University of Cincinnati

Date: 6/18/2018

I, Sounak Gupta, hereby submit this original work as part of the requirements for the degree of Doctor of Philosophy in Computer Science & Engineering.

It is entitled:

Pending Event Set Management in Parallel Discrete Event Simulation

Student's name: **Sounak Gupta**

This work and its defense approved by:

Committee chair: Philip Wilsey, Ph.D.

Committee member: Nael Abu-Ghazaleh, Ph.D.

Committee member: Fred Beyette, Ph.D.

Committee member: Ali Minai, Ph.D.

Committee member: Carla Purdy, Ph.D.

30512

Pending Event Set Management

in

Parallel Discrete Event Simulation

A dissertation submitted to the

Graduate School

of the University of Cincinnati

in partial fulfillment of the

requirements for the degree of

Doctor of Philosophy

in the Department of Electrical Engineering and Computer Science

of the College of Engineering and Applied Sciences

June 18, 2018

by

Sounak Gupta

B.E. , Bengal Engineering and Science University, Shibpur

June 2009

Committee Chair: Philip A. Wilsey , Ph.D.

Abstract

In Parallel Discrete Event Simulation (PDES), the *pending event set* refers to the set of events available for execution. These pending events are aggressively scheduled for execution in a Time Warp synchronized parallel simulation without strict enforcement of the causal relationship between events [1, 2]. For most discrete event simulation models, event processing granularity is generally quite small. On many-core and multi-core platforms, this decrease in granularity aggravates contention for the shared data structures which store these pending events. As the number of cores increase, a key challenge lies in providing effective, contention-free event list management for scheduling events. *Lock contention*, *sorting*, and *scheduling order* are the prime contributors to contention for access to the pending events set.

Possible solutions to this problem include atomic read-write operations [3], hardware transactional memory [4,5], or synchronization-friendly data structures [6,7]. The focus is on choosing efficient data structures for the pending event set and optimization of scheduling techniques that can improve the performance of the Time Warp synchronized parallel simulation. The following design concepts for optimizing the pending event set are explored in this dissertation:

1. an exploration of a variety of different data structures that are commonly used in the management of pending event set. In addition the management of the pending event set using a *Ladder Queue* [8] data structure is explored. The Ladder Queue forms a self-adjusting hierarchically partitioned priority queue that makes it particularly attractive for managing the pending event set;

2. the elimination of sorting within the Ladder Queue partitions. Events are then scheduled from the lowest partition without concerns for their time order and causal independence of these events is assumed;

3. an atomic read-write access to the the Ladder Queue partition that holds the smallest available events is explored;

4. Objects containing groups of events are organized in the pending event set. Each group is dequeued with one access and the collection of events are scheduled. Several different options for the definition of these groups and how these groups are scheduled is explored.

Experimental results show that fully-sorted Ladder Queue is over 20% faster than STL MultiSet and Splay Tree. It was also observed that Ladder Queue with unsorted lowest partition is at least 25% faster than fully-sorted Ladder Queue for all discrete event simulation models studied. The atomic read-write access to this lowest partition of the Ladder Queue allows simulation models to run 25-150% faster than the fully-sorted Ladder Queue. Furthermore, studies indicate that scheduling events in groups provides up to 100% gain in performance for some simulation models. The experiments also show that the modularity-based partitioning of LPs makes a simulation run as fast as Ladder Queue with atomic read-write operations for some kernel and model configurations. Overall Ladder Queue with atomic read-write access to its unsorted lowest partition has the best performance among the scheduling data structures studied and multiple schedule queues is the choice for scheduling strategy. Combining the two together results in the most effective scheduling mechanism. There is over 100% boost in performance for some models when this combination is compared to any of the other efficient configurations mentioned above.

Each of the aforementioned concepts explored address the contention to the shared data structures that store the pending event set. The work on *group* and *bag-centric* scheduling draws inspiration from the study on profile-driven data collected from discrete event simulations [9]. This study helped guide the design and optimization of scheduling strategies for the pending event set in a Time Warp synchronized parallel simulation.

Acknowledgements

First and foremost, I would like to express my sincere gratitude to my advisor Dr. Wilsey for his constant guidance, encouragement, and support throughout my journey as a graduate student. His deep insights into the dynamics of parallel simulation, data-driven approach to research, and exhaustive review of my drafts have motivated me to improve as a researcher.

I would like to thank the rest of my thesis committee, namely Dr. Fred Beyette, Dr. Ali Minai, Dr. Carla Purdy and Dr. Nael Abu-Ghazaleh, for their insightful comments and questions which have encouraged me to analyze my research from a broader perspective.

I also take this opportunity to thank my fellow lab-mates and collaborators, especially Tom Dickman and Doug Weber.

This acknowledgement would remain incomplete without any mention about my parents. I am grateful to them for giving me a happy childhood and a good education. I am also grateful to my fiancée Anuja Roy for patiently tolerating my chaotic working hours and geekish hobbies. I also thank my cousin, Kingshuk Mallik, for those pointless intellectual debates and help with my graduate applications and formatting of this thesis.

Last but not least, this journey would not have been memorable and enjoyable without friendly fellow travellers. I would like to thank them, especially Saibal Ghosh, Souvik Sarkar, Abhrojyoti Mondal, Sauradeep Bhowmick and Dr. Maitreya Dutta (in no particular order).

Contents

1 Introduction **2**

1.1 Thesis Overview 5

2 Background **7**

2.1 DES 7

2.2 PDES 8

2.2.1 Time Warp 9

2.3 ARCHITECTURE 11

2.3.1 Shared Memory Multiprocessors 11

2.3.2 Clustered Systems 12

2.4 COMMUNICATION 14

2.4.1 Message Passing 14

2.4.2 Shared Memory 15

3 Related Work **17**

3.1 Georgia Tech Time Warp (GTW) 17

3.2 Clustered Time Warp (CTW) 19

3.3 ROSS 20

3.3.1 ROSS-MT 21

3.4 WARPED 21

3.5 ROOT-SIM 22

4 WARPED2 **24**

4.1 Conceptual Overview . . . 24
4.1.1 Local Time Warp Components . . . 25
4.1.2 Global Time Warp Components . . . 27
4.1.3 Communication Manager . . . 28
4.1.4 Partitioner . . . 28
4.2 Journey of an Event . . . 29
4.2.1 How are Events Ordered? . . . 29
4.2.2 Pending Event Set . . . 30
4.2.3 Processing of Events . . . 31
4.2.4 Rollbacks & Cancellation . . . 32

5 PENDING EVENT SET 35
5.1 DATA STRUCTURES . . . 35
5.1.1 STL MultiSet and Splay Tree . . . 36
5.1.2 Ladder Queue . . . 37
5.1.3 Ladder Queue with Unsorted Bottom . . . 42
5.1.4 Ladder Queue with Lock-free Unsorted Bottom . . . 44
5.2 SCHEDULING TECHNIQUES . . . 50
5.2.1 Multiple Schedule Queues . . . 51
5.2.2 Chains . . . 53
5.2.3 Block Scheduling . . . 57
5.2.4 Bags . . . 61

6 Experiments 68
6.1 Setup . . . 68
6.2 Performance Metrics . . . 69
6.3 Benchmarks . . . 70
6.3.1 Epidemic Disease Propagation . . . 70
6.3.2 Portable Cellular Service (PCS) . . . 74

6.3.3 Traffic . . . 75
6.4 Analysis of Performance . . . 77
6.4.1 Schedule Queue Data Structure . . . 77
6.4.2 Multiple Schedule Queues . . . 87
6.4.3 Chain Scheduling . . . 95
6.4.4 Block Scheduling . . . 103
6.4.5 Bags with Static Window Size . . . 111
6.4.6 Bags with Fractional Time Window . . . 119
6.5 Summary of Performance Analysis . . . 126

7 Performance vs. Sequential **131**
7.1 Quantitative Analysis . . . 131
7.1.1 Top Performers . . . 131
7.1.2 Worst Performers . . . 140
7.1.3 Anomalies in Performance . . . 141
7.2 WARPED-2 on SMP . . . 141

8 Conclusion and Future Research **146**
8.1 Conclusion . . . 146
8.2 Future Work . . . 148
8.2.1 Hybrid Group Scheduling . . . 148
8.2.2 Load Balancing on Multi-Core Processors . . . 148

A All Results **159**
A.1 TRAFFIC-10K LPS . . . 159
A.2 TRAFFIC-1M LPS . . . 232
A.3 EPIDEMIC-WS-10K LPS . . . 289
A.4 EPIDEMIC-WS-100K LPS . . . 362
A.5 EPIDEMIC-BA-10K LPS . . . 431
A.6 EPIDEMIC-BA-100K LPS . . . 504

A.7 PCS-10K LPs 573

List of Figures

2.1 Discrete Event Simulation 8
2.2 Violation of Event Causality 9
2.3 Data Structures of Logical Process in Time Warp 10
2.4 A Symmetric Multiprocessor 12
2.5 A Non-Uniform Memory Access System 13
2.6 Beowulf Cluster 13
2.7 Message Passing Communication Model 15
2.8 Shared Memory Communication Model 16

4.1 Time Warp Components in WARPED2 26
4.2 Round Robin Partitioning in WARPED2 29
4.3 WARPED2 Pending Event Set 32

5.1 The Ladder Queue Structure 38
5.2 LADDER QUEUE with Unsorted Bottom 43
5.3 Insert Operation on Lock-Free List 46
5.4 Dequeue Operation on Lock-Free List 47
5.5 Multiple Schedule Queues in WARPED2 51
5.6 Traffic Model : Local Event Chains 53
5.7 Epidemic Disease Propagation Model : Local Event Chains 54
5.8 Portable Cellular Service Model : Local Event Chains 54
5.9 WARPED2 Scheduling Technique : Chains 55
5.10 WARPED2 Scheduling Technique : Blocks 57

5.11 Louvain-based Partitions (shown in different colors) . . . 62
5.12 Distribution of Communities in 'epidemic-10k-ws' model (refer to Table 6.3 for model specifications) . . . 62
5.13 WARPED2 Scheduling Technique : Bags . . . 63

6.1 Diffusion of Population . . . 72
6.2 Probabilistic Timed Transition System . . . 72
6.3 Epidemic Model . . . 73
6.4 Portable Cellular Service Model . . . 74
6.5 Sequence of events at each intersection . . . 75
6.6 Impact of the schedule queue data structure: Epidemic Model (*10,000* LPs, *Barabsi-Albert* diffusion network) . . . 80
6.7 Impact of the schedule queue data structure: Epidemic Model (*10,000* LPs, *Watts-Strogatz* diffusion network) . . . 82
6.8 Impact of the schedule queue data structure: Traffic Model (*10,000* LPs) . . . 84
6.9 Impact of the schedule queue data structure: PCS Model (*10,000* LPs) . . . 86
6.10 Impact of the multiple schedule queues: Epidemic Model (*10,000* LPs, *Barabsi-Albert* diffusion network) . . . 88
6.11 Impact of the multiple schedule queues: Epidemic Model (*10,000* LPs, *Watts-Strogatz* diffusion network) . . . 90
6.12 Impact of the multiple schedule queues: Traffic Model (*10,000* LPs) . . . 92
6.13 Impact of the multiple schedule queues: PCS Model (*10,000* LPs) . . . 94
6.14 Performance of Event Chains: Epidemic Model (*10,000* LPs, *Barabsi-Albert* diffusion network) . . . 97
6.15 Performance of Event Chains: Epidemic Model (*10,000* LPs, *Watts-Strogatz* diffusion network) . . . 98
6.16 Performance of Event Chains: Traffic Model (*10,000* LPs) . . . 100
6.17 Performance of Event Chains: Traffic Model (*1,048,576* LPs) . . . 102

6.18 Performance of Event Blocks: Epidemic Model (*10,000* LPs, *Barabsi-Albert* diffusion network) . . . 104
6.19 Performance of Event Blocks: Epidemic Model (*10,000* LPs, *Watts-Strogatz* diffusion network) . . . 106
6.20 Performance of Event Blocks: Traffic Model (*10,000* LPs) . . . 108
6.21 Performance of Event Blocks: Traffic Model (*1,048,576* LPs) . . . 110
6.22 Performance of Bags with Static Window Size: Epidemic Model (*10,000* LPs, *Barabsi-Albert* diffusion network) . . . 112
6.23 Performance of Bags with Static Window Size: Epidemic Model (*10,000* LPs, *Watts-Strogatz* diffusion network) . . . 114
6.24 Performance of Bags with Static Window Size: Traffic Model (*10,000* LPs) . . . 116
6.25 Performance of Bags with Static Window Size: PCS Model (*10,000* LPs) . . . 118
6.26 Performance of Bags with Fractional Time Window: Epidemic Model (*10,000* LPs, *Barabsi-Albert* diffusion network) . . . 120
6.27 Performance of Bags with Fractional Time Window: Epidemic Model (*10,000* LPs, *Watts-Strogatz* diffusion network) . . . 122
6.28 Performance of Bags with Fractional Time Window: Traffic Model (*10,000* LPs) . . . 124
6.29 Performance of Bags with Fractional Time Window: PCS Model (*10,000* LPs) . . . 126
6.30 Traffic for Multiple Ladder Queue-based Schedule Queues with Lockfree Unsorted Bottom (*10,000* LPs) . . . 128
6.31 Epidemic (Watts-Strogatz network) for Multiple Ladder Queue-based Schedule Queues with Lockfree Unsorted Bottom (*10,000* LPs) . . . 129

7.1 Top performers for Traffic (*10,000* LPs) model . . . 132
7.2 Top performers for Traffic (*1,048,576* LPs) model . . . 133
7.3 Top performers for Epidemic-BA (*10,000* LPs) model . . . 134
7.4 Top performers for Epidemic-WS (*10,000* LPs) model . . . 135
7.5 Top performers for Epidemic-BA (*100,000* LPs) model . . . 136
7.6 Top performers for Epidemic-WS (*100,000* LPs) model . . . 137

7.7 Top performers for PCS (*10,000* LPs) model . . . 138
7.8 Top performers for Traffic (*10,000* LPs) model . . . 142
7.9 Top performers for Traffic (*1,048,576* LPs) model . . . 143
7.10 Top performers for Epidemic-BA (*10,000* LPs) model . . . 143
7.11 Top performers for Epidemic-WS (*10,000* LPs) model . . . 144
7.12 Top performers for Epidemic-BA (*100,000* LPs) model . . . 144
7.13 Top performers for Epidemic-WS (*100,000* LPs) model . . . 145
7.14 Top performers for PCS (*10,000* LPs) model . . . 145

A.1 traffic 10k/plots/chains/threads vs count key chainsize 8 . . . 161
A.2 traffic 10k/plots/chains/threads vs chainsize key count 1 . . . 163
A.3 traffic 10k/plots/chains/threads vs chainsize key count 4 . . . 165
A.4 traffic 10k/plots/chains/threads vs chainsize key count 16 . . . 167
A.5 traffic 10k/plots/chains/threads vs count key chainsize 4 . . . 169
A.6 traffic 10k/plots/chains/threads vs count key chainsize 12 . . . 171
A.7 traffic 10k/plots/chains/threads vs chainsize key count 8 . . . 173
A.8 traffic 10k/plots/chains/threads vs chainsize key count 2 . . . 175
A.9 traffic 10k/plots/chains/threads vs count key chainsize 16 . . . 177
A.10 traffic 10k/plots/scheduleq/threads vs count key type ladder-queue 96 . . . 179
A.11 traffic 10k/plots/scheduleq/threads vs count key type stl-multiset . . . 181
A.12 traffic 10k/plots/scheduleq/threads vs count key type unsorted-bottom 50 . . . 183
A.13 traffic 10k/plots/scheduleq/threads vs count key type lockfree-unsorted-bottom 50 . . . 185
A.14 traffic 10k/plots/scheduleq/threads vs type key count 16 . . . 187
A.15 traffic 10k/plots/scheduleq/threads vs type key count 4 . . . 189
A.16 traffic 10k/plots/scheduleq/threads vs count key type lockfree-unsorted-bottom 96 . . . 191
A.17 traffic 10k/plots/scheduleq/threads vs count key type splay-tree . . . 193
A.18 traffic 10k/plots/scheduleq/threads vs count key type unsorted-bottom 96 . . . 195
A.19 traffic 10k/plots/scheduleq/threads vs count key type ladder-queue 50 . . . 197
A.20 traffic 10k/plots/scheduleq/threads vs type key count 8 . . . 199

A.21 traffic 10k/plots/scheduleq/threads vs count key type lockfree-unsorted-bottom 16 201
A.22 traffic 10k/plots/scheduleq/threads vs type key count 2 203
A.23 traffic 10k/plots/scheduleq/threads vs count key type ladder-queue 16 205
A.24 traffic 10k/plots/scheduleq/threads vs type key count 1 207
A.25 traffic 10k/plots/scheduleq/threads vs count key type unsorted-bottom 16 209
A.26 traffic 10k/plots/bags/threads vs staticwindow key fractionwindow 1.0 211
A.27 traffic 10k/plots/bags/threads vs fractionwindow key staticwindow 0 213
A.28 traffic 10k/plots/blocks/threads vs blocksize key count 4 215
A.29 traffic 10k/plots/blocks/threads vs blocksize key count 2 217
A.30 traffic 10k/plots/blocks/threads vs count key blocksize 256 219
A.31 traffic 10k/plots/blocks/threads vs blocksize key count 8 221
A.32 traffic 10k/plots/blocks/threads vs count key blocksize 32 223
A.33 traffic 10k/plots/blocks/threads vs blocksize key count 16 225
A.34 traffic 10k/plots/blocks/threads vs blocksize key count 1 227
A.35 traffic 10k/plots/blocks/threads vs count key blocksize 128 229
A.36 traffic 10k/plots/blocks/threads vs count key blocksize 64 231
A.37 traffic 1m/plots/chains/threads vs count key chainsize 8 233
A.38 traffic 1m/plots/chains/threads vs chainsize key count 1 235
A.39 traffic 1m/plots/chains/threads vs chainsize key count 4 238
A.40 traffic 1m/plots/chains/threads vs chainsize key count 16 240
A.41 traffic 1m/plots/chains/threads vs count key chainsize 4 242
A.42 traffic 1m/plots/chains/threads vs count key chainsize 12 244
A.43 traffic 1m/plots/chains/threads vs chainsize key count 8 246
A.44 traffic 1m/plots/chains/threads vs chainsize key count 2 248
A.45 traffic 1m/plots/chains/threads vs count key chainsize 16 250
A.46 traffic 1m/plots/scheduleq/threads vs count key type stl-multiset 252
A.47 traffic 1m/plots/scheduleq/threads vs count key type lockfree-unsorted-bottom 50 254
A.48 traffic 1m/plots/scheduleq/threads vs type key count 16 256

A.49 traffic 1m/plots/scheduleq/threads vs type key count 4 258
A.50 traffic 1m/plots/scheduleq/threads vs count key type lockfree-unsorted-bottom 96 260
A.51 traffic 1m/plots/scheduleq/threads vs count key type splay-tree 262
A.52 traffic 1m/plots/scheduleq/threads vs type key count 8 264
A.53 traffic 1m/plots/scheduleq/threads vs count key type lockfree-unsorted-bottom 16 266
A.54 traffic 1m/plots/scheduleq/threads vs type key count 2 268
A.55 traffic 1m/plots/scheduleq/threads vs type key count 1 270
A.56 traffic 1m/plots/blocks/threads vs blocksize key count 4 272
A.57 traffic 1m/plots/blocks/threads vs blocksize key count 2 274
A.58 traffic 1m/plots/blocks/threads vs count key blocksize 256 276
A.59 traffic 1m/plots/blocks/threads vs blocksize key count 8 278
A.60 traffic 1m/plots/blocks/threads vs count key blocksize 32 280
A.61 traffic 1m/plots/blocks/threads vs blocksize key count 16 282
A.62 traffic 1m/plots/blocks/threads vs blocksize key count 1 284
A.63 traffic 1m/plots/blocks/threads vs count key blocksize 128 286
A.64 traffic 1m/plots/blocks/threads vs count key blocksize 64 288
A.65 epidemic 10k ws/plots/chains/threads vs count key chainsize 8 290
A.66 epidemic 10k ws/plots/chains/threads vs chainsize key count 1 292
A.67 epidemic 10k ws/plots/chains/threads vs chainsize key count 4 295
A.68 epidemic 10k ws/plots/chains/threads vs chainsize key count 16 297
A.69 epidemic 10k ws/plots/chains/threads vs count key chainsize 4 299
A.70 epidemic 10k ws/plots/chains/threads vs count key chainsize 12 301
A.71 epidemic 10k ws/plots/chains/threads vs chainsize key count 8 303
A.72 epidemic 10k ws/plots/chains/threads vs chainsize key count 2 305
A.73 epidemic 10k ws/plots/chains/threads vs count key chainsize 16 307
A.74 epidemic 10k ws/plots/scheduleq/threads vs count key type ladder-queue 96 309
A.75 epidemic 10k ws/plots/scheduleq/threads vs count key type stl-multiset 311
A.76 epidemic 10k ws/plots/scheduleq/threads vs count key type unsorted-bottom 50 313

A.77 epidemic 10k ws/plots/scheduleq/threads vs count key type lockfree-unsorted-bottom 50 . 315
A.78 epidemic 10k ws/plots/scheduleq/threads vs type key count 16 317
A.79 epidemic 10k ws/plots/scheduleq/threads vs type key count 4 319
A.80 epidemic 10k ws/plots/scheduleq/threads vs count key type lockfree-unsorted-bottom 96 . 321
A.81 epidemic 10k ws/plots/scheduleq/threads vs count key type splay-tree 323
A.82 epidemic 10k ws/plots/scheduleq/threads vs count key type unsorted-bottom 96 325
A.83 epidemic 10k ws/plots/scheduleq/threads vs count key type ladder-queue 50 327
A.84 epidemic 10k ws/plots/scheduleq/threads vs type key count 8 329
A.85 epidemic 10k ws/plots/scheduleq/threads vs count key type lockfree-unsorted-bottom 16 . 331
A.86 epidemic 10k ws/plots/scheduleq/threads vs type key count 2 333
A.87 epidemic 10k ws/plots/scheduleq/threads vs count key type ladder-queue 16 335
A.88 epidemic 10k ws/plots/scheduleq/threads vs type key count 1 337
A.89 epidemic 10k ws/plots/scheduleq/threads vs count key type unsorted-bottom 16 339
A.90 epidemic 10k ws/plots/bags/threads vs staticwindow key fractionwindow 1.0 341
A.91 epidemic 10k ws/plots/bags/threads vs fractionwindow key staticwindow 0 343
A.92 epidemic 10k ws/plots/blocks/threads vs blocksize key count 4 345
A.93 epidemic 10k ws/plots/blocks/threads vs blocksize key count 2 347
A.94 epidemic 10k ws/plots/blocks/threads vs count key blocksize 256 349
A.95 epidemic 10k ws/plots/blocks/threads vs blocksize key count 8 351
A.96 epidemic 10k ws/plots/blocks/threads vs count key blocksize 32 353
A.97 epidemic 10k ws/plots/blocks/threads vs blocksize key count 16 355
A.98 epidemic 10k ws/plots/blocks/threads vs blocksize key count 1 357
A.99 epidemic 10k ws/plots/blocks/threads vs count key blocksize 128 359
A.100 epidemic 10k ws/plots/blocks/threads vs count key blocksize 64 361
A.101 epidemic 100k ws/plots/chains/threads vs count key chainsize 8 363
A.102 epidemic 100k ws/plots/chains/threads vs chainsize key count 1 365
A.103 epidemic 100k ws/plots/chains/threads vs chainsize key count 4 368
A.104 epidemic 100k ws/plots/chains/threads vs chainsize key count 16 370

A.105 epidemic 100k ws/plots/chains/threads vs count key chainsize 4 372
A.106 epidemic 100k ws/plots/chains/threads vs count key chainsize 12 374
A.107 epidemic 100k ws/plots/chains/threads vs chainsize key count 8 376
A.108 epidemic 100k ws/plots/chains/threads vs chainsize key count 2 378
A.109 epidemic 100k ws/plots/chains/threads vs count key chainsize 16 380
A.110 epidemic 100k ws/plots/scheduleq/threads vs count key type ladder-queue 96 382
A.111 epidemic 100k ws/plots/scheduleq/threads vs count key type stl-multiset 384
A.112 epidemic 100k ws/plots/scheduleq/threads vs count key type unsorted-bottom 50 386
A.113 epidemic 100k ws/plots/scheduleq/threads vs count key type lockfree-unsorted-bottom 50 388
A.114 epidemic 100k ws/plots/scheduleq/threads vs type key count 16 390
A.115 epidemic 100k ws/plots/scheduleq/threads vs type key count 4 392
A.116 epidemic 100k ws/plots/scheduleq/threads vs count key type lockfree-unsorted-bottom 96 394
A.117 epidemic 100k ws/plots/scheduleq/threads vs count key type splay-tree 396
A.118 epidemic 100k ws/plots/scheduleq/threads vs count key type unsorted-bottom 96 398
A.119 epidemic 100k ws/plots/scheduleq/threads vs count key type ladder-queue 50 400
A.120 epidemic 100k ws/plots/scheduleq/threads vs type key count 8 402
A.121 epidemic 100k ws/plots/scheduleq/threads vs count key type lockfree-unsorted-bottom 16 404
A.122 epidemic 100k ws/plots/scheduleq/threads vs type key count 2 406
A.123 epidemic 100k ws/plots/scheduleq/threads vs count key type ladder-queue 16 408
A.124 epidemic 100k ws/plots/scheduleq/threads vs type key count 1 410
A.125 epidemic 100k ws/plots/scheduleq/threads vs count key type unsorted-bottom 16 412
A.126 epidemic 100k ws/plots/blocks/threads vs blocksize key count 4 414
A.127 epidemic 100k ws/plots/blocks/threads vs blocksize key count 2 416
A.128 epidemic 100k ws/plots/blocks/threads vs count key blocksize 256 418
A.129 epidemic 100k ws/plots/blocks/threads vs blocksize key count 8 420
A.130 epidemic 100k ws/plots/blocks/threads vs count key blocksize 32 422
A.131 epidemic 100k ws/plots/blocks/threads vs blocksize key count 16 424
A.132 epidemic 100k ws/plots/blocks/threads vs blocksize key count 1 426

A.133 epidemic 100k ws/plots/blocks/threads vs count key blocksize 128 428
A.134 epidemic 100k ws/plots/blocks/threads vs count key blocksize 64 430
A.135 epidemic 10k ba/plots/chains/threads vs count key chainsize 8 432
A.136 epidemic 10k ba/plots/chains/threads vs chainsize key count 1 434
A.137 epidemic 10k ba/plots/chains/threads vs chainsize key count 4 437
A.138 epidemic 10k ba/plots/chains/threads vs chainsize key count 16 439
A.139 epidemic 10k ba/plots/chains/threads vs count key chainsize 4 441
A.140 epidemic 10k ba/plots/chains/threads vs count key chainsize 12 443
A.141 epidemic 10k ba/plots/chains/threads vs chainsize key count 8 445
A.142 epidemic 10k ba/plots/chains/threads vs chainsize key count 2 447
A.143 epidemic 10k ba/plots/chains/threads vs count key chainsize 16 449
A.144 epidemic 10k ba/plots/scheduleq/threads vs count key type ladder-queue 96 451
A.145 epidemic 10k ba/plots/scheduleq/threads vs count key type stl-multiset 453
A.146 epidemic 10k ba/plots/scheduleq/threads vs count key type unsorted-bottom 50 455
A.147 epidemic 10k ba/plots/scheduleq/threads vs count key type lockfree-unsorted-bottom 50 . 457
A.148 epidemic 10k ba/plots/scheduleq/threads vs type key count 16 459
A.149 epidemic 10k ba/plots/scheduleq/threads vs type key count 4 461
A.150 epidemic 10k ba/plots/scheduleq/threads vs count key type lockfree-unsorted-bottom 96 . 463
A.151 epidemic 10k ba/plots/scheduleq/threads vs count key type splay-tree 465
A.152 epidemic 10k ba/plots/scheduleq/threads vs count key type unsorted-bottom 96 467
A.153 epidemic 10k ba/plots/scheduleq/threads vs count key type ladder-queue 50 469
A.154 epidemic 10k ba/plots/scheduleq/threads vs type key count 8 471
A.155 epidemic 10k ba/plots/scheduleq/threads vs count key type lockfree-unsorted-bottom 16 . 473
A.156 epidemic 10k ba/plots/scheduleq/threads vs type key count 2 475
A.157 epidemic 10k ba/plots/scheduleq/threads vs count key type ladder-queue 16 477
A.158 epidemic 10k ba/plots/scheduleq/threads vs type key count 1 479
A.159 epidemic 10k ba/plots/scheduleq/threads vs count key type unsorted-bottom 16 481
A.160 epidemic 10k ba/plots/bags/threads vs staticwindow key fractionwindow 1.0 483

A.161 epidemic 10k ba/plots/bags/threads vs fractionwindow key staticwindow 0 485
A.162 epidemic 10k ba/plots/blocks/threads vs blocksize key count 4 487
A.163 epidemic 10k ba/plots/blocks/threads vs blocksize key count 2 489
A.164 epidemic 10k ba/plots/blocks/threads vs count key blocksize 256 491
A.165 epidemic 10k ba/plots/blocks/threads vs blocksize key count 8 493
A.166 epidemic 10k ba/plots/blocks/threads vs count key blocksize 32 495
A.167 epidemic 10k ba/plots/blocks/threads vs blocksize key count 16 497
A.168 epidemic 10k ba/plots/blocks/threads vs blocksize key count 1 499
A.169 epidemic 10k ba/plots/blocks/threads vs count key blocksize 128 501
A.170 epidemic 10k ba/plots/blocks/threads vs count key blocksize 64 503
A.171 epidemic 100k ba/plots/chains/threads vs count key chainsize 8 505
A.172 epidemic 100k ba/plots/chains/threads vs chainsize key count 1 507
A.173 epidemic 100k ba/plots/chains/threads vs chainsize key count 4 510
A.174 epidemic 100k ba/plots/chains/threads vs chainsize key count 16 512
A.175 epidemic 100k ba/plots/chains/threads vs count key chainsize 4 514
A.176 epidemic 100k ba/plots/chains/threads vs count key chainsize 12 516
A.177 epidemic 100k ba/plots/chains/threads vs chainsize key count 8 518
A.178 epidemic 100k ba/plots/chains/threads vs chainsize key count 2 520
A.179 epidemic 100k ba/plots/chains/threads vs count key chainsize 16 522
A.180 epidemic 100k ba/plots/scheduleq/threads vs count key type ladder-queue 96 524
A.181 epidemic 100k ba/plots/scheduleq/threads vs count key type stl-multiset 526
A.182 epidemic 100k ba/plots/scheduleq/threads vs count key type unsorted-bottom 50 528
A.183 epidemic 100k ba/plots/scheduleq/threads vs count key type lockfree-unsorted-bottom 50 . 530
A.184 epidemic 100k ba/plots/scheduleq/threads vs type key count 16 532
A.185 epidemic 100k ba/plots/scheduleq/threads vs type key count 4 534
A.186 epidemic 100k ba/plots/scheduleq/threads vs count key type lockfree-unsorted-bottom 96 . 536
A.187 epidemic 100k ba/plots/scheduleq/threads vs count key type splay-tree 538
A.188 epidemic 100k ba/plots/scheduleq/threads vs count key type unsorted-bottom 96 540

A.189 epidemic 100k ba/plots/scheduleq/threads vs count key type ladder-queue 50 542
A.190 epidemic 100k ba/plots/scheduleq/threads vs type key count 8 544
A.191 epidemic 100k ba/plots/scheduleq/threads vs count key type lockfree-unsorted-bottom 16 . 546
A.192 epidemic 100k ba/plots/scheduleq/threads vs type key count 2 548
A.193 epidemic 100k ba/plots/scheduleq/threads vs count key type ladder-queue 16 550
A.194 epidemic 100k ba/plots/scheduleq/threads vs type key count 1 552
A.195 epidemic 100k ba/plots/scheduleq/threads vs count key type unsorted-bottom 16 554
A.196 epidemic 100k ba/plots/blocks/threads vs blocksize key count 4 556
A.197 epidemic 100k ba/plots/blocks/threads vs blocksize key count 2 558
A.198 epidemic 100k ba/plots/blocks/threads vs count key blocksize 256 560
A.199 epidemic 100k ba/plots/blocks/threads vs blocksize key count 8 562
A.200 epidemic 100k ba/plots/blocks/threads vs count key blocksize 32 564
A.201 epidemic 100k ba/plots/blocks/threads vs blocksize key count 16 566
A.202 epidemic 100k ba/plots/blocks/threads vs blocksize key count 1 568
A.203 epidemic 100k ba/plots/blocks/threads vs count key blocksize 128 570
A.204 epidemic 100k ba/plots/blocks/threads vs count key blocksize 64 572
A.205 pcs 10k/plots/chains/threads vs count key chainsize 8 574
A.206 pcs 10k/plots/chains/threads vs chainsize key count 1 576
A.207 pcs 10k/plots/chains/threads vs chainsize key count 4 579
A.208 pcs 10k/plots/chains/threads vs chainsize key count 16 581
A.209 pcs 10k/plots/chains/threads vs count key chainsize 4 583
A.210 pcs 10k/plots/chains/threads vs count key chainsize 12 585
A.211 pcs 10k/plots/chains/threads vs chainsize key count 8 587
A.212 pcs 10k/plots/chains/threads vs chainsize key count 2 589
A.213 pcs 10k/plots/chains/threads vs count key chainsize 16 591
A.214 pcs 10k/plots/scheduleq/threads vs count key type ladder-queue 96 593
A.215 pcs 10k/plots/scheduleq/threads vs count key type stl-multiset 595
A.216 pcs 10k/plots/scheduleq/threads vs count key type unsorted-bottom 50 597

A.217 pcs 10k/plots/scheduleq/threads vs count key type lockfree-unsorted-bottom 50 599
A.218 pcs 10k/plots/scheduleq/threads vs type key count 16 601
A.219 pcs 10k/plots/scheduleq/threads vs type key count 4 603
A.220 pcs 10k/plots/scheduleq/threads vs count key type lockfree-unsorted-bottom 96 605
A.221 pcs 10k/plots/scheduleq/threads vs count key type splay-tree 607
A.222 pcs 10k/plots/scheduleq/threads vs count key type unsorted-bottom 96 609
A.223 pcs 10k/plots/scheduleq/threads vs count key type ladder-queue 50 611
A.224 pcs 10k/plots/scheduleq/threads vs type key count 8 613
A.225 pcs 10k/plots/scheduleq/threads vs count key type lockfree-unsorted-bottom 16 615
A.226 pcs 10k/plots/scheduleq/threads vs type key count 2 617
A.227 pcs 10k/plots/scheduleq/threads vs count key type ladder-queue 16 619
A.228 pcs 10k/plots/scheduleq/threads vs type key count 1 621
A.229 pcs 10k/plots/scheduleq/threads vs count key type unsorted-bottom 16 623
A.230 pcs 10k/plots/bags/threads vs staticwindow key fractionwindow 1.0 625
A.231 pcs 10k/plots/bags/threads vs fractionwindow key staticwindow 0 627
A.232 pcs 10k/plots/blocks/threads vs blocksize key count 4 629
A.233 pcs 10k/plots/blocks/threads vs blocksize key count 2 631
A.234 pcs 10k/plots/blocks/threads vs count key blocksize 256 633
A.235 pcs 10k/plots/blocks/threads vs blocksize key count 8 635
A.236 pcs 10k/plots/blocks/threads vs count key blocksize 32 637
A.237 pcs 10k/plots/blocks/threads vs blocksize key count 16 639
A.238 pcs 10k/plots/blocks/threads vs blocksize key count 1 641
A.239 pcs 10k/plots/blocks/threads vs count key blocksize 128 643
A.240 pcs 10k/plots/blocks/threads vs count key blocksize 64 645

List of Tables

6.1 WARPED2 setup 68
6.2 Experimental Setup 69
6.3 List of Epidemic Configurations 73
6.4 List of Traffic Configurations 77

7.1 Experimental Setup for SMP machine 141

A.1 TRAFFIC MODEL setup 159
A.2 LARGE TRAFFIC MODEL setup 232
A.3 EPIDEMIC MODEL WITH WATTS-STROGATZ setup 289
A.4 LARGE EPIDEMIC MODEL WITH WATTS-STROGATZ setup 362
A.5 EPIDEMIC MODEL WITH BARABASI-ALBERT setup 431
A.6 LARGE EPIDEMIC MODEL WITH BARABASI-ALBERT setup 504
A.7 PCS MODEL setup 573

List of Algorithms

1 Event Processing in GTW [20, 24] . . . 18

2 WARPED2 Event Processing Loop . . . 33

3 LADDER QUEUE Dequeue Operation . . . 39

4 LADDER QUEUE Insert Operation . . . 40

5 LADDER QUEUE Modified Insert Operation . . . 42

6 Lock-Free List Insert Event . . . 46

7 Lock-Free List Dequeue Event . . . 47

8 LADDER QUEUE Lock-Free Insert Operation . . . 48

9 LADDER QUEUE Lock-Free Dequeue Operation . . . 49

10 WARPED2 Event Processing Loop (Detailed) . . . 50

11 WARPED2 Event Processing Loop for Multiple Schedule Queues . . . 52

12 WARPED2 Event Processing Loop for Chain Scheduling . . . 58

13 WARPED2 Event Processing Loop for Block Scheduling . . . 60

14 WARPED2 Event Processing Loop for Bags . . . 67

Chapter 1

Introduction

This principal focus of this dissertation is *Pending Event Set Management* for Parallel Discrete Event Simulation (PDES) running on stand-alone multi-core processors. A Discrete Event Simulation (DES) models the behavior of a physical system whose state can change in discrete time intervals due to stimulus provided by events generated within the system. Events in DES are sequentially processed on a single processor and this unacceptably slow for large simulation models. Parallel Discrete Event Simulation (PDES) [1, 10] is an alternative to sequential DES that uses parallel algorithms to efficiently execute and synchronize a DES on a parallel compute platform. There are several possible synchronization mechanisms for PDES; some with a centralized mechanism and others with a distributed synchronization mechanism. This thesis explores the design of the pending event set for a PDES synchronized using the Time Warp Mechanism [2, 10]. Time Warp is a distributed synchronization mechanism that can be used with a variety of parallel processing architectures such as shared-memory multiprocessors, clusters, or other systems that support parallel execution [11, 12].

The rapid growth of multi-core and many-core processors in recent years is encouraging programmers to explore the benefits of hardware support for software-driven parallelism. However, the scalability of massively parallel software systems is limited by contention among threads for shared resources. A software system designed for processors with low number of cores will not necessarily scale up proportionately when executed on a many-core platform. This contention issue becomes particularly acute during scaling up of Time Warp synchronized [2, 13] PDES on many-core processors.

Time Warp is a checkpoint and rollback recovery mechanism that allows a discrete event simulation to process events greedily without explicit synchronization. It operates under the assumption that events will mostly arrive in time for the parallel simulator to process them in their intended order and, therefore, minimize the frequency of rollbacks. The events in a simulation are causally ordered based on their *timestamp*. For any given architecture, the 'critical path' is a reference to the time needed to execute all events using any conservative simulation mechanism (*i.e.* without any causal violations) [14]. This 'critical path' is, therefore, the lower bound on the time against which all parallel simulations can be compared. Due to the relatively fine-grained computational nature of most discrete event simulation models, efficient shared access to the *pending event set* scheduling mechanisms is of paramount importance to obtaining peak performance from the parallel simulation engine. In particular, the shared data structures holding the pending event set data and the mechanisms of safe access to them can have significant impact on the overall performance of the simulation system. This is especially critical in shared memory, many-core processing systems.

In light of these developments, the objectives to peak performance are to minimize contention to the shared data structures that hold the pending event set, while also allowing available events to be scheduled close to the critical path in order to minimize rollbacks. Lock contention, sorting, and order of event execution are the key aspects that amplify contention for the shared data structures of the pending event set. The focus here is to explore the limits of intra-node distributed event processing so that a Time Warp synchronized parallel simulation can deliver scalable speedups with minimal overall rollback.

The significant avenues explored in this dissertation are:

- **Ladder Queue**: The *Ladder Queue* [8] is a variant of the well-known *Calendar Queue* [15]. It is a novel data structure that can self-select the partition size for the months of the calendar. This sizing problem is one of the main challenges that makes traditional Calendar Queues difficult to employ. Thus, the *Ladder Queue* [8] potentially provides the performance benefits of a *Calendar Queue* without the complicated issues accompanying the sizing of time interval boundaries for each *month (or partition)*. Dickman *et al* [6] proposed that the Ladder Queue can efficiently hold and schedule the smallest timestamped events from each Logical Process (LP) assigned to the multi-processor node. The 'ladder' structure also presents some additional opportunities for design space optimization that have been outlined in the next two bullets. A detailed description of these design

space optimizations is presented in Section 5.1.2 of this dissertation.

- **Ladder Queue with Unsorted Bottom**: The Ladder Queue organizes stored events into partitions that can be arranged similar to rungs on a ladder. Conventionally, the Ladder Queue does not sort events within a rung partition until the events in a particular partition are needed for execution. Since these rungs partition events within fairly small time intervals, Gupta and Wilsey [3] proposed that events in the lowest partition can be processed without sorting. The hypothesis is that pending events contained within a rung are highly likely to be causally independent and they can therefore be scheduled for execution without further sorting based on their timestamp [3]. The details of this implementation/-variation of the Ladder Queue are presented in Section 5.1.3.

- **Ladder Queue with Lock-Free Access to Unsorted Bottom**: Partitioning the pending event set into a *Ladder Queue* data structure provides direct access to the lowest timestamped events from the LPs. However, since the *Ladder Queue* is a shared data structure, the locking costs and the fine-grained nature of discrete event simulation can trigger considerable contention and negatively impact overall performance. Gupta and Wilsey [3] proposed that atomic operations can be used to avoid lock contention for the lowest time window in a *Ladder Queue*. The implementation details of this are presented in Section 5.1.4.

- **Group Scheduling**: A traditional parallel simulation engine will schedule events one at a time for execution. Results from a related project on profiling discrete event simulation (called *DESMetrcs*) [9] suggests that there may be significant causal independence between the first few events in each LP's input event queue. Gupta and Wilsey [7] examined this concept with a preliminary implementation and defined two different mechanisms for defining *event groups* to help reduce the frequency of accesses to the pending event set. This approach further extends *causal independence* among events at the head of the pending event set (which contains the lowest timestamped events). The hypothesis is that scheduling groups of events at a time will help minimize contention to the pending event set. Two different group scheduling strategies were explored, namely: *block scheduling* and *chain scheduling*. The design details for event groups is presented in Sections 5.2.2 and 5.2.3.

- **Profile-driven Bag Scheduling**: In another study by the DESMetrics group, Crawford *et al* [16]

examined the modularity of LPs based on events communicated between the LPs in a Discrete Event Simulation. Their results suggest that it is possible to partition the LPs into groups with higher inter-LP event exchanges (which mirrors results from other empirical studies [17]). This tighter coupling also suggests that it might be possible to expand the group scheduling of events as outlined in the above bullet based on their modularity. This novel group scheduling practice organizes the lowest timestamped input events from each LPs into a bag for scheduling. A detailed study of this approach is contained in Section 5.2.4.

1.1 Thesis Overview

The remainder of this dissertation is organized as follows:

Chapter 2 a description of background information on parallel simulation and parallel computing. This knowledge is needed to understand the core concepts and ideas put forward in this dissertation.

Chapter 3 overviews several well-known parallel discrete event simulation kernels that are based on the Time Warp-based synchronization protocol. These implementations have significant features that have somewhat contributed to the overall design of the WARPED2 simulation kernel which is the target platform used in this dissertation.

Chapter 4 provides a detailed description of the WARPED2 the Time Warp-synchronized Parallel Discrete Event Simulation kernel developed by Dr. Philip A. Wilsey and his students at University of Cincinnati. The software architecture is discussed in detail and it serves as the experimental foundation for the ideas on organizing the *pending event set* that is explored in this research and described in Chapter 5.

Chapter 5 explains the core ideas and hypothesis are examined in this dissertation for implementing efficient data structures and algorithms within a Time Warp synchronized parallel simulation kernel executing on shared memory, many-core processors.

Chapter 6 presents an overview of the experimental setup and details of the simulation model benchmarks. Performance data from these experiments along with detailed analysis is also included in Section 6.4.

Chapter 7 presents a consolidated overview of the most effective scheduler configurations found from the quantitative study in Chapter 6.

Chapter 8 presents some final concluding remarks and suggestions for future research.

Chapter 2

Background

An overview of Parallel Discrete Event Simulation (PDES) and its Time Warp synchronization mechanism is contained in this chapter. In addition to the specific paper citations contained in this chapter, an excellent survey and book on PDES are also available [1, 10].

2.1 Discrete Event Simulation

There are many types and modeling styles for simulation and Figure 2.1 illustrates the broad categories of these. A physical system that is modelled as a sequence of events that are created at discrete time intervals is called *Discrete Event Simulation (DES)* [18]. The work in this dissertation will focus on the efficient implementation of the event processing engine in a parallel simulation kernel that is designed to execute DES models.

A *Discrete Event Simulation* model consists of three main data structures, namely:

Simulation clock: that records the simulated time of the model under study. This clock can be used for two primary purposes:

- measuring the progress of the simulation, and
- determining the order of events to be processed.

State variables: The snapshot of the simulated system at any specific point in time (otherwise called state) can be described accurately by these set of variables.

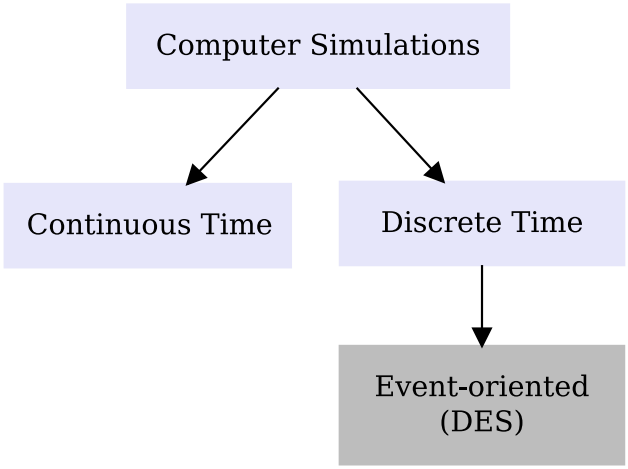


Figure 2.1: Discrete Event Simulation

Pending event set: A set of events that are yet to be processed.

A physical system can be described by a set of physical processes grouped together to form a *Simulation Model*. Each of these physical processes corresponds to a *Logical Process* (LP). The LPs communicate among themselves using timestamped events. The timestamp marks the simulation time used for ordering and processing of events. The state of the system is updated only when an event is processed.

A *Sequential Discrete Event Simulation* is a type of simulation where events are globally sorted and processed sequentially one at a time. A list holds all the events that are waiting to be processed and sorts them in increasing order of timestamp. The event with the lowest timestamp is processed first. The state of the system is updated and the simulation clock advances every time an event is processed. It is also likely that one or more new events are produced when an event is processed. This sequential mechanism is too slow and inefficient for large simulations and can be improved with parallelization.

2.2 Parallel Discrete Event Simulation

The method for running a discrete event simulation on a parallel computing platform is called Parallel Discrete Event Simulation (PDES). The parallel computing platform can be a shared memory multiprocessor, a distributed memory cluster, or a combination of both. A synchronization mechanism is used to coordinate parallel processing of events from different LPs while preserving the causal order for the final result [19]. The *Time Warp Mechanism* [1, 2, 10] is one such synchronization mechanism that we will explore further in

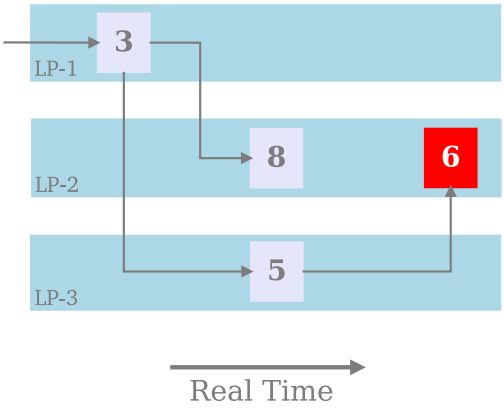


Figure 2.2: Violation of Event Causality

this dissertation. It is an *optimistic synchronization mechanism* which does not strictly preserve the causal order of events at all times and sometimes temporarily allows violation by the simulation executive. The next subsection provides a brief overview of the Time Warp mechanism.

2.2.1 Time Warp

The Time Warp mechanism is an optimistic method of simulation. It is based on the *Virtual Time* paradigm [2] which is a method for ordering events in distributed systems without requiring any knowledge of real time. *Virtual Time* is looked upon as the simulation time in case of parallel discrete event simulation.

Events from different LPs can be processed independently in an optimistically synchronized mechanism such as Time Warp. No consideration is required for situations when lower timestamped events are received *i.e.,* when the causal order of event is violated. Figure 2.2 depicts a causality violation scenario by showing the timing diagram for events in three LPs. An event with *timestamp = 3* is processed by LP_1 and two events are generated as a result, one with *timestamp = 8* is sent to LP_2 and another with *timestamp = 5* is sent to LP_3. LP_3, on processing this received event, generates an event for LP_2 with *timestamp = 6*. This event has a lower receive time than the previous event (*timestamp = 8*) received by LP_2. If this previously received event is processed by LP_2 before the event with lower receive time is received, it would lead to incorrect update of the state as well as new events could be sent to itself or other LPs incorrectly.

Rollback is the process of undoing the effects of incorrectly processed event(s) when a causality violation is detected at an LP. *Straggler event* is that incoming event with a timestamp lower than the LP simulation clock (the LP has advanced prematurely). The straggler event denotes a causality violation and triggers a

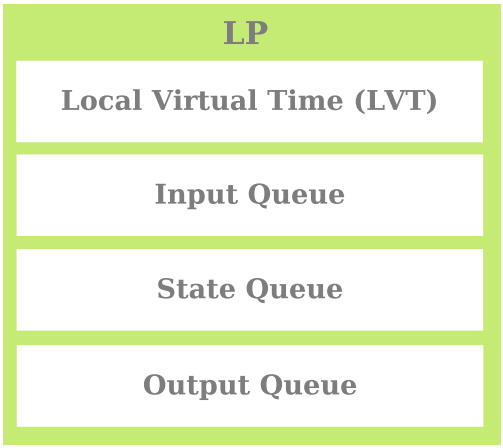


Figure 2.3: Data Structures of Logical Process in Time Warp

rollback. According to Jefferson [2], three data structures per LP are necessary to handle rollbacks: (1) Input Queue, (2) Output Queue, and (3) State Queue. Figure 2.3 shows the Input Queue which contains all past and future event, the Output Queue contains all previously sent events, and the State Queue holds snapshots of previous states.

On rollback, the state of an LP is restored back to a snapshot that was saved at a timestamp prior to the straggler event's receive time. The Output Queue keeps track of all events, with send time greater than the straggler event, that were sent incorrectly. Those events are are re-sent but this time as an *anti-message* or *negative event*. An anti-messages is a copy of the positive event that has already been sent, but with a crutial differentiator — a bit within the event payload is set to indicate it is a negative event. The receiving LP does not process an anti-message in the same way as it processes a positive event. The anti-message's job is to cancel out the positive counterpart in the input queue. The anti-message can also be a straggler if it causes causality violation for the receiving LP. In that scenario, a rollback is triggered in the manner mentioned above. This recursive process stops when the simulation is rid of causality violations.

The minimum timestamp of all unprocessed events and anti-messages at any given point during the simulation is called the *Global Virtual Time* (GVT) of the simulation. Events that have been sent but not yet received are also tracked for this metric [20]. Timestamp of the smallest unprocessed event in an LP is referred to as its *Local Virtual Time* (LVT). Though irrelevant as a metric on its own, *LVT* is essential to determine the *GVT* in a distributed environment. *GVT* calculation is a complex problem of estimation. Several algorithms for GVT estimation have been discussed in [21]. Estimation of *GVT* allows us to fix a

lower bound on how far back in time a rollback can occur. This is useful for selective elimination of events from the input and output queues and the states from the state queue. Only those events and states that have timestamp lower than GVT are eliminated since those will never be used again. This step is useful for memory reclamation as well as to commit I/O operations that cannot be undone. The process is called *fossil collection* and it does not necessarily have to be GVT-based. There are several other methods of fossil collection as discussed in [21].

2.3 Architecture of Parallel Processing Systems

Parallel processing systems [11,12] are generally characterized based on the manner in which processors and memories are grouped together. A single machine which shares a common address space for all processes is called a *shared memory* system. These systems can either have a single physical memory unit or multiple memory units. Multiple machines with separate process address connected over interconnected network is called a *clustered system*.

2.3.1 Shared Memory Multiprocessors

Symmetric Multiprocessor (SMP) consists of a group of identical processing cores connected to a single shared memory with full access to the input/output devices in the system. This arrangement allows uniform time access to the shared memory from each core. In general, this architecture fails to scale up linearly with increasing core counts due to the steep increase in memory contention. As a results, the number of processor cores in these configurations is generally bounded by some small number (generally ≤ 16, although this number continues to increase). Figure 2.4 illustrates a typical *SMP* system.

Non-Uniform Memory Access (NUMA) processor has multiple cores and multiple memories with variable access times to different memory locations based on the transport path between the processing cores and the memory module. In general, each core has a local memory module that provides fast access to its contents while access to a non-local memory module (generally local to some other core) is accessed with a longer access time. Due to the higher access time for remote memory units, the common programming practice is to maximize local memory accesses for a processor. Compared to *symmetric multiprocessors*,

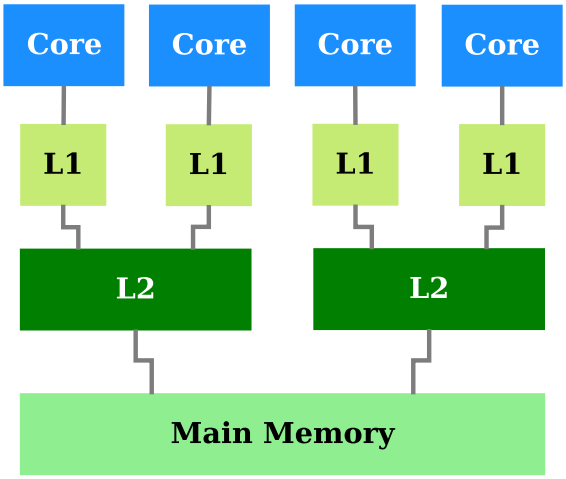


Figure 2.4: A Symmetric Multiprocessor

contention to single memory does not necessarily increase when number of processors is increased in *NUMA* systems. As a result, the latter system is capable of larger scale-up than SMP systems. Figure 2.5 illustrates a typical NUMA system.

2.3.2 Clustered Systems

A *Beowulf Cluster* (or simply cluster) is a loosely coupled set of machines (also called nodes) connected together over a local network. It is designed to appear as a single machine to the user. All nodes on the cluster execute the same program concurrently by launching multiple processes on each machine. A cluster allows the program currently being processed to communicate among processes using some type of message passing. This message passing is generally handled by a parallel communication software such as *Message Passing Interface (MPI)* [22] or *Parallel Virtual Machine (PVM)* [23]. Figure 2.6 illustrates a commonly used version of the *Beowulf cluster*.

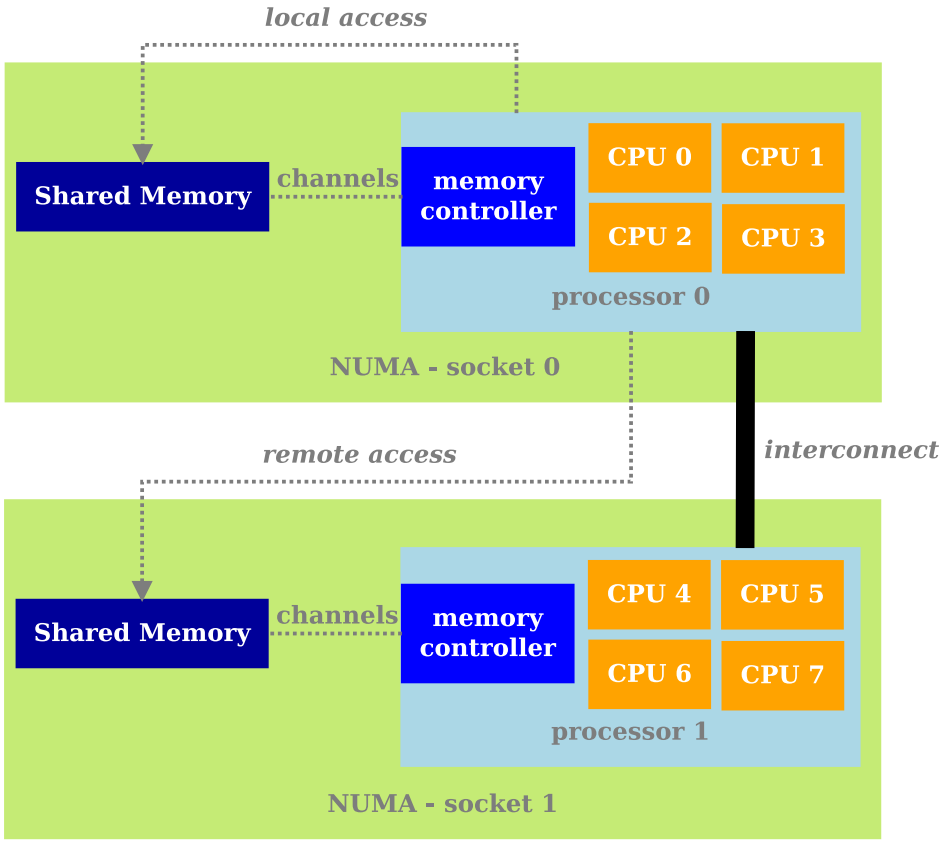


Figure 2.5: A Non-Uniform Memory Access System

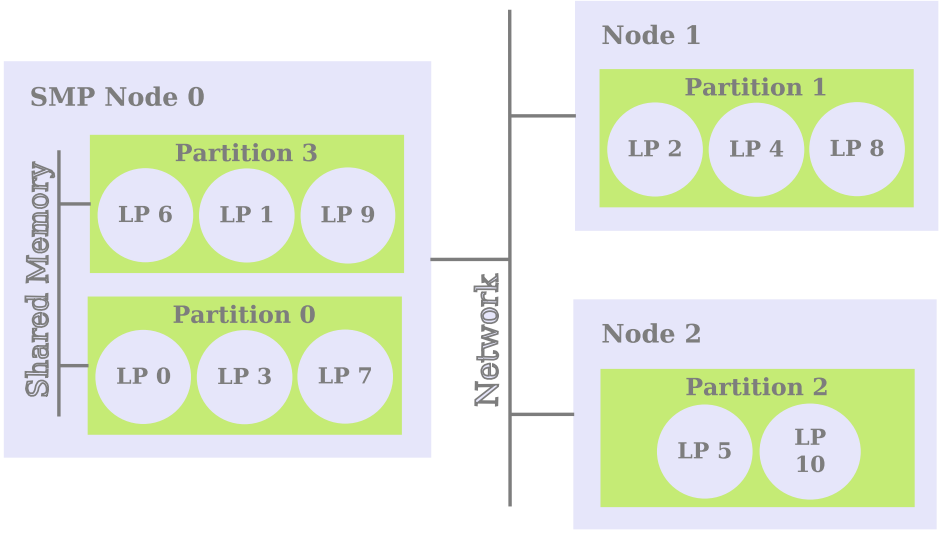


Figure 2.6: Beowulf Cluster

2.4 Communication in Parallel Systems

Any application capable of parallel and independent execution on multiple processes/threads may need to exchange information between these processes/threads. Processes/threads can communicate amongst themselves in either of these two ways:

- *Message Passing*: explicitly pass messages to each other using well defined message formats, or
- *Shared Memory*: shared data structures accessible to all processes/threads.

.

There are fundamental differences between these two communication methods and both have their strengths and weaknesses. Sections 2.4.1 and 2.4.2 explore these two communication models in further details.

2.4.1 Message Passing

Processes communicate only through serialized messages in a message passing system. These messages can be passed either in *synchronous mode* or in *asynchronous mode*. To enable the sender and receiver to communicate via serialized messages, the format of the messages must be pre-defined.

Synchronous message passing requires a specific ordering of the send/receive operations for each process so that the sender/receiver processes can operate together in synchronized fashion. Until a message is received by the receiver, a sender's send operation will remain blocked. Similarly, a receiver's receive operation will remain blocked until the sent message is fully received. These strict rules make every process follow a predictable and synchronized communication pattern. The whole system might suffer from a slow down since processes are not allowed to execute other operations while communication operations are ongoing.

Asynchronous message passing allows processes to continue with routine executions without blocking immediately after the start of send/receive operations. All pending operations are held in temporary queues and can be processed at any time and in any order. As a result, these processes do not have to follow a predictable communication pattern when communicating in asynchronous mode.

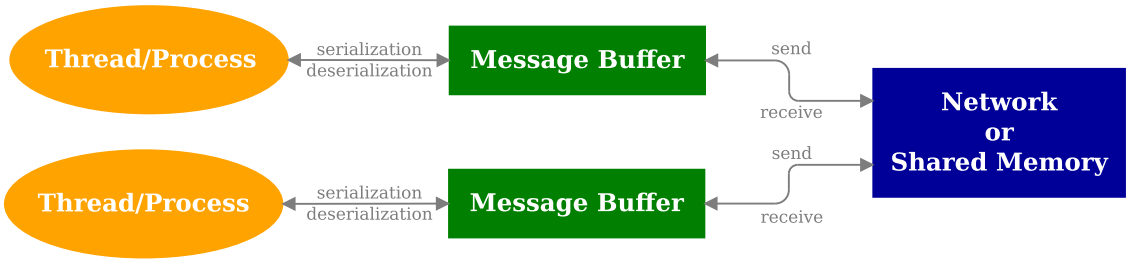


Figure 2.7: Message Passing Communication Model

If the workload can be partitioned sensibly to keep remote communication at a minimum, the number of cooperating processes can be scaled to essentially any size. This is the chief advantage of *message passing*. These processes can utilize separate address spaces on different machines while communicating over the local network. However, it takes time to serialize, deserialize, and copy messages. This latency is significantly higher when compared to the processing speed of modern processors. This latency is the chief disadvantage of remote messaging. There might be further delays when processes use an interconnection network because of extra communication latency. Fine-grained parallel applications need to send and receive lots of small messages which is not something message passing is ideally suited. Message passing is illustrated in Figure 2.7.

Message Passing Interface (MPI)

MPI [22] is an extensive and popular message passing standard popularly used for parallel applications. It supports both synchronous and asynchronous forms of communication and is a standard specified for developers and other MPI users. Several implementations of the MPI library exist, most popular among them being *MPICH* and *OpenMPI*.

2.4.2 Shared Memory

In parallel applications, it is possible for processes to communicate via shared data structures and share a common address space. A producer can be allowed to insert data directly into the data structures of a remote machine and the consumers can then remove this data for usage. Compared to message passing schemes, data transmission via this shared data structure scheme takes less time. Pointers can be also be used instead of copying large datasets to further improve the speed. To prevent multiple processes from corrupting the

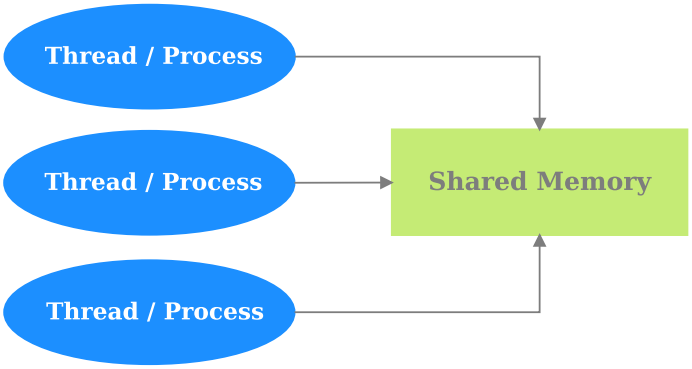


Figure 2.8: Shared Memory Communication Model

stored data by simultaneously performing read-modify-write operations on the same data, access to the shared data structures is protected usually using lock synchronization mechanisms. Locks protect entire sections of code that are executed by different processes when accessing the same data structures such as mutexes or semaphores. If too many processes contend for the lock simultaneously, the performance of shared memory data structures may be adversely affected. This makes it difficult to scale systems that use shared memory only as means of communication. A simple shared memory system is illustrated in Figure 2.8.

Chapter 3

Related Work

There are several popular implementations of Time Warp synchronized parallel simulation engines. An overview of some of the most popular ones is presented in this chapter.

3.1 Georgia Tech Time Warp (GTW)

Georgia Tech Time Warp [24] (GTW) is a general purpose Time Warp Simulator that is not used anymore. However, it is significant in that it was one of the most popular shared-memory PDES engine available. The shared memory multiprocessors such as the SparcStation and SGI PowerChallenge that *GTW* was designed for are now obsolete. Inspite of being out-dated, *GTW* set the template for most modern Time Warp simulators in use today. A single multi-threaded process runs a simulation model in *GTW*. Communication between threads is handled via shared data structures bound to a single processor.

A thread processes events from a statically allocated LP group in a simulation model. This ensures that the same processing core is used for processing events from a LP. Each thread has its own set of data structures to manage the distributed pending event set for its assigned LP group. The pending event set for each processor consists of three main data structures listed below [24]:

1. *Message Queue* is a linked list that holds positive messages for LPs. Each message is mapped to the processor that handles that LP. Tasks running on any processor can access this shared data structure. Access is usually synchronized via lock.

2. *Cancel Queue* is similar to *Message Queue* but, instead of positive messages, it holds negative messages (also known as anti-messages). This queue also requires a lock for synchronized access.

3. *Event Queue* is a composite data structure that is used to directly schedule events for processing. It holds both unprocessed and processed events. The event queue consists of two data structures, one each for processed and unprocessed events. A doubly linked list holds the processed events while a priority queue holds the unprocessed events. The priority queue can be configured by the user to be either a calendar queue or a skew heap.

Positive messages sent between LPs are inserted directly into the *Message Queue* while negative ones are directly inserted into the *Cancel Queue*. In order to process events, each thread first moves events from the *Message Queue* to the *Event Queue* and then processes the rollbacks. Messages from the *Cancel Queue* are removed next for event cancellations and any associated rollbacks are processed. The thread then processes one or more of the smallest events from the *Event Queue* and adds those events to the processed event list. The procedure mentioned above is repeated by all processors until the end of the simulation. Figure 1 presents the pseudo-code for *GTW's* main event processing loop.

while *Event Queue is not empty* **do**
    Transfer messages from Message Queue to Event Queue
    Process any rollbacks
    Remove anti-messages from Cancel Queue
    Cancel events and process associated rollbacks
    Remove one or more smallest events from unprocessed event pool in Event Queue
    process those events and move them to processed event pool

Algorithm 1: Event Processing in GTW [20, 24]

Threads can remove multiple events from the *Message Queues* and *Cancel Queues* in order to avoid frequent contention for access to these queues. This type of processing is called batch processing and the number of events in any batch represents the batch interval. All events in a batch are processed serially without consideration for rollbacks or cancellations.

In *GTW*, there is no need to send anti-messages explicitly since all communication is over shared memory. *GTW* calls its cancellation method *direct cancellation* because only a pointer to the event is needed for event cancellation. GVT calculation also becomes faster on a shared memory platform since shared

data structures can be used more effectively instead of messages passed between processors. This approach, however, limits the use of *GTW* to only a single multiprocessor machine, that too optimized for specific architectures.

The developer of a simulation model is responsible for partitioning the LPs among processors. This partitioning must be done during initialization of the simulation model. This is an unreasonable expectation. To effectively partition the LPs, the model developer would need to understand some the features of the underlying architecture such as the number of processors. Additionally, this static partitioning approach does not allow dynamic run-time balancing as there are no separate input queues for each LP. All unprocessed events for each processor are held within a single message queue.

3.2 Clustered Time Warp (CTW)

Clustered Time Warp [25] (CTW) employs a novel hybrid approach to processing events in *clusters*. Events within a LP group (or *cluster*) are processed sequentially while synchronization between different clusters is via the Time Warp mechanism. CTW was developed primarily to support digital logic simulation. Digital logic simulation tends to have localized computation within LP groups and is suited for the computational framework of *CTW*. In addition, some simulation models such as digital logic have events with fine computational granularity and lots of LPs. In a Time Warp simulator, this can cause significant increase in rollback count and memory footprint. Although *CTW* supports shared memory multiprocessors, it only uses shared memory with a custom message passing system. Thus, *CTW* is best suited for NUMA architectures and is not recommended for execution on a Beowulf Cluster.

Each LP cluster contains a timezone table, an output queue, and a set of LPs. Each LP has an input queue and a state queue. Based on events received from different LP clusters, the timezones (and timestamps) are recorded in the timezone table. A new timezone is added to the timezone table when an event is received from a remote cluster. Since anti-messages can only be sent between clusters and between LPs inside a cluster, a single output queue per cluster is sufficient.

CTW's rollback scheme is called *clustered rollback*. When a cluster receives a straggler event, rollback will occur for all intra-cluster LPs that have processed events with timestamps greater than the straggler event. *Local rollback*, which is an alternative to *clustered rollback*, allows the straggler event to be inserted

into the input queue of the receiver LP. The LP will trigger a rollback when it detects this straggler event. Even though *clustered rollbacks* may slow down computation by triggering rollbacks unnecessarily in some LPs, it eliminates the need to store processed events. This reduced memory footprint led *CTW*'s designers to prefer *clustered rollbacks*.

The timezone table in *CTW* is used to determine the frequency of state savings. Here the timezone of the last processed event is looked up before processing an event. The state is saved if this event belongs to a different timezone. This infrequent approach can broadly be classified into two categories, namely *local* and *clustered* checkpointing. In *local checkpointing*, all LPs save their state every time an event is processed in a new timezone, even if the event was received from a remote cluster. In *clustered checkpointing*, only the LP that receives an event from a remote cluster saves its state. A higher state saving frequency of the latter approach means more events must be saved for coast forwarding during state restoration. This increase in rollback computation and higher memory consumption was the reason why *CTW*'s designers preferred the *local checkpointing* approach.

3.3 Rensselaer's Optimistic Simulation System (ROSS)

Rensselaer's Optimistic Simulation System (ROSS) [26] is a general purpose simulator that started life as a re-implementation of *GTW*. Its capabilities were enhanced steadily over the years and now it can run both conservatively and optimistically synchronized parallel simulations as well as sequential simulations. It is widely used as a Timewarp-synchronized optimistic parallel simulator.

Although the event scheduling mechanism is similar to *GTW*, *ROSS* supports several priority queue implementations. There are also choices for algorithms used in fossil collection, state saving, and GVT calculation. A major difference between *GTW* and *ROSS* lies in the latter's use of processes instead of threads. *ROSS* uses MPI-based message passing instead of shared memory for inter-process communication.

ROSS, borrowing ideas from *GTW*, maps every LP to a process. Each process contains its own pending event set data structures and since these are not shared among processes, locks are unnecessary. Although similar to the data structures in *GTW*, *ROSS* uses different naming convention as mentioned below:

1. *Event Queue* for a process holds all the positive events for all LPs linked to that process. In addition, all

remote events, both positive and negative, are held in the *Event Queue*. The structure is implemented as a linked list and is analogous to the *Message Queue* in *GTW*.

2. *Cancel Queue* is similar to the data structure used in *GTW* but without the lock (which is unnecessary here). It is a linked list that holds negative events for all LPs for the corresponding process.

3. *Priority Queue* holds events in increasing order of timestamp. *ROSS* allows the user to configure the type of implementation, options available are Calendar Queue [15], heap [27], Splay Tree [28], or AVL tree [29]. The *Priority Queue* is analogous to the *unprocessed event queue* in *GTW*.

ROSS partitions the LPs on any process into groups called *Kernel Processes (KPs)* in order to reduce the time taken to fossil collect LPs. Similar to *clustered rollback* in *CTW*, all LPs in a *KP* undergo rollback and fossil collection together.

Instead of relying solely on traditional copy state saving to rollback LPs to a previous state, *ROSS* also uses *reverse computation* [30].

3.3.1 ROSS-MT

ROSS-MT [31] is a multi-threaded version of *ROSS*. Unlike the latter, *ROSS-MT* is optimized to use shared memory for inter-thread communication. As message passing using MPI is completely absent, all events are directly inserted into the *event queues*. *Event Queues* are further divided by possible senders in order to reduce the added contention on them. The memory management in *ROSS-MT* suits NUMA architecture.

3.4 WARPED

WARPED [32, 33] is a general-purpose discrete event simulator. It follows Jefferson's classic model unlike the Time Warp simulators that came before it. Here each LP has its own input, output, and state queues. WARPED was initially a process-based solution with communication via message passing only. With the development of multicore processors, in order to improve concurrent processing of events, each process was extended into multiple threads. The complexity of WARPED became unmaintainable after few years because multiple researchers kept adding new features and algorithms to it. Though configurable and modular in design, WARPED became too complex for new developers to learn and enhance.

This has led to the development of WARPED2, which is based on the design and architecture of WARPED. Chapter 4 provides a detailed overview of WARPED2.

3.5 The ROme OpTimistic Simulator (ROOT-Sim)

The ROme OpTimistic Simulator [34] (ROOT-Sim), a general purpose Time Warp simulator, shares several common characteristics with WARPED. Both use MPI-based message passing and can be classified as classic Time Warp implementations since each LP has its own *input*, *state* and *output* queues. ROOT-Sim is distinctly different from other Time Warp simulators when it comes to internal instrumentation. Memory usage can be optimized using *Dynamic Memory Logger and Restorer (DyMeLoR)*. *DyMeLoR* analyzes the performance of simulation models to understand which is a better fit — *copy-state saving* or *incremental state saving*. It can also transparently make a runtime switch between these two state savings strategies.

Committed and Consistent Global Snapshot (CCGS) is a service that ROOT-Sim introduced for transparently rebuilding the global snapshot of all LP states after each GVT calculation. During every GVT calculation, each LP has access to its portion of the global snapshot. This service allows any simulation model to implement its own custom global snapshot algorithm.

ROme simulator does not advocate the use of shared memory event processing pool for the threads on an SMP node [35, 36]. Instead, it relies heavily on a partitioned pending event structure with dynamic thread count in each kernel instance. Depending on the current workload, the number of threads can be scaled up or down.

In some of their recent papers [37–40], they have explored how all threads can fully share the workload of events by loosely coupling simulation objects and threads. Though this fully shared pending event pool will allow concurrent processing of any event, it is necessary to design parallel "insertion" and "dequeue" operations that are efficient. Loosely based on the Calendar Queue [15], they claim that this scalable lock-free event pool is accessible concurrently with *O(1)* amortized time complexity for both "insert" and "dequeue" operations.

Speculative processing and rollback techniques are currently used for *causality maintenance* by several Time Warp-based PDES systems. Pellegrini *et al* [41] question the effectiveness of this approach and propose an alternative preemptive approach which requires the CPU to be dynamically reassigned to past

unprocessed events (or operations such as *rollback*). According to the results they present, this approach allows the simulation to deviate less from the critical path by reacting promptly to the causal violations. This reduces the overall number of *causality violation* and is ideal for multi-core *x86-64* platforms.

Chapter 4

The WARPED2 Simulation Kernel

WARPED2 is a C++-based Time Warp synchronized parallel discrete event simulation kernel. It is extensively used for research on parallel simulations on multi-core processors and clusters. The kernel also provides a complete set of APIs that anyone can use to construct a simulation model. However, a fair amount of knowledge about Time Warp concepts is necessary at the beginning due to the low-level nature of some of the APIs (*e.g.,* state definition and copy constructors for the state). The WARPED2 kernel and model git repositories are publicly available at https://github.com/wilseypa/warped2 and https://github.com/wilseypa/warped2-models. WARPED2 supports two types of simulation, namely:

- sequential simulation, and
- Time Warp synchronized parallel simulation.

4.1 Conceptual Overview

In order to make WARPED2 easy to configure, extend and maintain, the simulation kernel uses a modular design. At startup, all components are configured and created individually. The sub-components are accessed through pointers. The kernel's event dispatcher supports both *Sequential* and Time Warp simulation. Sequential simulator is a fairly simple software as it contains only a single list of unprocessed events and does not require integration of any other component. On the other hand, the Time Warp event dispatcher

requires integration of several components. Each component provides specialized algorithms that deals with one of the following:

- event scheduling
- state saving
- cancellation
- GVT
- termination
- interprocess communication
- statistics

Figure 4.1 illustrates the WARPED2 implementation of Time Warp.

The components in Time Warp can be broadly classified as either **local** or **global**. *Local* components are responsible for controlling the node-specific activities of all LPs on that node, namely event processing, rollbacks and fossil collection. *Global* components, on the other hand, are responsible for cluster-wide control issues such as GVT, termination detection and calculation of statistics. Communication to all processes in a cluster environment is essential for determination of global state of the system.

4.1.1 Local Time Warp Components

The principle local components of WARPED2 are:

- The **Event Set** contains all *unprocessed* and *processed* events for the LPs on a node. The important data structures include:
  - *Input Queue* for each LP, and
  - *Schedule Queue* for events waiting to be processed soon.

The *Event Set* is responsible for storage and scheduling of unprocessed events for execution. It also processes rollbacks, and takes care of fossil collection for processed events. The *Event Set* has been designed for a multi-threaded environment and so the data structures that store pending events are designed for thread-safe concurrent or serialized access.

- The **Output Manager** is the module that holds all the *Output Queues* necessary for storage and tracking of outgoing events. WARPED2 supports only *aggressive cancellation* [10]. The *Output Manager* allows the kernel to do the following:
  - add newly generated events to its output queues,
  - dequeue events from output queues for processing a rollback, and
  - fossil collection of old output events to free up space on the output queues.
- The **State Manager** is the module that holds all the saved *LP states* inside its state queues. WARPED2

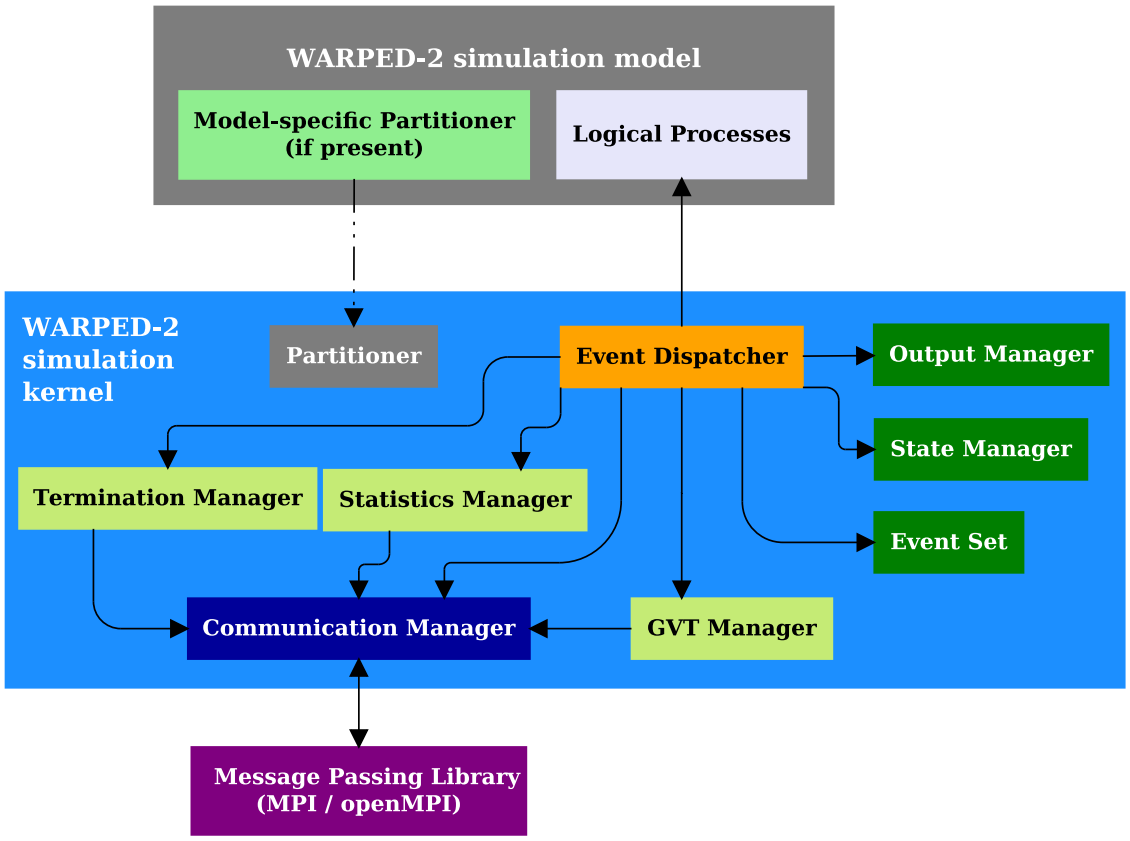


Figure 4.1: Time Warp Components in WARPED2

supports only *periodic state saving* [42]. The *State Manager* allows the kernel to do the following:

- saving the state of an LP,
- restore the state of an LP in case of a rollback, and
- fossil collection of old states in order to free up space in the state queues.

4.1.2 Global Time Warp Components

The major global components of WARPED2 are:

- The **GVT Manager** is the module responsible for tracking the global progress of any simulation. The progress of a simulation is measured in terms of *Global Virtual Time (GVT)* which can be calculated using either shared memory or message passing algorithms [20, 43, 44]. In WARPED2, the GVT is calculated via a hybrid approach:

  Step 1: multiple worker threads on each node regularly report the lowest timestamp for that node, and

  Step 2: nodes communicate their locally calculated lowest timestamp to other nodes for computation of the global progress as a minimum of these reported timestamps.

  WARPED2 currently supports the following two GVT calculation modes:

  - Synchronous [20] mode focuses on synchronized global reduction between all threads from all processes, and
  - Asynchronous [43] mode uses Mattern's message passing algorithm [43] and Fujimoto's shared memory algorithm [20].

  A more detailed explanation of the GVT algorithms of WARPED2 are available in [21].

- **Termination Manager** is the module in WARPED2 that initiates termination when it realizes all processes in the simulation have become inactive. Similar to the *GVT Manager*, the *Termination Manager* requires a hybrid approach involving worker threads and *Communication Manager*.

- **Statistics Manager** records local statistics on a distributed simulation. It also provides reduction methods to allow WARPED2 use the local statistics to compute and report consolidated statistics for the whole simulation.

4.1.3 Communication Manager

The *Communication Manager* is the module that facilitates connection between the global Time Warp components and the underlying message passing library. In WARPED2, the entire interprocess communication, including remote events sent to or received from another process, is channeled through the this Communication Manager. The manager accepts registration from any class that needs interprocess communication. This registration requires the class to inform the message type and provide a callback function for receiving events.

4.1.4 Partitioner

The *Partitioner* is a module responsible for dividing the logical processes of a simulation into groups (or partitions). At the time of initialization, the *Partitioner* is asked to create a defined number of partitions from a provided list of LPs using a partitioning technique. The WARPED2 kernel provides support for the following types of partitioning strategies:

- *Round Robin* partitioning is cyclic binning of LPs into different partitions. Figure 4.2 illustrates this algorithm.

- *Profile-Guided* partitioning is a network statistics driven partitioning strategy. Alt *et al* [17] studied how this partitioning can be done to minimize the number of remote messages sent in a distributed environment. They presented data that showed the effectiveness of *METIS* [45] as an effective partitioning tool for the LPs. The *profile-guided* partitioning in WARPED2 is done by partitioning a weighted LP network graph using *METIS*, where weight is the count of events (or messages) exchanged between any two LPs. Section 5.2.4 explains the *event bag scheduling* technique which relies on *modular communities* of LPs found by the *Louvain* partitioning algorithm [46].

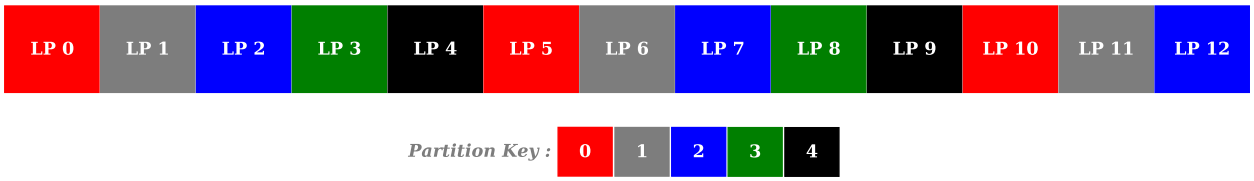


Figure 4.2: Round Robin Partitioning in WARPED2

- Users can define *Custom* partitioning strategies to suit the needs for the simulation model they are building.

Partitioning is an effective way to distribute workload across a parallel computing platform, especially if the simulation size is too large for any individual machine. Figure 2.6 shows how different *partitions* can be distributed on different nodes in a *Beowulf cluster*.

4.2 Journey of an Event

4.2.1 How are Events Ordered?

The implicit assumption for Time Warp is that every event is tagged with a totally ordered clock value based on virtual time [2]. This is essential to ensure that results from different simulation runs are deterministic [47]. Having a global tag for each event ensures the preservation of causal dependencies.
In order to ensure total ordering of events, WARPED2 uses the following 4-tuple scheme:

1. Receive Timestamp
2. Send Timestamp
3. Name of Sender LP
4. Generation

For practical purposes, the **Receive Timestamp** is the tagged clock value for each event. In situations where multiple events share the same *Receive Timestamp* value, the last three event parameters allow those events to be ordered.

Send Timestamp for virtual time systems is analogous to Lamport's logical clock [19] in real time distributed systems. It can ensure correct ordering of events as long as an LP sends only one event with a unique combination of *Send Timestamp*, *Receive Timestamp* and *Receiver LP*. Otherwise, strict ordering of events would not be preserved and WARPED2 forbids that.

Name of Sender LP is necessary when *Send Timestamp* fails to order all events correctly. Events received with the same combination of *Receive Timestamp* and *Send Timestamp* from different senders can be ordered using *Name of the Sender LP*. If this is not implemented, it is possible to get different results from different runs of the simulation [47].

Generation is a per-LP counter that keeps track of the number of events sent from a particular LP. In distributed memory systems, this counter value helps to differentiate between the same event which might get re-sent after a *rollback* [47]. Each event is tagged with a *Generation id* before being sent. This value is then used at the receiver's end for resolving any conflicts in the event order.

4.2.2 Pending Event Set

The set of events that are waiting to be processed is referred to as *Pending Event Set*. In WARPED2, each process maintains a pending event set for its dedicated set of LPs. This pending event set is logically separate from that of other processes. Events generated by a LP can either be sent *locally* to a LP present on the same process (or node) or can be sent to a LP on a *remote* process. *Local events* are inserted directly into the pending event set while *remote events* are inserted into a *remote event send queue*. The *manager thread* present on each process removes these *remote events* from the queue, forms *event messages* and sends these messages to the intended receiver process. The *manager thread* at the other end, on receiving an *event message*, unpacks this message to re-form the event and then inserts this event into the *pending event set* of the receiver. Each LP in the *pending event set* has its own *Input Queue* which temporarily holds all incoming events for that LP till they are processed. This queue is directly accessible to all threads and is protected by a lock.

The *Input Queue* holds both positive events and negative events (or anti-messages) and keeps all stored events sorted in increasing order at all times. In terms of sorting order, *anti-messages* have priority over their

positive counterparts. This mechanism ensures *anti-messages* are noticed first and, while processing it, the positive counterpart (if present in the *Input Queue*) can be *cancelled*. This helps to eliminate 'preventable' rollbacks [48]. The *Input Queue* only stores pointers to unprocessed events which ensures events are not unnecessarily copied. Event data replication will lead to an excessive use of available memory.

The *Schedule Queue* is a secondary data structure that is part of WARPED2's pending event set. The original design of WARPED2 assumes that this *Schedule Queue* is a *LTSF (Lowest TimeStamp First) Queue* (similar to an *Input Queue*) because it sorts lowest timestamped events from multiple LPs in increasing order. One event from each LP is *scheduled* into a common *Schedule Queue*. Scheduling an event here implies copying of event pointer into the *Schedule Queue*. The event is not removed from the *Input Queue* while it is *scheduled*. A worker thread "dequeues" the smallest event available in the *Schedule Queue* for processing. One or more *Schedule Queues* may be available for each process (or node). Since multiple worker threads can access a particular *Schedule Queue*, access to this data structure is serialized using a lock.

For each LP, it is necessary to keep track of the event that has been scheduled for the following reasons:

- It provides a way to identify whether the smallest event from an LP has been scheduled. When an event is into the *Input Queue* of an LP, it can be compared against the currently scheduled event. If it turns out that the newly inserted event is smaller, the new event can then be scheduled in place of the already scheduled event.
- It prevents multiple worker threads from processing events in the same LP. This might lead to an undetected case of *causality violation* causing out of order event commits to happen without corrections via rollback.

Figure 4.3 depicts the pending event set data structures used in WARPED2.

4.2.3 Processing of Events

At the start of simulation, each *Schedule Queue* is allocated one or more worker threads. Each thread processes events following the exact same procedure. A scheduled event is "dequeued" from a *Schedule Queue* but remains stored in the *Input Queue* until that event has been either processed or cancelled out. In

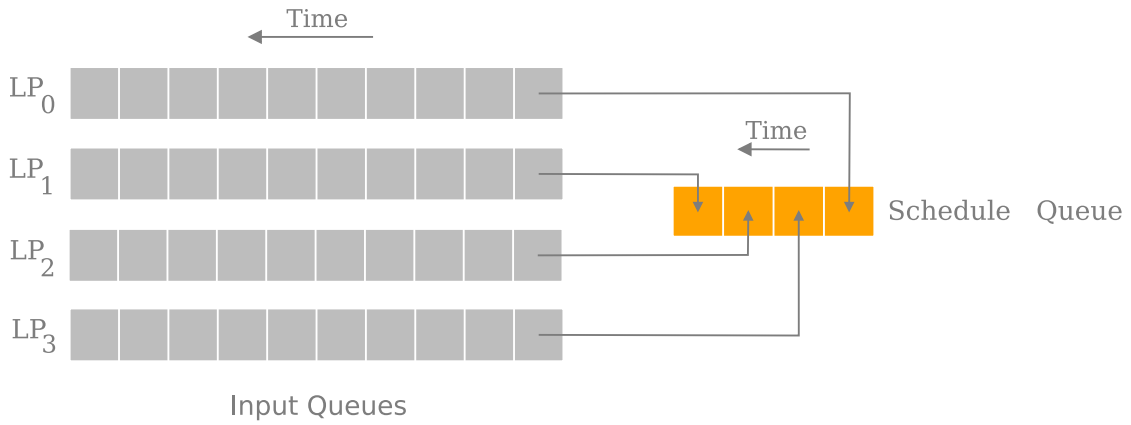


Figure 4.3: WARPED2 Pending Event Set

order to ensure the dequeued event is not a *straggler*, it is first compared against the last processed event for that receiving LP. If it turns out to be a straggler, rollback procedure is initiated for that LP. A *straggler* may also be an *anti-message* if its *positive* counterpart has already been processed. Due to the manner in which WARPED2 cancels positive event in *Input Queue* on receiving an *anti-message*, it can be assumed that if the scheduled event is an *anti-message*, it is a *straggler* and the LP needs to rollback. If the scheduled event is a positive one and not a *straggler*, then it is processed normally and the *LP State* is saved. Any new events that were generated are sent to the intended LPs. The recently processed event is then stored in the *Processed Queue* and a new event is scheduled from the LP into the *Schedule Queue*. Algorithm 2 summarizes the worker thread event processing loop. In general, the algorithm proceeds as follows: before an event is inserted into the *Input Queue*, it is compared to the event that is currently scheduled for that LP. The new event is immediately scheduled if the corresponding LP has currently scheduled event or if the new event is smaller than the currently scheduled event. This prompt initiative prevents a rollback from occurring.

4.2.4 Rollbacks & Cancellation

According to Jefferson [2], three data structures are necessary for the rollback and event cancellation process, namely: the *Input Queue*, the *Output Queue*, and the *State Queue*. The data structures used in WARPED2 are based on Jefferson's description. But while Jefferson's approach requires the *Input Queue* to hold both unprocessed and processed events, separate data structures are maintained in WARPED2 for ease of schedul-

```
while signal to terminate not detected do
    e ← getNextScheduledEvent()
    LP ← receiver of e

    if e < last processed event for LP then
        Rollback LP

    if e is an anti-message then
        Cancel event with e (if possible)
        Schedule new events for LP
        continue

    Process event e
    Save state of LP
    Send newly generated events to their destinations

    Move e to Processed Queue
    Replace scheduled event for LP with an event from the Input Queue (if available)
```

Algorithm 2: WARPED2 Event Processing Loop

ing.

A 3-tuple value is stored for each entry in the *State Queue*. The tuple attributes are:

- The address of the memory location holding the saved copy of the LP state.
- The address of event that produced the LP State. If there is a rollback, this event will be compared against the straggler event for identifying the restore point. It will also be compared against the GVT value during fossil collection.
- In order to ensure deterministic results, state of the LP's random number generator is saved. A linked list holds these states and are restored during a rollback.

Each entry into the *Output Queue* is a tuple similar to that in a *State Queue*. The tuple attributes are:

- address of the source event, and
- address of the sent event.

In order to figure out which sent events need to be cancelled out via anti-messages in case of a rollback, the source event is compared against the straggler event. *Processed Queue* of an LP holds the addresses of events processed from that LP.

Chapter 5

Optimizing WARPED2's Pending Event Set

This chapter is a continuation of the discussion on *Pending Event Set* started in Section 4.2.2. As mentioned in that section, the *Schedule Queue* is a data structure that stores lowest unprocessed pending events from all of the LPs in sorted order. Worker threads contest for dedicated access to this shared data structure in order to "dequeue" events for processing. Conventional designs visualize this *Schedule Queue* as a priority queue where events are arranged in increasing order of their timestamp. However, in this chapter, alternative ideas on sorting data structures and scheduling techniques will be presented to show it is not always necessary to fully sort all pending events. The remainder of this chapter is organized as follows. Section 5.1 presents the different data structures that can be used as *Schedule Queue* in WARPED2. Section 5.2 presents some scheduling techniques which, when used in combination with the data structures discussed in Section 5.1, can boost the performance of the WARPED2 simulator.

5.1 Data Structures for Scheduling Pending Events

Ronngren *et al* [49] proposed that all unprocessed and processed events along with each event's execution status can be stored inside a common data structure called *Linear List*. The *Linear List* is a doubly linked list [50] that is simple to implement. Though rollbacks and fossil collection become more efficient when using *Linear List*, the authors [49] state that *Linear List* struggles to insert events into a large event pool. As an alternative to linear list, they suggest that the improved skew heap [51] is a promising alternative.

Prasad *et al* [52, 53] proposed that, for medium to coarse-grained simulations, parallelized Calendar

Queues [15] can be used. Each processor maintains its own separate Calendar Queue. The result from their experiment showed that arranging the pending events into queues locally does not adversely impact the balance of the simulator when compared to the standard global queue-based arrangement.

Santoro *et al* [54] proposed a modified version of the Calendar Queue that uses an array and a hierarchical bitmap. They show that this modification allows the priority queue to access its stored events in constant-time with low overhead.

In WARPED2, several different data structures have previously been explored for managing the pending event set. Similar to the aforementioned research on priority queues, a reduction of the overhead involved in sorting events remains a major motivating factor. However, the advent of multi-core processors and the subsequent analysis of the contention issues involved in using shared data structures has raised concerns about the current design. In Sections 5.1.1, 5.1.2, 5.1.3 and 5.1.4 several different data structures are explored for creating an effective *Schedule Queue* in WARPED2.

5.1.1 STL MultiSet and Splay Tree

The *C++ STL MultiSet* and *Splay Tree* are both well-known tree-based implementations of a priority queue and are used extensively in different applications. *STL MultiSet* is a sorted multiple associative container present in the Standard Template Library [55] which is similar to a *Set* container but, unlike the latter, allows multiple instances of any element. It permits lookup, insertion, and removal of an element in *O(log n)* amortized time and is ideal for quickly verifying whether an element is present inside the container. The STL MultiSet has been implemented using *Red-Black Tree* [56], a self-balancing binary search trees and supports bidirectional iterators.

Similar to the *STL MultiSet*, the *Splay Tree* [28] is also a variation of self-adjusting binary search tree that allows for quick access to recently accessed elements. It can insert, lookup, and remove an element in *O(log n)* amortized time. The *Splay Tree* used in WARPED2 is a faithful implementation of the original data structure described in [28].

5.1.2 Ladder Queue

The *Ladder Queue* [8] is a priority queue which utilizes the bucket-based sorting philosophy of a Calendar Queue [15]. In case of a Calendar Queue, each bucket stores events within a certain time-window (or month) and the entire data structure must sometimes be resized at regular intervals in order to balance the changing range of event timestamp across the buckets. The Ladder Queue avoids the need for the resizing of buckets by dynamically splitting only the bucket with earliest event timestamps into multiple buckets once the number of events stored in that particular bucket exceed a pre-defined threshold. Figure 5.1 illustrates the principle components of a Ladder Queue.

Initially the Ladder Queue data structure is empty. Incoming events are inserted into the *Top* structure without sorting. The minimum and maximum timestamp values for events stored inside Top are updated (if needed) when any new event is inserted there. On receiving the first "dequeue event" request, the events in Top are transferred to $Rung_1$. The total bucket count in $Rung_1$ is configurable dynamically. The time window of events transferred from Top is uniformly partitioned into all the buckets available in $Rung_1$. The buckets store events without sorting in $Rung_1$ and there is an upper threshold for the number of events that can be stored in each bucket. When the number of events in the first non-empty bucket of the current rung exceed this threshold, a new lower rung is created. Events from the first non-empty bucket are then transferred to this new rung by splitting the bucket's time window uniformly across all buckets in the new rung. Figure 5.1 illustrates how this process works. Here events from the first non-empty bucket in $Rung_1$ is transferred to $Rung_2$ by splitting the time window of the bucket uniformly. Thus, the bucket size (defined as the number of elements in the bucket) of each bucket in $Rung_2$ is a sub-range of the bucket size in $Rung_1$.

The final step in the "dequeue event" process is transfer of the first non-empty bucket (which holds events with the smallest timestamps) to *Bottom*. The Bottom is a priority queue which sorts events in increasing order of their timstamp. The "dequeue" operation is then able to pull the smallest event stored in Bottom. Bottom then holds the smallest events available to the *Ladder Queue*. Events can be dequeued from Bottom until it becomes empty. Any new dequeue request at this point initiates another transfer of events from the rungs. The first non-empty bucket of the lowest available rung is transferred to the Bottom. When the events in the rungs and Bottom of the ladder are exhausted, these ladder elements are replenished by transferring events from the Top. Algorithm 3 explains how events are dequeued from the Ladder Queue.

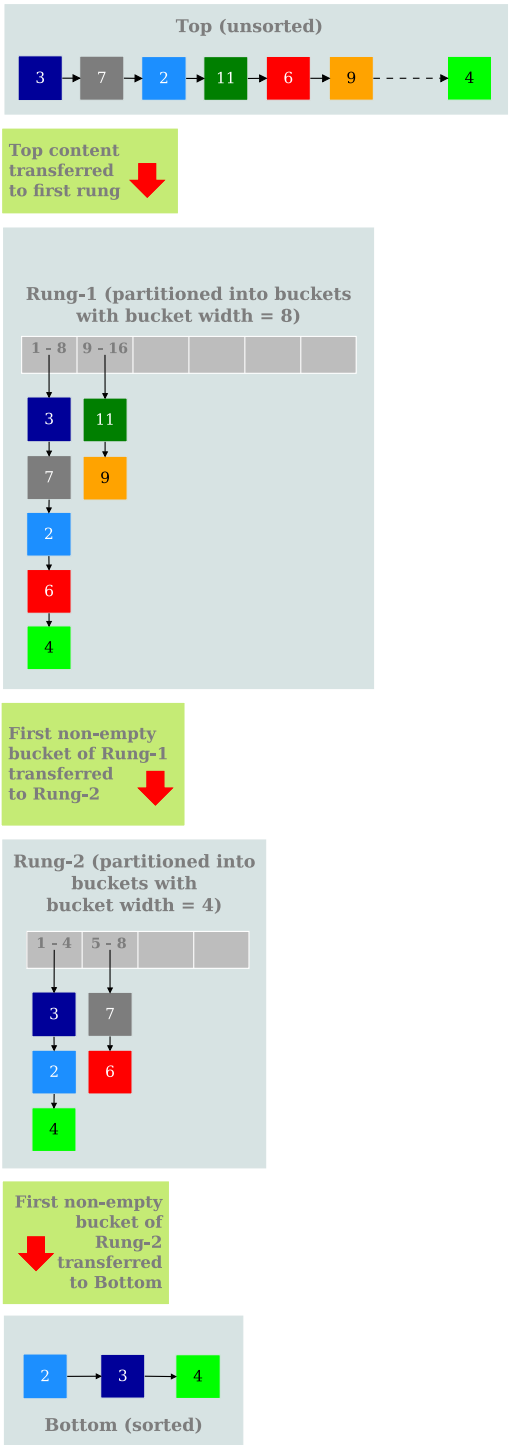


Figure 5.1: The Ladder Queue Structure

After the initial events to the Ladder Queue are pulled from Top into the ladder rungs, any new incoming events are inserted into that element of the Ladder Queue (Top, Rungs, or Bottom) based on the value of its timestamp as consistent with the current timestamp ranges contained in those elements. That is, if an event's timestamp lies within the time window of any rung, it is inserted into a specific bucket on that rung; the event is inserted into Bottom if the timestamp is lower than any of the available rungs; and finally any incoming event with timestamp greater than $Rung_1$'s time window is inserted into *Top*. Algorithm 4 explains the "insert event" operation in details.

if *Bottom not empty* **then**
    event $e \leftarrow$ dequeue event from head of $Bottom$
    return e

if *Rung(s) exist* **then**
    Find first non-empty bucket k in the lowest available rung x
    Transfer events from bucket k in rung x to $Bottom$
    Delete rung x if it has no non-empty buckets left
    event $e \leftarrow$ dequeue event from head of $Bottom$
    return e

/ Transfer events from Top into Rung(s) and Bottom */*

$Rung[1].\texttt{bucketWidth} \leftarrow \frac{Top.\texttt{maxTS} - Top.\texttt{minTS}}{Top.\texttt{size}}$

$Rung[1].\texttt{minTS} \leftarrow Top.\texttt{minTS}$
Transfer Top into $Rung[1]$
$Top.\texttt{minTS} \leftarrow Top.\texttt{maxTS}$
Find first non-empty bucket k in $Rung[1]$
Transfer events from bucket k in $Rung[1]$ to $Bottom$
Delete $Rung[1]$ if it has no non-empty buckets left

event $e \leftarrow$ dequeue event from head of $Bottom$
return e

Algorithm 3: LADDER QUEUE Dequeue Operation

if $e \rightarrow$ `timestamp()` $\geq Top$.`minTS` **then**
    Insert event e into tail of Top
    return

while $e \rightarrow$ `timestamp()` $< Rung[x]$.`minTS` **do**
    x++

if $x \in$ *valid rung* **then**

    bucket $k \leftarrow \frac{e \rightarrow \texttt{timestamp()} - Rung[x].\texttt{minTS}}{Rung[x].\texttt{bucketWidth}}$

    Insert event e into tail of bucket k of rung x
    return

if $Bottom$.`size` $> Bottom$.`sizeThreshold` **then**

    Create new rung whose index is y
    Transfer events from $Bottom$ to $Rung[y]$

    / Insert event e into the Rung[y] */*

    bucket $k \leftarrow \frac{e \rightarrow \texttt{timestamp()} - Rung[y].\texttt{minTS}}{Rung[y].\texttt{bucketWidth}}$

    Insert event e into tail of bucket k of rung y
    return

Insert event e into $Bottom$
return

Algorithm 4: LADDER QUEUE Insert Operation

What is exciting about the Ladder Queue?

From a conceptual standpoint, the Ladder Queue is able to store events from different *epochs*[1] separately in its different sub-structures. Events from one epoch (whose timestamps fall in the range t and $t + \Delta t$) are held in the Bottom and Rung elements while events from the next epoch (timestamps above $t + \Delta t$) are held in the Top element. Any incoming event is inserted into one of these three Ladder Queue elements based on its timestamp. When there are no more events left inside Bottom and Rung(s), a "dequeue" operation moves the ladder queue to its next epoch. The events in Top are then transferred to the Rung(s) and Bottom. The time window of an epoch (t to $t + \Delta t$) is defined by the minimum and maximum timestamp of events pulled from Top. The *Calendar Queue* [15] can also coarsely sort events into buckets based on time window (or epoch) but needs manual intervention to re-evaluate the time window's Δt interval. Manual intervention is not needed in Ladder Queue because its inherent design characteristics force the time window to be split whenever the event count in a Rung bucket or Bottom exceeds a certain threshold. Through experimental evaluation, Tang *et al* [8] proposed that the ideal value for this threshold is *50*. This setting will be revisited in the experimental assessment section of this dissertation (Chapter 6).

Is it necessary to split Bottom when its event count exceeds threshold?

Algorithm 4 shows how the events in the Bottom element are split between Rungs and Bottom when Bottom's event count exceeds a specified threshold. In a Ladder Queue, the stimulus for all dynamic adjustments to time window is *event count*. However, in the Pending Event Set of Time Warp, it turns out that event count in this context is insignificant. The focus here is to group all events which are within a certain time window. While dynamic splitting of time windows in the Ladder Queue is essential for scheduling an optimal group of events for execution, the hypothesis of this dissertation is that over-reliance on event count can lead to excessive and unnecessarily time-wasting splits. The time window splits and transfer of events from Top to Rungs and between Rungs are computationally not so expensive when compared to the split of Bottom and transfer of events from Bottom to the lowest Rung. Algorithm 5 shows how the "insert event" operation can be modified to eliminate the need for splitting the Bottom in case of an event overflow. This is the "insert event" operation used in Ladder Queue implementation of WARPED2.

[1] Each pull of events from Top into the Rungs and Bottom of the ladder queue is termed a new epoch.

if $e \rightarrow$ `timestamp()` $\geq Top$.`minTS` **then**
    Insert event e into tail of Top
    return

while $e \rightarrow$ `timestamp()` $< Rung[x]$.`minTS` **do**
    x++

if $x \in$ *valid rung* **then**

    bucket $k \leftarrow \frac{e \rightarrow \texttt{timestamp}() - Rung[x].\texttt{minTS}}{Rung[x].\texttt{bucketWidth}}$

    Insert event e into tail of bucket k of rung x
    return

Insert event e into $Bottom$
return

Algorithm 5: LADDER QUEUE Modified Insert Operation

5.1.3 Ladder Queue with Unsorted Bottom

Extending the narrative in Section 5.1.2, the Ladder Queue [8] structure can be modified for use as a partially sorted priority queue. Since the Ladder Queue efficiently groups events within a small time interval, Bottom most likely contains events that are causally independent. Any Time Warp-synchronized PDES simulator greedily processes events and has the ability to recover from causal violations (events processed out-of-order in an LP) if detected within a reasonable time interval. Under ideal circumstances, events inside Bottom would be causally independent and it should be possible to process them without the need to sort them based on their timestamp. Replacing the priority queue in the Bottom structure with any unsorted container should allow the simulator to process events without incurring frequent rollbacks and also save valuable time that otherwise would have been wasted on sorting. Even though *Unsorted Bottom* may not be an effective event scheduling strategy for a wide spectrum of simulation models, results presented in [3] indicate that it is suited for Time-Warp compatible models.

Similar to the modifications proposed to the "insert event" operation in Section 5.1.2, Bottom is not split any further when the event count there exceeds a specified threshold. Separate algorithms for "insert" and "dequeue" event are not presented in this section since it is mostly similar to Algorithms 5 and 3 in Section

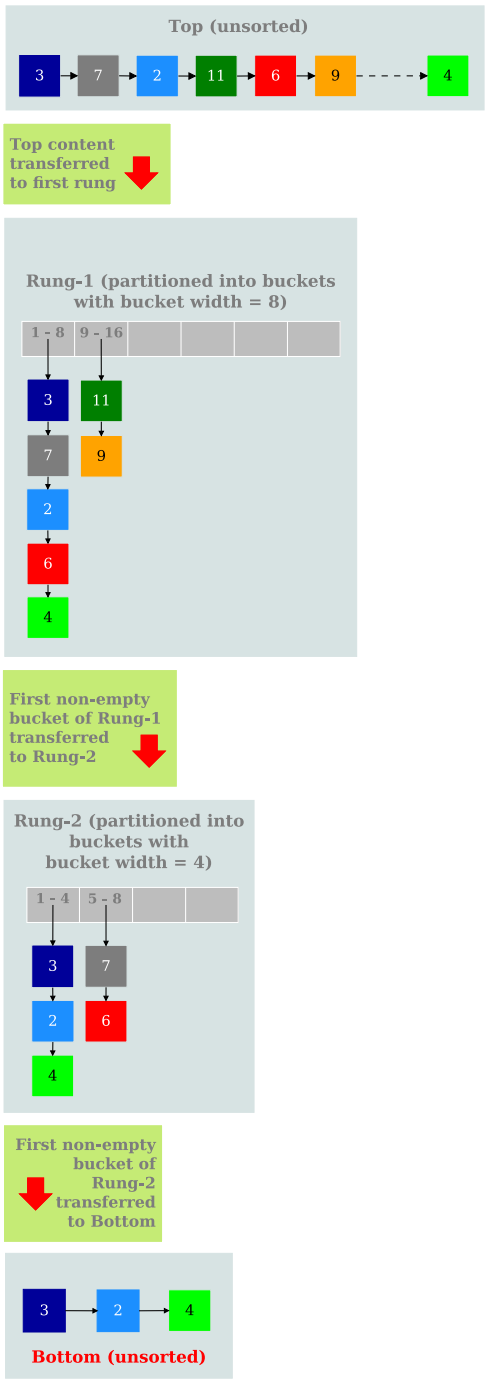


Figure 5.2: LADDER QUEUE with Unsorted Bottom

5.1.2.

How to track the minimum event timestamp for GVT calculation?

Section 4.1.2 gives an overview of the *GVT* estimation algorithm of WARPED2. In order to track the progress made by any individual node, it is necessary to track the lowest timestamp of events currently present in the *Schedule Queue(s)* on that node. If the Schedule Queue is a fully sorted priority queue (such as STL MultiSet, Splay Tree or Ladder Queue), each "dequeue event" operation is sufficient to figure out what is the smallest event present inside the Schedule Queue. However, for *Ladder Queue with Unsorted Bottom*, the "dequeue event" operation is not a reliable source for reading the lowest timestamp. This is because the Bottom structure is not sorted. In order to track the minimum timestamp, a record is kept of the minimum timestamp whenever a bucket is transferred from the lowest rung to the Bottom. This record is updated every time the Bottom runs out of events and another bucket is transferred to Bottom. This record is also updated if the "insert event" operation inserts an event into Bottom whose timestamp is smaller than the recorded minimum timestamp.

5.1.4 Ladder Queue with Lock-free Unsorted Bottom

In WARPED2, locks (Mutex [57] or Ticket Lock [58]) regulate access to the shared Schedule Queue(s). Algorithm 10 shows that `lockScheduleQueue()` and `unlockScheduleQueue()` are used to regulate access to the Schedule Queue when worker threads try to access it for dequeuing and inserting events. This coarse-grained locking strategy works efficiently for concurrent access to priority queues which do not have any hierarchical internal arrangement. However, when hierarchically-structured Ladder Queue with Unsorted Bottom is used as the Schedule Queue, a fine-grained locking strategy may improve the efficiency of concurrent operations on the queue. Based on the hypothesis proposed in Section 5.1.3 that eliminates the need to sort pending events inside Bottom, a *Lock-Free Unsorted List* [59–62] could serve as Bottom while the rest of the Ladder Queue (Top and Rungs) is protected by a common lock.

Survey of the literature available on non-blocking and lock-free data structures reveals several options for lock-free lists and queues [59–67]. Among these choices, a promising option is the `LFList`, a lock-free list developed by Zhang *et al* [62]. The LFList is easy to adopt into the Ladder Queue library but it

lacks a "dequeue" mechanism. A modified version of the LFList, proposed by Gupta *et al* [3], provides this dequeue functionality and also eliminates features which are unnecessary for WARPED2. The LFList uses backward link (`prev`) and thread ID (`tid`) which are relevant for doubly linked lists and for a wait-free queue derivative respectively. Such features are an overkill for unsorted Bottom in the Ladder Queue and requires only a singly-linked list to store unsorted events within a small time window. Preliminary results presented in [3] highlights the potential performance benefits of this approach.

The Ladder Queue (shown in Figures 5.1 and 5.2) allows three basic operations for manipulation of the pending event set, namely:

1. *Insert* a newly scheduled event from an Input Queue to the Schedule Queue,
2. *Dequeue* a scheduled event from the Schedule Queue for processing, and
3. *Erase* a scheduled event from the Schedule Queue when a straggler or anti-message is detected.

In the default design of WARPED2, a Schedule Queue lock makes these operations thread-safe for each Schedule Queue. As the number of worker threads per Schedule Queue increases, contention for this lock also increases. A lock-free list, on the other hand, uses compare-and-swap (CAS) instructions and can scale significantly better than a lock-protected queue as the number of threads increase. As mentioned above, Bottom can be implemented as a lock-free list but a common lock still needs to protect the internal Ladder structures, namely Top and Rung(s). This arrangement will allow threads to atomically dequeue from Bottom while another thread can acquire the lock in parallel and insert an event into Top or Rung(s) and re-organize the buckets. The set of Schedule Queue operations needed for a Ladder Queue with Lock-free Unsorted Bottom can be further simplified as follows:

1. If a newly scheduled event needs to be *inserted* into the lock-free Bottom, it can be inserted at the head of the list using one CAS operation. Threads will still have to contest for access to the common lock if this new event has to be inserted into Top or Rungs instead of Bottom.
2. A scheduled event can be *dequeued* from head of the lock-free Bottom using one CAS operation. If the Bottom is empty, threads will need access to the common lock for pulling events down from Top or Rungs into Bottom. The transfer of events from the chosen rung bucket to Bottom will be done using lock-free operations.

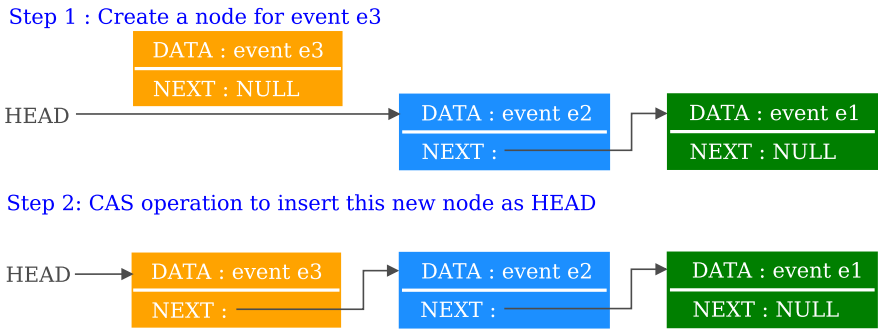


Figure 5.3: Insert Operation on Lock-Free List

3. *Erase* is not a necessary operation for Time Warp. The simulation can rollback to a consistent state if a scheduled event is not immediately removed from the Schedule Queue when a straggler or anti-message is detected. The erase operation is a relatively complex operation for lock-free lists and this complexity is avoidable for Time Warp-synchronized simulators at the expense of delayed rollbacks of longer length.

Lock-Free List in WARPED2

The Lock-Free List used to create the *Unsorted Bottom* for the Ladder Queue is a simple singly-linked list. Each node on this list has two attributes: `DATA` for storing the event and `NEXT` for storing the address of the next node on that list. The lock-free list only supports "insert" and "dequeue" operations on the `HEAD` of the list. Figures 5.3 and 5.4 illustrate these operations. Algorithms 6 and 7 provide further details.

Create a new node x
$x \rightarrow$DATA $\leftarrow$ event e

node $m \leftarrow$ HEAD
$x \rightarrow$NEXT $\leftarrow m$
while `CAS (&HEAD, &m, x)` $\neq$ `SUCCESS` **do**
  $m \leftarrow$ HEAD
  $x \rightarrow$NEXT $\leftarrow m$

Algorithm 6: Lock-Free List Insert Event

Algorithms 8 and 9 show how the "insert" and "dequeue" operations will work when a Lock-Free List is used to create a Ladder Queue with Lock-Free Unsorted Bottom. The `lock()` and `unlock()` functions acquire and release the common lock respectively and protects access to Top and any available Rungs.

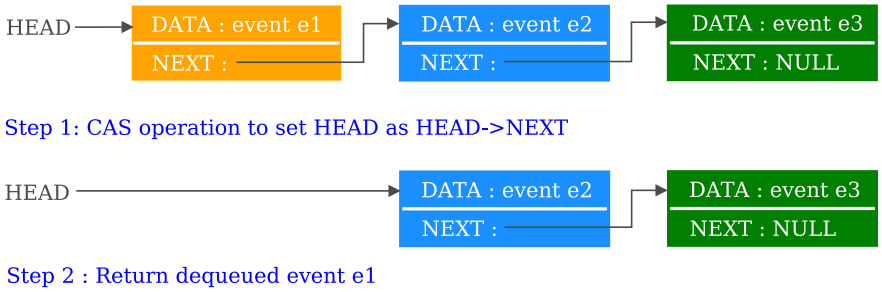


Figure 5.4: Dequeue Operation on Lock-Free List

node $m \leftarrow$ HEAD
if $m \in$ *NULL* **then**
  return NULL

while CAS (*&HEAD, &m, m→NEXT*) $\neq$ SUCCESS **do**
  $m \leftarrow$ HEAD
  if $m \in$ *NULL* **then**
    return NULL
return $m \rightarrow$DATA

Algorithm 7: Lock-Free List Dequeue Event

```
lock()
if e → timestamp() ≥ Top.minTS then
    Insert event e into tail of Top
    unlock()
    return

while e → timestamp() < Rung[x].minTS do
    x++

if x ∈ valid rung then

    bucket k ← (e→timestamp() − Rung[x].minTS) / Rung[x].bucketWidth

    Insert event e into tail of bucket k of rung x
    unlock()
    return
unlock()

Insert event e into lock-free Bottom
return
```

Algorithm 8: LADDER QUEUE Lock-Free Insert Operation

event $e \leftarrow$ dequeue from lock-free $Bottom$
if $e \in$ *valid event* **then**
    return e

`lock()`
if *Rung(s) exist* **then**
    Find first non-empty bucket k in the lowest available rung x
    Transfer events from bucket k in rung x to $Bottom$
    Delete rung x if it has no non-empty buckets left
    `unlock()`

    event $e \leftarrow$ dequeue event from lock-free $Bottom$
    return e

/ Transfer events from Top into Rung(s) and Bottom */*

$Rung[1].\texttt{bucketWidth} \leftarrow \frac{Top.\texttt{maxTS} - Top.\texttt{minTS}}{Top.\texttt{size}}$

$Rung[1].\texttt{minTS} \leftarrow Top.\texttt{minTS}$
Transfer Top into $Rung[1]$
$Top.\texttt{minTS} \leftarrow Top.\texttt{maxTS}$
Find first non-empty bucket k in $Rung[1]$
Transfer events from bucket k in $Rung[1]$ to $Bottom$
Delete $Rung[1]$ if it has no non-empty buckets left
`unlock()`

event $e \leftarrow$ dequeue event from lock-free $Bottom$
return e

Algorithm 9: LADDER QUEUE Lock-Free Dequeue Operation

5.2 Techniques for Scheduling Pending Events

The *Pending Event Set* in WARPED2 uses a two-level hierarchical arrangement. Figure 4.3 illustrates this design and Section 4.2.2 provides detailed explanation for the different different data structures that combine together to form the *pending event set*. However, Algorithm 2 only provides a simplistic overview of how events are processed in WARPED2. It does not provide any details about how locks are implemented to serialize access by multiple threads to the *Input Queue* and *Schedule Queue*. Algorithm 10 presents the missing details which will be needed to continue the discussion in later sections of this chapter.

while *signal to terminate not detected* **do**
- `lockScheduleQueue()`
- $e \leftarrow$ `getNextScheduledEvent()`
- `unlockScheduleQueue()`

- $LP \leftarrow$ receiver of e

- `reportMinTimeForGVT(` *thread_id* , $e \rightarrow$ `timestamp() )`

- **if** $e <$ *last processed event for* LP **then**
  - Rollback LP

- **if** e *is an anti-message* **then**
  - Cancel event with e (if possible)
  - Schedule new events for LP
  - **continue**

- Process event e
- Save state of LP
- Send newly generated events to their destinations

- Call `lockInputQueue()` for the LP

- Move e to Processed Queue

- `lockScheduleQueue()`
- Replace scheduled event for LP with an event from the Input Queue (if available)
- `unlockScheduleQueue()`

- Call `unlockInputQueue()` for the LP

Algorithm 10: WARPED2 Event Processing Loop (Detailed)

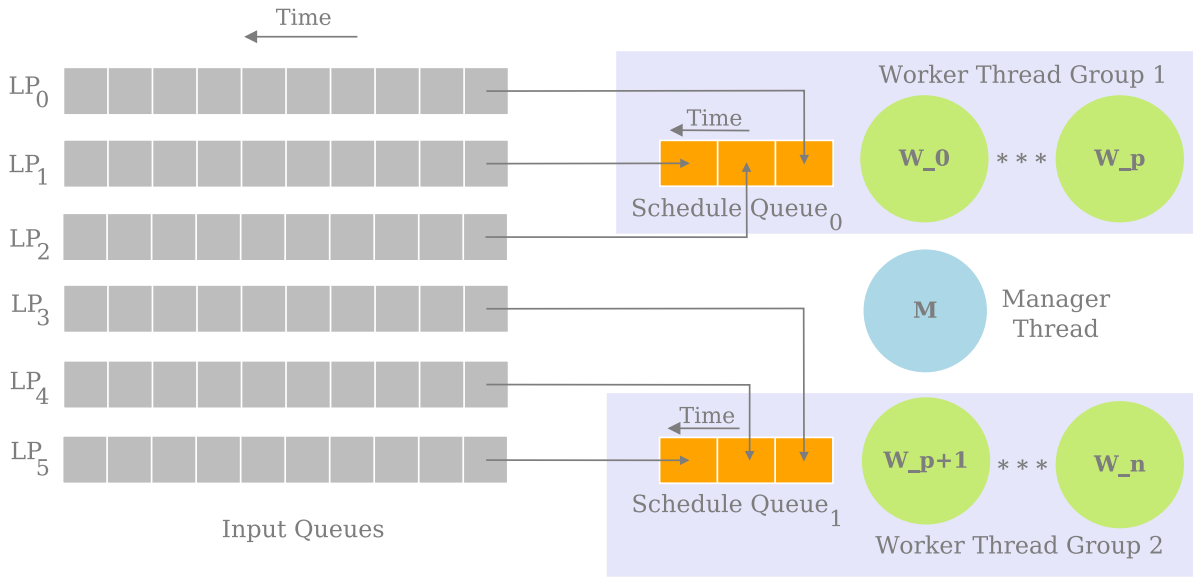


Figure 5.5: Multiple Schedule Queues in WARPED2

As shown in Algorithm 10, `lockInputQueue()` and `lockScheduleQueue()` protects access to the Input Queues and the Schedule Queue respectively. Each thread reports the timestamp of the event it is currently processing using `reportMinTimeForGVT()` for *GVT* estimation.

Should WARPED2 process pending events in *groups* or in *parallel* instead of one at a time?

Due to the rapid growth in number of on-chip cores, contention for shared data structures has increased [68]. Multiple threads attempt to access the *Pending Event Set* data structures for scheduling the next event for execution. Event executions are generally fine-grained in parallel simulation. This quickly leads to non-trivial contention for the pending event set. If threads are provided the option to process events in either groups or in parallel, they will spend less time in waiting for access to these shared data structures.

Section 5.2.1 describes a "divide-and-conquer" strategy of partitioning the *pending event set* into multiple *Schedule Queues* in order to boost parallel processing of pending events. Sections 5.2.2, 5.2.3 and 5.2.4 proposes techniques for scheduling pending events in groups.

5.2.1 Multiple Schedule Queues

In order to reduce the contention for common *Schedule Queue* on a multi-core processing platform, Dickman *et al* [6] proposed the use of multiple *Schedule Queues* in WARPED2. The *Input Queues* are split into groups

where each LP group has its own *Schedule Queue* and *Schedule Queue Lock*. The thread pool is also split into groups with each *Schedule Queue* having access to its own private pool of threads. Algorithm 11 shows how events can be processed using this arrangement.

while *signal to terminate not detected* **do**

$schedule\_queue\_id \leftarrow$ `getScheduleQueueId`($thread\_id$)
`lockScheduleQueue`($schedule\_queue\_id$)
$e \leftarrow$ `getNextScheduledEvent` (***schedule_queue_id***)
`unlockScheduleQueue`($schedule\_queue\_id$)

$LP \leftarrow$ receiver of e

`reportMinTimeForGVT`($thread\_id$, $e \rightarrow$ `timestamp()`)

if $e <$ *last processed event for* LP **then**
  Rollback LP

if e *is an anti-message* **then**
  Cancel event with e (if possible)
  Schedule new events for LP
  continue

Process event e
Save state of LP
Send newly generated events to their destinations

Call `lockInputQueue()` for the LP

Move e to Processed Queue

`lockScheduleQueue`($schedule\_queue\_id$)
Replace scheduled event for LP with an event from the Input Queue (if available)
`unlockScheduleQueue`($schedule\_queue\_id$)

Call `unlockInputQueue()` for the LP

Algorithm 11: WARPED2 Event Processing Loop for Multiple Schedule Queues

There is a greater risk of events being processed out-of-order in this arrangement. Multiple *Schedule Queues* processing events independently might process events "too greedily" and throw the scheduling mechanism off-balance. If that happens, the Time Warp mechanism will identify any causal violation and

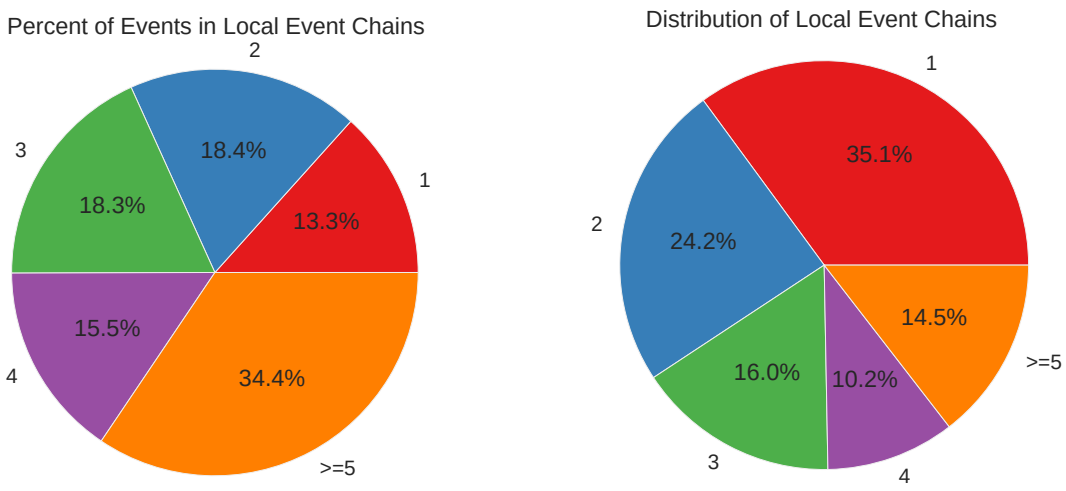


Figure 5.6: Traffic Model : Local Event Chains

initiate the recovery process via *rollbacks*. The motivation behind splitting the common Schedule Queue is to aim for overall gain in computing time by reducing contention for the *Schedule Queue lock*. Imbalance, which is a side-effect of this arrangement, may lead to loss of computing time due to extra rollbacks. As long as this loss does not override the gains made from reduction in lock contention, this arrangement would be a promising option on multi-core processing platforms.

5.2.2 Chains

Inferences drawn from Quantitative Analysis

As evident from the details presented so far in this thesis, one of the the key areas of focus for WARPED2 is *contention management* in the pending event set. Chapter 5 presents the various avenues that have been explored to date. In addition, some foundational studies were also conducted in the past few years on partitioning of simulation models [17] and transactional memory-based access to the pending event set [4], but with only limited success.

Wilsey [9] published a quantitative study on the runtime profile of events generated from the simulation traces of events processed. A key metric in that study was the *event chain*, which is a collection of pending events from an LP that could potentially be executed as a group. The rationale is, at any point in the simulation, an event chain would contain all events from an LP that are available for immediate execution in the pending event set. The event chain of an LP is treated as a single entity and the scheduling mechanism

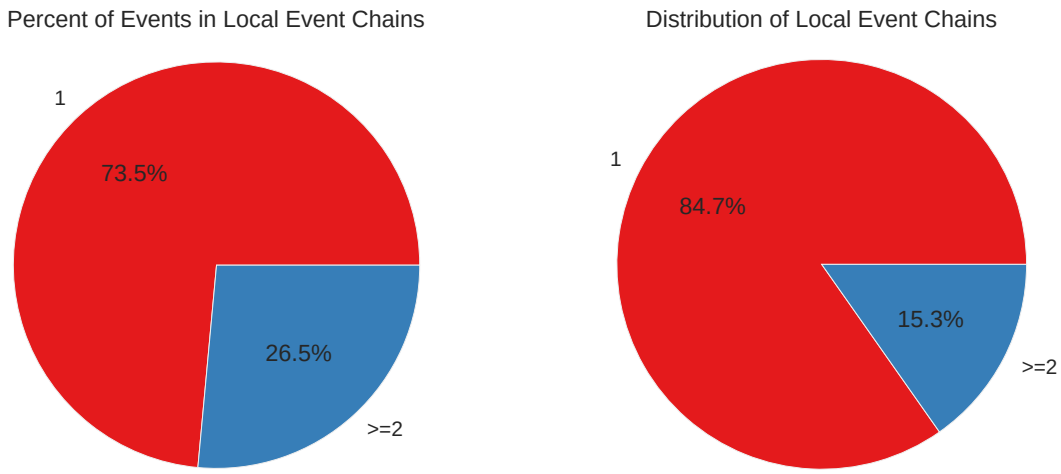


Figure 5.7: Epidemic Disease Propagation Model : Local Event Chains

progresses to the next event once all events in the current chain are processed. Chains can be classified into three categories [9], namely: *local*, *linked*, and *global*. However, for this discussion, only the local chains are of interest.

If multi-event sized chains are a majority, multiple events can be dequeued for execution by each processing thread. Plots on the left side in Figures 5.6, 5.8, and 5.7 show the percentages of total events in local event chains. This data is from a previous study [9] and it shows that majority of events in *Portable Cellular Service (PCS)* and *Traffic* models are part of event chains whose length exceeds 1. However, only about 27% of events in *Epidemic* model are part of chains whose length exceeds 1. Based on this observation,

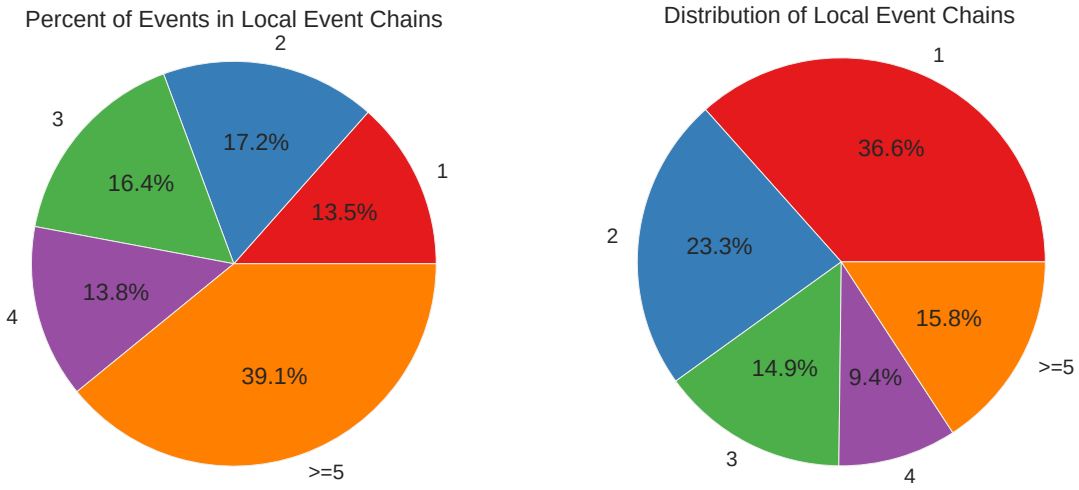


Figure 5.8: Portable Cellular Service Model : Local Event Chains

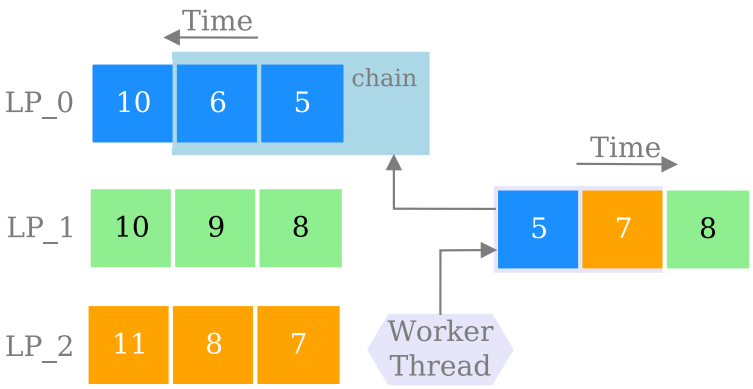


Figure 5.9: WARPED2 Scheduling Technique : Chains

it would be better from a performance standpoint if pending events from *PCS* and *Traffic* are scheduled in groups. For detailed description of the above-mentioned simulation models, please refer to Section 6.3.

The event chain data also helps us identify chains of variable lengths available at different points during the simulation. This helps us quantify the percentages of multi-event scheduling opportunities available. Plots on the right in Figures 5.6, 5.7 and 5.8 show the fraction of chains of length 1–4 and higher. If events from *Traffic* and *PCS* models are scheduled for processing in groups of size ≥ 2, the simulator may have to deal with causality violation 36% of the time. The *Epidemic* model has 85% chance of suffering causality violations when chain size is ≥ 2.

As evident from the discussion above, some models benefit by scheduling events in groups from the pending event set. This study presents a conservative snapshot of the event's availability. Time Warp synchronization, on the other hand, benefits from the delayed approach to enforcement of causal order. Thus, performance of a Time Warp simulation when events are scheduled in groups may exceed what the profile data suggests.

Chain Scheduling

Chain Scheduling technique is an attempt to understand the extent of parallelism available when a set of smallest available events from a LP are scheduled for processing. The goal is to schedule this set (or *chain*) of events without adversely hampering the performance of WARPED2 due to too many causal violations. Each "dequeue event" request will remove multiple events from the LP's *Input Queue* along with the event at the head of the *Schedule Queue*. These dequeued events are then processed in serial order.

It would be too optimistic to expect there will not be any causal violations when a chain of events is processed. However, Time Warp is a "checkpointing-and-rollback" based mechanism that can detect and rectify itself when it detects causal violations. This extra workload will add some delay to the overall computing time. On the other hand, "dequeuing" a set of events from the *Input Queue* saves valuable computing time that would otherwise be wasted on the wait to acquire the *Schedule Queue Lock* had all events been scheduled via the *Schedule Queue*. Thus, *Chain Scheduling* is a compromised design choice which can out-perform WARPED2's default scheduling mechanism (discussed in Section 4.2.3) if the computing time saved by reduced lock contention exceeds the time lost due to extra rollbacks.

There are two approaches for how the output events, generated as a result of this processing, can be inserted back into the simulation as pending events. These are:

- Wait till all events from the chain have been processed and then all the output events are inserted back into the simulation in bulk. This approach allows for efficient contention management since worker threads will not need to acquire locks inside pending event set frequently. However, any causal violation (due to delay in insertion of output events) will not be detected till at a later time. This could result in *rollbacks* of longer duration. While testing, this approach proved to be hugely inefficient compared to the approach discussed next. Therefore, WARPED2 does not currently support bulk sending of stored output events. A detailed study is available in [7].

- Insert the output events into the simulation as soon as they are generated. This approach will increase the contention overhead in the shared data structures of the pending event set but will allow the simulator to detect causality violations quicker than the the former approach. A detailed study is available in [7].

After all the "dequeued events" in a chain have been processed, the *Schedule Queue* is replenished with the smallest available unprocessed event from the LP whose events were present in that chain. The configurable parameter *chain size* is vital for adjusting the sliding time window within which events are processed in bulk. Thus, *chain size* impacts the performance of WARPED2 by affecting the frequency of rollbacks.

Algorithm 12 shows how chains are scheduled in WARPED2. The smallest available event from the

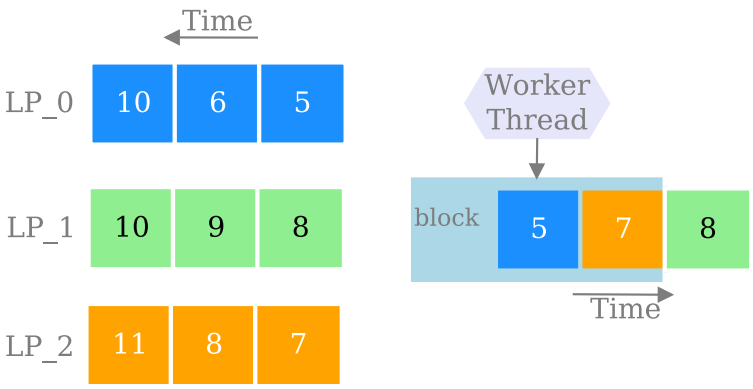


Figure 5.10: WARPED2 Scheduling Technique : Blocks

Schedule Queue is dequeued using `getNextScheduledEvent()` and then the n smallest available events from the corresponding *Input Queue* are read to form the *chain*. Another interesting point to note here is that each thread uses `reportMinTimeForGVT()` to report the smallest event timestamp for *GVT* calculation. Normally this would be a simple operation when scheduling only one event at a time for processing, but here only the timestamp of the smallest event in the chain is reported. In this case it happens to be the first event stored inside the `event_chain`.

5.2.3 Block Scheduling

The *Block Scheduling* technique is an attempt to understand the extent of parallelism available when a set of smallest available pending events from the *Schedule Queue* are scheduled for processing. The goal is to process this set (or *block*) of events in serial order without adversely hampering the performance of WARPED2 due to too many causal violations. No two events in this *block* belong to the same LP.

It would be too optimistic to expect there will not be any causal violations when a block of events is processed. Fortunately, the Time Warp is a "checkpointing-and-rollback" based mechanism that can detect and rectify itself when it detects causal violations. This extra workload will add some delay to the overall computing time. On the other hand, "dequeuing" a set of events from the *Schedule Queue* saves valuable computing time that would otherwise be wasted on the wait to acquire the *Schedule Queue Lock* for every scheduled event. Thus, *Block Scheduling* is a compromised design choice which can out-perform WARPED2's default scheduling mechanism (discussed in Section 4.2.3) if the computing time saved by reduced lock contention exceeds the time lost due to extra rollbacks.

```
while signal to terminate not detected do
    schedule_queue_id ← getScheduleQueueId(thread_id)
    lockScheduleQueue(schedule_queue_id)
    e ← getNextScheduledEvent(schedule_queue_id)
    unlockScheduleQueue(schedule_queue_id)

    LP ← receiver of e
    LP → lockInputQueue()
    event_chain ← read n events from the Input Queue
    LP → unlockInputQueue()
    reportMinTimeForGVT( thread_id , event_chain[0] → timestamp() )

    while event e ∈ event_chain do
        if e < last processed event for LP then
            Rollback LP
            break

        if e is an anti-message then
            Cancel event with e (if possible)
            continue

        Process event e
        Save state of LP
        Send newly generated events to their destinations

    Call lockInputQueue() for the LP

    Move processed events to Processed Queue

    lockScheduleQueue(schedule_queue_id)
    Replace scheduled event for LP with an event from the Input Queue (if available)
    unlockScheduleQueue(schedule_queue_id)

    Call unlockInputQueue() for the LP
```

Algorithm 12: WARPED2 Event Processing Loop for Chain Scheduling

In block scheduling, each "dequeue event" request will remove multiple events from the *Schedule Queue*. These dequeued events are then processed sequentially. There are two approaches as to how the output events, generated as a result of this processing, can be inserted back into the simulation as pending events. These are:

- Wait till all events from the block have been processed and then all the output events are inserted back into the simulation in bulk. This approach allows for efficient contention management since worker threads will not need to acquire locks inside pending event set frequently. However, any causal violation (due to delay in insertion of output events) will not be detected till at a later time. This could result in *rollbacks* of longer duration. While testing, this approach proved to be hugely inefficient compared to the approach discussed next. Thus, WARPED2 does not currently support bulk sending of stored output events. A detailed study is available in [7].

- Insert the output events into the simulation as soon as they are generated. This approach will increase the contention overhead in the shared data structures of the pending event set but will allow the simulator to detect causality violations quicker than the the former approach. A detailed study is available in [7].

After all the "dequeued events" in a block have been processed, the *Schedule Queue* is replenished with the smallest available unprocessed event from each of the LPs whose events were present in that block. Bulk replenishment of the *Schedule Queue* after processing a *block* of events would help reduce time wasted on contention management in the pending event set. Similar to *chain size*, the configurable parameter *block size* is vital for adjusting the sliding time window within which events are processed in bulk. That is, *block size* impacts the performance of WARPED2 by affecting the frequency of rollbacks.

Algorithm 13 shows how blocks are scheduled in WARPED2. The `getNextEventBlock()` dequeues n smallest available events from the *Schedule Queue*. Each thread reports the smallest event timestamp for *GVT* calculation using `reportMinTimeForGVT()`. Normally this would be a simple operation when scheduling only one event at a time for processing, but here only the timestamp of the smallest event in the block is reported. In this case, it happens to be the first event stored inside the `event_block`.

```
while signal to terminate not detected do
    schedule_queue_id ← getScheduleQueueId(thread_id)
    lockScheduleQueue(schedule_queue_id)
    event_block ← getNextEventBlock(schedule_queue_id)
    unlockScheduleQueue(schedule_queue_id)

    reportMinTimeForGVT( thread_id , event_block[0] → timestamp() )

    Create an empty LP_list
    while event e ∈ event_block do
        LP ← receiver of e
        Insert LP into LP_list

        if e < last processed event for LP then
            Rollback LP

        if e is an anti-message then
            Cancel event with e (if possible)
            continue

        Process event e
        Save state of LP
        Send newly generated events to their destinations

    for LP ∈ LP_list do
        Call lockInputQueue() for the LP

    lockScheduleQueue(schedule_queue_id)
    for LP ∈ LP_list do
        Move e to Processed Queue
        Replace scheduled event for LP with an event from the Input Queue (if available)
    unlockScheduleQueue(schedule_queue_id)

    for LP ∈ LP_list do
        Call unlockInputQueue() for the LP
```

Algorithm 13: WARPED2 Event Processing Loop for Block Scheduling

5.2.4 Bags

Why are bags an interesting option for scheduling pending events?

Extending the theme set by chain and block scheduling techniques, contention management is the key motivating factor for organizing pending events into bags. The goal of this design is to enable threads to process events from different bags in parallel without long waits for access to the shared data structures. This design relies on the hypothesis that pending events grouped into profile-driven partitions will mostly retain causal order even when these event partitions are processed in parallel. Thus, in theory, processing bags in parallel combines the best of both worlds: low contention for shared resources and minimal performance trade-off due to marginal increase in rollbacks.

The Louvain Partitioner

The *Louvain* method [46] is a greedy optimization method that can partition large weighted networks into communities. Its output is a dendogram that captures the hierarchies of communities.

Modularity [69] is a metric used to quantify how effectively the network was partitioned into different communities (or clusters). Its value ranges between *-1* and *1* and is a measure of edge density inside communities compared to edges outside communities. A high modularity index indicates dense intra-community nodal connections along with sparse inter-community nodal connections.

Though optimization should in theory yield the best possible solution, it is impractical due to the sheer volume of cluster combinations possible. As a result, the *Louvain* method uses a 2-step heuristic approach to iteratively optimize the *modularity* of partitions:

1. search for "small" communities through local optimization of modularity, and
2. group together nodes belonging to the same community and construct a secondary network between communities.

Accurate modularity optimization is considered to be a *NP-hard* problem. However, estimates suggest that the *Louvain* method has *O(n log n)* amortized time complexity [46]. Figure 5.11 illustrates a network partitioned into different communities (each color indicates a different community).

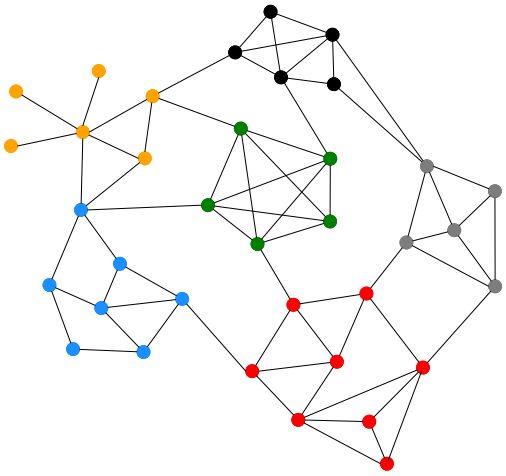


Figure 5.11: Louvain-based Partitions (shown in different colors)

Crawford *et al* [16] analyzed the distribution of communities in a simulation of epidemic disease propagation for *10,000* LPs (refer to Section 6.3.1 for details regarding the Epidemic model). They found *54* communities whose LP counts were distributed in the range *50-425* with the mean at approximately *190*. Figure 5.12 shows this distribution of communities.

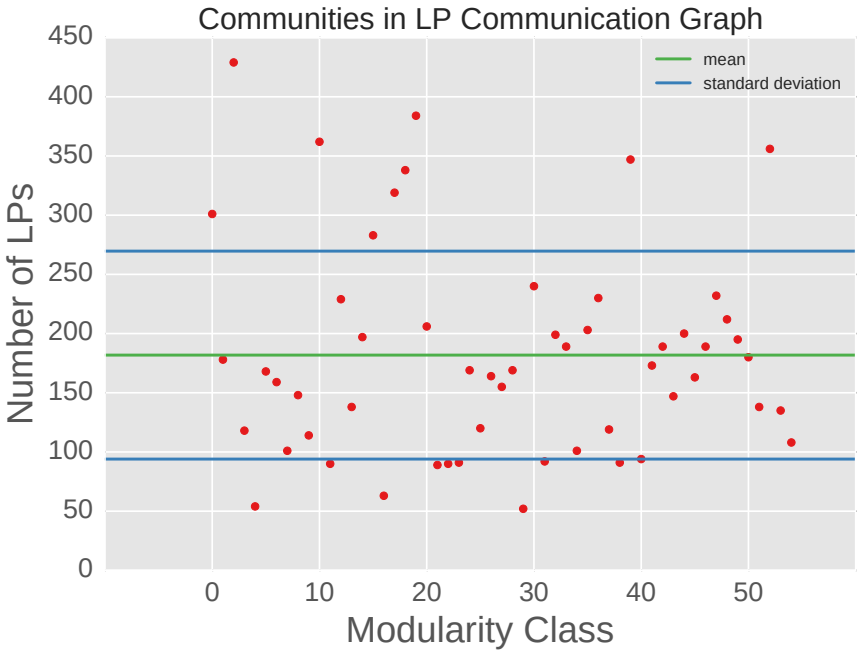


Figure 5.12: Distribution of Communities in 'epidemic-10k-ws' model (refer to Table 6.3 for model specifications)

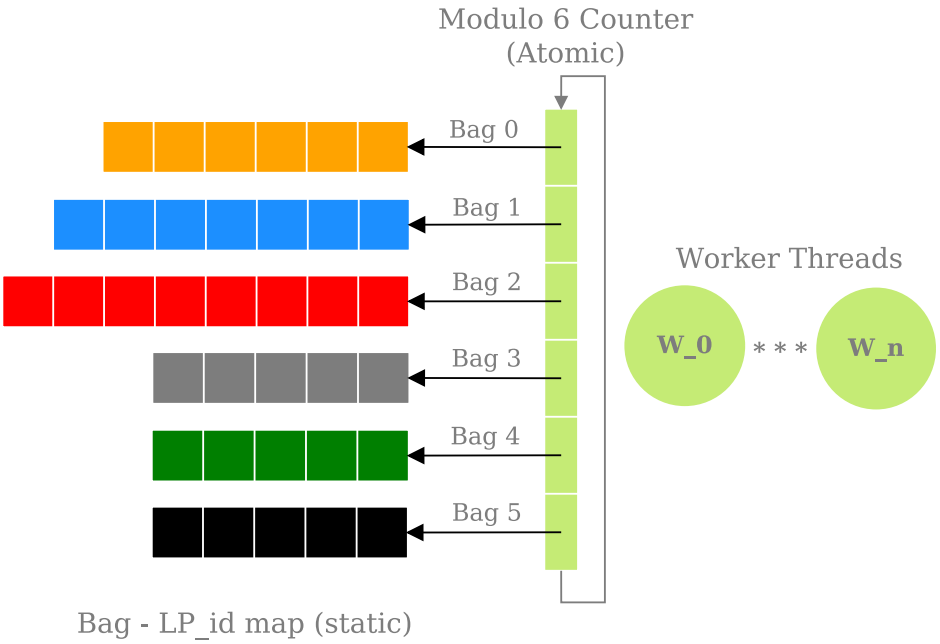


Figure 5.13: WARPED2 Scheduling Technique : Bags

Organizing Partitions into Bags

From the perspective of *Discrete Event Simulation*, let us assume that each node in Figure 5.11 represents a Logical Process (LP). The weight of any edge in the network equals number of events exchanged between the two LPs connected by that edge. A network partition (or community) with high modularity index ideally has dense connections between LPs inside a partition and sparse connections to LPs outside the partition. The hypothesis is that densely connected LPs are more likely to generate events which are *causally linked* than events generated by LPs which are sparesly connected. Based on this assumption, the LPs from each community can be grouped together to form a *bag* for that community. Figure 5.13 presents a static *Circular Array* whose each element points to a bag of LPs. The color and size of each *bag* in Figure 5.13 matches those of the corresponding *community* in Figure 5.11.

Each worker thread requests the *modulo N atomic counter* for access to the next available bag in the circular array. The size of the static circular array (also called *carousel*) equals the number of bags (N). Once the thread gets access to a bag, it locks access to that bag and copies the smallest available unprocessed event from the *Input Queue* of each LP mapped to that bag into an *array*. It then schedules a part or whole of this unsorted array for processing. There are three potential benefits to organizing LPs in the aforementioned

way:

1. The LPs in one bag generally share sparse connections to LPs in other bags. Assuming this sparse connection means events in one bag are more likely to be causally independent of events in other bags over a large time window, different threads should be able to process events from different bags in parallel without a drastic increase in rollbacks. The *modulo N counter*, built using *CAS* atomic operation, can allow a thread to access a bag from the carousel without the need for any central lock.

2. Dense connections between LPs inside a bag may signify that some or all events in the above-mentioned *array* are causally linked. However, as discussed in Section 5.1.3, it is reasonable to assume that events within a relatively small time window are causally independent of each other. Therefore, instead of processing all events from the *array*, a worker thread can only process those events whose timestamps are within a specified time window. Much like the *Unsorted Bottom* in the Ladder Queue, the events within this time window are not sorted before being processed. During every event processing cycle in WARPED2, the worker thread calculates the time window's lower and upper bounds using one of the following two configurable parameters: T

   - *Window Size (TS_{window})* :

   $$\begin{aligned} &[\, TS_{min}\ ,\ TS_{min} + TS_{window}\ ],\ \text{where} \\ &TS_{min} = \text{timestamp of smallest event in the array.} \end{aligned} \tag{5.1}$$

   - *Fraction of Bag's Total Window Size ($frac_{\text{BAG}}$)* :

   $$\begin{aligned} &[\, TS_{min}\ ,\ TS_{min} + (TS_{max} - TS_{min}) \times frac_{\text{BAG}}\ ],\ \text{where} \\ &TS_{min} = \text{timestamp of smallest event in the array , and} \\ &TS_{max} = \text{timestamp of largest event in the array.} \end{aligned} \tag{5.2}$$

3. The carousel and the bags linked to it are static structures. This means there is no loss of performance due to memory allocation and deallocation after initialization. Even the *array* used to temporarily copy the smallest unprocessed events for the bag can be reused for processing the next bag. This drastically reduces the fluctuations in memory usage and boosts performance through reduced need

for memory I/O.

After the worker thread finishes processing all events scheduled from a bag, it moves these events from the *Input Queue* to the *Processed Queue* of the corresponding LPs. It unlocks the bag it was currently processing and request the atomic counter for the next available bag in the carousel. It is worth noting here that the lock for a bag is an atomic flag that is used to prevent multiple worker threads from accessing a particular bag at the same time. It is unlikely that there is any significant contention for this flag because the modulo counter ensures serialized allocation of bags to waiting threads. Even if a thread is forced to wait while another thread finishes processing events from a bag, this wait is unlikely to be long since the thread currently holding the lock is processing events for the previous cycle, but was delayed due to some reason.

Distributed Tracking of Simulation's Progress

The distributed organization of bags complicates how worker threads can asynchronously report the smallest event timestamp for calculation of the *GVT*. As mentioned in Section 4.1.2, worker threads in WARPED2 work for a process and regularly report the lowest timestamp for that process. A worker thread learns the smallest timestamp among events in a bag when it acquires access to that bag. A *GVT cycle* for bags is considered complete when worker threads have processed events from all bags in the carousel in that cycle. Depending on the speed at which events are processed, a worker thread may process multiple bags during any GVT cycle but it only reports the smallest timestamp among all events it processed during that cycle. During an ongoing GVT cycle, the worker thread reports its minimum timestamp as the one recorded in the *previous* GVT cycle.

In order to keep track of the minimum timestamp among events processed during any ongoing GVT cycle while also retaining the minimum timestamp recorded in the previous cycle, each worker thread uses a data structure called `minTS_record` with the following attributes:

- Current GVT Cycle Count (`curr_cycle_count`)
- Minimum Timestamp recorded during the previous GVT cycle (`minTS_prev`)
- Minimum Timestamp recorded during the ongoing GVT cycle (`minTS_curr`)

Algorithm 14 presents how events are processed in bags. Some of the event set functions used here have been explained below:

- `lockInputQueue()` and `unlockInputQueue()` controls access to the Input Queues.
- `getBag()` fetches the next available bag from the carousel and locks it. It also increments the modulo N atomic counter using Compare-and-Swap (CAS) operation. If the `curr_cycle_count` is smaller than the current GVT cycle count, the worker thread realizes it has entered a new GVT cycle. In that case, it transfers `minTS_curr` to `minTS_prev` and updates `curr_cycle_count` to the current GVT cycle count.
- `releaseBag()` releases the lock a worker thread acquired on a bag before processing events from that bag.
- `fetchLPs()` fetches the list of LPs from the static *carousel* for a particular bag.
- `getEvent()` find the smallest available event from the Input Queue of an LP. The worker thread compares this event's timestamp to `minTS_curr` and updates the latter if the event's timestamp is smaller.
- `reportMinTimeForGVT()` is used by each thread to report the smallest timestamp for GVT calculation.

```
while signal to terminate not detected do
    bag_id ← getBag()
    LP_list ← fetchLPs(bag_id)

    buffer := array for temporarily storing events
    for LP ∈ LP_list do
        Call lockInputQueue() for the LP
        e ← getEvent(LP)
        Call unlockInputQueue() for the LP
        Insert e into buffer

    min_ts ← minimum timestamp from the previous GVT cycle for this thread
    reportMinTimeForGVT( min_ts )

    Calculate the upper limit (TS_upper) for admissible events. Refer to Equations 5.1 and 5.2.

    while event e ∈ buffer do
        if e → timestamp() > TS_upper then
            continue

        if e < last processed event for LP then
            Rollback LP

        if e is an anti-message then
            Cancel event with e (if possible)
            continue

        Process event e
        Save state of LP
        Send newly generated events to their destinations

        Call lockInputQueue() for the LP
        Move e to Processed Queue
        Call unlockInputQueue() for the LP

    releaseBag (bag_id)
```

Algorithm 14: WARPED2 Event Processing Loop for Bags

Chapter 6

Experiments

6.1 Setup

The focus of this thesis is to explore different options to explore how the performance of the *pending event set* can be improved in a Time Warp synchronized Discrete Event Simulator. The goal is to reduce the contention for shared data structures. As mentioned in the introduction, the key aspects that amplify contention for the shared data structures containing the pending event set are *lock contention*, *sorting*, and *scheduling order*. The ideas discussed in Chapter 5 are closely tied to the architecture of SMP / NUMA nodes and how its performance is affected by thread synchronization, and memory allocation. Therefore, all experiments for this dissertation have been performed on a single node NUMA machine.

Table 6.1 presents how the WARPED2 kernel was configured for the experiments.

Parameter	Values
Number of Worker Threads	4, 8, 16, 32 or 64
Number of Schedule Queues	1, 2, 4, 8 or 16
GVT Method	Asynchronous
GVT Period (milliseconds between each estimation)	1,000
State Save Period (events processed between state saves)	32
Bag Window Size	4, 16, 32, 64, 128, 256 or Full
Fraction of Bag Window	0.05, 0.25, 0.5, 0.75 or Full
Chain Size (events)	4, 8, 12 or 16
Block Size (LPs)	32, 64, 128 or 256

Table 6.1: WARPED2 setup

		Intel® Xeon® E5-2670
Processor	ISA	x86_64
	# Cores	8
	# Threads	16
	# Sockets	2
	Frequency	2.60 GHz
	L1 Data Cache	32 kB
	L1 Inst Cache	32 kB
	L2 Cache	256 kB
	L3 Cache	20 MiB
	Memory	64 GB
Runtime	OS Kernel	Linux 4.9.0-4-amd64
	C Library	Debian GLIBC 2.24-11+deb9u3
	Compiler	GCC v6.3.0
	MPI	MPICH v3.2
	Python	2.7.13
	Python-numpy	1.12.1
	Python-networkx	1.11

Table 6.2: Experimental Setup

The computational platform for the experiments performed and reported in this thesis is defined in Table 6.2.

6.2 Performance Metrics

The performance metrics used in this study are:

- **Simulation Runtime (in seconds):** This metric is useful for comparing performance of different configurations and to compare the effectiveness of different ideas discussed in Chapter 5.
- **Event Commitment Ratio** for a configuration equals

$$\frac{\text{Total number of events processed for that configuration}}{\text{Total number of committed events}}$$

  Total number of committed events is independent of the effects of different configurations. This metric is useful to understand how different scheduling techniques (discussed in Chapter 5) adversely impact performance of the simulation due to fluctuations in frequency of causal violations.

- **Events Processing Rate (per second)** for a configuration equals

$$\frac{\text{Total number of events processed for that configuration}}{\text{Simulation Runtime (in seconds)}}$$

  This metric is a good indicator of the degree of contention that exists in the pending event set. A high value on this metric indicates lower contention and is useful for analyzing the effectiveness of group scheduling and bags.

6.3 Simulation Models

This section describes three WARPED2 simulation models. These models have been used as benchmarks to study and compare the performance of the WARPED2 kernel for the design hypotheses presented in Chapter 5. Some of these models are based on simulation models developed elsewhere and are commonly used as test-benches by *Discrete Event Simulation* researchers.

6.3.1 Epidemic Disease Propagation

The *Epidemic* model is a Discrete Event Simulation model that simulates how an infectious disease would spread across a set of geographic locations. It is based on a reaction-diffusion epidemic model proposed by Perumalla *et al* [70].

In the simulation model, each *location* represents an LP. It is home to a population in which each individual has varying degree of susceptibility to the disease. A probabilistic reaction function has been used to model the intra-location transmission of the disease amongst its residents [71]. The spread of disease within an individual has been modelled as a finite state machine called *Probabilistic Timed Transition System (PTTS)* [71]. A diffusion network models the inter-location migration of individuals between locations. The design of diffusion network is based on one of these two available options:

- β model proposed by Watts *et al* [72] because it allows the model to mimic the "small-world" properties of the *human network*.

- Some properties of the *human network* mirror that of a "scale-free" network. As a result, the *diffu-*

sion network can also be configured to use the *Barabsi-Albert* model [73] which is an algorithmic approximation of a "scale-free" network.

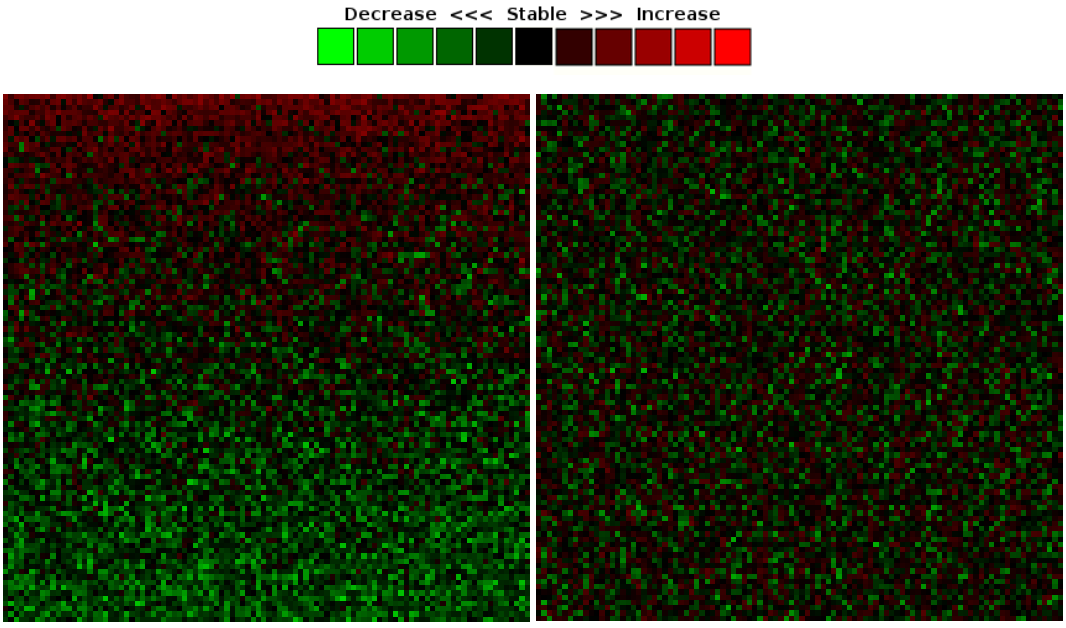
A limitation of the β-model [72] is that it produces an unrealistic degree distribution. Real networks, on the other hand, are often "scale-free" networks, in-homogeneous in degree, and have hubs and a scale-free degree distribution. Thus, real networks are better described by the *Barabsi-Albert (BA)* model [73] which supports preferential attachment. However, unlike the β-model, the *Barabsi-Albert* model cannot produce the high levels of clustering seen in real networks. Thus, neither the β model nor the *Barabsi-Albert* model are fully realistic. Figure 6.1 compares the diffusive behavior of people when using these two networks on a population of *1,000,000*. Both heatmaps show whether the population has increased, decreased or remained fairly stable at each of the *10,000* locations after the end of simulation. It is evident from the distribution of *red* and *green* across the simulation space that the *Barabsi-Albert* model has an in-homogeneous degree distribution and sparse clusters. The β-model, on the other hand, has a fairly regular degree distribution and denser clustering.

The LP state keeps track of the population residing in that location. Individuals can leave one location and travel to any of the connected locations. The connections depends on the diffusion network layout. The following attributes are tracked in each individual:

- ID (unique for each individual in the entire population),
- susceptibility to the disease (value $\in$ [0,1]),
- vaccination status : *yes / no*, and
- *PTTS* infection status (uninfected, latent, incubating, infectious, asympt or recovered). Refer to Figure 6.2 for details of the *PTTS*.

An exponential distribution mimics the creation of events within the epidemic model. The following are the event types used in this simulation:

- **Disease Update Trigger**: self-initiated by the LP for updating the infection status of its resident population,



Barabási–Albert Model Watts-Strogatz β Model

Figure 6.1: Diffusion of Population

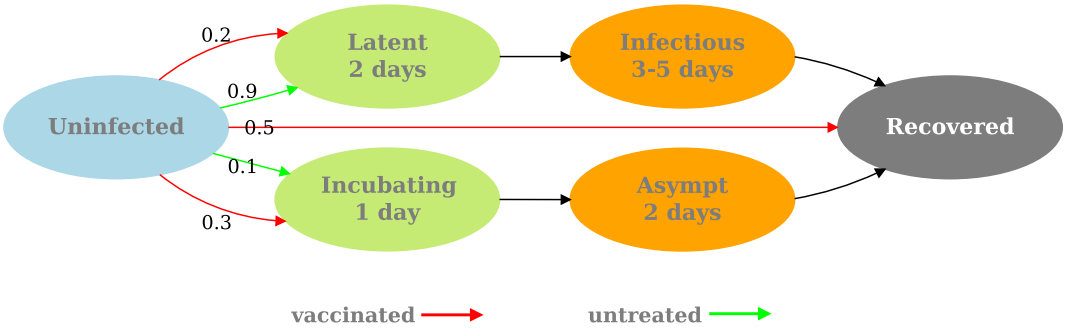


Figure 6.2: Probabilistic Timed Transition System

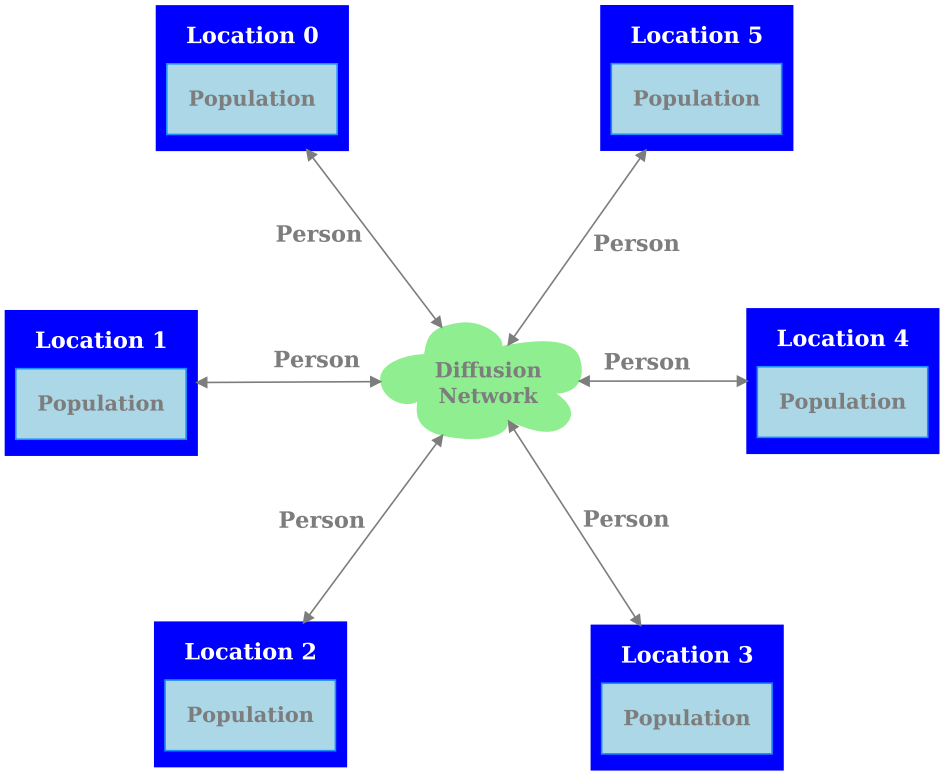


Figure 6.3: Epidemic Model

- **Diffusion Trigger**: self-initiated by the LP for sending one resident to any neighboring location with uniform randomness, and
- **Diffusion**: used when welcoming an arriving individual from a neighboring location.

Figure 6.3 presents an illustration of the model. Table 6.3 shows the different configurations of the epidemic model that have been used in Section 6.4 and appendix. Tables A.3, A.5, A.4 and A.6 provide the detailed configuration for these models.

Configuration	Diffusion Network	Number of locations	Population
epidemic-10k-ws	Watts-Strogatz	10,000	1,000,000
epidemic-10k-ba	Barabasi-Albert	10,000	1,000,000
epidemic-100k-ws	Watts-Strogatz	100,000	500,000
epidemic-100k-ba	Barabasi-Albert	100,000	500,000

Table 6.3: List of Epidemic Configurations

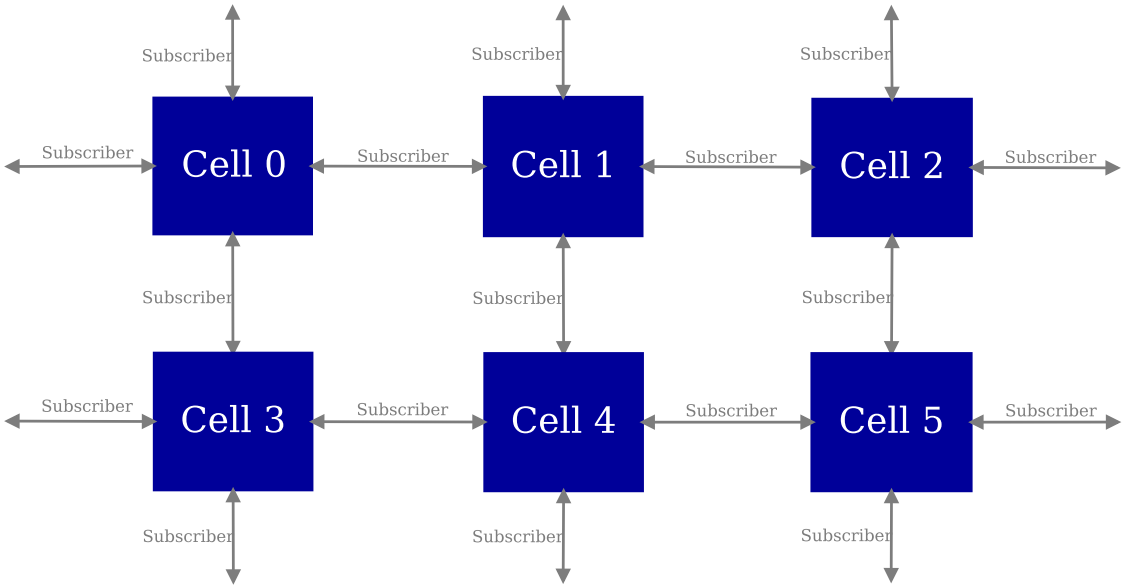


Figure 6.4: Portable Cellular Service Model

6.3.2 Portable Cellular Service (PCS)

This model simulates a wireless communication network capable of providing service to a number of subscribers (or *portables*). The network service area comprises of *cells*, each of which represents the smallest building block in the simulation. Each cell has a certain number of channels that can be used for making or receiving calls. Whenever there is an incoming or outgoing call, an available channel from the cell is allocated. In the event of channel unavailability, the call is *blocked* [74]. Each cell represents a logical process in this simulation and they are organized as a rectangular grid (refer to Figure 6.4). The test configuration uses a grid of size *100 x 100* which translates to a LP count of *10,000*. Table A.7 presents the configurations used for this model.

Portables can migrate to adjacent cells and there is a wrap around which occurs when a portable reaches one of the edges. Migration to an adjacent cell is a regular activity for portables and the time period is based on a Poisson distribution. A *handoff block* happens for a migrating portable when the ongoing call is dropped because all available channels are busy. The state of Logical Process keeps track of the following attributes for each cell:

- count of idle channels,
- count of call attempts,

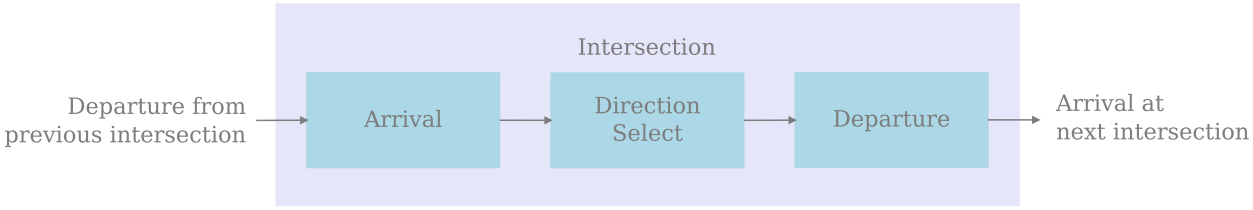


Figure 6.5: Sequence of events at each intersection

- count of channel blocks, and
- count of handoff blocks.

Call requests to a portable follow a Poisson distribution. The following are the types of events used in this model:

- *Next Call*: self-initiated by the cell to start a new call,
- *Call Completion*: self-initiated by the cell to set the call duration,
- *Portable Move Out*: self-initiated by the cell to send a portable to any of its adjacent cells with uniform randomness, and
- *Portable Move In*: receive a portable migrating from an adjacent cell.

6.3.3 Traffic

The traffic model simulates how cars move through a grid of intersections in a city. It models the nature of traffic flow — delays and choice of turns at intersections based on destination of the car. The model assumes that each road has three lanes in either direction and all intersections are four way. This model has been adapted from the ROSS [26] simulator for WARPED2.

Each intersection has been modeled as a Logical Processes and an event represents the movement of cars through and between intersections. Figure 6.5 shows the three phases of an event at an intersection, namely :

- *Arrival*: signifies the arrival of a car at an intersection from any of its adjacent intersection.

- *Direction Select* is a self-initiated event that helps the car to decide which direction to travel from the intersection it is currently at. This decision is based on the final destination the car is trying to reach and regulations controlling the flow of cars through that intersection.
- *Departure* is a self-initiated event which tells the intersection the car is currently located at that it is ready to travel to a different intersection.

The final destination for each car is a randomly assigned intersection. At the initial stage, all available cars are uniformly distributed across all intersections. Once a car arrives at an intersection, it tries to go in the direction that would get it closer to its target intersection. There are thresholds in place to help regulate and spread the flow of traffic. This limits congestion on certain roads and may lead to a car being denied permission to travel in the direction of its choice. The car can randomly choose to either wait or take an alternate route. Events follow an exponential distribution and a new one is created when an existing one is processed.

The status of traffic at each intersection at any point in time is tracked by the Logical Process state. The state has the following attributes which are used to decide whether a car can be allowed to travel in the direction of its choice:

- number of cars arriving from each direction,
- number of cars departing in each direction,
- total number of cars arriving at the intersection, and
- total number of cars departing from the intersection.

The grid has been modeled as a rectangular mesh to allow the traffic model to be scaled. The traffic model in *ROSS* adopts a similar approach. This grid arrangement is similar to the network structure used by *PCS* (as shown in Figure 6.4). Two configurations of traffic have been used for different studies in this thesis. Table 6.4 presents these two configurations and details for each have been presented in Tables A.1 and A.2 respectively.

Configuration	Grid Dimension	Number of Cells
traffic-10k	100 x 100	10,000
traffic-1m	1024 x 1024	1,048,576

Table 6.4: List of Traffic Configurations

6.4 Analysis of Performance

6.4.1 Schedule Queue Data Structure

This quantitative study is aimed at understanding the effectiveness of different data structures for storing and scheduling events for execution in WARPED-2. The data structures being studied are STL MultiSet, Splay Tree and Ladder Queue. Three separate variants of the Ladder Queue are under consideration, namely: (i) one with a fully sorted Bottom and accessed with a mutex lock, (ii) one with an unsorted Bottom and accessed with a mutex lock, and (iii) one with an unsorted Bottom that is accessed with a lock-free, atomic read-write access setup. The details for each data structure can be found in Section 5.1. In order to eliminate the side-effects of load imbalance in this study, only a single schedule queue has been used. The performance of these data structures for different thread counts (4, 8, 16, 32 and 64) has been presented. Additional plots for multiple schedule queues with different data structures can be found in the Appendix. Threshold is a critical parameter in Ladder Queue which triggers the split of a bucket when its event count exceeds this threshold. Here three separate values of threshold have been studied, namely 16, 50 (as recommended by in [8]) and 96.

Figure 6.6 (parts (a), (b), and (c)) show the performance impact of the different data structures used for the Schedule Queue. The number attached to the different Ladder Queue variants in the plot keys represent the threshold of a Ladder Queue. The significant observations from these plots have been explained below:

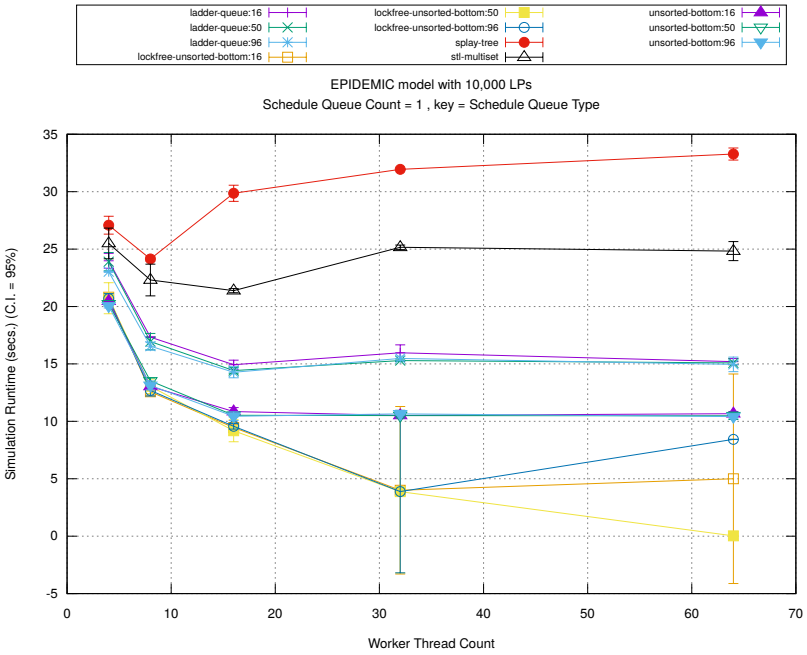
1. As expected, the Ladder Queue with Lock-free Unsorted Bottom outperforms the sorted Ladder Queue, Splay Tree and STL MultiSet by generally over 50% as the number of worker threads increase. However, it only marginally outperforms Ladder Queue with Unsorted Bottom.

2. In the original description of the Ladder Queue [8], Tang *et al* stipulate that the optimal value for threshold should be 50. However, as can be seen in the performance data of Figure 6.6, the setting for this threshold has nominal impact on the overall performance. Section 5.1.2 describes a modification

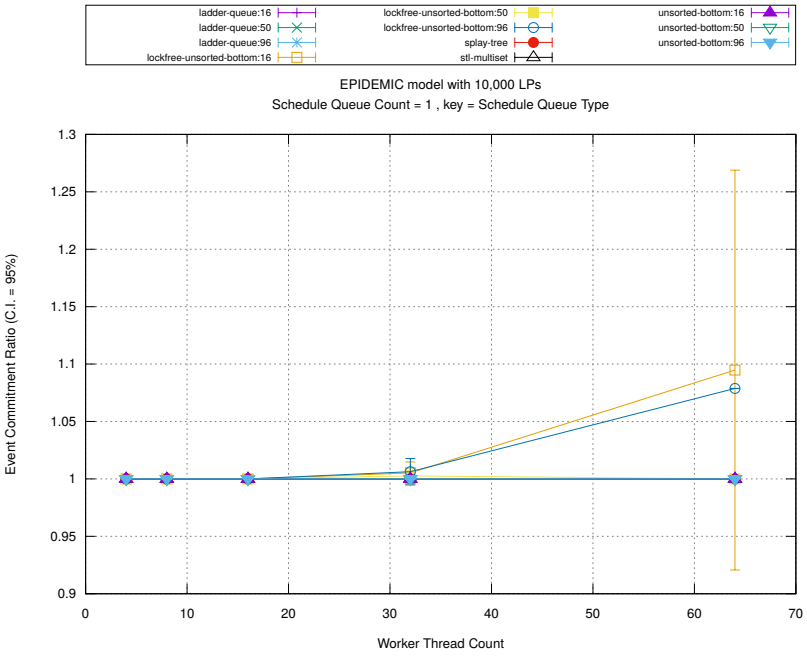
to the original Ladder Queue algorithm which eliminates the need to split Bottom when its event count exceeds this threshold. This is probably why we don't see a huge variation in performance for the three threshold values. The penalty of this split would have been higher for a smaller threshold but there would be increased likelihood of causal violations when a larger threshold is used. It is likely that if this threshold value is increased further than 100, a point will be reached when this choice of threshold will start to negatively impact performance. This is because for that threshold value and beyond, the Ladder Queue will lose its ability to effectively put causally independent events into Bottom.

3. The Ladder Queue with an unsorted bottom is generally 50% faster than the Ladder Queue with a sorted bottom. This is expected because, in case of the former, no time is wasted on sorting events inside Bottom. The Event Commitment Ratio (Figure 6.6(b)) indicates there were no rollbacks in either case which means no time was lost due to recovery from causal violations. The epidemic model generally tends to have low event density within a small time window. This means sorting inside Bottom is not a significant overhead in case of this model.

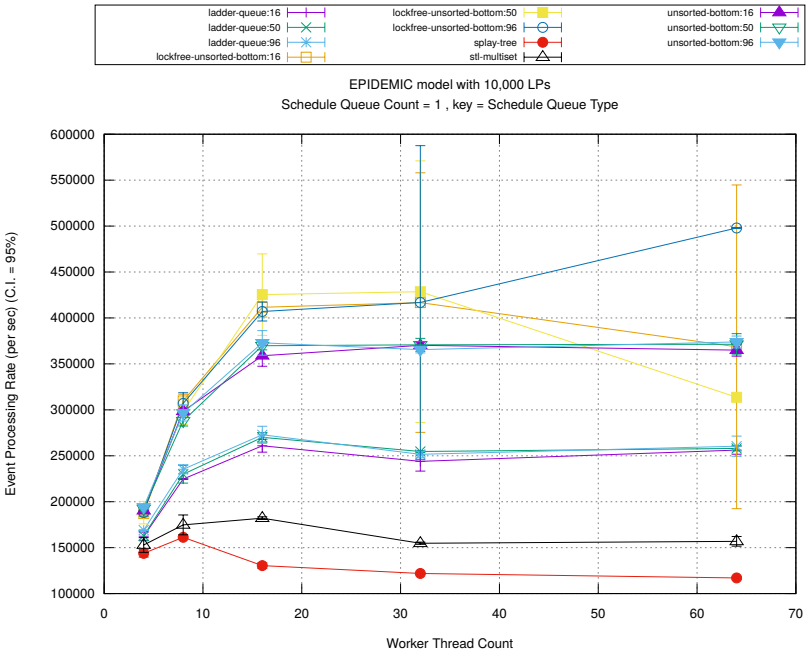
4. The performance of most data structures marginally improves with increase in number of worker threads till about 16 threads and then marginally degrades. On the other hand, Ladder Queue with Lock-free Unsorted Bottom shows slight improvement in performance when thread count goes above 16. This is because atomic read-write operations are not affected adversely by increase in number of threads. However, the performance of other data structures here is surprising because contention for access to the shared Schedule Queue should increase with increase in number of worker threads. This should significantly degrade the performance as the thread count increases beyond 16 but that is not what has been observed. The architecture of the NUMA machine used may provide some clues. As mentioned in Table 6.2, each processor has 2 sockets of 8 cores each and each core supports 2 hardware threads. This explains why the best performance is observed for 16 threads. However, it is difficult to speculate why the performance beyond 16 threads does not degrade significantly.



(a) Simulation Runtime (in seconds)

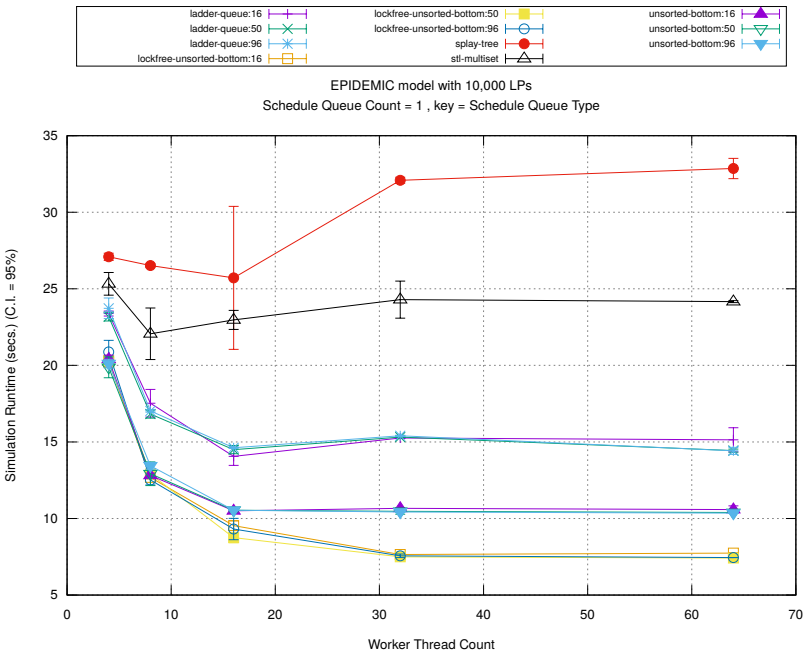


(b) Event Commitment Ratio



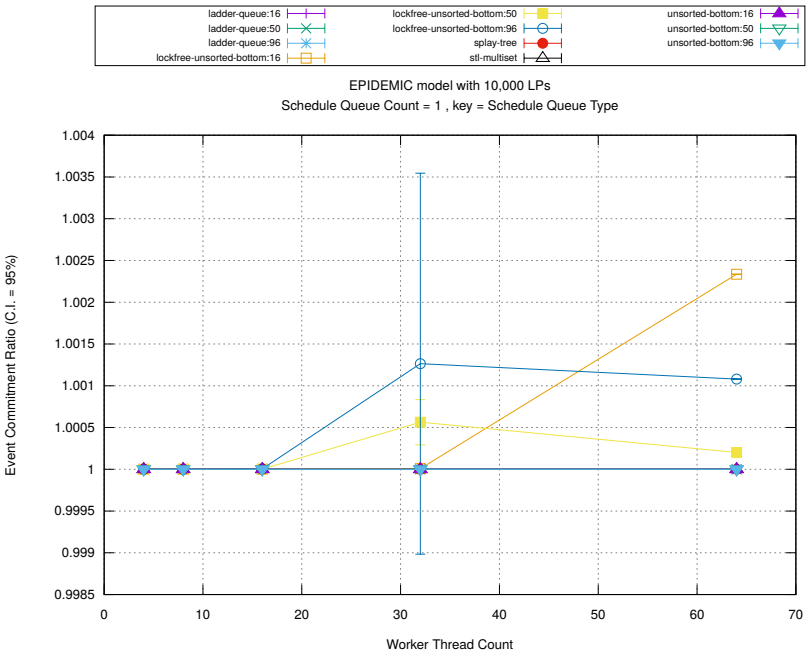
(c) Event Processing Rate (per second)

Figure 6.6: Impact of the schedule queue data structure: Epidemic Model (*10,000* LPs, *Barabsi-Albert* diffusion network)

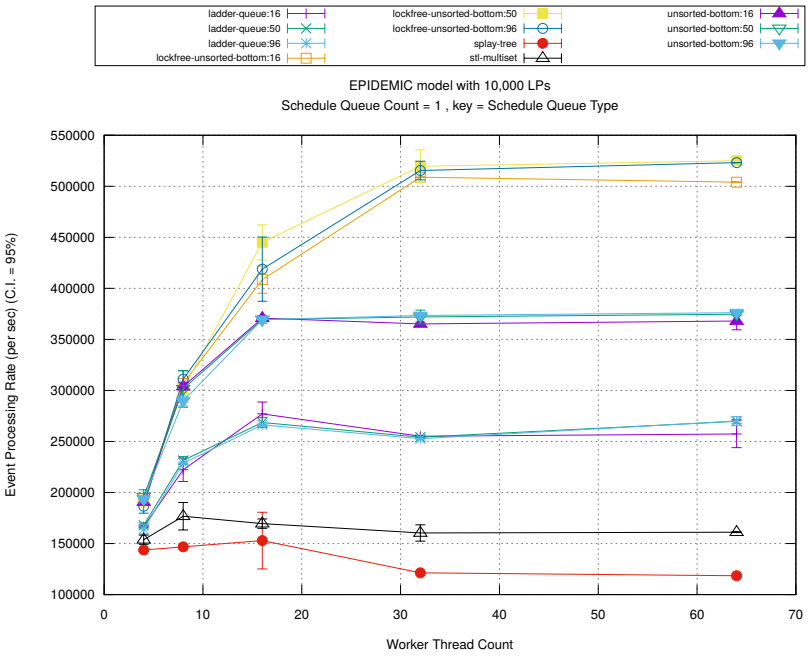


(a) Simulation Runtime (in seconds)

Figure 6.7 (parts (a), (b), and (c)) show that the performance results of the Epidemic model with *Watts-Strogatz* [72] diffusion network is generally similar to that of Epidemic model with *Barabsi-Albert* [73] diffusion network (shown in Figure 6.6).

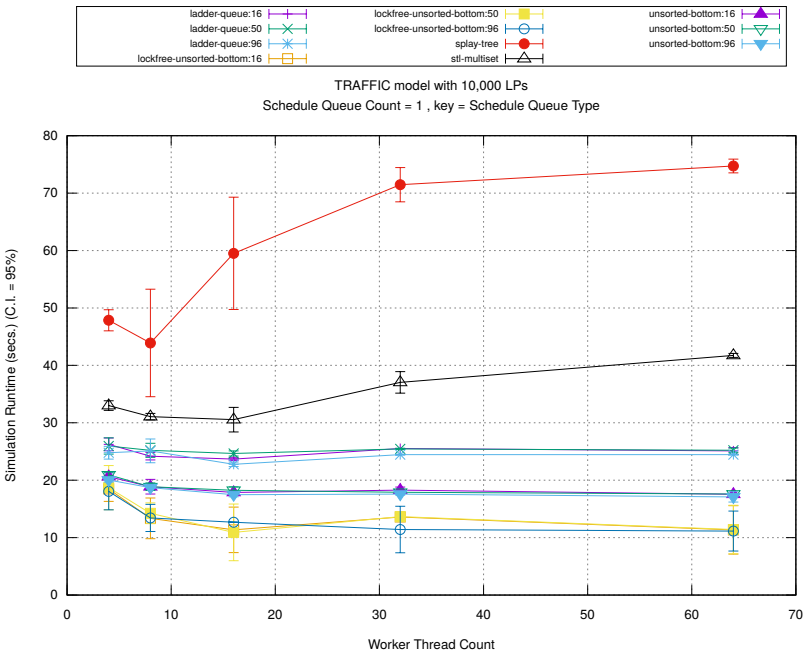


(b) Event Commitment Ratio



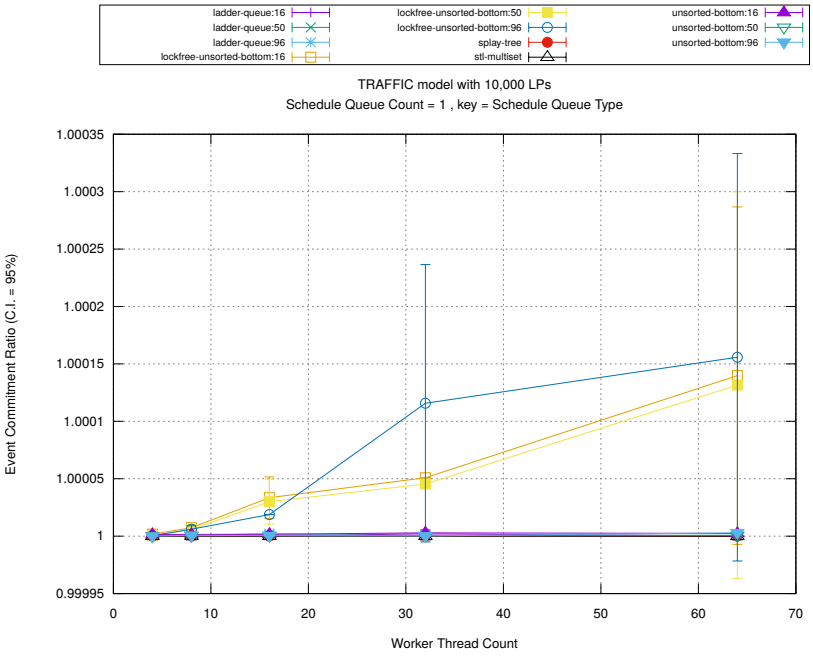
(c) Event Processing Rate (per second)

Figure 6.7: Impact of the schedule queue data structure: Epidemic Model (*10,000* LPs, *Watts-Strogatz* diffusion network)

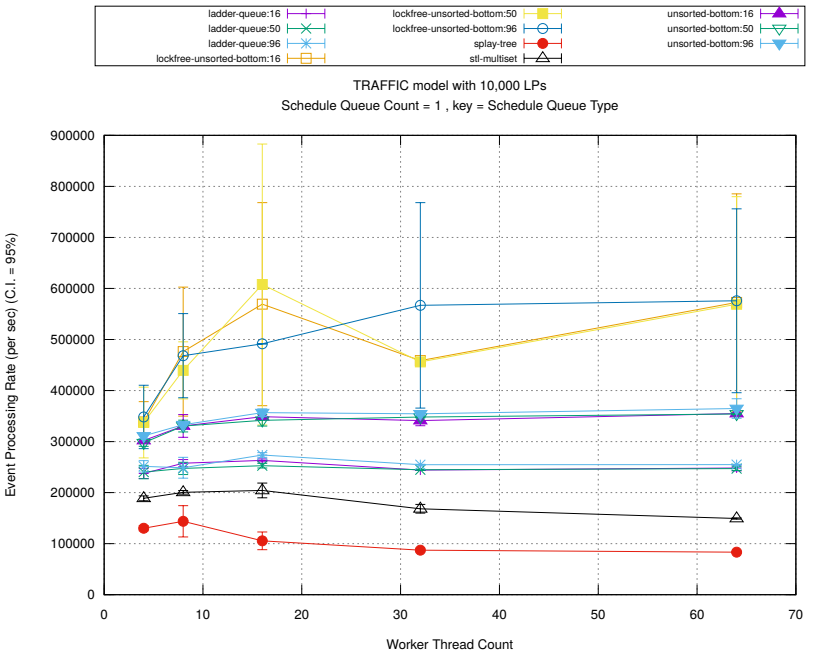


(a) Simulation Runtime (in seconds)

Compared to Epidemic Model plots in Figures 6.6 and 6.7, Traffic model behaves differently for Ladder Queue with Lock-free Unsorted Bottom. Figure 6.8 shows its performance is nearly 90% faster than Ladder Queue with Unsorted Bottom, which in turn is over 25% faster than sorted Ladder Queue. Traffic model tends to have higher event density within a small time window. The simulation's performance benefits from not having to sort a large number of events that are stored inside Bottom.

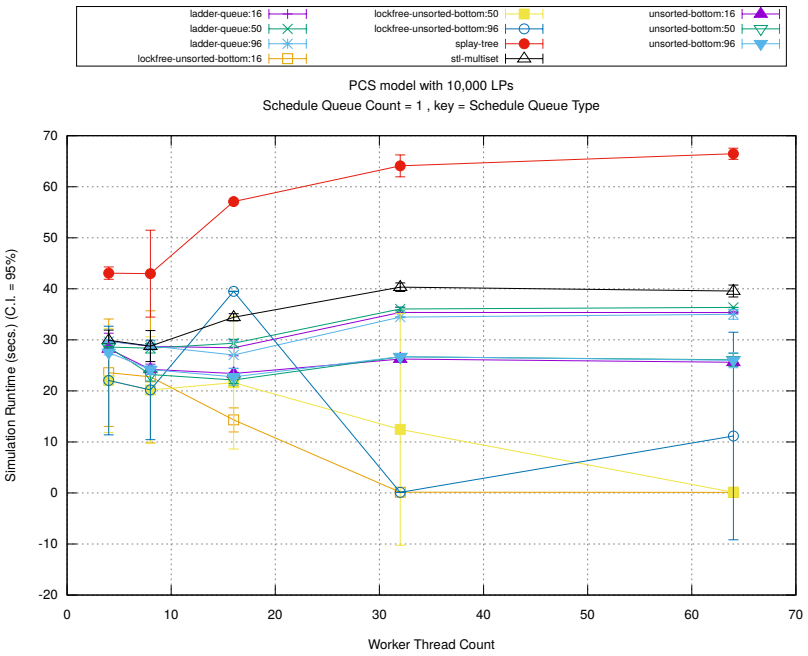


(b) Event Commitment Ratio



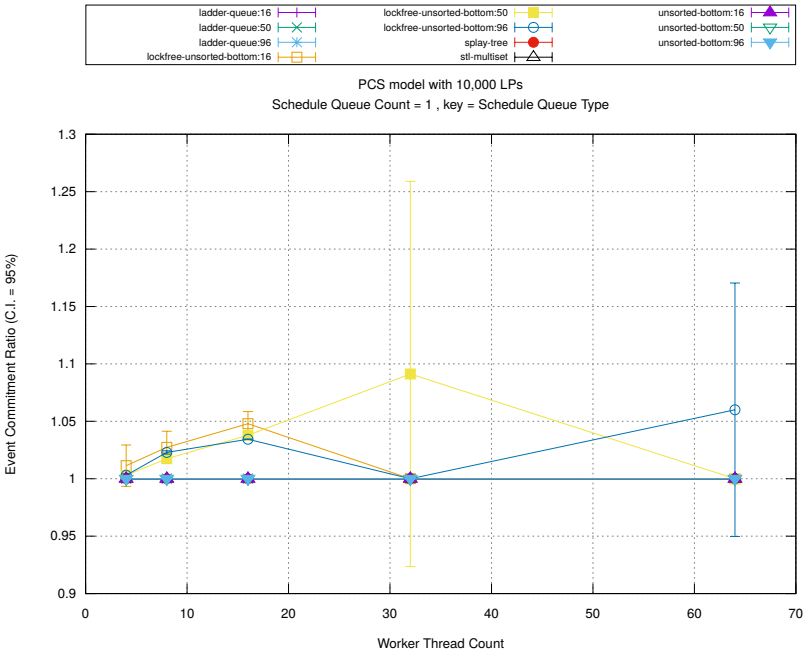
(c) Event Processing Rate (per second)

Figure 6.8: Impact of the schedule queue data structure: Traffic Model (*10,000* LPs)

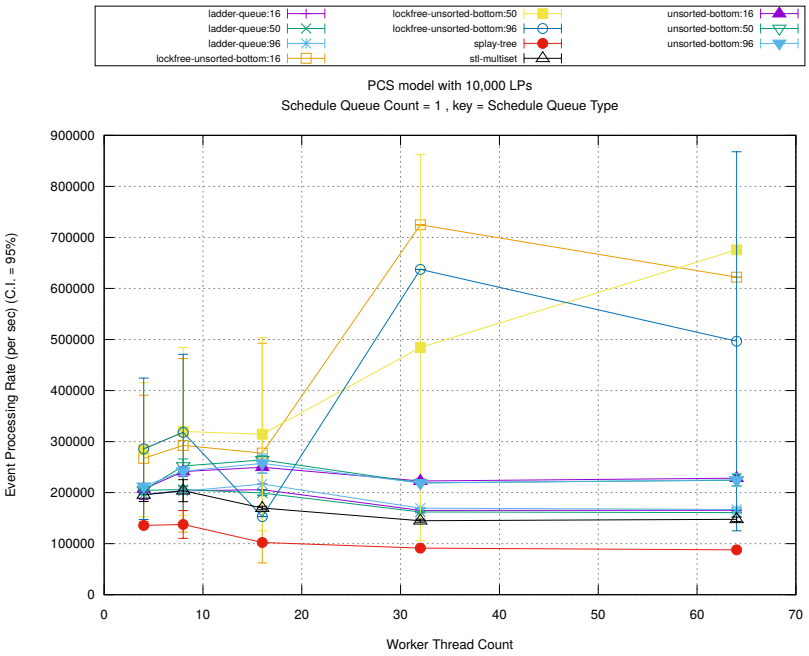


(a) Simulation Runtime (in seconds)

The PCS model, on the other hand, shows mixed performance for Ladder Queue with Lock-free Unsorted Bottom. In Figure 6.9, it starts off strongly but the performance tapers off with increase in number of worker threads. Ladder Queue with Lock-Free Unsorted Bottom is still faster than its locked counterpart but only by a small margin. Ladder Queue with Unsorted Bottom is around 20% faster than sorted Ladder Queue. Splay Tree is consistently the worst performer in every simulation model studied.

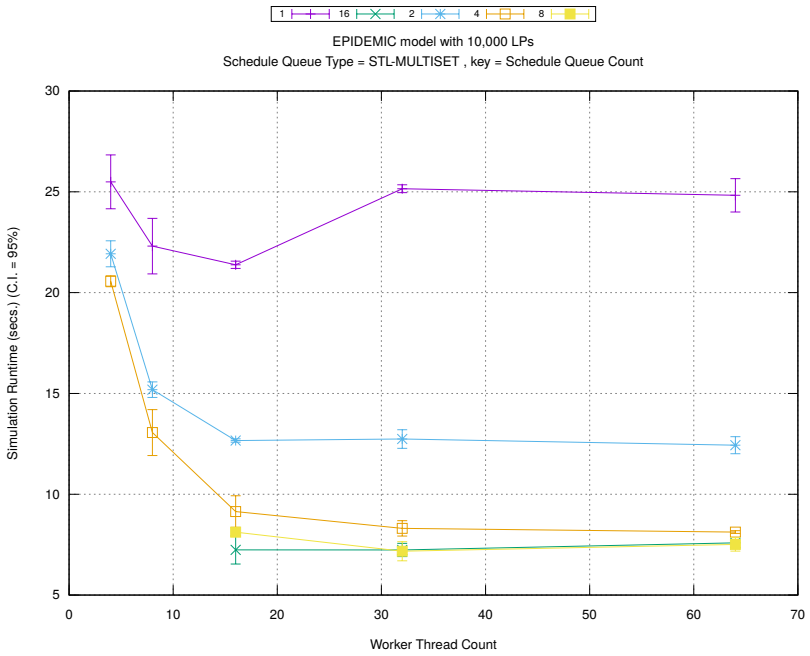


(b) Event Commitment Ratio



(c) Event Processing Rate (per second)

Figure 6.9: Impact of the schedule queue data structure: PCS Model (*10,000* LPs)

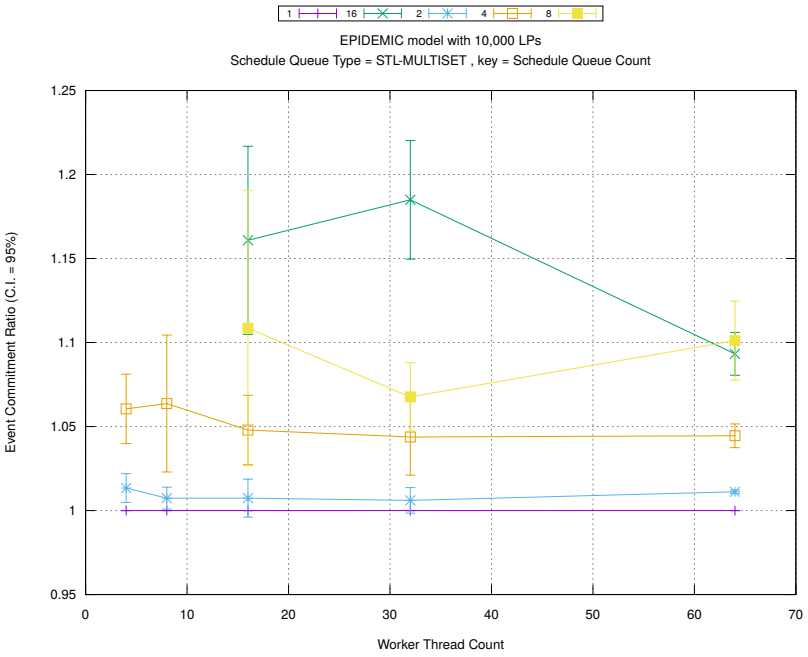


(a) Simulation Runtime (in seconds)

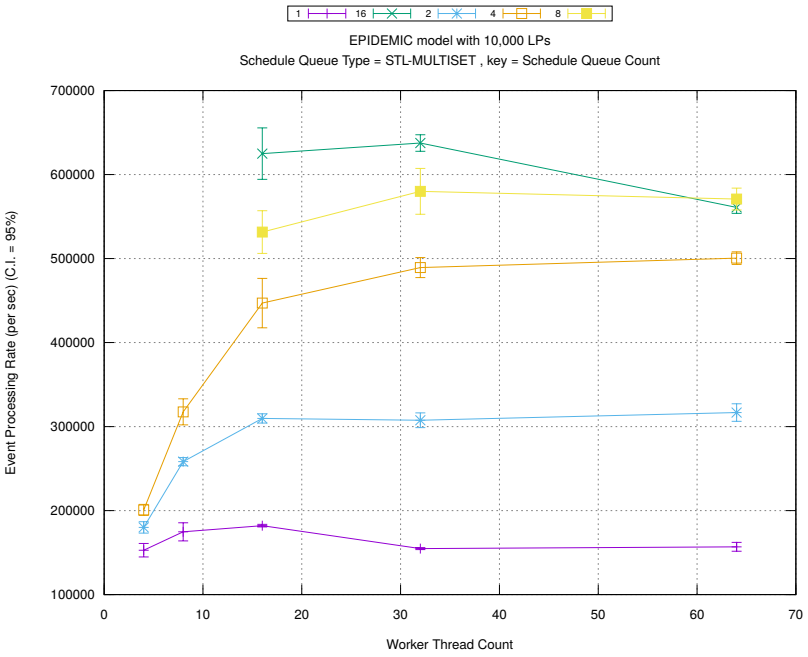
6.4.2 Multiple Schedule Queues

This quantitative study aims to understand the impact of distributed event processing by multiple schedule queues for different thread counts (4, 8, 16, 32 and 64). The schedule queue count never exceeds the thread count because each schedule queue is allocated atleast one thread for processing its stored events. Detailed description is available in Section 5.2.1. In order to retain clarity of presentation, only STL MultiSet is used as the underlying schedule queue data structure. The performance impact of multiple schedule queues when the underlying data structure is Splay Tree or Ladder Queue or any of its variants can be studied in plots presented in the Appendix.

Figure 6.10 shows the impact of partitioning the pending event set uniformly across multiple Schedule Queues. Section 5.2.1 explains this organization in details. The plots show that performance improves steadily as the number of schedule queues is increased up to 4, beyond which there is no further improvement. This is because there is a significant increase in number of rollbacks when 8 or 16 schedule queues are used.

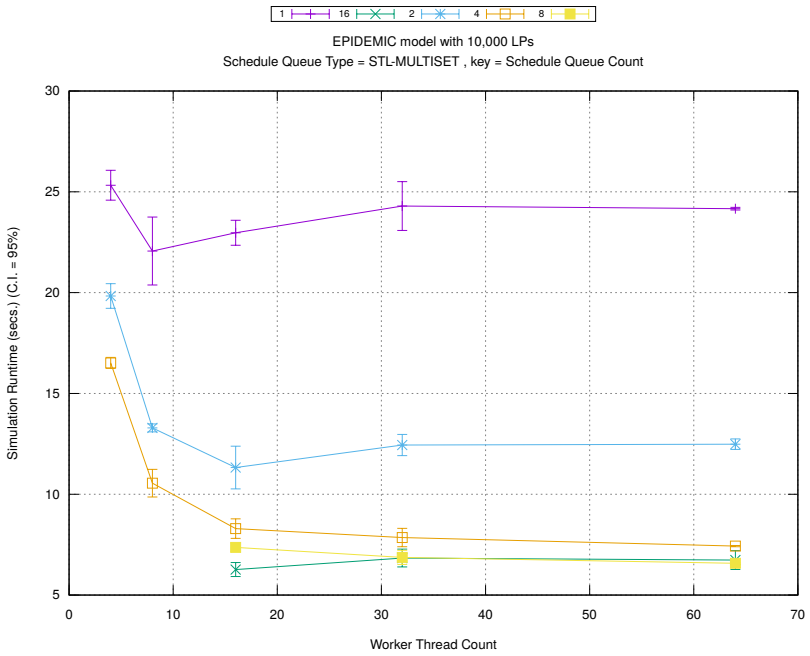


(b) Event Commitment Ratio



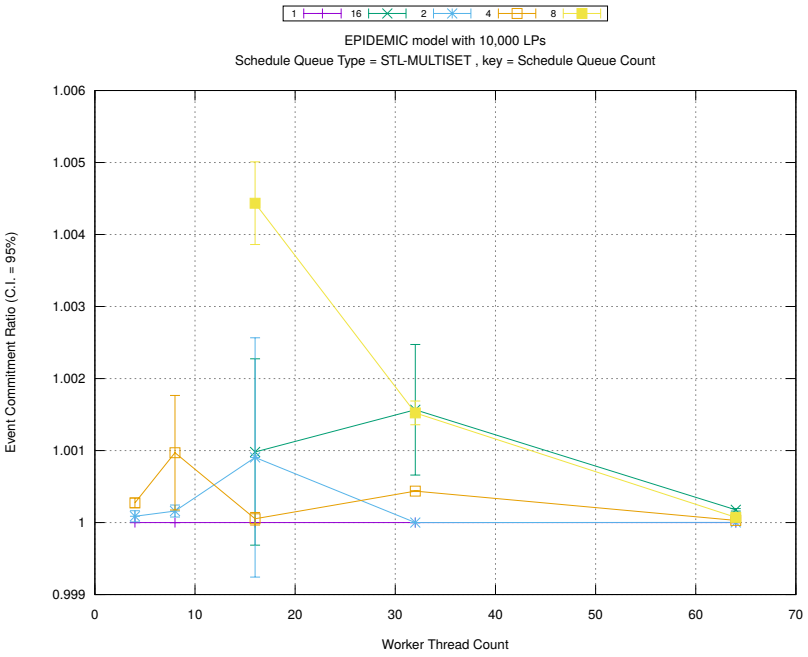
(c) Event Processing Rate (per second)

Figure 6.10: Impact of the multiple schedule queues: Epidemic Model (*10,000* LPs, *Barabsi-Albert* diffusion network)

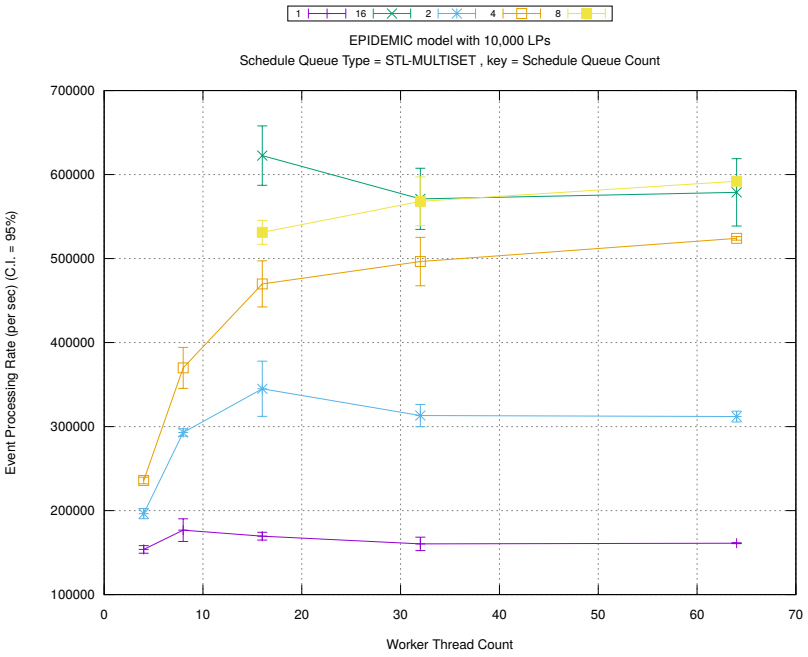


(a) Simulation Runtime (in seconds)

Figure 6.11 shows performance which is quite similar to that shown in Figure 6.10, except for the low rollback count for higher number of schedule queues. The low event event density within any time window of an epidemic model helps to preserve better balance in a distributed simulation by not throwing rollbacks too frequently.

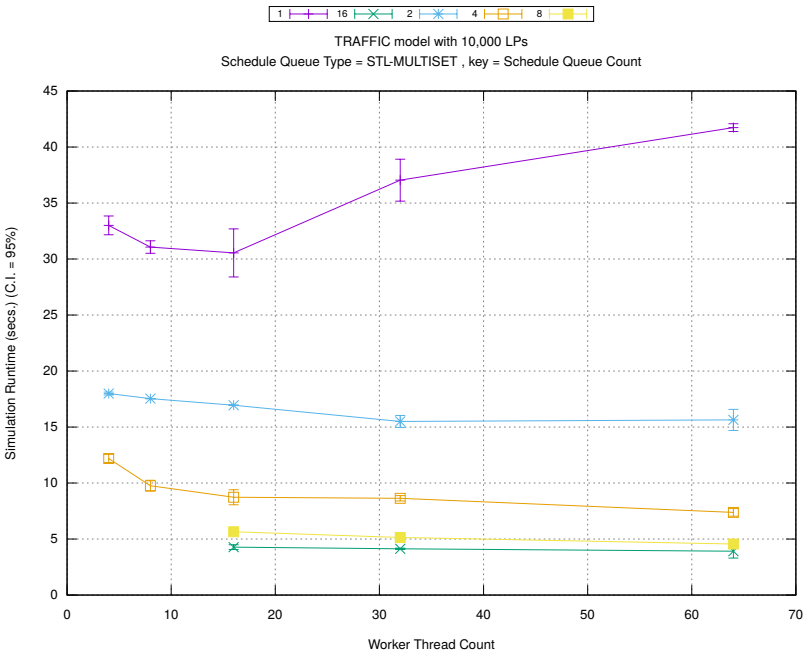


(b) Event Commitment Ratio



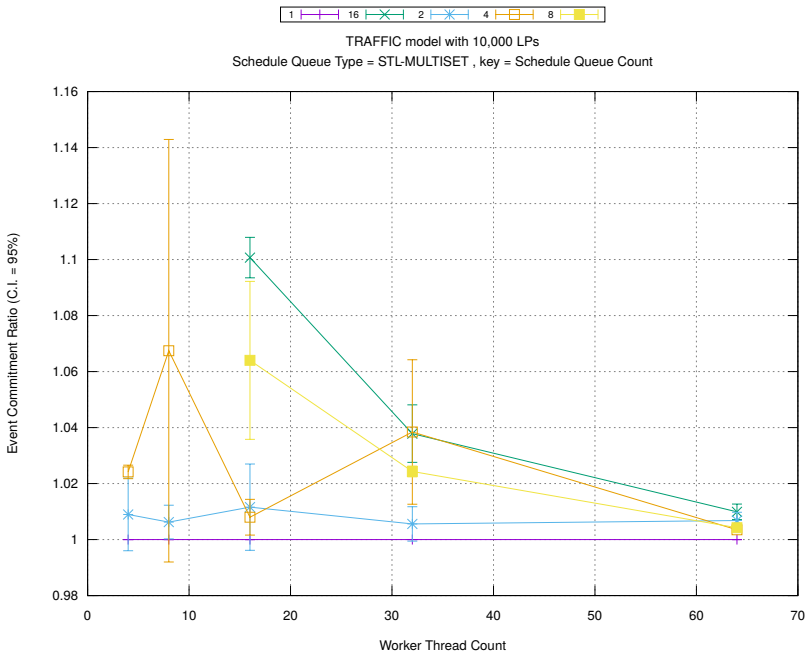
(c) Event Processing Rate (per second)

Figure 6.11: Impact of the multiple schedule queues: Epidemic Model (*10,000* LPs, *Watts-Strogatz* diffusion network)

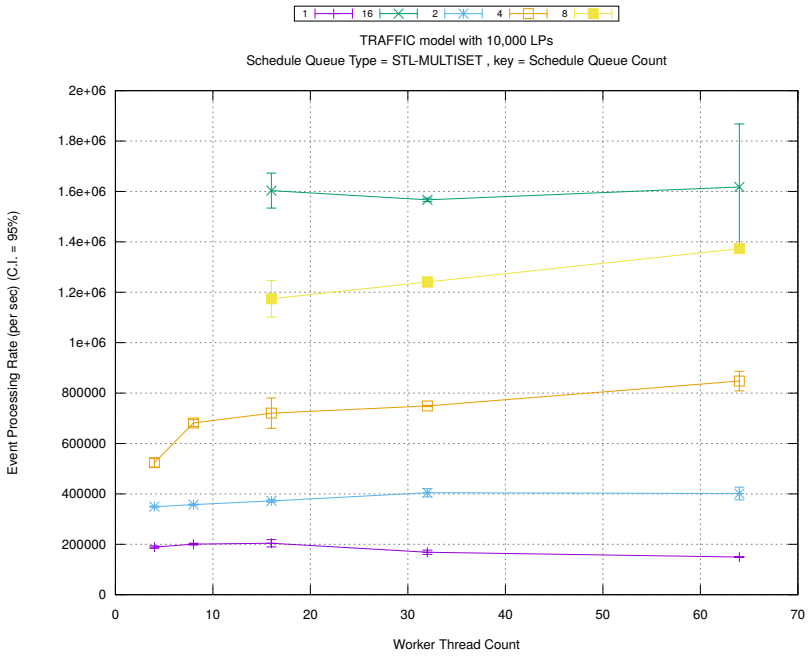


(a) Simulation Runtime (in seconds)

In comparison, Figure 6.12 shows improvement in performance for traffic model till the number of schedule queues is increased up to 8 beyond which there is no further gain. The behavior of rollbacks here is a bit surprising given the reduction in their number for thread count 32 and higher.

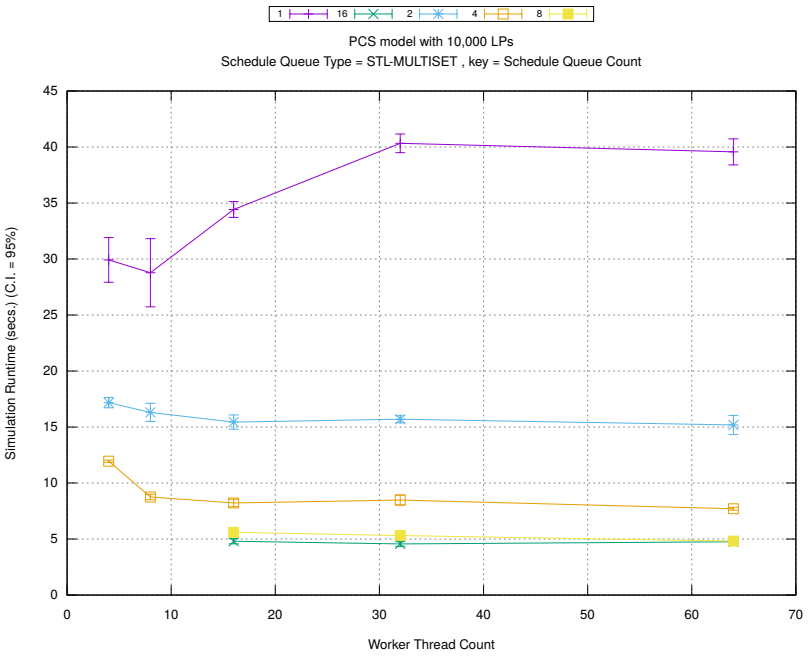


(b) Event Commitment Ratio



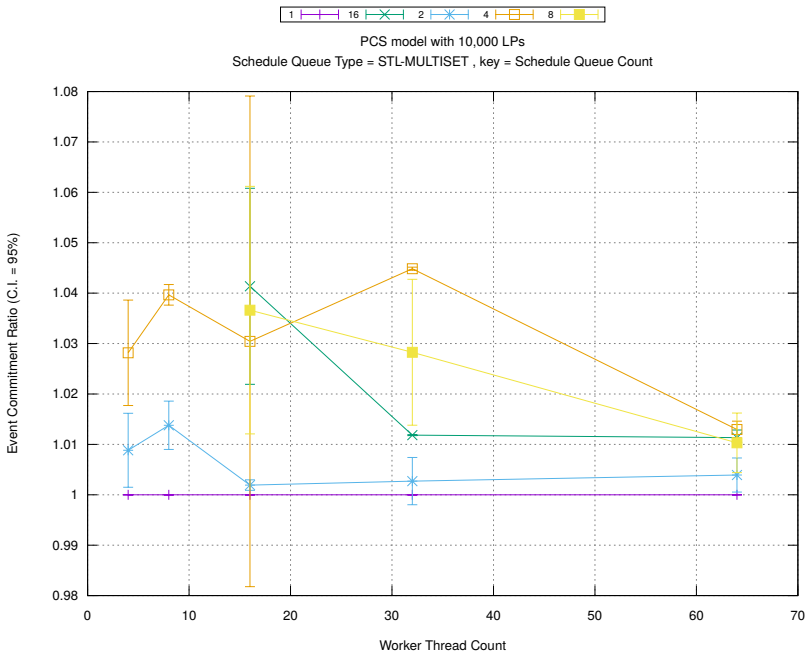
(c) Event Processing Rate (per second)

Figure 6.12: Impact of the multiple schedule queues: Traffic Model (*10,000* LPs)

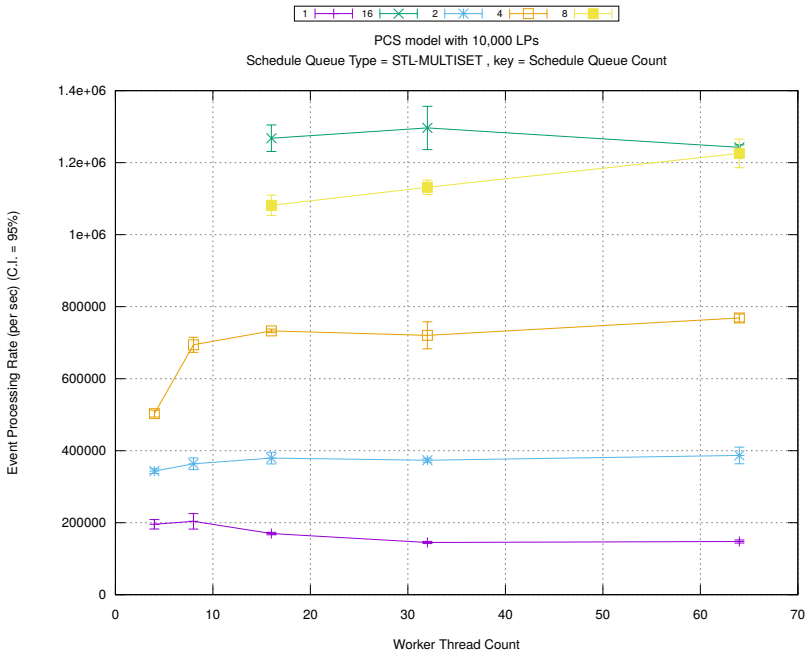


(a) Simulation Runtime (in seconds)

Figure 6.13 shows the performance of PCS model. It is similar to that of traffic model shown in Figure 6.12.



(b) Event Commitment Ratio



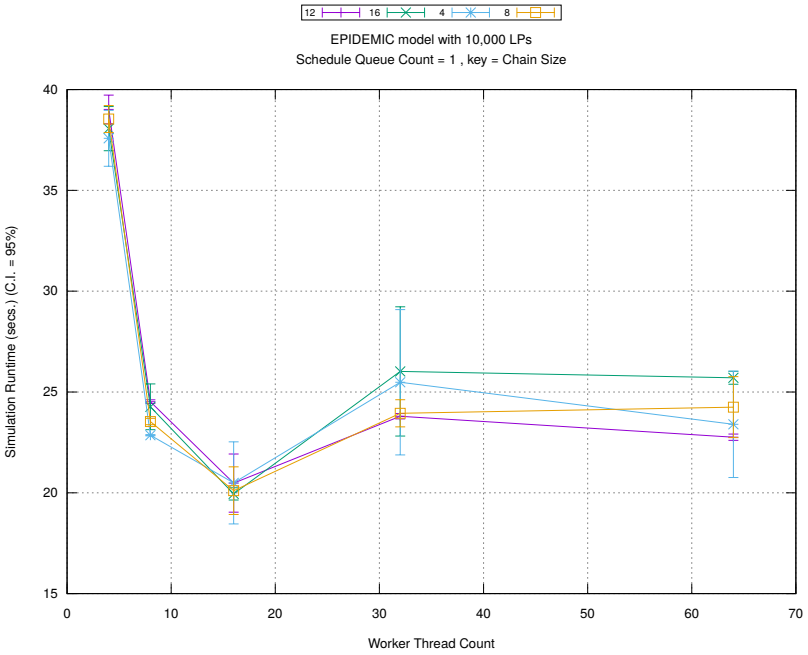
(c) Event Processing Rate (per second)

Figure 6.13: Impact of the multiple schedule queues: PCS Model (*10,000* LPs)

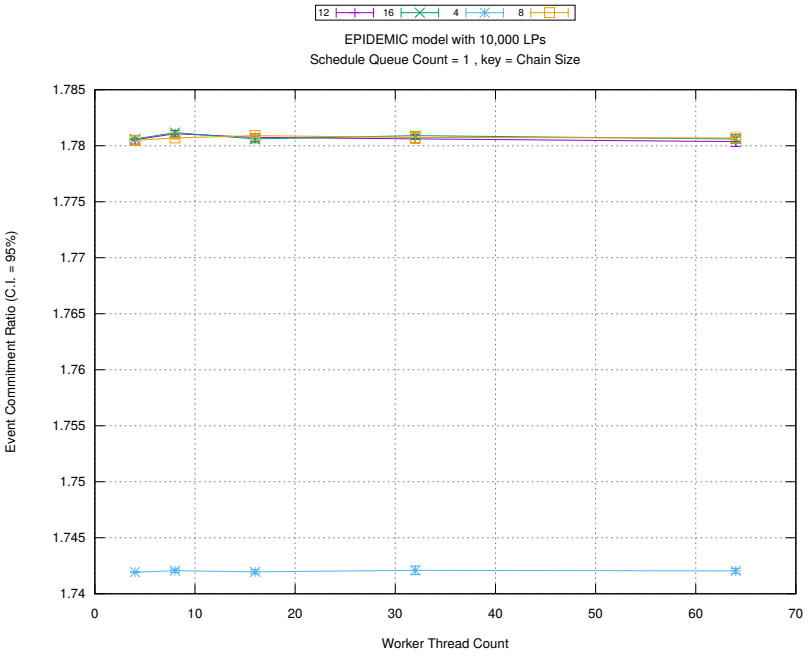
6.4.3 Chain Scheduling

This quantitative analysis aims to study the effect of different chain sizes on the performance of of WARPED-2. Section 5.2.2 presents a detailed description of event chain and its size selection parameter. In order to simplify this analysis, only one schedule queue has been used in order to avoid the adverse effects of load imbalance that often plagues simulations using multiple schedule queues. The performance of WARPED-2 when chains are used in combination with multiple schedule queues can be studied from plots presented in the Appendix. The underlying data structure for the schedule queue here is STL MultiSet and different thread counts used are 4, 8, 16, 32 and 64.

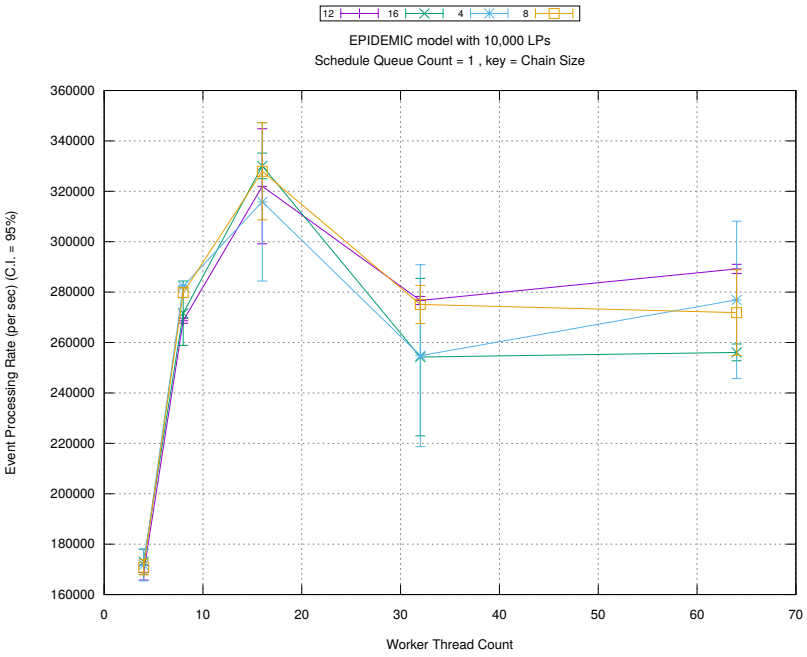
Figures 6.14 and 6.15 show the performance of Epidemic model when chains of varying sizes are used. Both configurations of the model have identical performance which remains fairly steady with increase in number of worker threads. Chain size only has marginal effect on performance. It is interesting to note that rollback count is invariant for varying number of worker threads and fairly similar for different chain sizes. The similar rollback count is likely because the local event chain size is mostly 1 (as shown in Figure 5.7). The performance of Chain Scheduling, which uses STL MultiSet as Schedule Queue, is similar to that of a single Schedule Queue (shown in Figure 6.6). Refer to Section 5.2.2 for a detailed explanation of event chain.



(a) Simulation Runtime (in seconds)

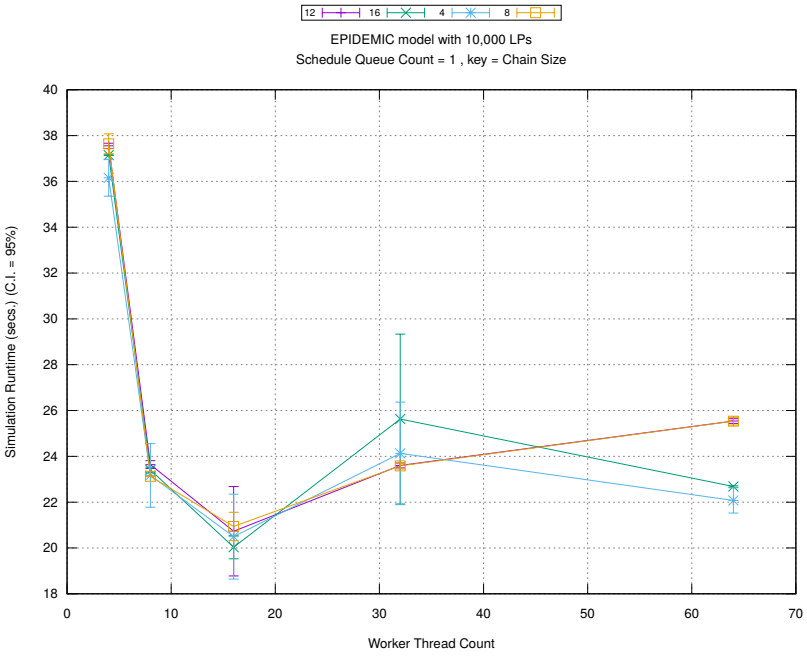


(b) Event Commitment Ratio

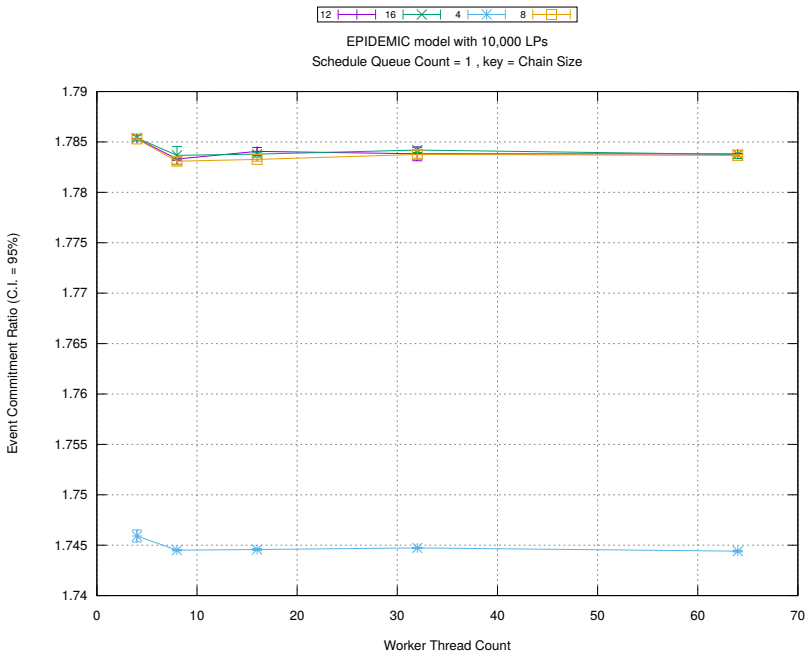


(c) Event Processing Rate (per second)

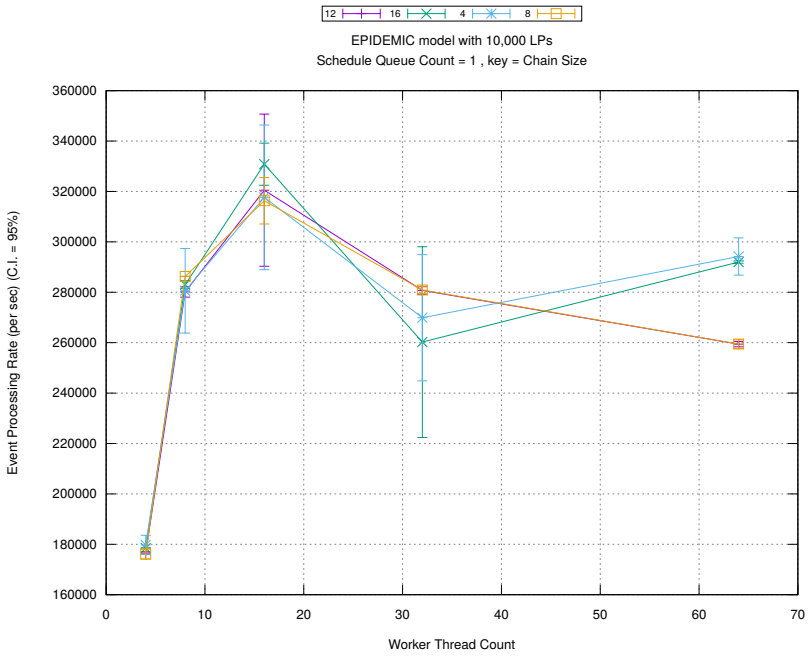
Figure 6.14: Performance of Event Chains: Epidemic Model (*10,000* LPs, *Barabsi-Albert* diffusion network)



(a) Simulation Runtime (in seconds)

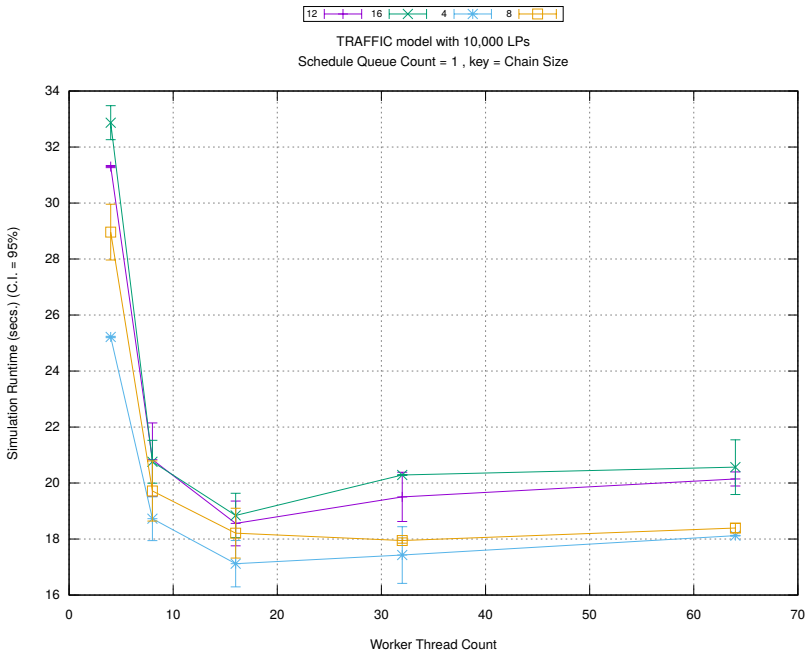


(b) Event Commitment Ratio



(c) Event Processing Rate (per second)

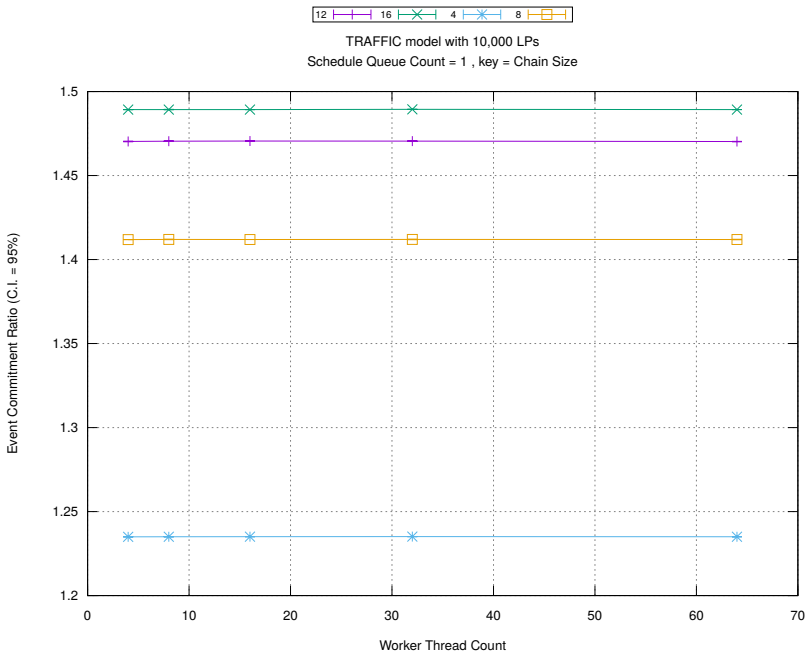
Figure 6.15: Performance of Event Chains: Epidemic Model (*10,000* LPs, *Watts-Strogatz* diffusion network)



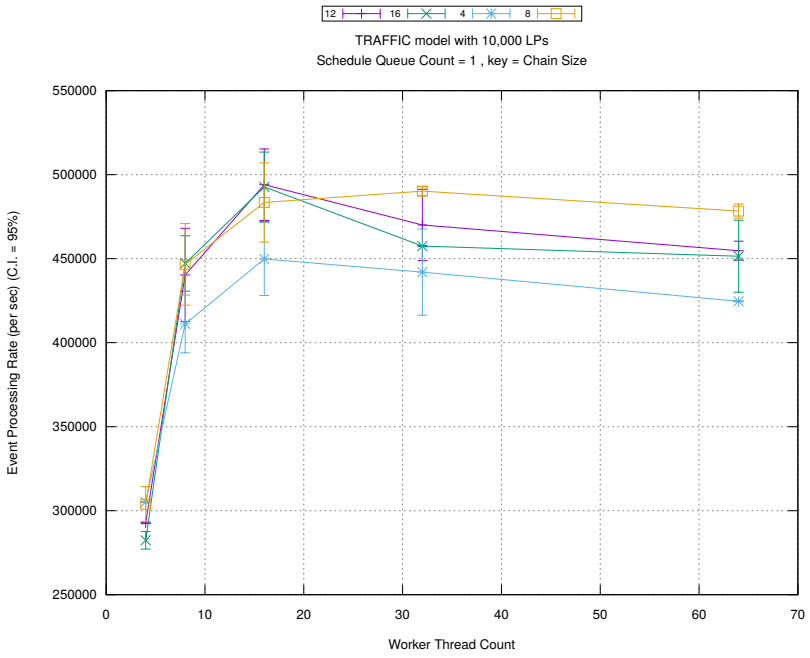
(a) Simulation Runtime (in seconds)

Figure 6.16 shows that the performance of Traffic model with 10,000 LPs remains invariant with change in chain size and worker thread count. Similar to Figures 6.14 and 6.15, rollback count is invariant for varying number of worker threads but is significantly different for different chain sizes.

For Traffic simulation with 1,048,576 LPs, Figure 6.17 shows an unexpected loss of performance for 16 threads. The simulation is fairly stable for other configurations.

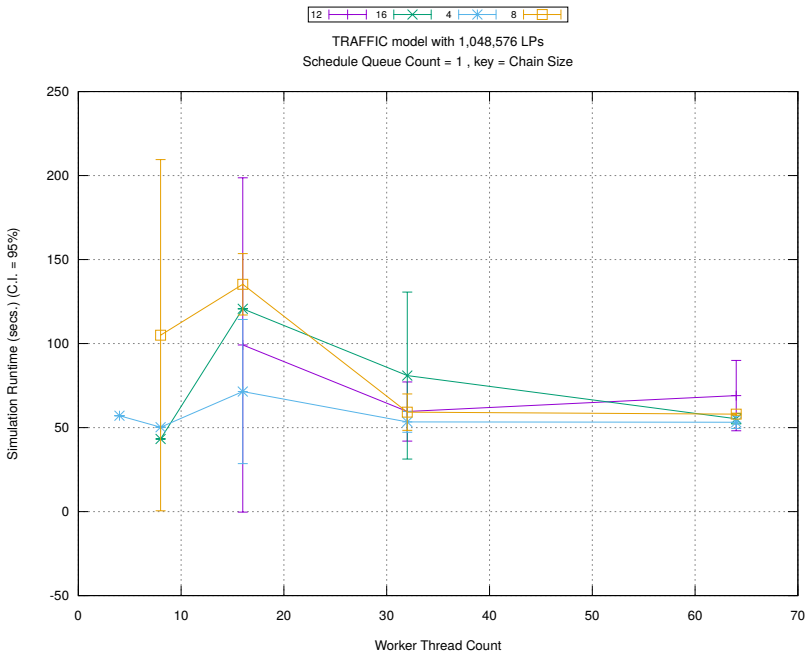


(b) Event Commitment Ratio

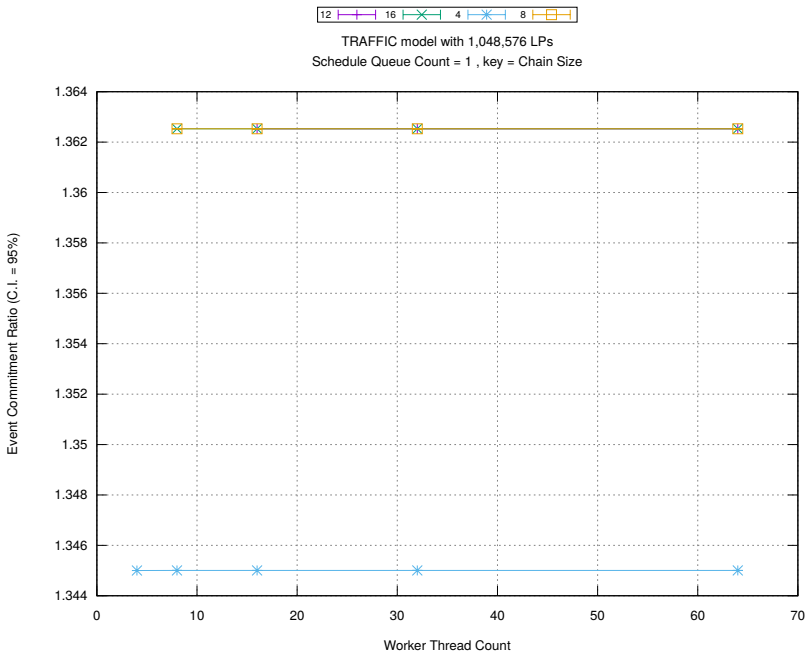


(c) Event Processing Rate (per second)

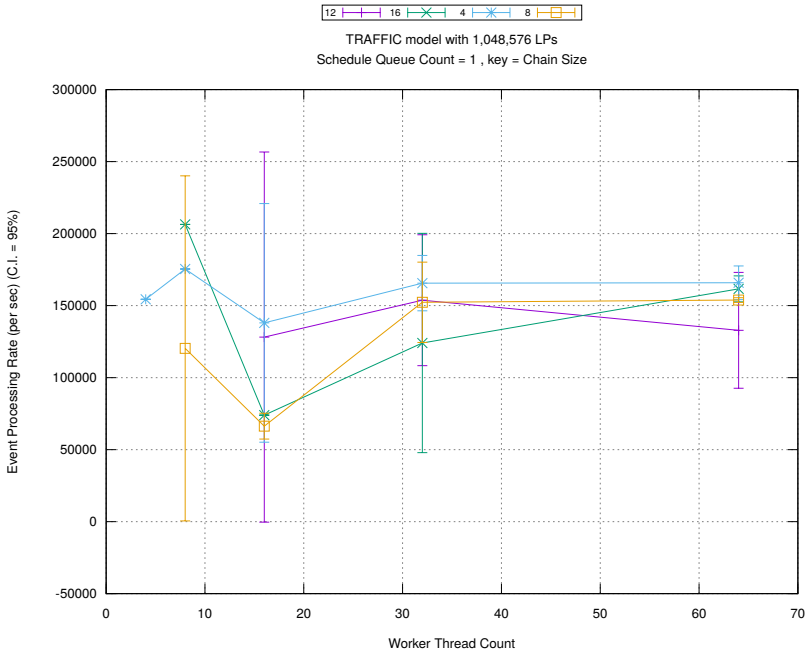
Figure 6.16: Performance of Event Chains: Traffic Model (*10,000* LPs)



(a) Simulation Runtime (in seconds)

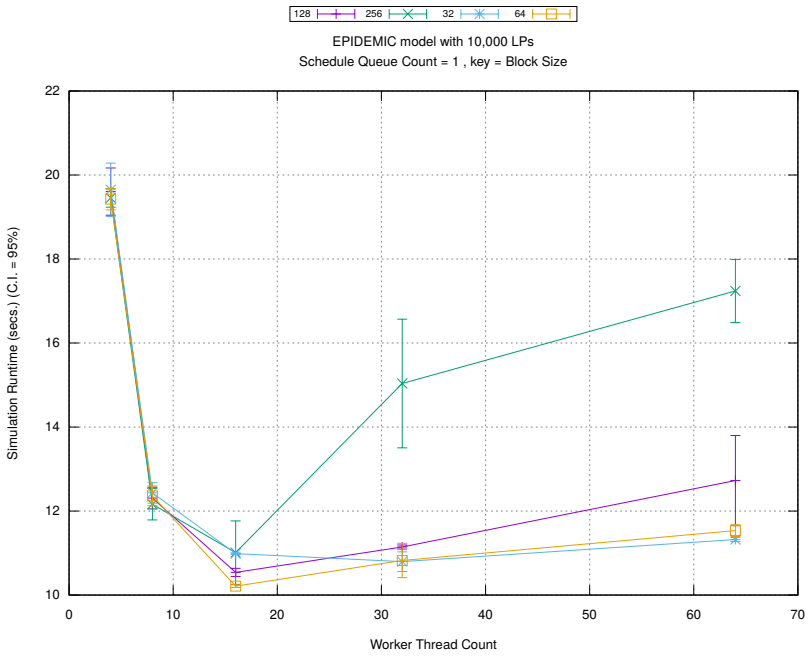


(b) Event Commitment Ratio



(c) Event Processing Rate (per second)

Figure 6.17: Performance of Event Chains: Traffic Model (*1,048,576* LPs)

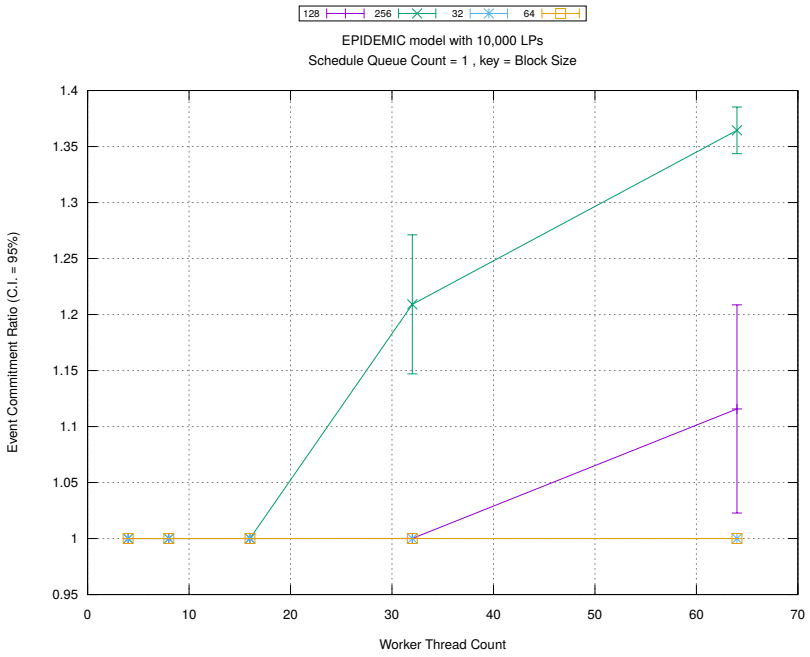


(a) Simulation Runtime (in seconds)

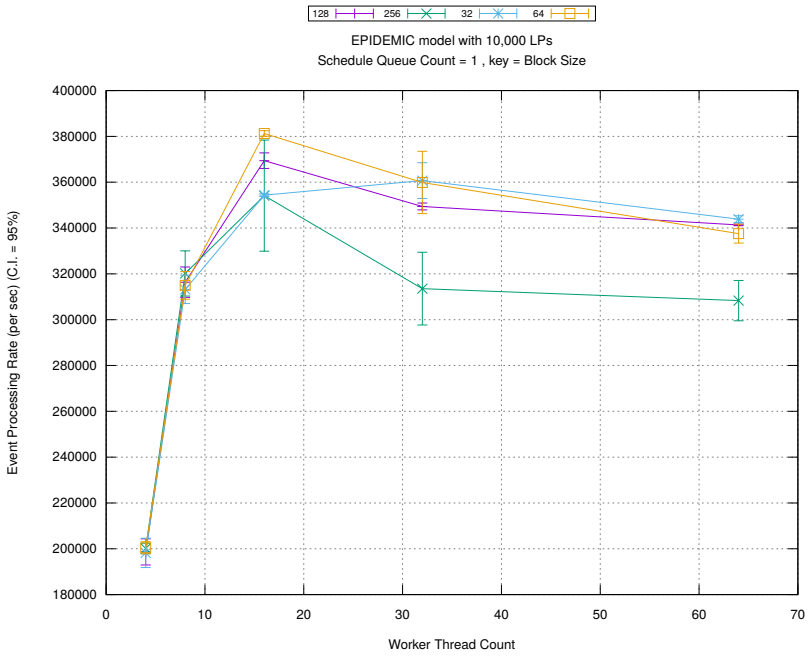
6.4.4 Block Scheduling

This quantitative analysis aims to study the effect of different block sizes on the performance of of WARPED-2. Section 5.2.3 presents a detailed description of blocks and its size selection parameter. In order to simplify this analysis, only one schedule queue has been used in order to avoid the adverse effects of load imbalance that often plagues simulations using multiple schedule queues. The performance of WARPED-2 when blocks are used in combination with multiple schedule queues can be studied from plots presented in the Appendix. The underlying data structure for the schedule queue here is STL MultiSet and different thread counts used are 4, 8, 16, 32 and 64.

Figures 6.18 and 6.19 show the performance of Epidemic model when blocks of varying sizes are used. Both configurations of the model have identical performance which remains fairly steady with increase in number of worker threads. Block size only has marginal effect on performance. It is interesting to note that block scheduling has better performance when compared to that of chain scheduling. Refer to Section 5.2.3 for a detailed explanation of event blocks.

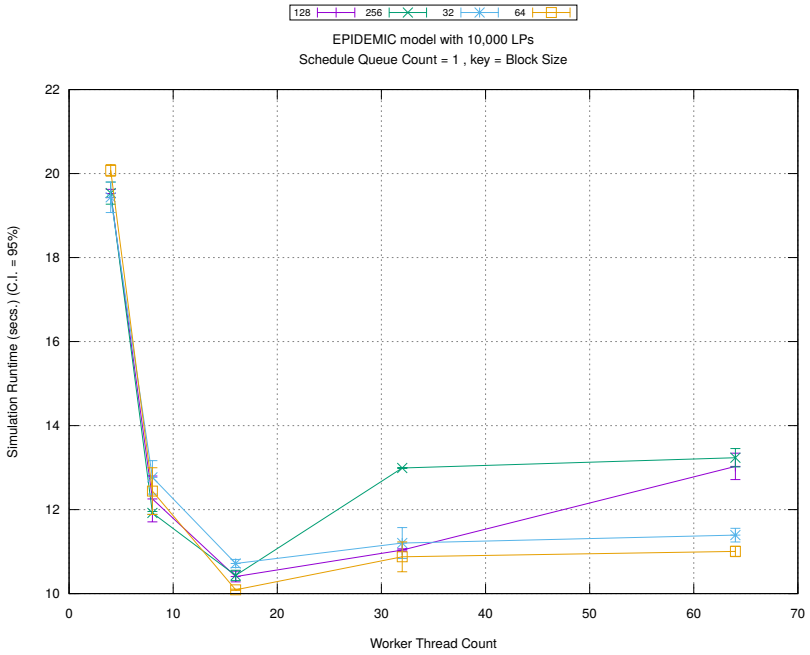


(b) Event Commitment Ratio

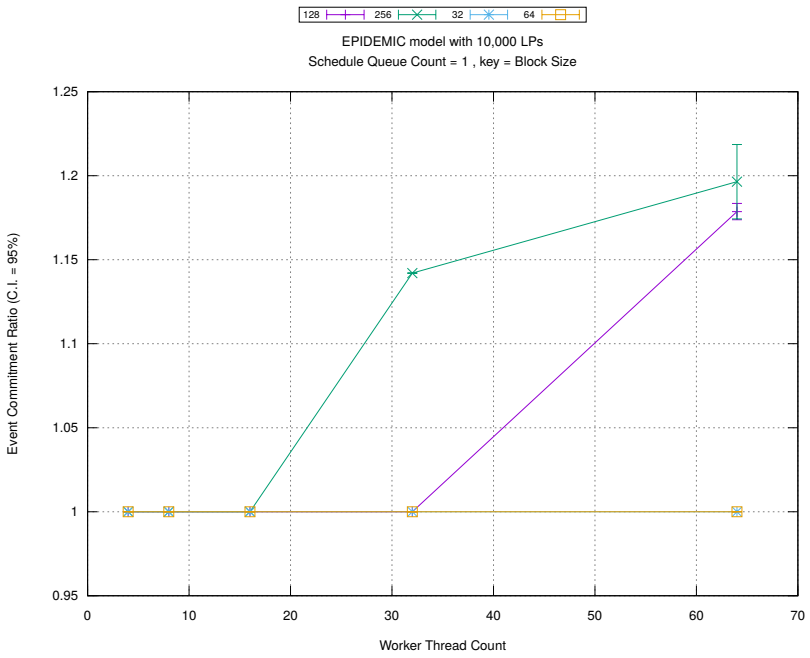


(c) Event Processing Rate (per second)

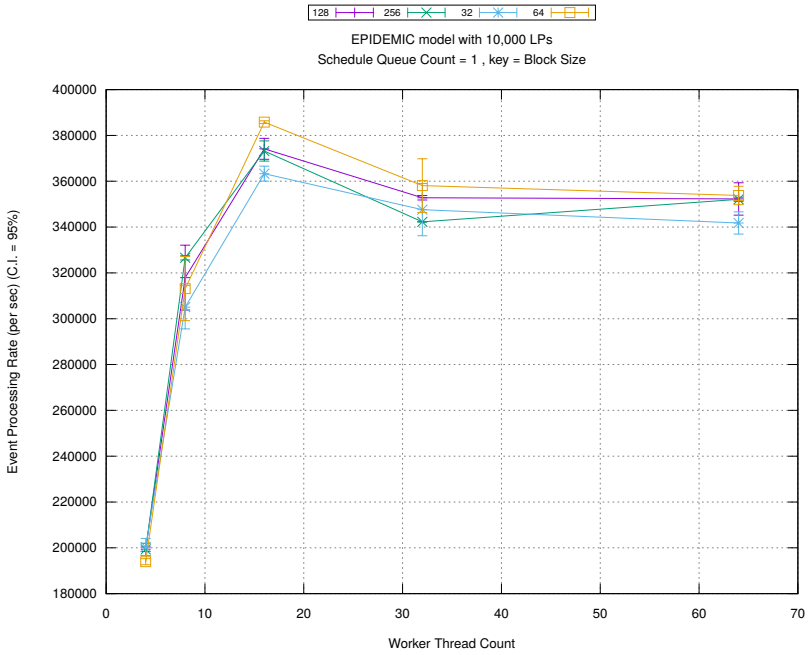
Figure 6.18: Performance of Event Blocks: Epidemic Model (*10,000* LPs, *Barabsi-Albert* diffusion network)



(a) Simulation Runtime (in seconds)

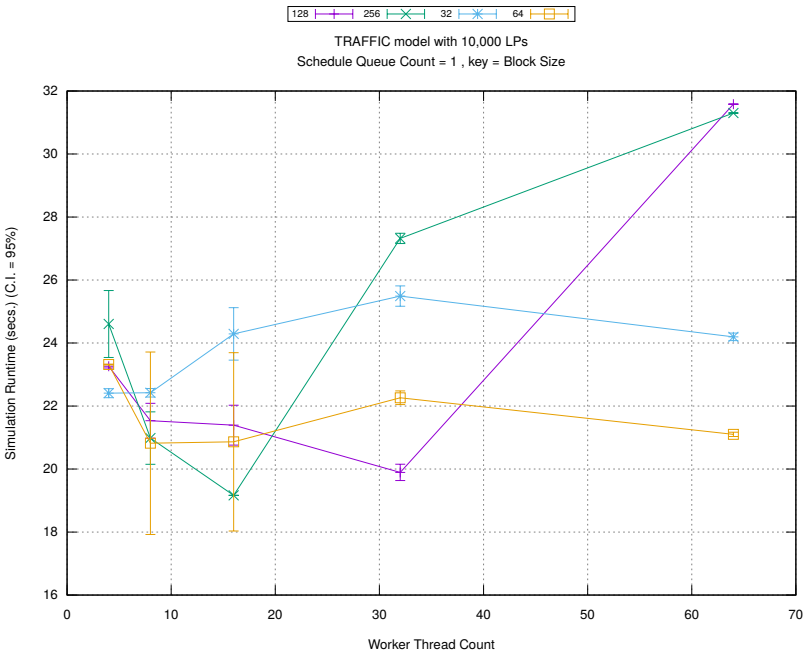


(b) Event Commitment Ratio



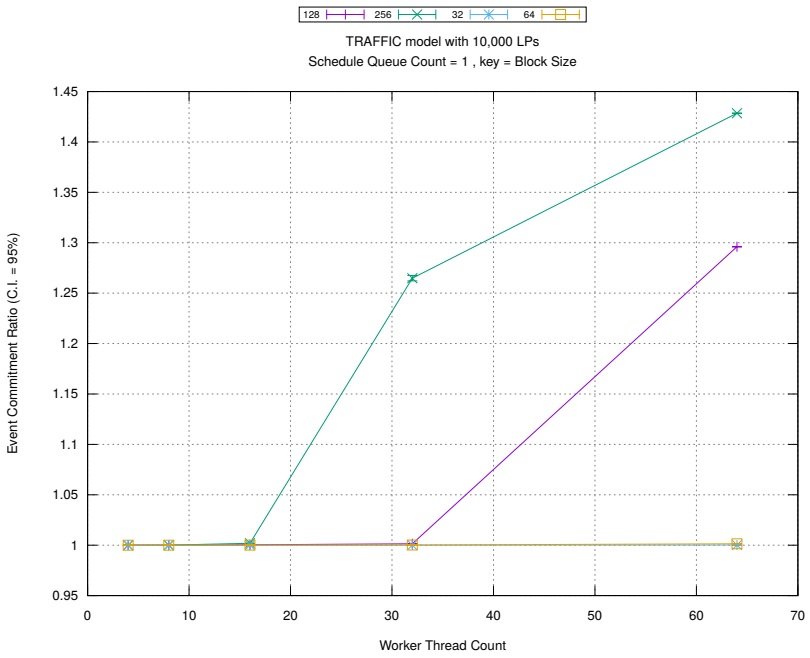
(c) Event Processing Rate (per second)

Figure 6.19: Performance of Event Blocks: Epidemic Model (*10,000* LPs, *Watts-Strogatz* diffusion network)

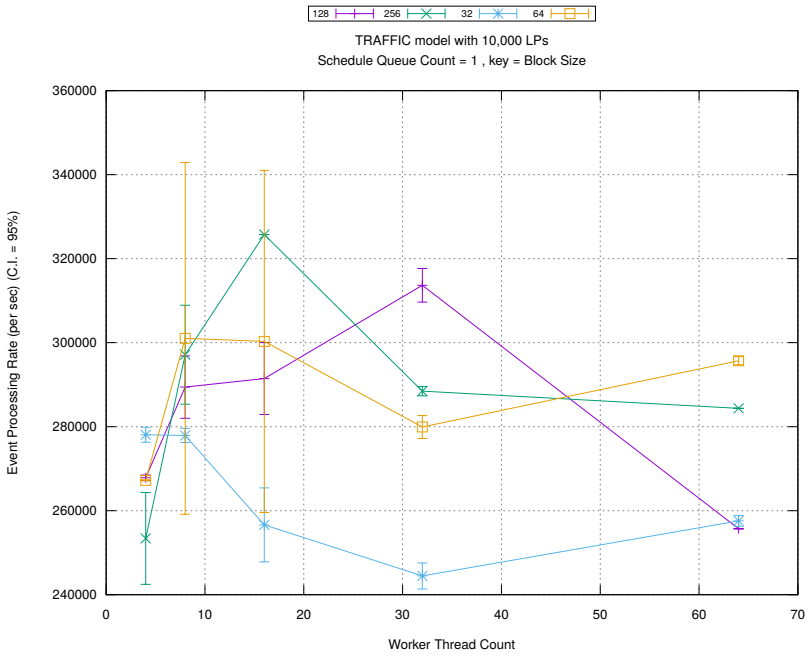


(a) Simulation Runtime (in seconds)

Figure 6.20 shows that the performance of Traffic model with 10,000 LPs remains invariant with change in block size and worker thread count.

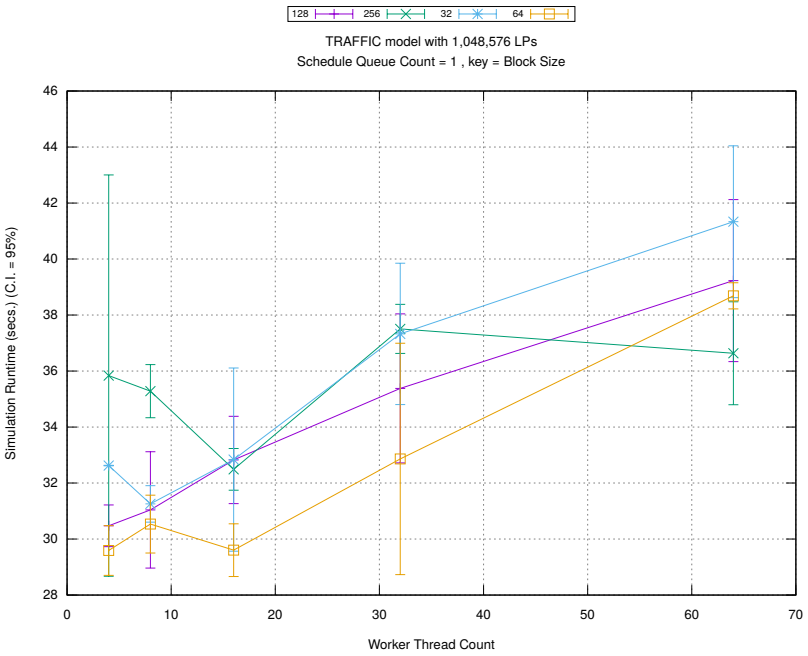


(b) Event Commitment Ratio



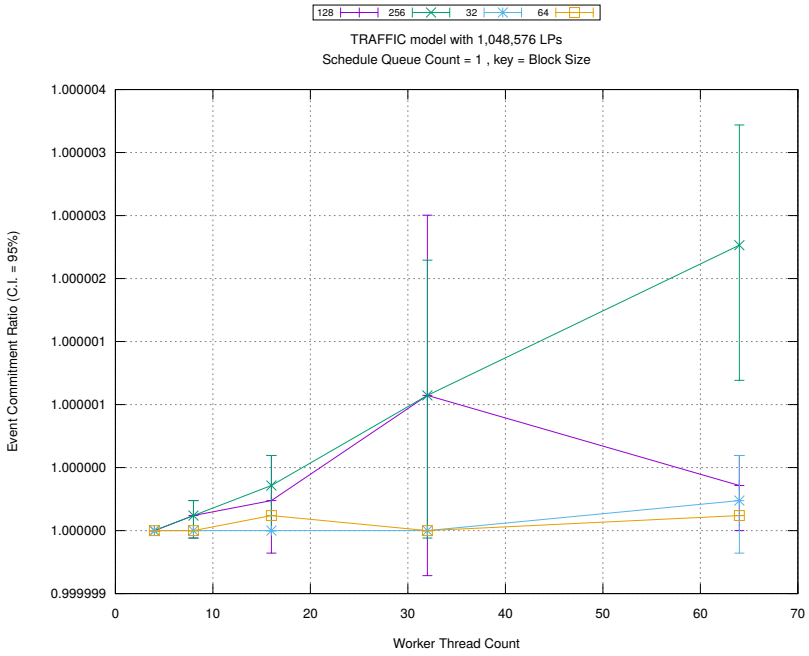
(c) Event Processing Rate (per second)

Figure 6.20: Performance of Event Blocks: Traffic Model (*10,000* LPs)

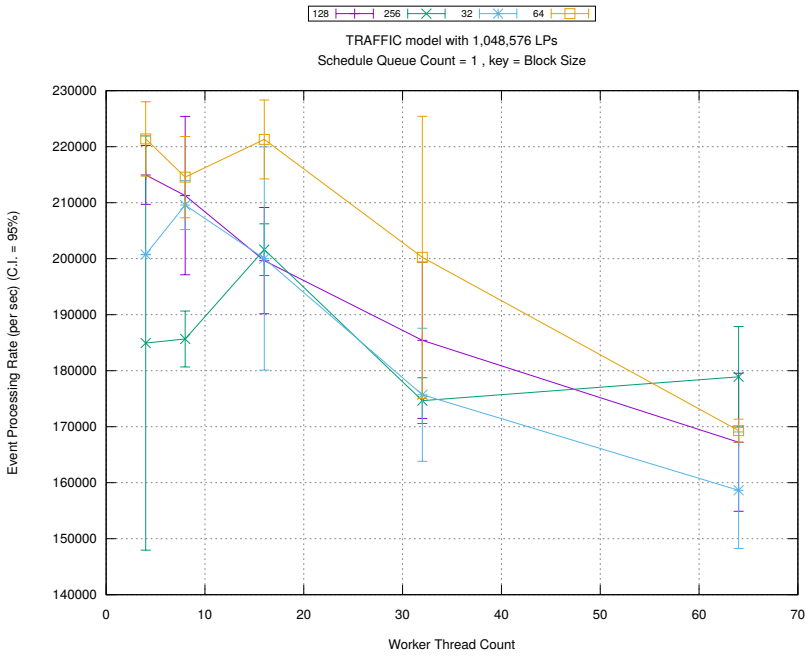


(a) Simulation Runtime (in seconds)

Figure 6.21 shows a Traffic simulation with 1,048,576 LPs behaves similar to the 10,000 LP traffic simulation.

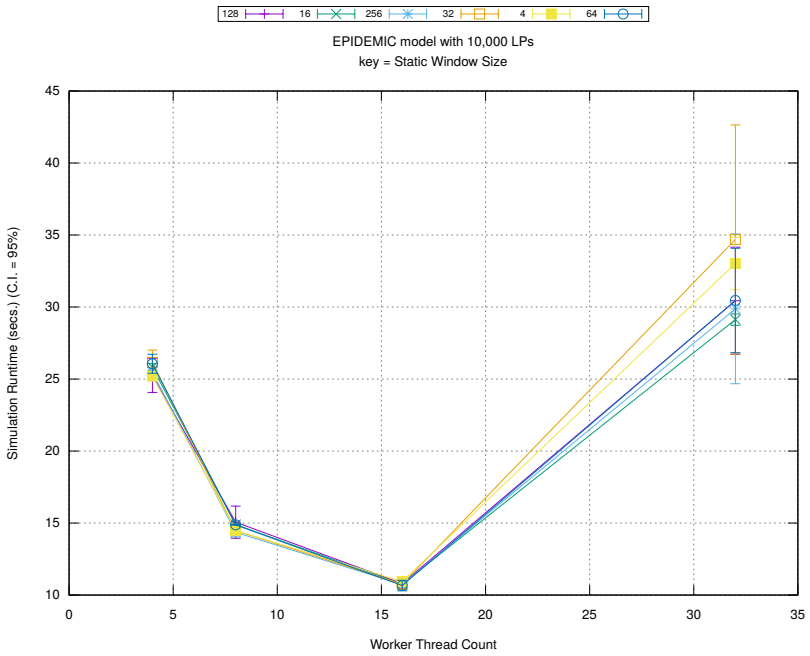


(b) Event Commitment Ratio



(c) Event Processing Rate (per second)

Figure 6.21: Performance of Event Blocks: Traffic Model (*1,048,576* LPs)

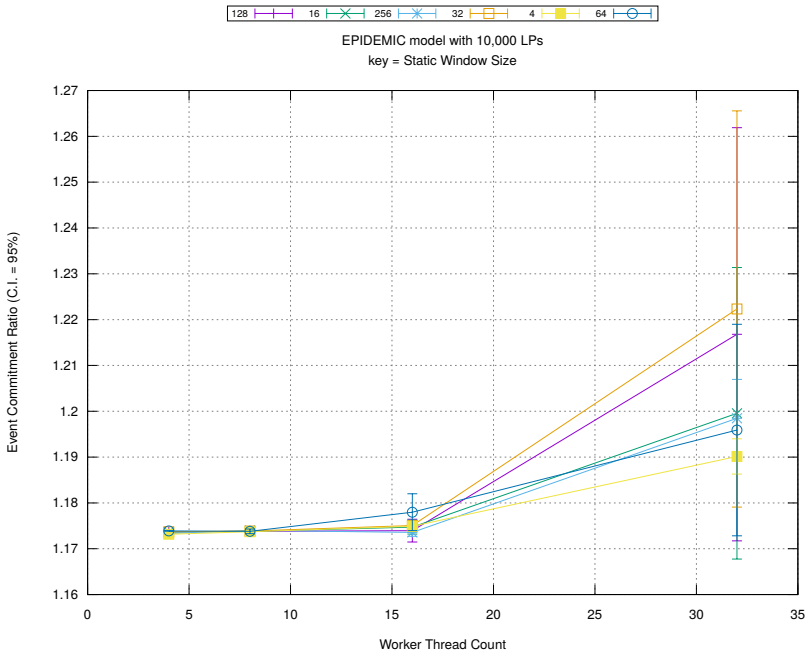


(a) Simulation Runtime (in seconds)

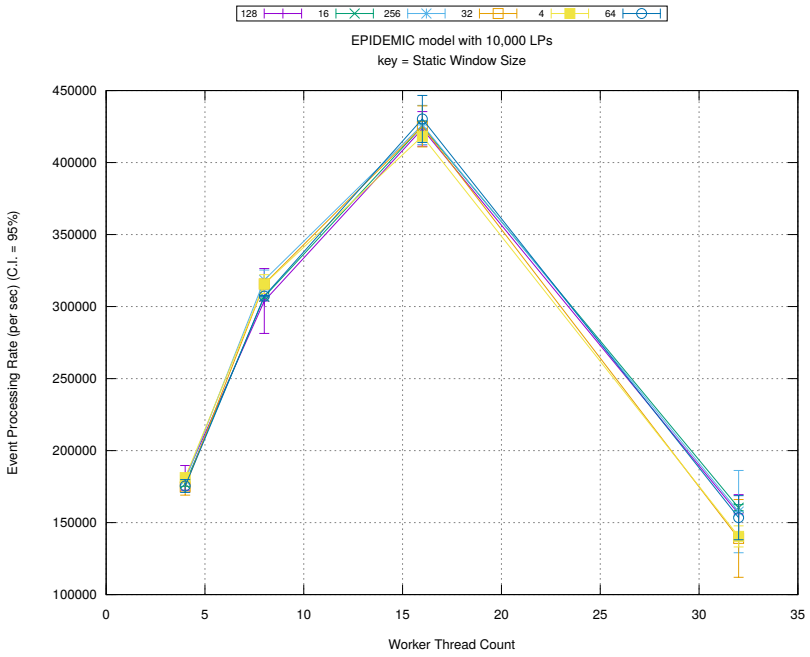
6.4.5 Bags with Static Window Size

In this study, the performance of bags and its static window size selection parameter are being quantitatively analyzed. Section 5.2.4 presents a detailed description of bags. The profile-driven partitioning for each benchmark model is driven by the event profile collected during sequential simulation. Configuration for this sequential simulation can be found in the configuration tables in the Appendix, namely Tables A.3, A.4, A.5, A.6, A.1, A.2 and A.7. The performance of WARPED-2 is being evaluated for different thread counts (4, 8, 16, 32 and 64) when using bags with different static window sizes, namely 4, 16, 32, 64, 128 and 256.

Figures 6.22 and 6.23 show the performance of Epidemic Model where events are scheduled from profile-driven LP partitions (refer to Section 5.2.4 for details). Both configurations show fairly similar performance which peaks at 16 threads. From the plots, it is evident that static window size has little or no impact up to thread count of 32. Similar to event chains, the rollback count remains invariant to increase in number of worker threads and static window size. Epidemic model using Watts-Strogatz diffusion network has lesser number of rollbacks compared to the Epidemic model using Barabasi-Albert diffusion network. It is worth noting that performance of bags is similar and sometimes superior to that of Ladder Queue with Lock-free Unsorted Bottom in Figures 6.6 and 6.7.

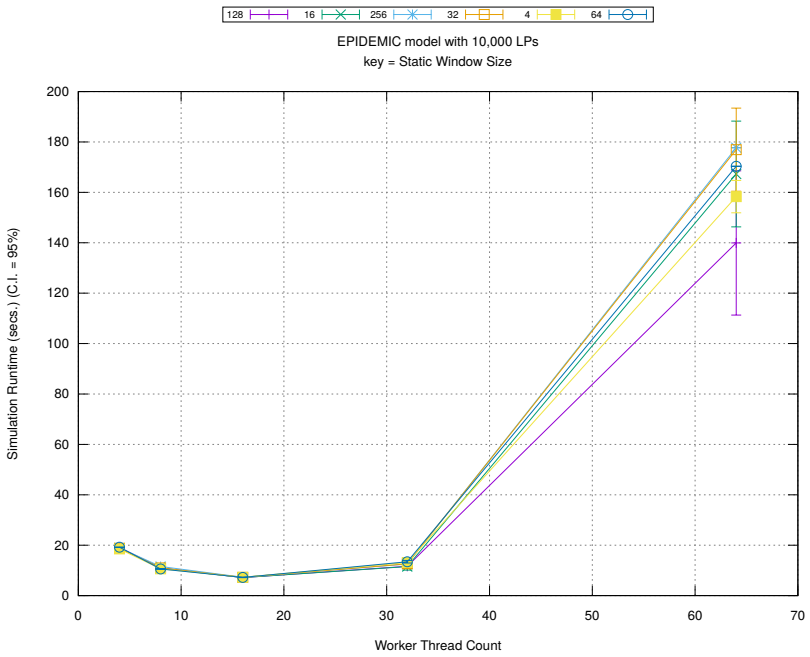


(b) Event Commitment Ratio

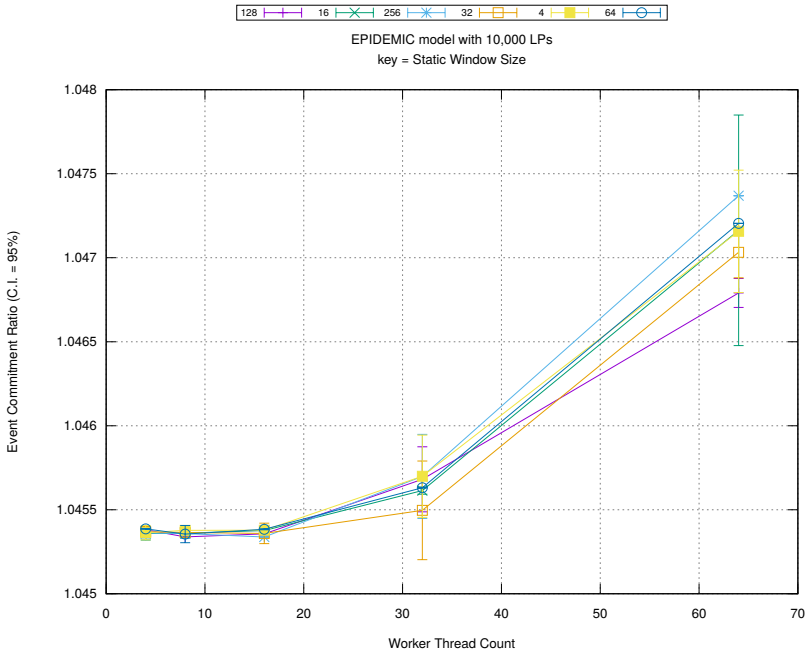


(c) Event Processing Rate (per second)

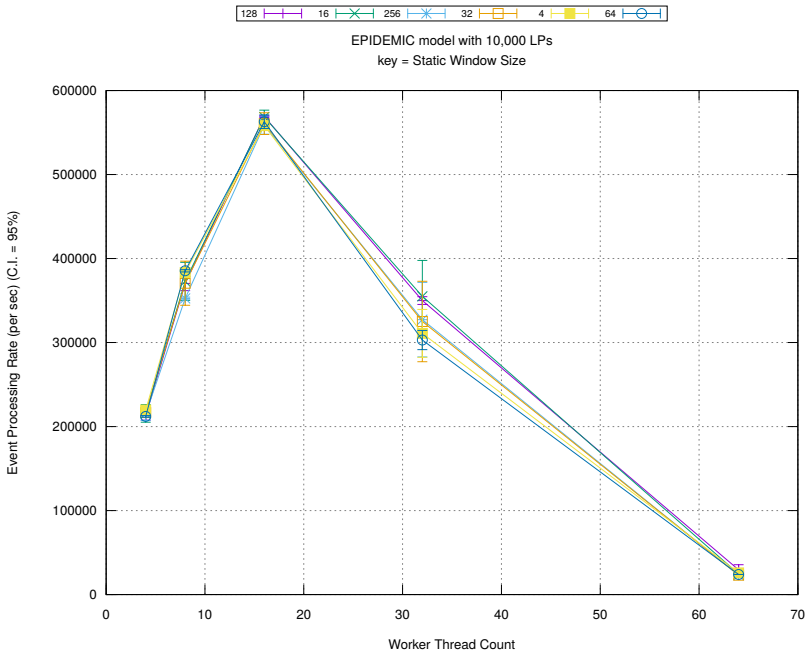
Figure 6.22: Performance of Bags with Static Window Size: Epidemic Model (*10,000* LPs, *Barabsi-Albert* diffusion network)



(a) Simulation Runtime (in seconds)

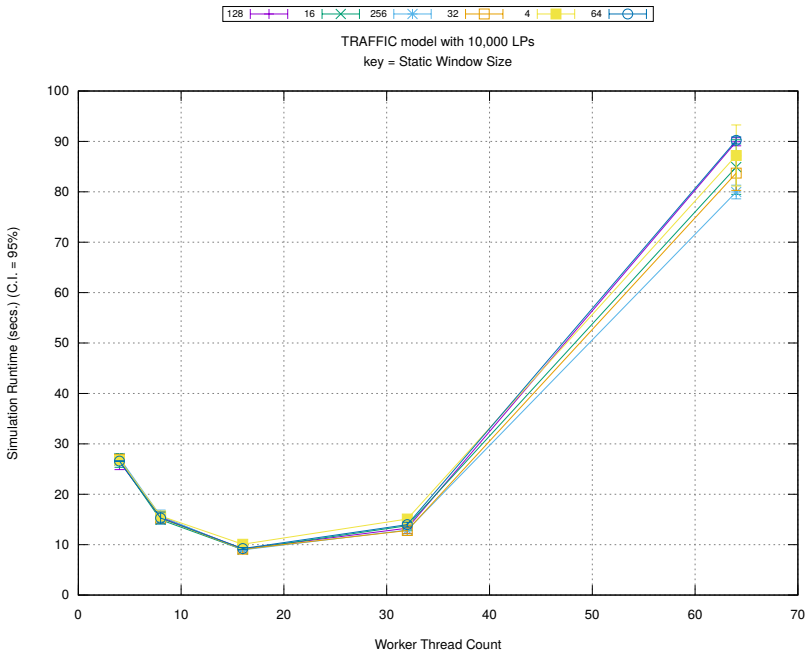


(b) Event Commitment Ratio



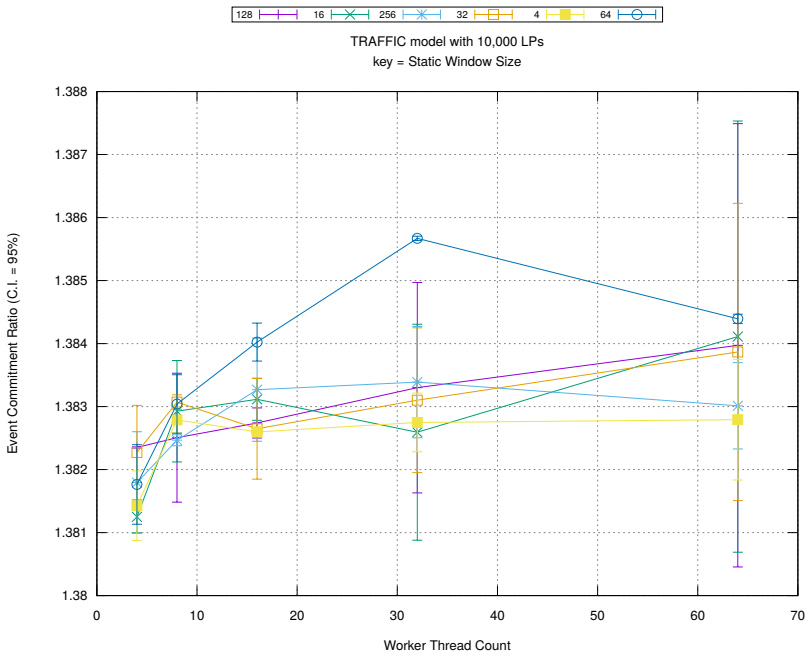
(c) Event Processing Rate (per second)

Figure 6.23: Performance of Bags with Static Window Size: Epidemic Model (*10,000* LPs, *Watts-Strogatz* diffusion network)

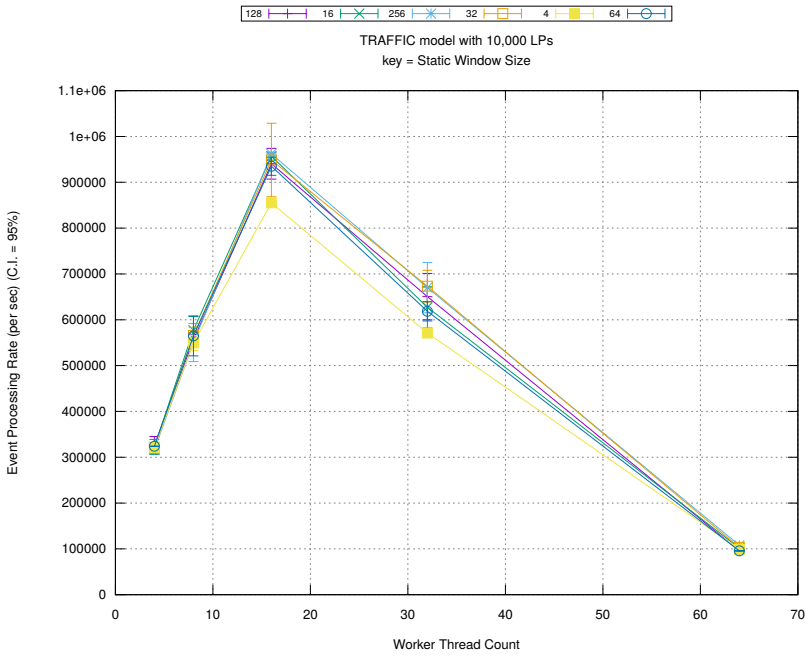


(a) Simulation Runtime (in seconds)

Figures 6.24 and 6.25 show the performance of bags when using static window size for Traffic and PCS models respectively. Similar to the behavior shown by the two different configurations of the Epidemic model, Traffic and PCS models also have rollback count that remains invariant with increase in number of worker threads. Peak performance is for 16 threads. The static window size parameter has very little or no impact on the simulation.

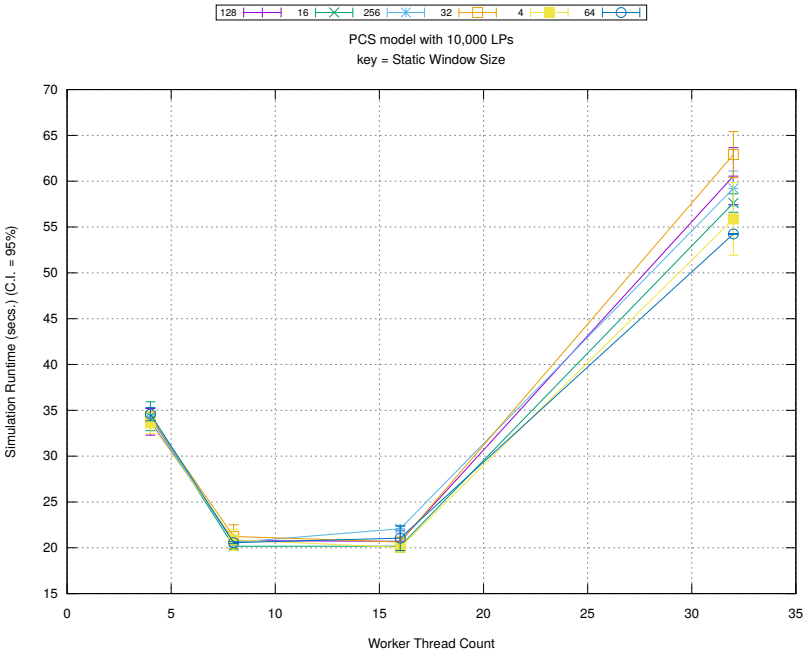


(b) Event Commitment Ratio

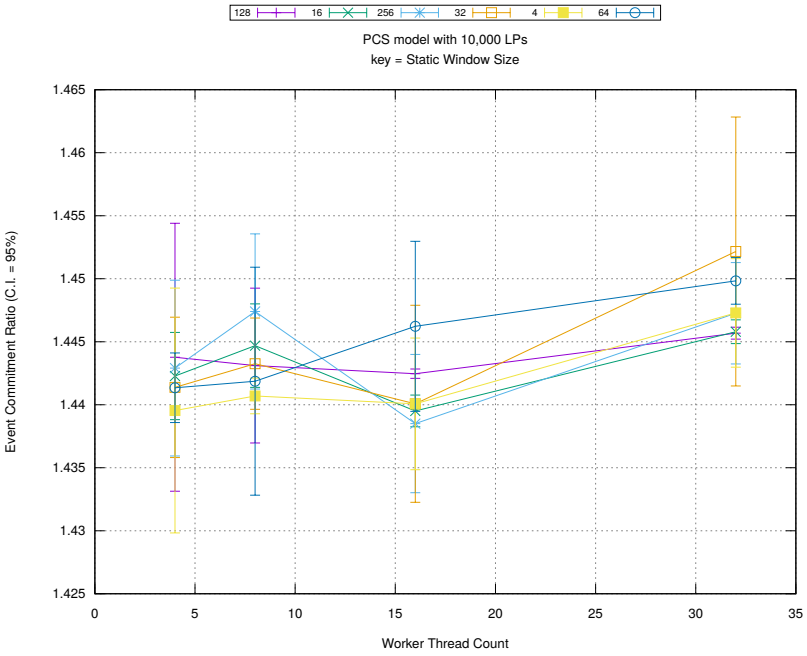


(c) Event Processing Rate (per second)

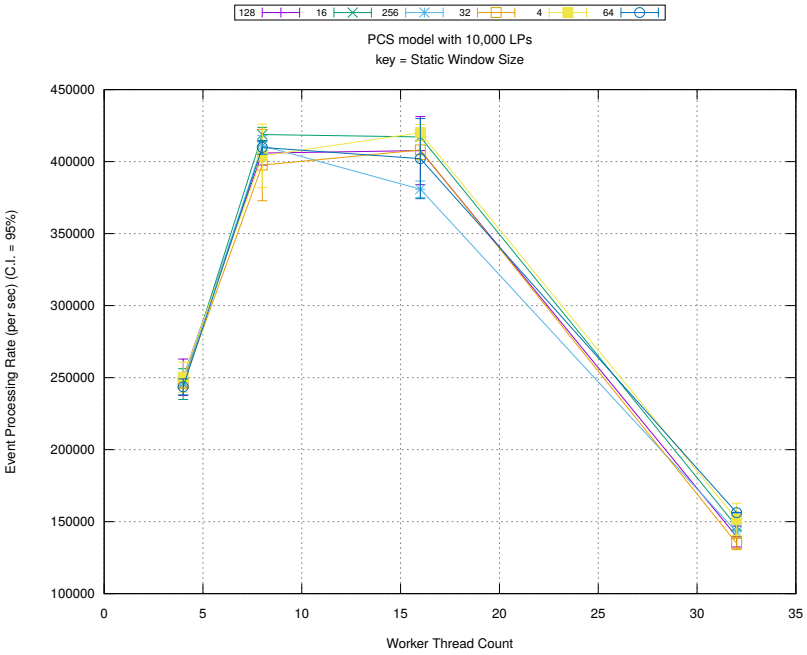
Figure 6.24: Performance of Bags with Static Window Size: Traffic Model (*10,000* LPs)



(a) Simulation Runtime (in seconds)

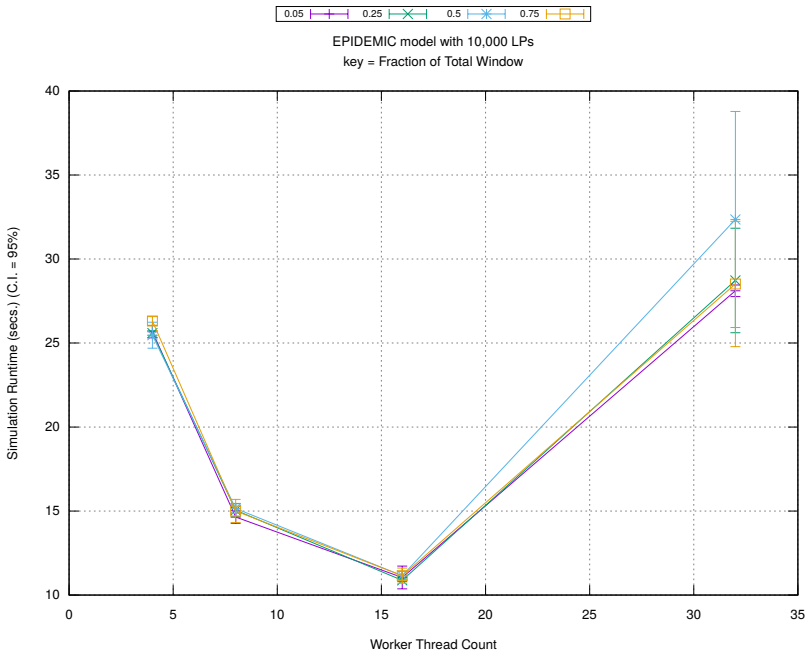


(b) Event Commitment Ratio



(c) Event Processing Rate (per second)

Figure 6.25: Performance of Bags with Static Window Size: PCS Model (*10,000* LPs)

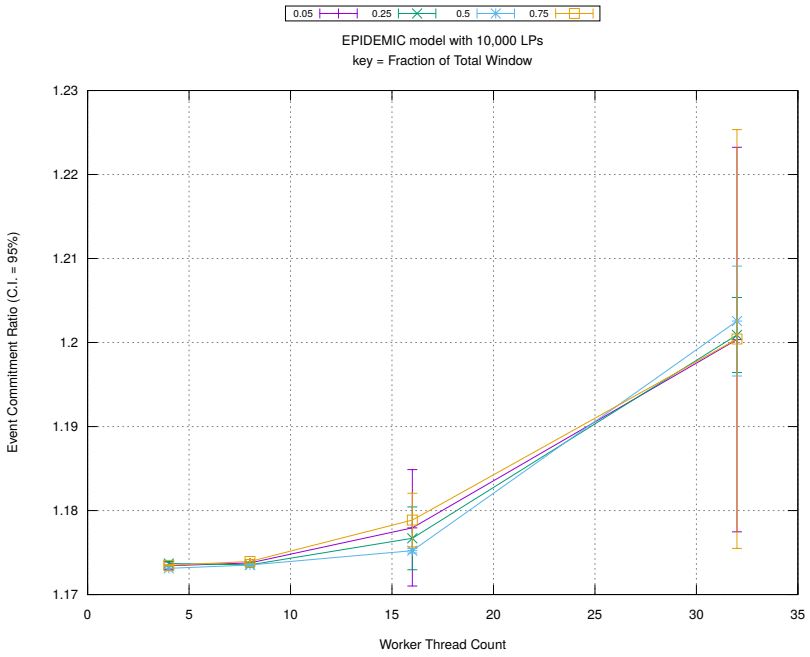


(a) Simulation Runtime (in seconds)

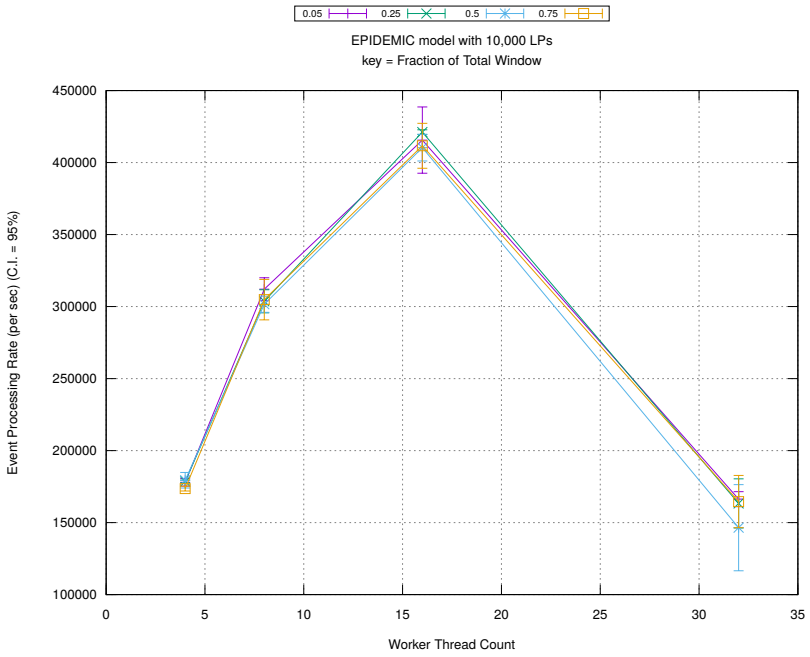
6.4.6 Bags with Fractional Time Window

Similar to Section 6.4.5, the performance of bags and its fractional window selection parameter are being quantitatively analyzed in this study. The performance of WARPED-2 is being evaluated for different thread counts (4, 8, 16, 32 and 64) when using bags with fractional window. The fractional parametric values evaluated are 0.05, 0.25, 0.50 and 0.75.

Figures 6.26 and 6.27 show the performance of Epidemic Model where events are scheduled from profile-driven LP partitions (refer to Section 5.2.4 for details). Both configurations show fairly similar performance which peaks at 16 threads. From the plots, it is evident that fraction of time window has little or no impact. Similar to event chains and bags using static window size, the rollback count remains fairly invariant to increase in number of worker threads and fraction of time window. Epidemic model using Watts-Strogatz diffusion network has lesser number of rollbacks compared to the Epidemic model using Barabasi-Albert diffusion network.

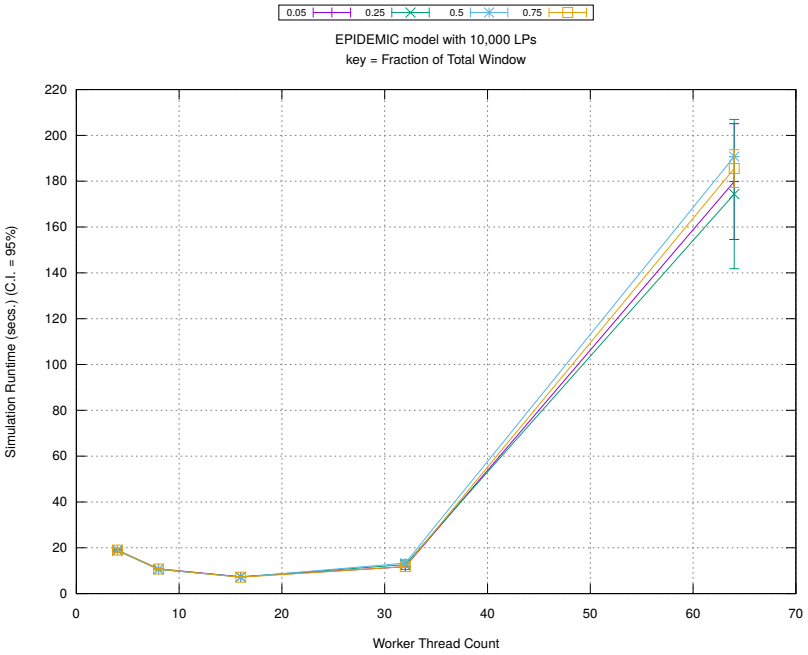


(b) Event Commitment Ratio

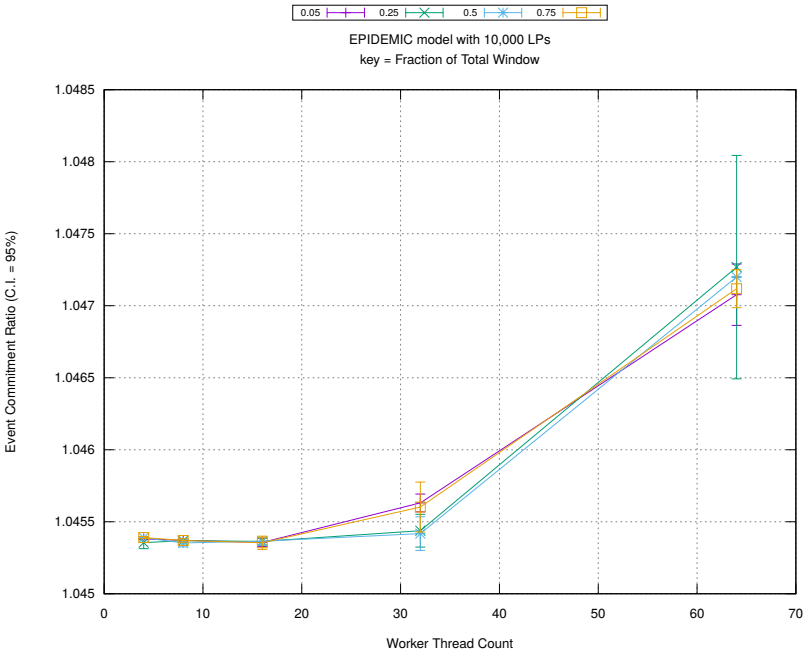


(c) Event Processing Rate (per second)

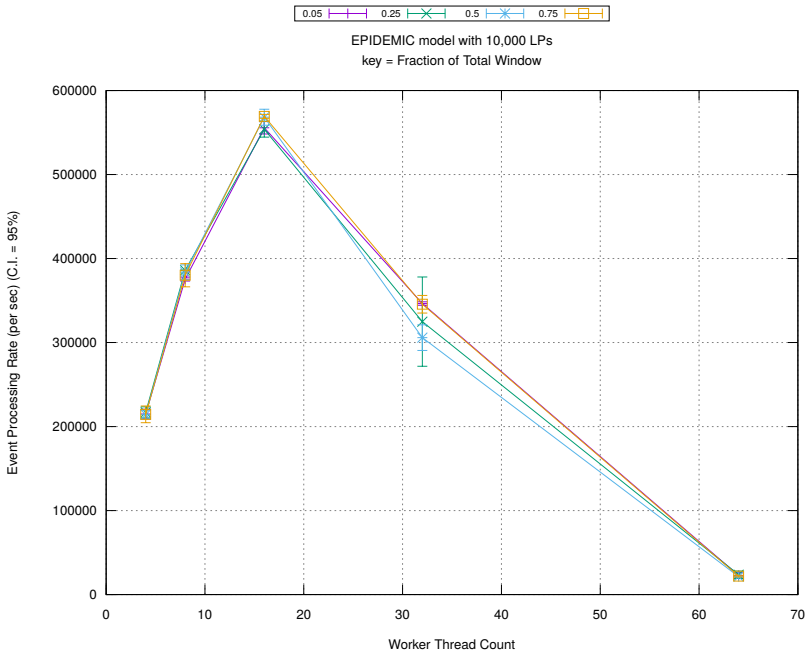
Figure 6.26: Performance of Bags with Fractional Time Window: Epidemic Model (*10,000* LPs, *Barabsi-Albert* diffusion network)



(a) Simulation Runtime (in seconds)

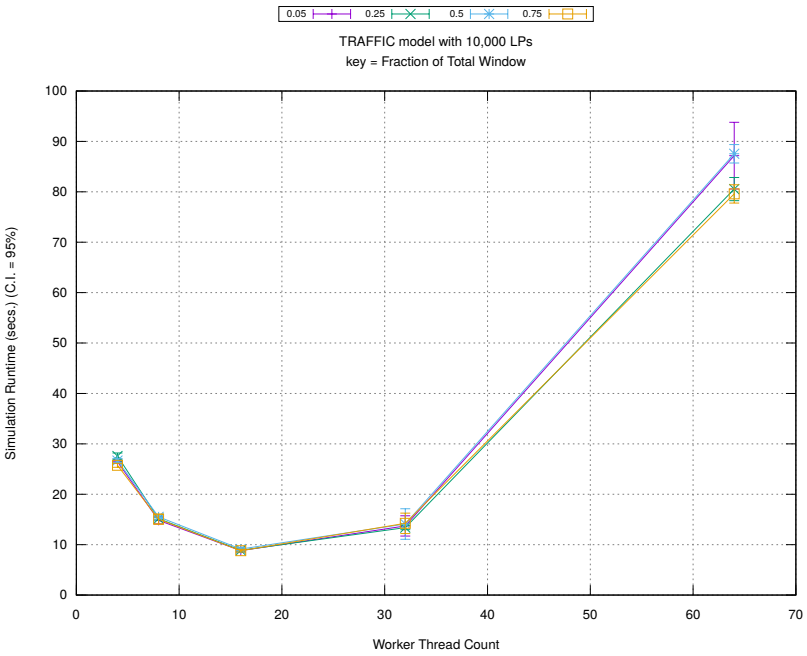


(b) Event Commitment Ratio



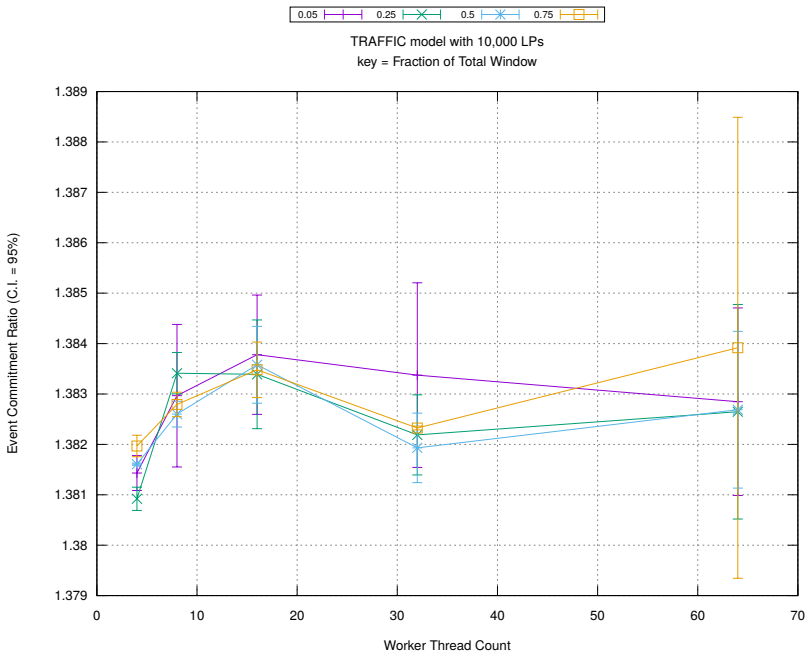
(c) Event Processing Rate (per second)

Figure 6.27: Performance of Bags with Fractional Time Window: Epidemic Model (*10,000* LPs, *Watts-Strogatz* diffusion network)

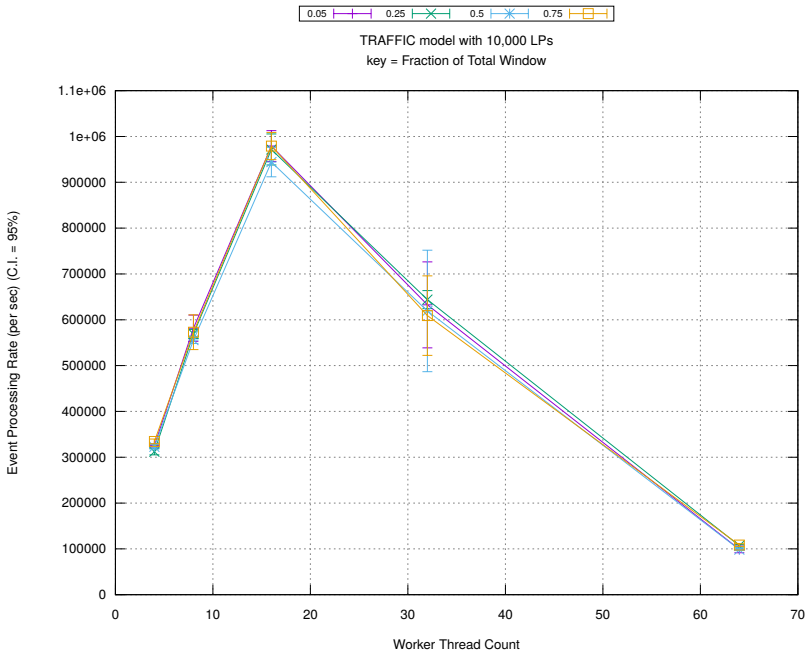


(a) Simulation Runtime (in seconds)

Figures 6.28 and 6.29 show the performance of bags when using fractional time window for Traffic and PCS models respectively. Similar to the behavior shown by the two different configurations of the Epidemic model, Traffic and PCS models also have rollback count that remains invariant with increase in number of worker threads. Peak performance is for 16 threads. The fractional time window parameter has very little or no impact on the simulation.

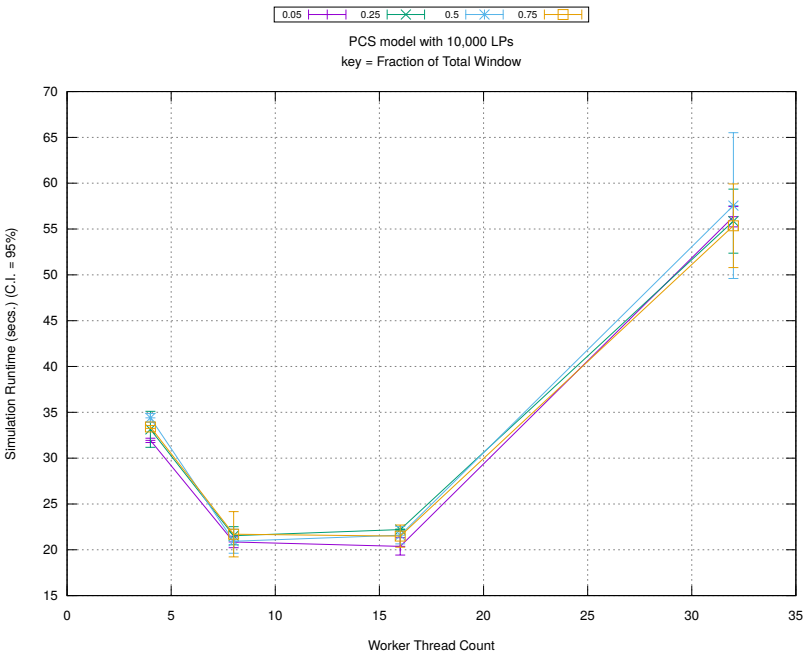


(b) Event Commitment Ratio

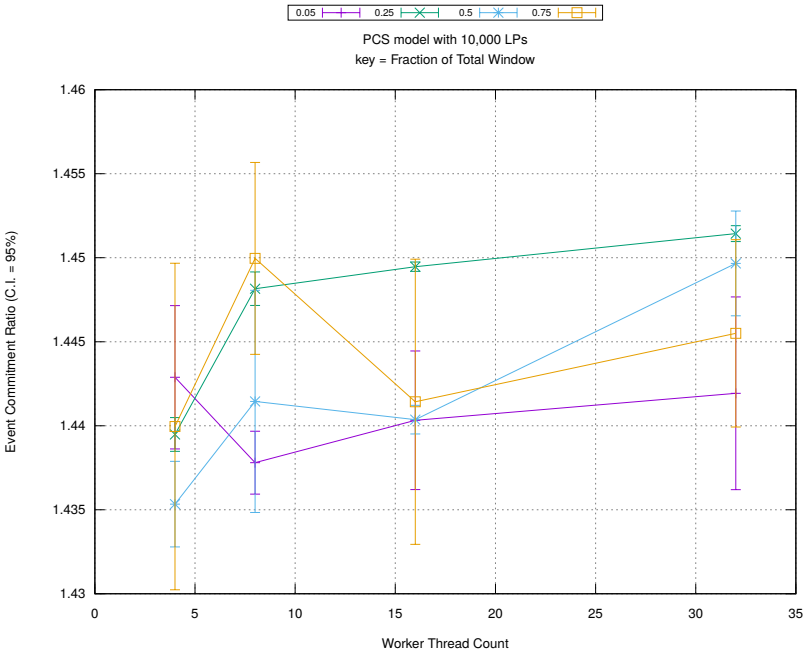


(c) Event Processing Rate (per second)

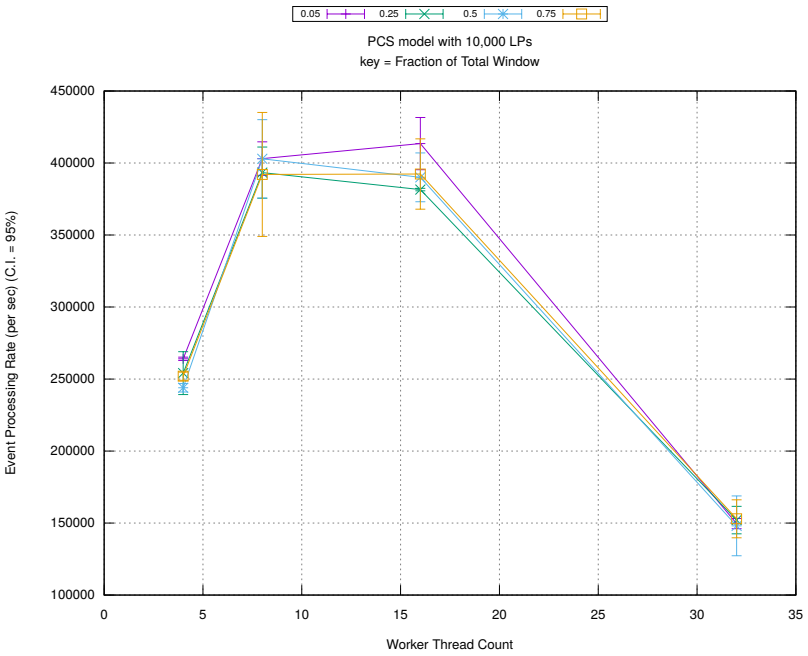
Figure 6.28: Performance of Bags with Fractional Time Window: Traffic Model (*10,000* LPs)



(a) Simulation Runtime (in seconds)



(b) Event Commitment Ratio



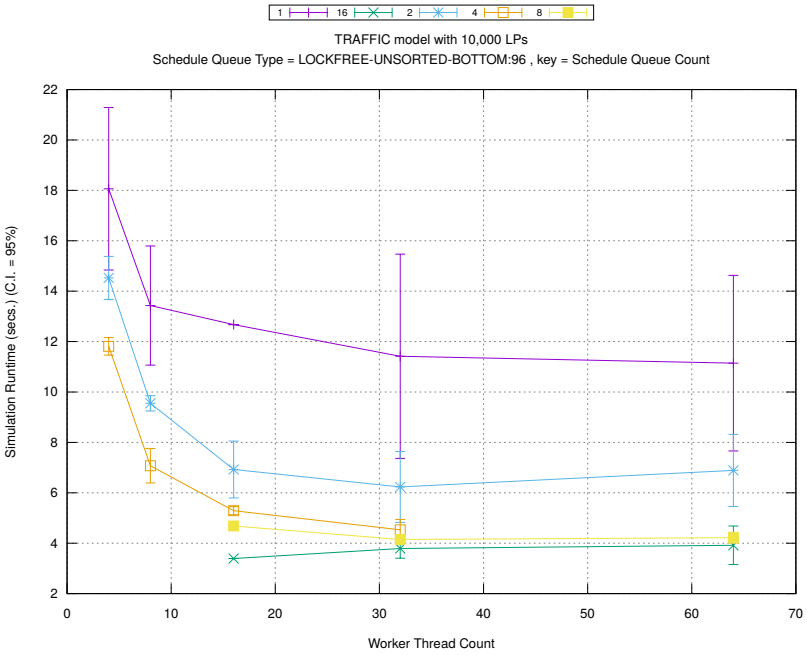
(c) Event Processing Rate (per second)

Figure 6.29: Performance of Bags with Fractional Time Window: PCS Model (*10,000* LPs)

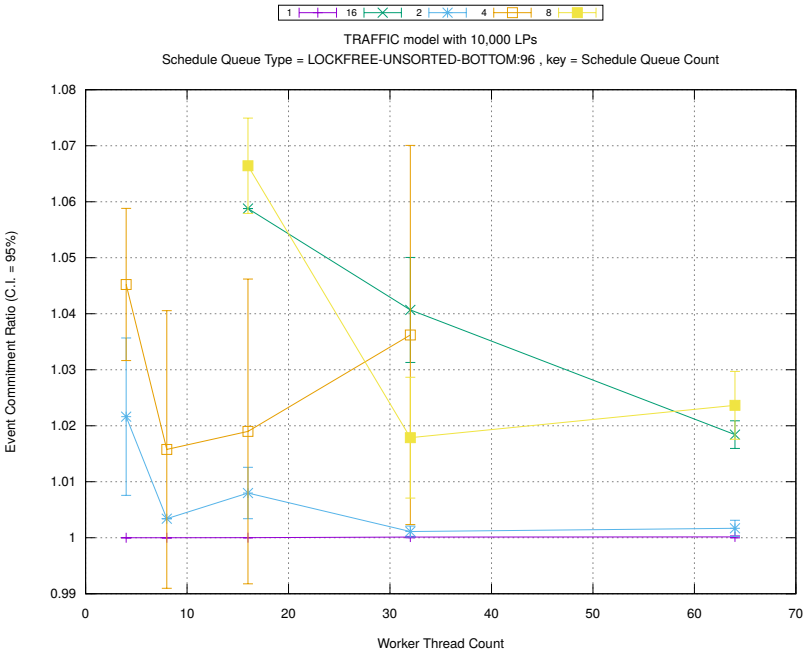
6.5 Summary of Performance Analysis

Based on the data presented in Section 6.4, it can be observed that different models react differently to the various options for schedule queue data structures and scheduling mechanisms. In spite of these subtle differences in behavior, a common pattern does emerge. Ladder Queue with lock-free access to unsorted bottom is generally the most efficient candidate for the schedule queue. Its event processing rate is only eclipsed by multiple schedule queues. A possible configuration, therefore, can be multiple schedule queues with each schedule queue being a Ladder Queue with Lockfree Unsorted Bottom. Figures 6.30 and 6.31 present the performance of the aforementioned configuration for two separate models. The network structure and event density of these two models, namely Epidemic model with Watts-Strogatz network and Traffic model, are different which should aid in model-specific comparisons. Since the threshold parameter in a Ladder Queue has only negligible effect on simulation performance, threshold value of 96 is the only one used for this presentation. Plots for multiple Ladder Queue-based schedule queues using Lockfree Unsorted Bottom and other threshold values can be studied in the Appendix.

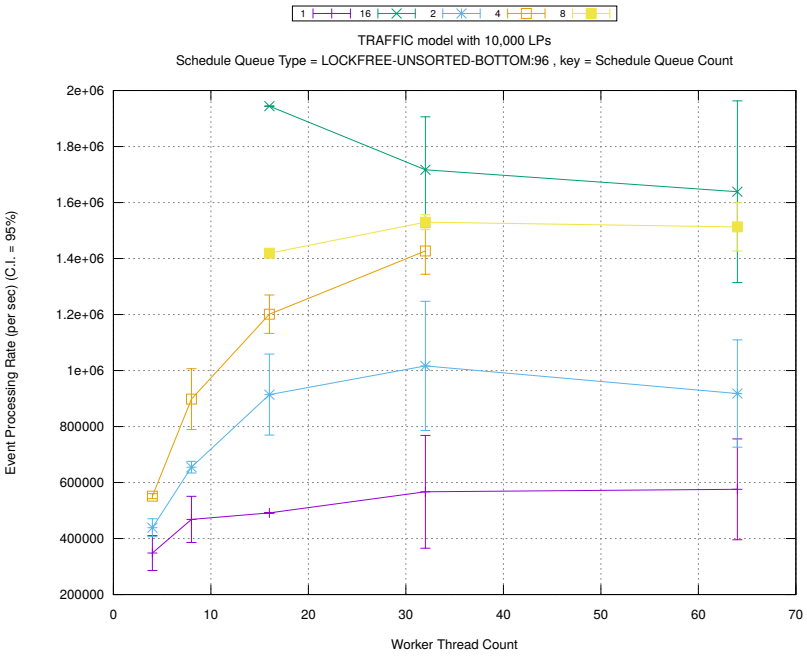
Figure 6.30 shows gain in performance for the traffic model when compared against Figure 6.12 (multi-



(a) Simulation Runtime (in seconds)

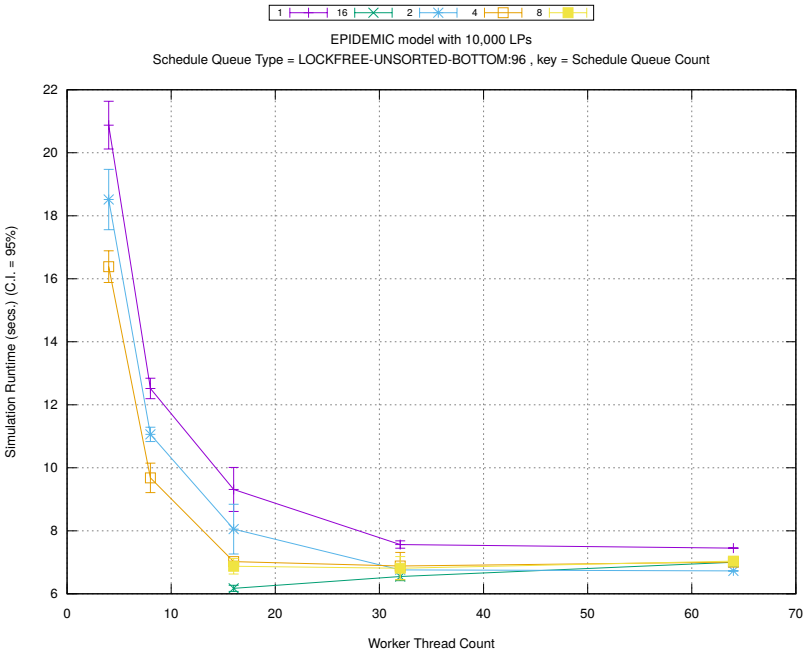


(b) Event Commitment Ratio

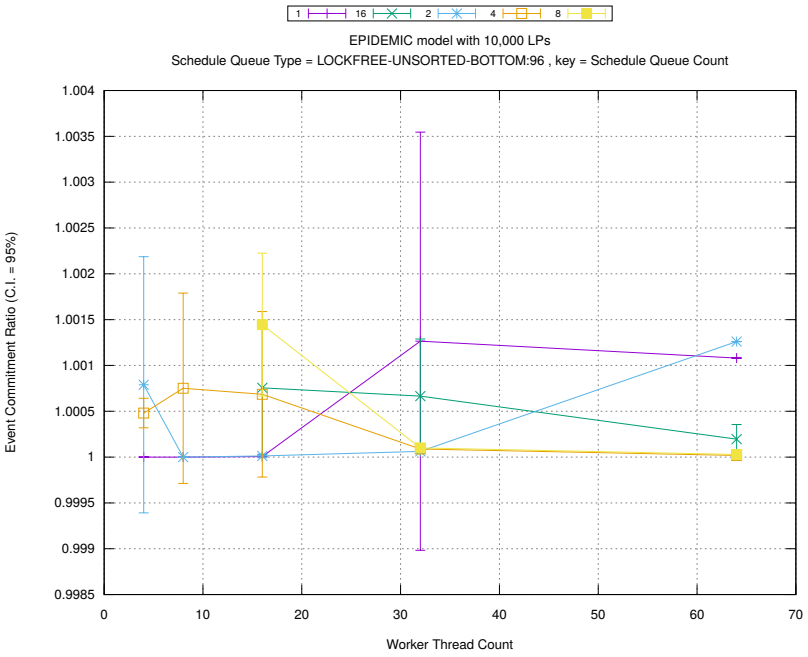


(c) Event Processing Rate (per second)

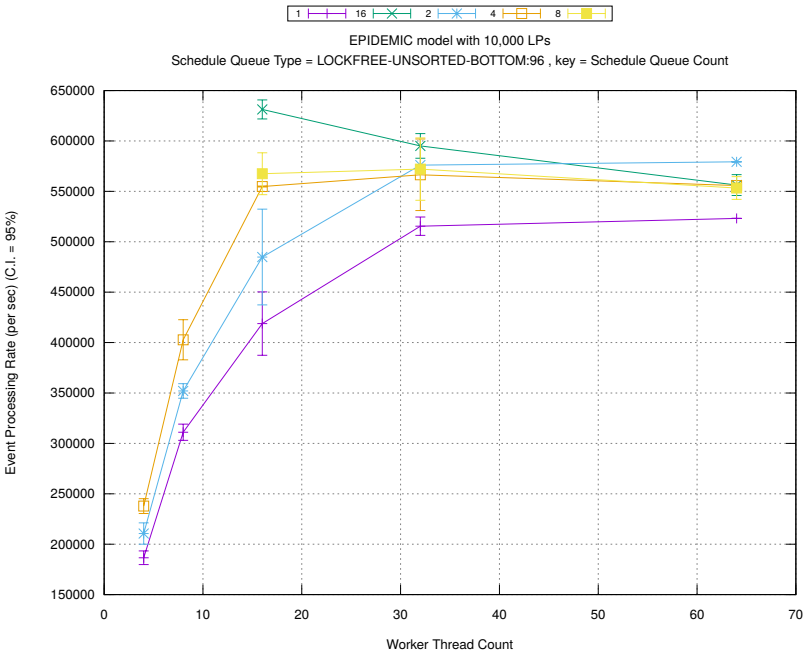
Figure 6.30: Traffic for Multiple Ladder Queue-based Schedule Queues with Lockfree Unsorted Bottom (*10,000* LPs)



(a) Simulation Runtime (in seconds)



(b) Event Commitment Ratio



(c) Event Processing Rate (per second)

Figure 6.31: Epidemic (Watts-Strogatz network) for Multiple Ladder Queue-based Schedule Queues with Lockfree Unsorted Bottom (*10,000* LPs)

ple STL MultiSet-based schedule queues) or Figure 6.8 (single schedule queue with different data structure options). The gain is substantial when event processing rate is compared but marginal when simulation times are compared.

Figure 6.31 shows gain in performance for the epidemic model when compared against Figure 6.11 (multiple STL MultiSet-based schedule queues) or Figure 6.7 (single schedule queue with different data structure options). The gain for event processing rate is marginal by comparison. Epidemic model has sparse event density when compared to traffic model. This explains the increase in event processing rate for traffic. However, this higher event density also increases the risk of causality violation in traffic. Infact there is a 5-10% increase in rollbacks for traffic while epidemic's rollback count increases by less than 1%. As a result, any performance gains in event processing rate for traffic is mostly cancelled by the increased number of rollbacks it needs to take care of.

Chapter 7

Performance Comparison to Sequential Simulation

Speedup with respect to Sequential Simulation, for a WARPED-2 configuration, is a metric used in this chapter to present a consolidated overview of the most effective scheduling data structures and scheduling strategies found during the quantitative study presented in Chapter 6 and Appendix. The metric equals

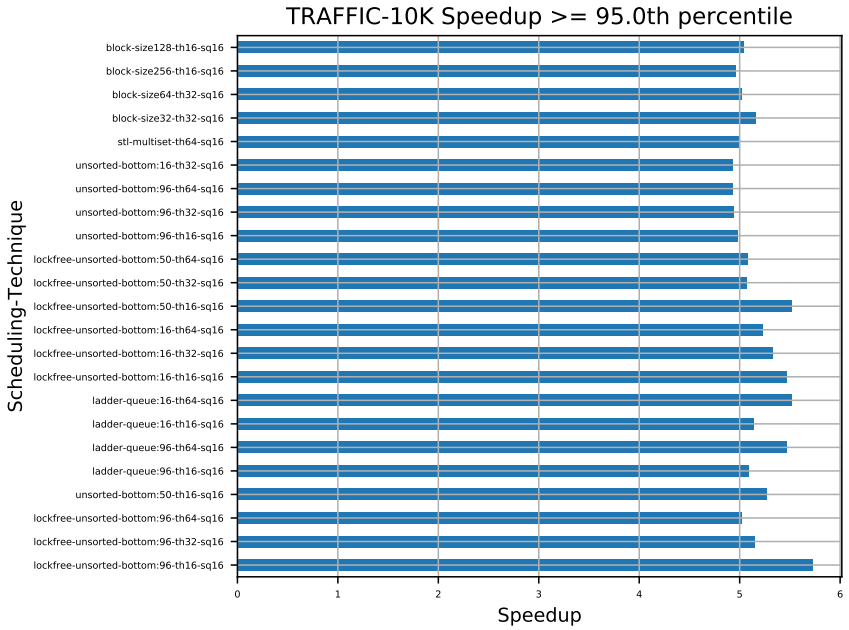
$$\frac{\text{Time needed by Sequential Simulation to reach a certain timestamp}}{\text{Time needed by that Time Warp configuration to reach the same timestamp (or GVT)}}$$

According to [14], this speedup value represents the 'supercritical speedup' of any optimistically-synchronized parallel simulation such as WARPED-2.

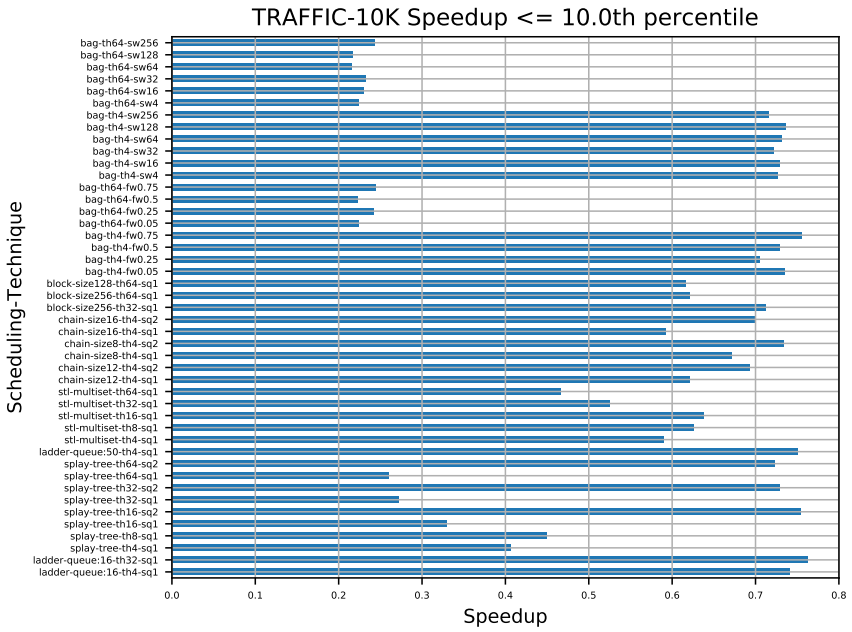
7.1 Quantitative Analysis of Configurations

7.1.1 Top Performers

Figures 7.1(a) and 7.2(a) show that Ladder Queue with Lock-free Unsorted Bottom is a top performing configuration for the two Traffic models. The choice of schedule queue count varies for the two models: 16 schedule queues is a top performer for the 10,000 LPs Traffic model while 4 or 8 schedule queues work for the 1,000,000 LPs Traffic model. Speedup against sequential simulation for both Traffic models are similar across a range of scheduling techniques. The difference in schedule queue count is most likely due to higher

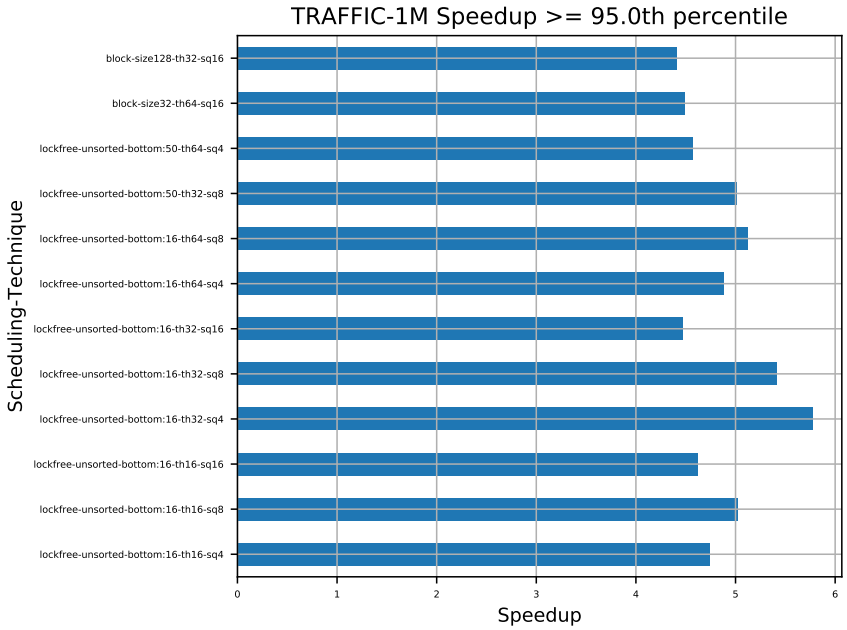


(a) Top Performers

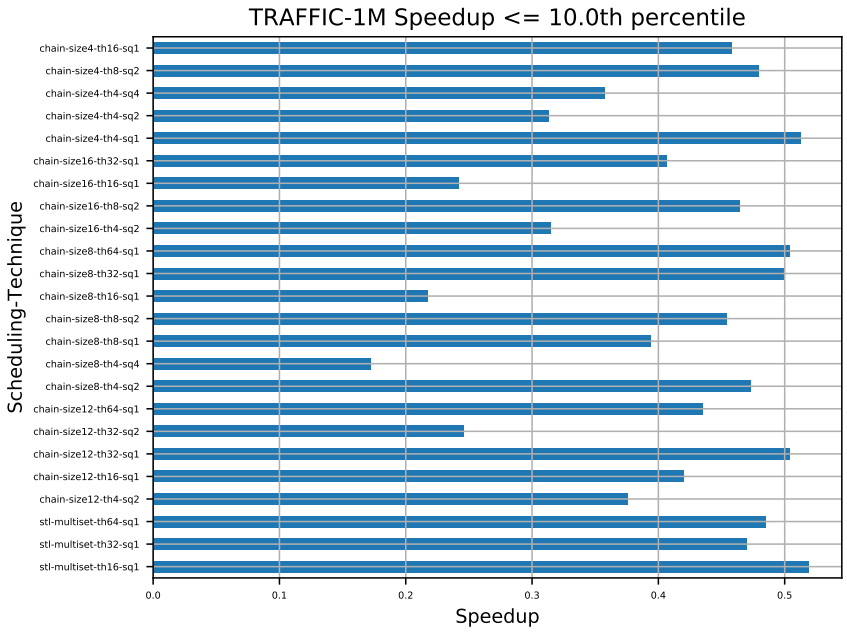


(b) Worst Performers

Figure 7.1: Top performers for Traffic (*10,000* LPs) model

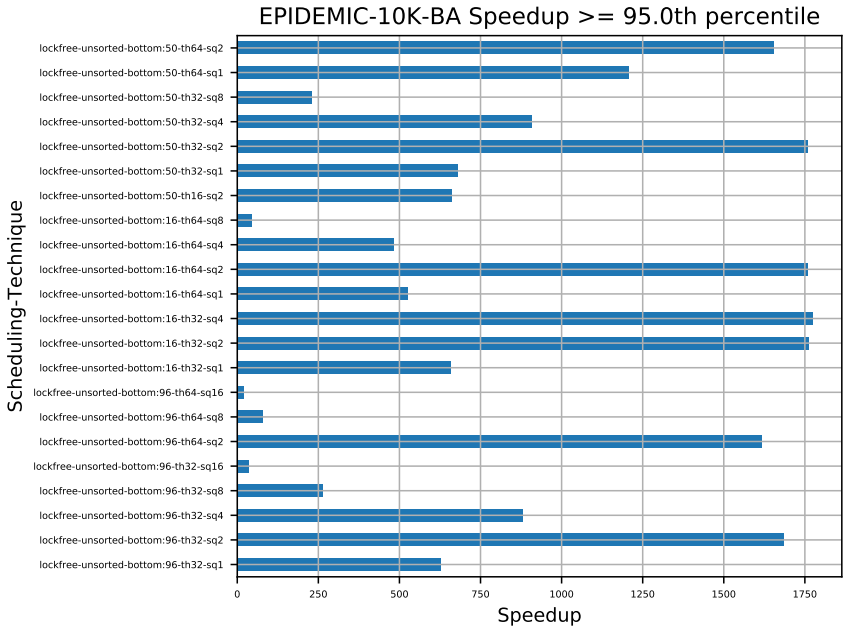


(a) Top Performers

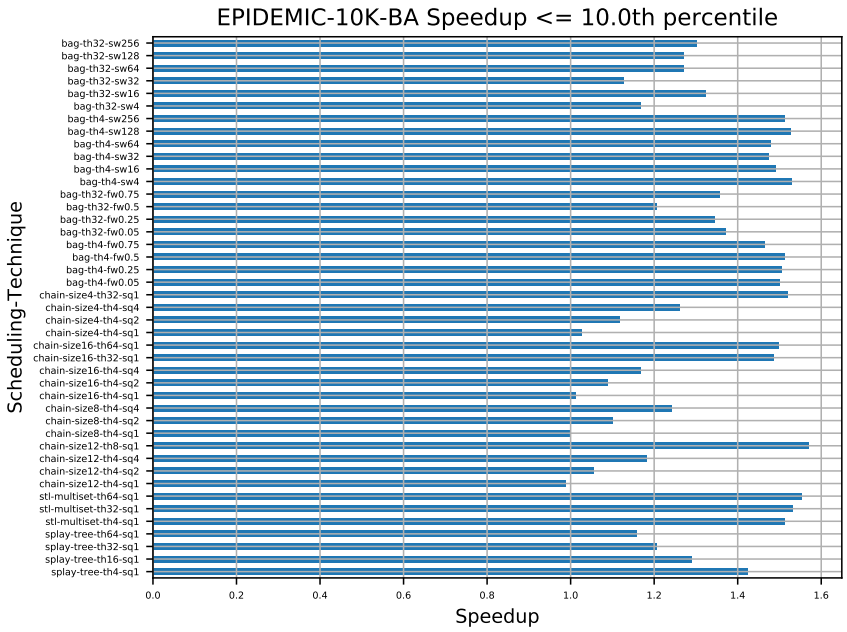


(b) Worst Performers

Figure 7.2: Top performers for Traffic (*1,048,576* LPs) model

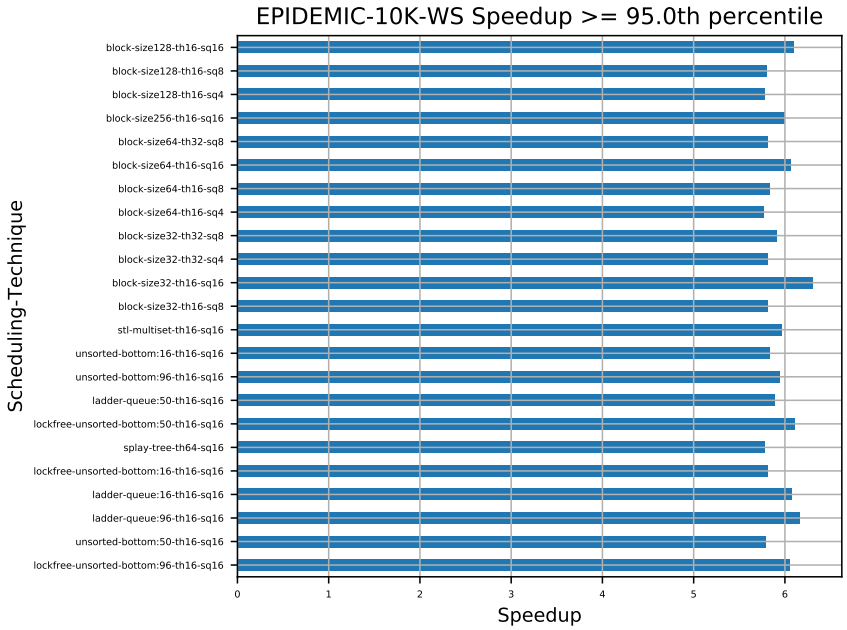


(a) Top Performers

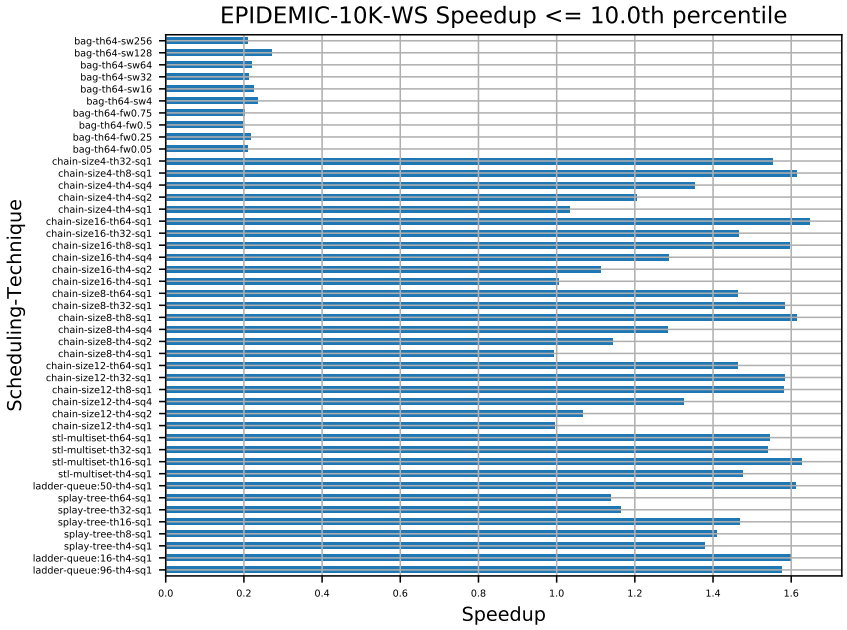


(b) Worst Performers

Figure 7.3: Top performers for Epidemic-BA (*10,000* LPs) model

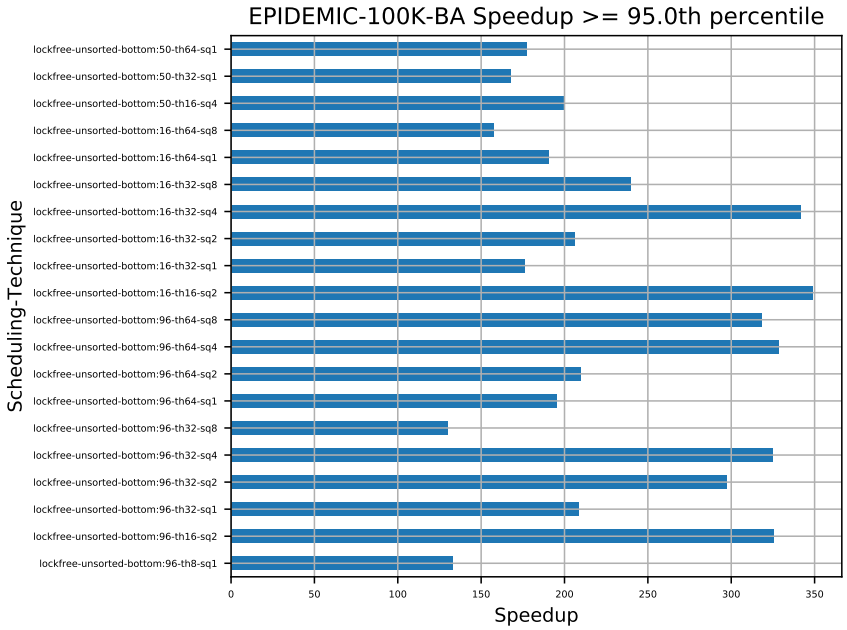


(a) Top Performers

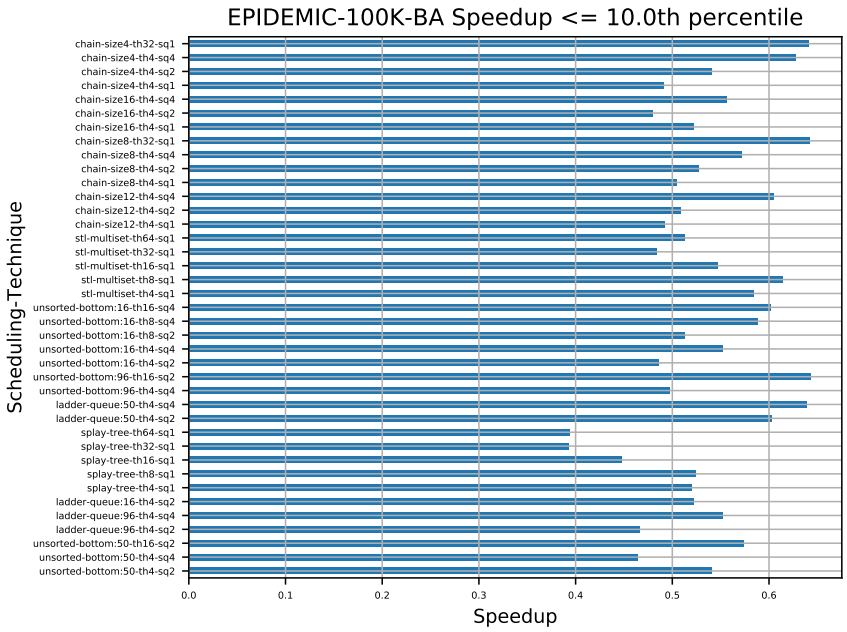


(b) Worst Performers

Figure 7.4: Top performers for Epidemic-WS (*10,000* LPs) model

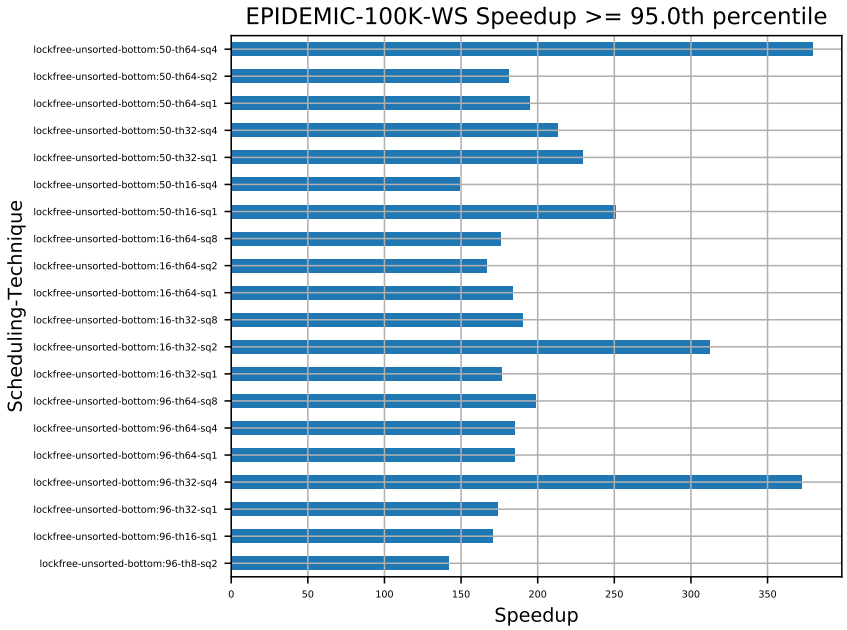


(a) Top Performers

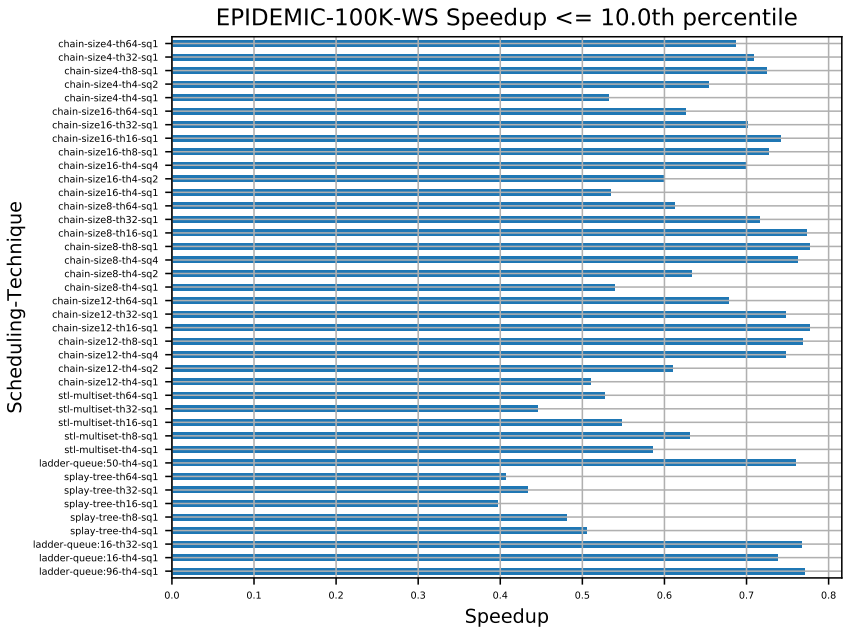


(b) Worst Performers

Figure 7.5: Top performers for Epidemic-BA (*100,000* LPs) model

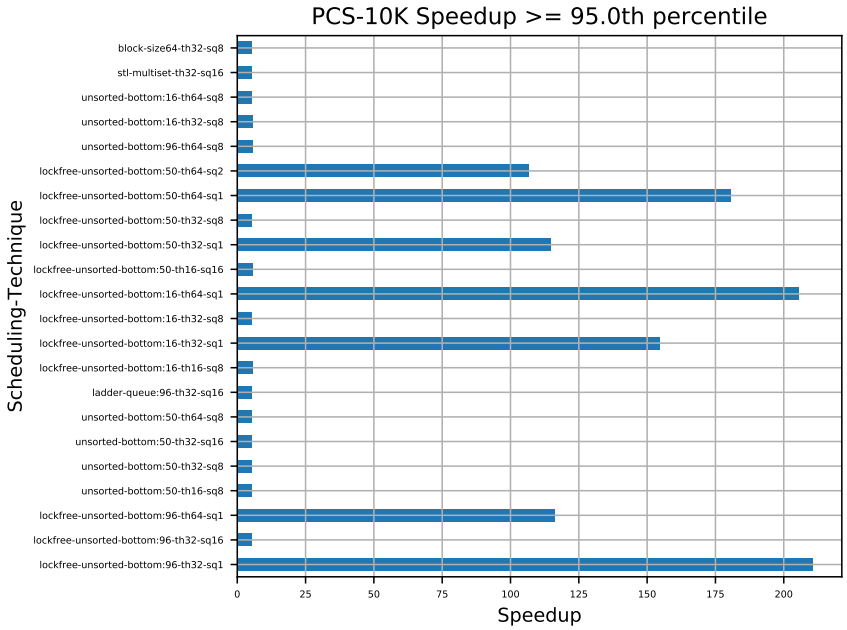


(a) Top Performers

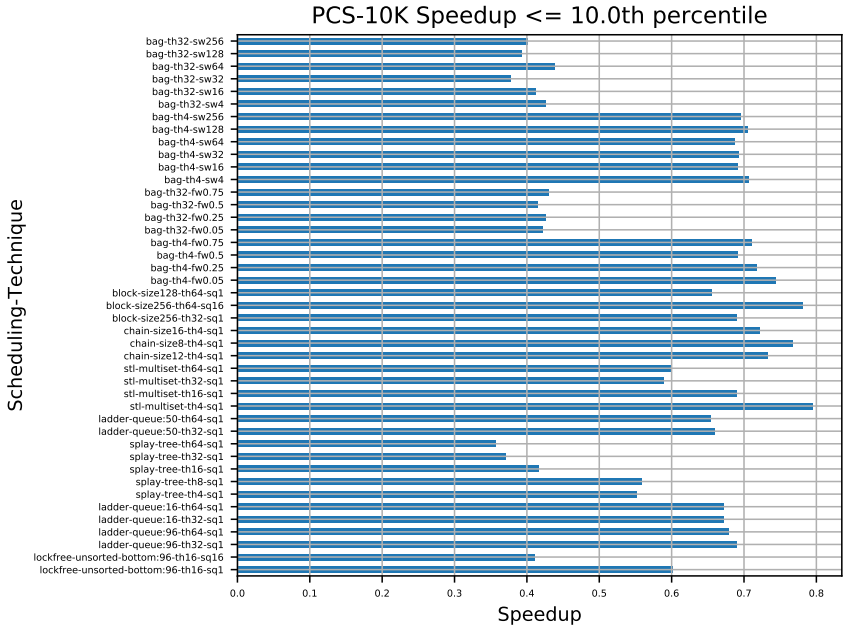


(b) Worst Performers

Figure 7.6: Top performers for Epidemic-WS (*100,000* LPs) model



(a) Top Performers



(b) Worst Performers

Figure 7.7: Top performers for PCS (*10,000* LPs) model

event processing ratio for the larger Traffic model.

Figure 7.7(a) shows Ladder Queue with Lock-free Unsorted Bottom is the clear choice for the PCS model. PCS seems to prefer one schedule queue most likely due to high causal event density. This observation is significant because although both Traffic and PCS models use mesh network, their performance window do not correlate so closely.

In case of the two Epidemic models with 10,000 LPs, the performance pattern looks strikingly different when Epidemic with Watts-Strogatz network is compared to Epidemic with Barabasi-Albert network. Figure 7.4(a) shows block scheduling is quite effective for Watts-Strogatz network but Figure 7.3(a) shows Ladder Queue with Lock-free Unsorted Bottom dominates for Barabasi-Albert network. The change in network structure seems to drastically affect the density of causally-linked events. It can also be observed that it is easier to pinpoint a good scheduling technique for Barabasi-Albert based Epidemic model when compared to Watts-strogatz based model. In case of the latter model, many scheduling configurations show good speedup.

Figures 7.5(a) and 7.6(a) show that, for the Epidemic models with 100,000 LPs, Ladder Queue with Lock-free Unsorted Bottom is the clear choice. Unlike Epidemic models with 10,000 LPs, both models here seem to prefer low number of schedule queues (2, 4 and sometimes 8). The smaller 10,000 LPs Epidemic models prefer 8 or 16 schedule queues. In general, a wide variety of scheduling techniques work for the Watts-Strogatz network while the preference is less wide for Barabasi-Albert based Epidemic simulation. Additionally, Barabasi-Albert based Epidemic model prefers lower number of schedule queues. This is indicative of higher density of causally linked events in case of the BA model. It explains why a wide variety of scheduling techniques work for the WS model while the selection window is smaller for the BA model.

Summary

The overwhelming choice for most models is Ladder Queue wih Lock-free Unsorted Bottom. Larger models prefer lower schedule queue count (2,4 or maybe even 8) while smaller models generally work better with higher number of schedule queues (8 or 16). Bags and chains are generally absent from the 95th percentile group for all models. In few cases like Traffic model, Block scheduling is a powerful option as well.

Additionally, the underlying network seems to play a key role in determining which scheduling techniques are effective for a model, but PCS and Traffic defy this trend.

7.1.2 Worst Performers

Figures 7.1(b), 7.2(b), 7.7(b), 7.5(b) and 7.6(b) show that scheduling techniques in the lowest 10th percentile are all slower than sequential simulation, except for Epidemic models with 10,000 LPs (Figures 7.3(b) and 7.4(b)). Most scheduling techniques (type of schedule queue, blocks and chains) use one schedule queue for a wide variety of worker threads. It is expected that contention will slow down simulation but it is surprising that the speedup is less than 1.

Chain and Bag scheduling techniques are some of the worst performers for the Epidemic models. The performance of Chain scheduling suffers more than Block scheduling when LP count increases for the Traffic model. Both Chain and Block scheduling techniques perform very poorly for the 10,000 LP sized Traffic model but performance of Block scheduling improves when LP count increases. Usually Block scheduling techniques perform poorly for larger block sizes of 64, 128 or 256.

These observations for Chain, Bag and Block scheduling indicates that over-aggressive estimates of causal independence are ineffective. The simulation models which suffer most from these over-aggressive thresholds are Traffic and PCS - both have a higher density of causally-linked events when compared to the Epidemic models.

Summary

The lowest 10th percentile results are good for understanding the effects of contention. Since most of the scheduling techniques here use one schedule queue, we can assume load imbalance is not an issue for the simulation models. This means only model-specific causal density and contention can affect performance of different scheduling techniques. Epidemic models are of special interest here because these models have the lowest density of causally-linked events. So contention for shared pending event set is its primary hindrance for the Epidemic models. Based on observations from Figures 7.3(b) and 7.4(b), Epidemic models with 10,000 LPs suffer less than other models and shows speedup > 1. This observation indicates that causal violations due to over-aggressive thresholds adversely affect performance equally or even more than

contention for the shared event pool.

7.1.3 Anomalies in Performance

Epidemic model with Barabasi-Albert network and 10,000 LPs shows extremely high speedup values in Figure 7.3(a). This behavior indicates that some aspect of the processing architecture is making WARPED-2 simulations extremely efficient. Since these simulations were executed on NUMA-based computing platforms, the effect of distributed caches is worth exploring. Section 7.2 presents a small study on a SMP node which does not have such distributed cache arrangements. This study is useful for pinpointing that cache effects in NUMA machines do play a major role in artificially boosting the performance of WARPED-2.

7.2 Performance of WARPED-2 on SMP machine

Figures 7.8, 7.9, 7.14, 7.12, 7.13, 7.10 and 7.11 show the speedup vs. sequential simulation for all simulation models. These simulations were executed on a SMP machine (refer to Table 7.1). All configurations in these experiments use 6 worker threads because this study only aims to capture how cache effects affect performance on a SMP machine when compared to a NUMA machine. Section 7.1.3 presents the anomalies that exist in the data collected on the NUMA-based machines.

		Intel® Xeon® E5675
Processor	ISA	x86_64
	# Cores	6
	# Threads	12
	# Sockets	1
	Frequency	3.06 GHz
	Cache	12 MiB
	Memory	28 GB
Runtime	OS Kernel	4.9.0-6-amd64
	C Library	Debian GLIBC 2.24-11+deb9u3
	Compiler	GCC v6.3.0
	MPI	MPICH v3.2
	Python	2.7.13
	Python-numpy	1.12.1
	Python-networkx	1.11

Table 7.1: Experimental Setup for SMP machine

Figure 7.8: Top performers for Traffic (*10,000* LPs) model

The following observations can be made from the data:

1. The speedup numbers from the SMP machine are low for all simulation models. This indicates that the NUMA architecture is making the performance of WARPED-2 look better in Section 7.1 than it should have been.

2. Ladder Queue with Lock-free Unsorted Bottom is generally the best option but not by a huge margin unlike what we observed from the NUMA experiments. That is partly due to use of lower number of threads (6) here. Locked queues do not suffer a lot from contention issues for lower thread count and so they tend to perform at a level comparable to Ladder Queue with Lock-free Unsorted Bottom when only 6 worker threads are used.

3. The results show that 6 threads with 6 schedule queues is generally not a good option, mostly due to load imbalance which triggers too many rollbacks. The configurations that seem to perform well are 6 threads with 2 or 3 schedule queues.

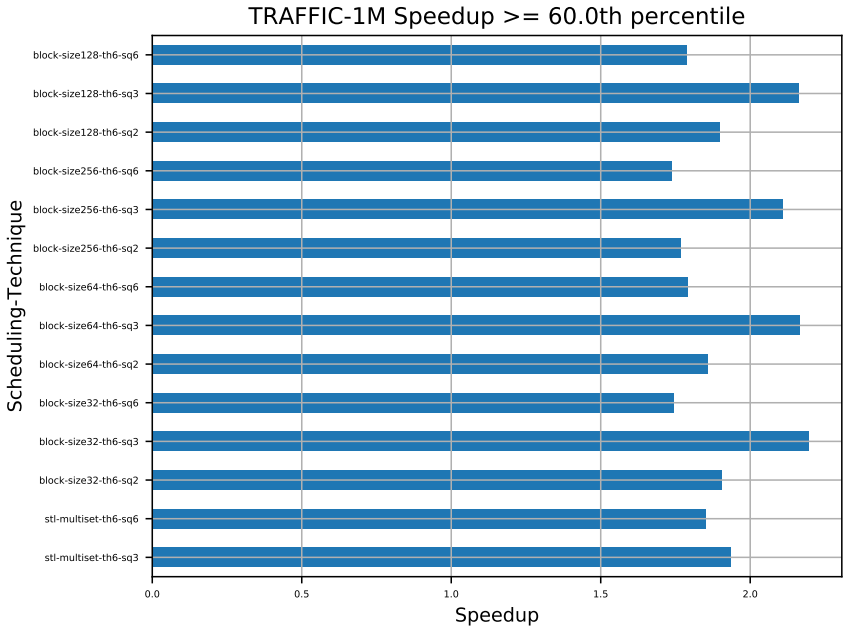


Figure 7.9: Top performers for Traffic (*1,048,576* LPs) model

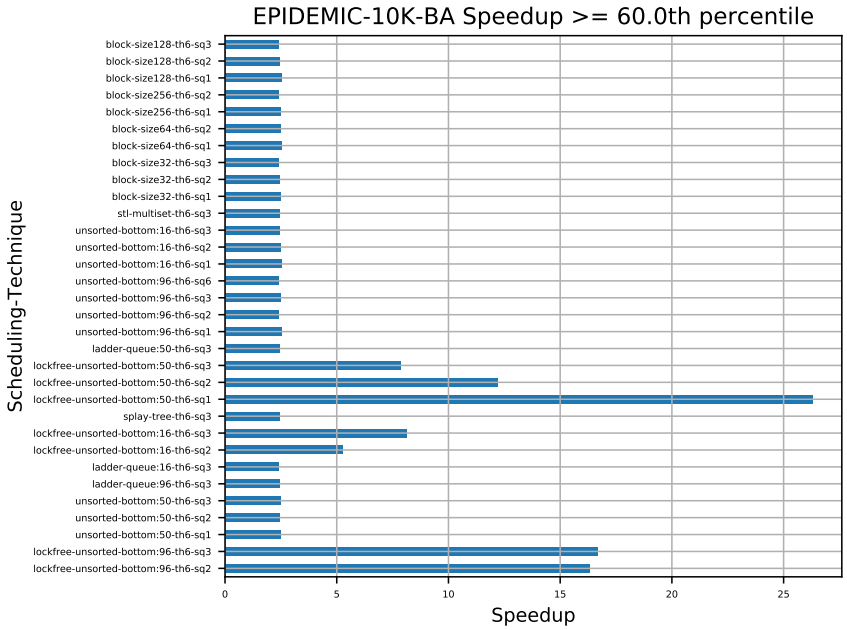


Figure 7.10: Top performers for Epidemic-BA (*10,000* LPs) model

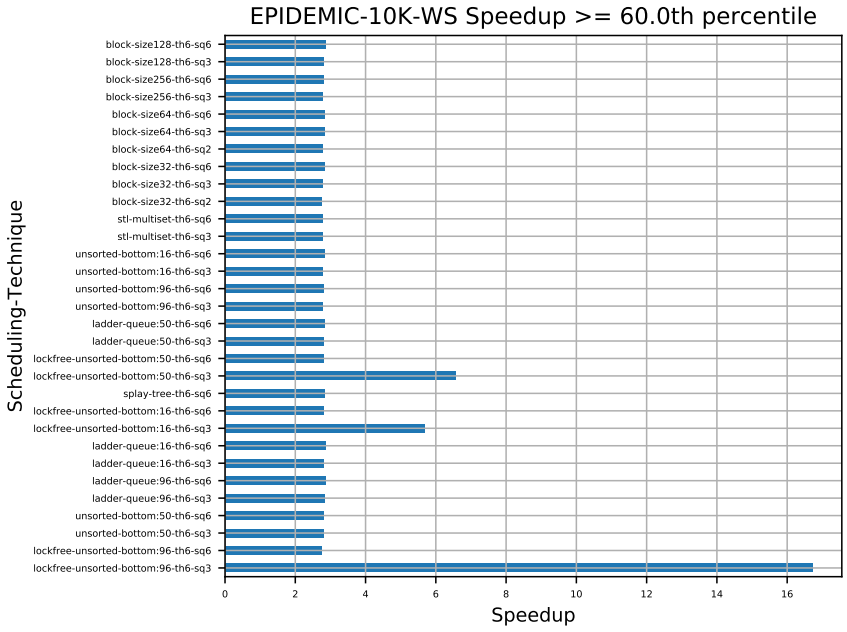


Figure 7.11: Top performers for Epidemic-WS (*10,000* LPs) model

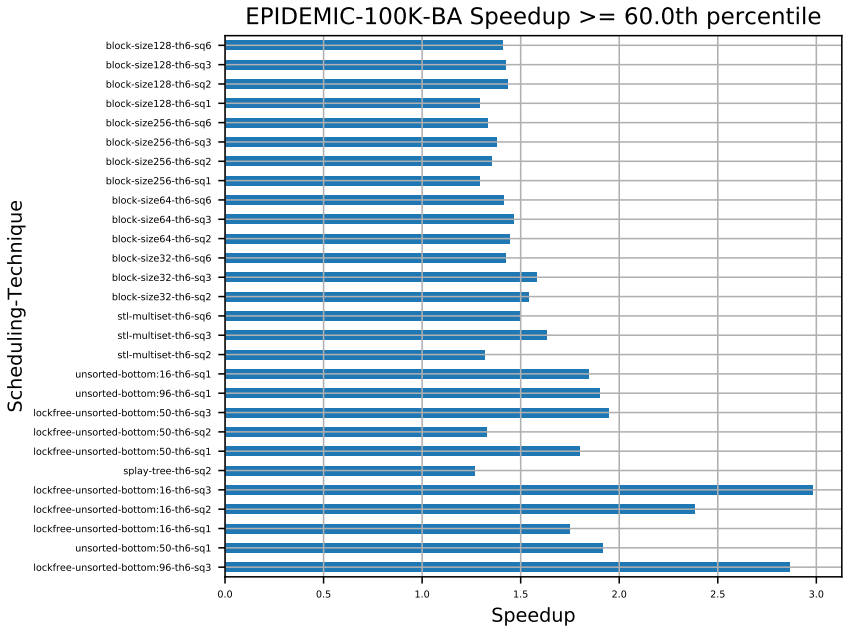


Figure 7.12: Top performers for Epidemic-BA (*100,000* LPs) model

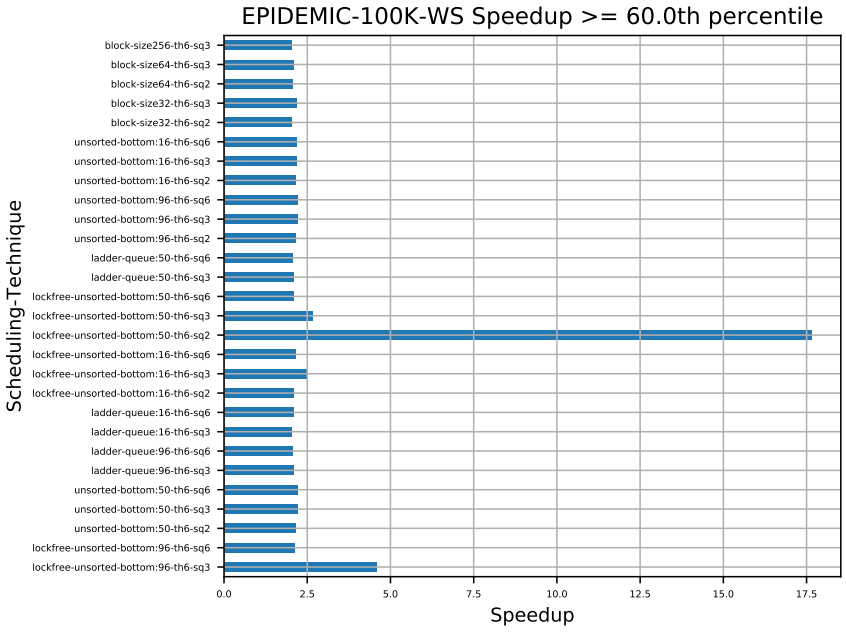


Figure 7.13: Top performers for Epidemic-WS (*100,000* LPs) model

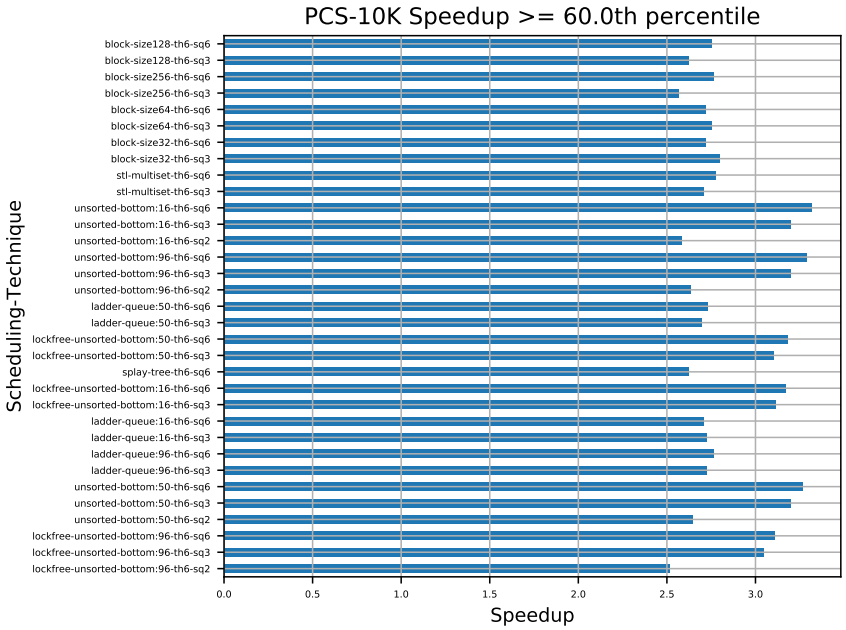


Figure 7.14: Top performers for PCS (*10,000* LPs) model

Chapter 8

Conclusions and Suggestions for Future Research

8.1 Conclusion

This dissertation studies the problem of contention that prevents Time Warp-synchronized Parallel Discrete Event Simulations from scaling up substantially on a multi-core computing platform. The data structure that holds the pending events is a major point of contention for threads trying to access and process these scheduled events in parallel. Several different data structures, namely STL MultiSet [55], Splay Tree [28] and Ladder Queue [8], have been explored and discussed in details. The Ladder Queue, a hierarchically organized priority queue, is of particular interest because of its ability to self-split pending events into time-bound partitions without requiring any manual intervention like the Calendar Queue [15]. This self-split of pending events forces the smallest available events to accumulate inside the lowest partition of the Ladder structure. If one assumes that these events within the lowest partition are causally independent, there is no further need to sort these events. This approach saves computational time otherwise wasted on sorting and also reduces contention to this shared pool by increasing availability. The contention can be further reduced through atomic read-write operations on this lowest partition. Experimental results presented in Section 6.4.1 show that atomic operations on the unsorted lowest partition in a Ladder Queue improves performance of parallel simulation by over 100% when compared to a sorted Ladder Queue.

A parallel approach to contention management lies in exploration of alternative scheduling techniques. Traditional PDES kernels maintain a common pending event pool which is accessed by threads for processing events. This organization prevents a simulation from scaling up massively on large multi-core computing platforms. Splitting this common pending event pool and distributing it uniformly among groups of threads was successfully explored by Dickman *et al* [6]. The benefit from reduction in contention compensates for the extra rollbacks due to imbalance in distributed scheduling. Experimental results presented in Section 6.4.2 show that performance of simulation can be improved by upto 150% when the pending event pool is split into several pools, each having its own group of threads.

The combination of multiple schedule queues with each schedule queue being a Ladder Queue with lockfree unsorted bottom is the best scheduling strategy for all simulation models studied in this thesis. The gain in performance from this configuration depends heavily on the event density of each model and tends to favor the models with relatively high event density. This is because this configuration improves the event processing rate by reducing contention. A models with high event density is ideally placed to benefit more from this reduction in contention. Section 6.5 presents a detailed discussion on this topic.

Wilsey's [9] analysis of profile-driven data collected from Discrete Event Simulation showed that it is possible for threads to schedule multiple events for processing without causing too many causal violations. Unlike traditional PDES where one event is scheduled at a time, scheduling multiple events at once from a pending event pool reduces the overall time a thread wastes in waiting for access to this event pool. Gupta and Wilsey [3] explored two different approaches: one where multiple events are scheduled from a common pending event pool, and another where multiple events are scheduled from the LP that holds the smallest unprocessed event. Experimental results presented in Sections 6.4.3 and 6.4.4 show that this arrangement can speed up performance by upto 100% for some simulation models. The results validate the findings in [9] that not all simulation models benefit from events processed in groups.

Alt and Wilsey [17] showed that network statistics driven partitioning of LPs into 'close-knit' communities is effective for reducing remote messaging in a distributed environment. This dissertation explores the use of modularity-based techniques [46] to partition LPs into dense communities which are sparsely connected to other communities. Section 5.2.4 explains that it is possible to process a groups (or bag) of pending events from each community in parallel without drastic increase in number of causal violations.

This arrangement extends the existing benefits of processing events in groups (as discussed in the previous paragraph). This organization also allows cyclic atomic scheduling of bags to a waiting thread. Experimental results presented in Sections 6.4.5 and 6.4.6 show that, for some models, the performance of bag scheduling technique equals that of Ladder Queue with Lock-free Unsorted Bottom.

8.2 Suggestions for Future Work

8.2.1 Hybrid Group Scheduling

Sections 5.2.2 and 5.2.3 explain how events can be scheduled for processing using chains and blocks respectively. An idea that has not been explored in this dissertation involves combining the two scheduling strategies together in order to schedule events from different LPs that are within a bounding time window. Event block can serve as the first-level filter for defining the bounding time window. Each LP within the block can then be allowed to schedule multiple events within this specified time window for processing. The assumption here would be that events within this time window are generally causally independent of each other.

8.2.2 Load Balancing on Multi-Core Processors

For the past two decades, researchers have been exploring ways to balance workload in a Parallel Discrete Event Simulation. Several approaches have been proposed. Deelman and Szymanski [75] studied dynamic load balancing for unbalanced simulation of spatially explicit problems. Their proposed min-max based approach balances the workload for a ring of processes by using an estimate for arrival time of future events. Schlagenhaft *et al* [76] proposed that clusters of LPs can be moved between processing elements. Alt and Wilsey [17] showed that LPs can be partitioned using METIS [45] and assigned to different nodes on a clusters by minimizing the number of remote events communicated over the network.

The world of parallel computing has evolved significantly in the past two decades. The advent of multi-core processors has made it necessary to balance the workload distributed among threads within the same node. Dickman *et al* [6] studied the problem of intra-node load balancing for multiple schedule queues. They proposed that LPs with the smallest events can migrate within a ring of schedule queues in order

to keep the simulation balanced and closer to the critical path. While this approach works well for some simulation models, the design is sensitive to cache architecture of the host processor. The performance of such an arrangement is significantly poor on NUMA-based systems. Linden *et al* [77] describes a similar load migration protocol for fine-grained dynamic load balancing, albeit one where LPs are visualized as voxels.

In WARPED2, the pending events are partitioned and processed by several threads in parallel. While the event processing rate for each thread is approximately equal, the rate at which events are committed varies for each thread. Regulating the event processing rate across a group of threads may help keep the simulation closer to the critical path. This should reduce the number of rollbacks and result in an overall improvement in simulation time. Further research on this topic is required before arriving at a possible solution. An event profiler for Time Warp-based PDES may prove to be a valuable tool for this study.

Bibliography

[1] R. M. Fujimoto, *Parallel and Distributed Simulation Systems*. Wiley Interscience, Jan. 2000. i, 2, 7, 8

[2] D. Jefferson, "Virtual time," *ACM Transactions on Programming Languages and Systems*, vol. 7, pp. 405–425, July 1985. i, 2, 8, 9, 10, 29, 32

[3] S. Gupta and P. A. Wilsey, "Lock-free pending event set management in time warp," in *Proceedings of the 2Nd ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*, SIGSIM PADS '14, (New York, NY, USA), pp. 15–26, ACM, 2014. i, 4, 42, 45, 147

[4] J. Hay and P. A. Wilsey, "Experiments with hardware-based transactional memory in parallel simulation," in *Proceedings of the 3rd ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*, (London, UK), pp. 75–86, ACM, 2015. i, 53

[5] E. Santini, M. Ianni, A. Pellegrini, and F. Quaglia, "Hardware-transactional-memory based speculative parallel discrete event simulation of very fine grain models," in *Proceedings of the 2015 IEEE 22Nd International Conference on High Performance Computing (HiPC)*, (Washington, DC, USA), pp. 145–154, IEEE Computer Society, 2015. i

[6] T. Dickman, S. Gupta, and P. A. Wilsey, "Event pool structures for pdes on many-core beowulf clusters," in *Proceedings of the 2013 ACM SIGSIM conference on Principles of advanced discrete simulation*, pp. 103–114, May 2013. i, 3, 51, 147, 148

[7] S. Gupta and P. A. Wilsey, "Quantitative driven optimization of a time warp kernel," in *Proceedings of the 2017 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*, pp. 27–38, ACM, 2017. i, 4, 56, 59

[8] W. T. Tang, R. S. M. Goh, and I. L.-J. Thng, "Ladder queue: An o(1) priority queue structure for large-scale discrete event simulation," *ACM Transactions on Modeling and Computer Simulation*, vol. 15, pp. 175–204, July 2005. i, 3, 37, 41, 42, 77, 146

[9] P. A. Wilsey, "Some properties of events executed in discrete-event simulation models," in *Proceedings of the 2016 annual ACM Conference on SIGSIM Principles of Advanced Discrete Simulation*, (Banf, Alberta, Canada), pp. 165–176, ACM, 2016. ii, 4, 53, 54, 147

[10] R. Fujimoto, "Parallel discrete event simulation," *Communications of the ACM*, vol. 33, pp. 30–53, Oct. 1990. 2, 7, 8, 26

[11] D. E. Culler, A. Gupta, and J. P. Singh, *Parallel Computer Architecture: A Hardware/Software Approach*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1997. 2, 11

[12] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 5th ed., 2011. 2, 11

[13] D. Jefferson, B. Beckman, F. Wieland, L. Blume, M. Di Loreto, P. Hontalas, P. Laroche, K. Sturdevant, J. Tupman, V. Warren, J. Wedel, H. Younger, and S. Bellenot, "Distributed simulation and the Time Warp operating system," in *Proceedings of the 12^{th} SIGOPS — Symposium of Operating Systems Principles*, pp. 77–93, 1987. 2

[14] D. Jefferson and P. L. Reiher, "Supercritical speedup," in *Proceedings of the 24^{th} Annual Simulation Symposium* (A. H. Rutan, ed.), pp. 159–168, IEEE Computer Society Press, Apr. 1991. 3, 131

[15] R. Brown, "Calendar queues: A fast O(1) priority queue implementation for the simulation event set problem," *Communications of the ACM*, vol. 31, pp. 1220–1227, Oct. 1988. 3, 21, 22, 36, 37, 41, 146

[16] P. Crawford, S. J. Eidenbenz, P. D. Barnes, and P. A. Wilsey, "Some properties of communication behaviors in discrete-event simulation models," in *Simulation Conference (WSC), 2017 Winter*, pp. 1025–1036, IEEE, 2017. 4, 62

[17] A. Alt and P. A. Wilsey, "Profile driven partitioning of parallel simulation models," in *Proceedings of the 2014 Winter Simulation Conference*, pp. 2750–2761, IEEE Press, 2014. 5, 28, 53, 147, 148

[18] A. M. Law and W. D. Kelton, *Simulation Modeling and Analysis*. McGraw-Hill, 3rd ed., 2000. 7

[19] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Communications of ACM*, vol. 21, pp. 558–565, July 1978. 8, 30

[20] R. M. Fujimoto and M. Hybinette, "Computing global virtual time in shared-memory multiprocessors," Aug. 1994. xxi, 10, 18, 27

[21] D. Weber, "Time warp simulation on multi-core processors and clusters," *Master's thesis, University of Cincinnati, Cincinnati, OH*, 2016. 10, 11, 27

[22] W. Gropp, E. Lusk, and A. Skjellum, *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. Cambridge, MA: MIT Press, 1994. 12, 15

[23] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. S. Sunderam, *PVM: Parallel virtual machine: a users' guide and tutorial for networked parallel computing*. MIT press, 1994. 12

[24] S. Das, R. Fujimoto, K. Panesar, D. Allison, and M. Hybinette, "GTW: a Time Warp system for shared memory multiprocessors," in *Proceedings of the 1994 Winter Simulation Conference* (J. D. Tew, S. Manivannan, D. A. Sadowski, and A. F. Seila, eds.), pp. 1332–1339, Dec. 1994. xxi, 17, 18

[25] H. Avril and C. Tropper, "Clustered time warp and logic simulation," in *Proceedings of the 9th Workshop on Parallel and Distributed Simulation*, pp. 112–119, 1995. 19

[26] C. D. Carothers, D. Bauer, and S. Pearce, "Ross: A high-performance, low memory, modular time warp system," *Journal of Parallel and Distributed Computing*, pp. 53–60, 2000. 20, 75

[27] J. W. J. Williams, "Algorithm 232: heapsort," *Communications of the ACM*, vol. 7, no. 6, pp. 347–348, 1964. 21

[28] D. Sleator and R. Tarjan, "Self adjusting binary search trees," *Journal of the ACM*, vol. 32, no. 3, pp. 652–686, 1985. 21, 36, 146

[29] G. Andelson-Velskii and E. Landis, "An algorithm for the organisation of information," *Soviet. Math*, vol. 3, no. 1259-1262, p. 128, 1962. 21

[30] C. D. Carothers, K. S. Perumalla, and R. M. Fujimoto, "Efficient optimistic parallel simulations using reverse computation," in *Proceedings of the thirteenth workshop on Parallel and distributed simulation, PADS'99*, pp. 126–135, May 1999. 21

[31] D. Jagtap, N. Abu-Ghazaleh, and D. Ponomarev, "Optimization of parallel discrete event simulator for multi-core systems," in *Parallel Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*, pp. 520–531, May 2012. 21

[32] D. E. Martin, T. J. McBrayer, and P. A. Wilsey, "WARPED: A Time Warp simulation kernel for analysis and application development," in *29th Hawaii International Conference on System Sciences (HICSS-29)* (H. El-Rewini and B. D. Shriver, eds.), vol. Volume I, pp. 383–386, Jan. 1996. 21

[33] R. Radhakrishnan, D. E. Martin, M. Chetlur, D. M. Rao, and P. A. Wilsey, "An Object-Oriented Time Warp Simulation Kernel," in *Proceedings of the International Symposium on Computing in Object-Oriented Parallel Environments (ISCOPE'98)* (D. Caromel, R. R. Oldehoeft, and M. Tholburn, eds.), vol. LNCS 1505, pp. 13–23, Springer-Verlag, Dec. 1998. 21

[34] A. Pellegrini, R. Vitali, and F. Quaglia, "The rome optimistic simulator: Core internals and programming model," in *Proceedings of the 4th International ICST Conference on Simulation Tools and Techniques*, SIMUTools '11, (ICST, Brussels, Belgium, Belgium), pp. 96–98, ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2011. 22

[35] R. Vitali, A. Pellegrini, and F. Quaglia, "Towards symmetric multi-threaded optimistic simulation kernels," in *Proceedings of the 2012 ACM/IEEE/SCS 26th Workshop on Principles of Advanced and Distributed Simulation*, PADS '12, (Washington, DC, USA), pp. 211–220, IEEE Computer Society, 2012. 22

[36] R. Vitali, A. Pellegrini, and F. Quaglia, "Assessing load-sharing within optimistic simulation platforms," in *Proceedings of the Winter Simulation Conference*, WSC '12, pp. 289:1–289:13, Winter Simulation Conference, 2012. 22

[37] R. Marotta, M. Ianni, A. Pellegrini, and F. Quaglia, "A lock-free o(1) event pool and its application to share-everything pdes platforms," in *Proceedings of the 20th International Symposium on Distributed*

Simulation and Real-Time Applications, DS-RT '16, (Piscataway, NJ, USA), pp. 53–60, IEEE Press, 2016. 22

[38] R. Marotta, M. Ianni, A. Pellegrini, and F. Quaglia, "A non-blocking priority queue for the pending event set," in *Proceedings of the 9th EAI International Conference on Simulation Tools and Techniques*, SIMUTOOLS'16, (ICST, Brussels, Belgium, Belgium), pp. 46–55, ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2016. 22

[39] M. Ianni, R. Marotta, A. Pellegrini, and F. Quaglia, "Towards a fully non-blocking share-everything pdes platform," in *Proceedings of the 21st International Symposium on Distributed Simulation and Real Time Applications*, DS-RT '17, (Piscataway, NJ, USA), pp. 25–32, IEEE Press, 2017. 22

[40] R. Marotta, M. Ianni, A. Pellegrini, and F. Quaglia, "A conflict-resilient lock-free calendar queue for scalable share-everything pdes platforms," in *Proceedings of the 2017 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*, SIGSIM-PADS '17, (New York, NY, USA), pp. 15–26, ACM, 2017. 22

[41] A. Pellegrini and F. Quaglia, "A fine-grain time-sharing time warp system," *ACM Trans. Model. Comput. Simul.*, vol. 27, pp. 10:1–10:25, May 2017. 22

[42] J. Fleischmann and P. A. Wilsey, "Comparative analysis of periodic state saving techniques in Time Warp simulators," in *Proc. of the 9th Workshop on Parallel and Distributed Simulation (PADS 95)*, pp. 50–58, June 1995. 27

[43] F. Mattern, "Efficient algorithms for distributed snapshots and global virtual time approximation," *Journal of Parallel and Distributed Computing*, vol. 18, pp. 423–434, Aug. 1993. 27

[44] S. Bellenot, "Global virtual time algorithms," in *Distributed Simulation*, pp. 122–127, Society for Computer Simulation, Jan. 1990. 27

[45] G. Karypis and V. Kumar, "A fast and high quality multilevel scheme for partitioning irregular graphs," *SIAM Journal on Scientific Computing*, vol. 20, no. 1, pp. 359–34, 1998. 28, 148

[46] V. D. Blondel, J.-L. Guillaume, R. Lambiotte, and E. Lefebvre, "Fast unfolding of communities in large networks," *Journal of statistical mechanics: theory and experiment*, vol. 2008, no. 10, p. P10008, 2008. 28, 61, 147

[47] R. Rönngren and M. Liljenstam, "On event ordering in parallel discrete event simulation," in *Proceedings of the Thirteenth Workshop on Parallel and Distributed Simulation*, PADS '99, (Washington, DC, USA), pp. 38–45, IEEE Computer Society, 1999. 29, 30

[48] B. D. Lubachevsky *et al.*, "Rollback sometimes works...if filtered," in *Winter Simulation Conference*, pp. 630–639, Society for Computer Simulation, Dec. 1989. 31

[49] R. Rönngren, R. Ayani, R. M. Fujimoto, and S. R. Das, "Efficient implementation of event sets in time warp," in *Proceedings of the 1993 workshop on Parallel and distributed simulation*, pp. 101–108, May 1993. 35

[50] A. Newell, J. C. Shaw, and H. A. Simon, "Empirical explorations of the logic theory machine: a case study in heuristic," in *Papers presented at the February 26-28, 1957, western joint computer conference: Techniques for reliability*, pp. 218–230, ACM, 1957. 35

[51] D. D. Sleator and R. E. Tarjan, "Self-adjusting heaps," *SIAM Journal on Computing*, vol. 15, no. 1, pp. 52–69, 1986. 35

[52] S. Prasad and B. Naquib, "Effectiveness of global event queues in rollback reduction and load balancing," in *Proceedings of the 9th Workshop on Parallel and Distributed Simulation (PADS 95)*, pp. 187–190, 1995. 35

[53] S. K. Prasad, S. I. Sawant, and B. Naqib, "Using parallel data structures in optimistic discrete event simulation of varying granularity on shared-memory computers," in *Algorithms and Architectures for Parallel Processing, 1995. ICAPP 95. IEEE First ICA/sup 3/PP., IEEE First International Conference on*, vol. 1, pp. 365–374, IEEE, 1995. 35

[54] T. Santoro and F. Quaglia, "A low-overhead constant-time ltf scheduler for optimistic simulation systems," in *Computers and Communications (ISCC), 2010 IEEE Symposium on*, pp. 948–953, IEEE, 2010. 36

[55] D. Musser and A. Stepanov, "Generic programming," invited paper," in *ISSAC*, vol. 88, 1989. 36, 146

[56] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, "Introduction to algorithms second edition," 2001. 36

[57] E. W. Dijkstra, "Solution of a problem in concurrent programming control," *Commun. ACM*, vol. 8, pp. 569–, Sept. 1965. 44

[58] J. M. Mellor-Crummey and M. L. Scott, "Algorithms for scalable synchronization on shared-memory multiprocessors," *ACM Transactions on Computer Systems (TOCS)*, vol. 9, no. 1, pp. 21–65, 1991. 44

[59] M. Fomitchev and E. Ruppert, "Lock-free linked lists and skip lists," in *Proceedings of the twenty-third annual ACM symposium on Principles of distributed computing*, pp. 50–59, ACM, 2004. 44

[60] J. D. Valois, "Lock-free linked lists using compare-and-swap," in *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '95, pp. 214–222, 1995. 44

[61] T. L. Harris, "A pragmatic implementation of non-blocking linked-lists," in *Proceedings of the 15th International Conference on Distributed Computing*, DISC '01, (London, UK, UK), pp. 300–314, Springer-Verlag, 2001. 44

[62] K. Zhang, Y. Zhao, Y. Yang, Y. Liu, and M. Spear, "Practical non-blocking unordered lists," in *International Symposium on Distributed Computing*, pp. 239–253, Springer, 2013. 44

[63] M. M. Michael and M. L. Scott, "Correction of a memory management method for lock-free data structures," tech. rep., University of Rochester, Rochester, NY, USA, 1995. 44

[64] M. M. Michael and M. L. Scott, "Simple, fast, and practical non-blocking and blocking concurrent queue algorithms," in *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '96, pp. 267–275, 1996. 44

[65] M. M. Michael and M. L. Scott, "Nonblocking algorithms and preemption-safe locking on multiprogrammed shared memory multiprocessors," *Journal of Parallel and Distributed Computing*, vol. 51, pp. 1–26, May 1998. 44

[66] M. M. Michael, "High performance dynamic lock-free hash tables and list-based sets," in *Proceedings of the Fourteenth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '02, pp. 73–82, 2002. 44

[67] M. M. Michael, "Hazard pointers: Safe memory reclamation for lock-free objects," *IEEE Trans. Parallel Distrib. Syst.*, vol. 15, pp. 491–504, June 2004. 44

[68] A. Ghuloum, "Face the inevitable, embrace parallelism," *Communications of the ACM*, vol. 52, pp. 36–38, Sept. 2009. 51

[69] M. E. Newman, "Modularity and community structure in networks," *Proceedings of the national academy of sciences*, vol. 103, no. 23, pp. 8577–8582, 2006. 61

[70] K. S. Perumalla and S. K. Seal, "Discrete event modeling and massively parallel execution of epidemic outbreak phenomena," *Simulation*, vol. 88, pp. 768–783, July 2012. 70

[71] C. Barrett, K. Bisset, S. Eubank, X. Feng, and M. Marathe, "Episimdemics: An efficient algorithm for simulating the spread of infectious disease over large realistic social networks," in *High Performance Computing, Networking, Storage and Analysis, 2008. SC 2008. International Conference for*, pp. 1–12, Nov 2008. 70

[72] D. J. Watts and S. H. Strogatz, "Collective dynamics of 'small-world' networks," *Nature*, vol. 393, pp. 440–442, June 1998. 70, 71, 81, 289, 362

[73] A.-L. Barabási and R. Albert, "Emergence of scaling in random networks," *science*, vol. 286, no. 5439, pp. 509–512, 1999. 71, 81, 431, 504

[74] Y.-B. Lin and P. A. Fishwick, "Asynchronous parallel discrete event simulation," *IEEE Transactions on Systems, Man and Cybernetics, Part A: Systems and Humans*, vol. 26, pp. 397–412, July 1996. 74

[75] E. Deelman and B. K. Szymanski, "Dynamic load balancing in parallel discrete event simulation for spatially explicit problems," in *Parallel and Distributed Simulation, 1998. PADS 98. Proceedings. Twelfth Workshop on*, pp. 46–53, IEEE, 1998. 148

[76] R. Schlagenhaft, M. Ruhwandl, C. Sporrer, and H. Bauer, "Dynamic load balancing of a multi-cluster simulator on a network of workstations," in *ACM SIGSIM Simulation Digest*, vol. 25(1), pp. 175–180, IEEE Computer Society, 1995. 148

[77] J. Lindén, P. Bauer, S. Engblom, and B. Jonsson, "Fine-grained local dynamic load balancing in pdes," in *Proceedings of the 2018 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*, pp. 201–212, ACM, 2018. 149

Appendix A

Performance Results from all Configurations and Simulation Models

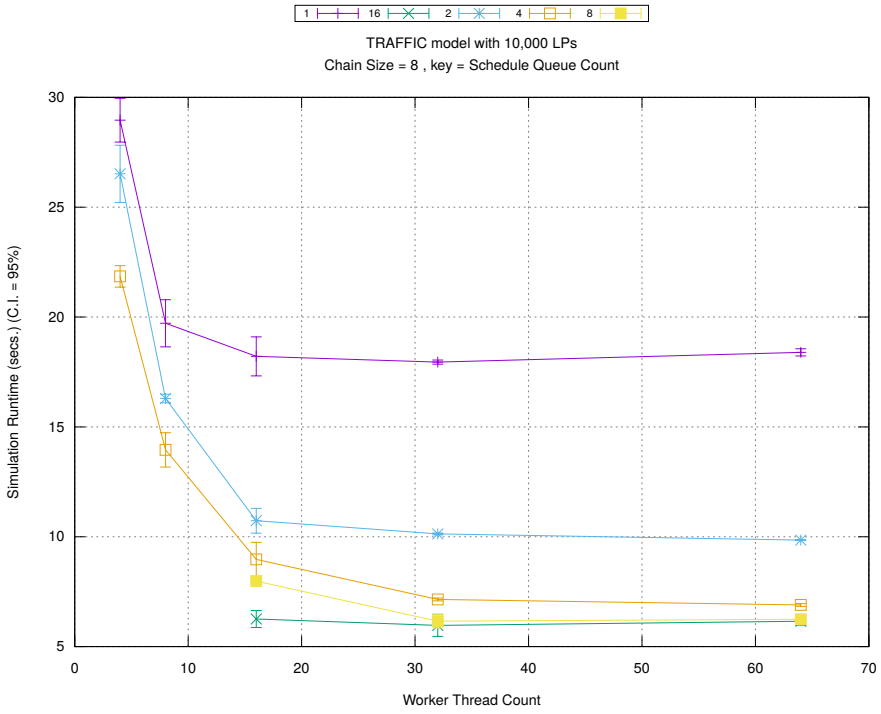
Section 6.3 describes the different benchmarks used in the quantitative study of different optimizations discussed in Chapter 5. The following sections present all the data collected for this study. Tables in each section provide configuration details for all benchmark models.

A.1 Traffic Model Configured with 10,000 LPs

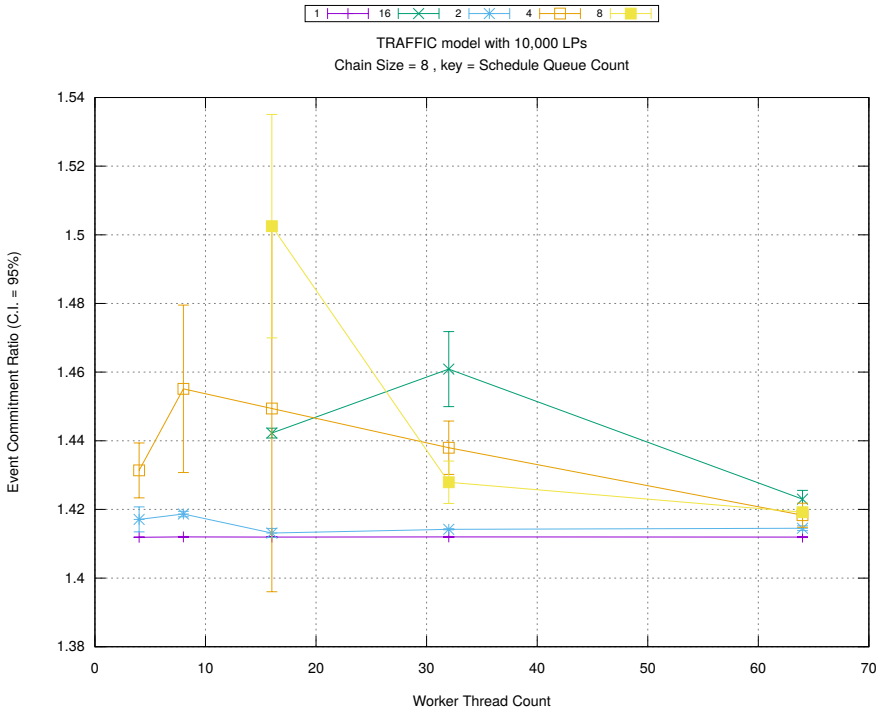
Table A.1 shows the configuration for this model.

Parameter	Values
Number of Intersections (or LPs)	10,000
Type of network connecting LPs	4-directional grid
Grid Size	100x100
Number of Cars initially at each intersection	25
Mean car arrival interval at each intersection	400 timestamp units
Simulation Time	10,000 timestamp units
Sequential Simulation Time for calculating modularity	6000 timestamp units

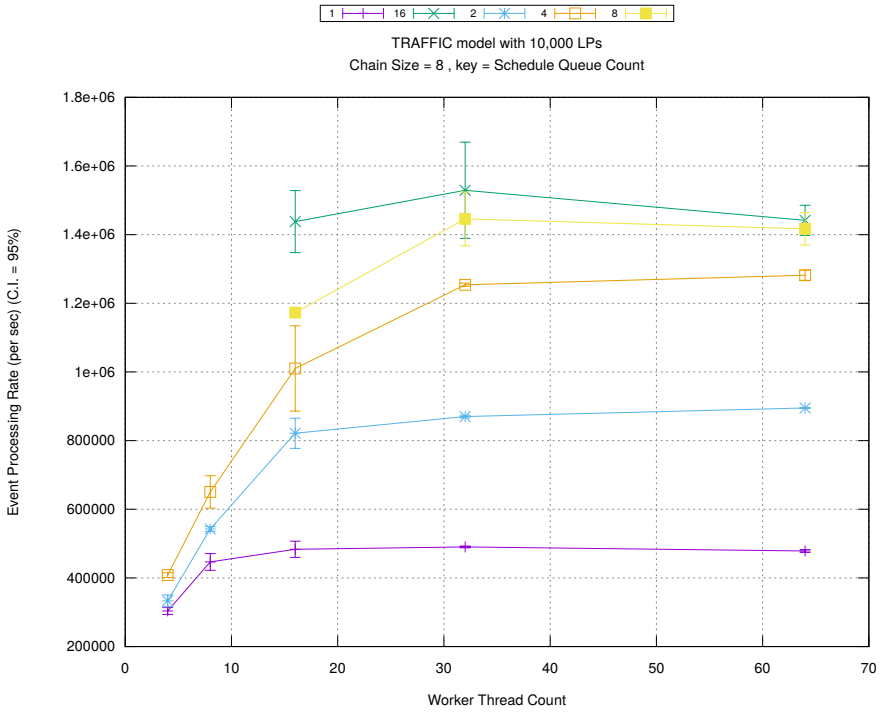
Table A.1: TRAFFIC MODEL setup



(a) Simulation Runtime (in seconds)

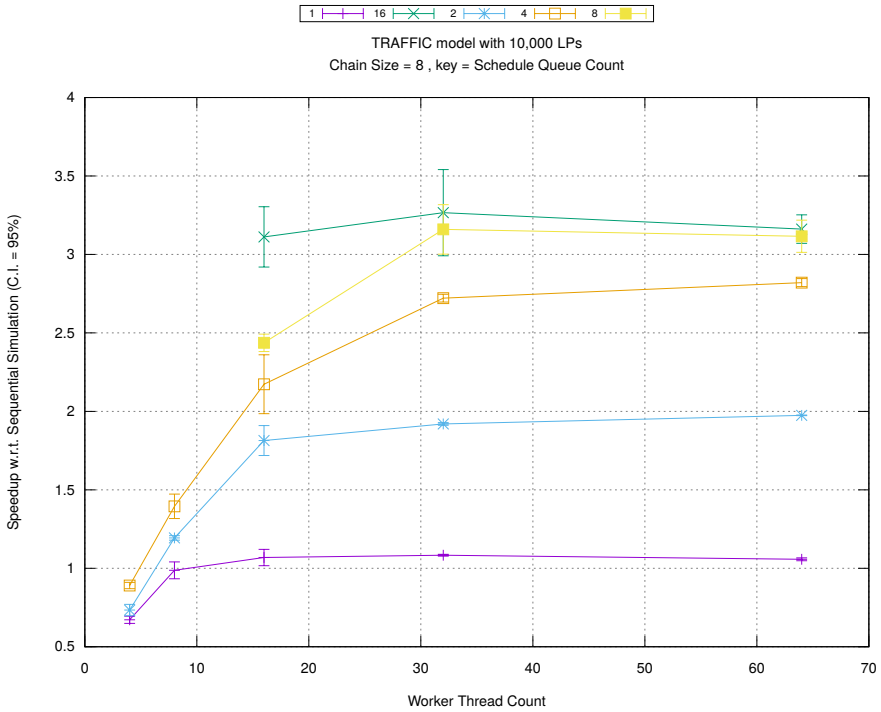


(b) Event Commitment Ratio

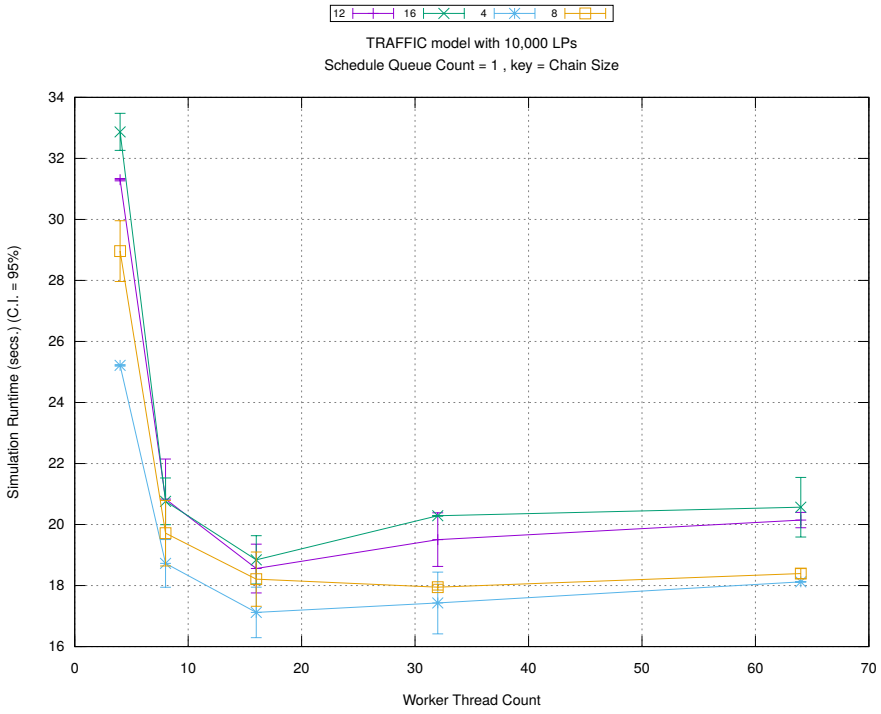


(c) Event Processing Rate (per second)

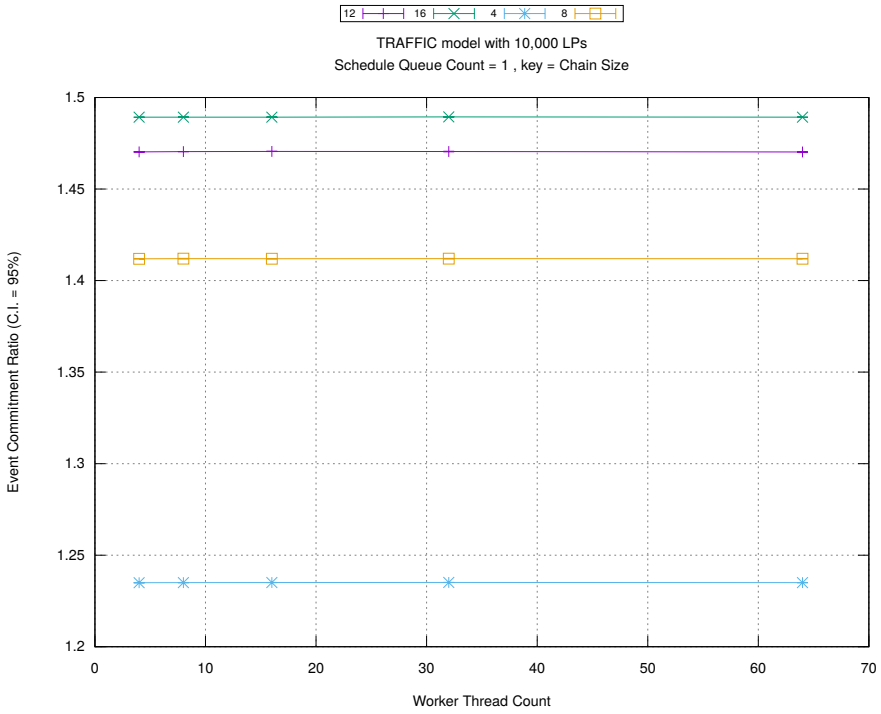
Figure A.1: traffic 10k/plots/chains/threads vs count key chainsize 8



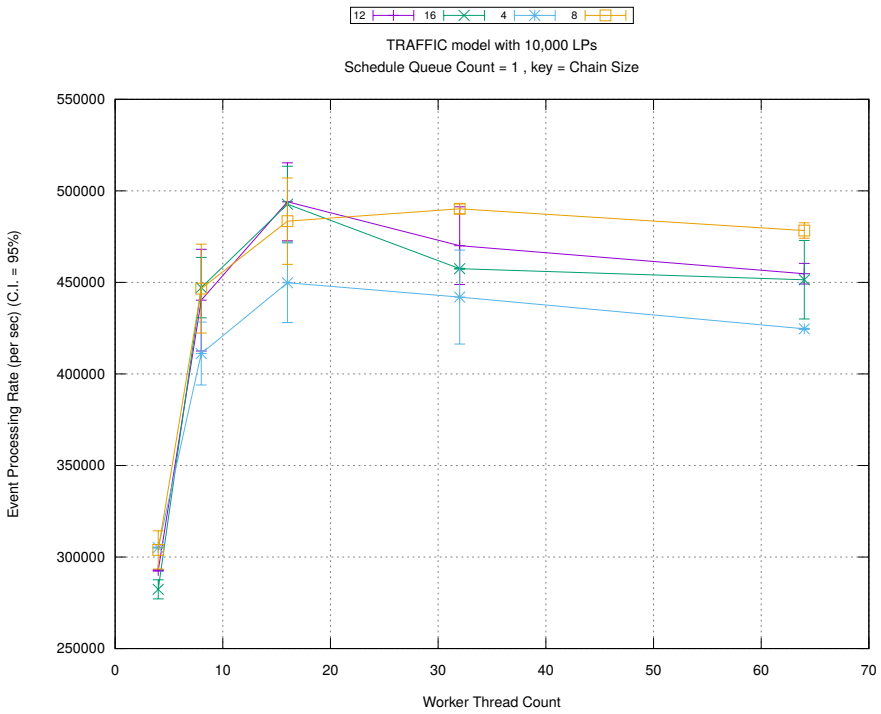
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

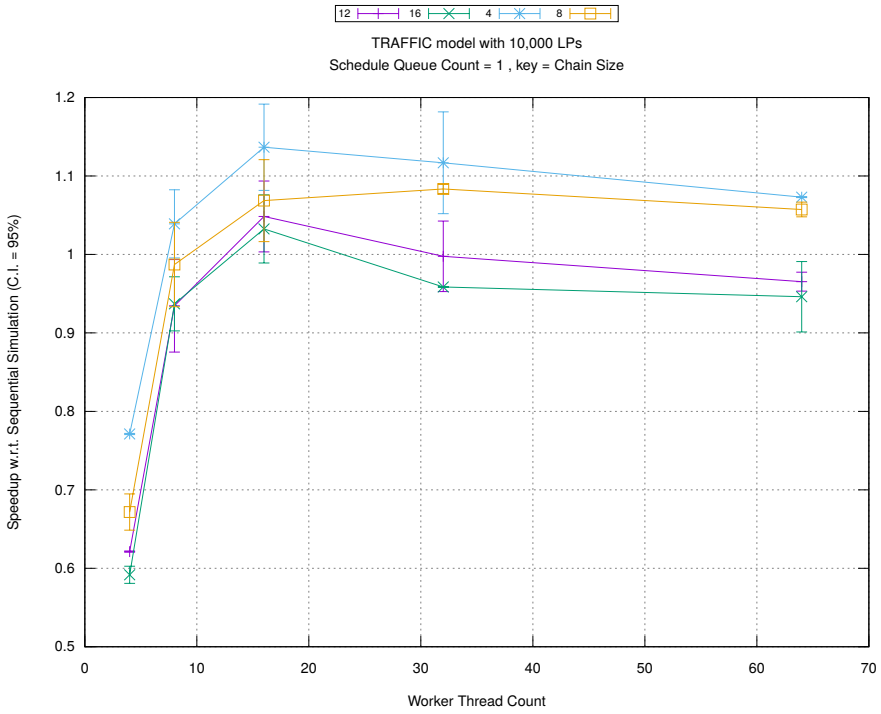


(b) Event Commitment Ratio

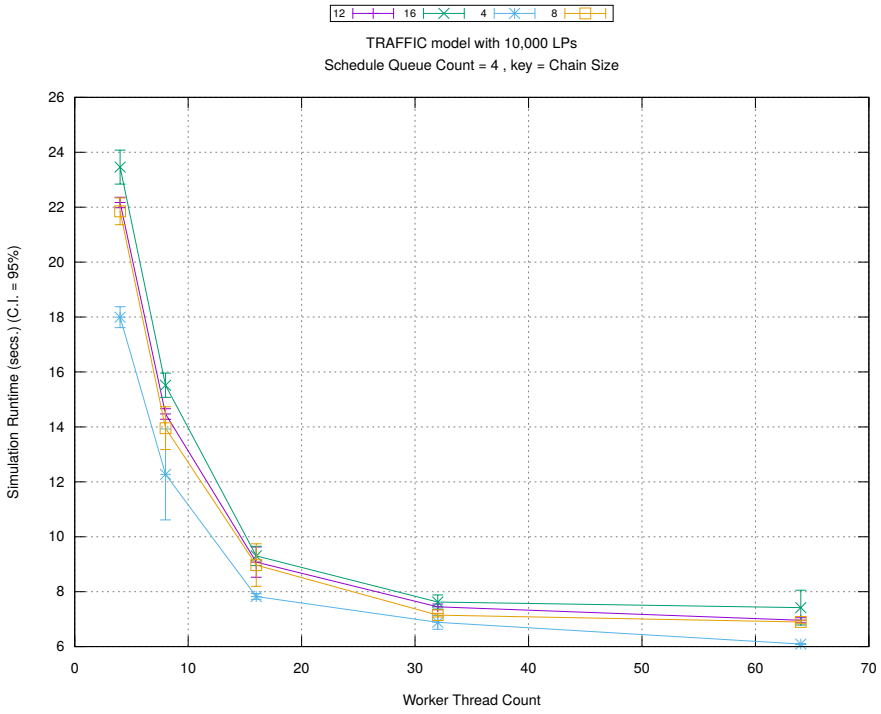


(c) Event Processing Rate (per second)

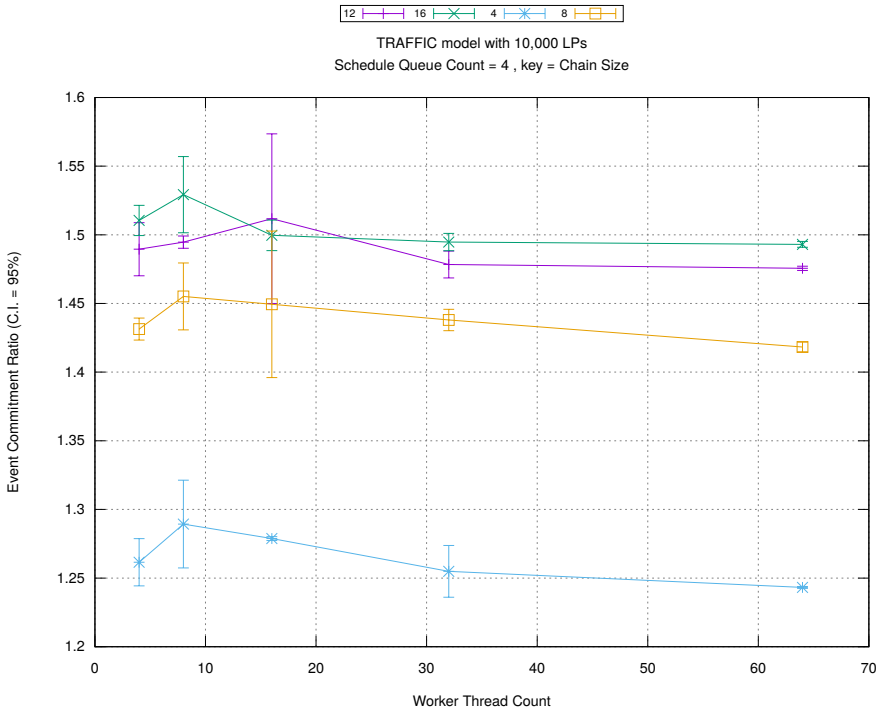
Figure A.2: traffic 10k/plots/chains/threads vs chainsize key count 1



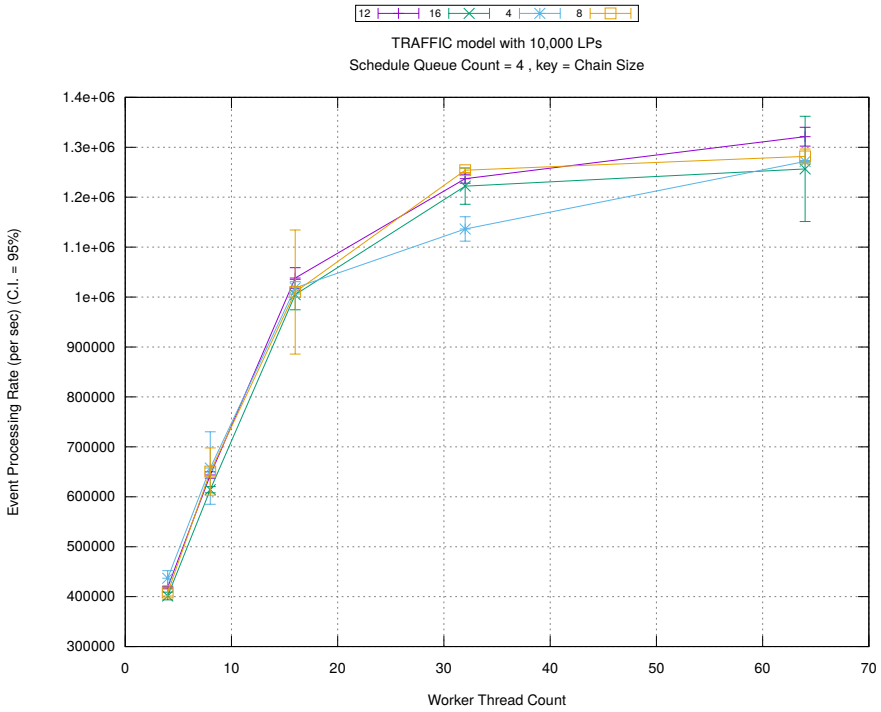
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

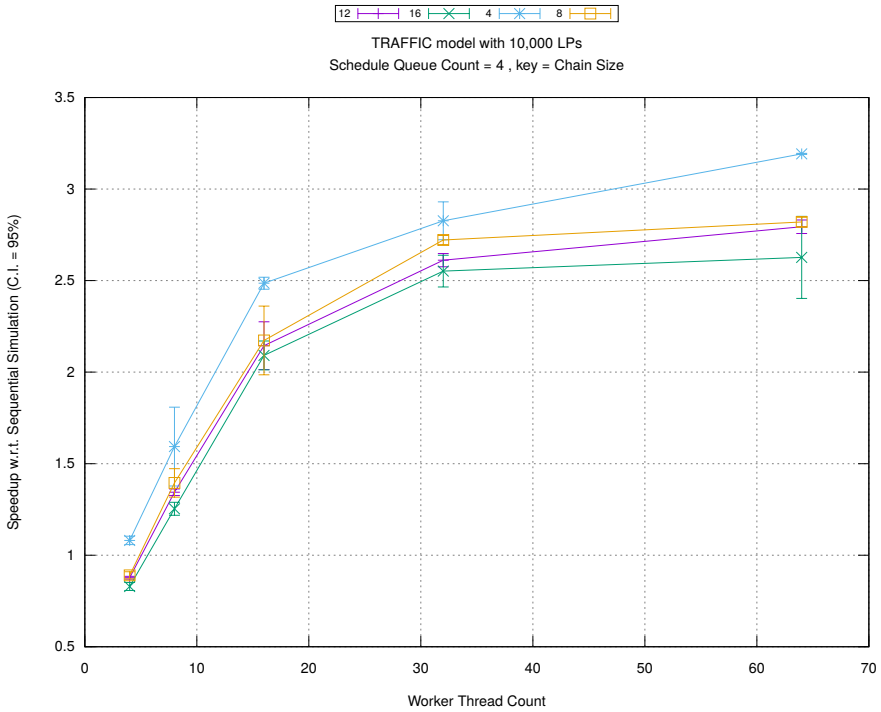


(b) Event Commitment Ratio

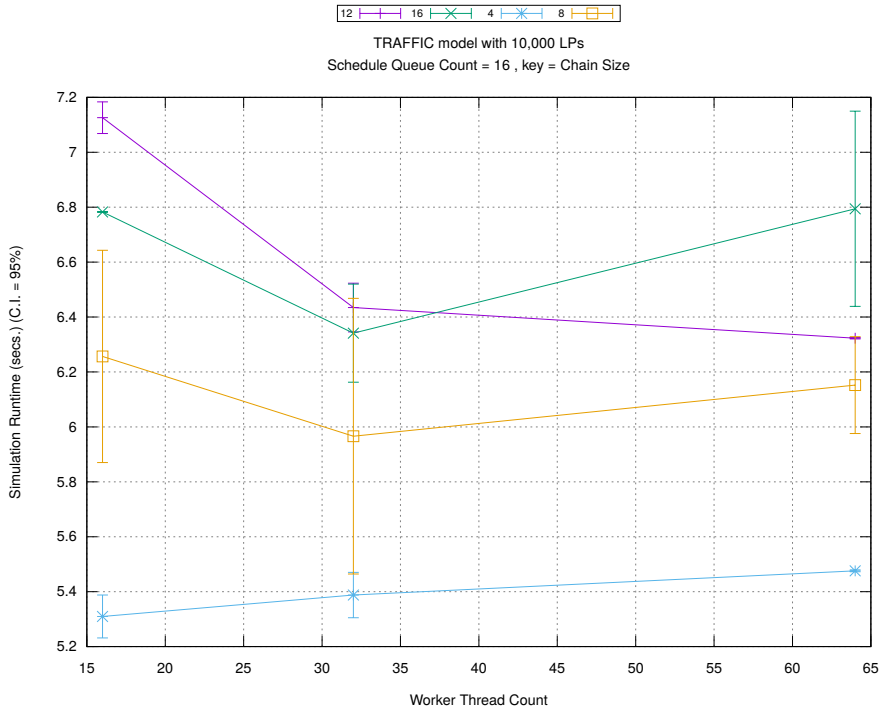


(c) Event Processing Rate (per second)

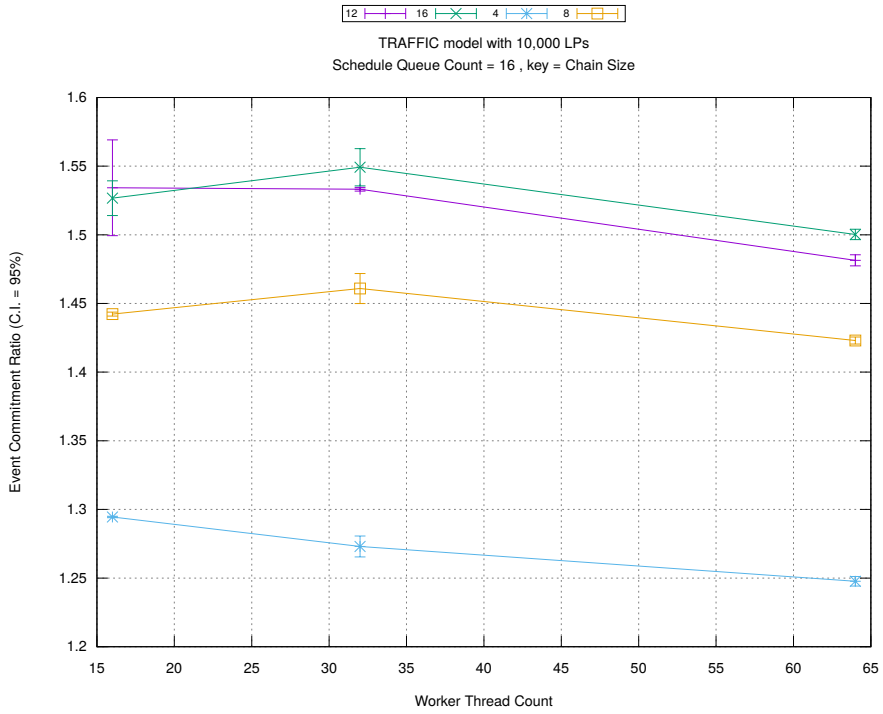
Figure A.3: traffic 10k/plots/chains/threads vs chainsize key count 4



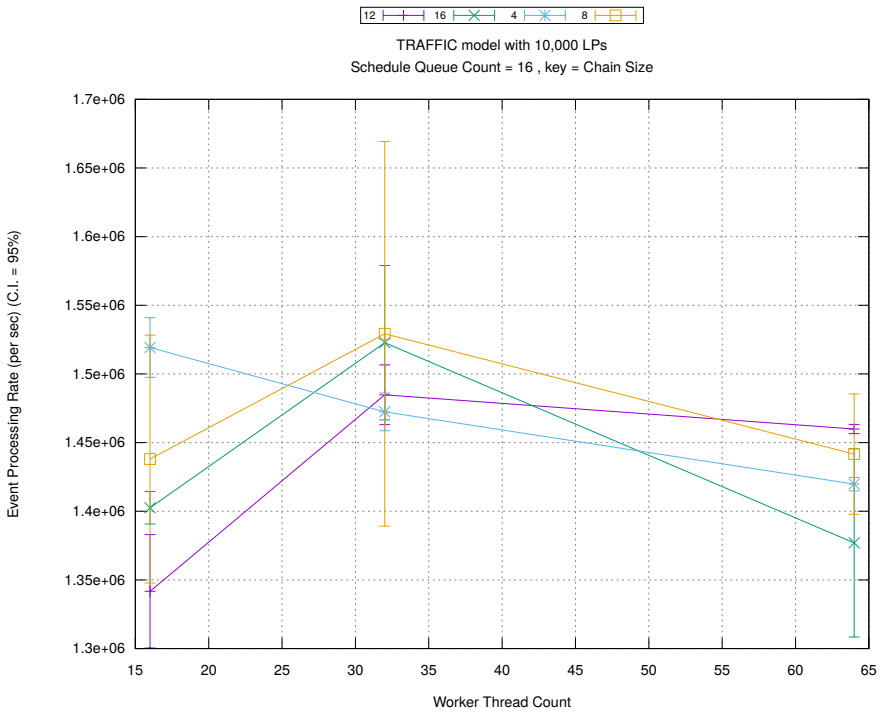
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

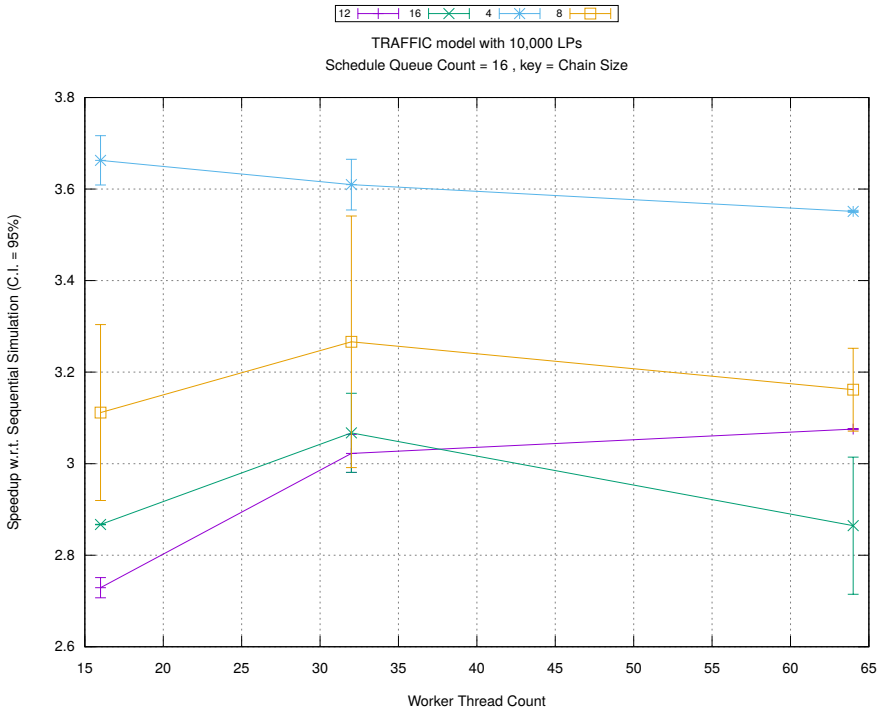


(b) Event Commitment Ratio

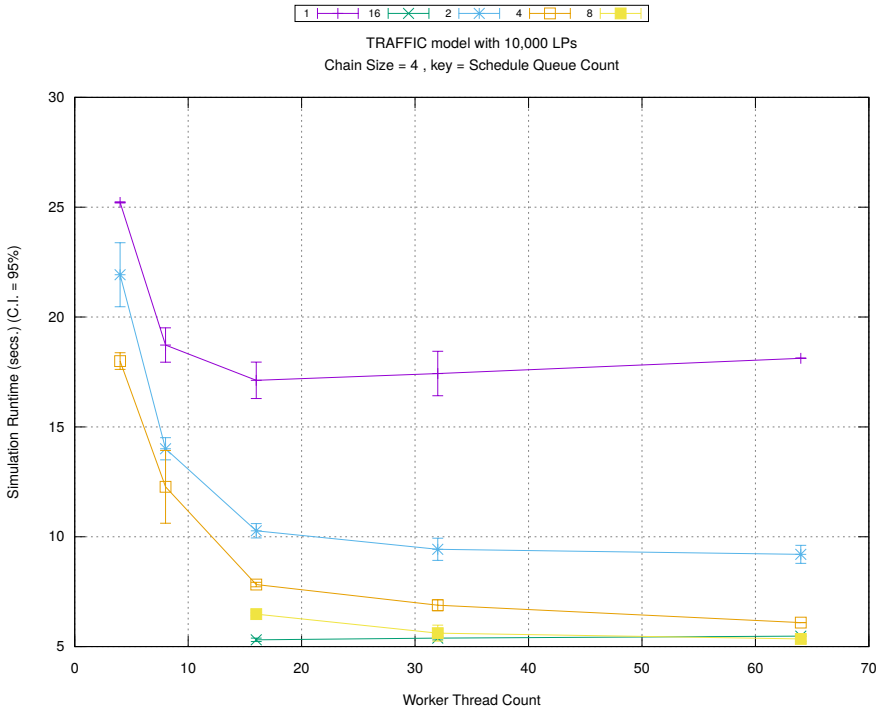


(c) Event Processing Rate (per second)

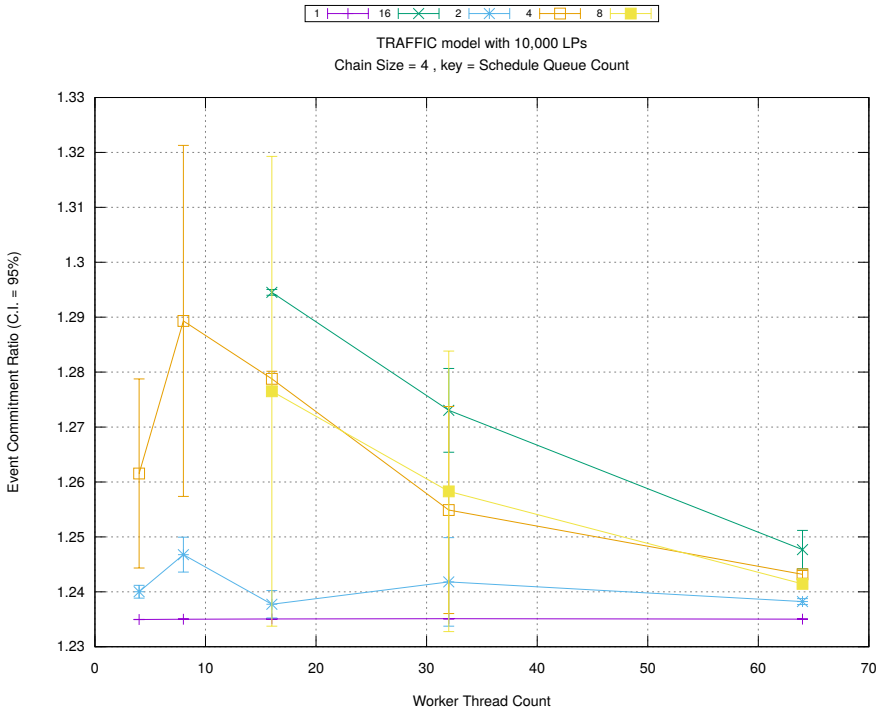
Figure A.4: traffic 10k/plots/chains/threads vs chainsize key count 16



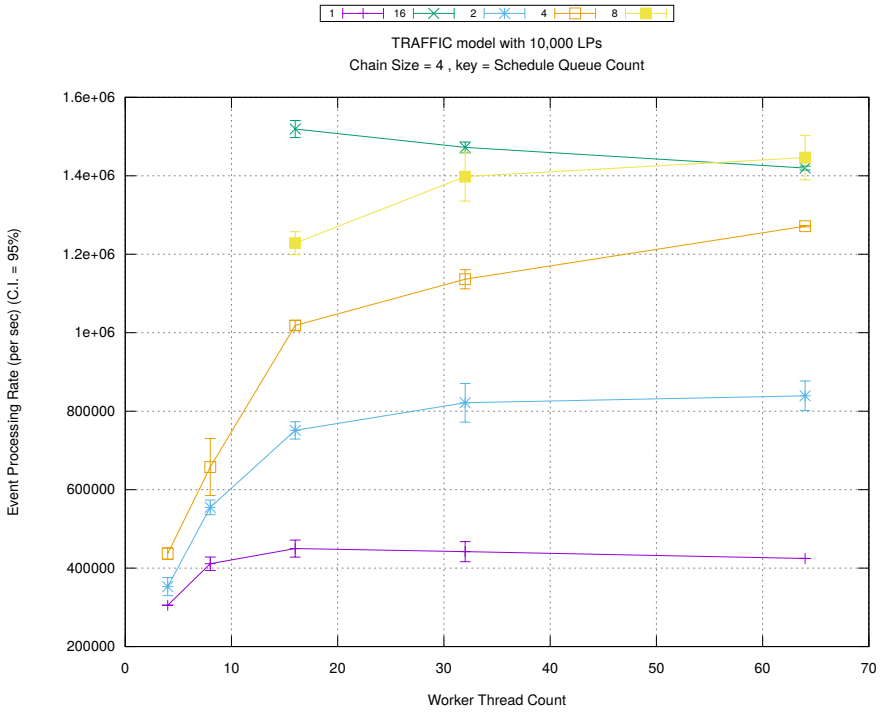
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

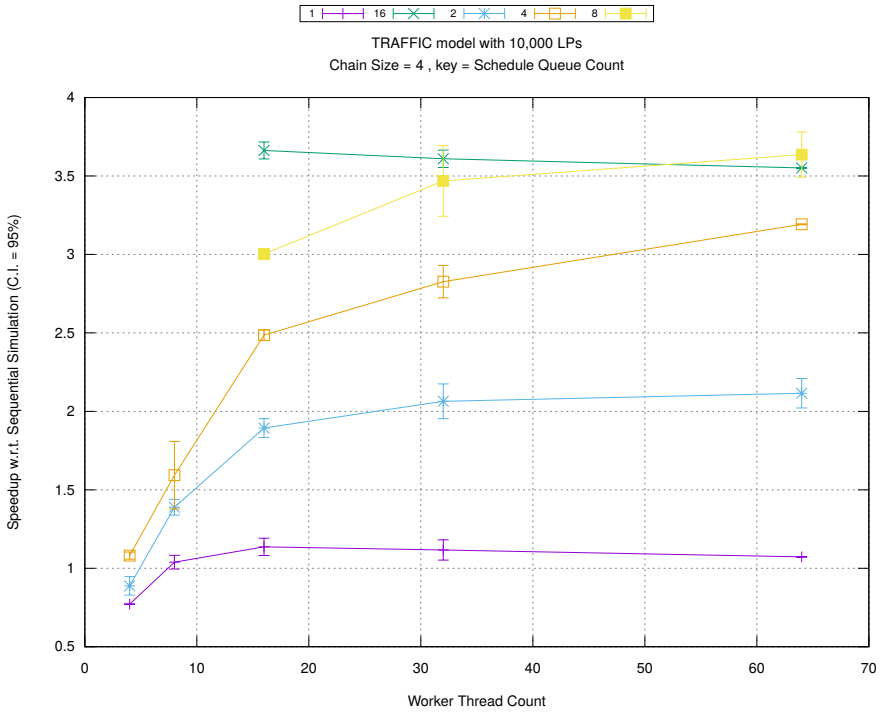


(b) Event Commitment Ratio

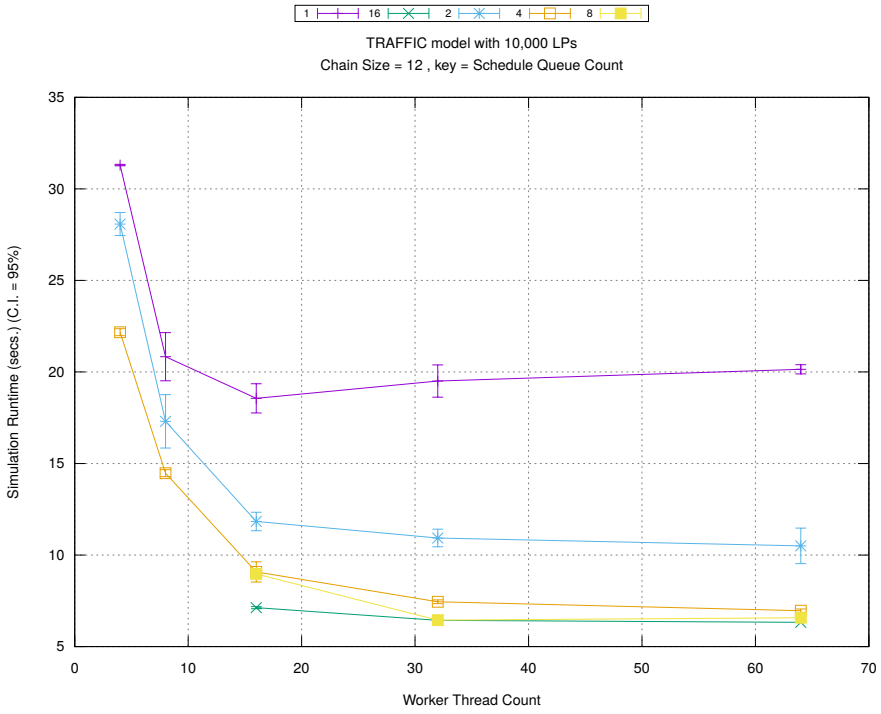


(c) Event Processing Rate (per second)

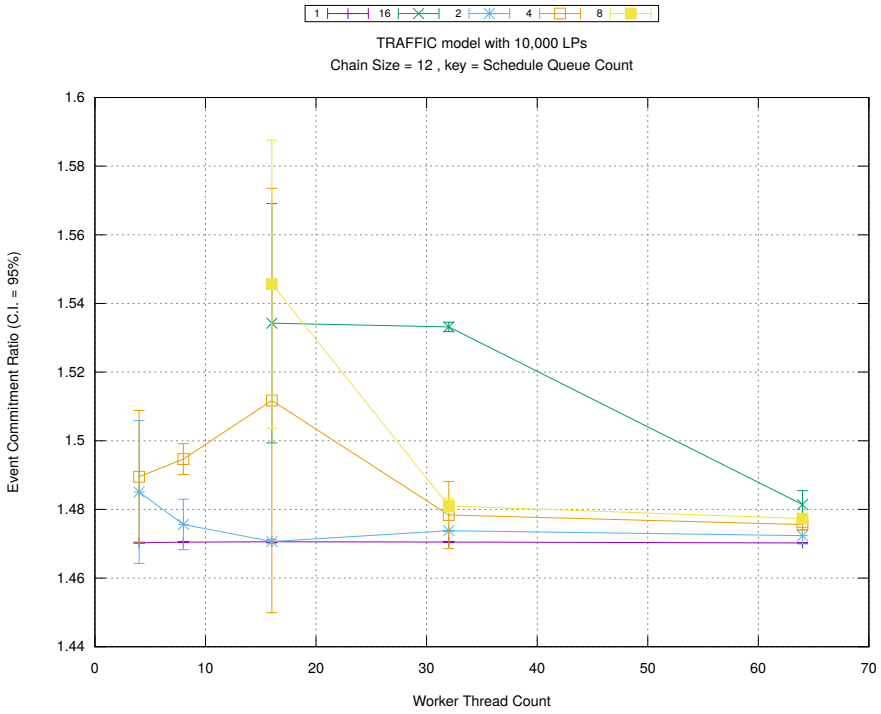
Figure A.5: traffic 10k/plots/chains/threads vs count key chainsize 4



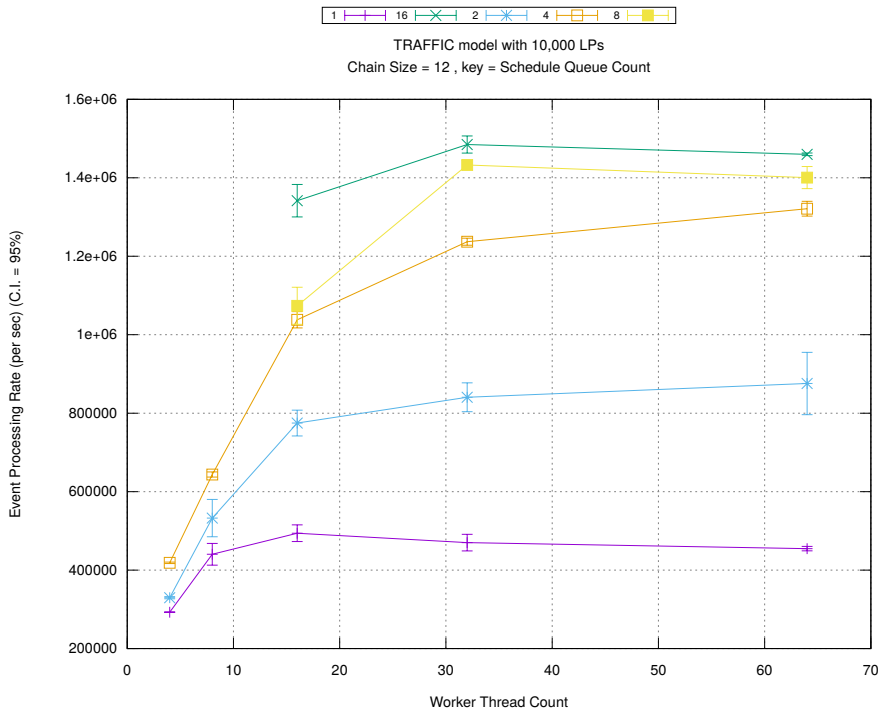
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

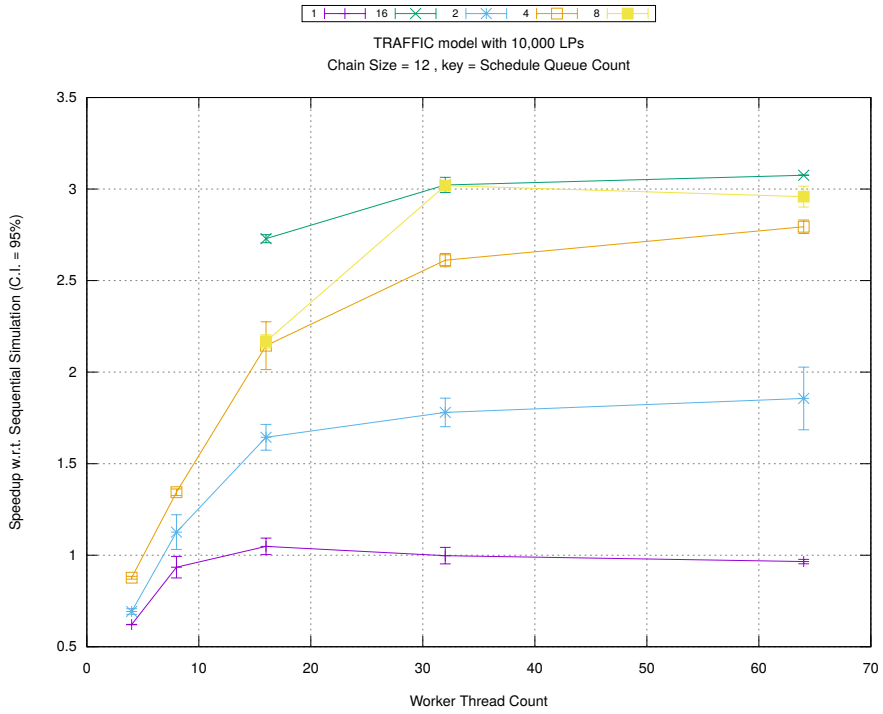


(b) Event Commitment Ratio

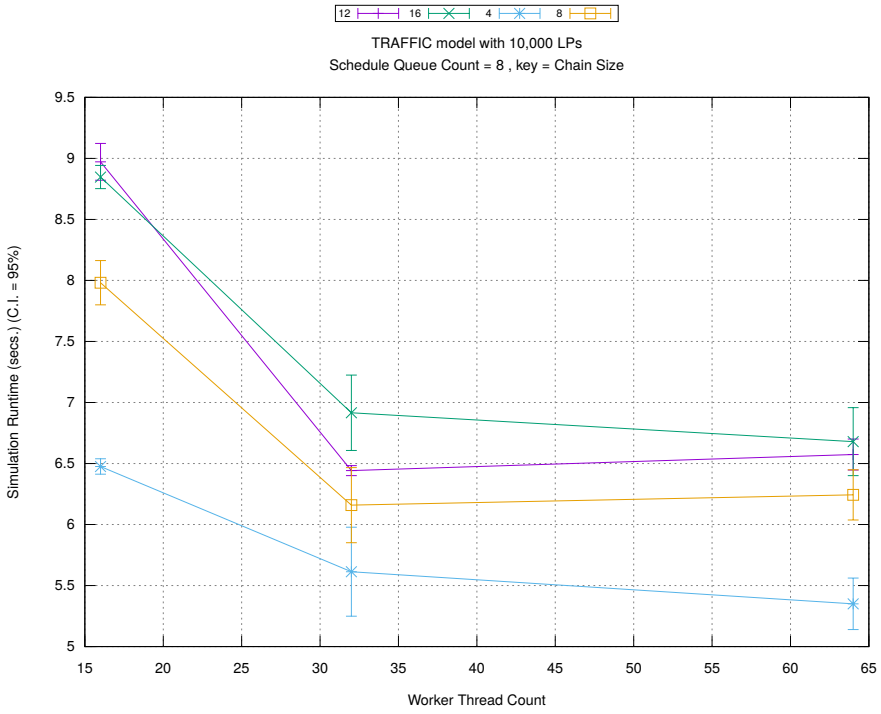


(c) Event Processing Rate (per second)

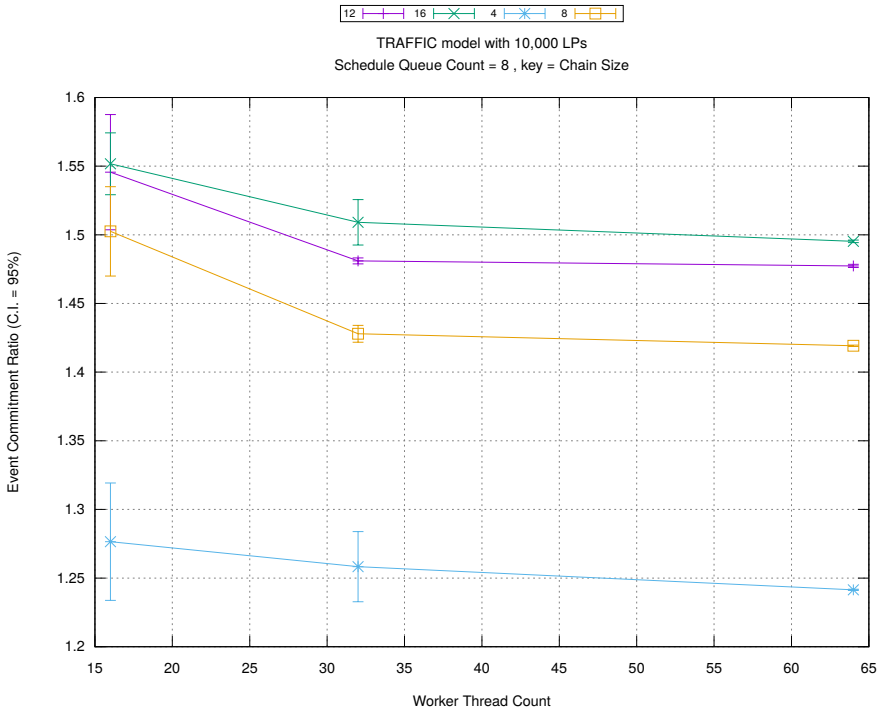
Figure A.6: traffic 10k/plots/chains/threads vs count key chainsize 12



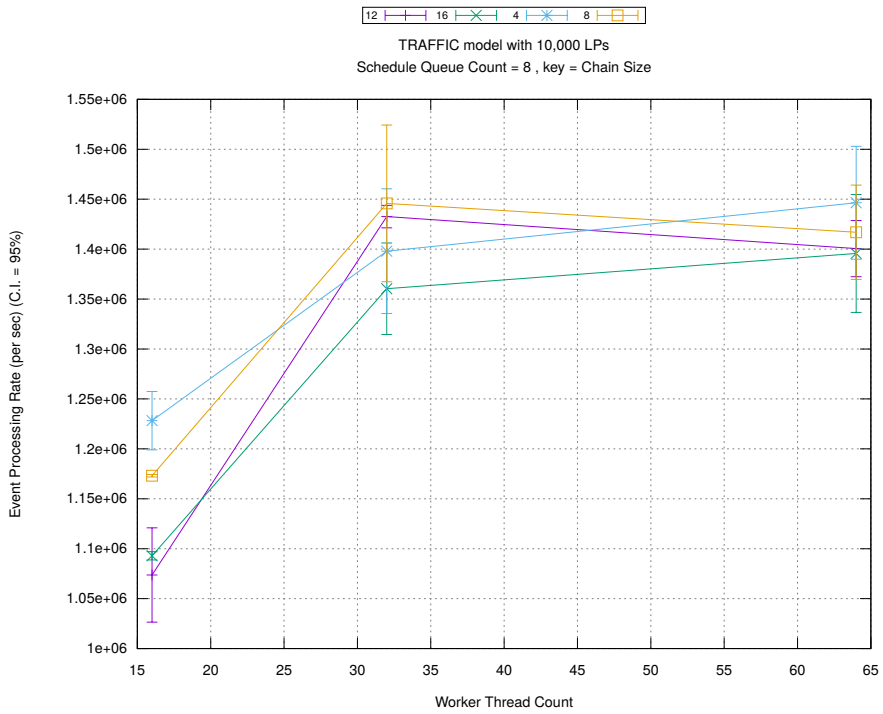
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

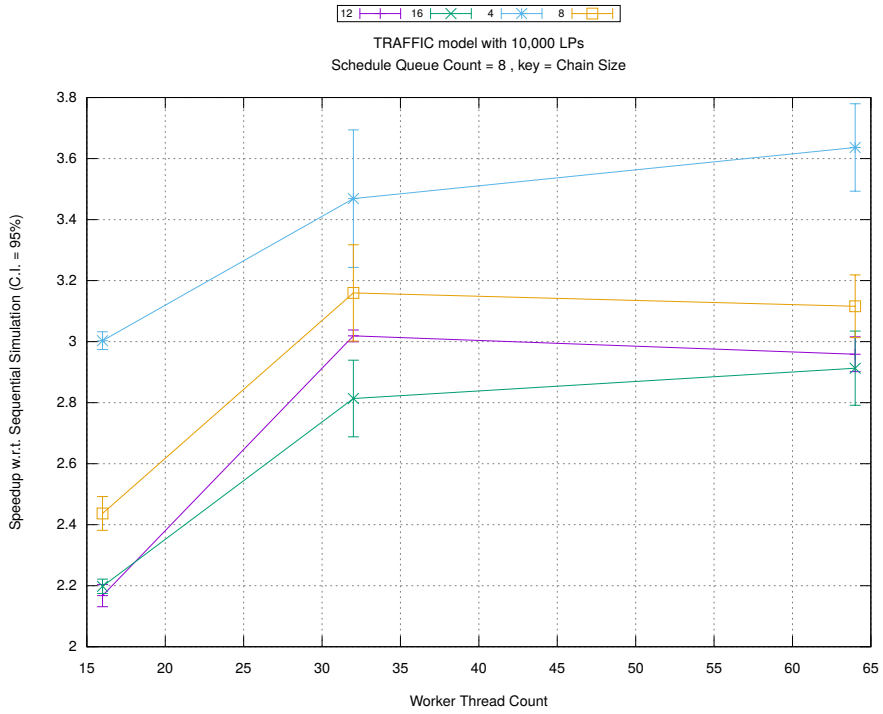


(b) Event Commitment Ratio

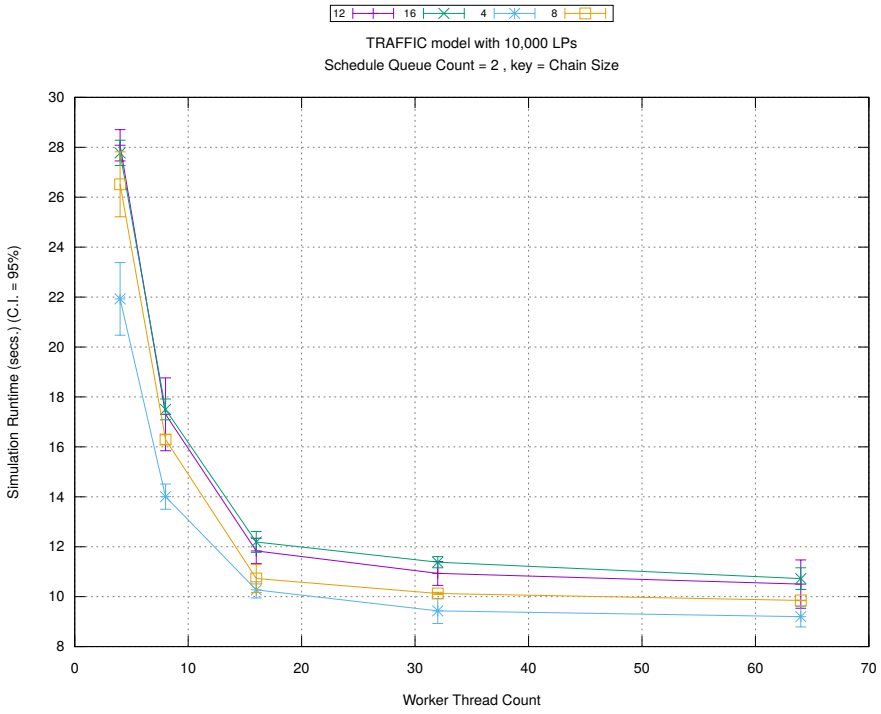


(c) Event Processing Rate (per second)

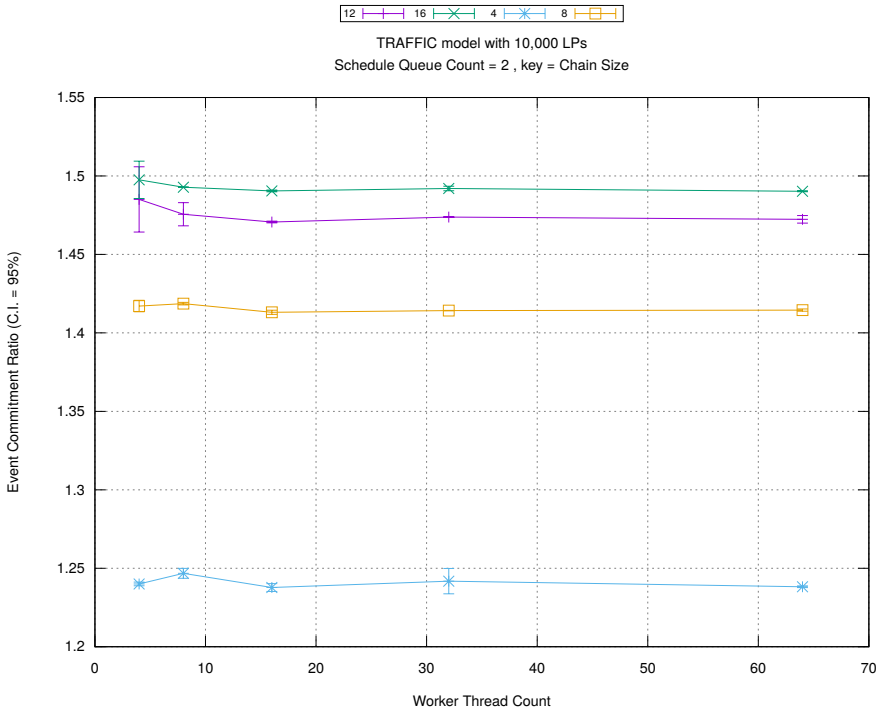
Figure A.7: traffic 10k/plots/chains/threads vs chainsize key count 8



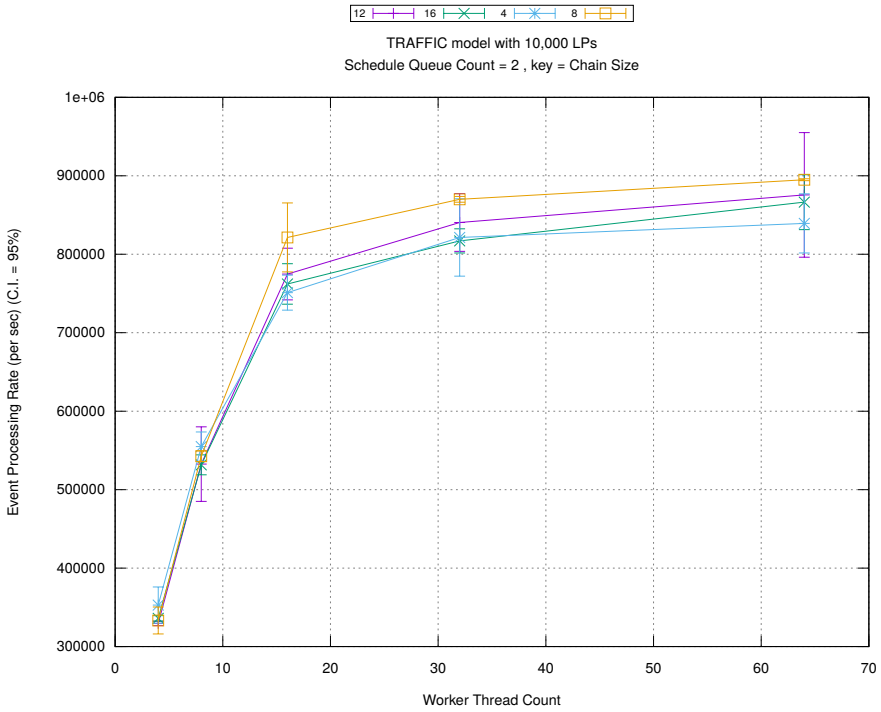
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

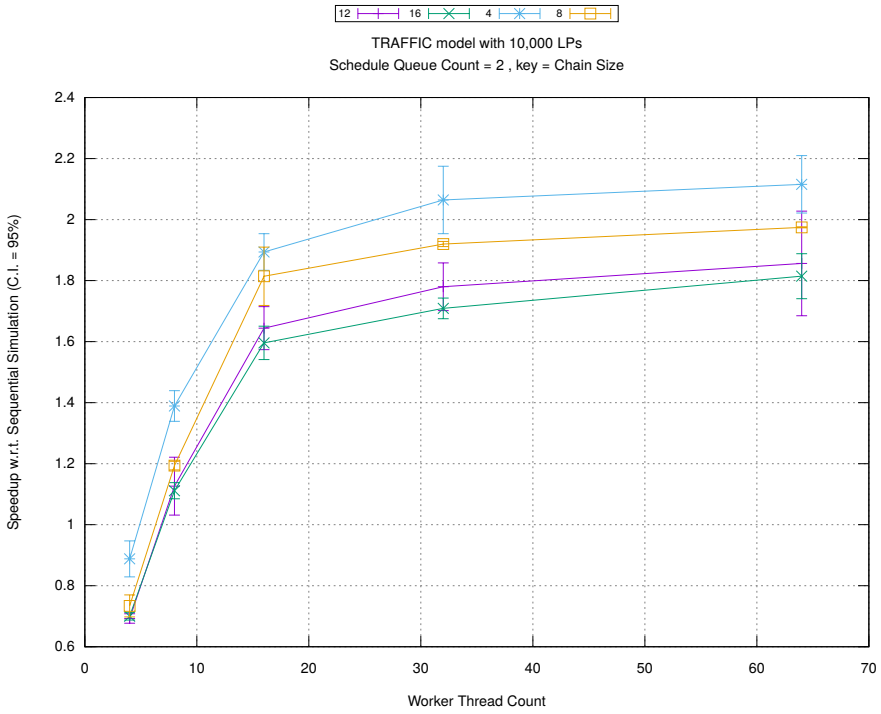


(b) Event Commitment Ratio

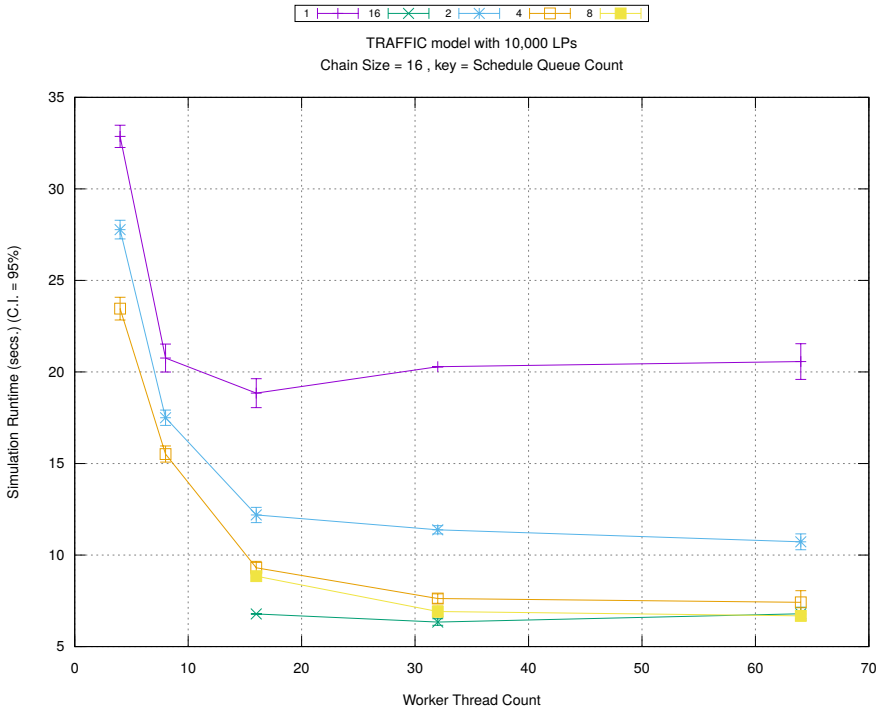


(c) Event Processing Rate (per second)

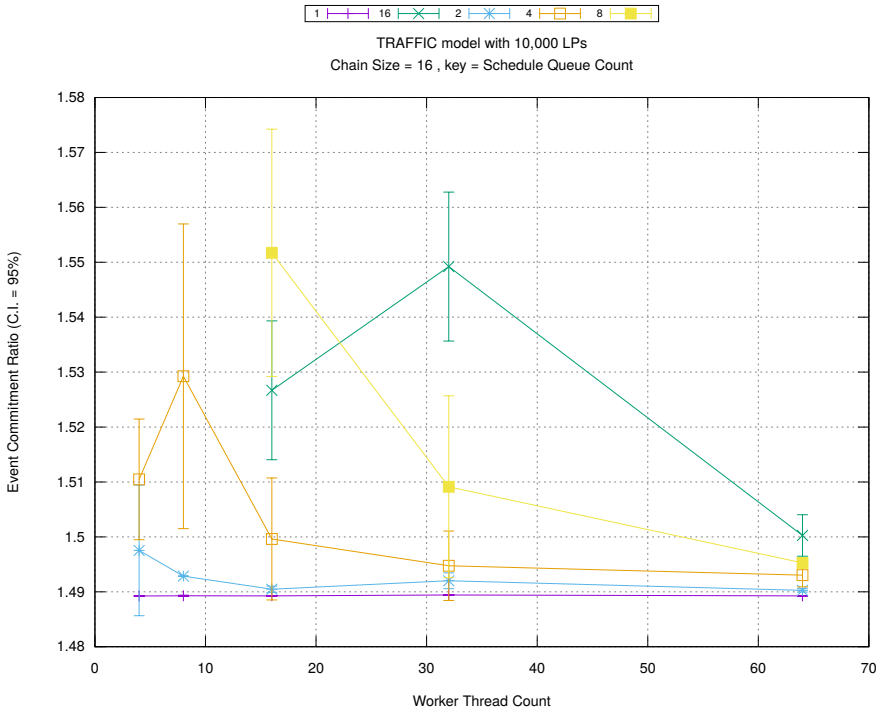
Figure A.8: traffic 10k/plots/chains/threads vs chainsize key count 2



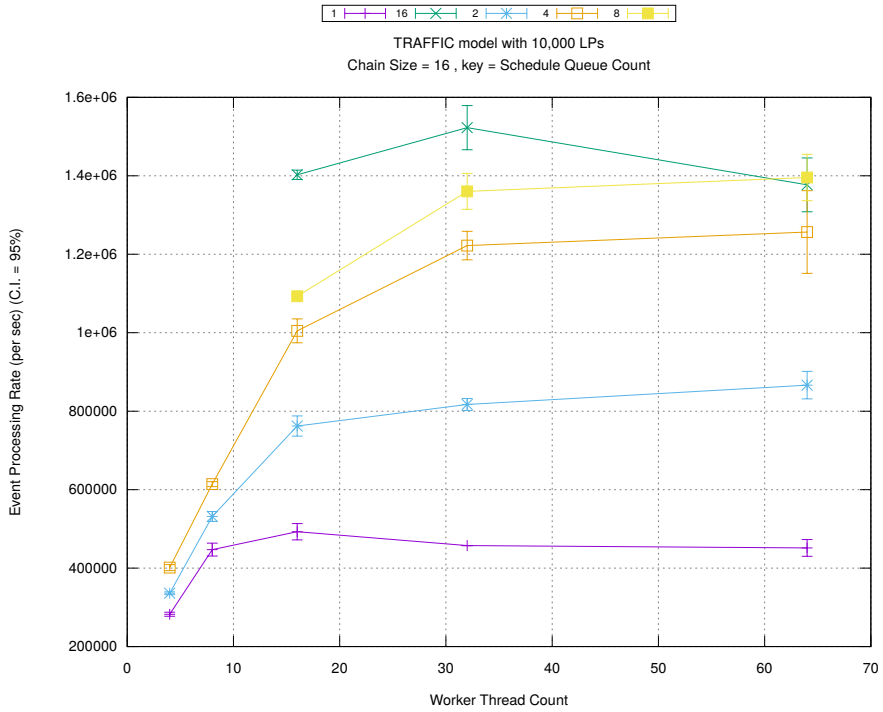
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

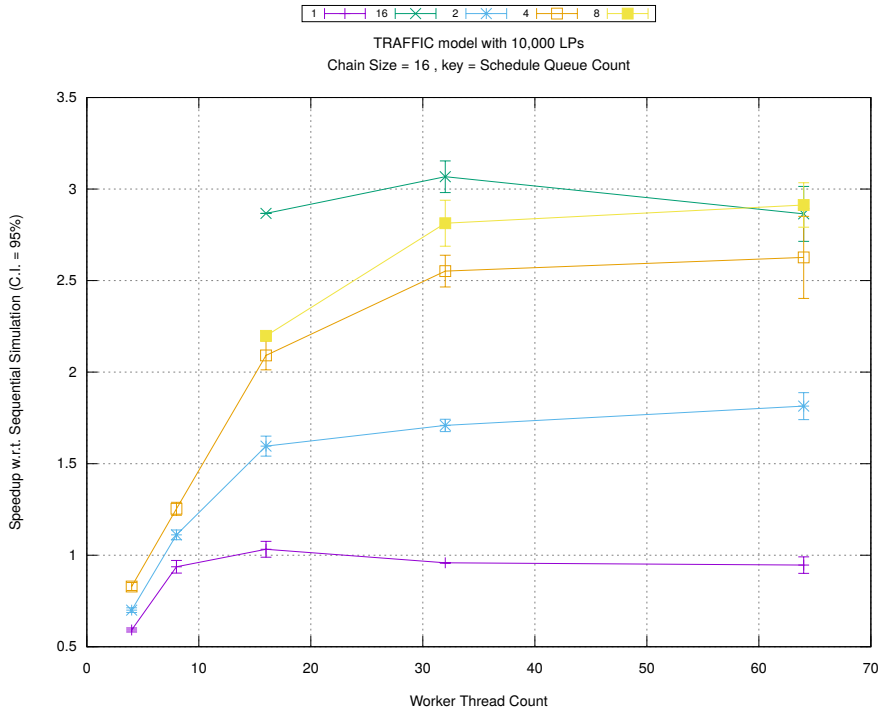


(b) Event Commitment Ratio

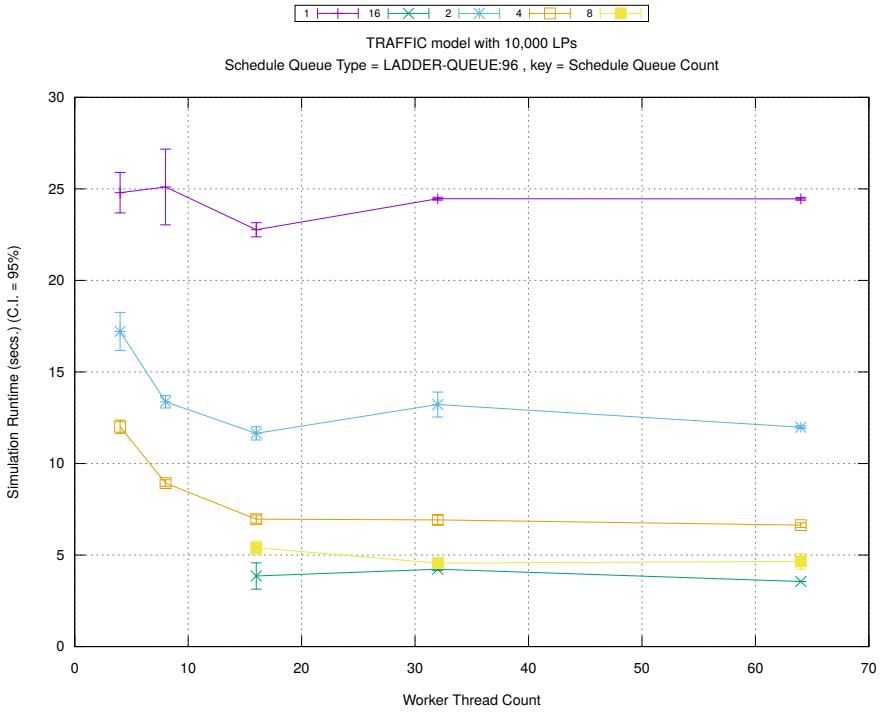


(c) Event Processing Rate (per second)

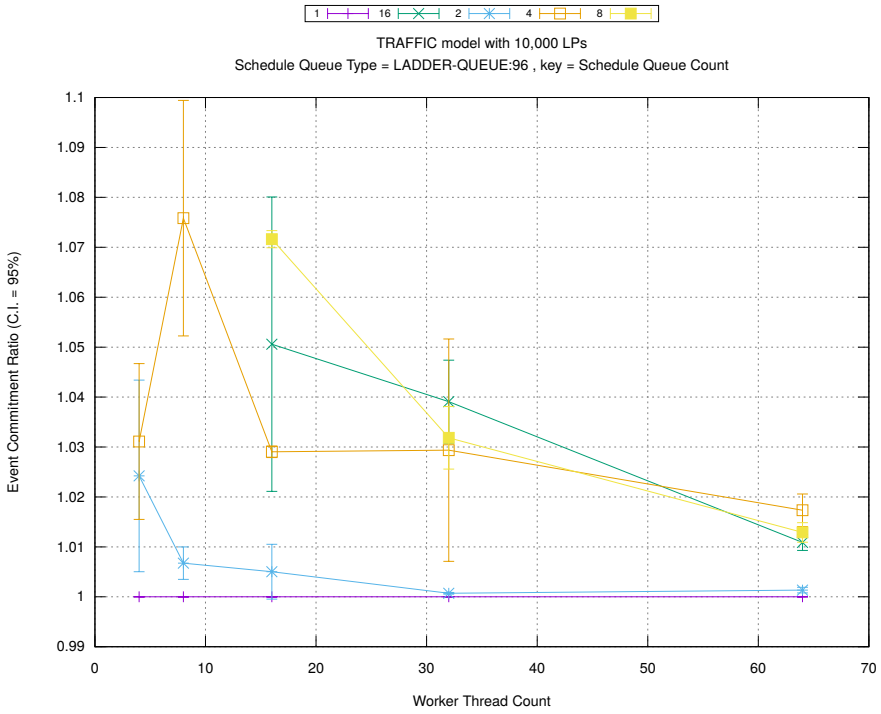
Figure A.9: traffic 10k/plots/chains/threads vs count key chainsize 16



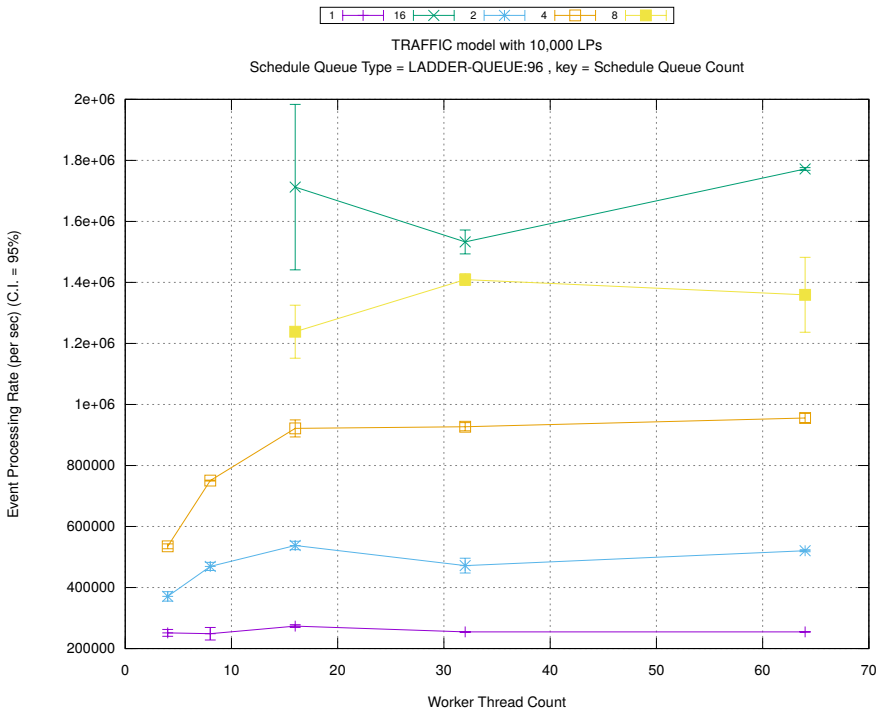
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

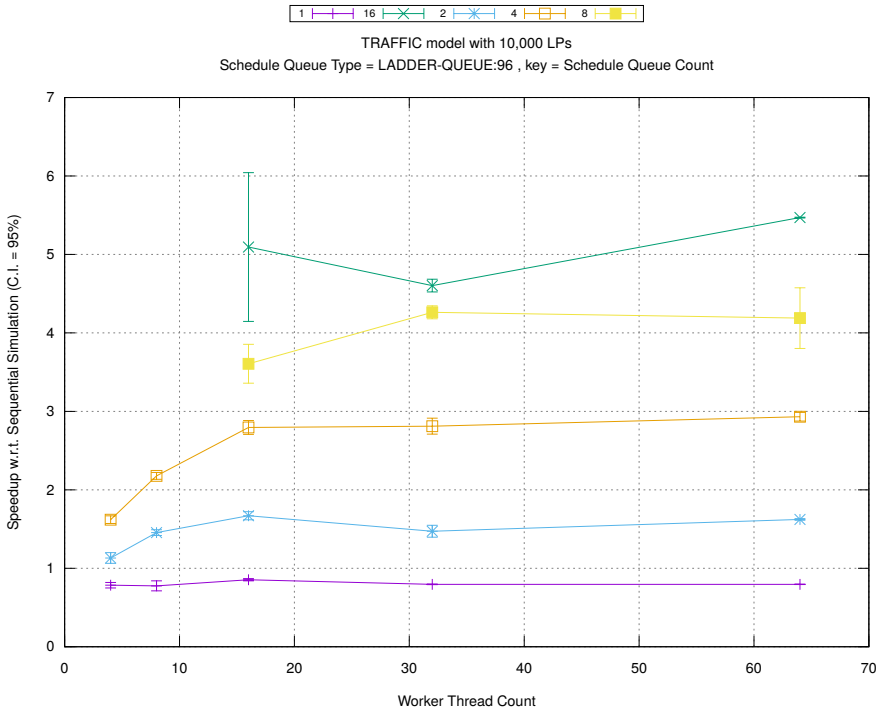


(b) Event Commitment Ratio

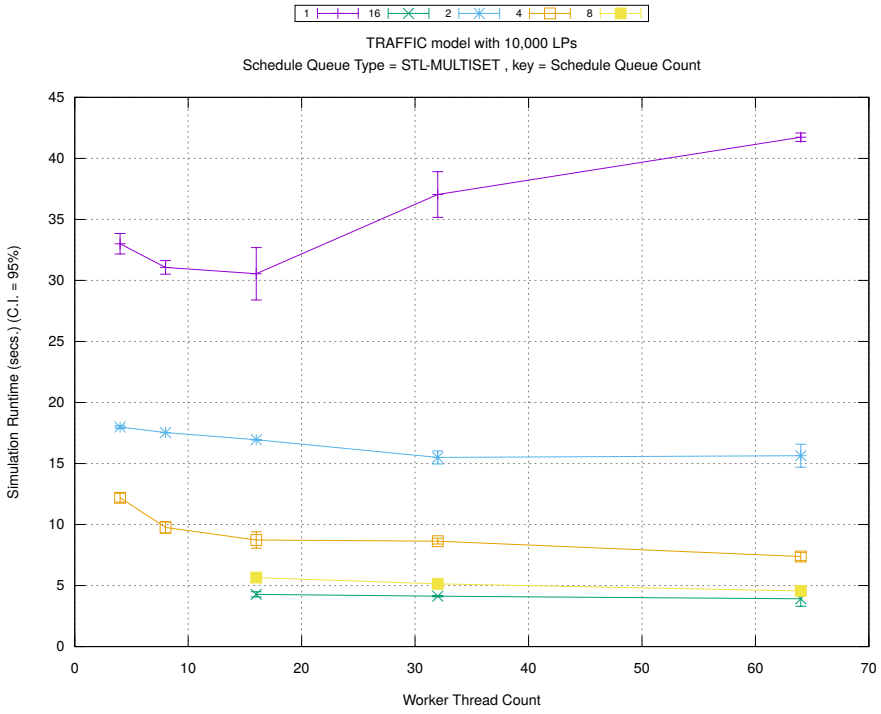


(c) Event Processing Rate (per second)

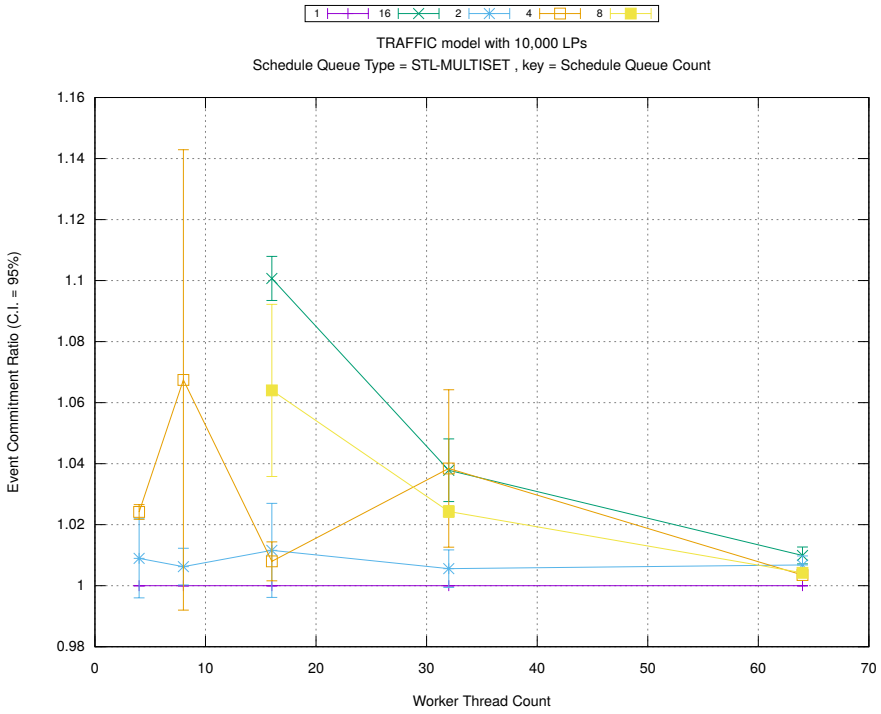
Figure A.10: traffic 10k/plots/scheduleq/threads vs count key type ladder-queue 96



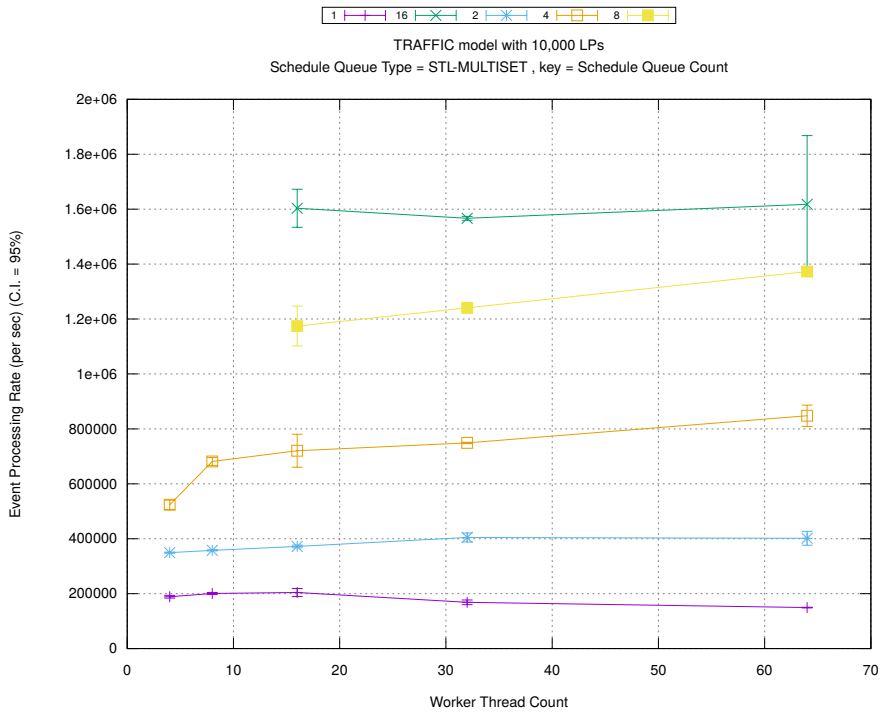
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

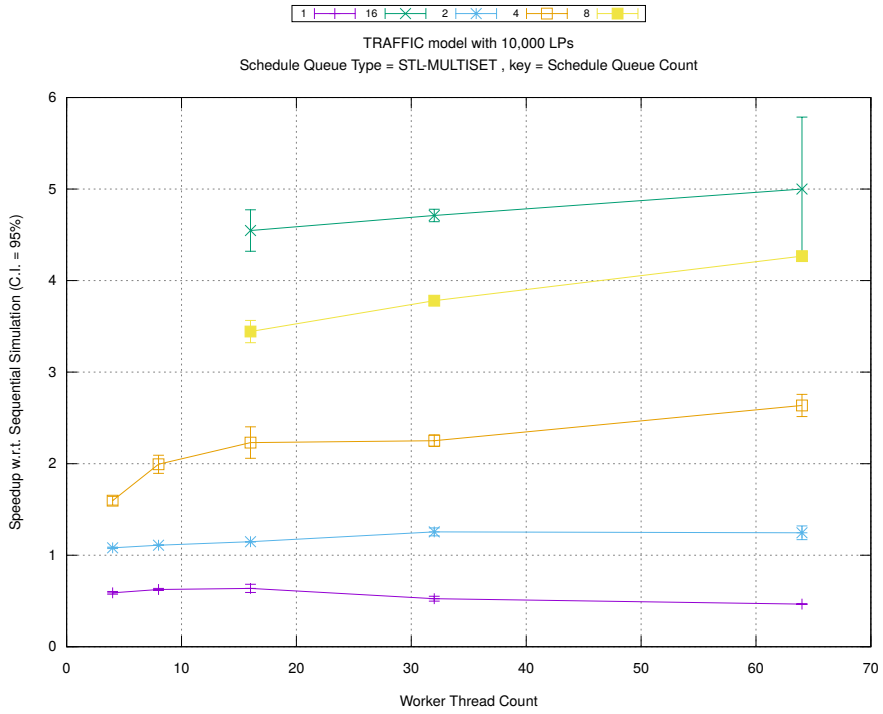


(b) Event Commitment Ratio

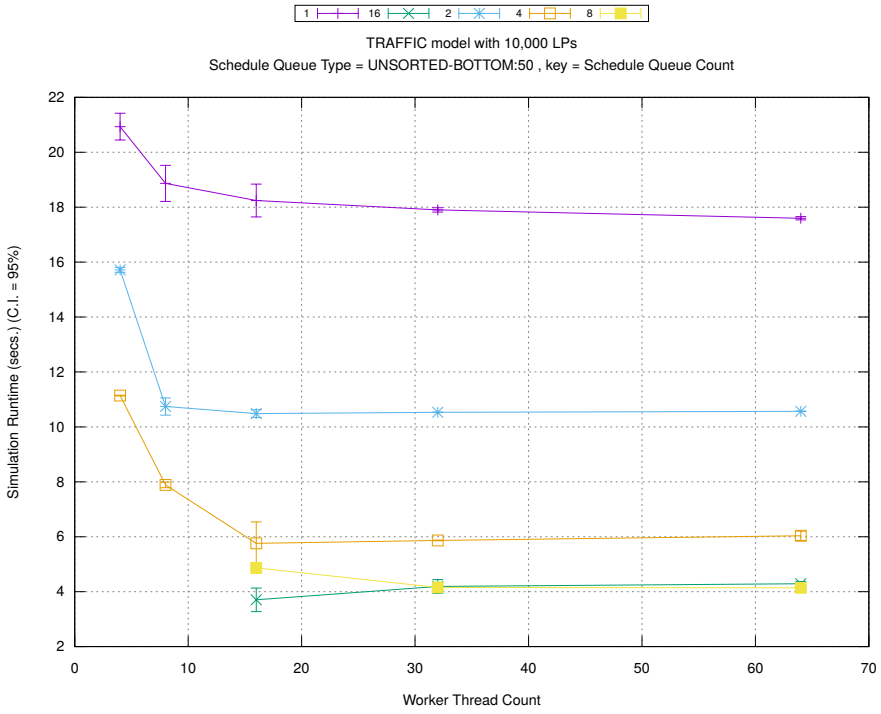


(c) Event Processing Rate (per second)

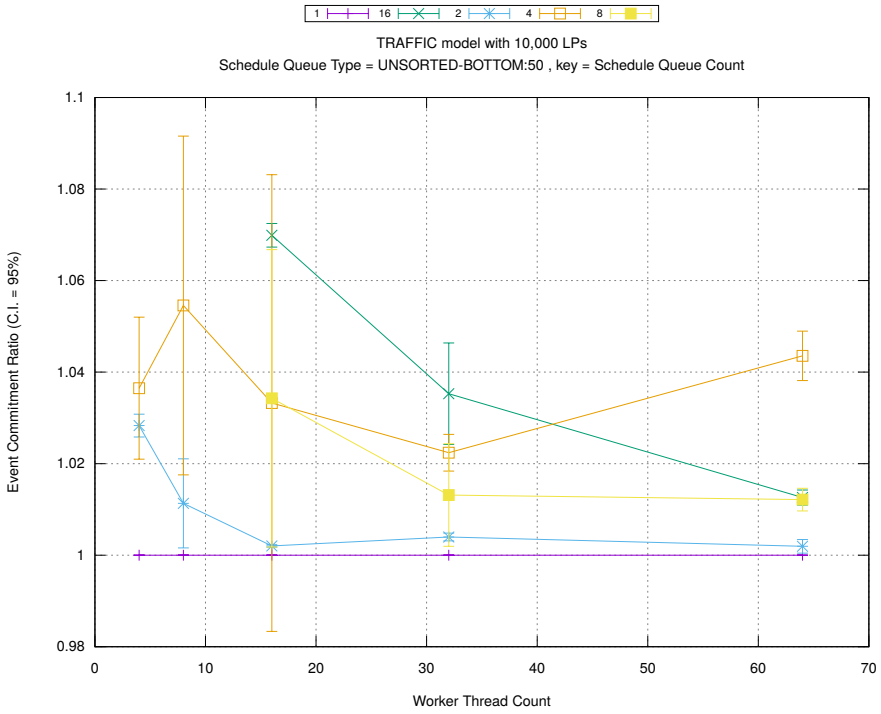
Figure A.11: traffic 10k/plots/scheduleq/threads vs count key type stl-multiset



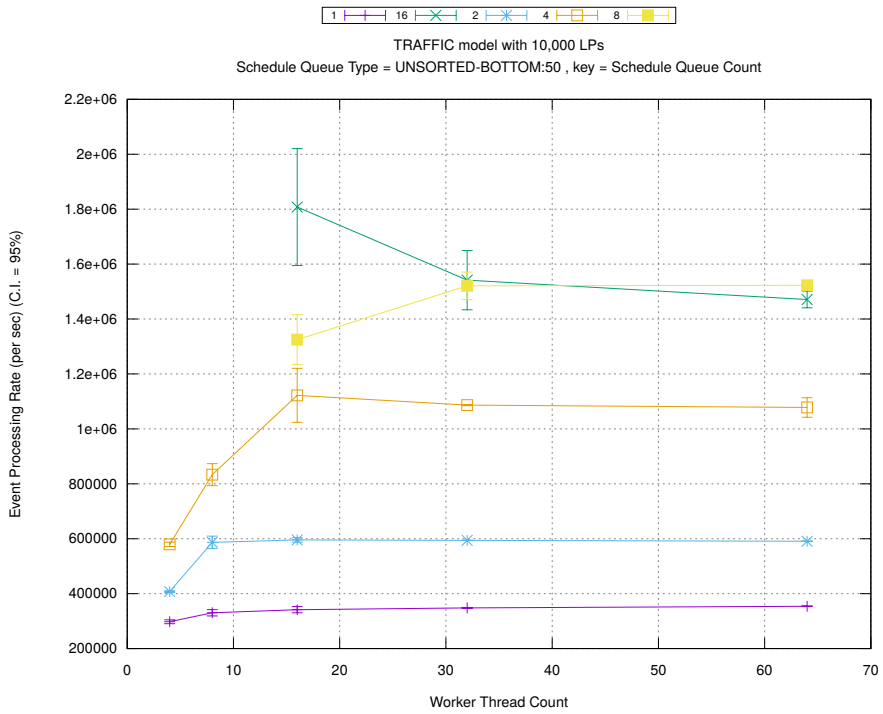
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

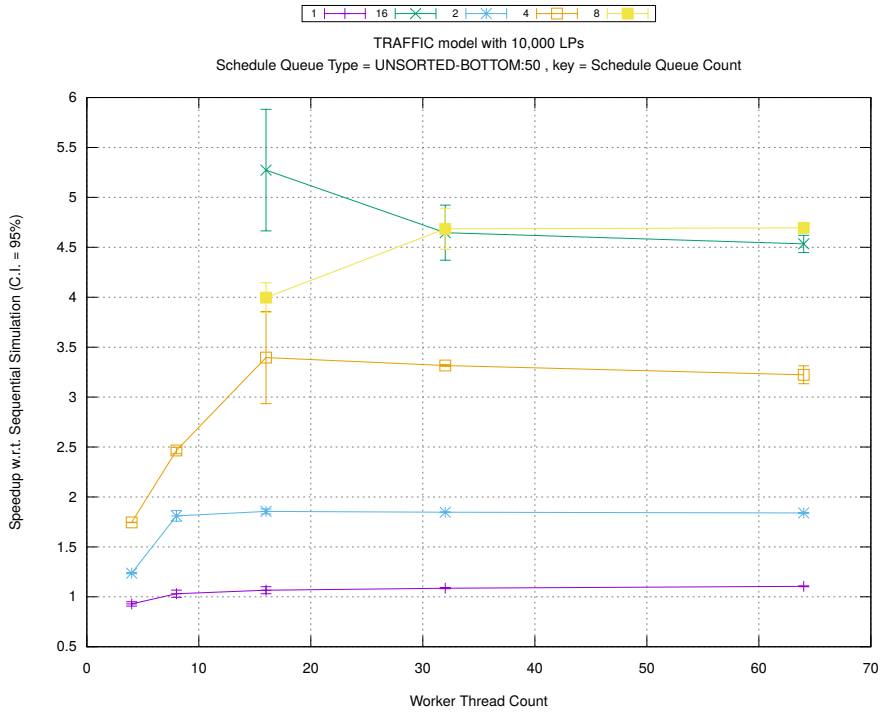


(b) Event Commitment Ratio

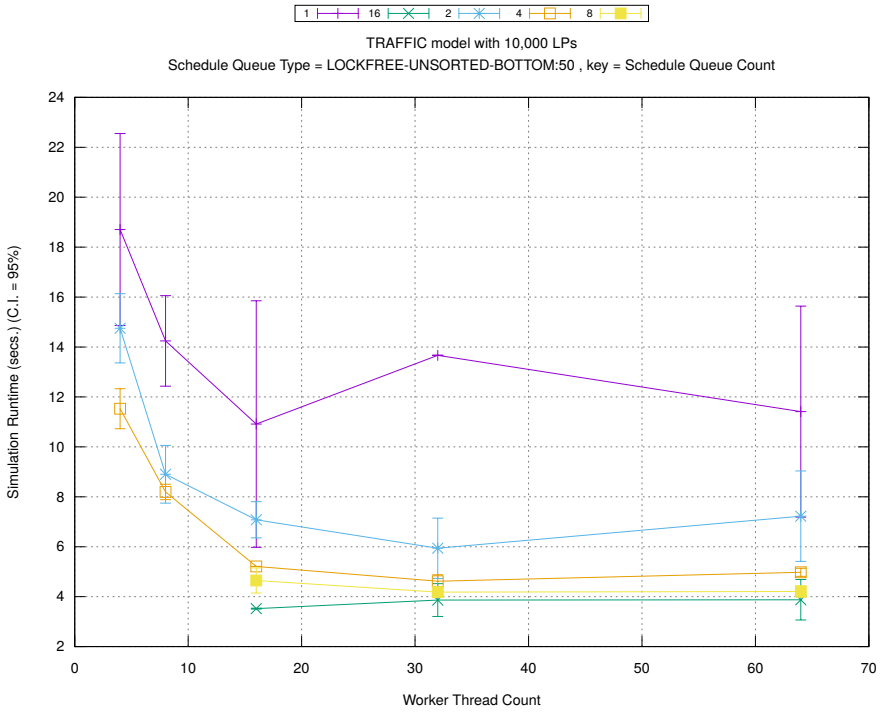


(c) Event Processing Rate (per second)

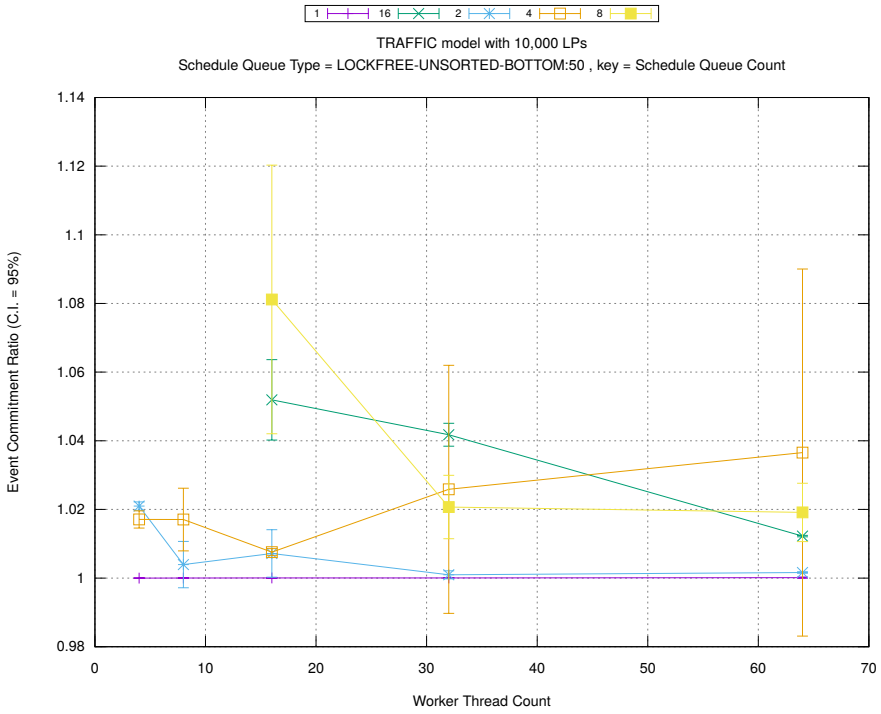
Figure A.12: traffic 10k/plots/scheduleq/threads vs count key type unsorted-bottom 50



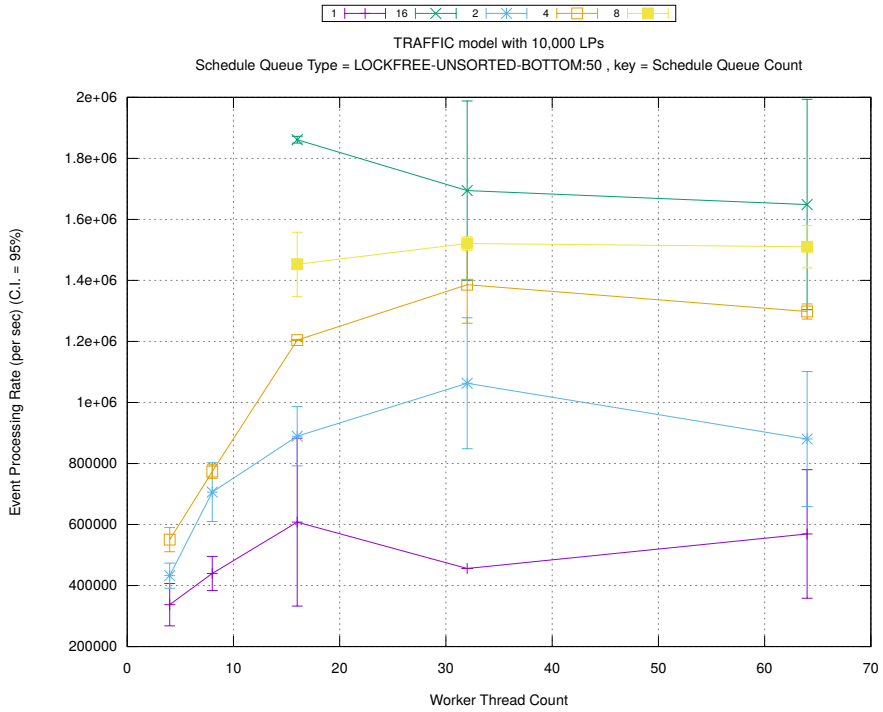
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

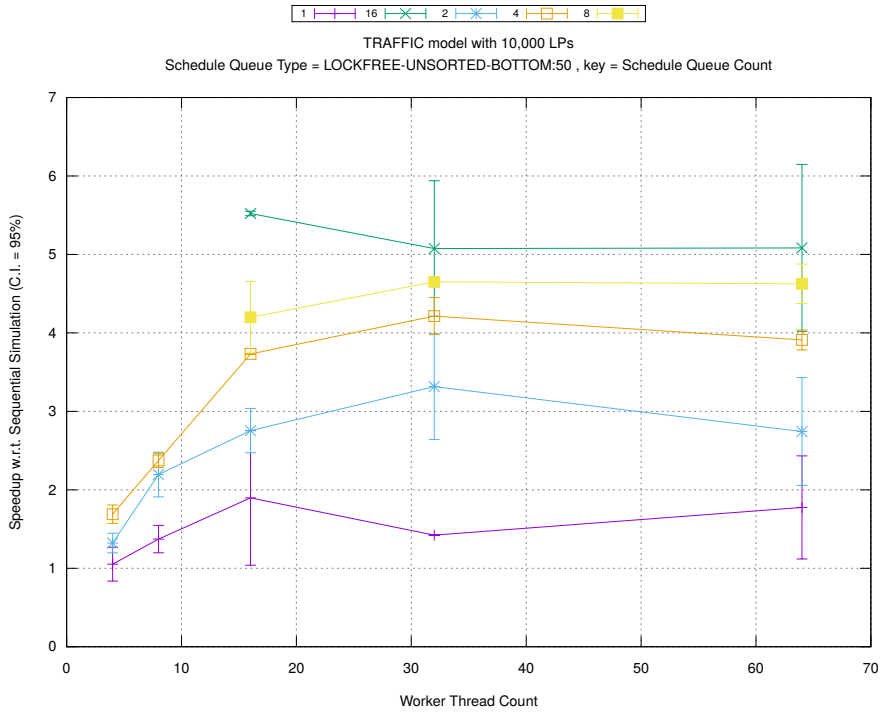


(b) Event Commitment Ratio

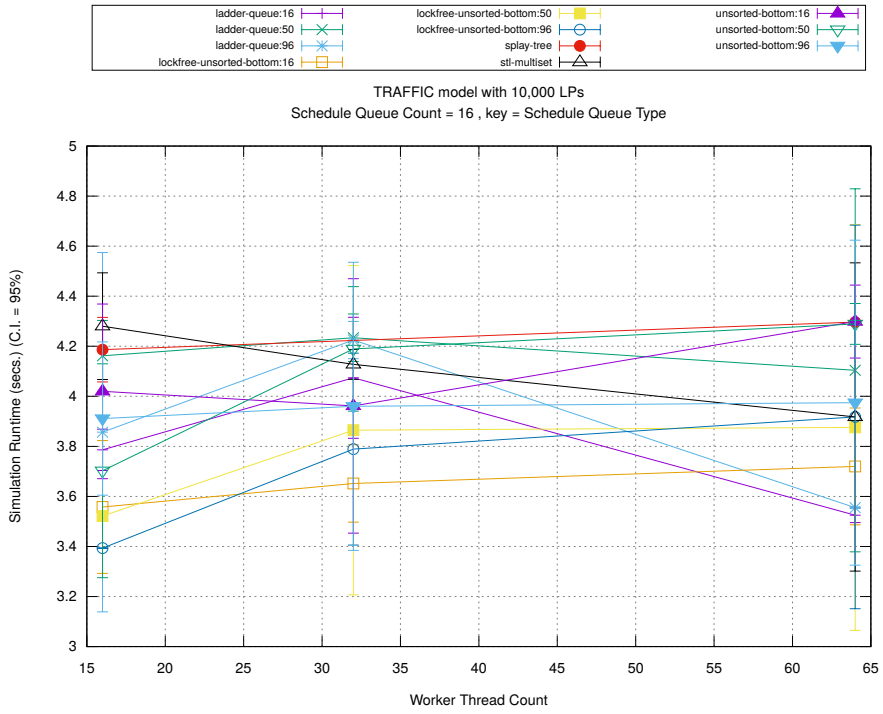


(c) Event Processing Rate (per second)

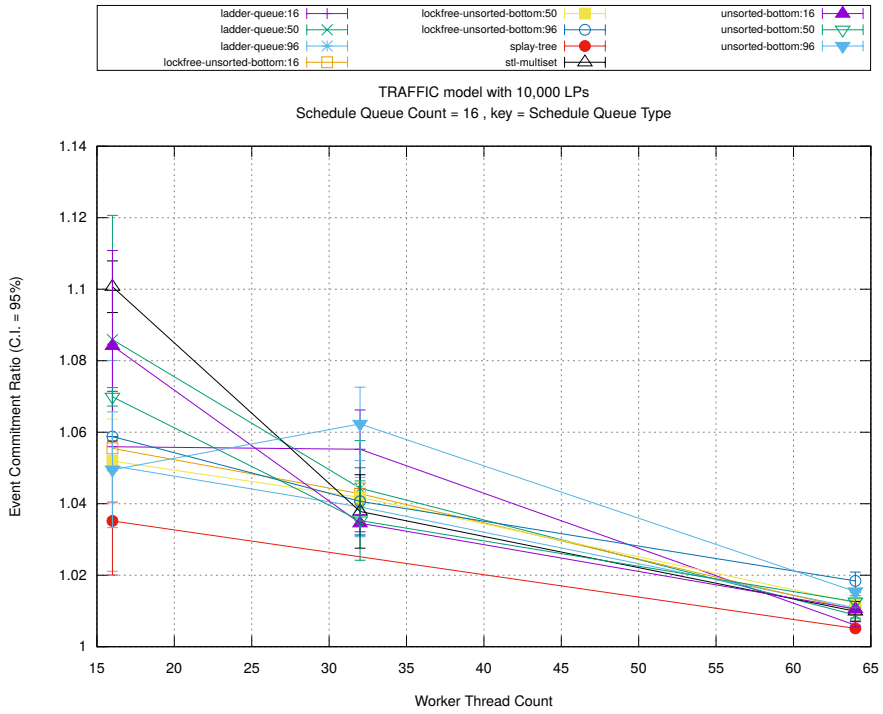
Figure A.13: traffic 10k/plots/scheduleq/threads vs count key type lockfree-unsorted-bottom 50



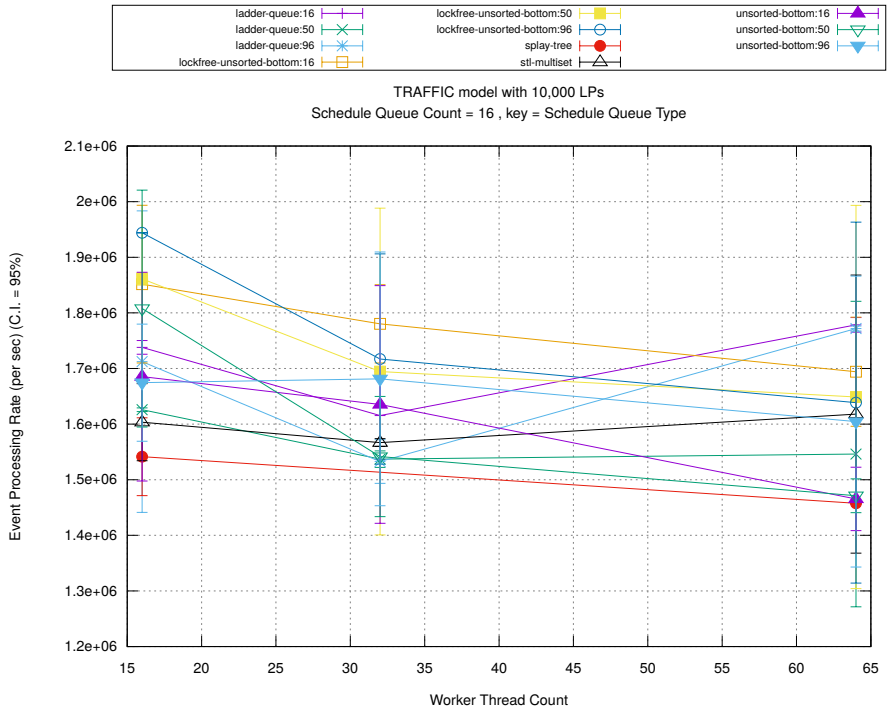
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

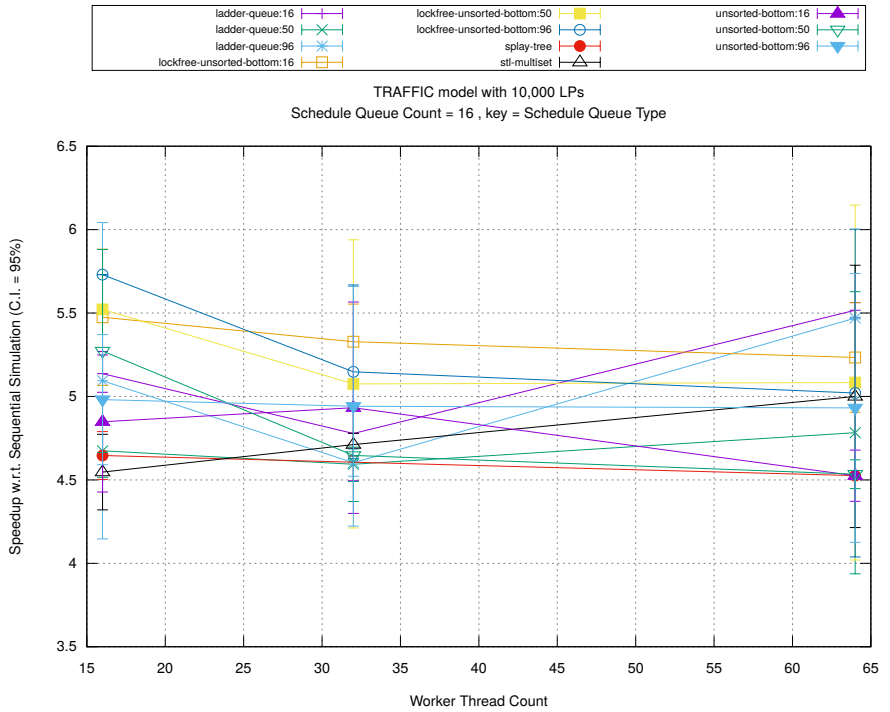


(b) Event Commitment Ratio

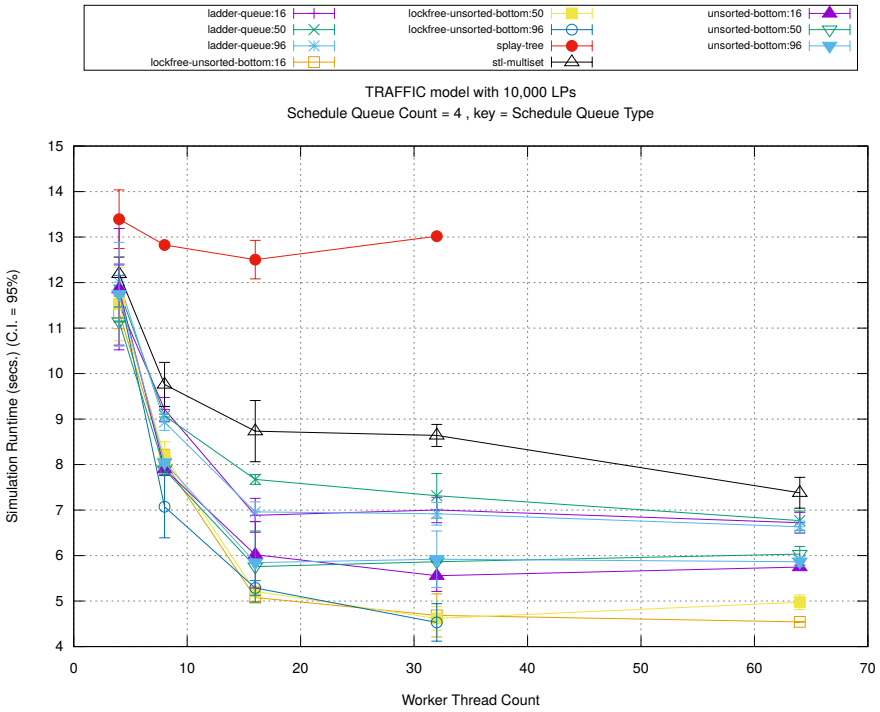


(c) Event Processing Rate (per second)

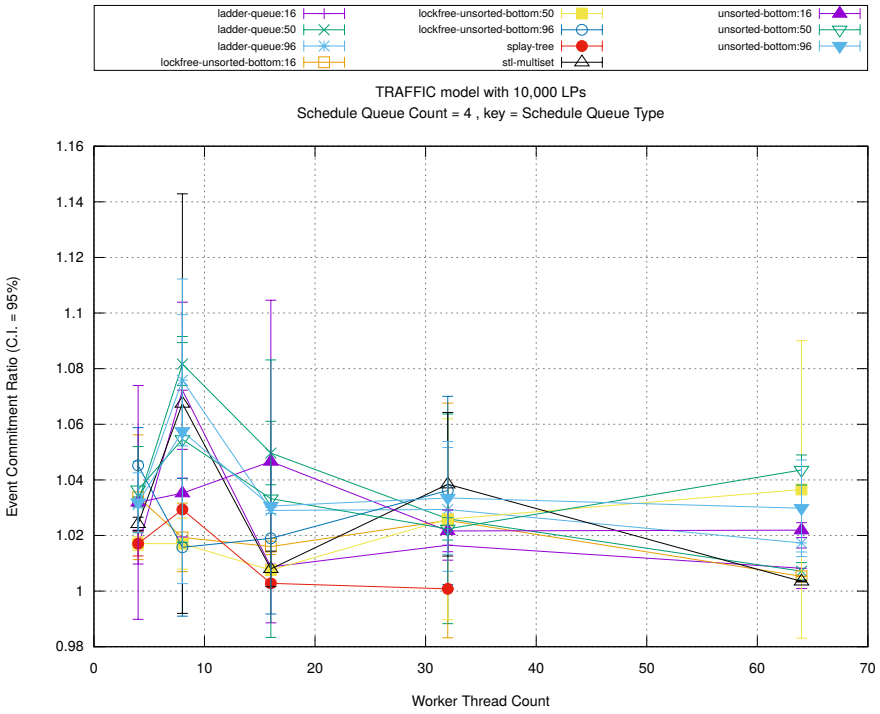
Figure A.14: traffic 10k/plots/scheduleq/threads vs type key count 16



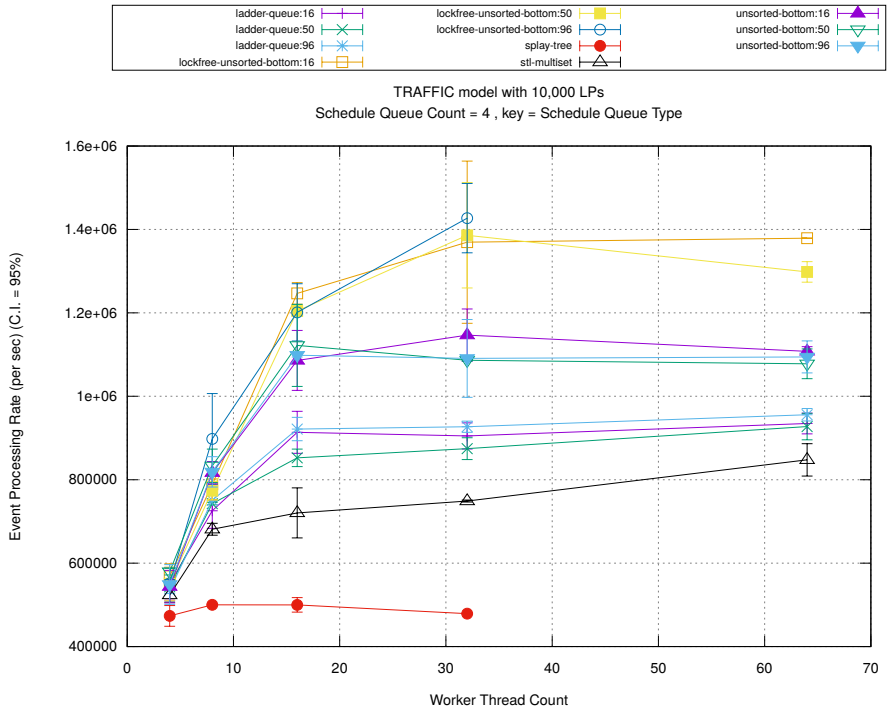
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

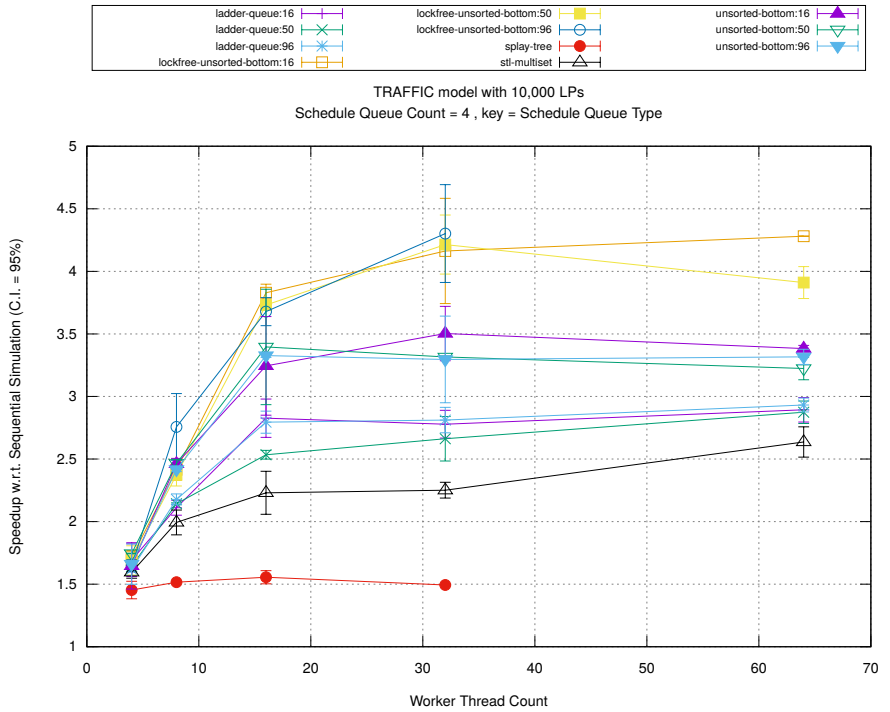


(b) Event Commitment Ratio

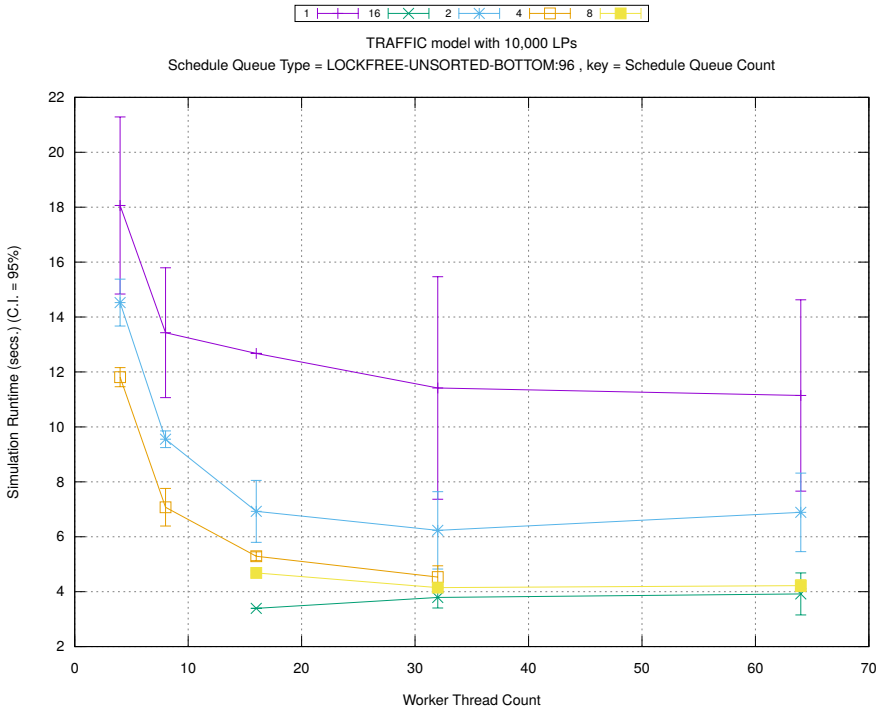


(c) Event Processing Rate (per second)

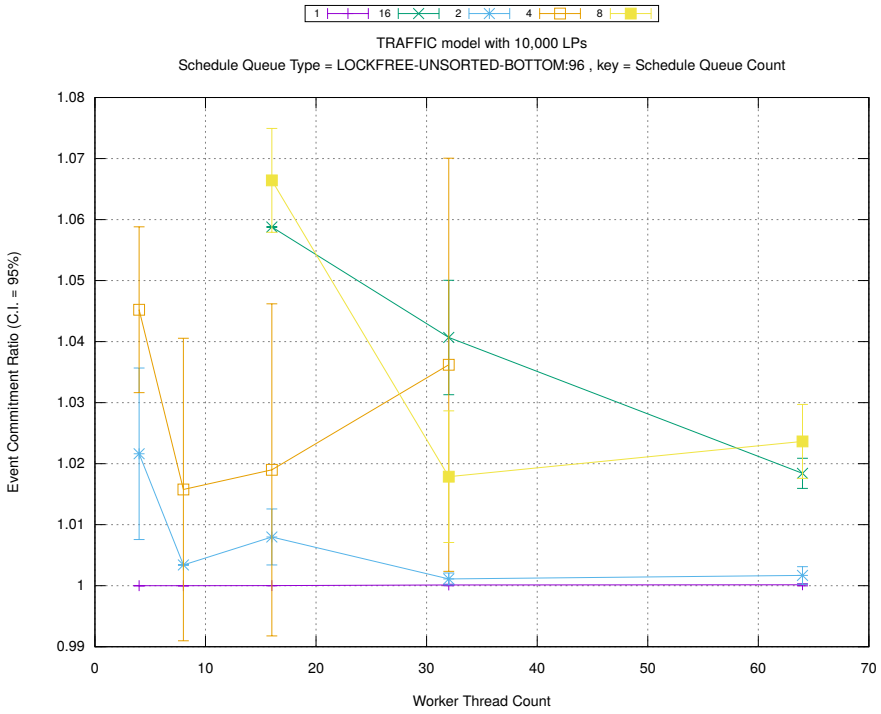
Figure A.15: traffic 10k/plots/scheduleq/threads vs type key count 4



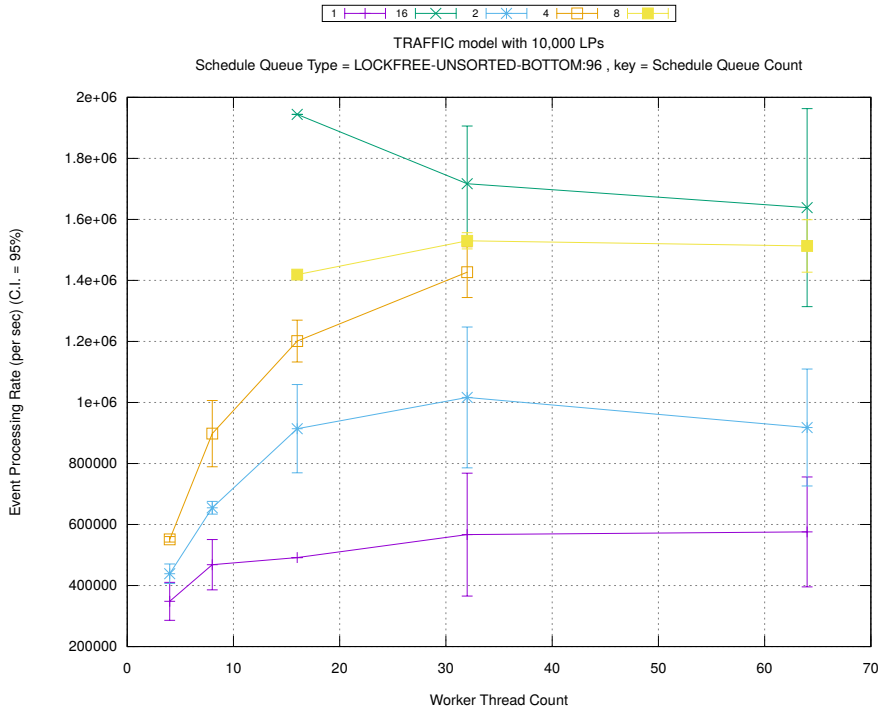
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

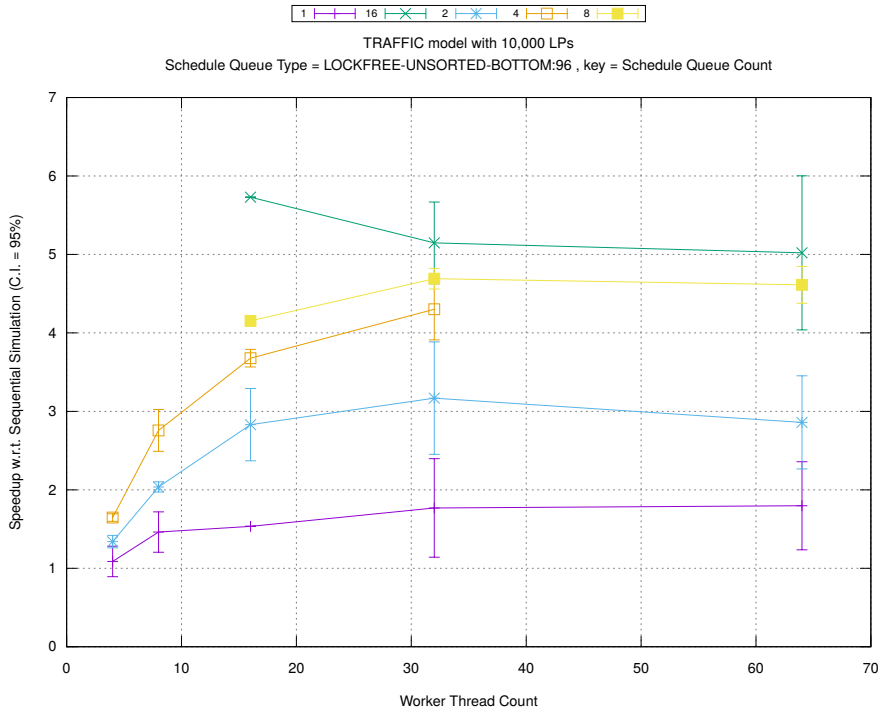


(b) Event Commitment Ratio

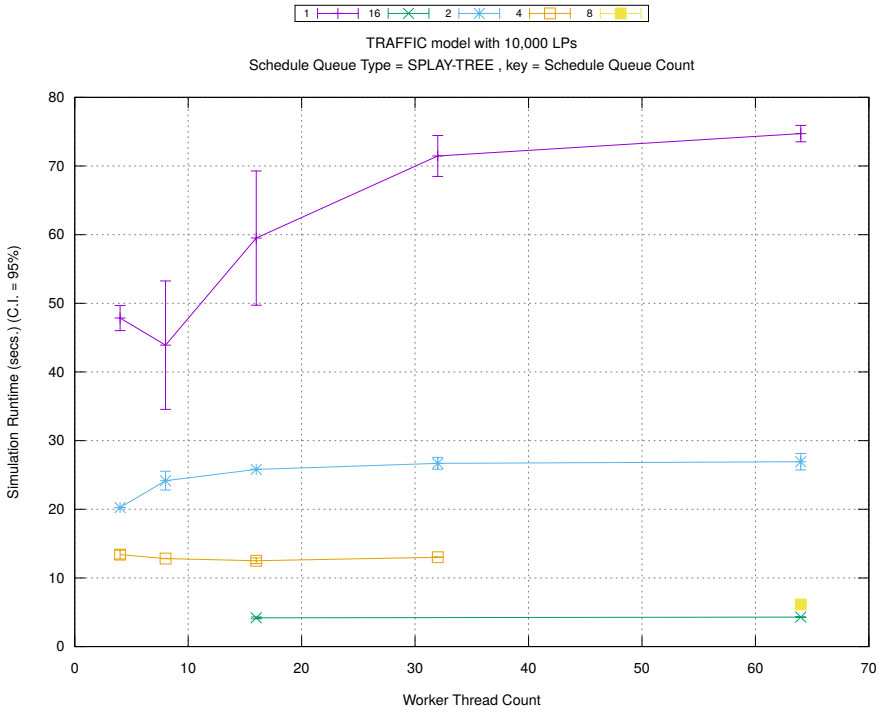


(c) Event Processing Rate (per second)

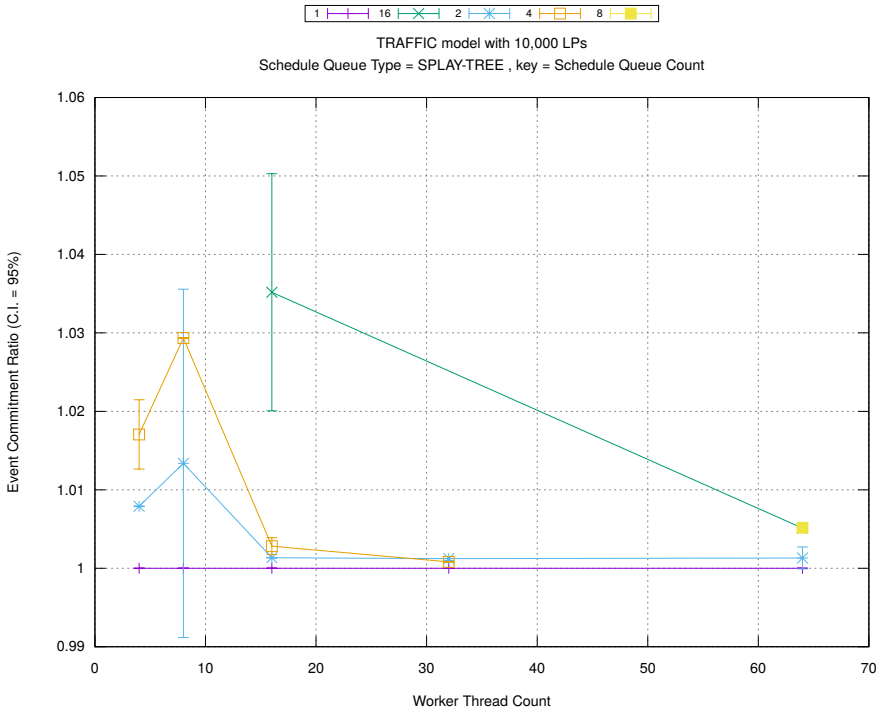
Figure A.16: traffic 10k/plots/scheduleq/threads vs count key type lockfree-unsorted-bottom 96



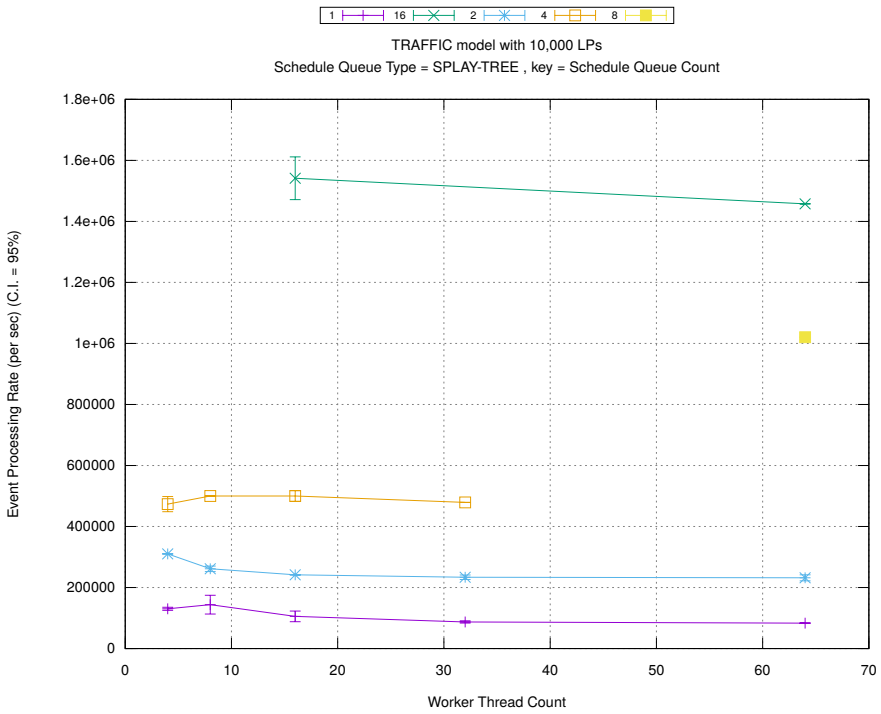
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

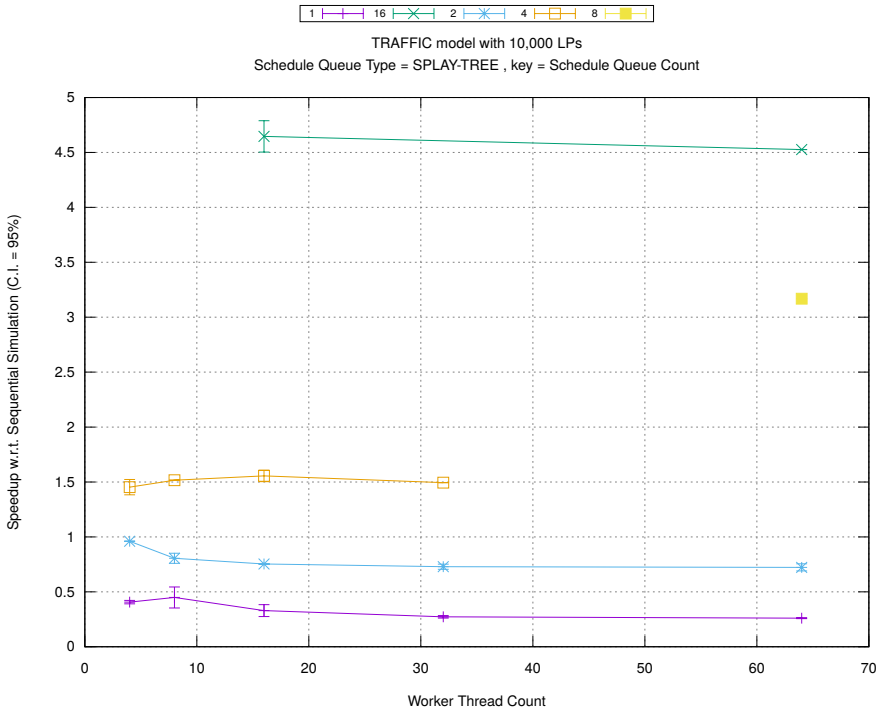


(b) Event Commitment Ratio

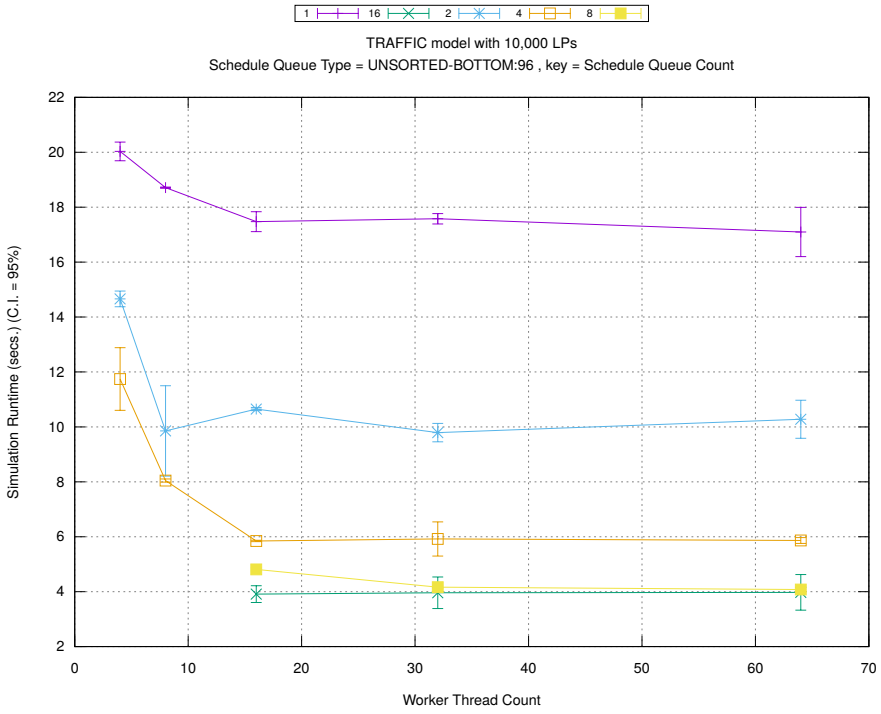


(c) Event Processing Rate (per second)

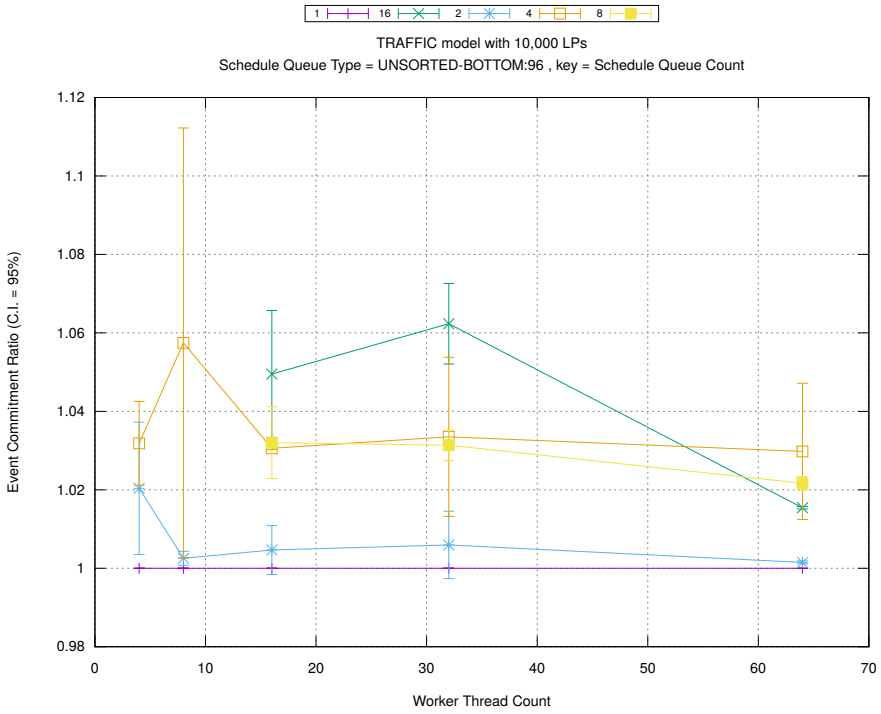
Figure A.17: traffic 10k/plots/scheduleq/threads vs count key type splay-tree



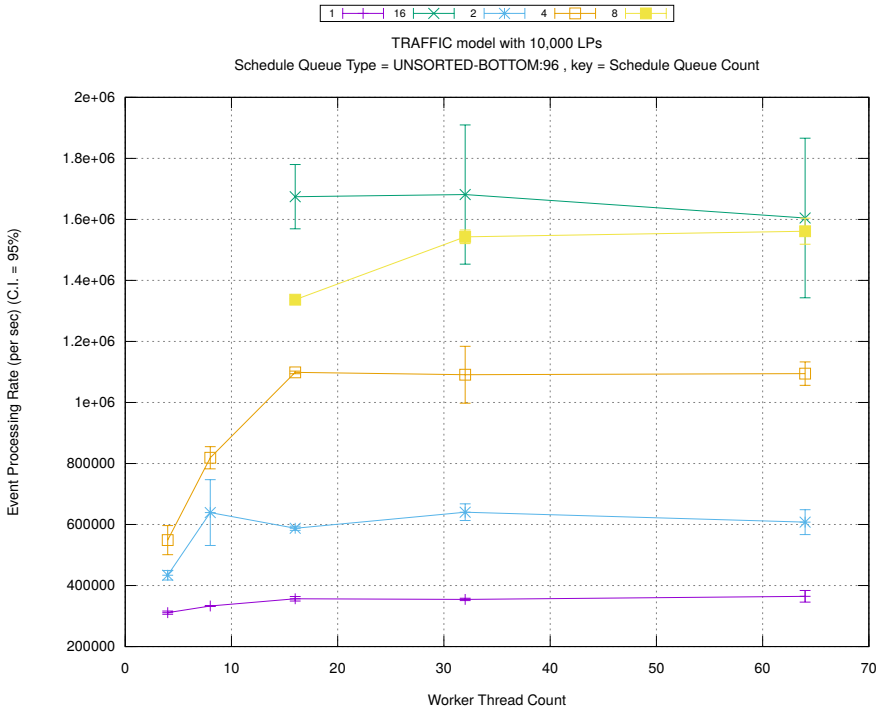
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

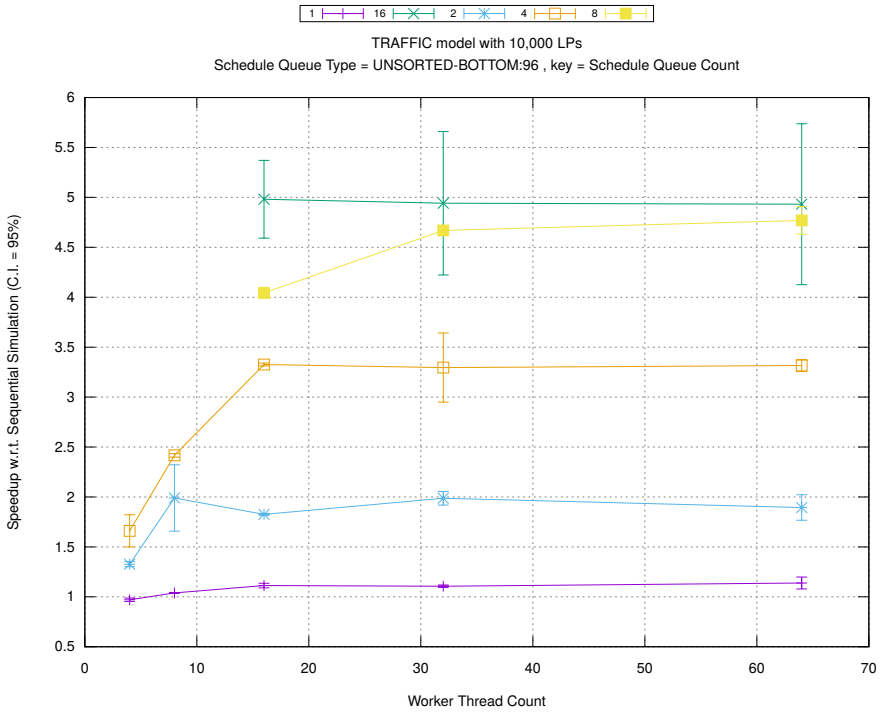


(b) Event Commitment Ratio

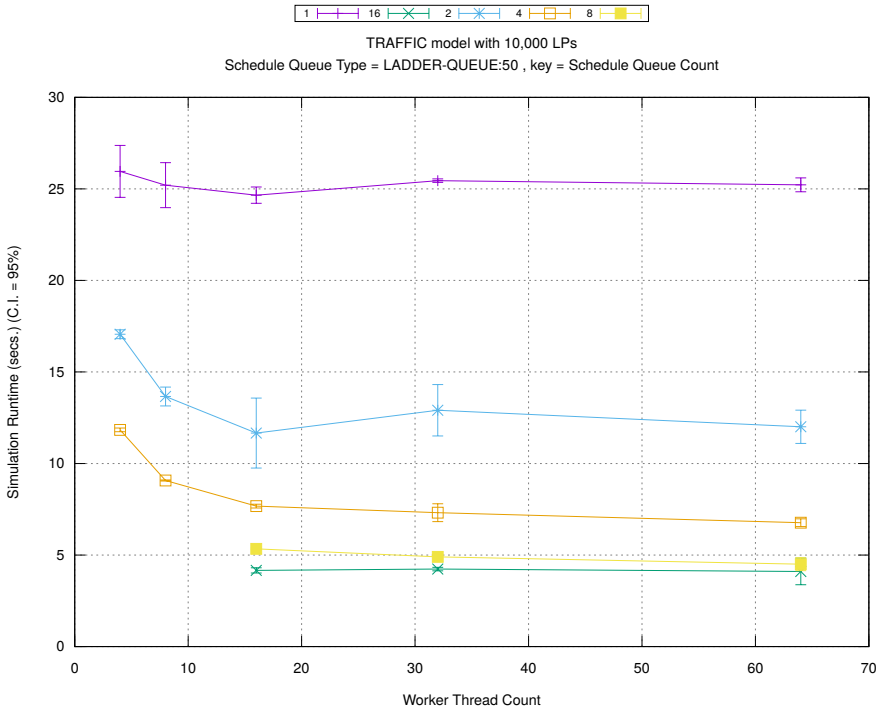


(c) Event Processing Rate (per second)

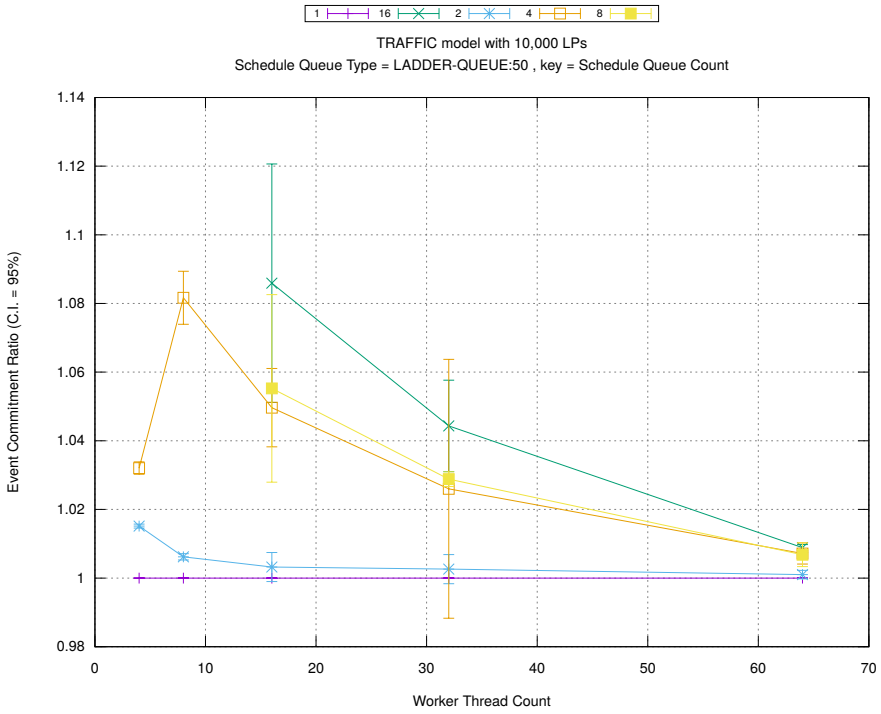
Figure A.18: traffic 10k/plots/scheduleq/threads vs count key type unsorted-bottom 96



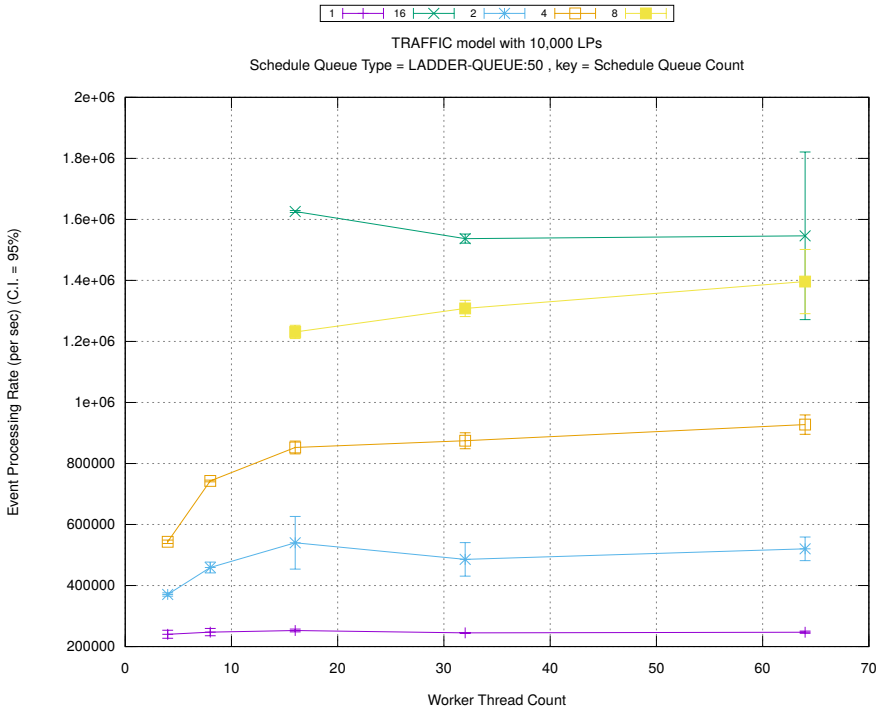
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

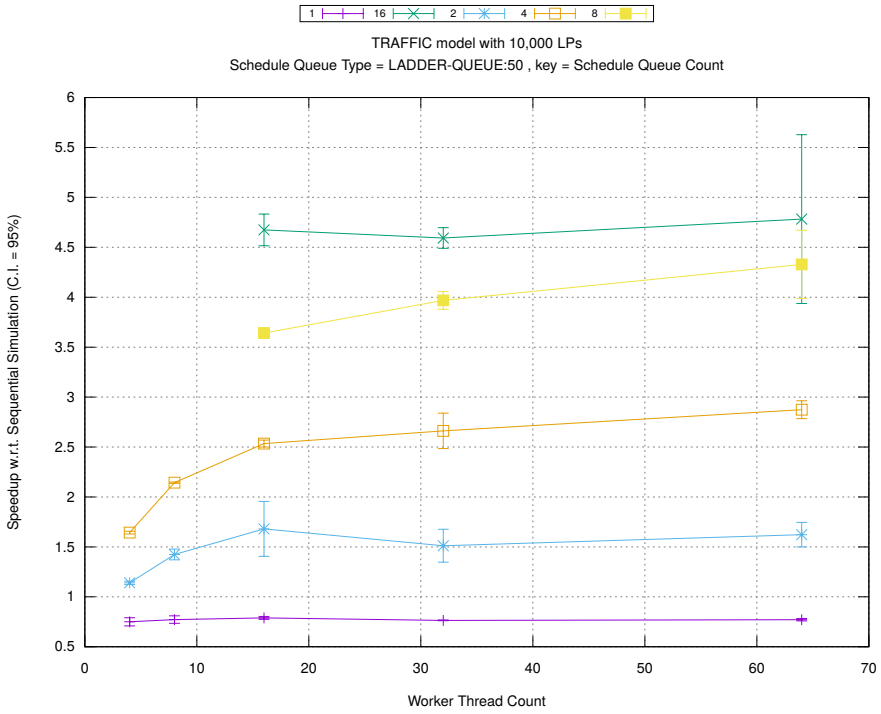


(b) Event Commitment Ratio

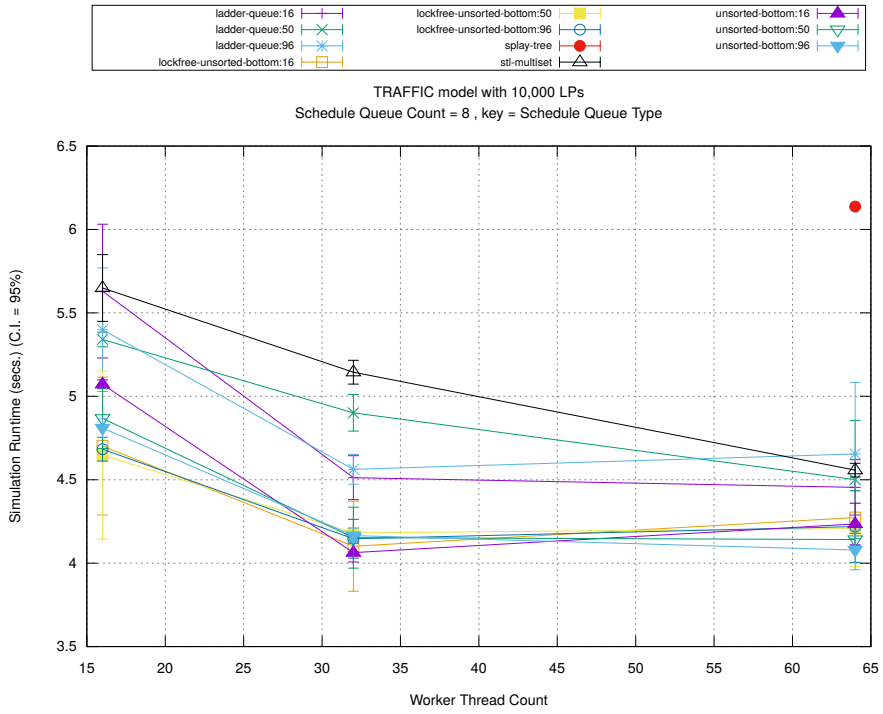


(c) Event Processing Rate (per second)

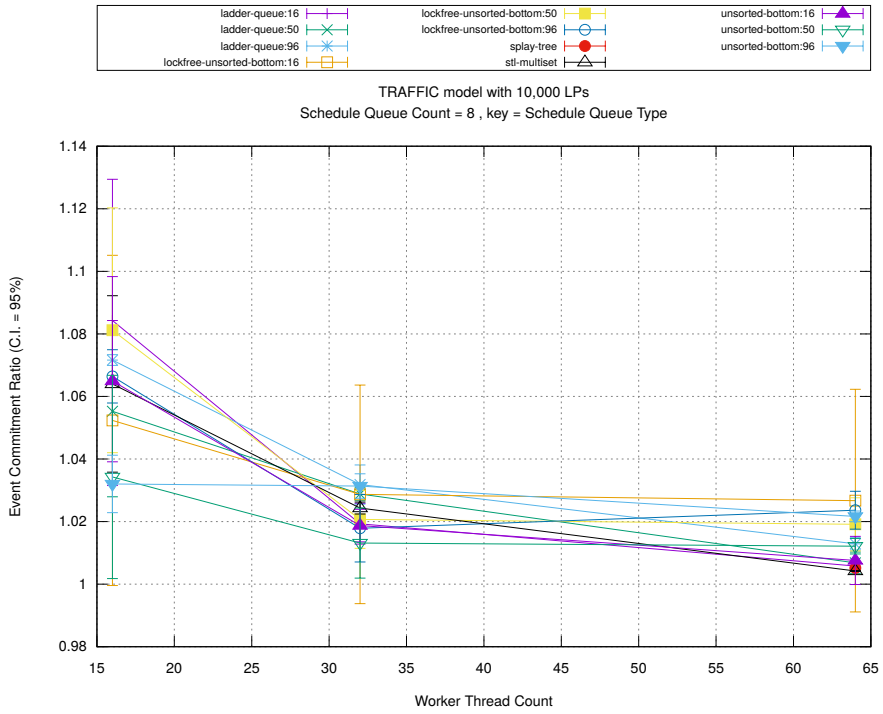
Figure A.19: traffic 10k/plots/scheduleq/threads vs count key type ladder-queue 50



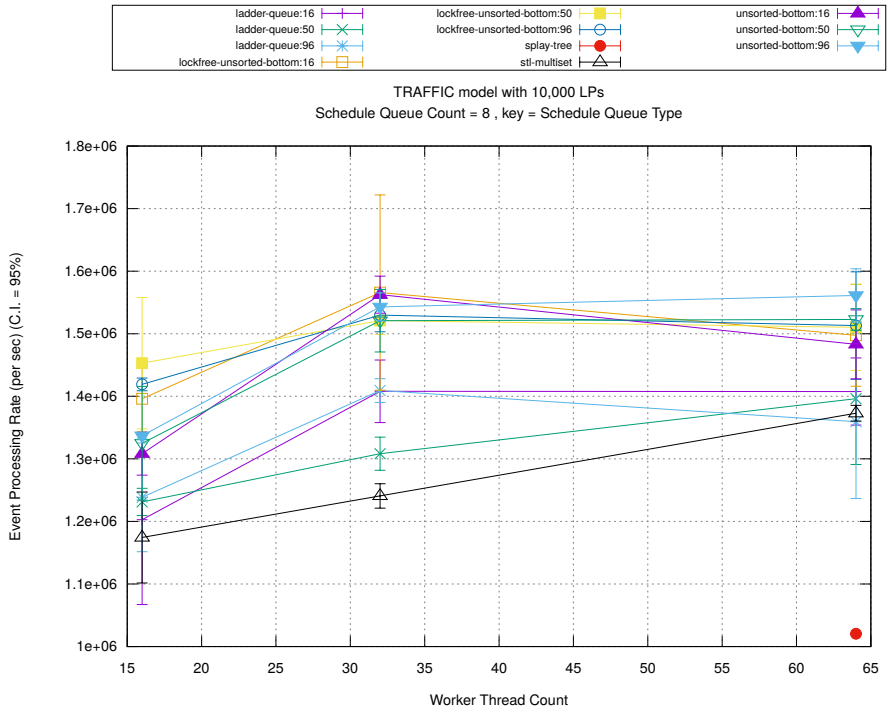
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

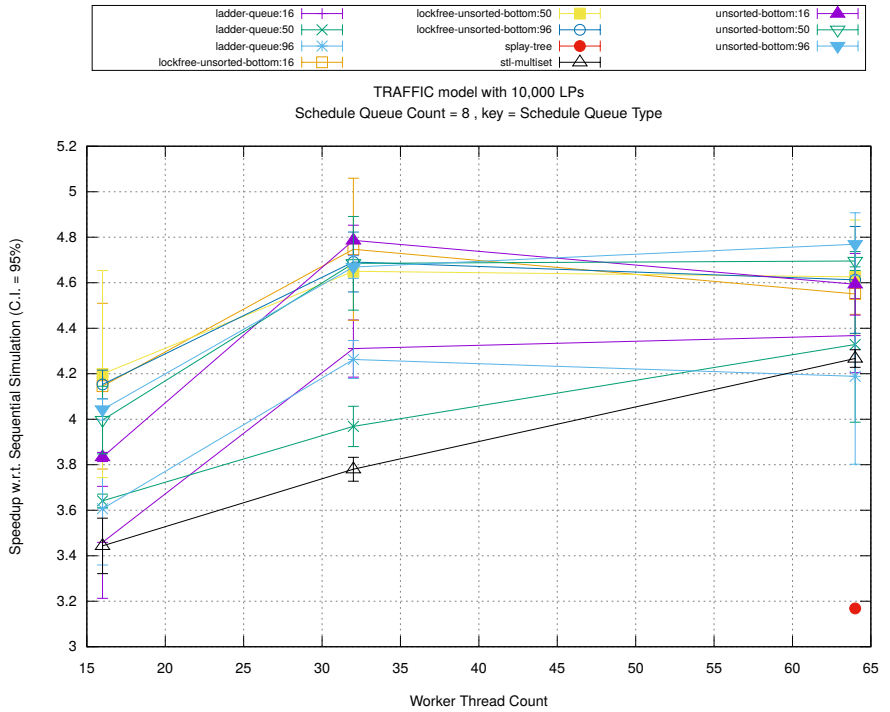


(b) Event Commitment Ratio

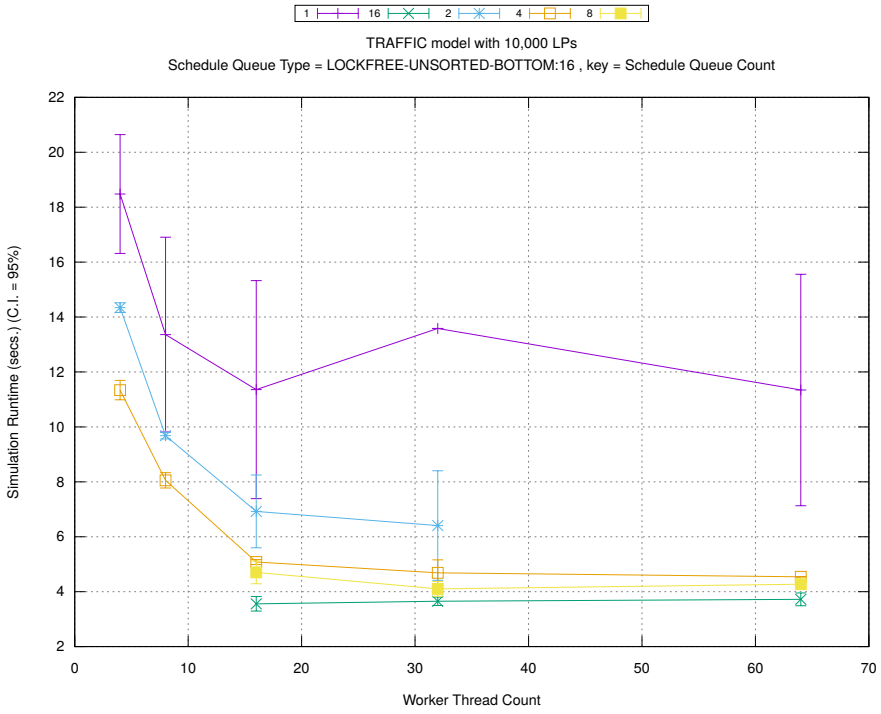


(c) Event Processing Rate (per second)

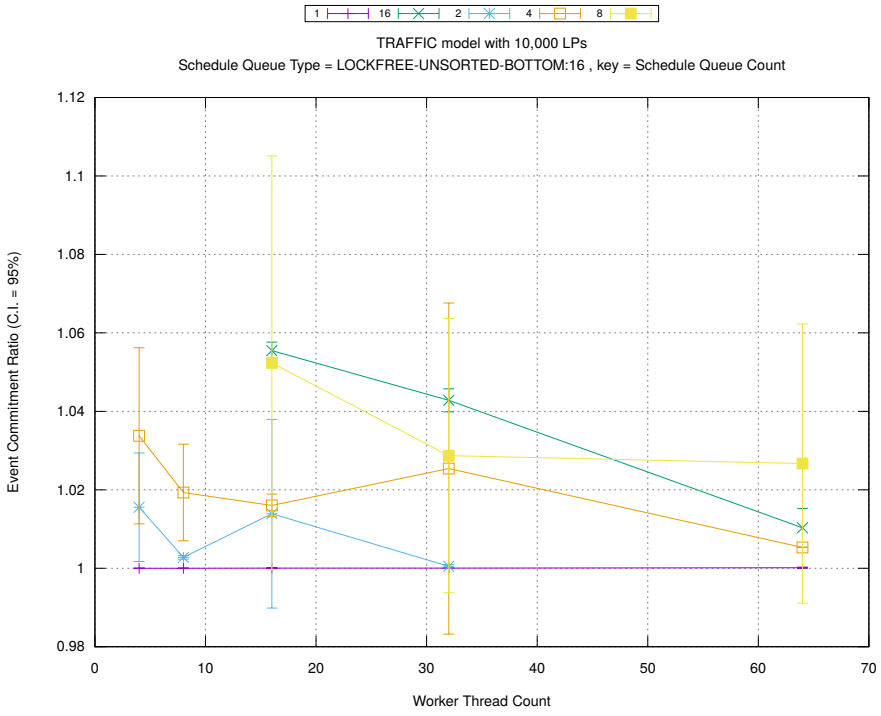
Figure A.20: traffic 10k/plots/scheduleq/threads vs type key count 8



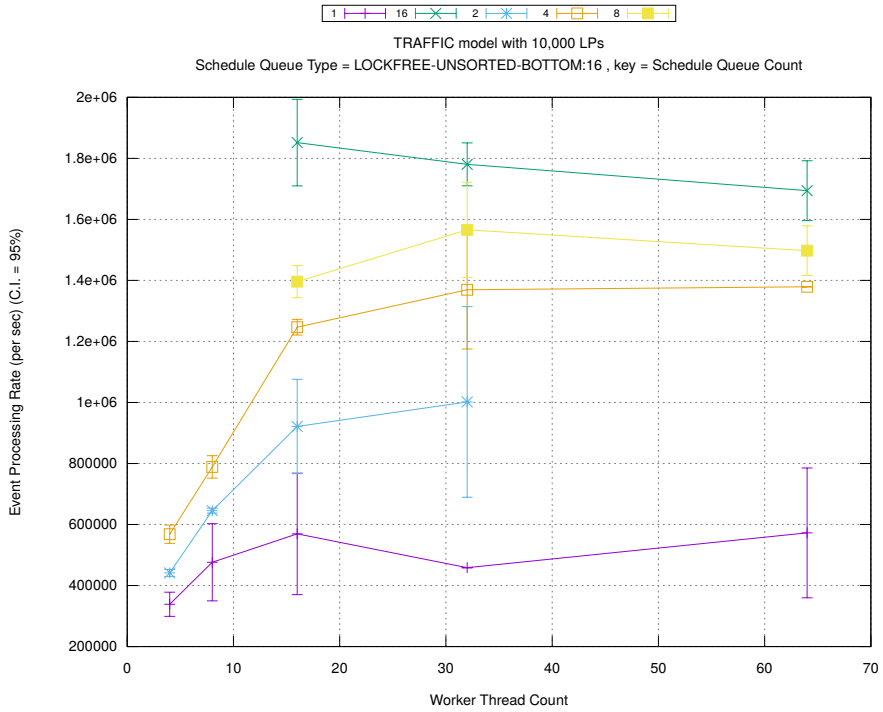
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

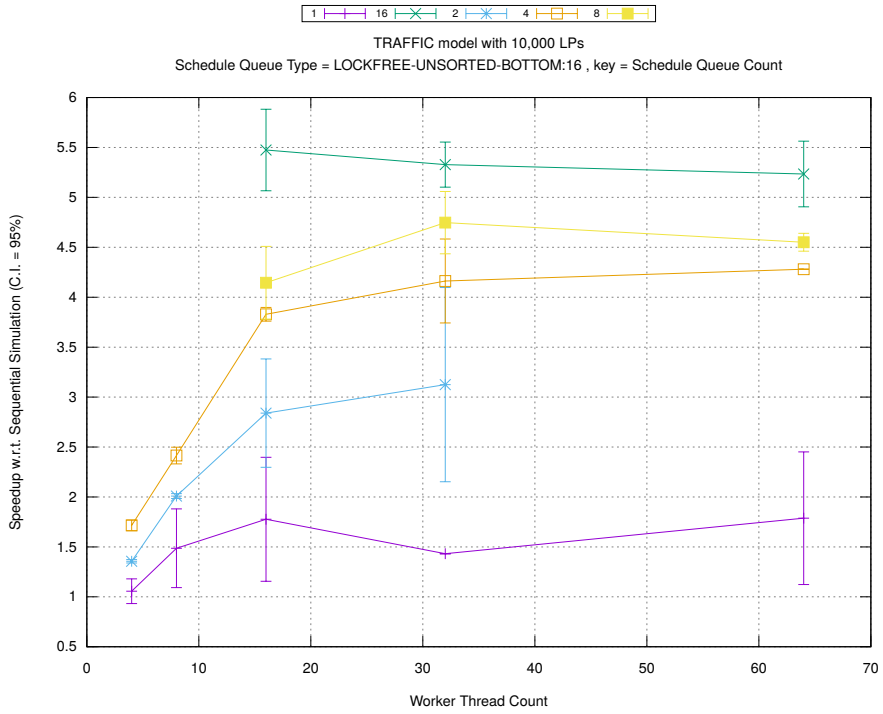


(b) Event Commitment Ratio

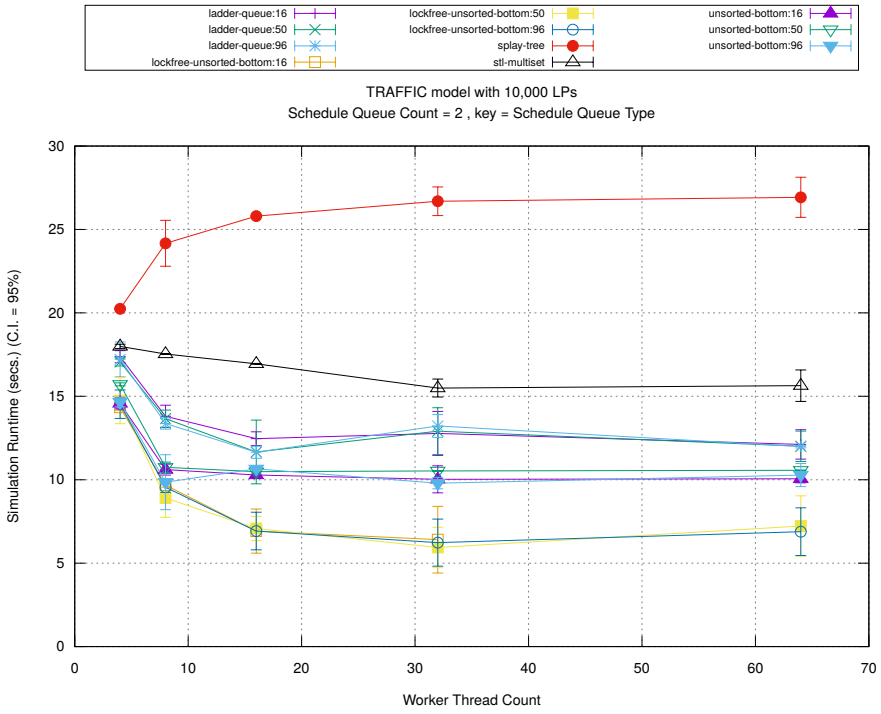


(c) Event Processing Rate (per second)

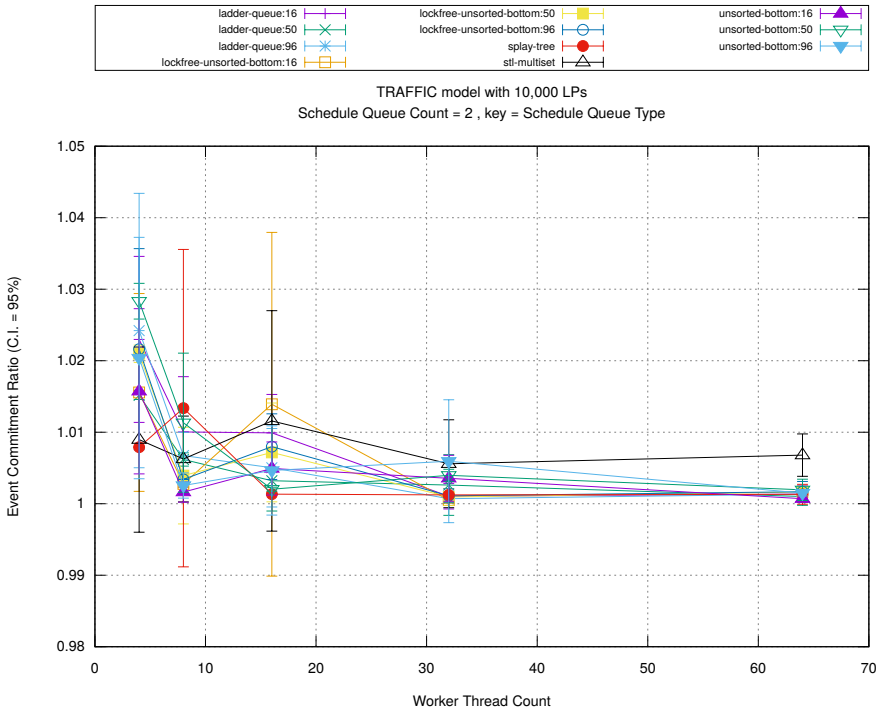
Figure A.21: traffic 10k/plots/scheduleq/threads vs count key type lockfree-unsorted-bottom 16



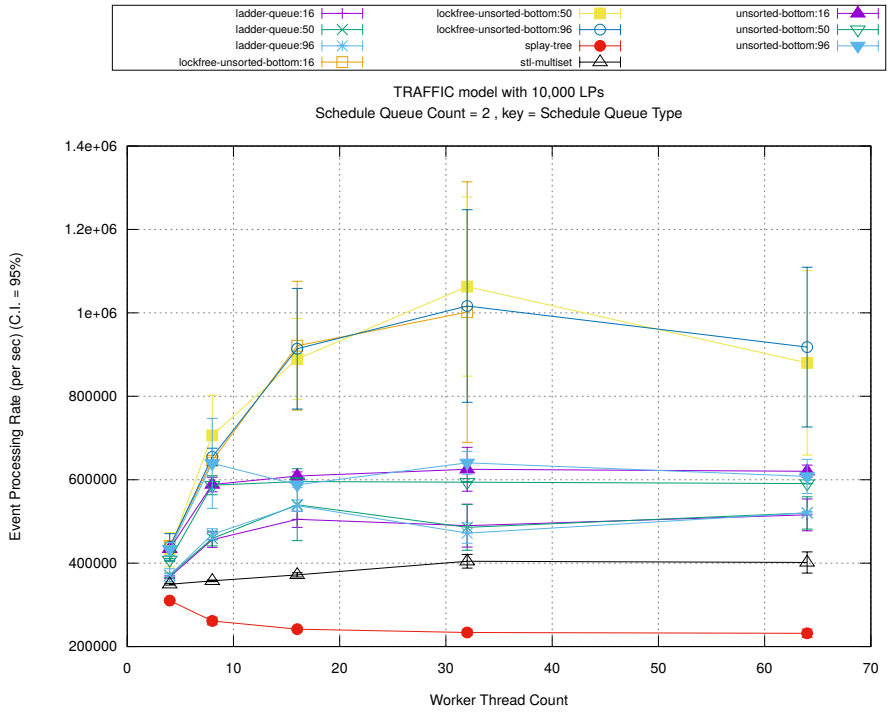
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

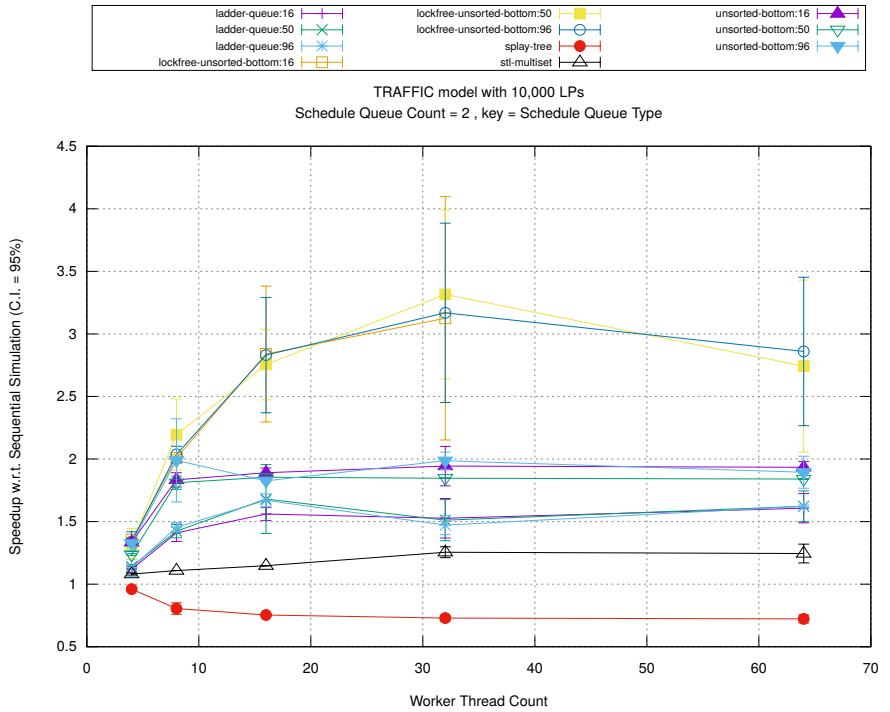


(b) Event Commitment Ratio

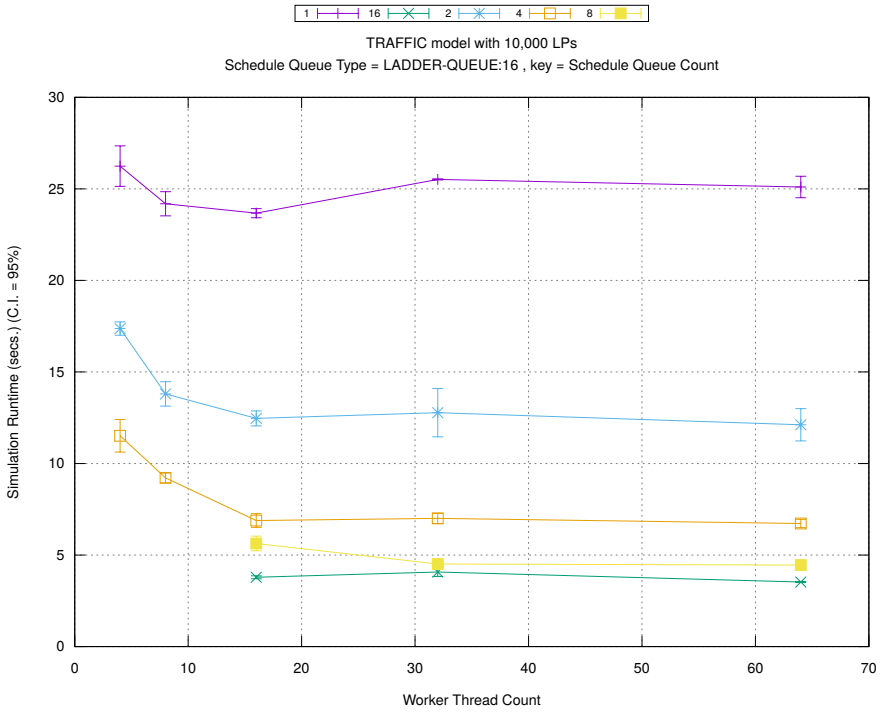


(c) Event Processing Rate (per second)

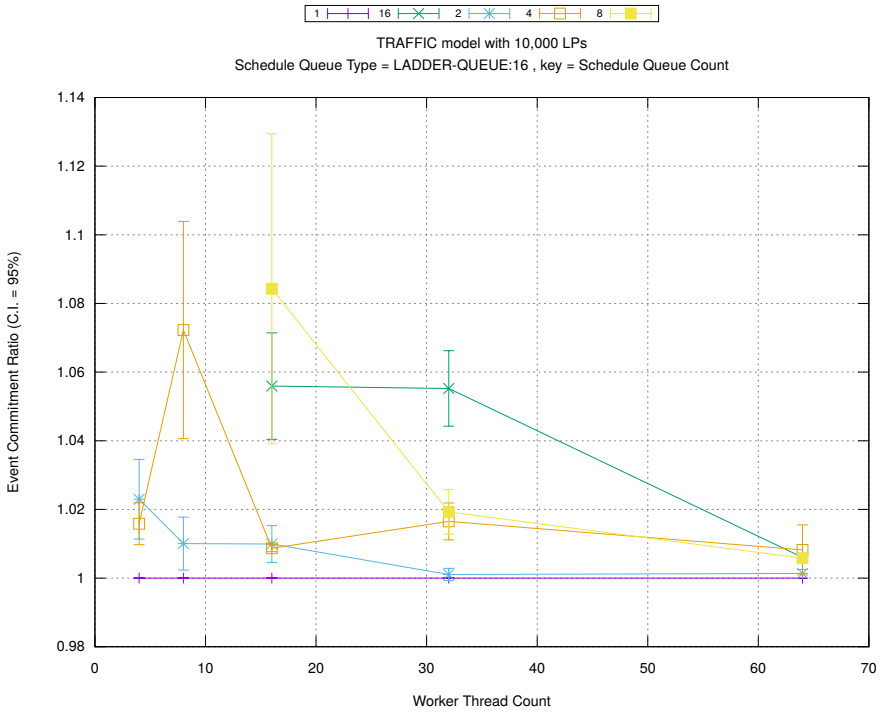
Figure A.22: traffic 10k/plots/scheduleq/threads vs type key count 2



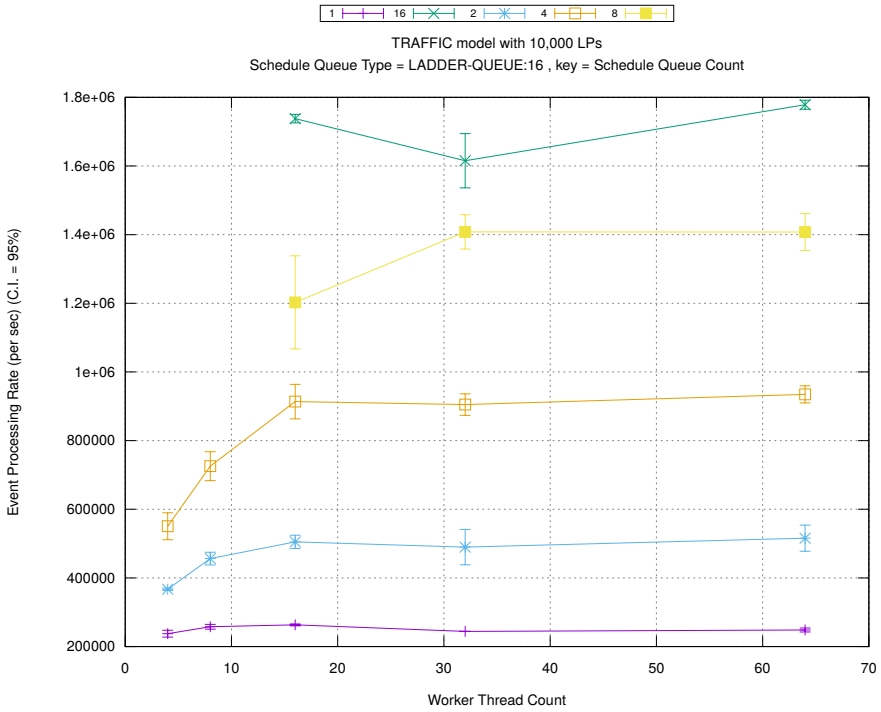
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

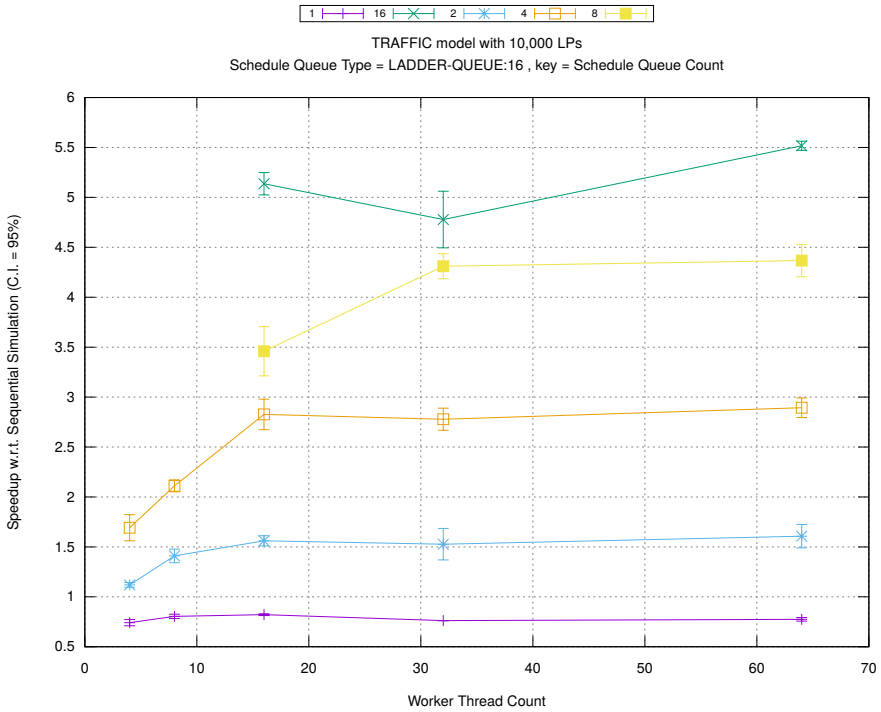


(b) Event Commitment Ratio

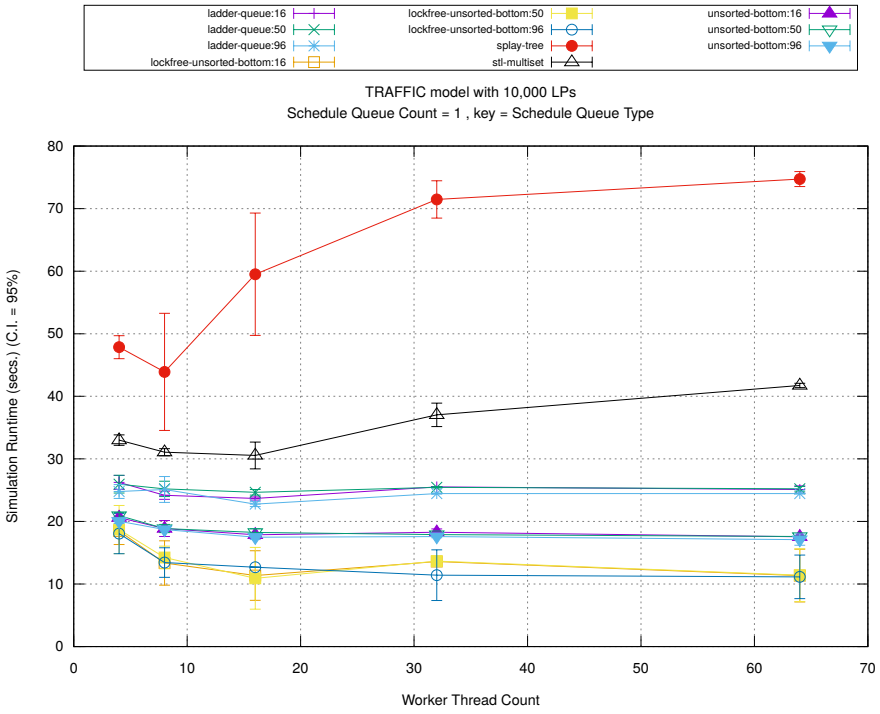


(c) Event Processing Rate (per second)

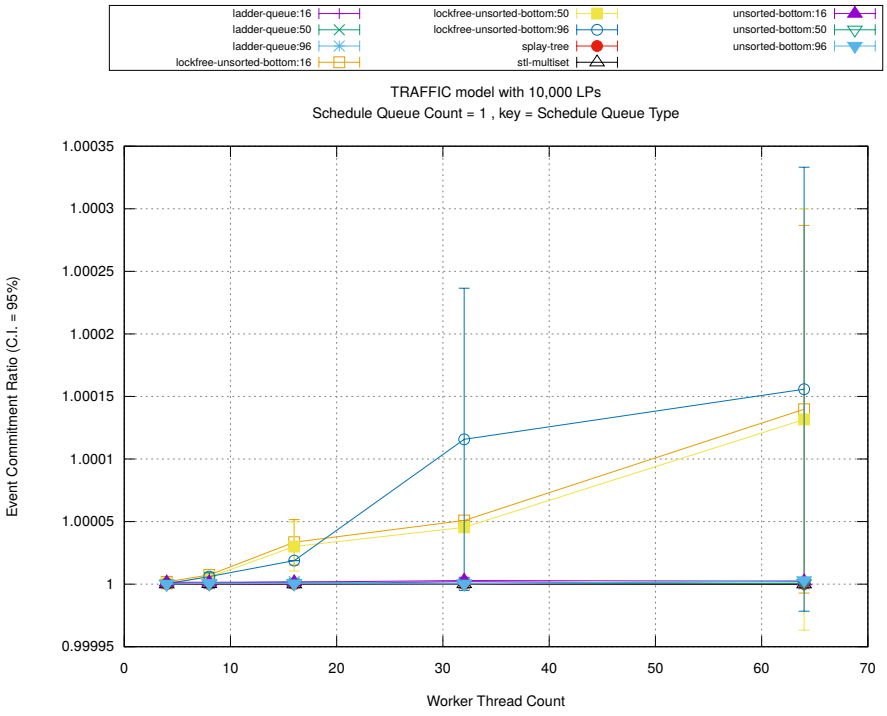
Figure A.23: traffic 10k/plots/scheduleq/threads vs count key type ladder-queue 16



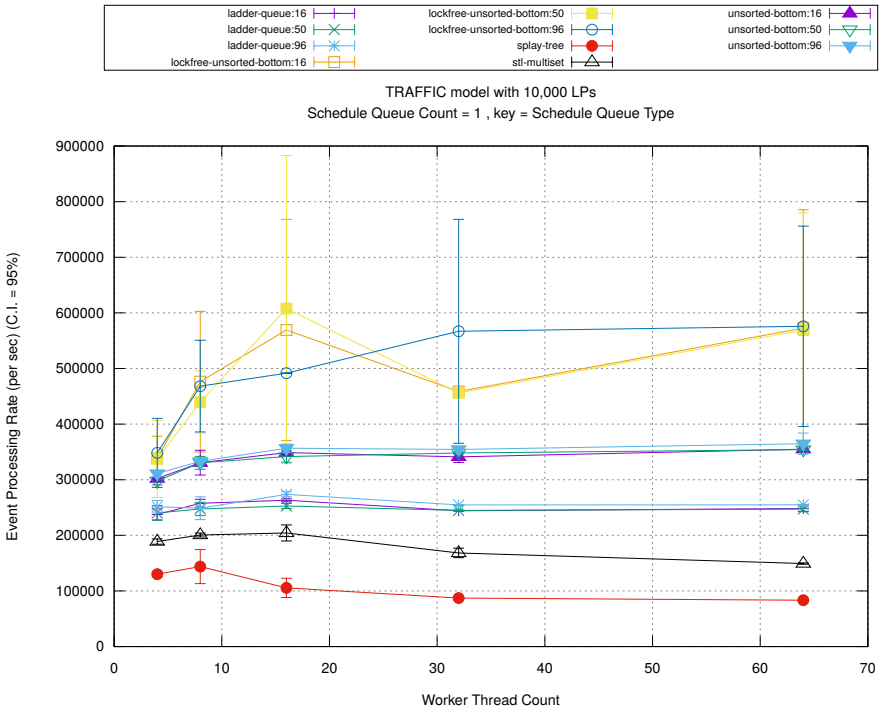
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

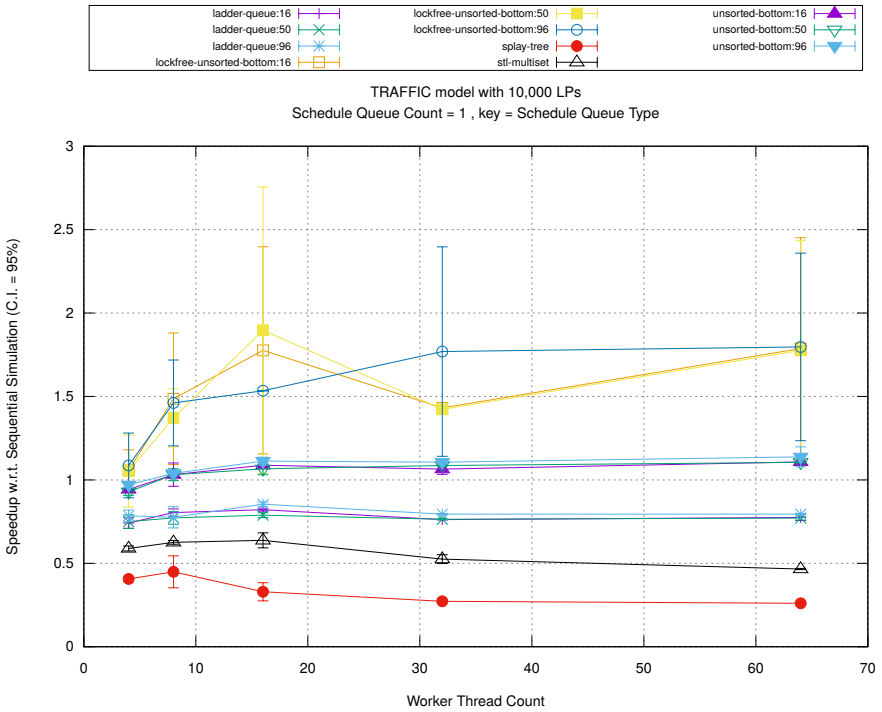


(b) Event Commitment Ratio

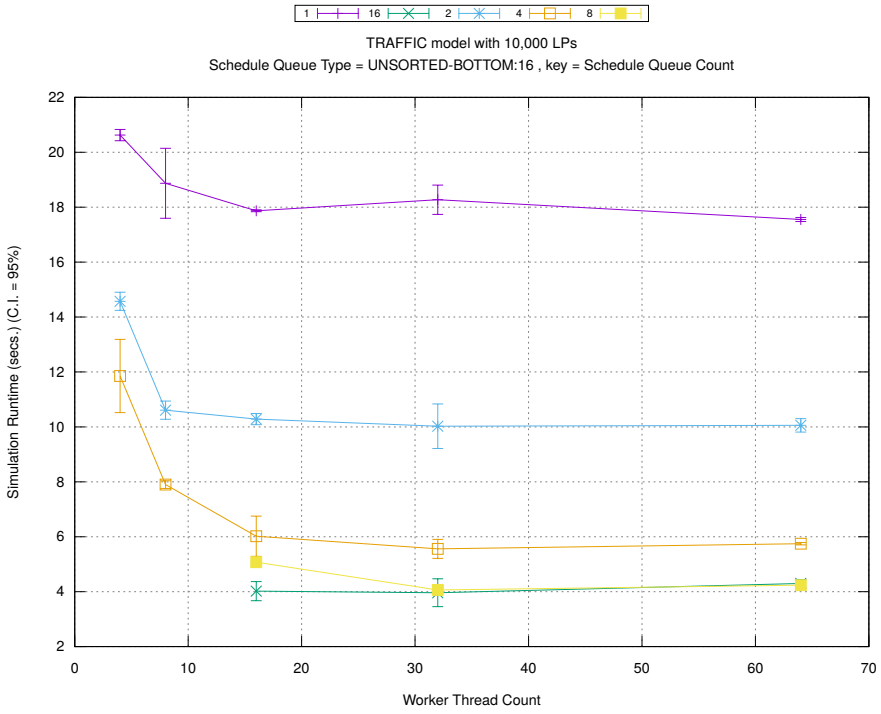


(c) Event Processing Rate (per second)

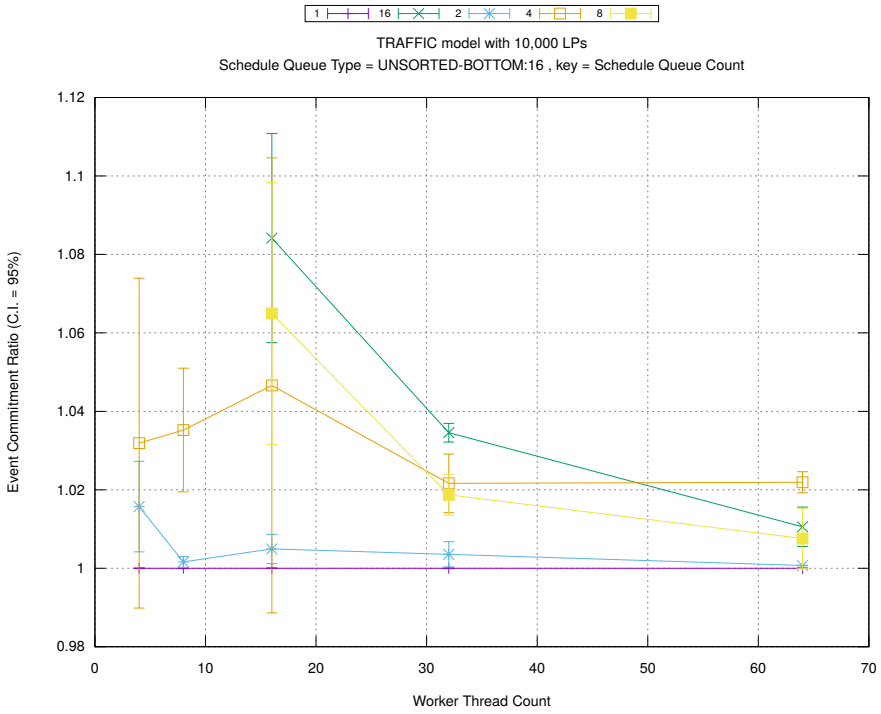
Figure A.24: traffic 10k/plots/scheduleq/threads vs type key count 1



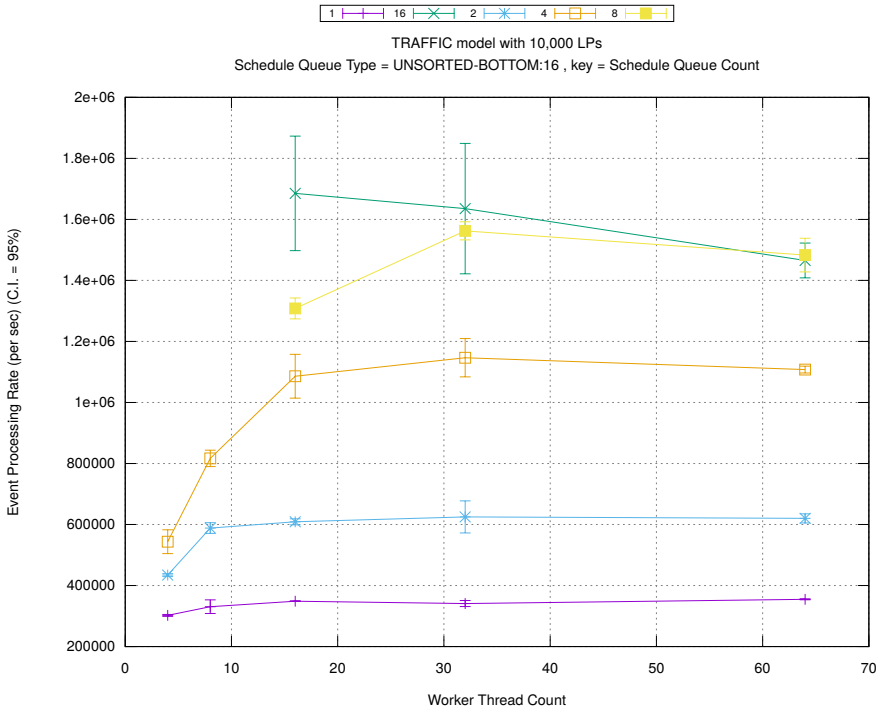
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

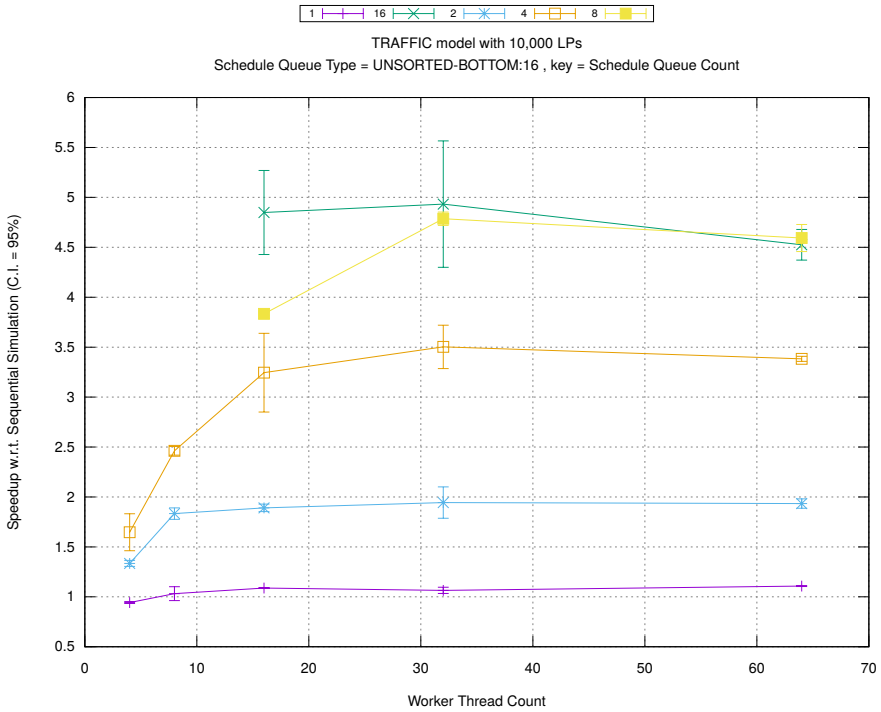


(b) Event Commitment Ratio

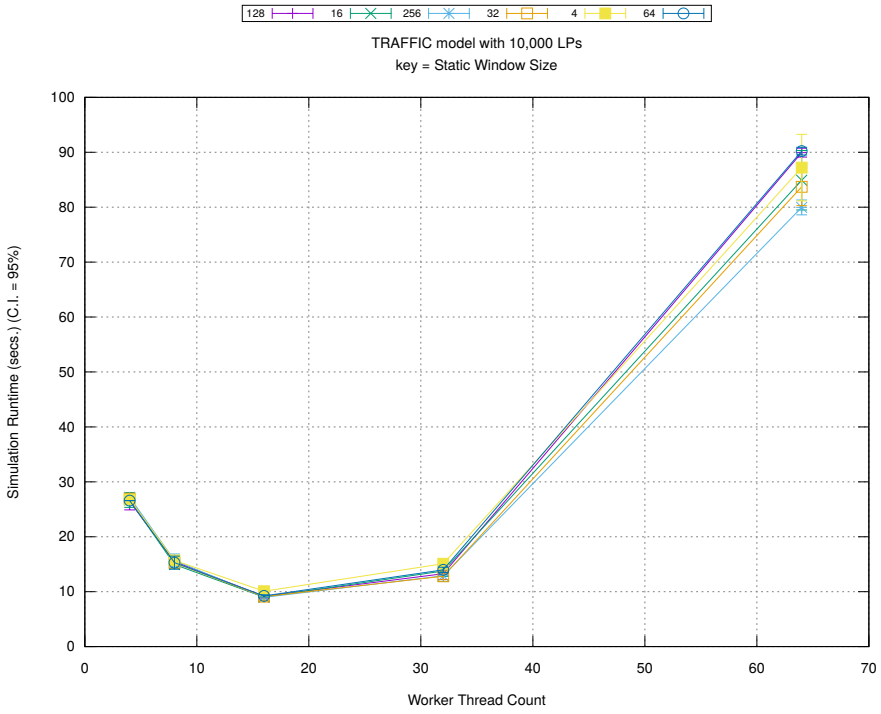


(c) Event Processing Rate (per second)

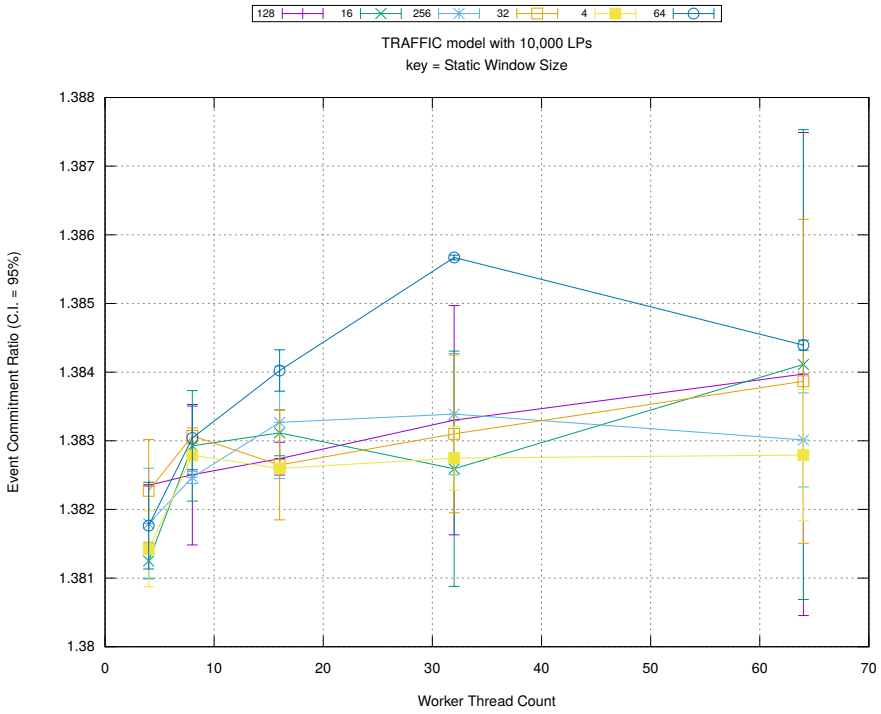
Figure A.25: traffic 10k/plots/scheduleq/threads vs count key type unsorted-bottom 16



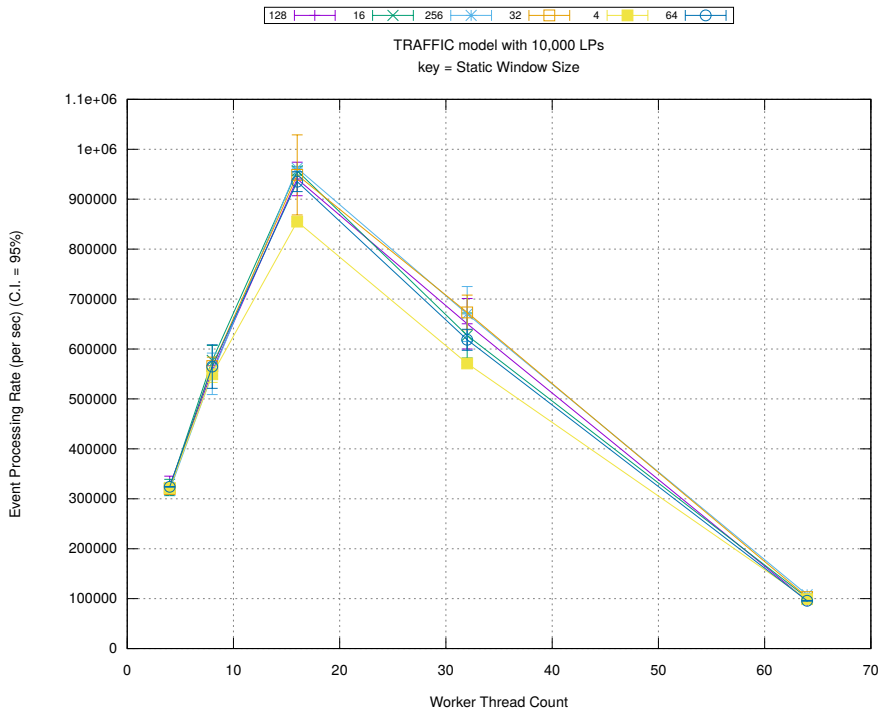
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

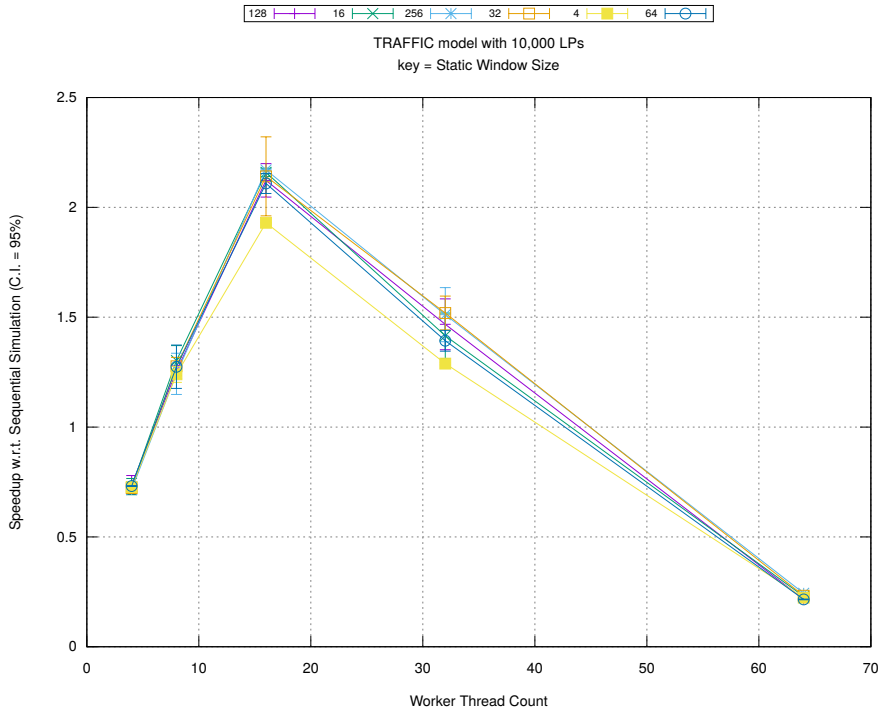


(b) Event Commitment Ratio

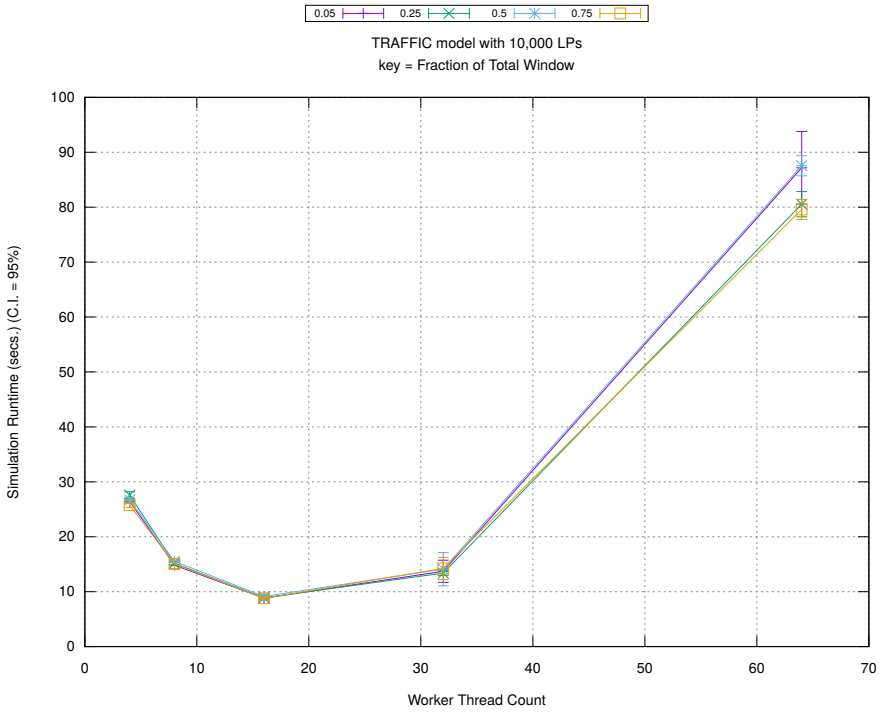


(c) Event Processing Rate (per second)

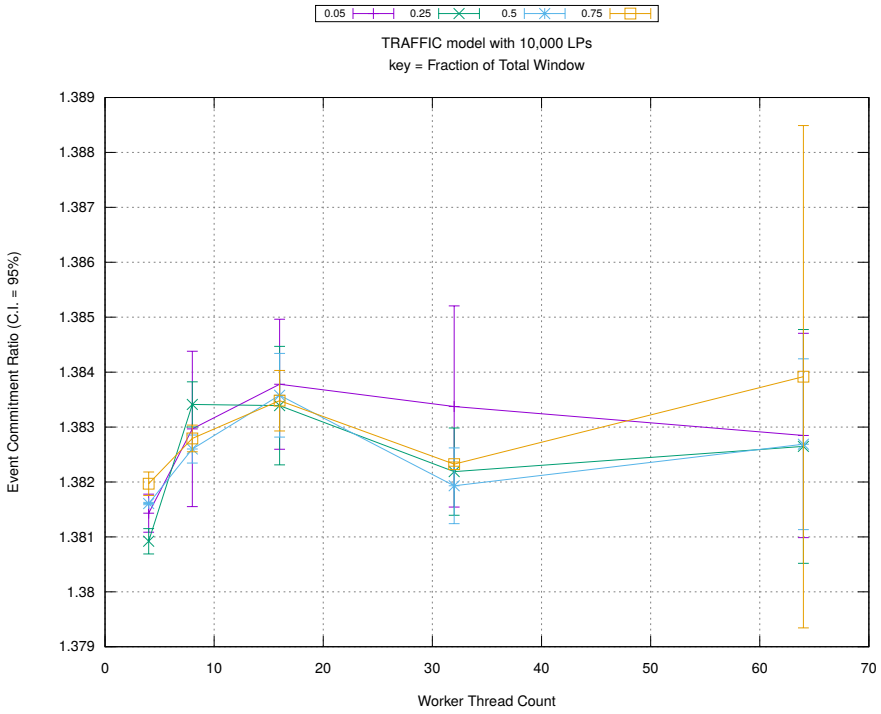
Figure A.26: traffic 10k/plots/bags/threads vs staticwindow key fractionwindow 1.0



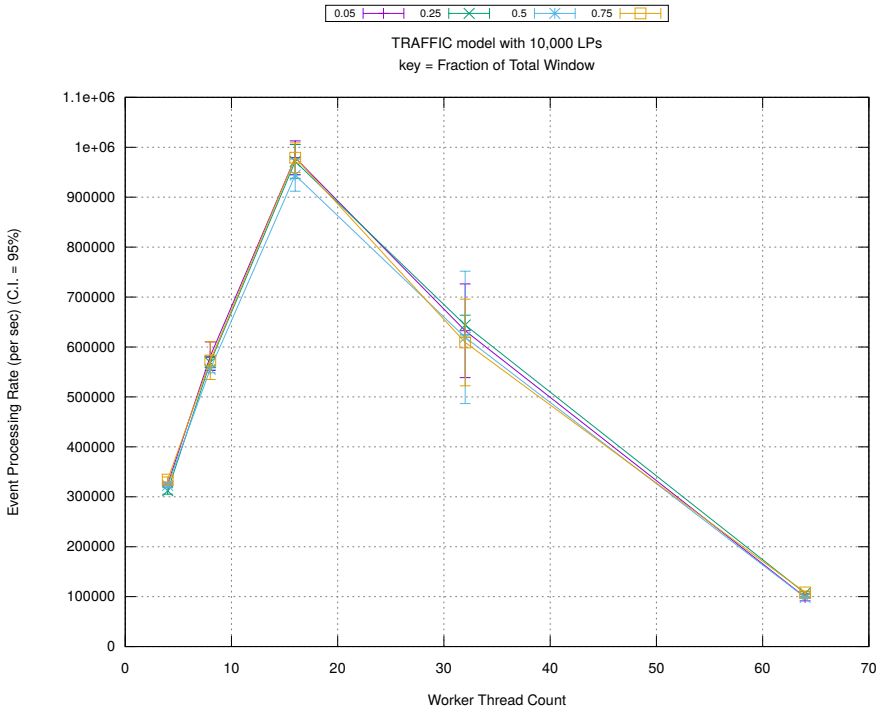
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

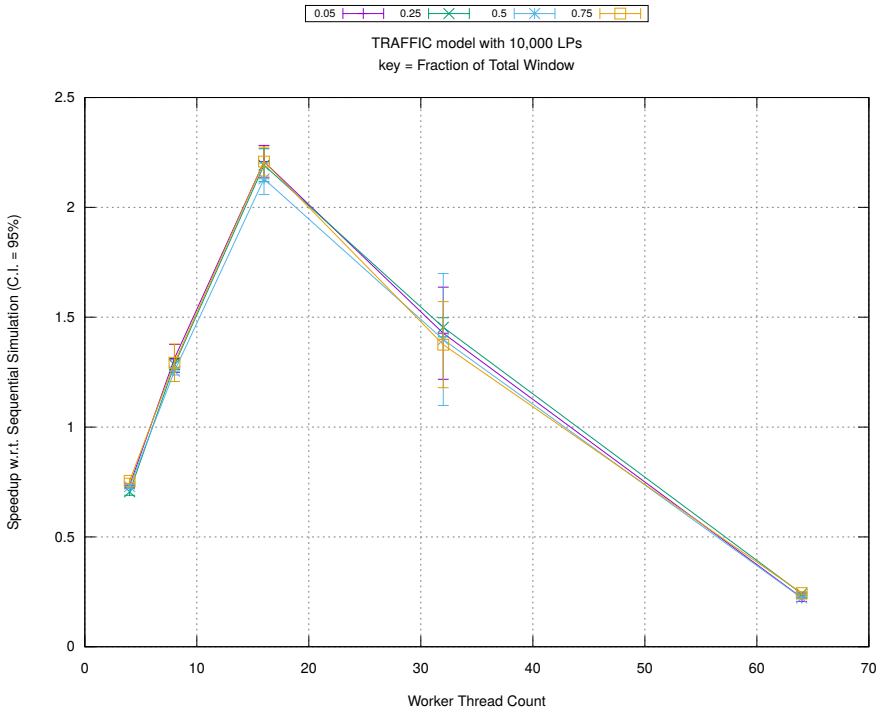


(b) Event Commitment Ratio

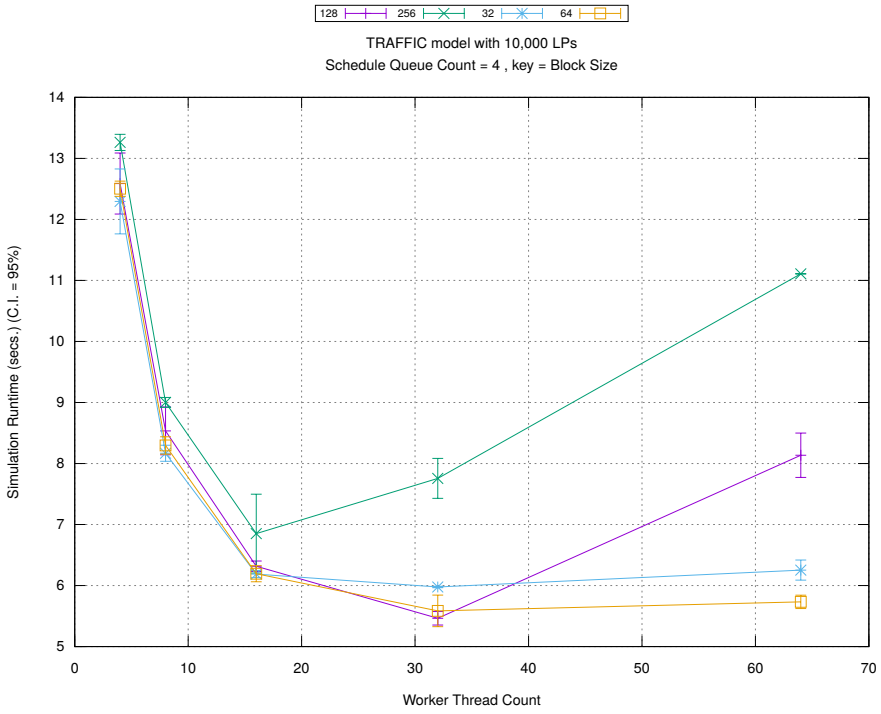


(c) Event Processing Rate (per second)

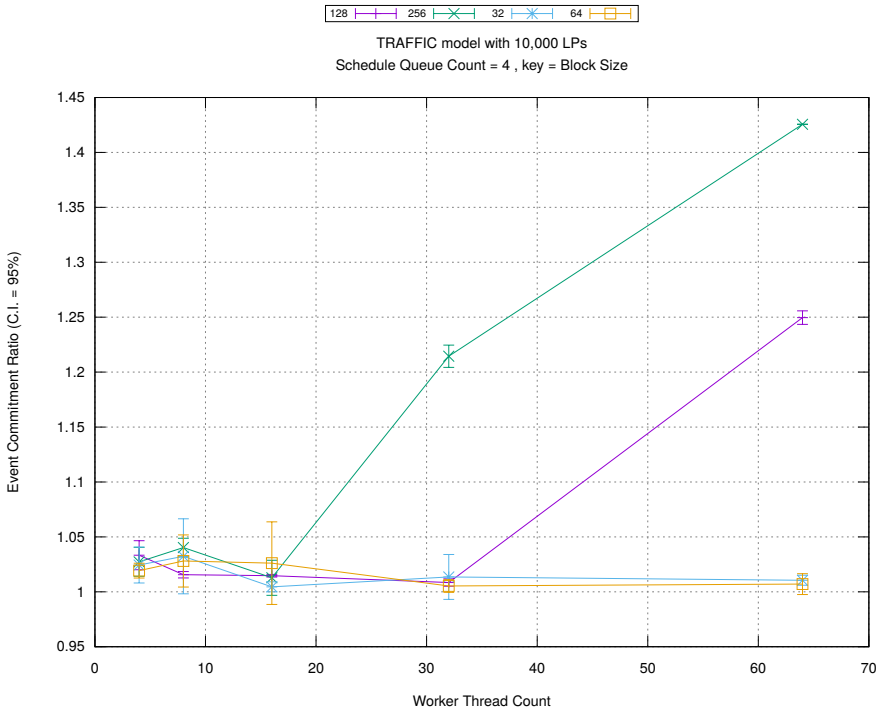
Figure A.27: traffic 10k/plots/bags/threads vs fractionwindow key staticwindow 0



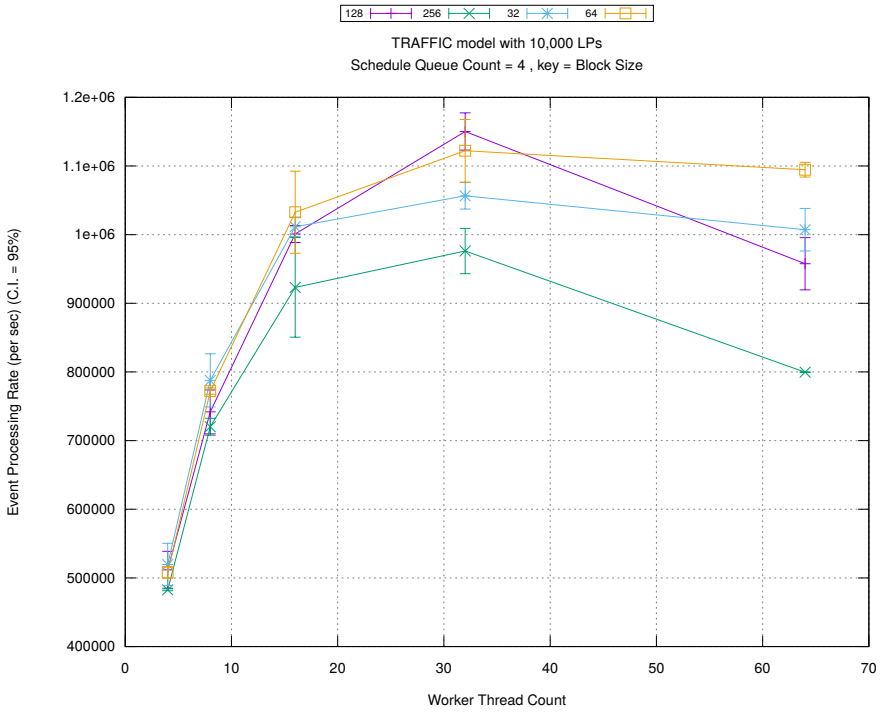
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

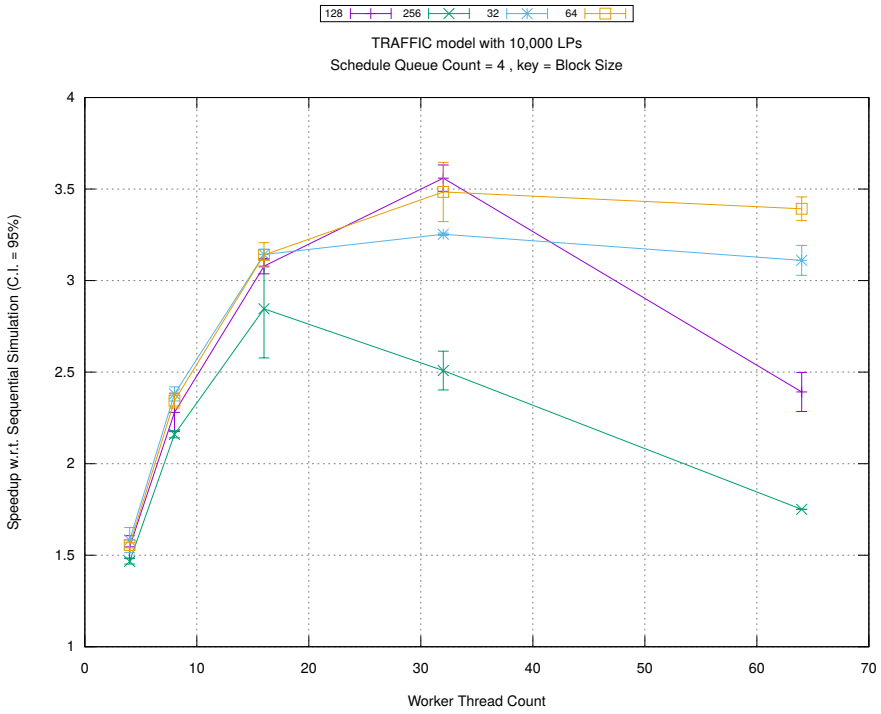


(b) Event Commitment Ratio

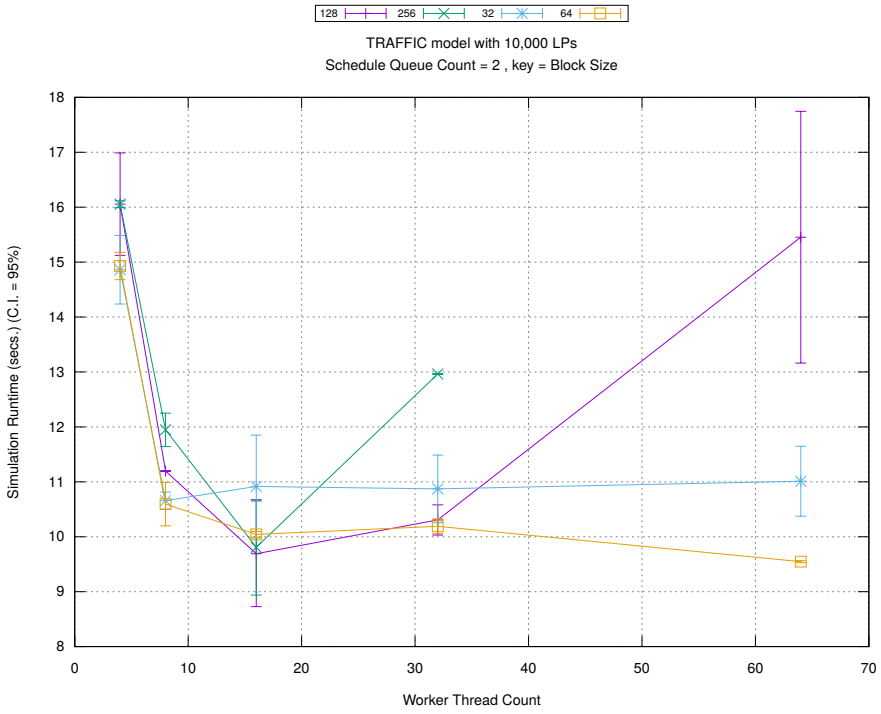


(c) Event Processing Rate (per second)

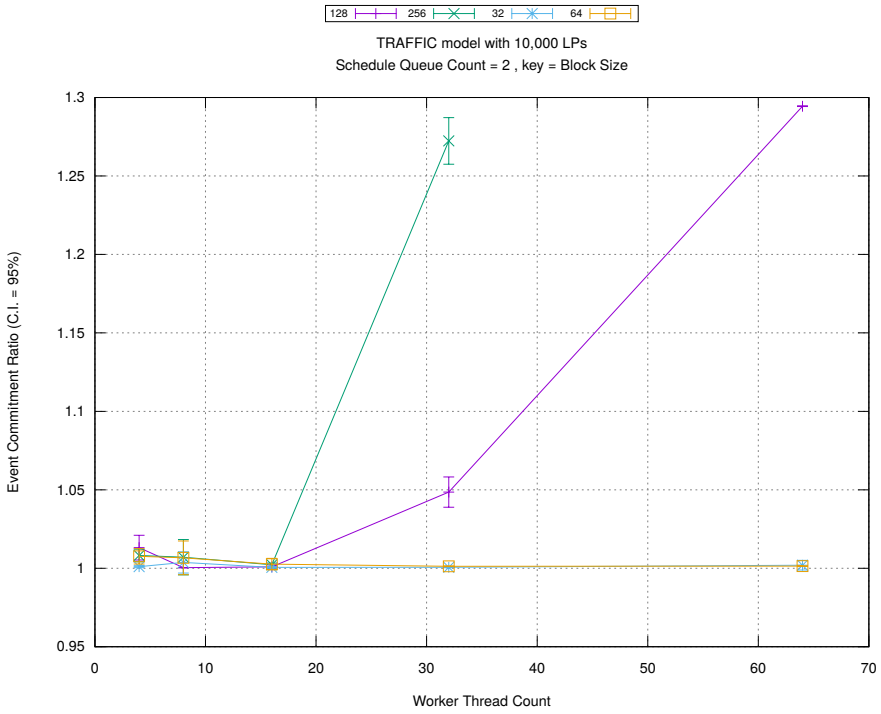
Figure A.28: traffic 10k/plots/blocks/threads vs blocksize key count 4



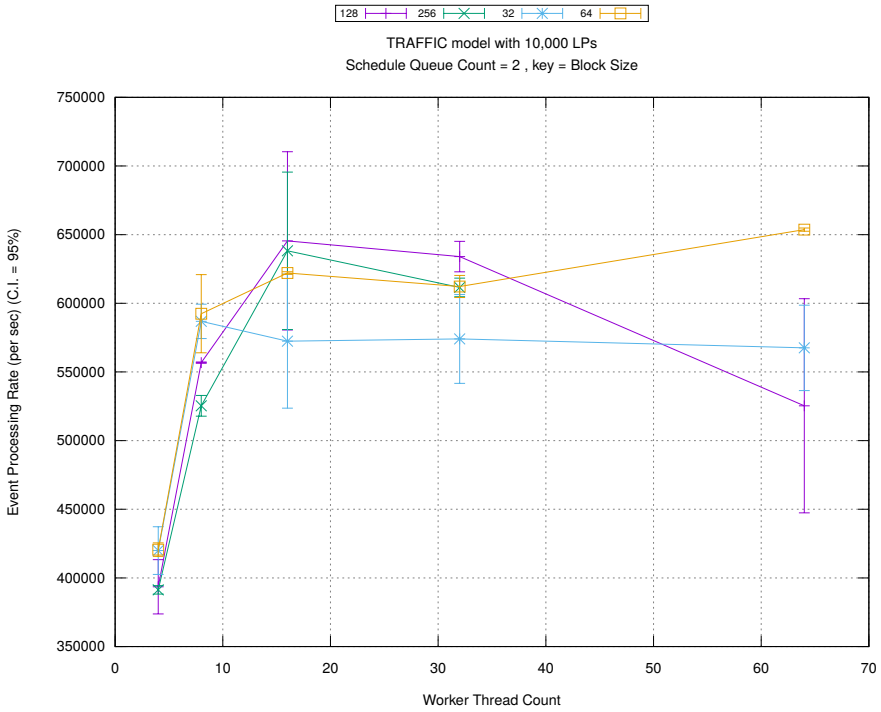
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

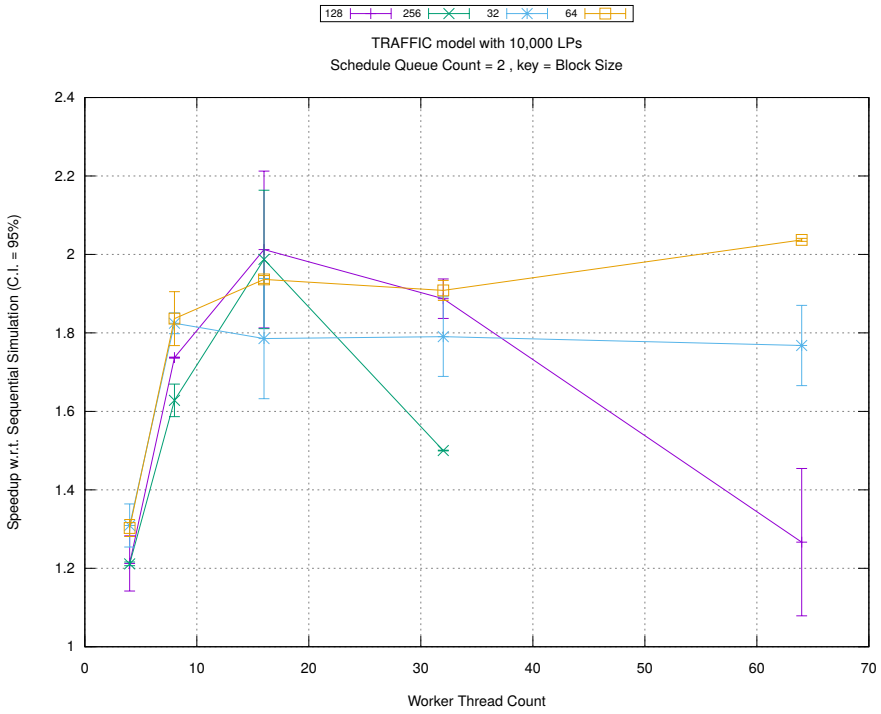


(b) Event Commitment Ratio

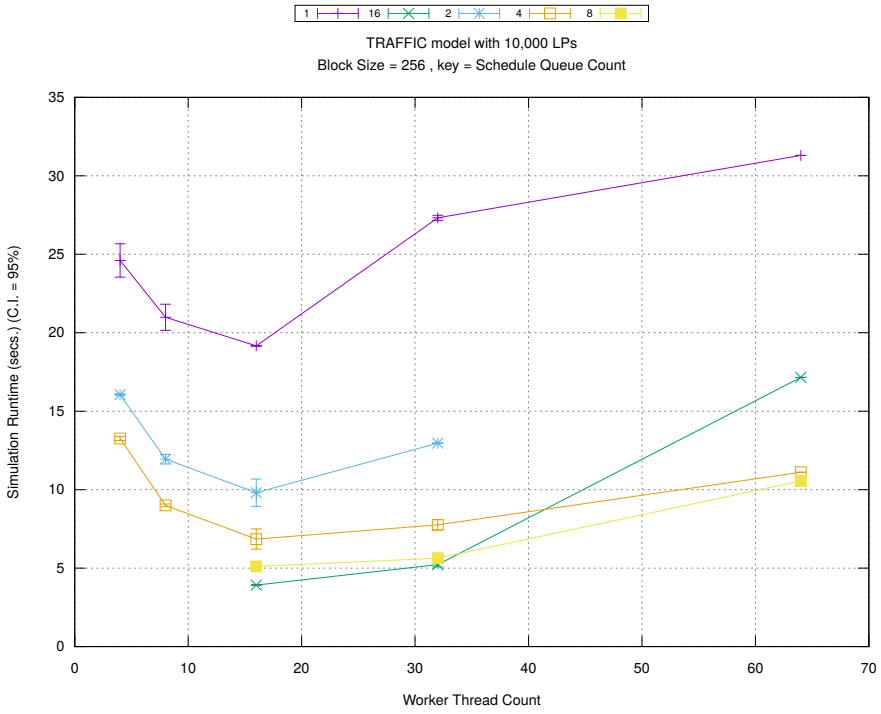


(c) Event Processing Rate (per second)

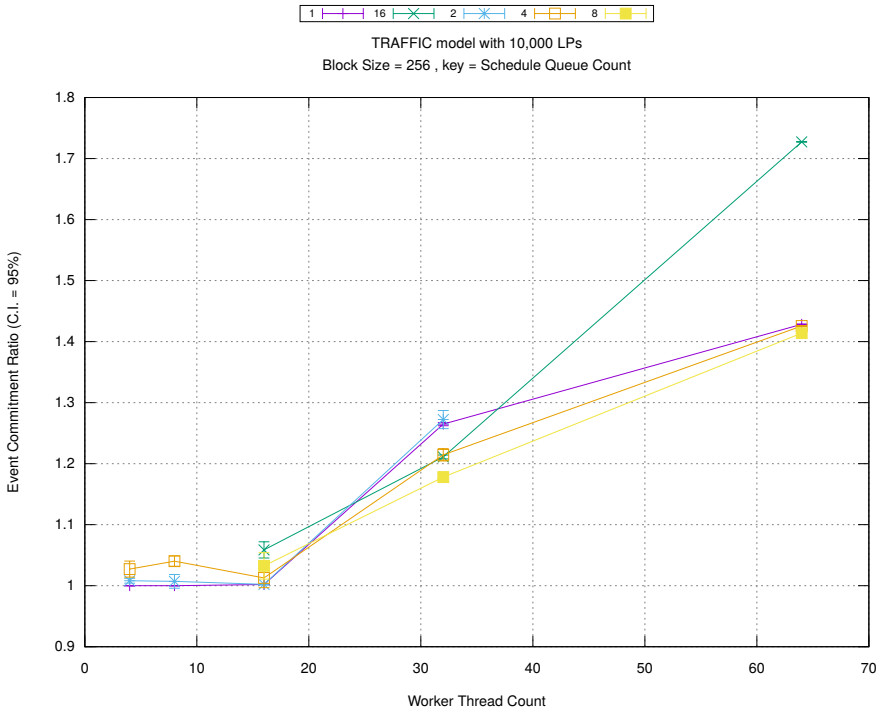
Figure A.29: traffic 10k/plots/blocks/threads vs blocksize key count 2



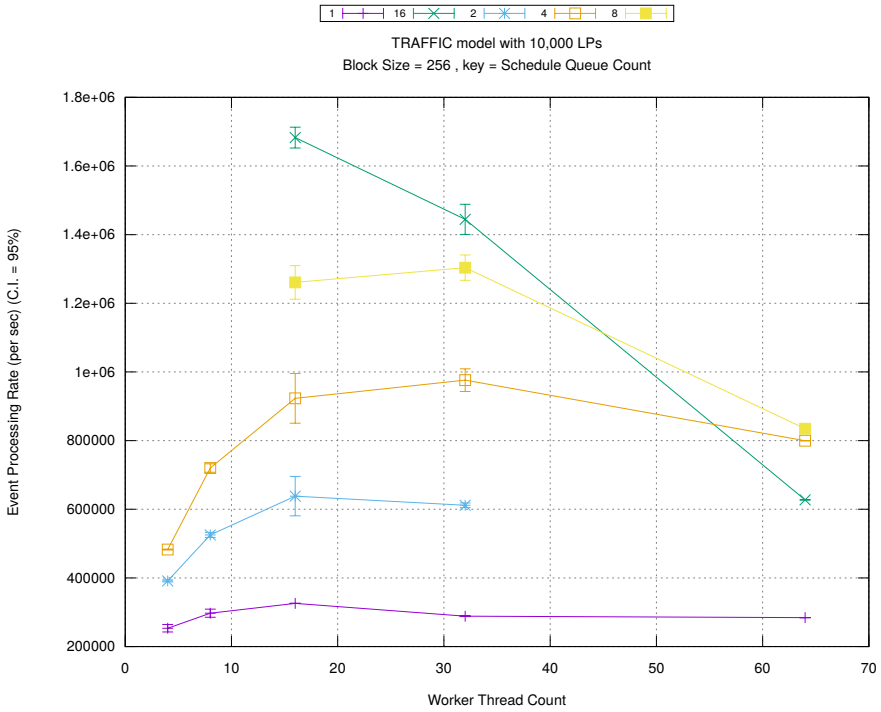
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

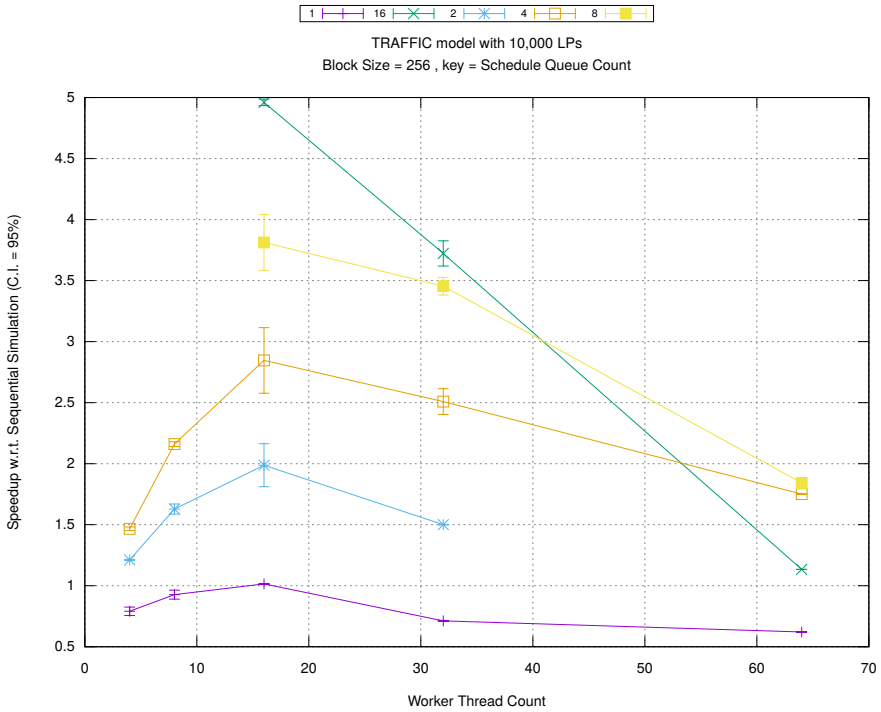


(b) Event Commitment Ratio

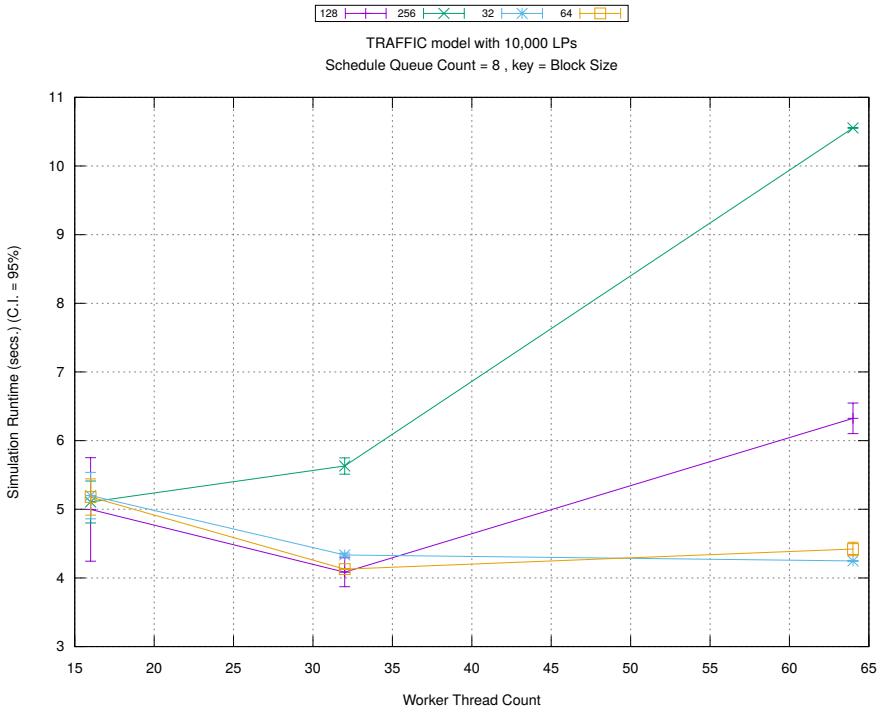


(c) Event Processing Rate (per second)

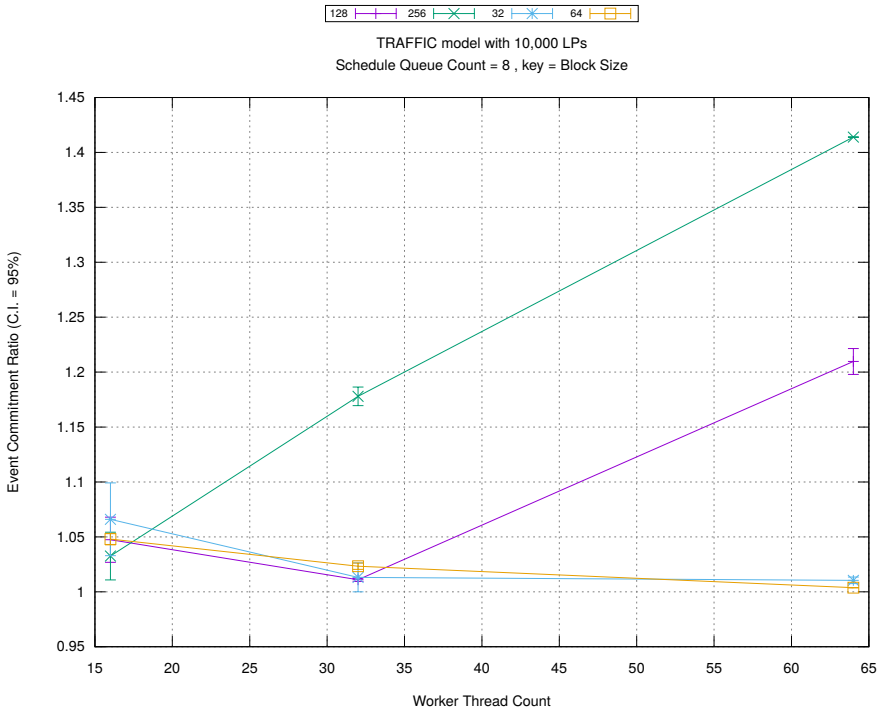
Figure A.30: traffic 10k/plots/blocks/threads vs count key blocksize 256



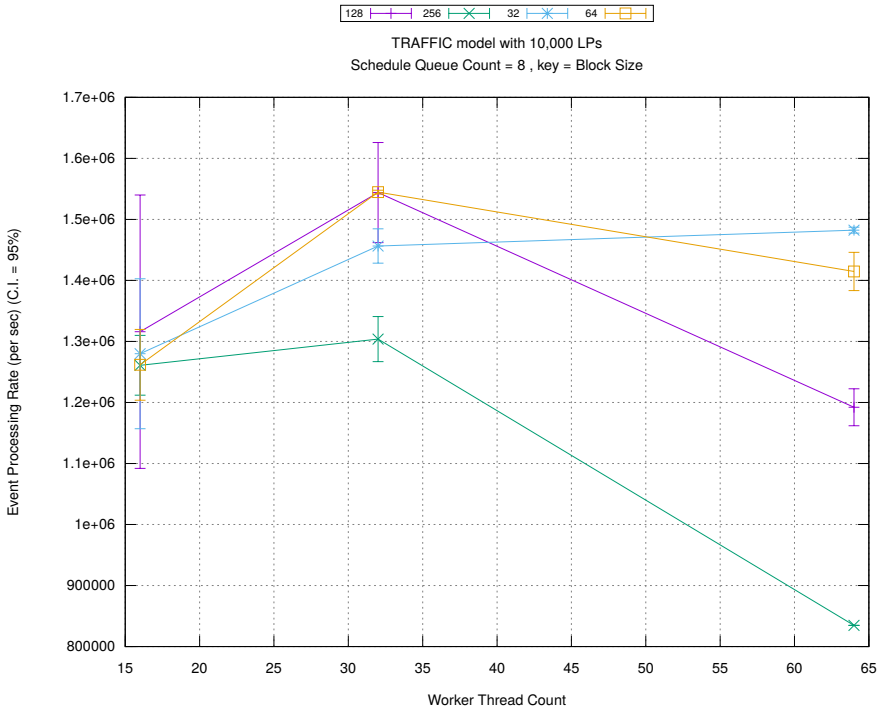
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

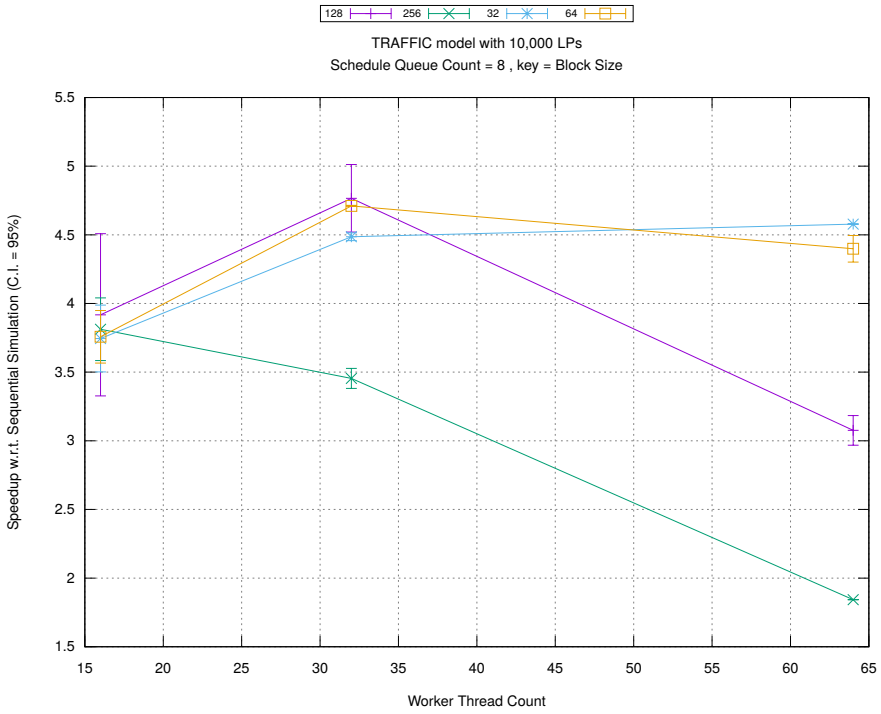


(b) Event Commitment Ratio

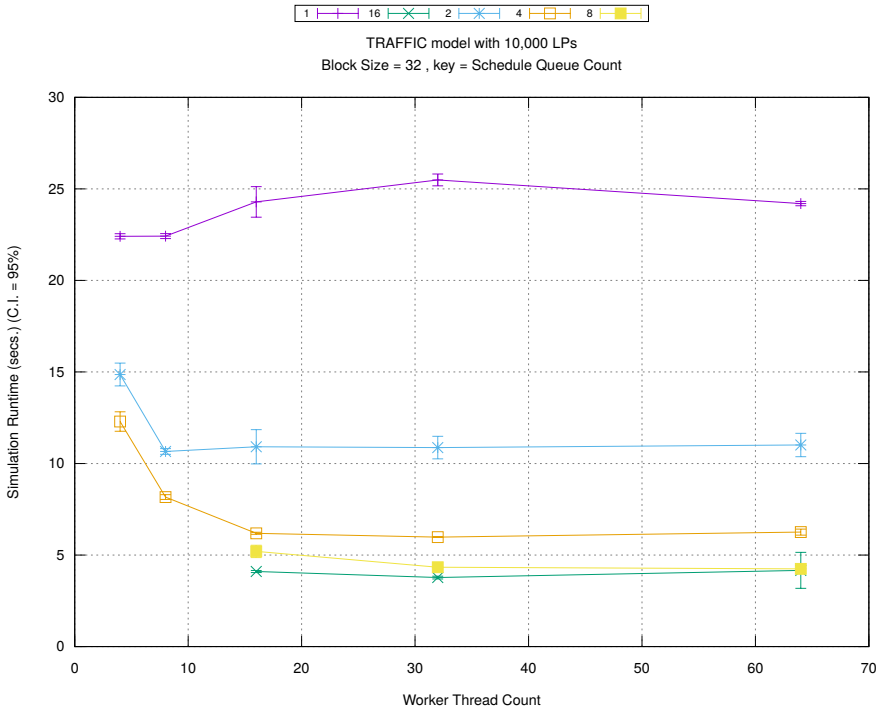


(c) Event Processing Rate (per second)

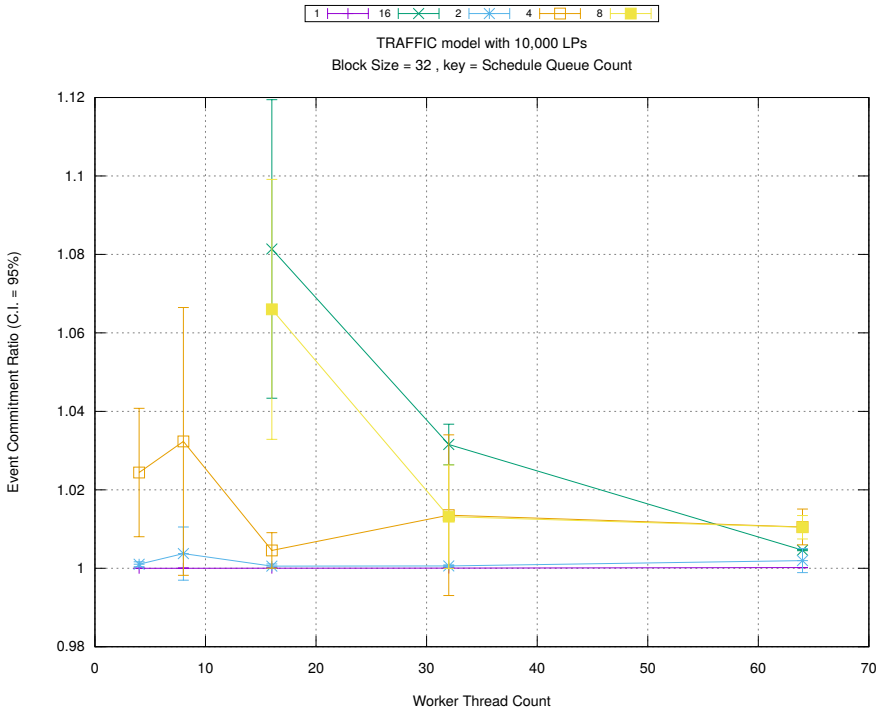
Figure A.31: traffic 10k/plots/blocks/threads vs blocksize key count 8



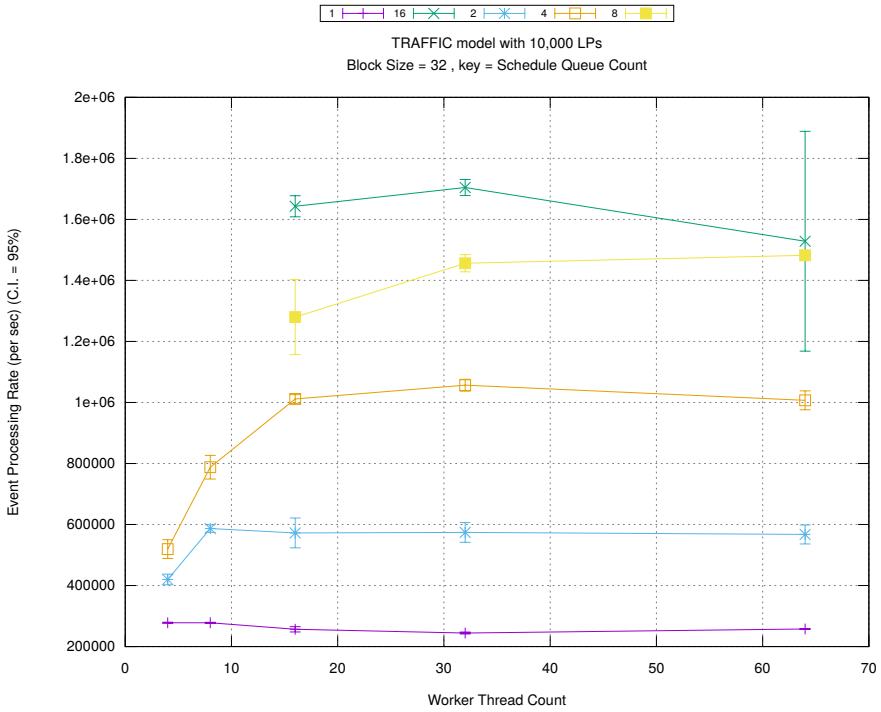
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

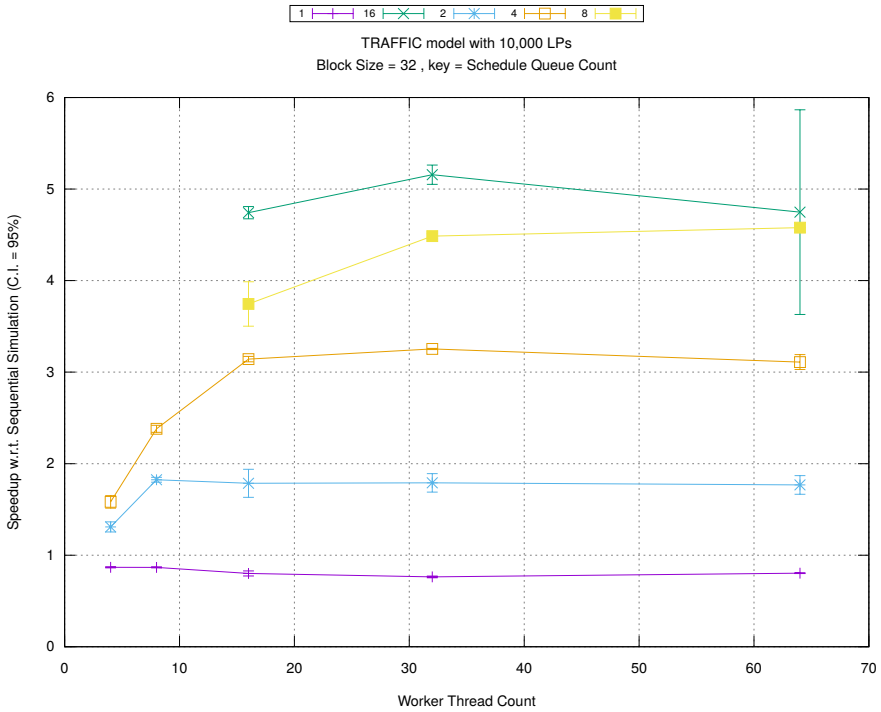


(b) Event Commitment Ratio

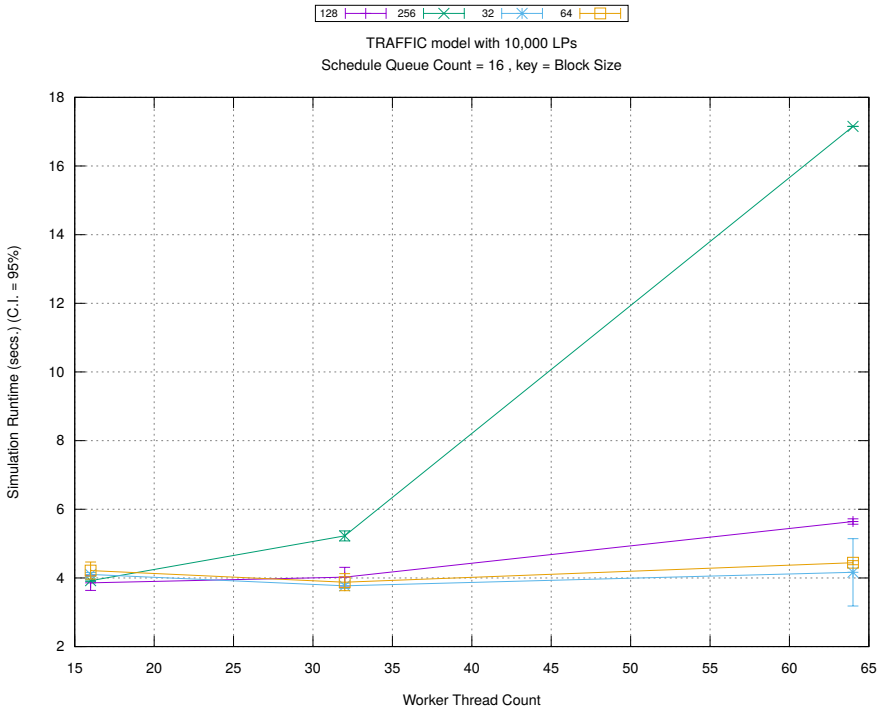


(c) Event Processing Rate (per second)

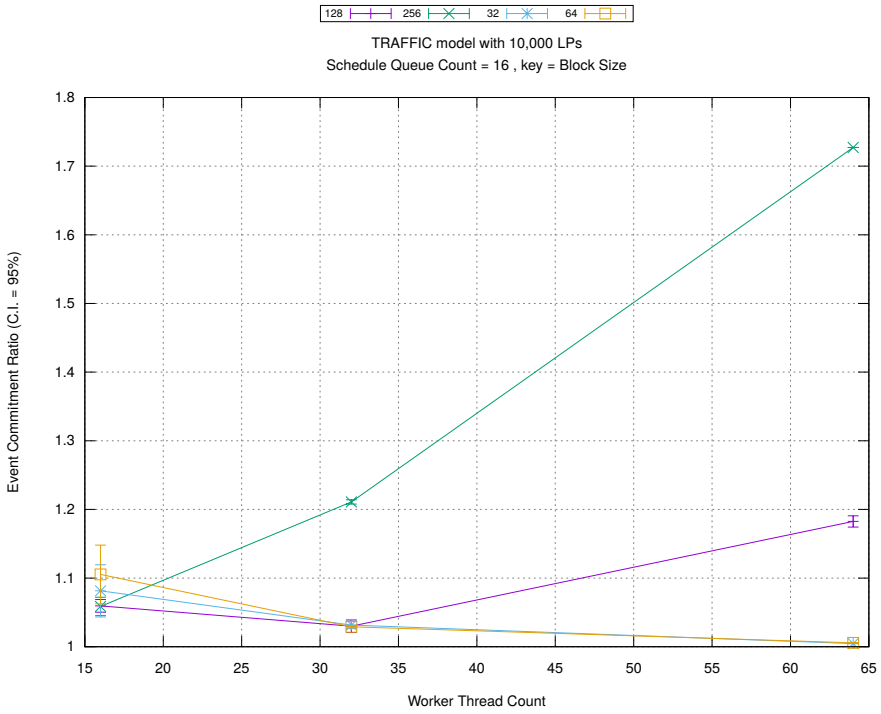
Figure A.32: traffic 10k/plots/blocks/threads vs count key blocksize 32



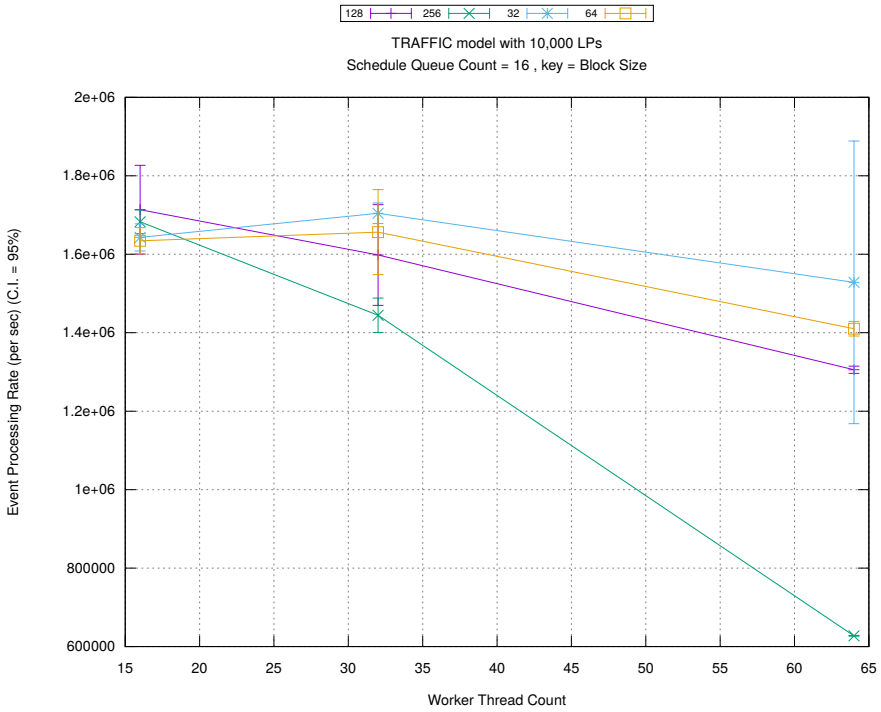
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

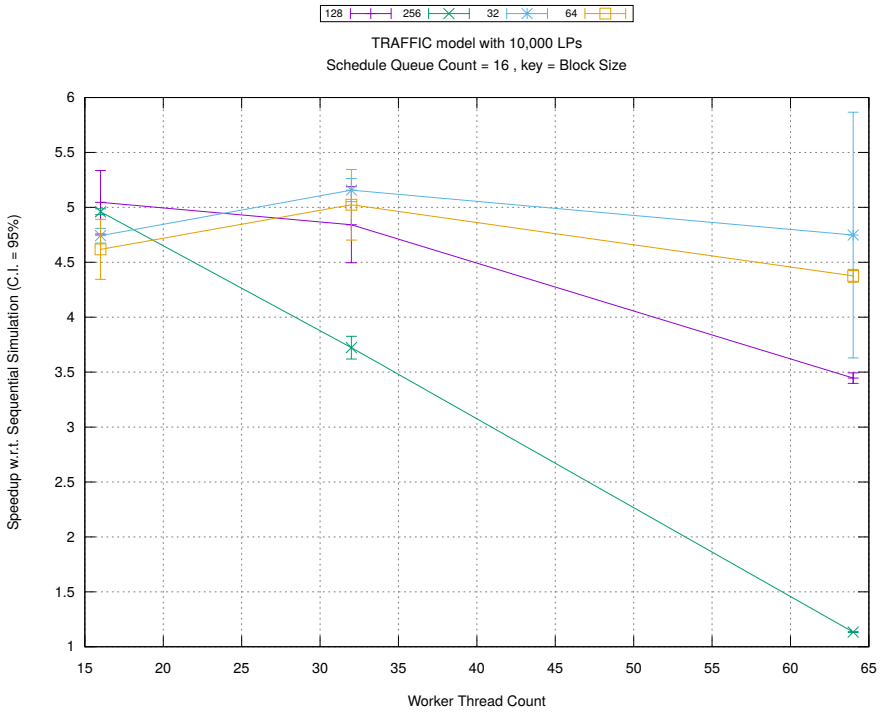


(b) Event Commitment Ratio

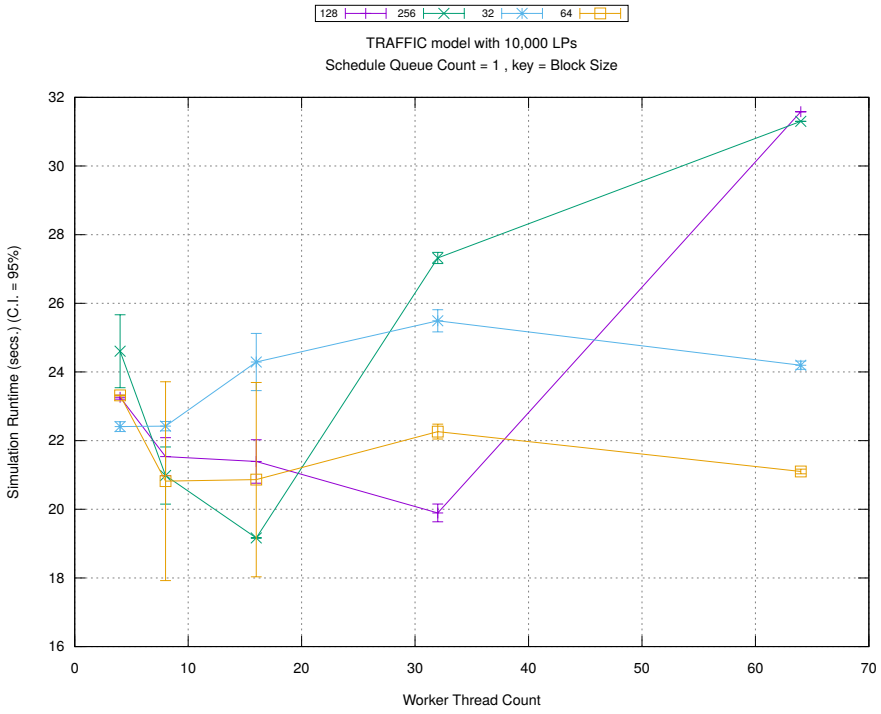


(c) Event Processing Rate (per second)

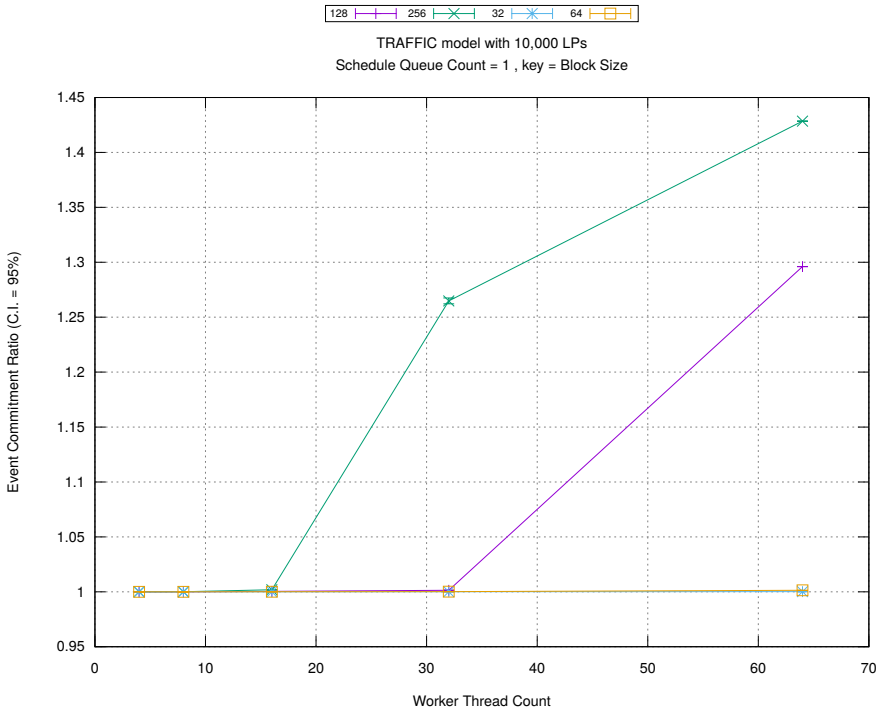
Figure A.33: traffic 10k/plots/blocks/threads vs blocksize key count 16



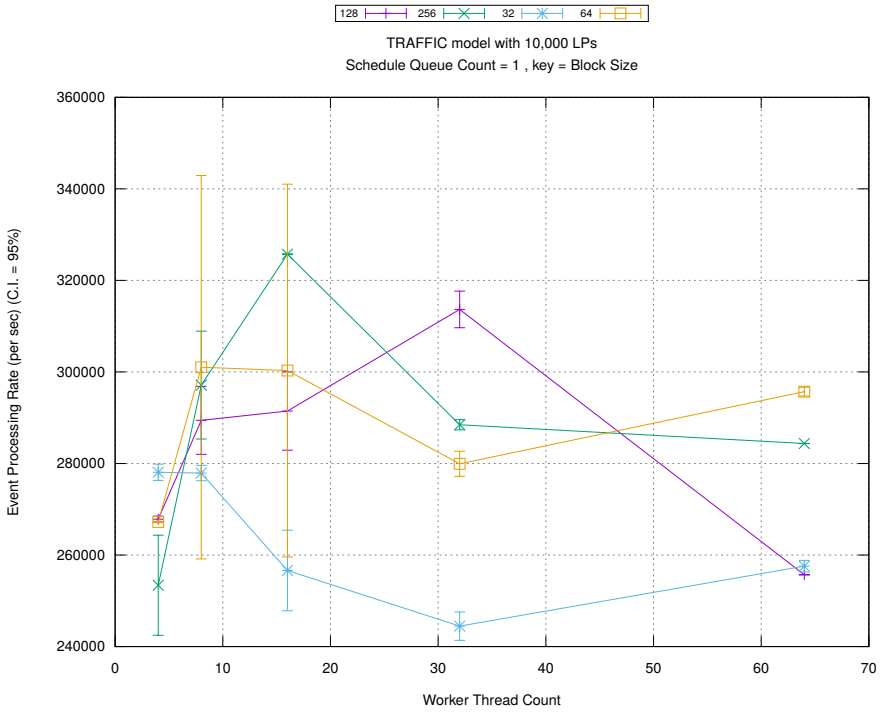
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

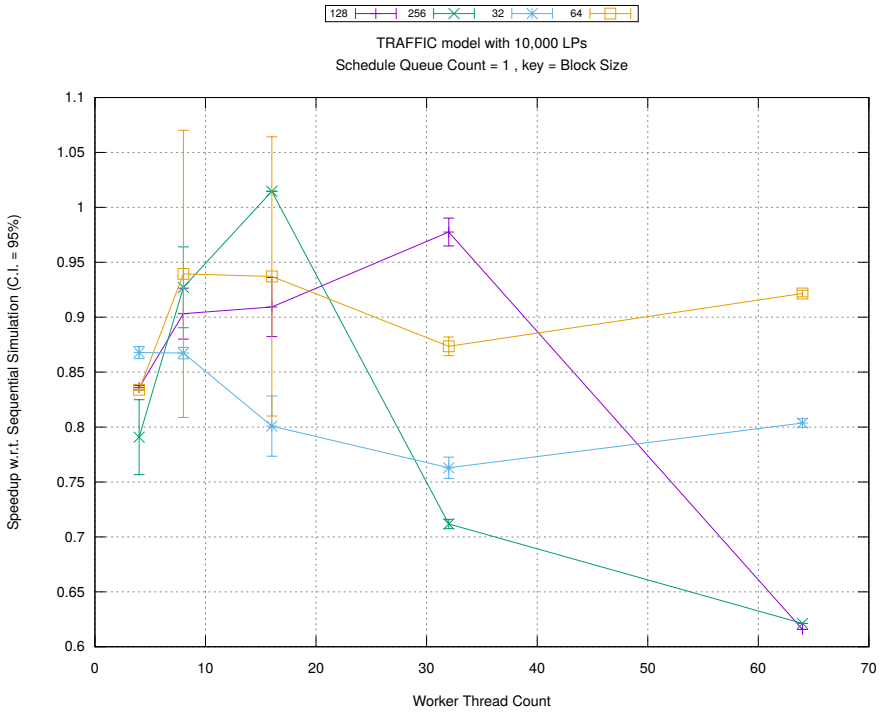


(b) Event Commitment Ratio

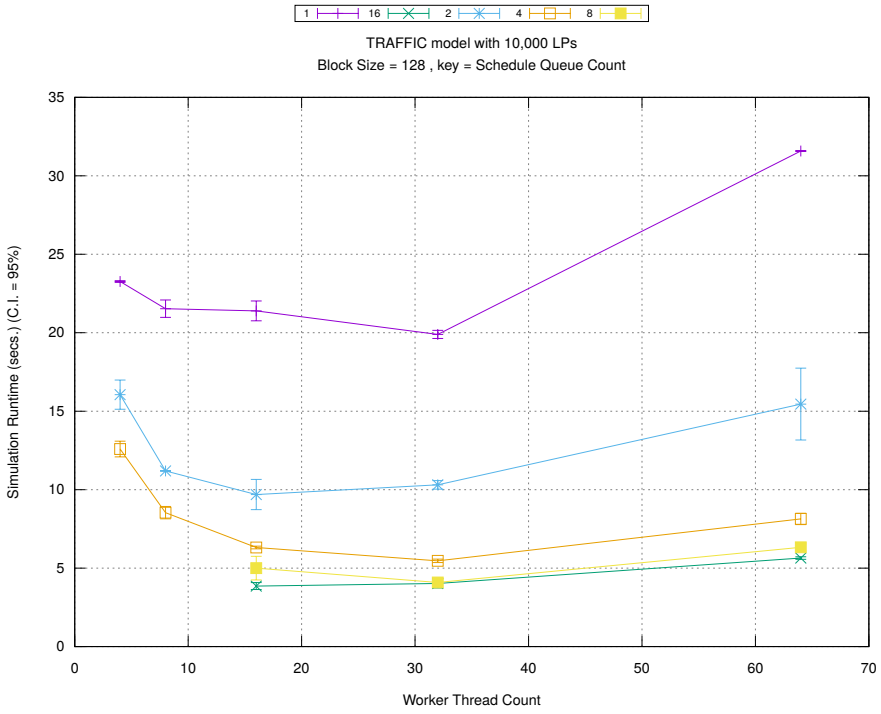


(c) Event Processing Rate (per second)

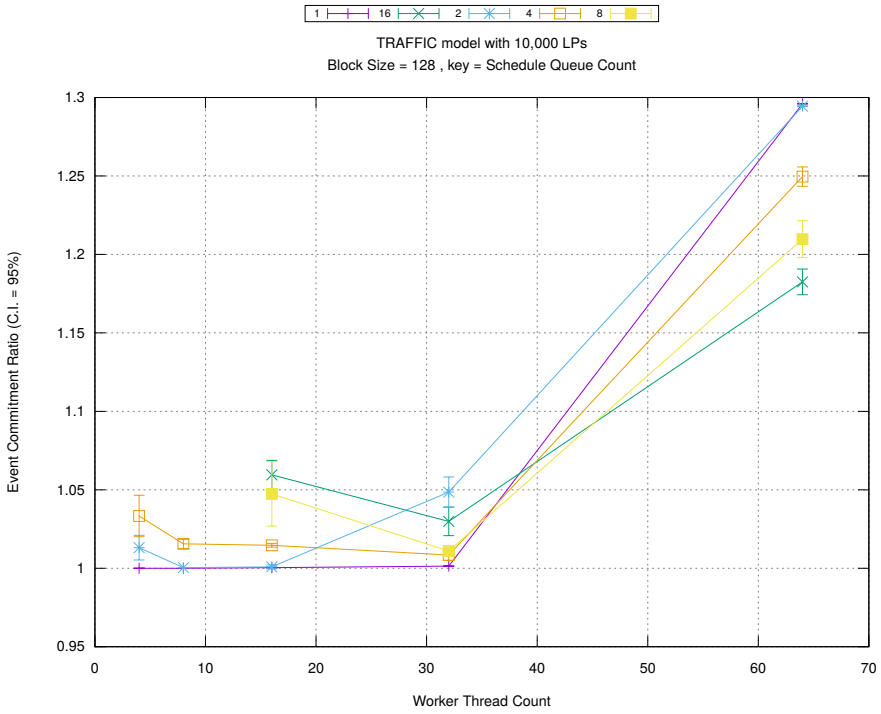
Figure A.34: traffic 10k/plots/blocks/threads vs blocksize key count 1



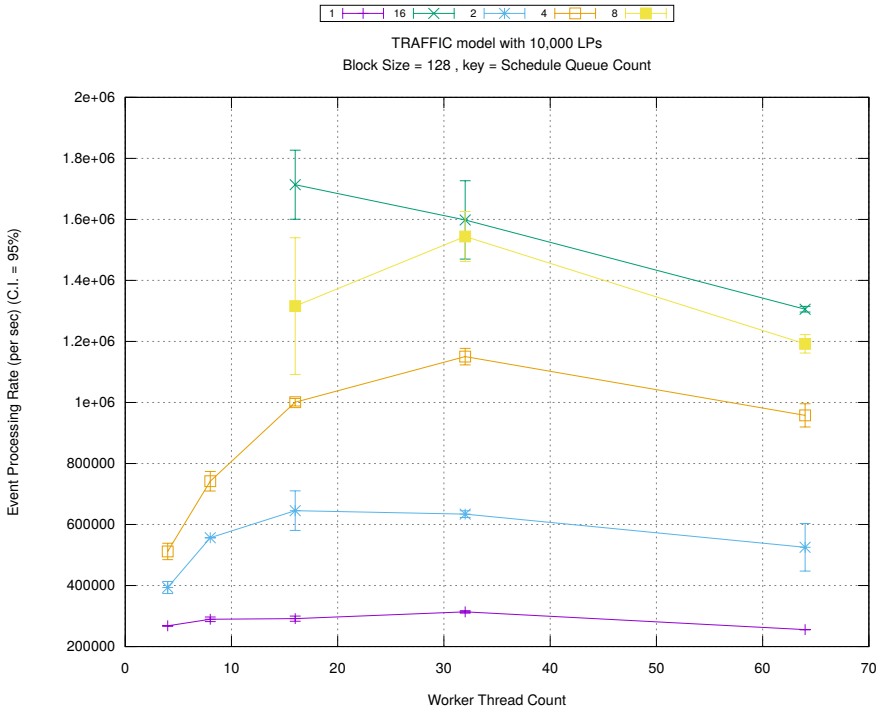
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

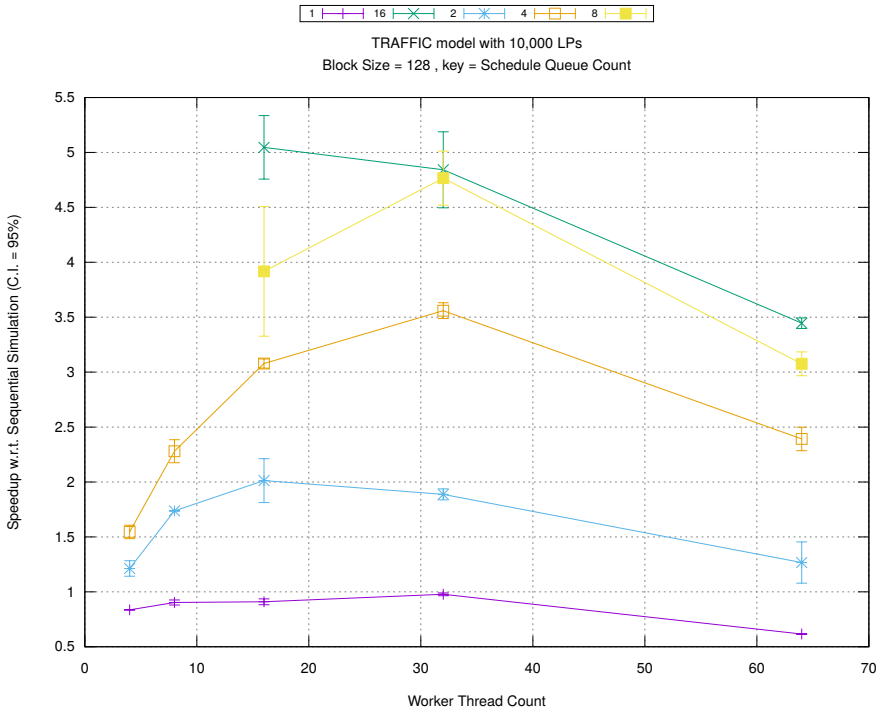


(b) Event Commitment Ratio

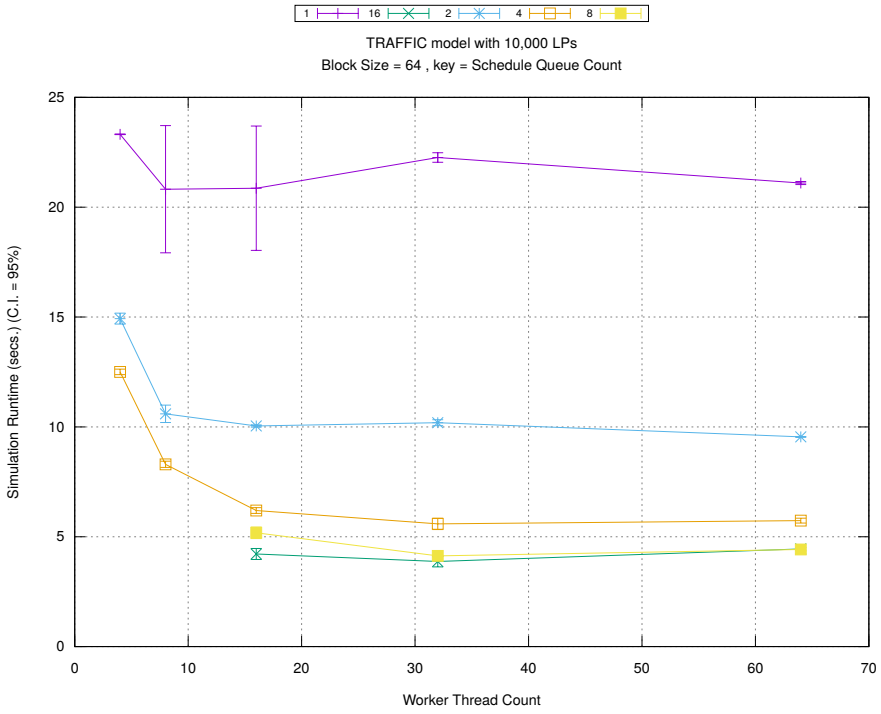


(c) Event Processing Rate (per second)

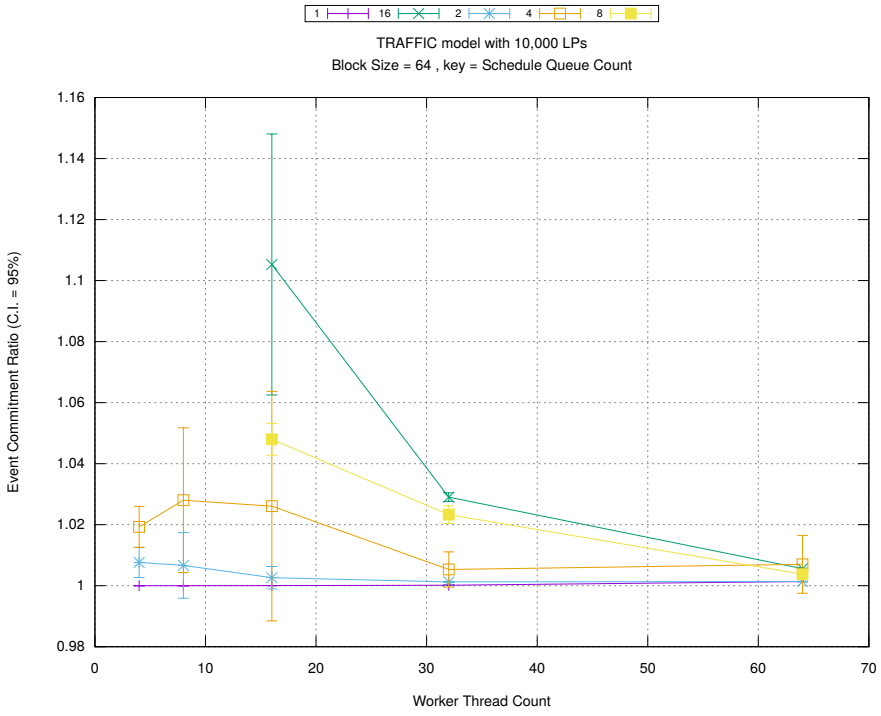
Figure A.35: traffic 10k/plots/blocks/threads vs count key blocksize 128



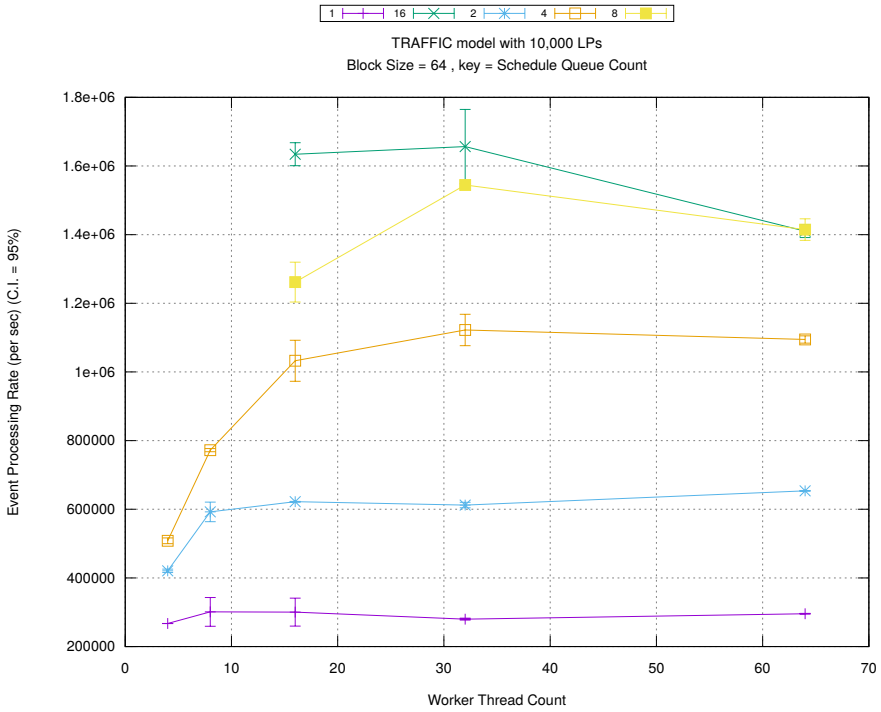
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

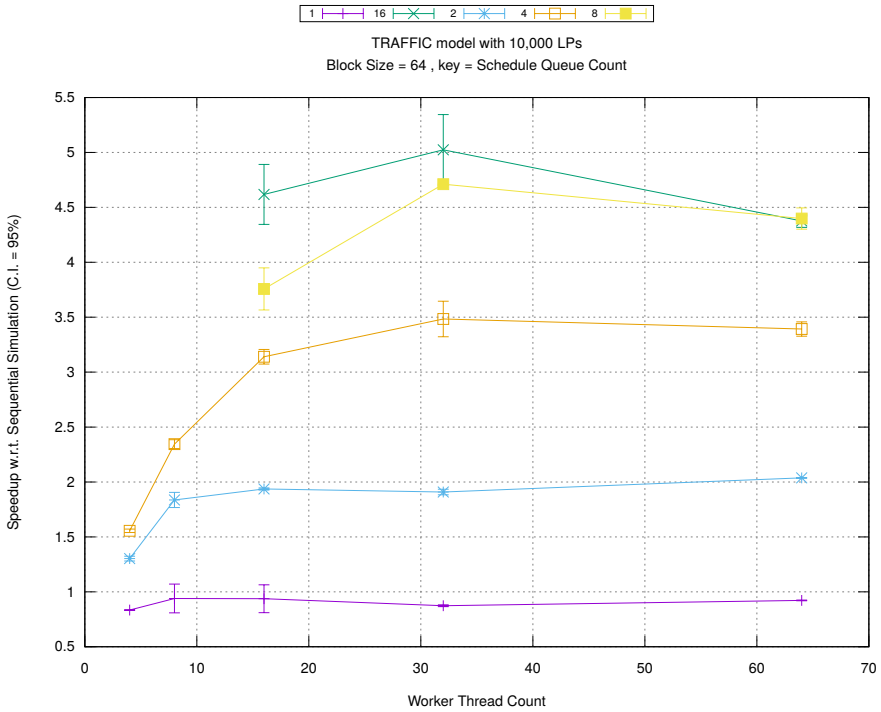


(b) Event Commitment Ratio



(c) Event Processing Rate (per second)

Figure A.36: traffic 10k/plots/blocks/threads vs count key blocksize 64



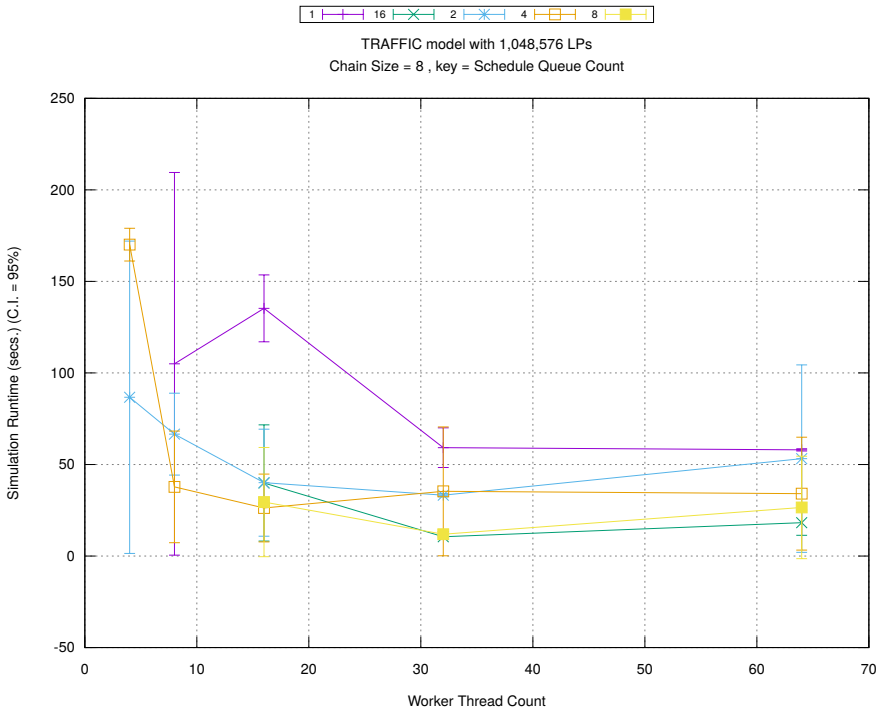
(d) Speedup w.r.t. Sequential Simulation

A.2 Traffic Model Configured with 1,048,576 LPs

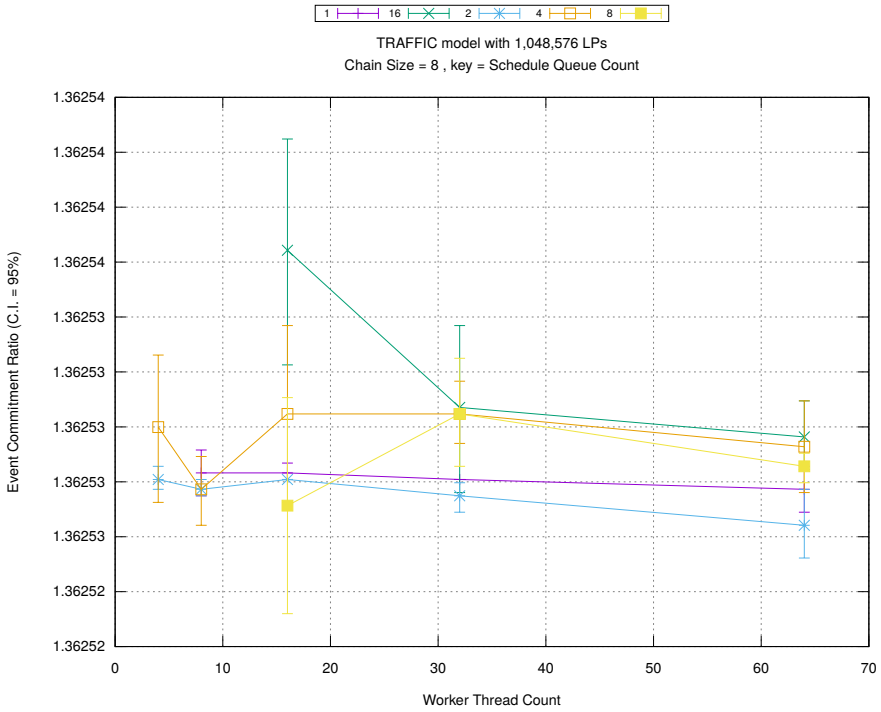
Table A.2 shows the configuration for this model.

Parameter	Values
Number of Intersections (or LPs)	1,048,576
Type of network connecting LPs	4-directional grid
Grid Size	1024x1024
Number of Cars initially at each intersection	25
Mean car arrival interval at each intersection	400 timestamp units
Simulation Time	500 timestamp units
Sequential Simulation Time for calculating modularity	300 timestamp units

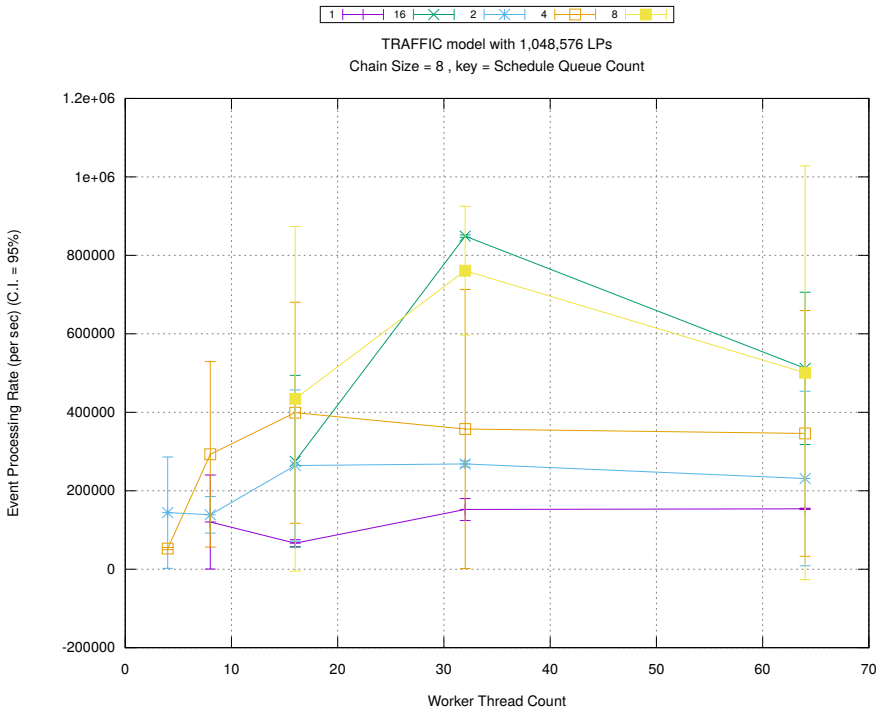
Table A.2: LARGE TRAFFIC MODEL setup



(a) Simulation Runtime (in seconds)

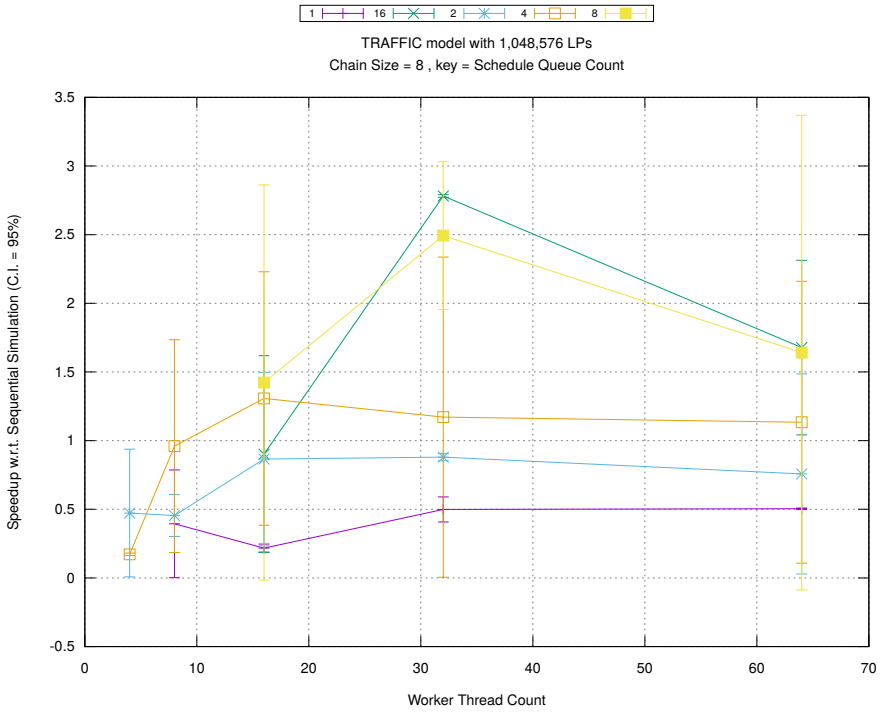


(b) Event Commitment Ratio

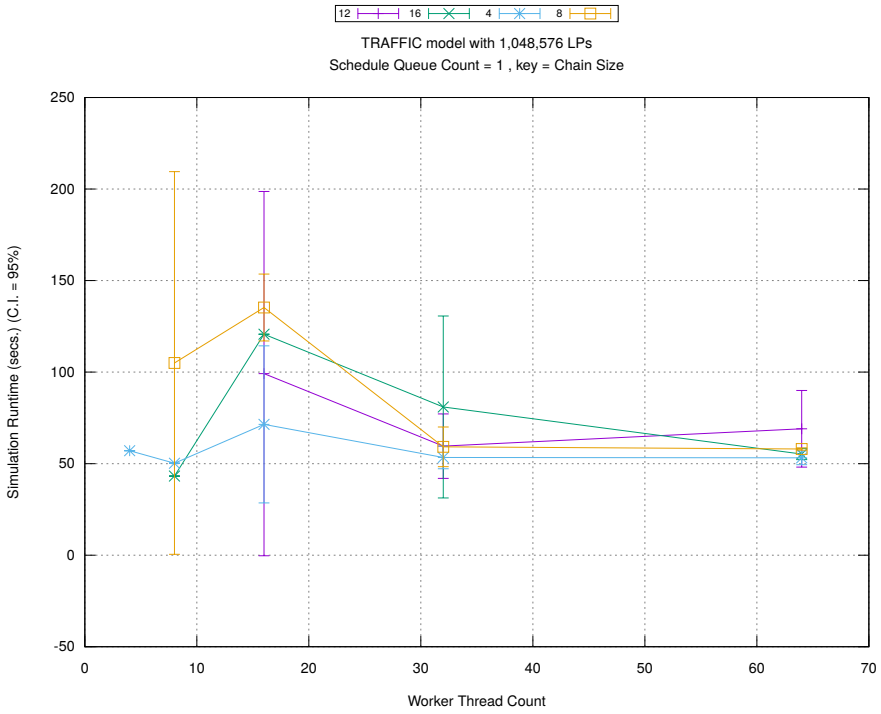


(c) Event Processing Rate (per second)

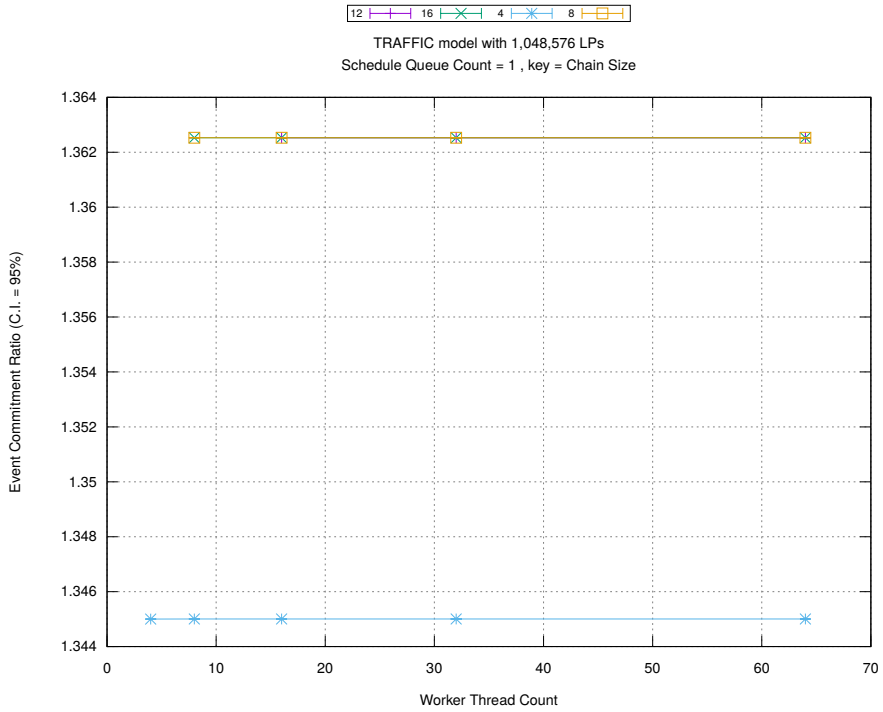
Figure A.37: traffic 1m/plots/chains/threads vs count key chainsize 8



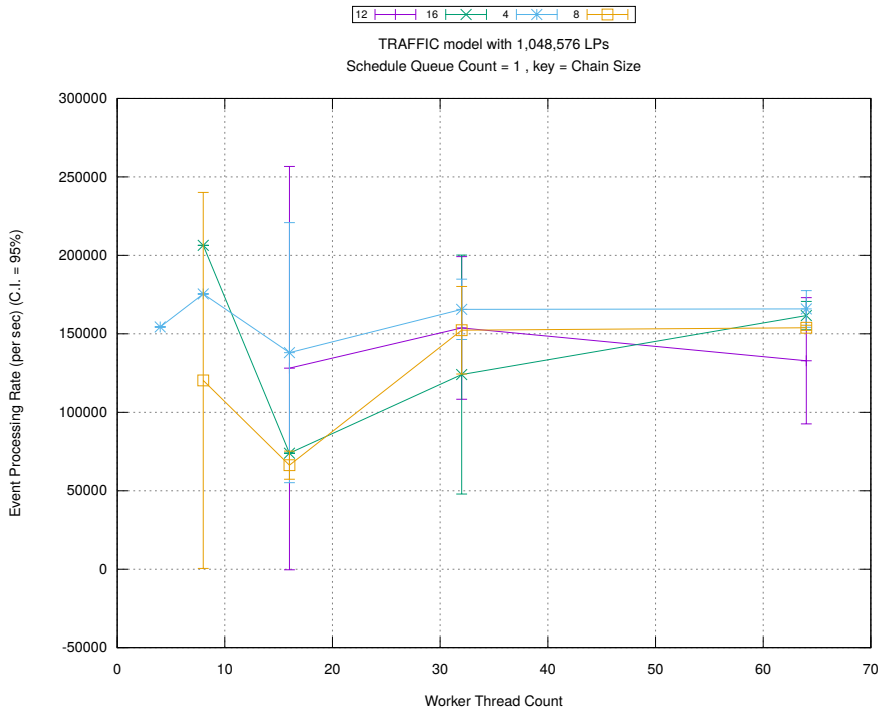
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

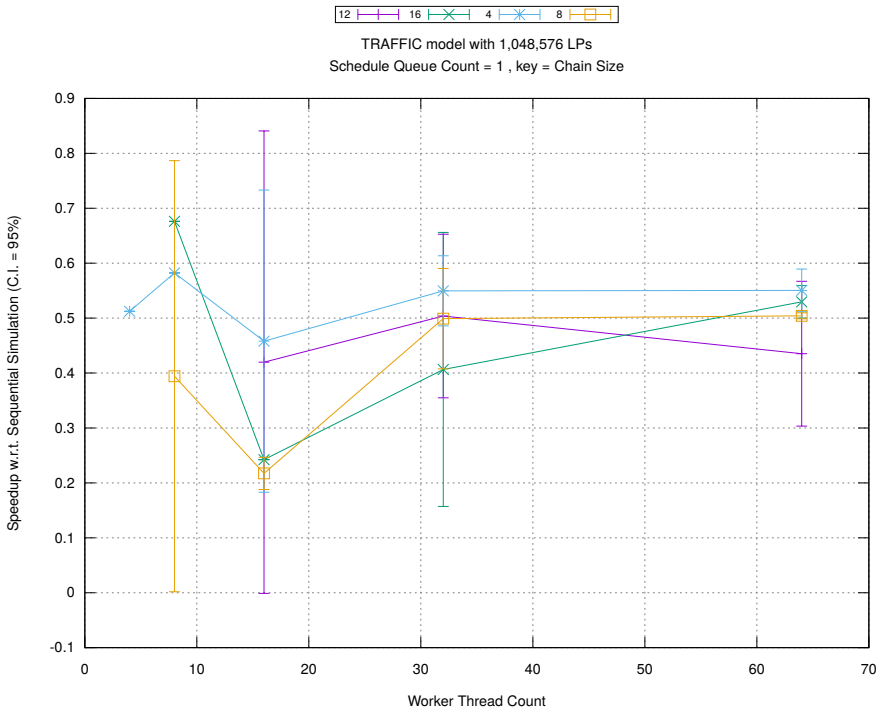


(b) Event Commitment Ratio

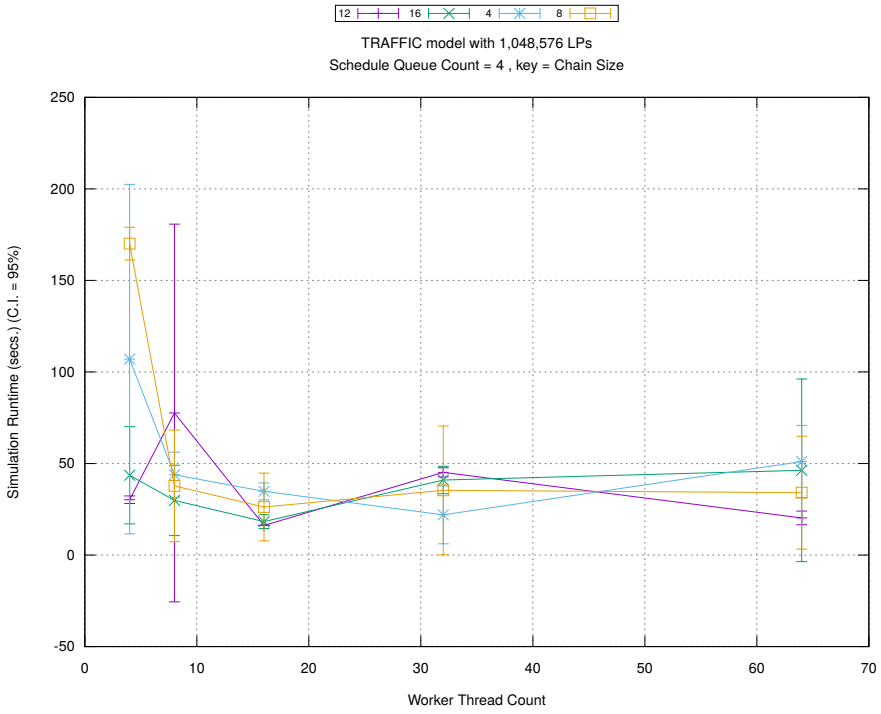


(c) Event Processing Rate (per second)

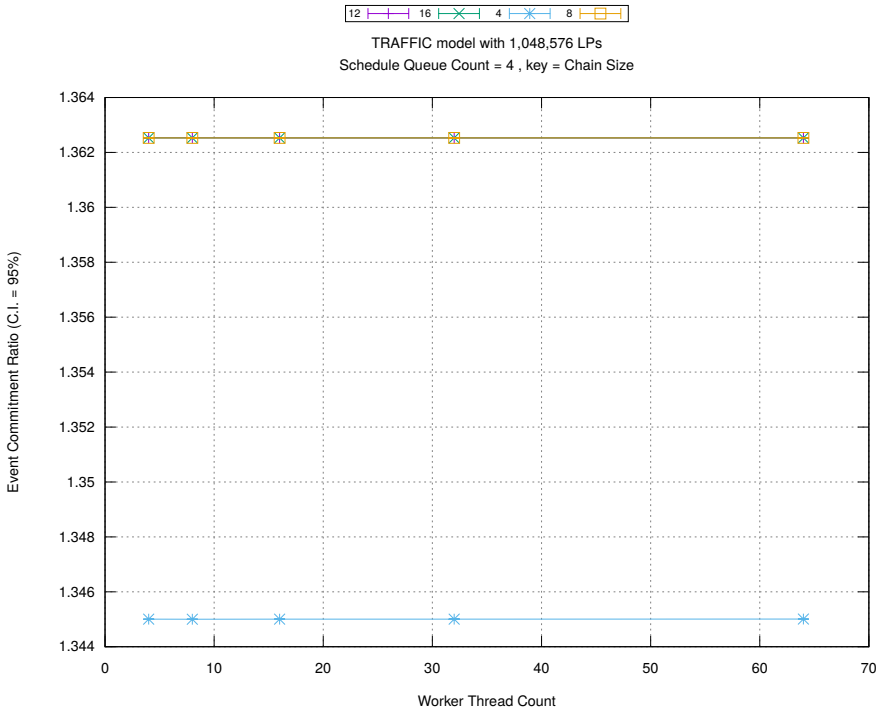
Figure A.38: traffic 1m/plots/chains/threads vs chainsize key count 1



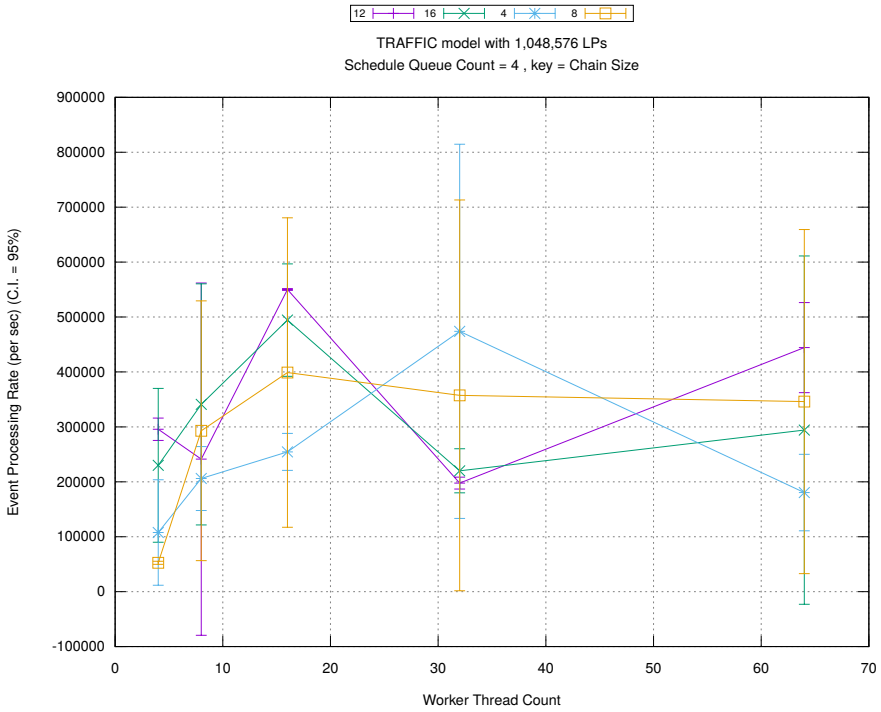
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

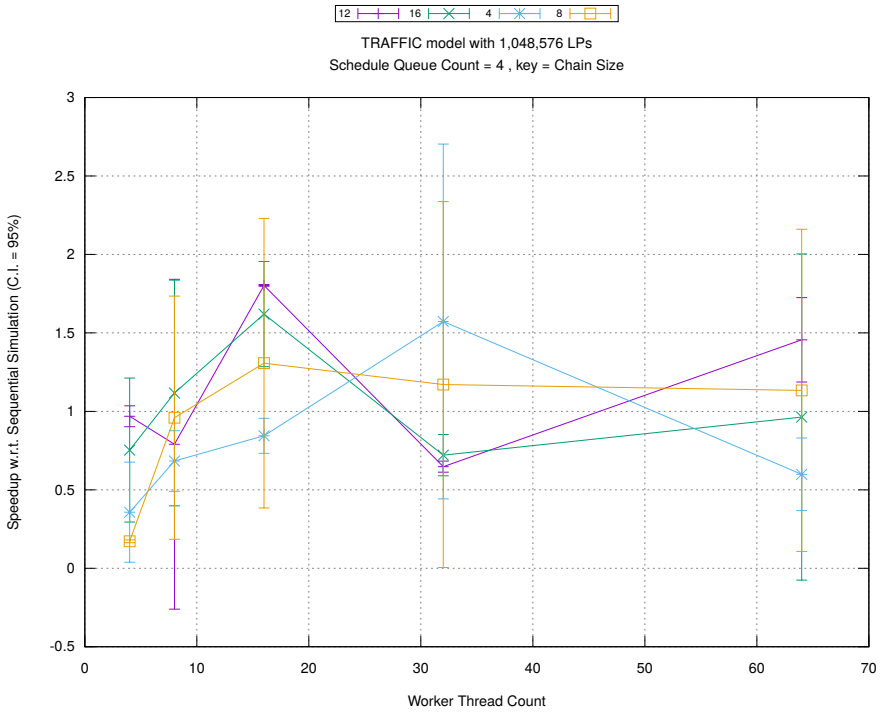


(b) Event Commitment Ratio

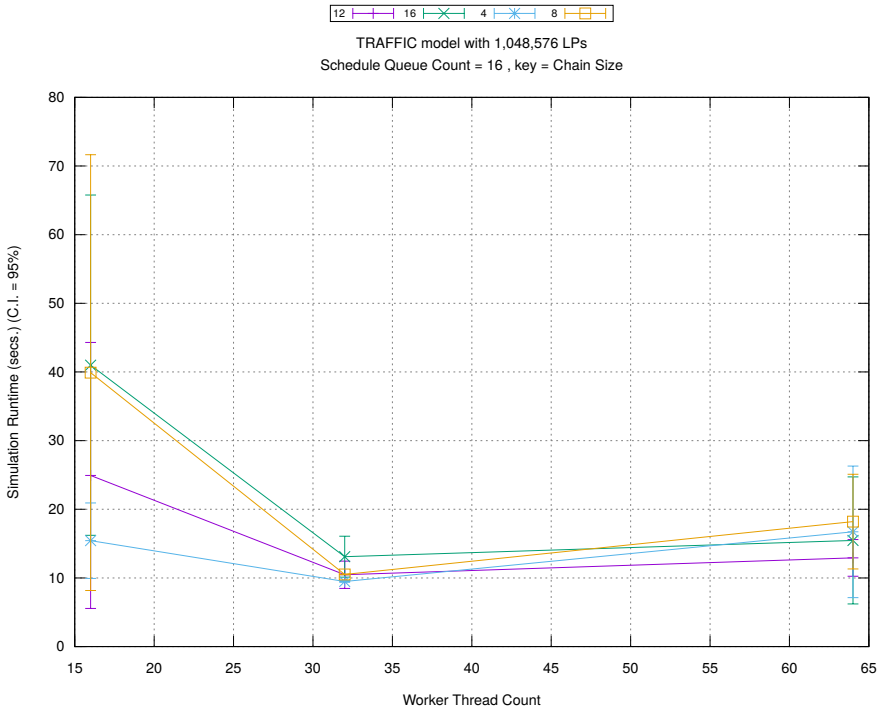


(c) Event Processing Rate (per second)

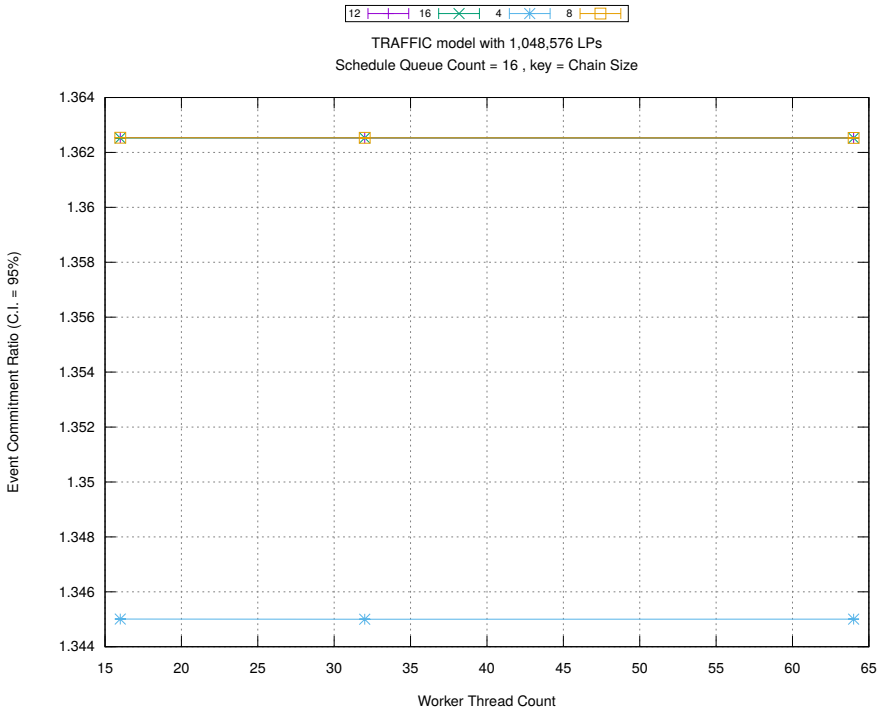
Figure A.39: traffic 1m/plots/chains/threads vs chainsize key count 4



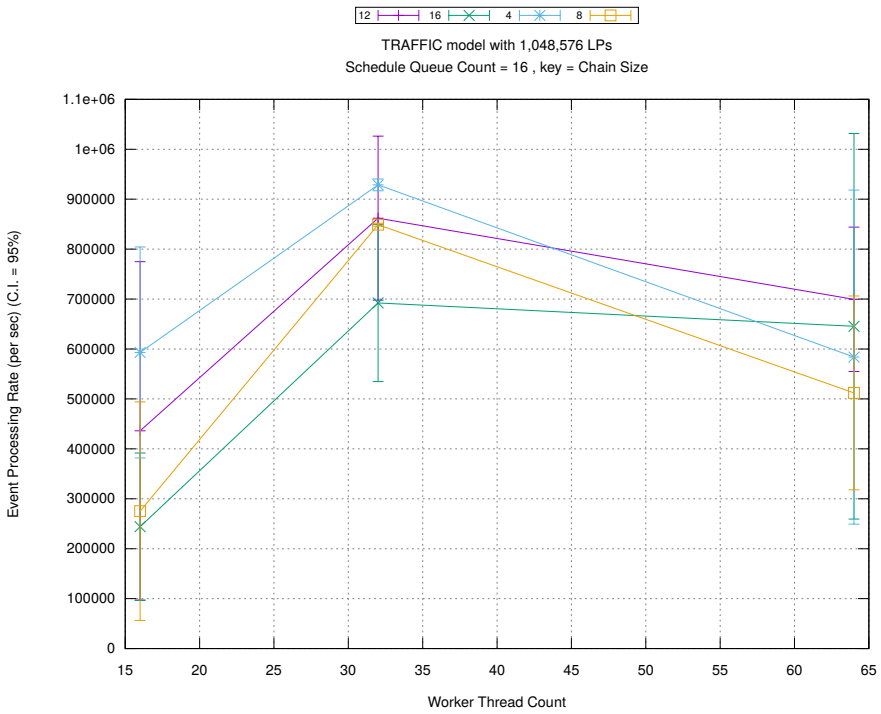
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

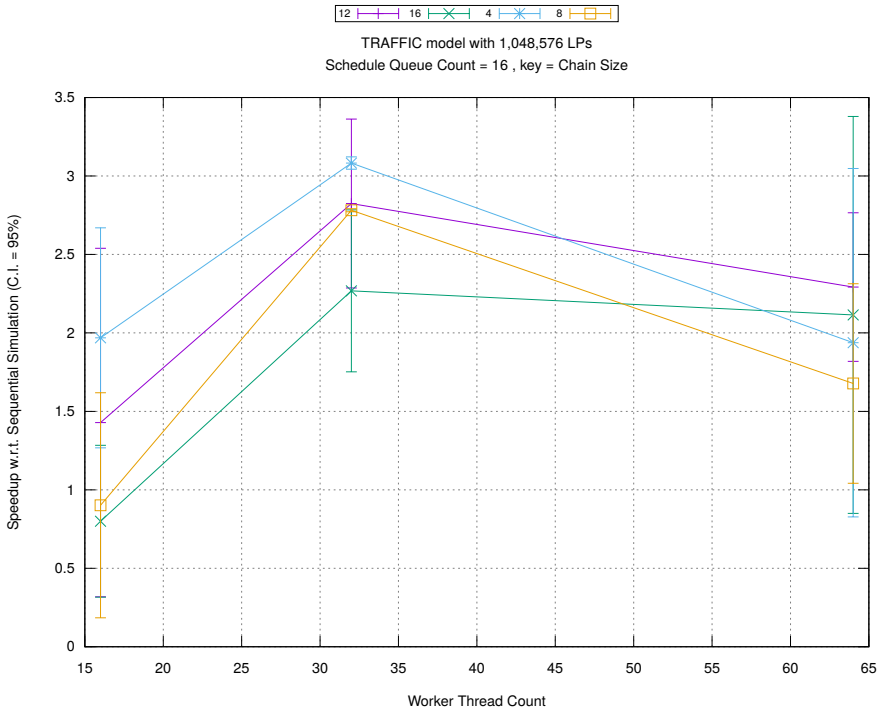


(b) Event Commitment Ratio

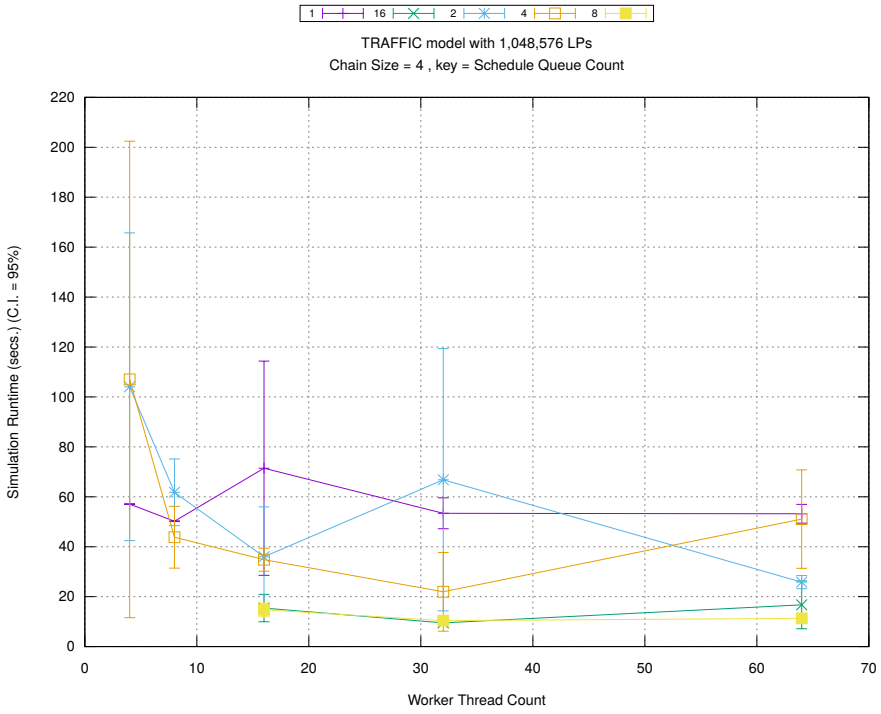


(c) Event Processing Rate (per second)

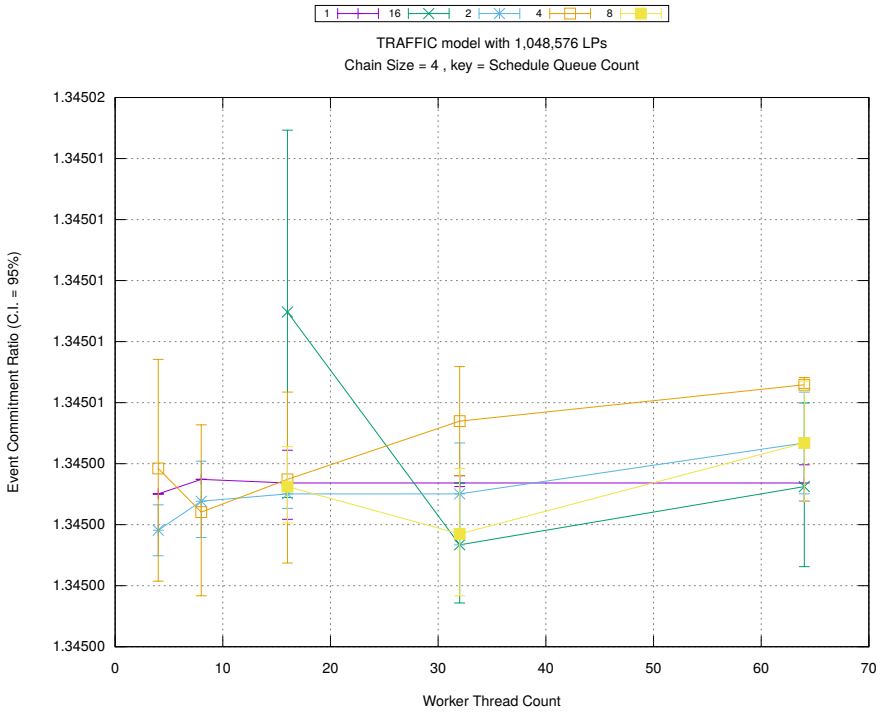
Figure A.40: traffic 1m/plots/chains/threads vs chainsize key count 16



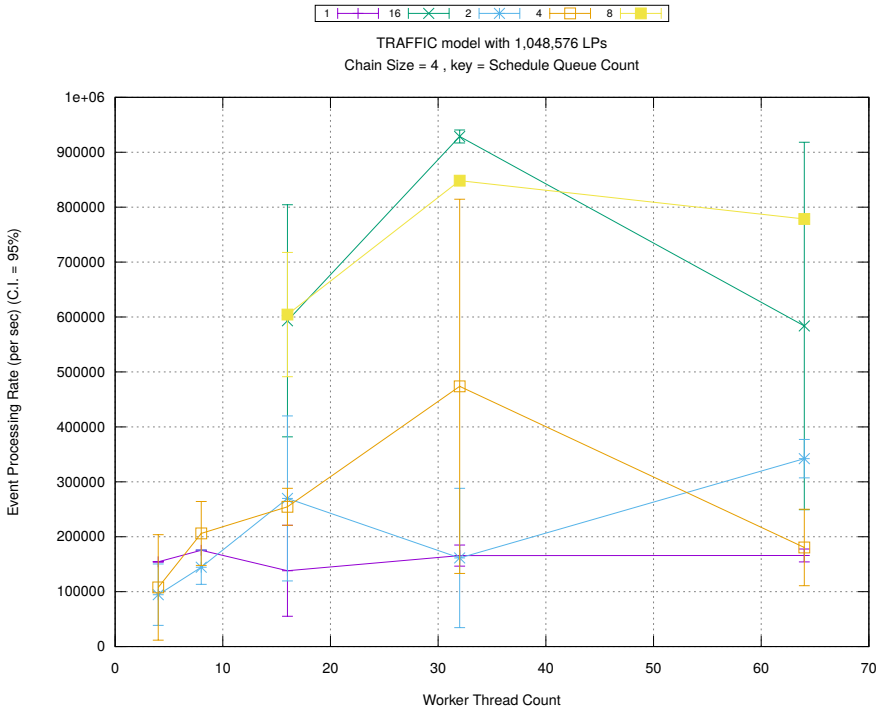
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

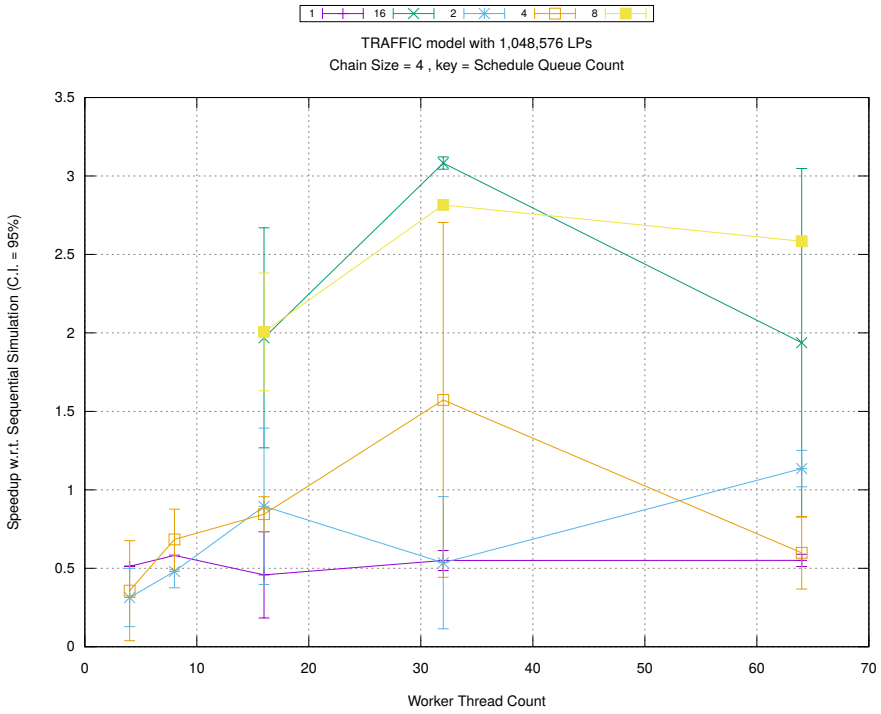


(b) Event Commitment Ratio

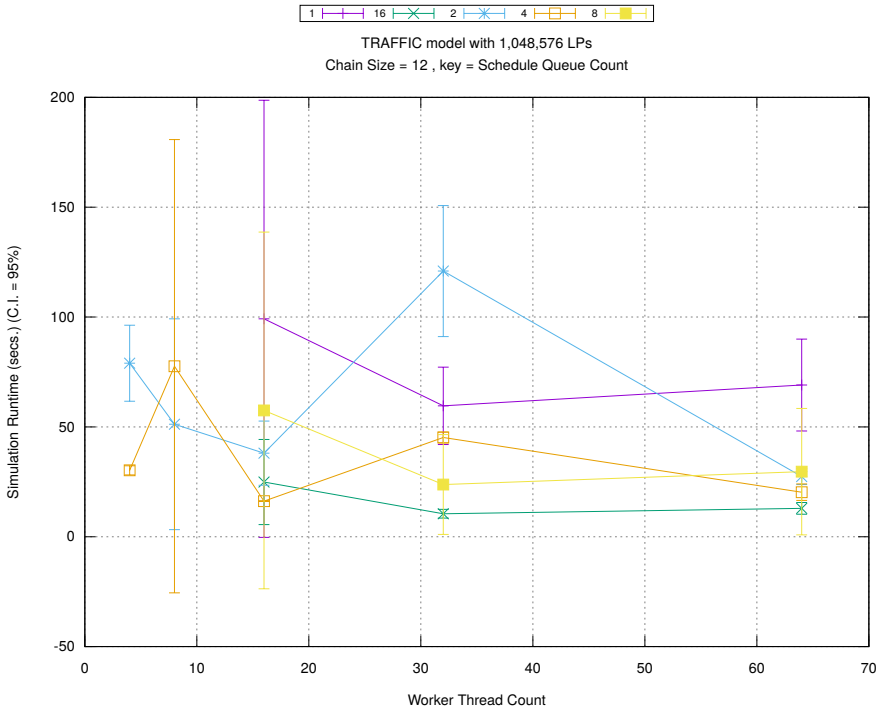


(c) Event Processing Rate (per second)

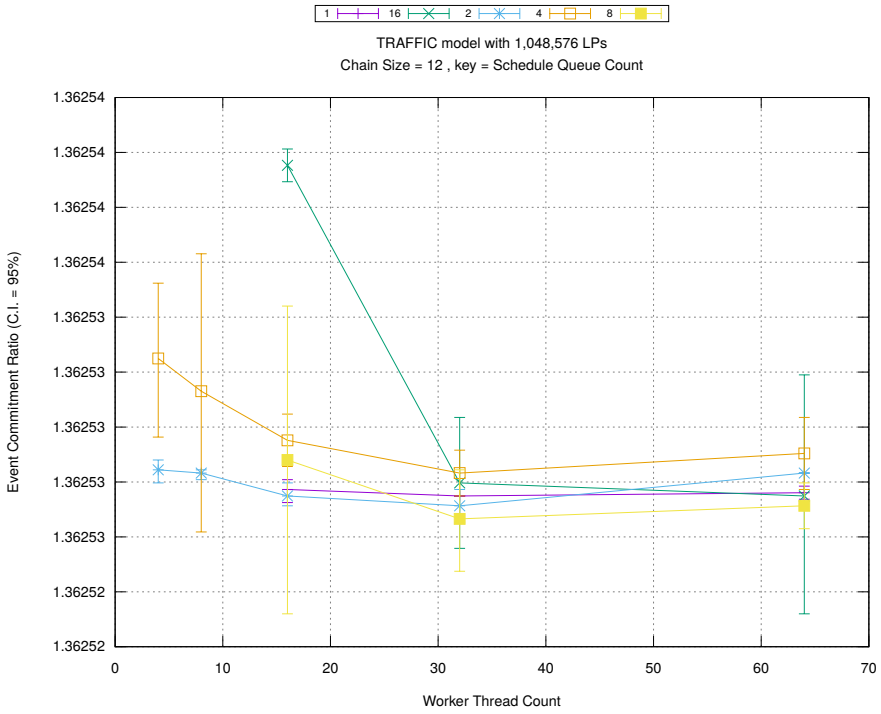
Figure A.41: traffic 1m/plots/chains/threads vs count key chainsize 4



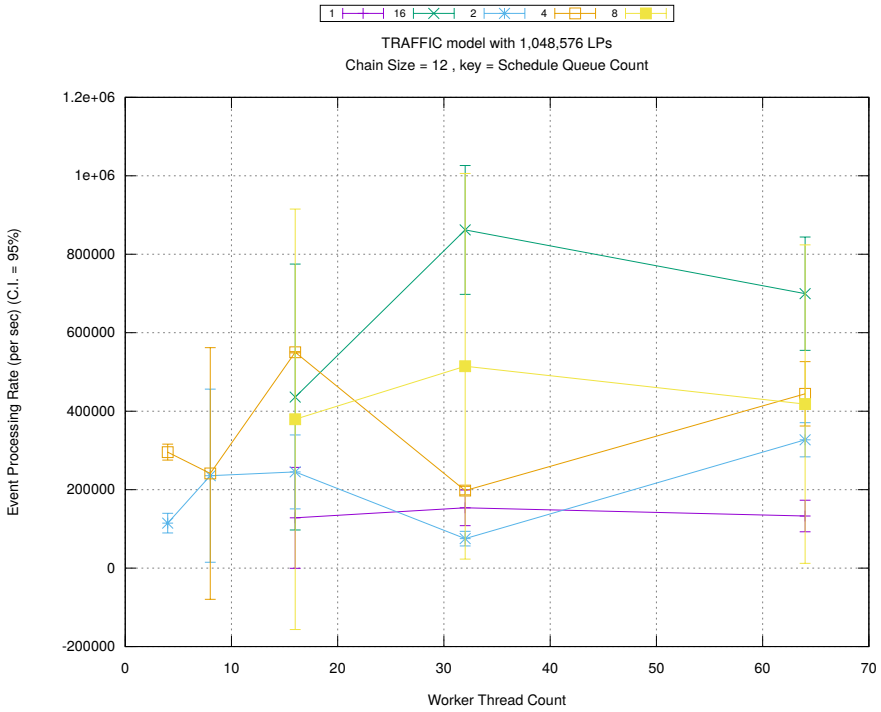
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

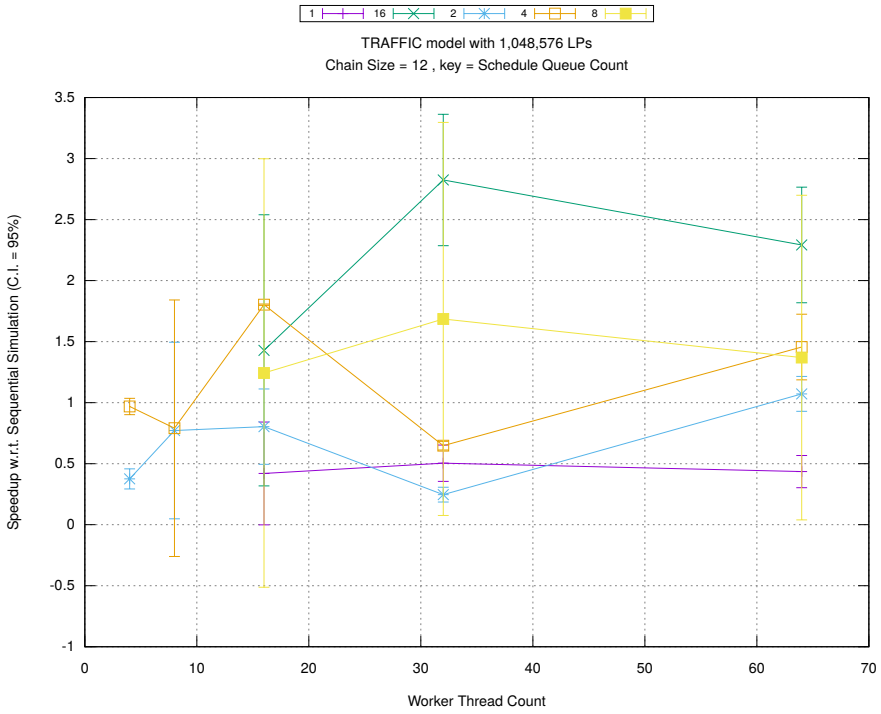


(b) Event Commitment Ratio

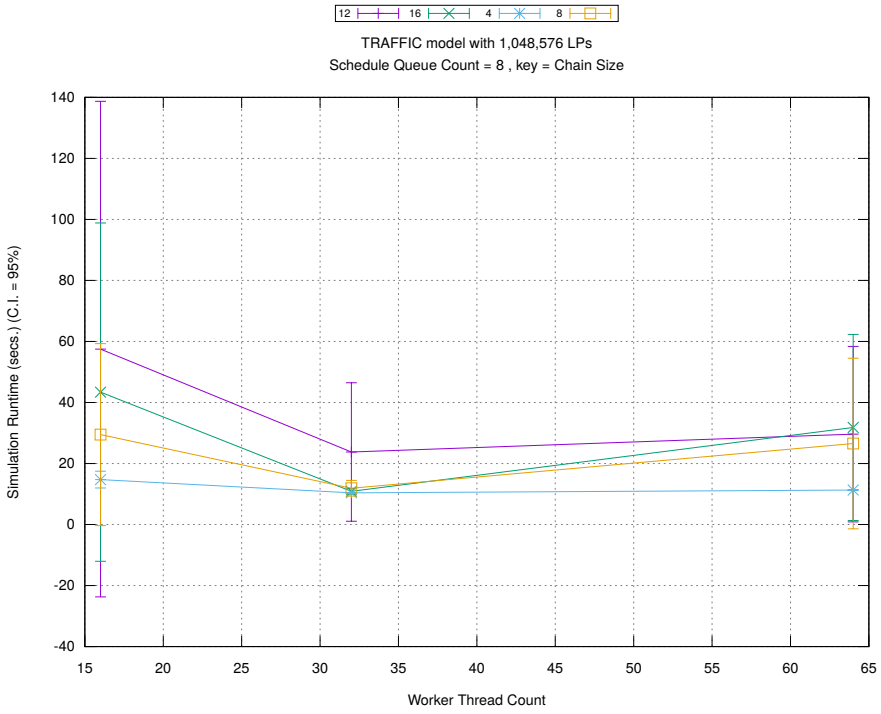


(c) Event Processing Rate (per second)

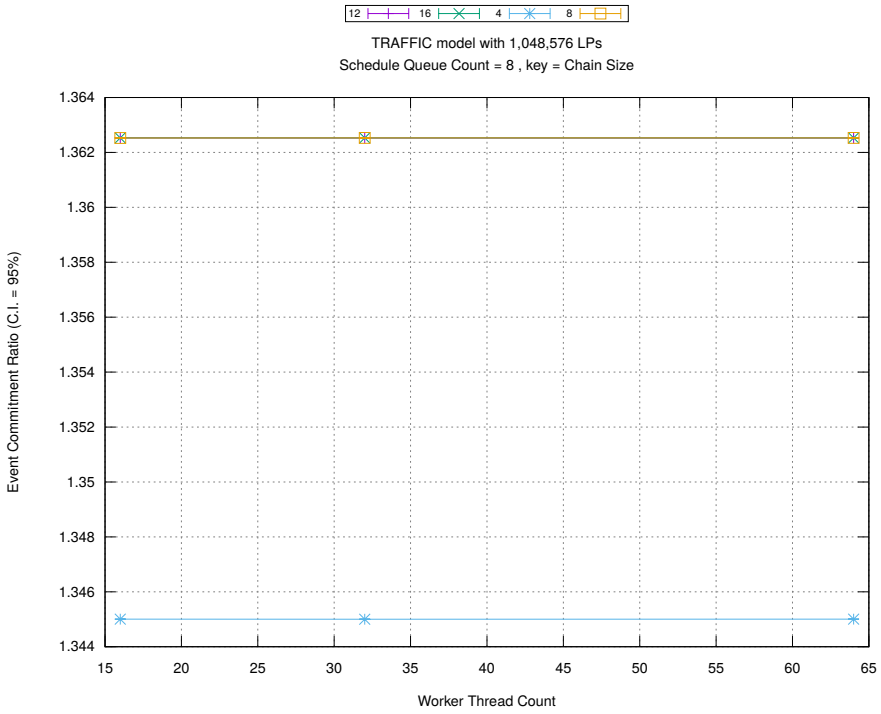
Figure A.42: traffic 1m/plots/chains/threads vs count key chainsize 12



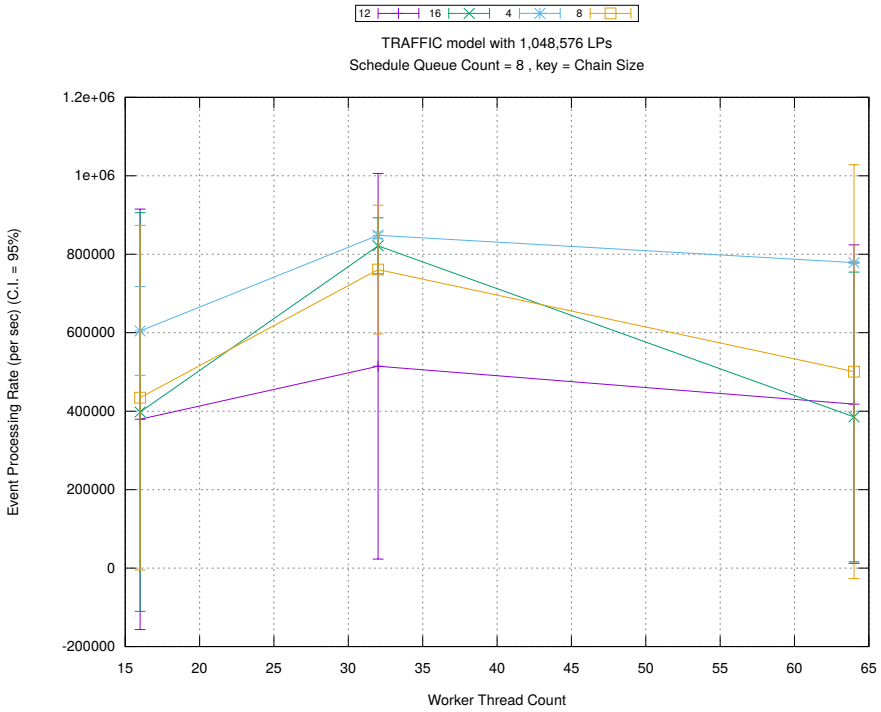
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

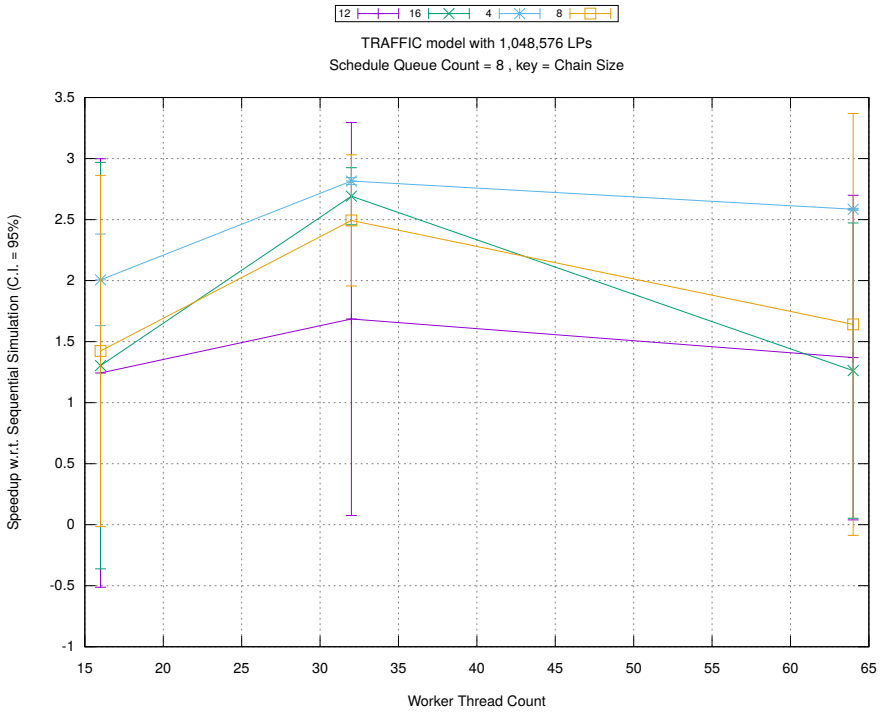


(b) Event Commitment Ratio

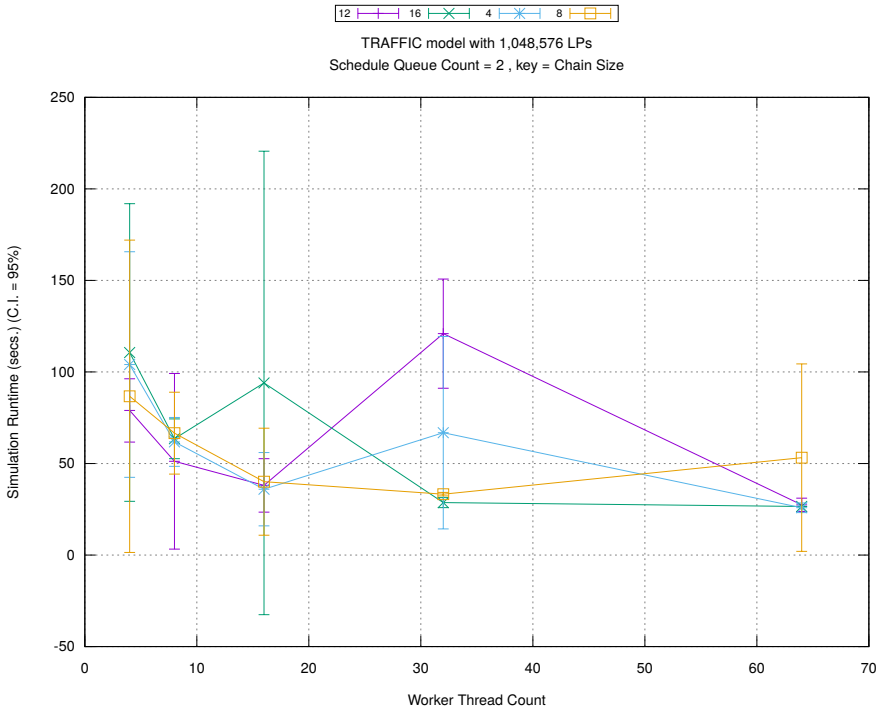


(c) Event Processing Rate (per second)

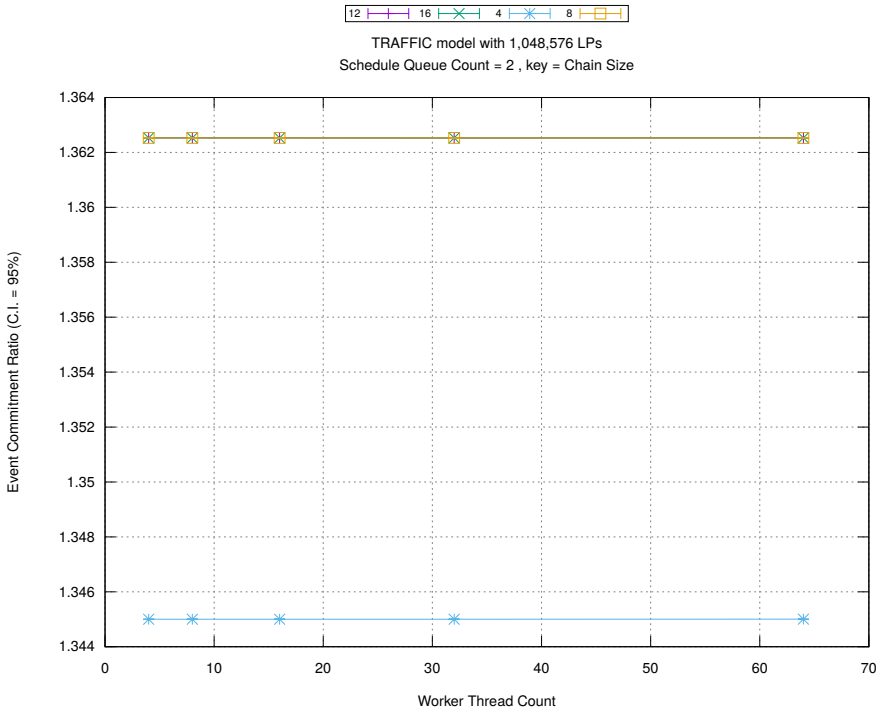
Figure A.43: traffic 1m/plots/chains/threads vs chainsize key count 8



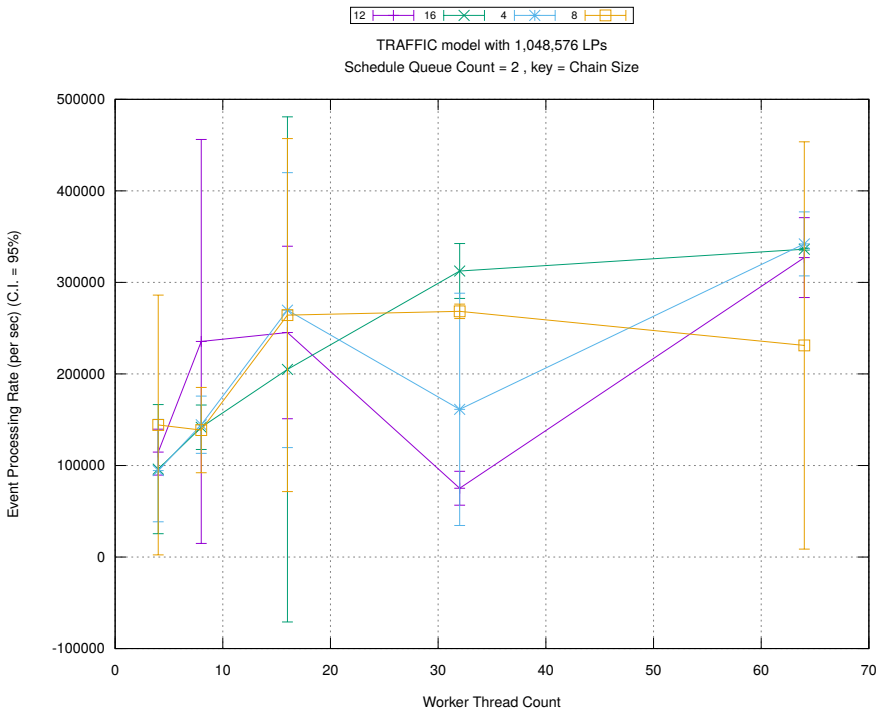
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

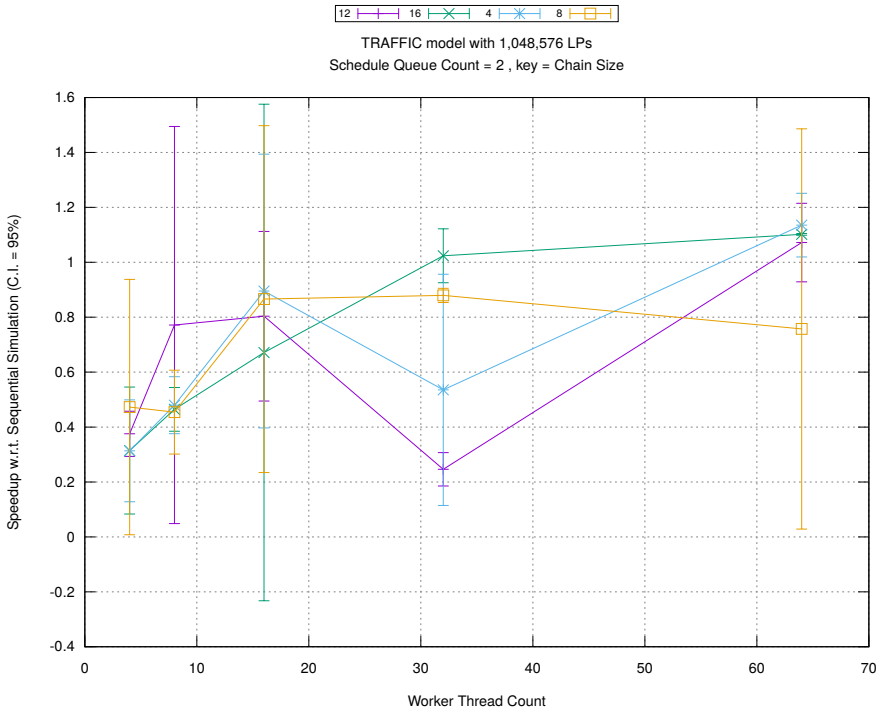


(b) Event Commitment Ratio

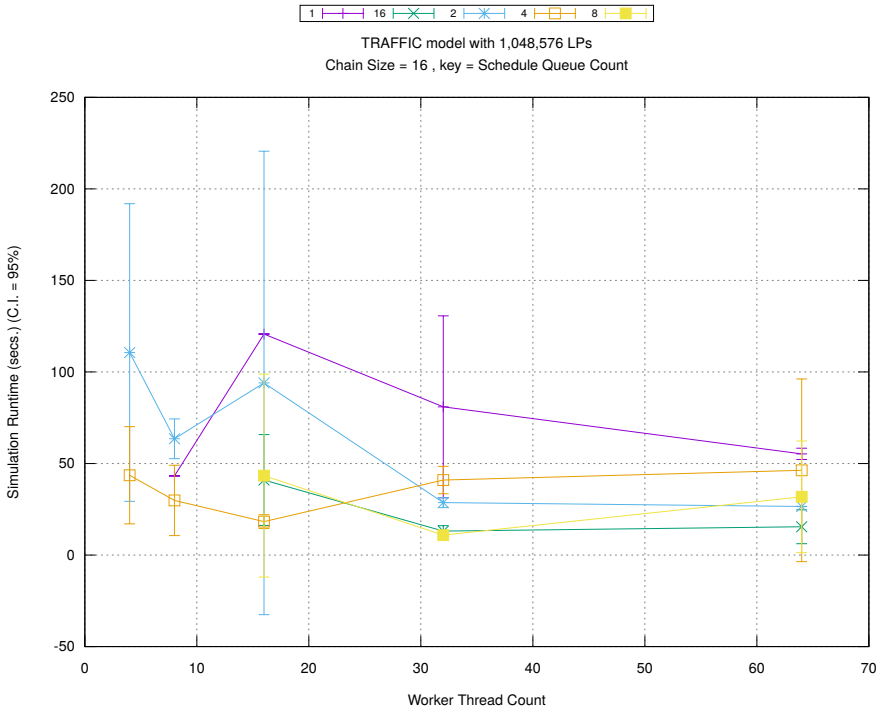


(c) Event Processing Rate (per second)

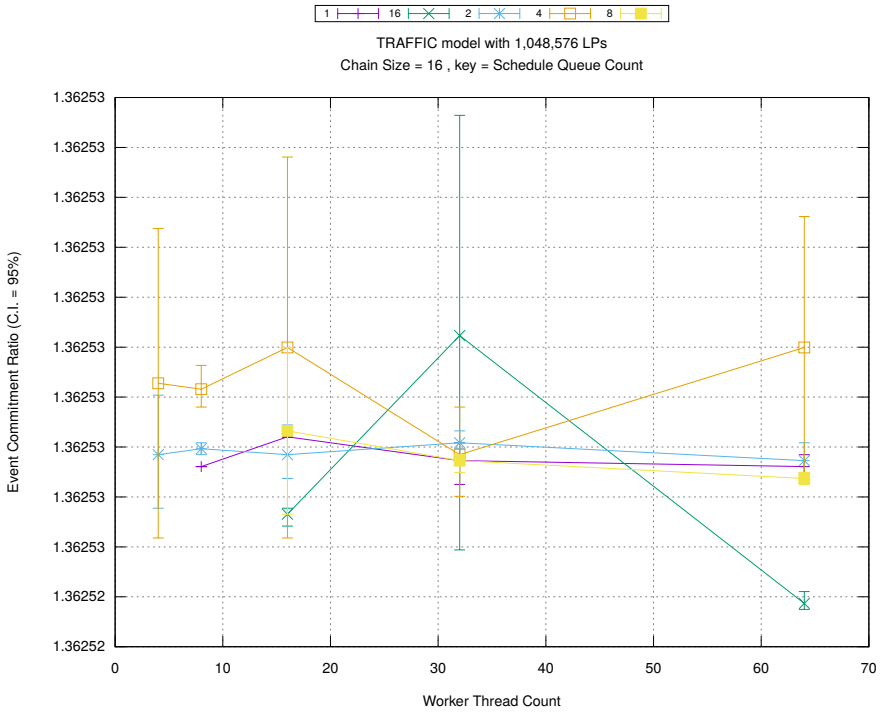
Figure A.44: traffic 1m/plots/chains/threads vs chainsize key count 2



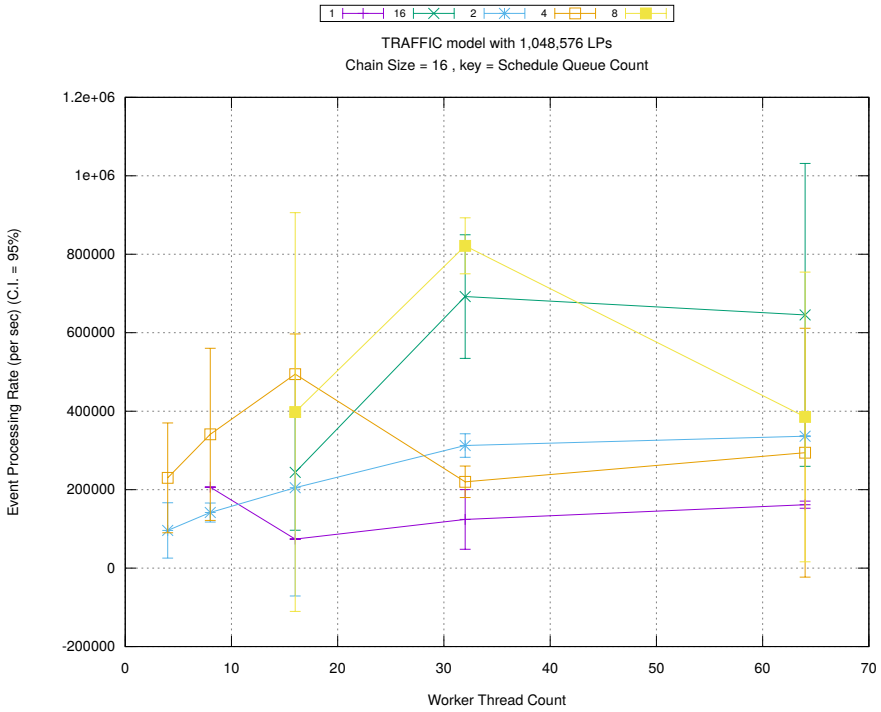
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

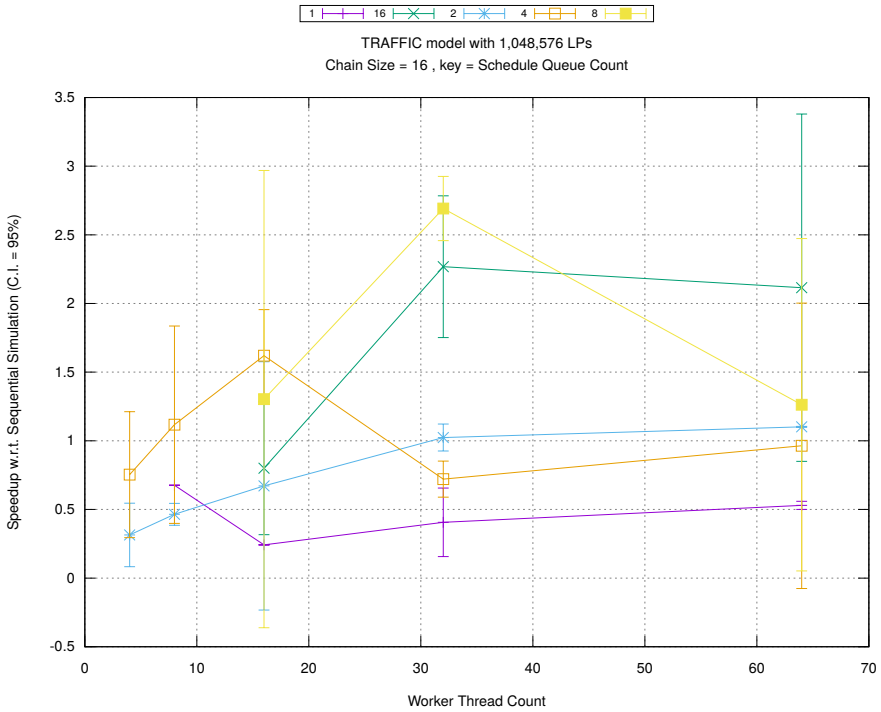


(b) Event Commitment Ratio

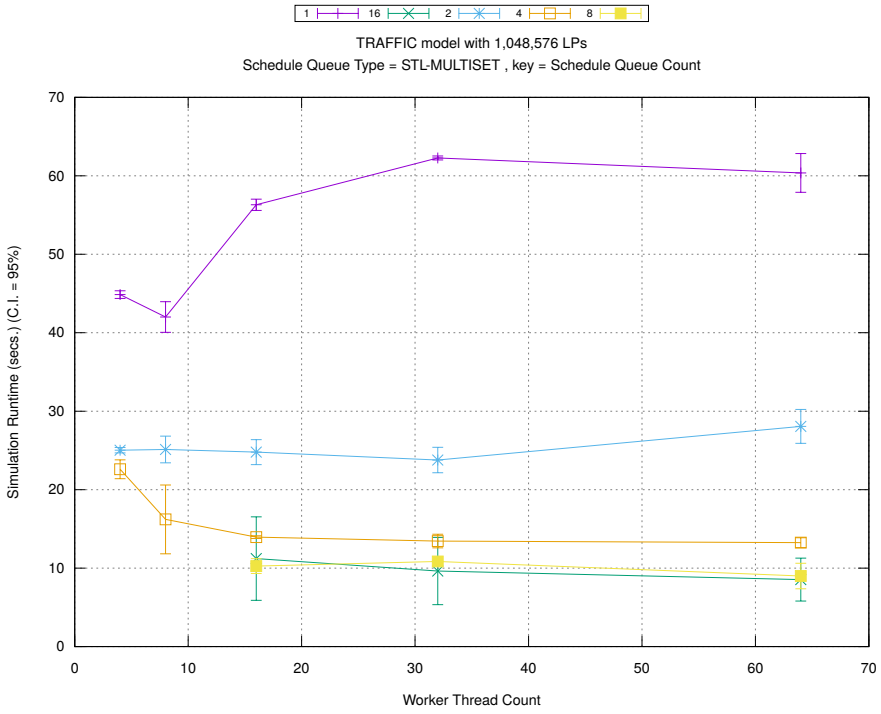


(c) Event Processing Rate (per second)

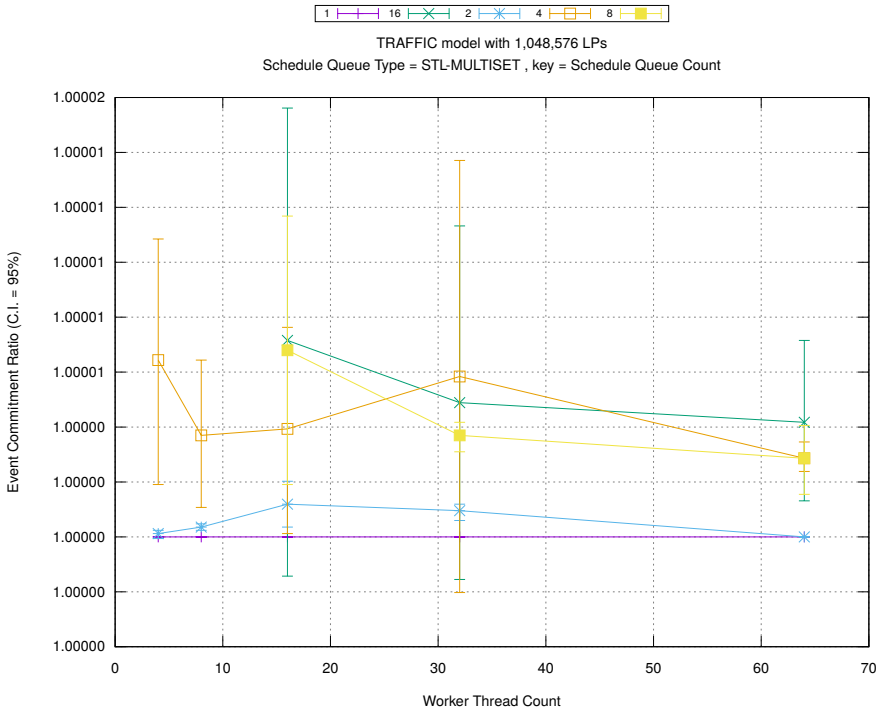
Figure A.45: traffic 1m/plots/chains/threads vs count key chainsize 16



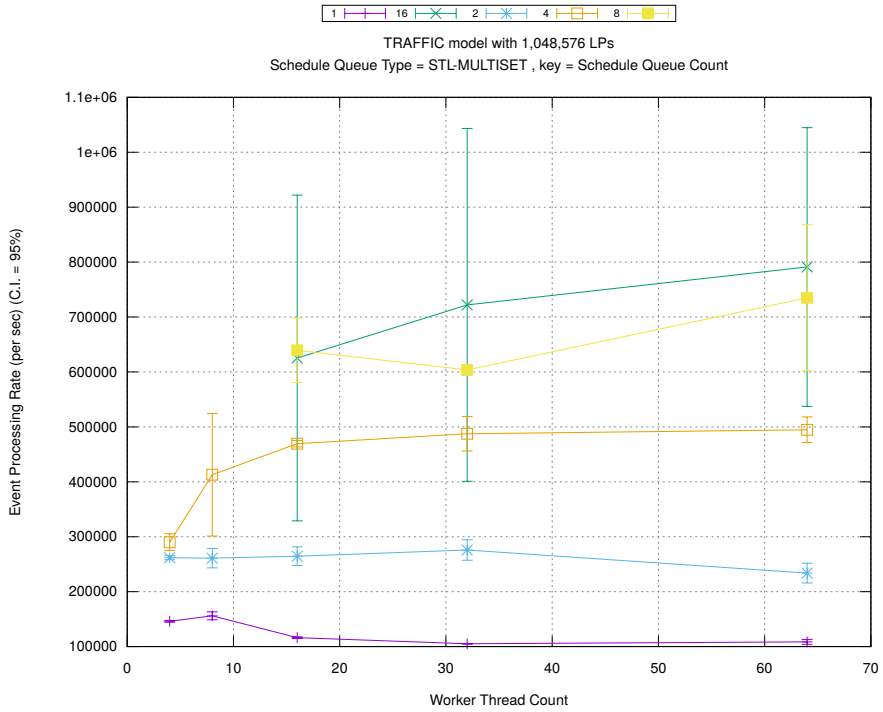
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

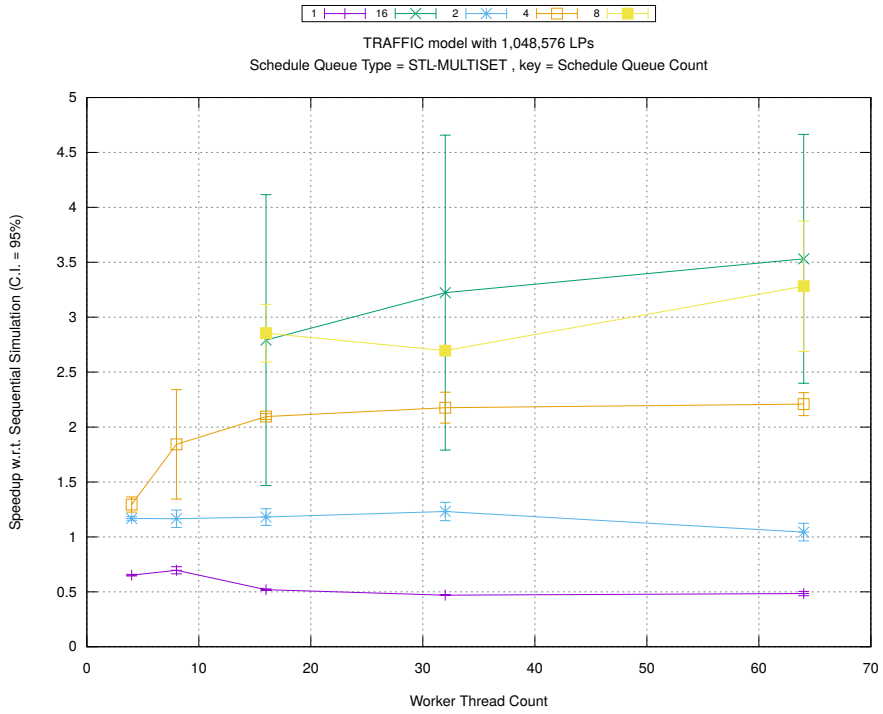


(b) Event Commitment Ratio

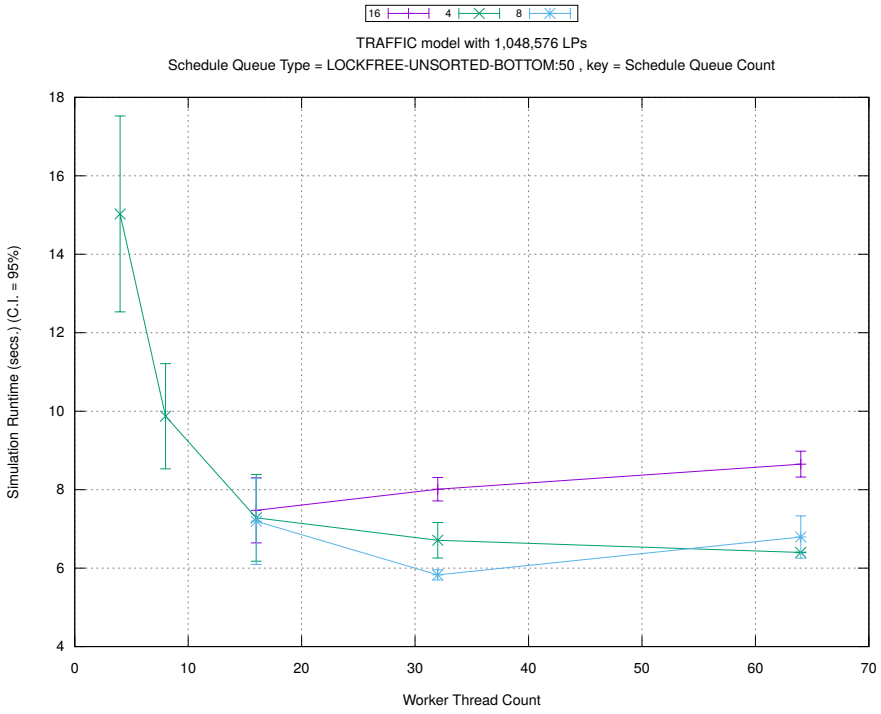


(c) Event Processing Rate (per second)

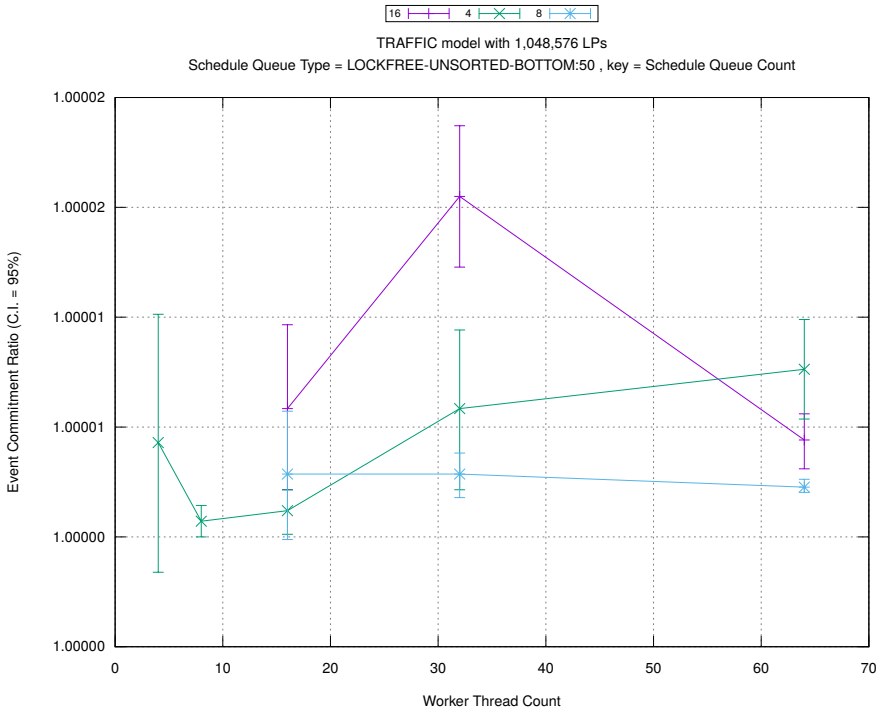
Figure A.46: traffic 1m/plots/scheduleq/threads vs count key type stl-multiset



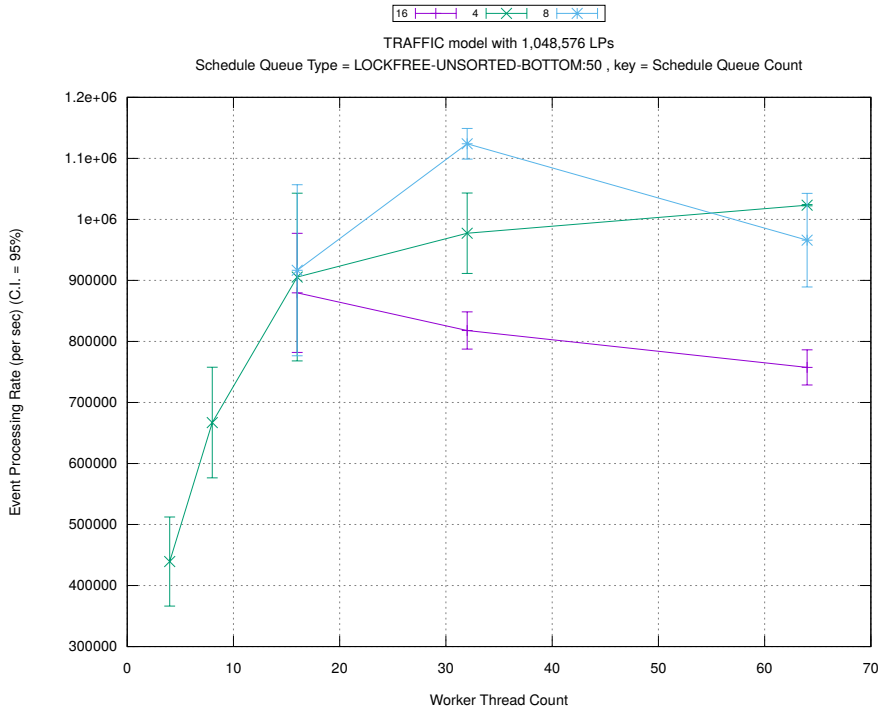
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

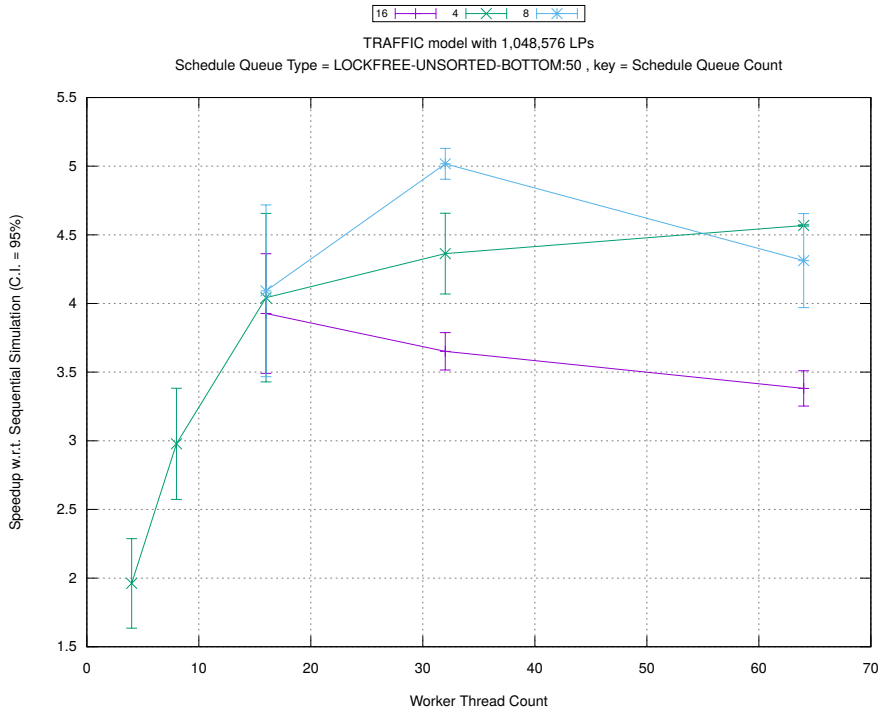


(b) Event Commitment Ratio

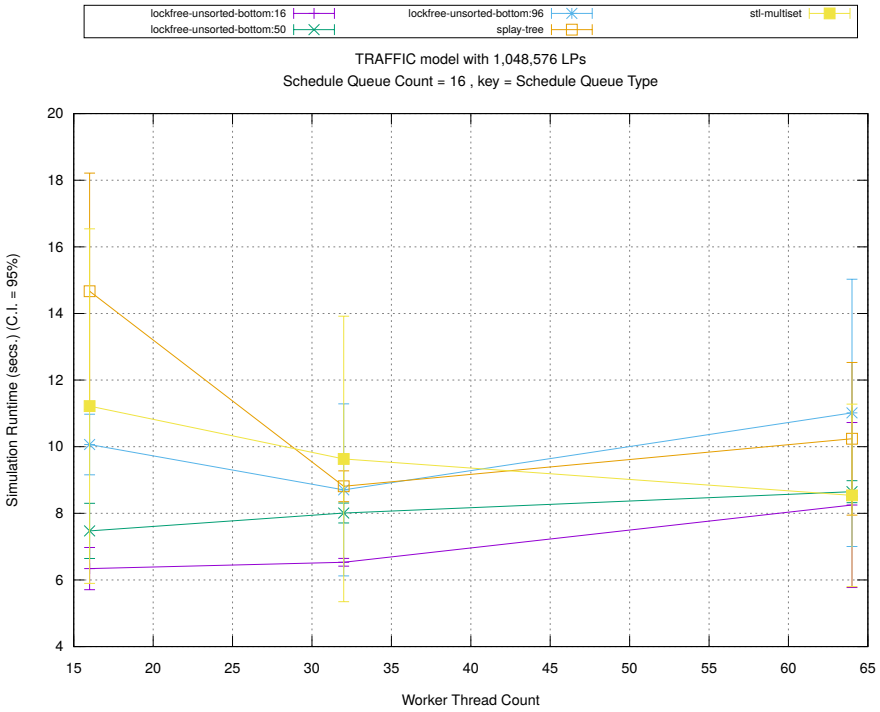


(c) Event Processing Rate (per second)

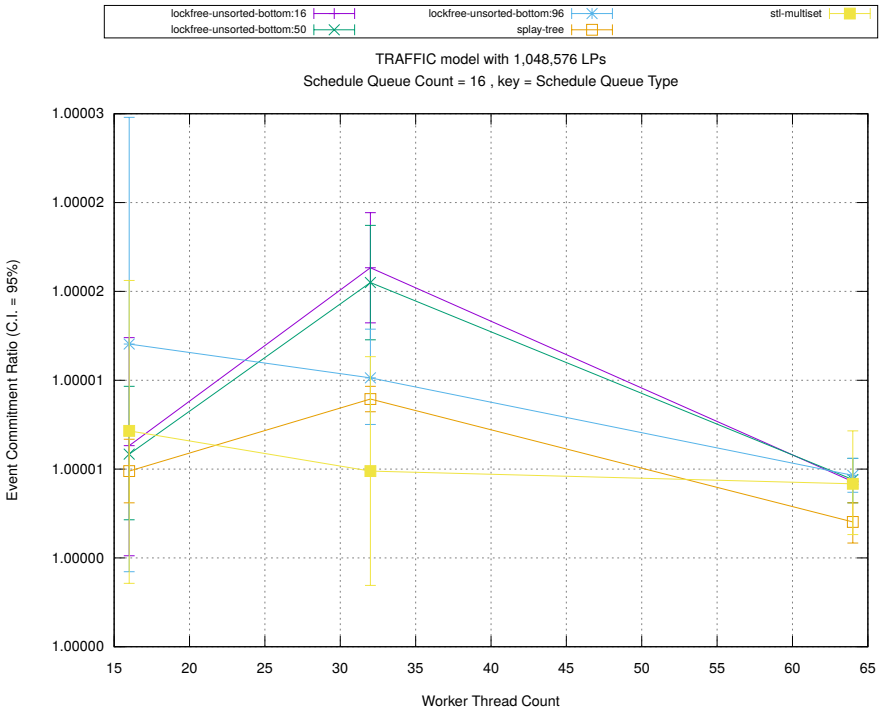
Figure A.47: traffic 1m/plots/scheduleq/threads vs count key type lockfree-unsorted-bottom 50



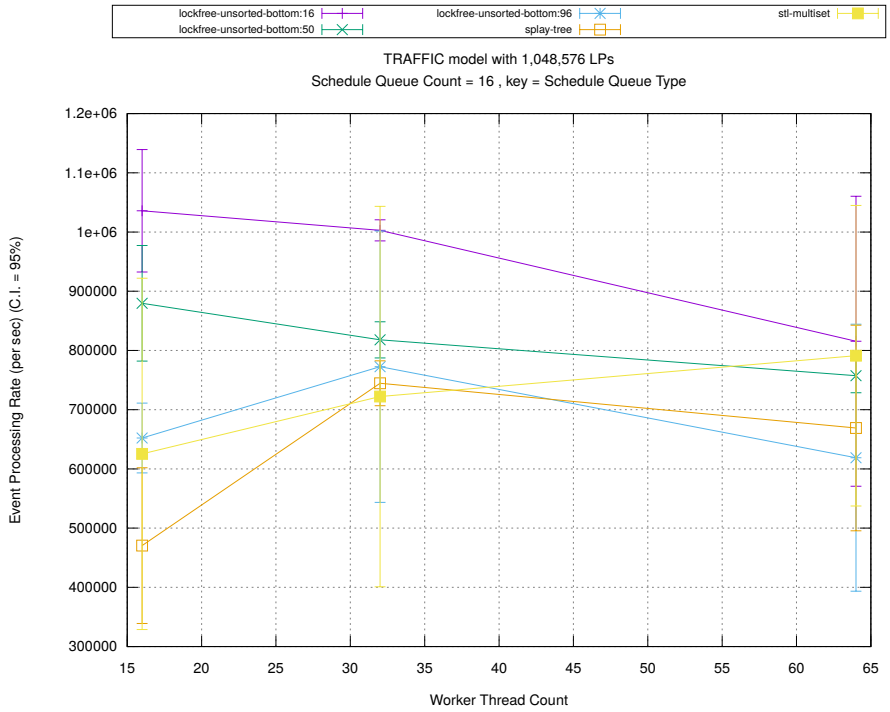
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

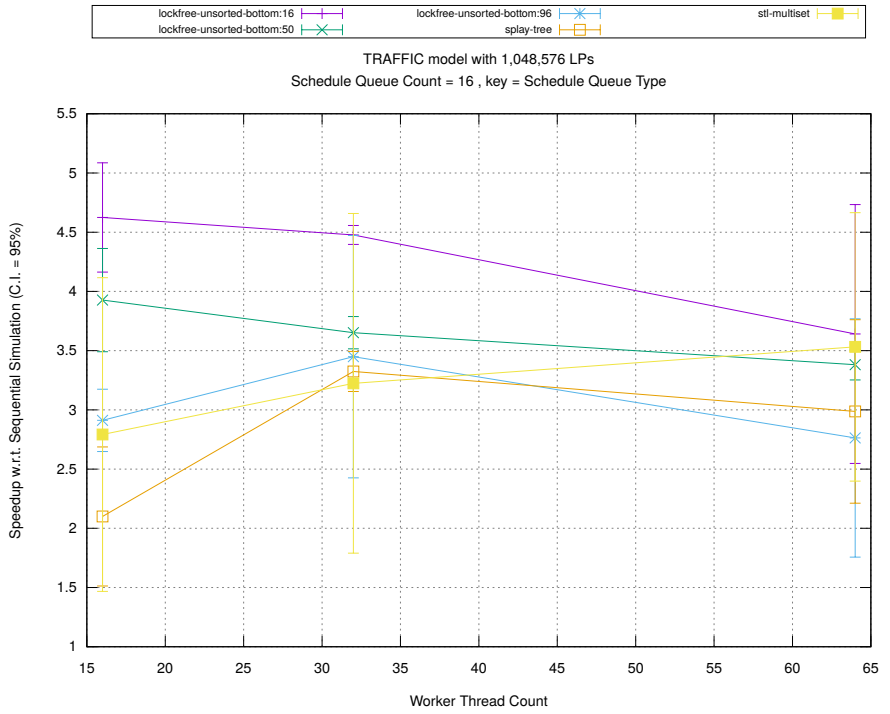


(b) Event Commitment Ratio

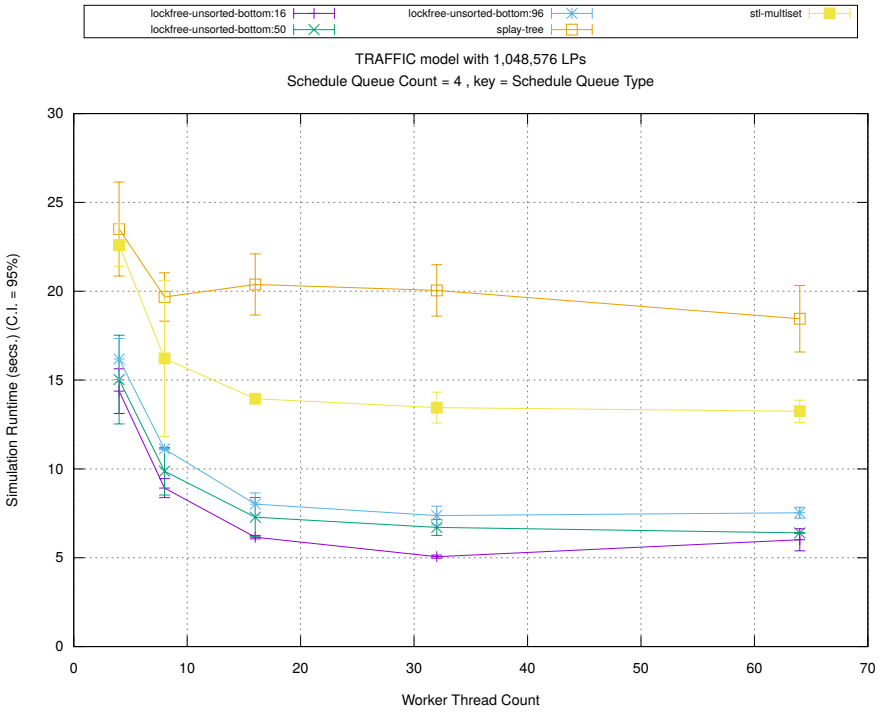


(c) Event Processing Rate (per second)

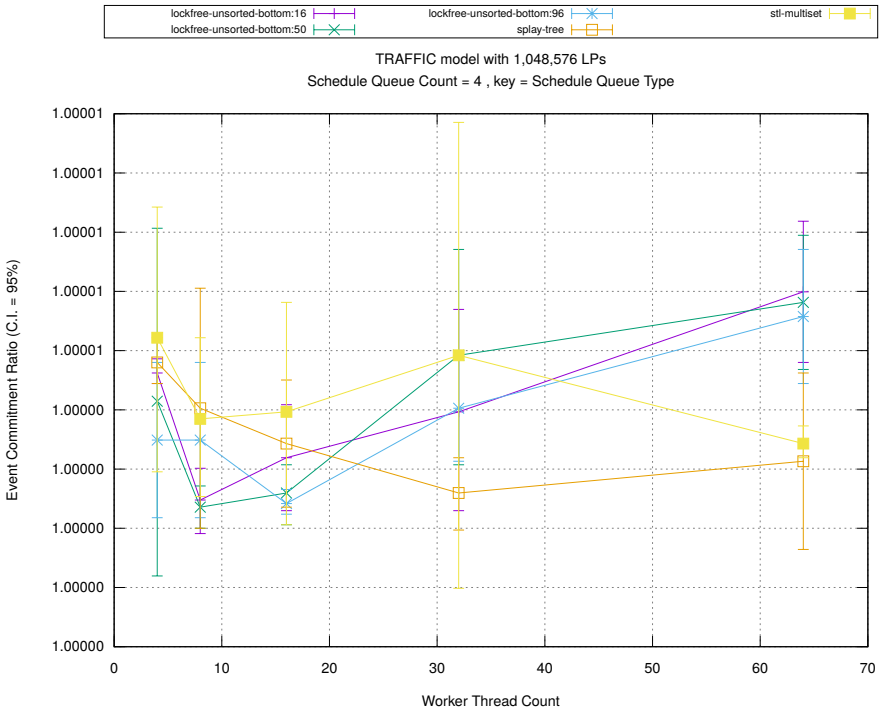
Figure A.48: traffic 1m/plots/scheduleq/threads vs type key count 16



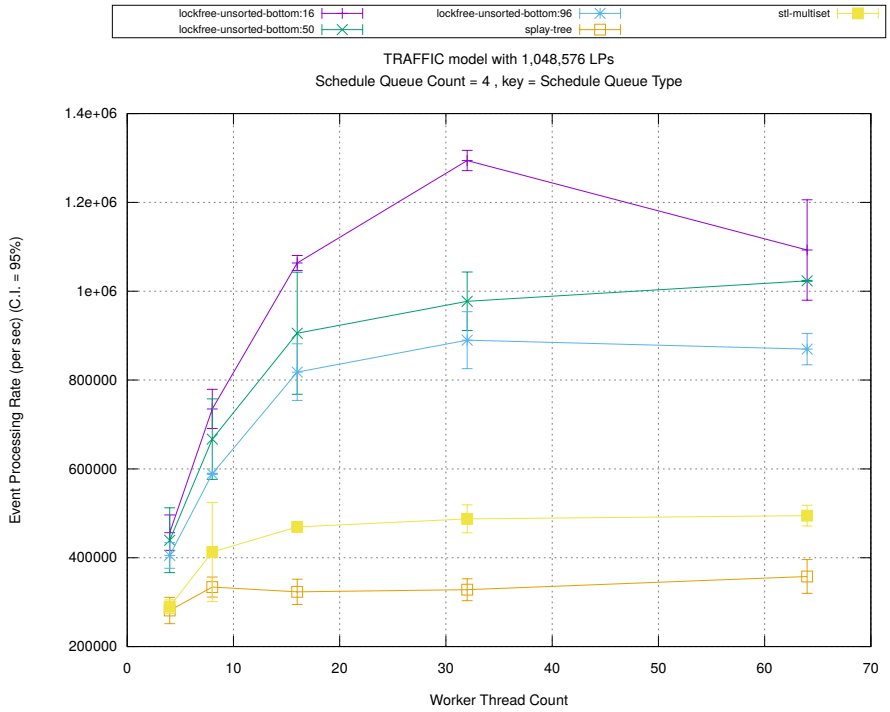
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

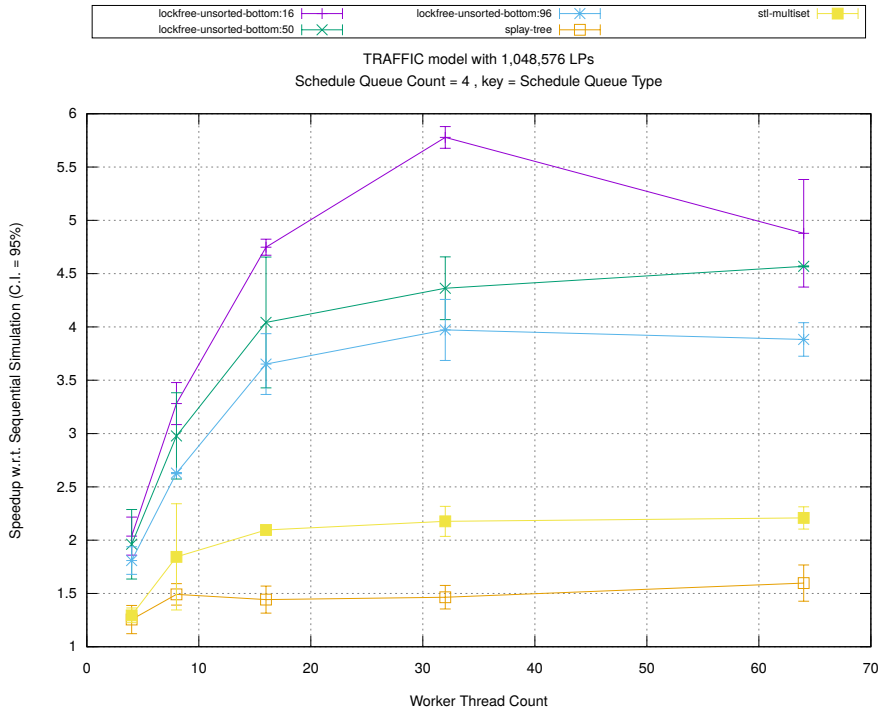


(b) Event Commitment Ratio

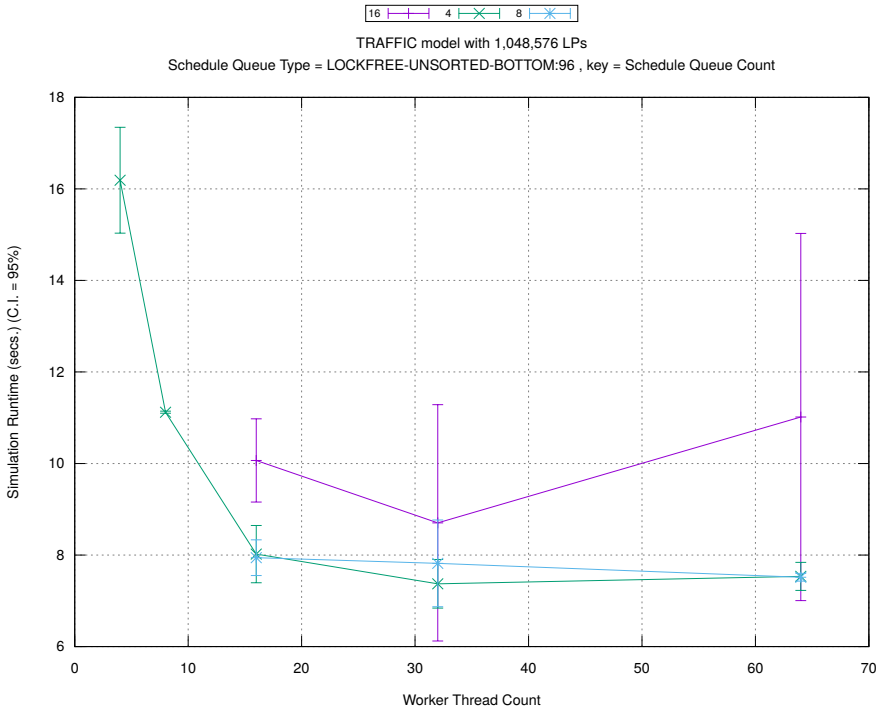


(c) Event Processing Rate (per second)

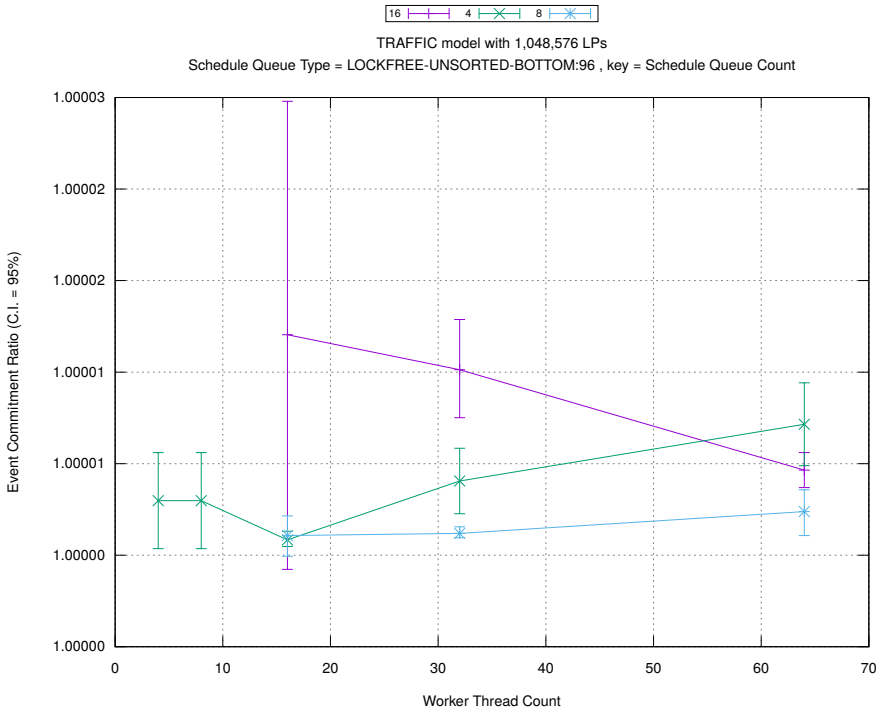
Figure A.49: traffic 1m/plots/scheduleq/threads vs type key count 4



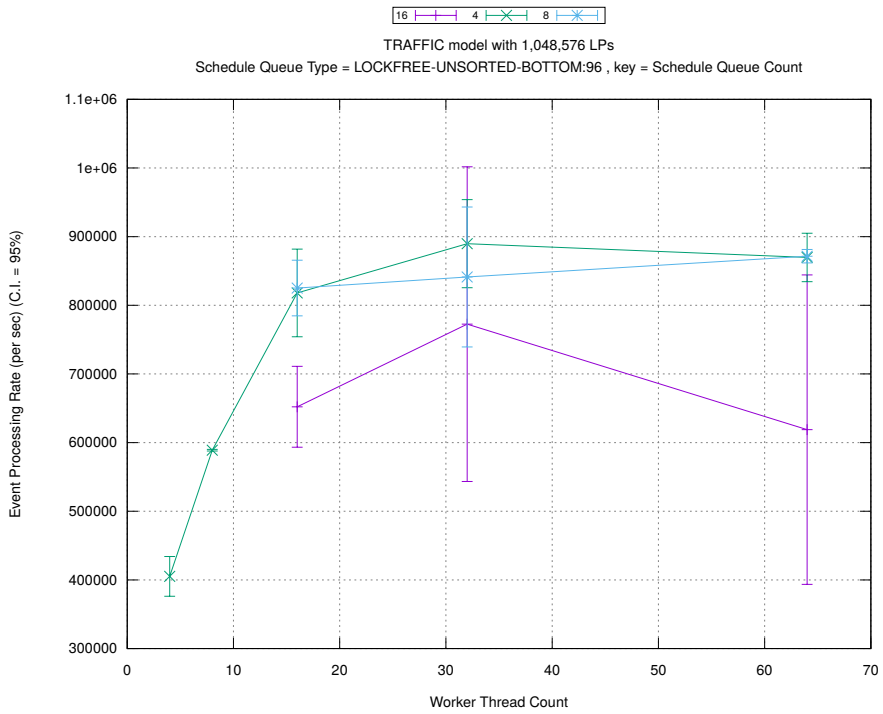
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

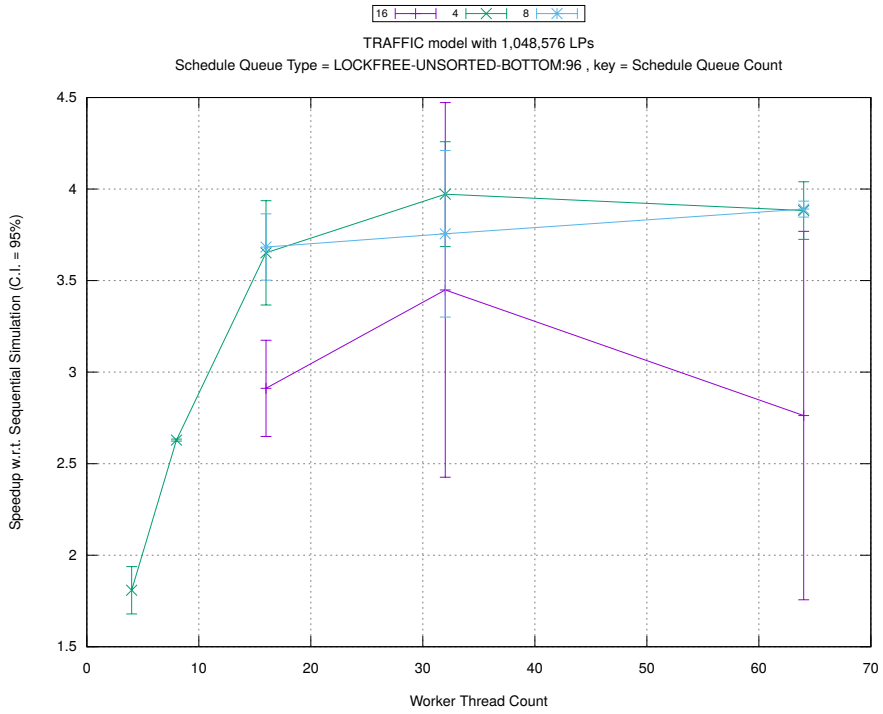


(b) Event Commitment Ratio

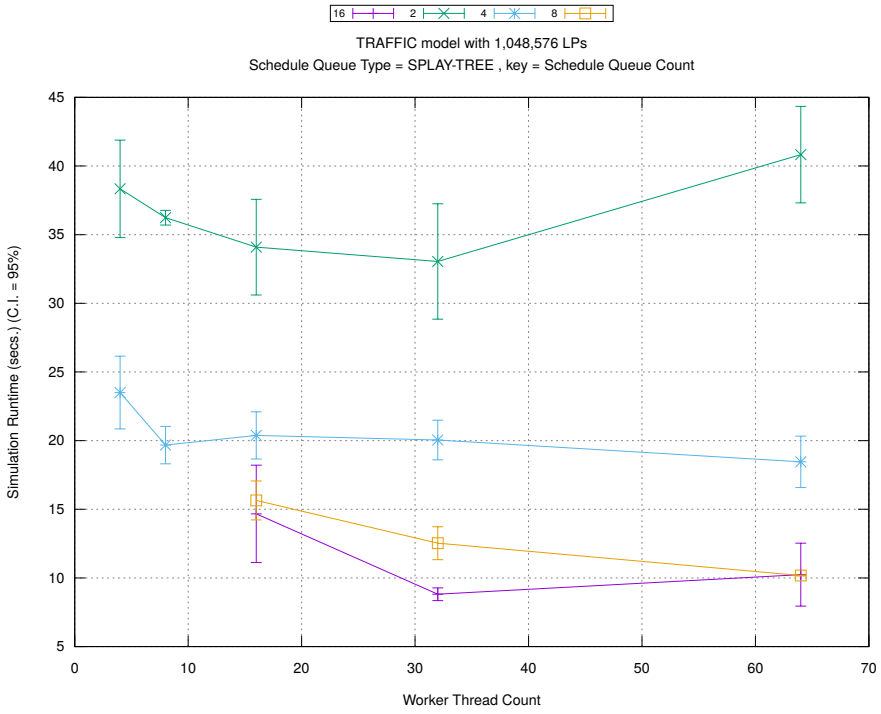


(c) Event Processing Rate (per second)

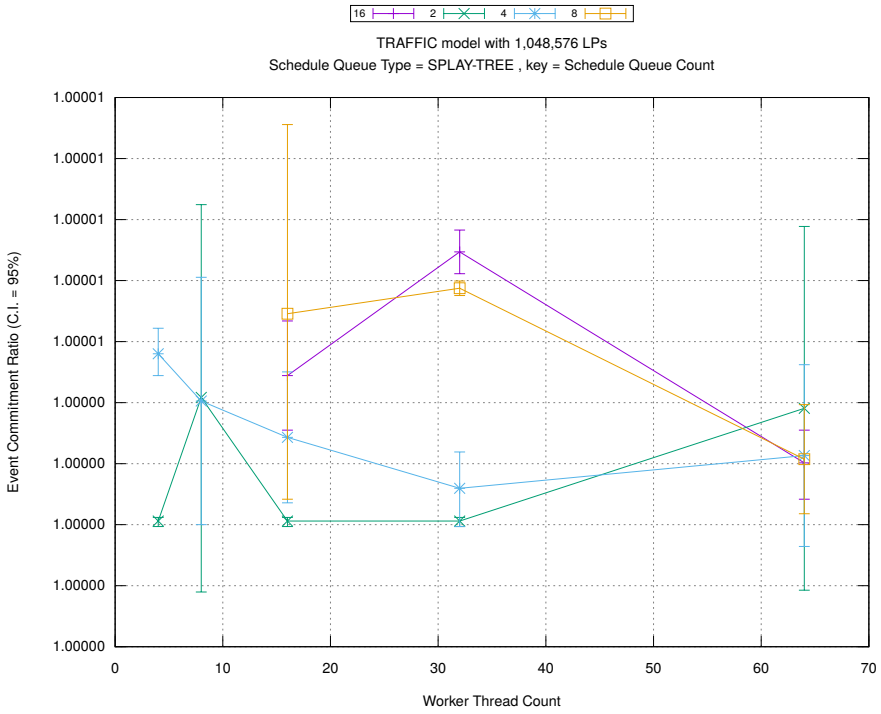
Figure A.50: traffic 1m/plots/scheduleq/threads vs count key type lockfree-unsorted-bottom 96



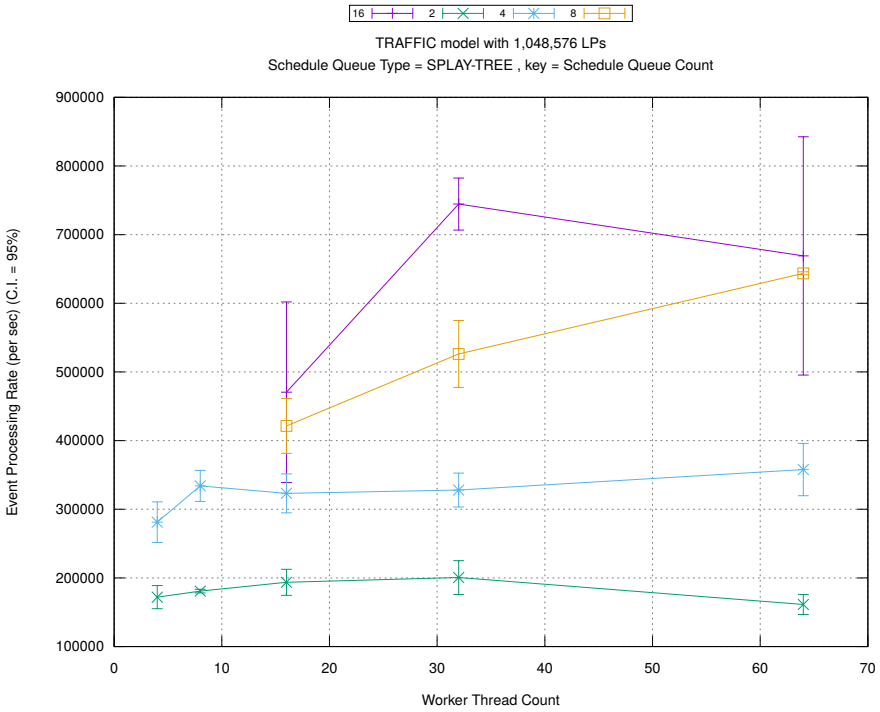
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

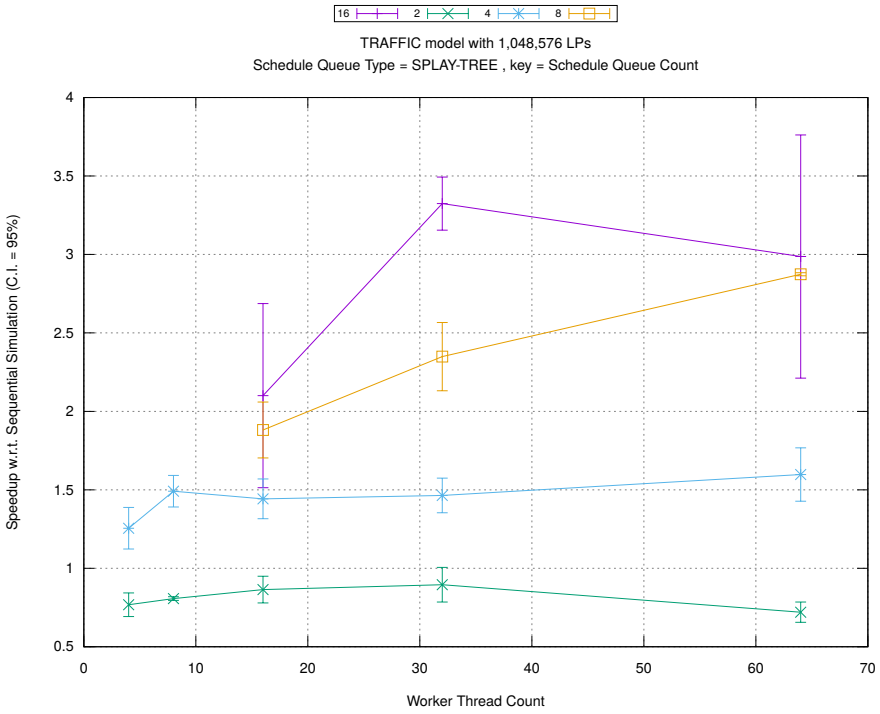


(b) Event Commitment Ratio

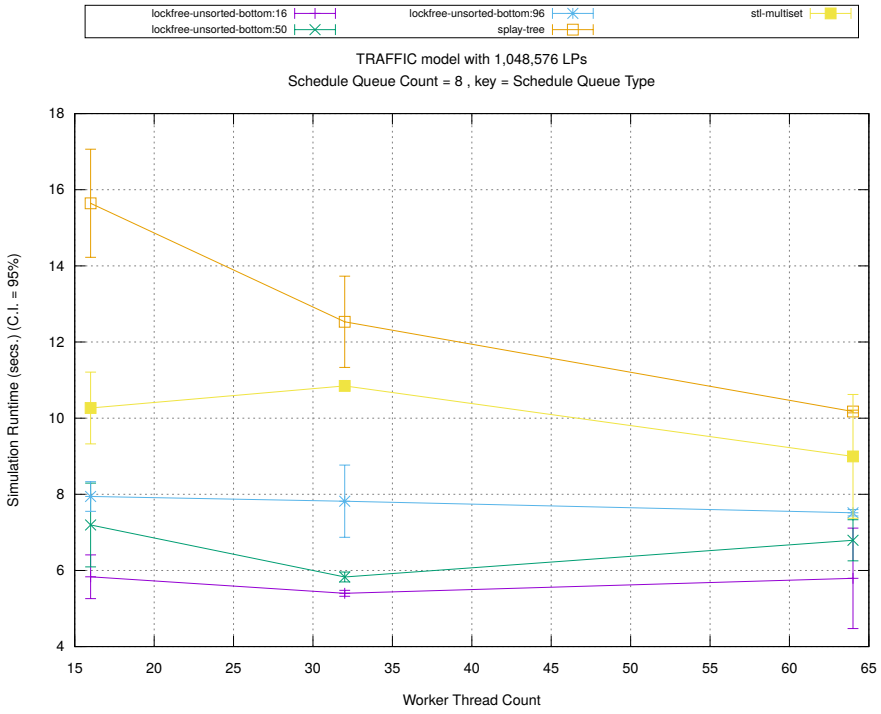


(c) Event Processing Rate (per second)

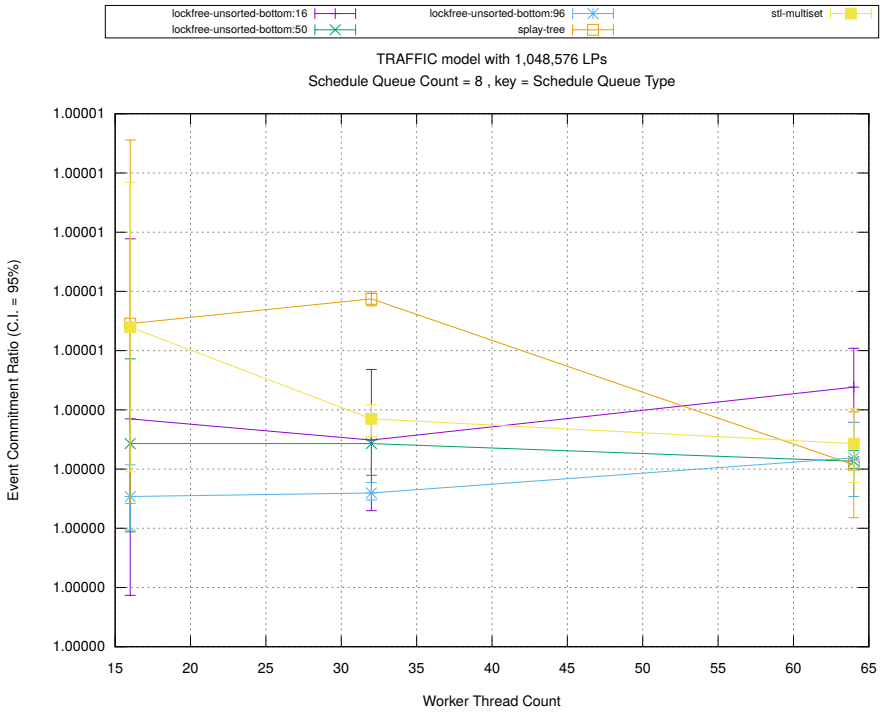
Figure A.51: traffic 1m/plots/scheduleq/threads vs count key type splay-tree



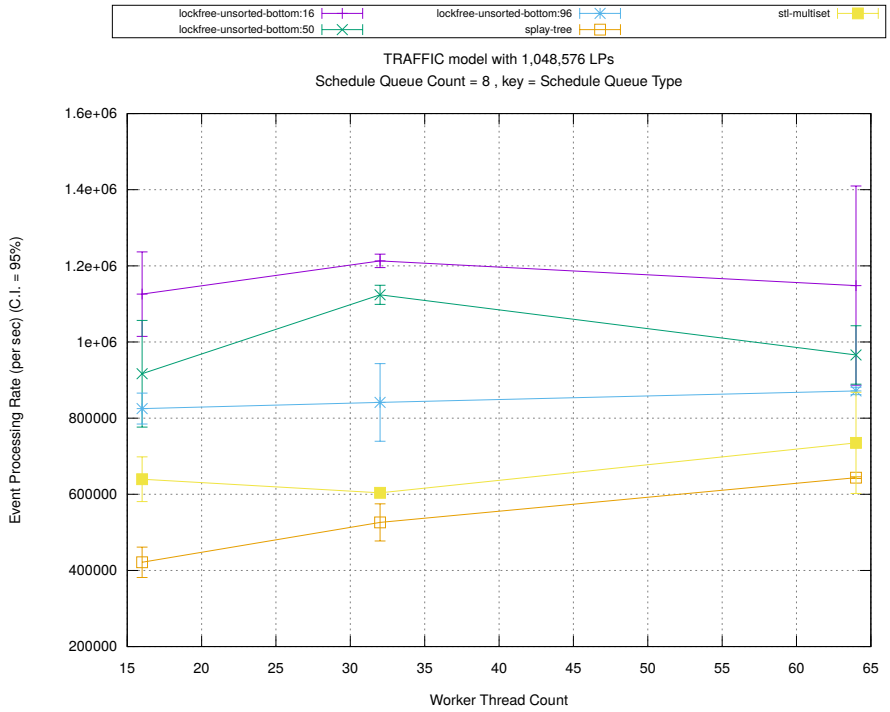
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

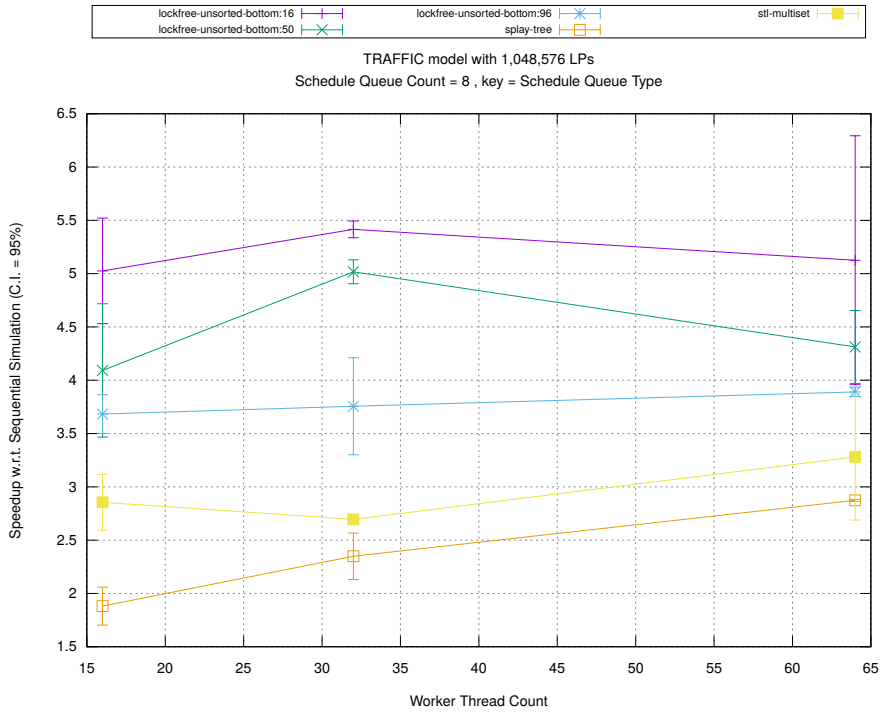


(b) Event Commitment Ratio

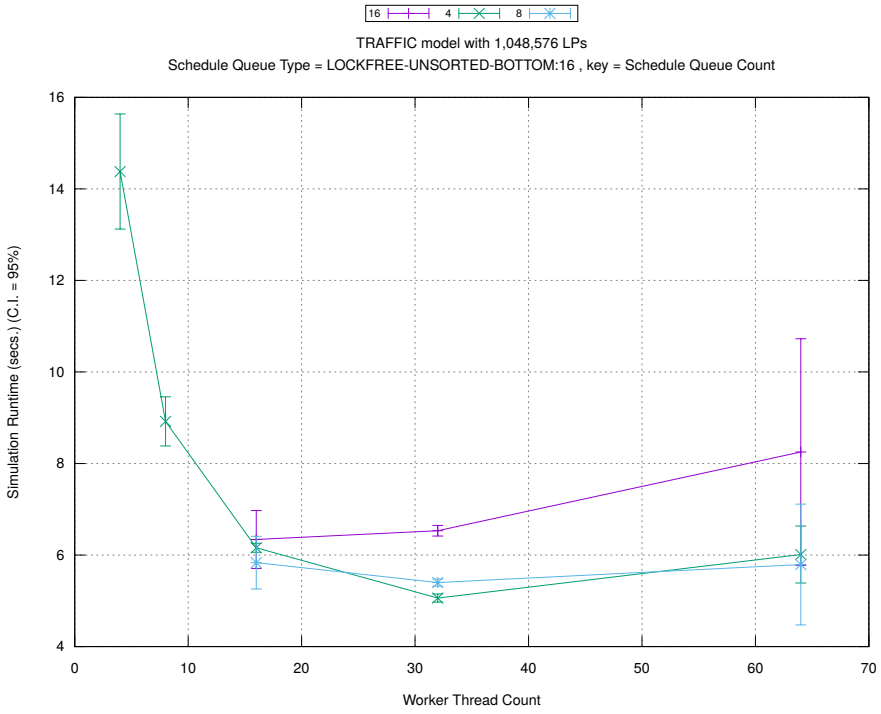


(c) Event Processing Rate (per second)

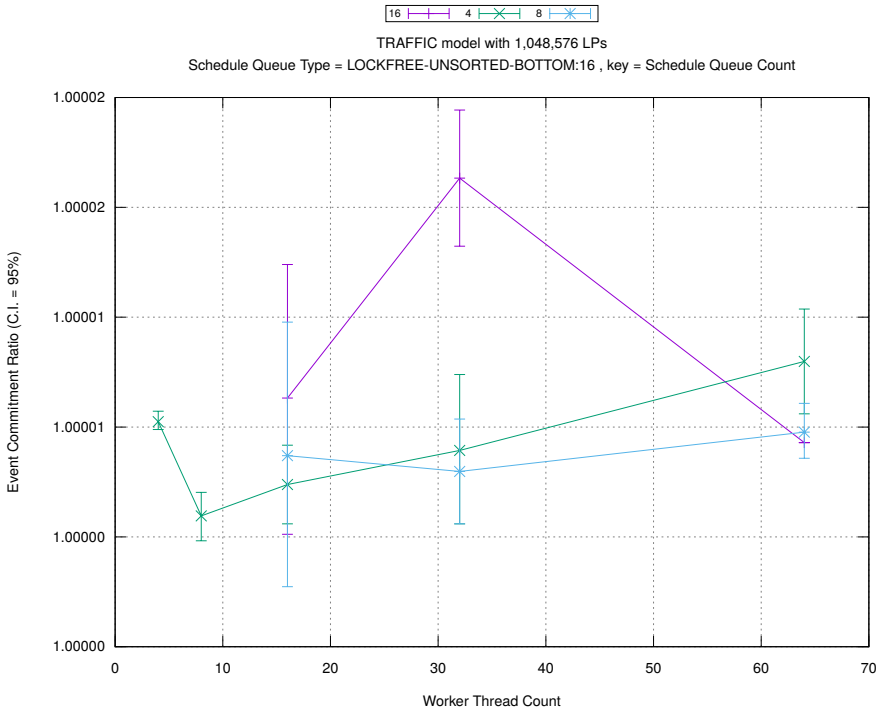
Figure A.52: traffic 1m/plots/scheduleq/threads vs type key count 8



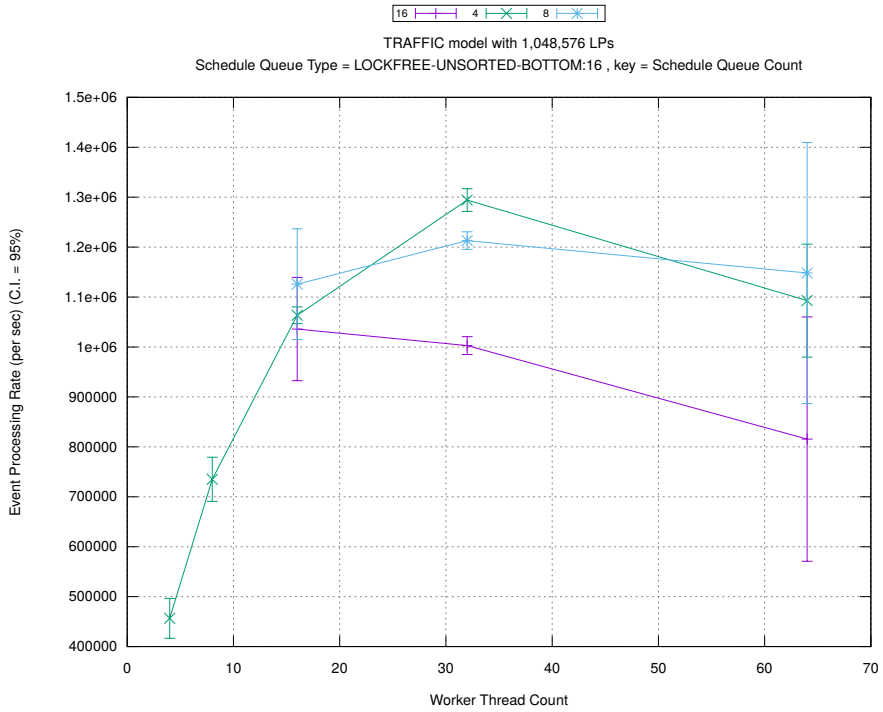
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

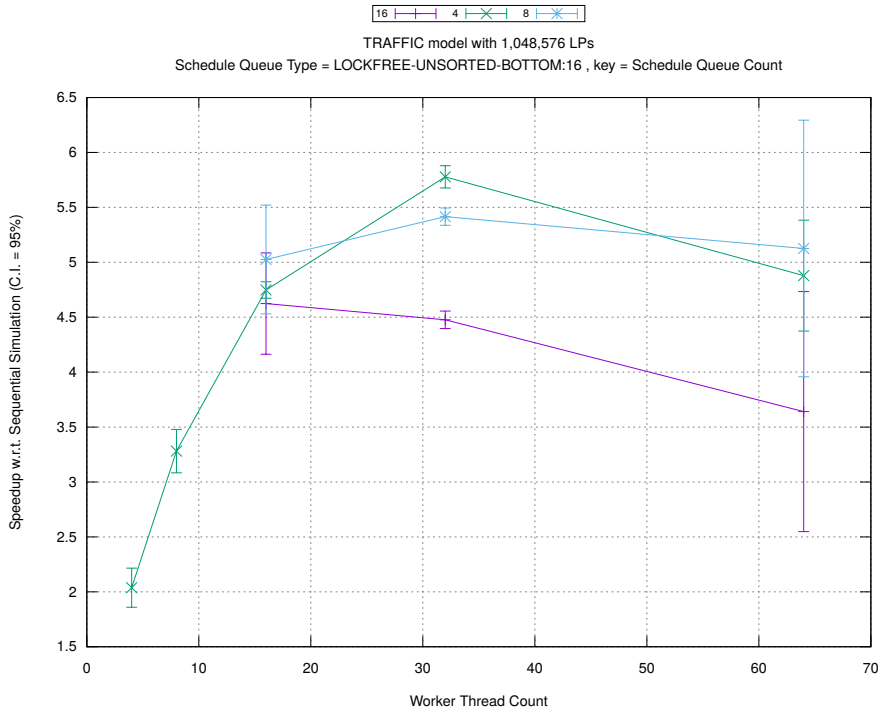


(b) Event Commitment Ratio

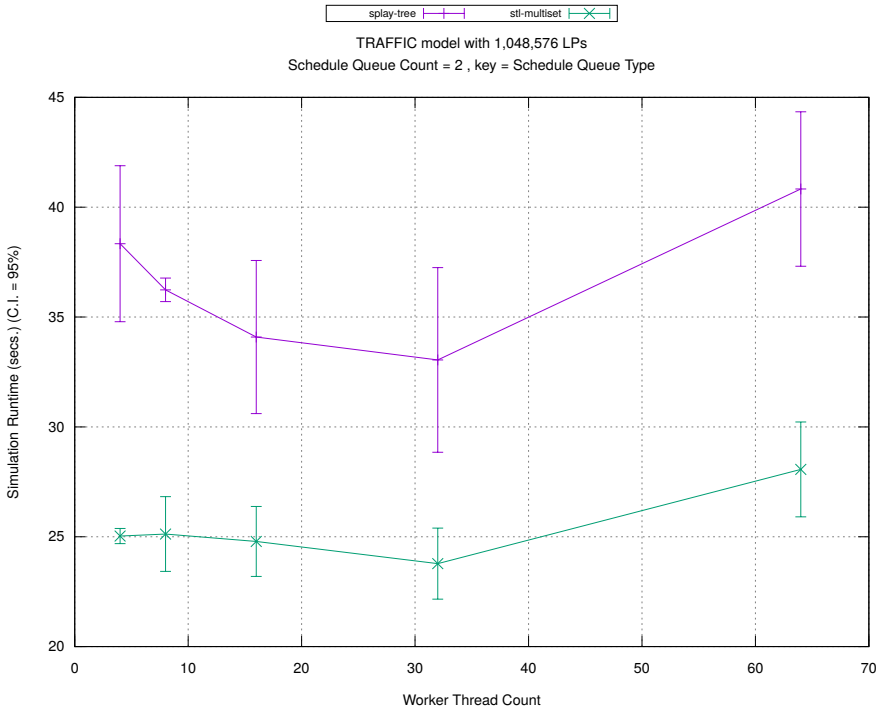


(c) Event Processing Rate (per second)

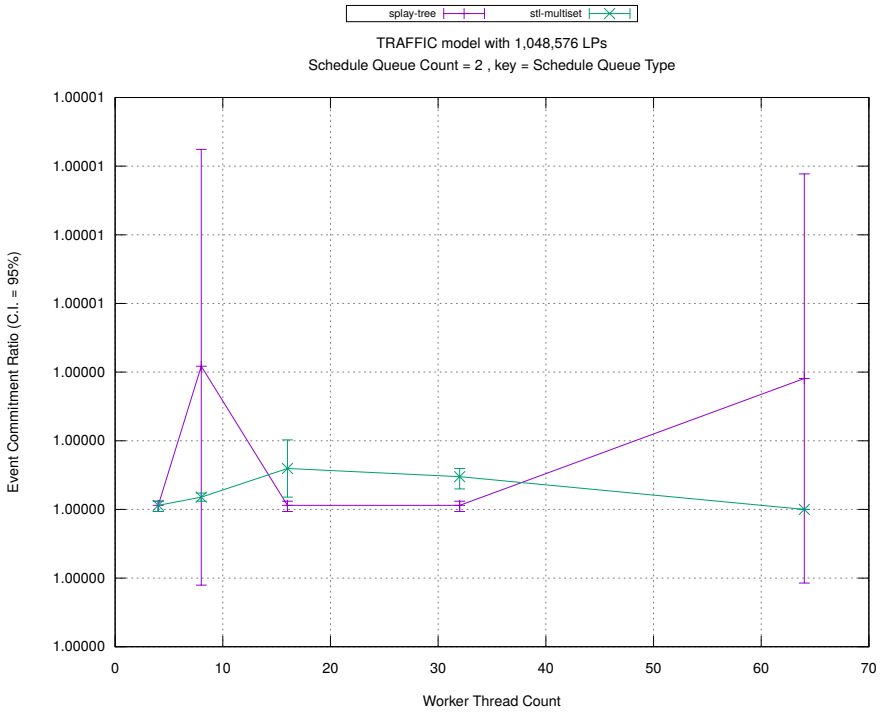
Figure A.53: traffic 1m/plots/scheduleq/threads vs count key type lockfree-unsorted-bottom 16



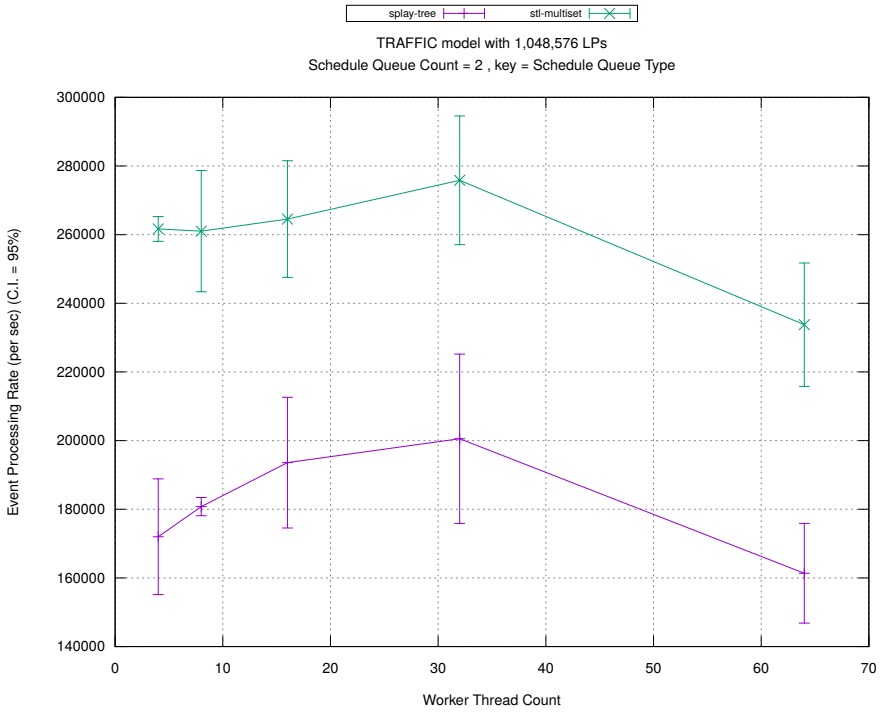
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

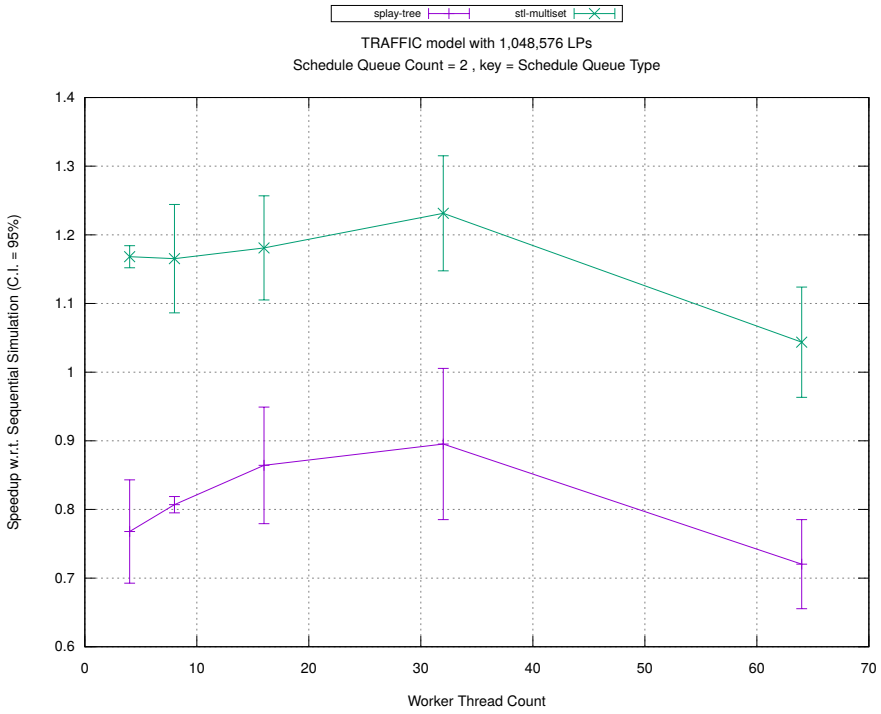


(b) Event Commitment Ratio

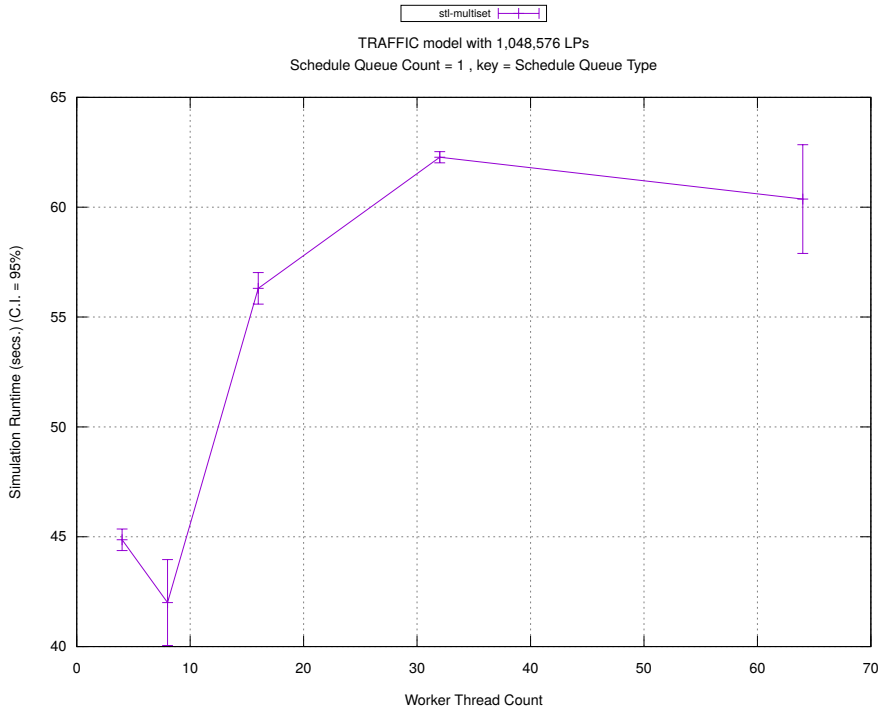


(c) Event Processing Rate (per second)

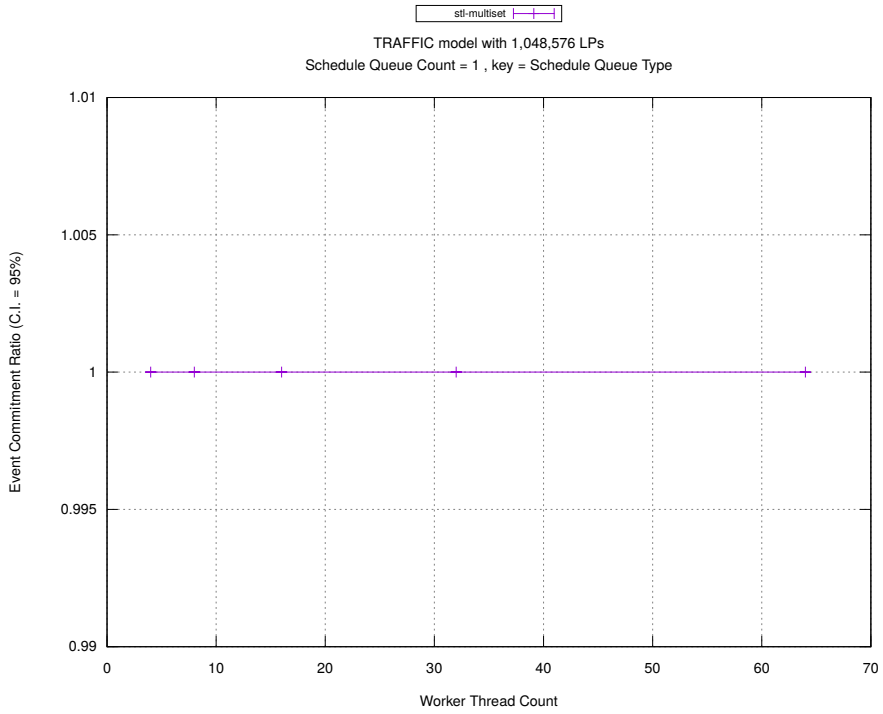
Figure A.54: traffic 1m/plots/scheduleq/threads vs type key count 2



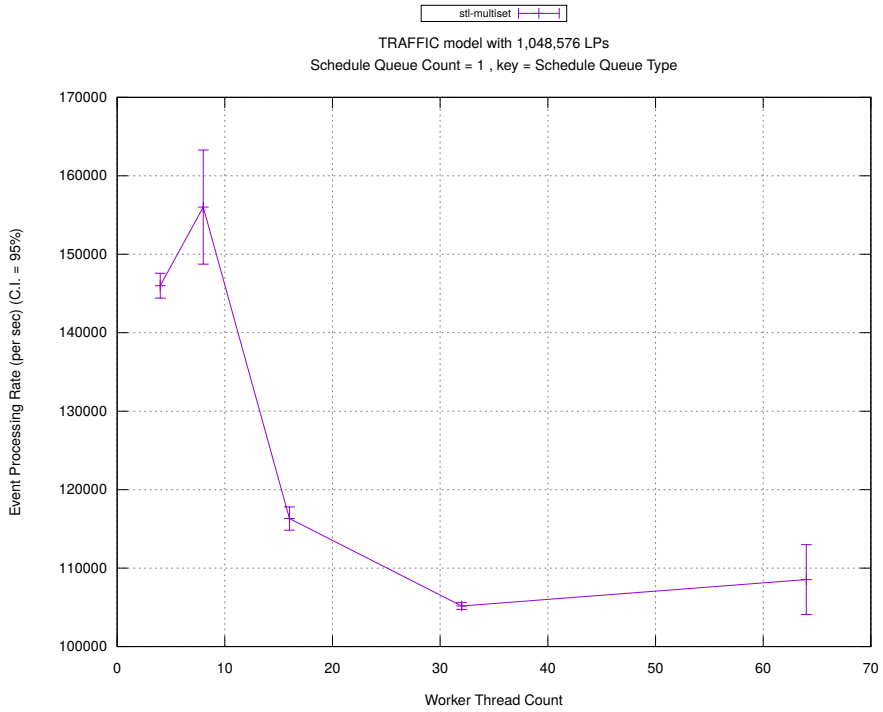
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

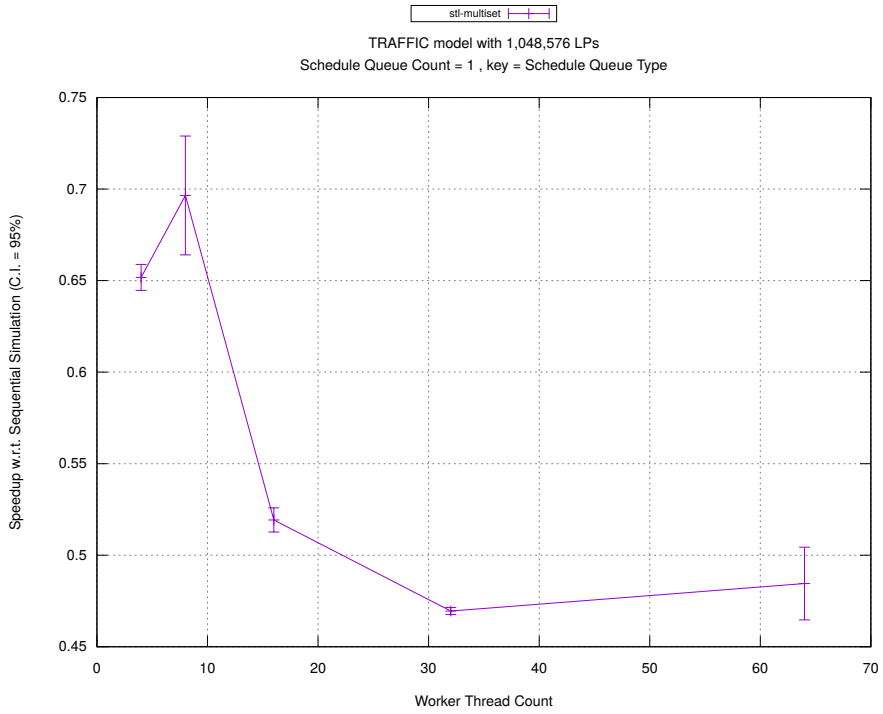


(b) Event Commitment Ratio

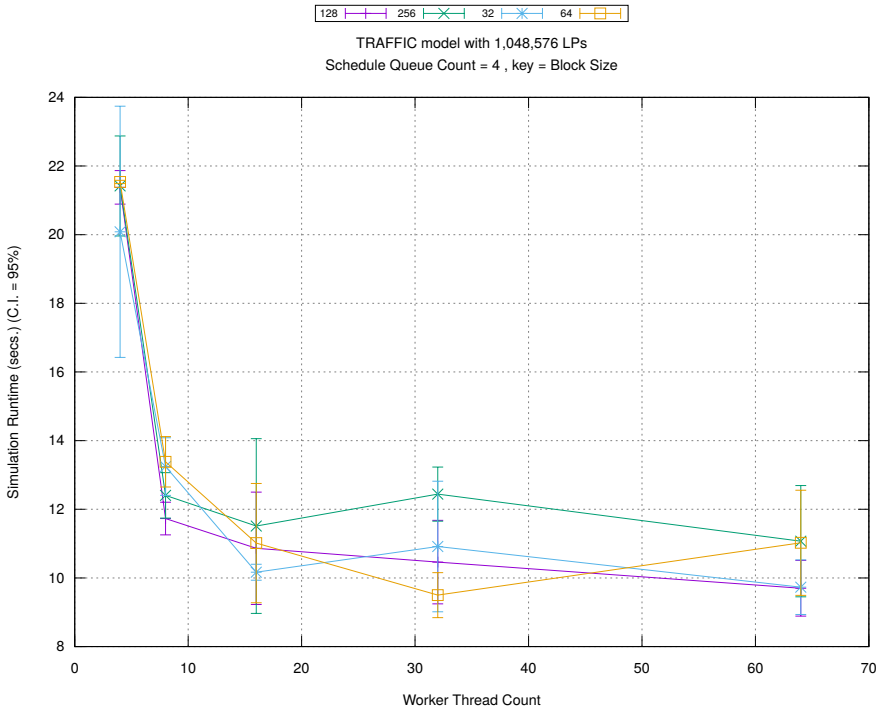


(c) Event Processing Rate (per second)

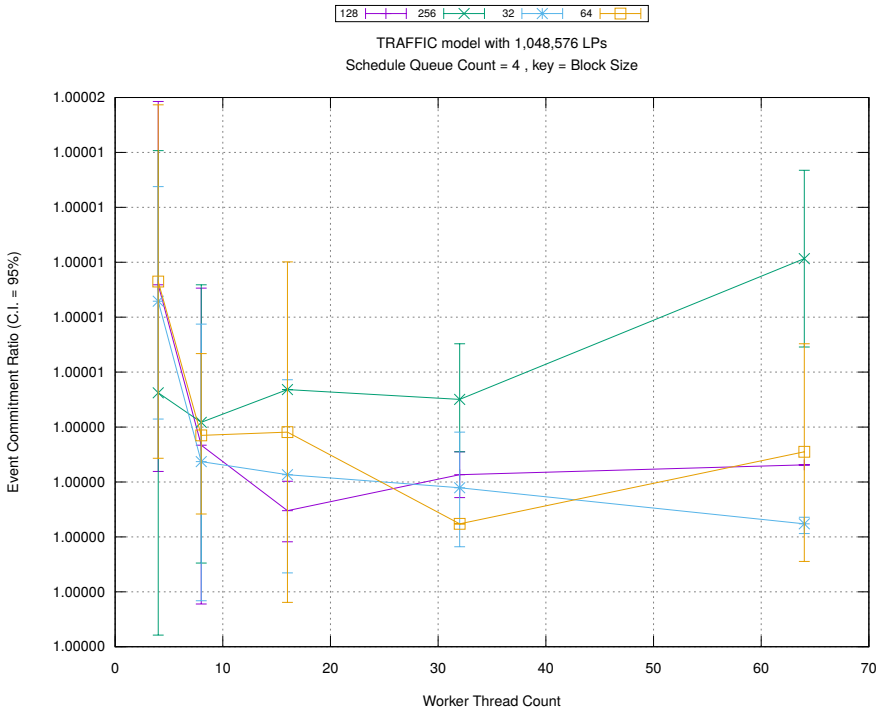
Figure A.55: traffic 1m/plots/scheduleq/threads vs type key count 1



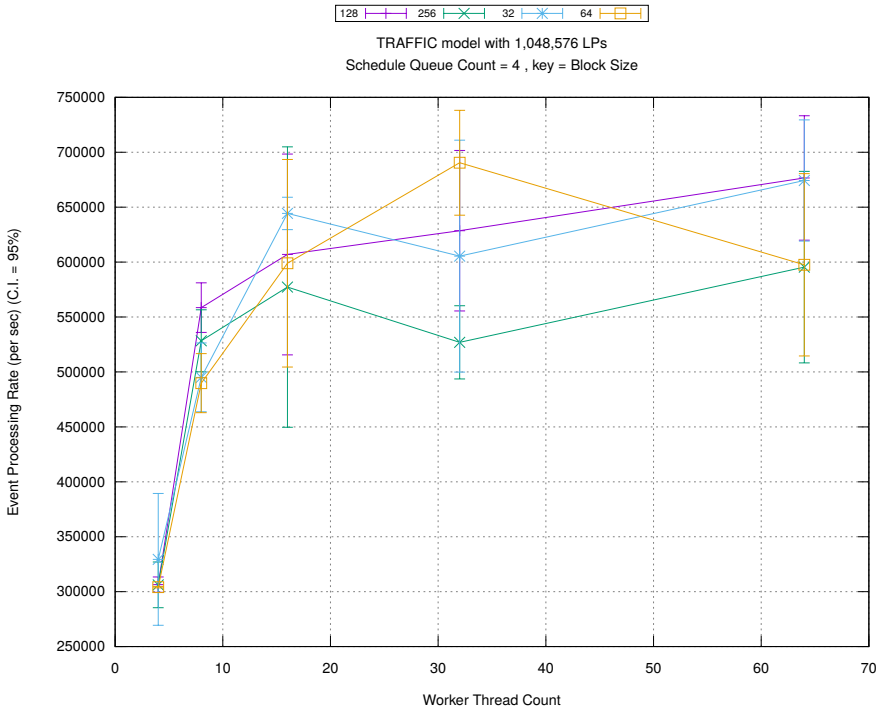
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

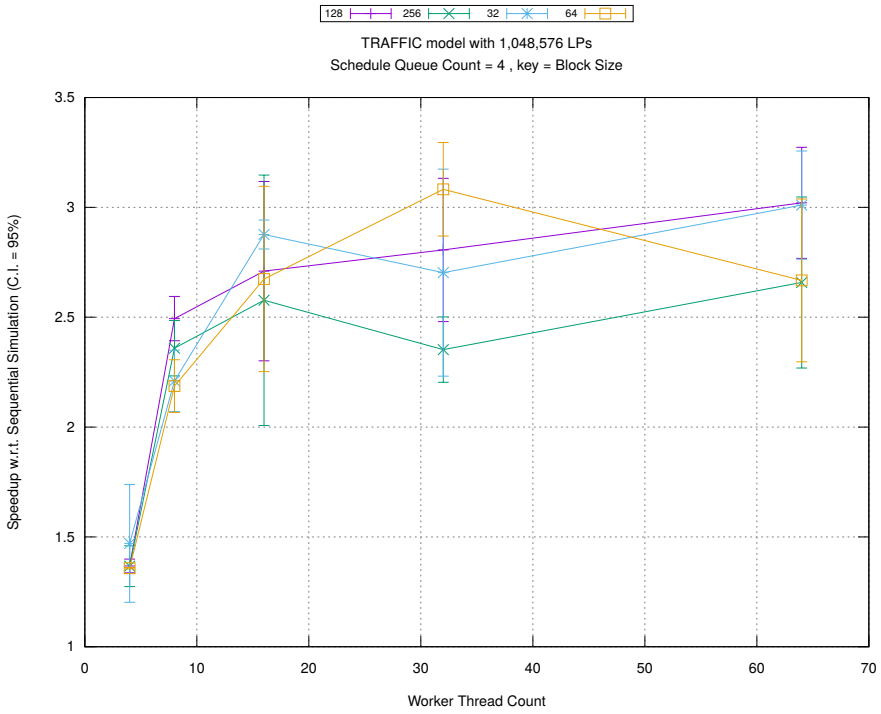


(b) Event Commitment Ratio

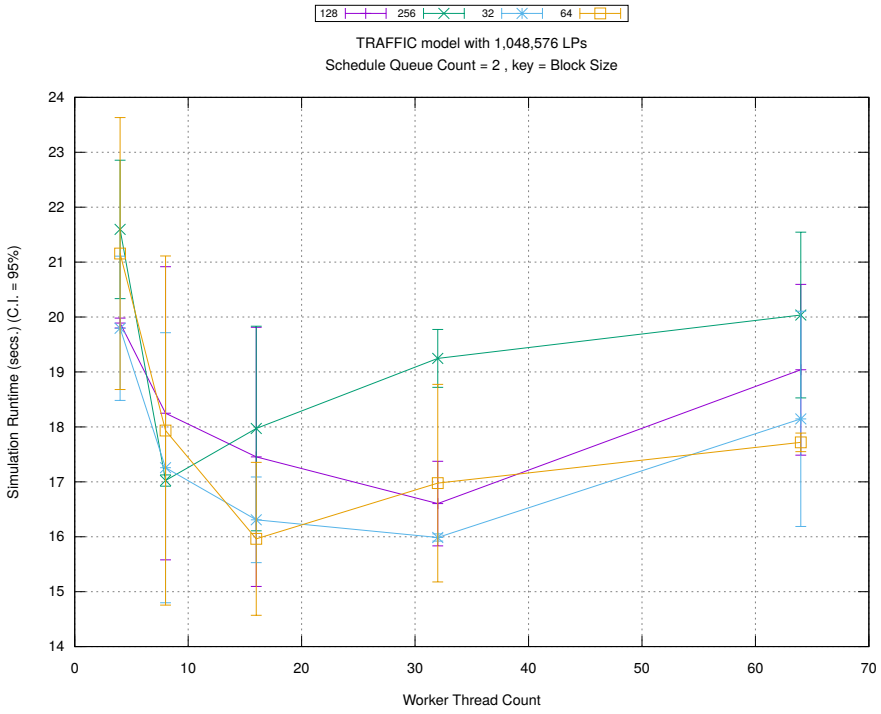


(c) Event Processing Rate (per second)

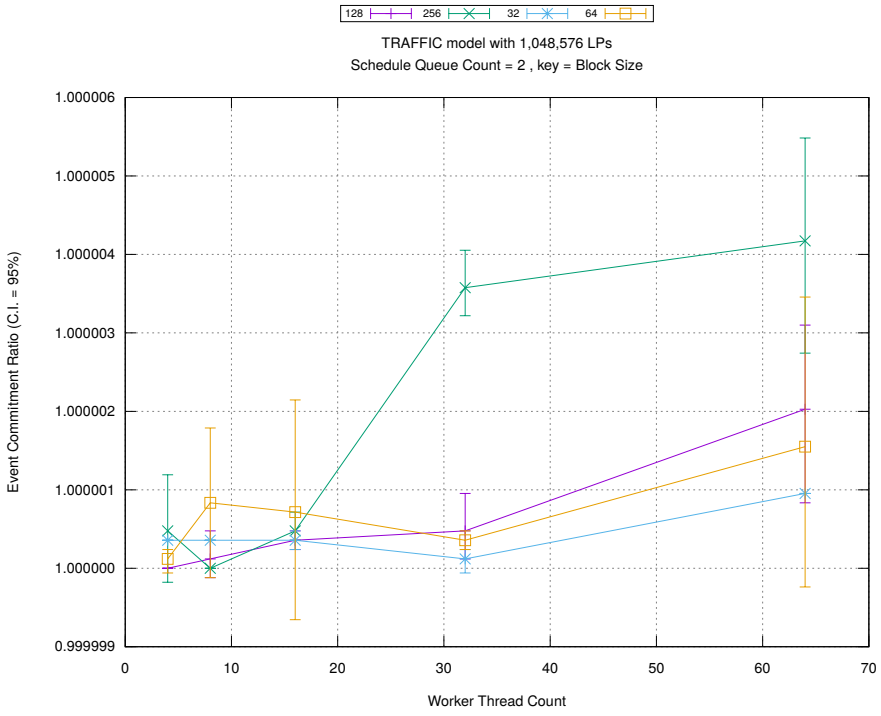
Figure A.56: traffic 1m/plots/blocks/threads vs blocksize key count 4



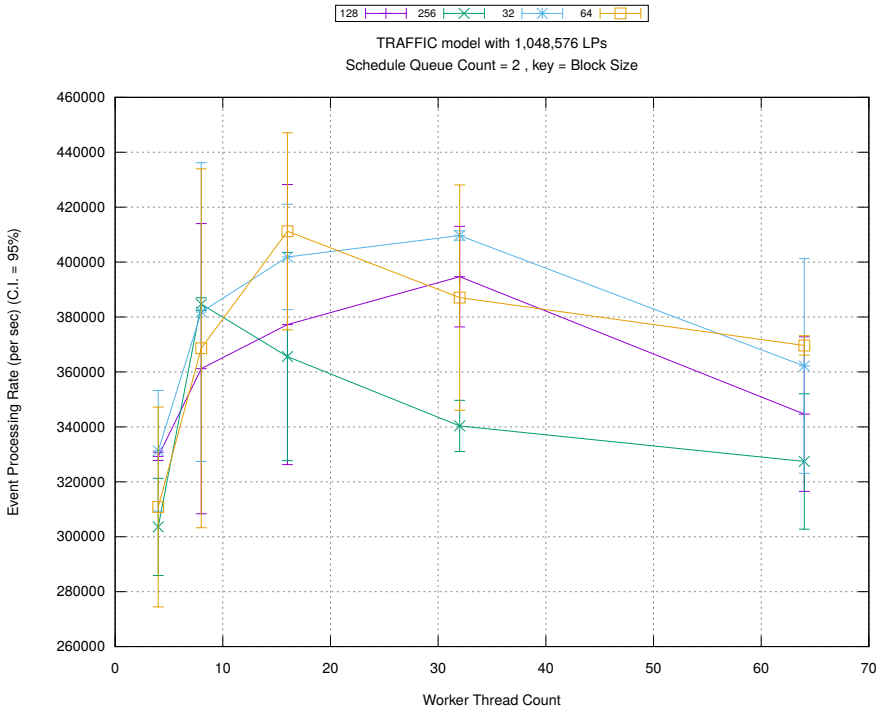
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

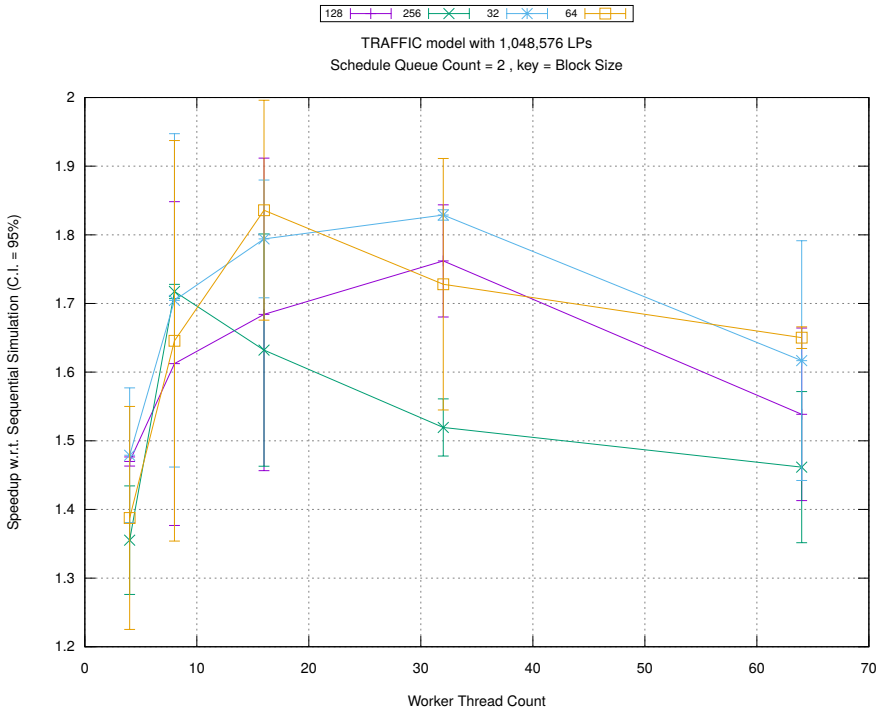


(b) Event Commitment Ratio

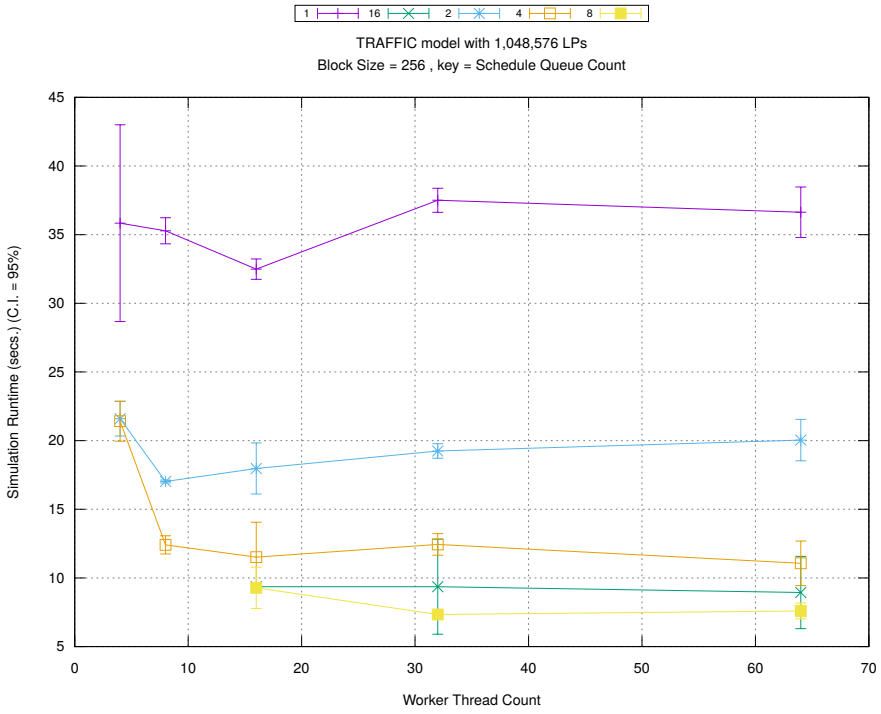


(c) Event Processing Rate (per second)

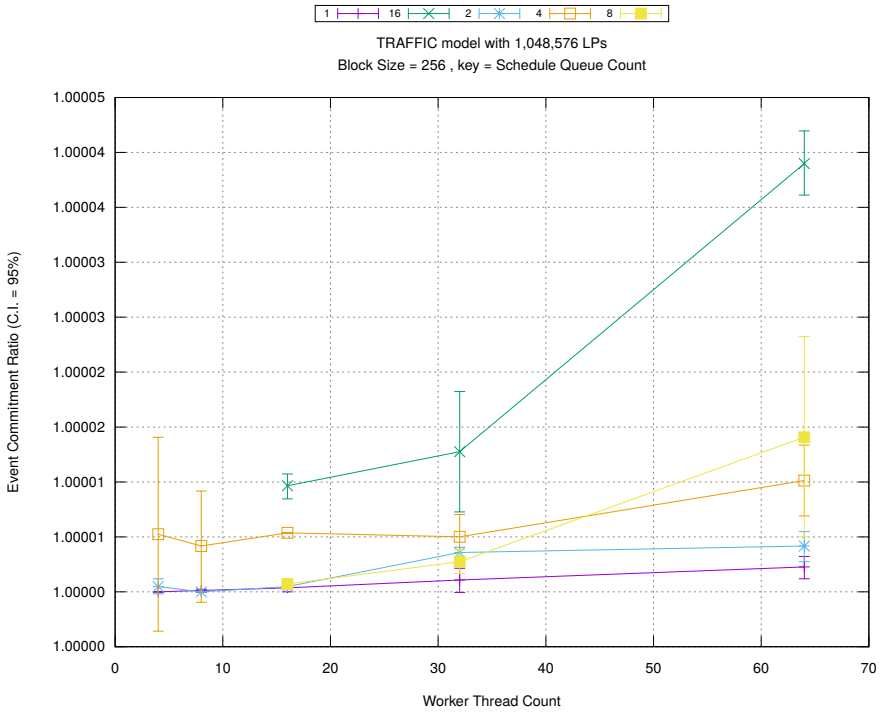
Figure A.57: traffic 1m/plots/blocks/threads vs blocksize key count 2



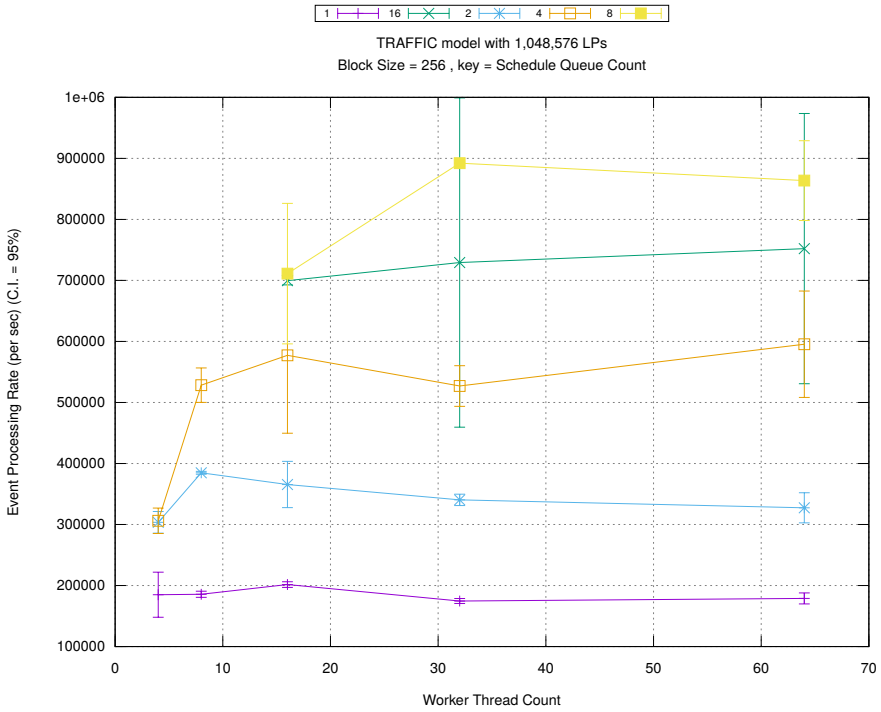
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

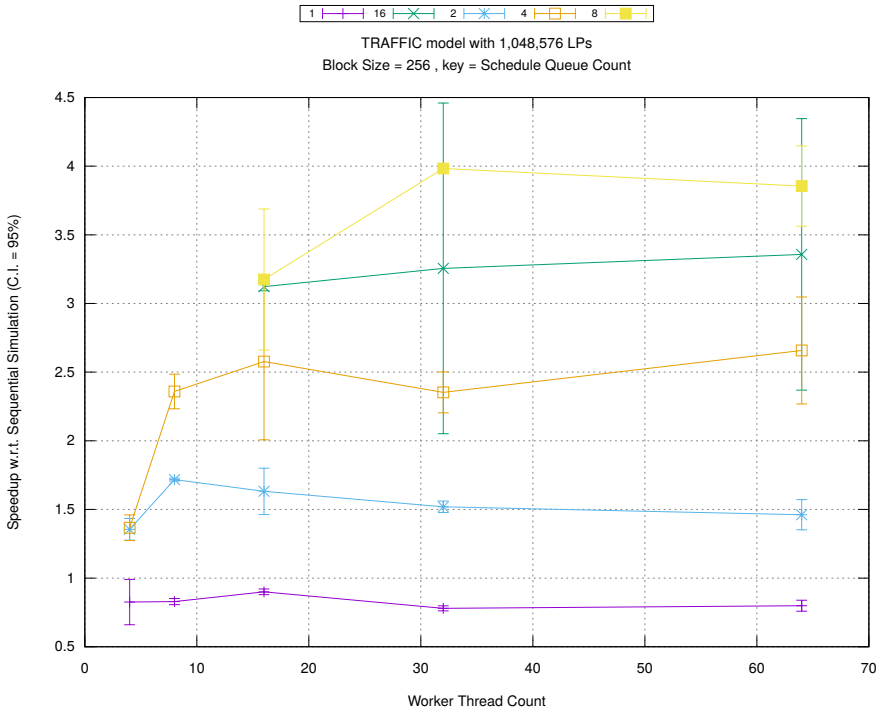


(b) Event Commitment Ratio

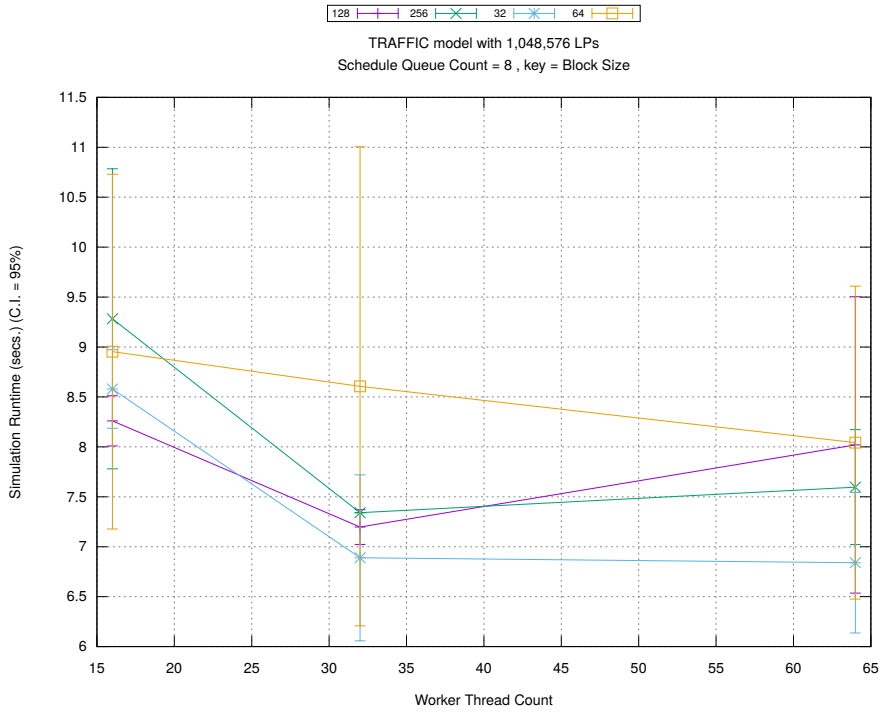


(c) Event Processing Rate (per second)

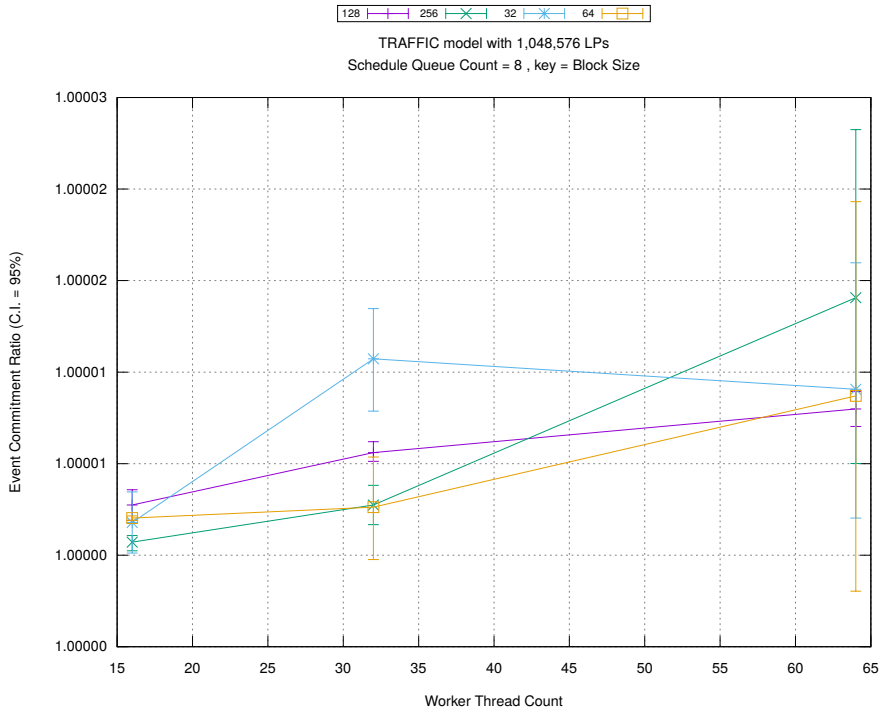
Figure A.58: traffic 1m/plots/blocks/threads vs count key blocksize 256



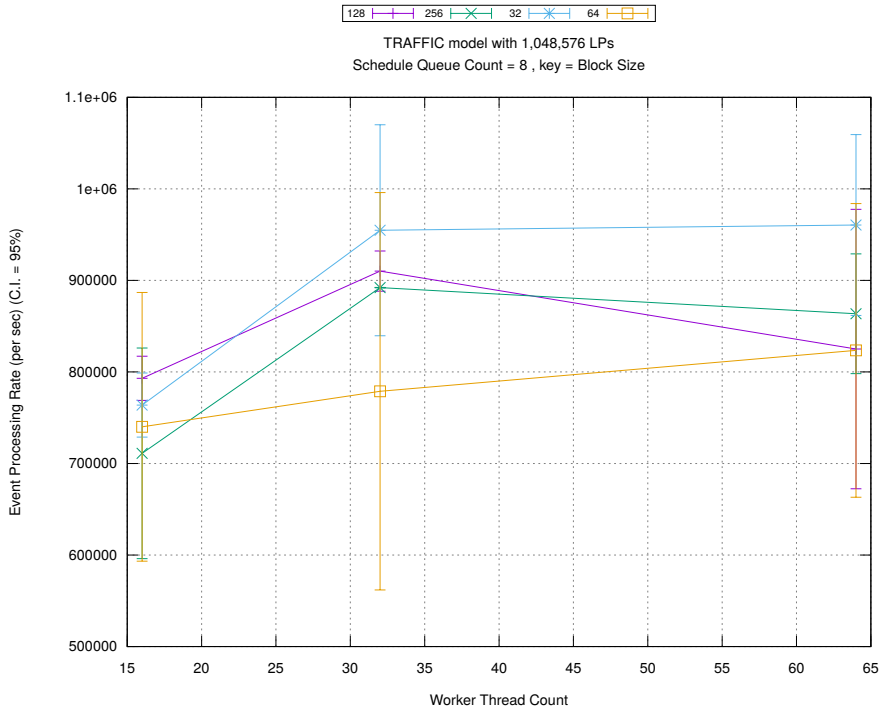
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

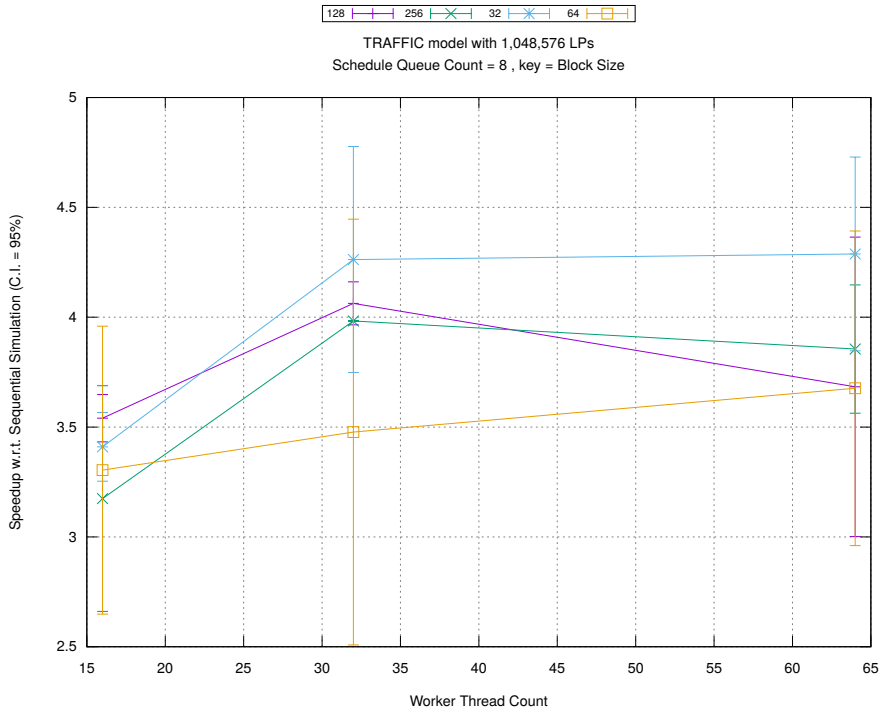


(b) Event Commitment Ratio

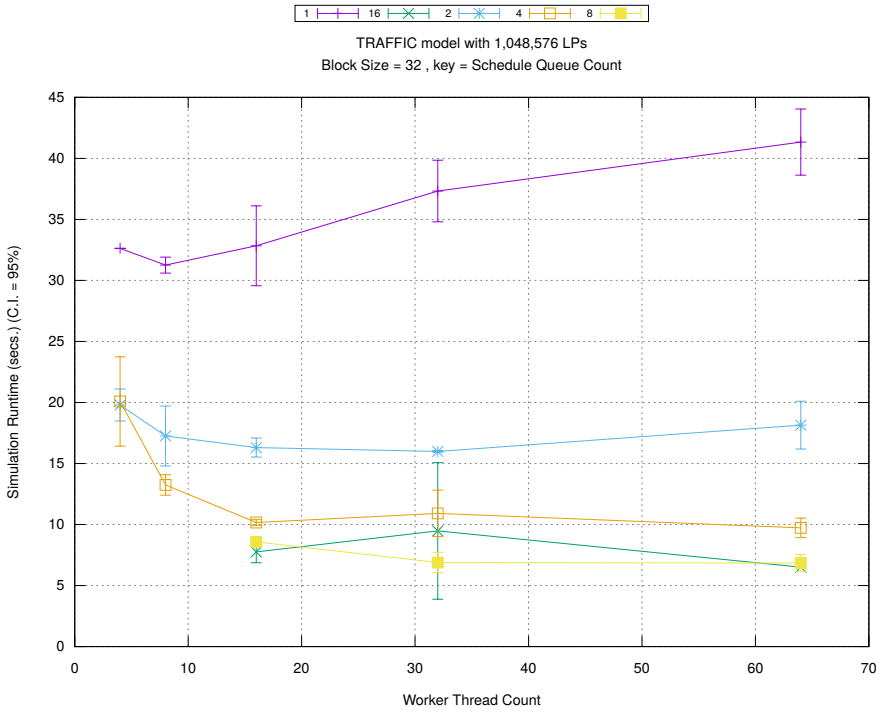


(c) Event Processing Rate (per second)

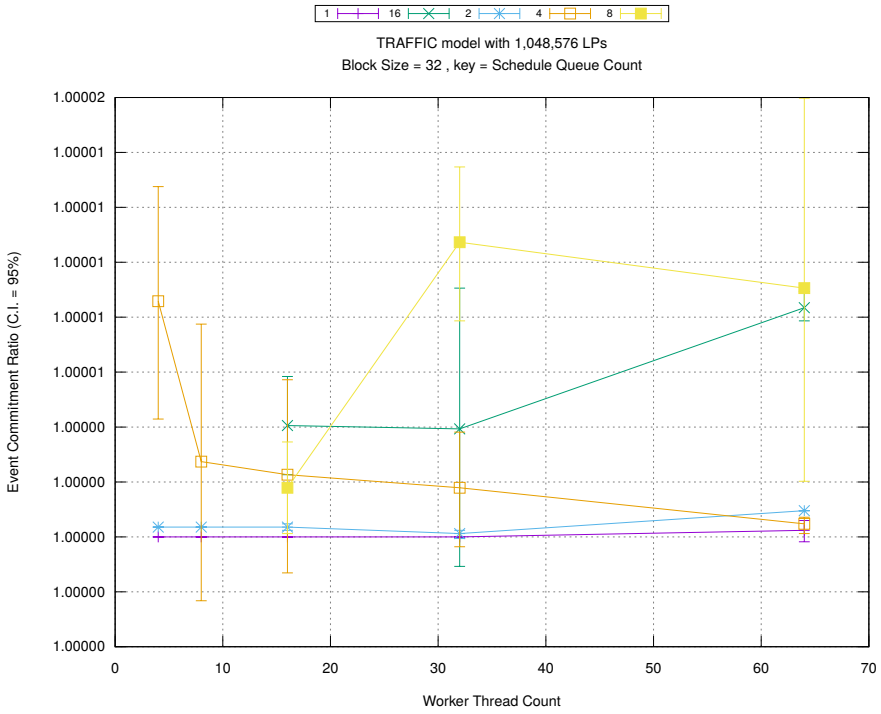
Figure A.59: traffic 1m/plots/blocks/threads vs blocksize key count 8



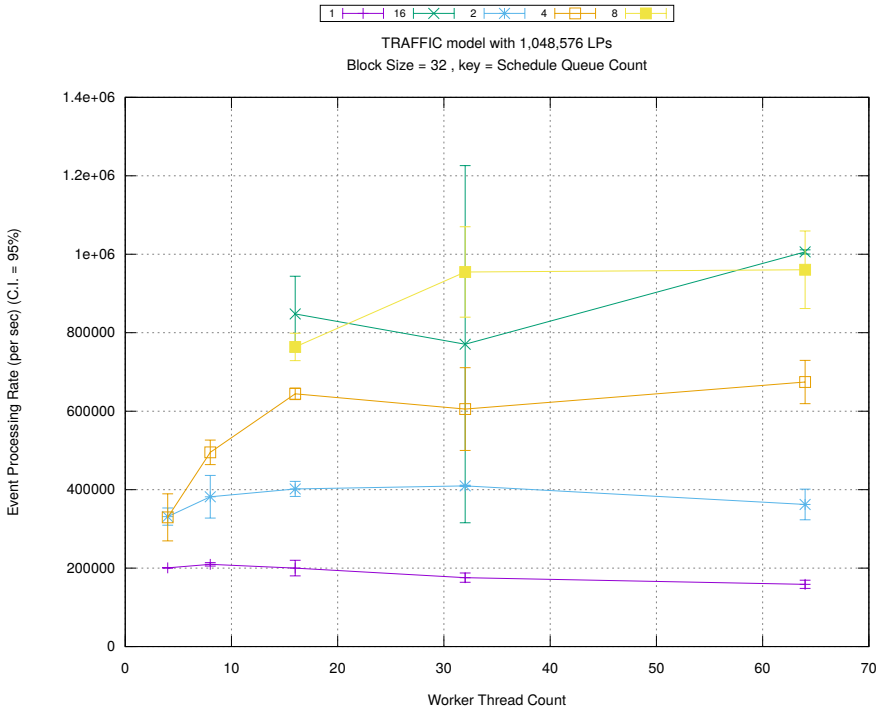
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

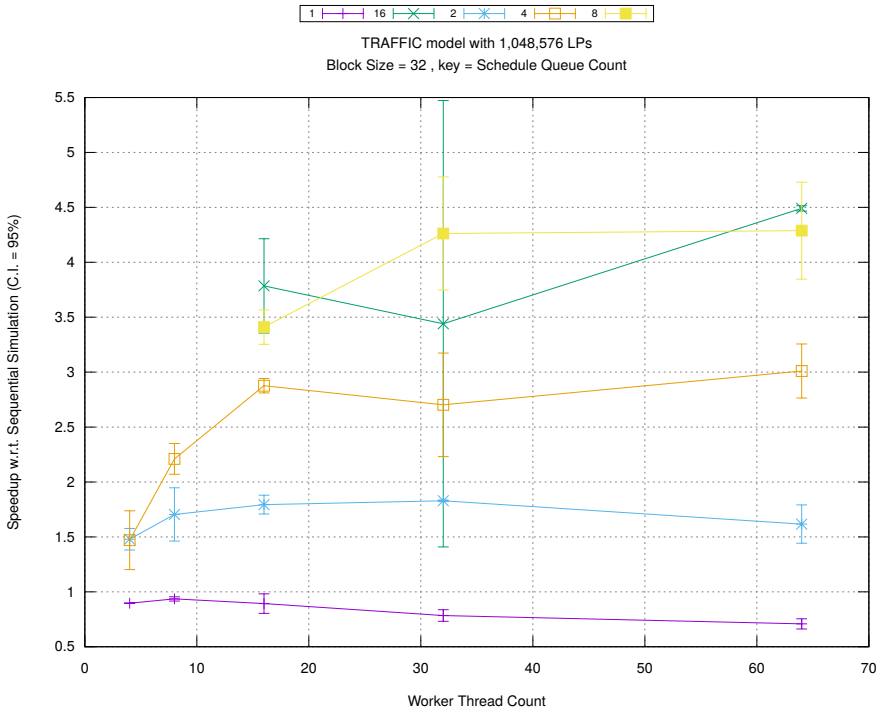


(b) Event Commitment Ratio

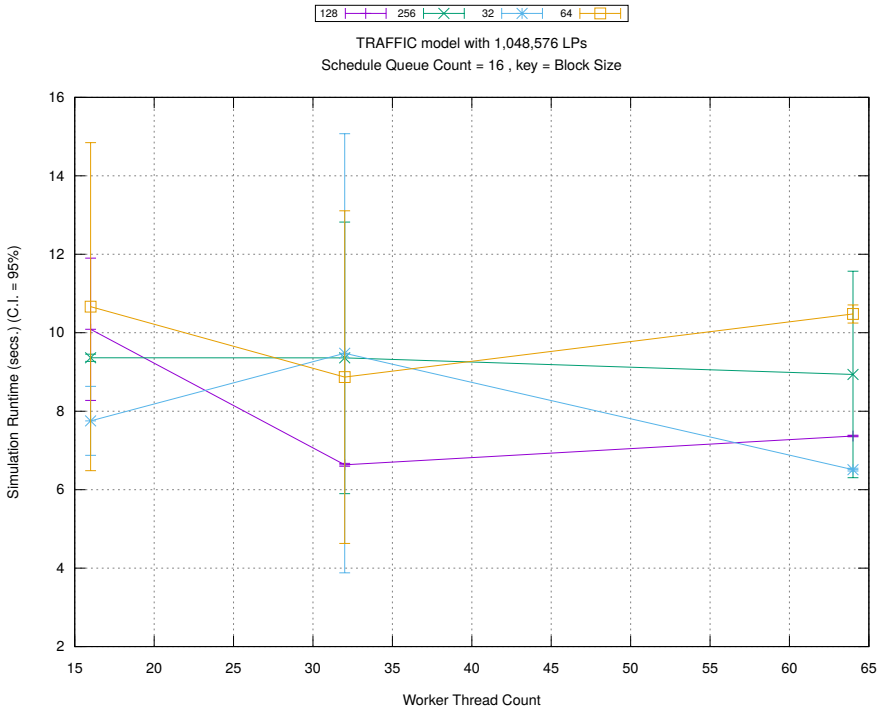


(c) Event Processing Rate (per second)

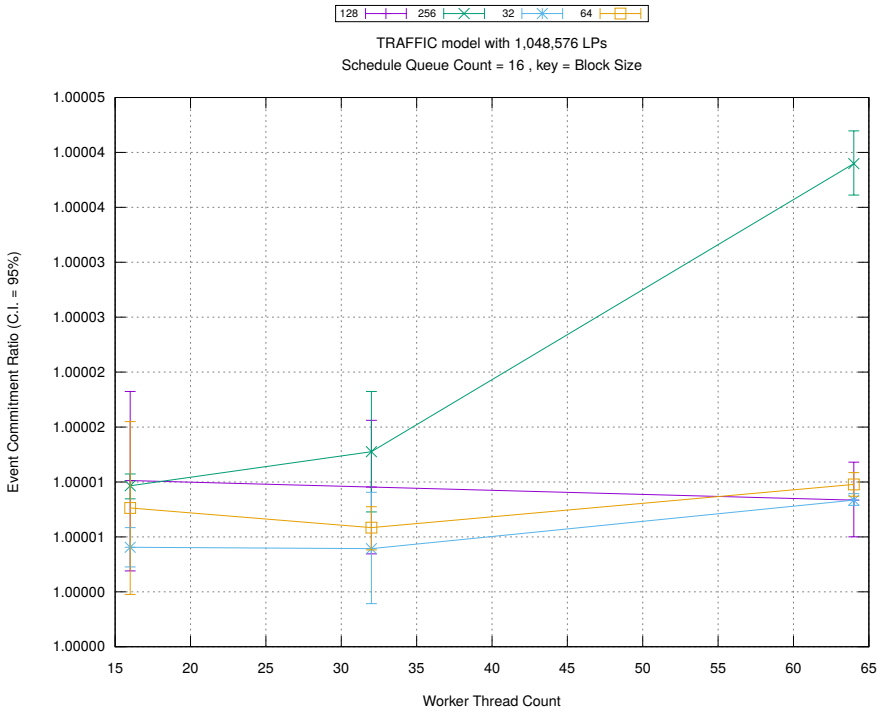
Figure A.60: traffic 1m/plots/blocks/threads vs count key blocksize 32



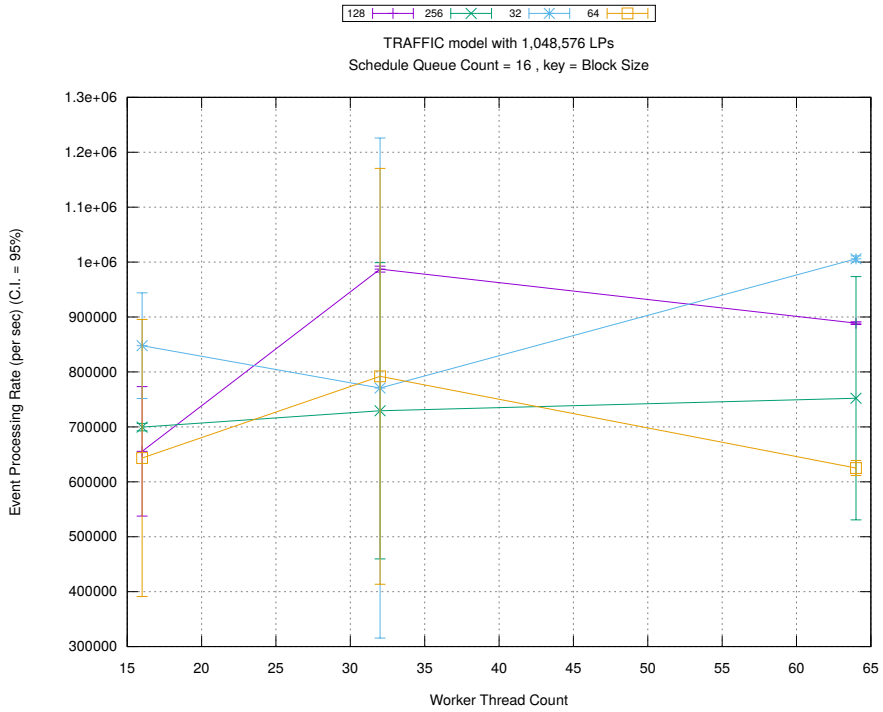
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

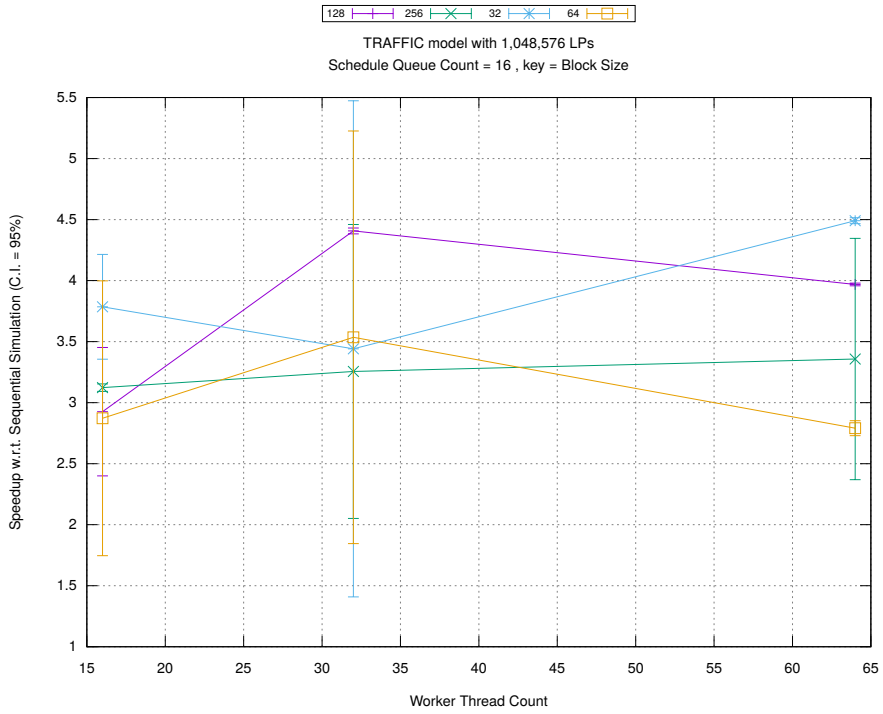


(b) Event Commitment Ratio

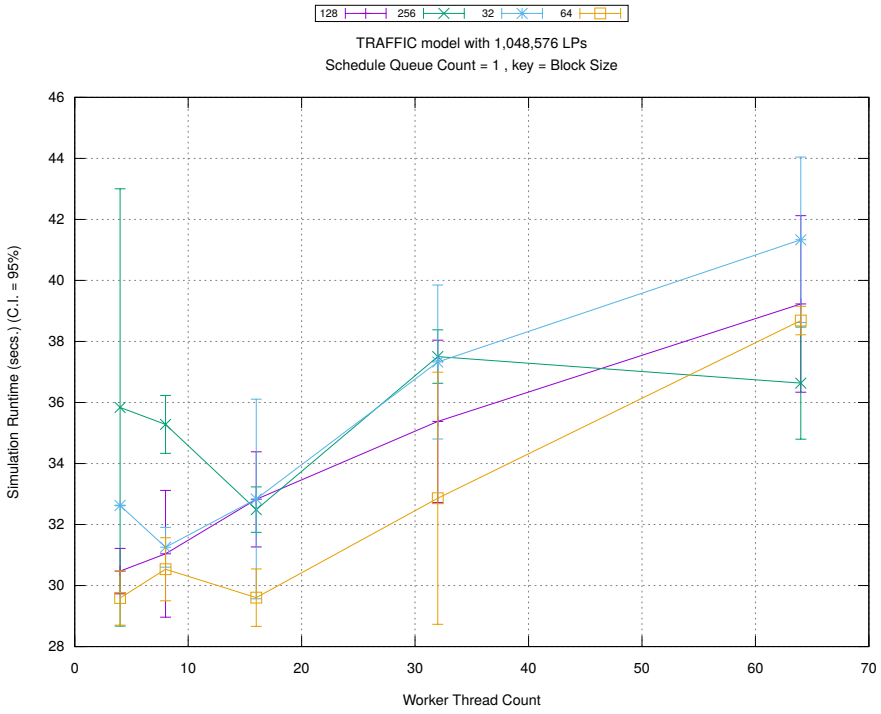


(c) Event Processing Rate (per second)

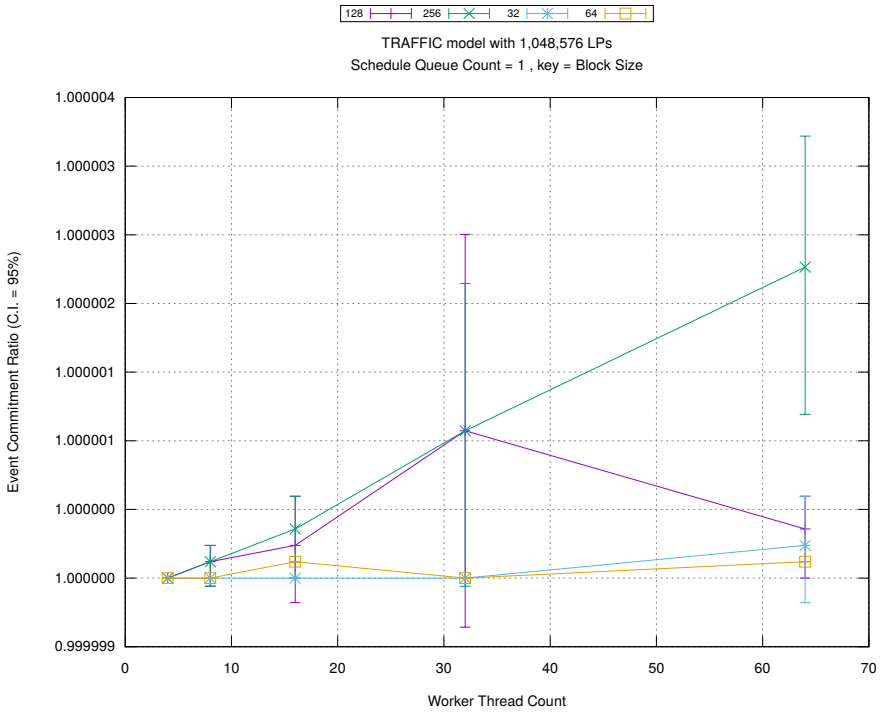
Figure A.61: traffic 1m/plots/blocks/threads vs blocksize key count 16



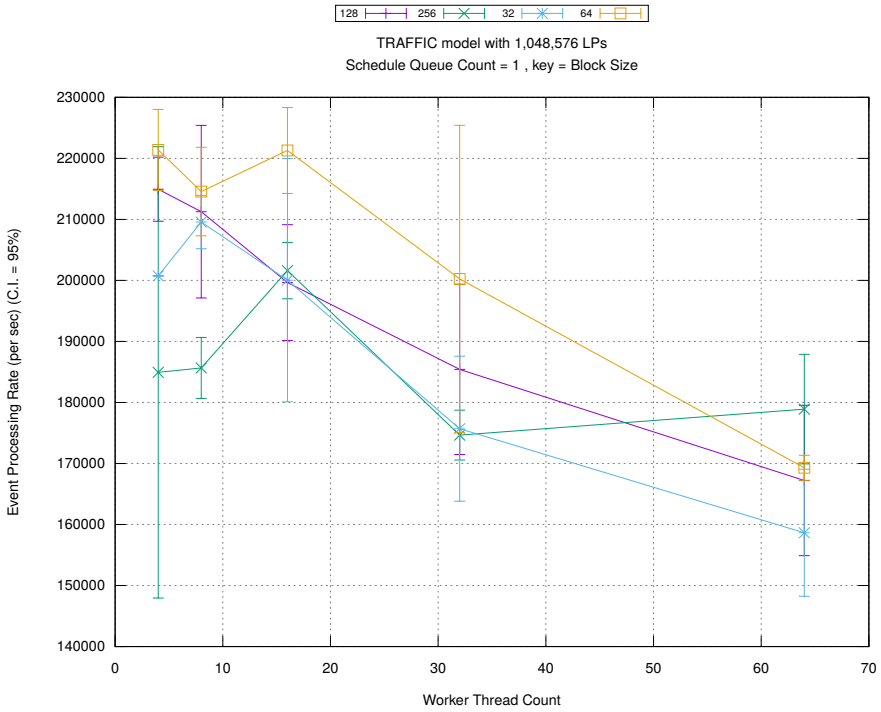
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

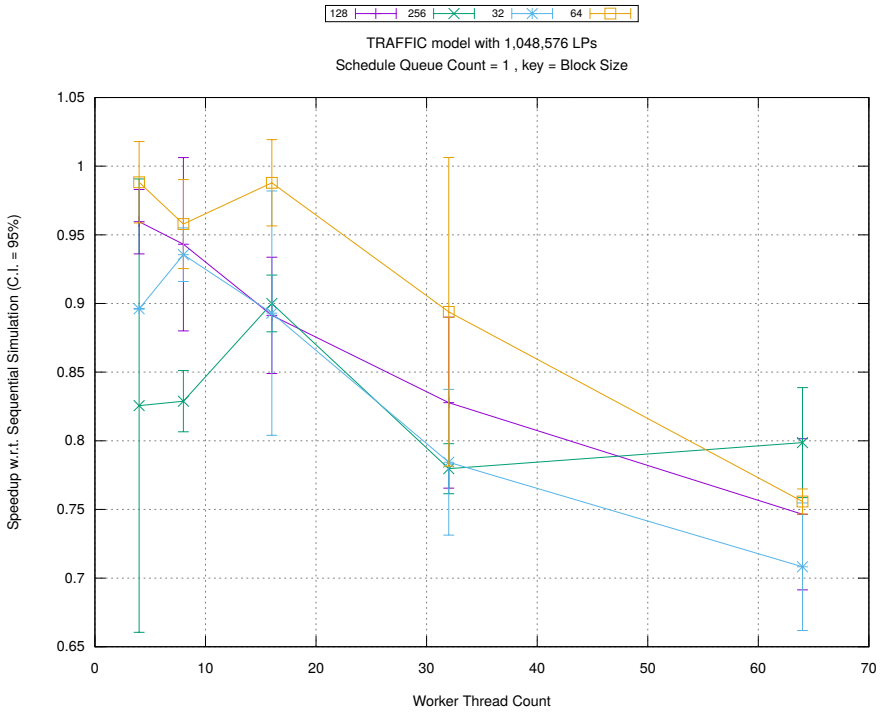


(b) Event Commitment Ratio

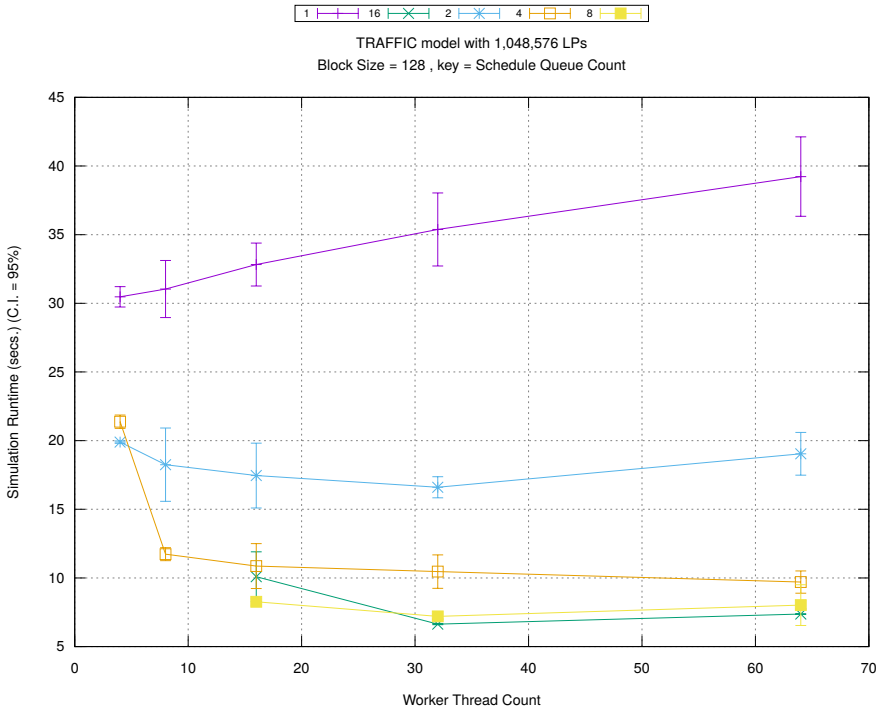


(c) Event Processing Rate (per second)

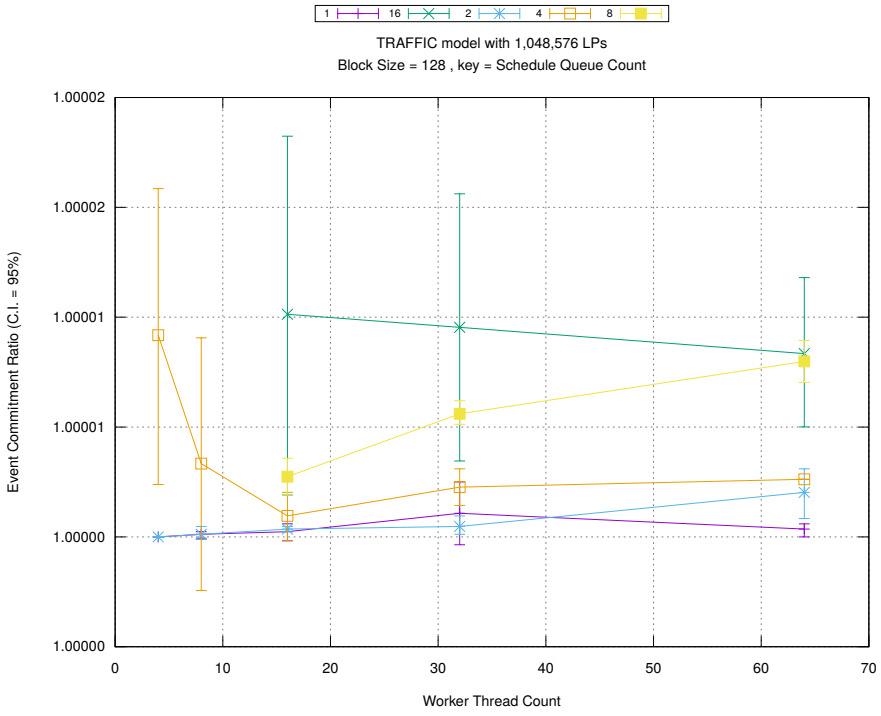
Figure A.62: traffic 1m/plots/blocks/threads vs blocksize key count 1



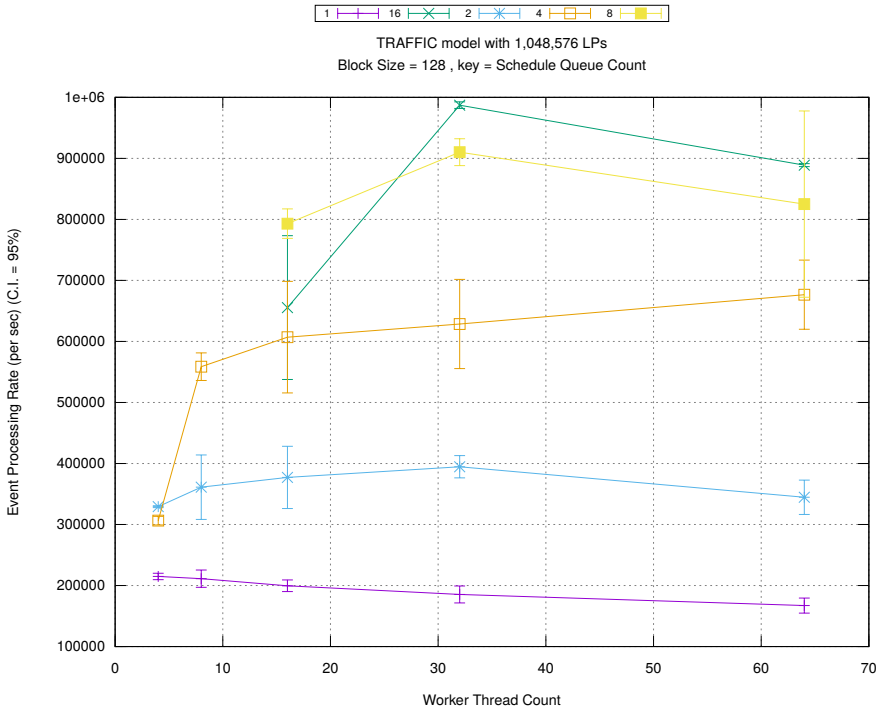
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

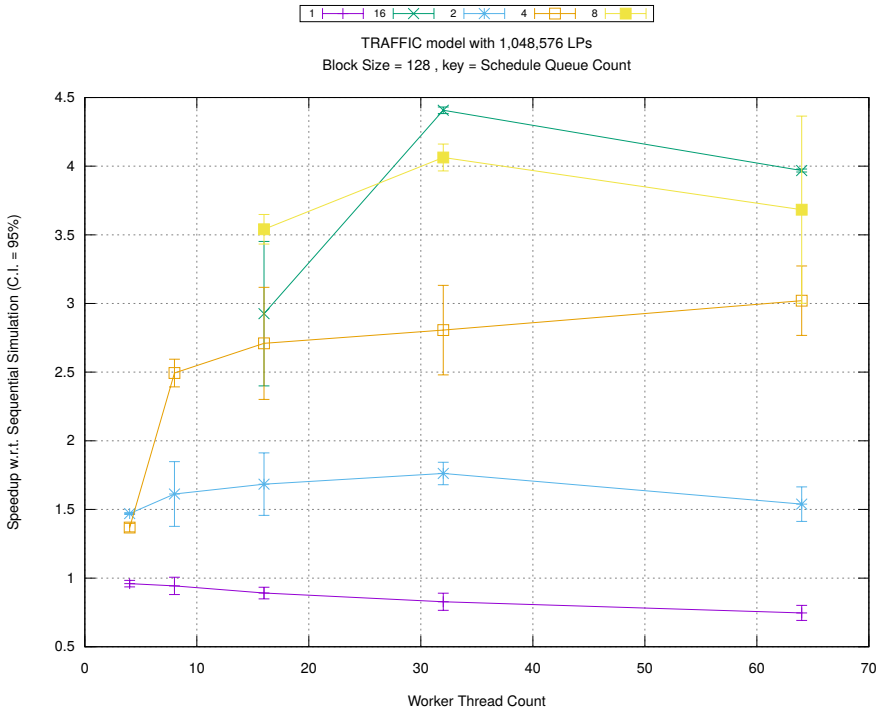


(b) Event Commitment Ratio

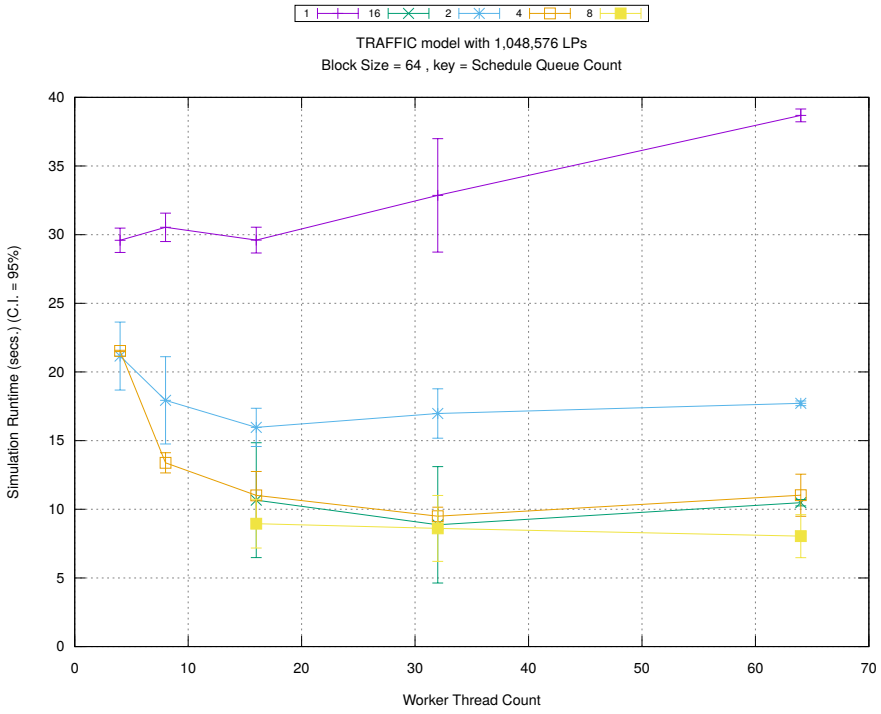


(c) Event Processing Rate (per second)

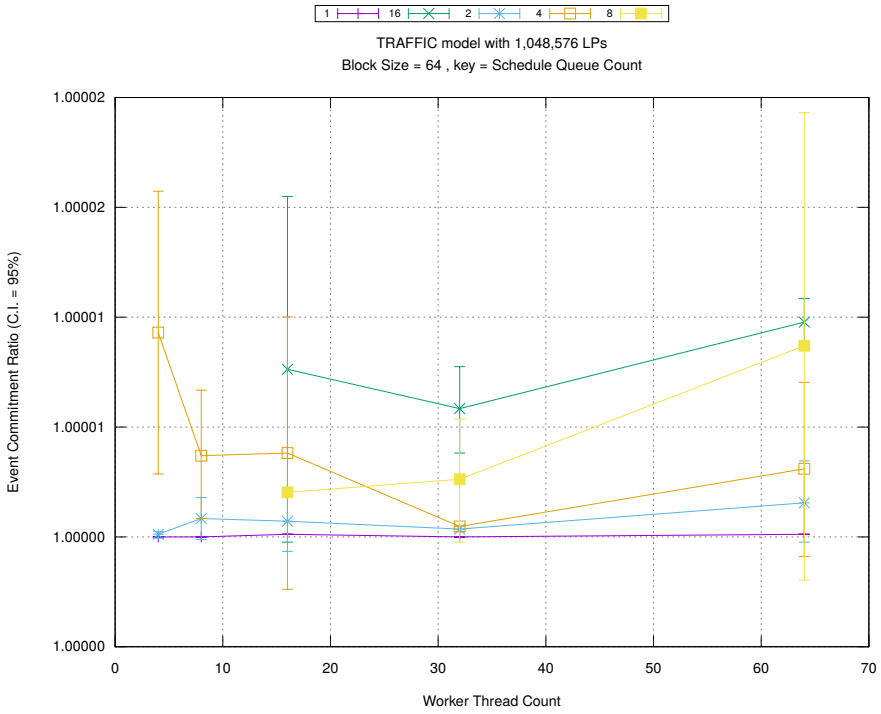
Figure A.63: traffic 1m/plots/blocks/threads vs count key blocksize 128



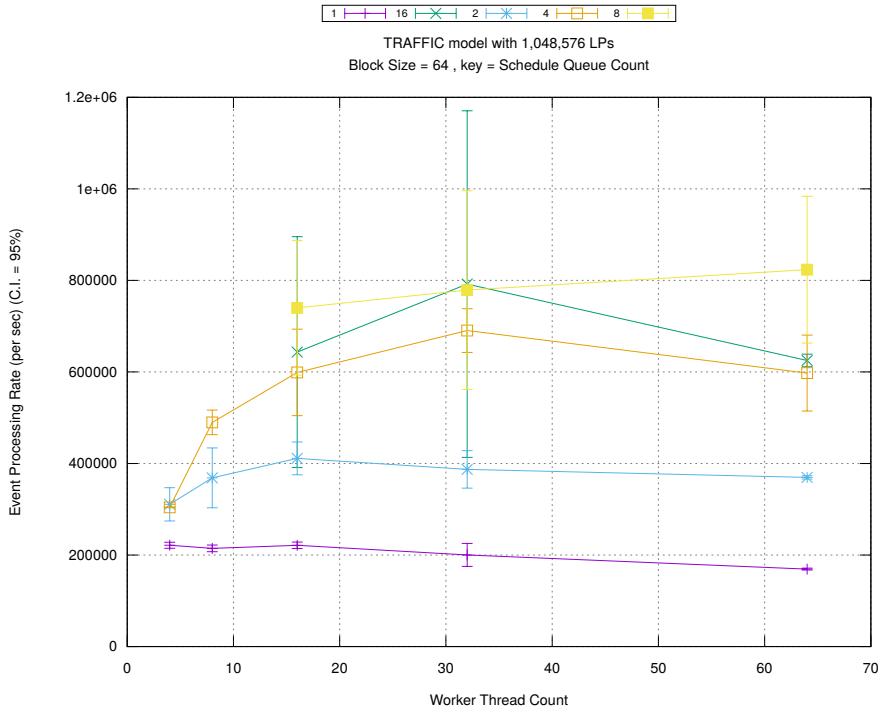
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

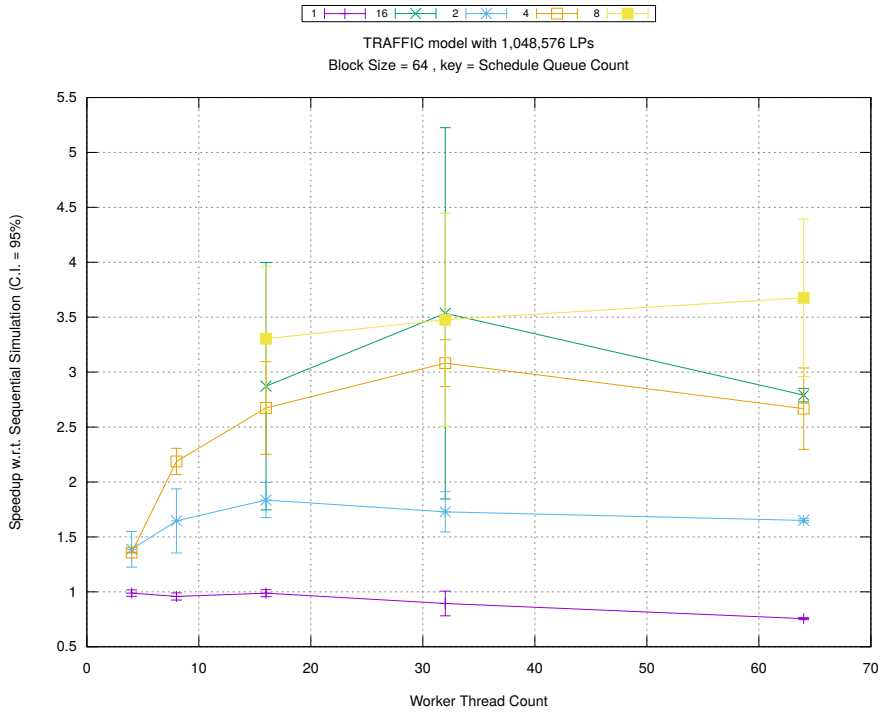


(b) Event Commitment Ratio



(c) Event Processing Rate (per second)

Figure A.64: traffic 1m/plots/blocks/threads vs count key blocksize 64



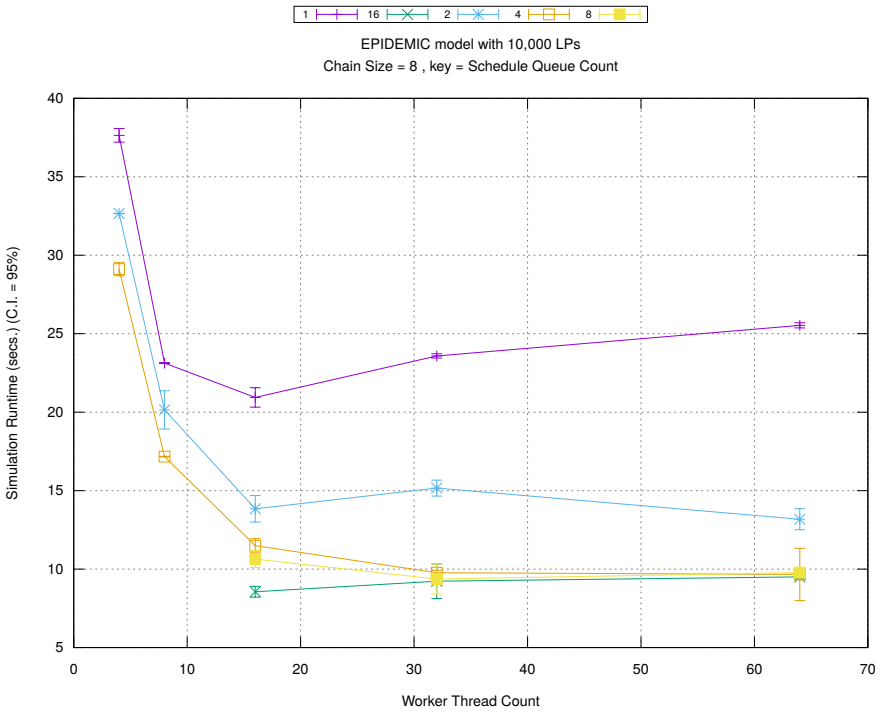
(d) Speedup w.r.t. Sequential Simulation

A.3 Epidemic Model with 10,000 LPs and Watts-Strogatz Network

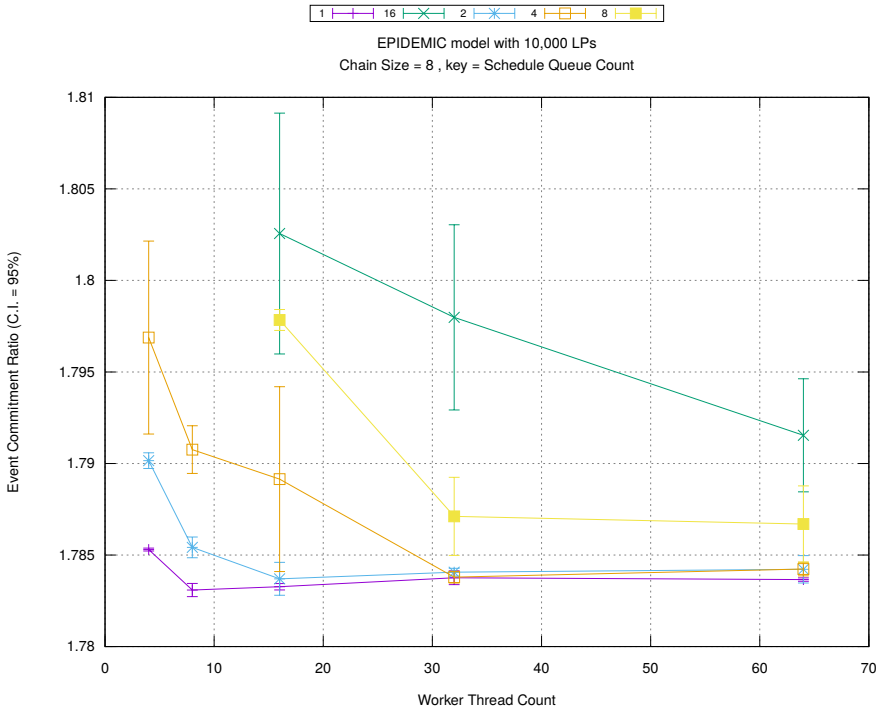
Table A.3 shows the configuration for this model.

Parameter	Values
Number of Intersections (or LPs)	10,000
Type of network connecting LPs	Watts-Strogatz [72]
Population Size	1,000,000
Simulation Time	15,000 timestamp units
Sequential Simulation Time for calculating modularity	8,000 timestamp units

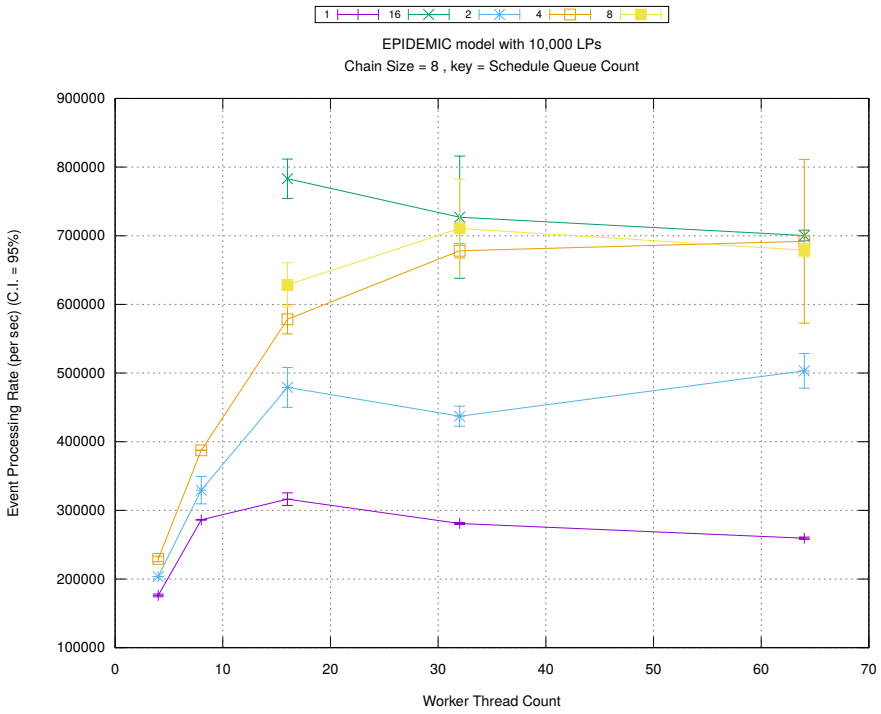
Table A.3: EPIDEMIC MODEL WITH WATTS-STROGATZ setup



(a) Simulation Runtime (in seconds)

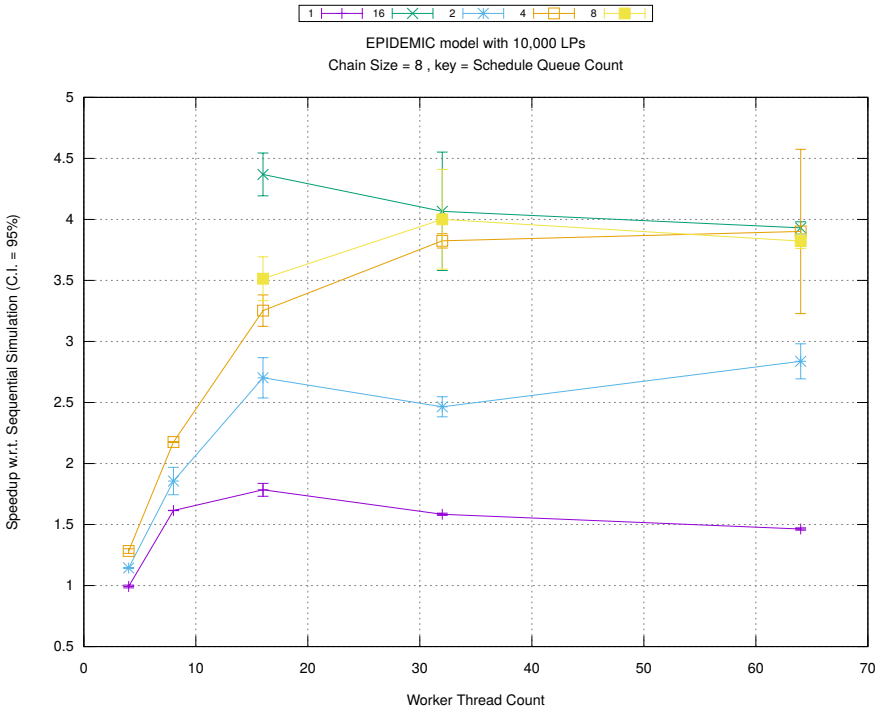


(b) Event Commitment Ratio

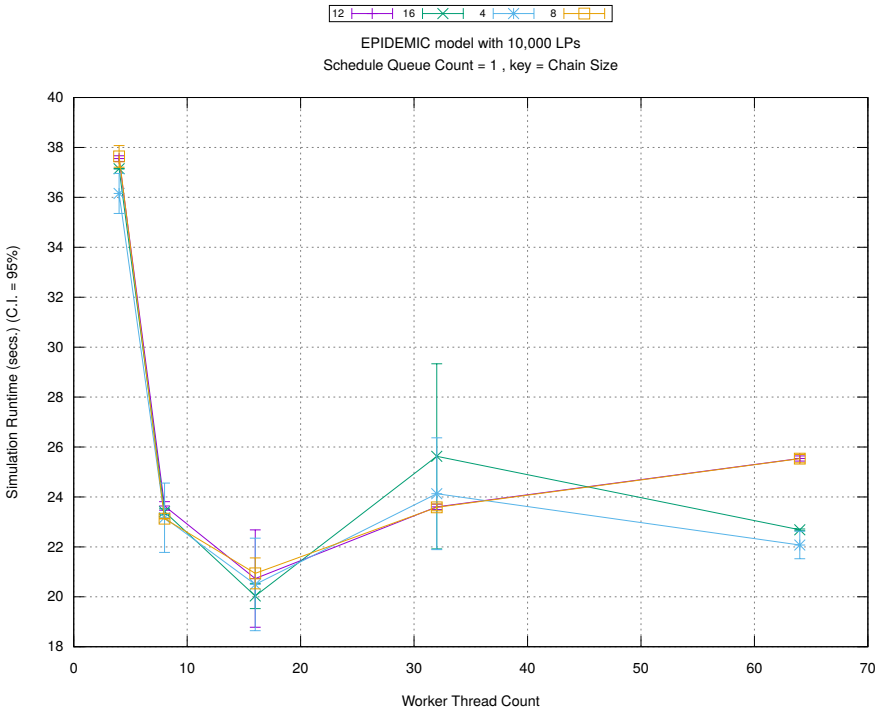


(c) Event Processing Rate (per second)

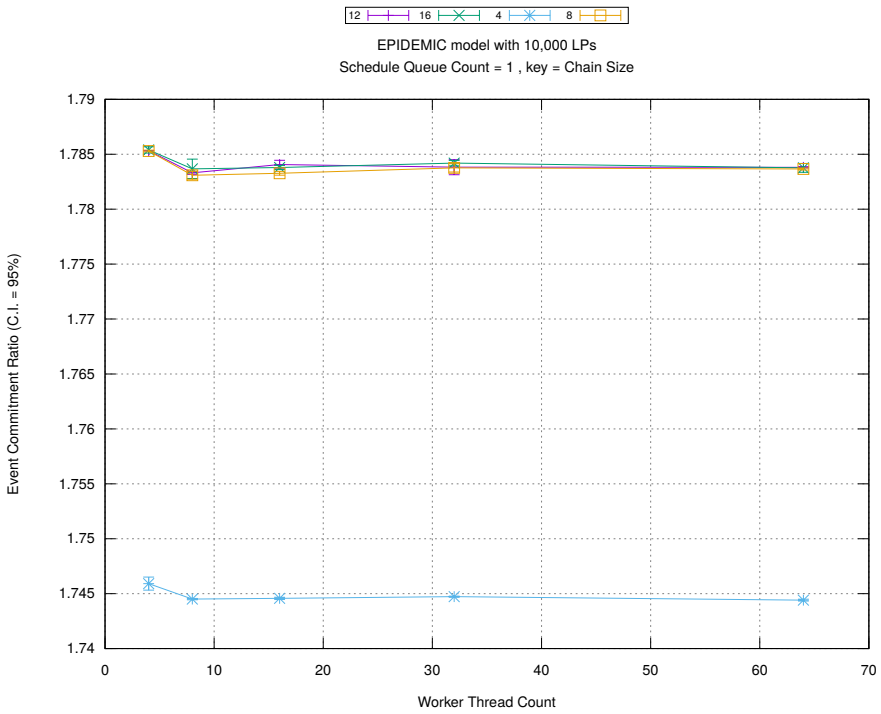
Figure A.65: epidemic 10k ws/plots/chains/threads vs count key chainsize 8



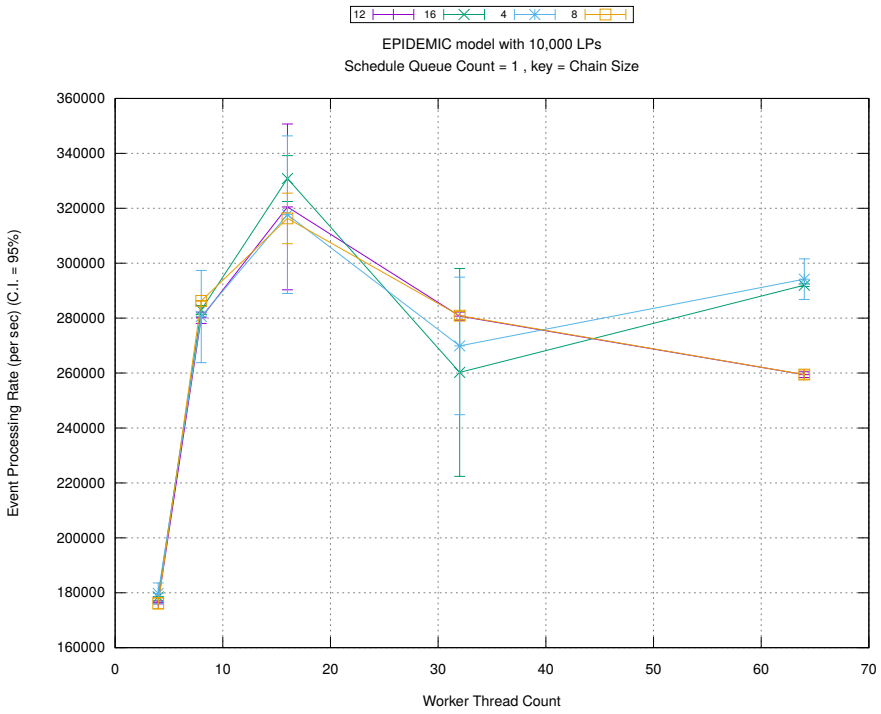
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

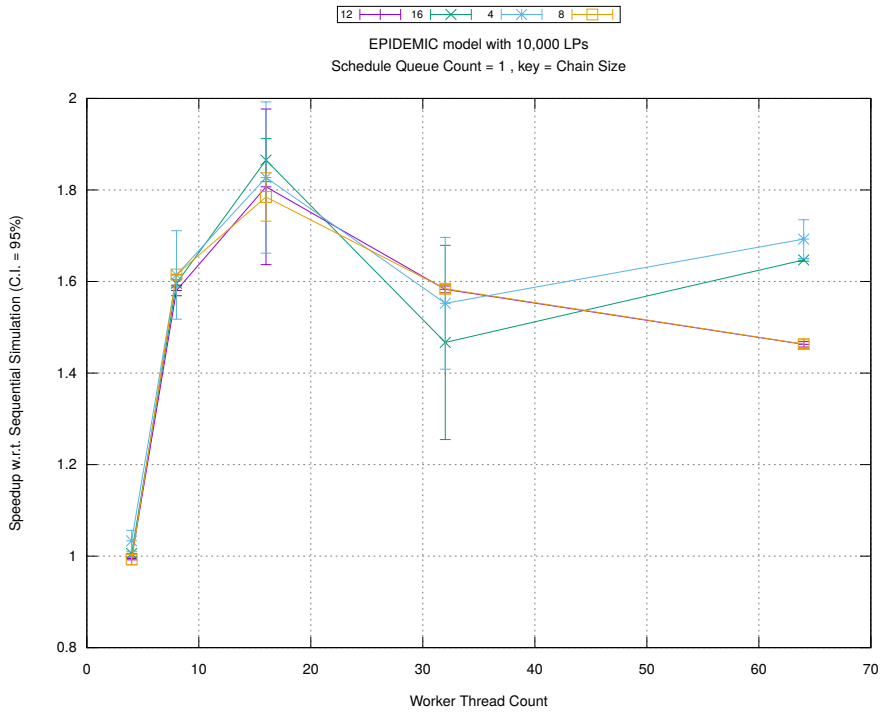


(b) Event Commitment Ratio

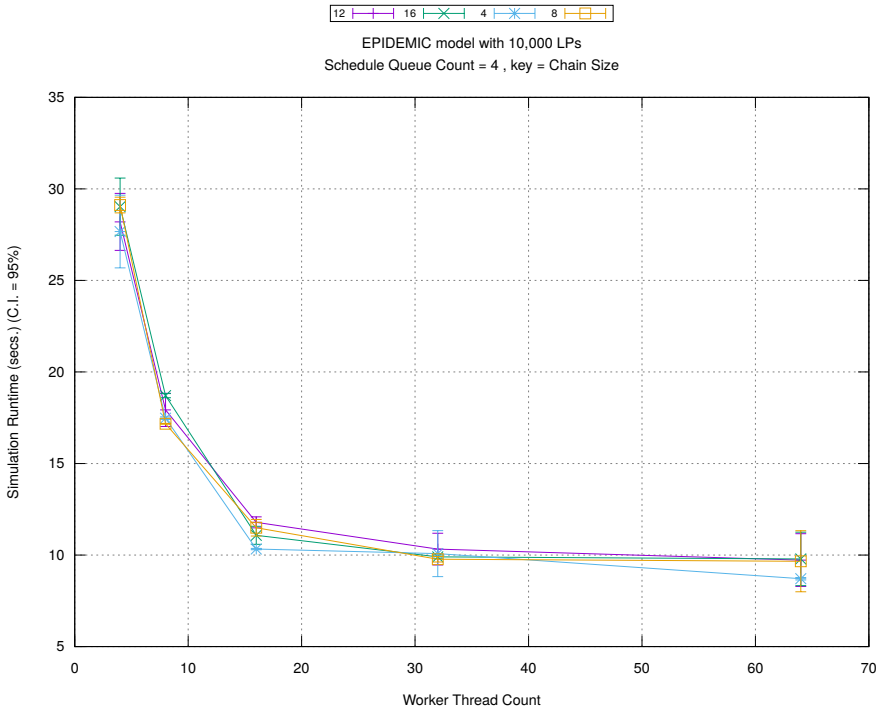


(c) Event Processing Rate (per second)

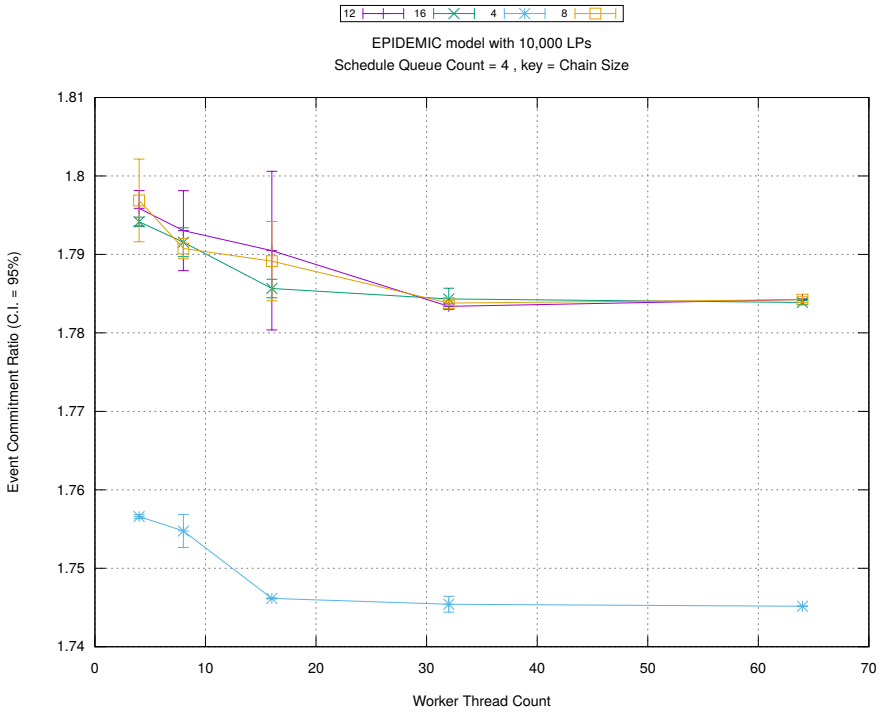
Figure A.66: epidemic 10k ws/plots/chains/threads vs chainsize key count 1



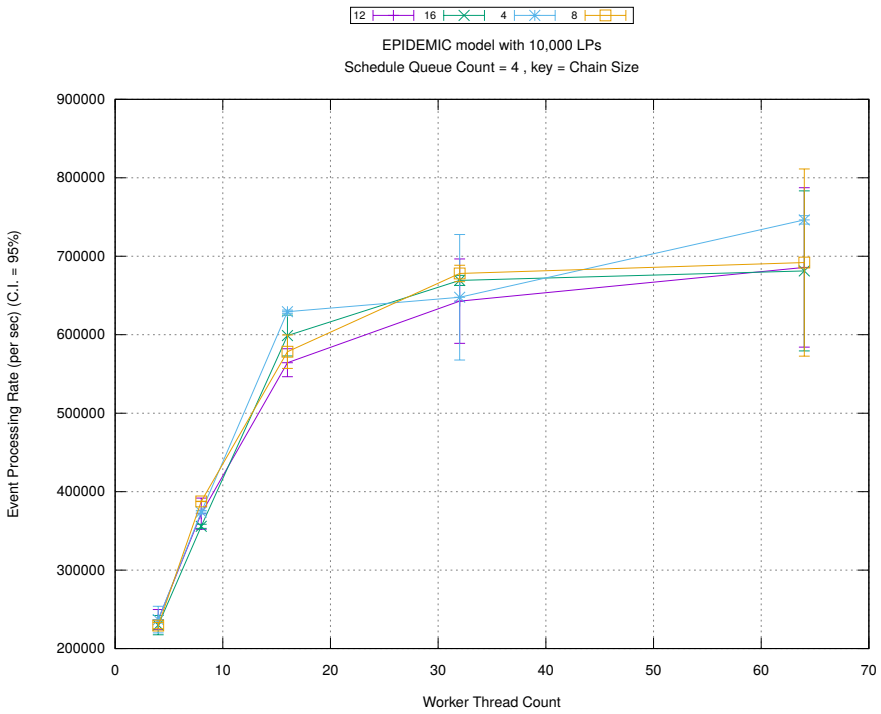
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

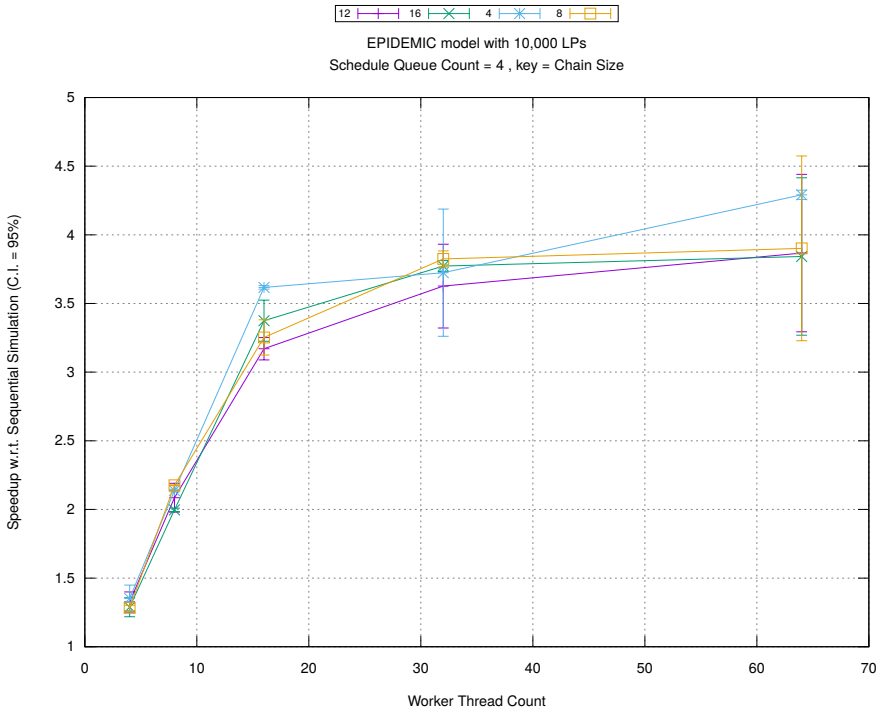


(b) Event Commitment Ratio

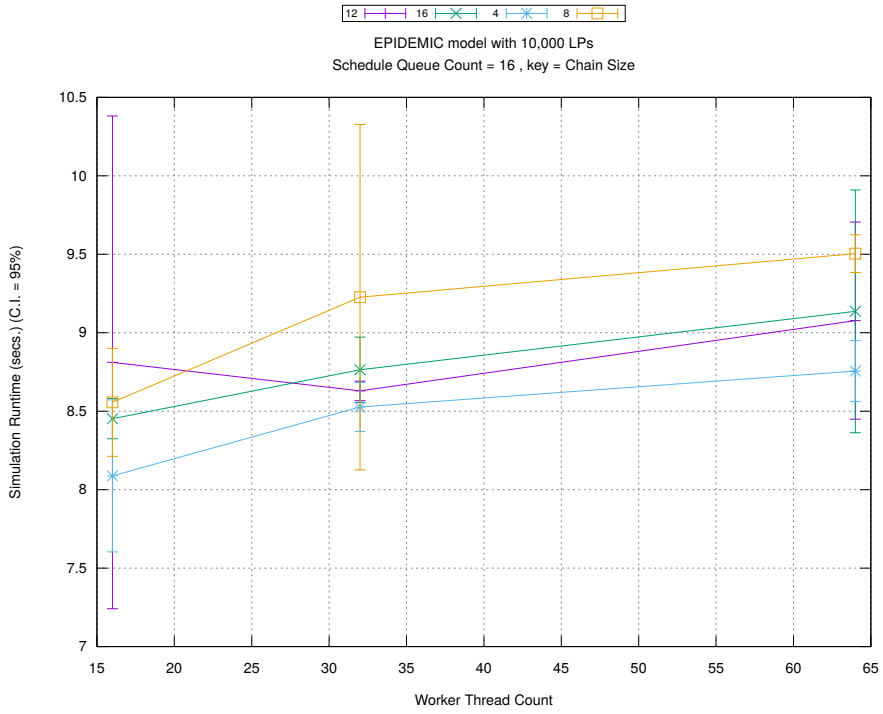


(c) Event Processing Rate (per second)

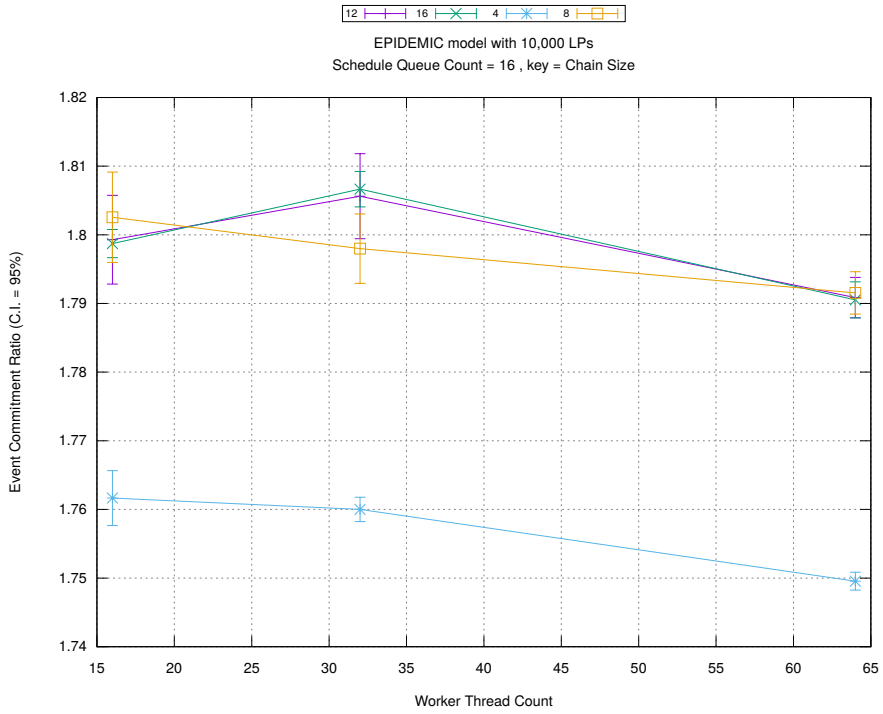
Figure A.67: epidemic 10k ws/plots/chains/threads vs chainsize key count 4



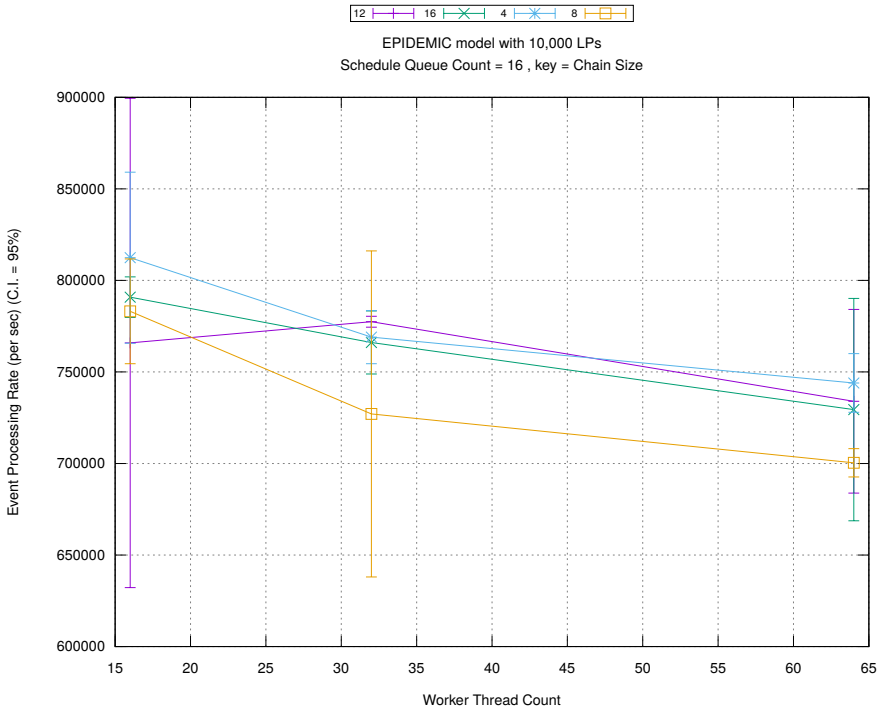
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

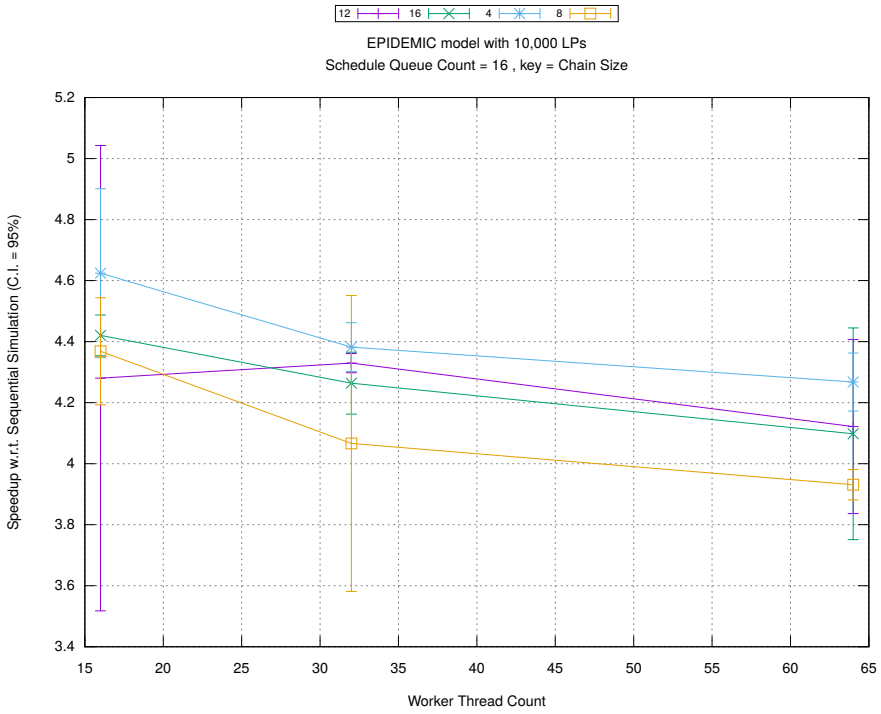


(b) Event Commitment Ratio

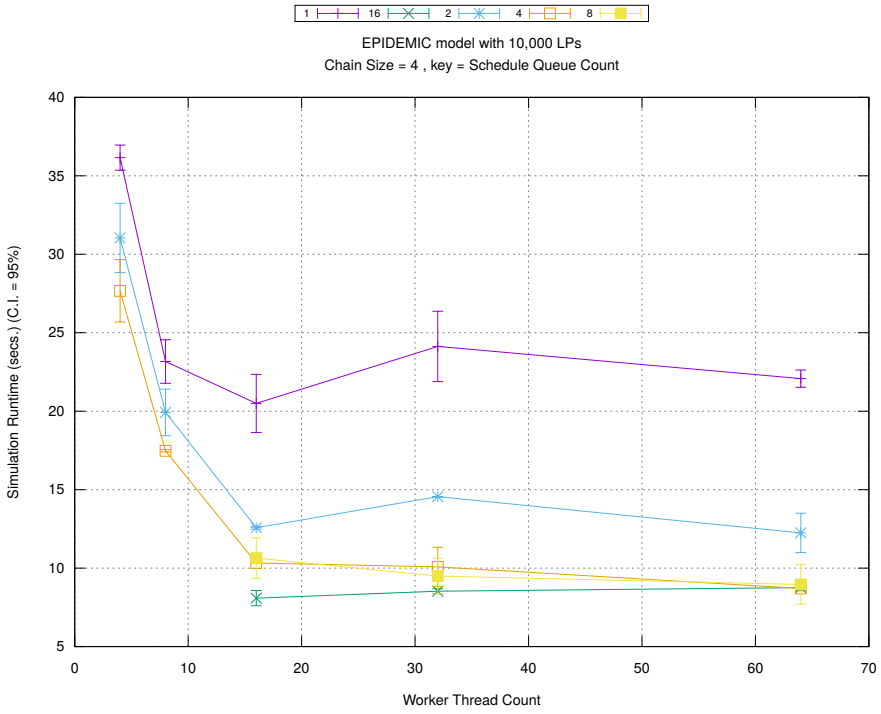


(c) Event Processing Rate (per second)

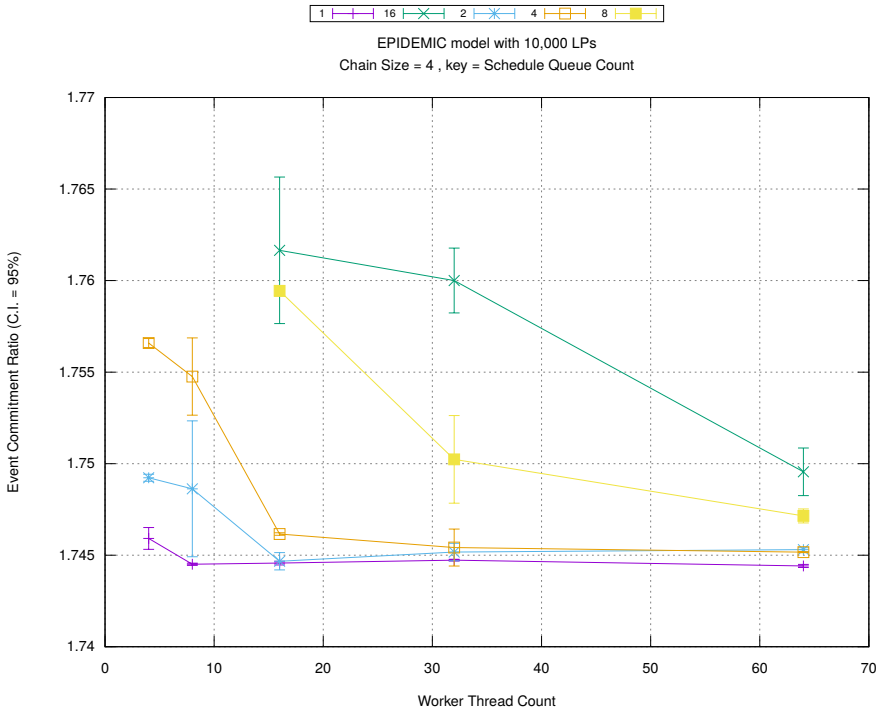
Figure A.68: epidemic 10k ws/plots/chains/threads vs chainsize key count 16



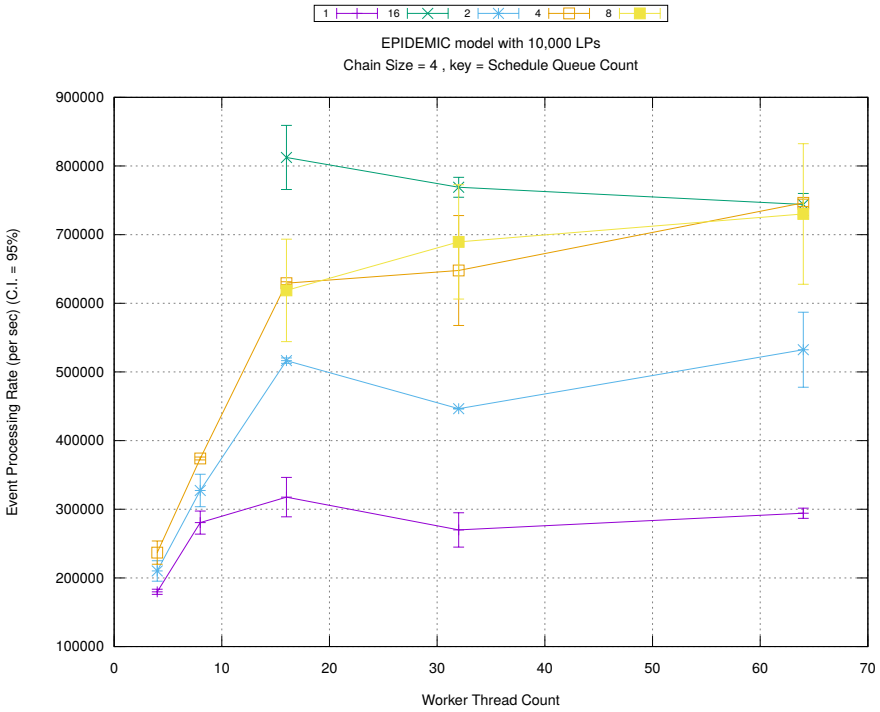
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

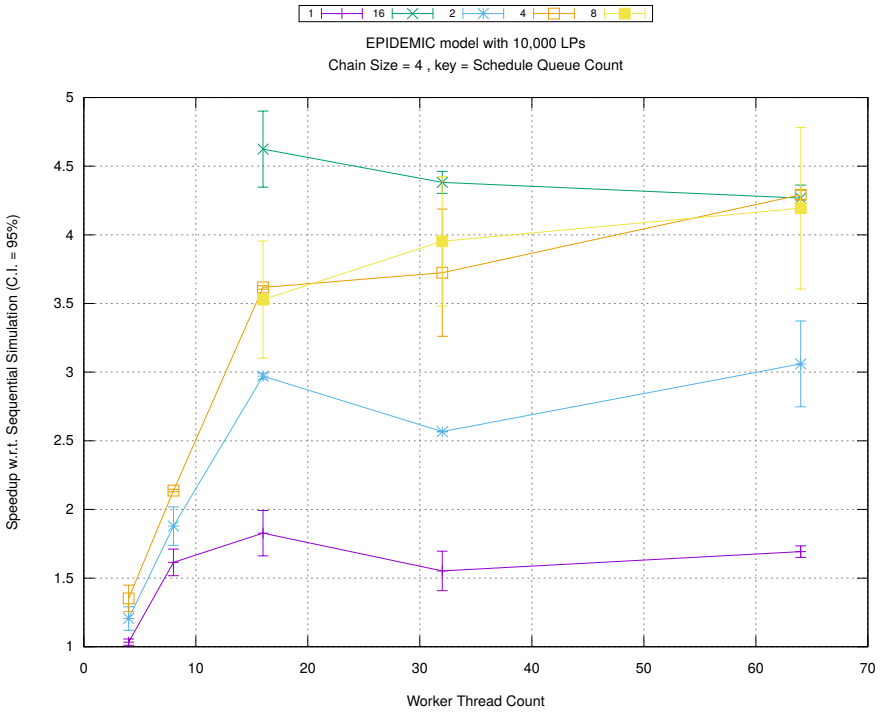


(b) Event Commitment Ratio

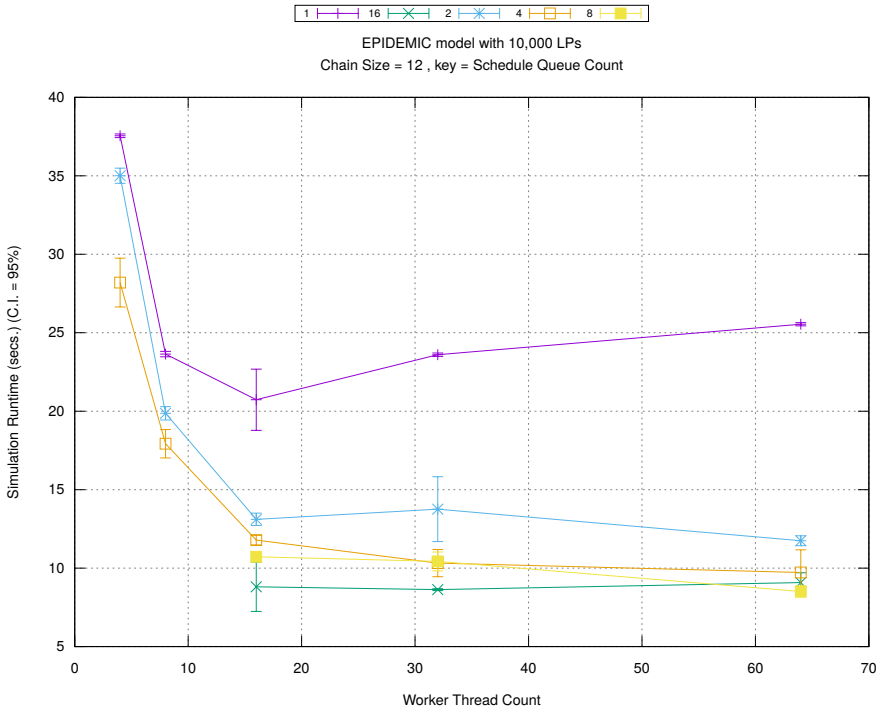


(c) Event Processing Rate (per second)

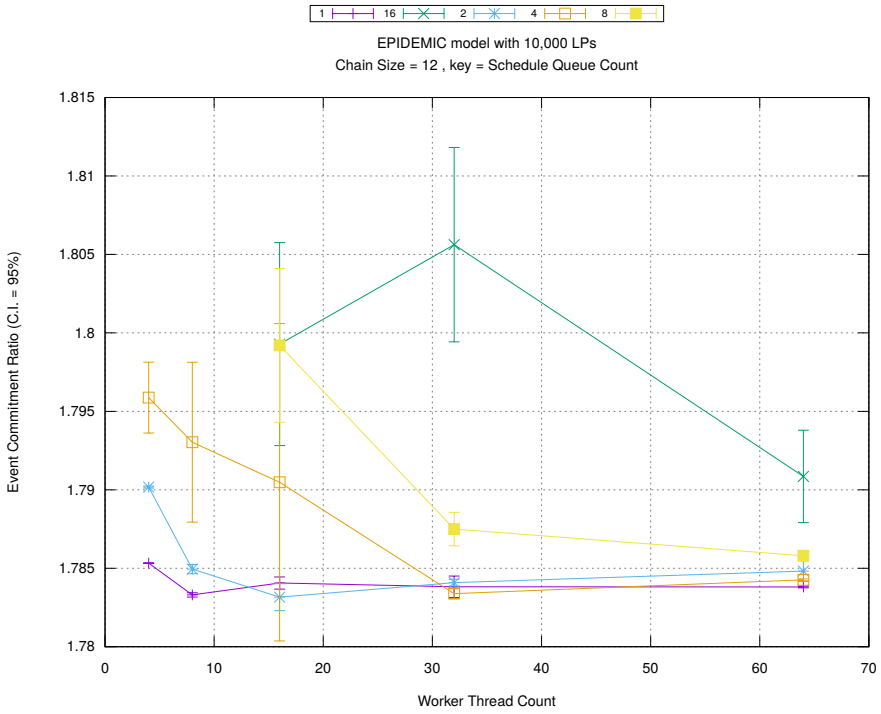
Figure A.69: epidemic 10k ws/plots/chains/threads vs count key chainsize 4



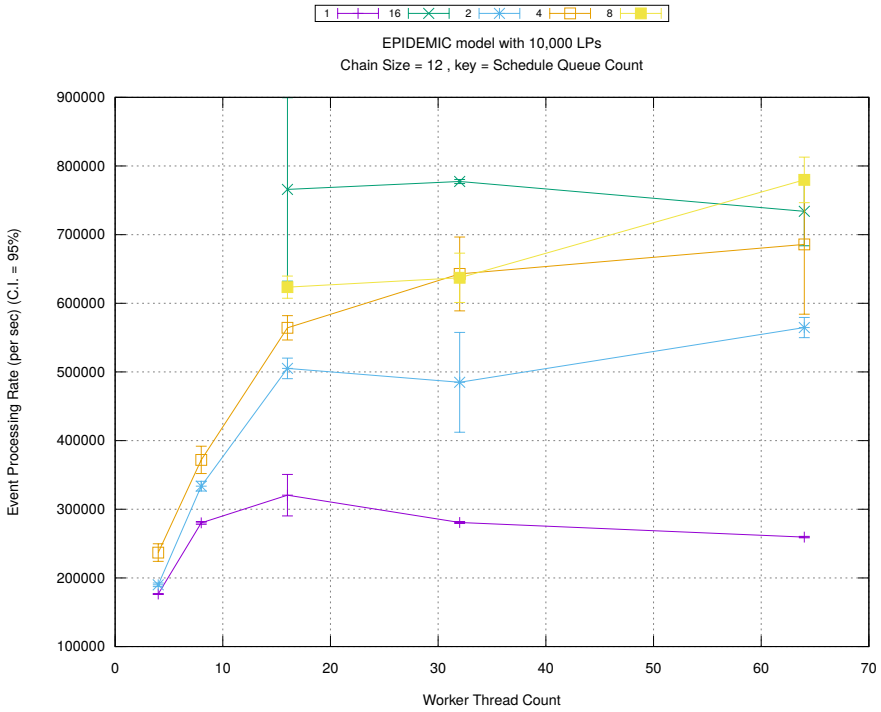
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

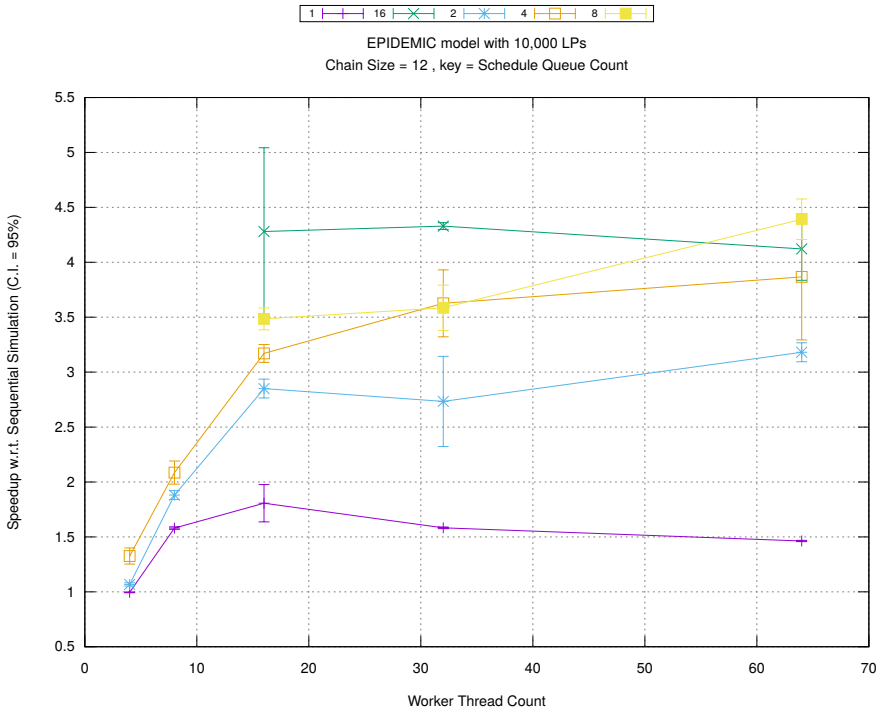


(b) Event Commitment Ratio

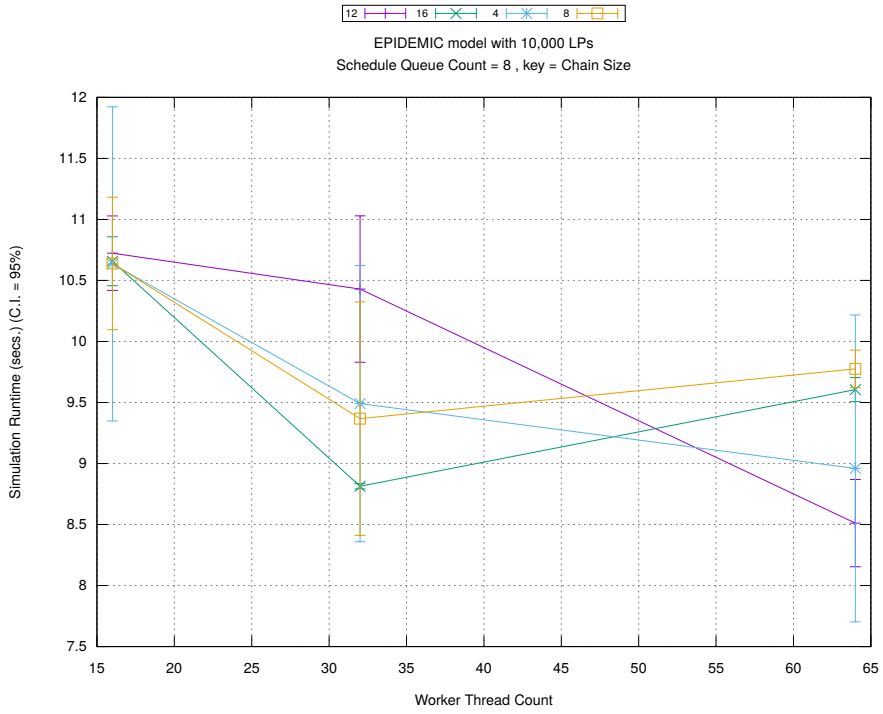


(c) Event Processing Rate (per second)

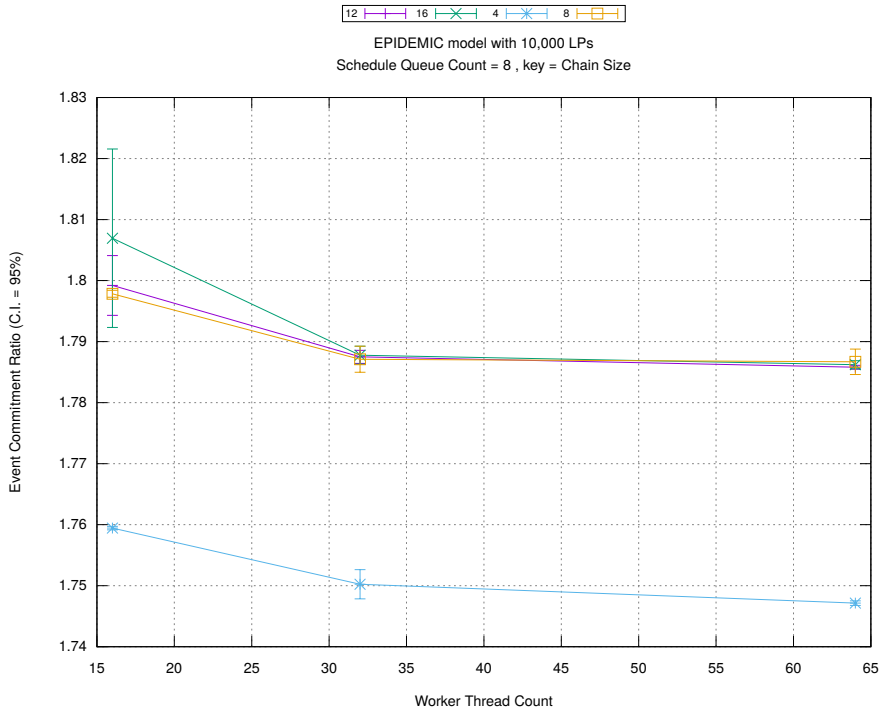
Figure A.70: epidemic 10k ws/plots/chains/threads vs count key chainsize 12



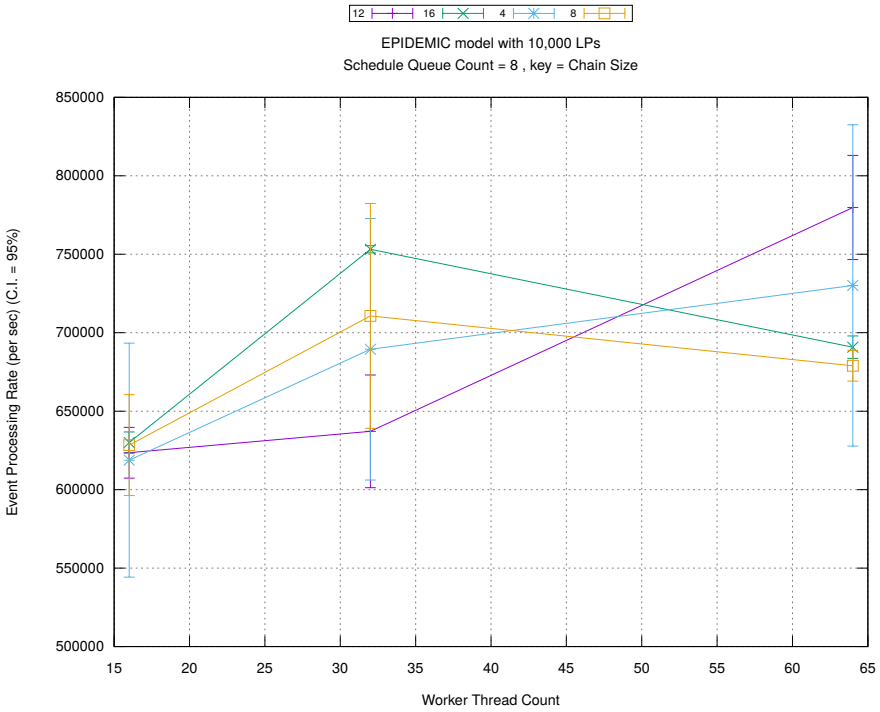
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

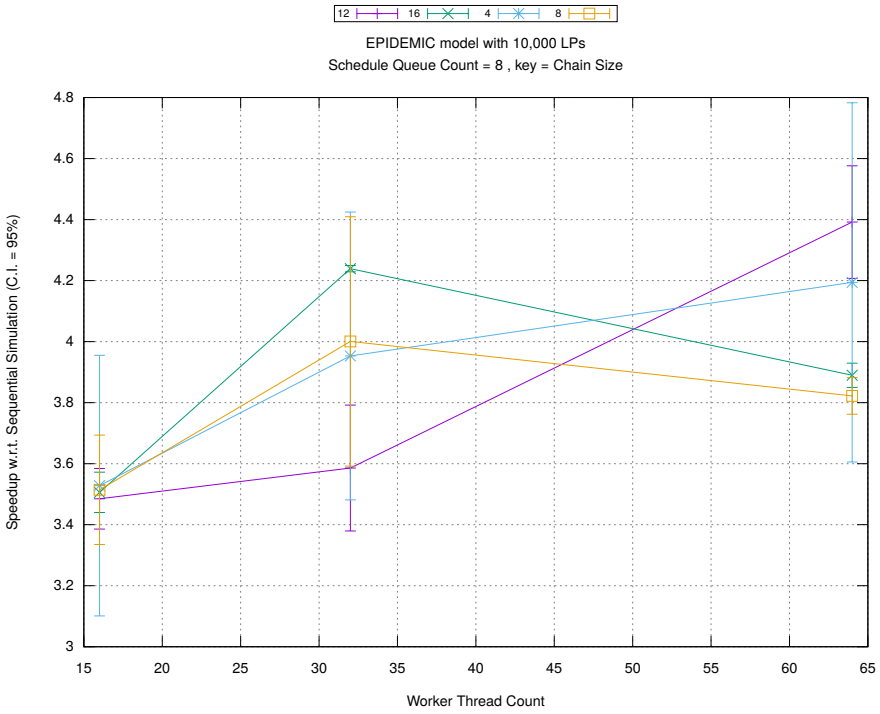


(b) Event Commitment Ratio

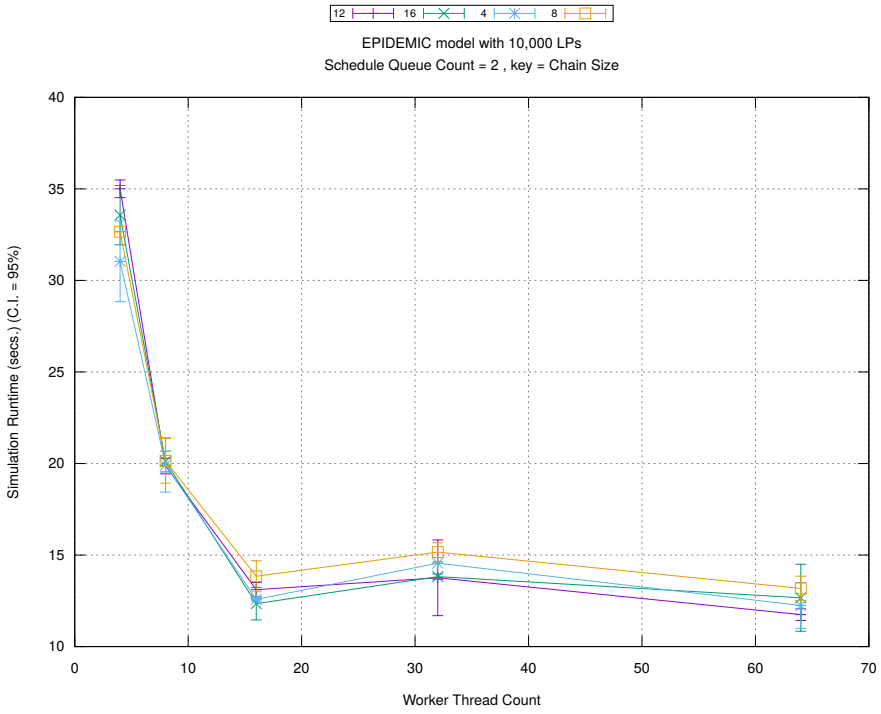


(c) Event Processing Rate (per second)

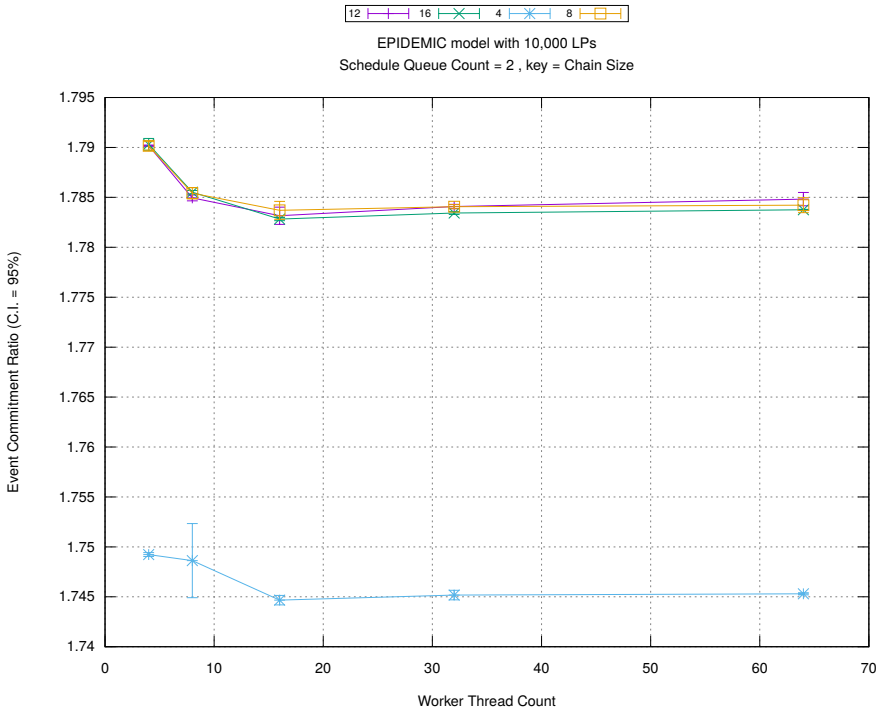
Figure A.71: epidemic 10k ws/plots/chains/threads vs chainsize key count 8



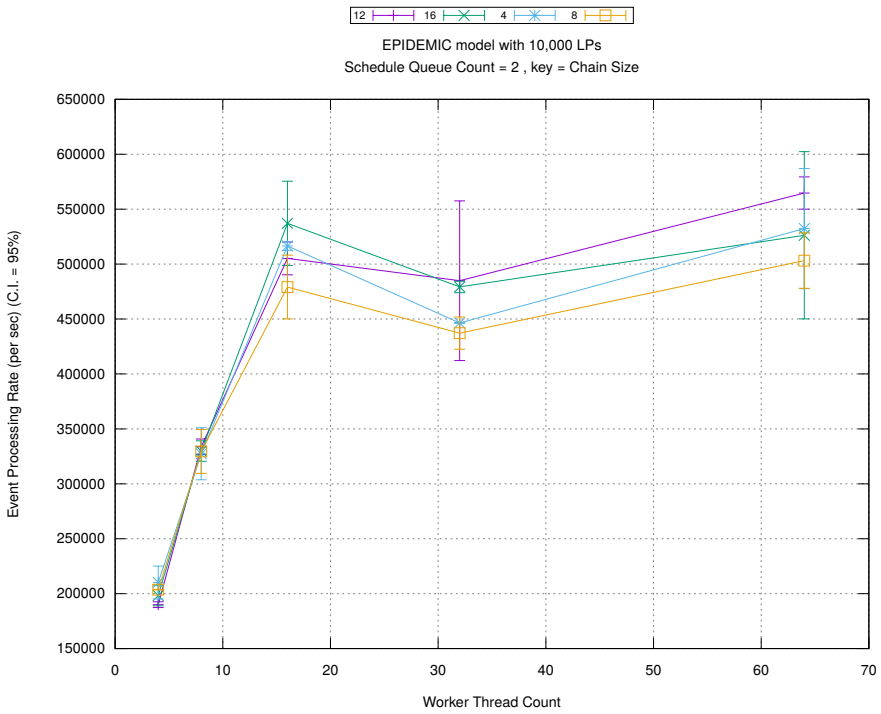
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

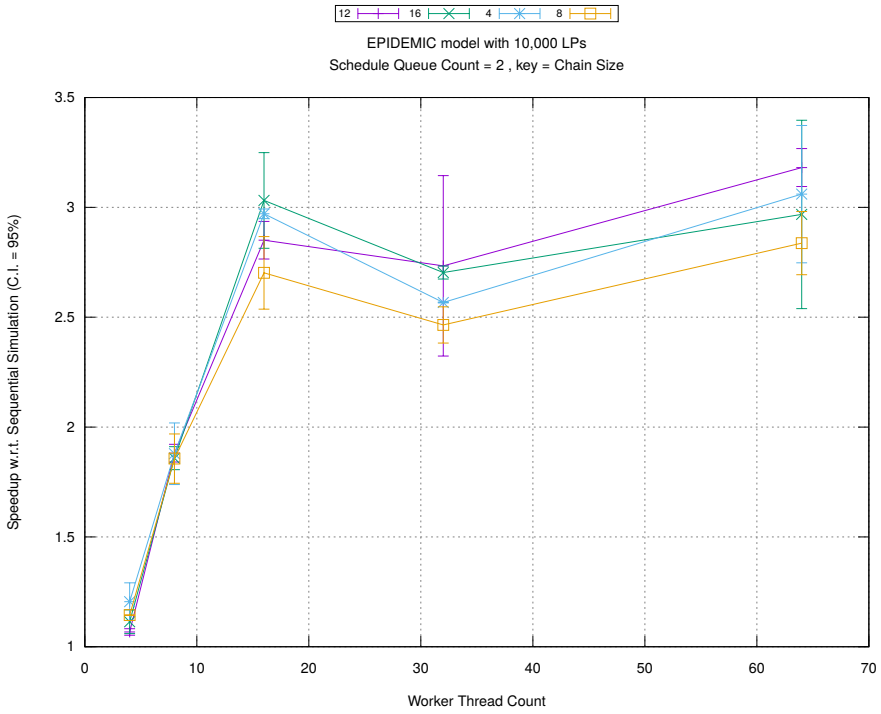


(b) Event Commitment Ratio

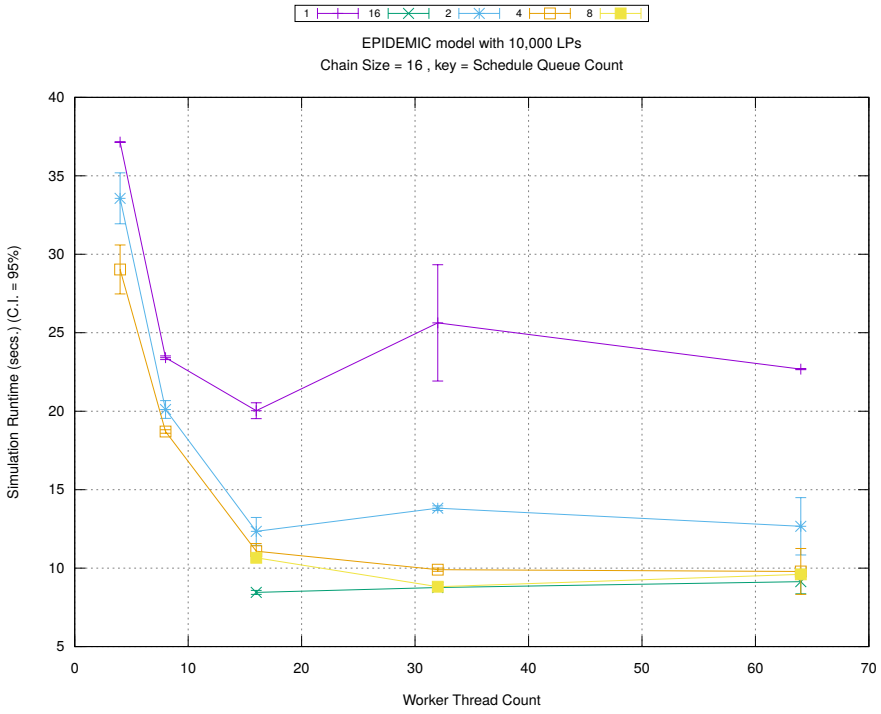


(c) Event Processing Rate (per second)

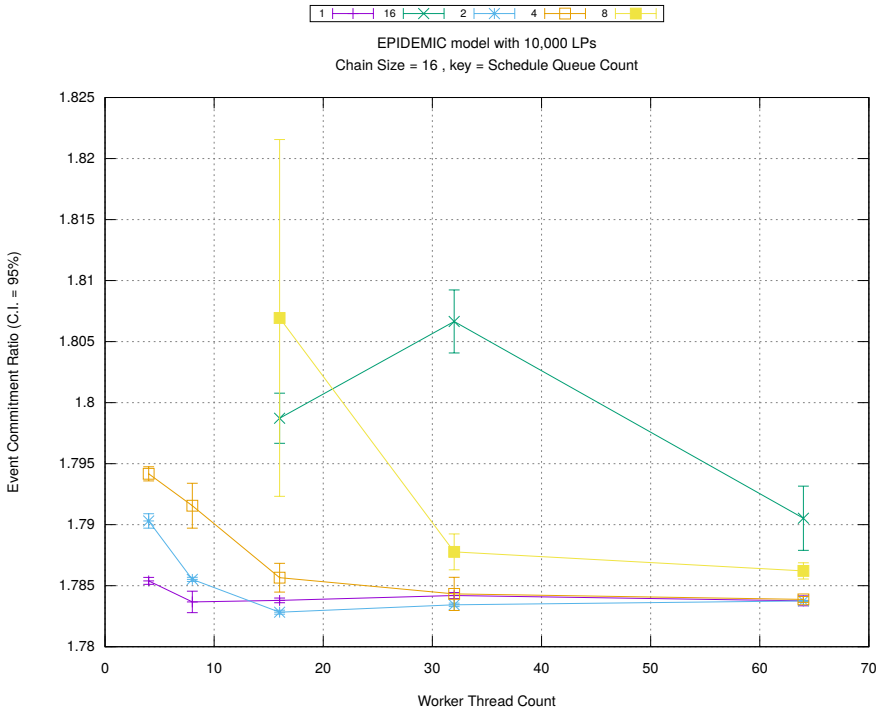
Figure A.72: epidemic 10k ws/plots/chains/threads vs chainsize key count 2



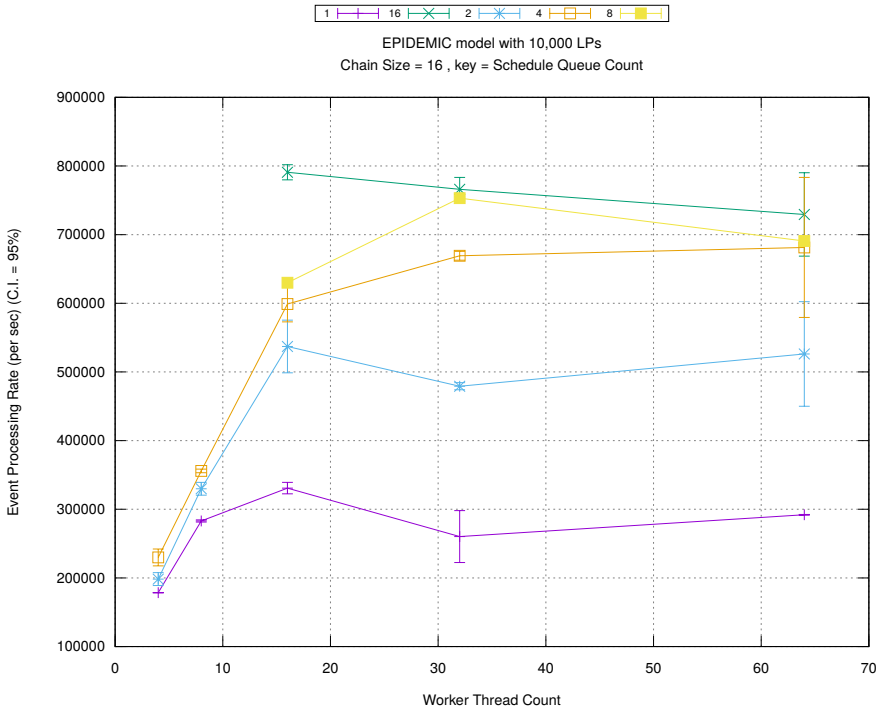
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

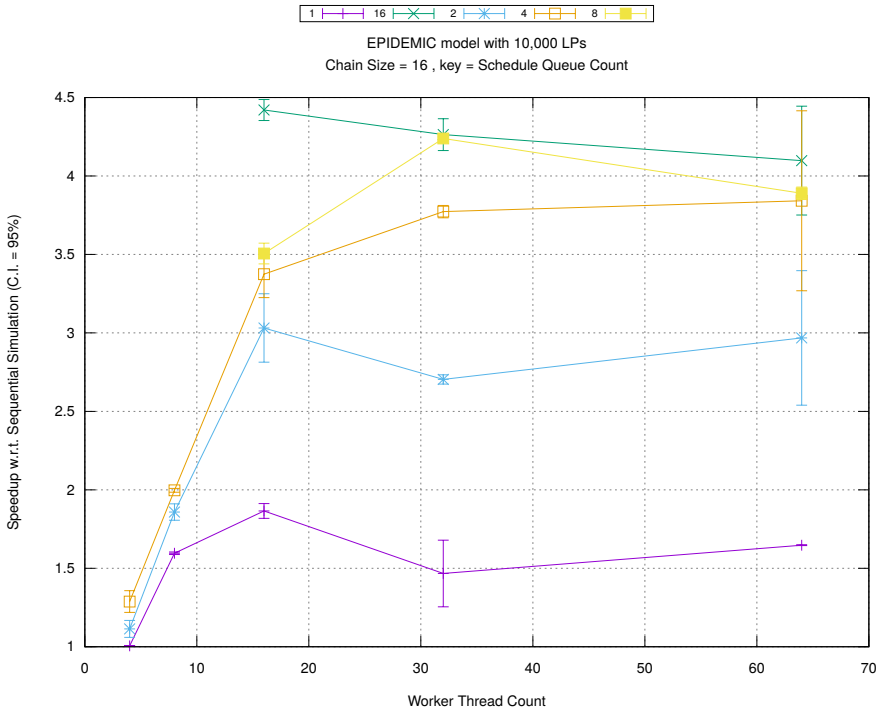


(b) Event Commitment Ratio

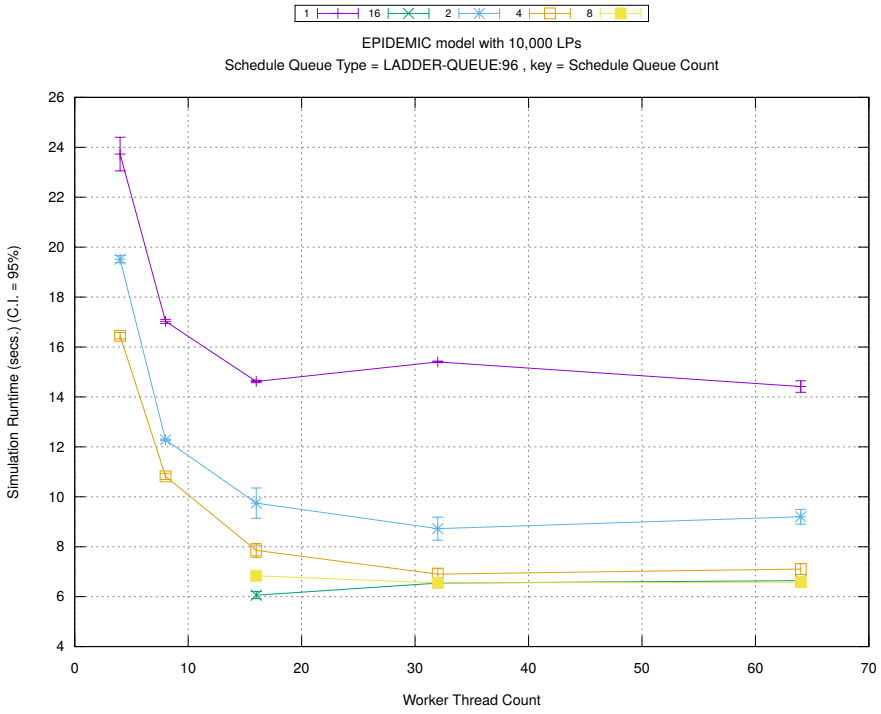


(c) Event Processing Rate (per second)

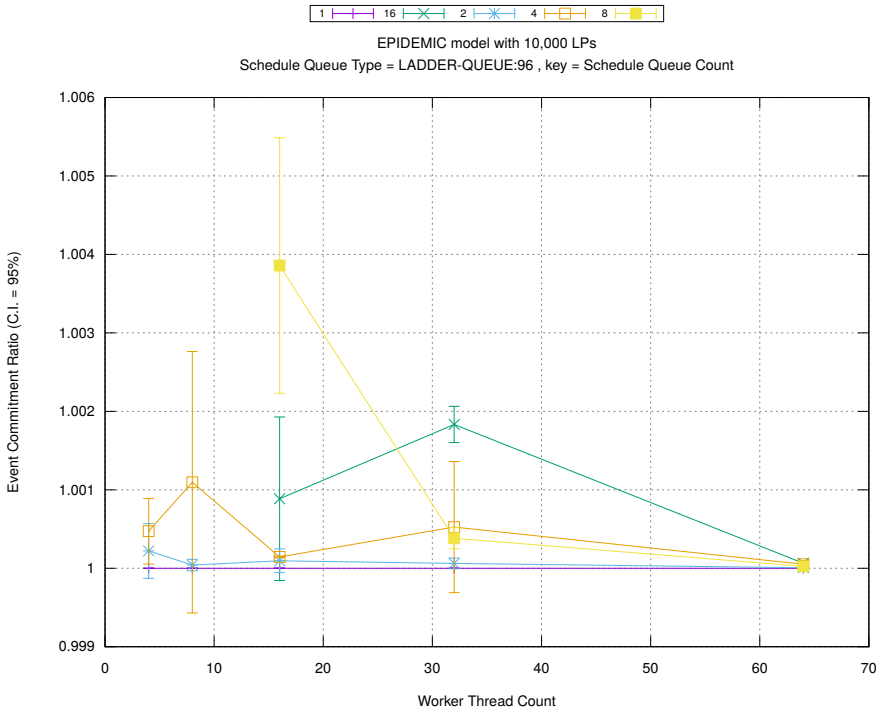
Figure A.73: epidemic 10k ws/plots/chains/threads vs count key chainsize 16



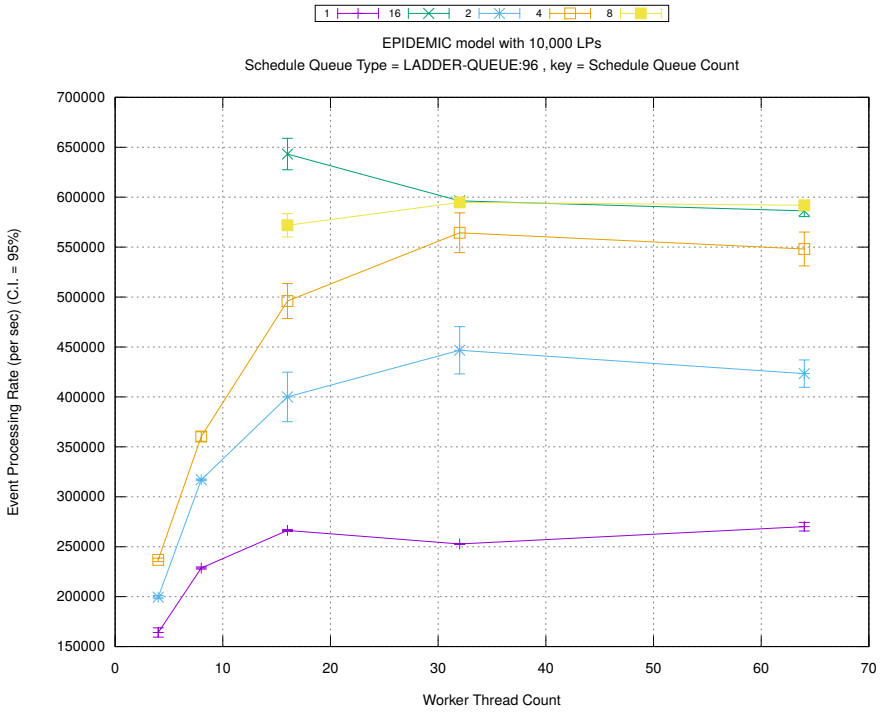
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

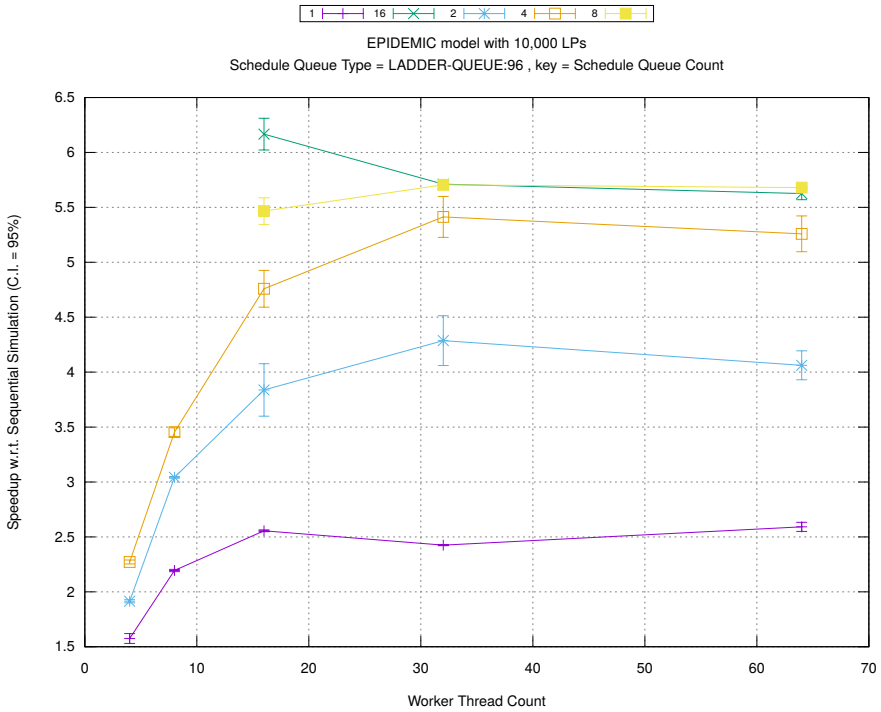


(b) Event Commitment Ratio

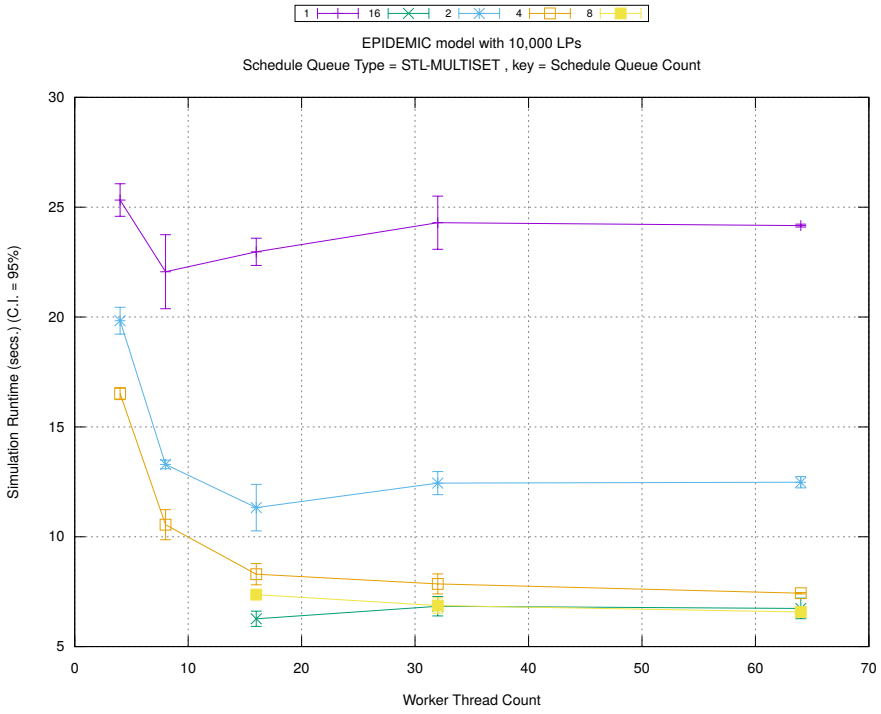


(c) Event Processing Rate (per second)

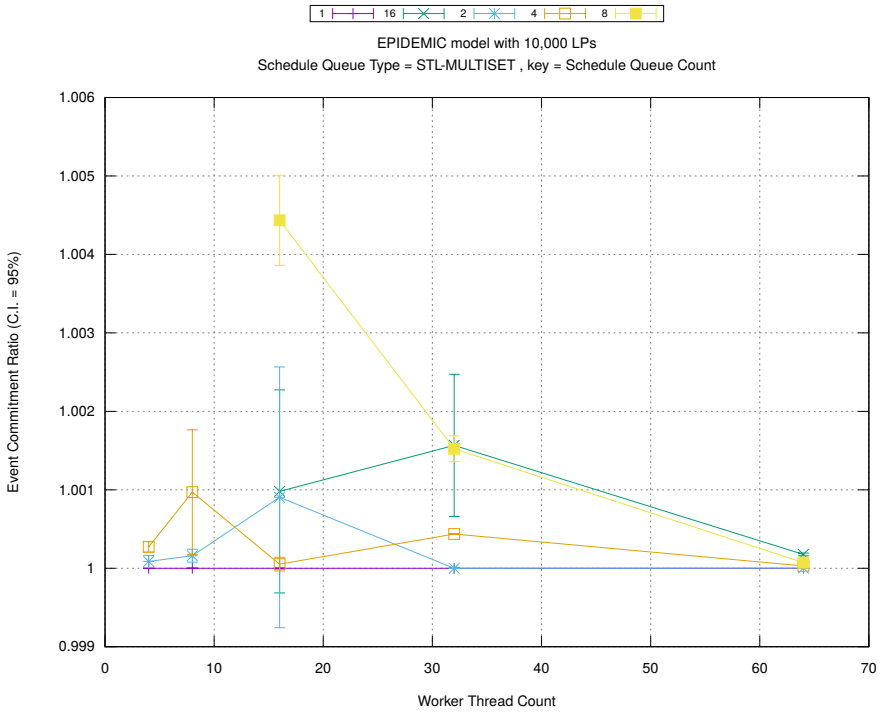
Figure A.74: epidemic 10k ws/plots/scheduleq/threads vs count key type ladder-queue 96



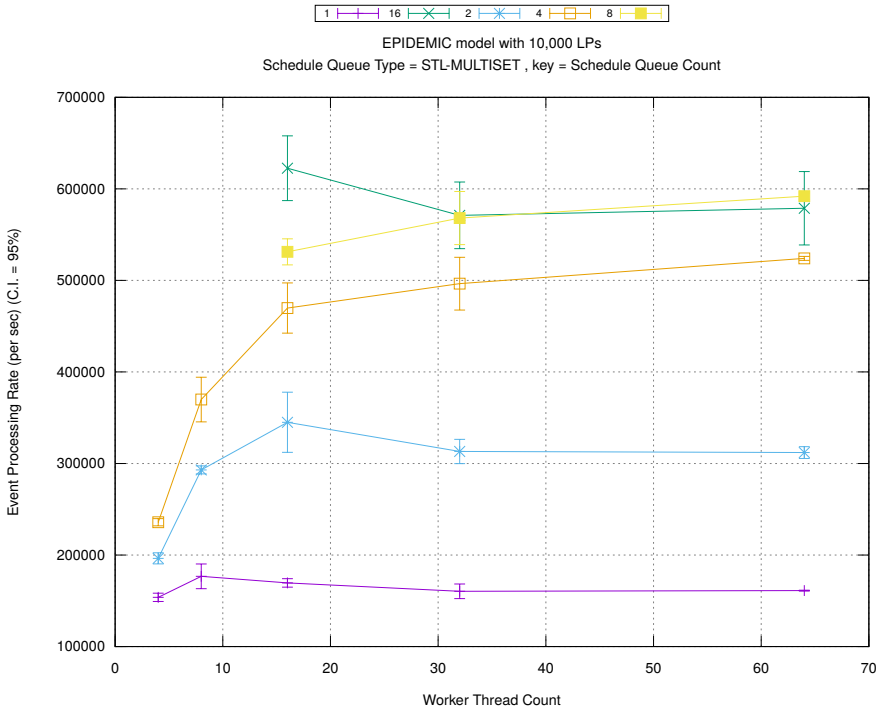
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

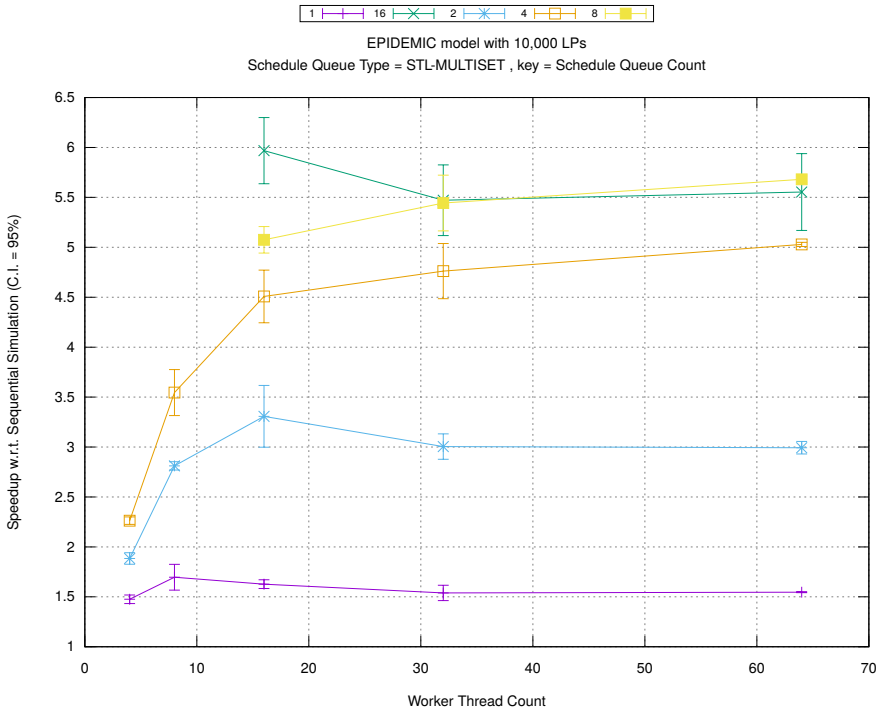


(b) Event Commitment Ratio

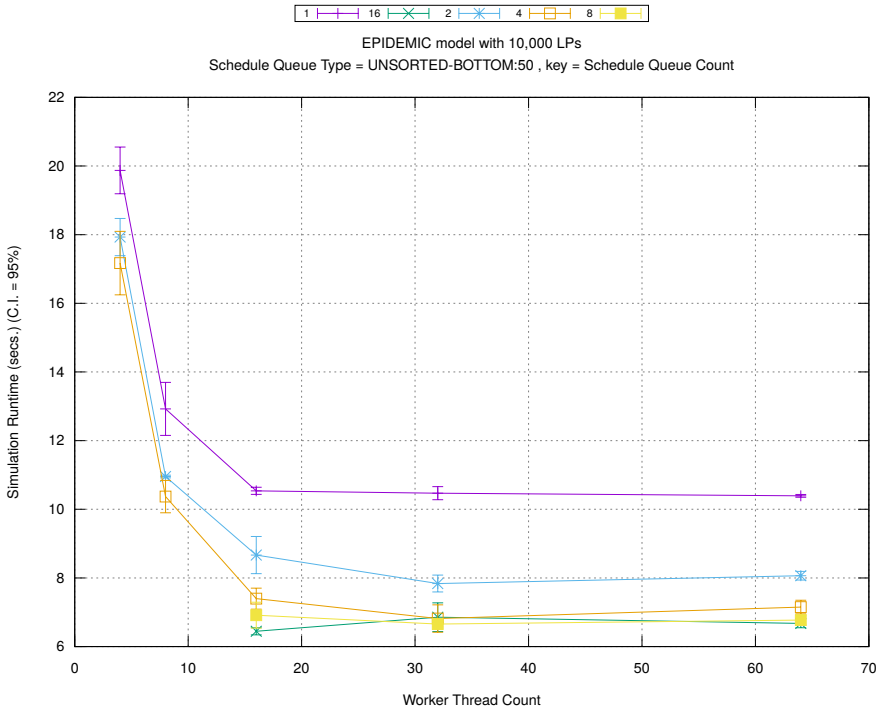


(c) Event Processing Rate (per second)

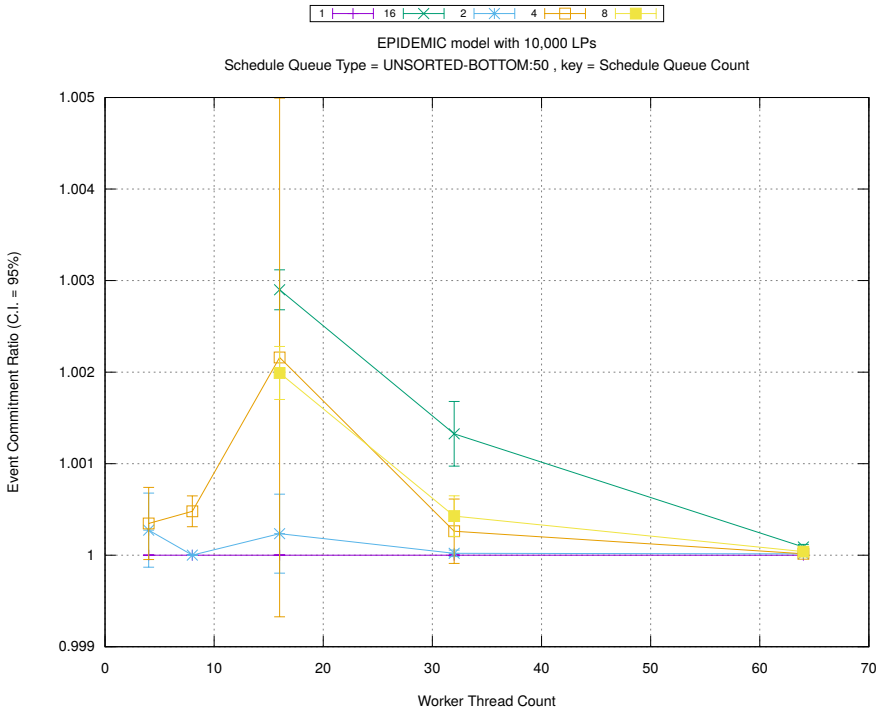
Figure A.75: epidemic 10k ws/plots/scheduleq/threads vs count key type stl-multiset



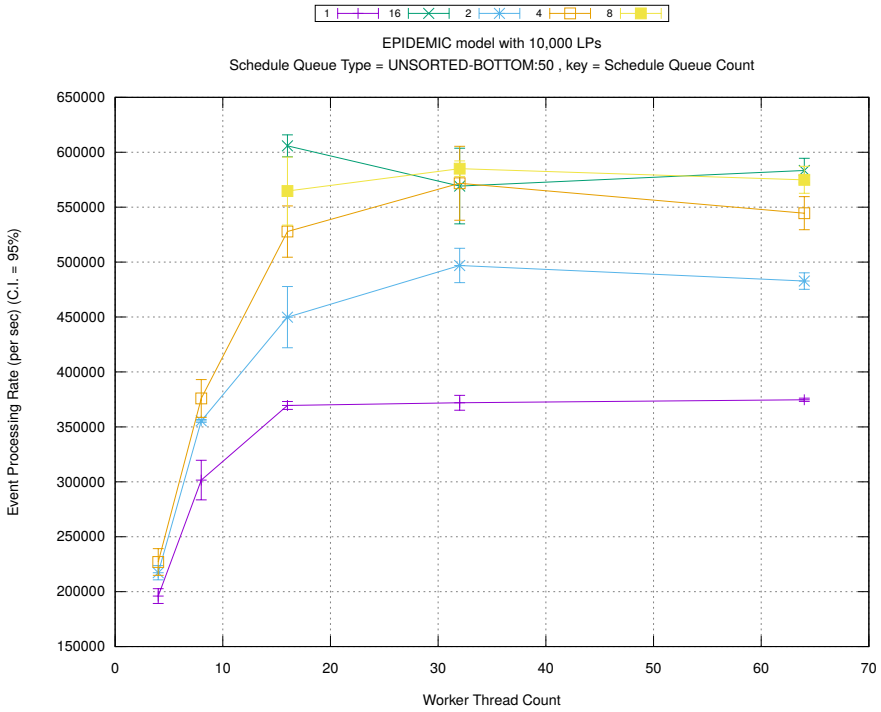
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

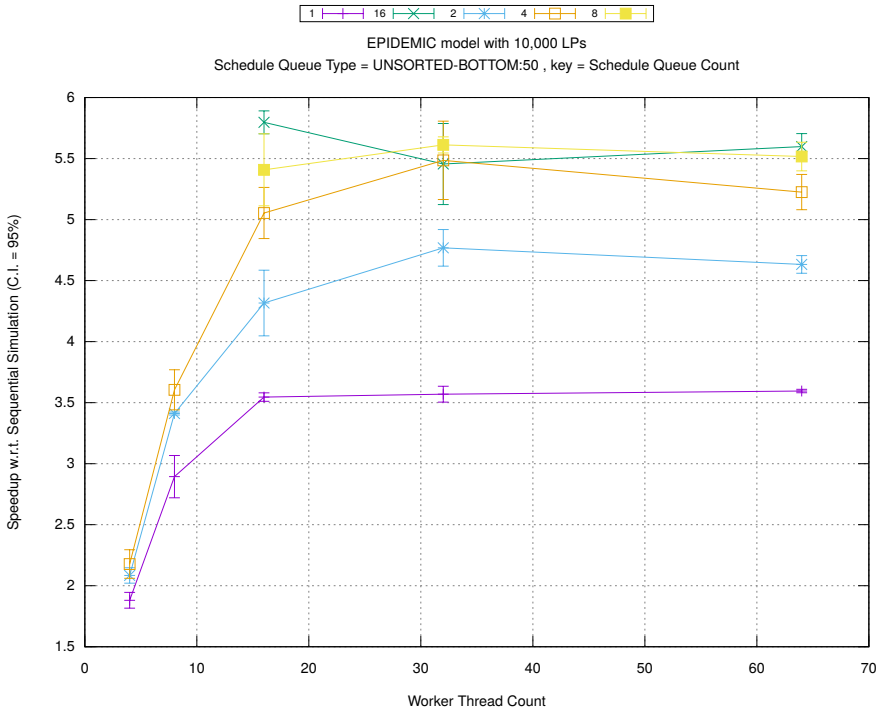


(b) Event Commitment Ratio

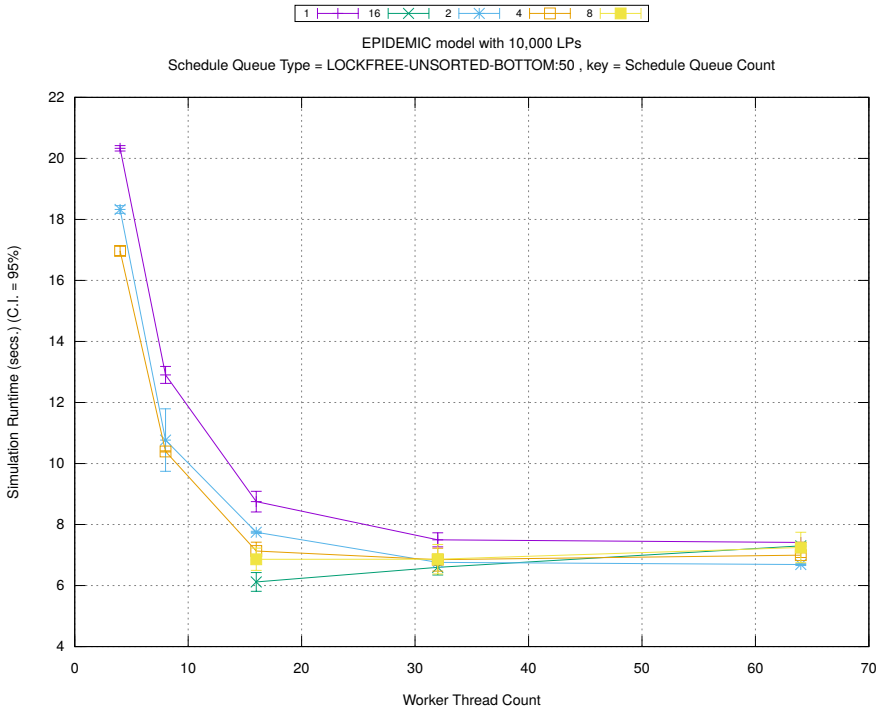


(c) Event Processing Rate (per second)

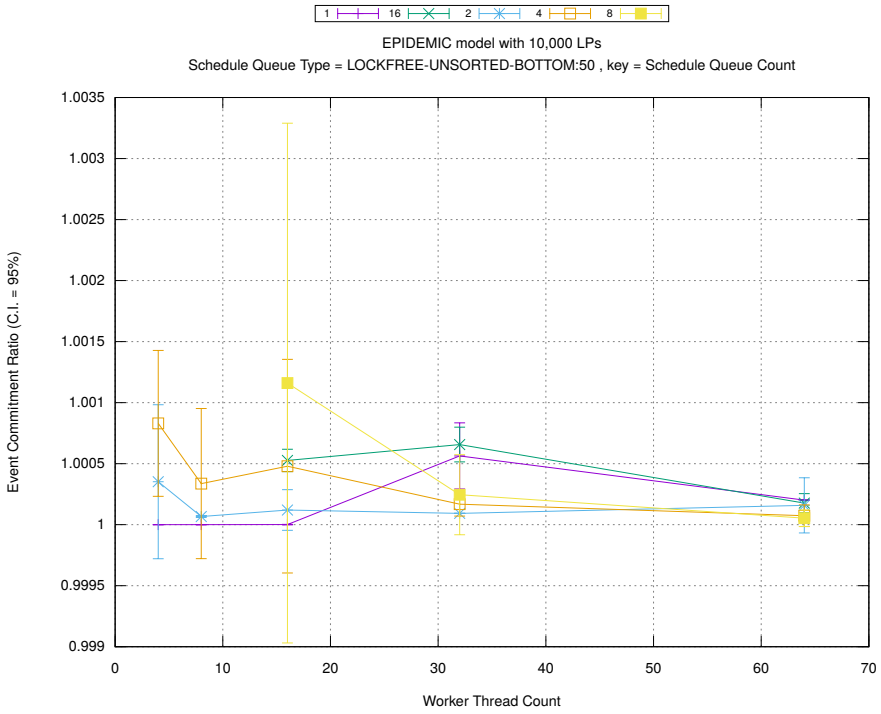
Figure A.76: epidemic 10k ws/plots/scheduleq/threads vs count key type unsorted-bottom 50



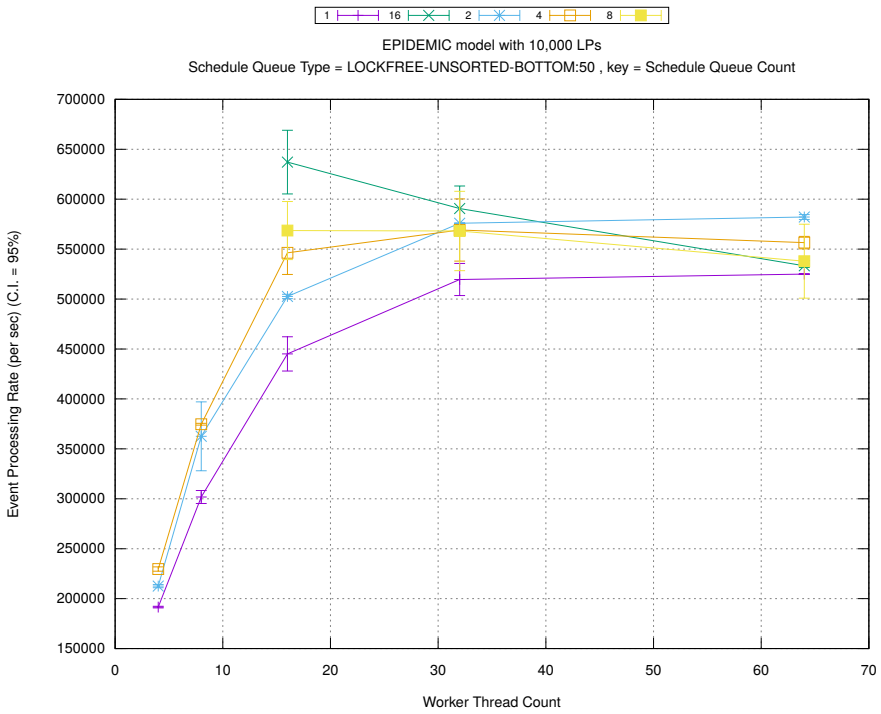
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

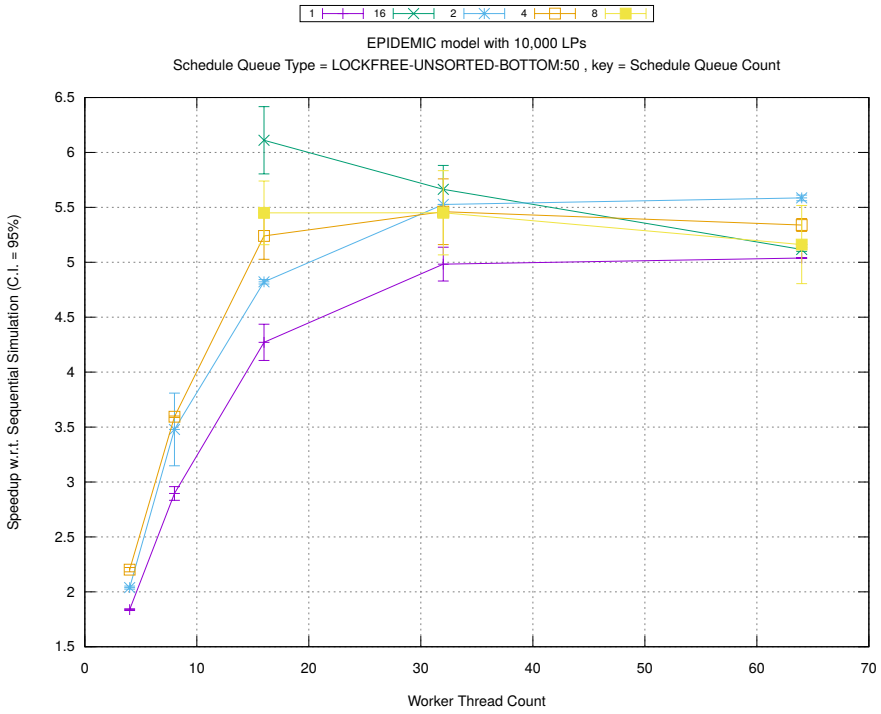


(b) Event Commitment Ratio

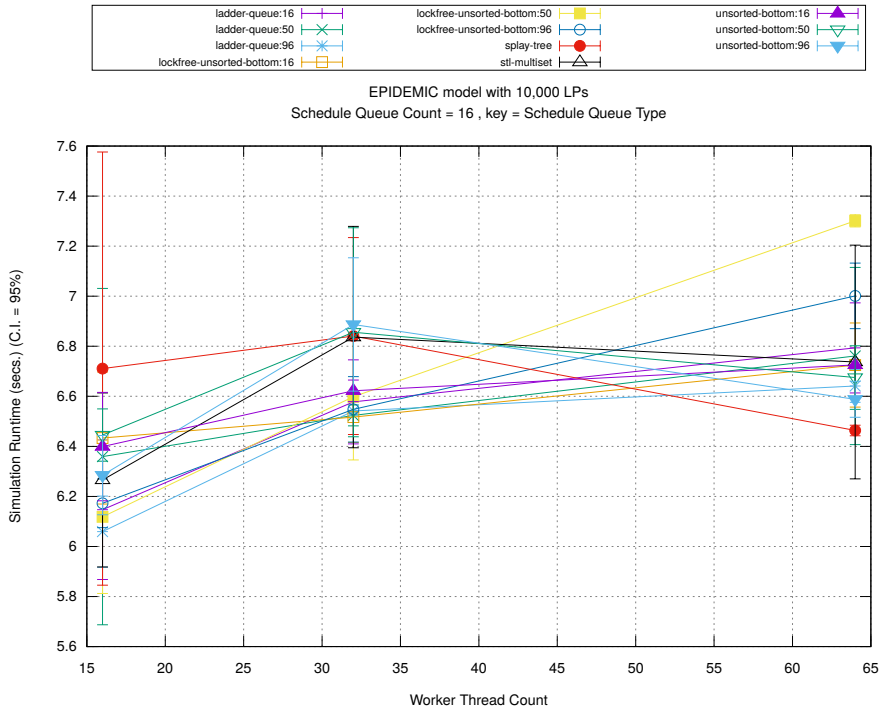


(c) Event Processing Rate (per second)

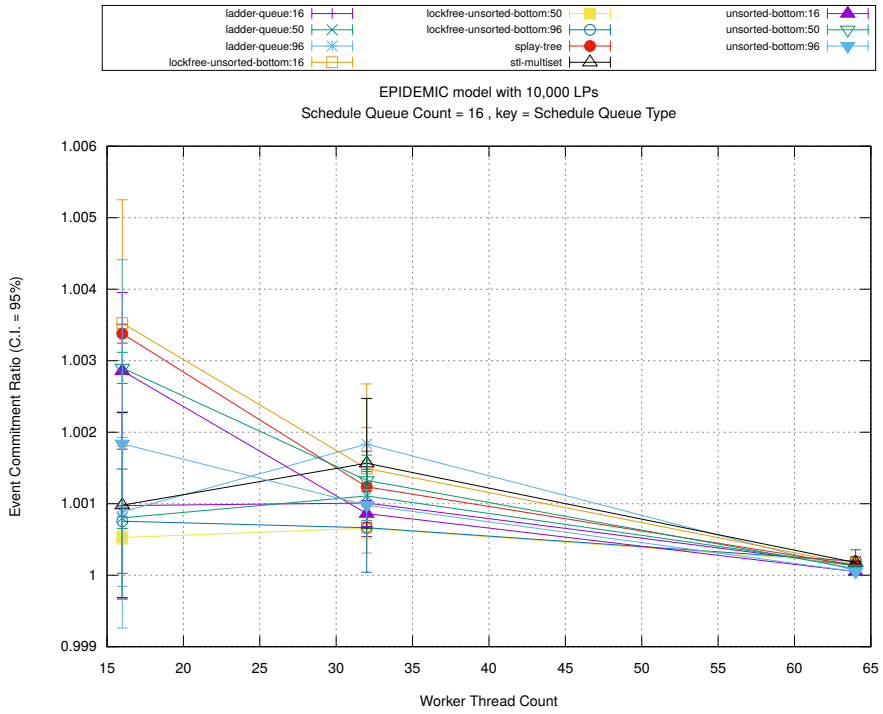
Figure A.77: epidemic 10k ws/plots/scheduleq/threads vs count key type lockfree-unsorted-bottom 50



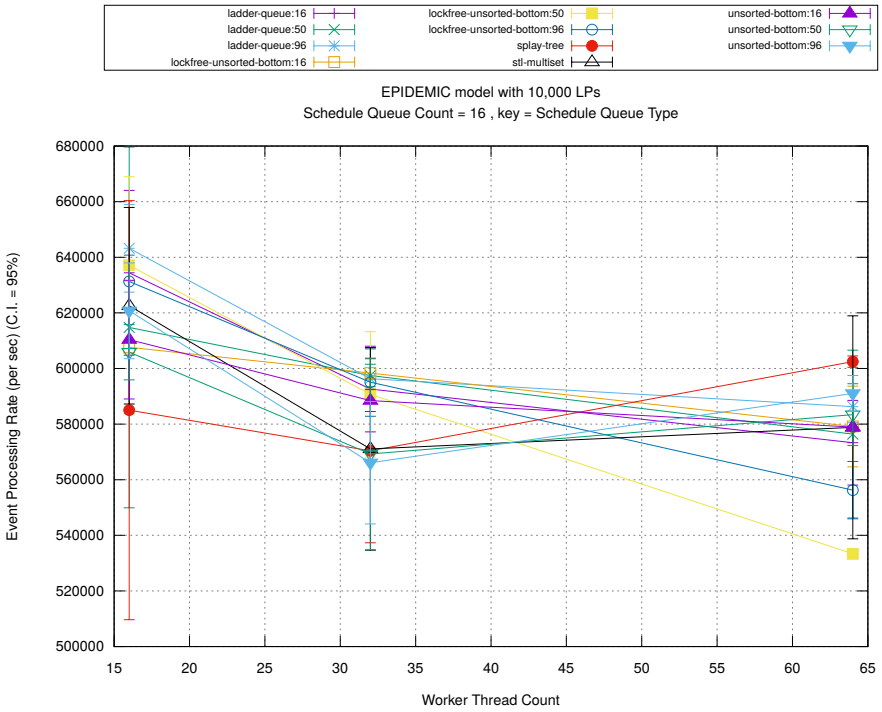
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

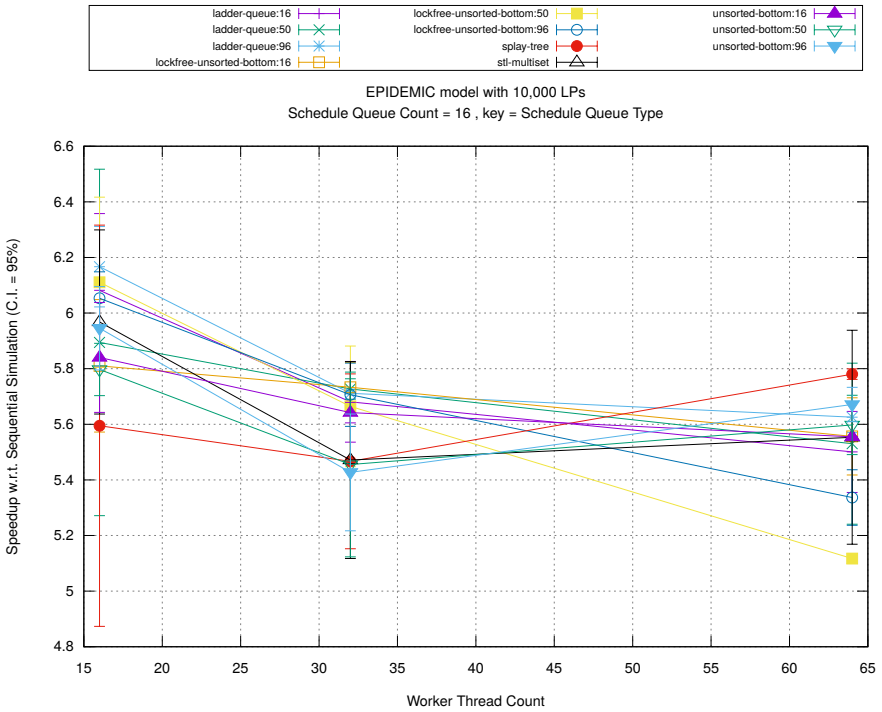


(b) Event Commitment Ratio

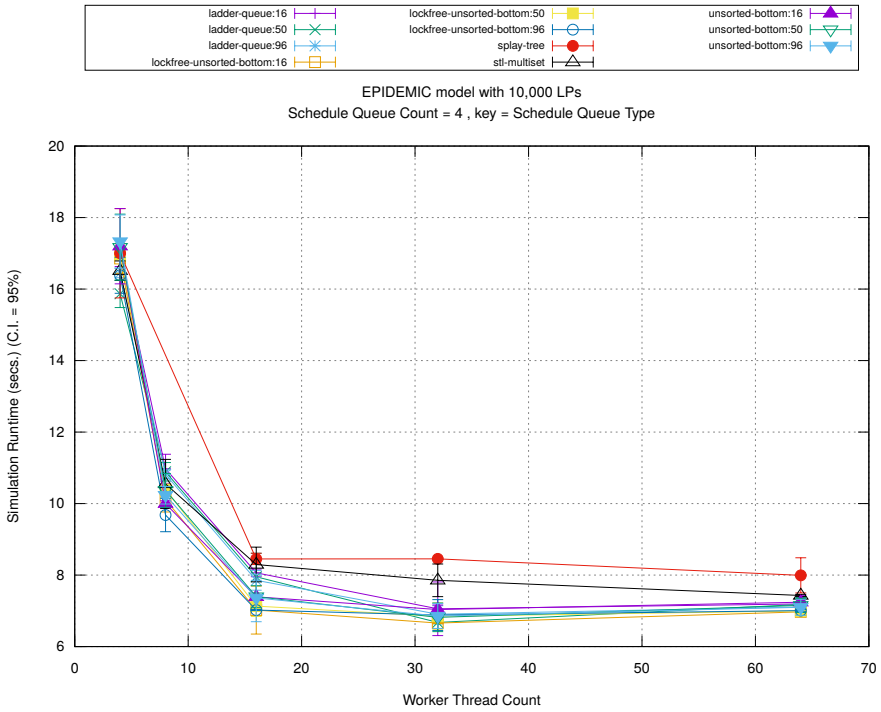


(c) Event Processing Rate (per second)

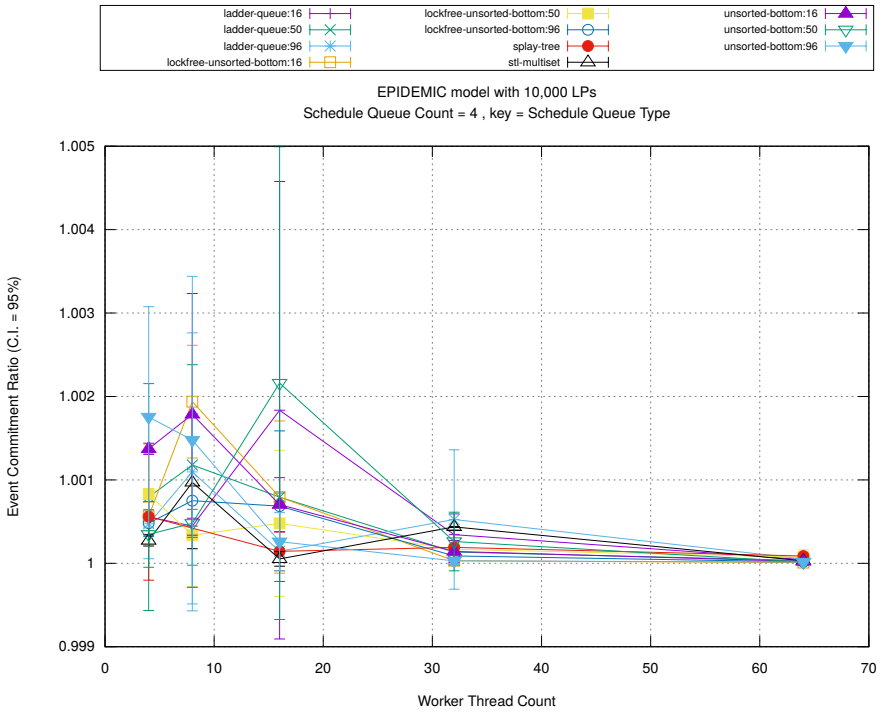
Figure A.78: epidemic 10k ws/plots/scheduleq/threads vs type key count 16



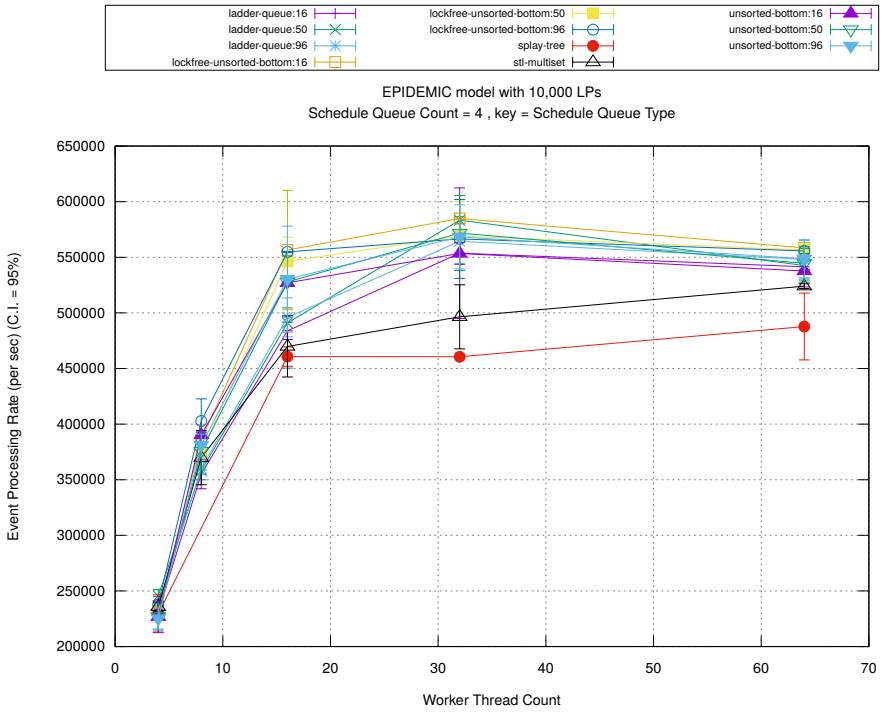
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

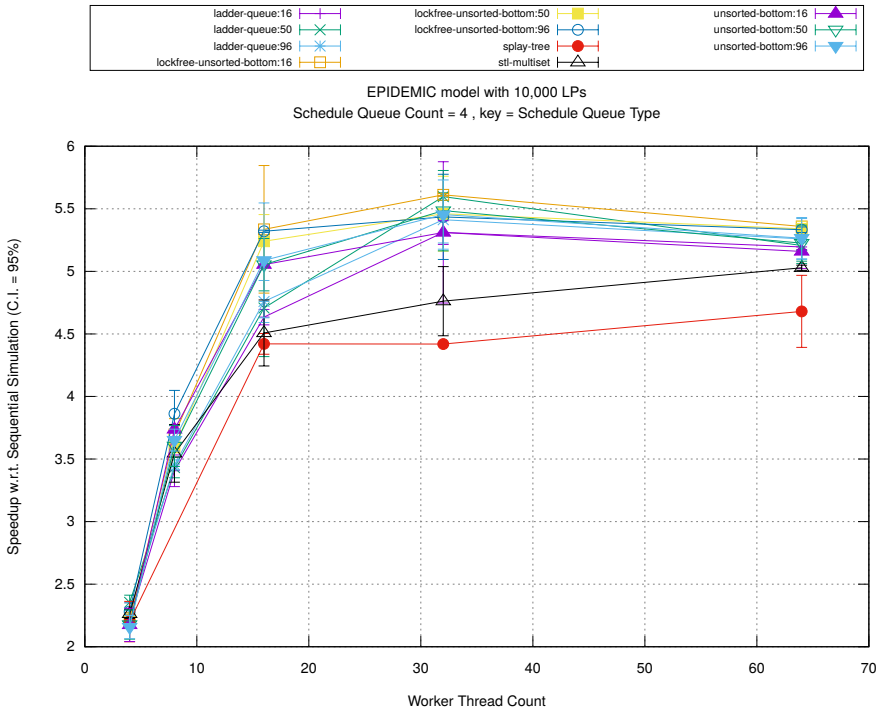


(b) Event Commitment Ratio

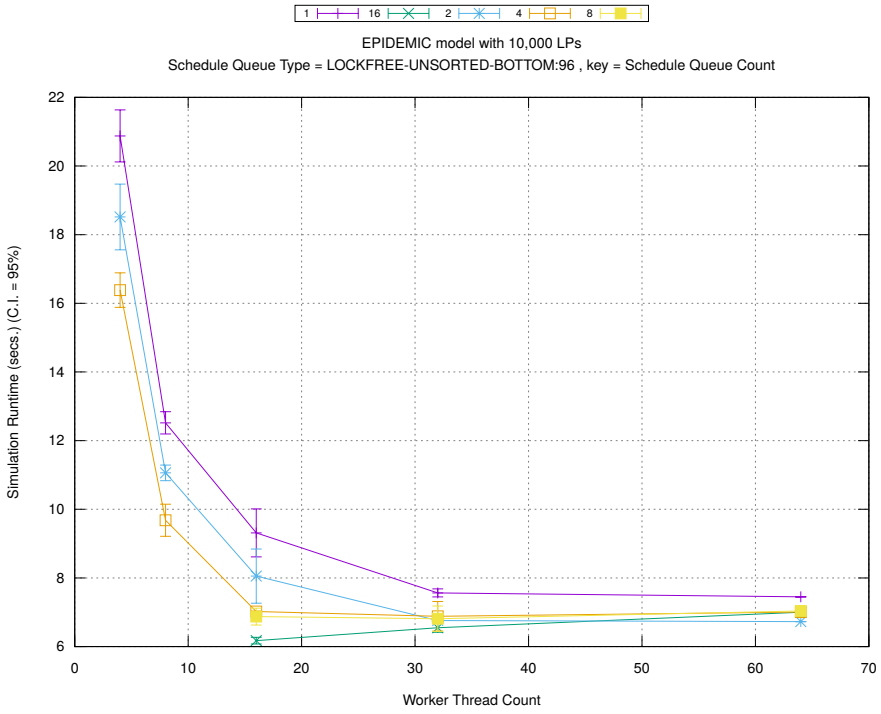


(c) Event Processing Rate (per second)

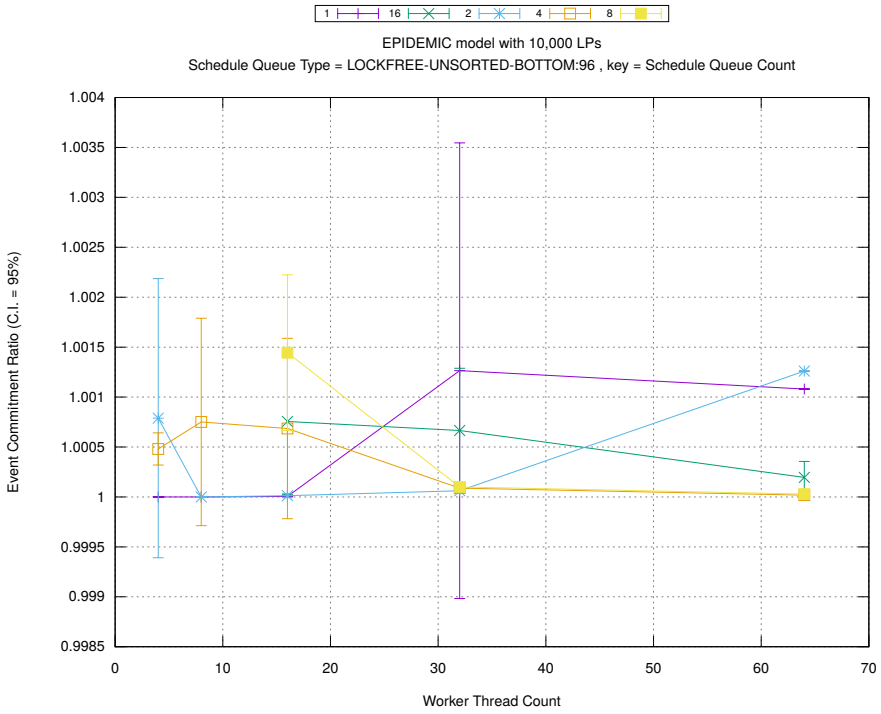
Figure A.79: epidemic 10k ws/plots/scheduleq/threads vs type key count 4



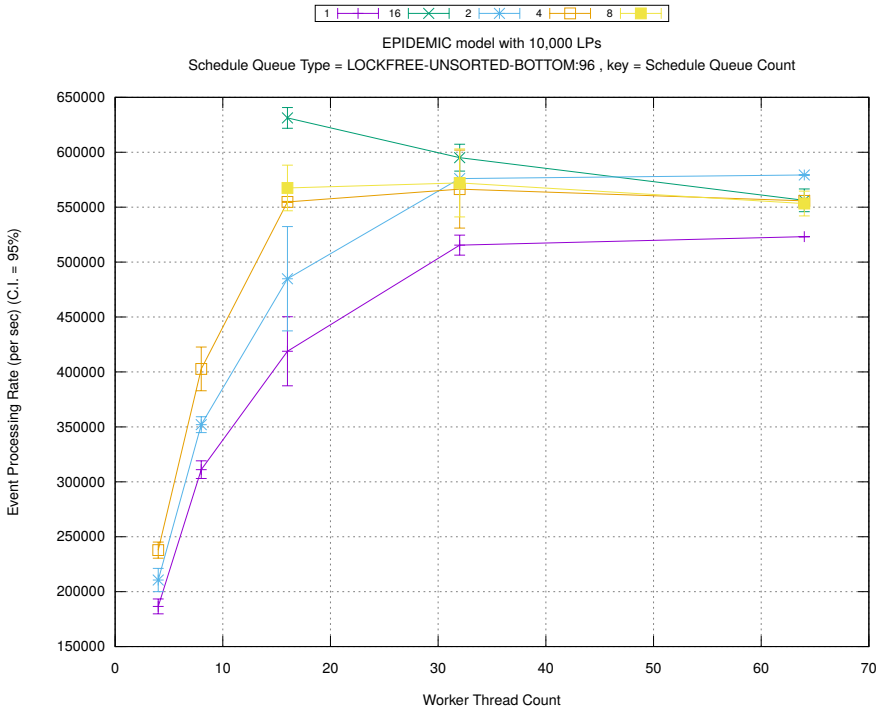
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

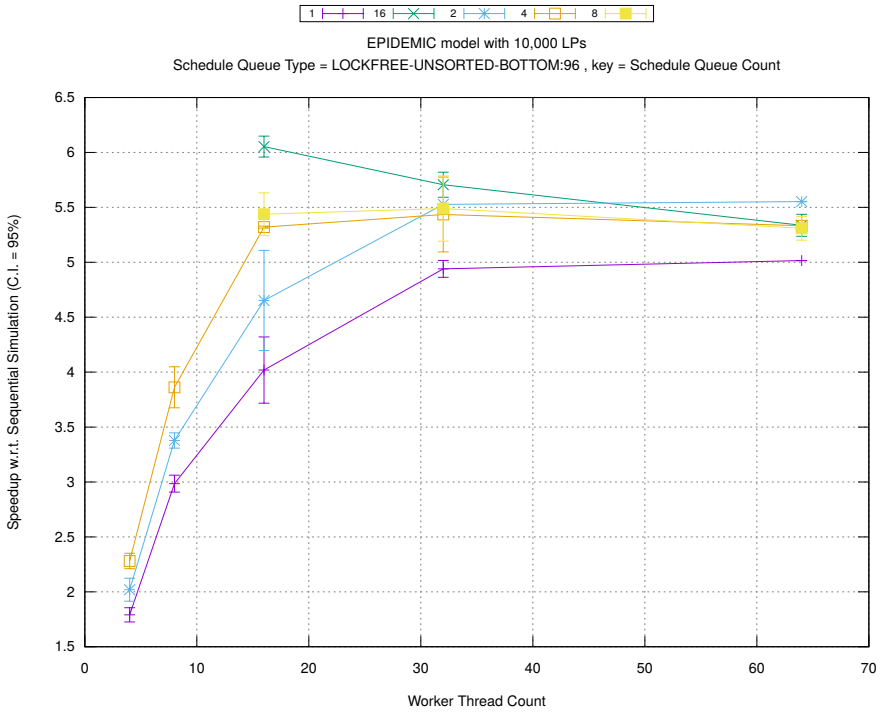


(b) Event Commitment Ratio

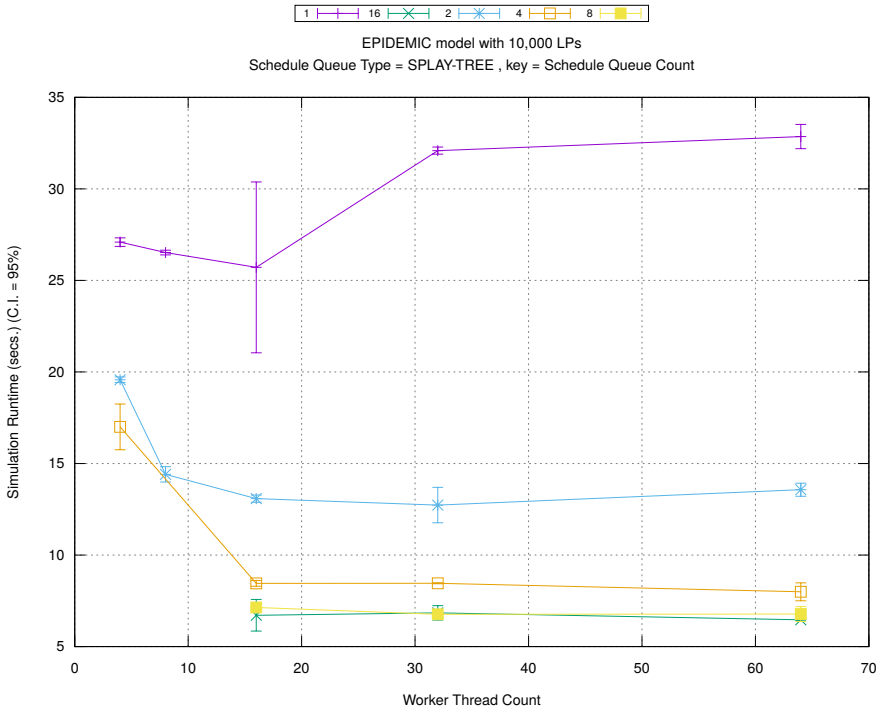


(c) Event Processing Rate (per second)

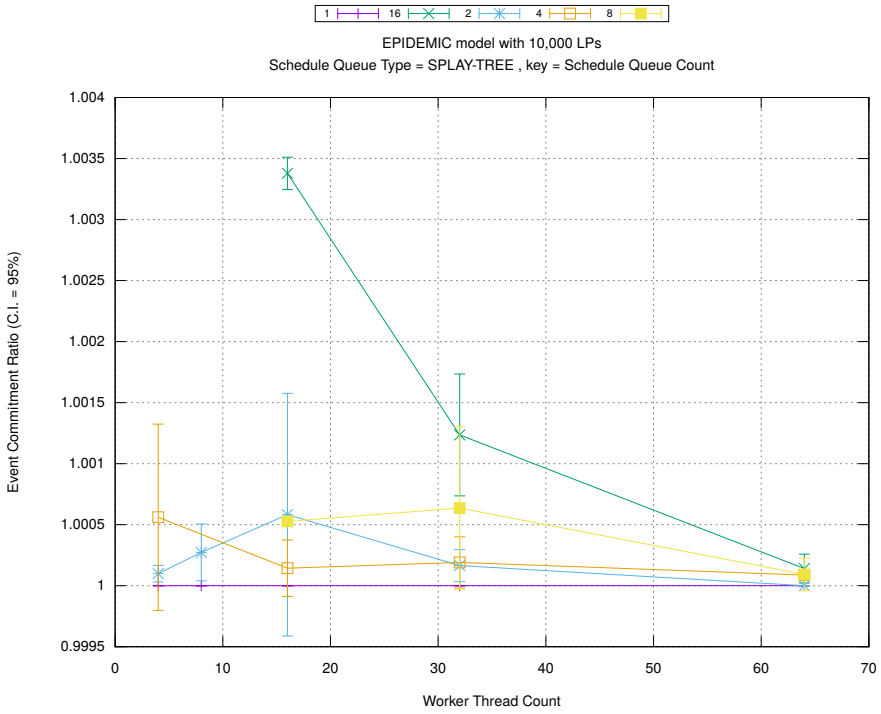
Figure A.80: epidemic 10k ws/plots/scheduleq/threads vs count key type lockfree-unsorted-bottom 96



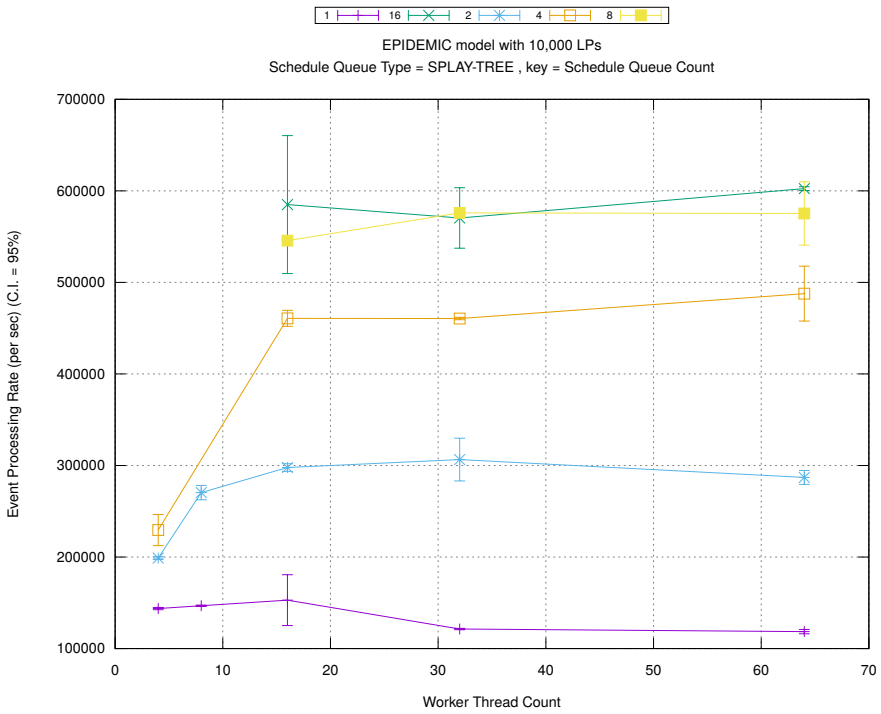
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

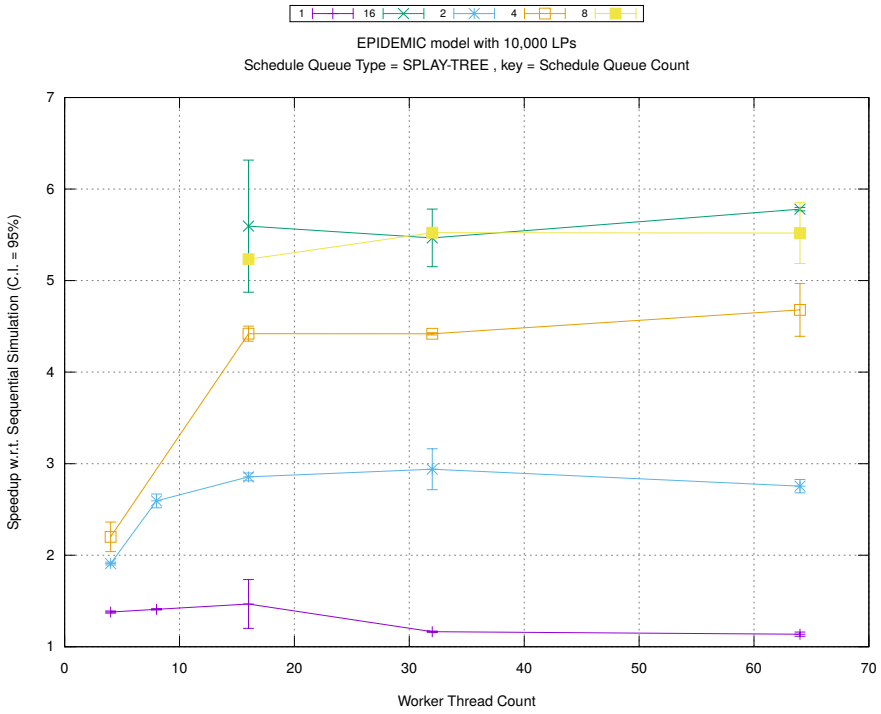


(b) Event Commitment Ratio

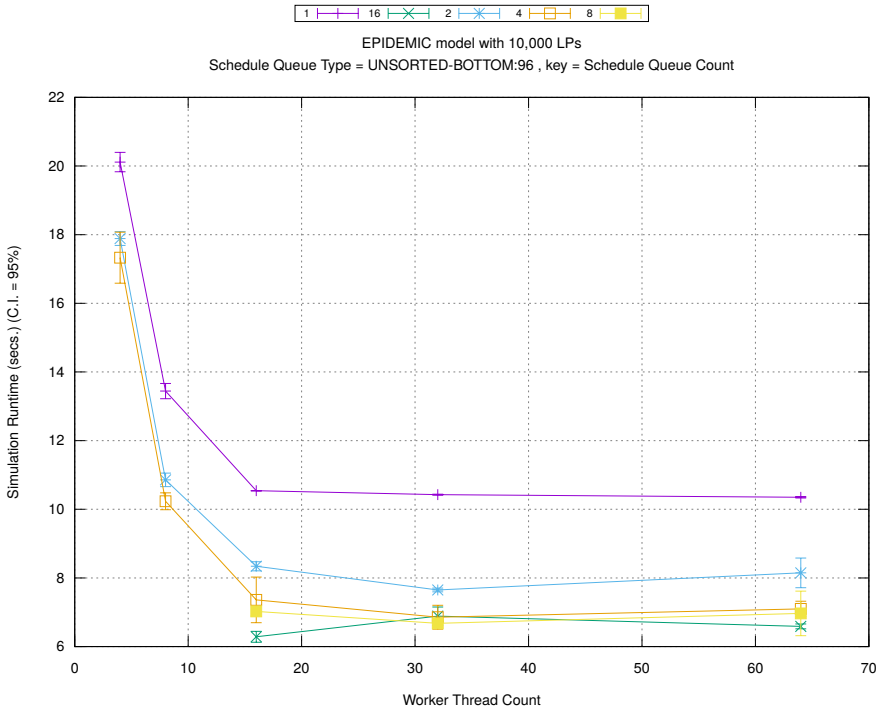


(c) Event Processing Rate (per second)

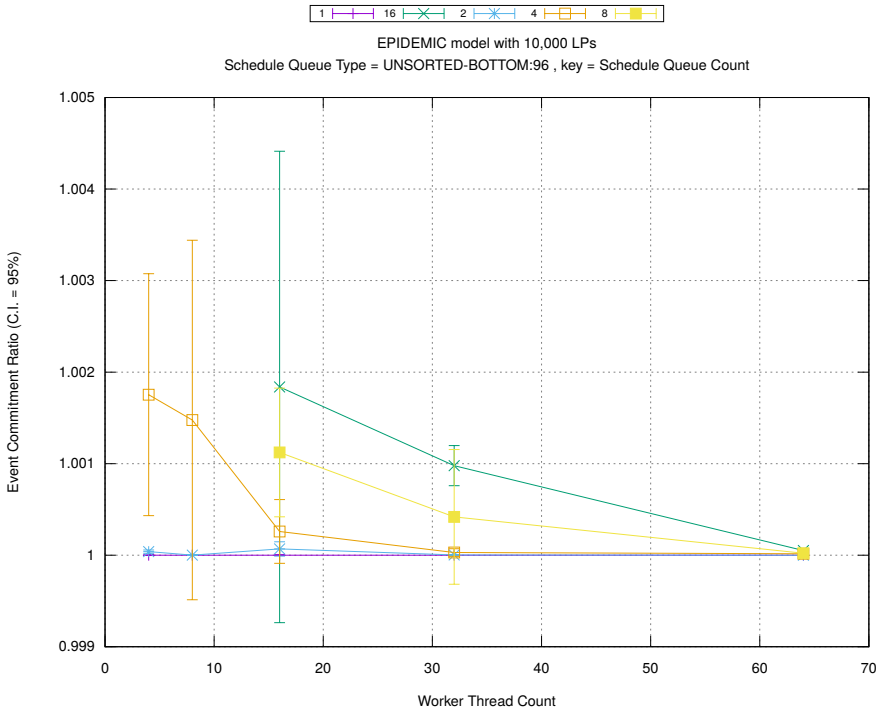
Figure A.81: epidemic 10k ws/plots/scheduleq/threads vs count key type splay-tree



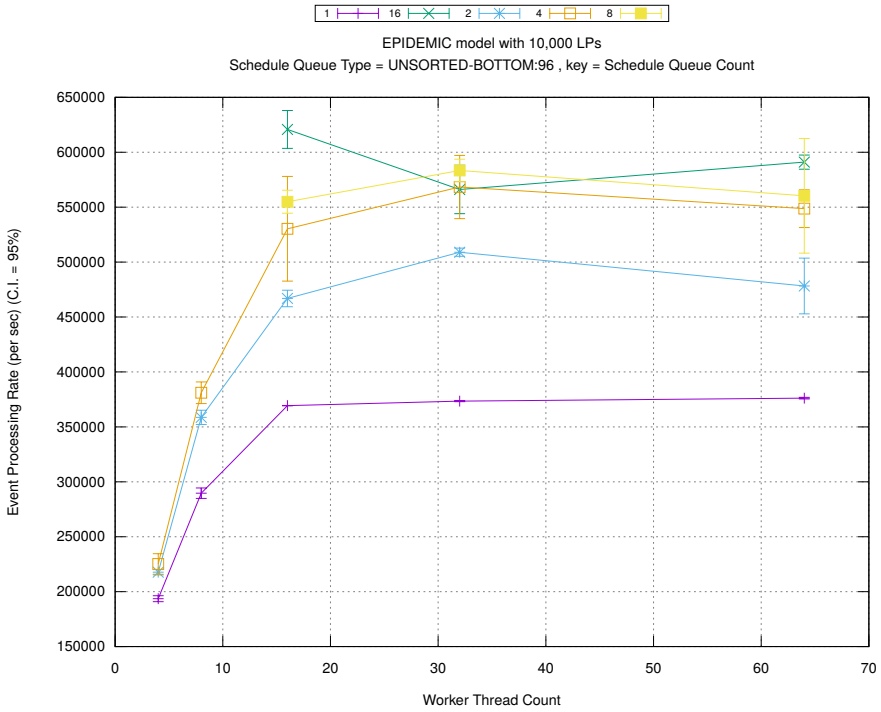
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

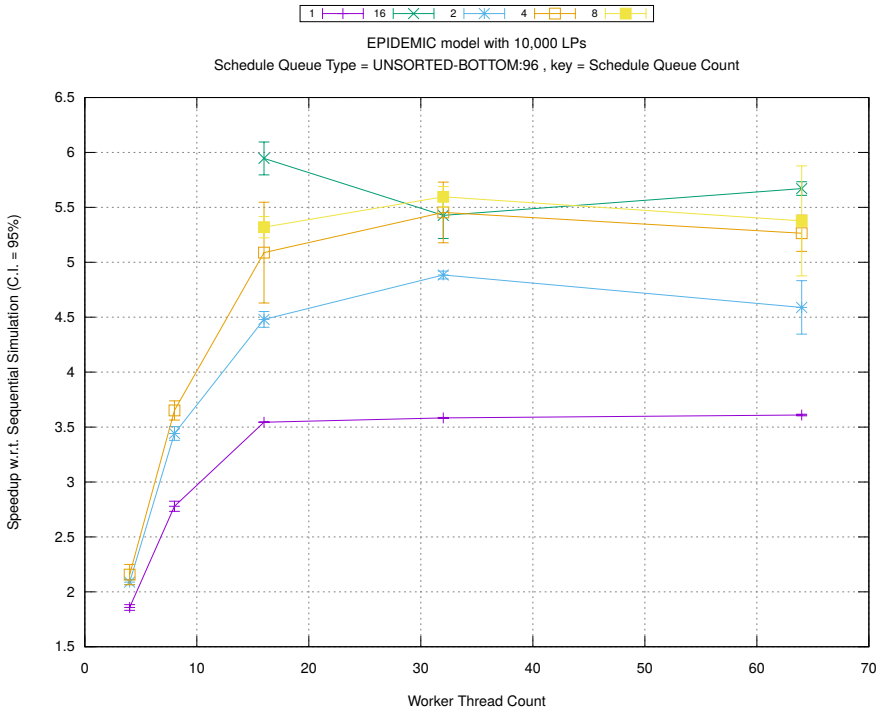


(b) Event Commitment Ratio

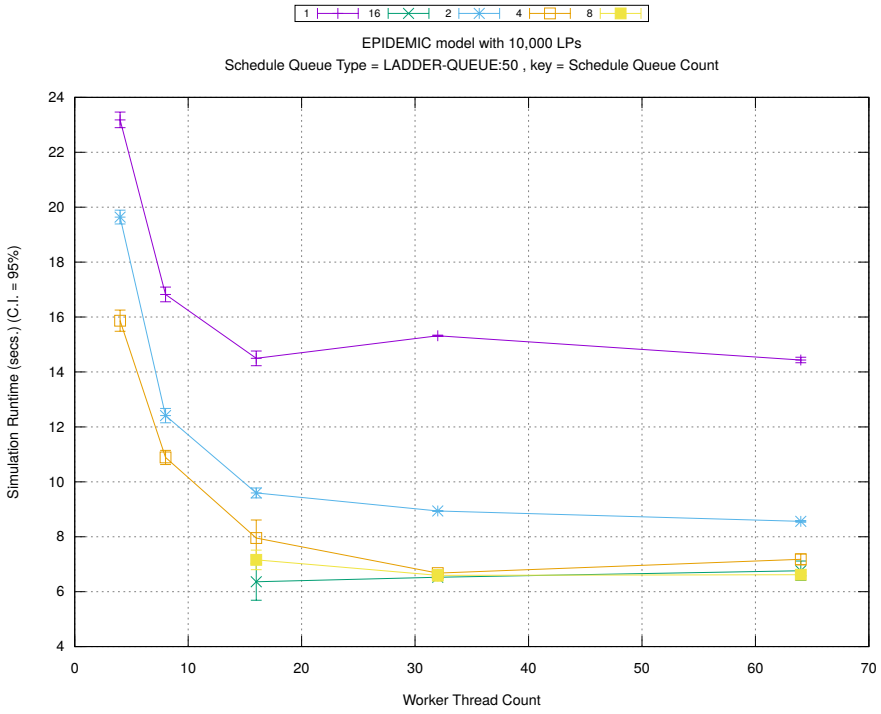


(c) Event Processing Rate (per second)

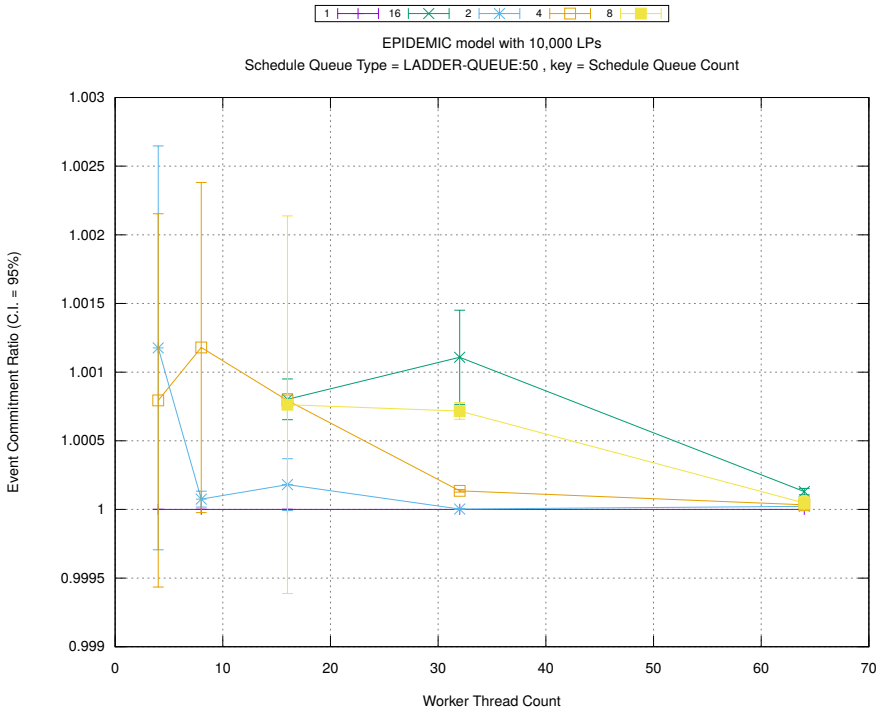
Figure A.82: epidemic 10k ws/plots/scheduleq/threads vs count key type unsorted-bottom 96



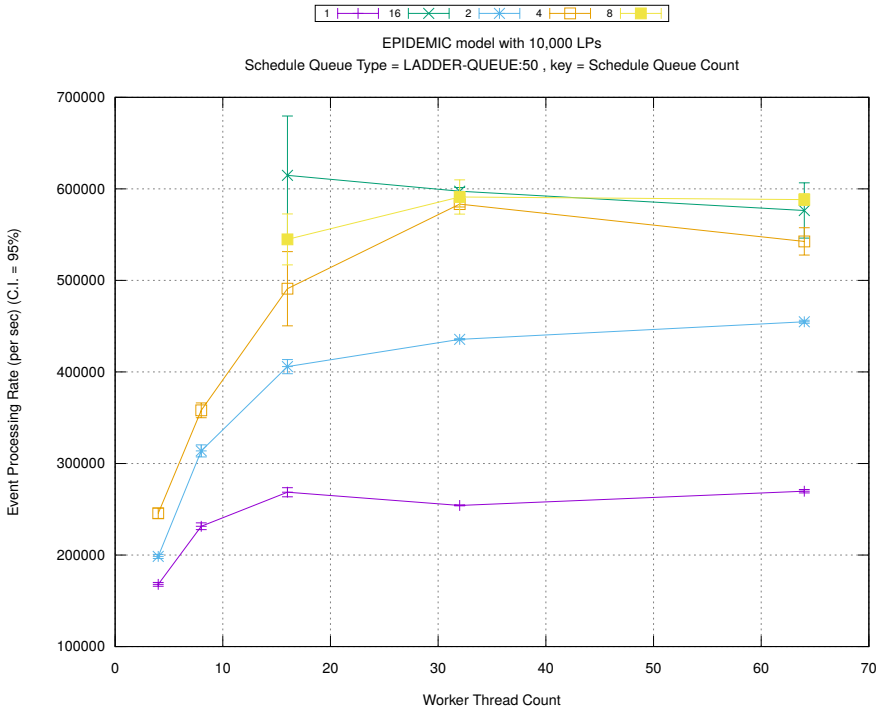
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

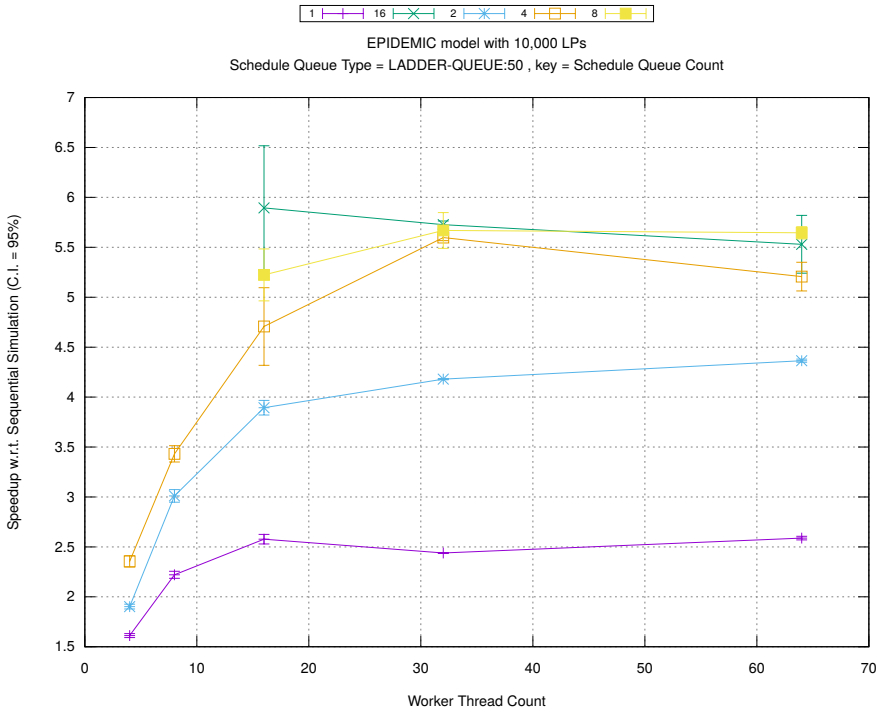


(b) Event Commitment Ratio

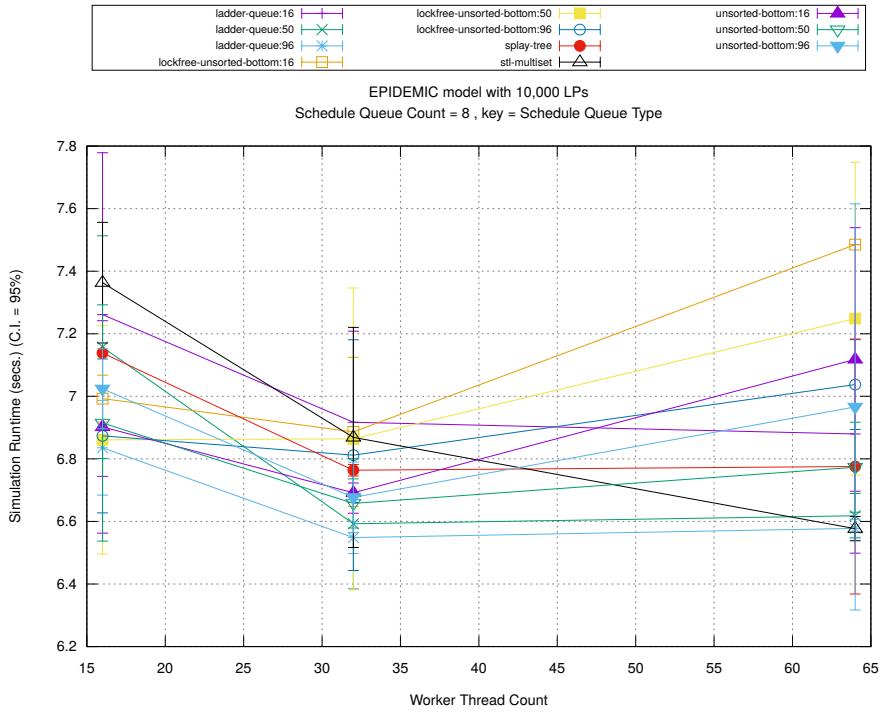


(c) Event Processing Rate (per second)

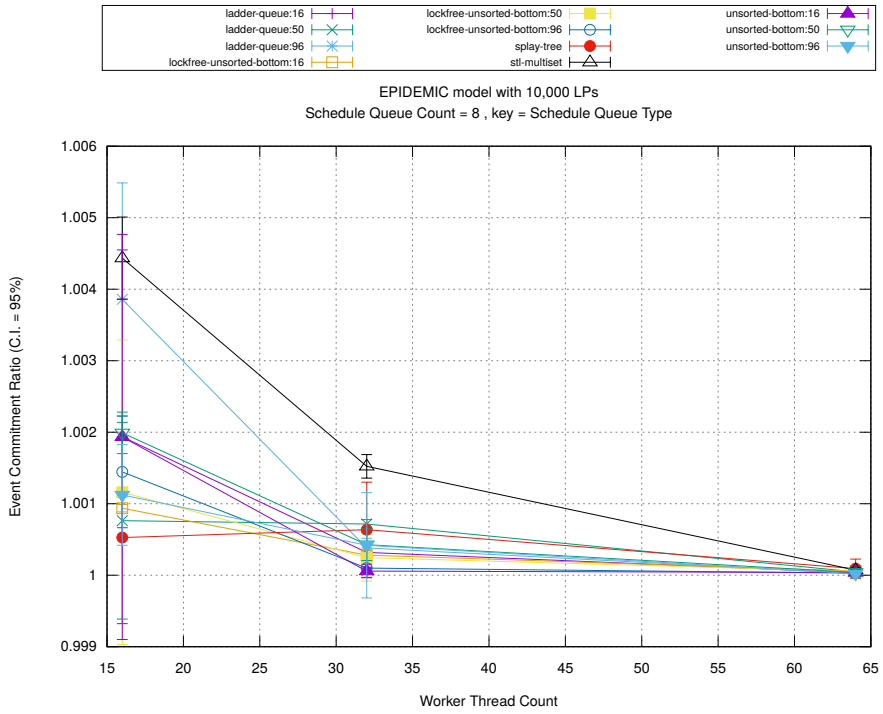
Figure A.83: epidemic 10k ws/plots/scheduleq/threads vs count key type ladder-queue 50



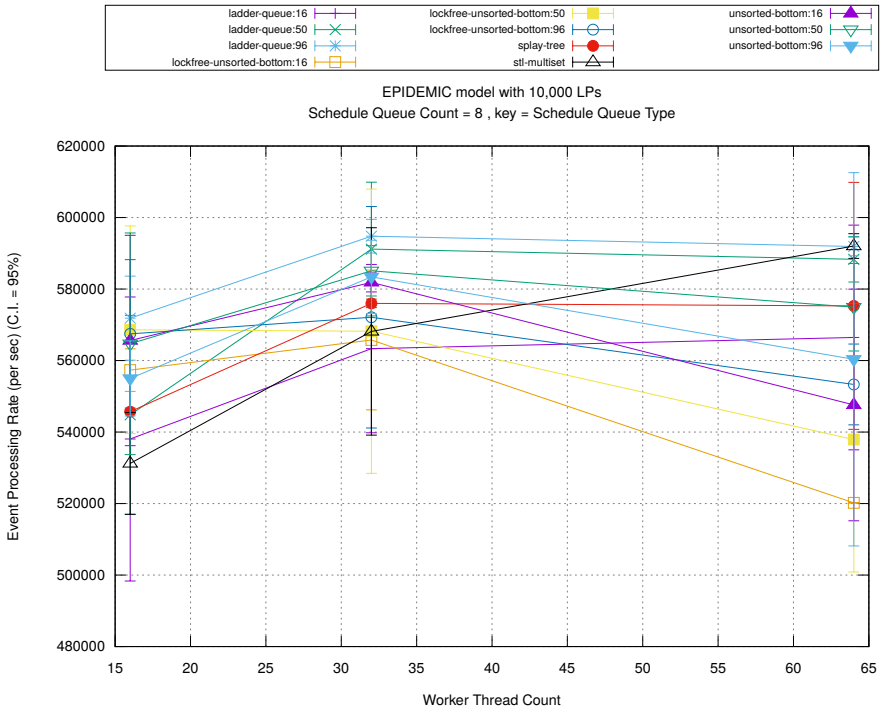
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

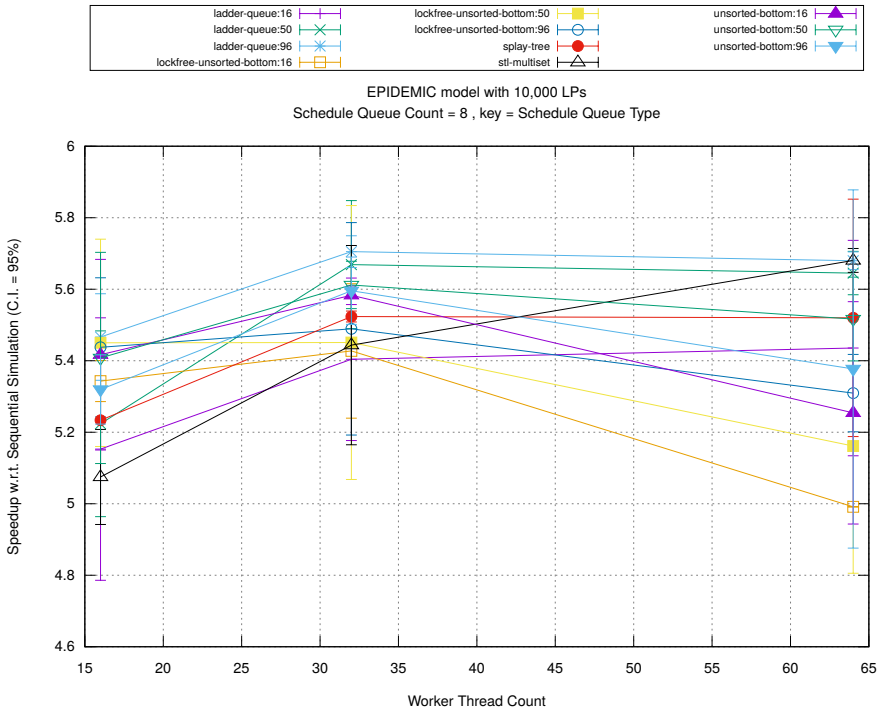


(b) Event Commitment Ratio

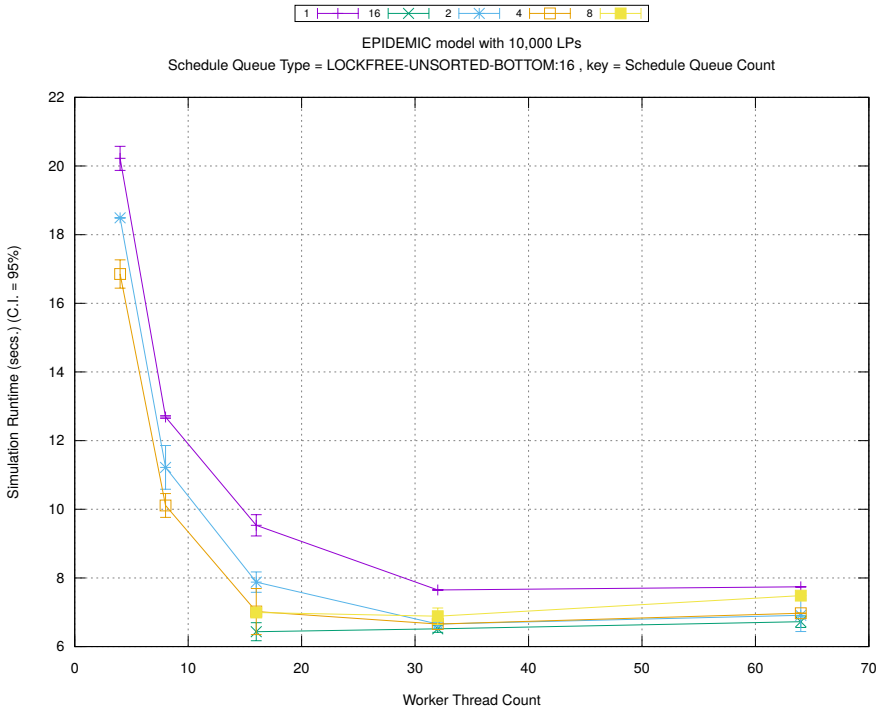


(c) Event Processing Rate (per second)

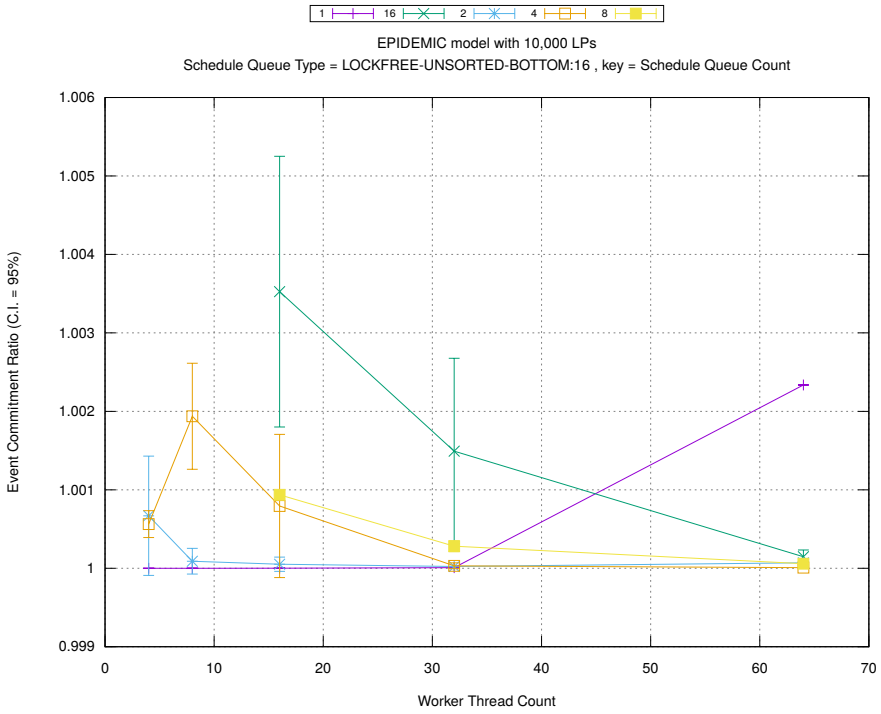
Figure A.84: epidemic 10k ws/plots/scheduleq/threads vs type key count 8



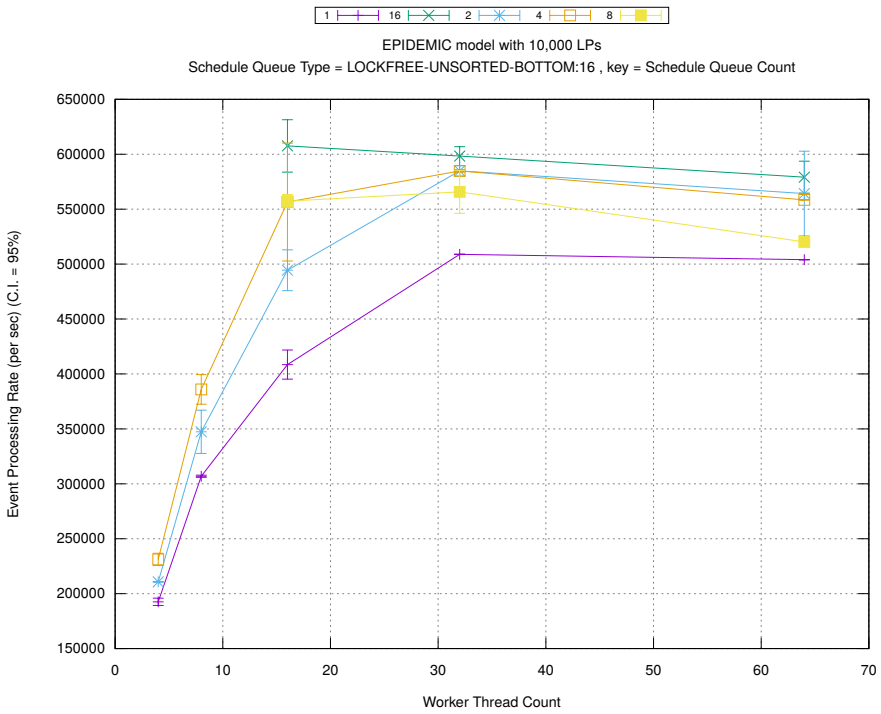
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

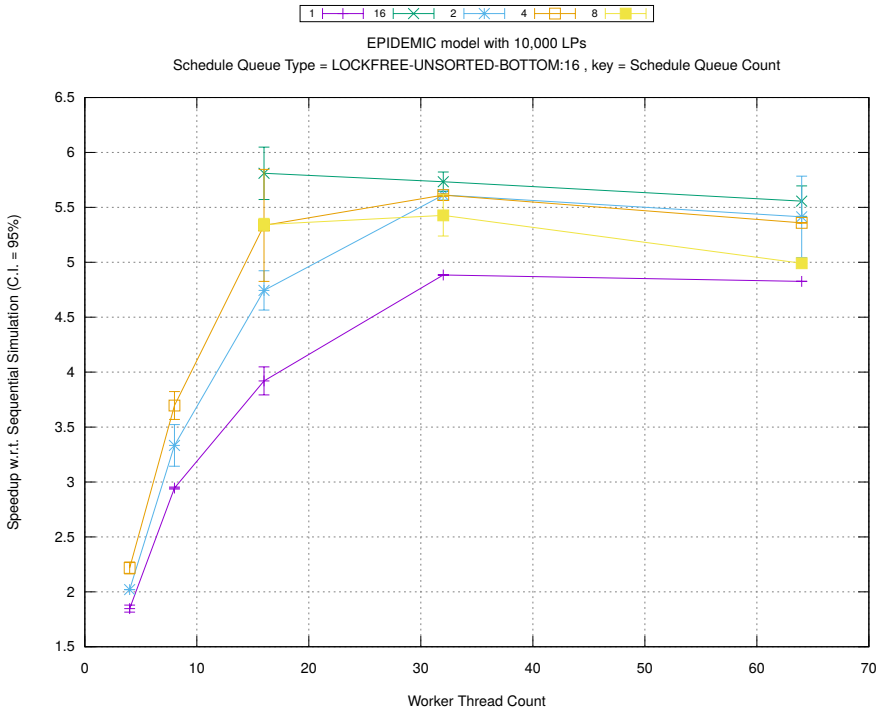


(b) Event Commitment Ratio

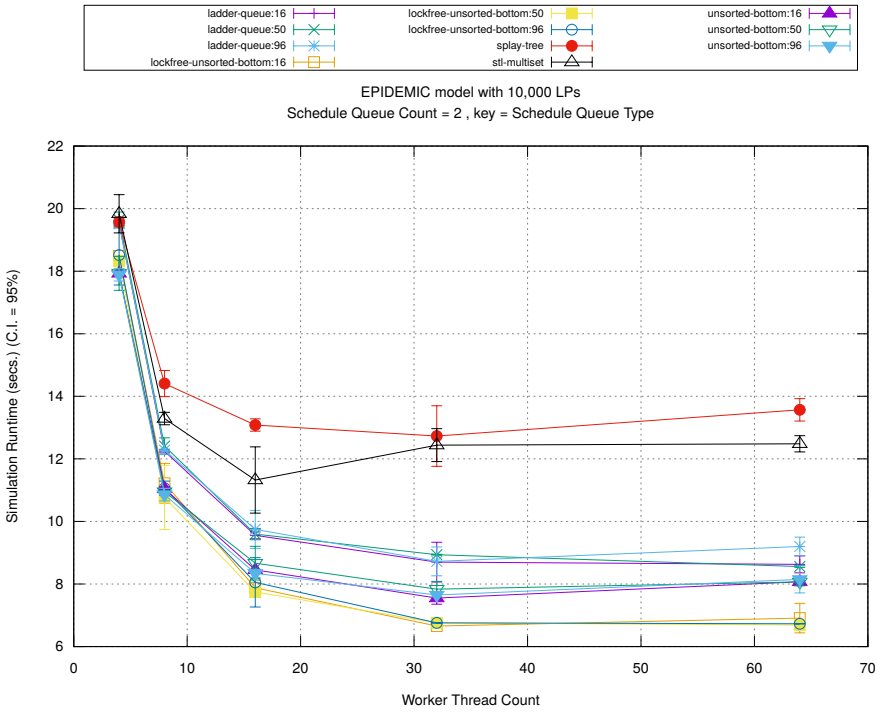


(c) Event Processing Rate (per second)

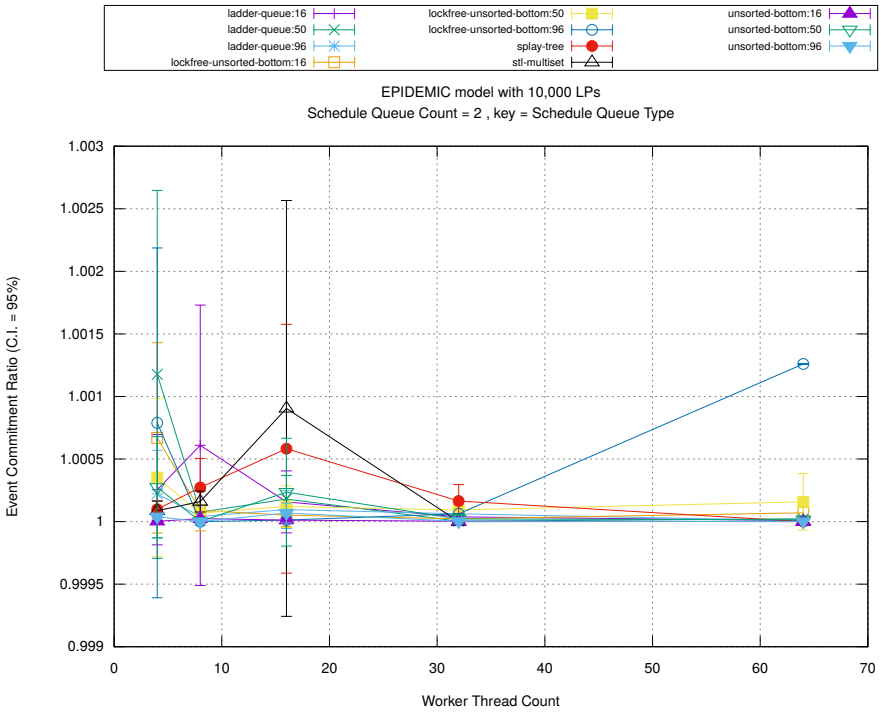
Figure A.85: epidemic 10k ws/plots/scheduleq/threads vs count key type lockfree-unsorted-bottom 16



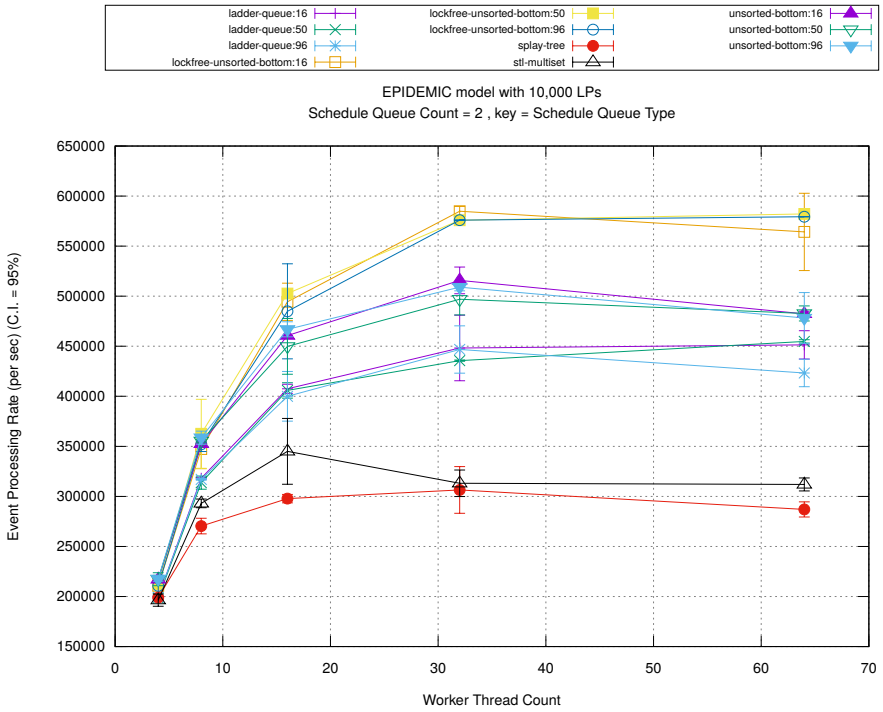
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

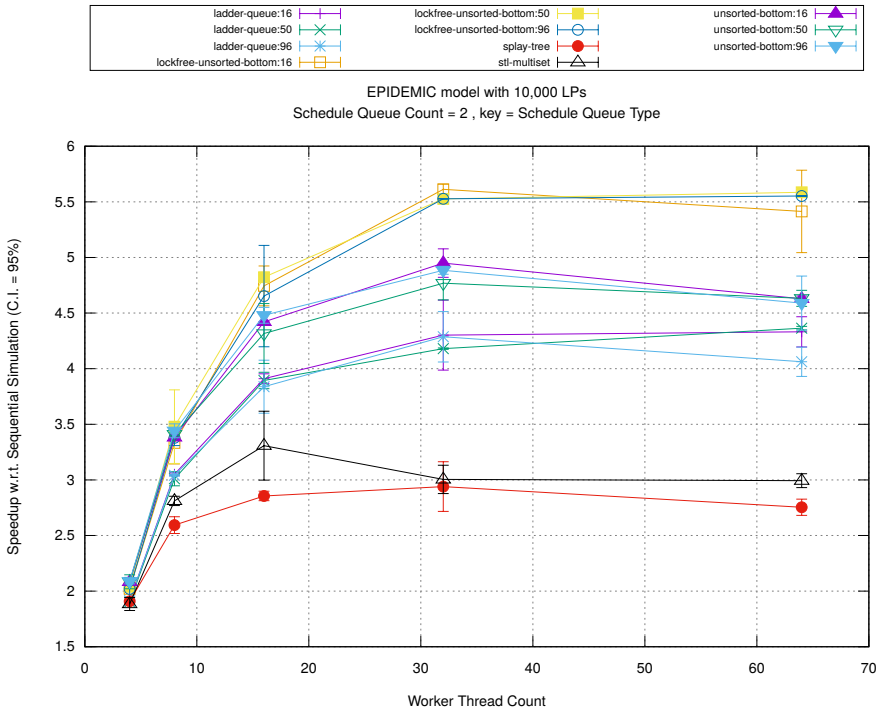


(b) Event Commitment Ratio

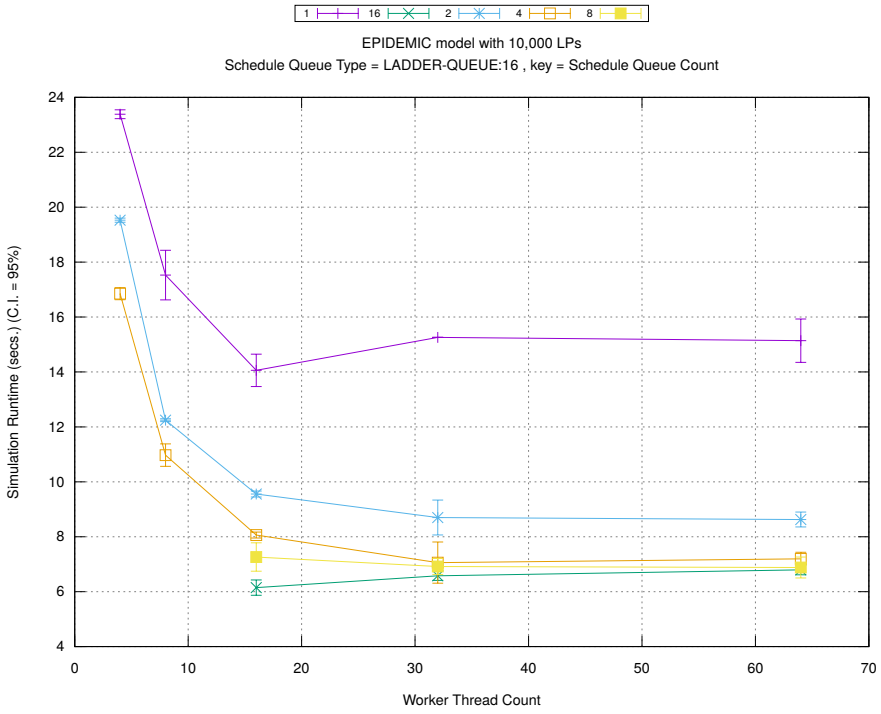


(c) Event Processing Rate (per second)

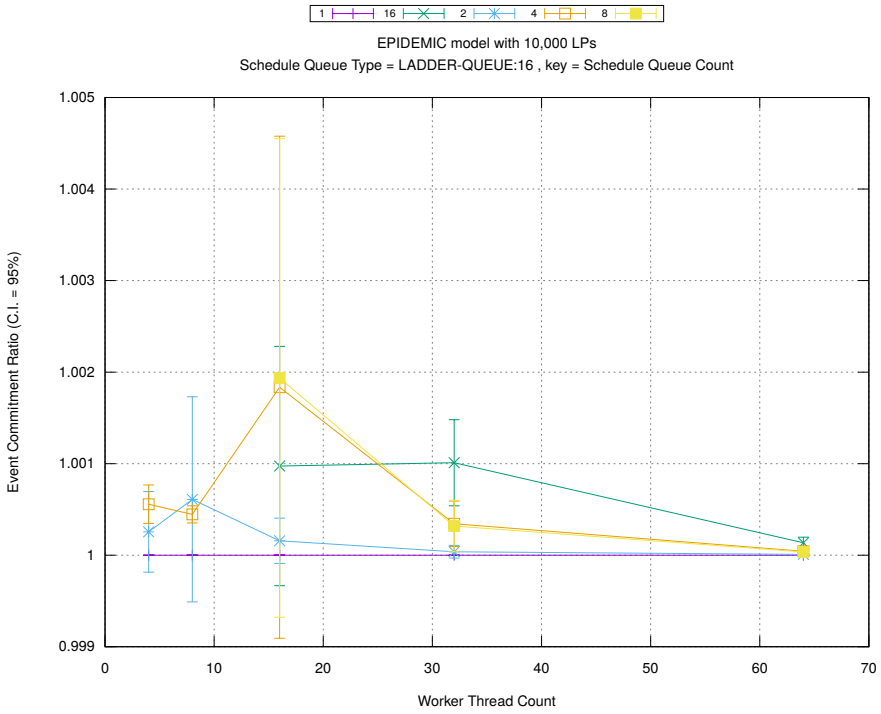
Figure A.86: epidemic 10k ws/plots/scheduleq/threads vs type key count 2



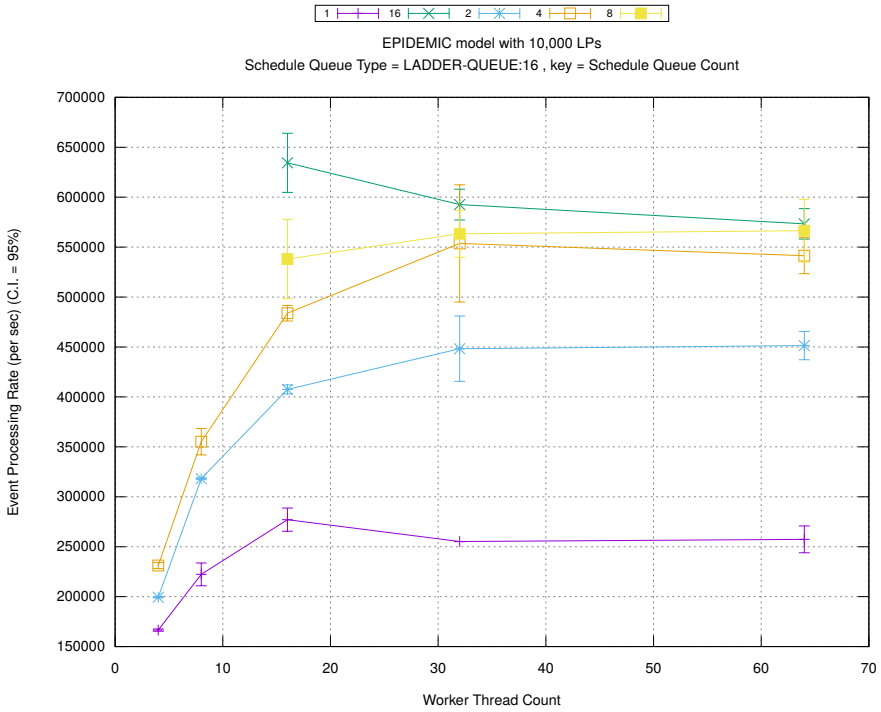
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

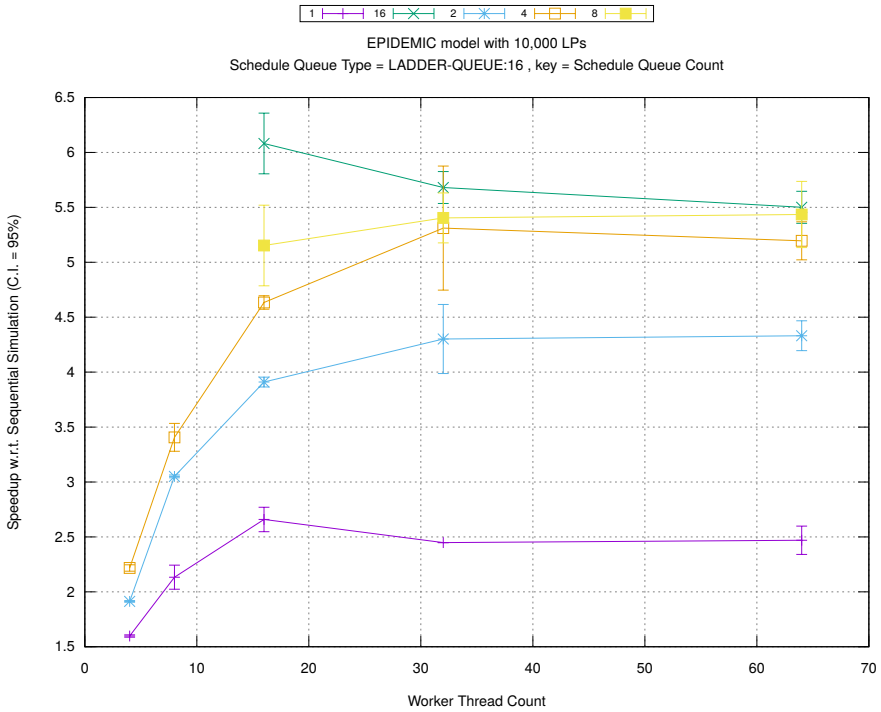


(b) Event Commitment Ratio

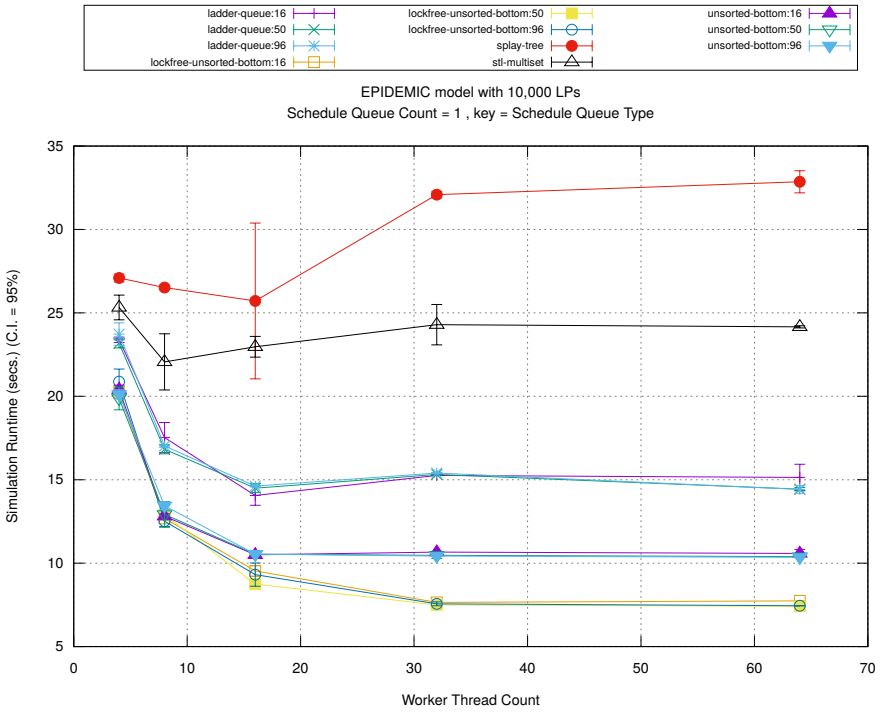


(c) Event Processing Rate (per second)

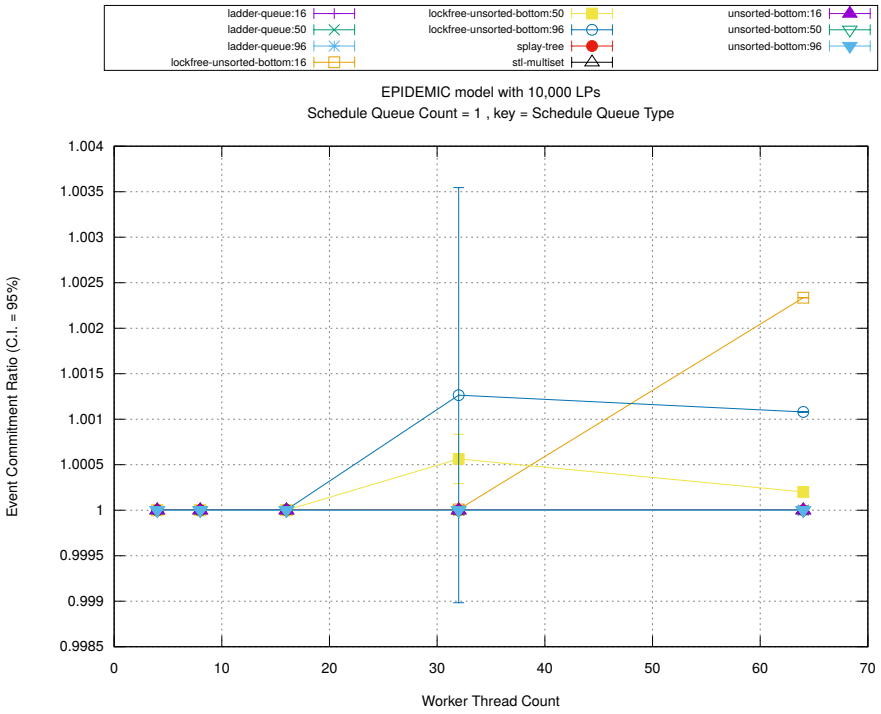
Figure A.87: epidemic 10k ws/plots/scheduleq/threads vs count key type ladder-queue 16



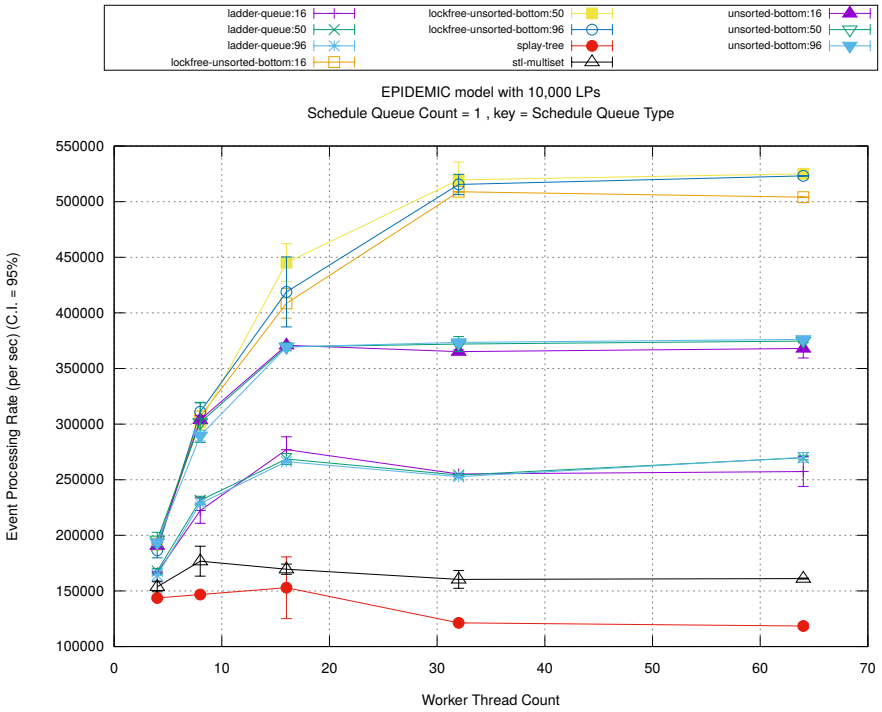
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

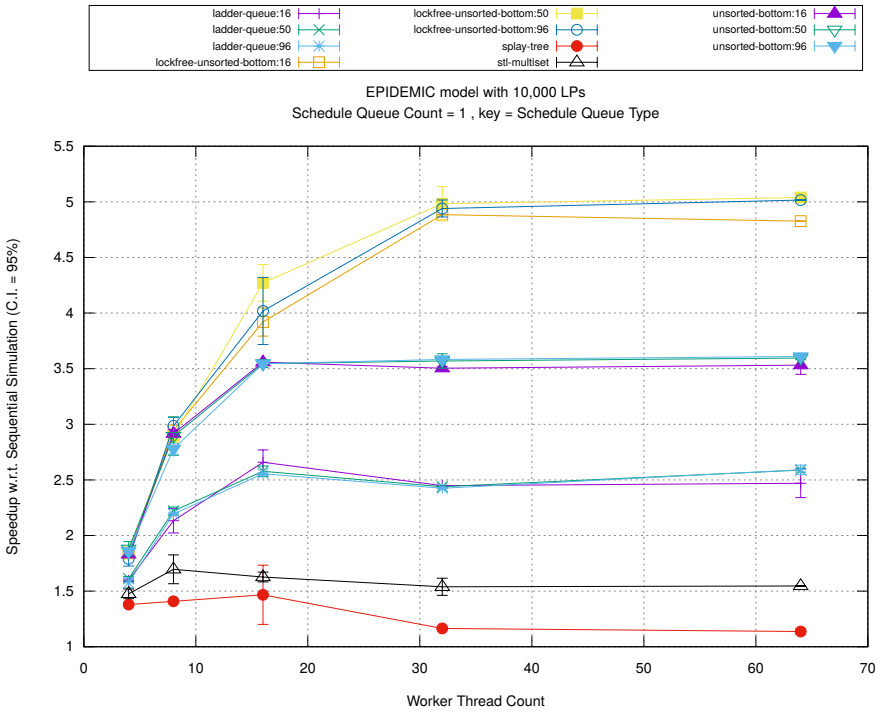


(b) Event Commitment Ratio

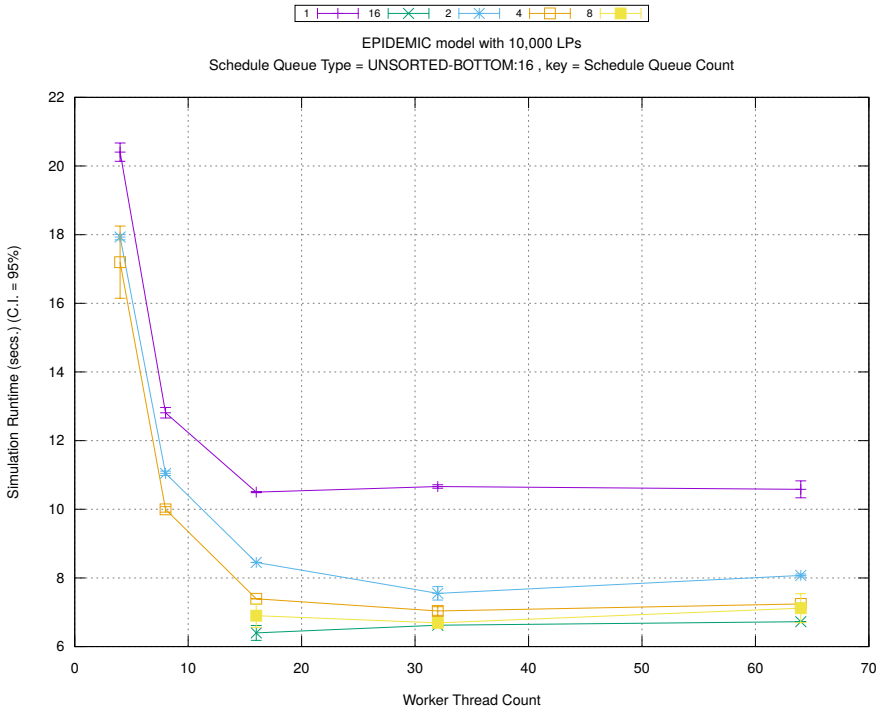


(c) Event Processing Rate (per second)

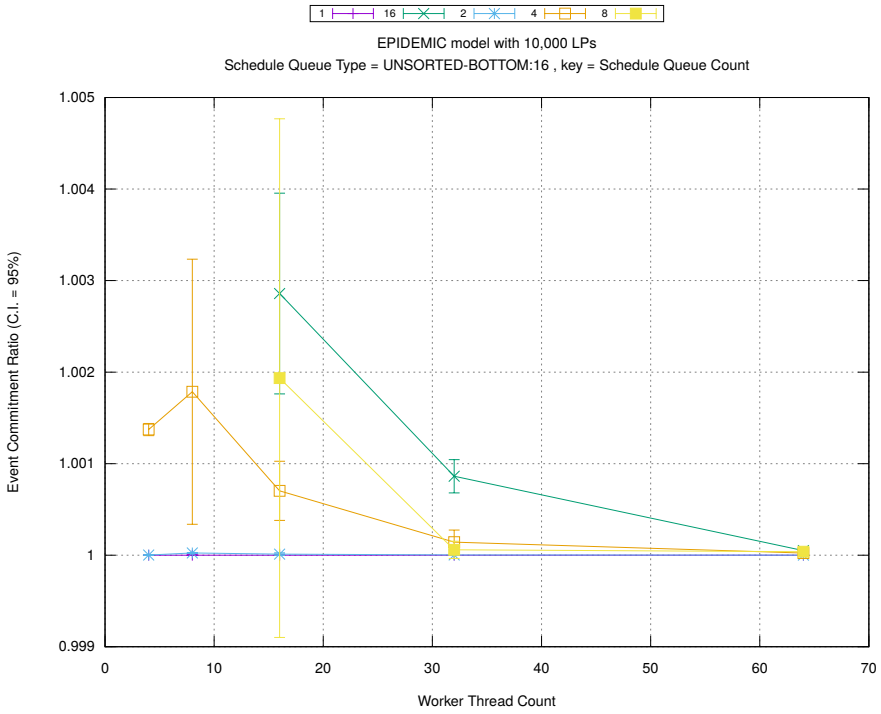
Figure A.88: epidemic 10k ws/plots/scheduleq/threads vs type key count 1



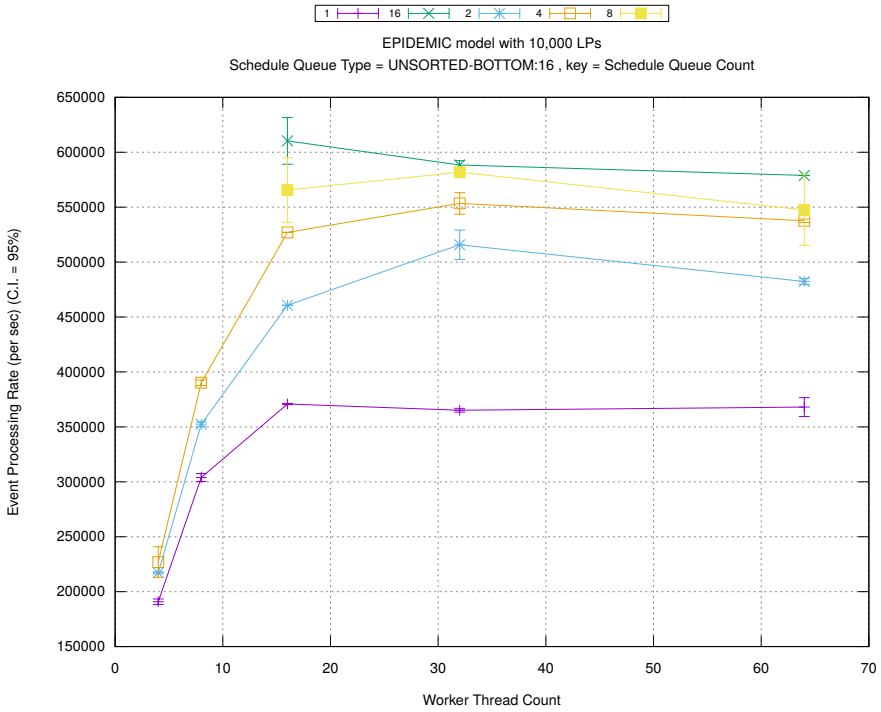
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

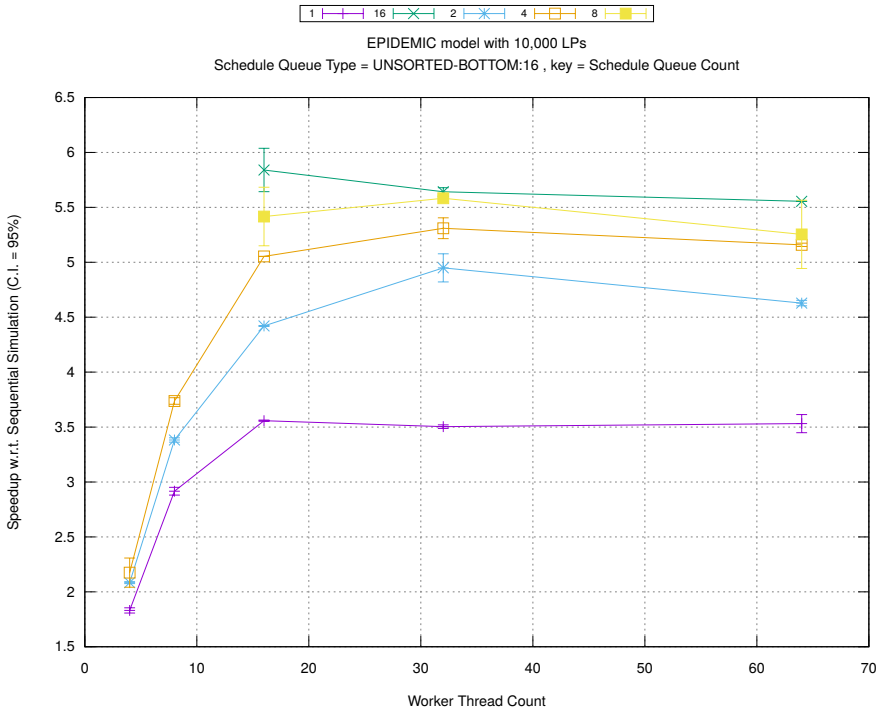


(b) Event Commitment Ratio

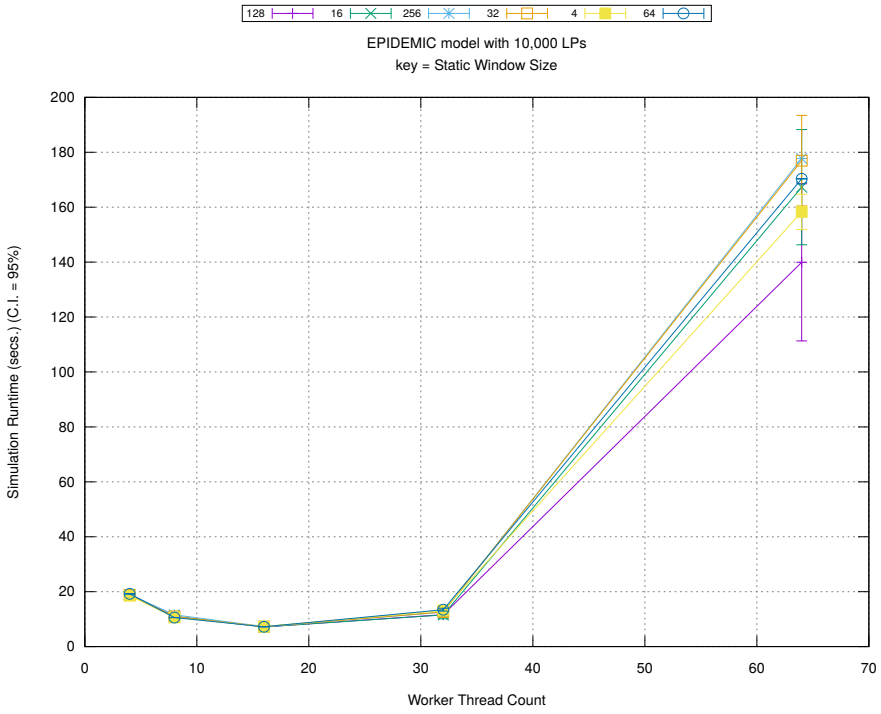


(c) Event Processing Rate (per second)

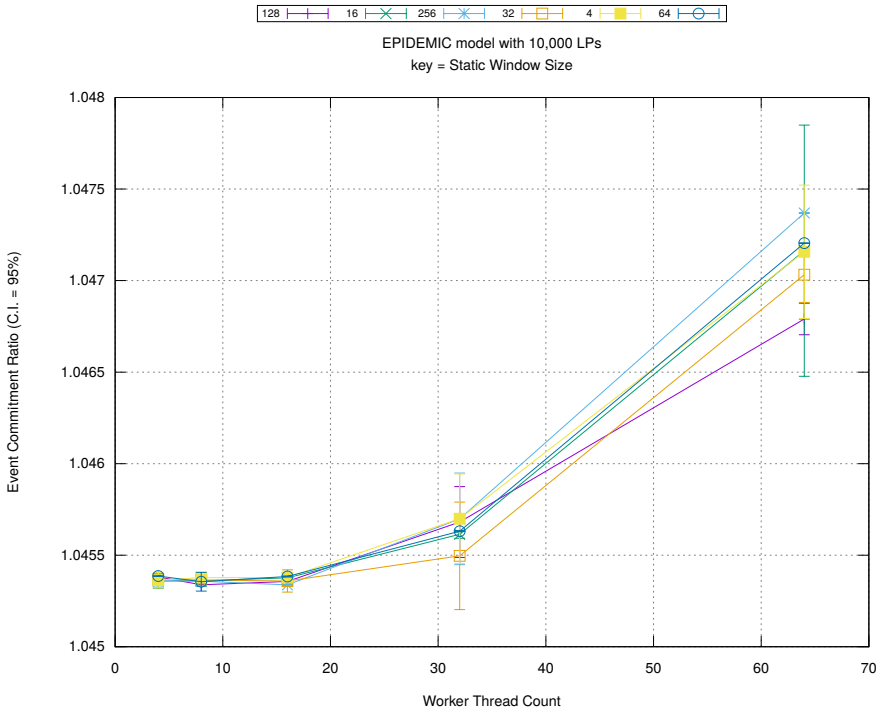
Figure A.89: epidemic 10k ws/plots/scheduleq/threads vs count key type unsorted-bottom 16



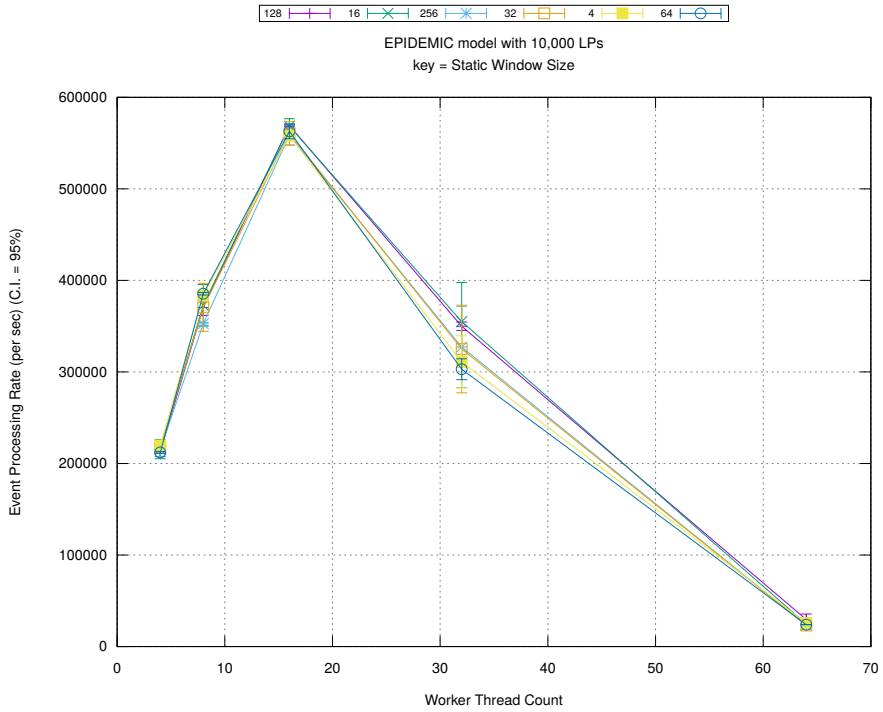
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

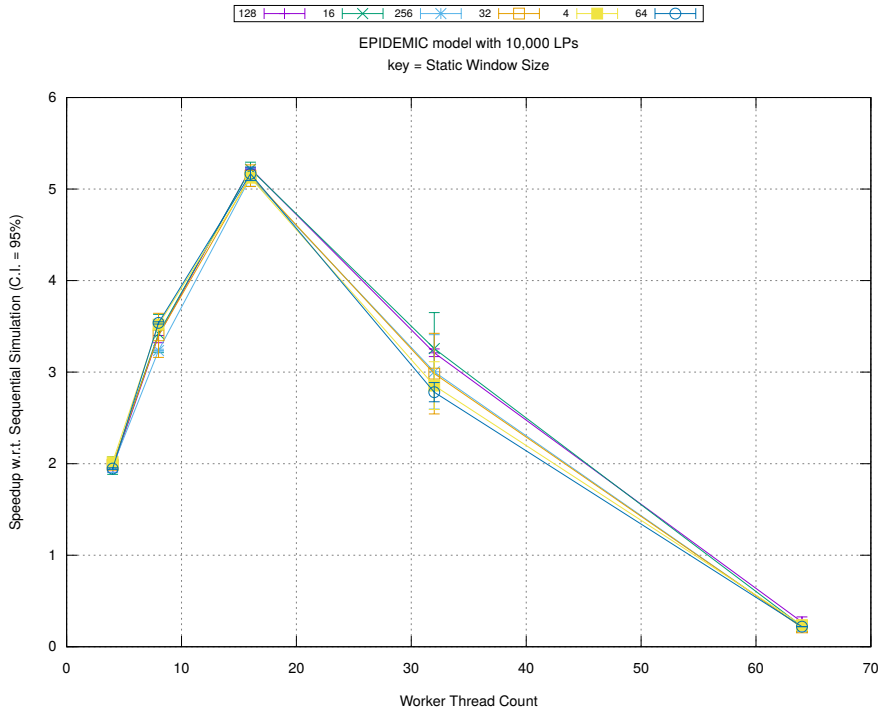


(b) Event Commitment Ratio

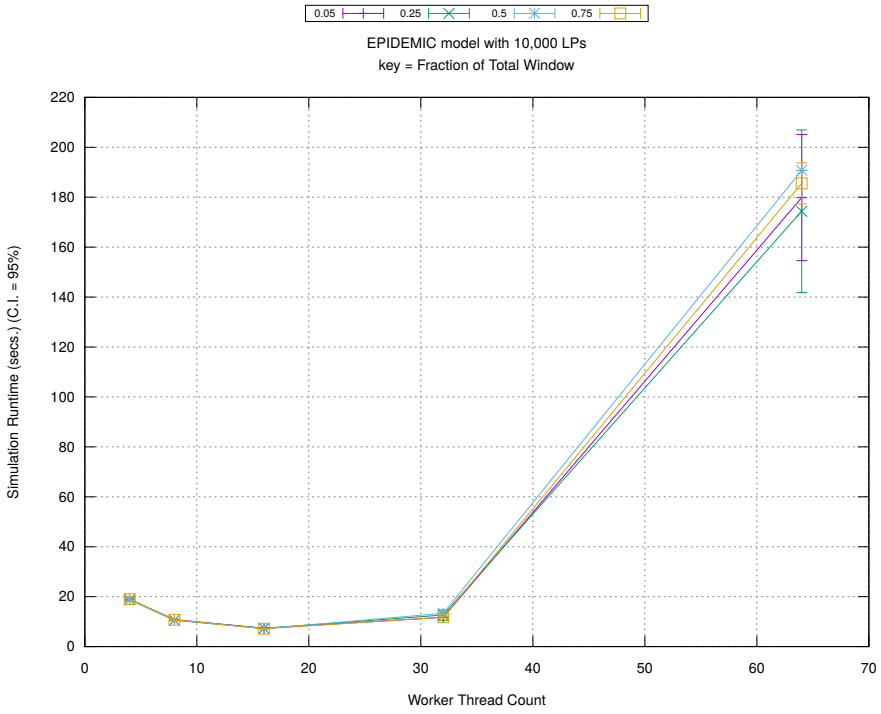


(c) Event Processing Rate (per second)

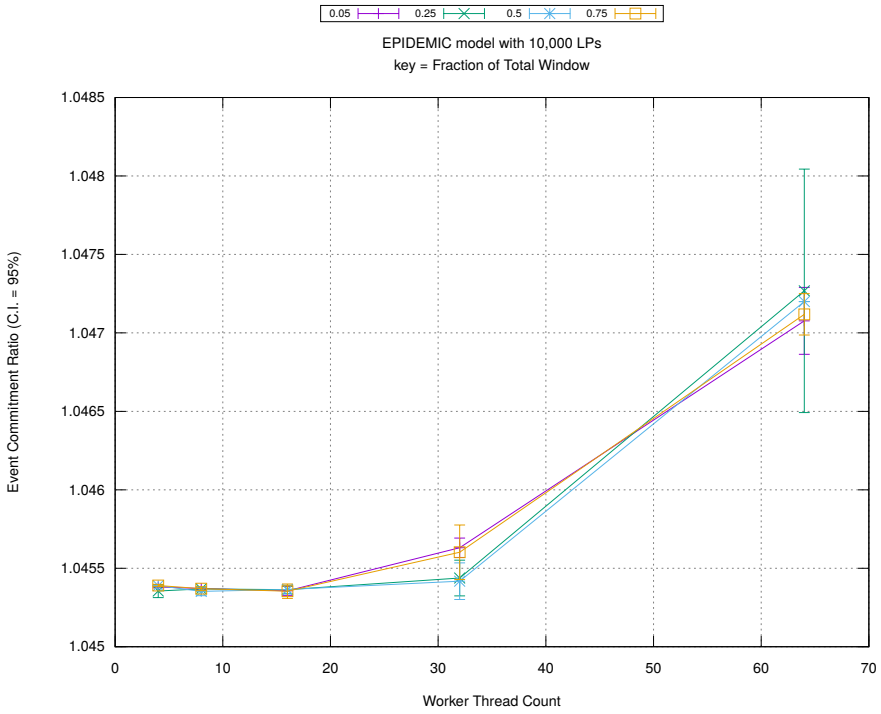
Figure A.90: epidemic 10k ws/plots/bags/threads vs staticwindow key fractionwindow 1.0



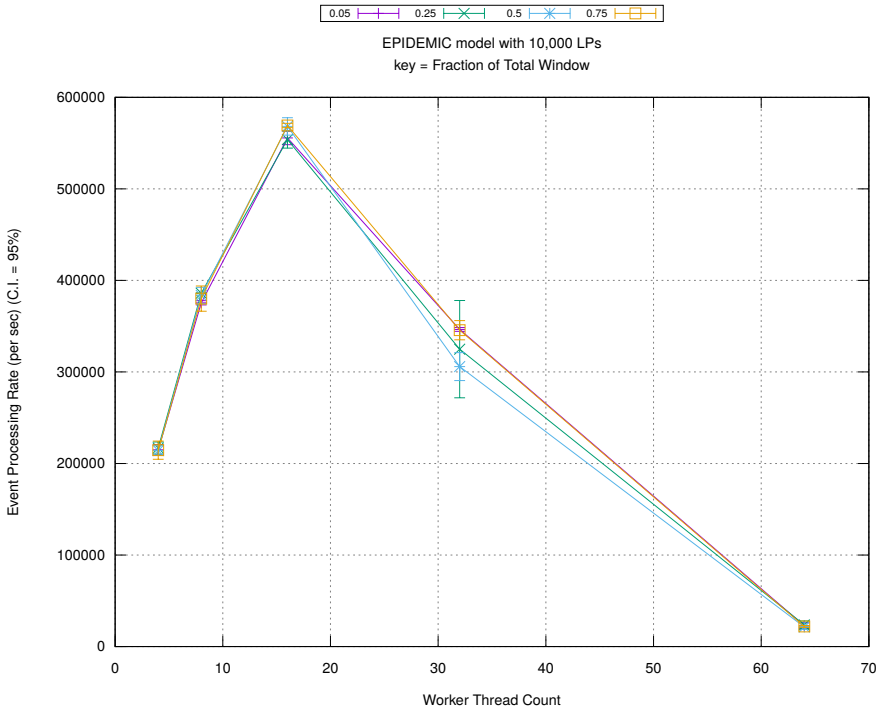
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

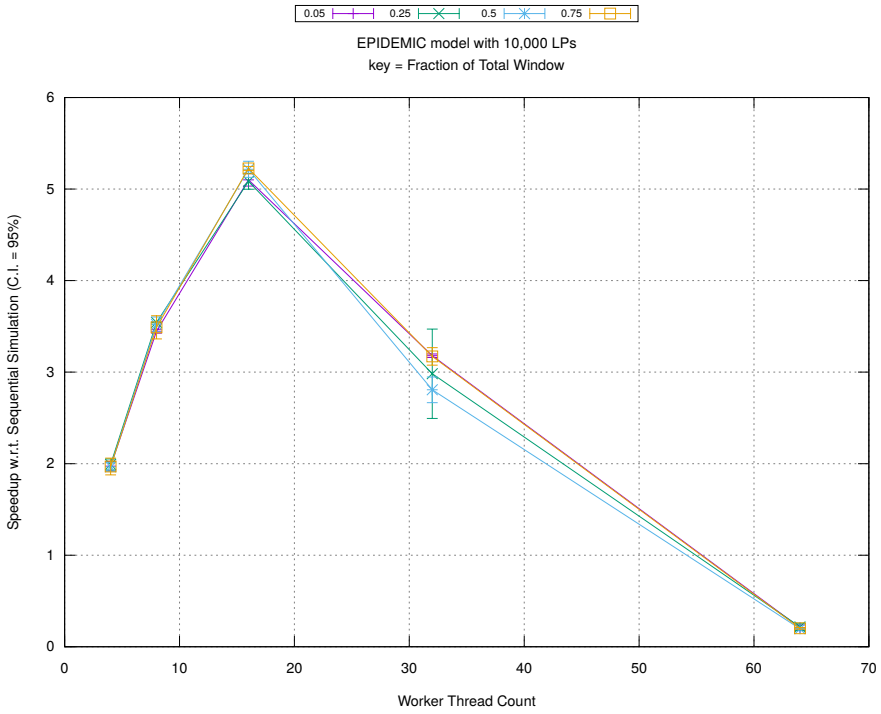


(b) Event Commitment Ratio

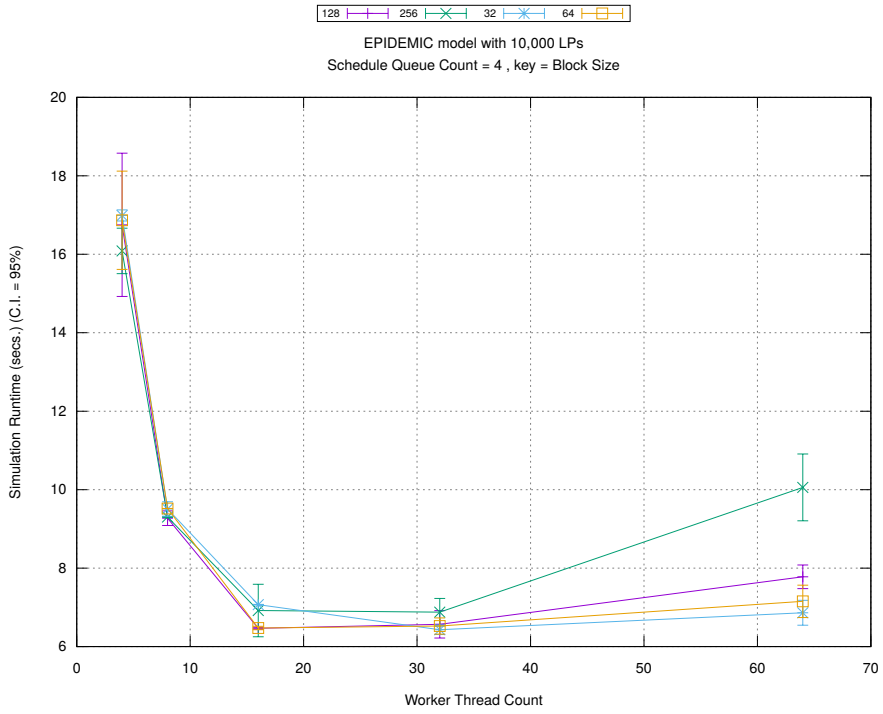


(c) Event Processing Rate (per second)

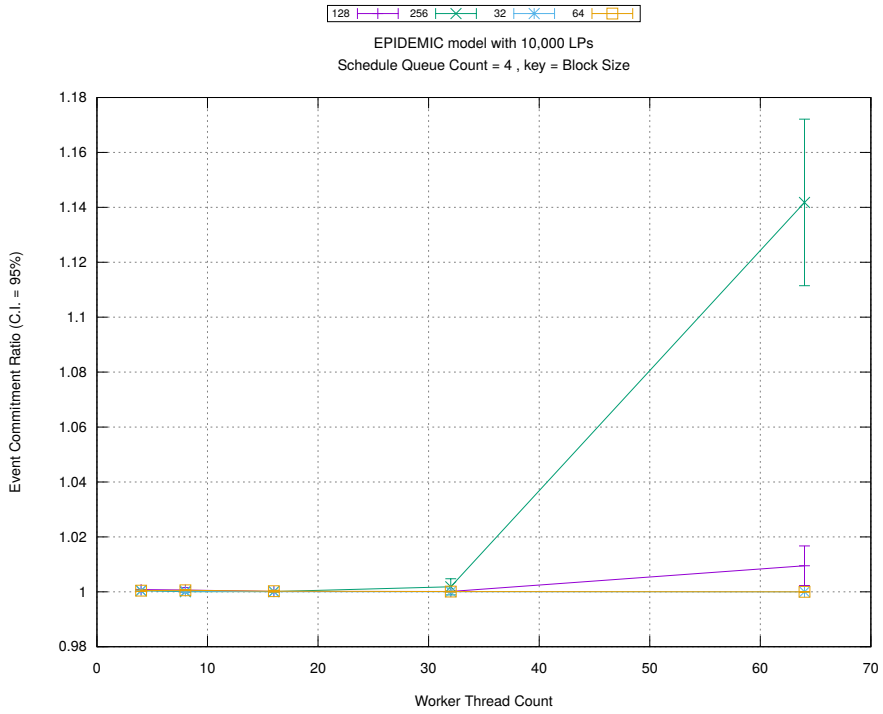
Figure A.91: epidemic 10k ws/plots/bags/threads vs fractionwindow key staticwindow 0



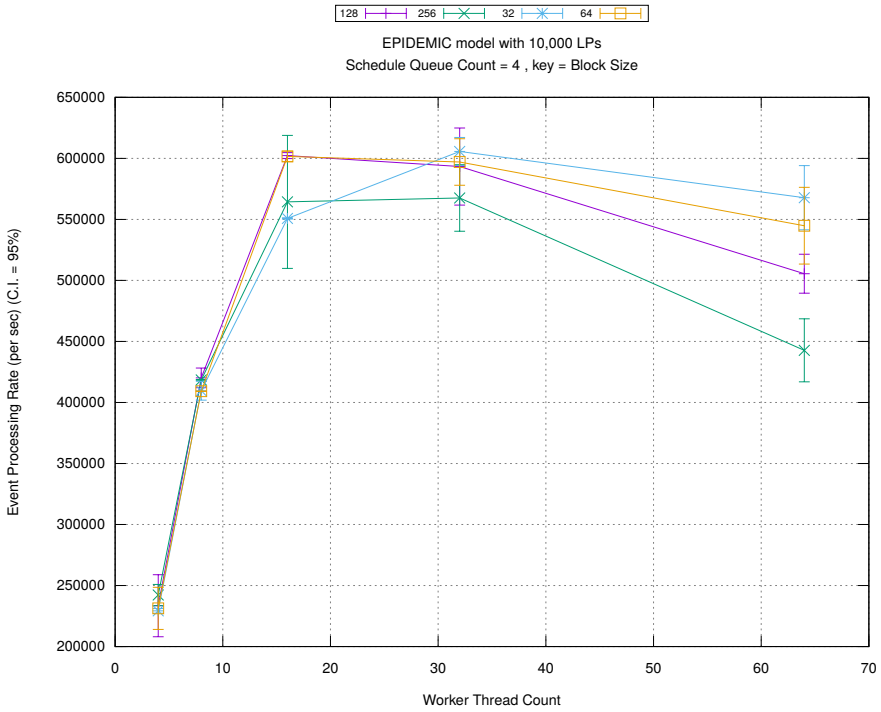
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

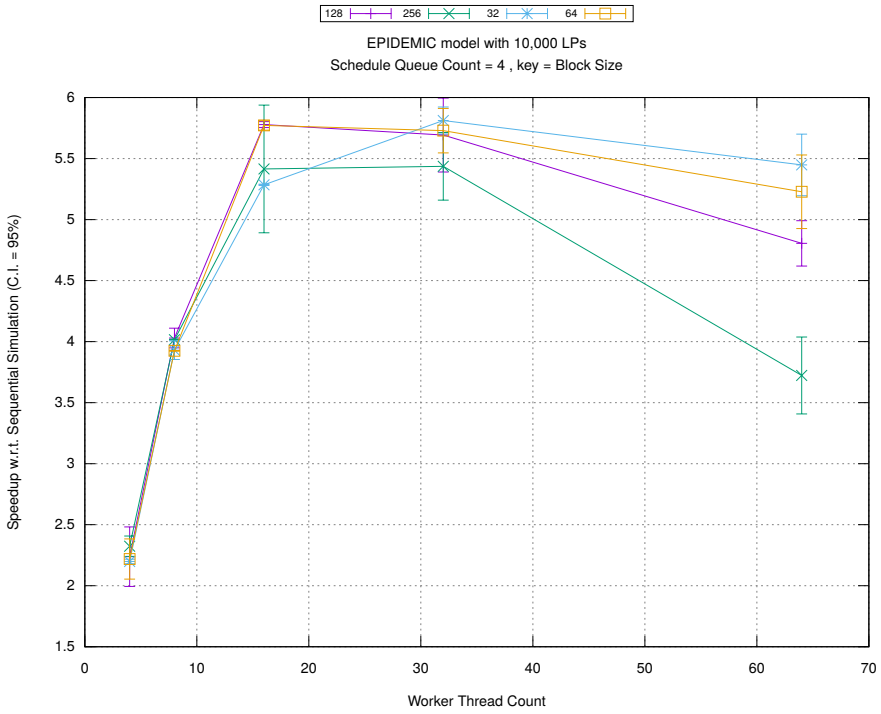


(b) Event Commitment Ratio

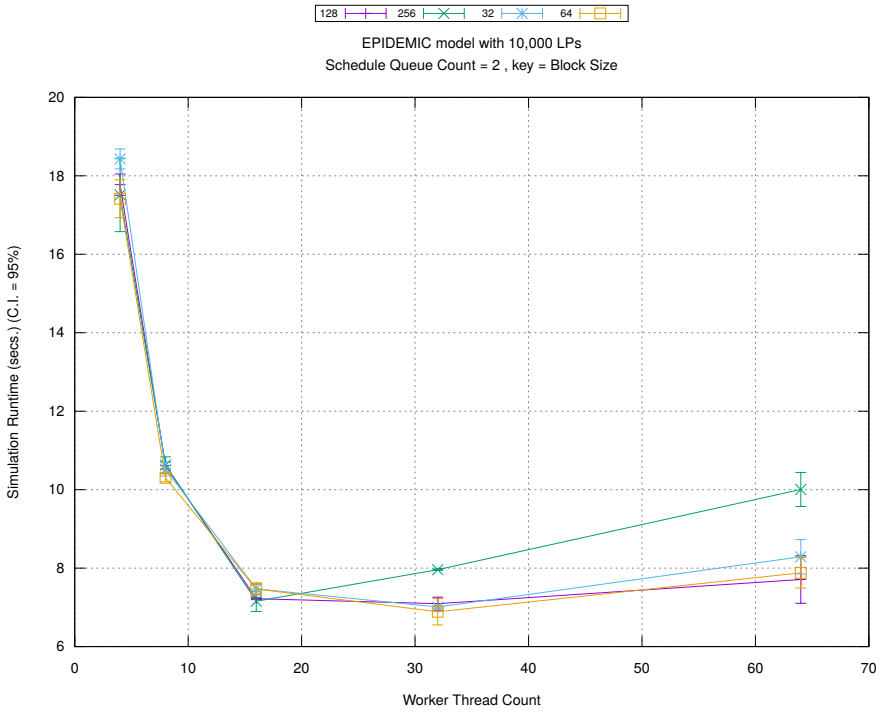


(c) Event Processing Rate (per second)

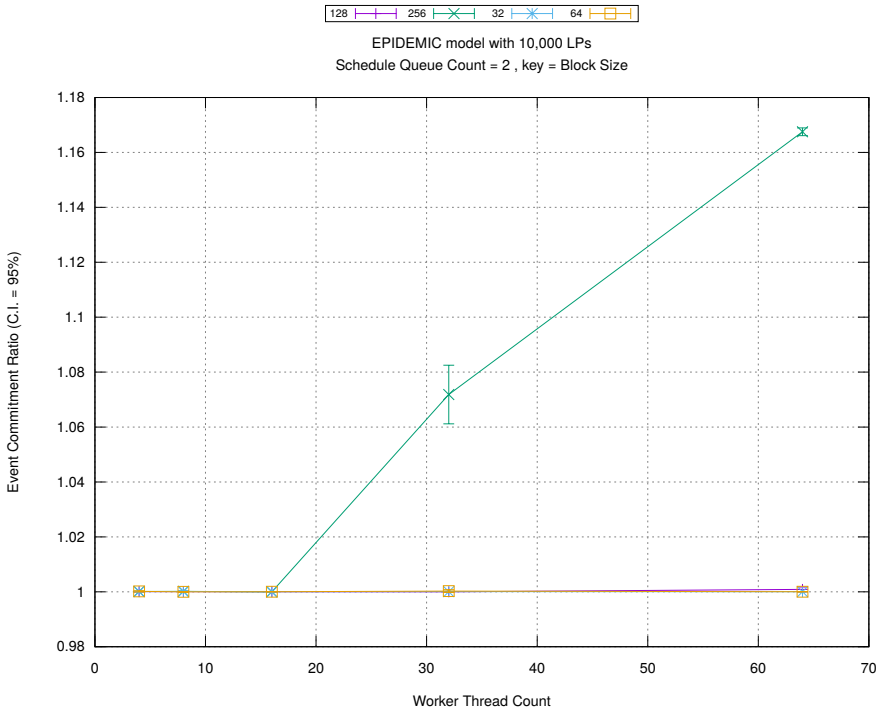
Figure A.92: epidemic 10k ws/plots/blocks/threads vs blocksize key count 4



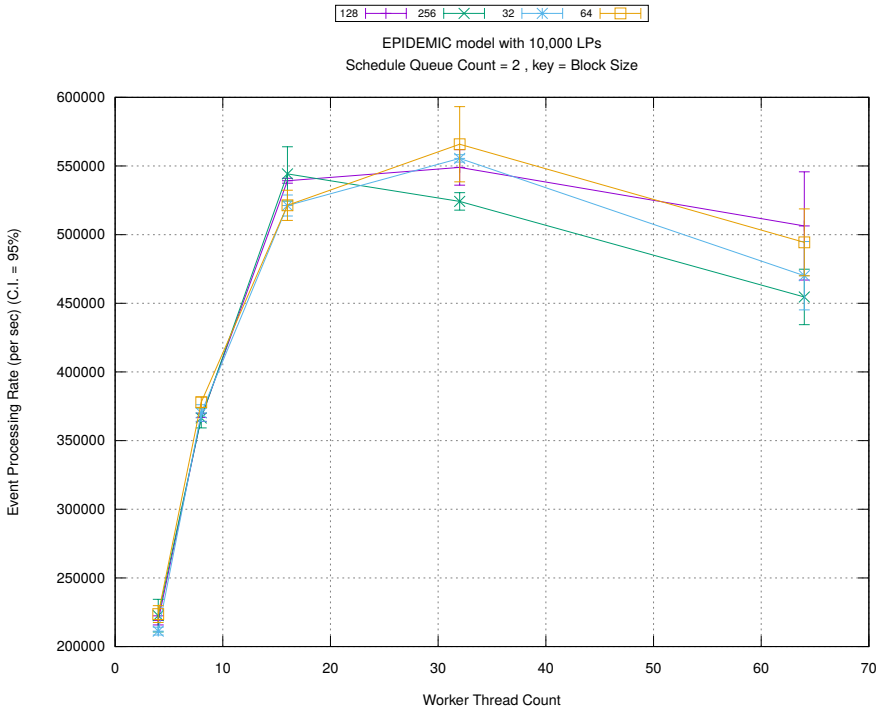
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

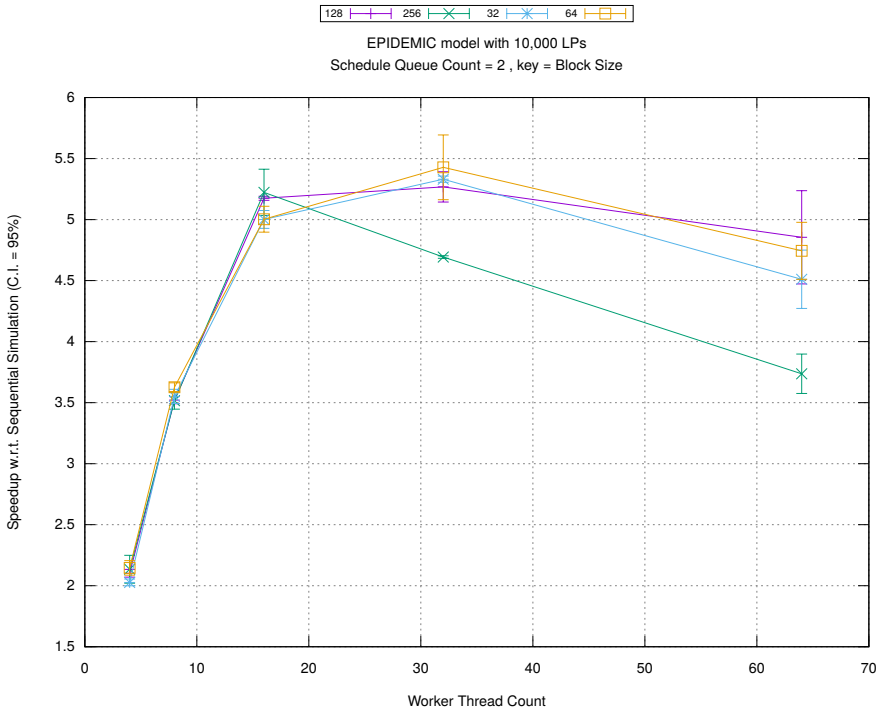


(b) Event Commitment Ratio

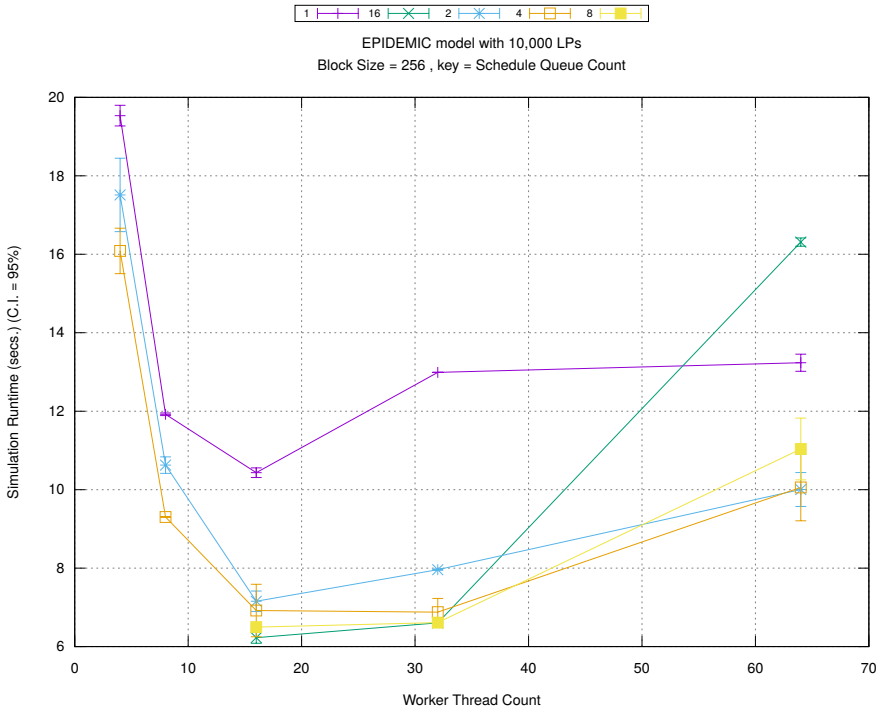


(c) Event Processing Rate (per second)

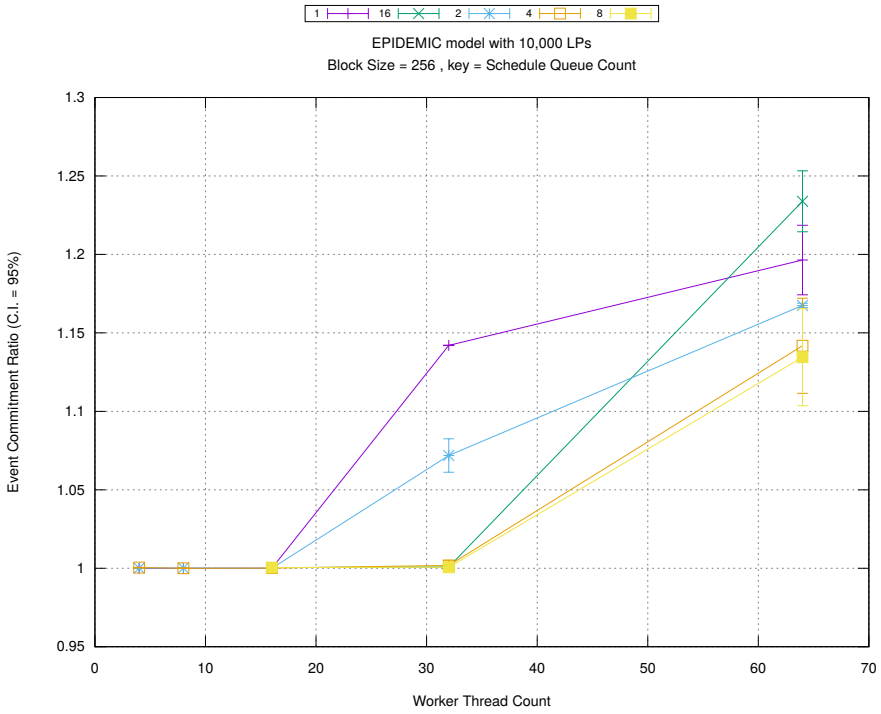
Figure A.93: epidemic 10k ws/plots/blocks/threads vs blocksize key count 2



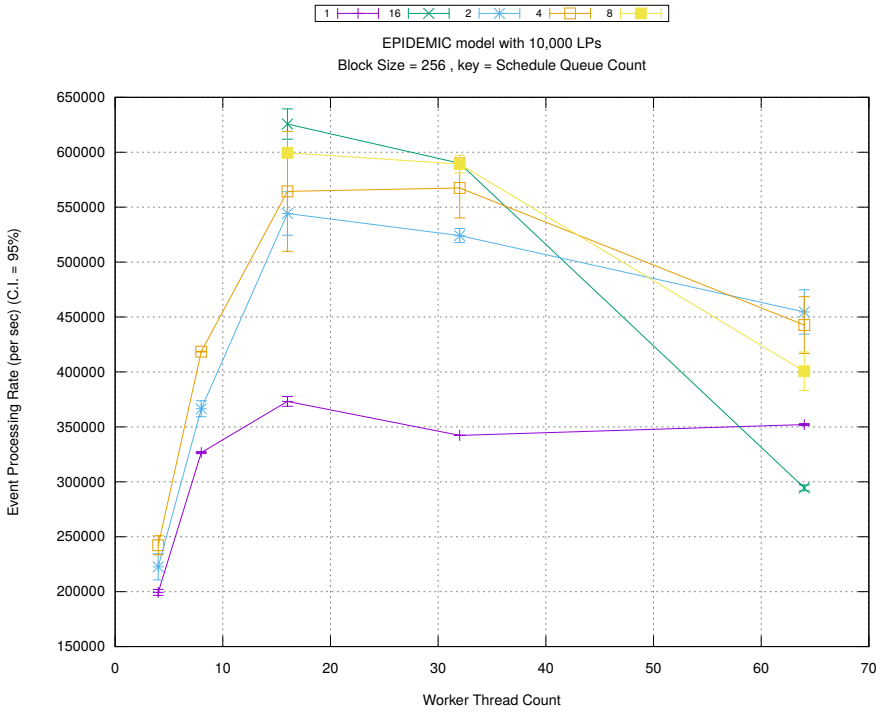
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

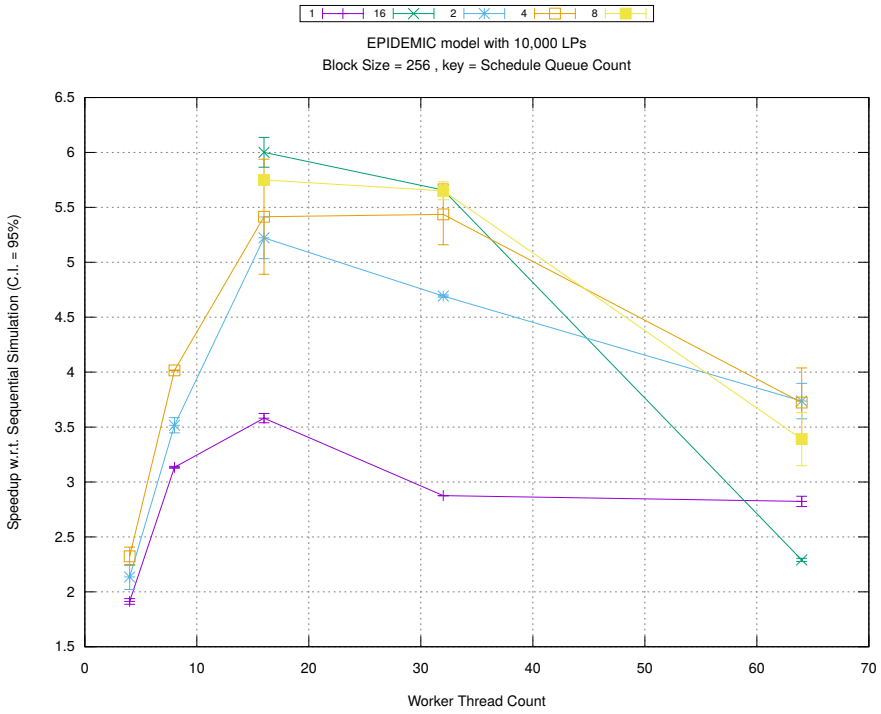


(b) Event Commitment Ratio

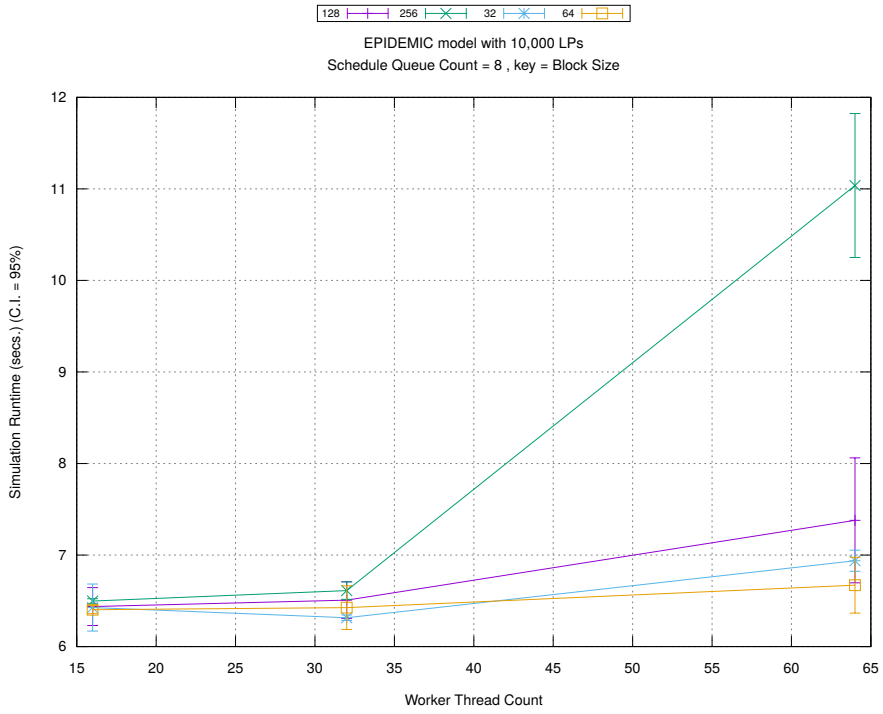


(c) Event Processing Rate (per second)

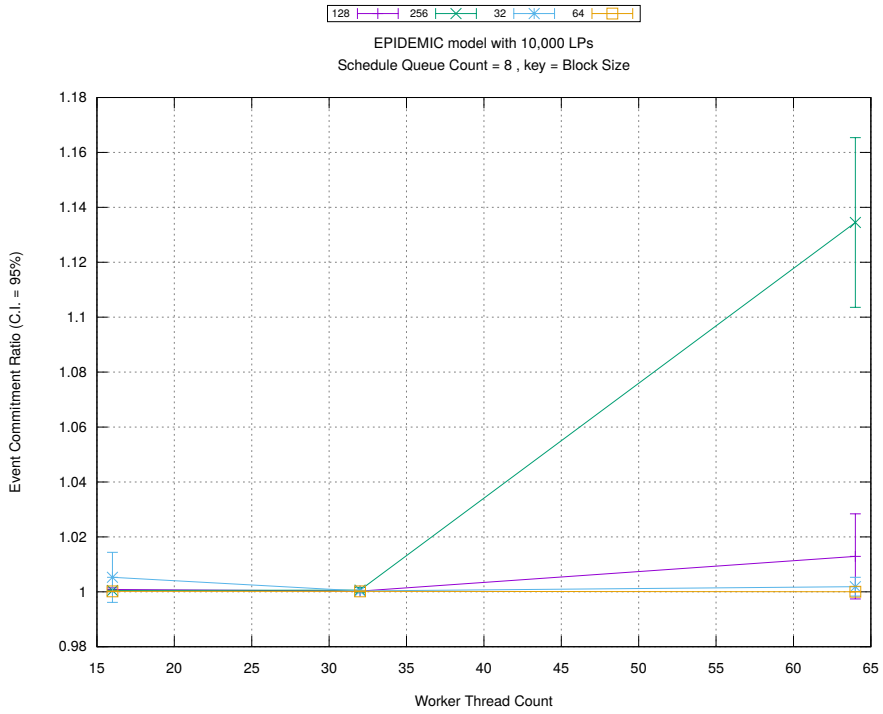
Figure A.94: epidemic 10k ws/plots/blocks/threads vs count key blocksize 256



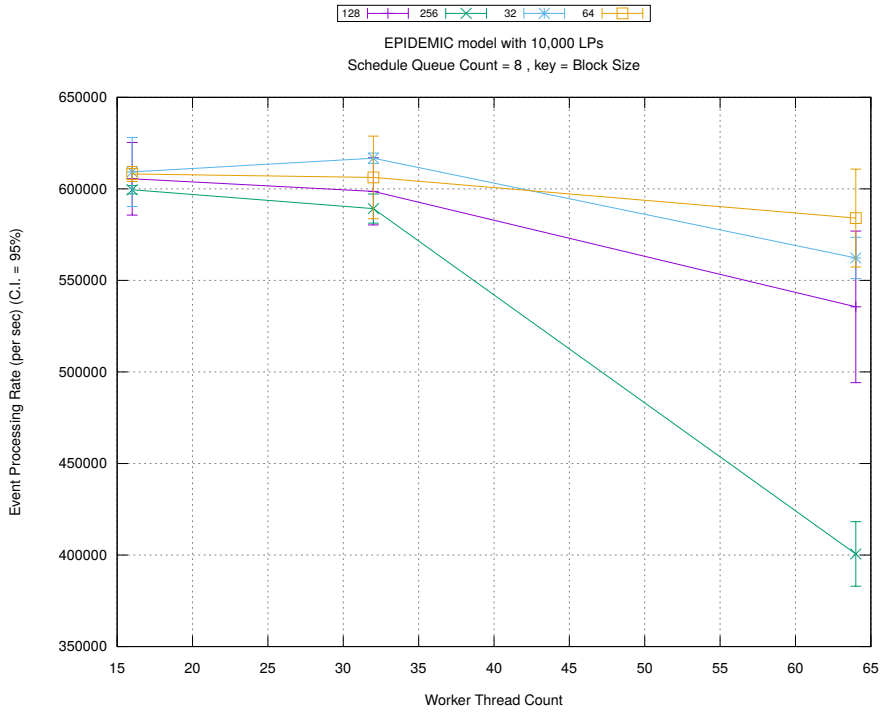
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

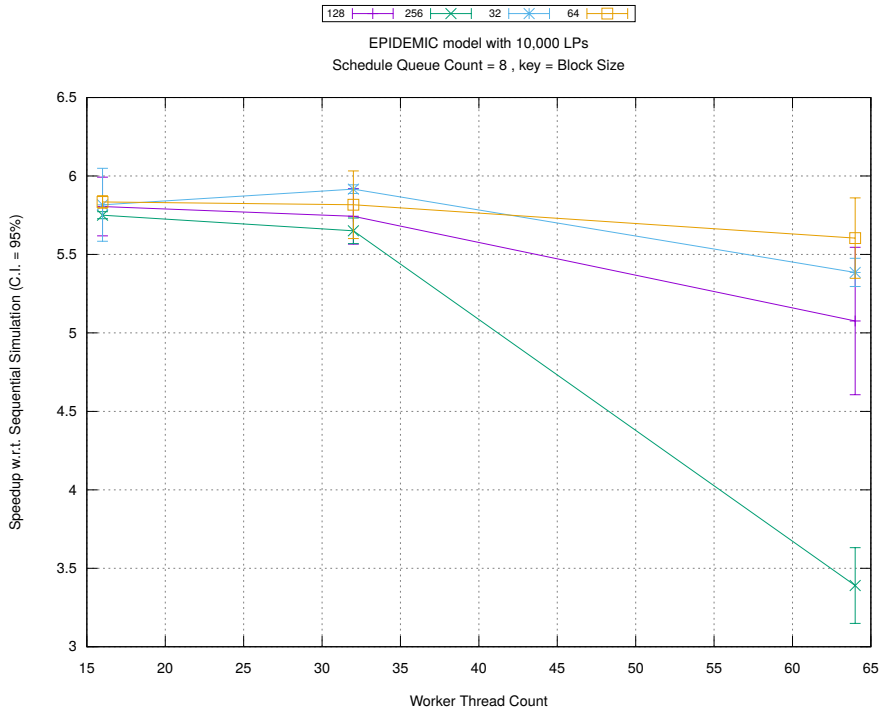


(b) Event Commitment Ratio

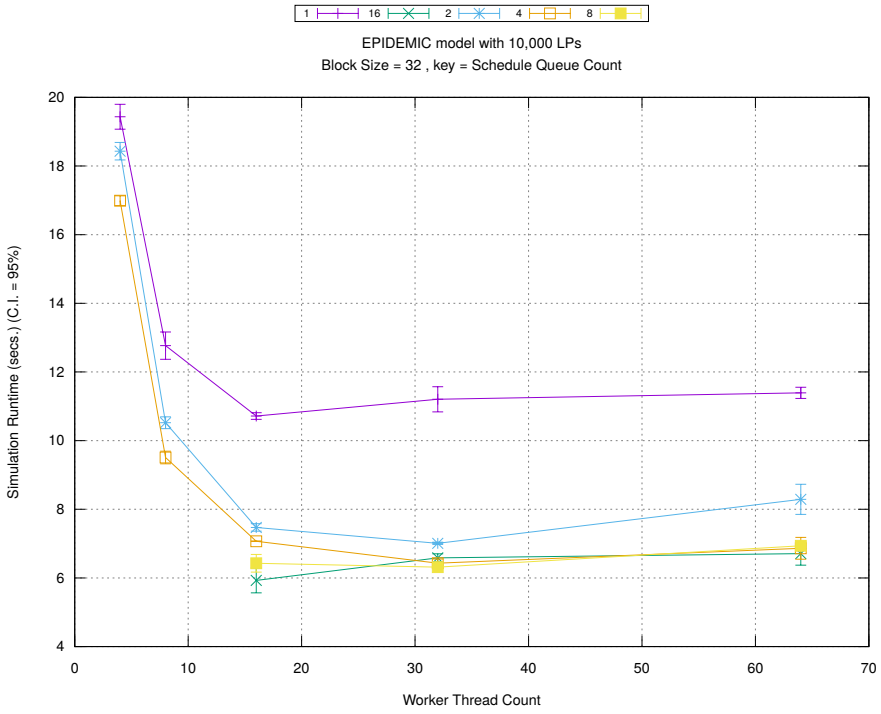


(c) Event Processing Rate (per second)

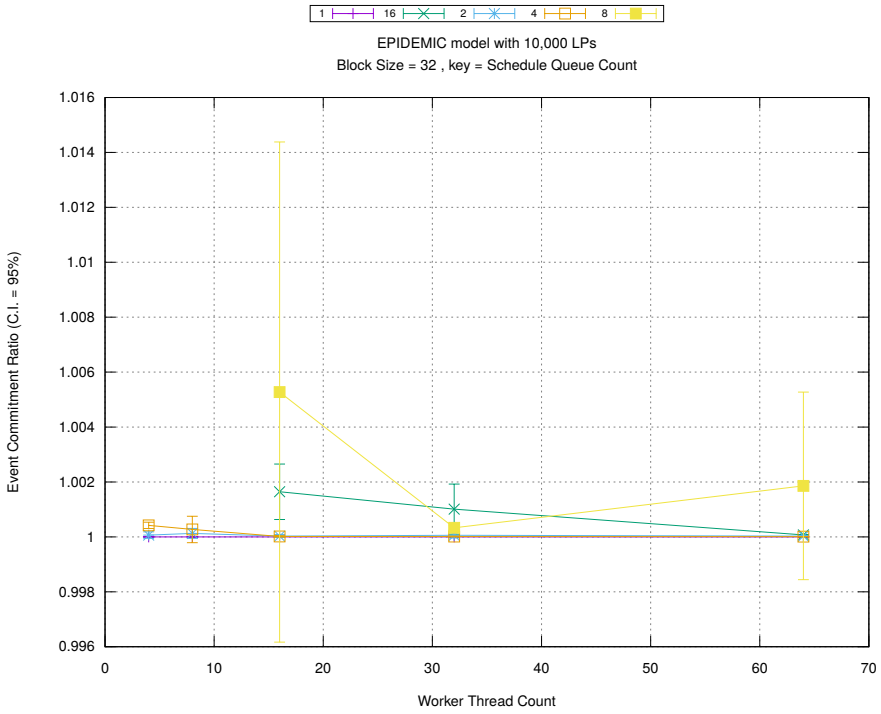
Figure A.95: epidemic 10k ws/plots/blocks/threads vs blocksize key count 8



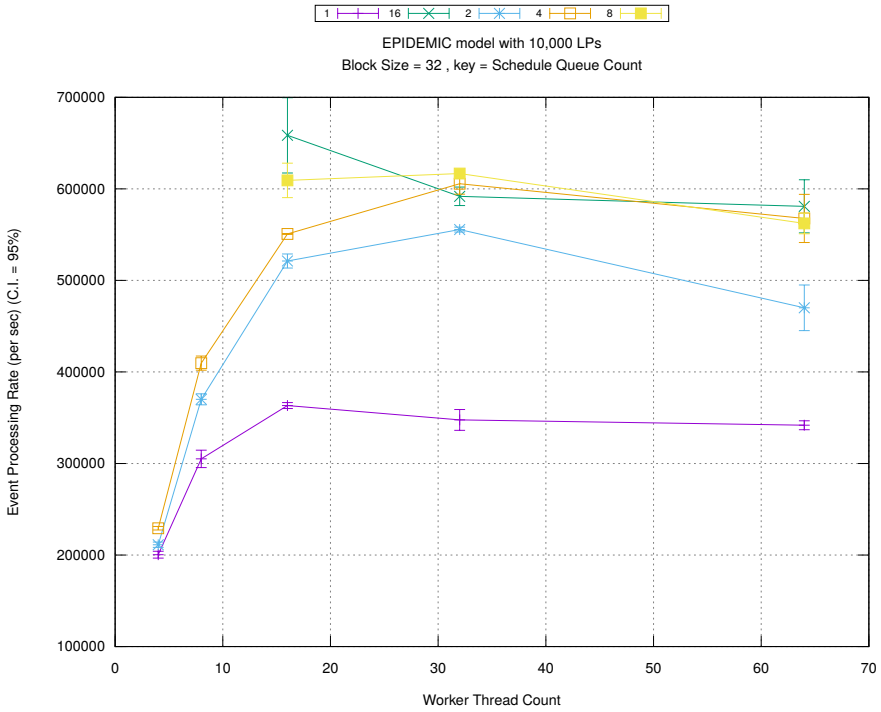
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

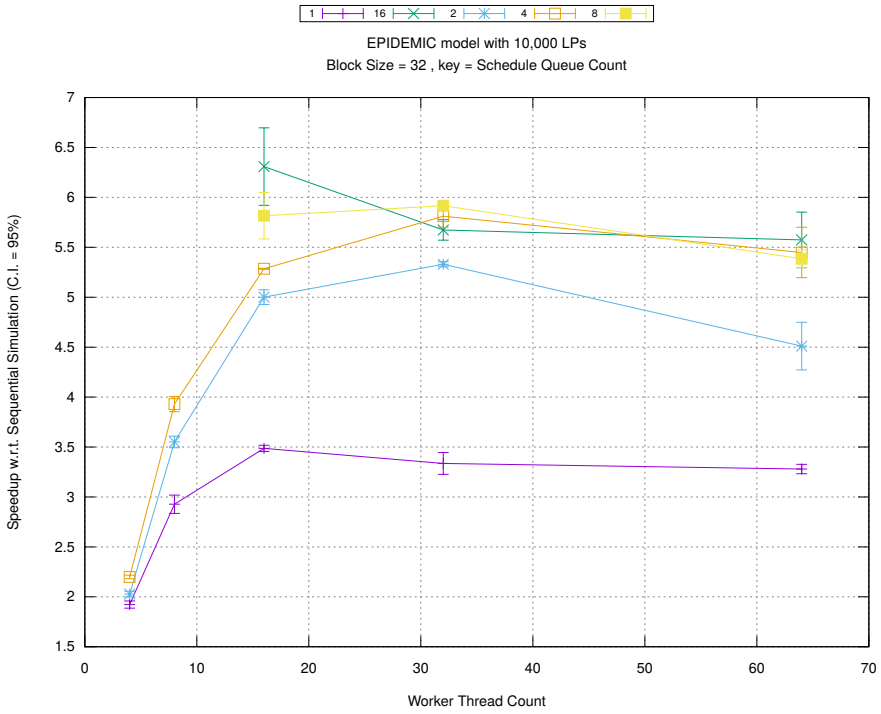


(b) Event Commitment Ratio

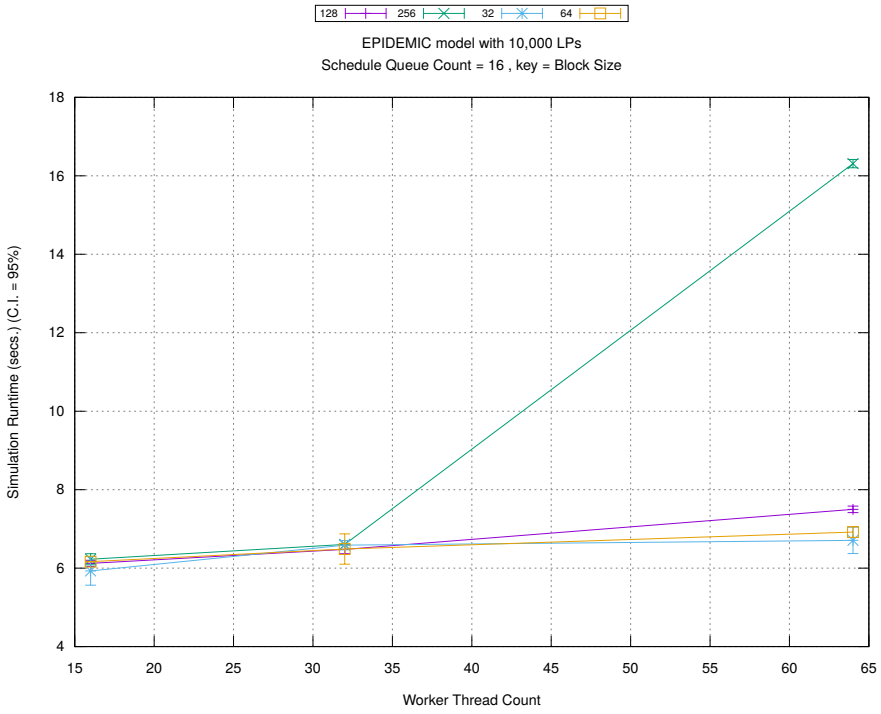


(c) Event Processing Rate (per second)

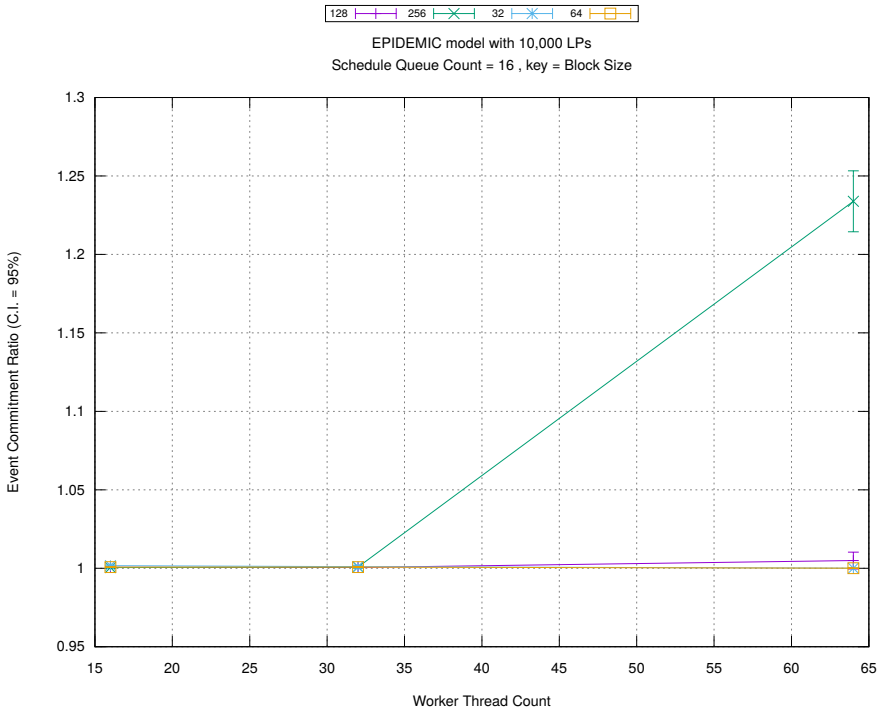
Figure A.96: epidemic 10k ws/plots/blocks/threads vs count key blocksize 32



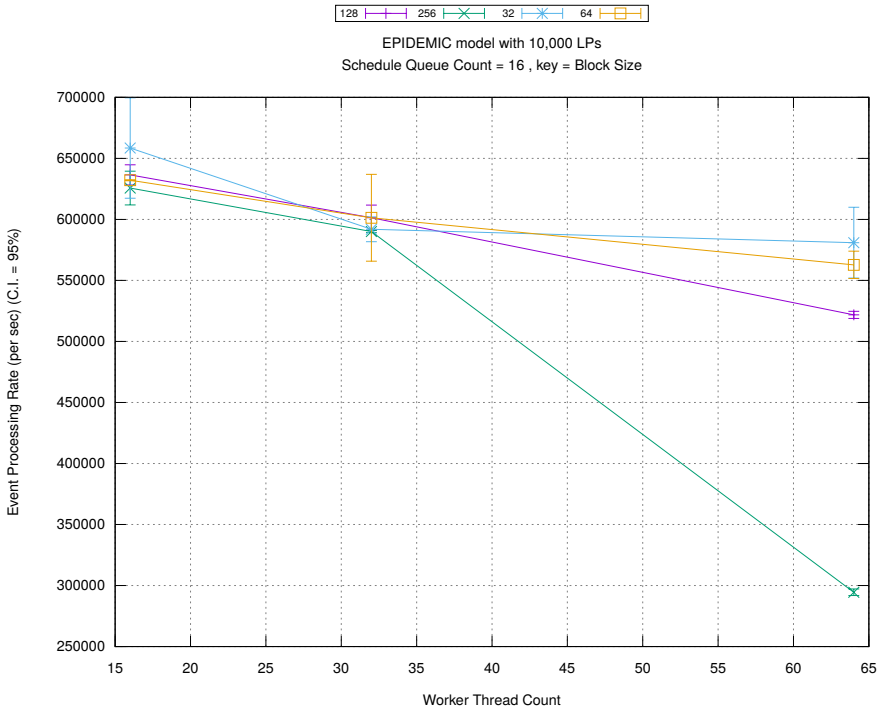
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

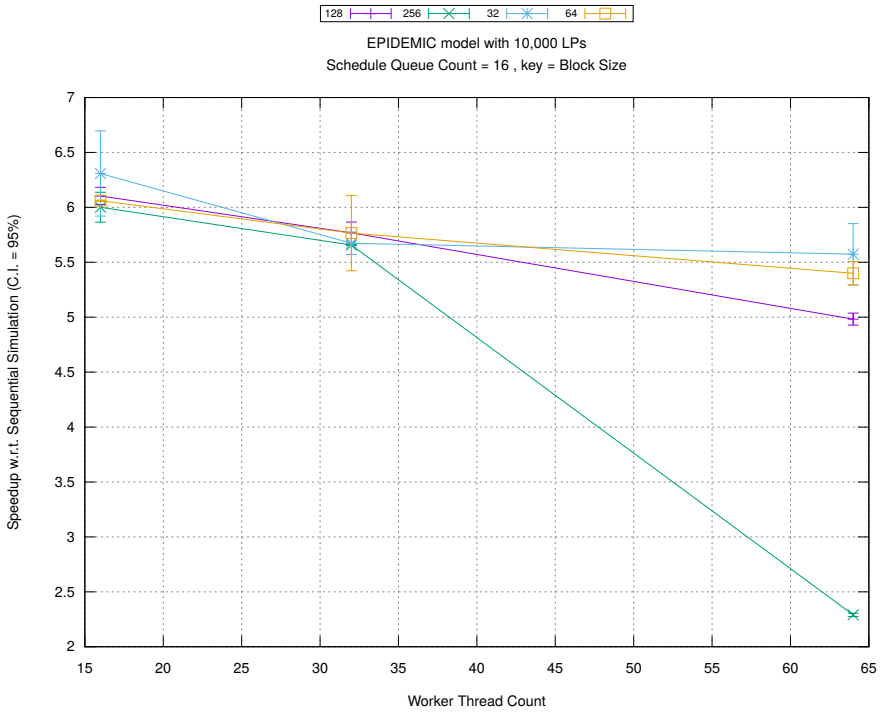


(b) Event Commitment Ratio

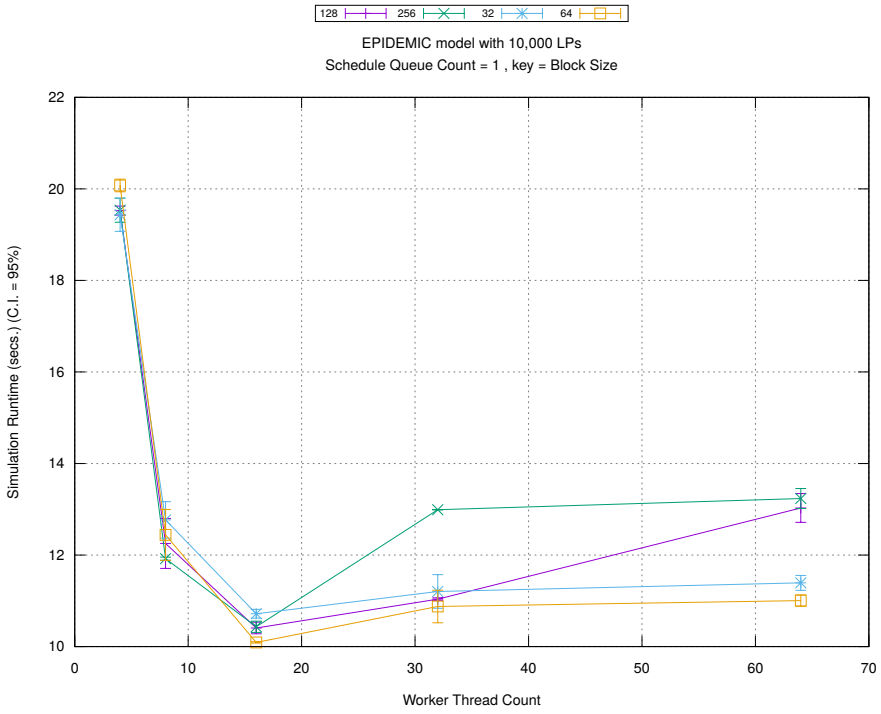


(c) Event Processing Rate (per second)

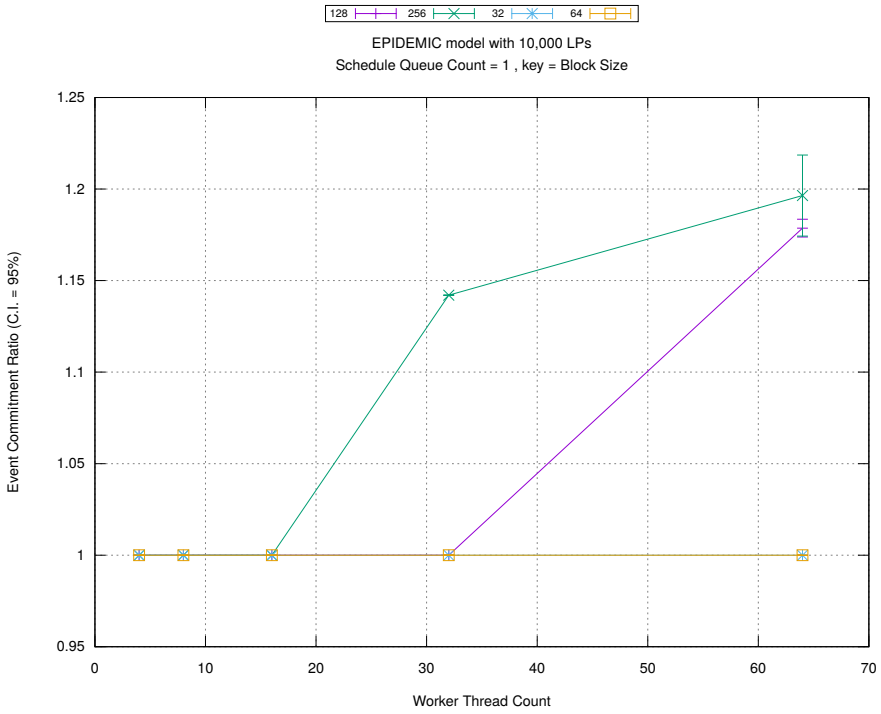
Figure A.97: epidemic 10k ws/plots/blocks/threads vs blocksize key count 16



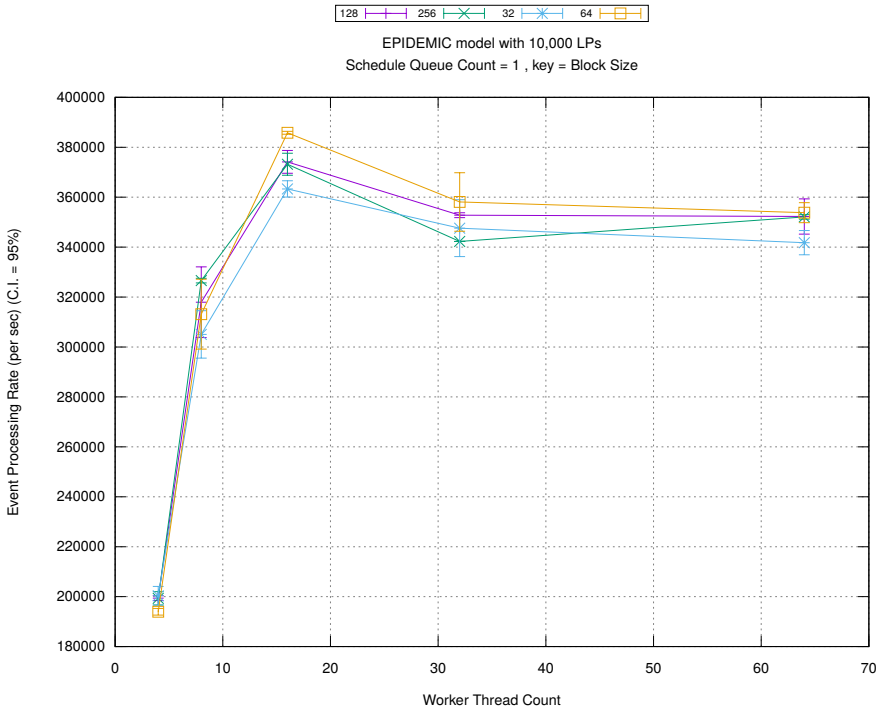
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

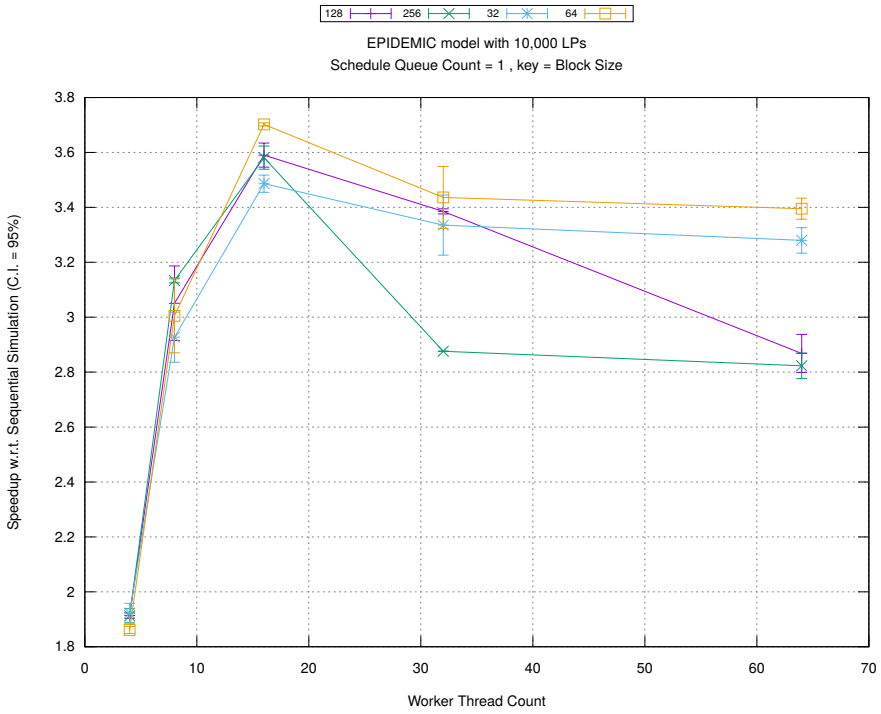


(b) Event Commitment Ratio

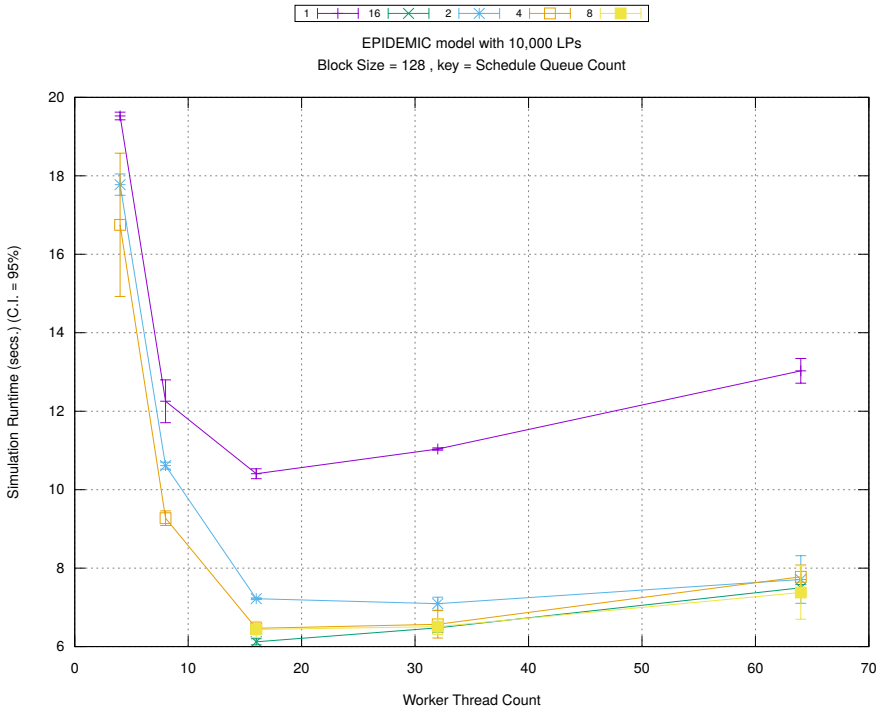


(c) Event Processing Rate (per second)

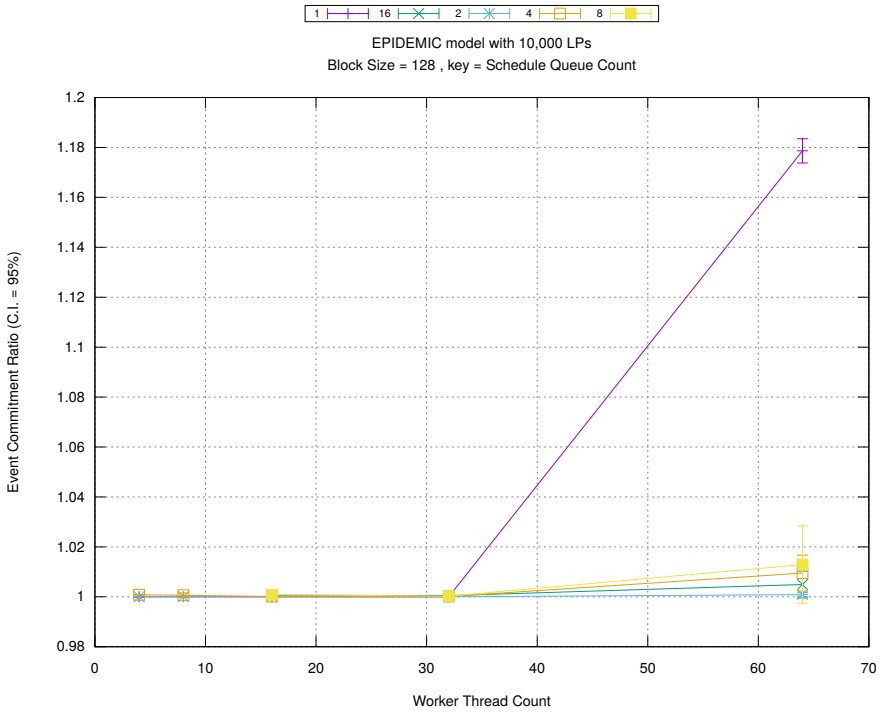
Figure A.98: epidemic 10k ws/plots/blocks/threads vs blocksize key count 1



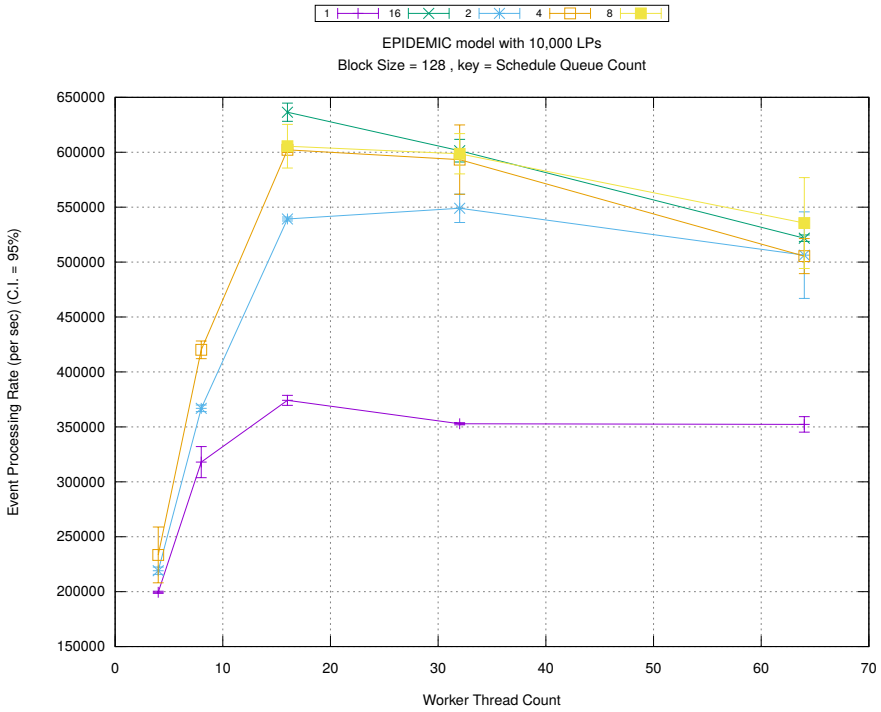
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

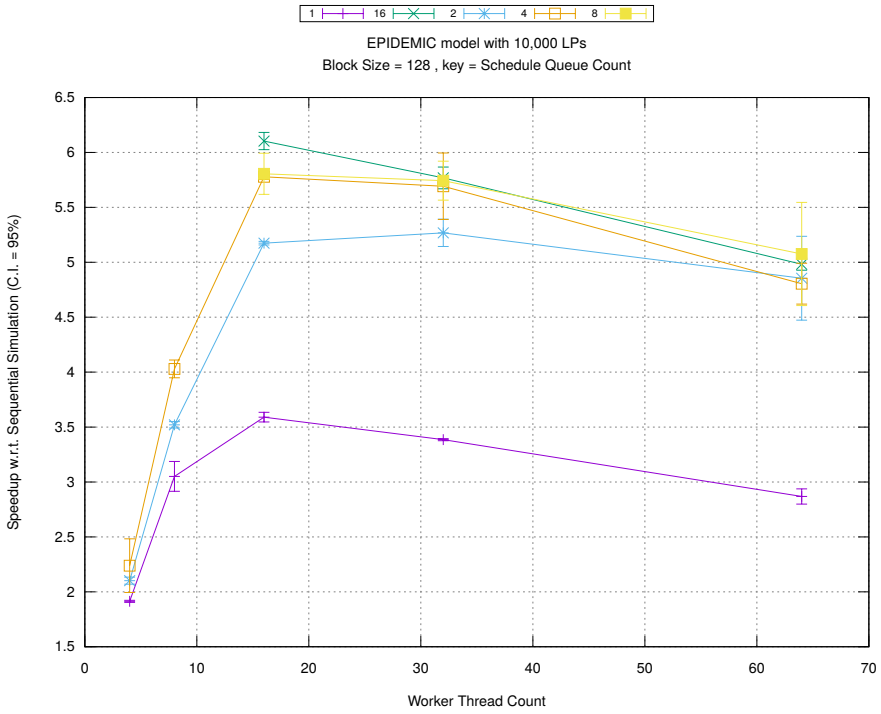


(b) Event Commitment Ratio

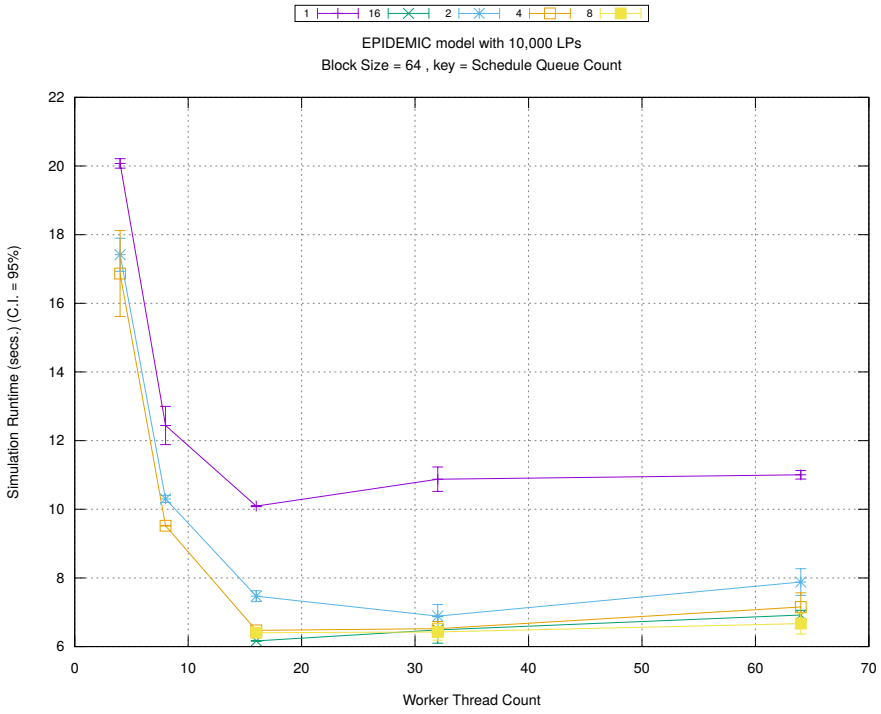


(c) Event Processing Rate (per second)

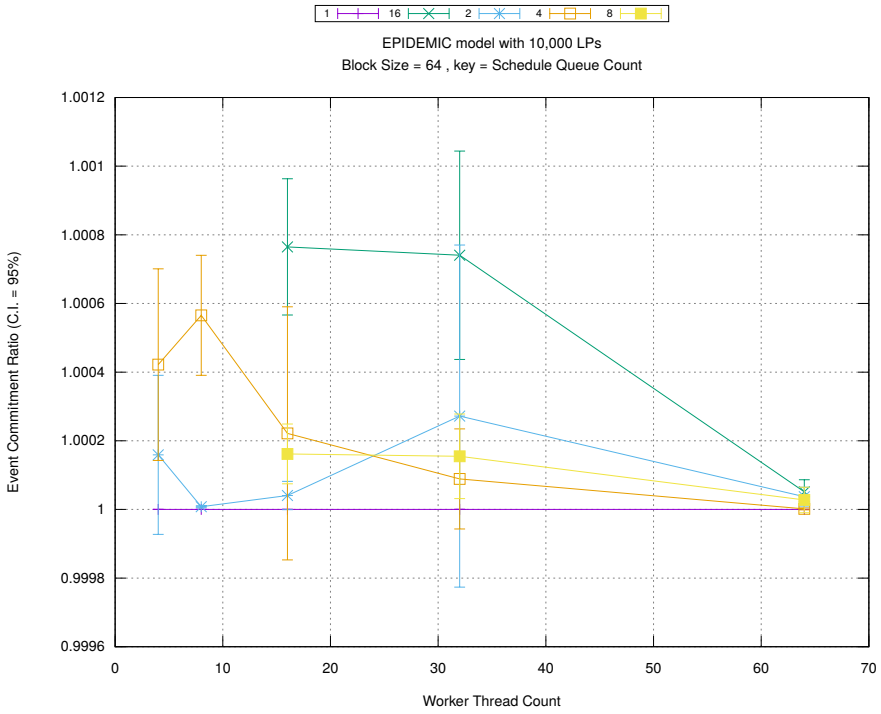
Figure A.99: epidemic 10k ws/plots/blocks/threads vs count key blocksize 128



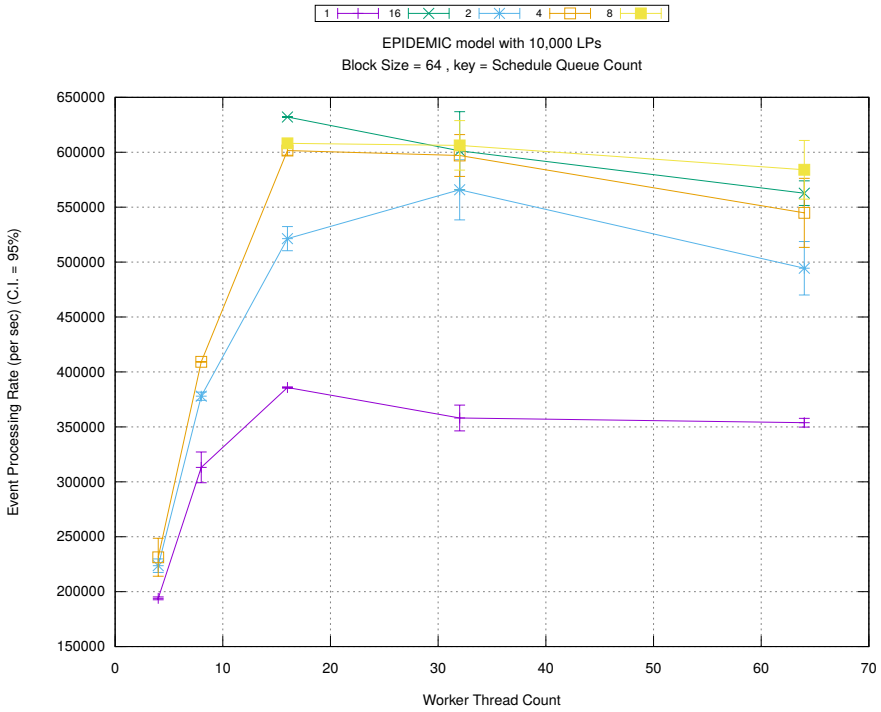
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

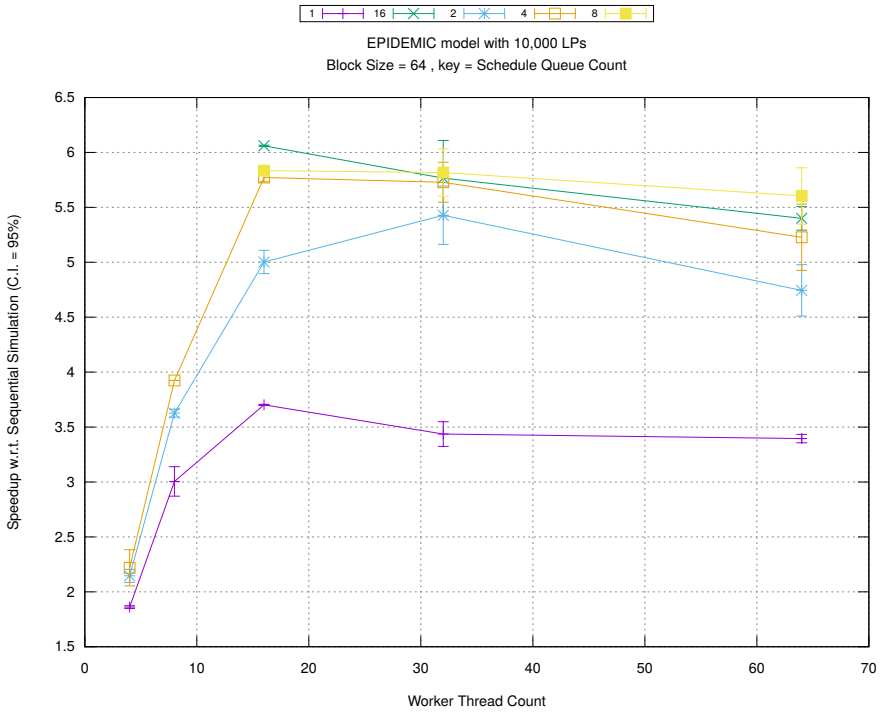


(b) Event Commitment Ratio



(c) Event Processing Rate (per second)

Figure A.100: epidemic 10k ws/plots/blocks/threads vs count key blocksize 64



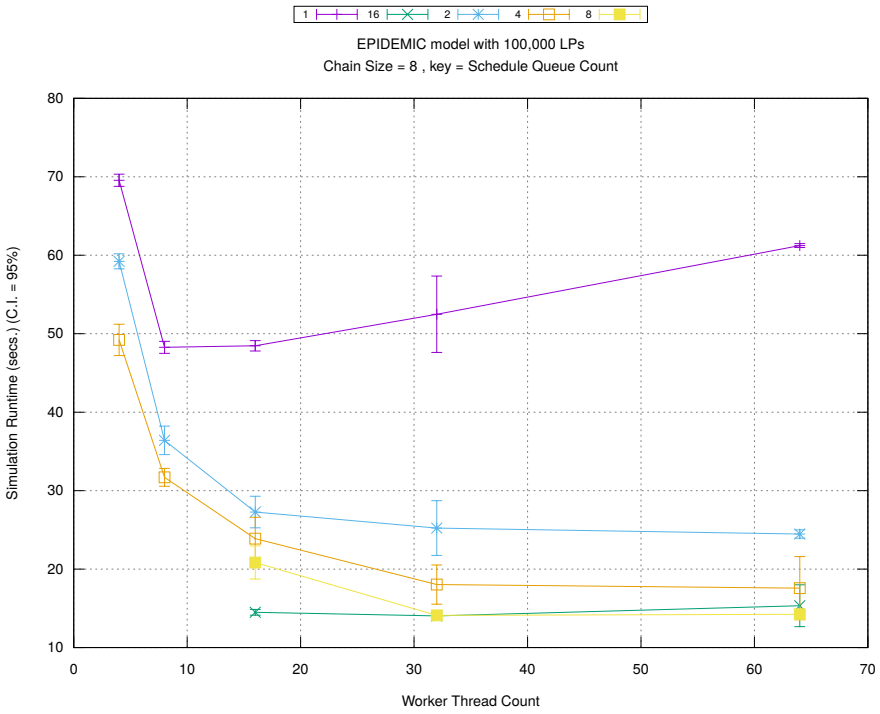
(d) Speedup w.r.t. Sequential Simulation

A.4 Epidemic Model with 100,000 LPs and Watts-Strogatz Network

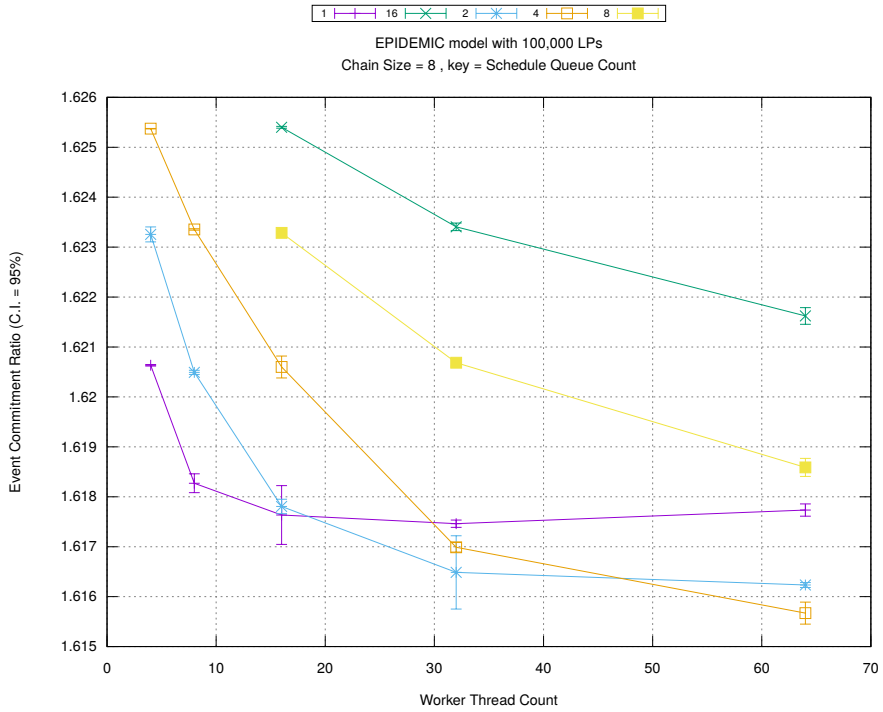
Table A.4 shows the configuration for this model.

Parameter	Values
Number of Intersections (or LPs)	100,000
Type of network connecting LPs	Watts-Strogatz [72]
Population Size	500,000
Simulation Time	6,000 timestamp units
Sequential Simulation Time for calculating modularity	200 timestamp units

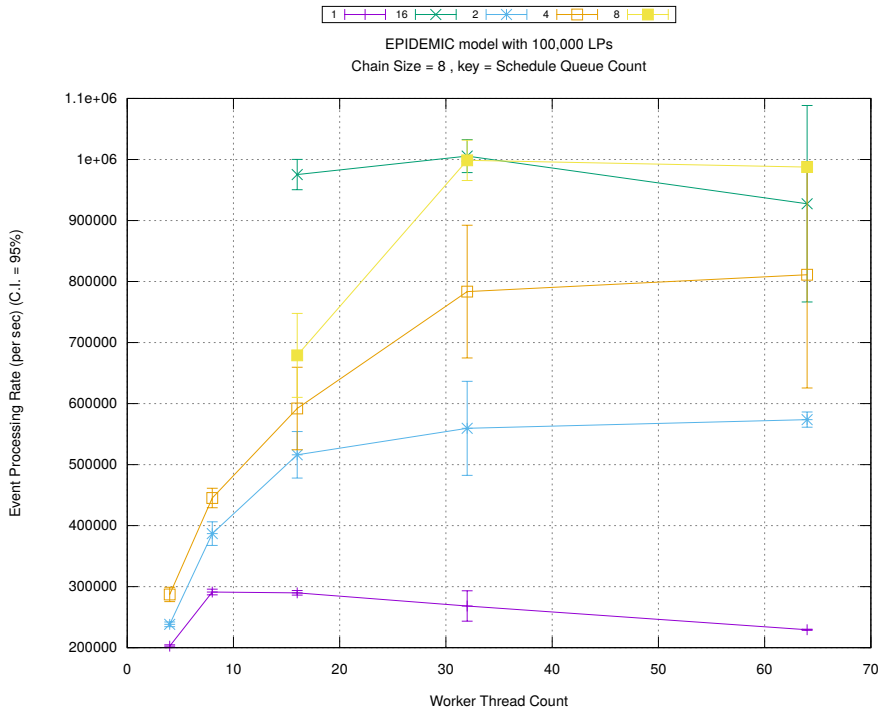
Table A.4: LARGE EPIDEMIC MODEL WITH WATTS-STROGATZ setup



(a) Simulation Runtime (in seconds)

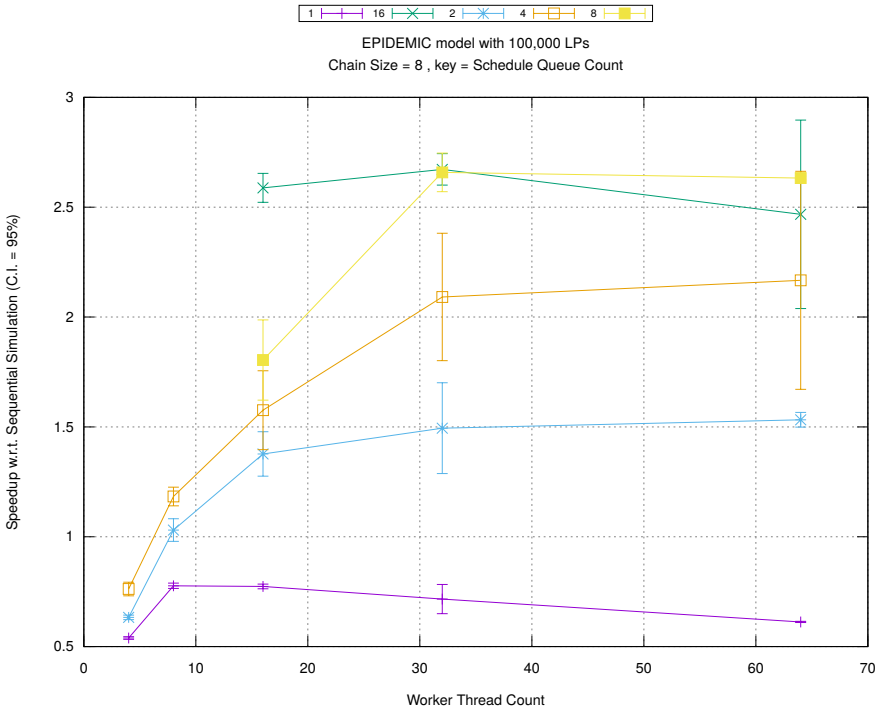


(b) Event Commitment Ratio

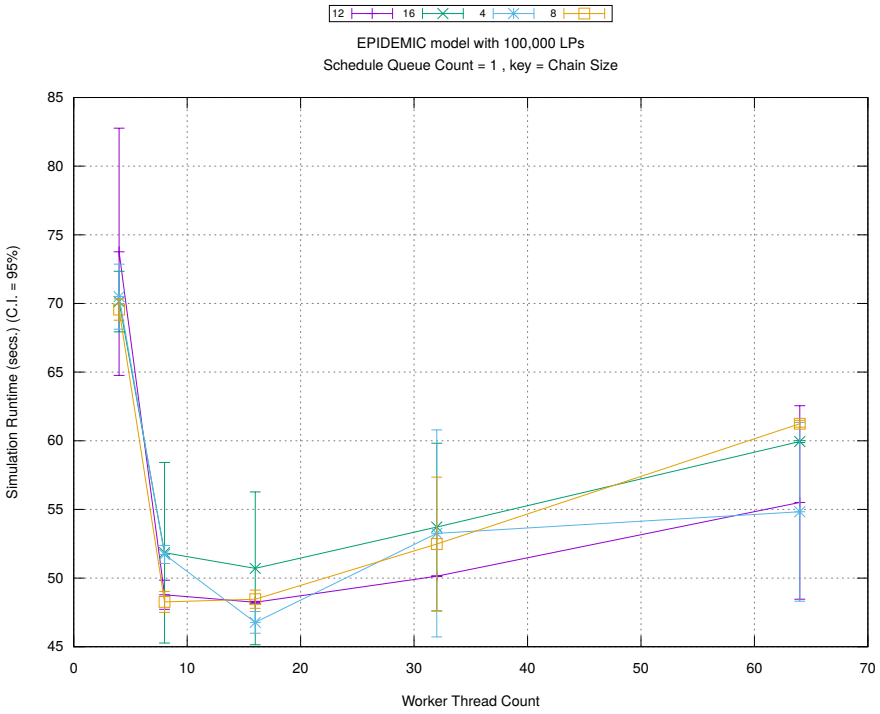


(c) Event Processing Rate (per second)

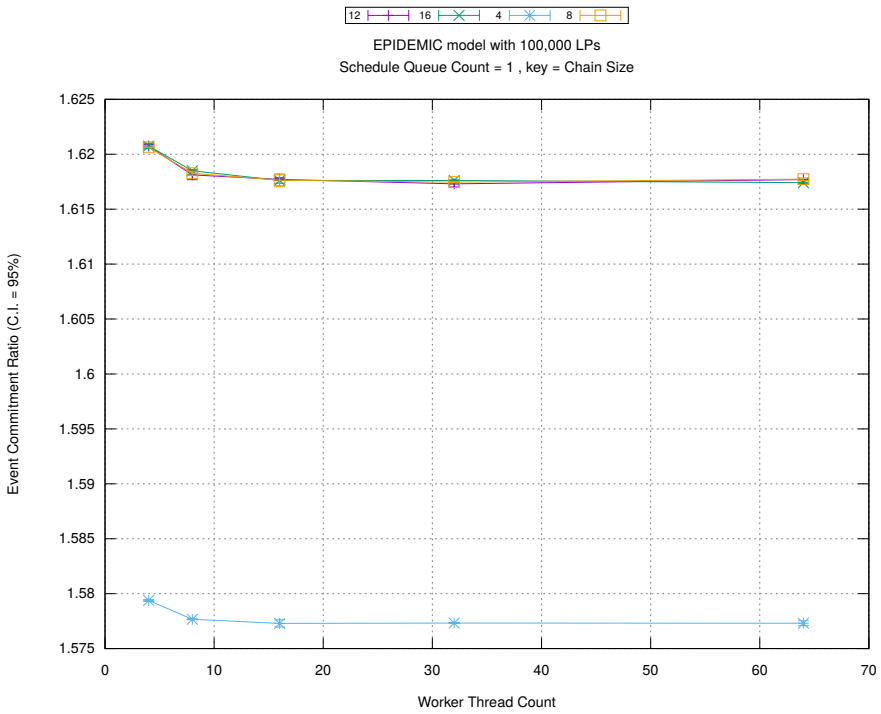
Figure A.101: epidemic 100k ws/plots/chains/threads vs count key chainsize 8



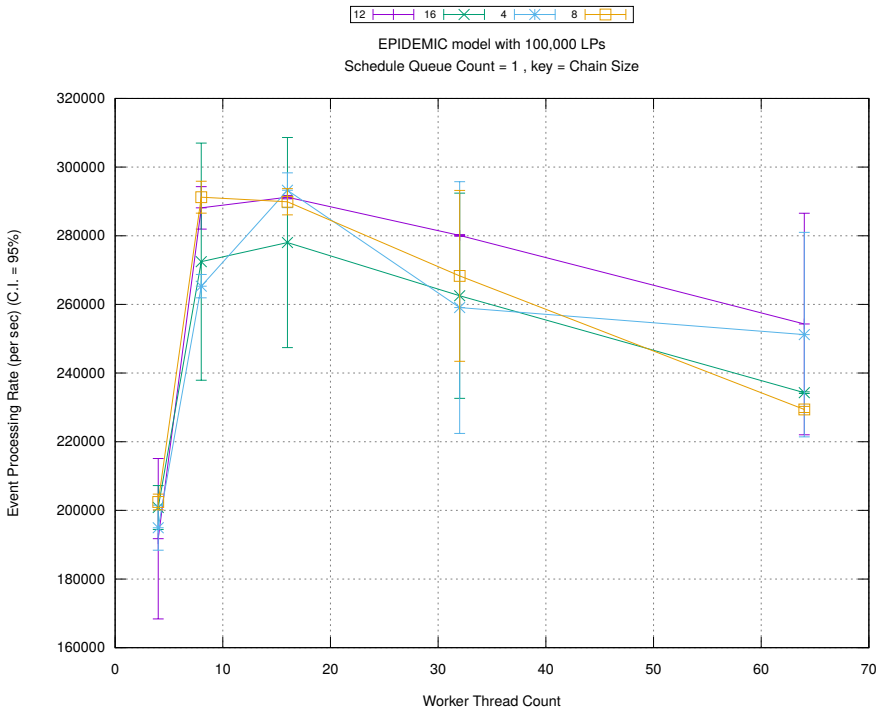
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

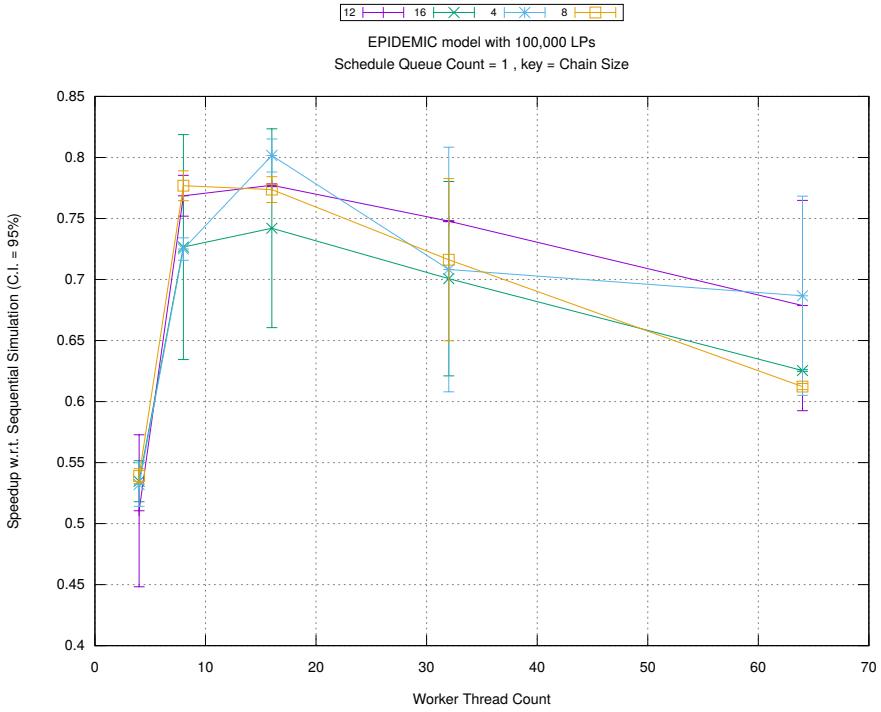


(b) Event Commitment Ratio

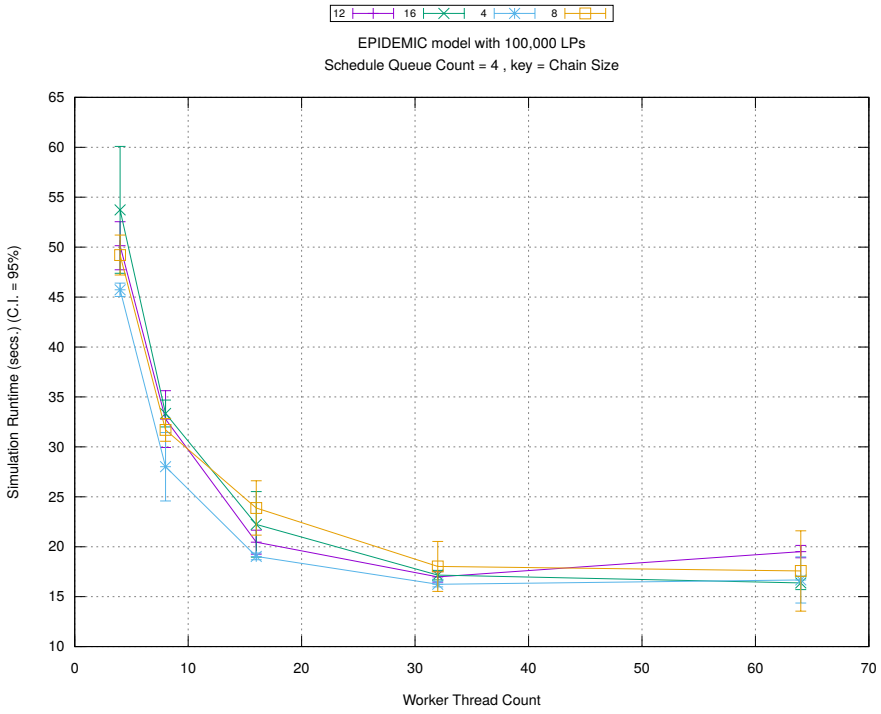


(c) Event Processing Rate (per second)

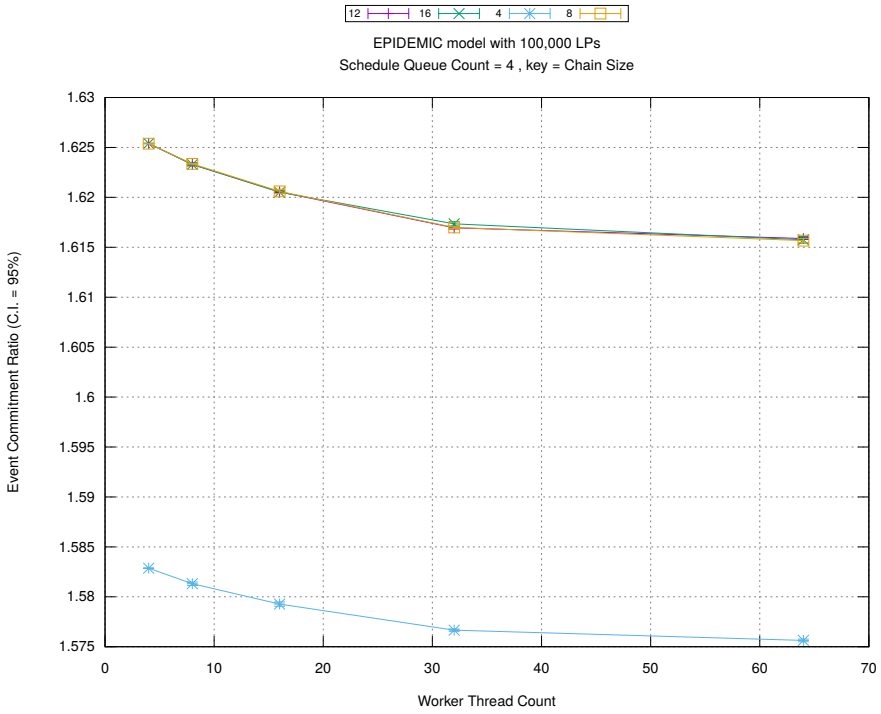
Figure A.102: epidemic 100k ws/plots/chains/threads vs chainsize key count 1



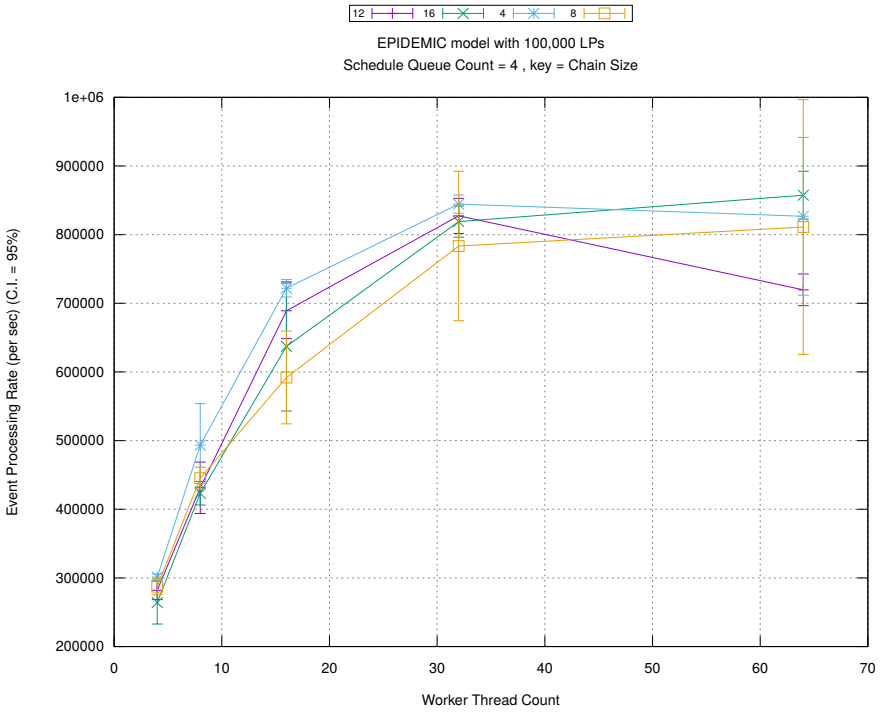
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

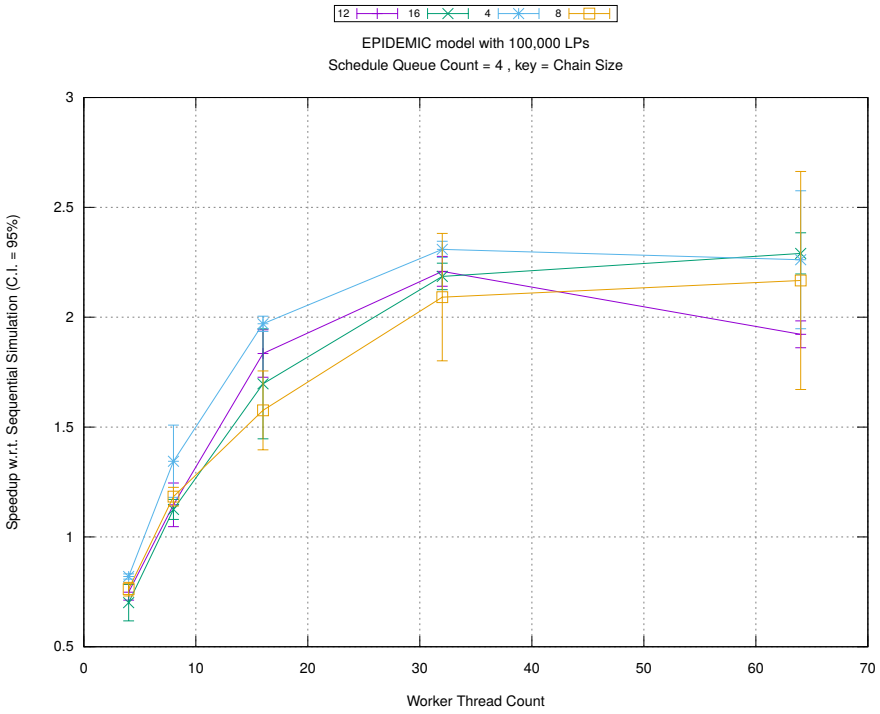


(b) Event Commitment Ratio

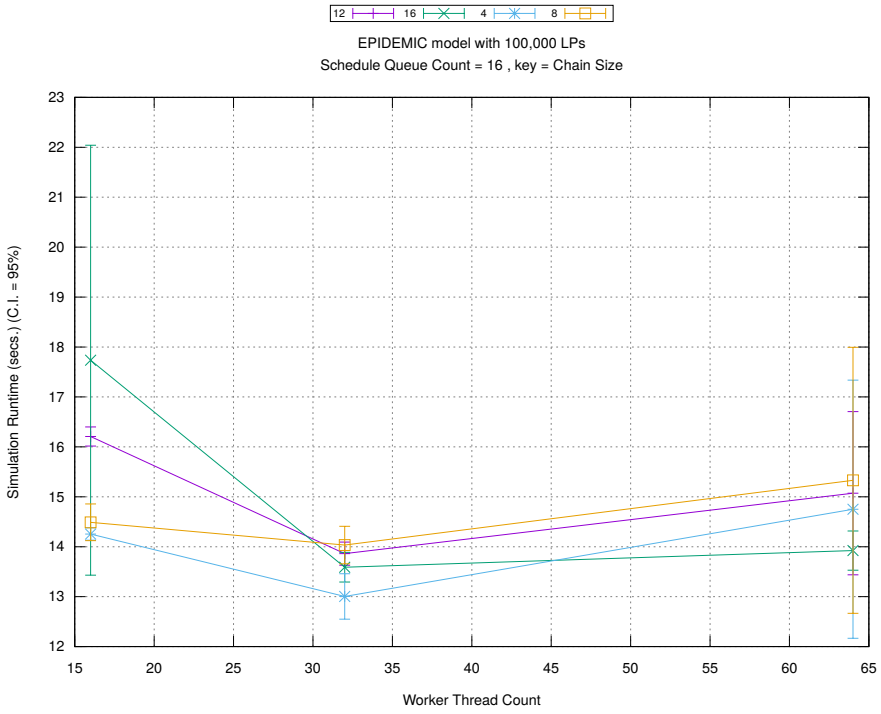


(c) Event Processing Rate (per second)

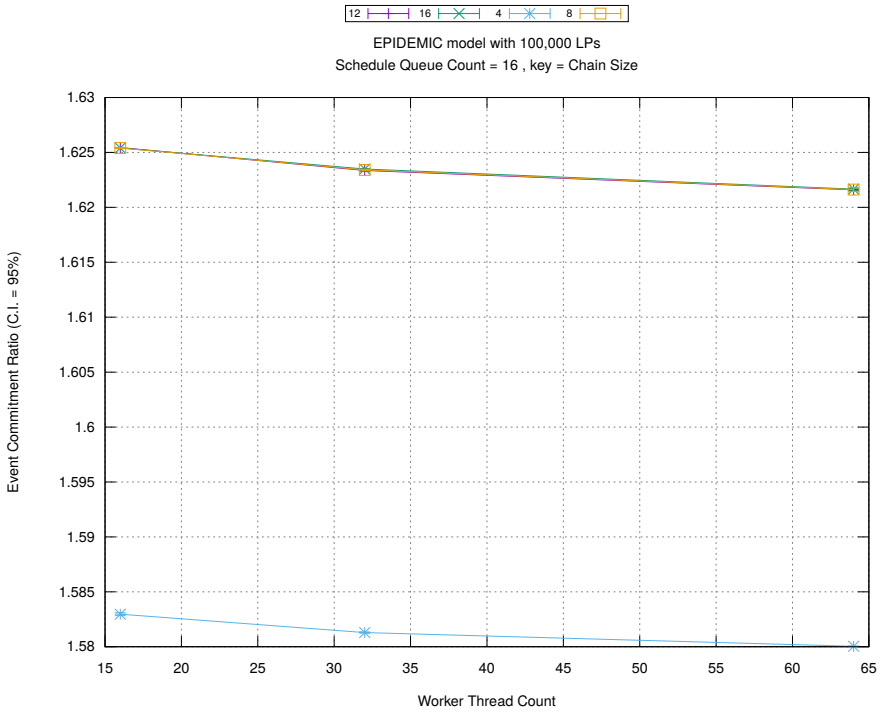
Figure A.103: epidemic 100k ws/plots/chains/threads vs chainsize key count 4



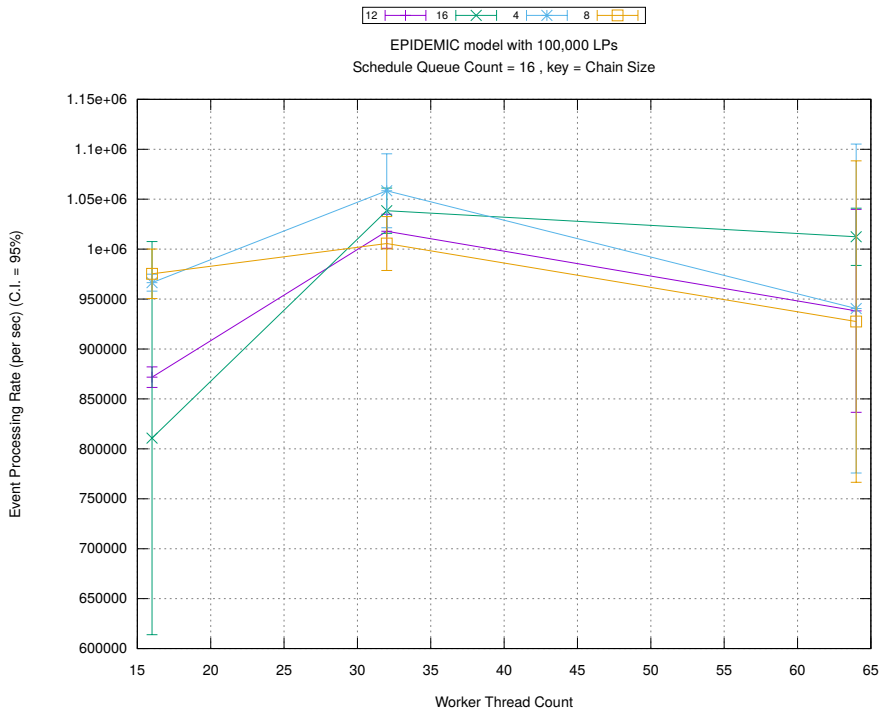
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

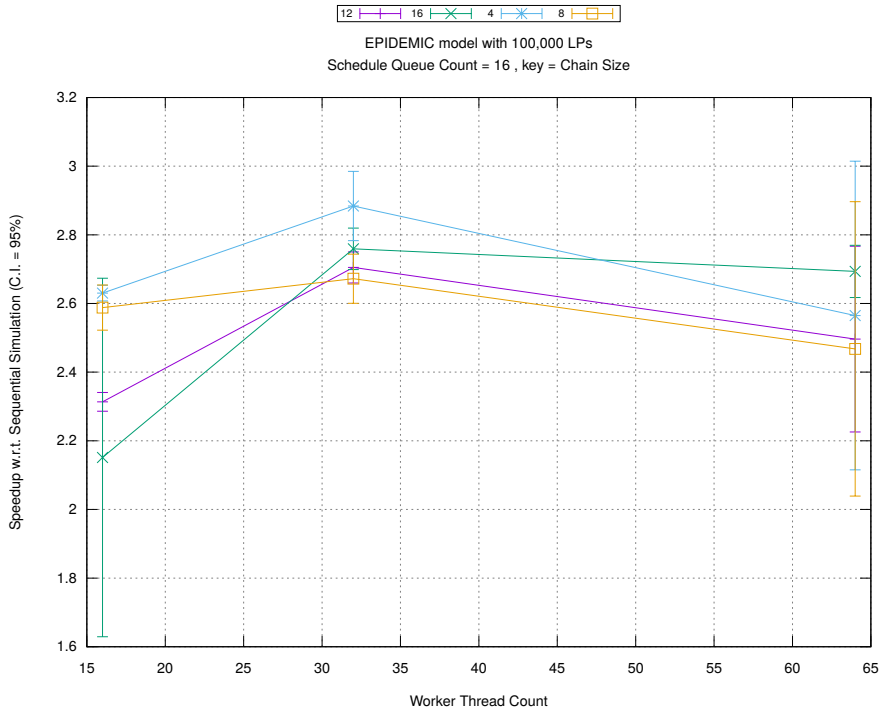


(b) Event Commitment Ratio

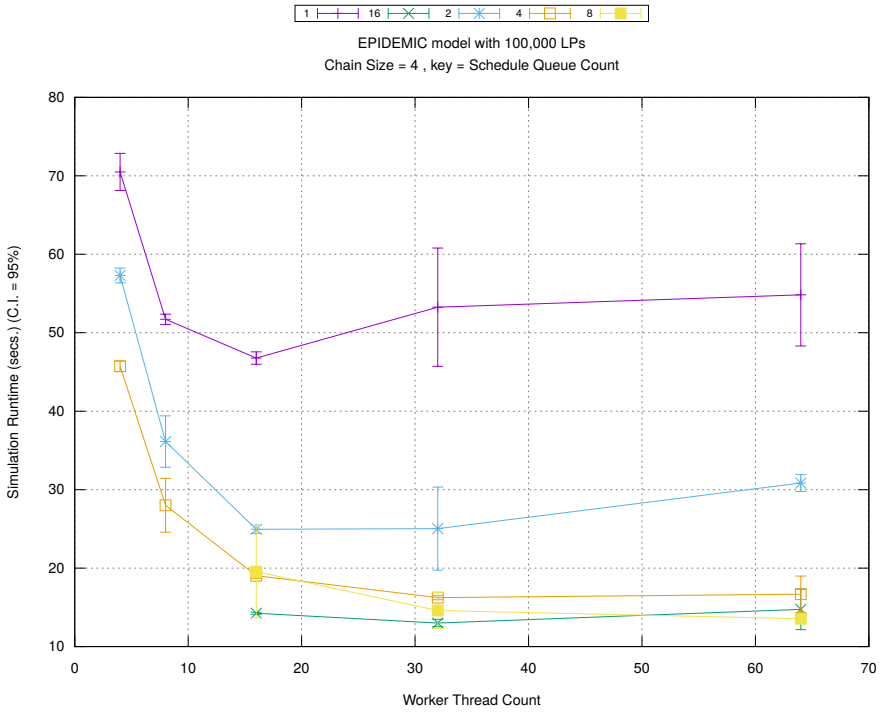


(c) Event Processing Rate (per second)

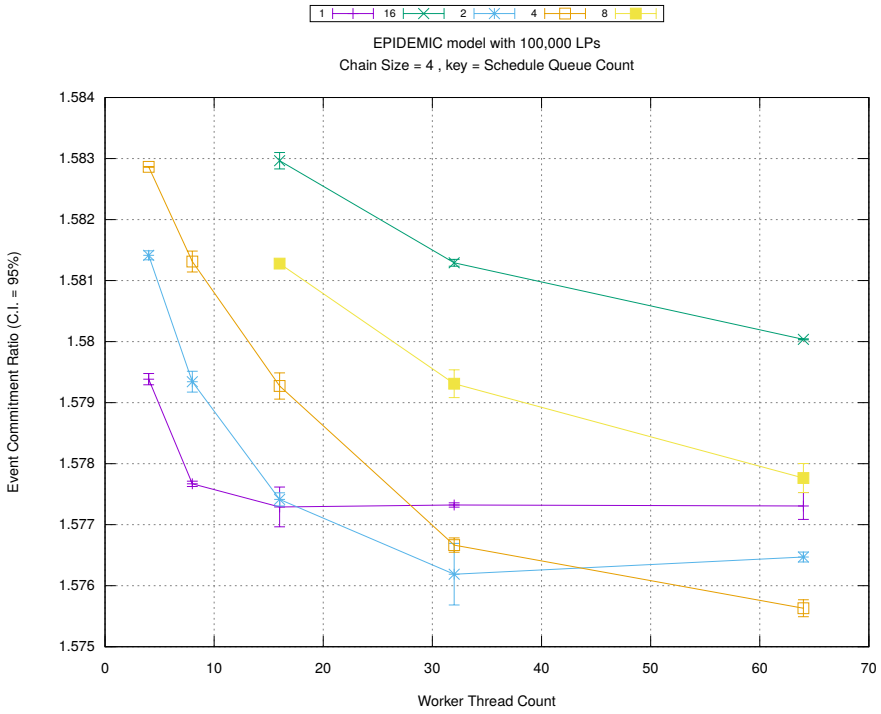
Figure A.104: epidemic 100k ws/plots/chains/threads vs chainsize key count 16



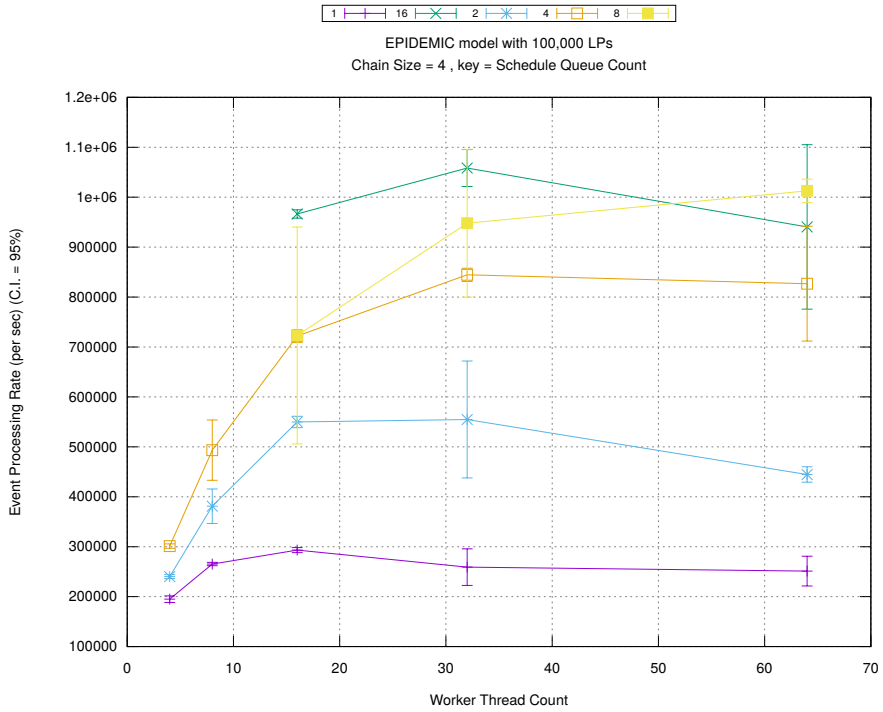
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

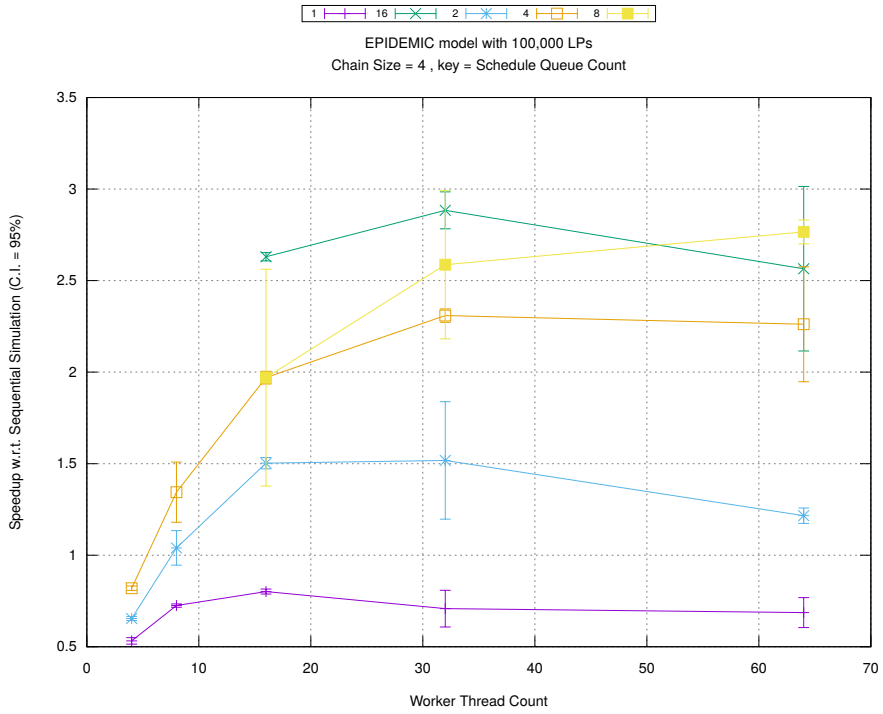


(b) Event Commitment Ratio

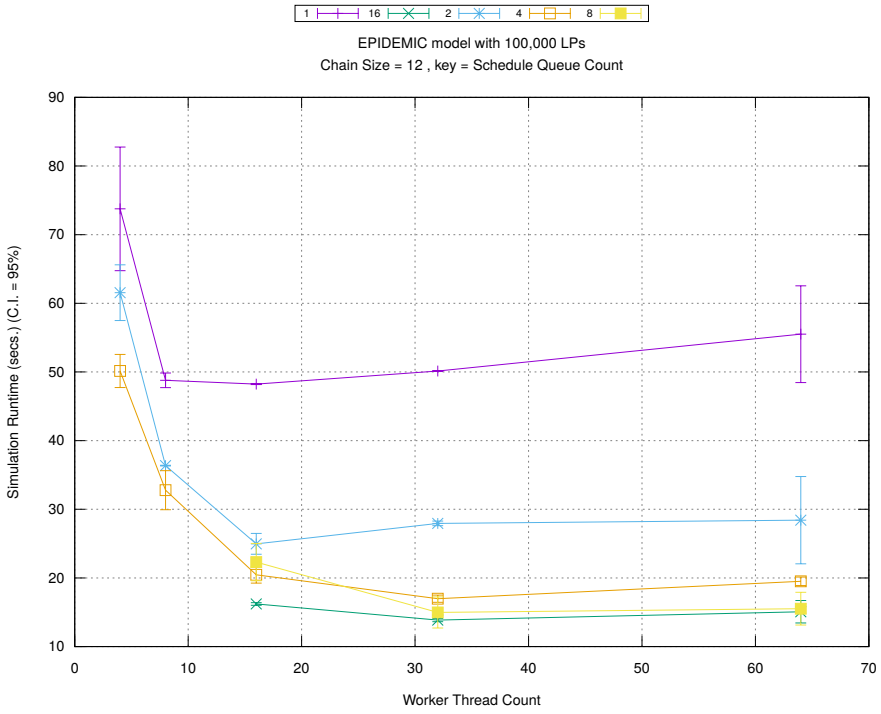


(c) Event Processing Rate (per second)

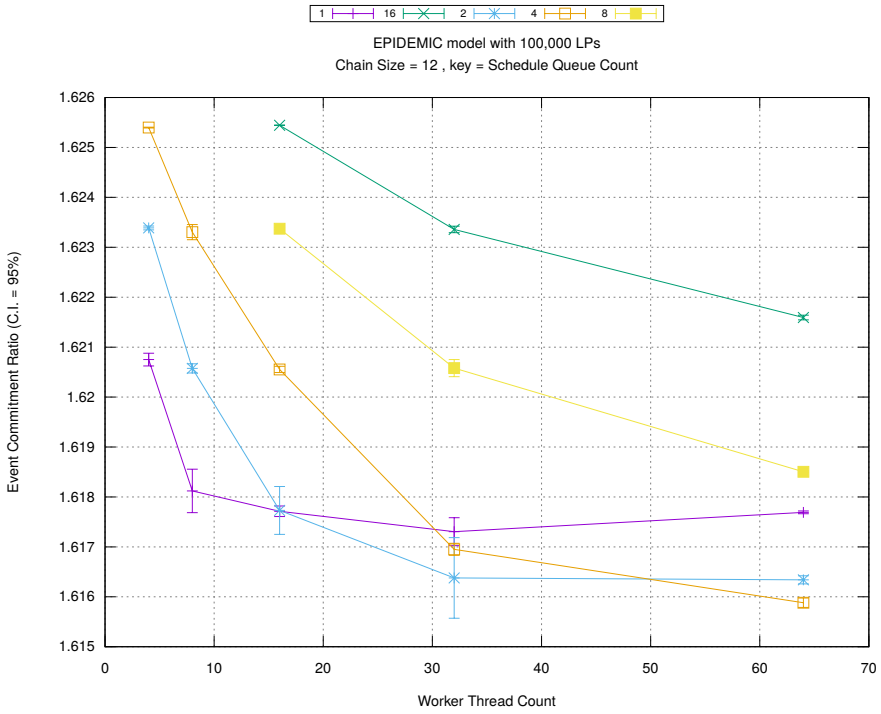
Figure A.105: epidemic 100k ws/plots/chains/threads vs count key chainsize 4



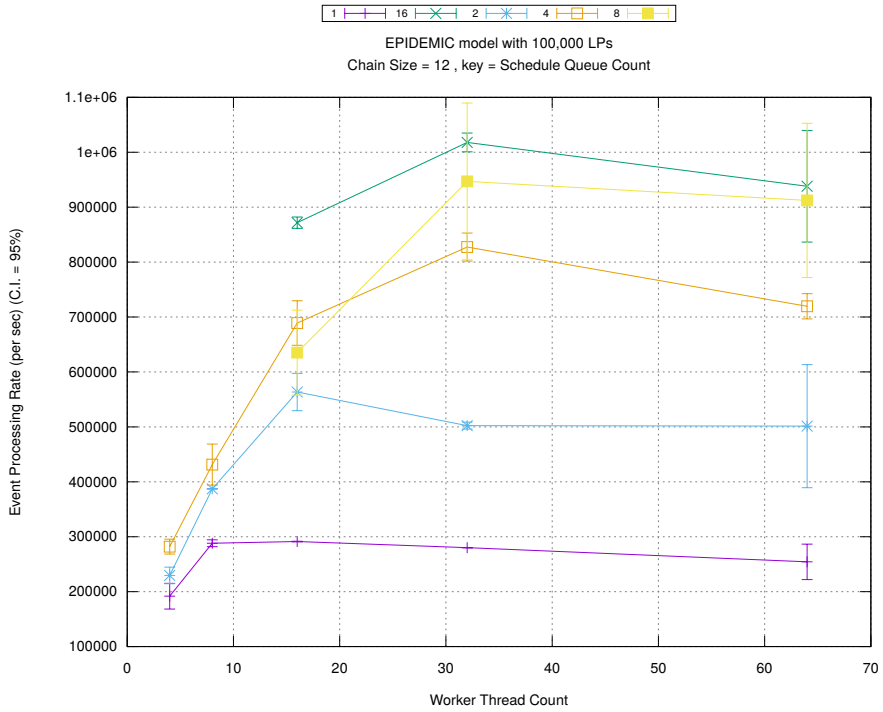
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

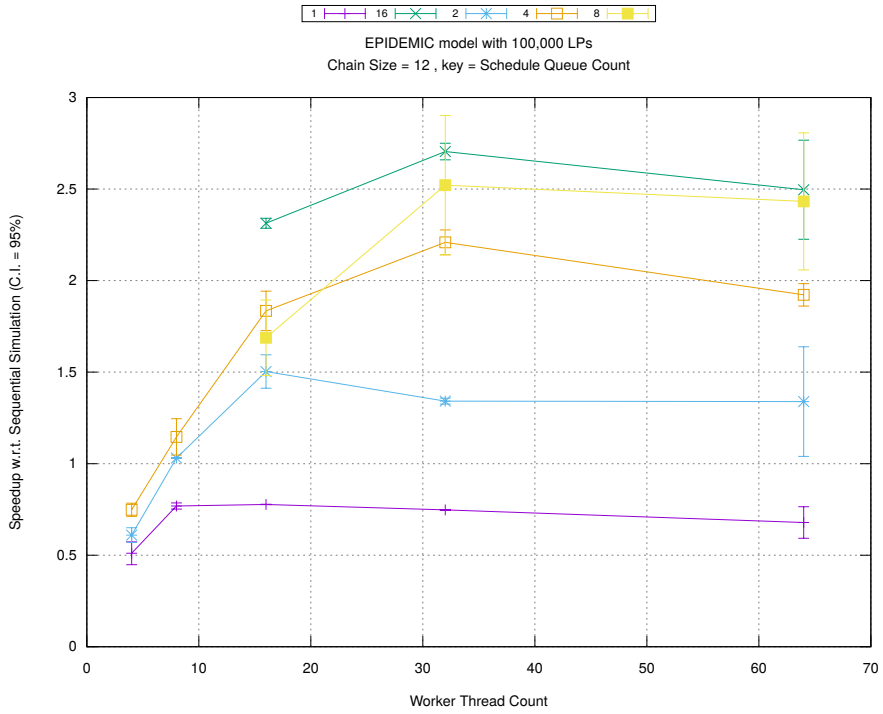


(b) Event Commitment Ratio

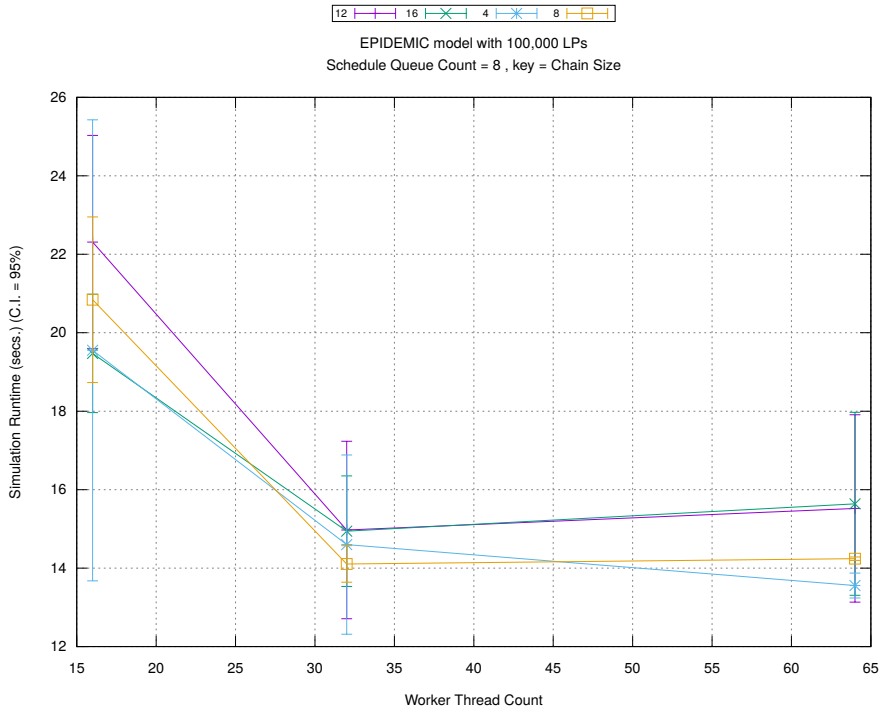


(c) Event Processing Rate (per second)

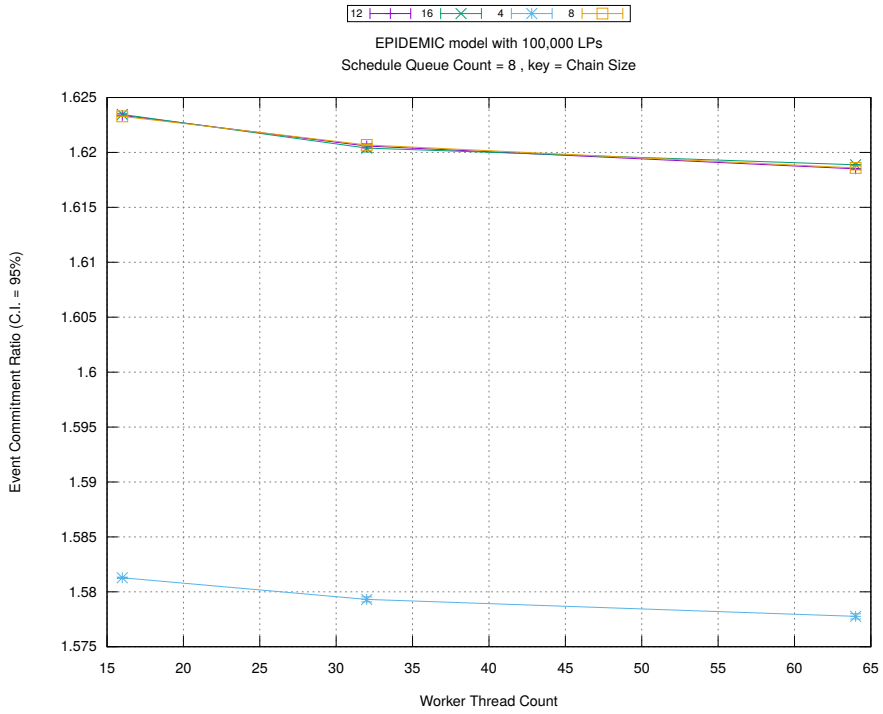
Figure A.106: epidemic 100k ws/plots/chains/threads vs count key chainsize 12



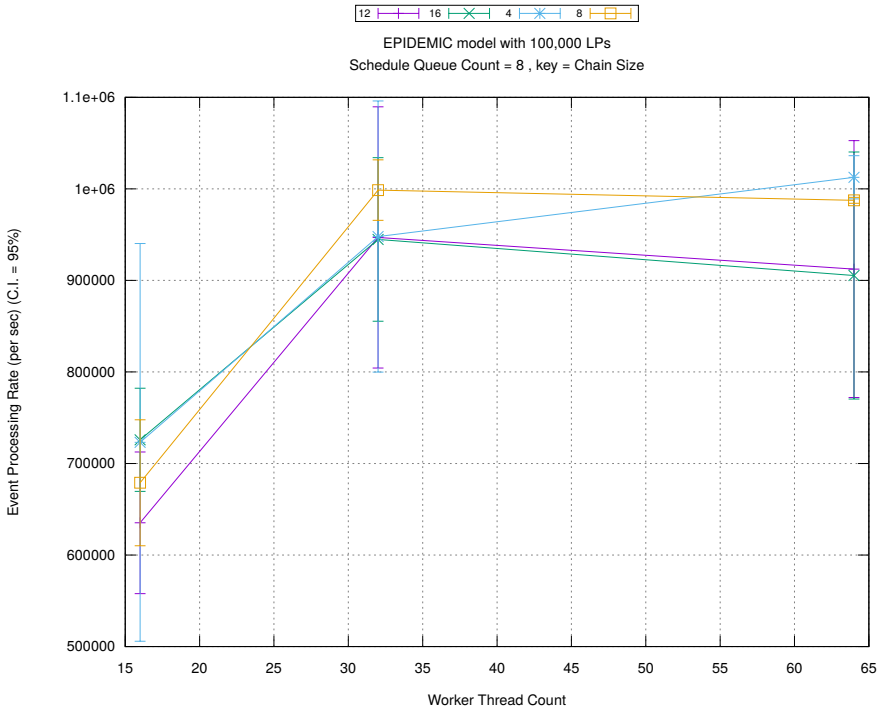
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

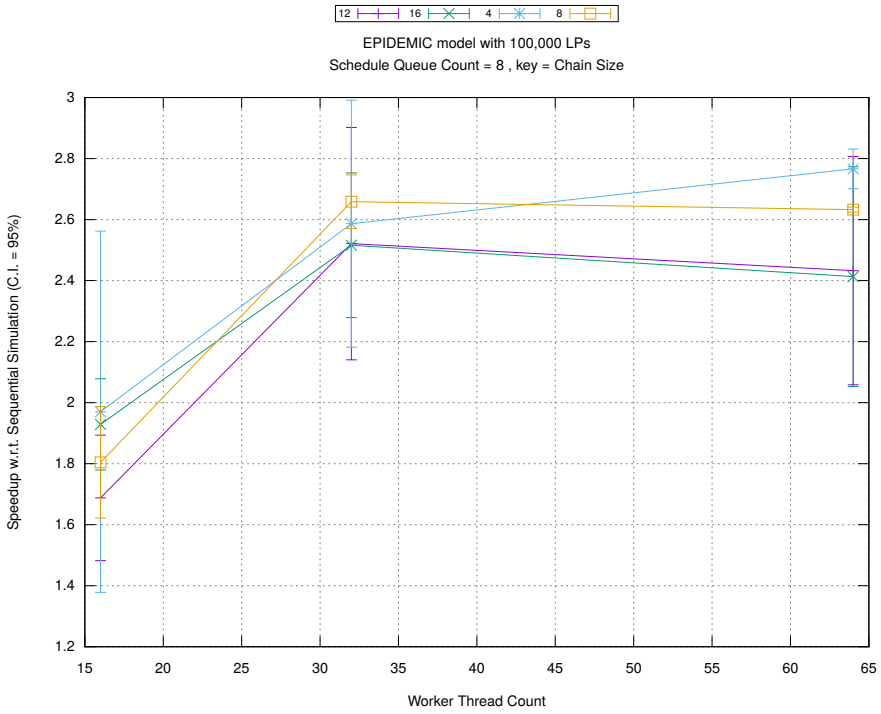


(b) Event Commitment Ratio

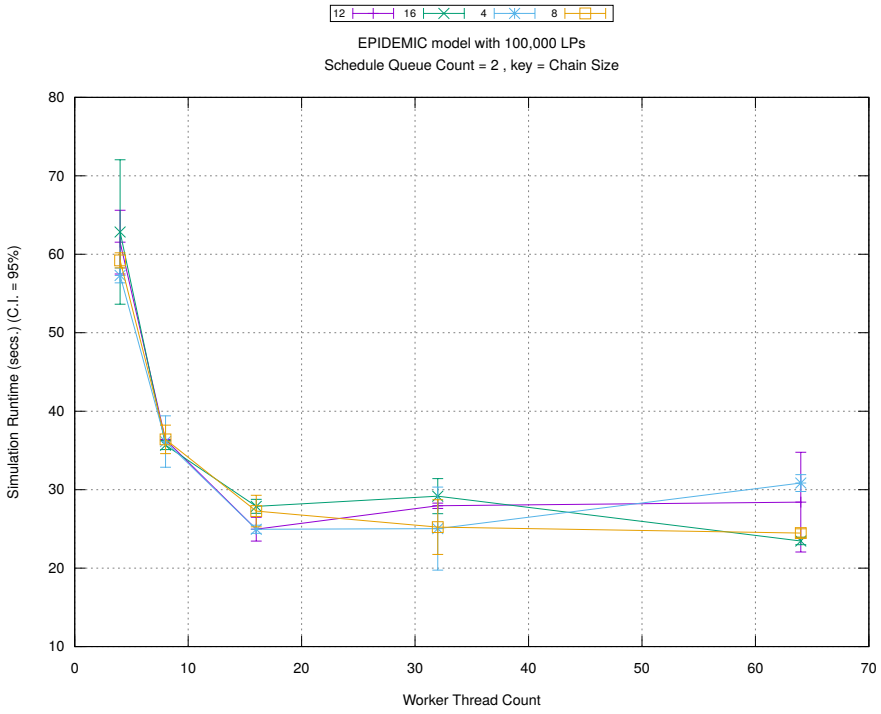


(c) Event Processing Rate (per second)

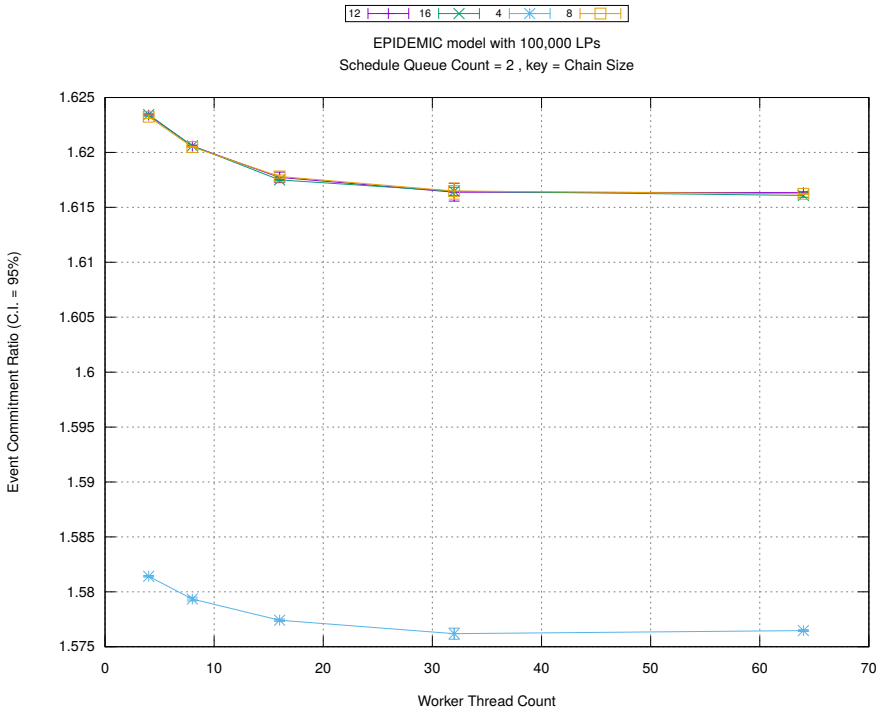
Figure A.107: epidemic 100k ws/plots/chains/threads vs chainsize key count 8



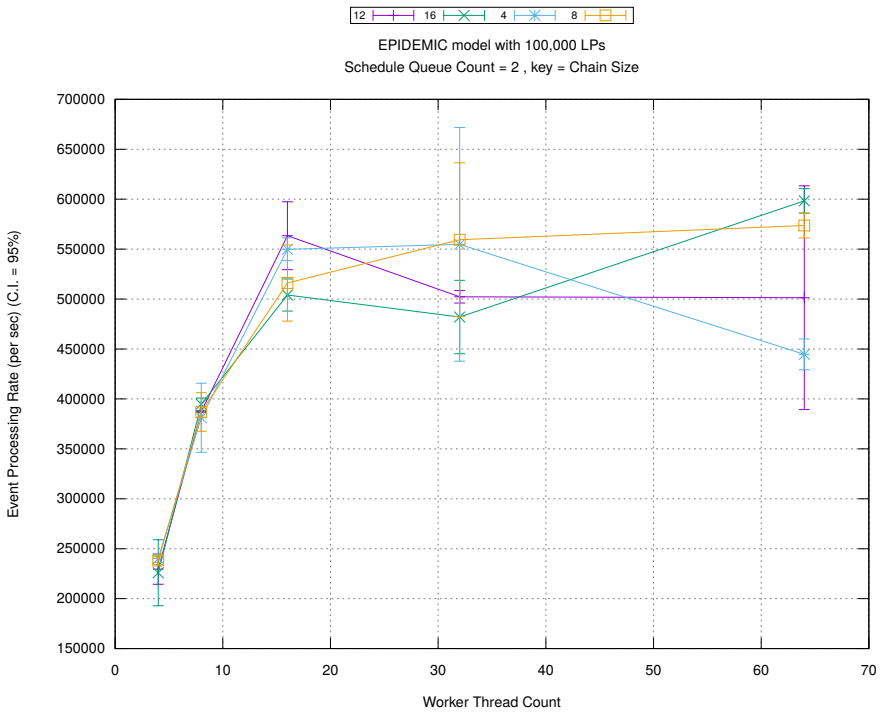
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

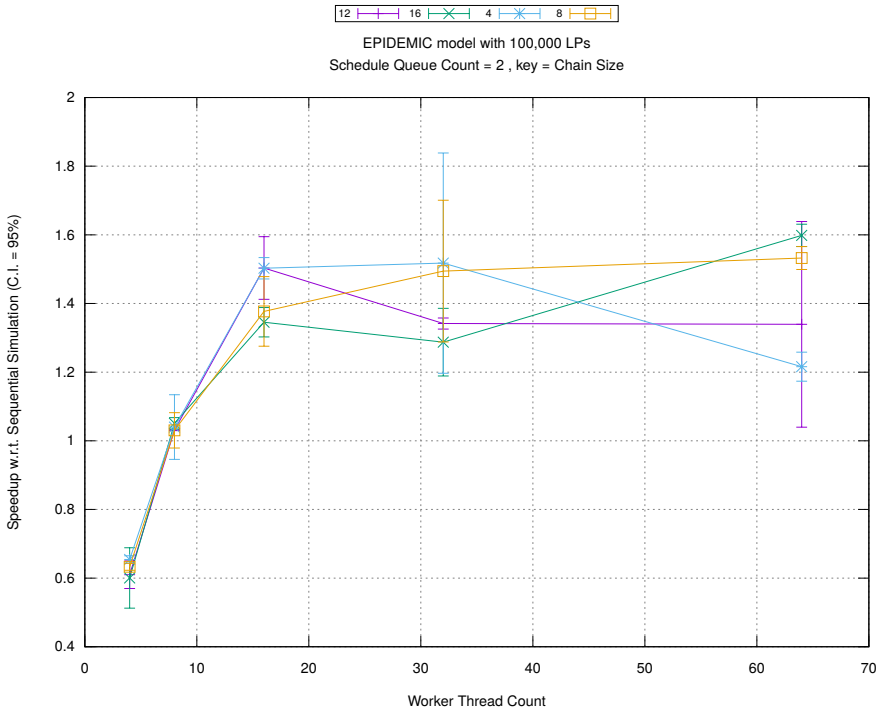


(b) Event Commitment Ratio

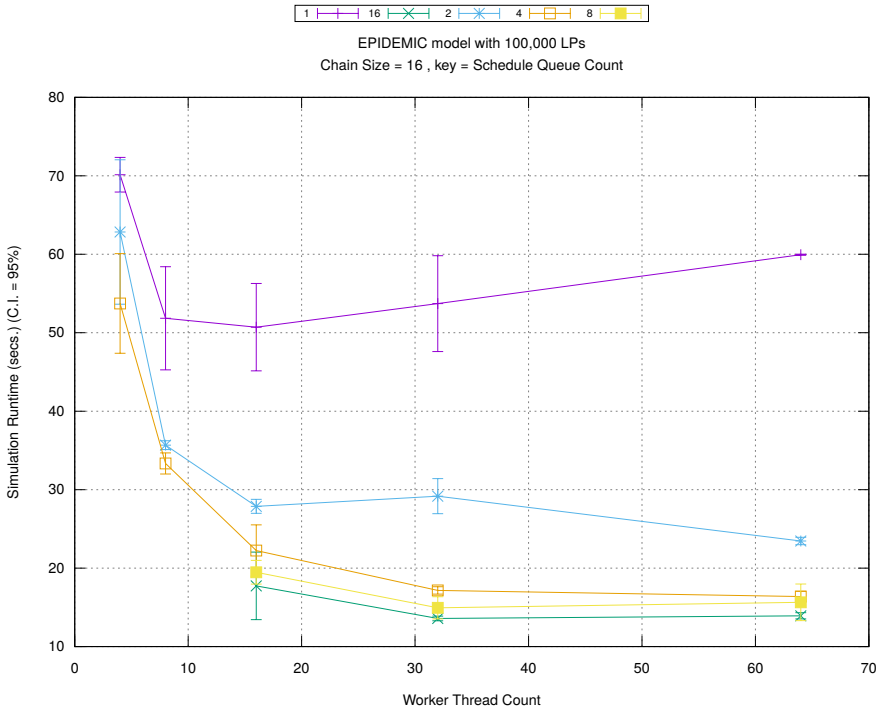


(c) Event Processing Rate (per second)

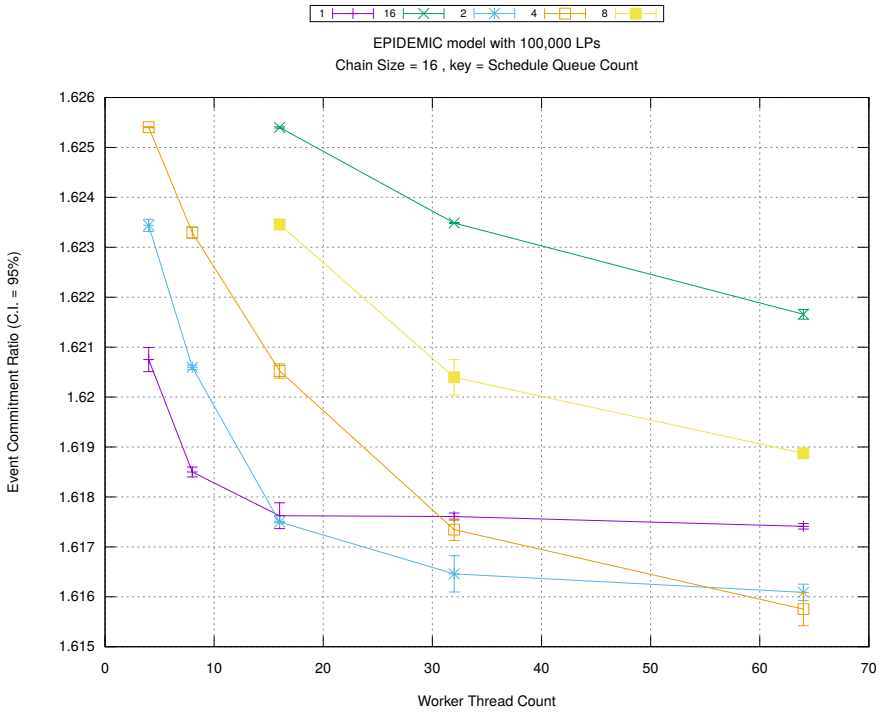
Figure A.108: epidemic 100k ws/plots/chains/threads vs chainsize key count 2



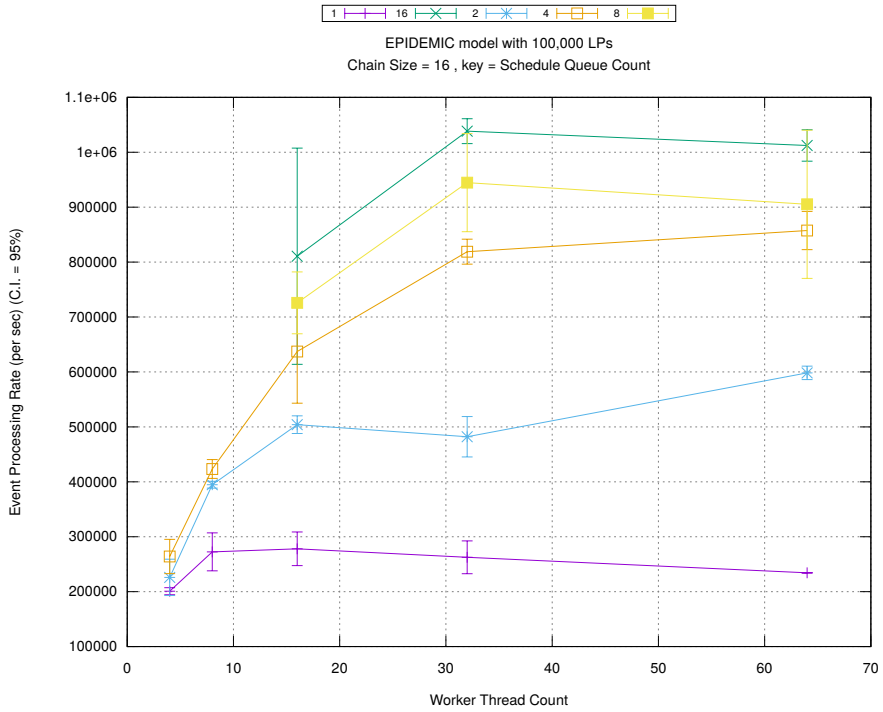
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

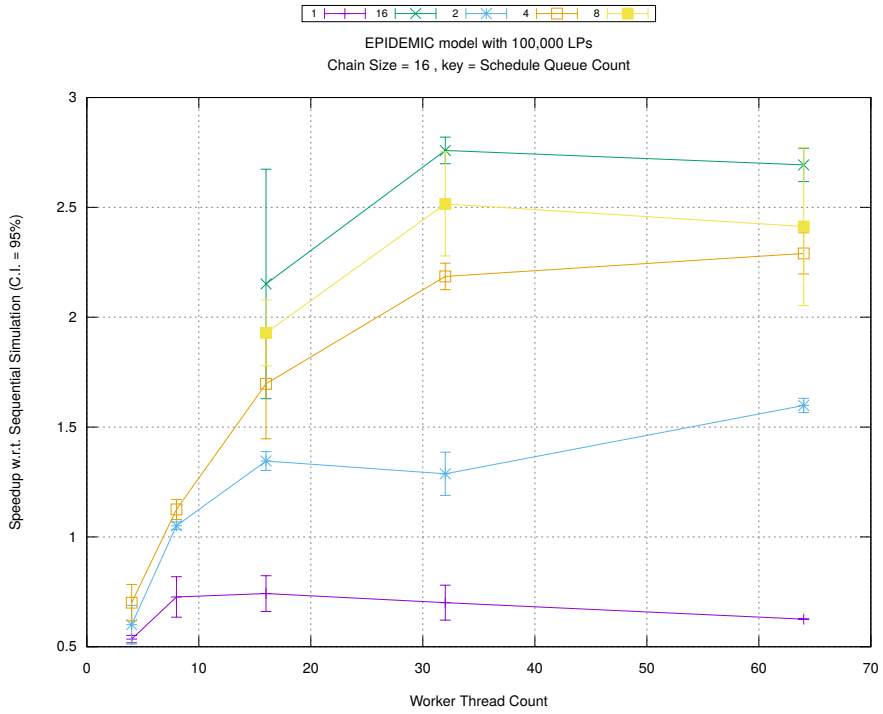


(b) Event Commitment Ratio

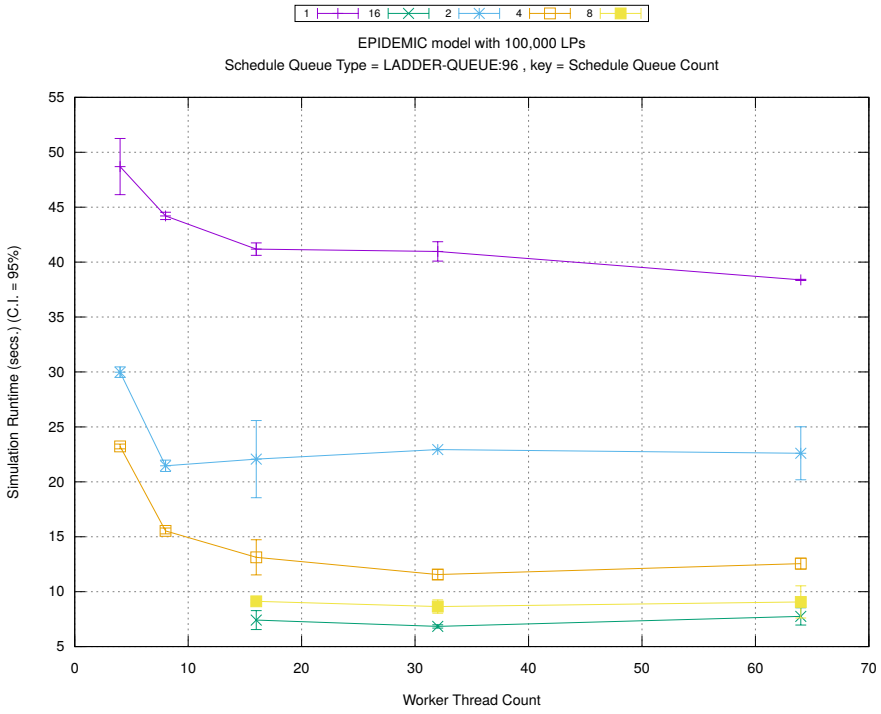


(c) Event Processing Rate (per second)

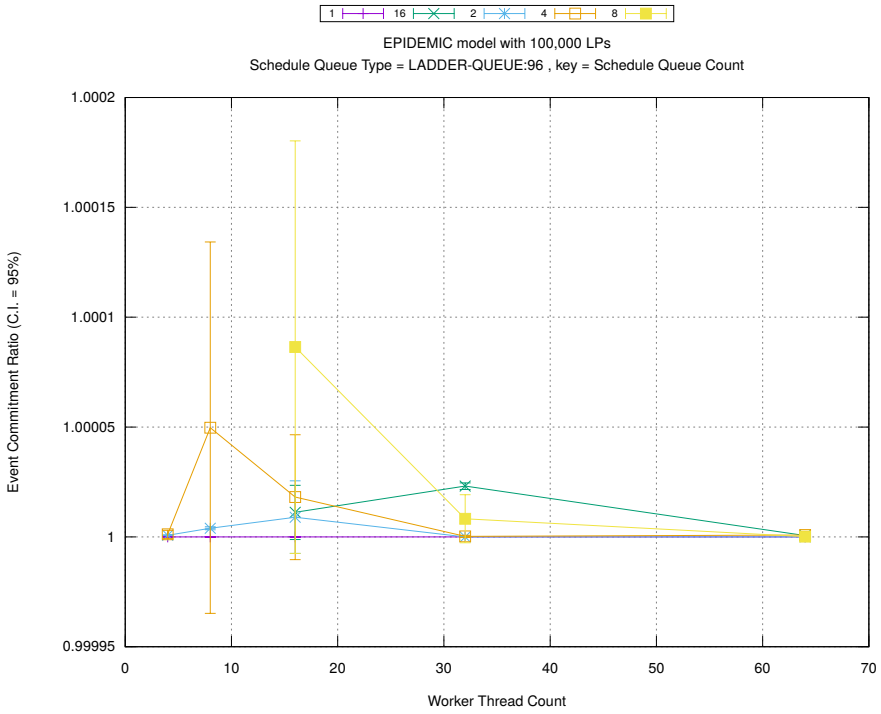
Figure A.109: epidemic 100k ws/plots/chains/threads vs count key chainsize 16



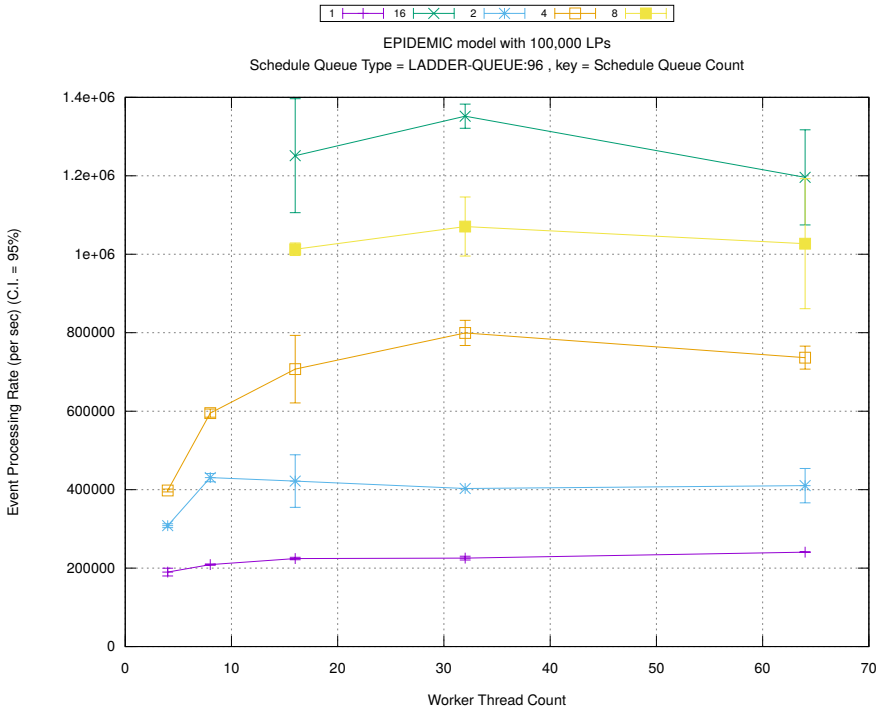
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

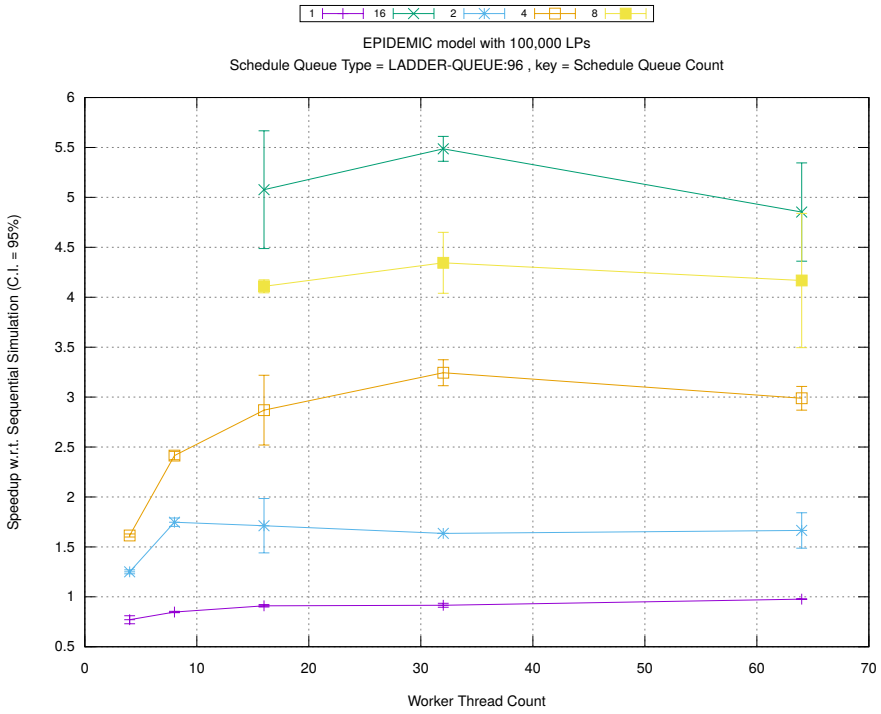


(b) Event Commitment Ratio

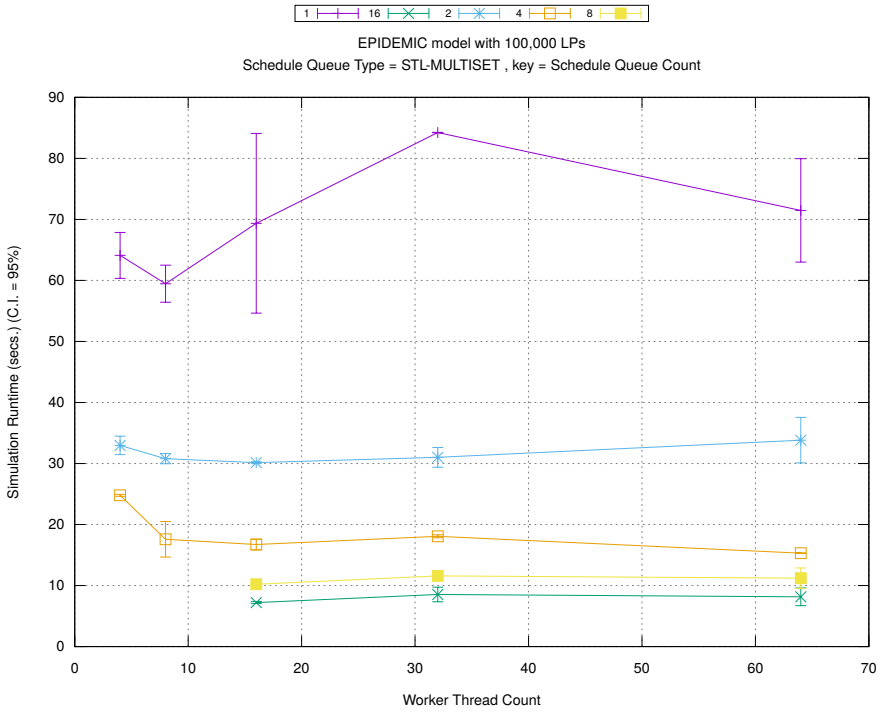


(c) Event Processing Rate (per second)

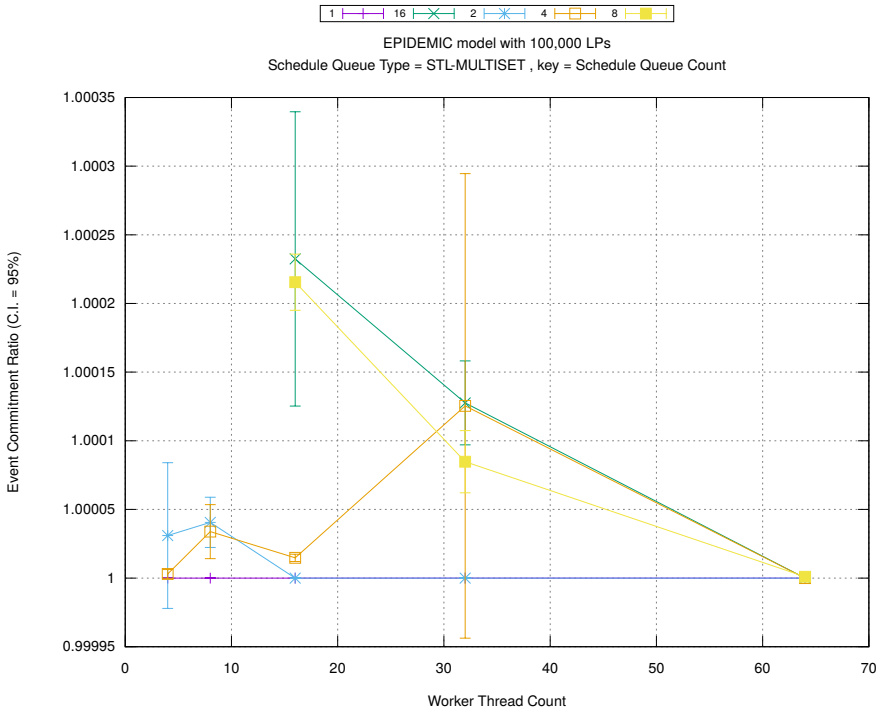
Figure A.110: epidemic 100k ws/plots/scheduleq/threads vs count key type ladder-queue 96



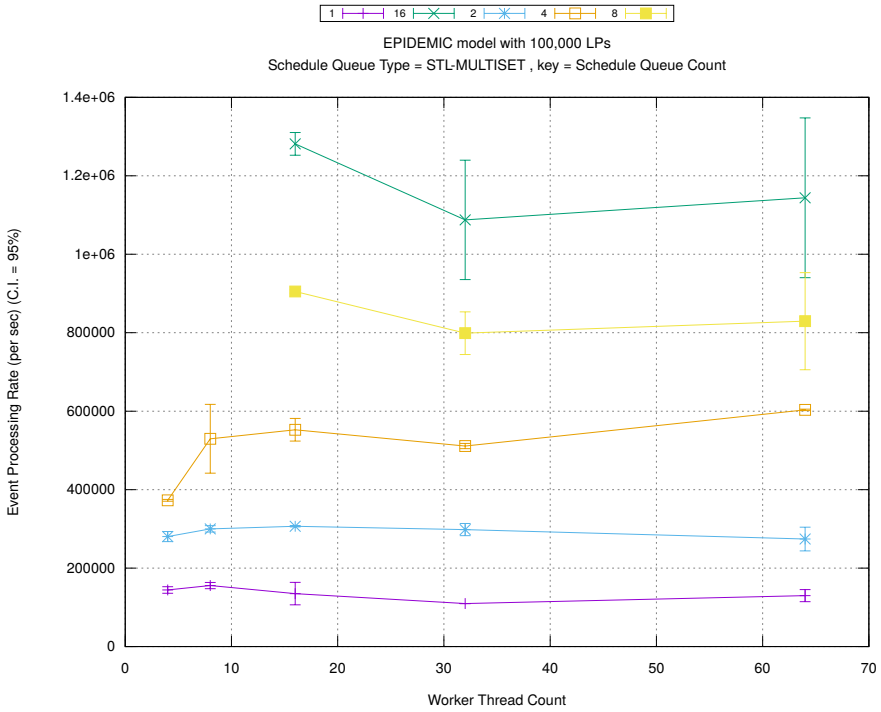
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

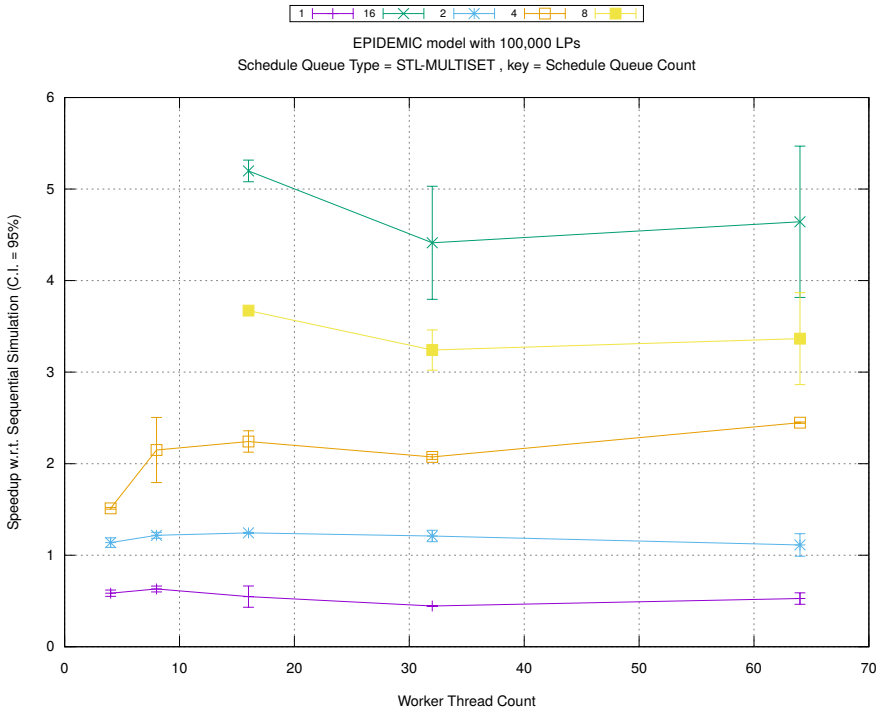


(b) Event Commitment Ratio

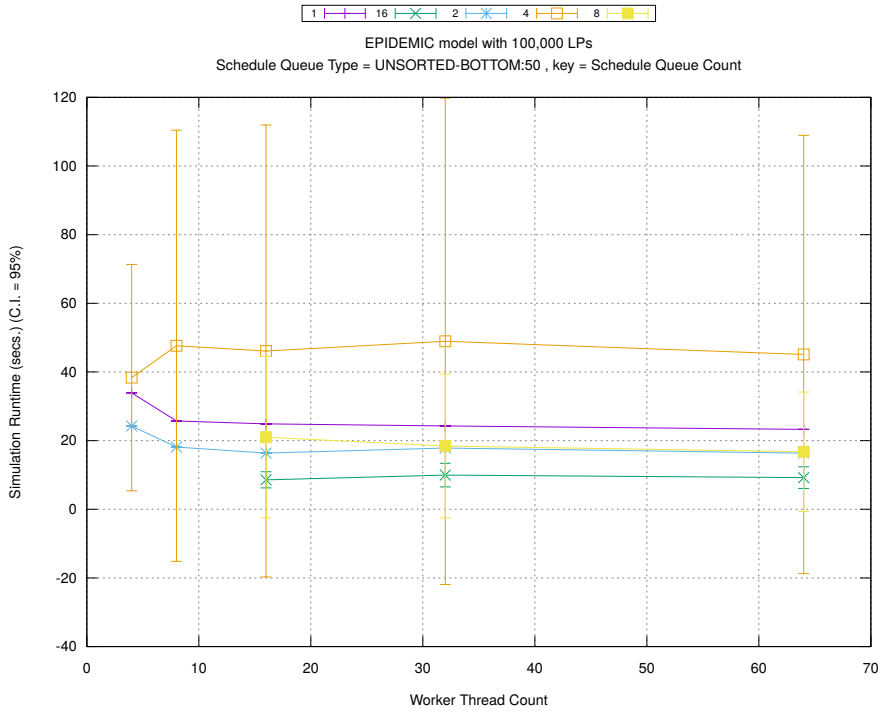


(c) Event Processing Rate (per second)

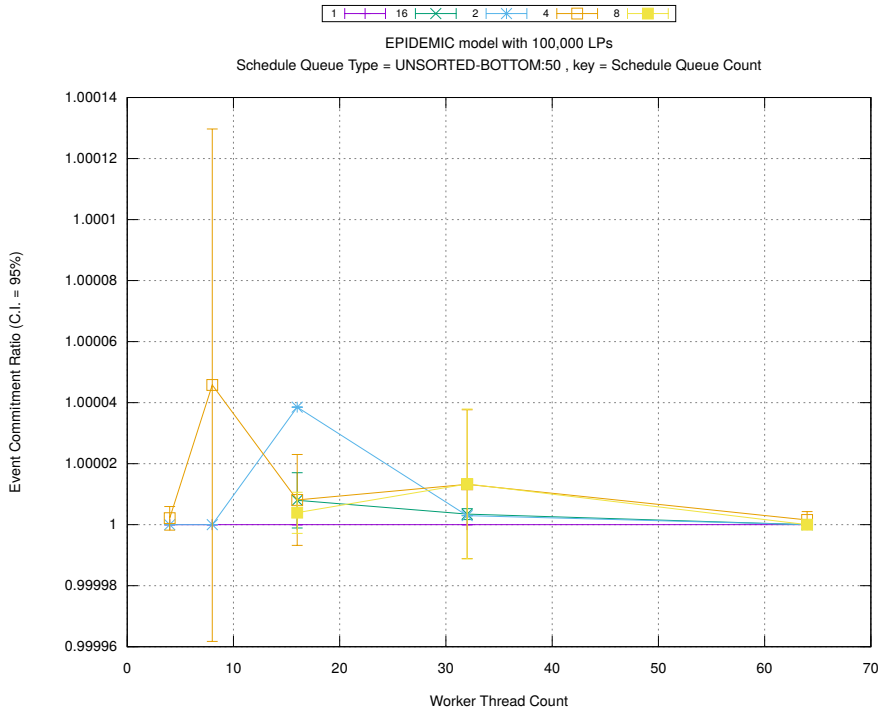
Figure A.111: epidemic 100k ws/plots/scheduleq/threads vs count key type stl-multiset



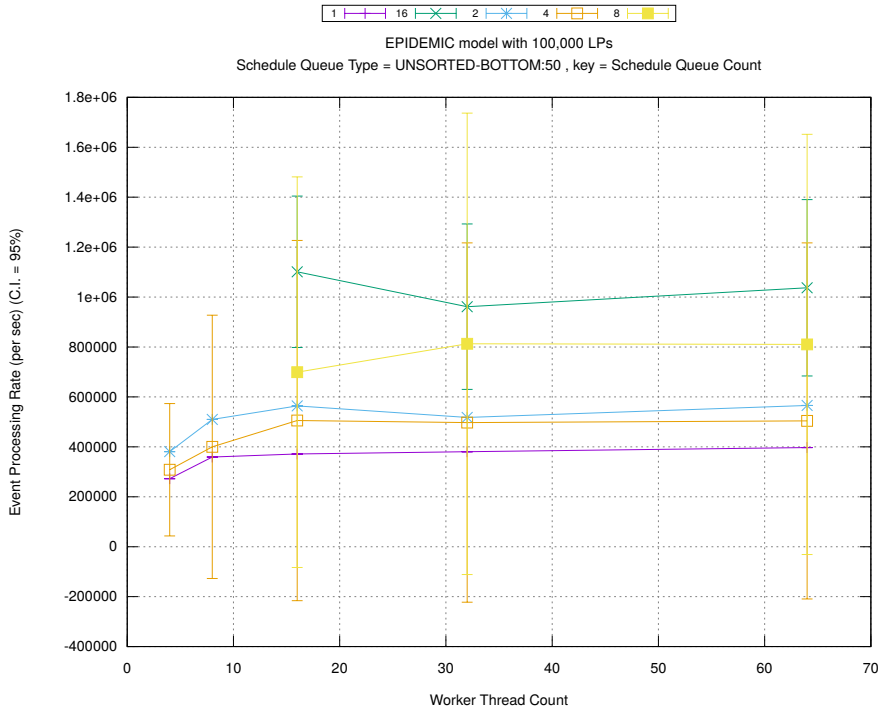
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

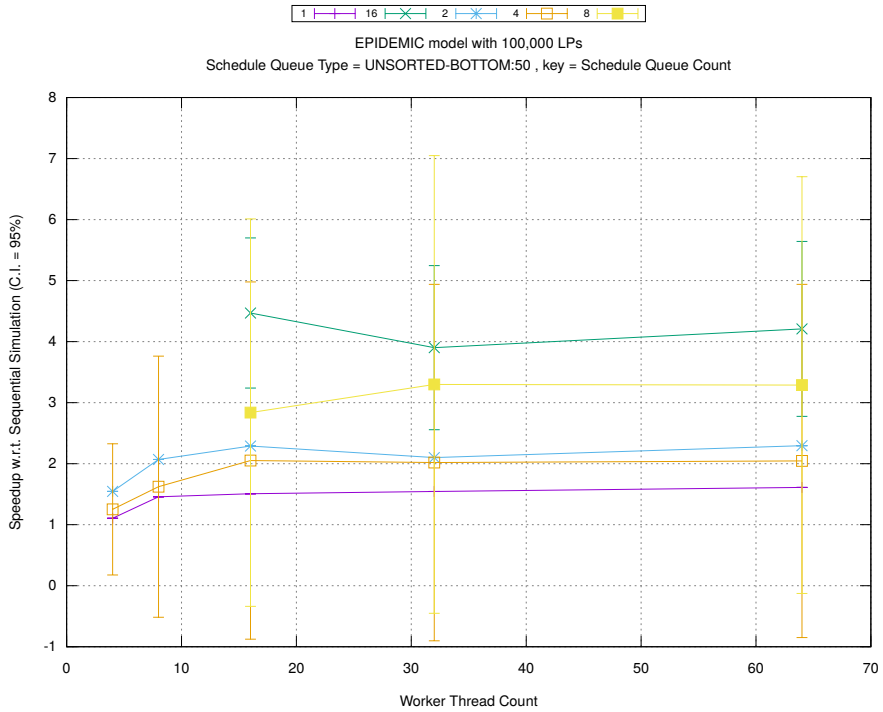


(b) Event Commitment Ratio

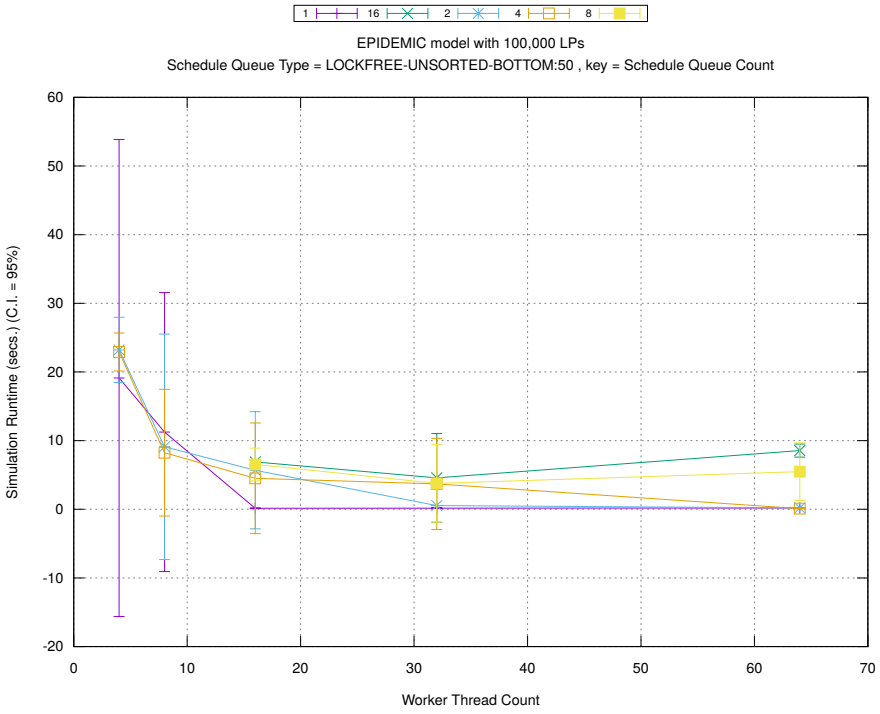


(c) Event Processing Rate (per second)

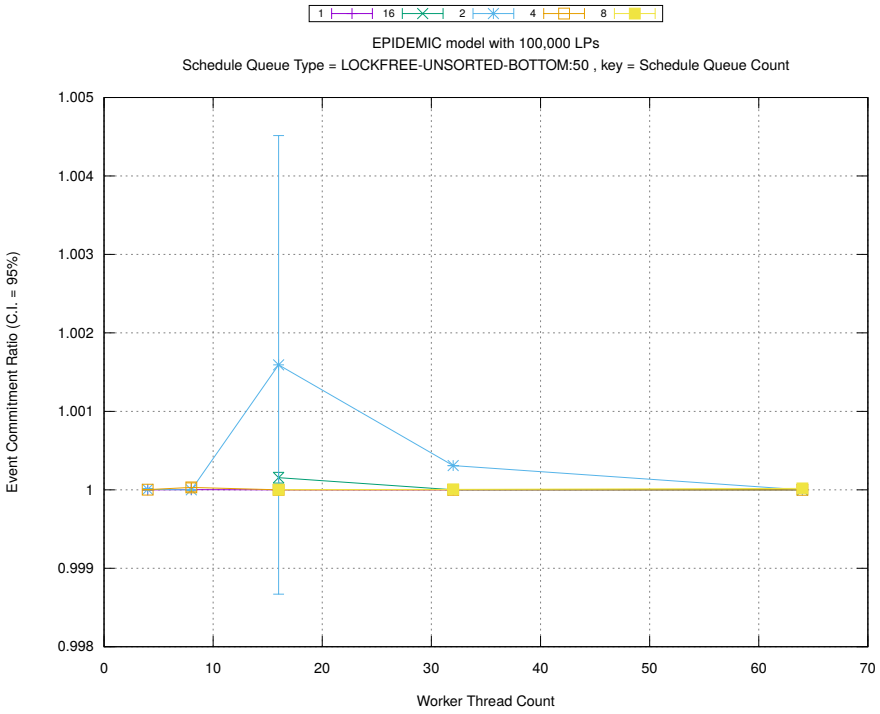
Figure A.112: epidemic 100k ws/plots/scheduleq/threads vs count key type unsorted-bottom 50



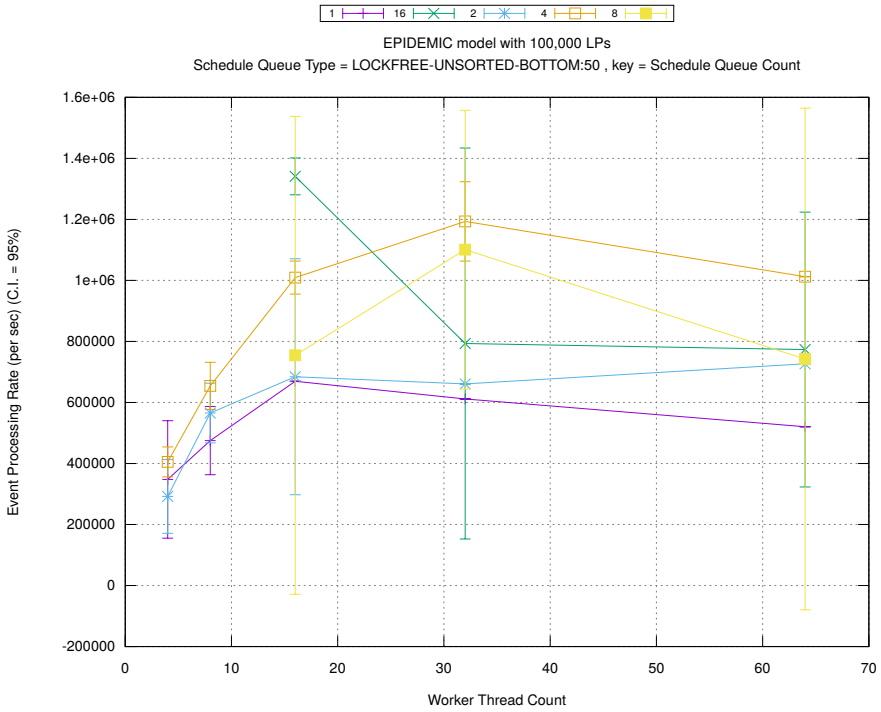
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

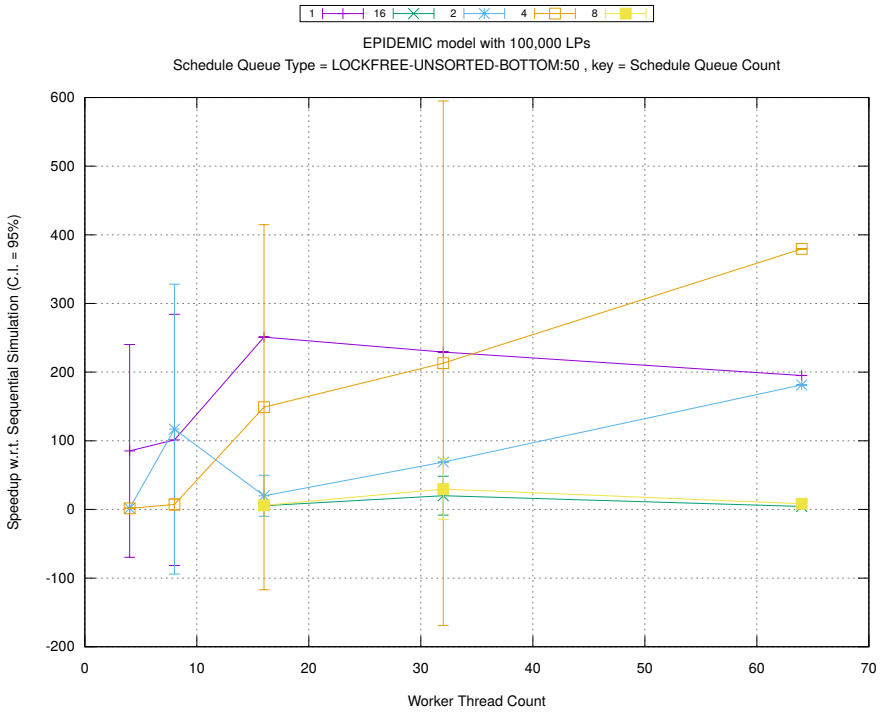


(b) Event Commitment Ratio

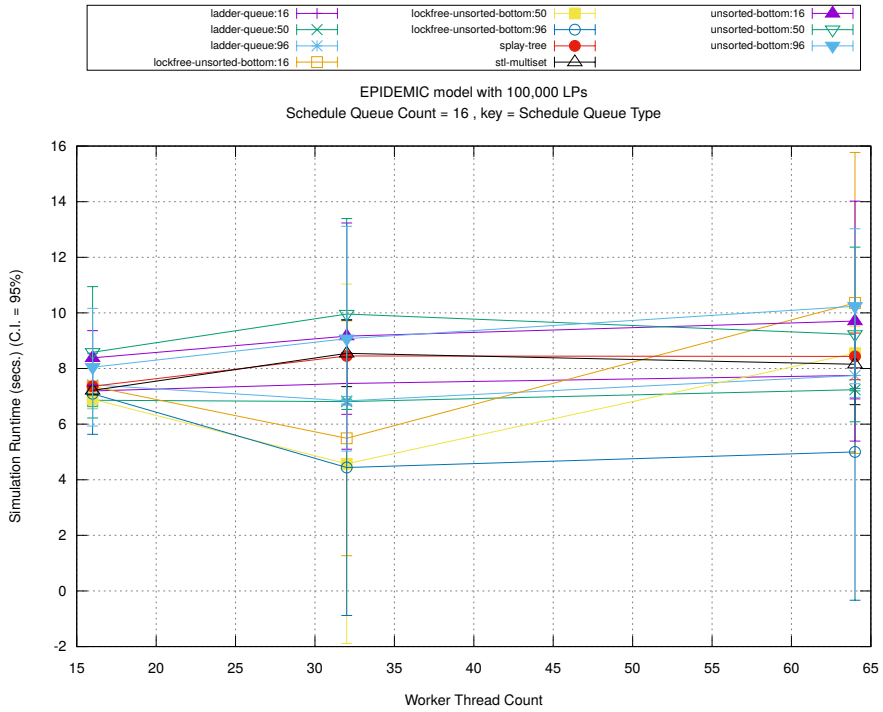


(c) Event Processing Rate (per second)

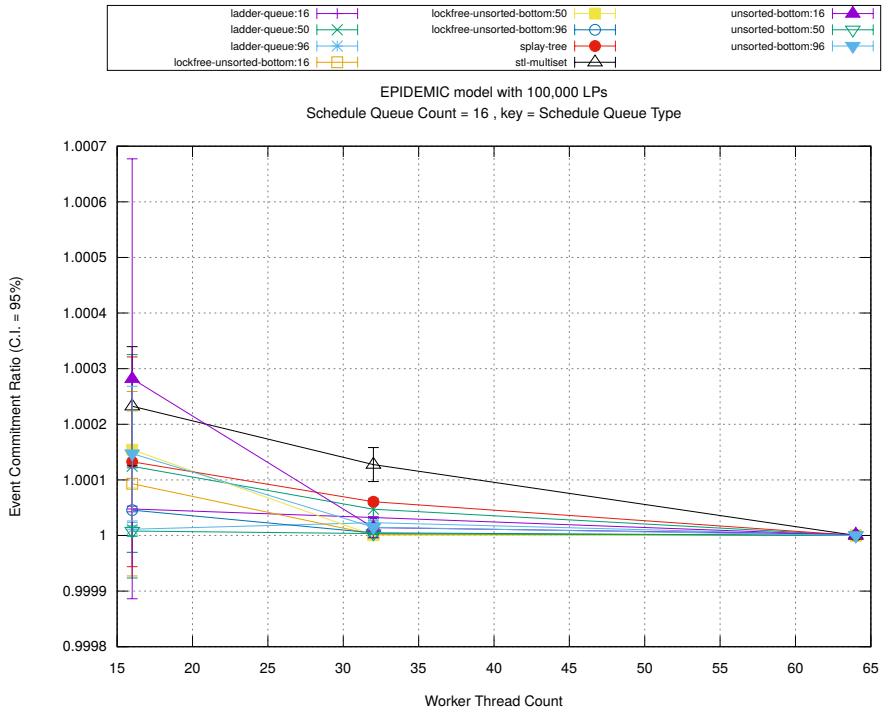
Figure A.113: epidemic 100k ws/plots/scheduleq/threads vs count key type lockfree-unsorted-bottom 50



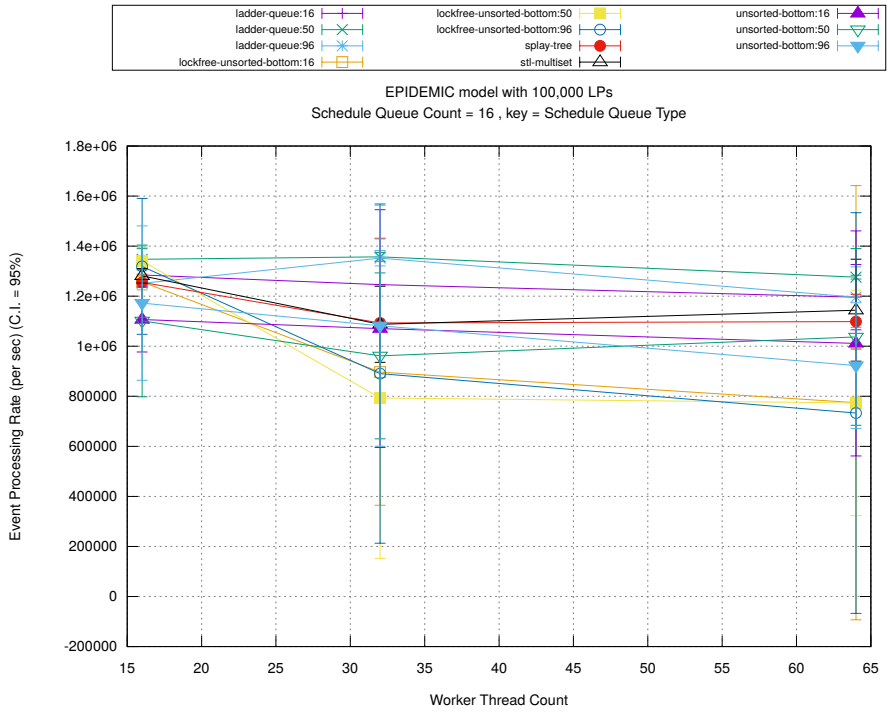
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

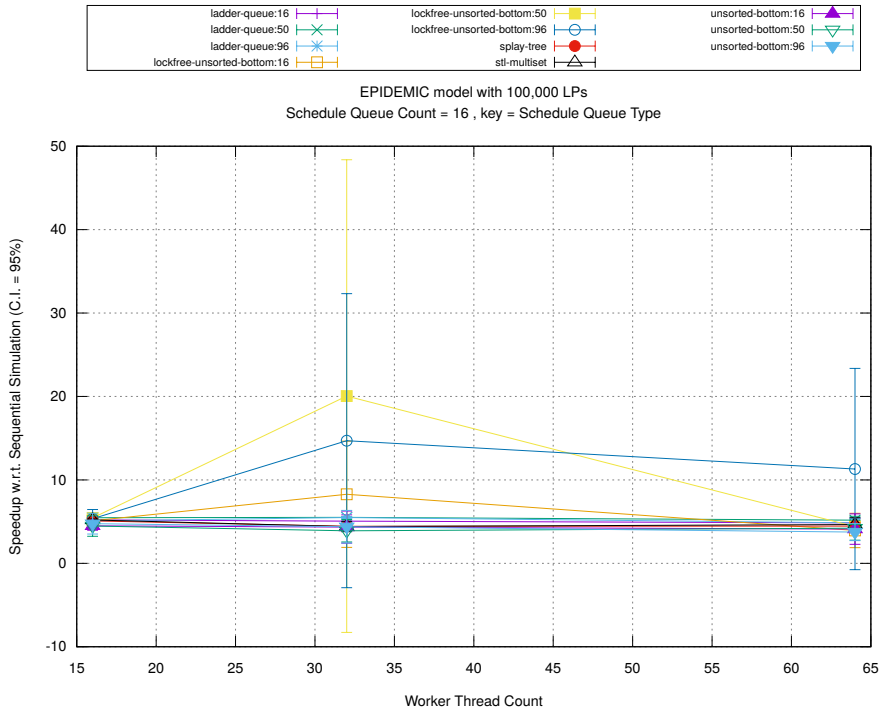


(b) Event Commitment Ratio

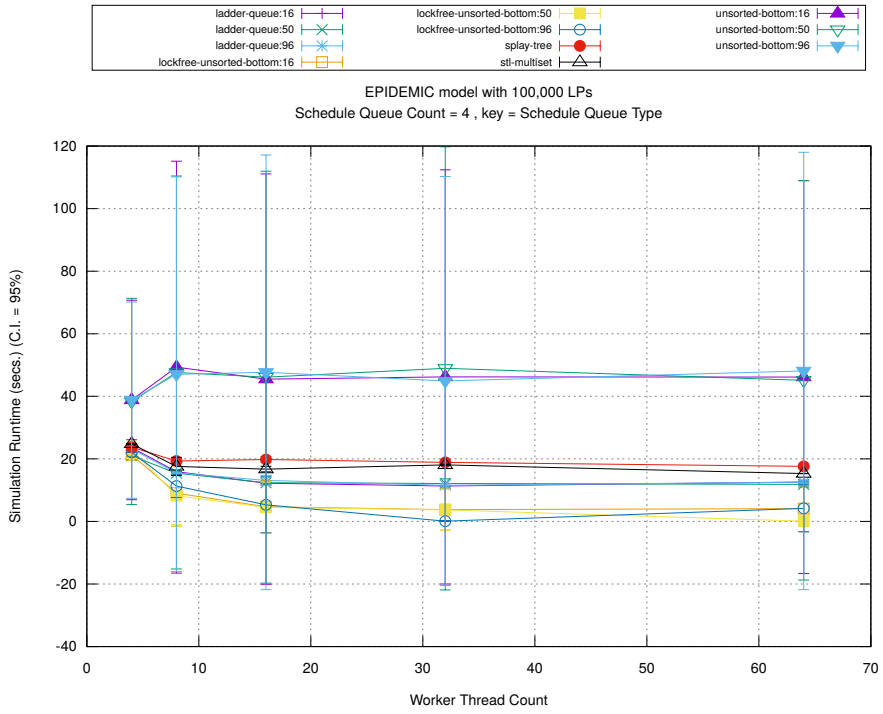


(c) Event Processing Rate (per second)

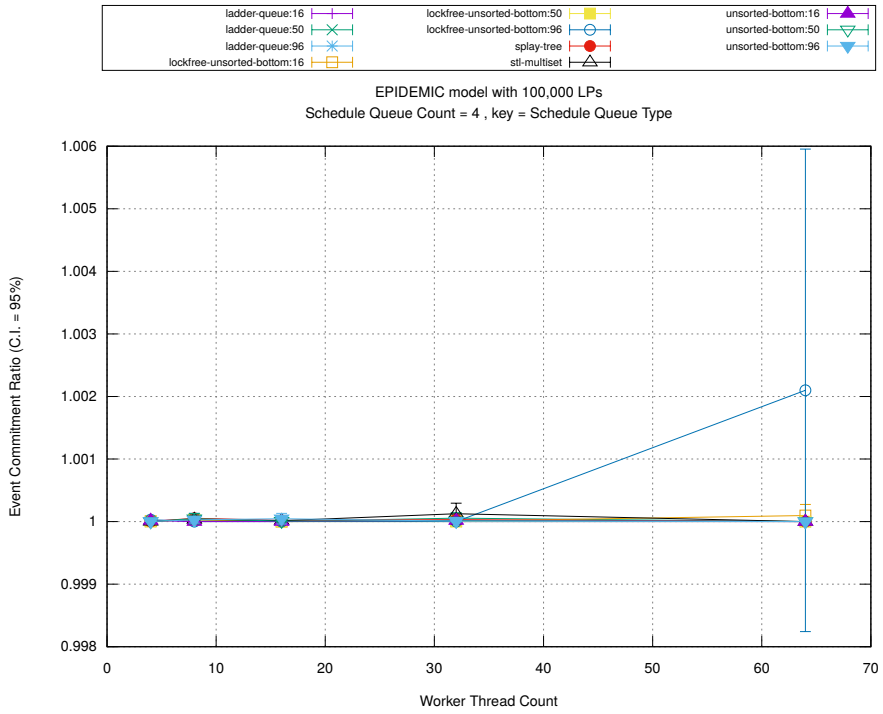
Figure A.114: epidemic 100k ws/plots/scheduleq/threads vs type key count 16



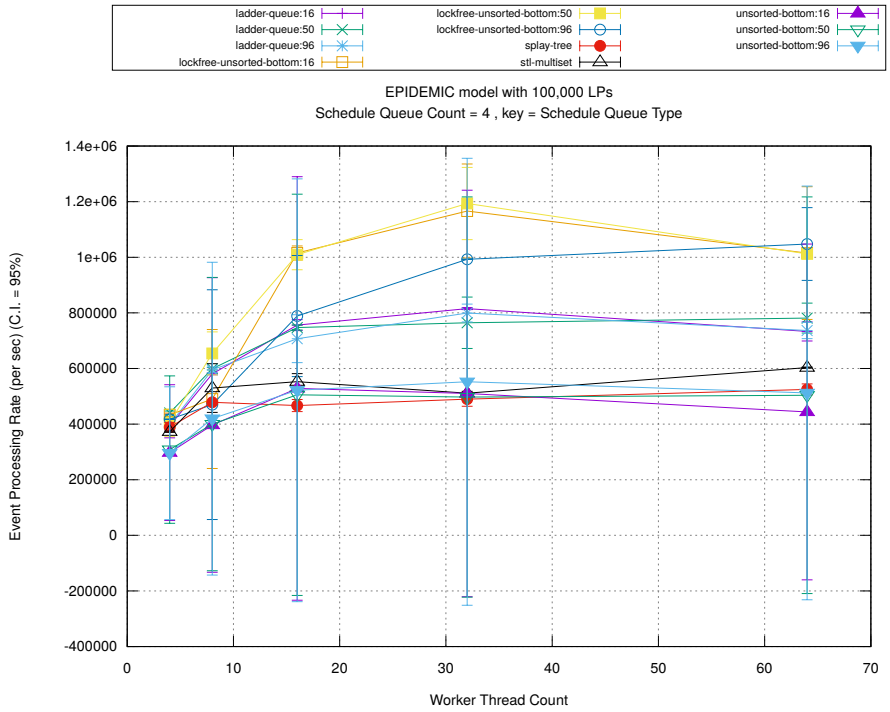
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

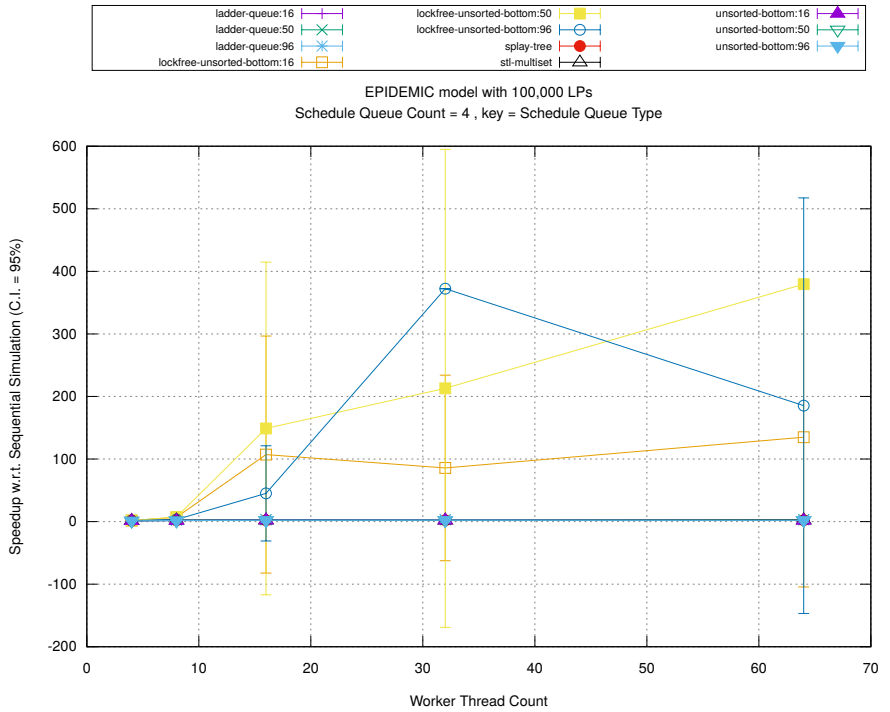


(b) Event Commitment Ratio

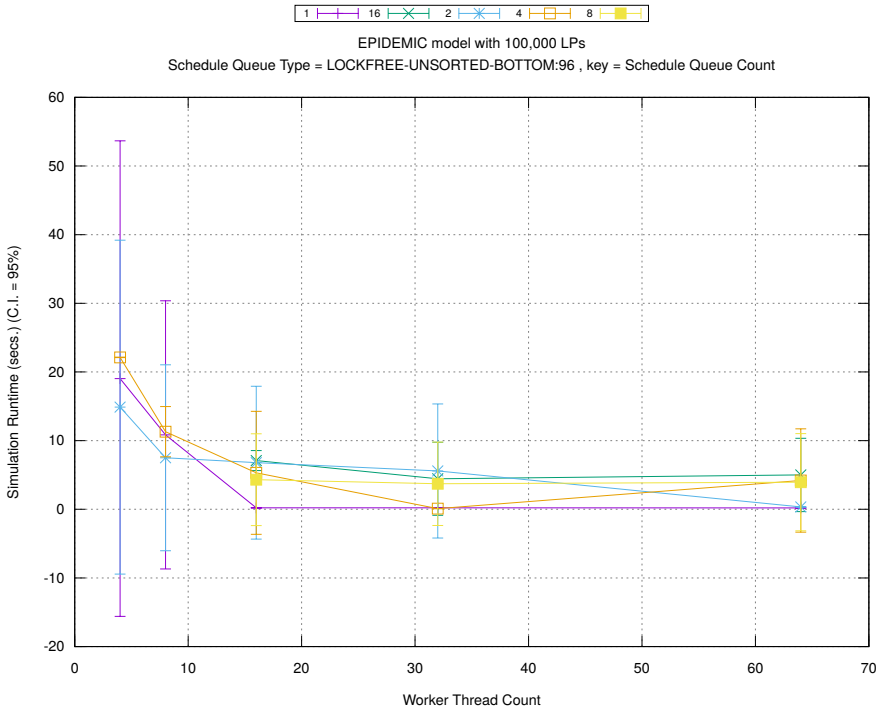


(c) Event Processing Rate (per second)

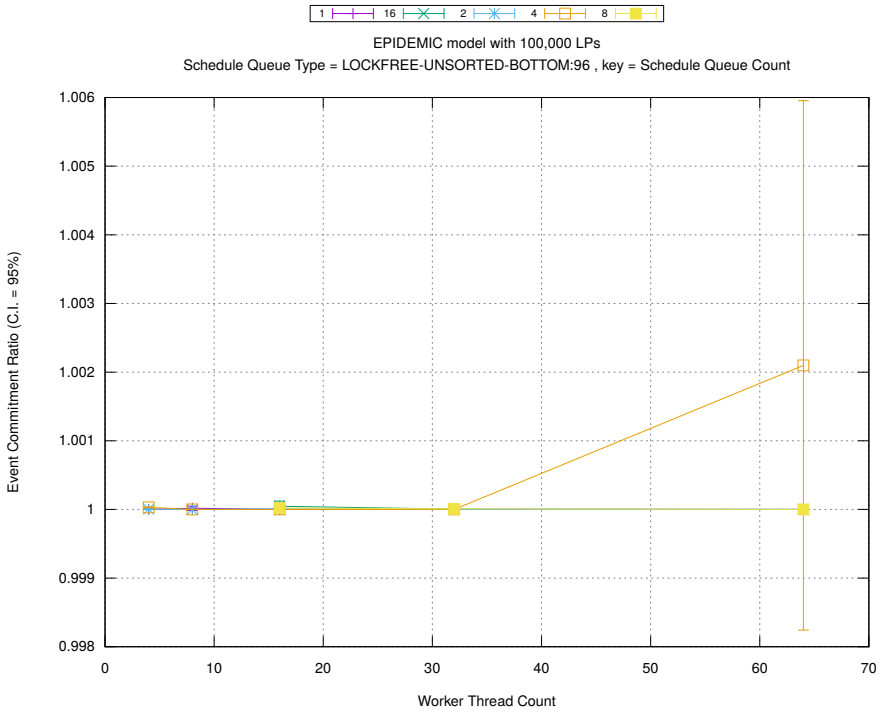
Figure A.115: epidemic 100k ws/plots/scheduleq/threads vs type key count 4



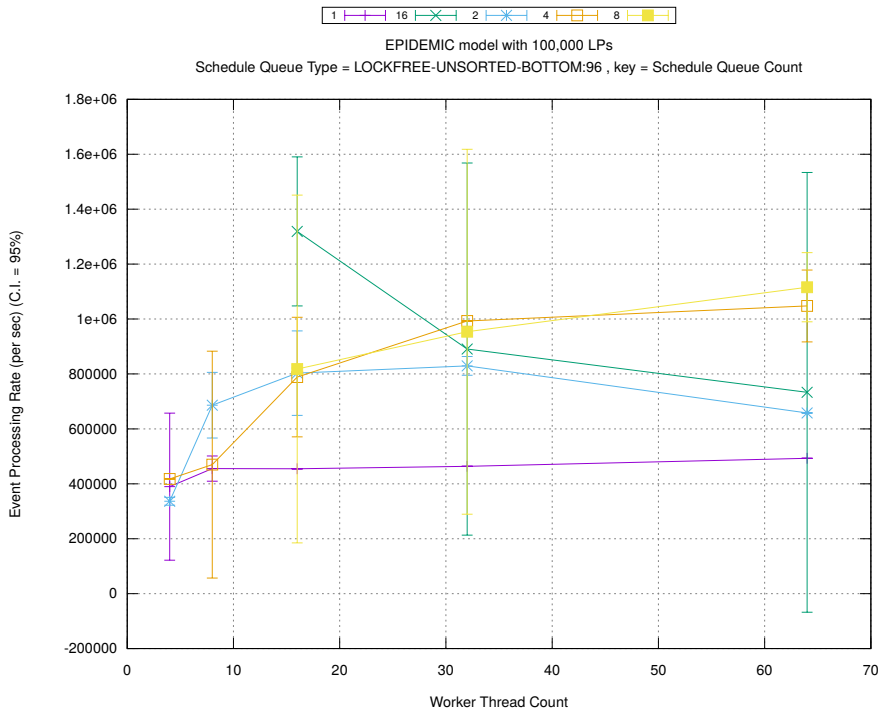
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

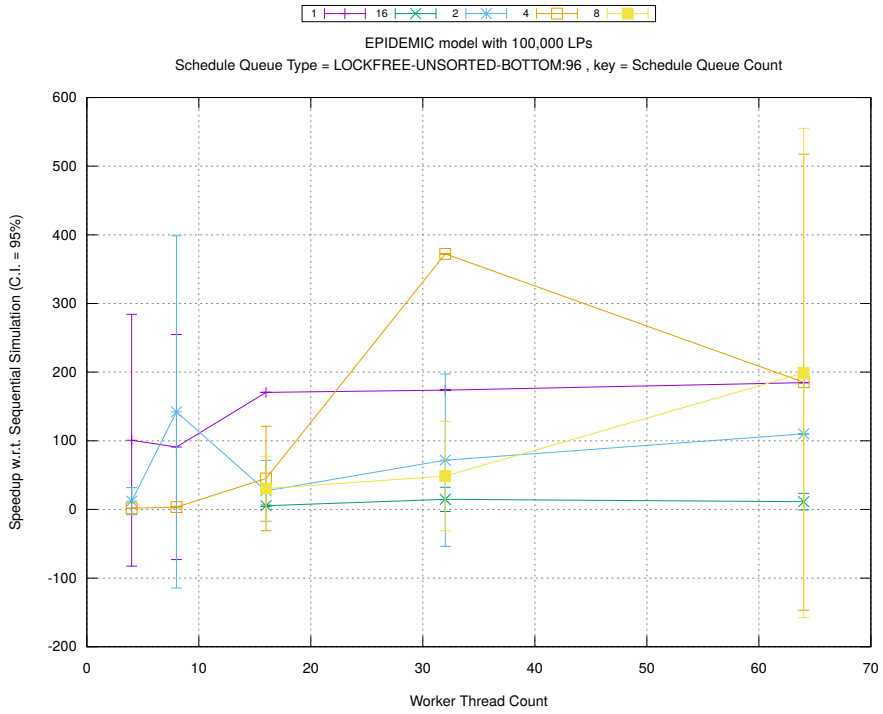


(b) Event Commitment Ratio

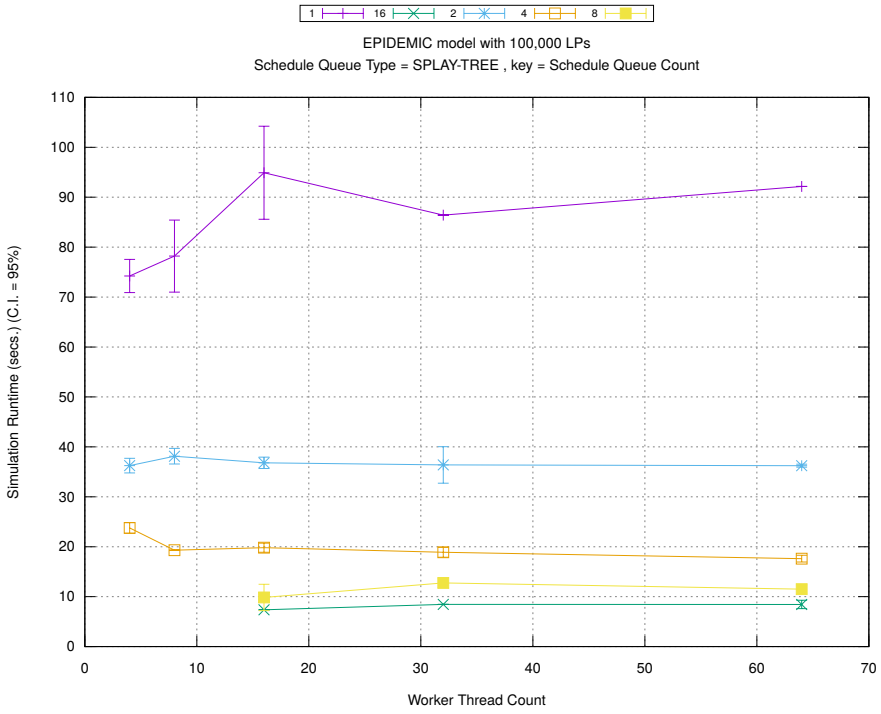


(c) Event Processing Rate (per second)

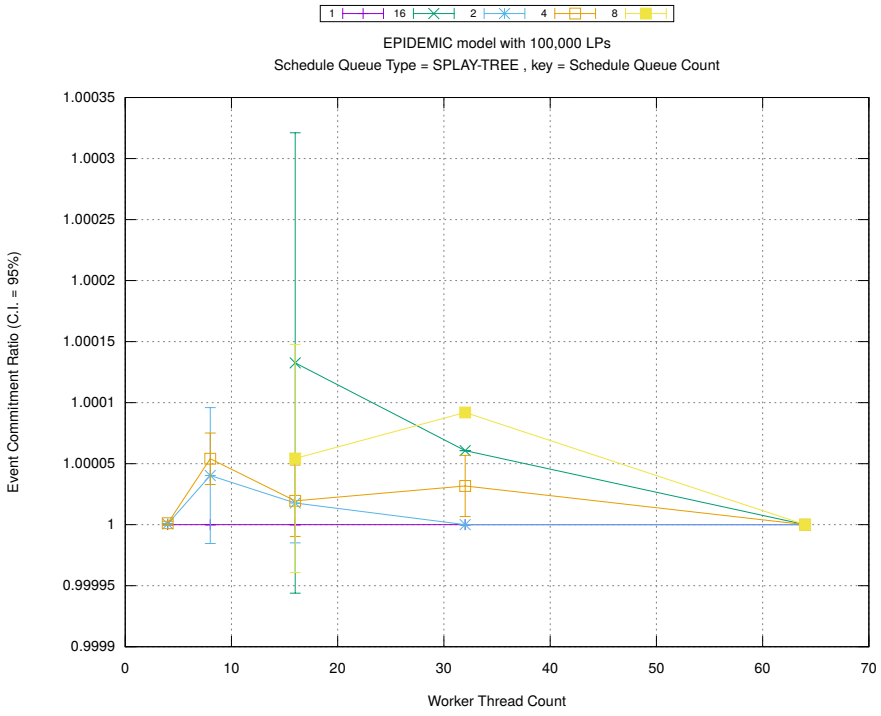
Figure A.116: epidemic 100k ws/plots/scheduleq/threads vs count key type lockfree-unsorted-bottom 96



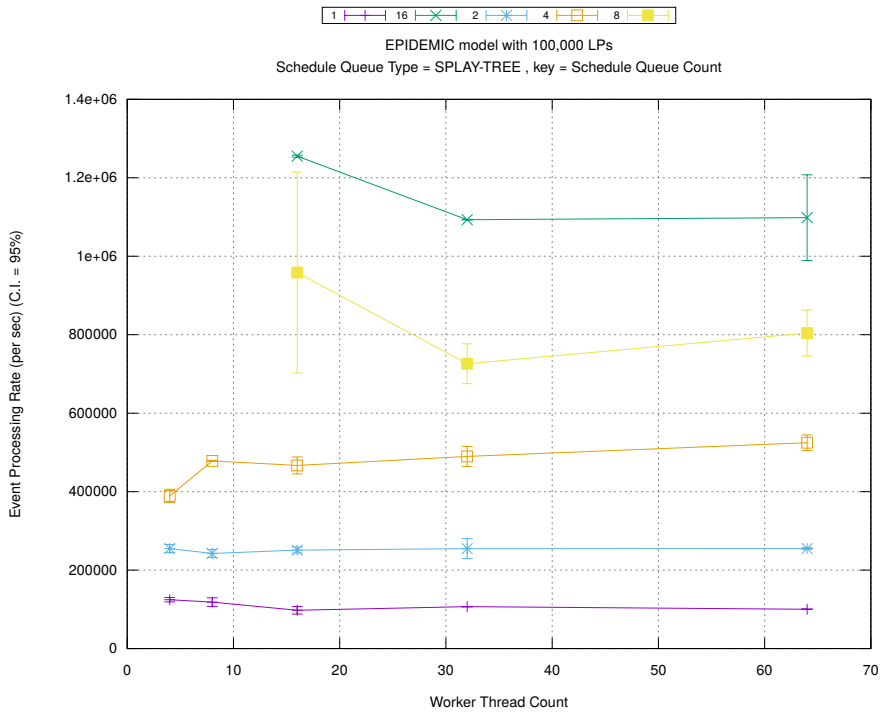
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

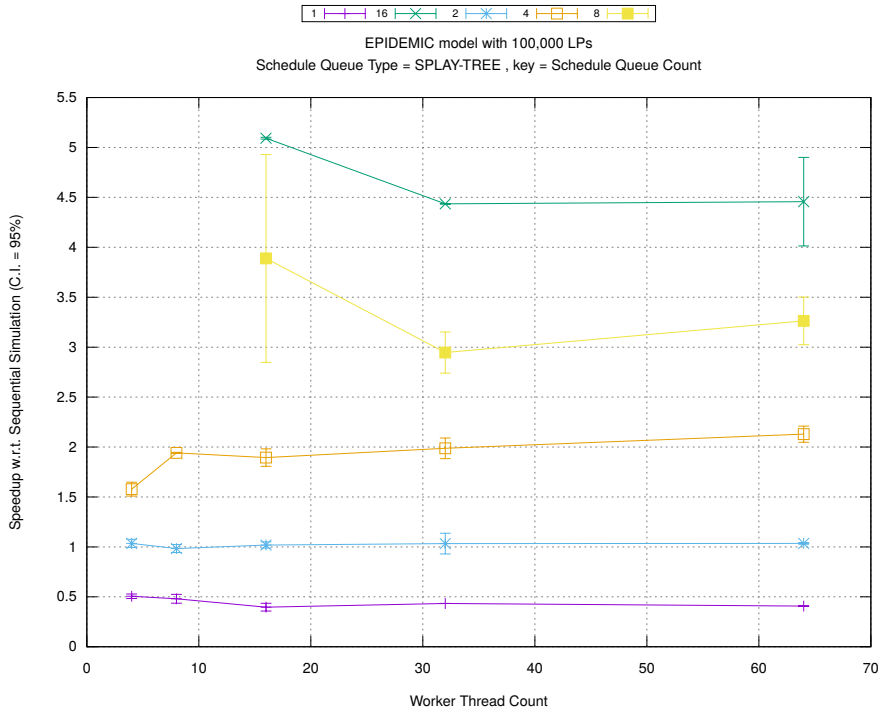


(b) Event Commitment Ratio

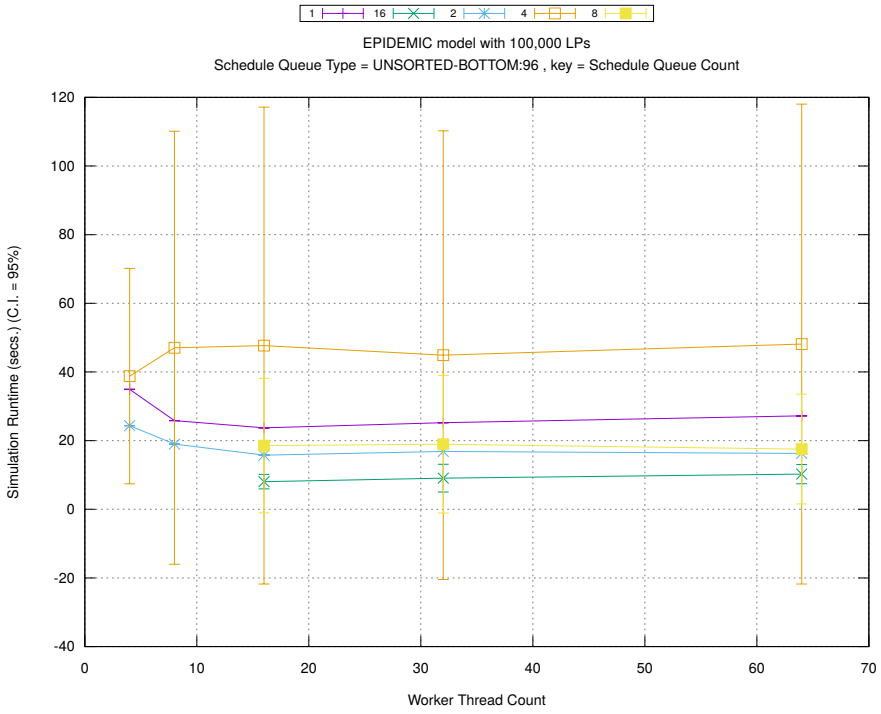


(c) Event Processing Rate (per second)

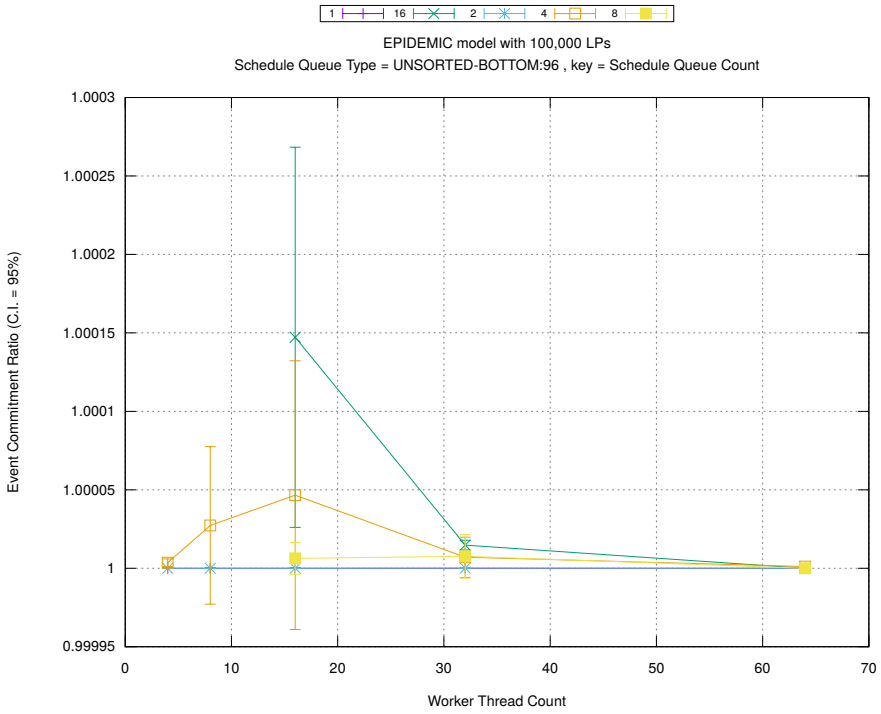
Figure A.117: epidemic 100k ws/plots/scheduleq/threads vs count key type splay-tree



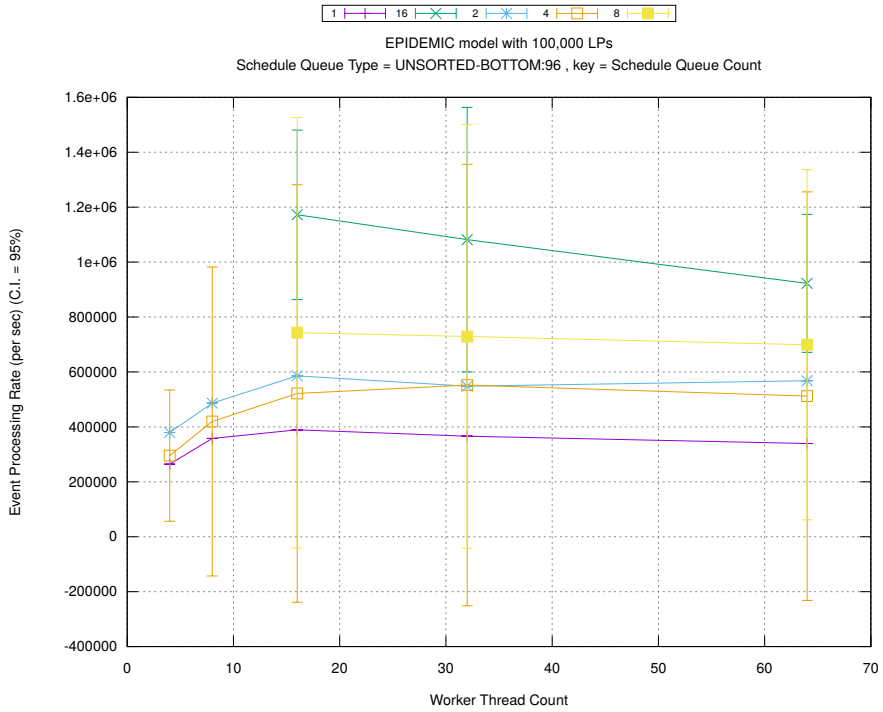
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

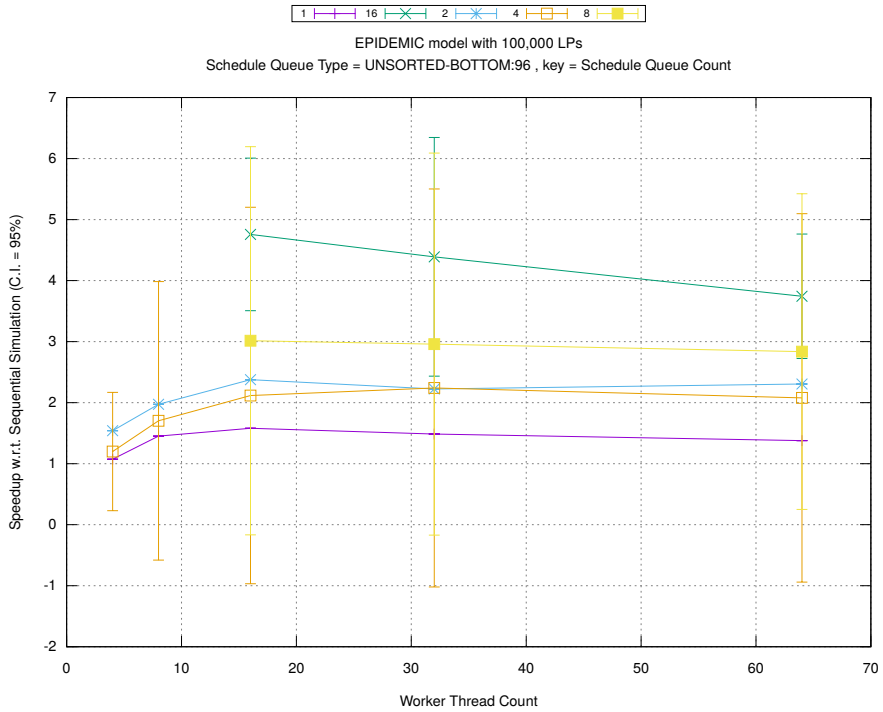


(b) Event Commitment Ratio

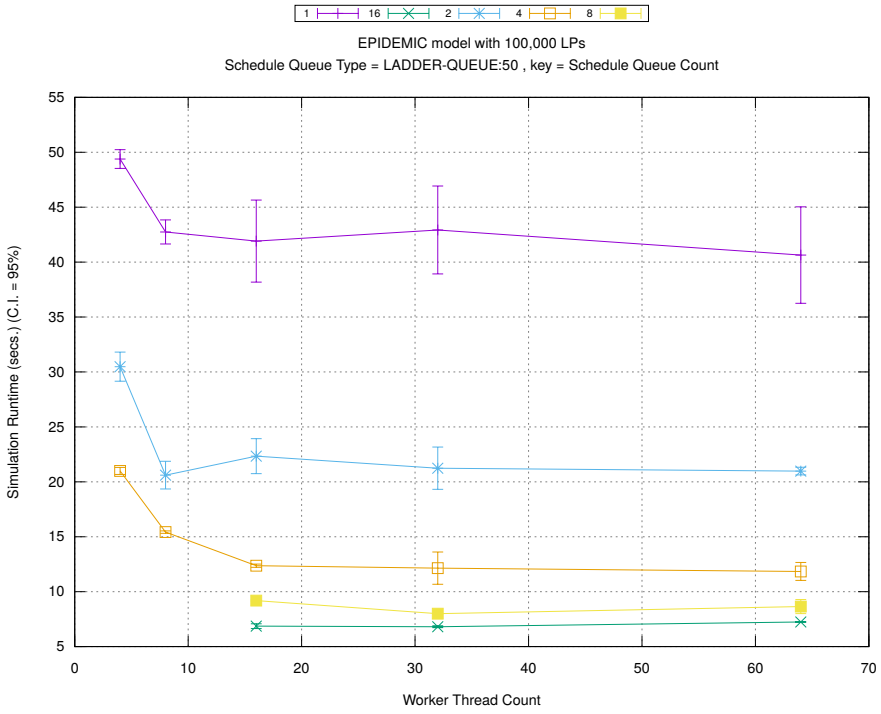


(c) Event Processing Rate (per second)

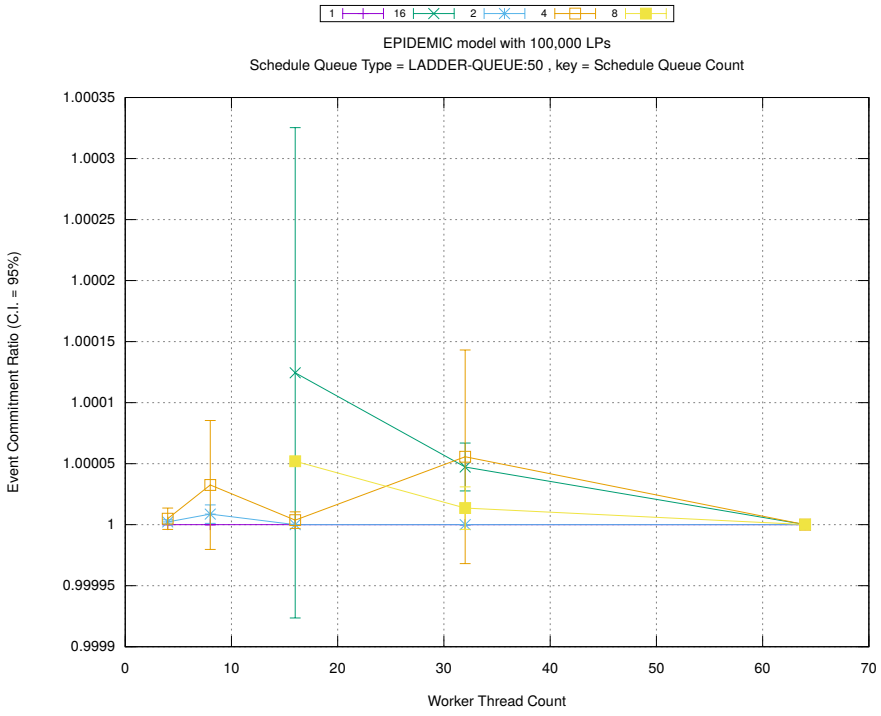
Figure A.118: epidemic 100k ws/plots/scheduleq/threads vs count key type unsorted-bottom 96



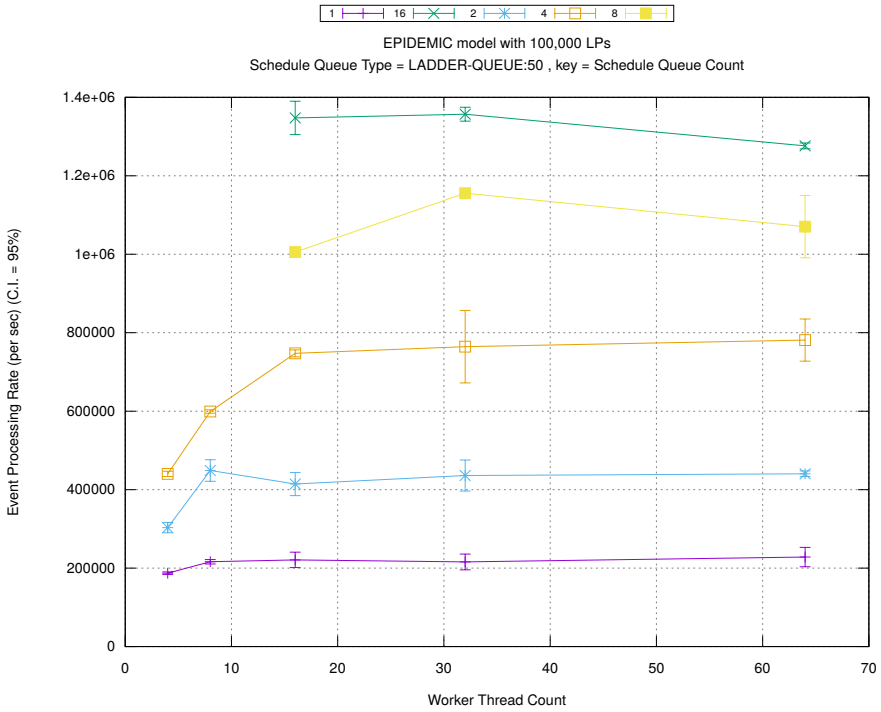
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

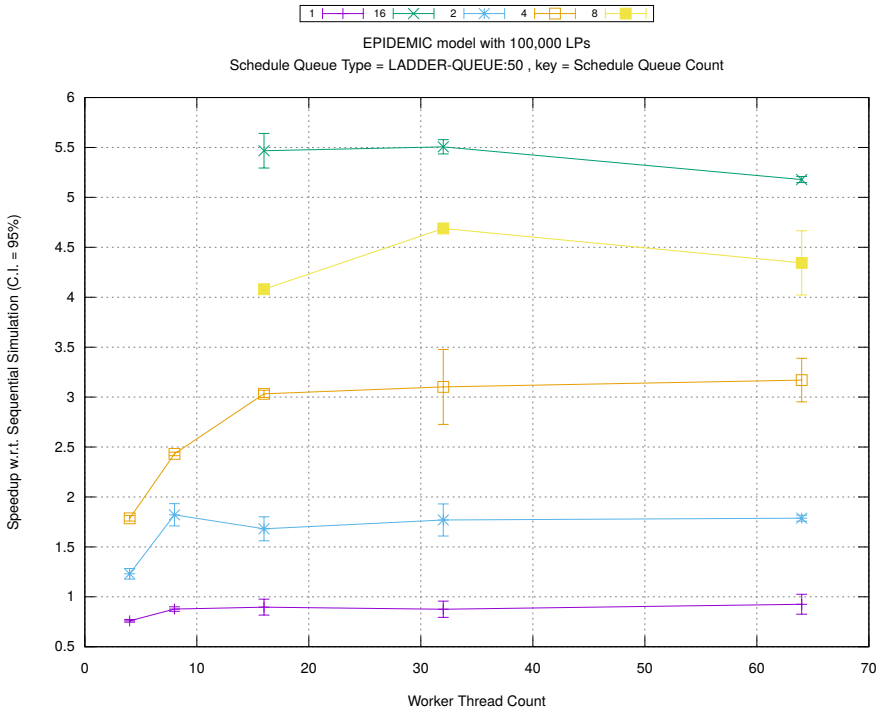


(b) Event Commitment Ratio

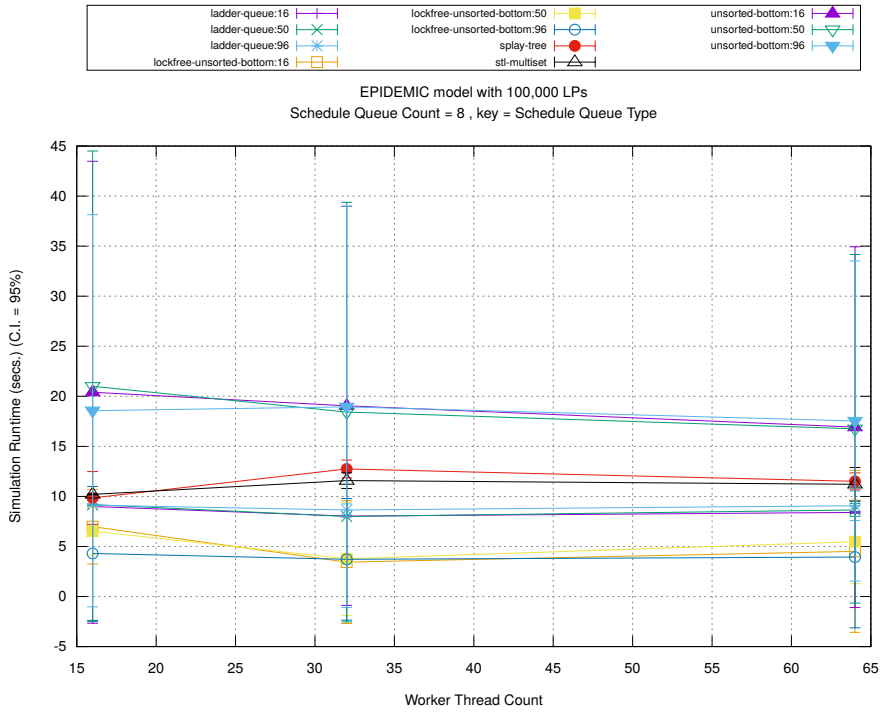


(c) Event Processing Rate (per second)

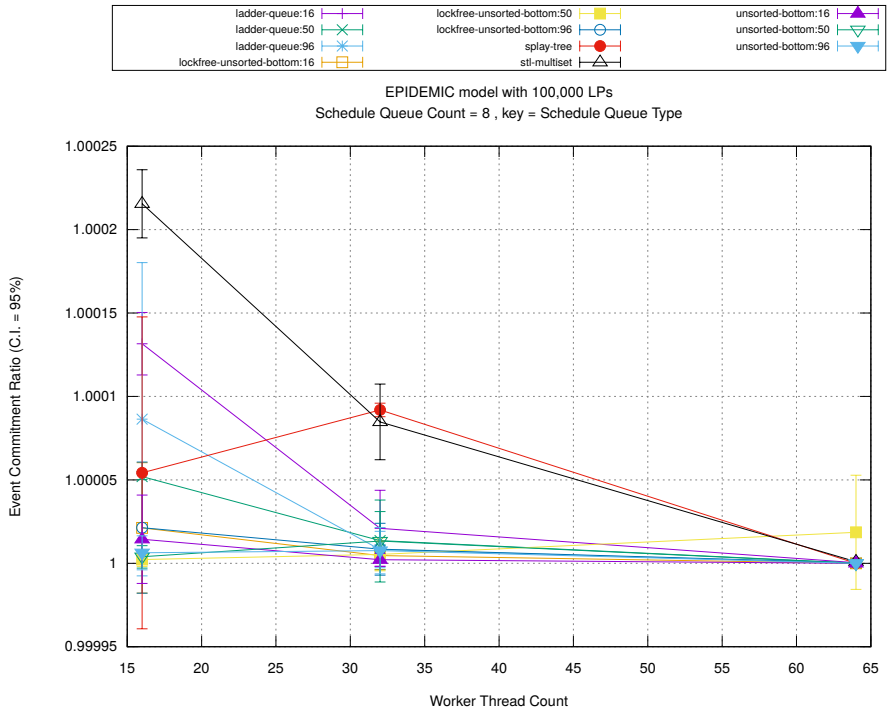
Figure A.119: epidemic 100k ws/plots/scheduleq/threads vs count key type ladder-queue 50



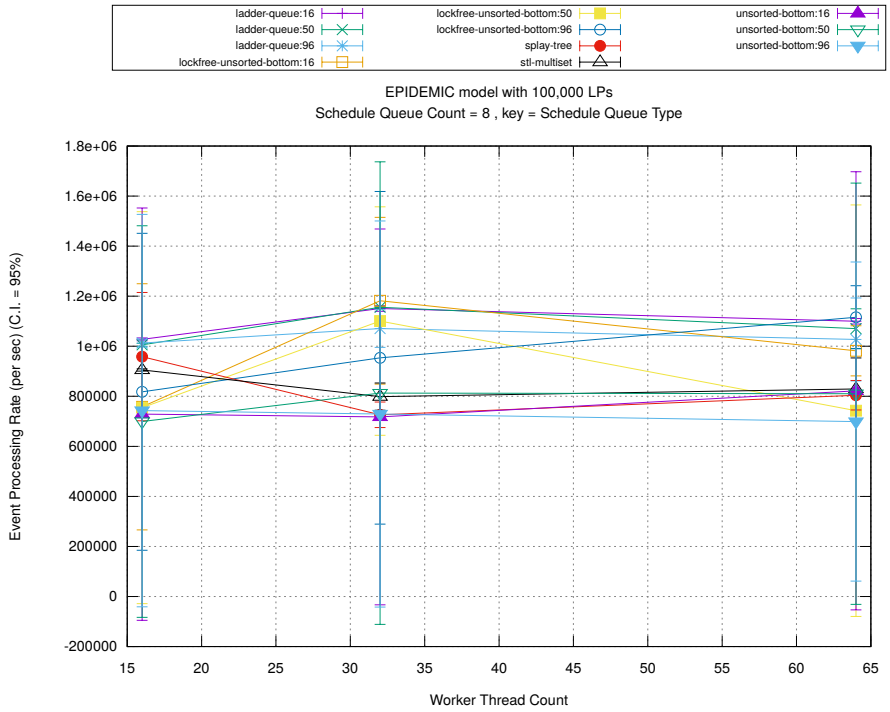
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

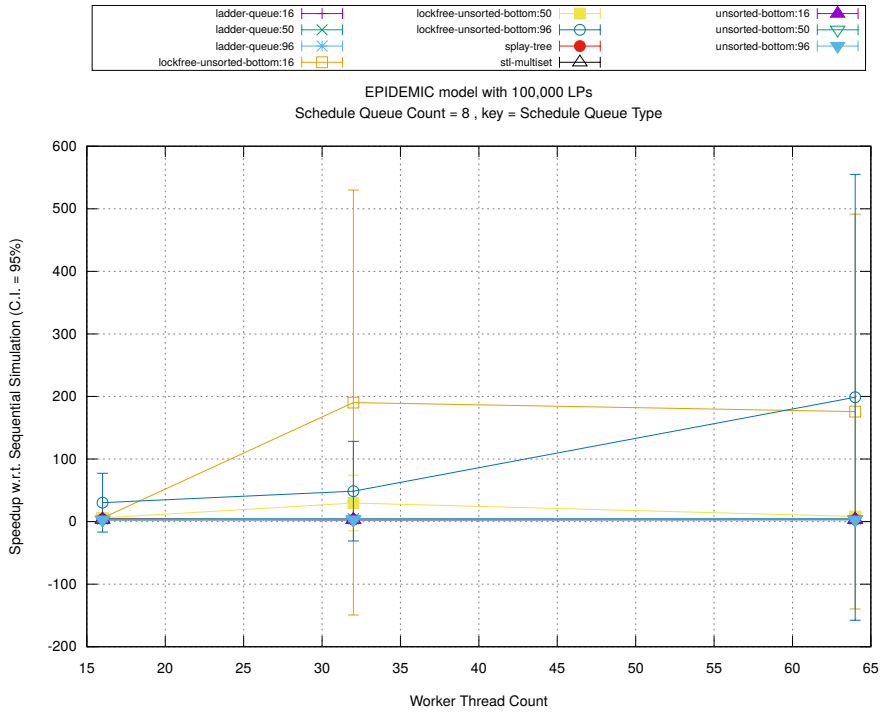


(b) Event Commitment Ratio

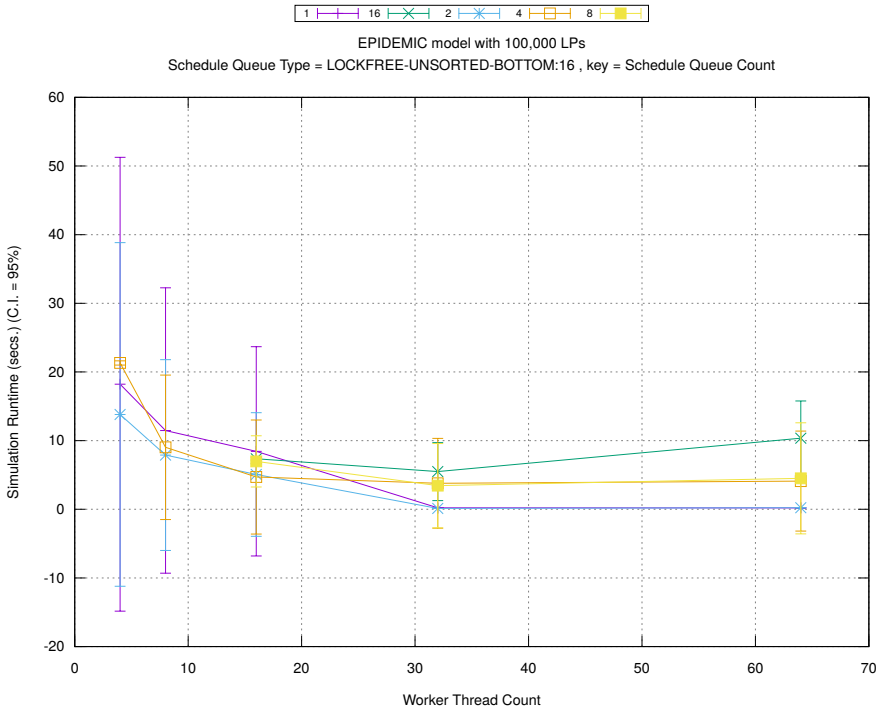


(c) Event Processing Rate (per second)

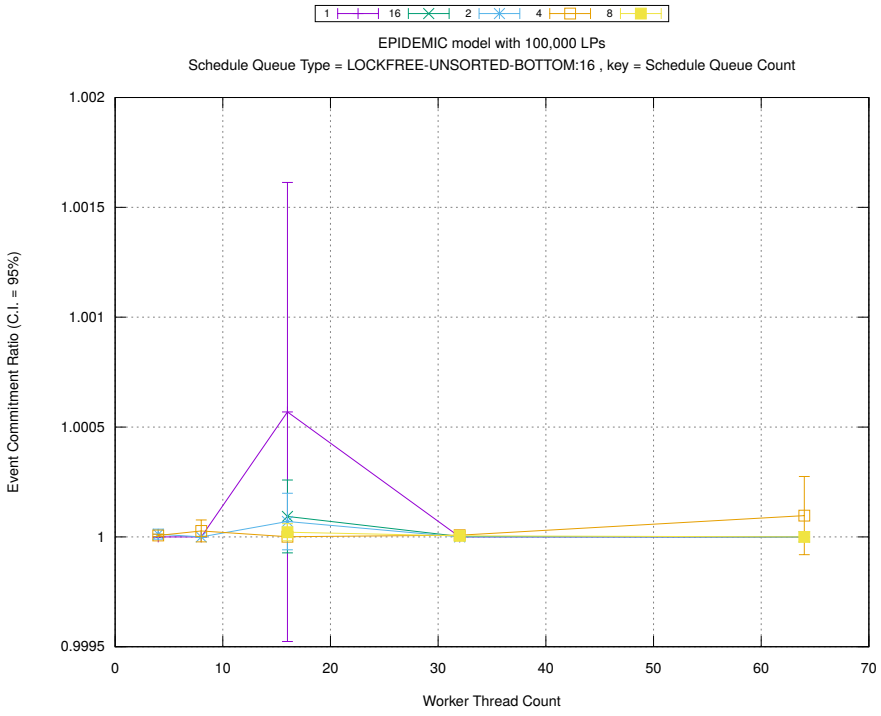
Figure A.120: epidemic 100k ws/plots/scheduleq/threads vs type key count 8



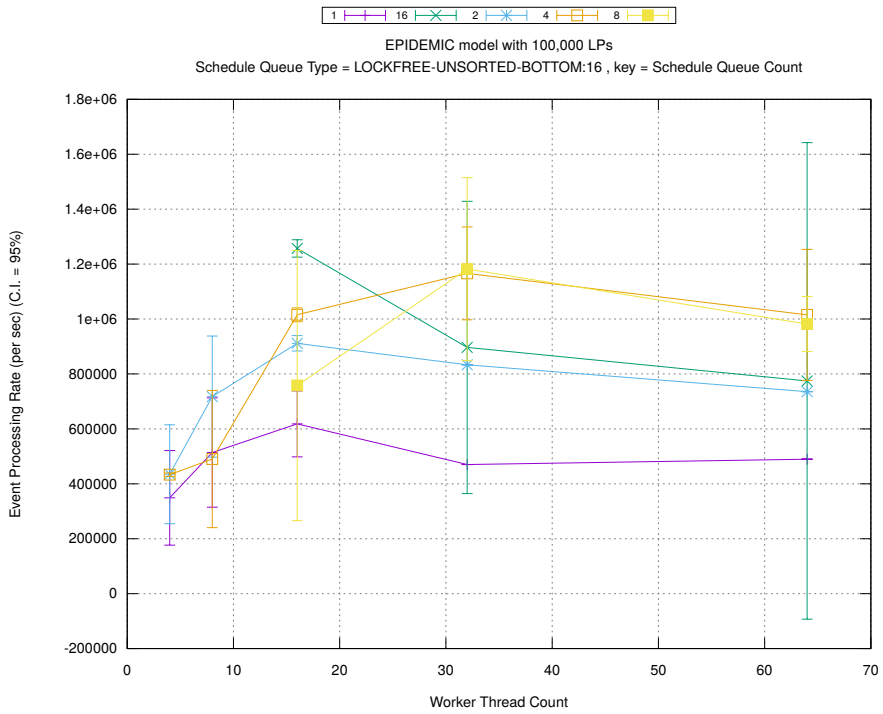
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

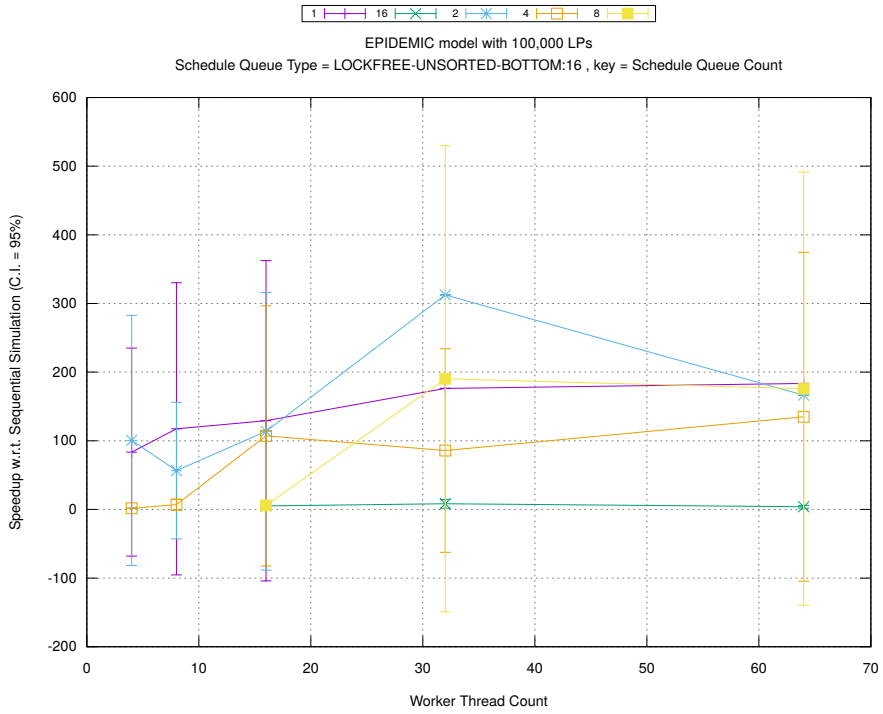


(b) Event Commitment Ratio

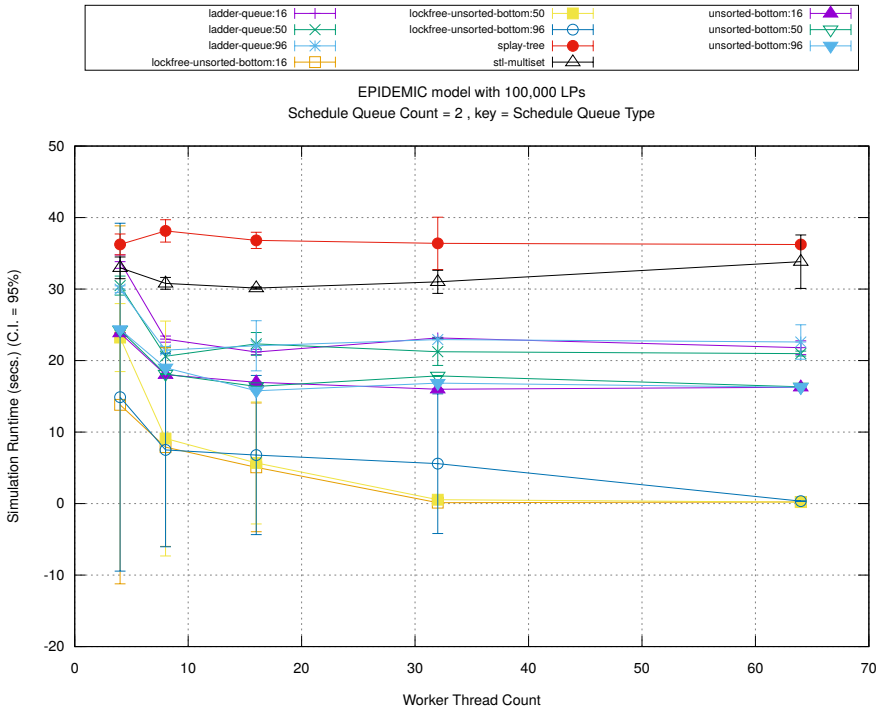


(c) Event Processing Rate (per second)

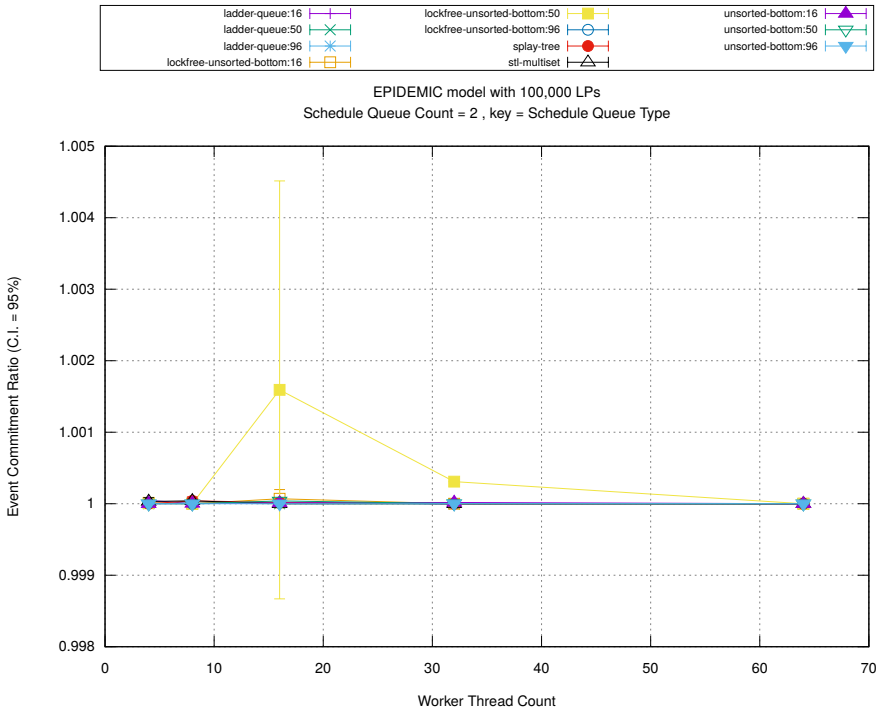
Figure A.121: epidemic 100k ws/plots/scheduleq/threads vs count key type lockfree-unsorted-bottom 16



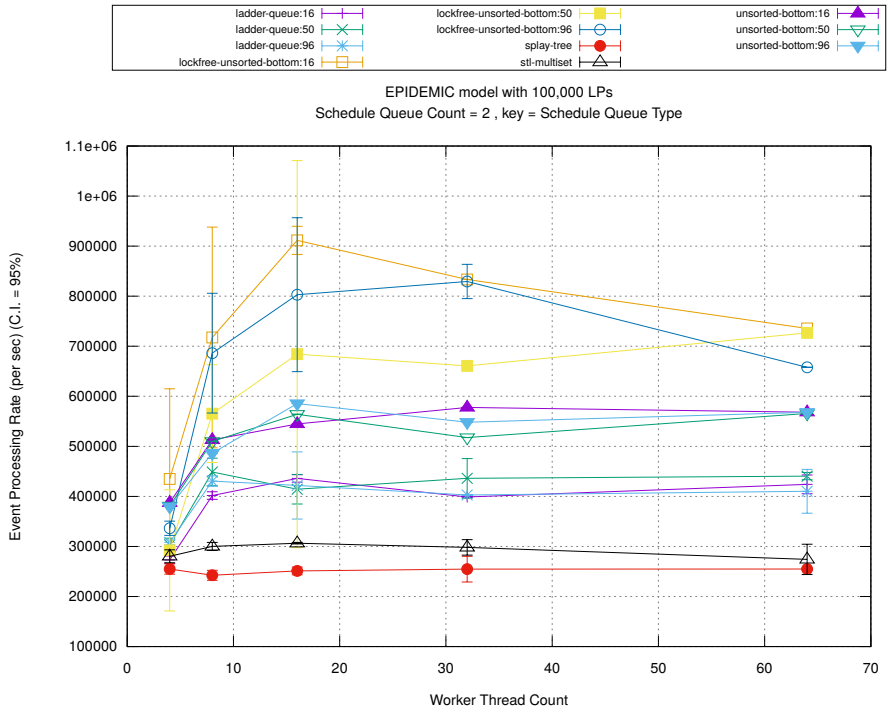
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

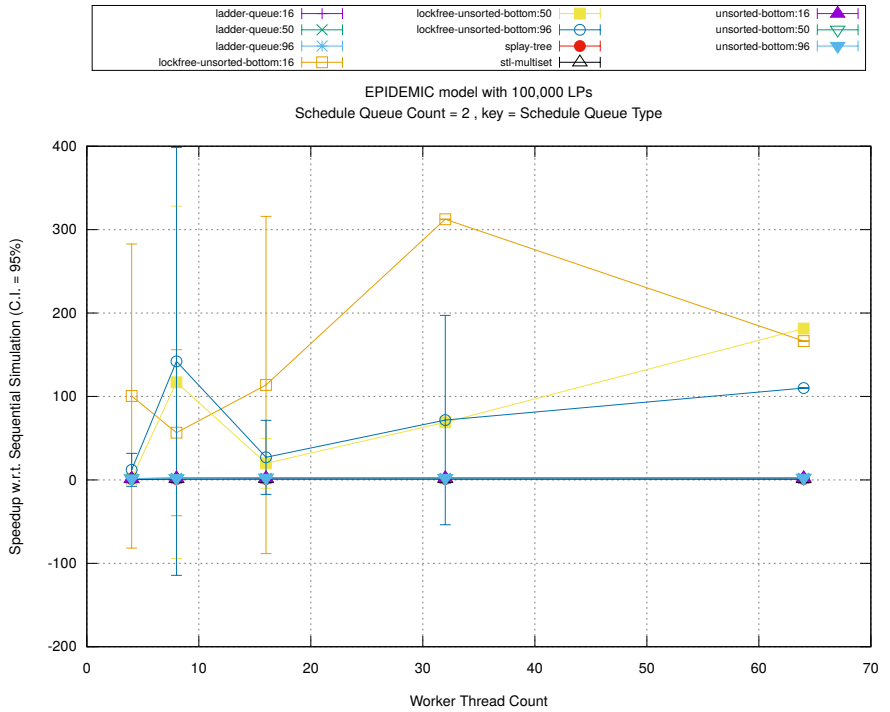


(b) Event Commitment Ratio

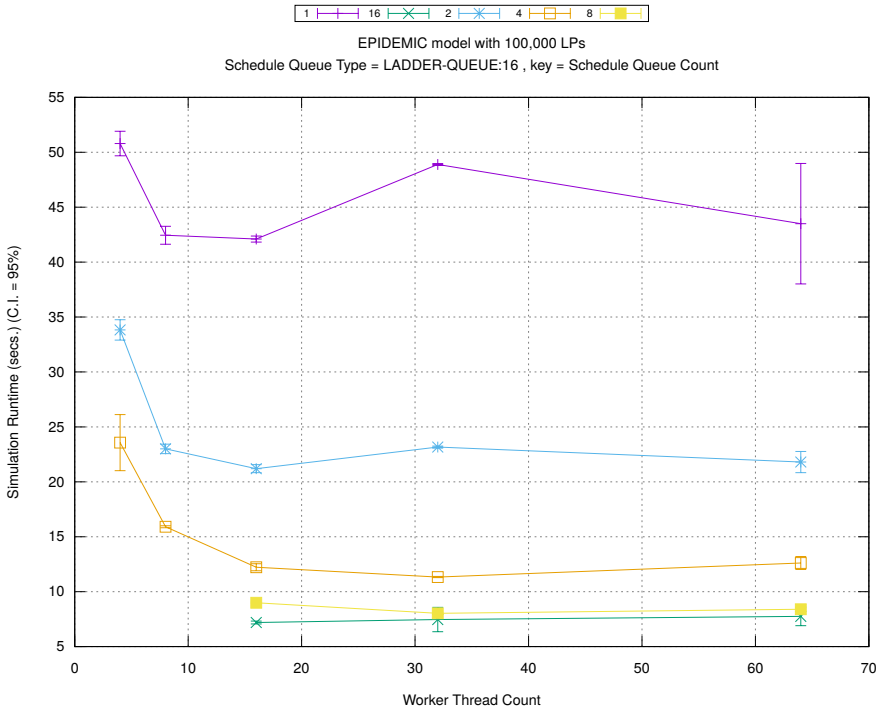


(c) Event Processing Rate (per second)

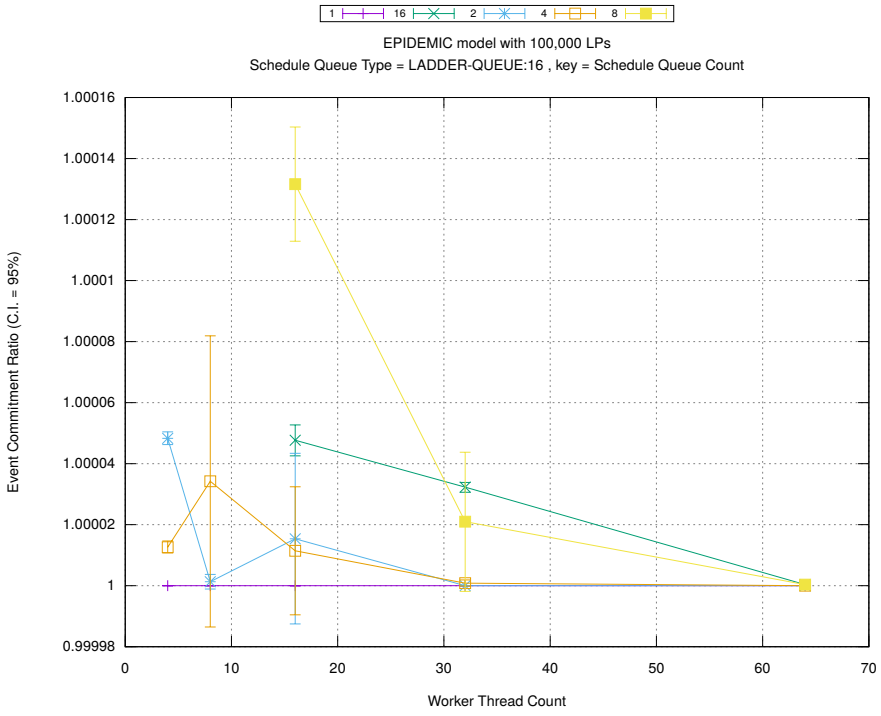
Figure A.122: epidemic 100k ws/plots/scheduleq/threads vs type key count 2



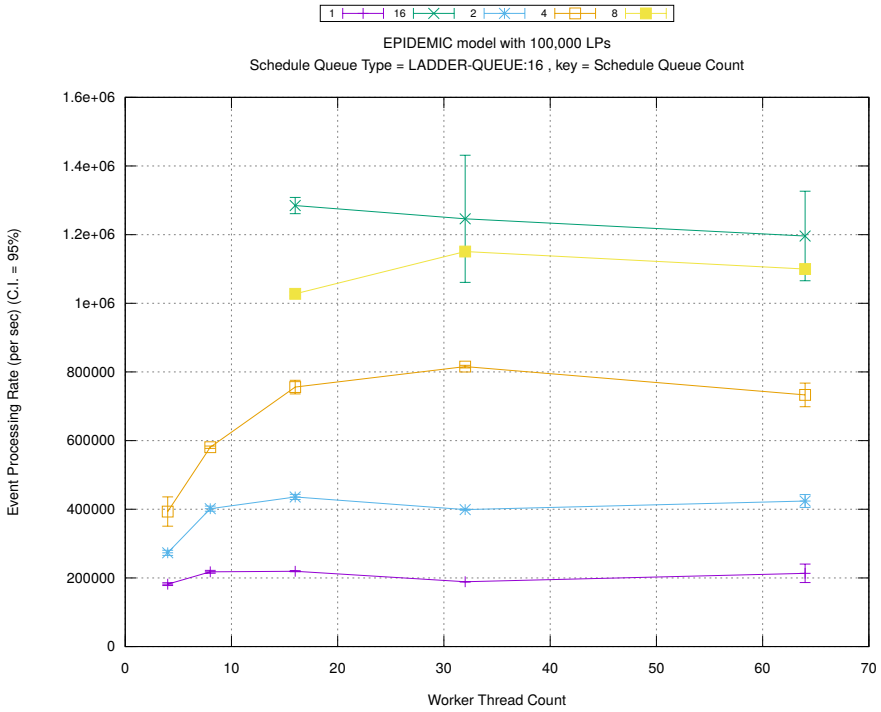
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

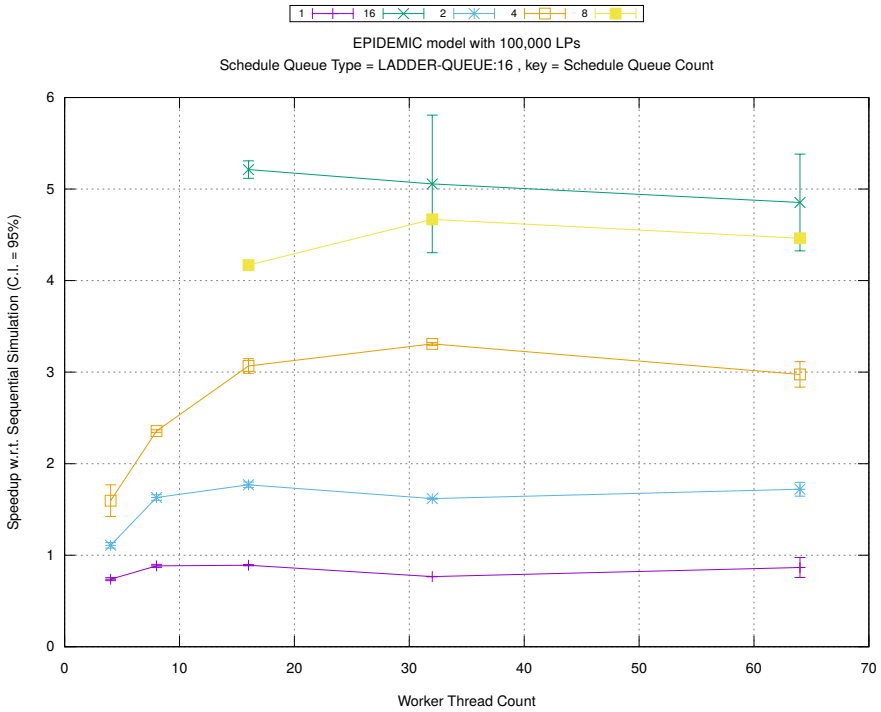


(b) Event Commitment Ratio

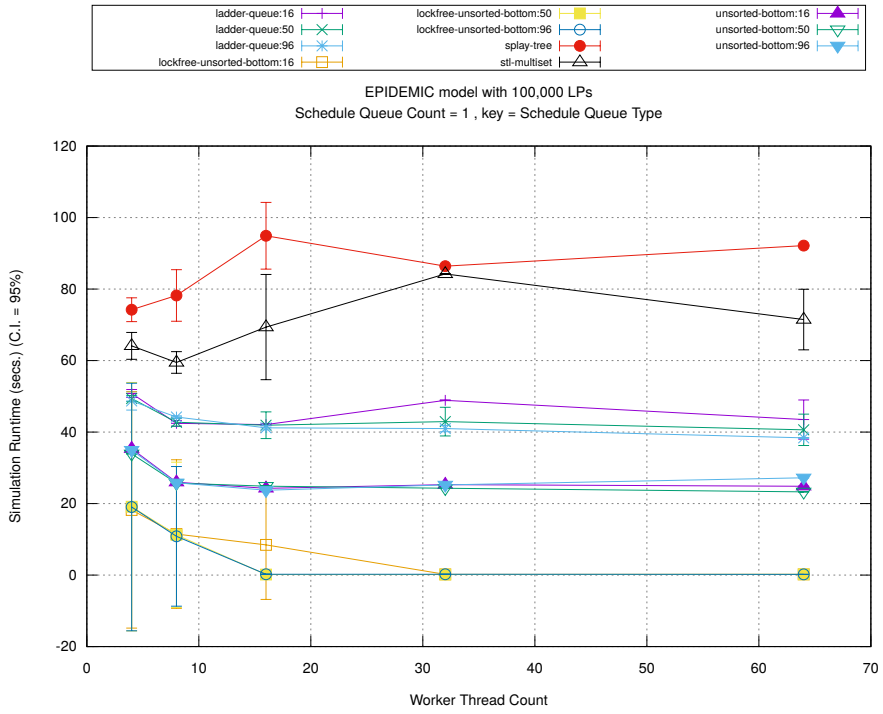


(c) Event Processing Rate (per second)

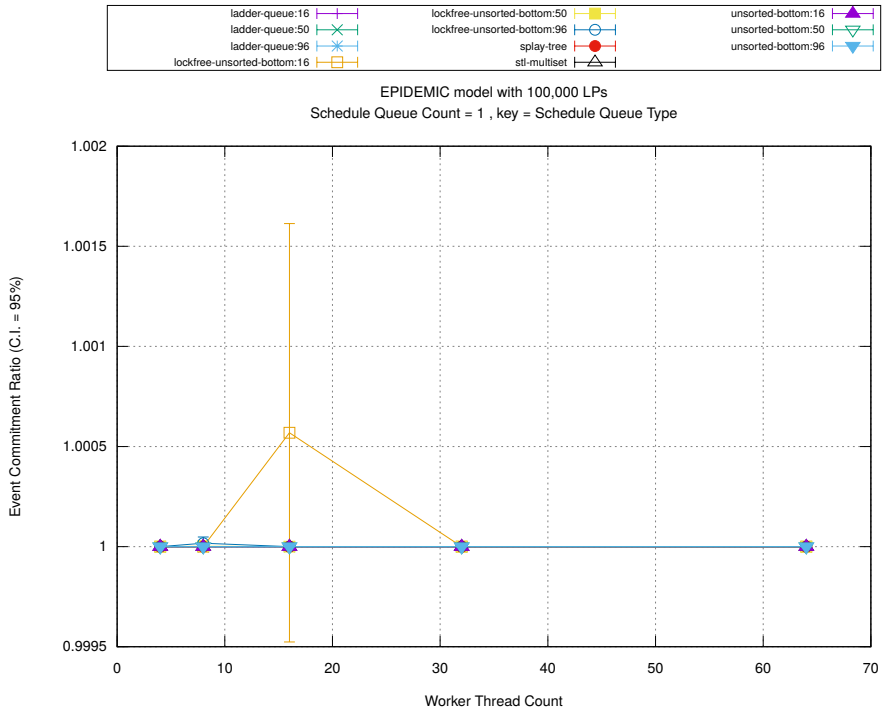
Figure A.123: epidemic 100k ws/plots/scheduleq/threads vs count key type ladder-queue 16



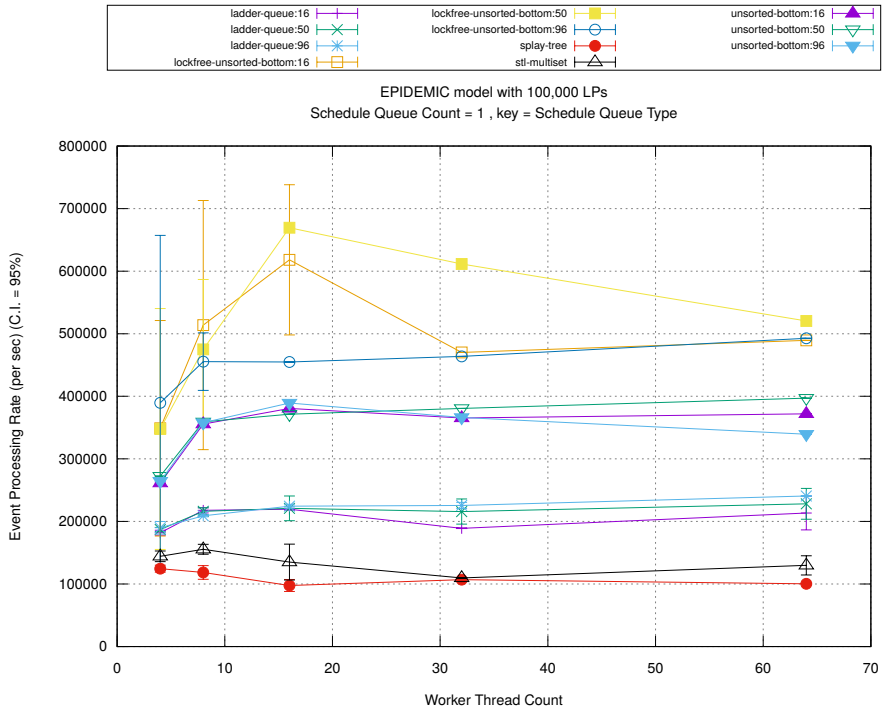
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

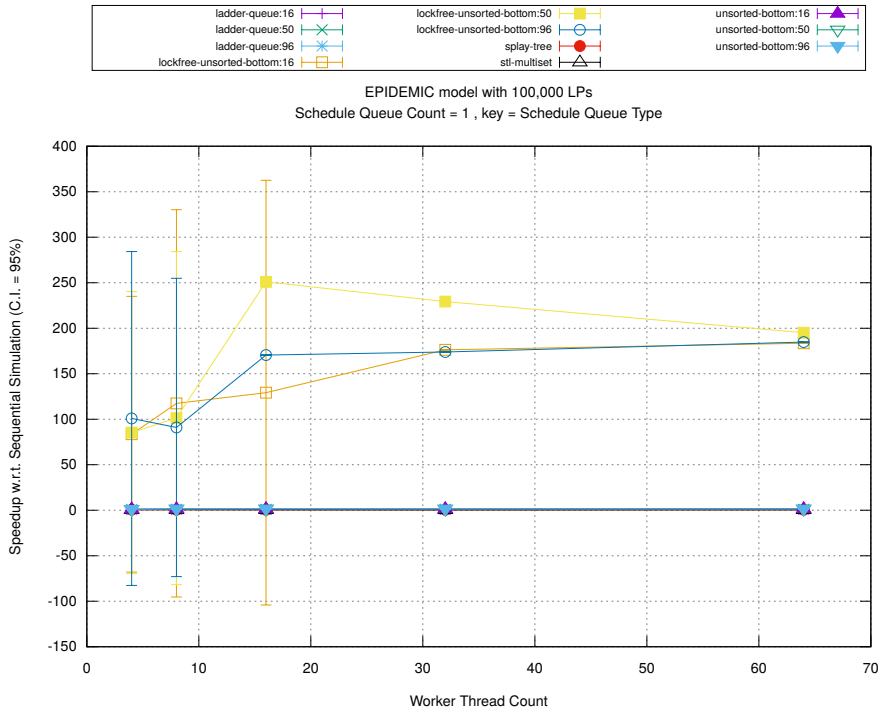


(b) Event Commitment Ratio

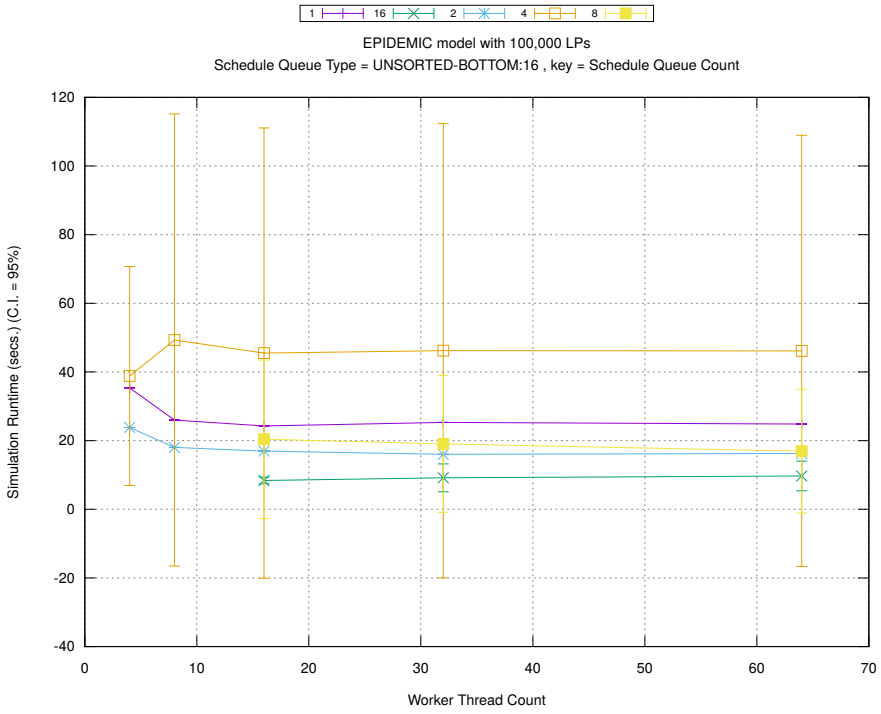


(c) Event Processing Rate (per second)

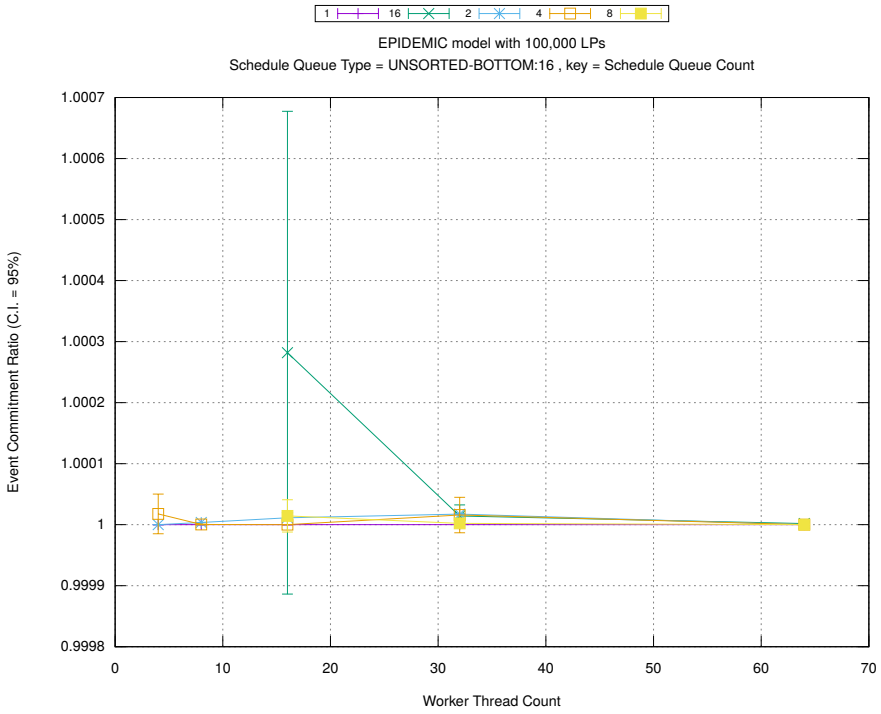
Figure A.124: epidemic 100k ws/plots/scheduleq/threads vs type key count 1



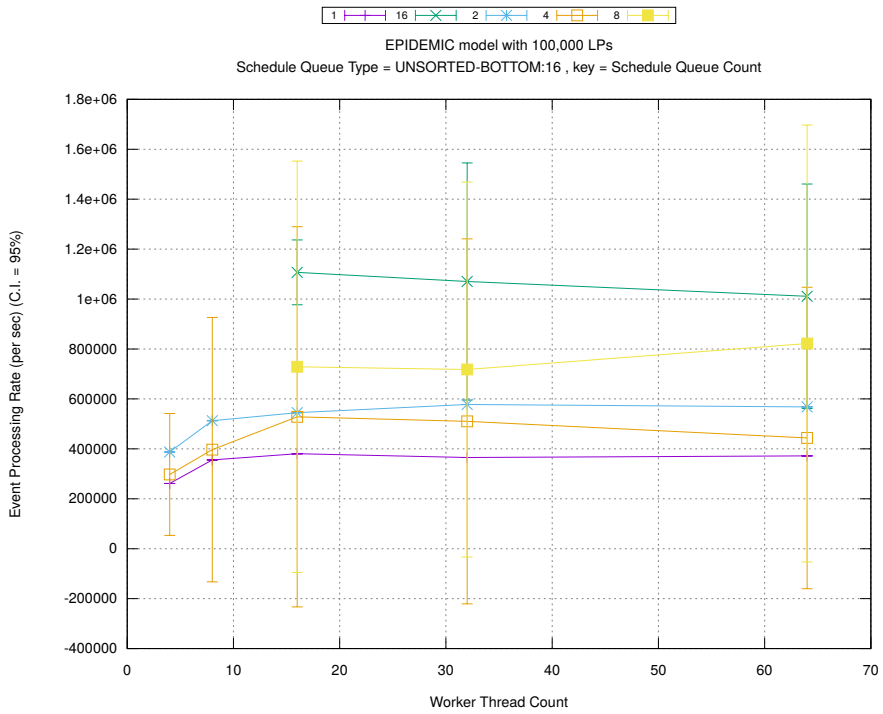
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

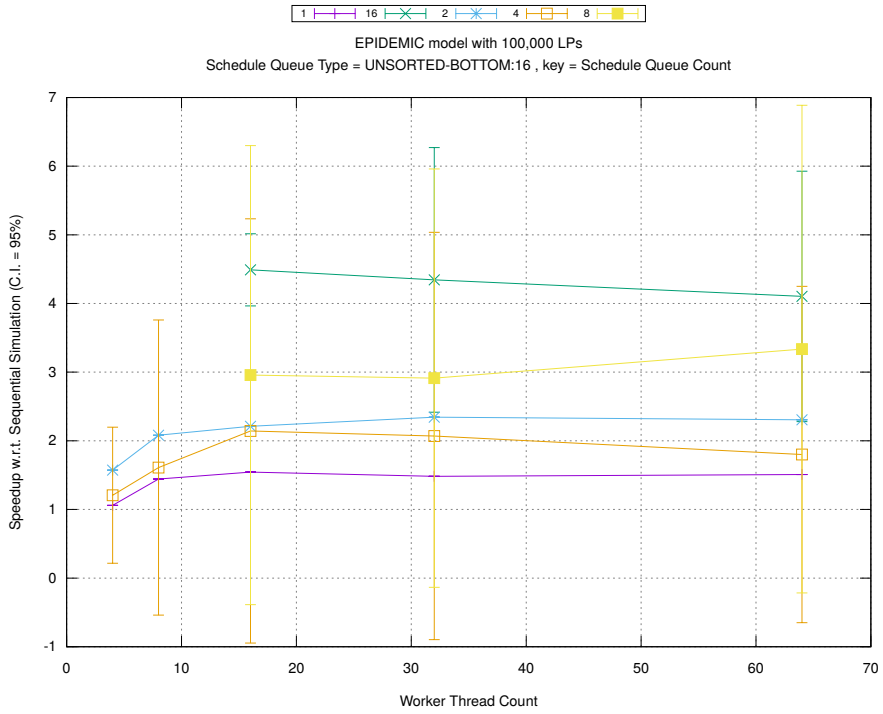


(b) Event Commitment Ratio

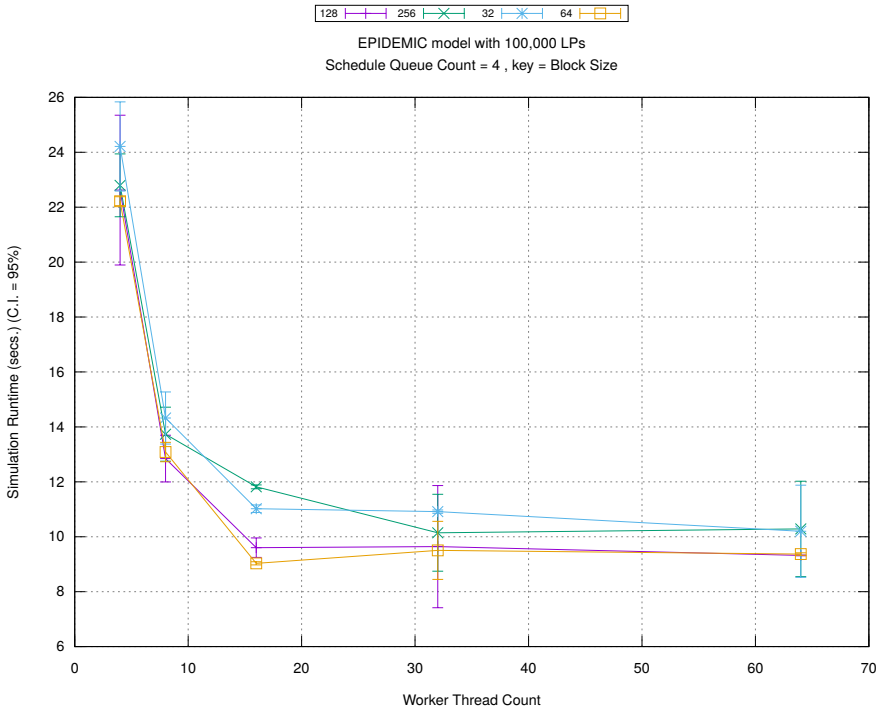


(c) Event Processing Rate (per second)

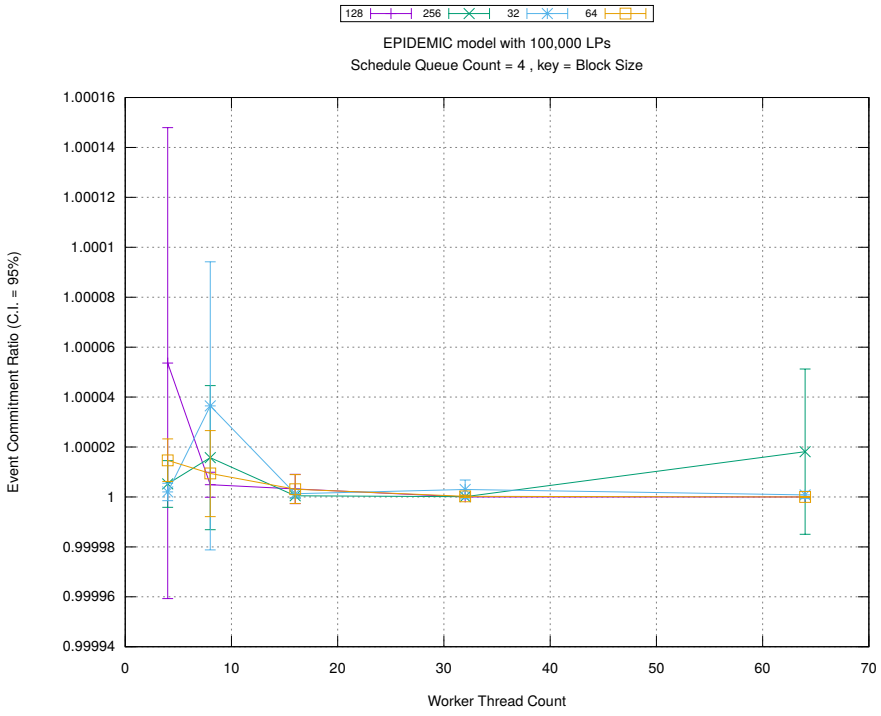
Figure A.125: epidemic 100k ws/plots/scheduleq/threads vs count key type unsorted-bottom 16



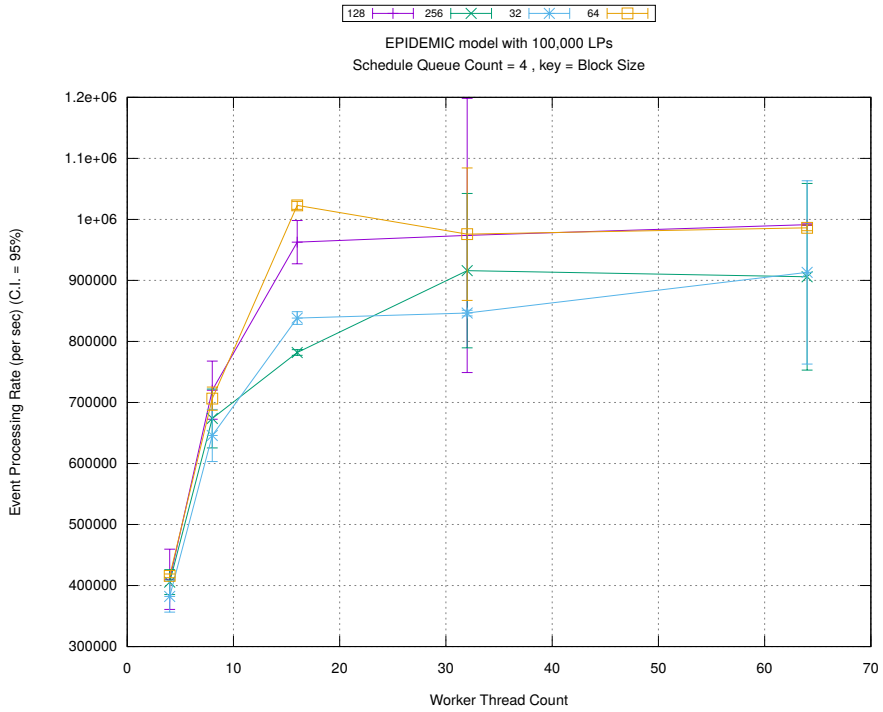
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

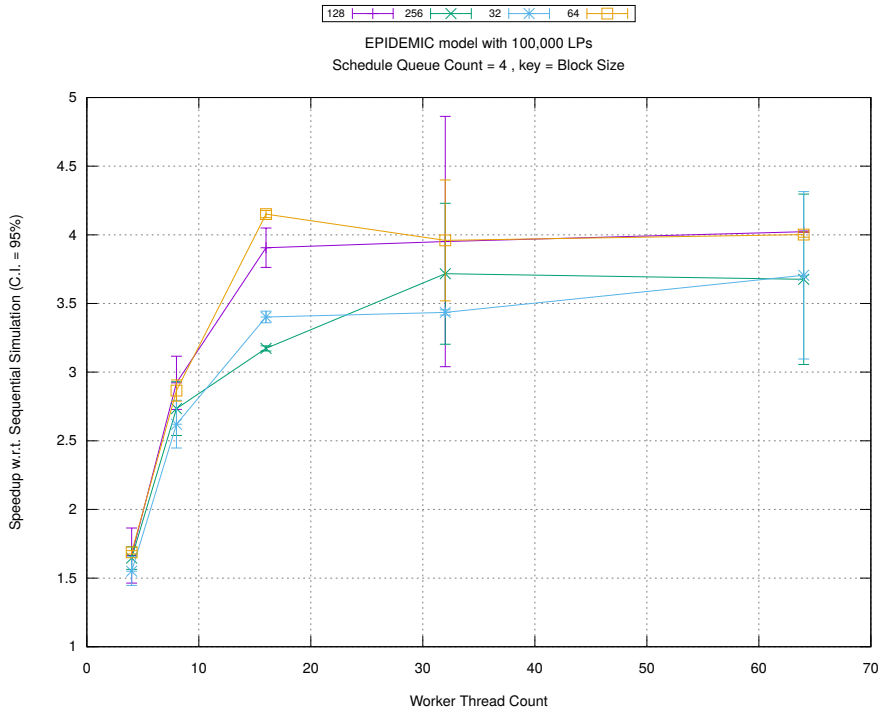


(b) Event Commitment Ratio

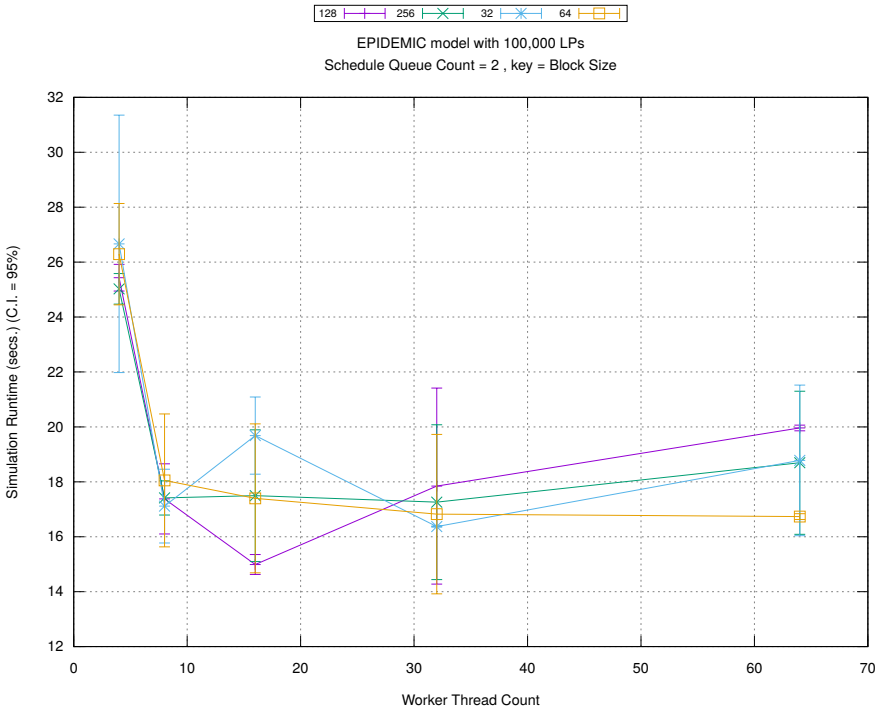


(c) Event Processing Rate (per second)

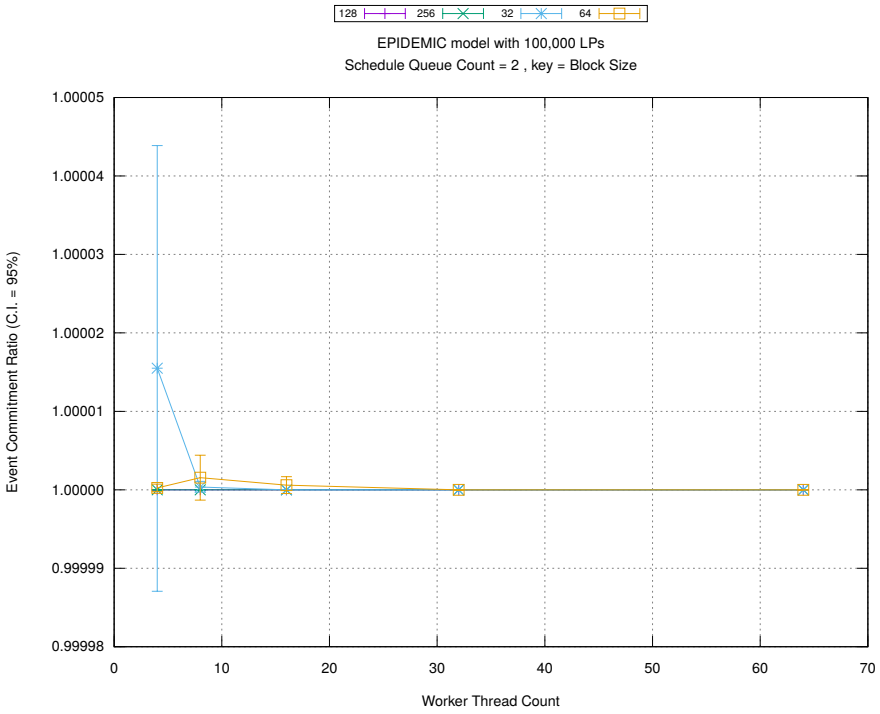
Figure A.126: epidemic 100k ws/plots/blocks/threads vs blocksize key count 4



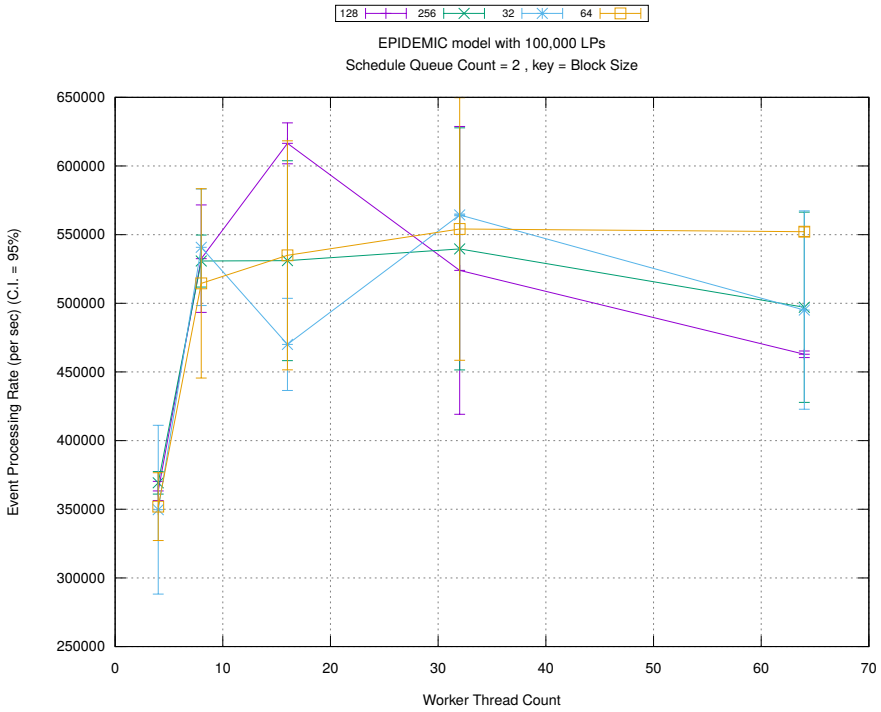
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

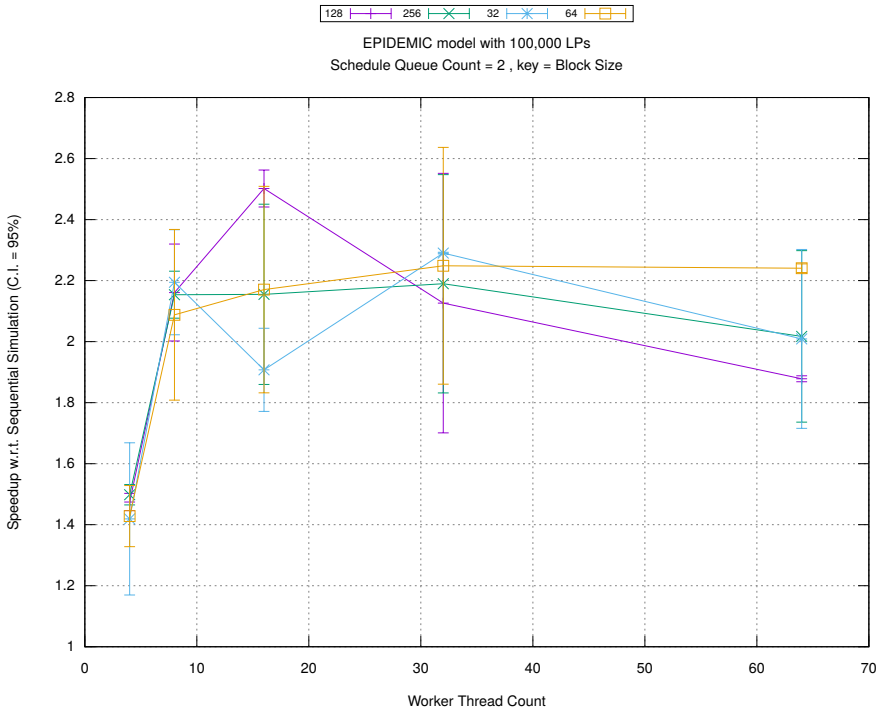


(b) Event Commitment Ratio

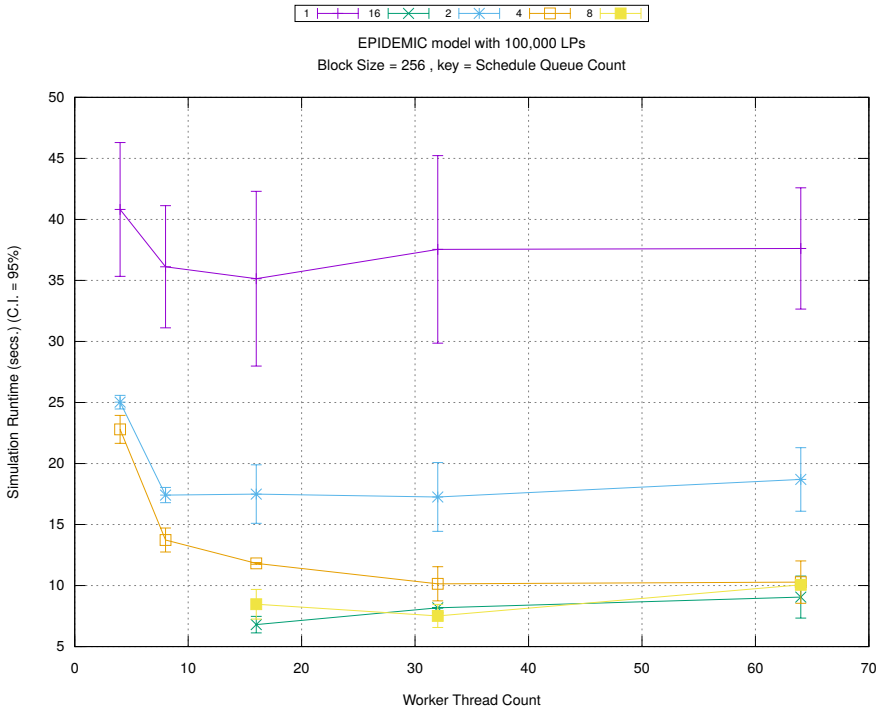


(c) Event Processing Rate (per second)

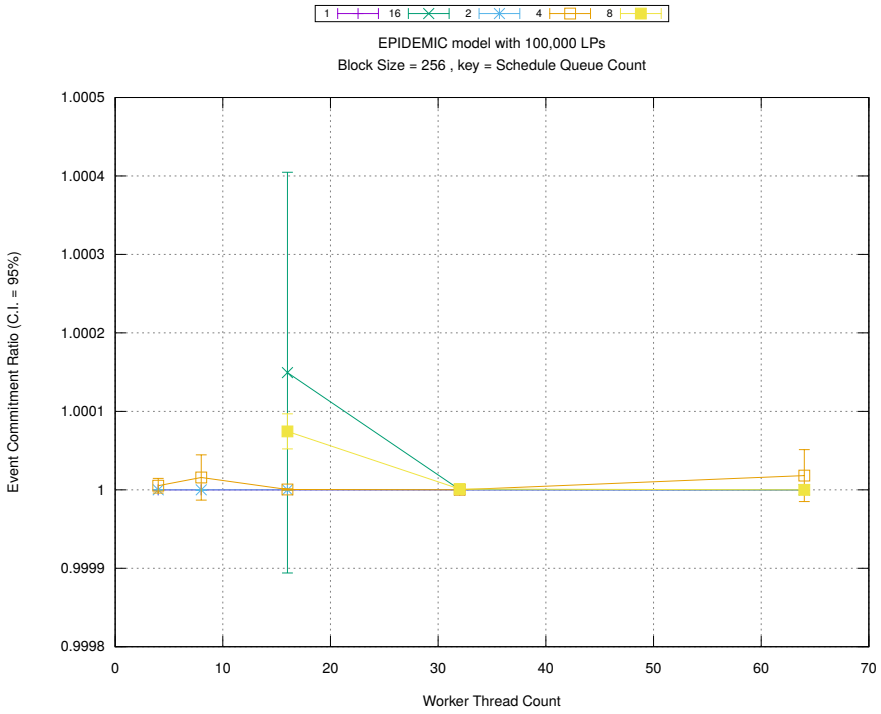
Figure A.127: epidemic 100k ws/plots/blocks/threads vs blocksize key count 2



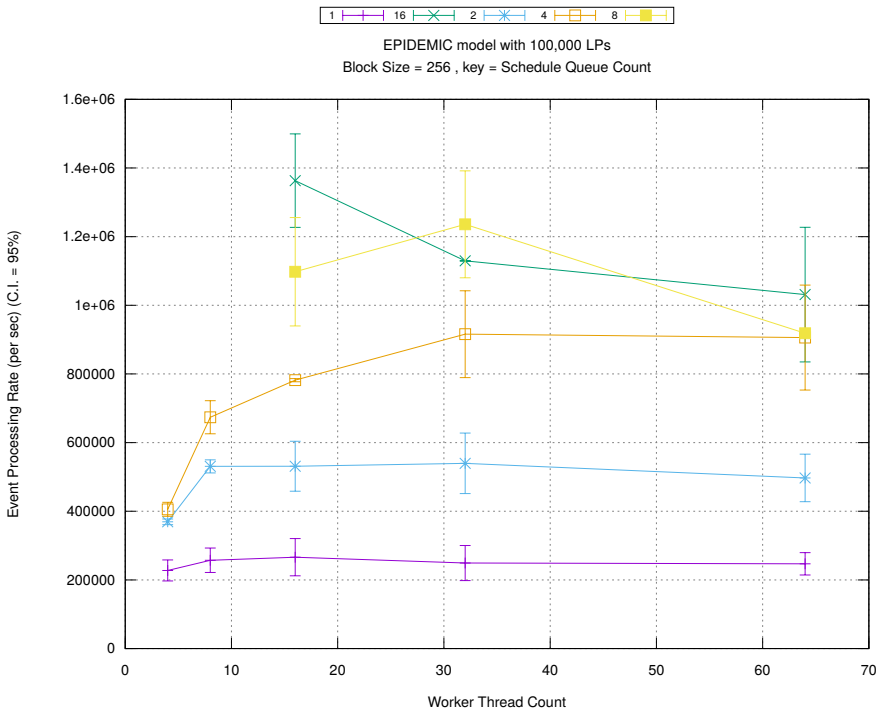
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

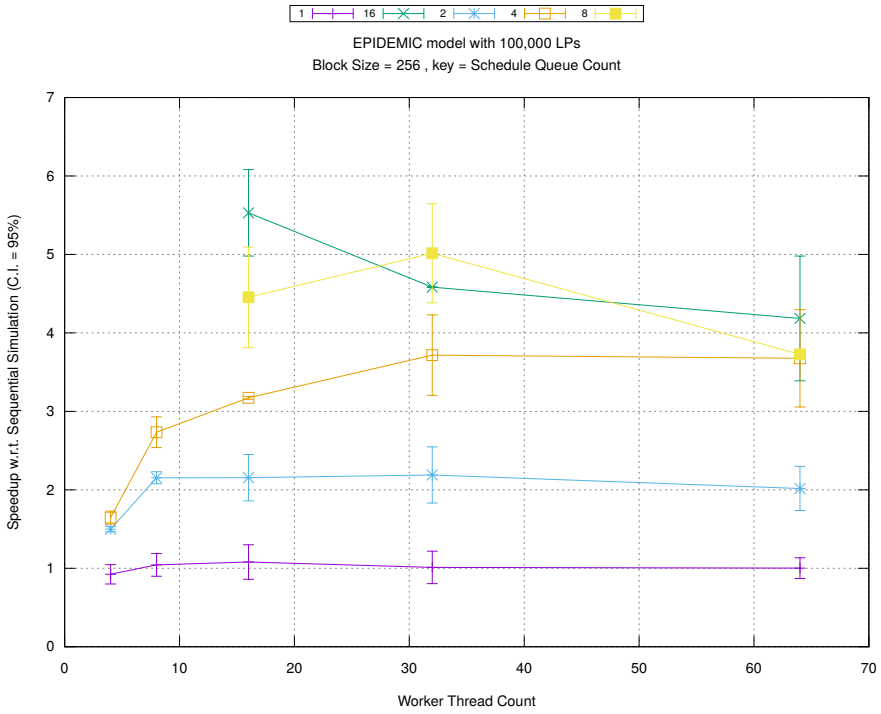


(b) Event Commitment Ratio

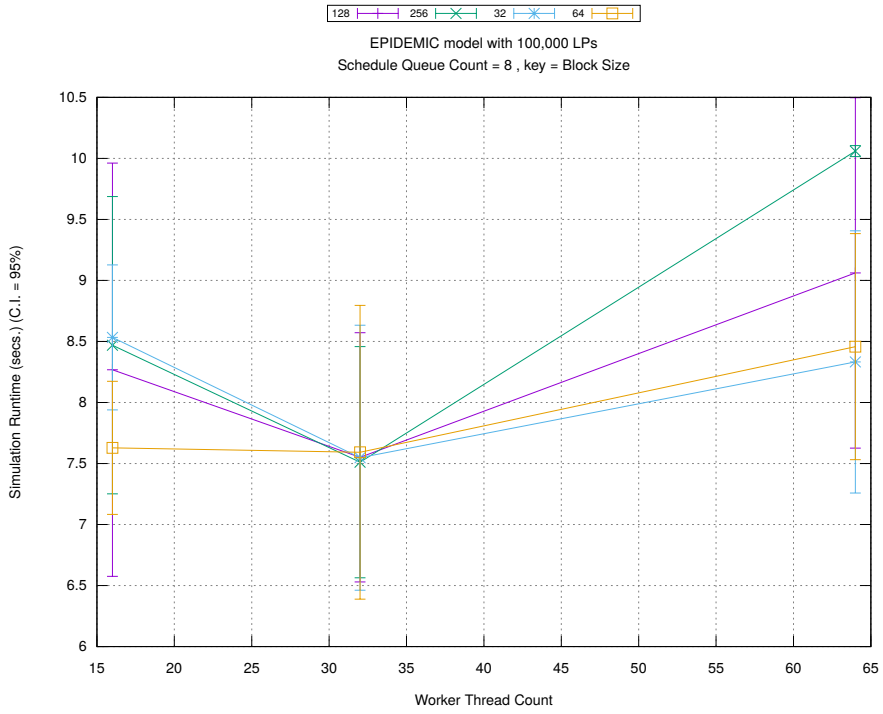


(c) Event Processing Rate (per second)

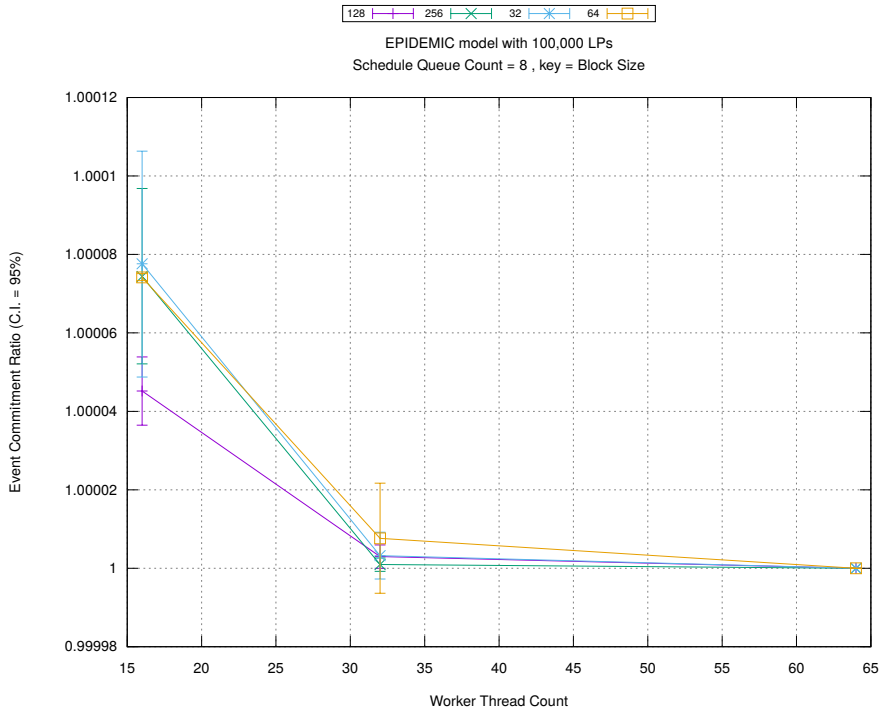
Figure A.128: epidemic 100k ws/plots/blocks/threads vs count key blocksize 256



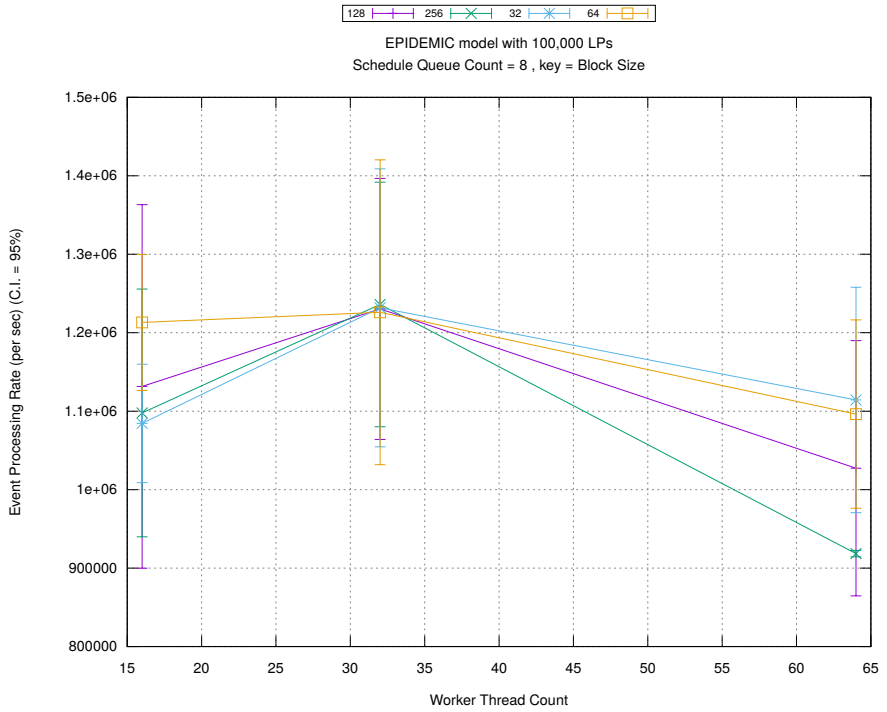
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

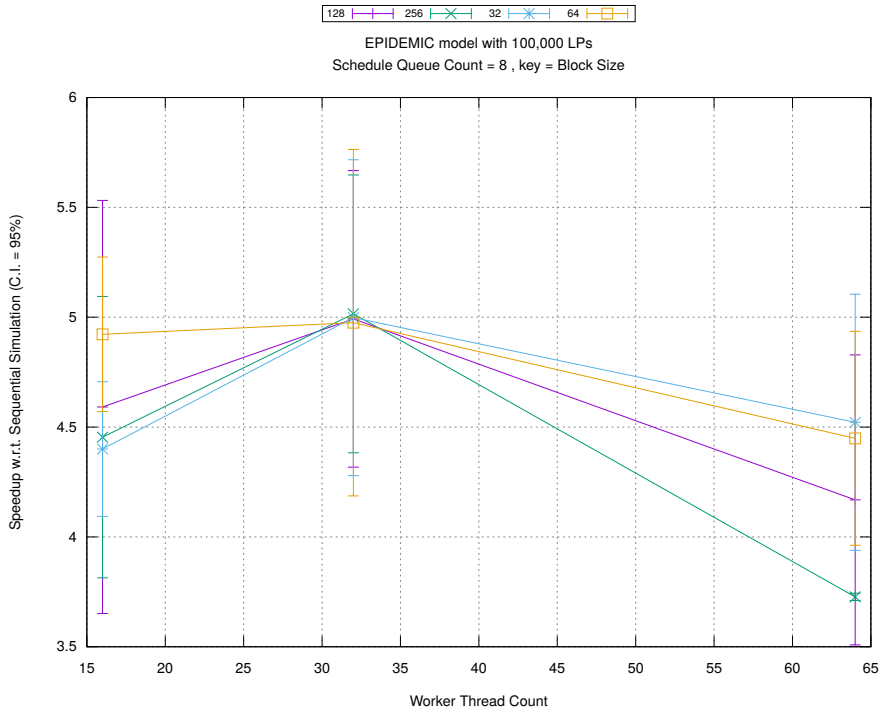


(b) Event Commitment Ratio

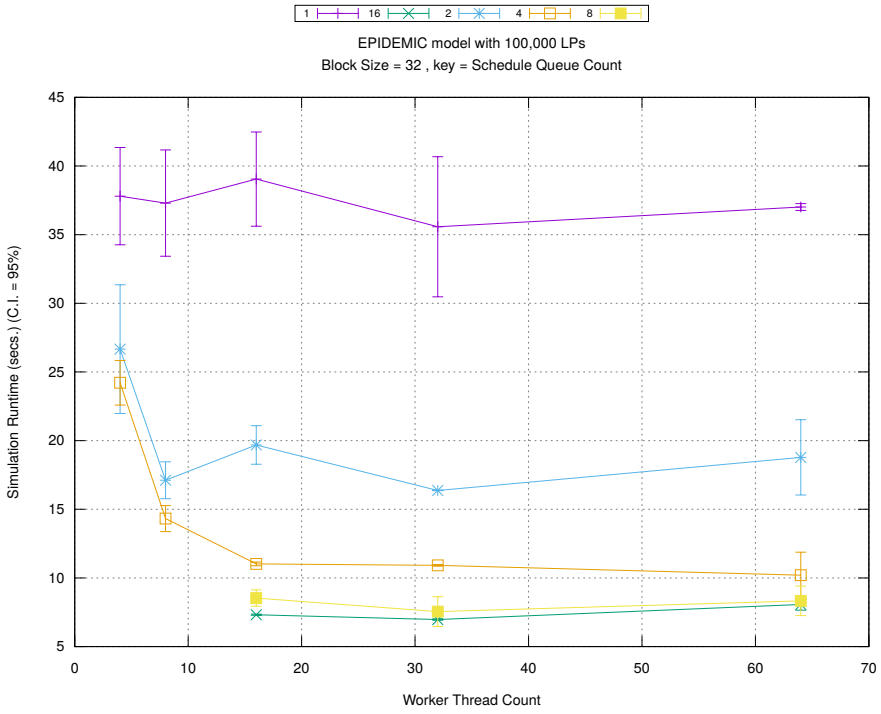


(c) Event Processing Rate (per second)

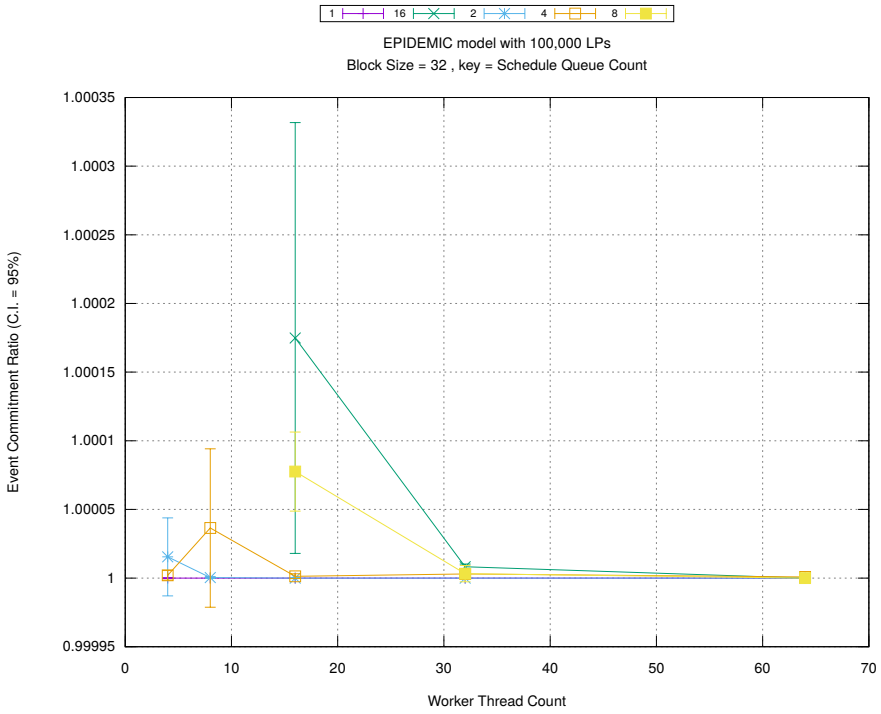
Figure A.129: epidemic 100k ws/plots/blocks/threads vs blocksize key count 8



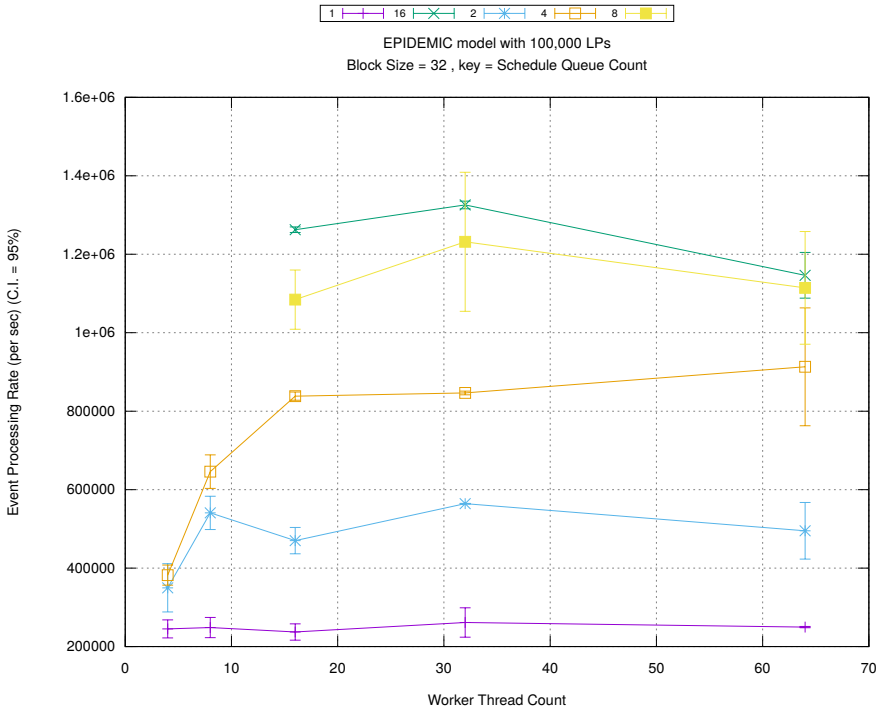
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

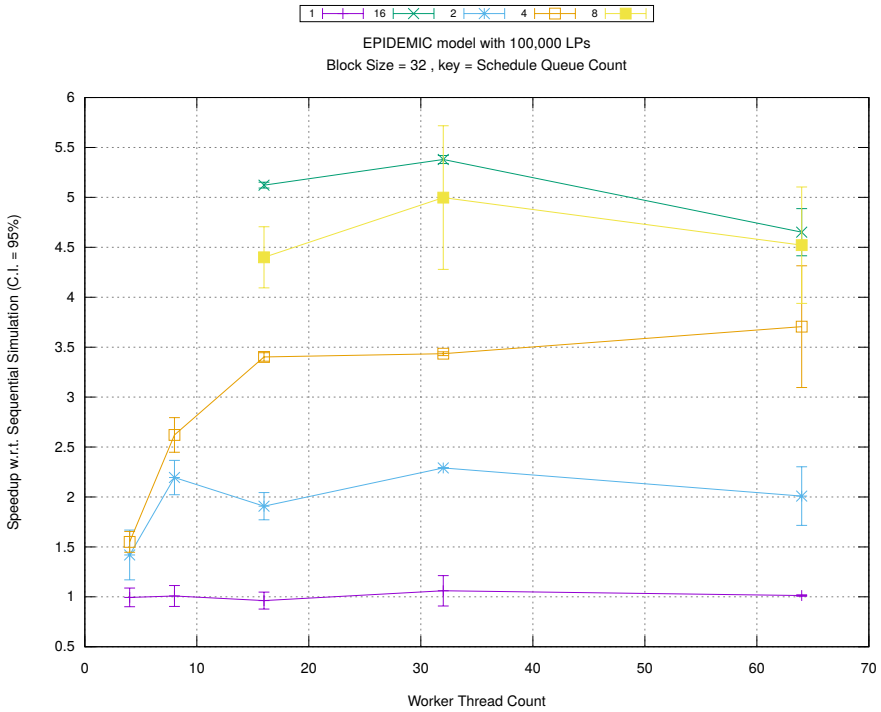


(b) Event Commitment Ratio

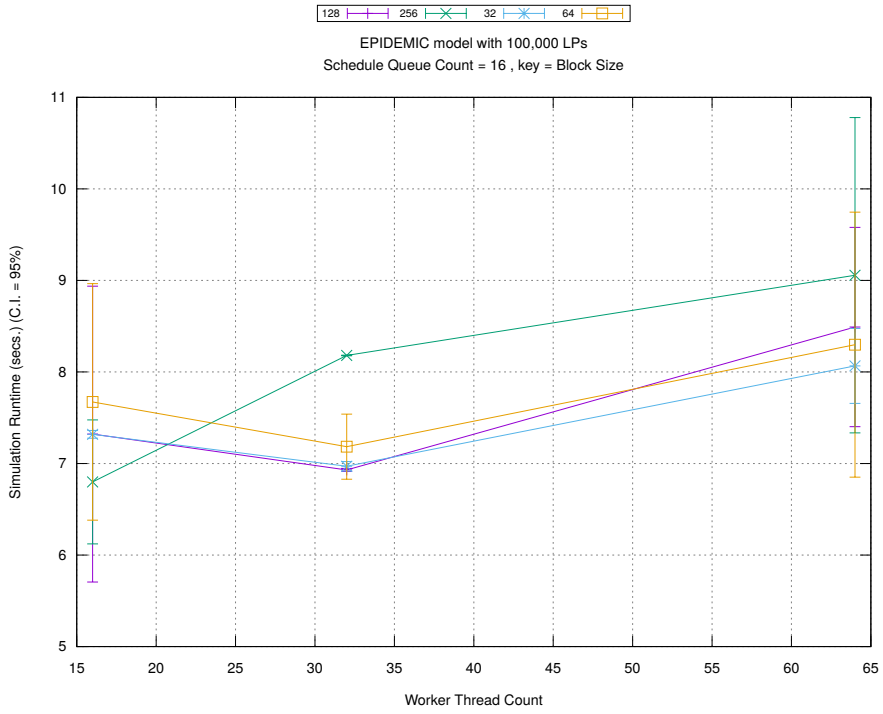


(c) Event Processing Rate (per second)

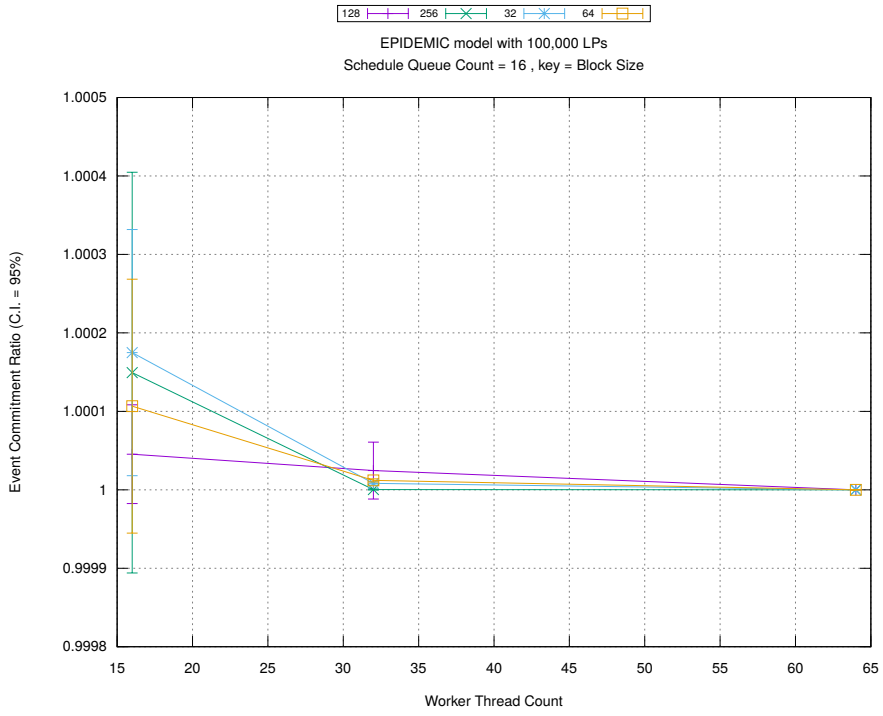
Figure A.130: epidemic 100k ws/plots/blocks/threads vs count key blocksize 32



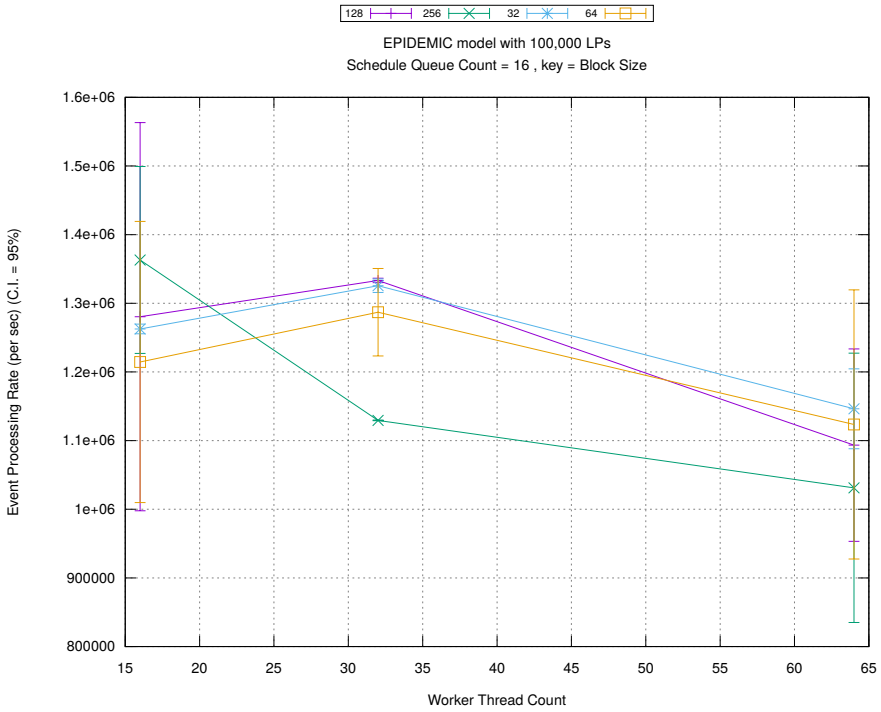
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

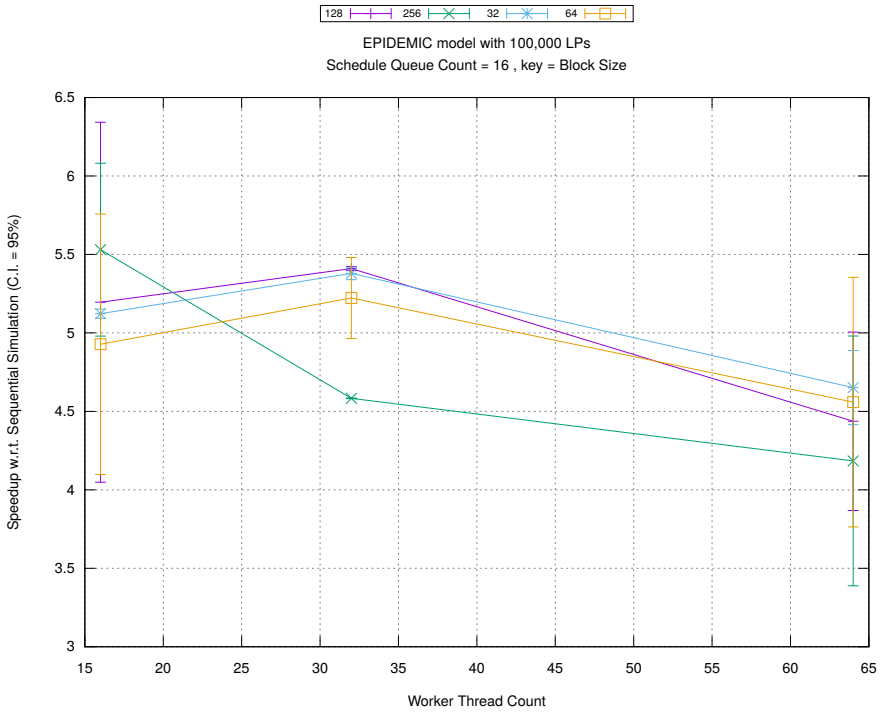


(b) Event Commitment Ratio

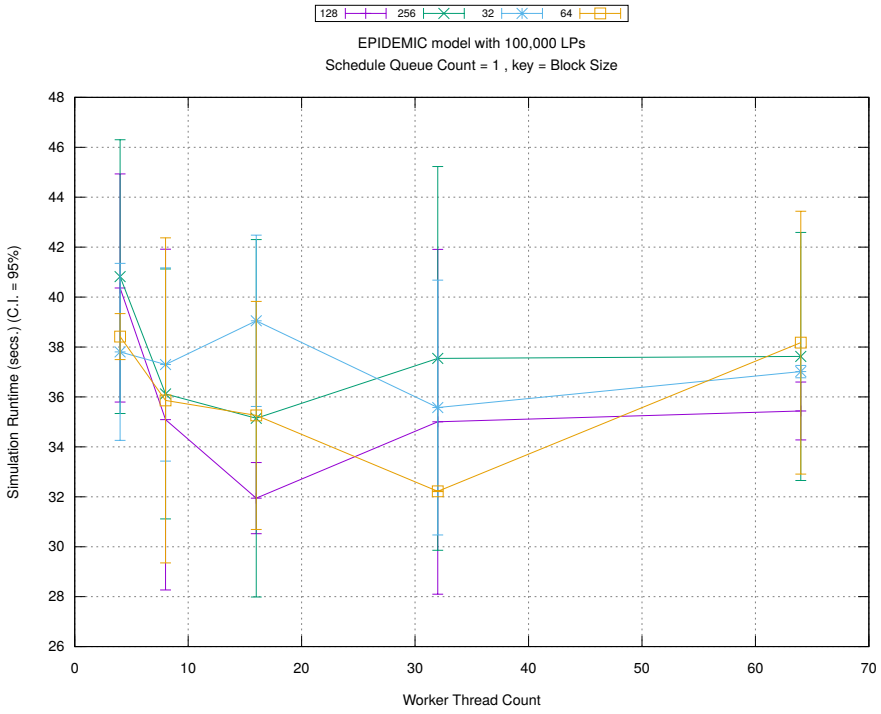


(c) Event Processing Rate (per second)

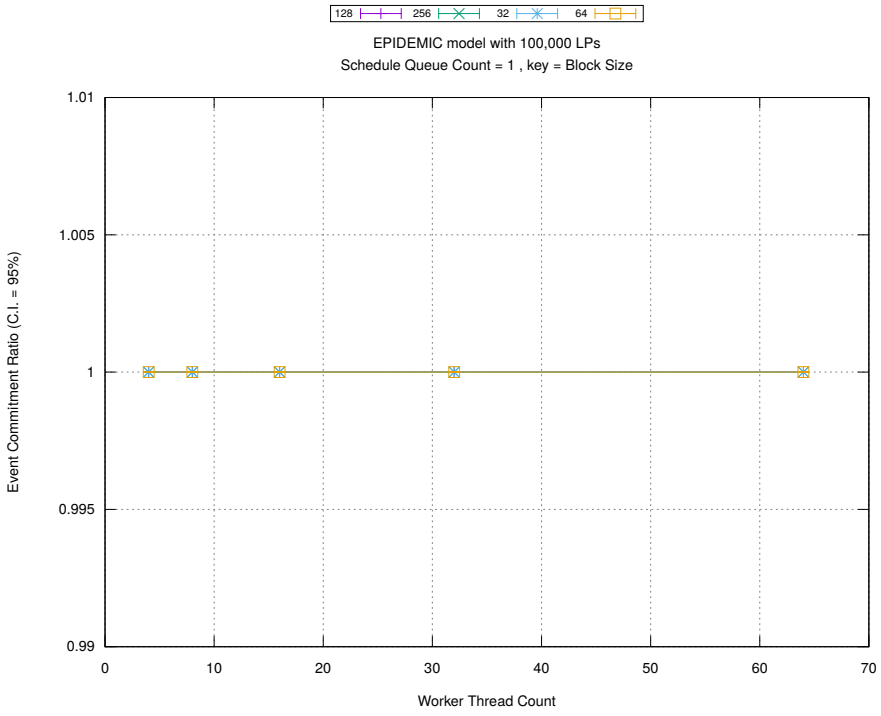
Figure A.131: epidemic 100k ws/plots/blocks/threads vs blocksize key count 16



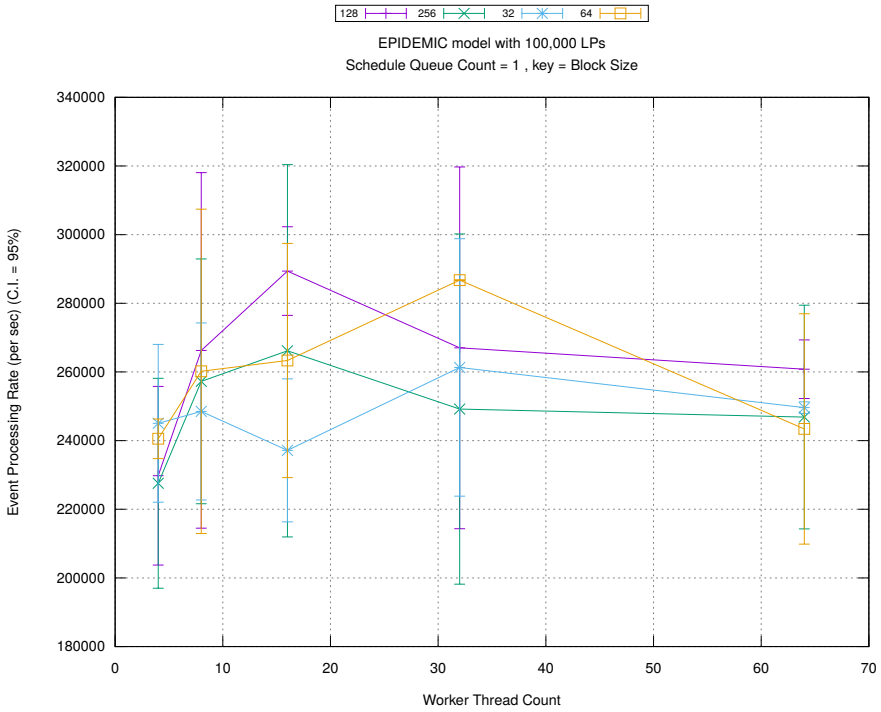
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

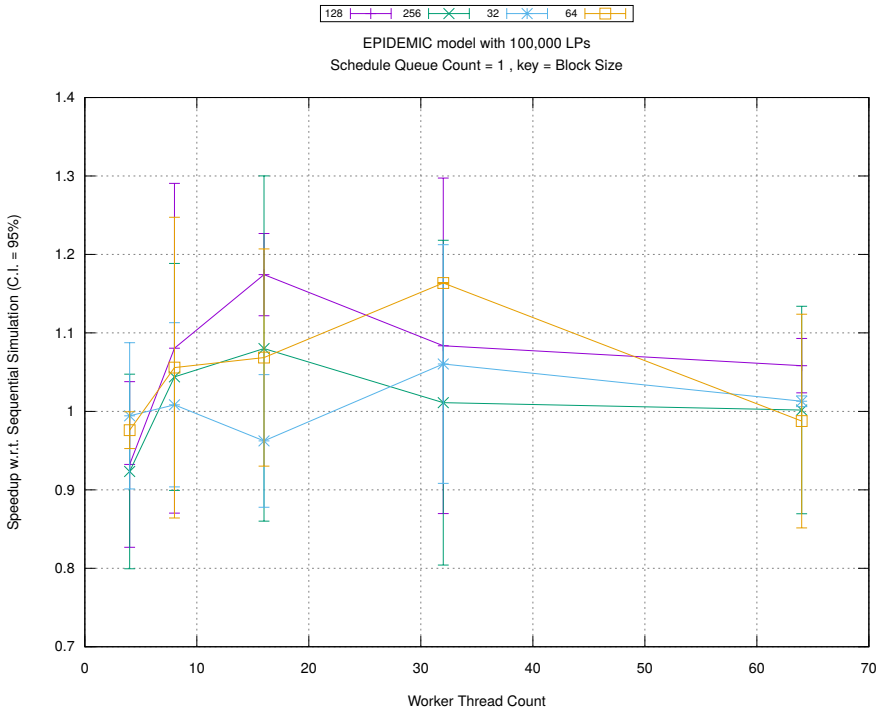


(b) Event Commitment Ratio

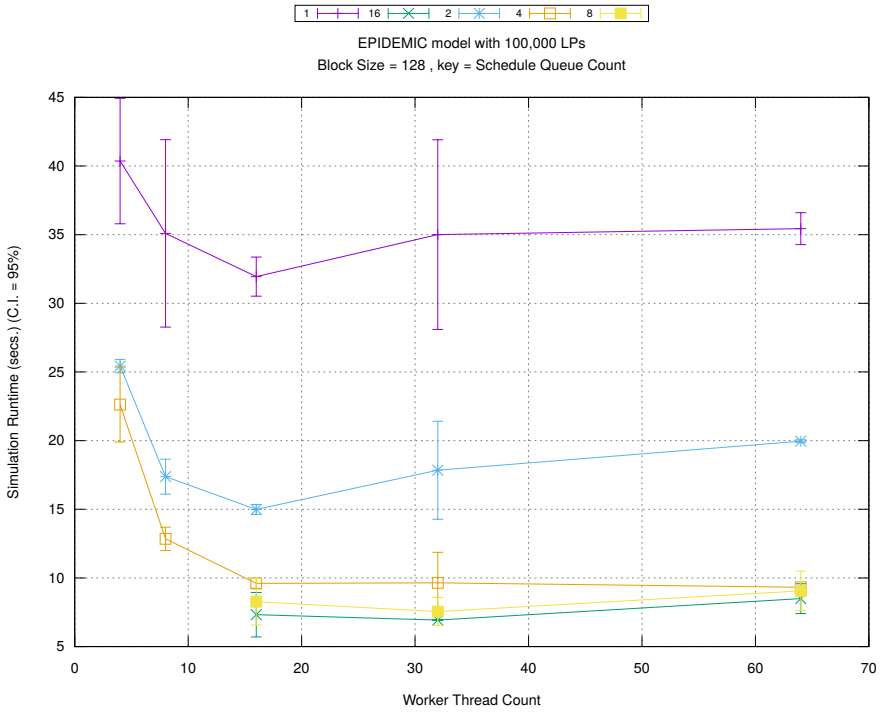


(c) Event Processing Rate (per second)

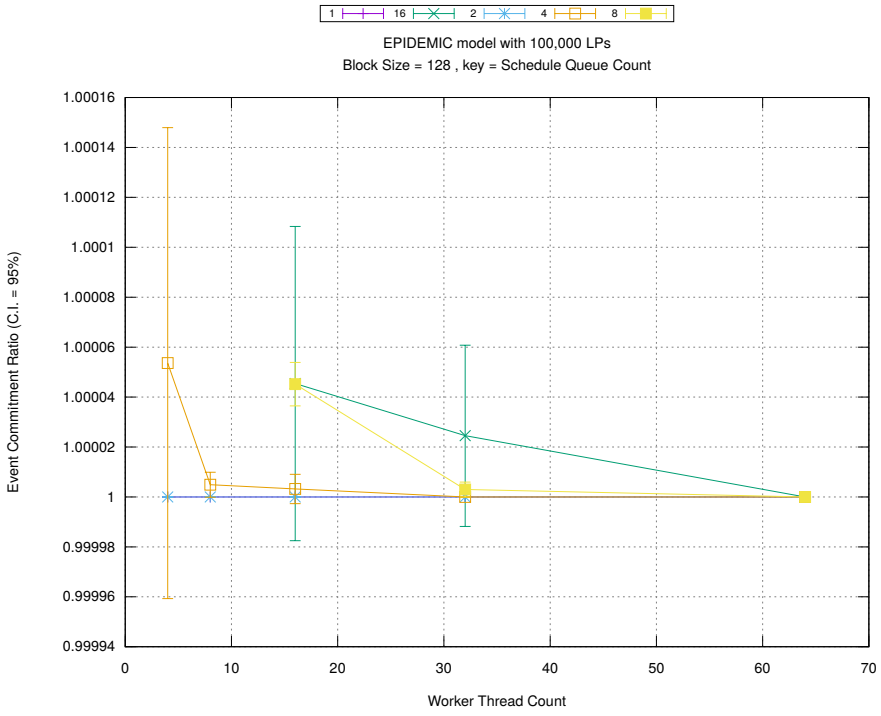
Figure A.132: epidemic 100k ws/plots/blocks/threads vs blocksize key count 1



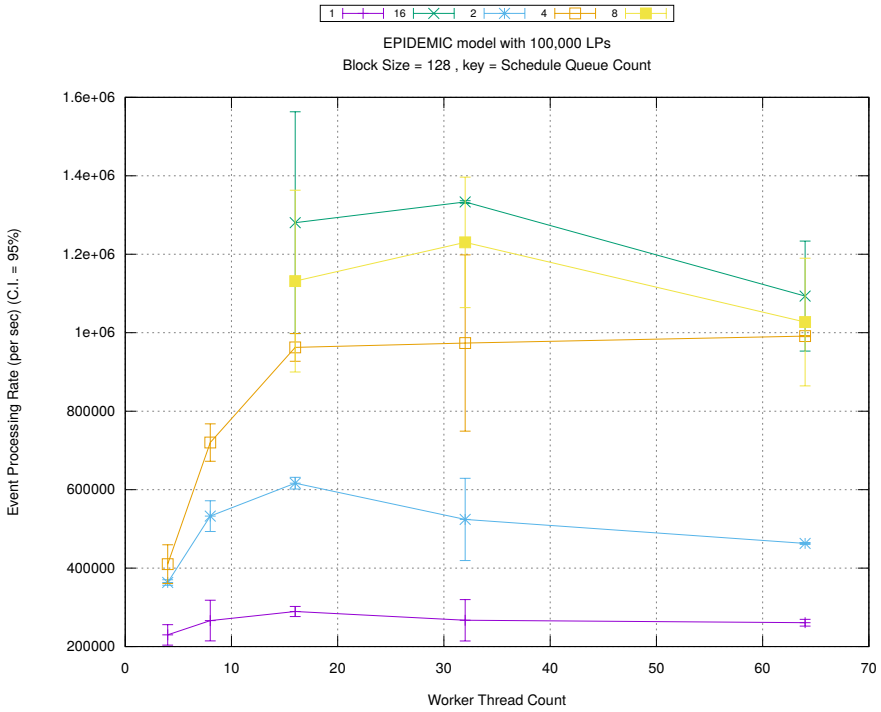
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

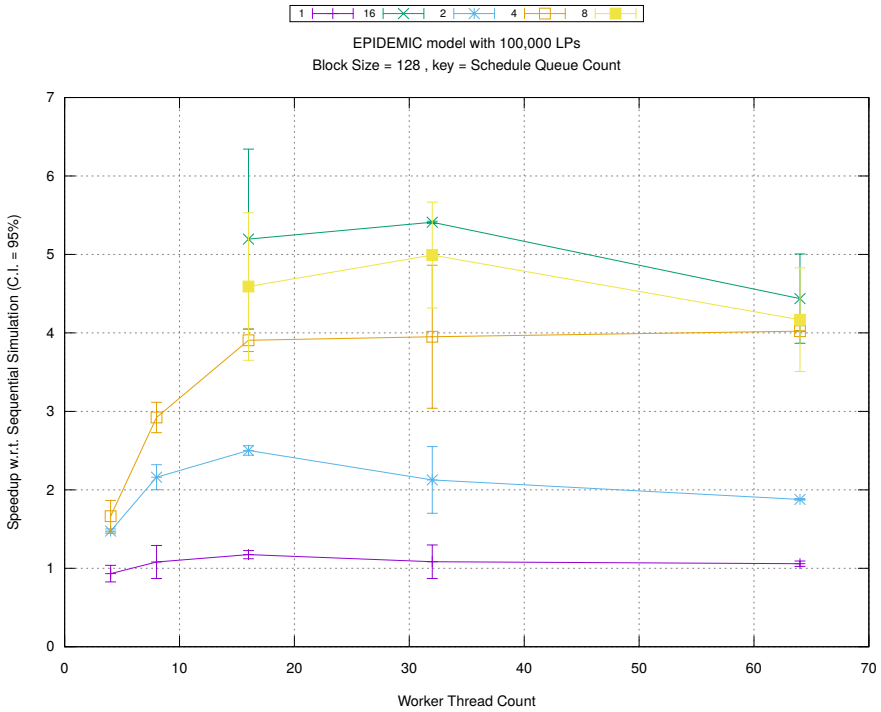


(b) Event Commitment Ratio

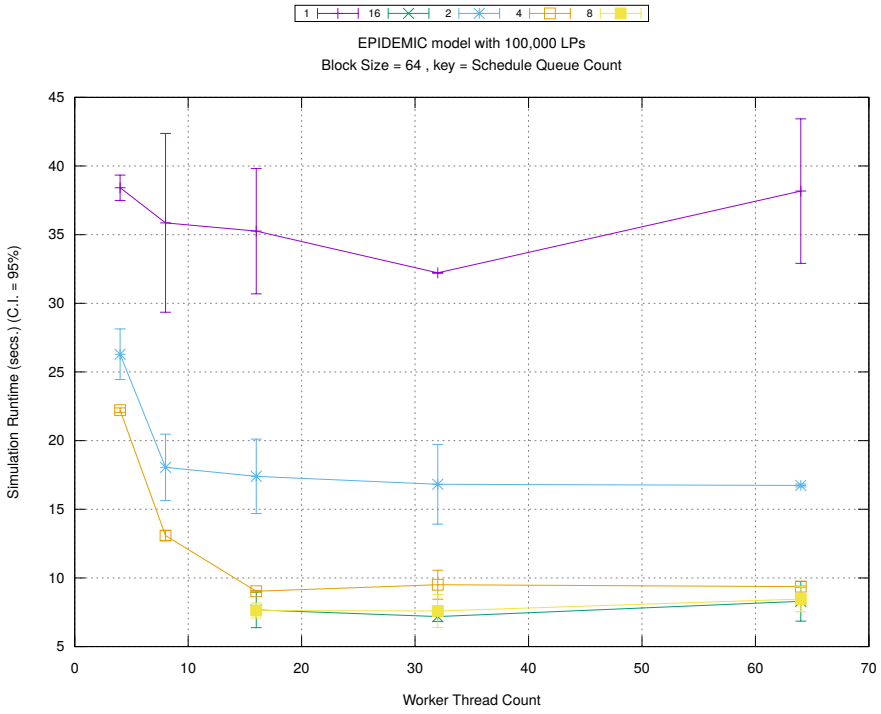


(c) Event Processing Rate (per second)

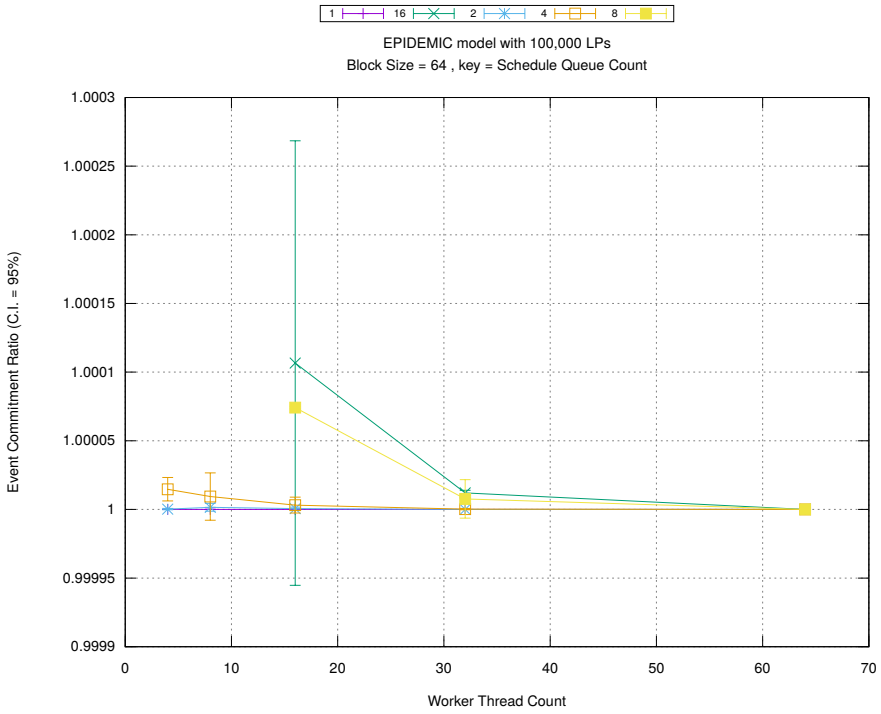
Figure A.133: epidemic 100k ws/plots/blocks/threads vs count key blocksize 128



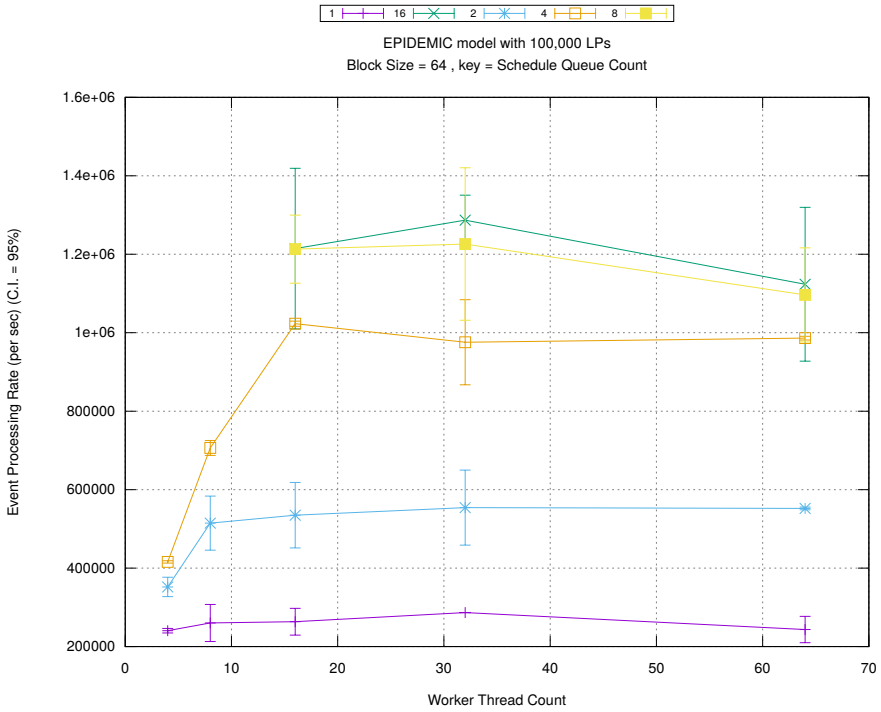
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

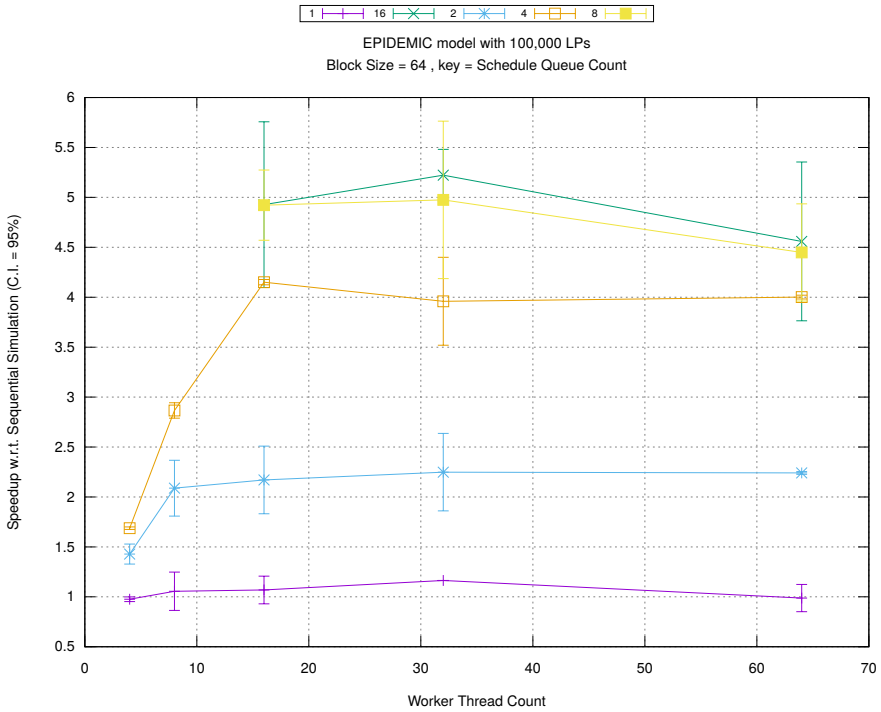


(b) Event Commitment Ratio



(c) Event Processing Rate (per second)

Figure A.134: epidemic 100k ws/plots/blocks/threads vs count key blocksize 64



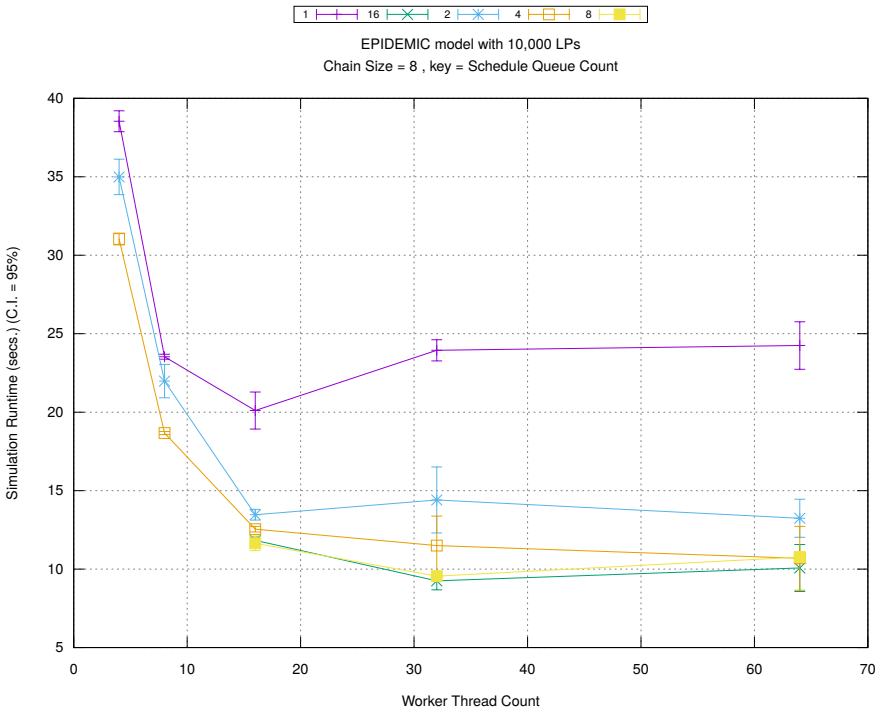
(d) Speedup w.r.t. Sequential Simulation

A.5 Epidemic Model with 10,000 LPs and Barabasi-Albert Network

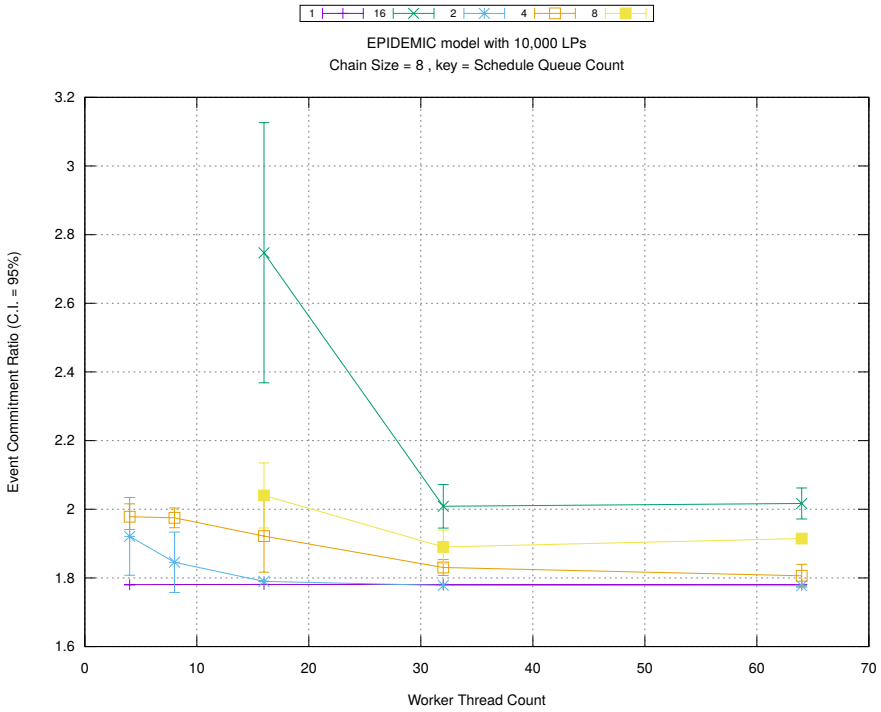
Table A.5 shows the configuration for this model.

Parameter	Values
Number of Intersections (or LPs)	10,000
Type of network connecting LPs	Barabasi-Albert [73]
Population Size	1,000,000
Simulation Time	15,000 timestamp units
Sequential Simulation Time for calculating modularity	8,000 timestamp units

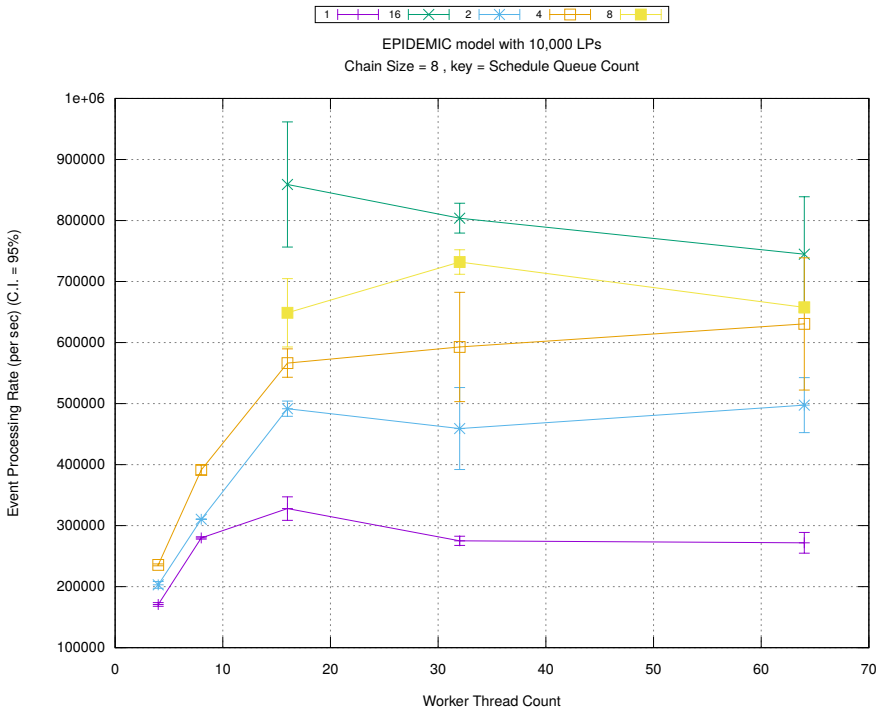
Table A.5: EPIDEMIC MODEL WITH BARABASI-ALBERT setup



(a) Simulation Runtime (in seconds)

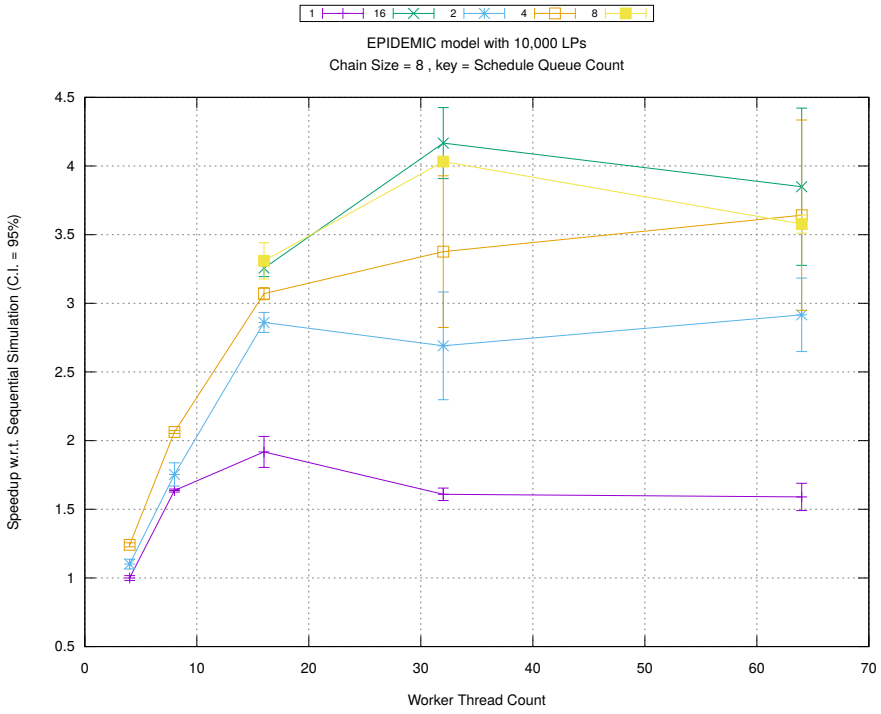


(b) Event Commitment Ratio

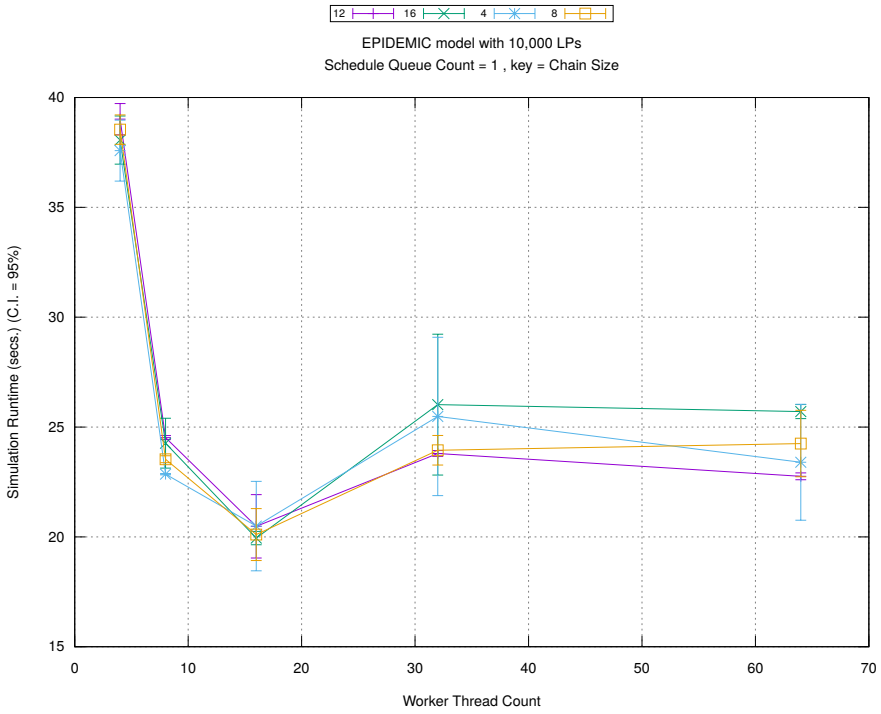


(c) Event Processing Rate (per second)

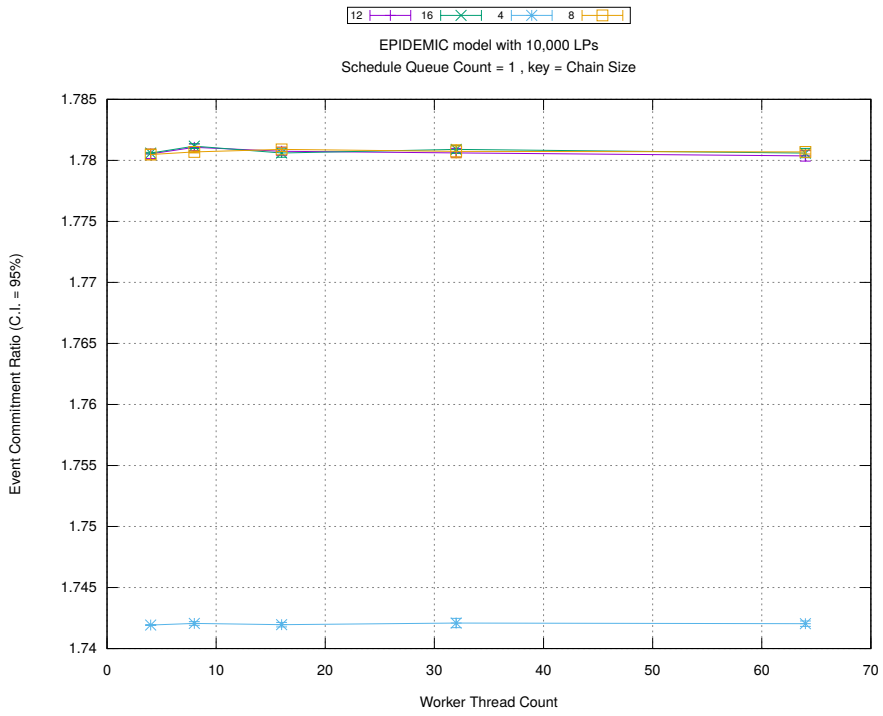
Figure A.135: epidemic 10k ba/plots/chains/threads vs count key chainsize 8



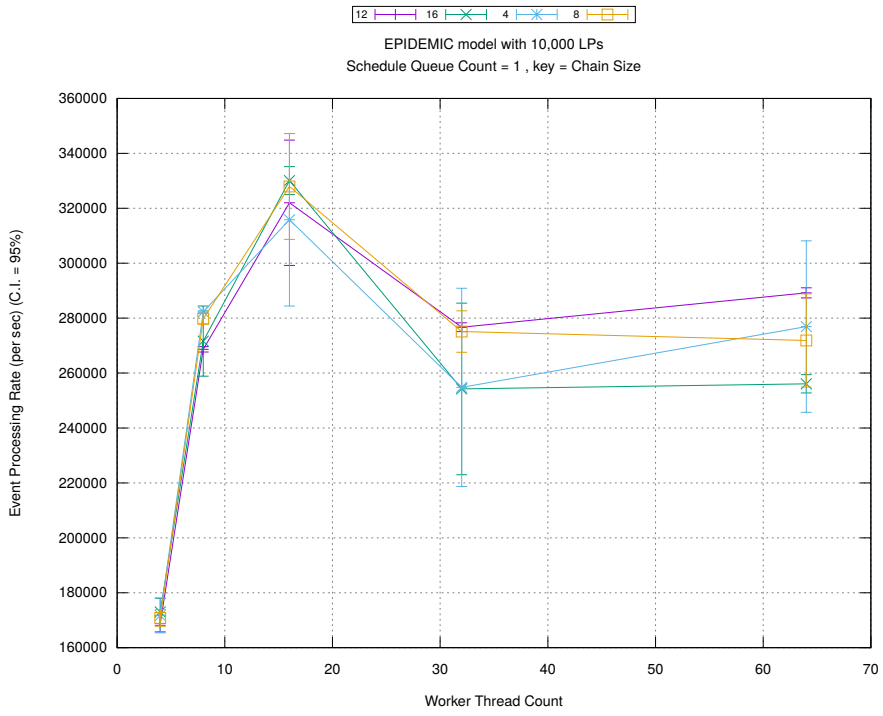
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

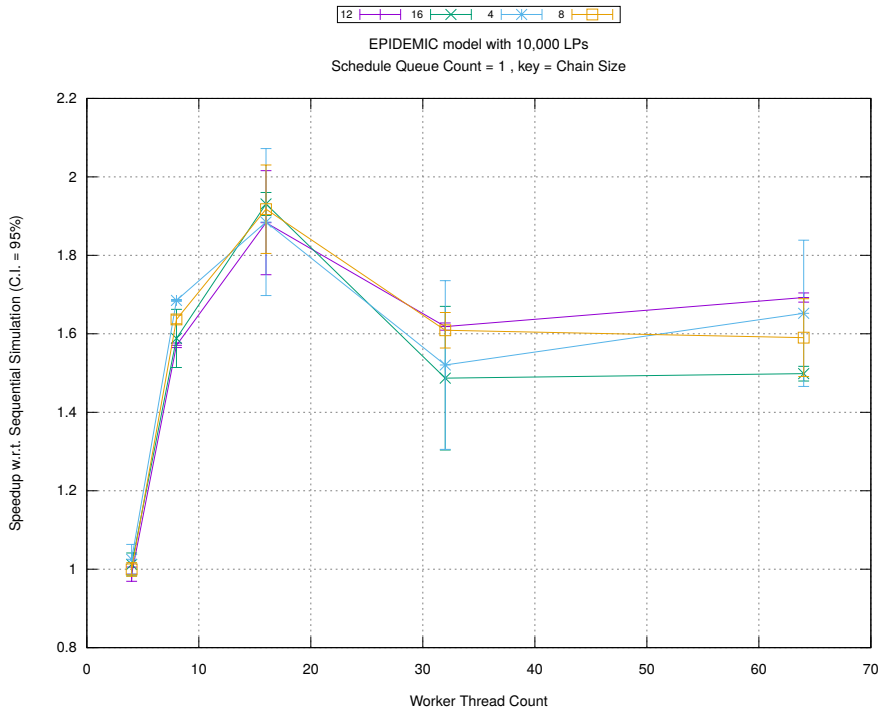


(b) Event Commitment Ratio

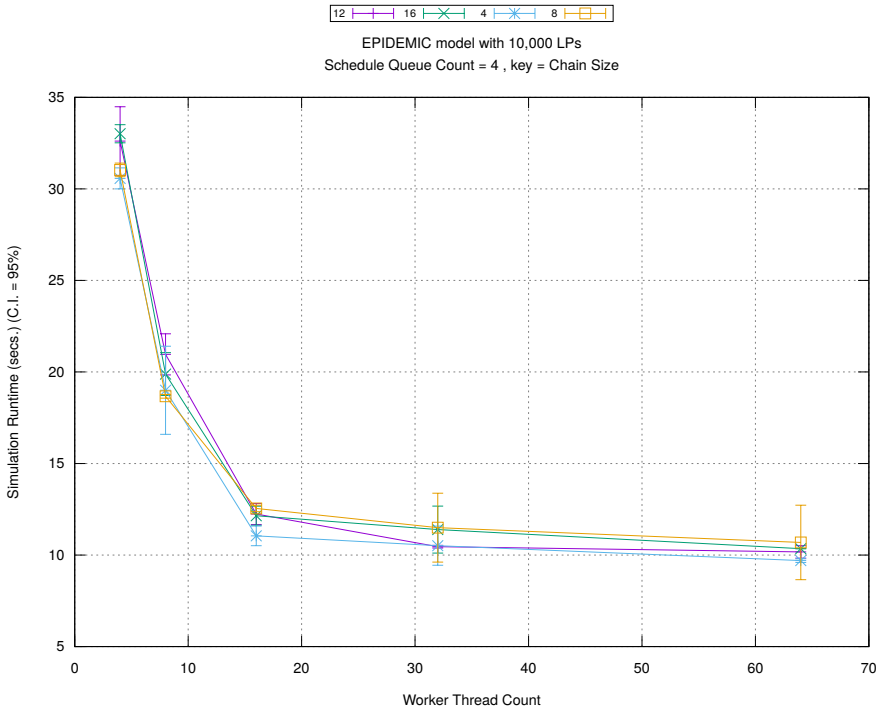


(c) Event Processing Rate (per second)

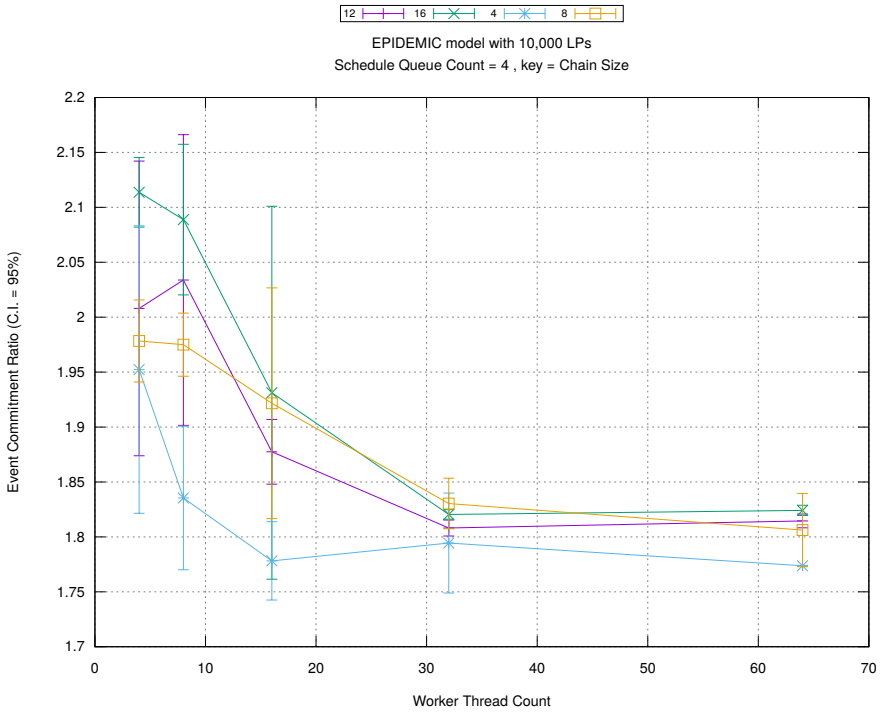
Figure A.136: epidemic 10k ba/plots/chains/threads vs chainsize key count 1



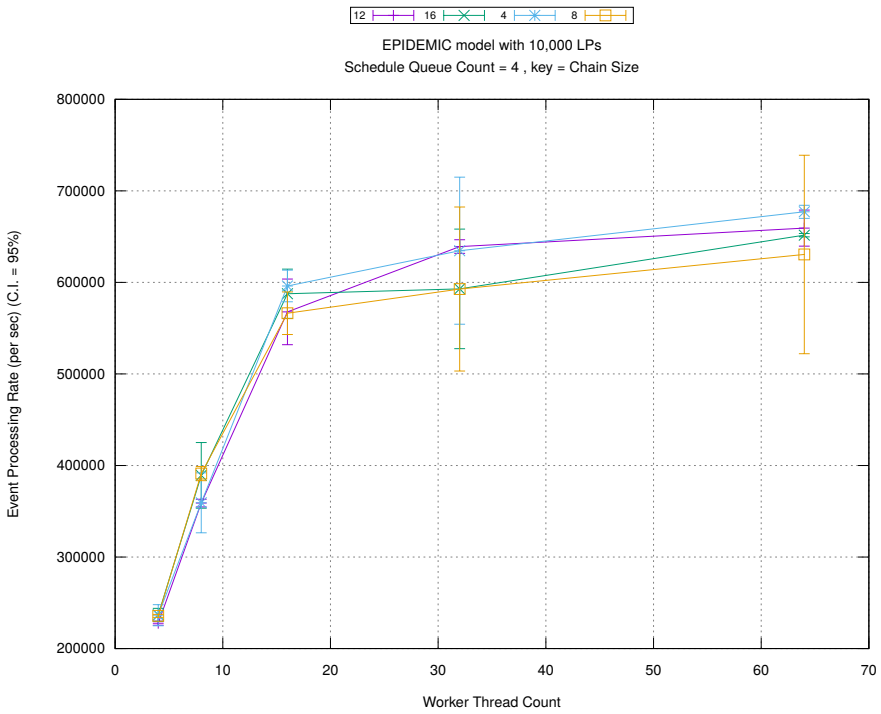
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

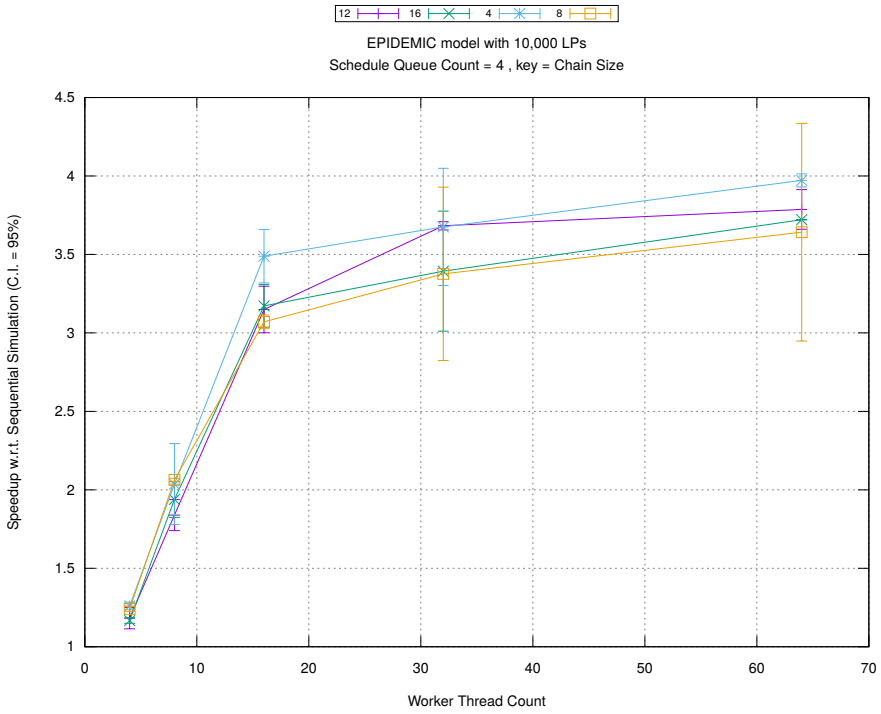


(b) Event Commitment Ratio

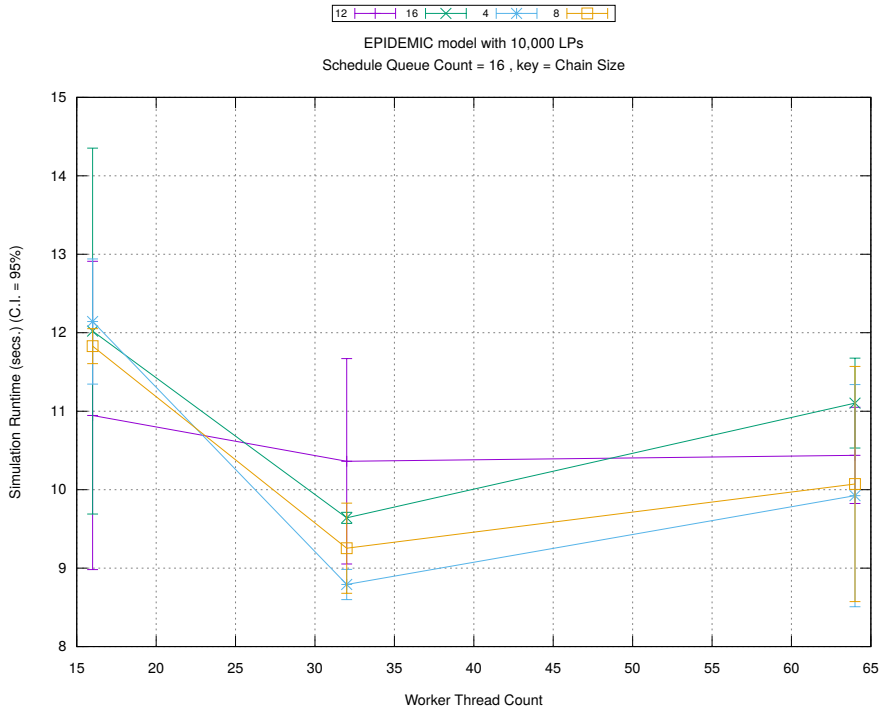


(c) Event Processing Rate (per second)

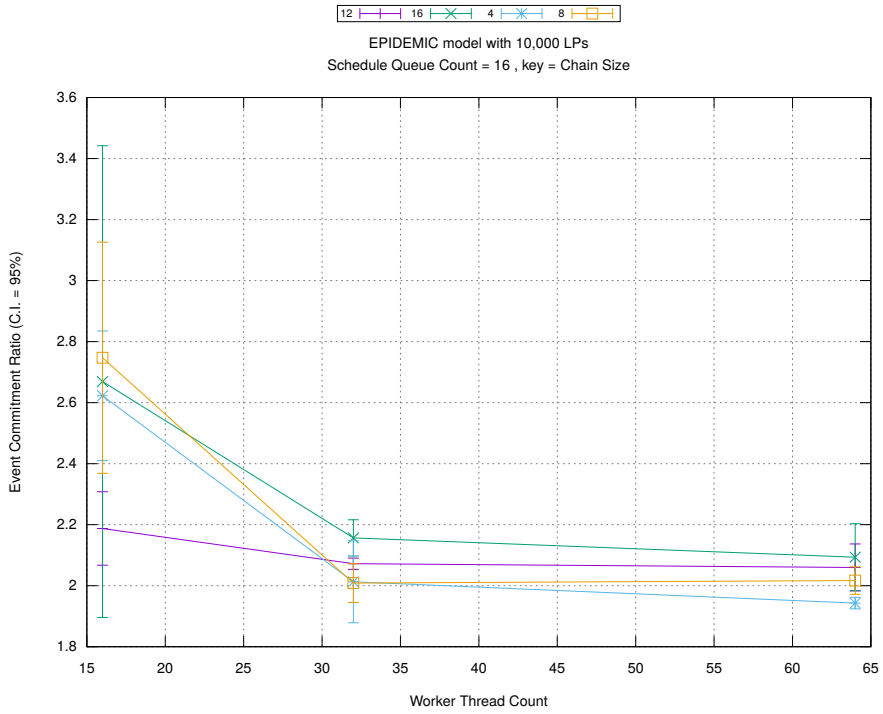
Figure A.137: epidemic 10k ba/plots/chains/threads vs chainsize key count 4



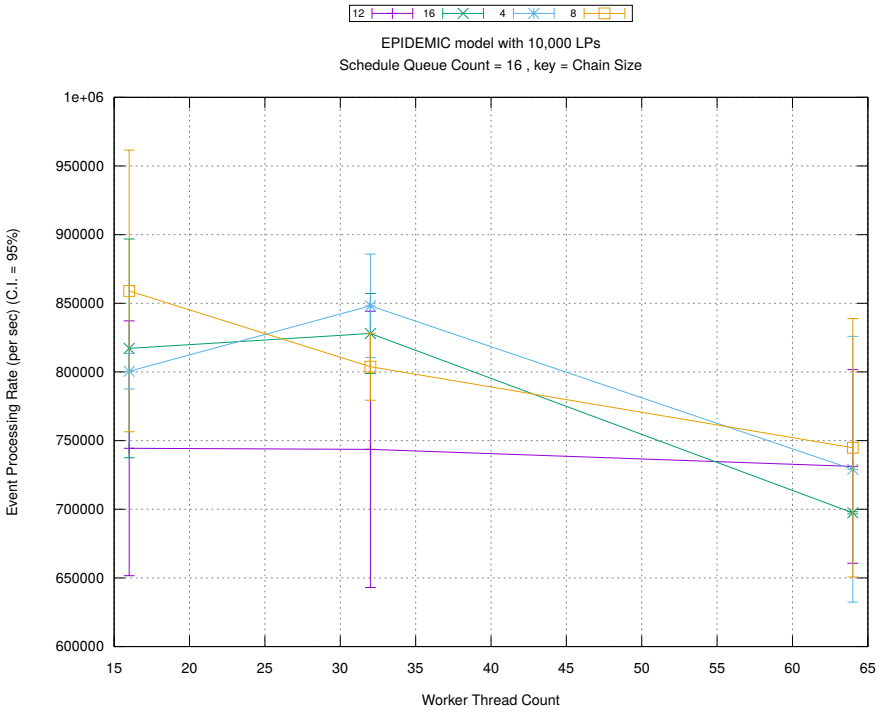
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

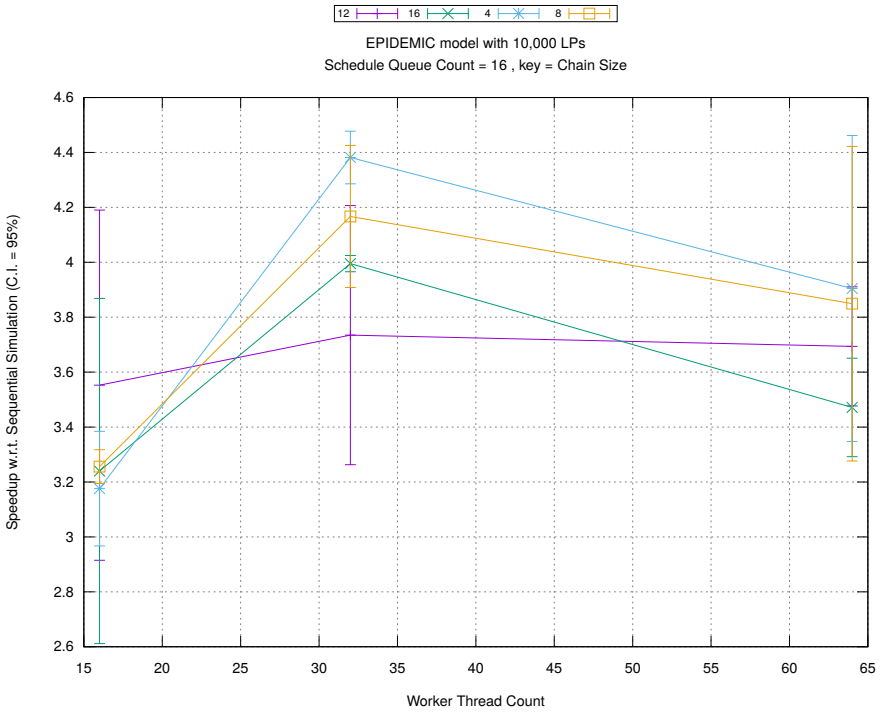


(b) Event Commitment Ratio

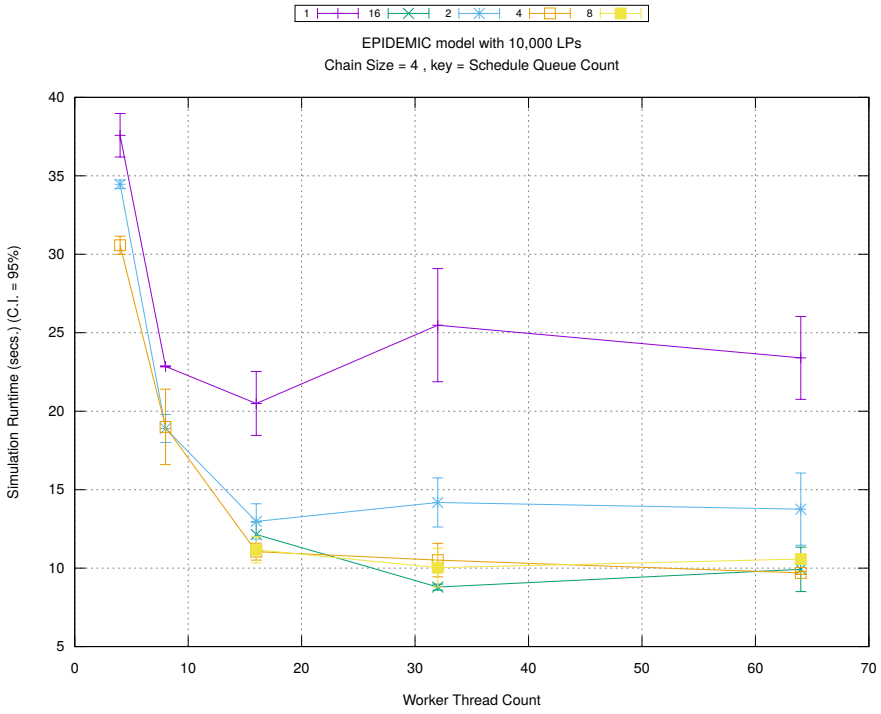


(c) Event Processing Rate (per second)

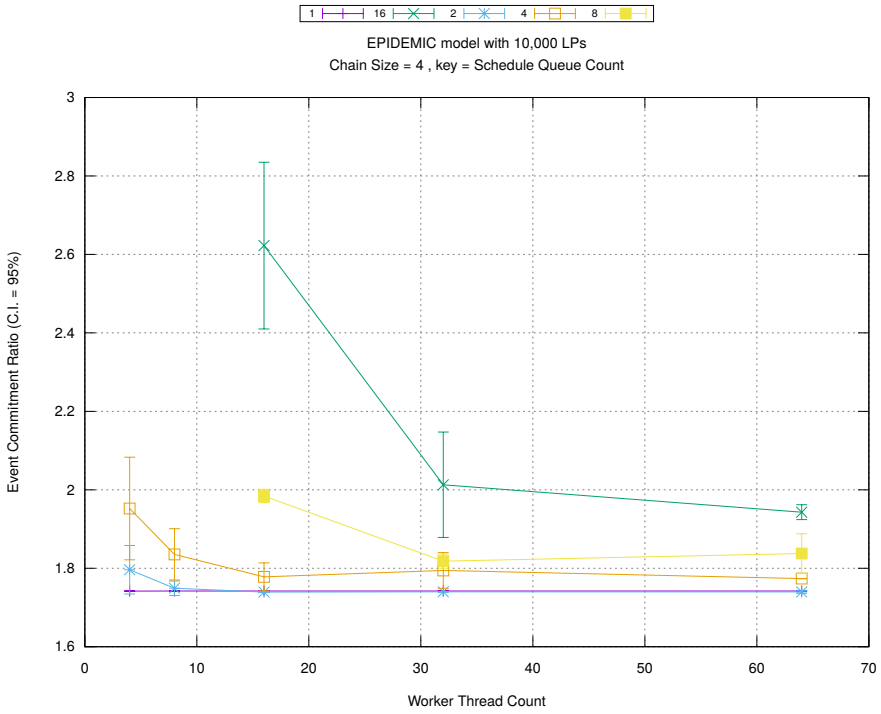
Figure A.138: epidemic 10k ba/plots/chains/threads vs chainsize key count 16



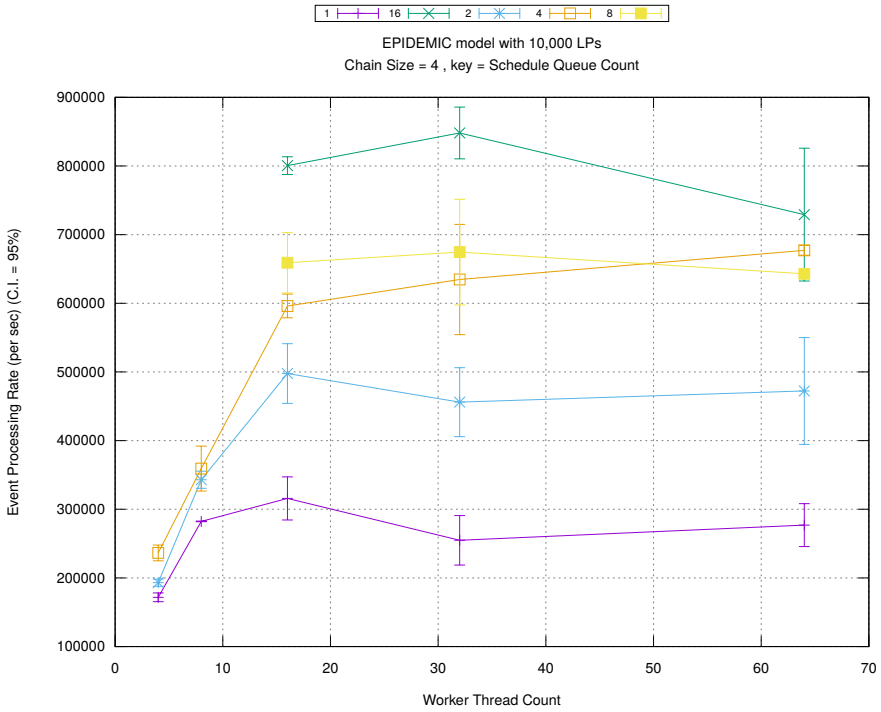
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

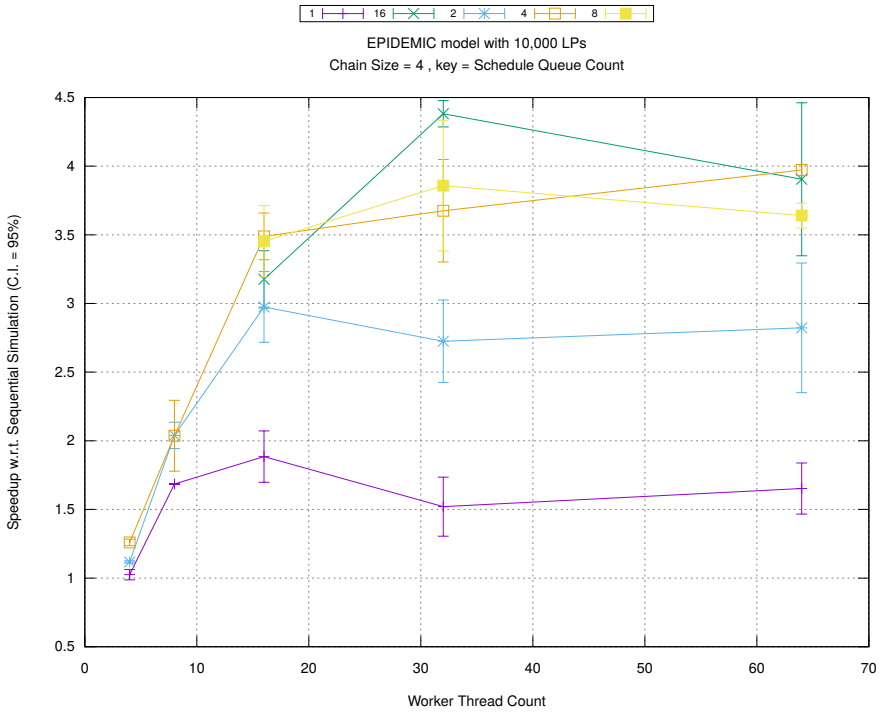


(b) Event Commitment Ratio

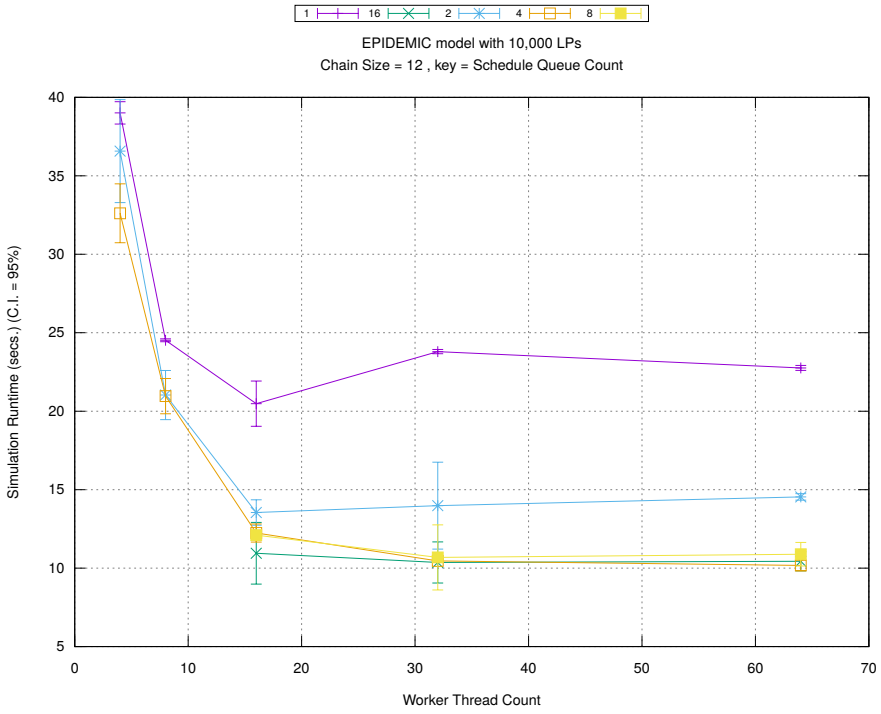


(c) Event Processing Rate (per second)

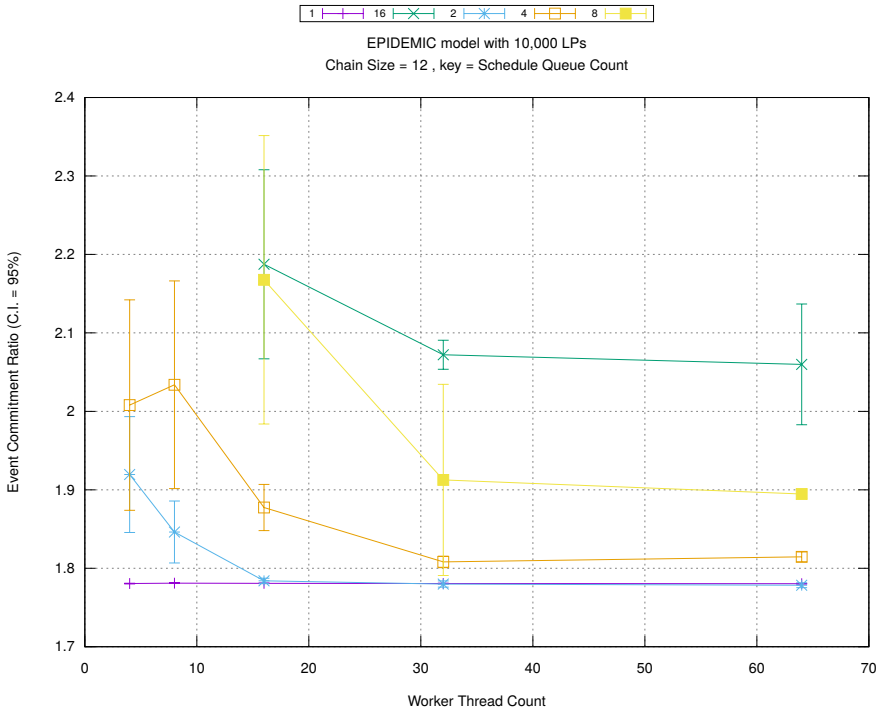
Figure A.139: epidemic 10k ba/plots/chains/threads vs count key chainsize 4



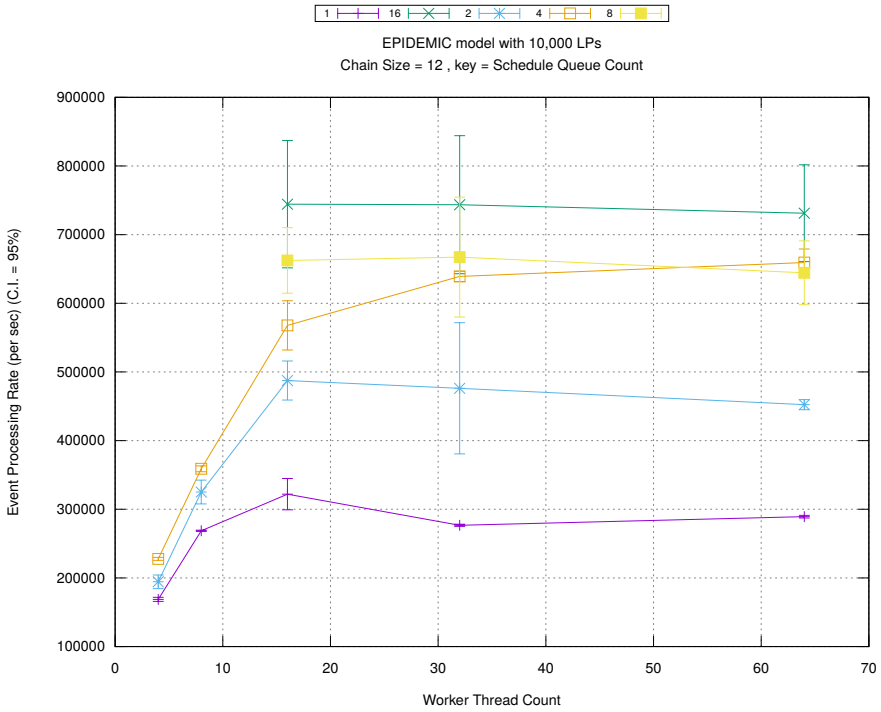
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

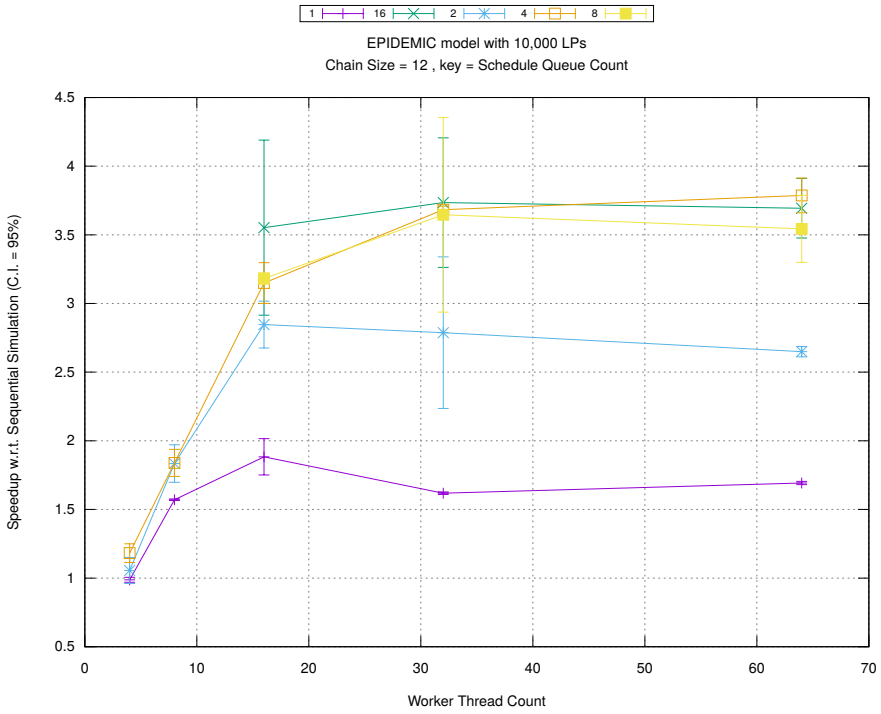


(b) Event Commitment Ratio

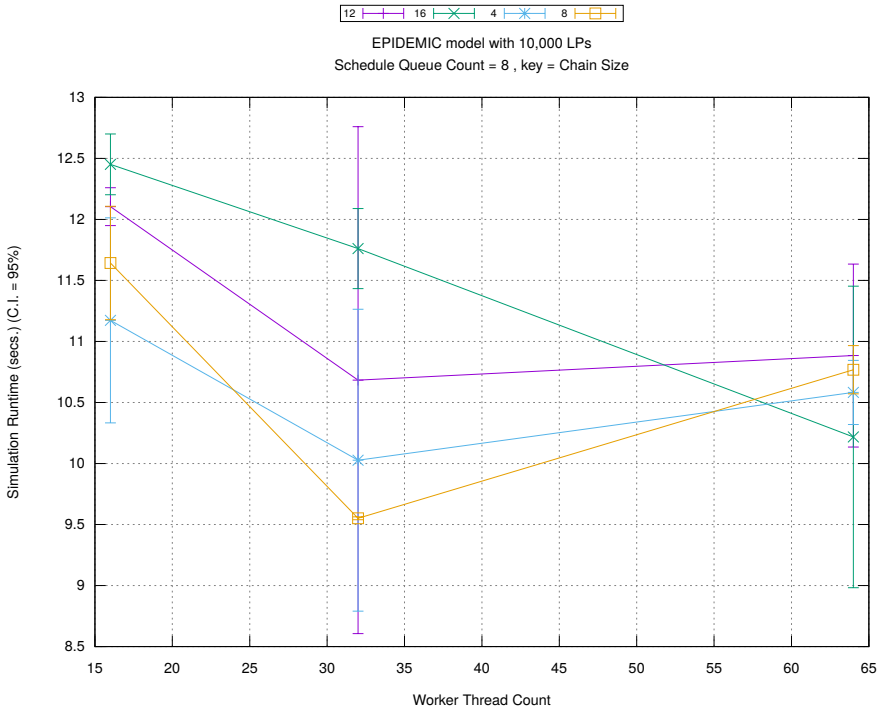


(c) Event Processing Rate (per second)

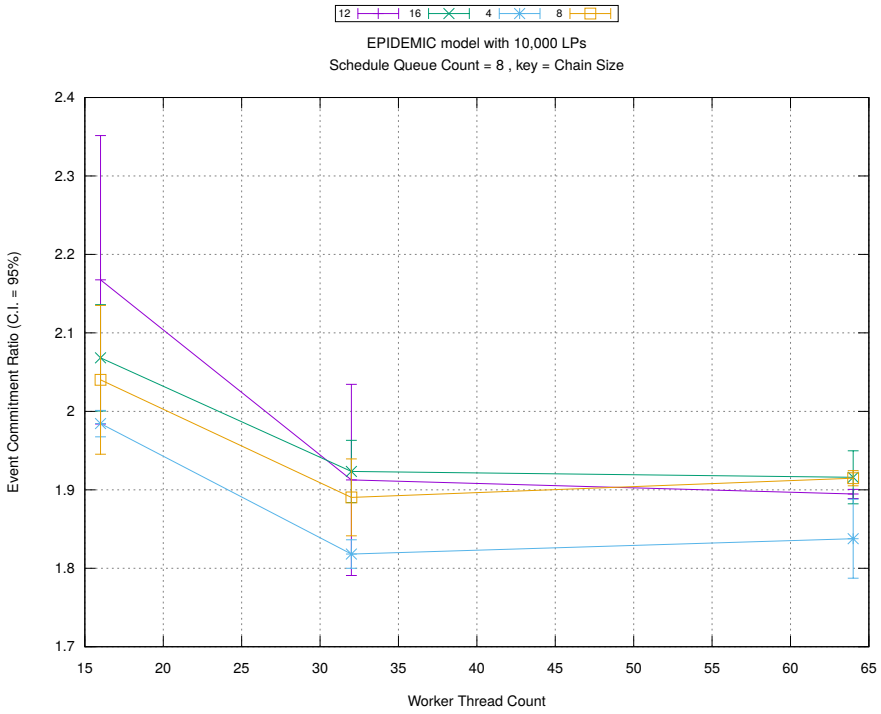
Figure A.140: epidemic 10k ba/plots/chains/threads vs count key chainsize 12



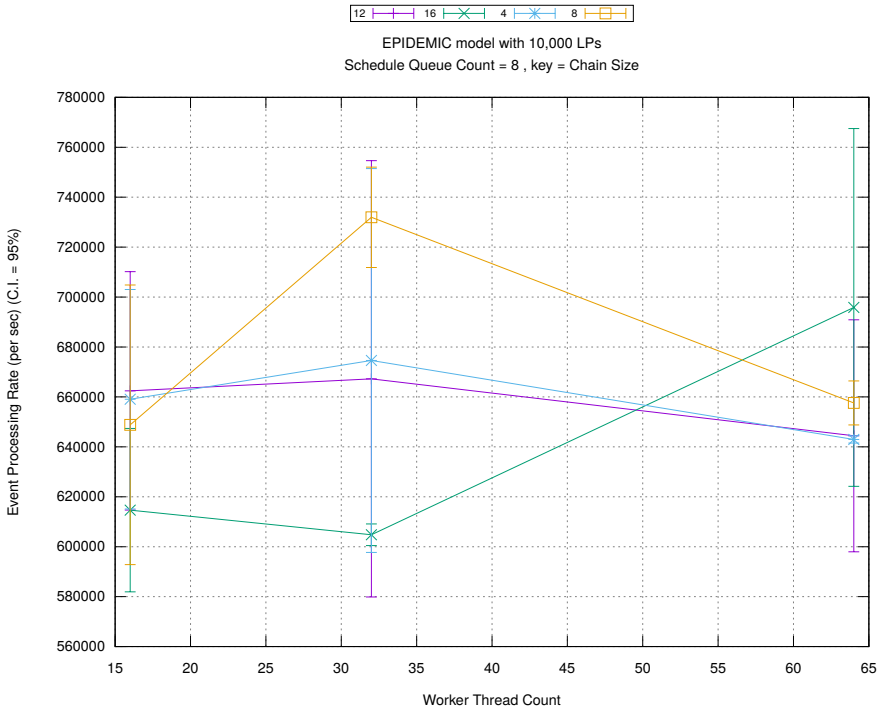
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

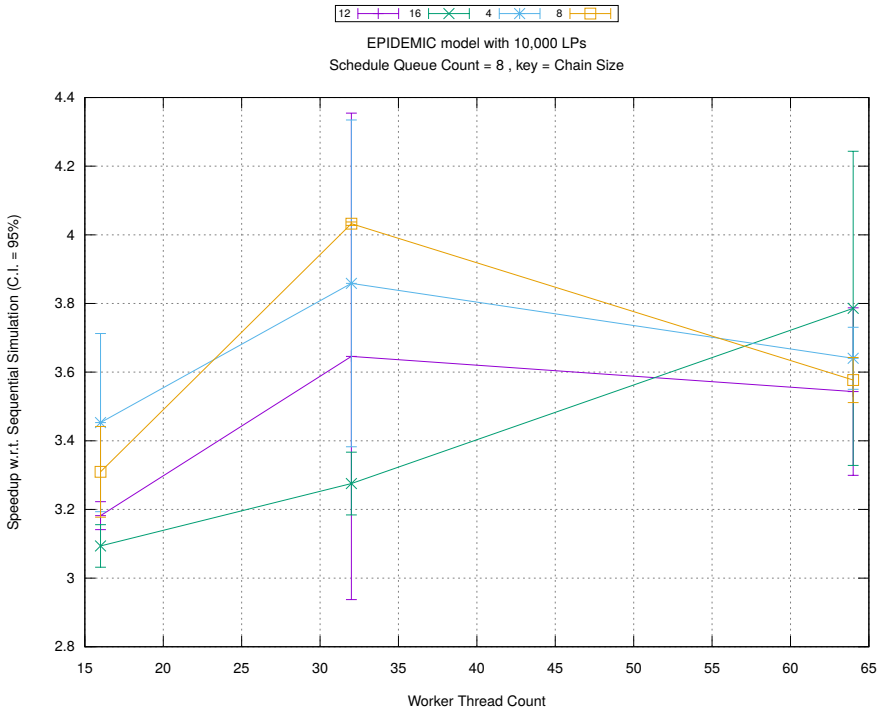


(b) Event Commitment Ratio

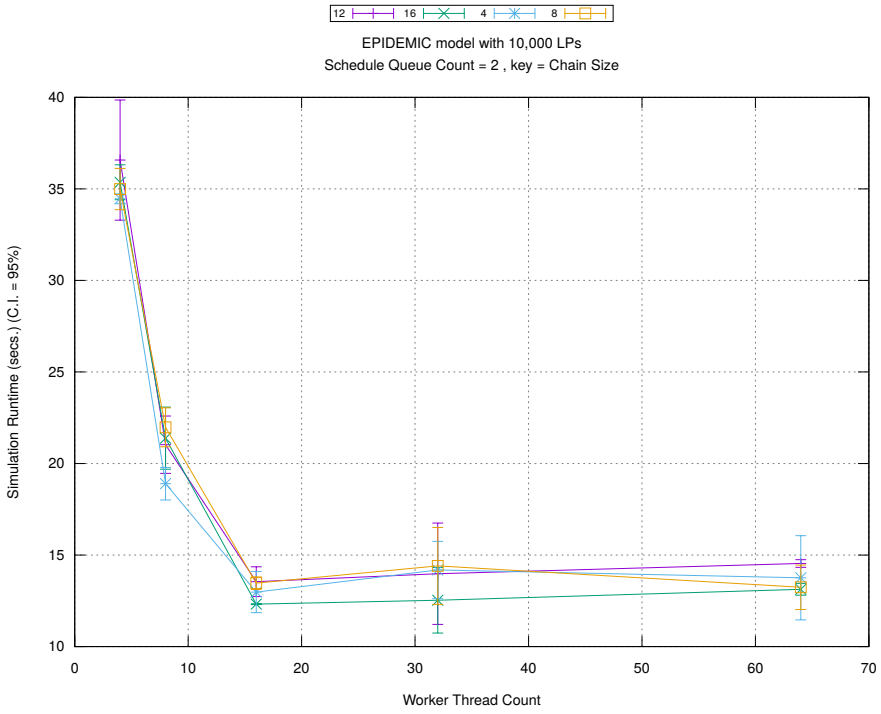


(c) Event Processing Rate (per second)

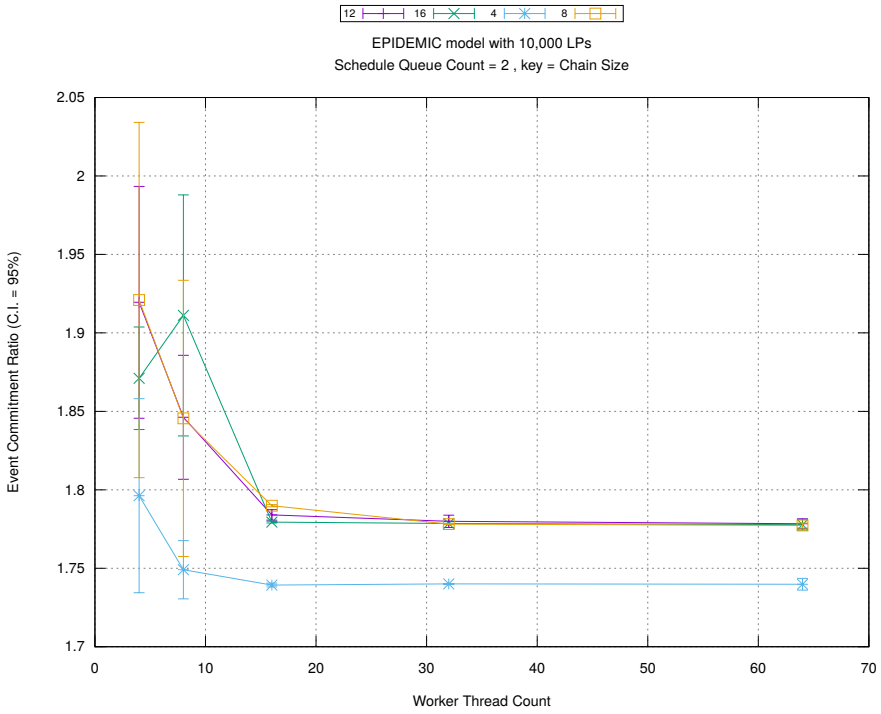
Figure A.141: epidemic 10k ba/plots/chains/threads vs chainsize key count 8



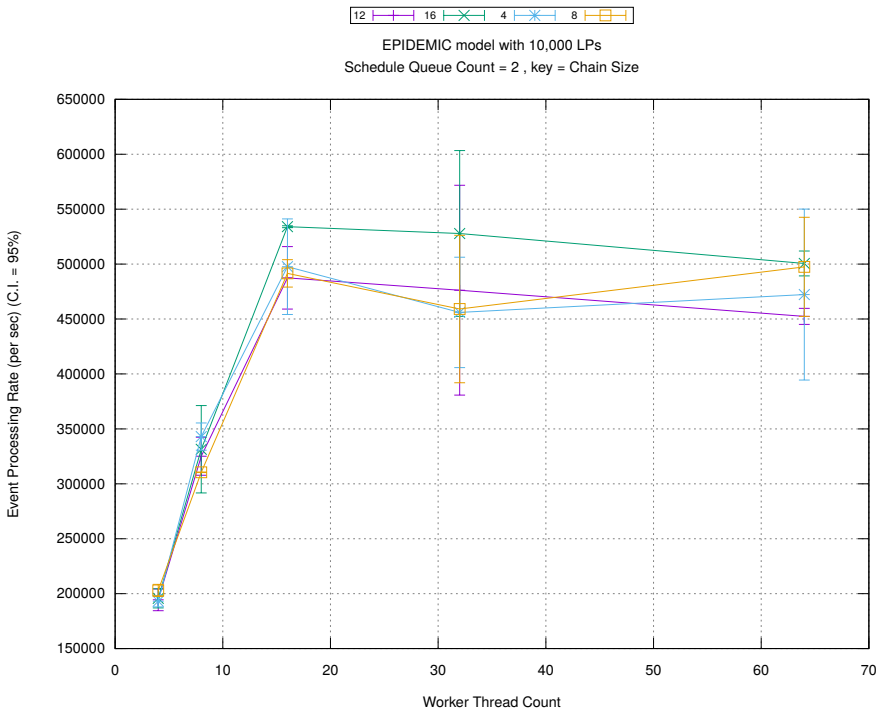
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

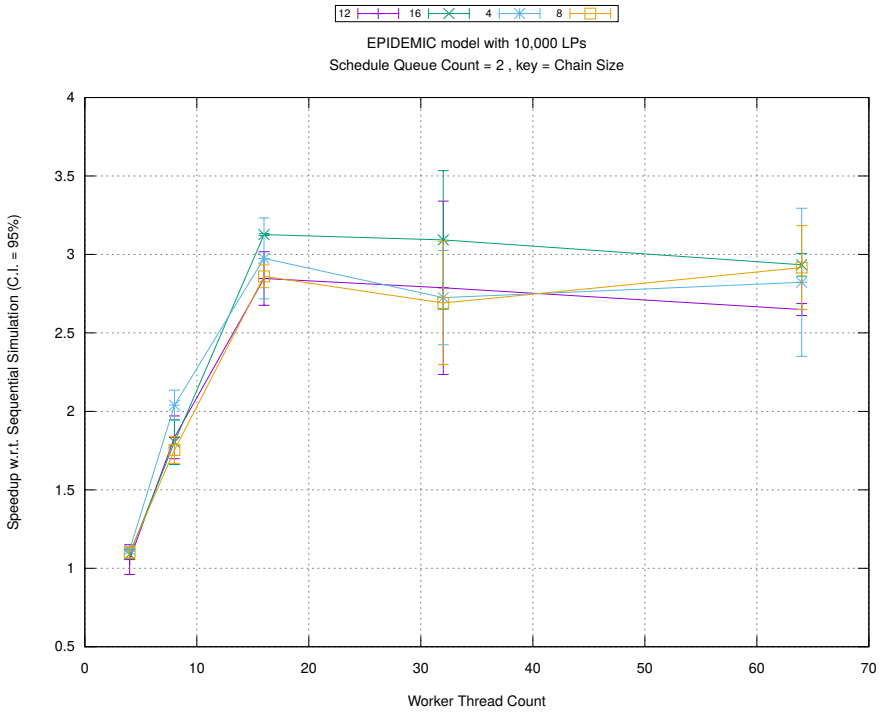


(b) Event Commitment Ratio

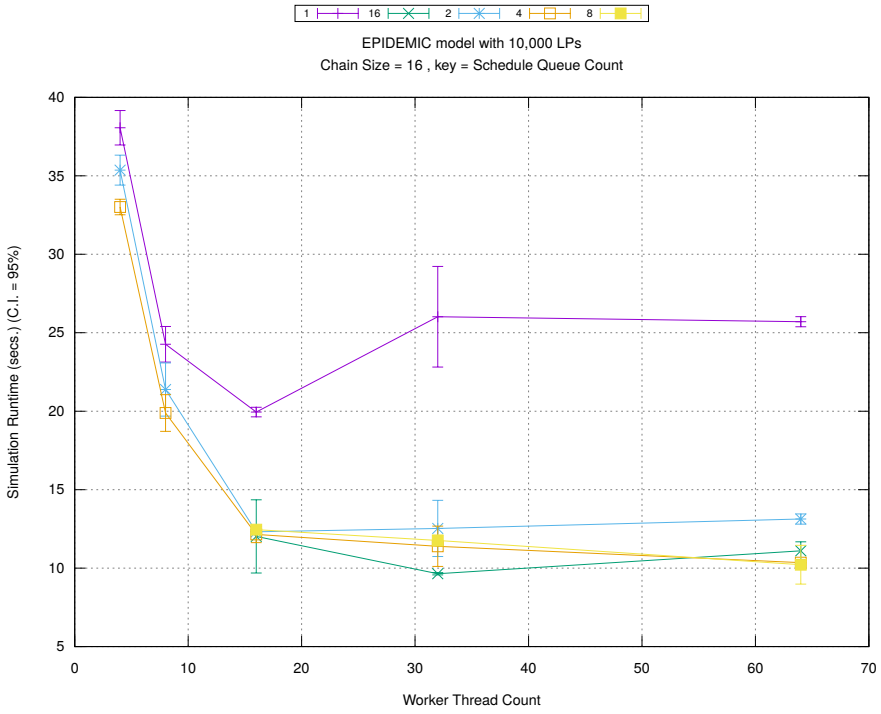


(c) Event Processing Rate (per second)

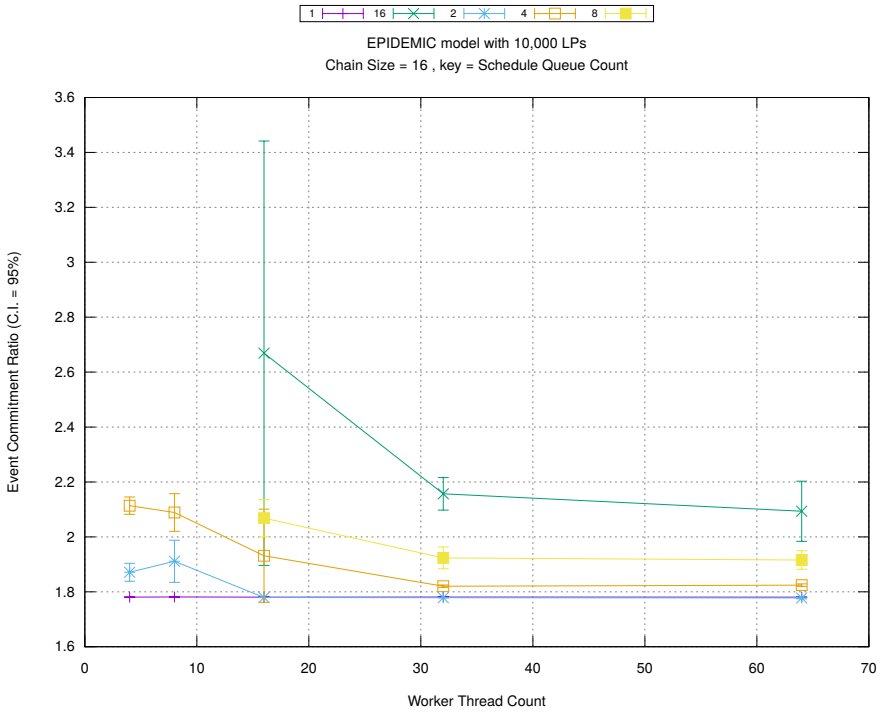
Figure A.142: epidemic 10k ba/plots/chains/threads vs chainsize key count 2



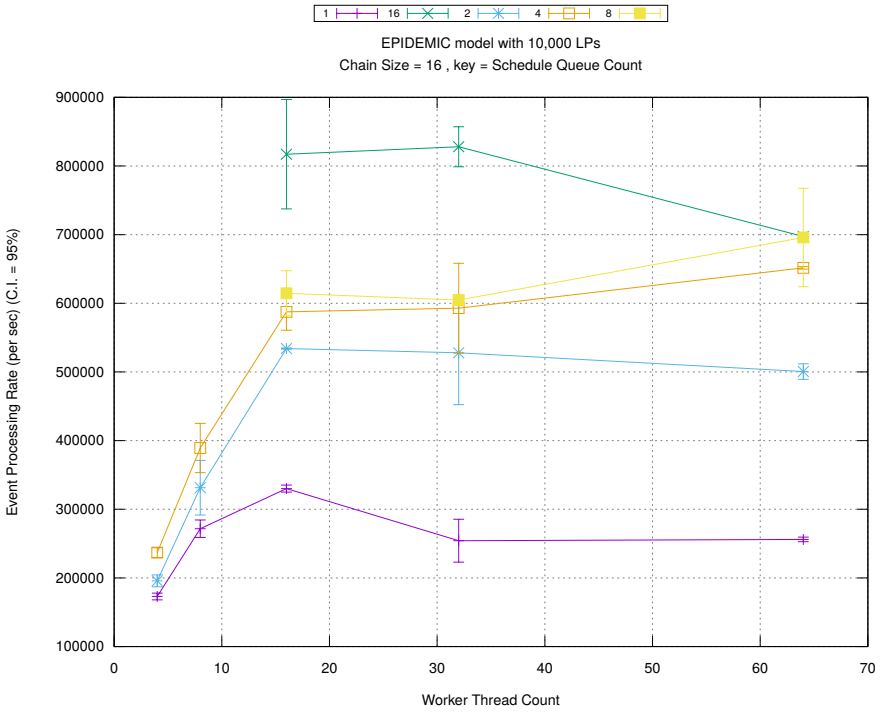
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

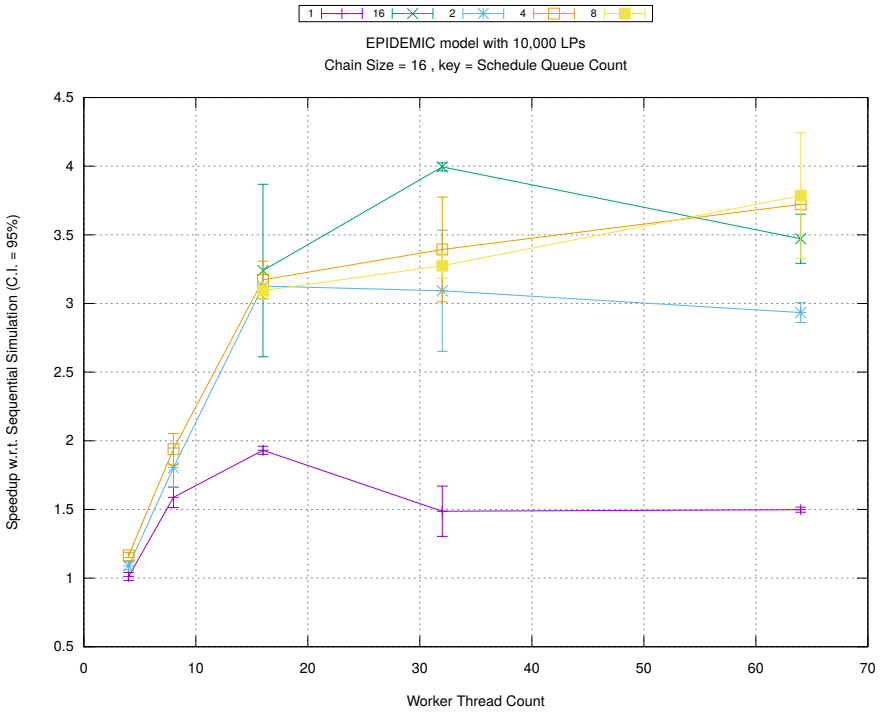


(b) Event Commitment Ratio

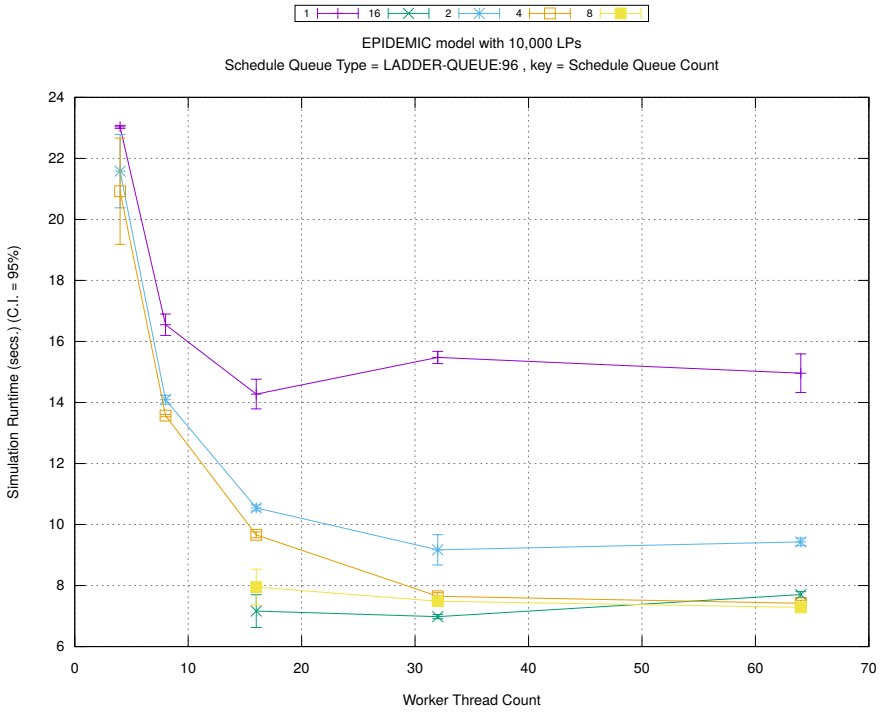


(c) Event Processing Rate (per second)

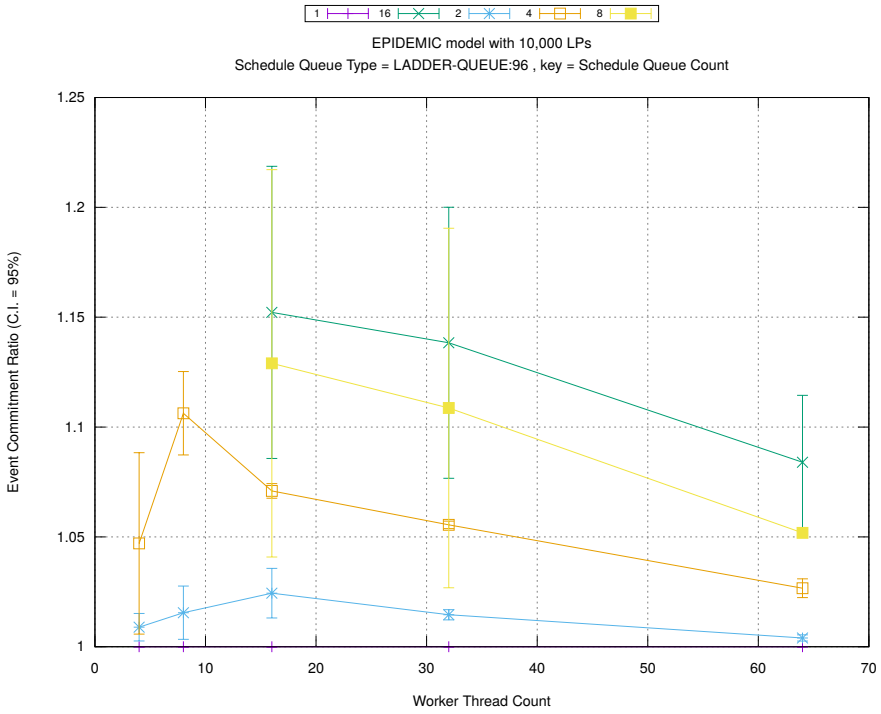
Figure A.143: epidemic 10k ba/plots/chains/threads vs count key chainsize 16



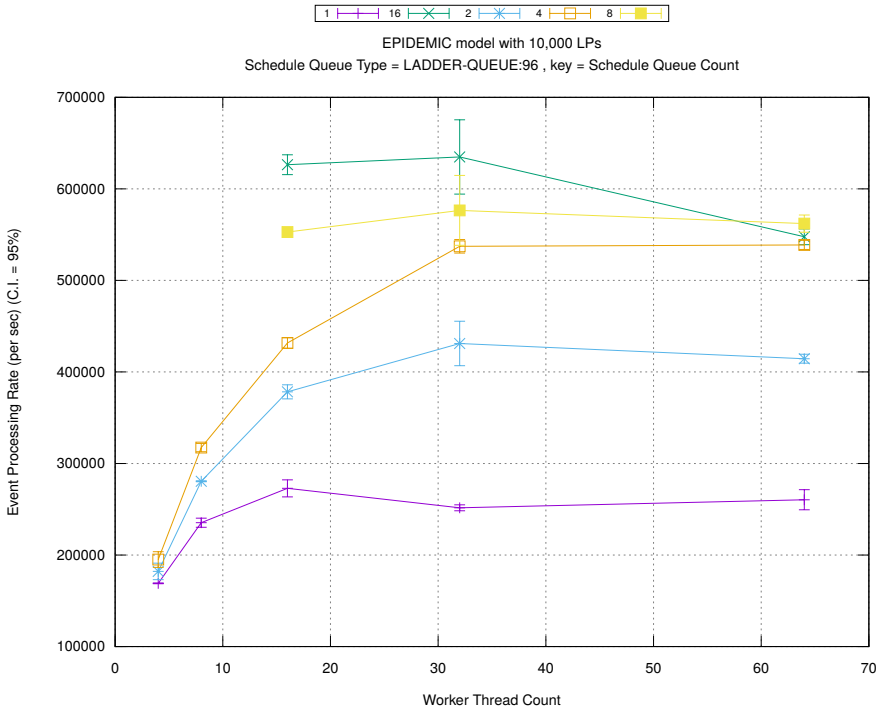
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

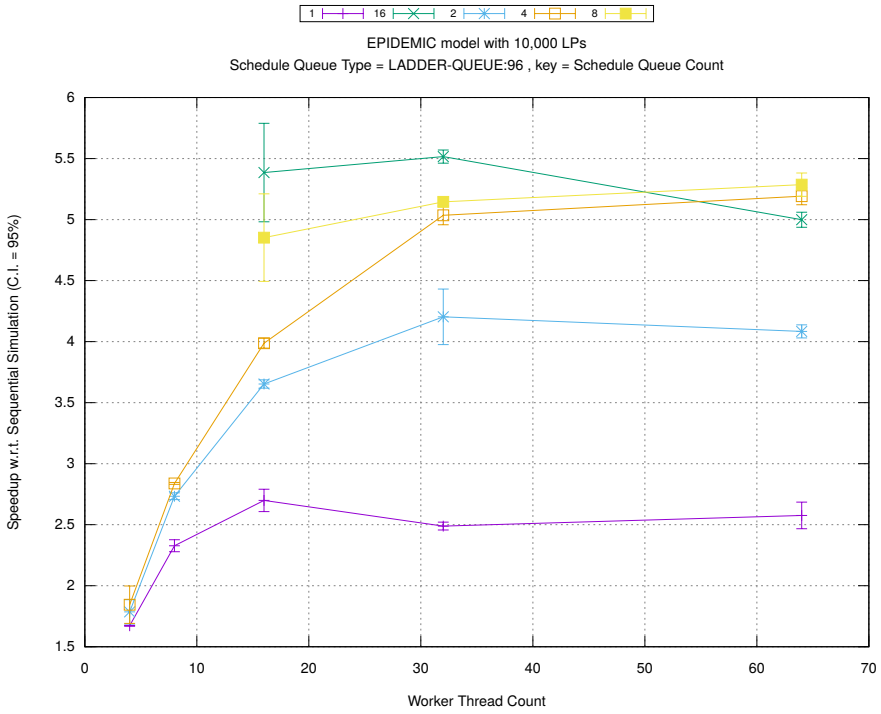


(b) Event Commitment Ratio

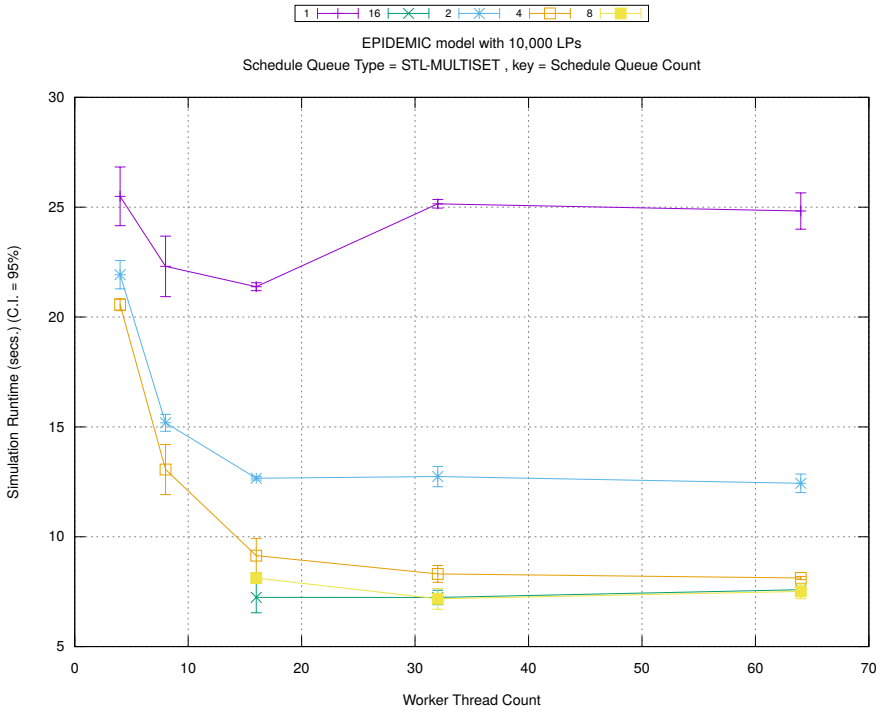


(c) Event Processing Rate (per second)

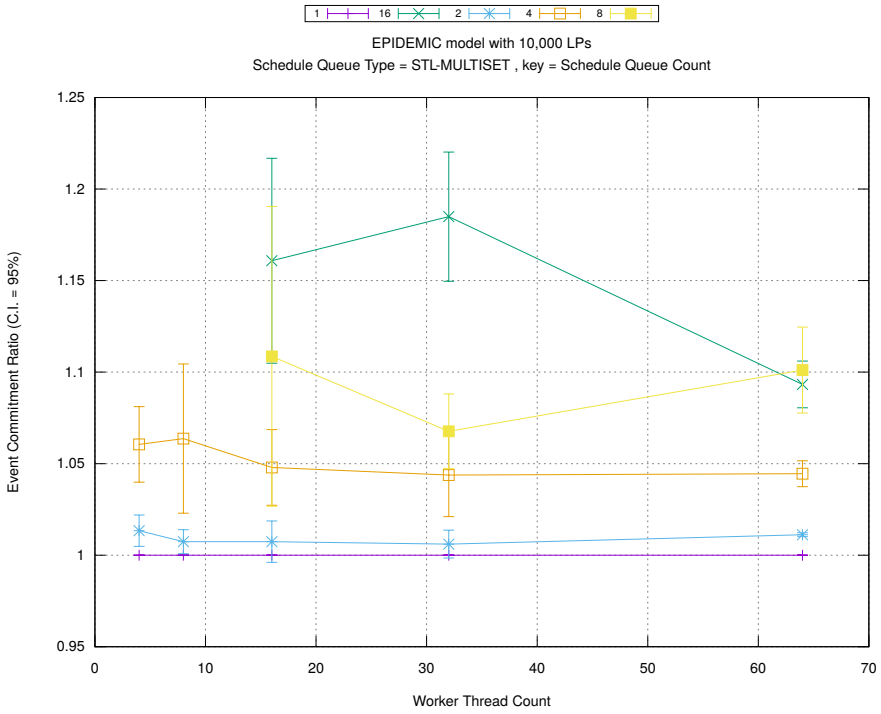
Figure A.144: epidemic 10k ba/plots/scheduleq/threads vs count key type ladder-queue 96



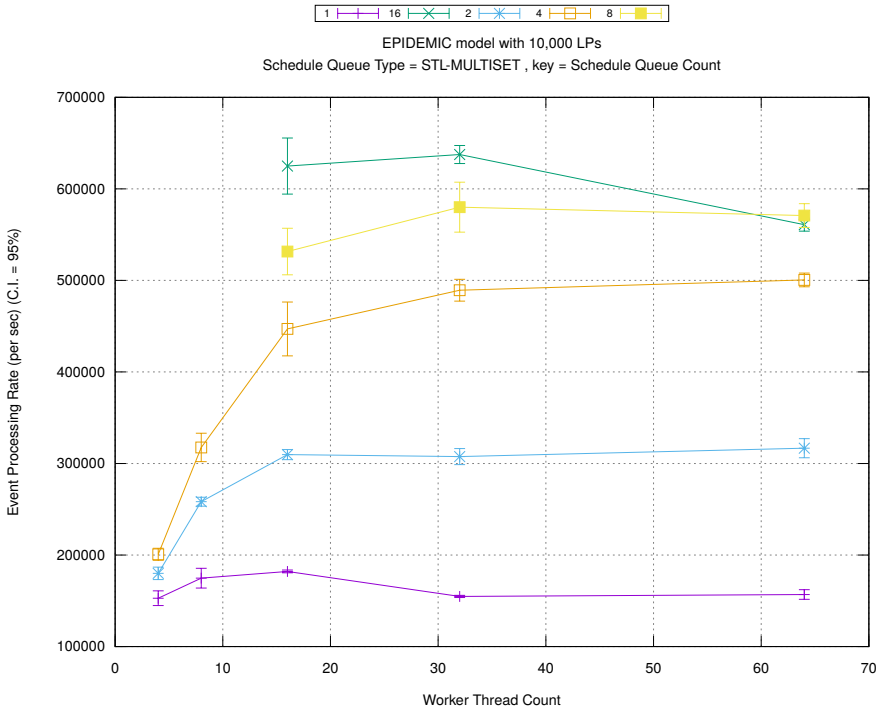
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

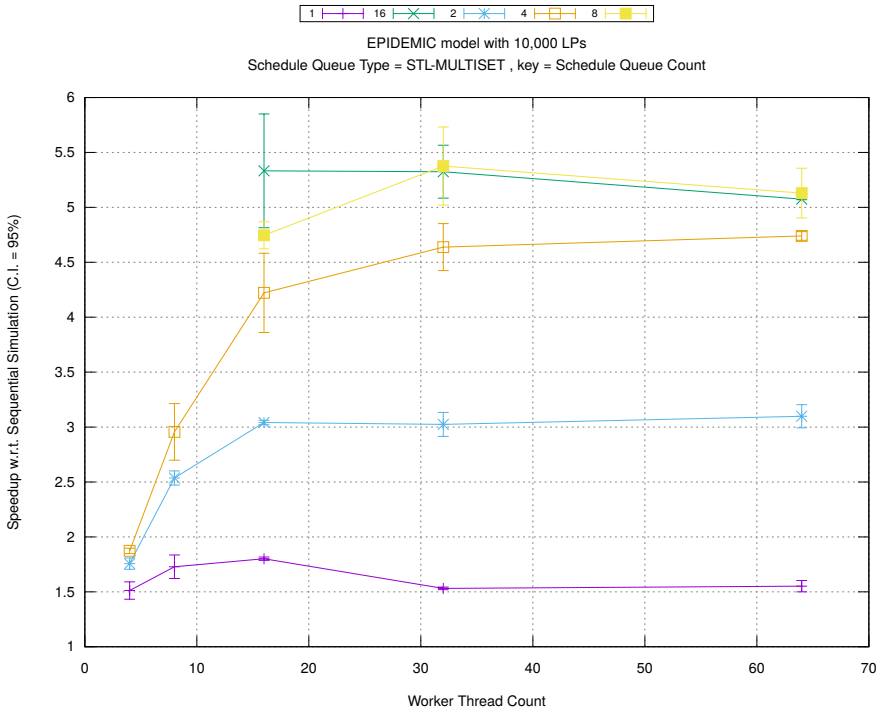


(b) Event Commitment Ratio

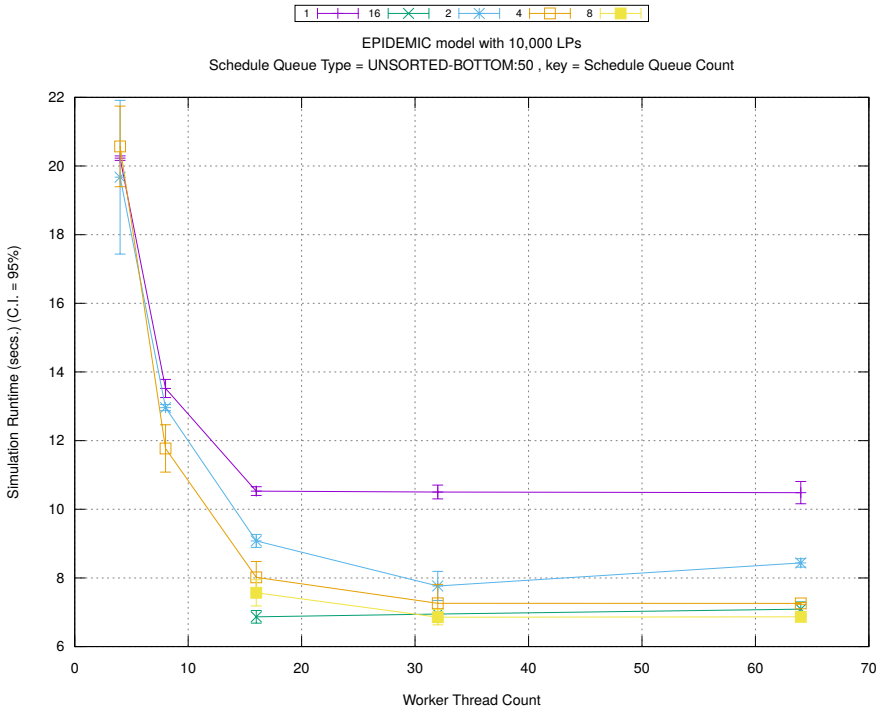


(c) Event Processing Rate (per second)

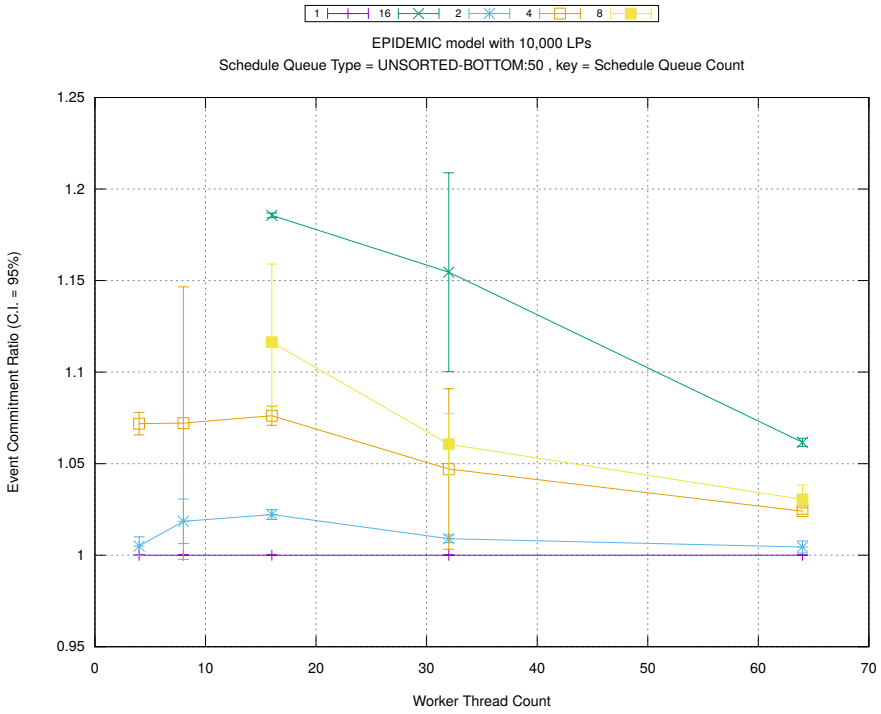
Figure A.145: epidemic 10k ba/plots/scheduleq/threads vs count key type stl-multiset



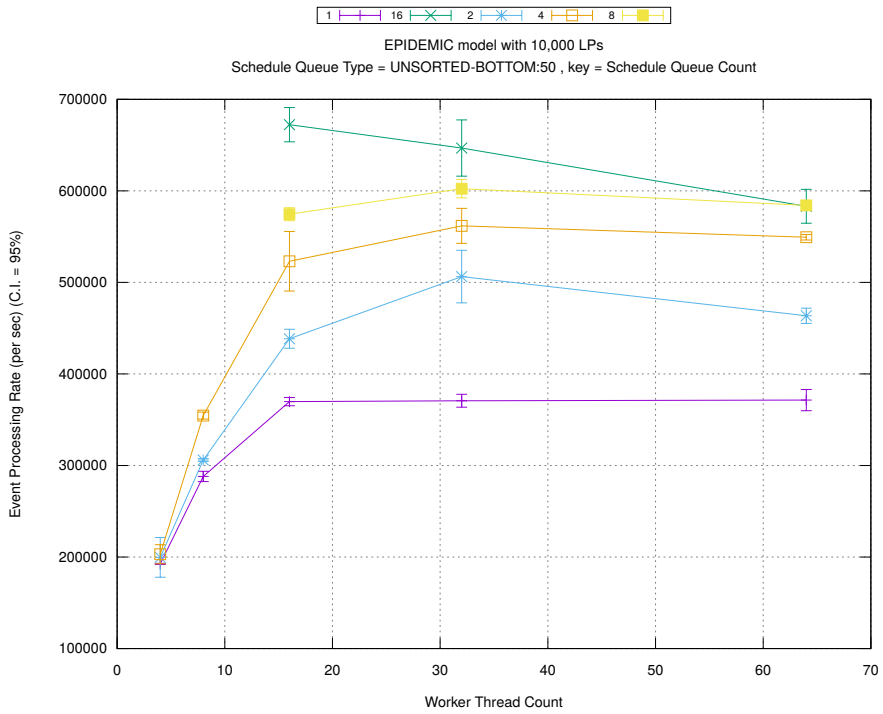
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

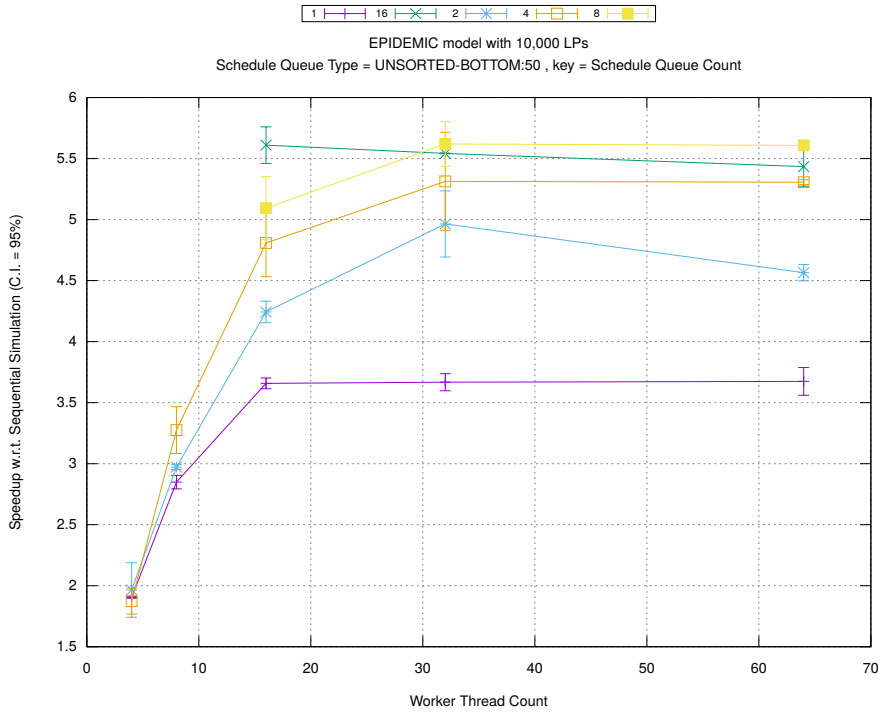


(b) Event Commitment Ratio

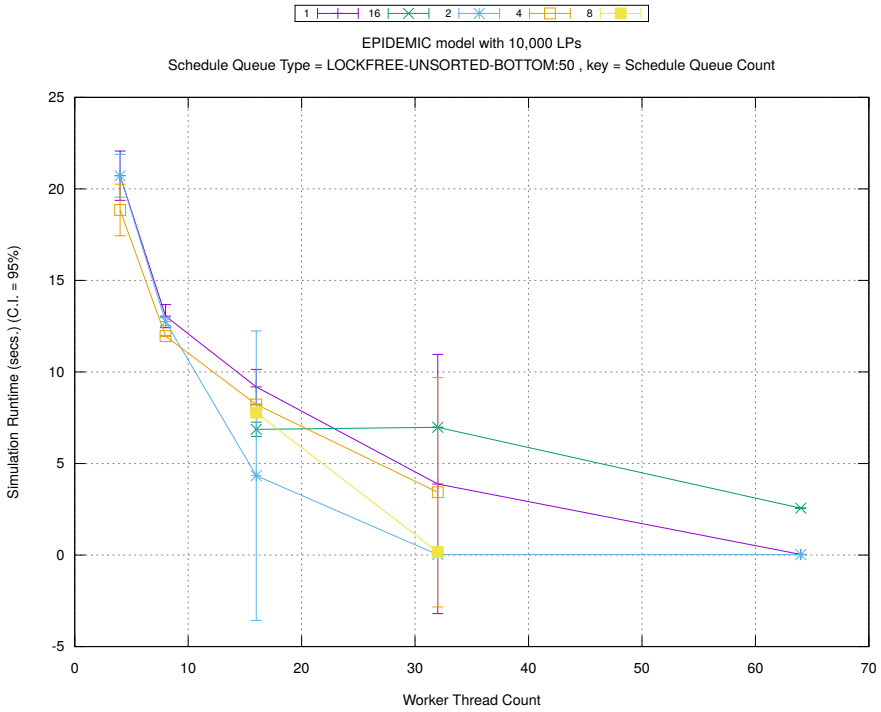


(c) Event Processing Rate (per second)

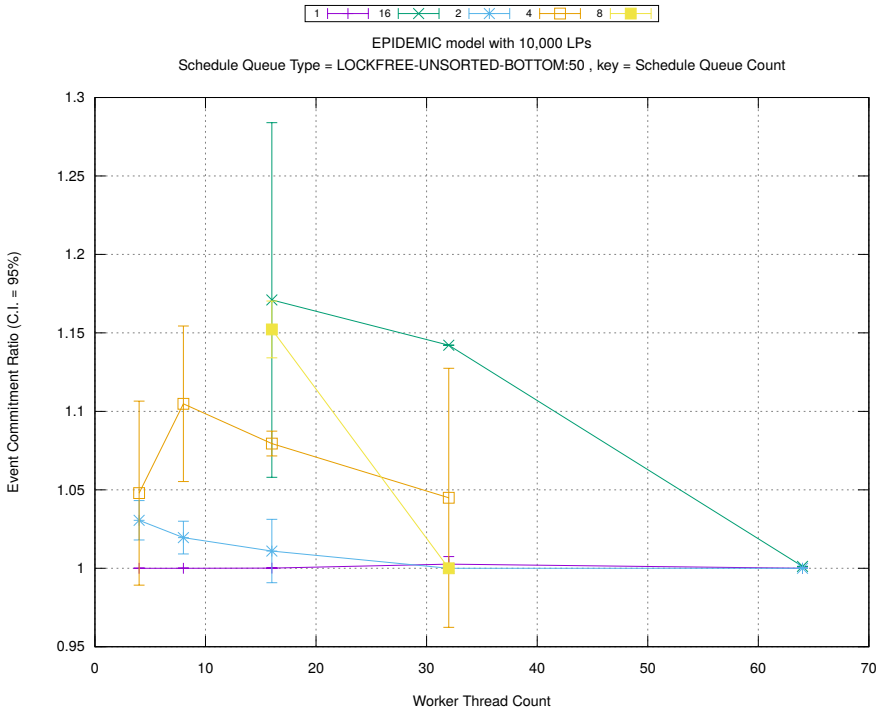
Figure A.146: epidemic 10k ba/plots/scheduleq/threads vs count key type unsorted-bottom 50



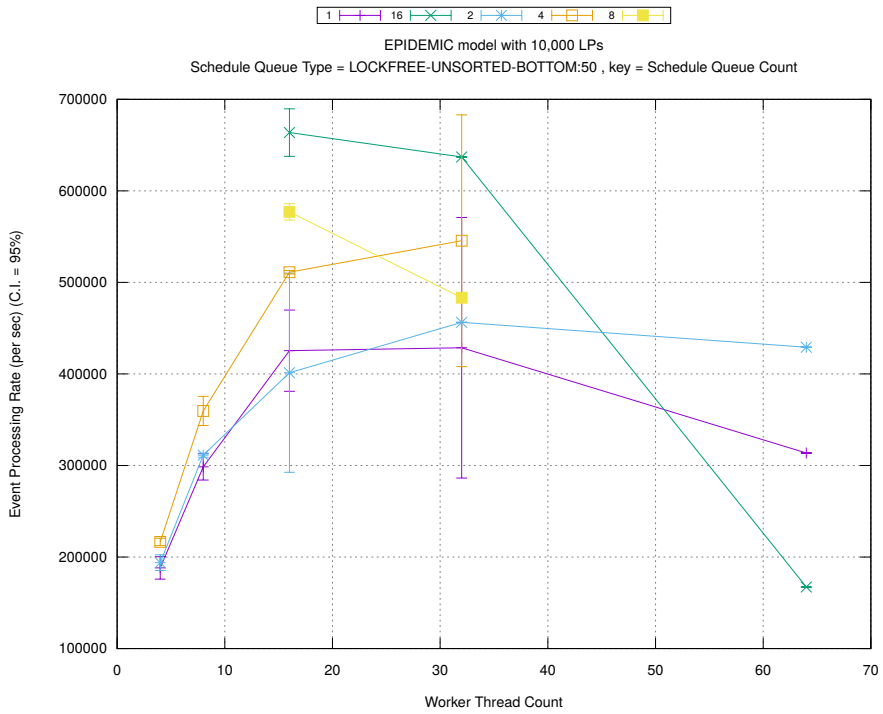
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

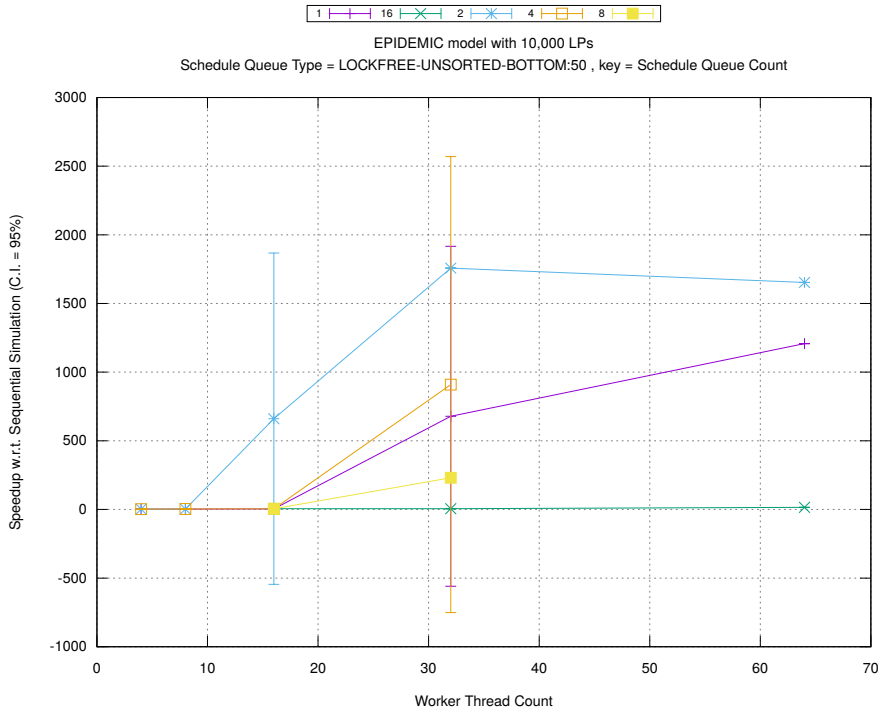


(b) Event Commitment Ratio

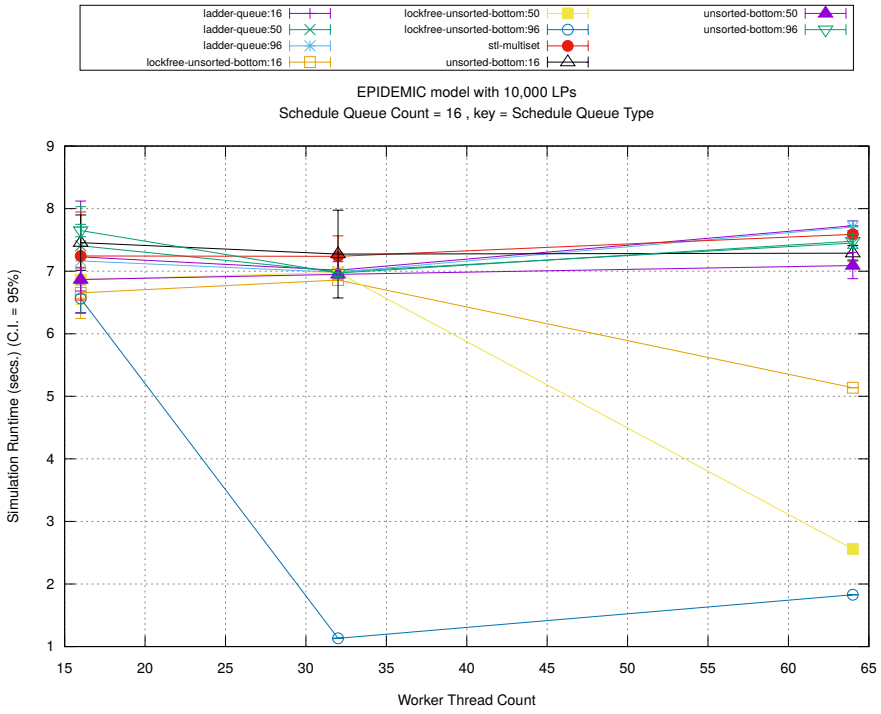


(c) Event Processing Rate (per second)

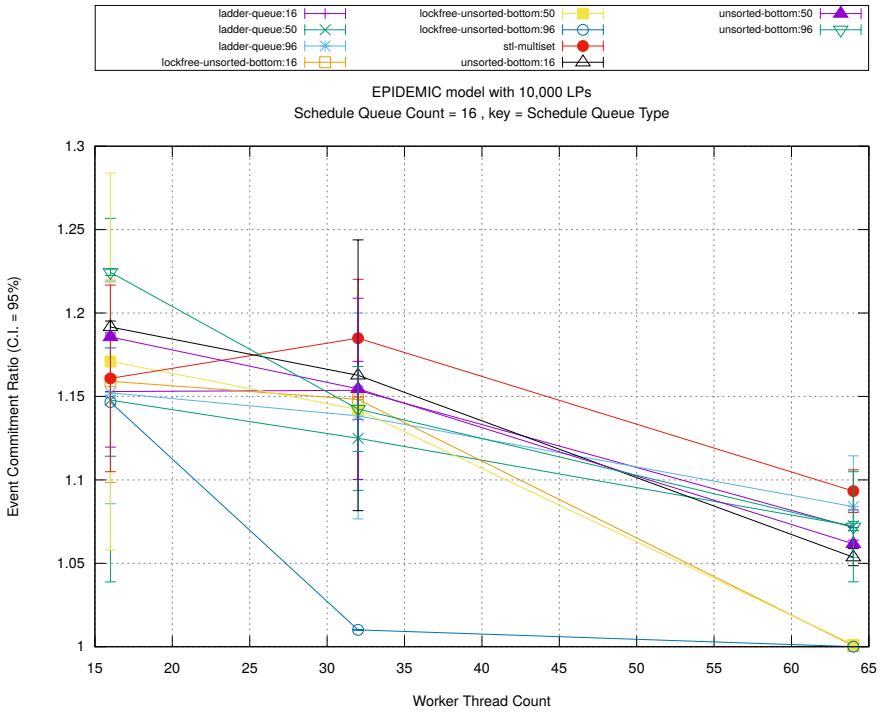
Figure A.147: epidemic 10k ba/plots/scheduleq/threads vs count key type lockfree-unsorted-bottom 50



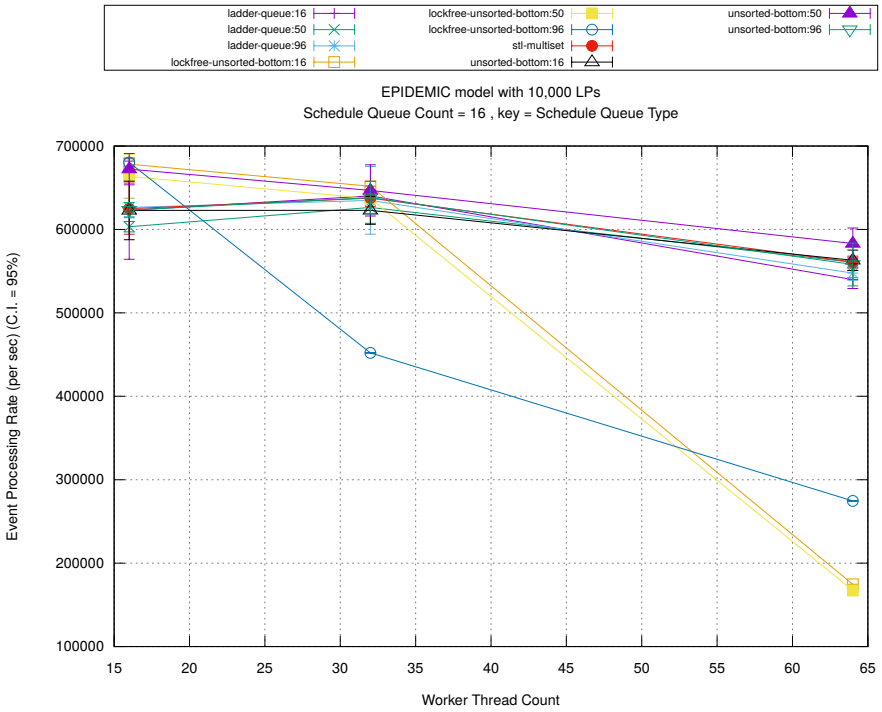
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

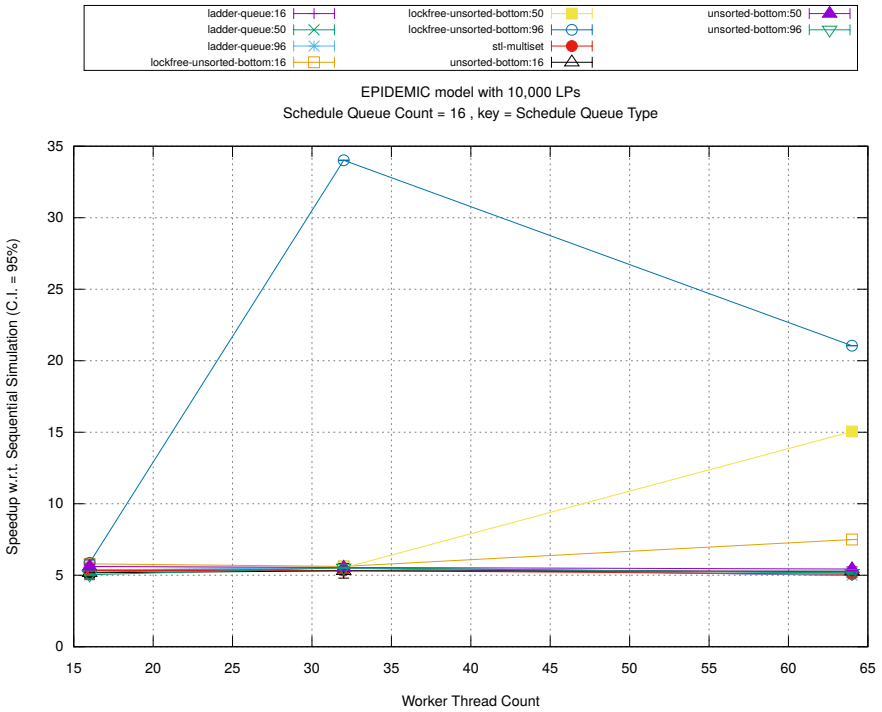


(b) Event Commitment Ratio

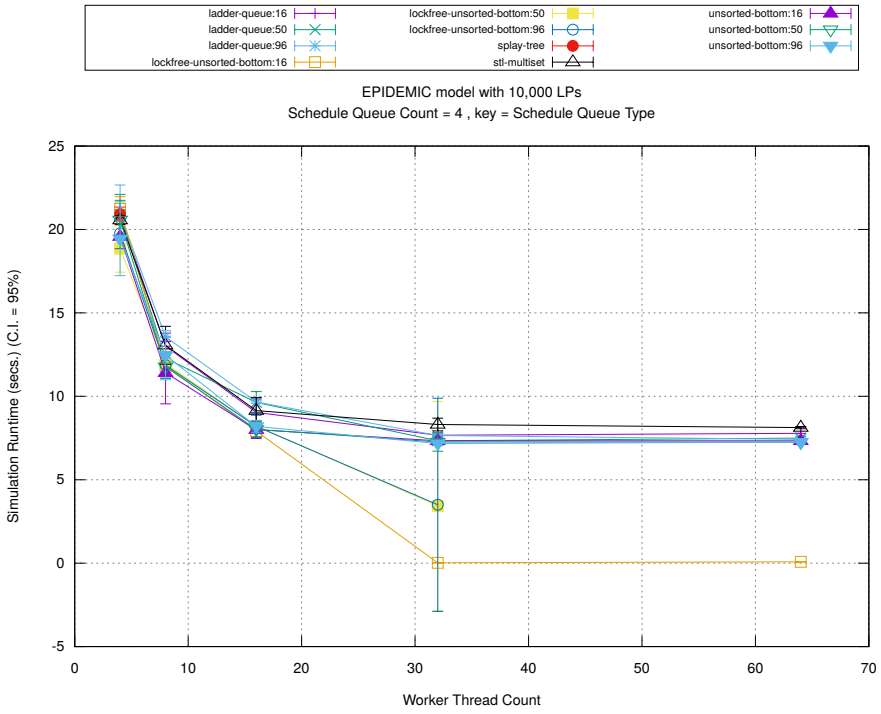


(c) Event Processing Rate (per second)

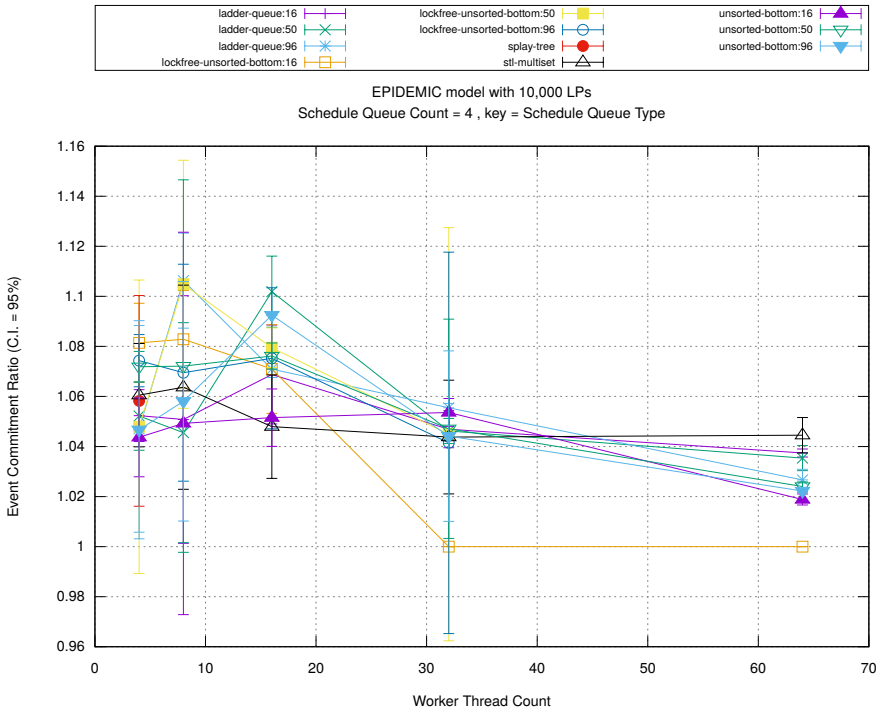
Figure A.148: epidemic 10k ba/plots/scheduleq/threads vs type key count 16



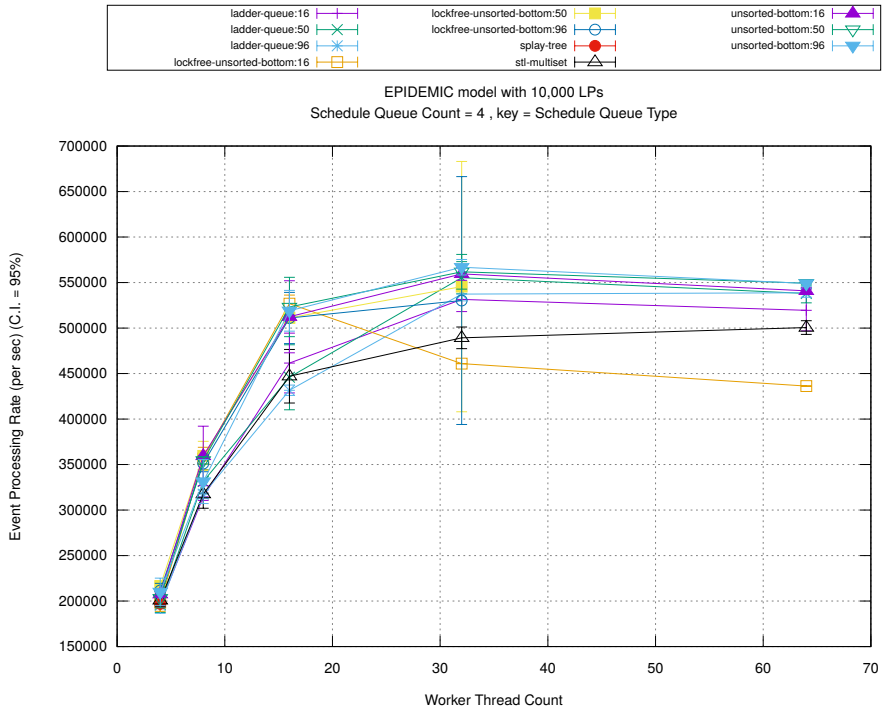
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

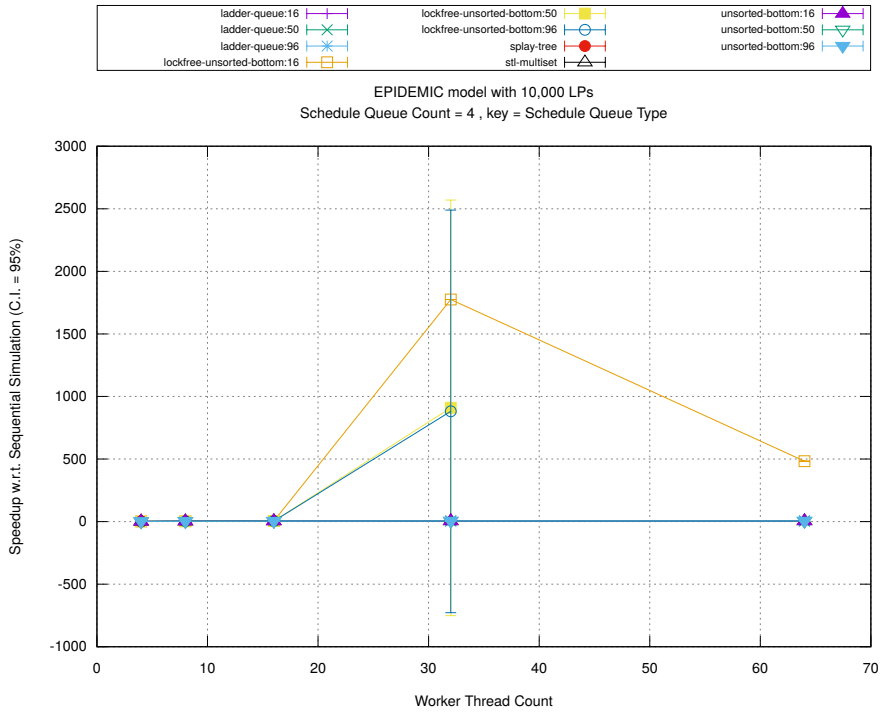


(b) Event Commitment Ratio

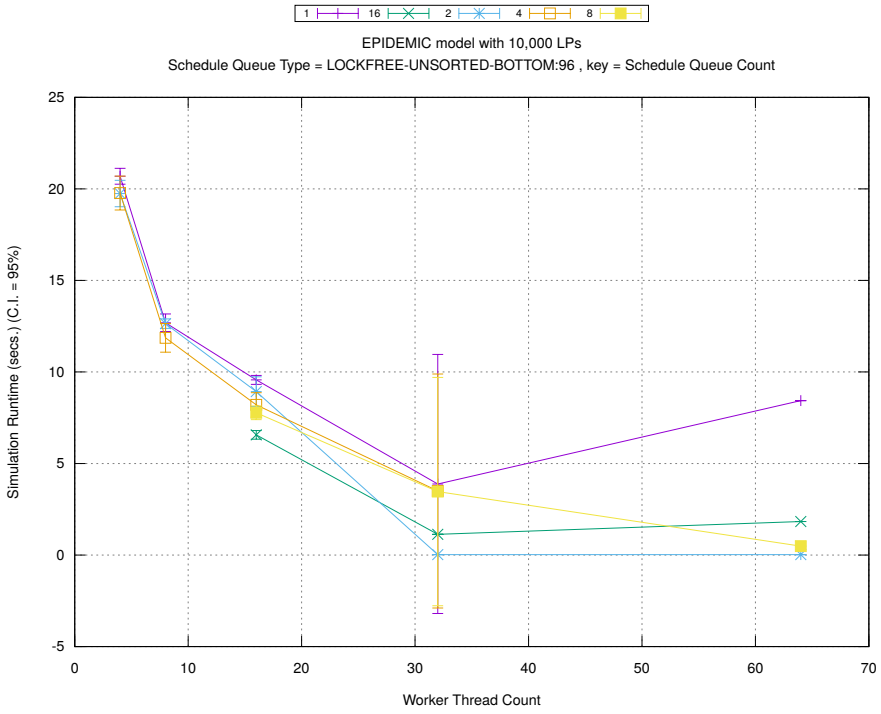


(c) Event Processing Rate (per second)

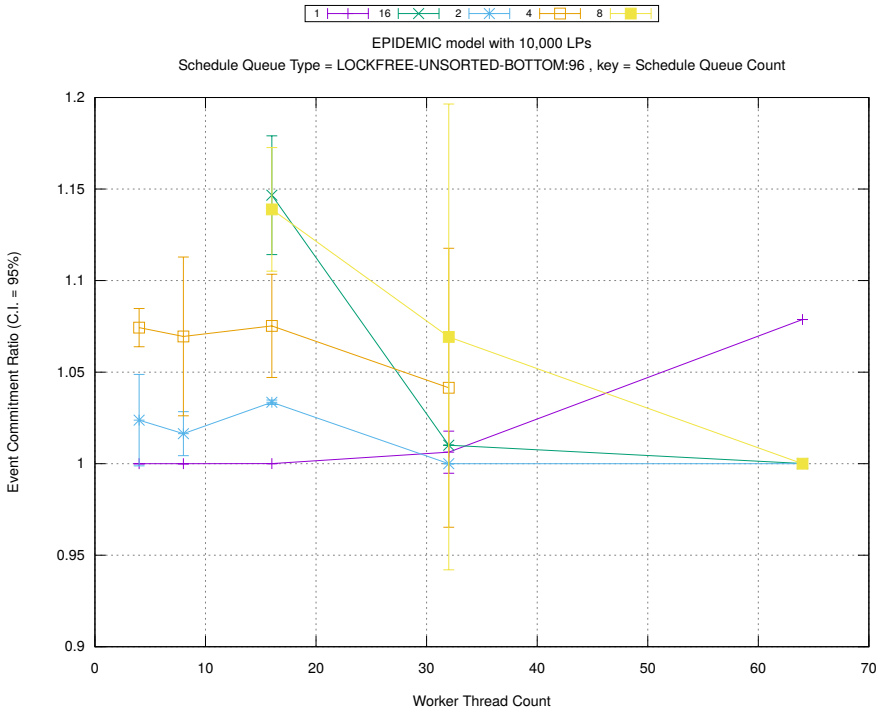
Figure A.149: epidemic 10k ba/plots/scheduleq/threads vs type key count 4



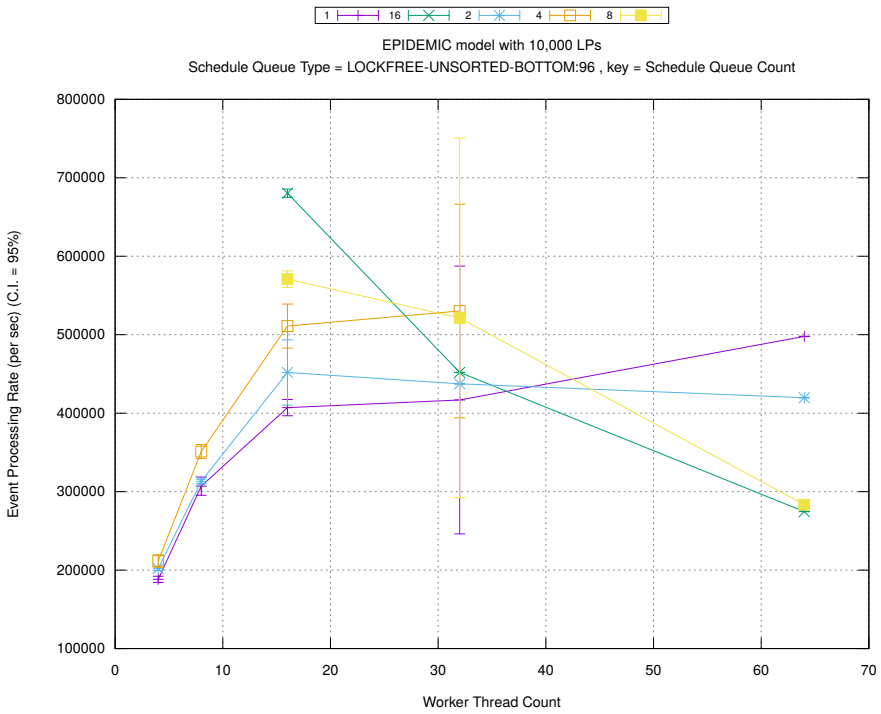
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

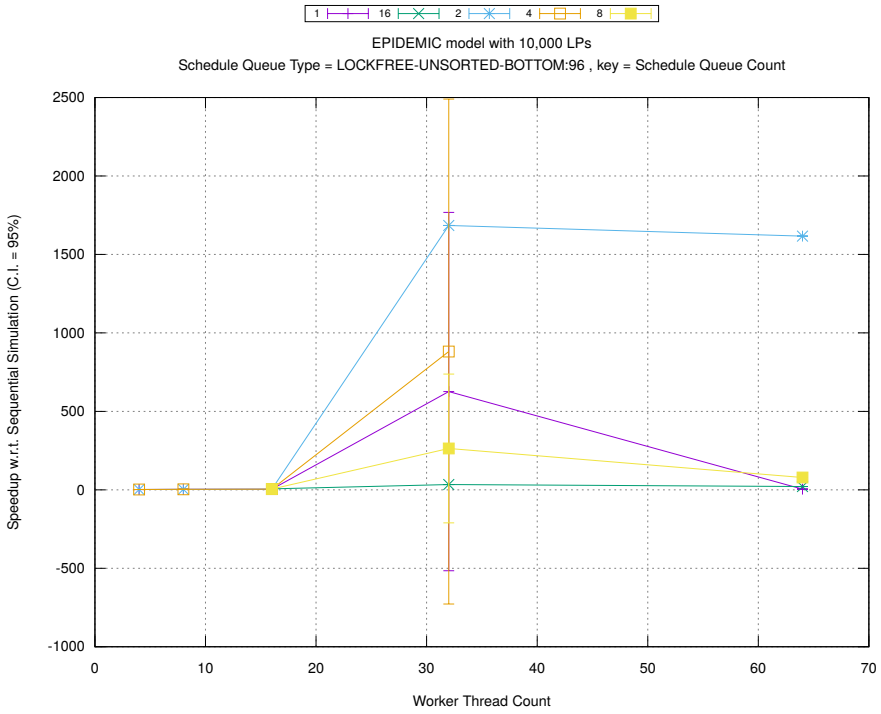


(b) Event Commitment Ratio

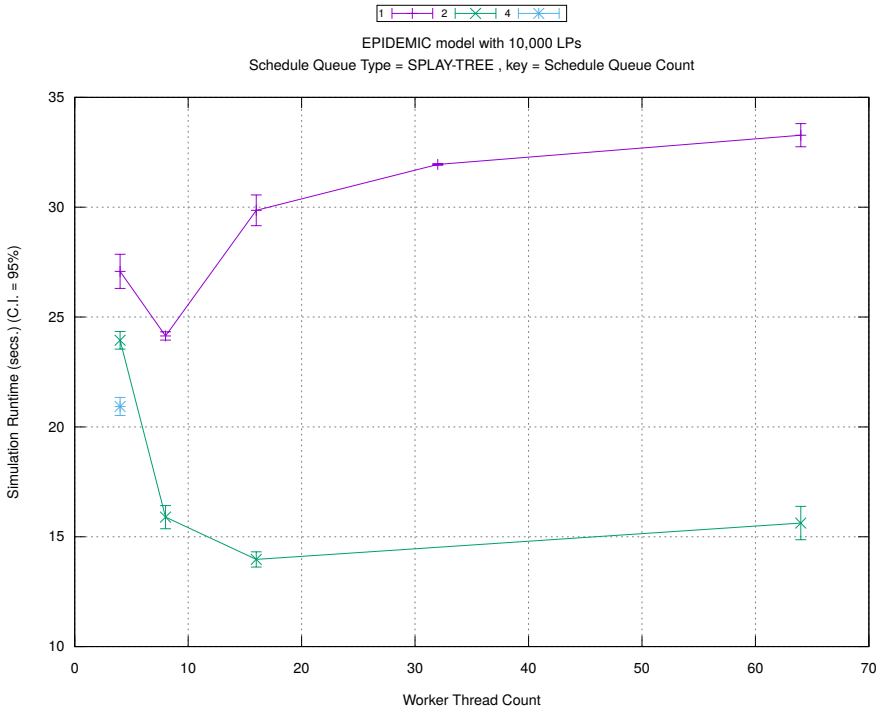


(c) Event Processing Rate (per second)

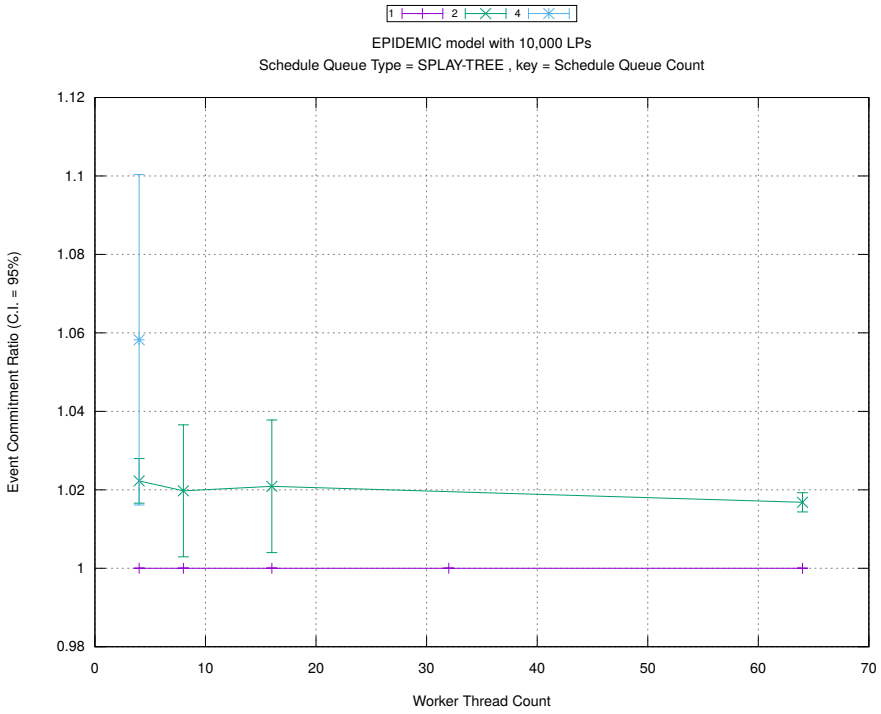
Figure A.150: epidemic 10k ba/plots/scheduleq/threads vs count key type lockfree-unsorted-bottom 96



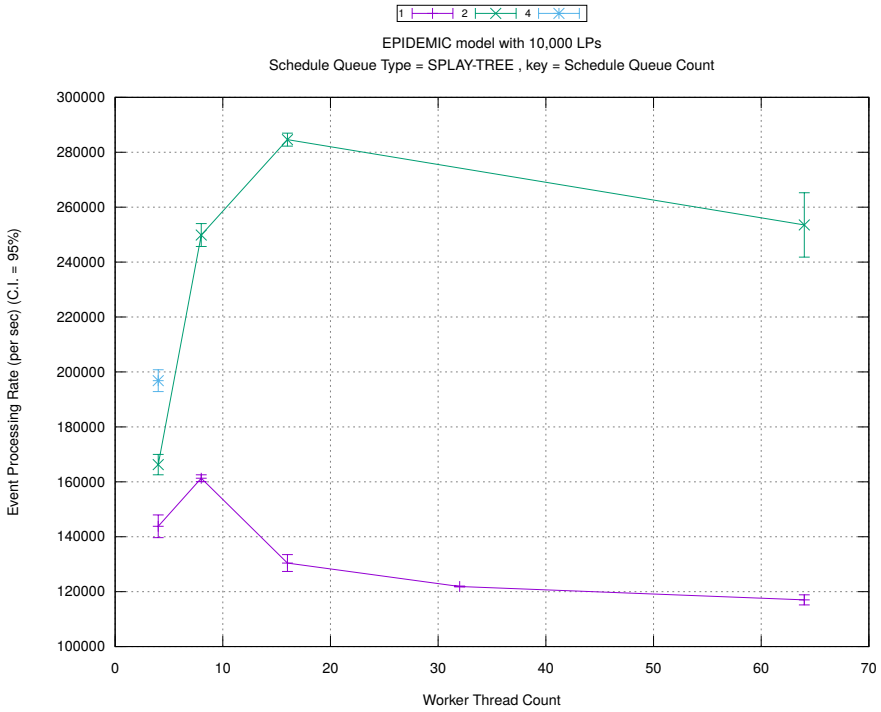
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

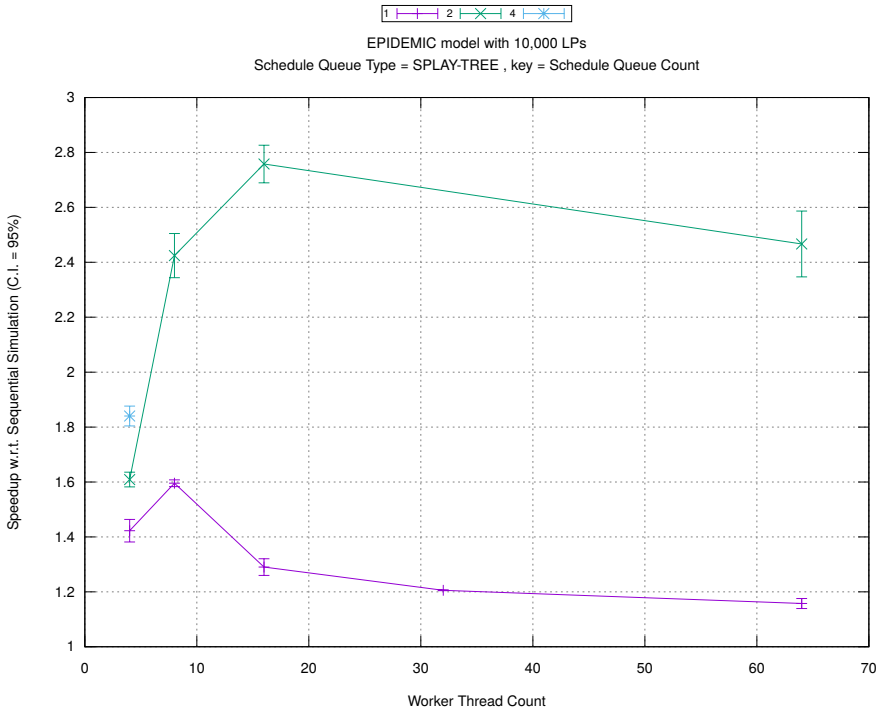


(b) Event Commitment Ratio

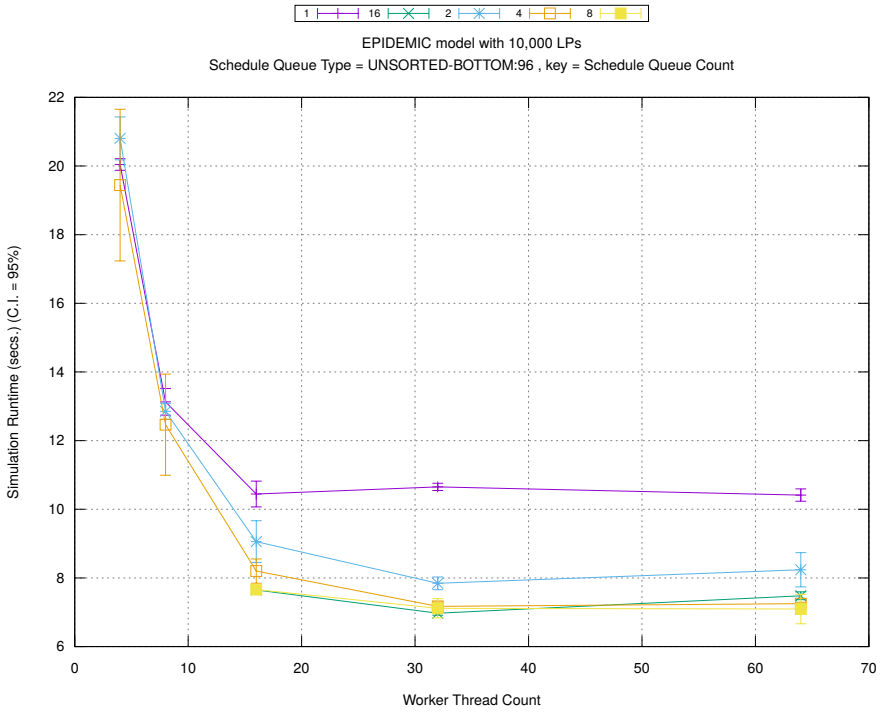


(c) Event Processing Rate (per second)

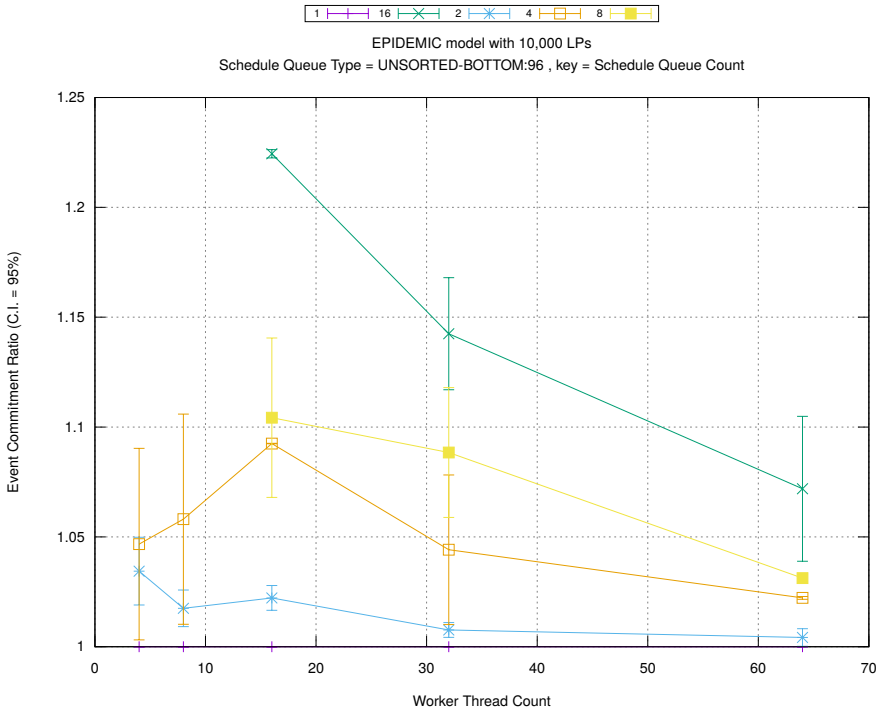
Figure A.151: epidemic 10k ba/plots/scheduleq/threads vs count key type splay-tree



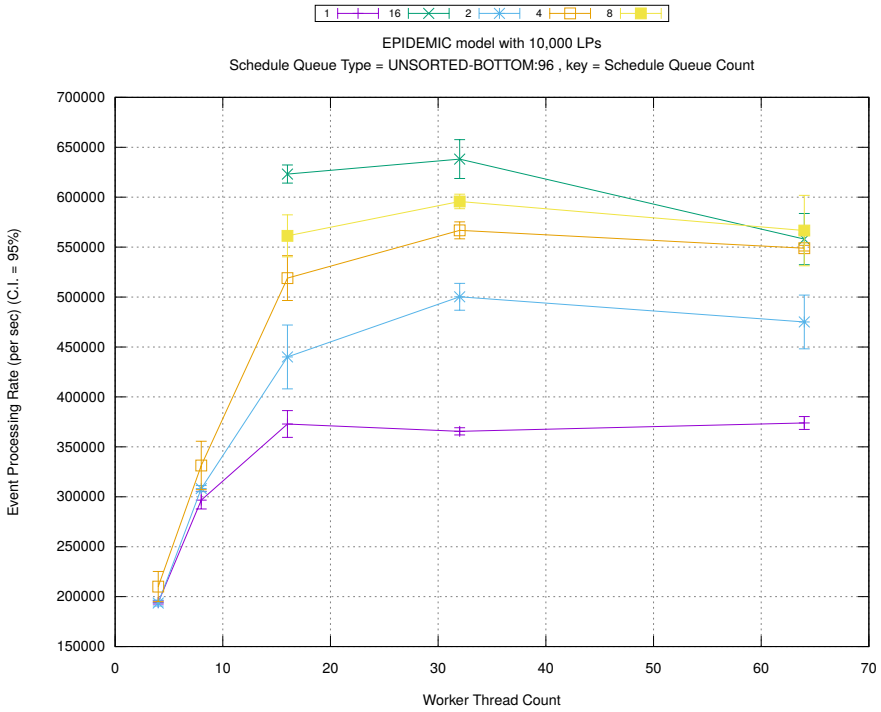
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

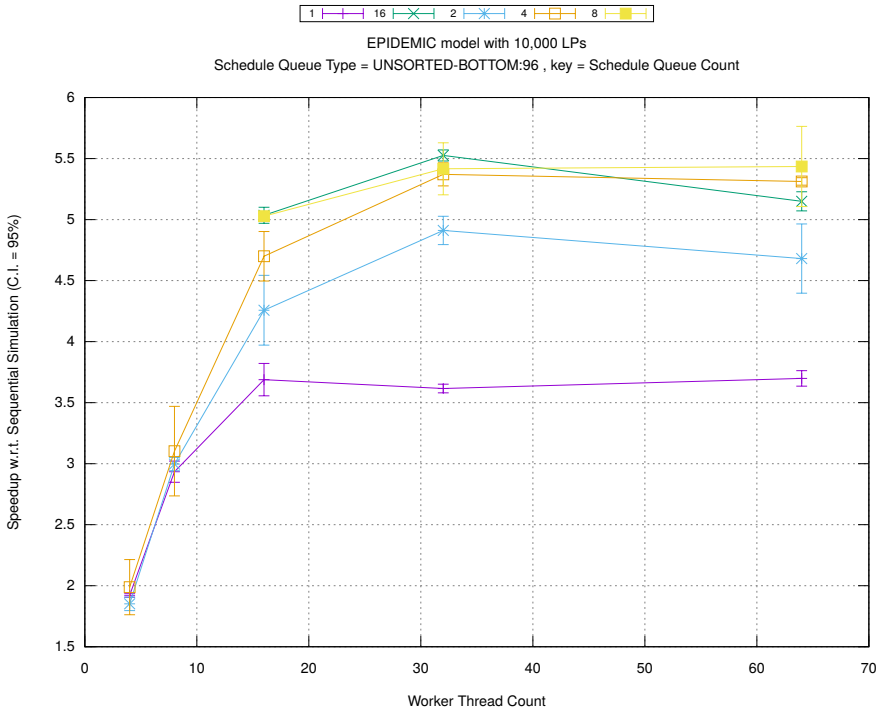


(b) Event Commitment Ratio

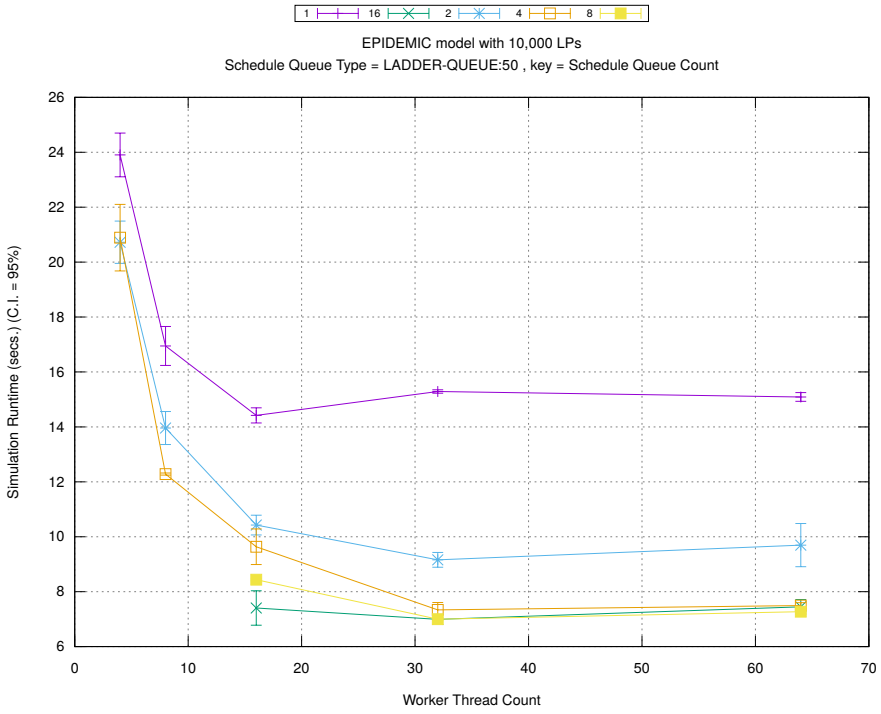


(c) Event Processing Rate (per second)

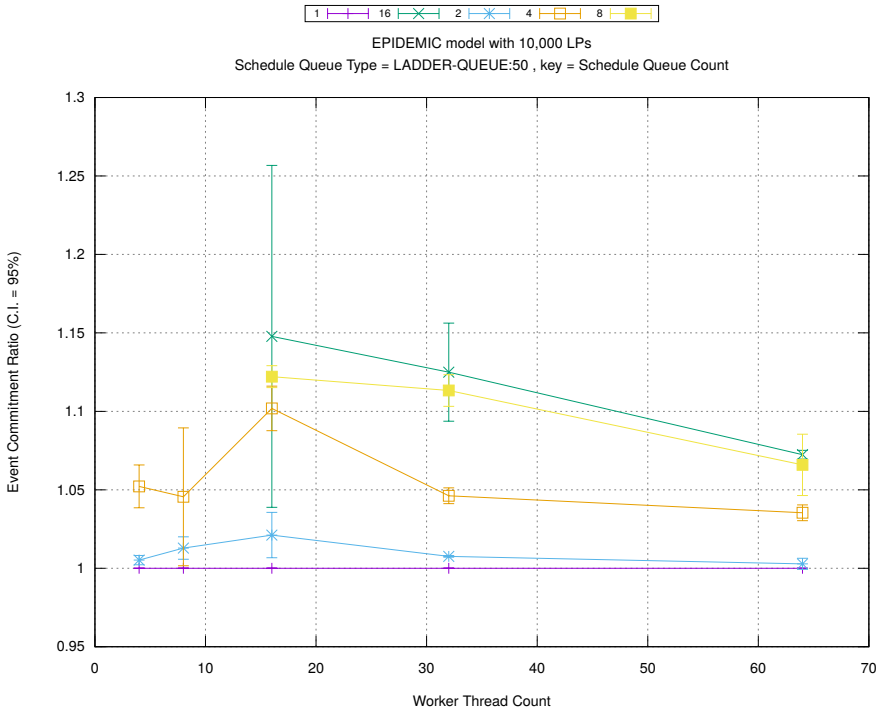
Figure A.152: epidemic 10k ba/plots/scheduleq/threads vs count key type unsorted-bottom 96



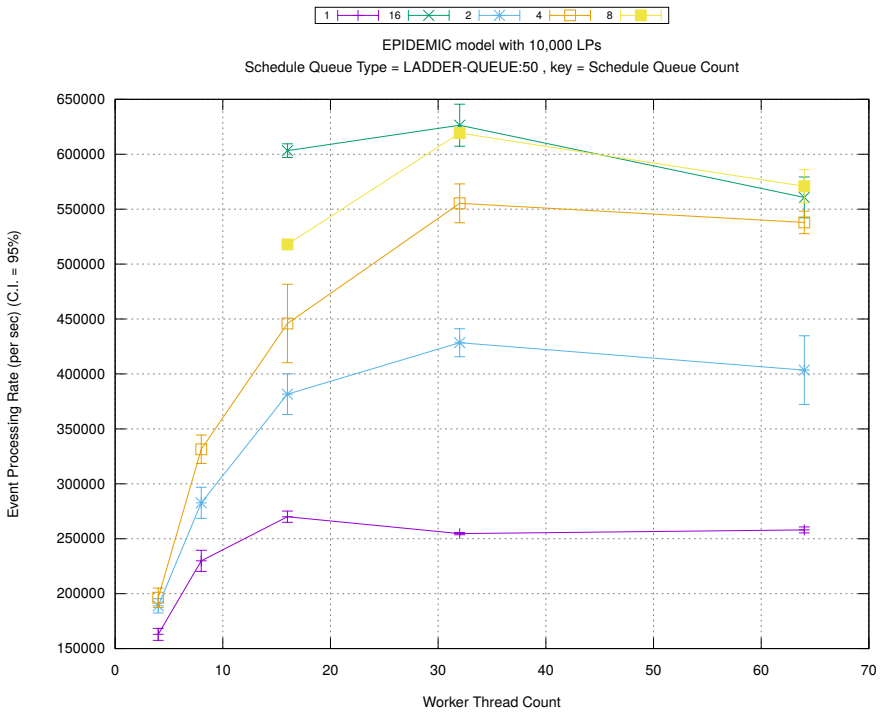
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

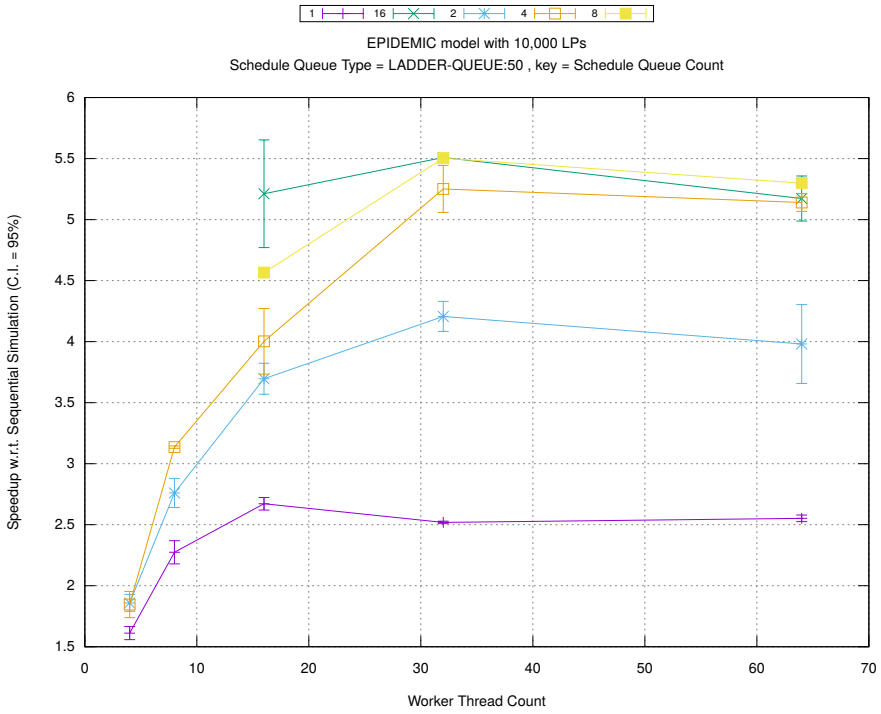


(b) Event Commitment Ratio

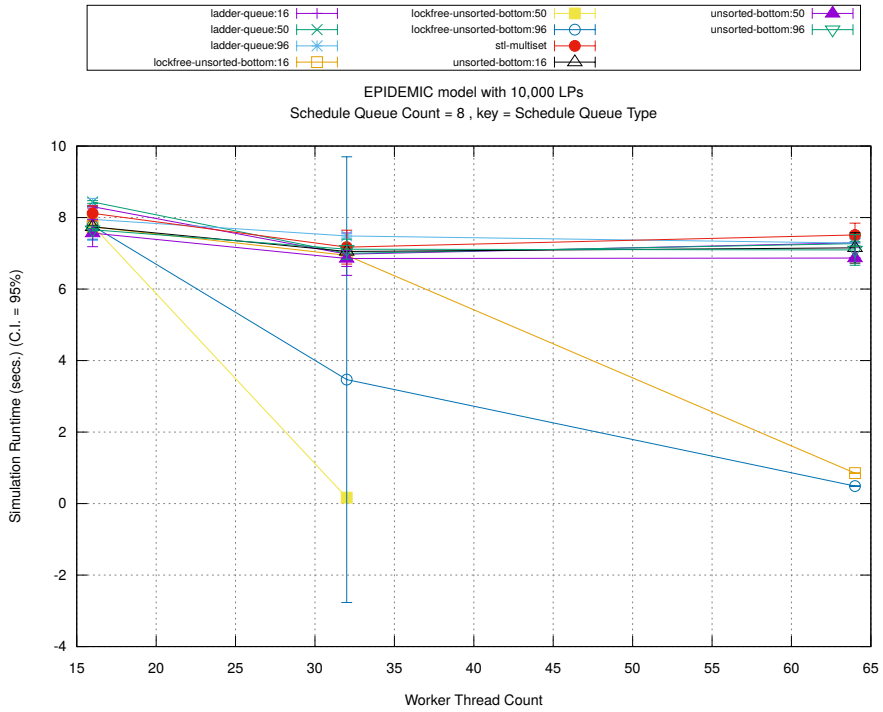


(c) Event Processing Rate (per second)

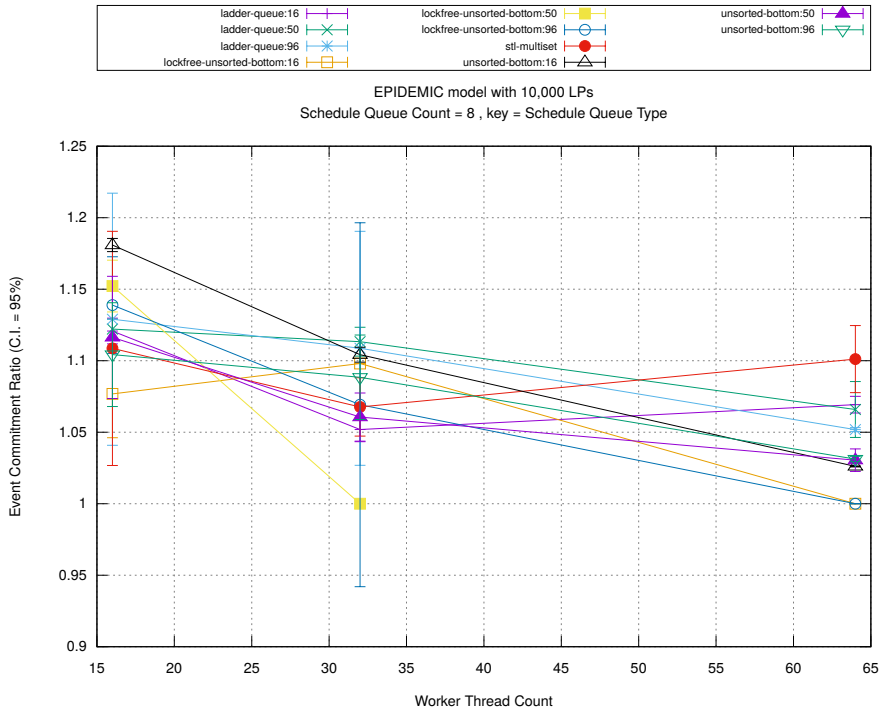
Figure A.153: epidemic 10k ba/plots/scheduleq/threads vs count key type ladder-queue 50



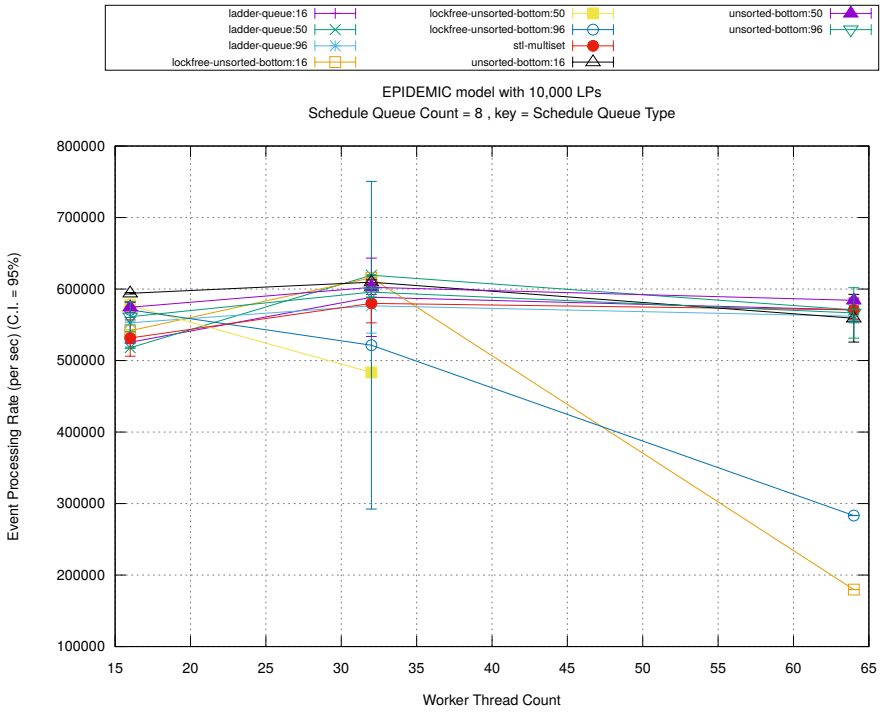
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

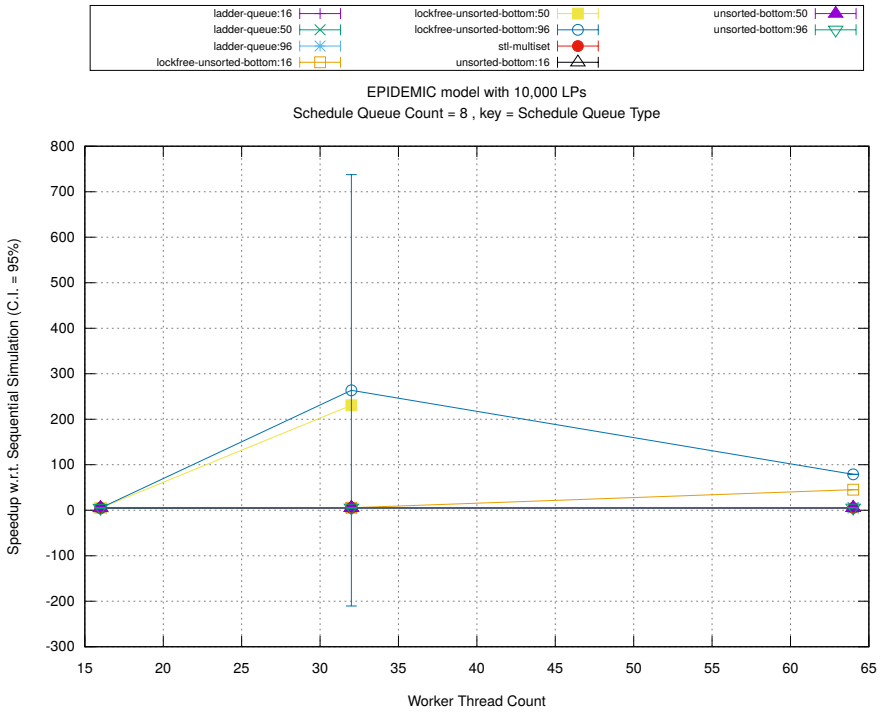


(b) Event Commitment Ratio

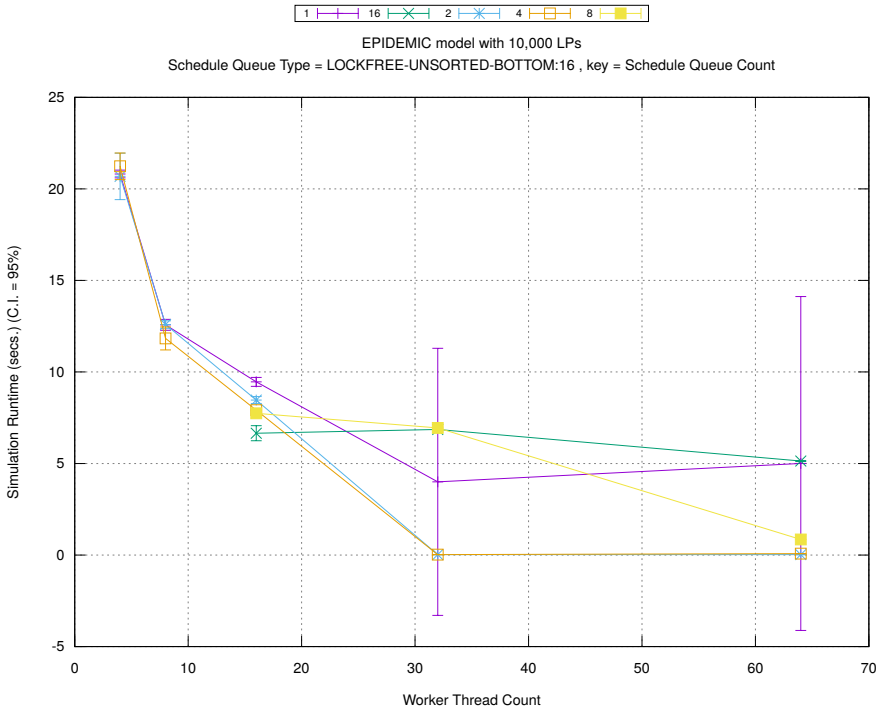


(c) Event Processing Rate (per second)

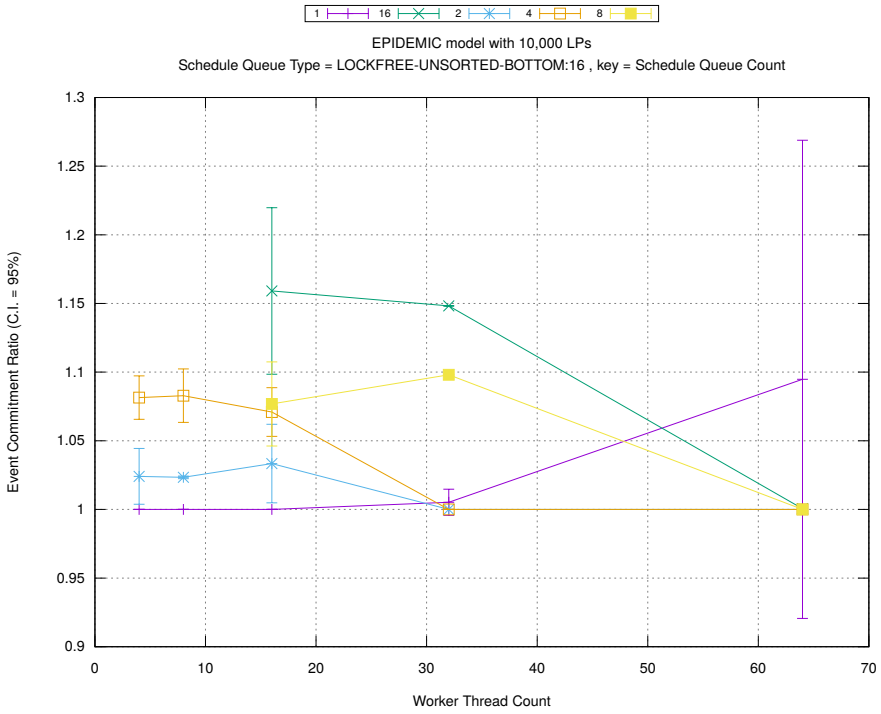
Figure A.154: epidemic 10k ba/plots/scheduleq/threads vs type key count 8



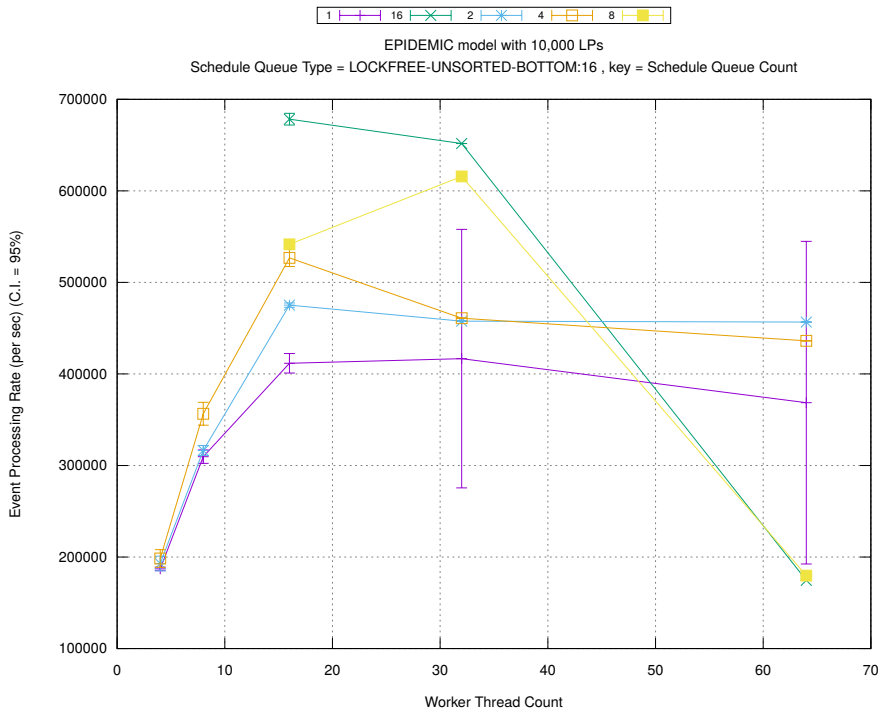
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

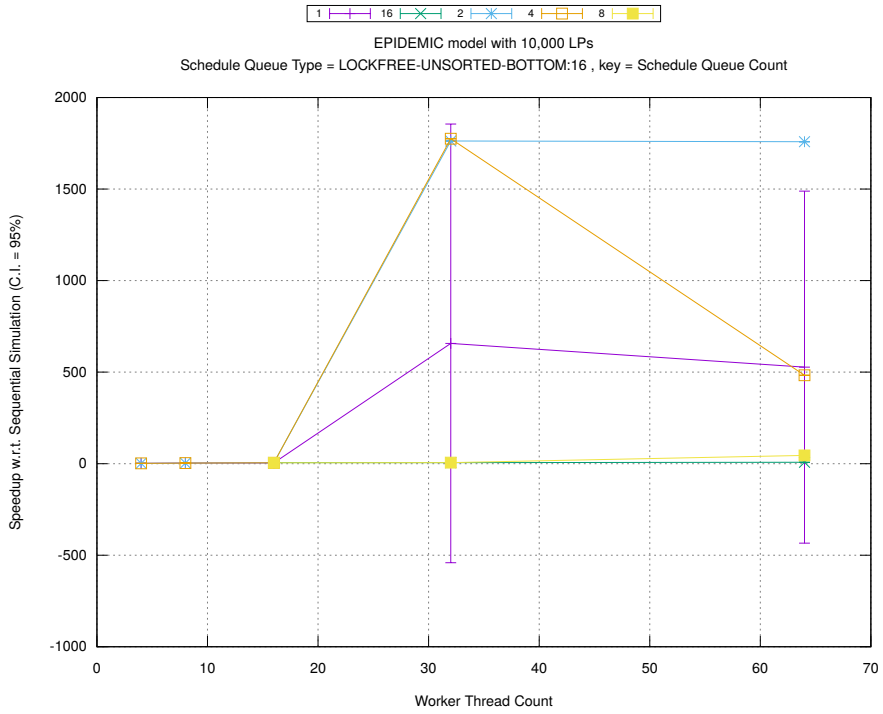


(b) Event Commitment Ratio

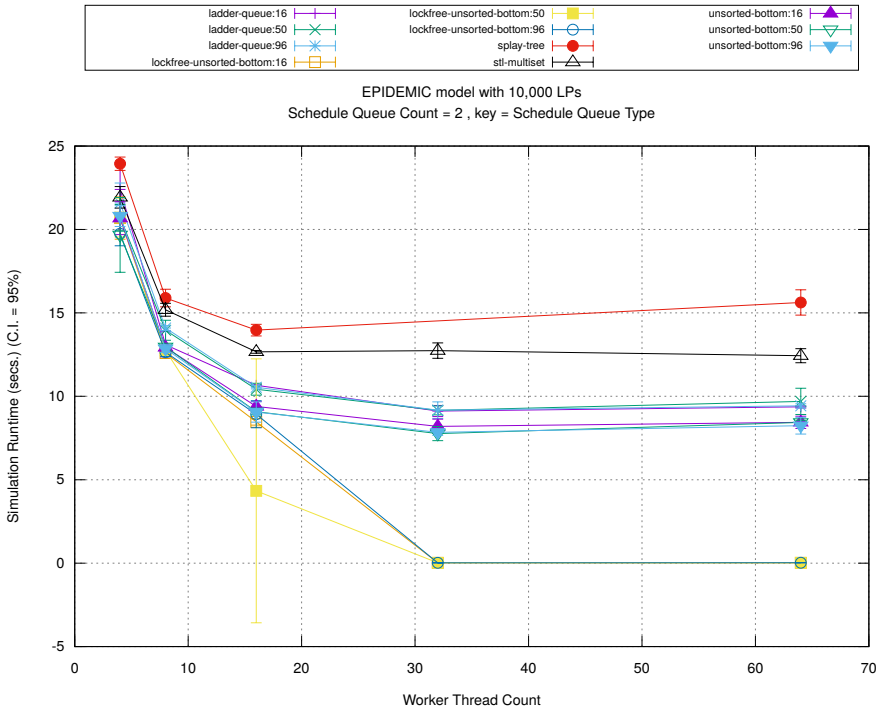


(c) Event Processing Rate (per second)

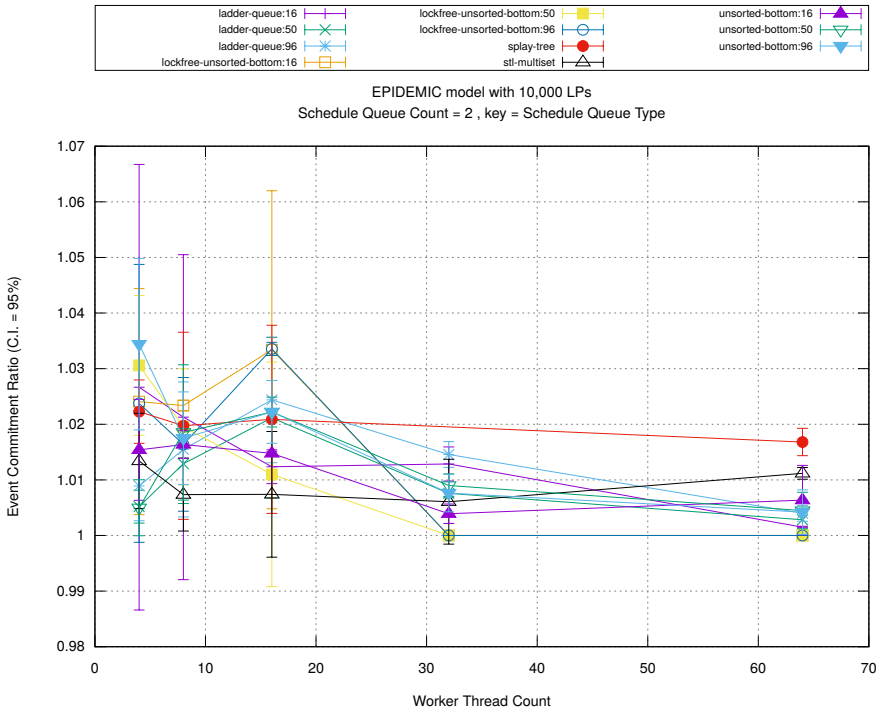
Figure A.155: epidemic 10k ba/plots/scheduleq/threads vs count key type lockfree-unsorted-bottom 16



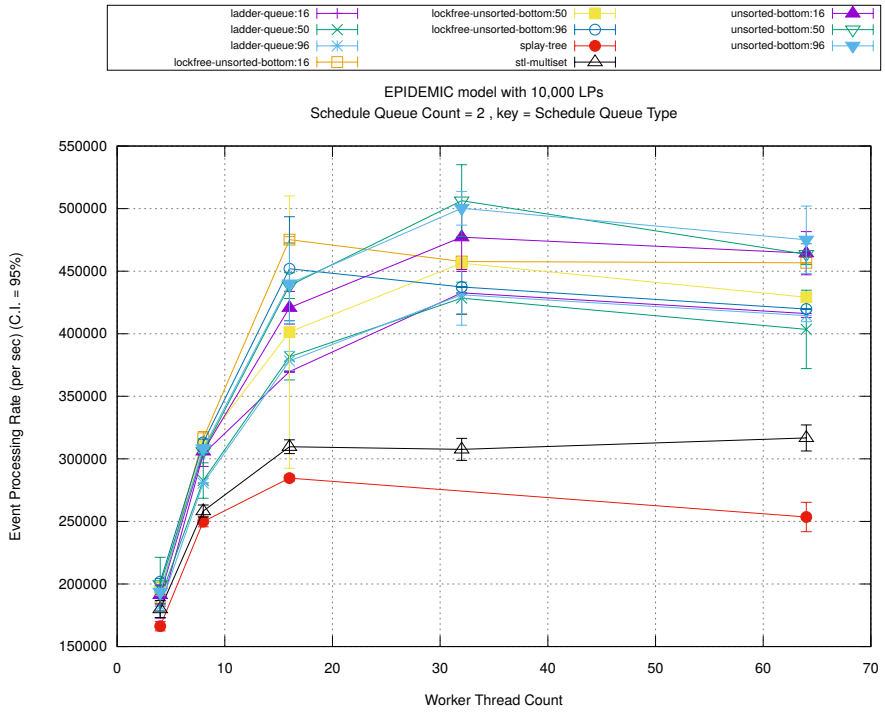
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

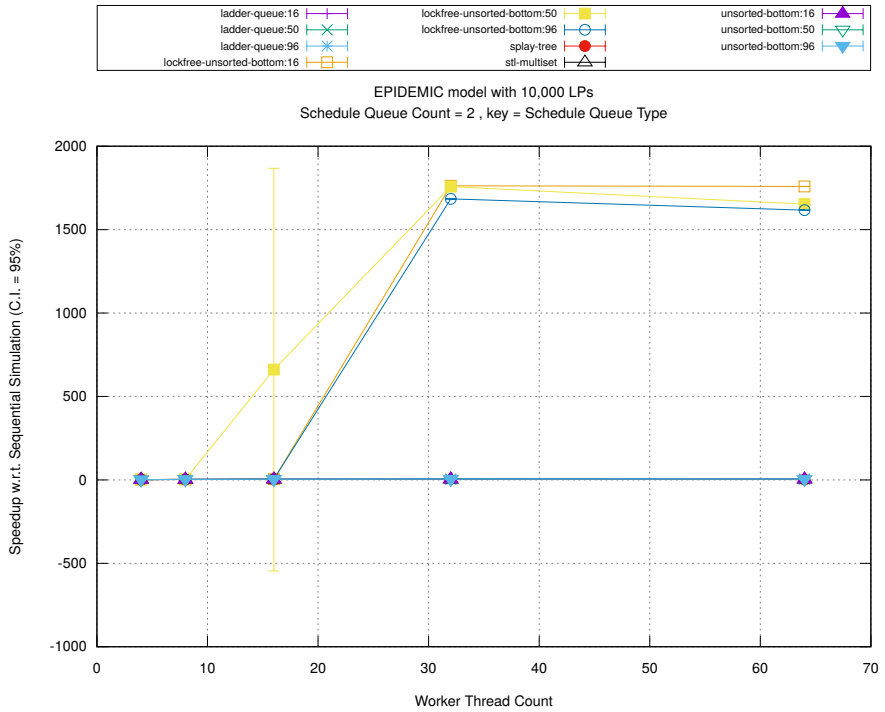


(b) Event Commitment Ratio

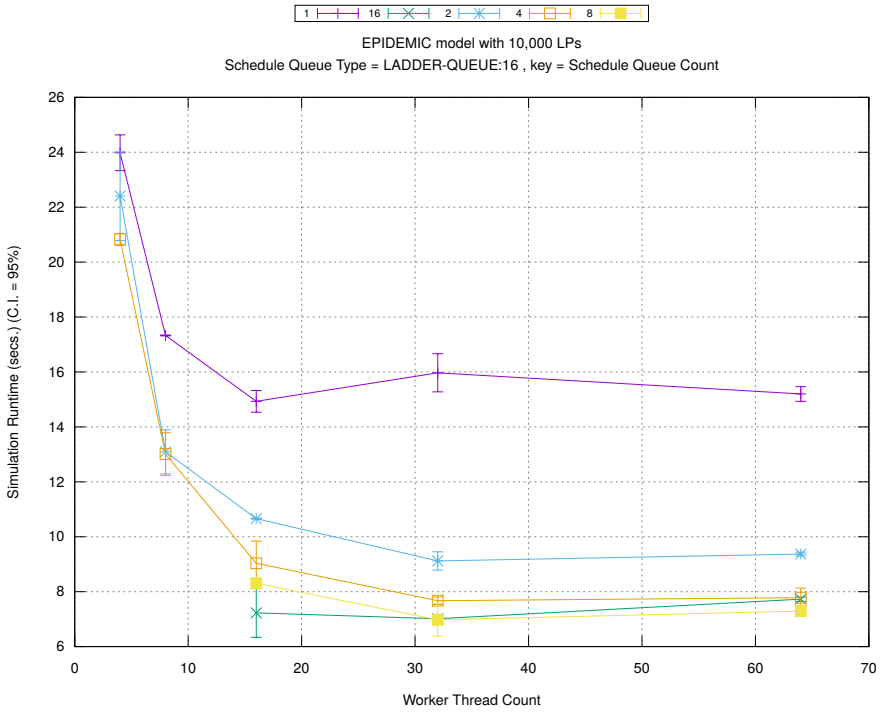


(c) Event Processing Rate (per second)

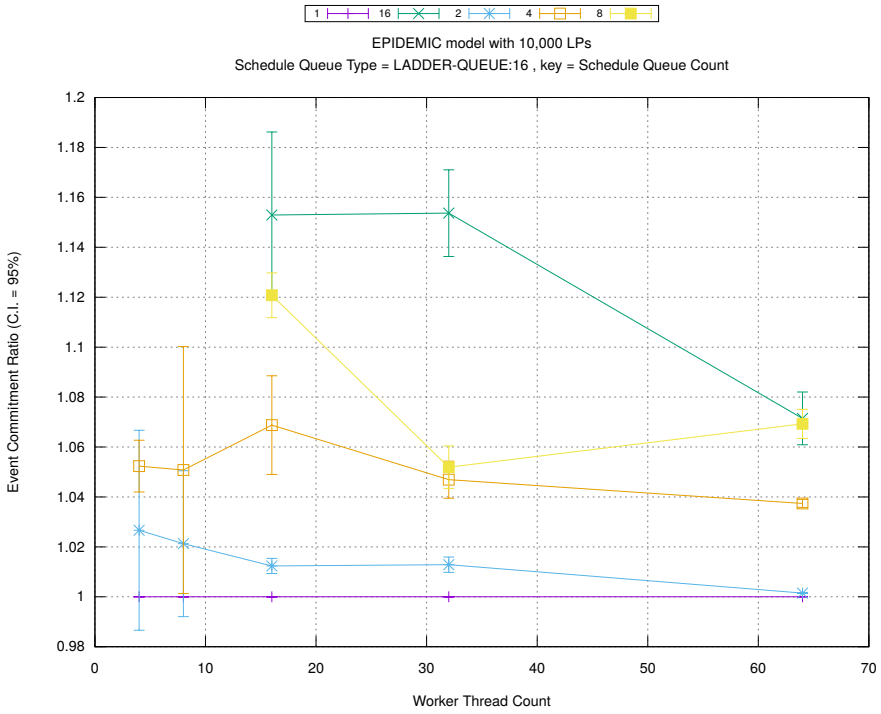
Figure A.156: epidemic 10k ba/plots/scheduleq/threads vs type key count 2



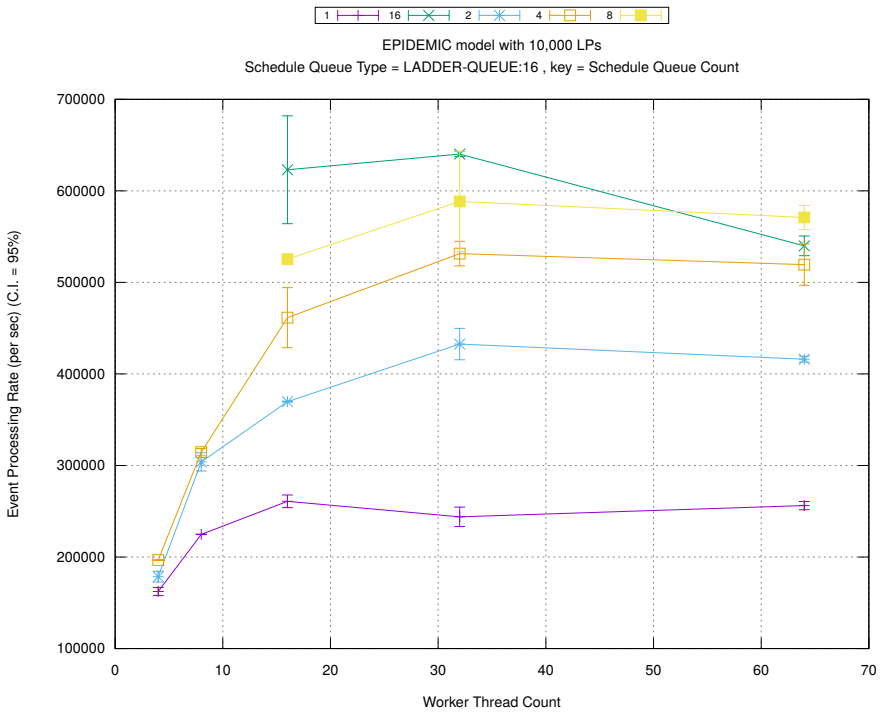
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

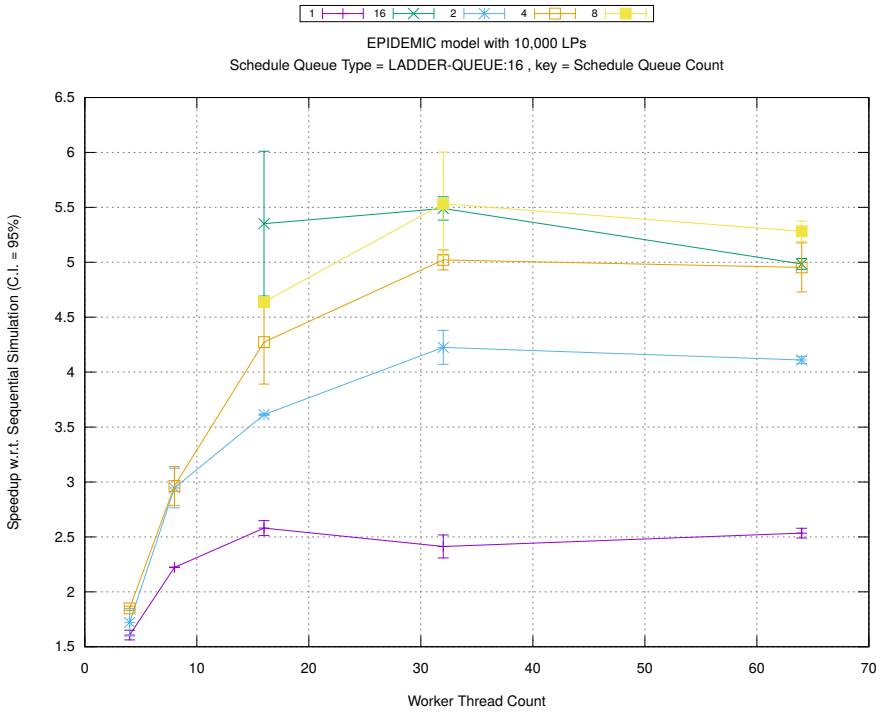


(b) Event Commitment Ratio

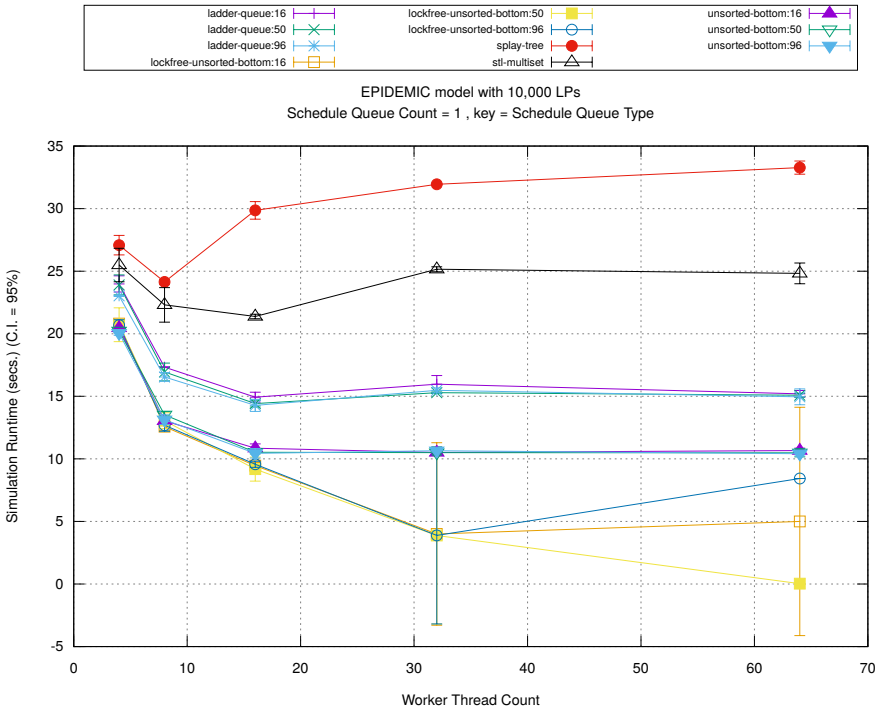


(c) Event Processing Rate (per second)

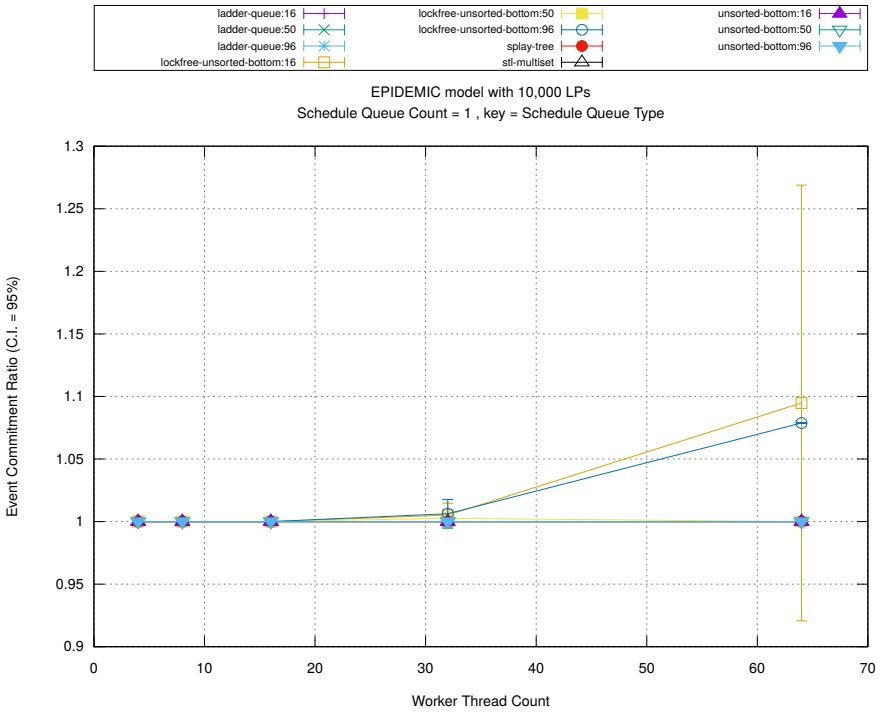
Figure A.157: epidemic 10k ba/plots/scheduleq/threads vs count key type ladder-queue 16



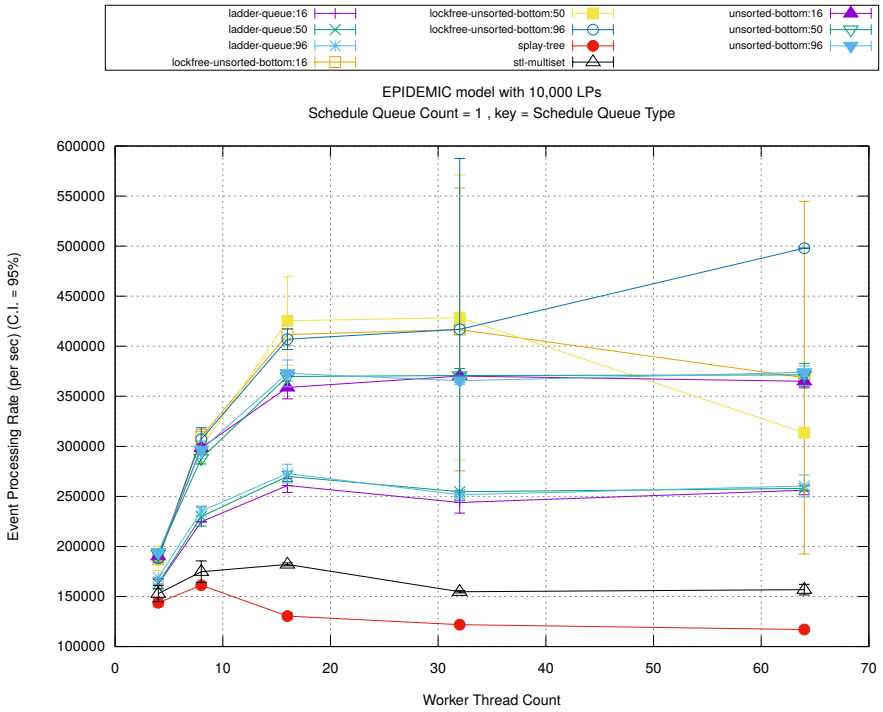
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

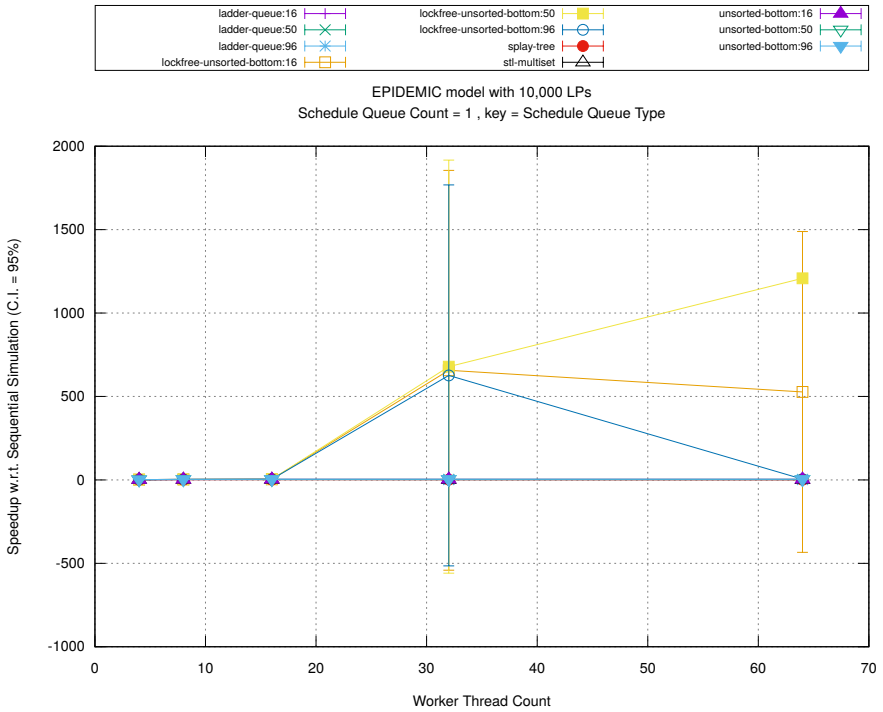


(b) Event Commitment Ratio

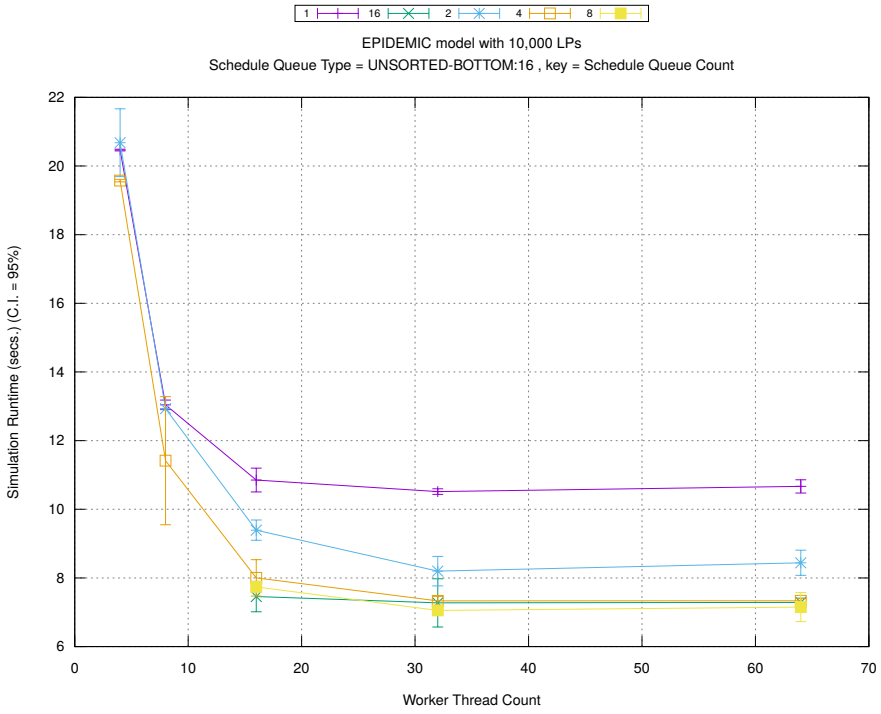


(c) Event Processing Rate (per second)

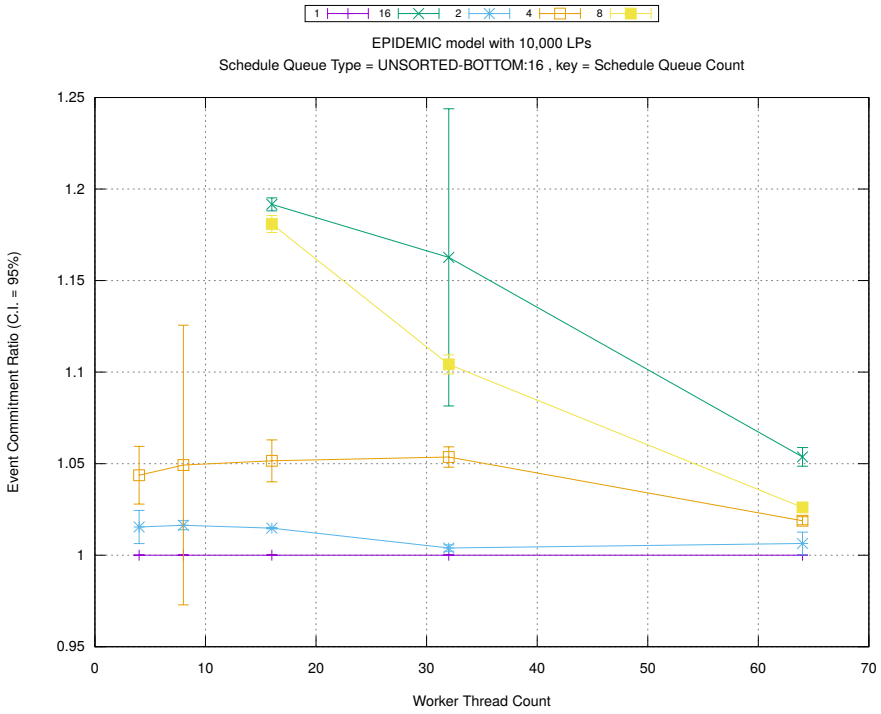
Figure A.158: epidemic 10k ba/plots/scheduleq/threads vs type key count 1



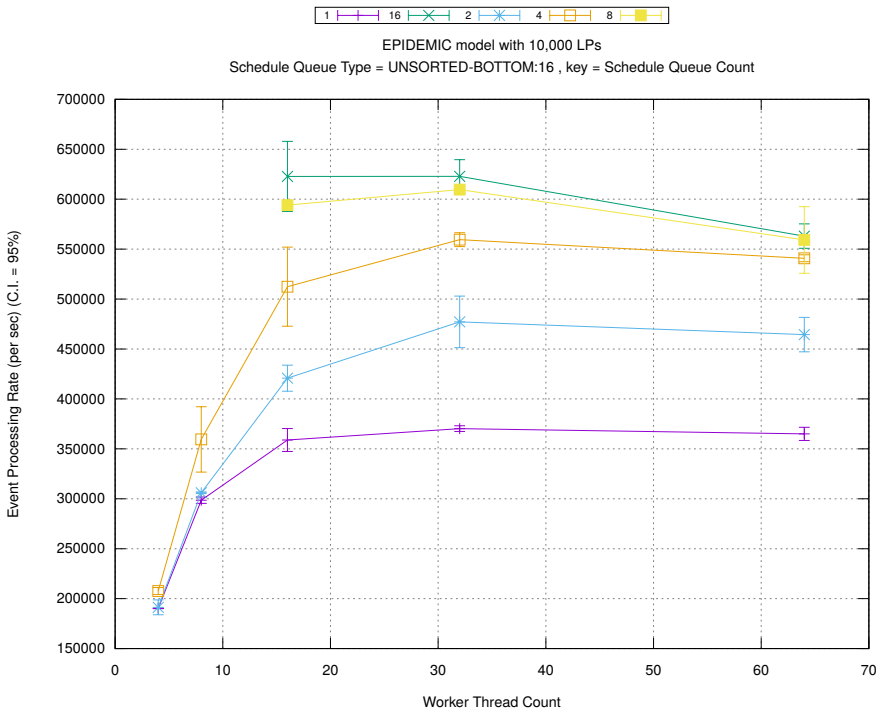
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

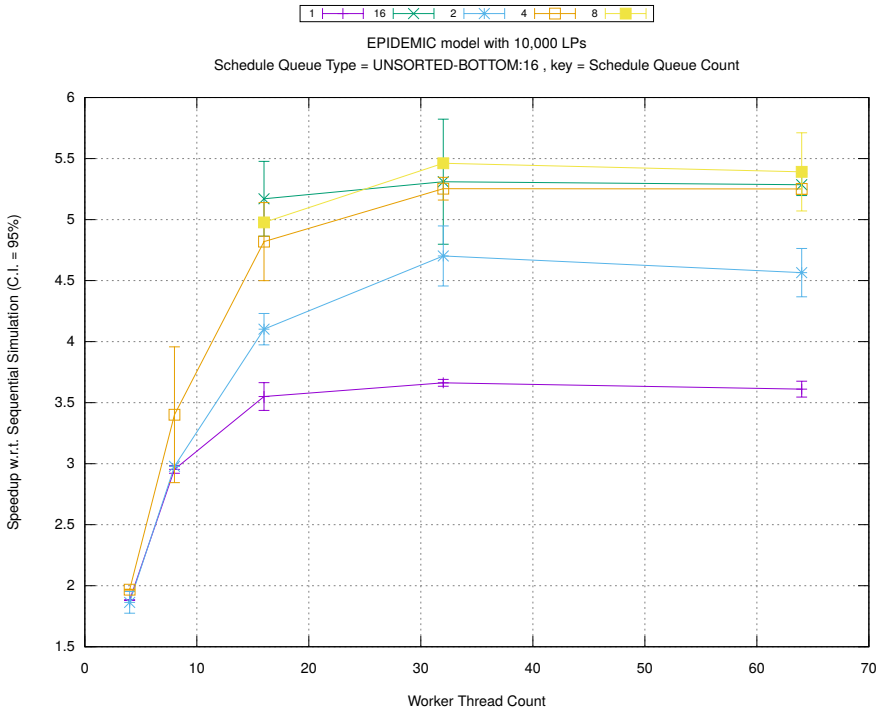


(b) Event Commitment Ratio

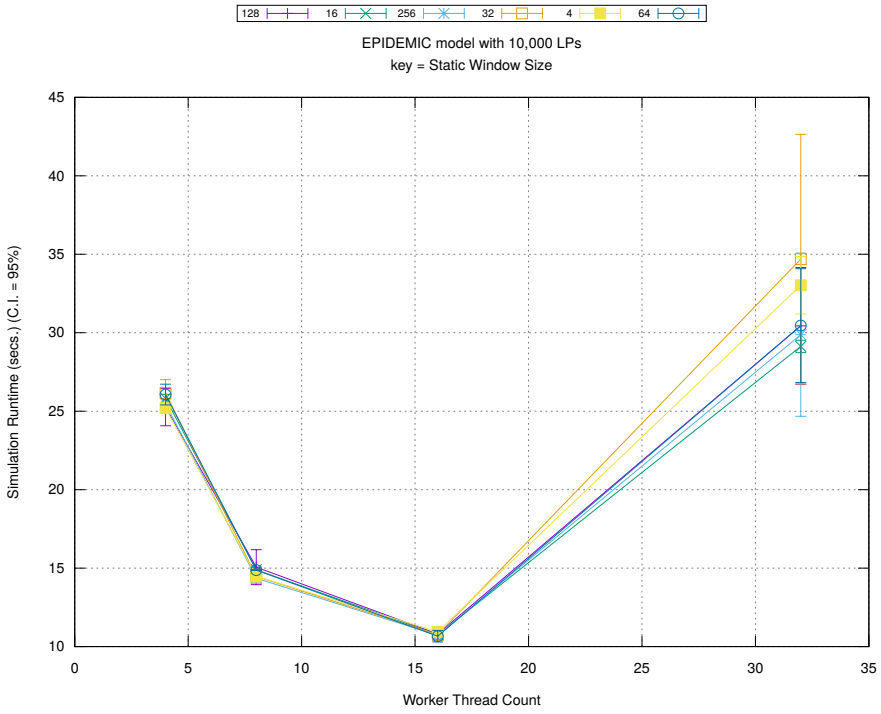


(c) Event Processing Rate (per second)

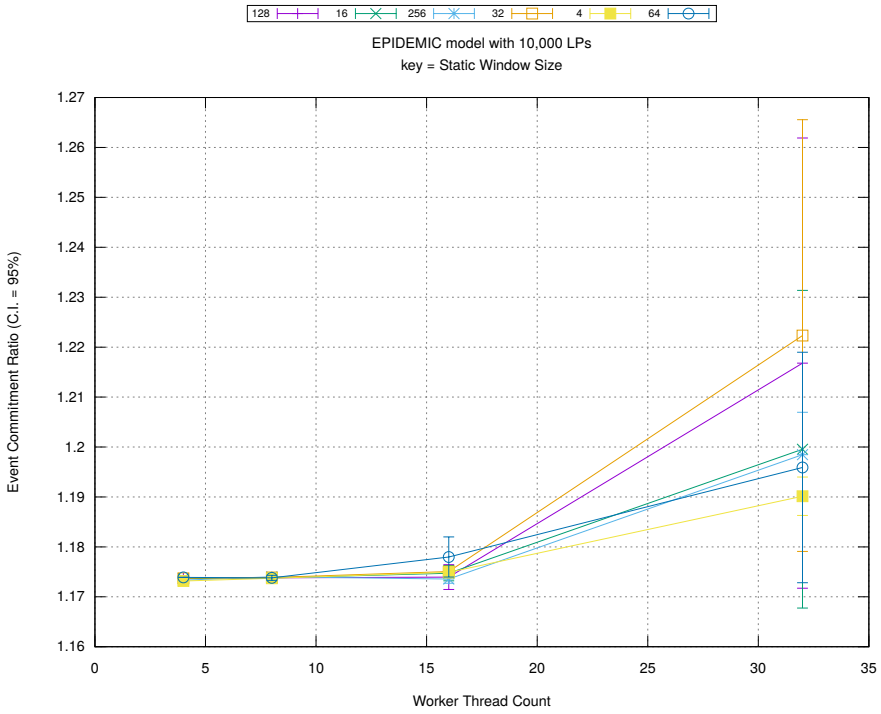
Figure A.159: epidemic 10k ba/plots/scheduleq/threads vs count key type unsorted-bottom 16



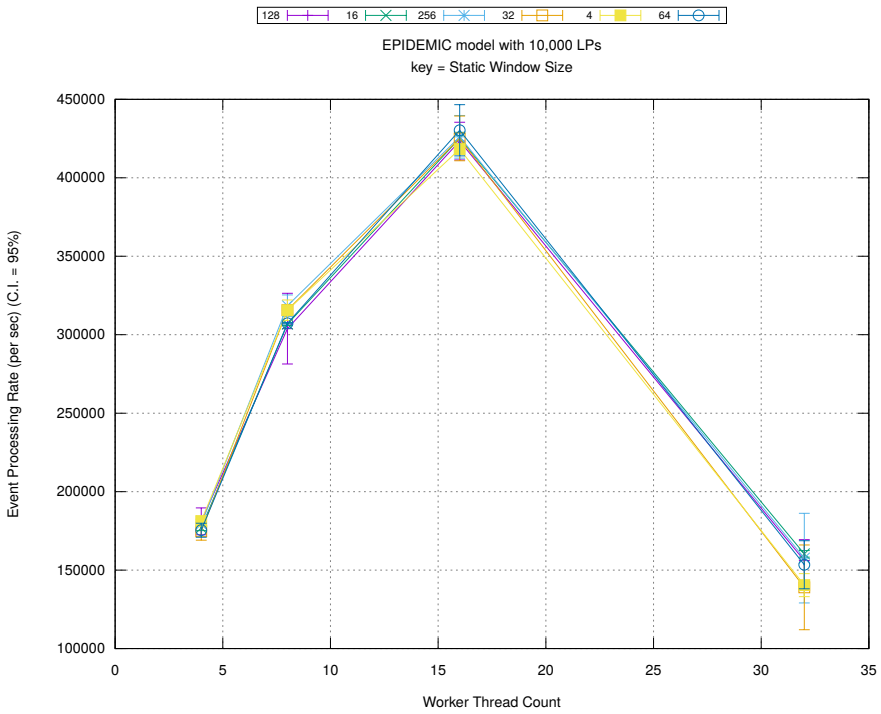
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

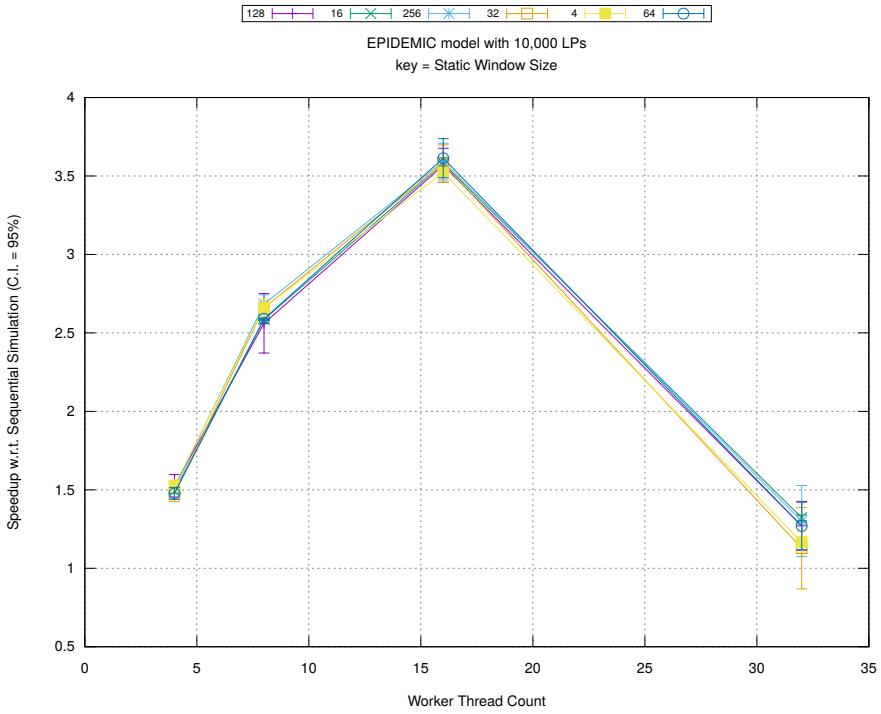


(b) Event Commitment Ratio

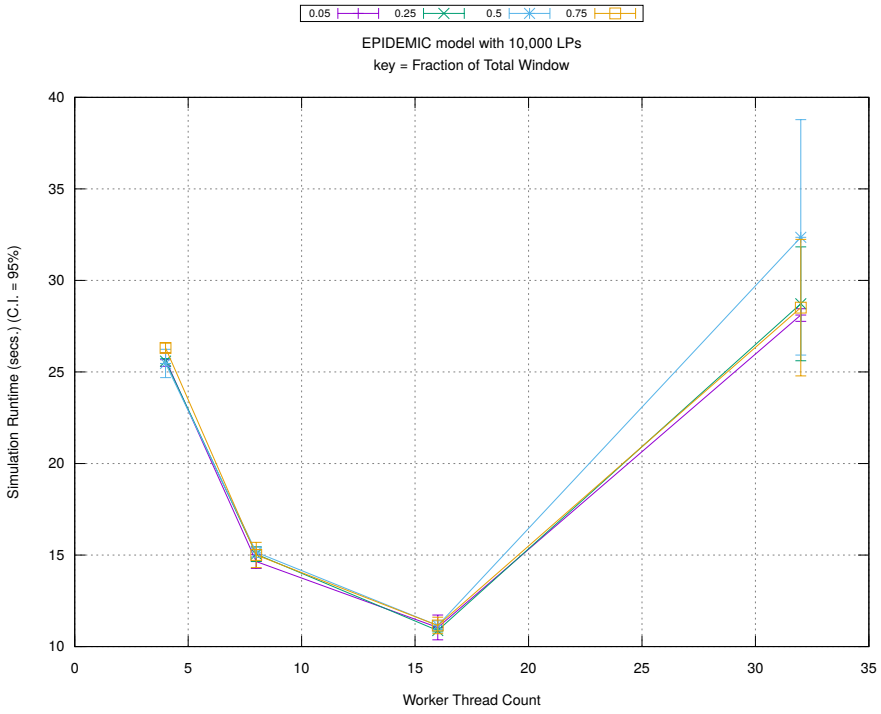


(c) Event Processing Rate (per second)

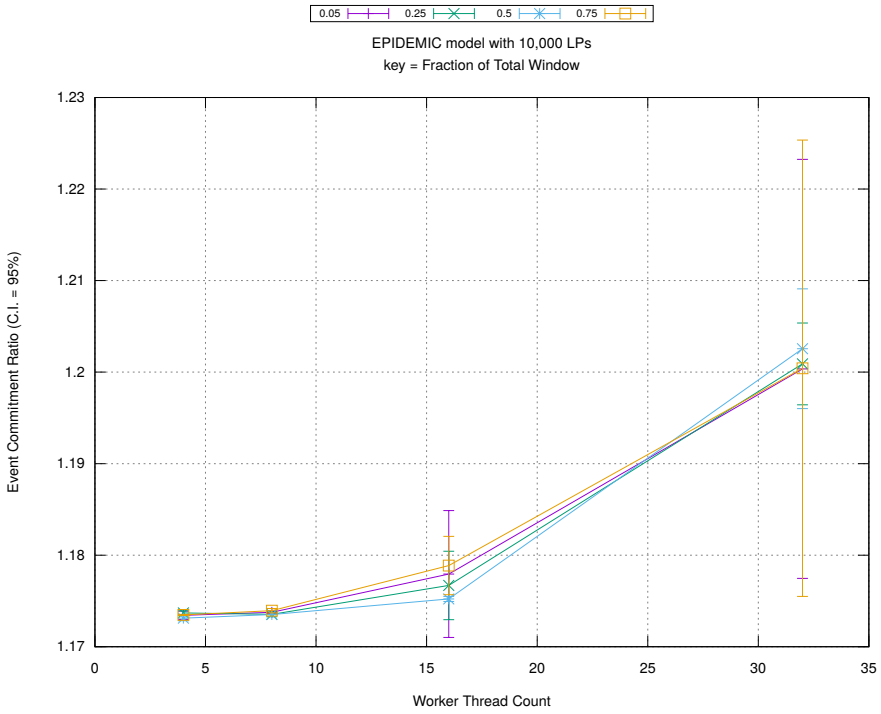
Figure A.160: epidemic 10k ba/plots/bags/threads vs staticwindow key fractionwindow 1.0



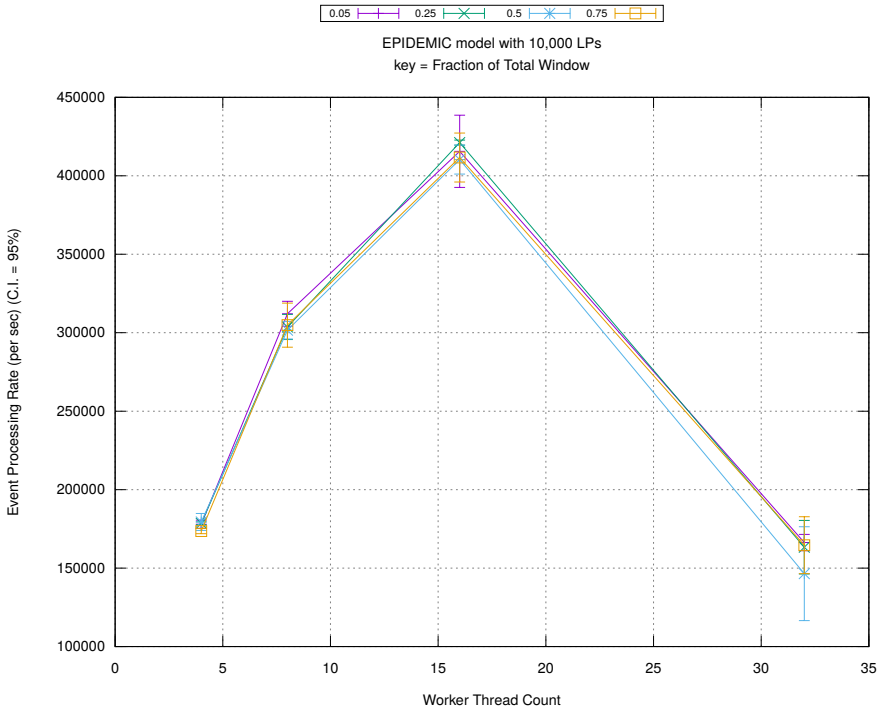
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

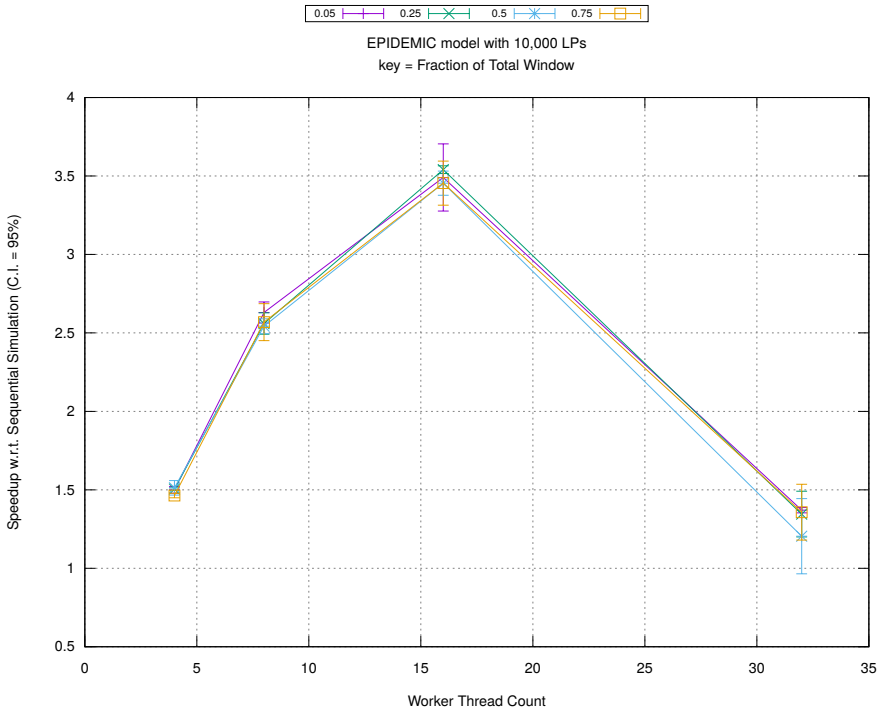


(b) Event Commitment Ratio

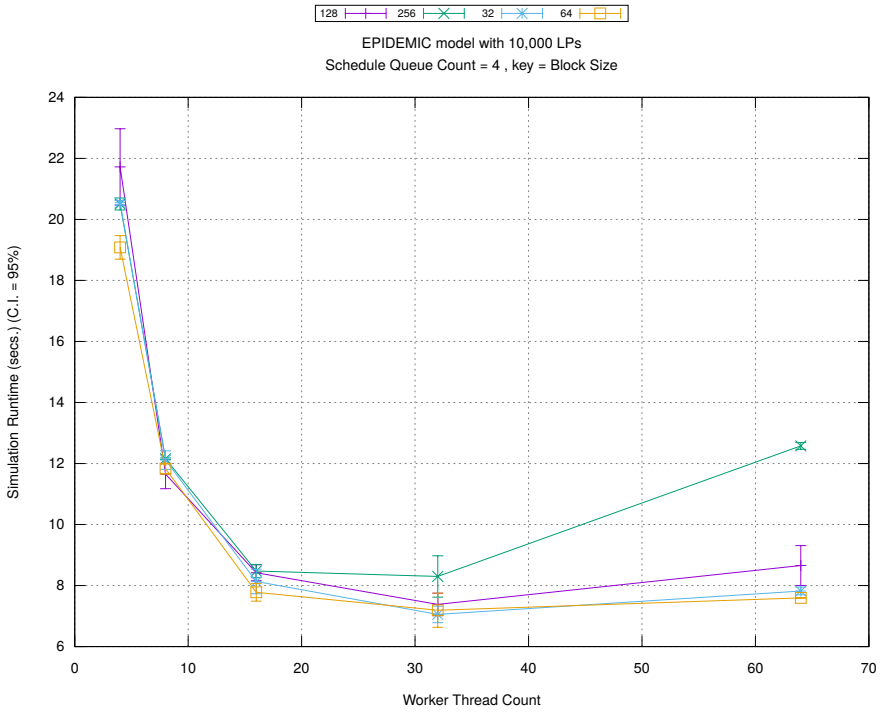


(c) Event Processing Rate (per second)

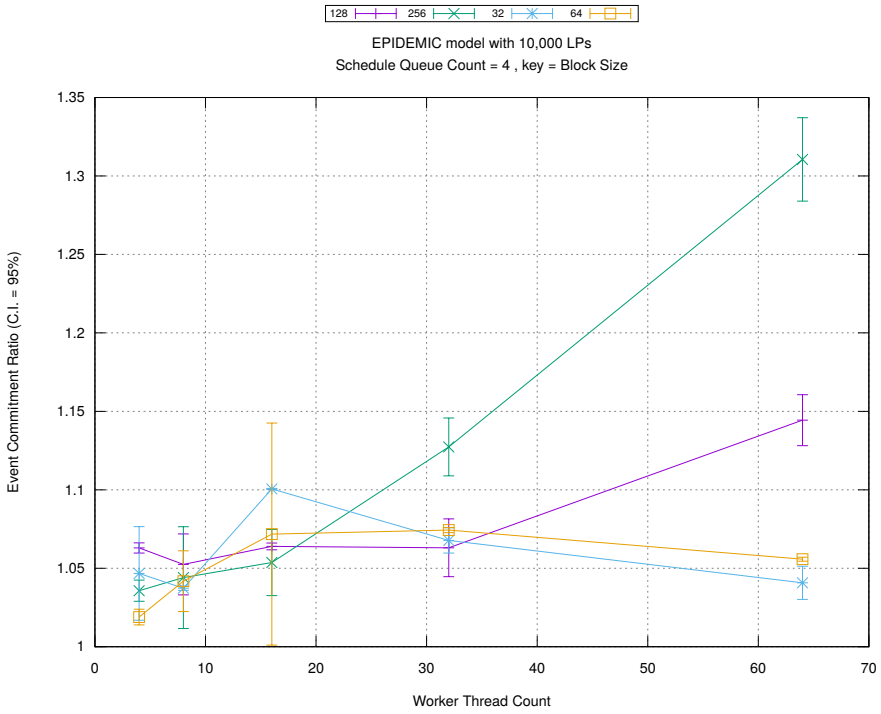
Figure A.161: epidemic 10k ba/plots/bags/threads vs fractionwindow key staticwindow 0



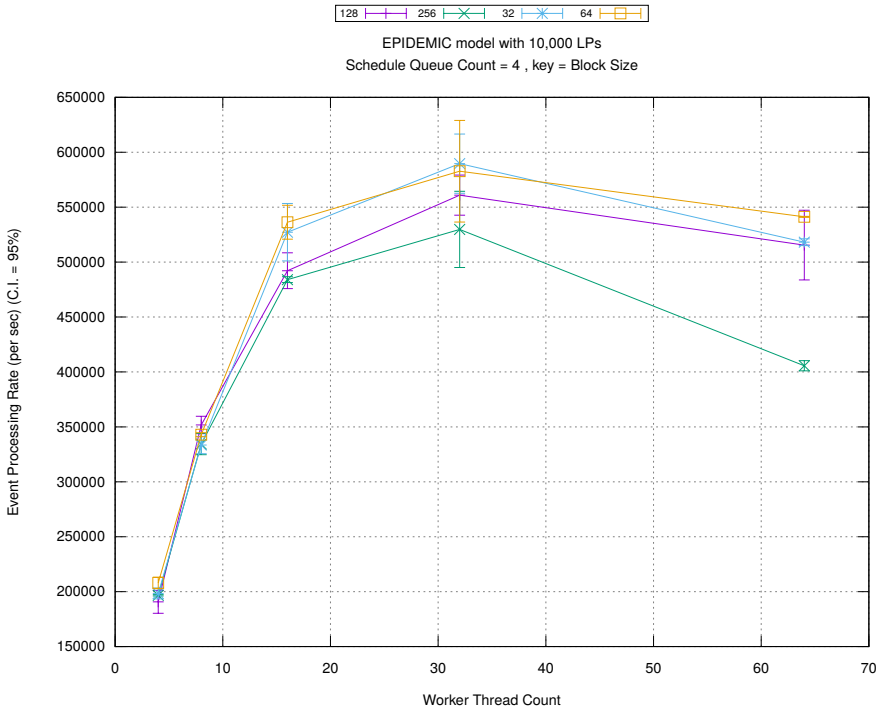
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

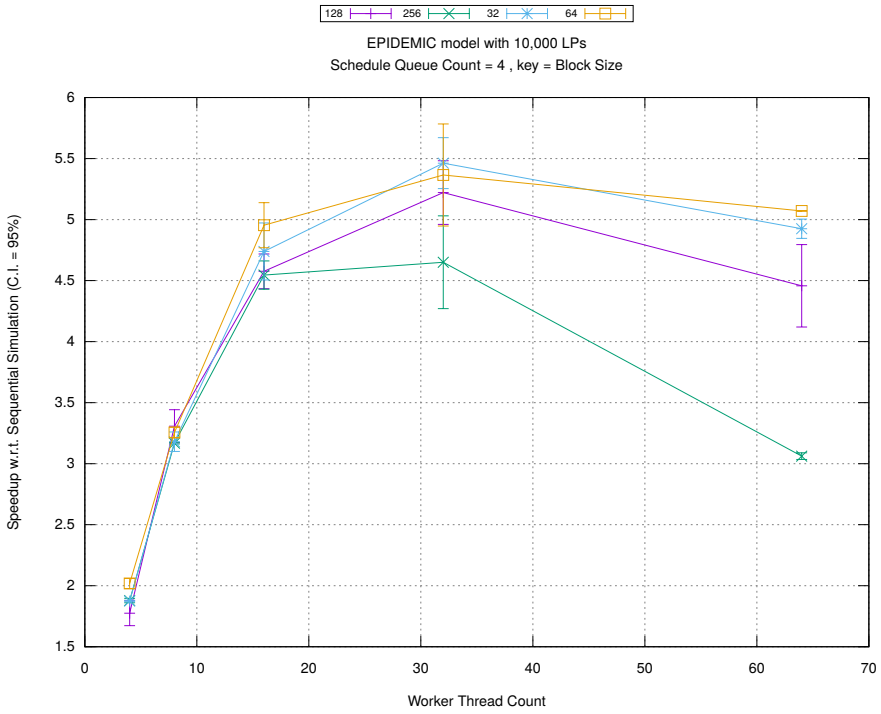


(b) Event Commitment Ratio

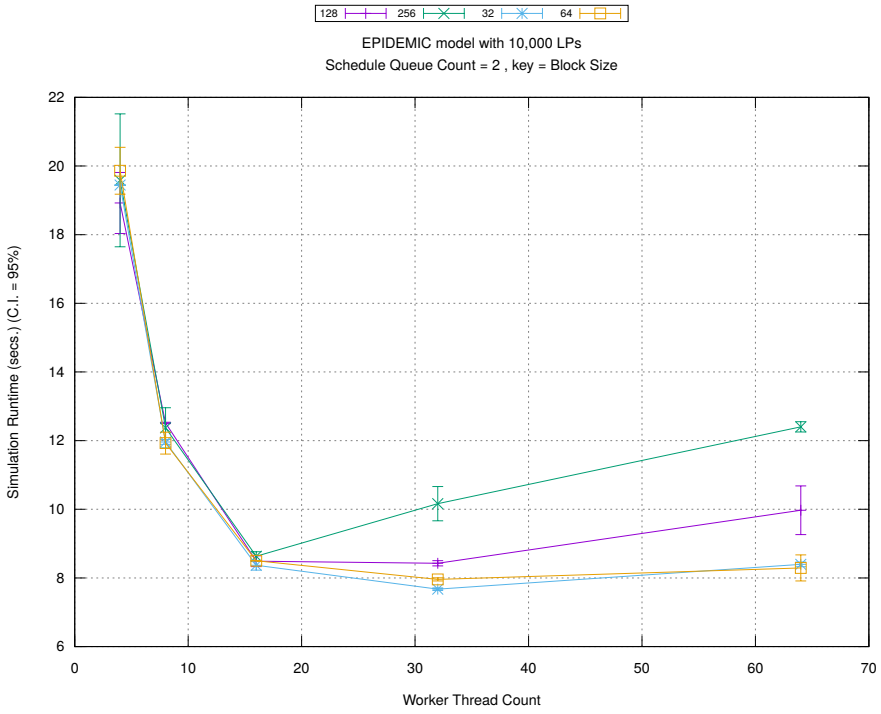


(c) Event Processing Rate (per second)

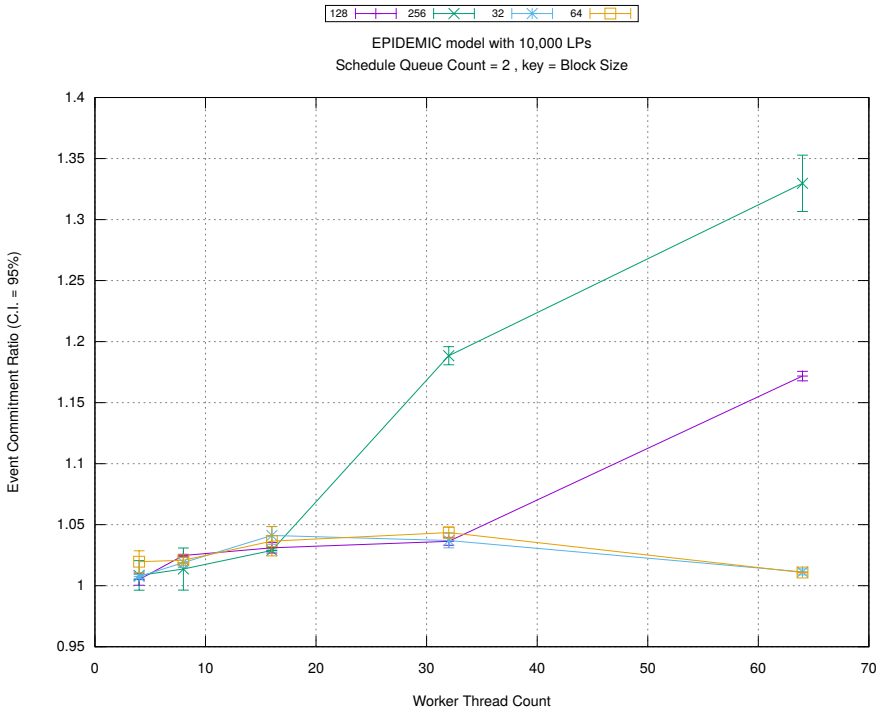
Figure A.162: epidemic 10k ba/plots/blocks/threads vs blocksize key count 4



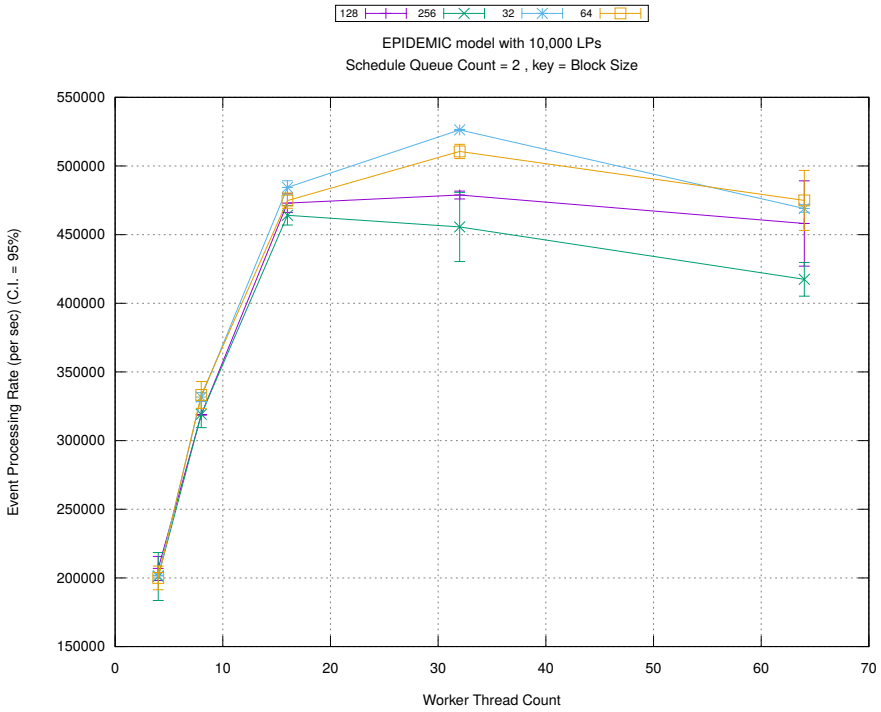
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

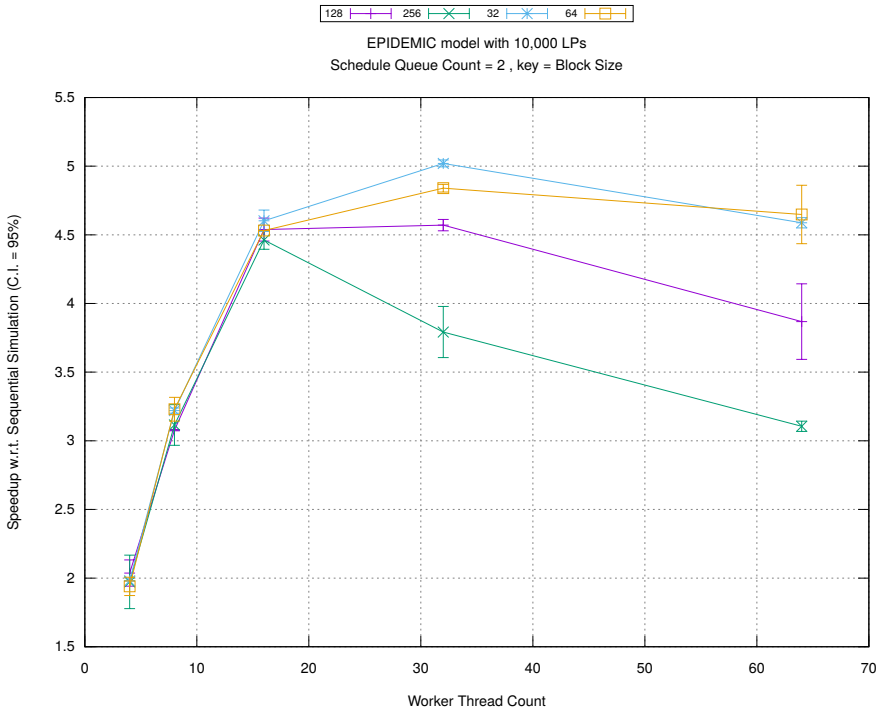


(b) Event Commitment Ratio

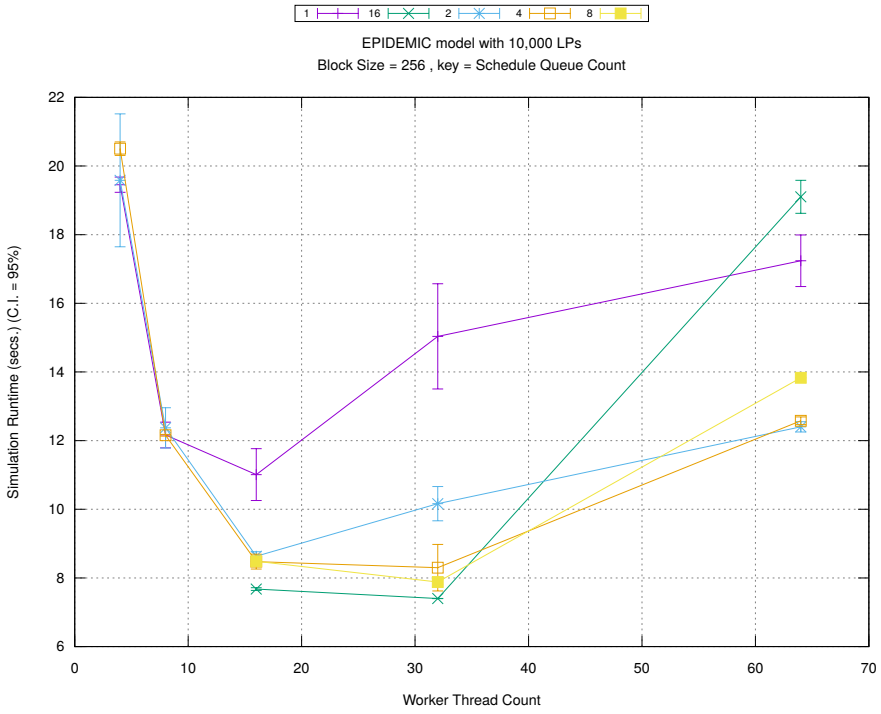


(c) Event Processing Rate (per second)

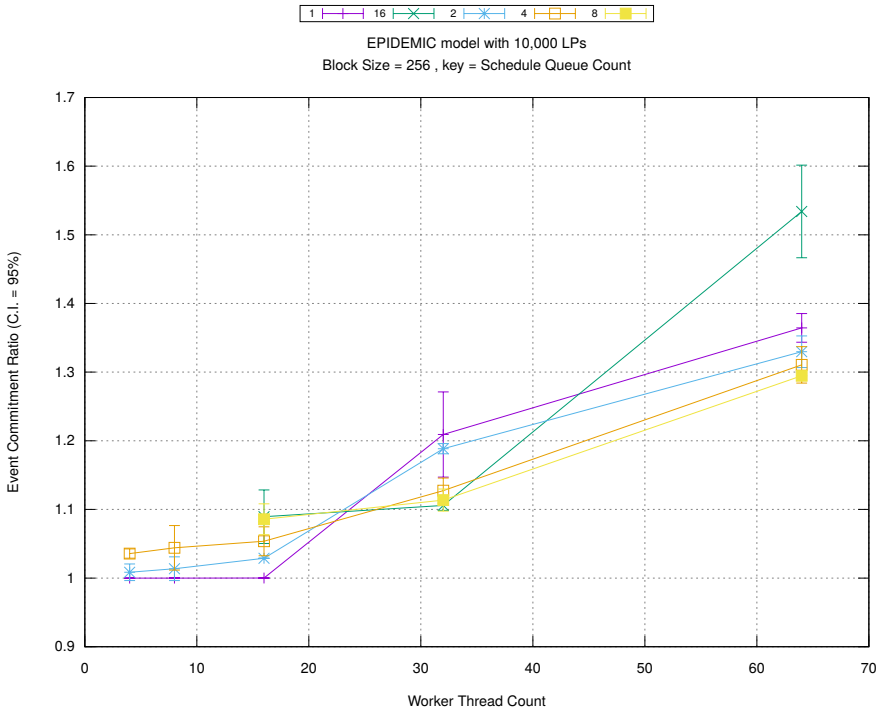
Figure A.163: epidemic 10k ba/plots/blocks/threads vs blocksize key count 2



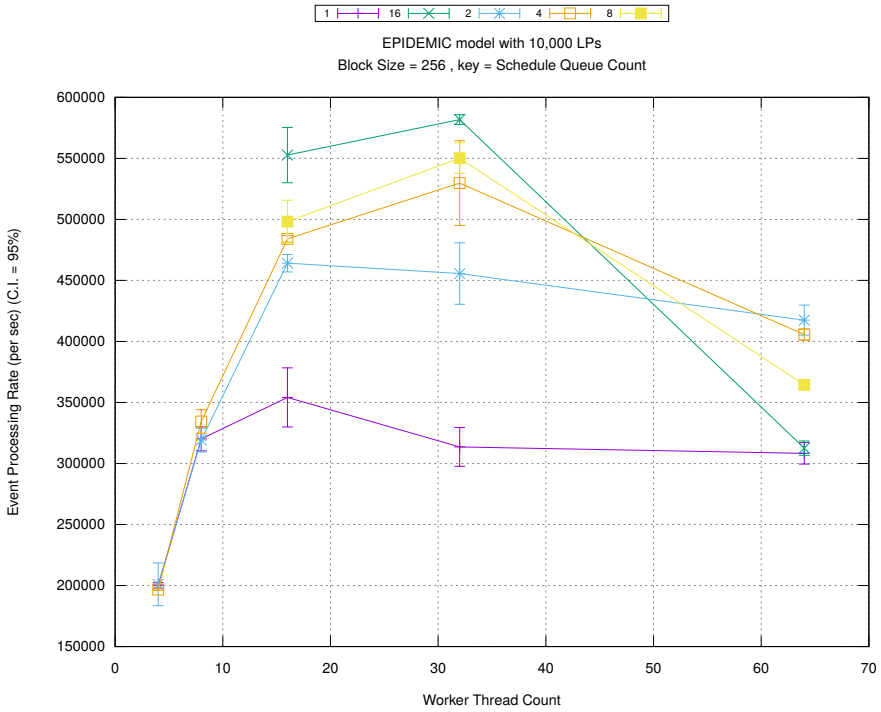
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

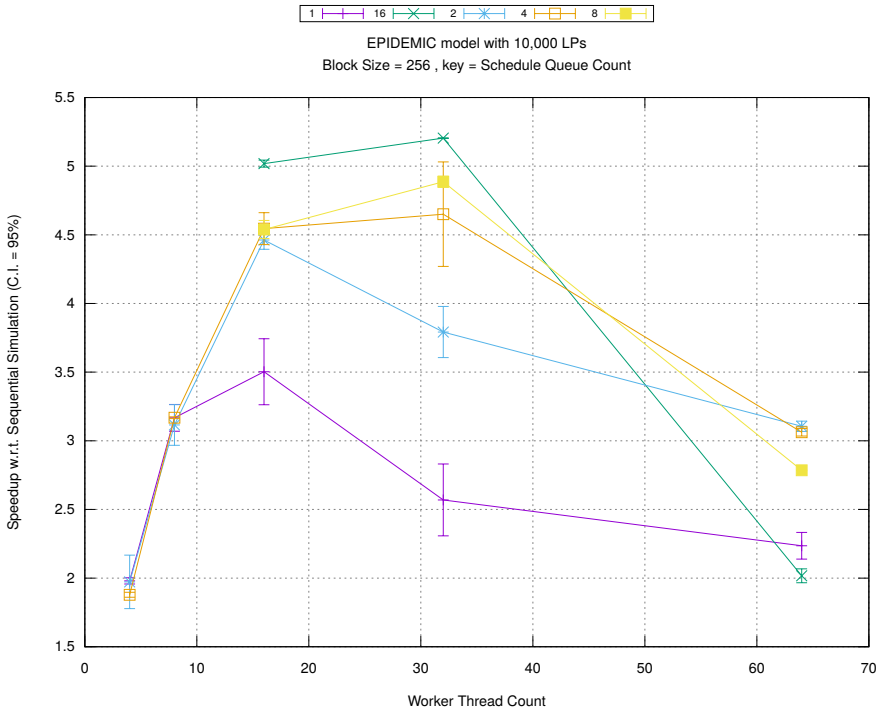


(b) Event Commitment Ratio

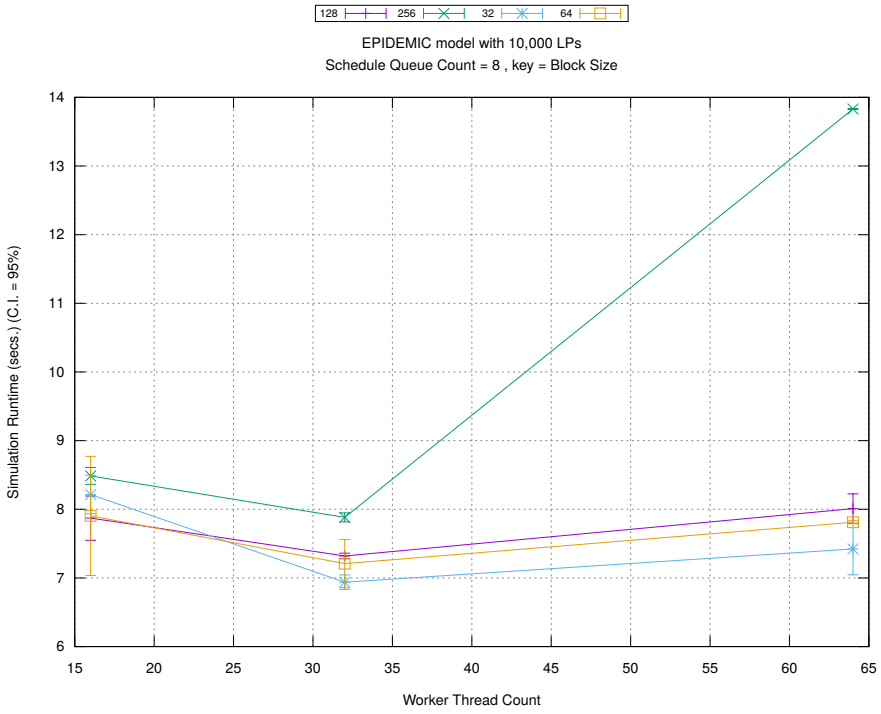


(c) Event Processing Rate (per second)

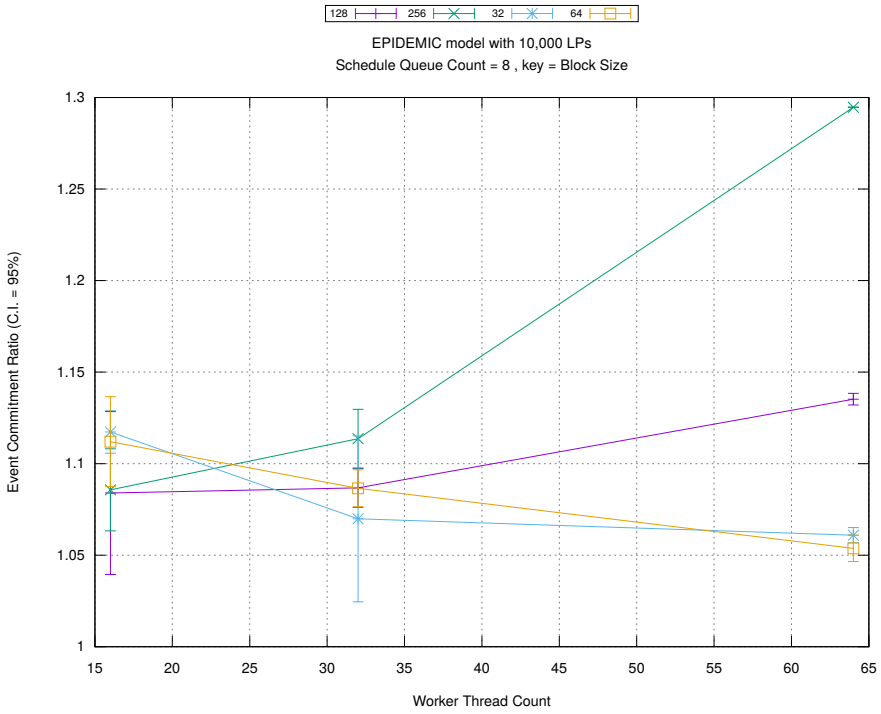
Figure A.164: epidemic 10k ba/plots/blocks/threads vs count key blocksize 256



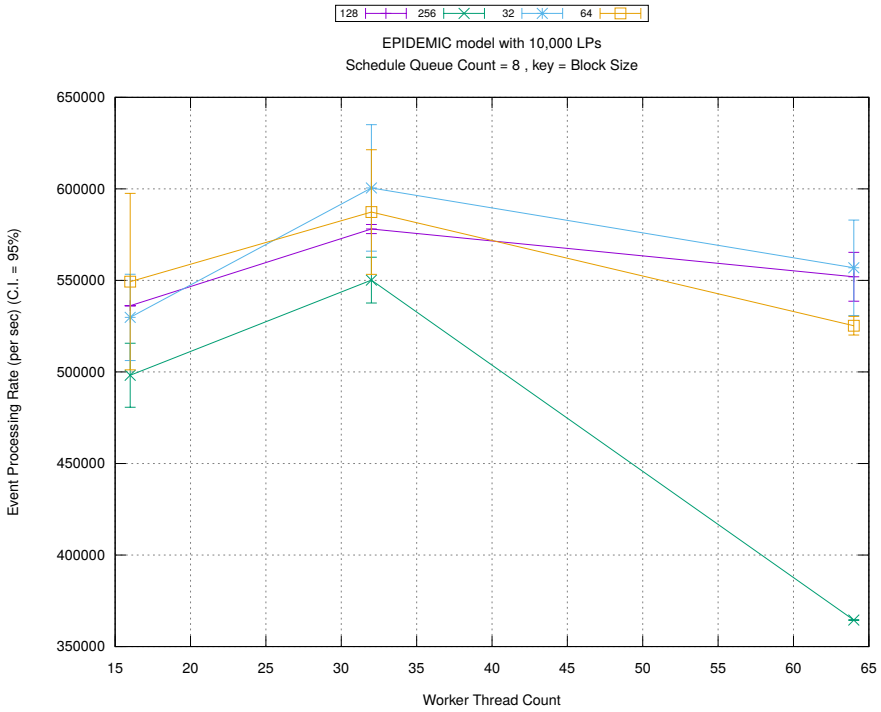
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

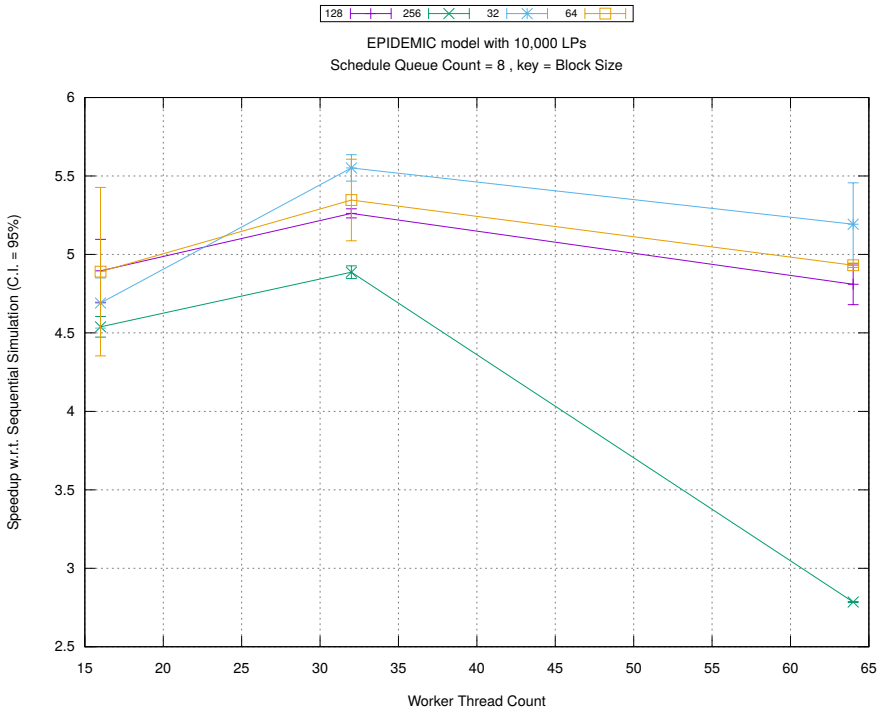


(b) Event Commitment Ratio

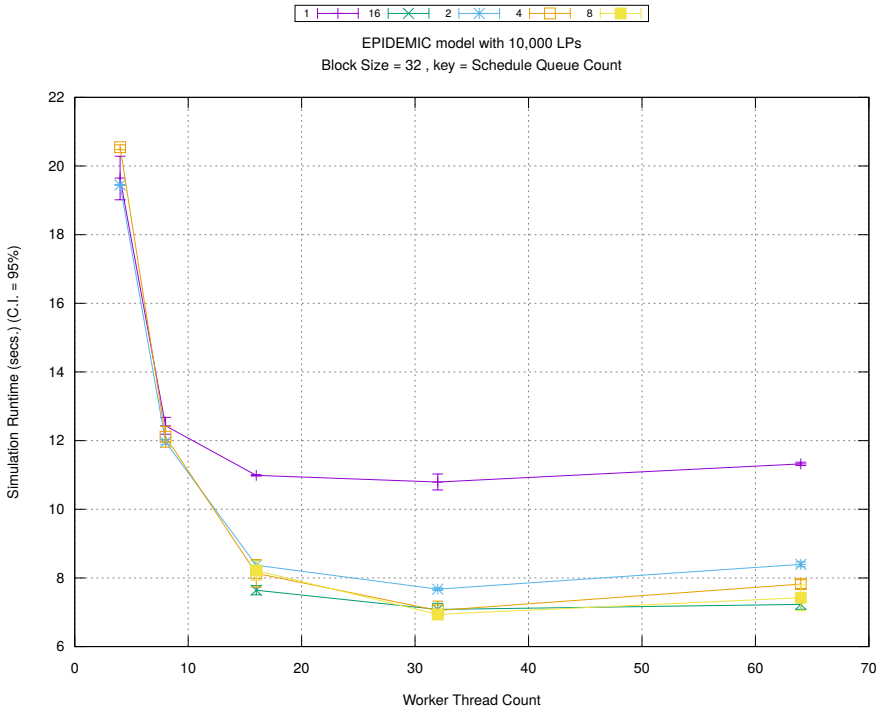


(c) Event Processing Rate (per second)

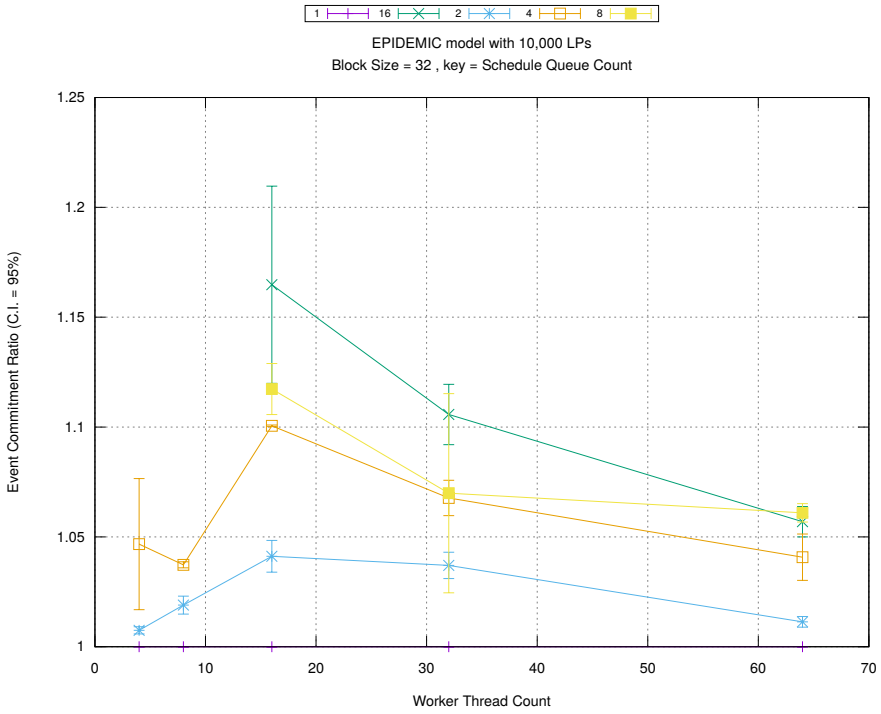
Figure A.165: epidemic 10k ba/plots/blocks/threads vs blocksize key count 8



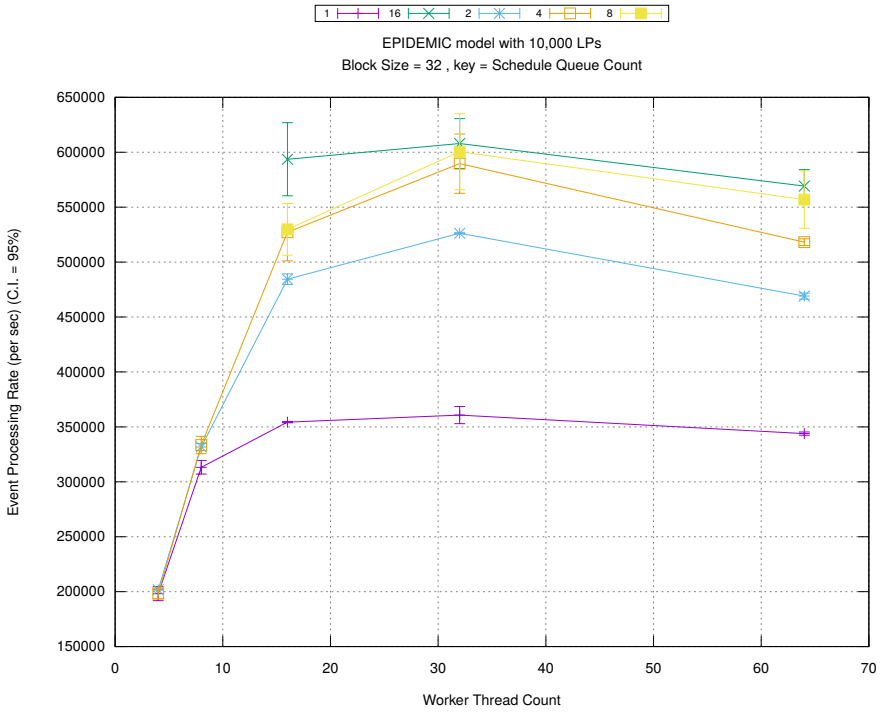
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

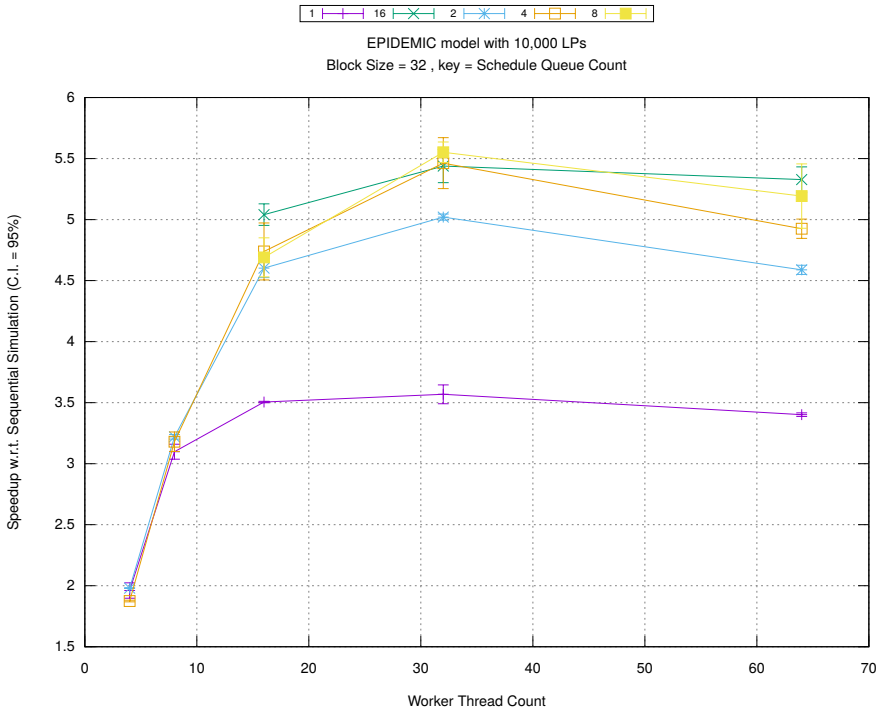


(b) Event Commitment Ratio

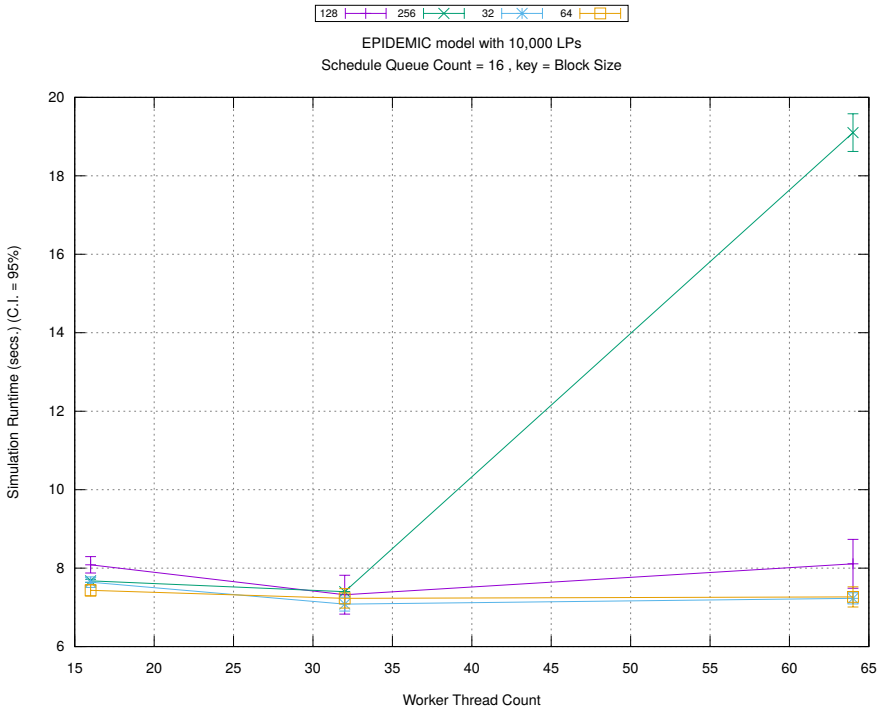


(c) Event Processing Rate (per second)

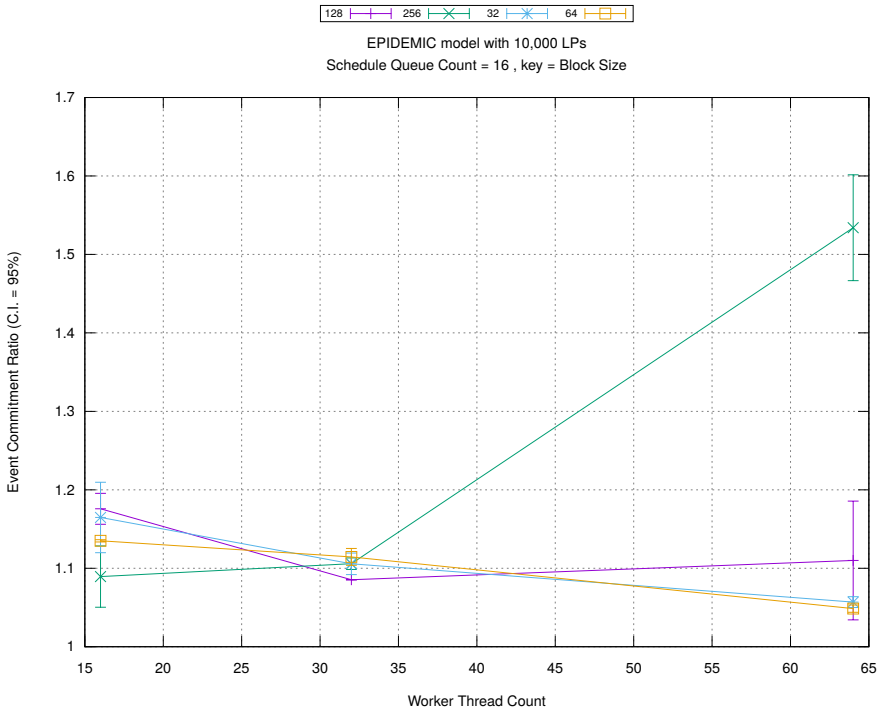
Figure A.166: epidemic 10k ba/plots/blocks/threads vs count key blocksize 32



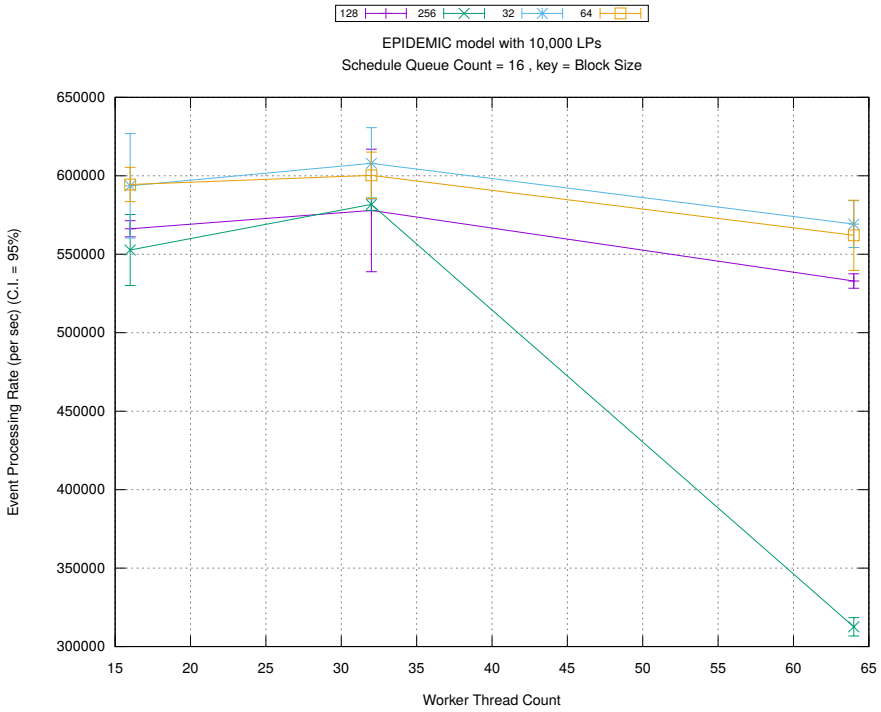
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

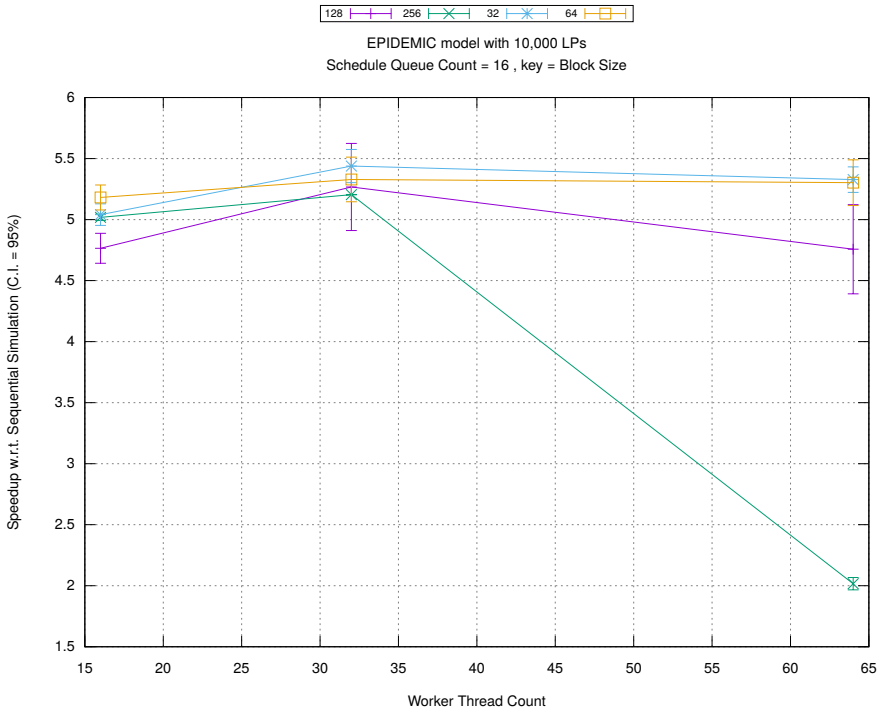


(b) Event Commitment Ratio

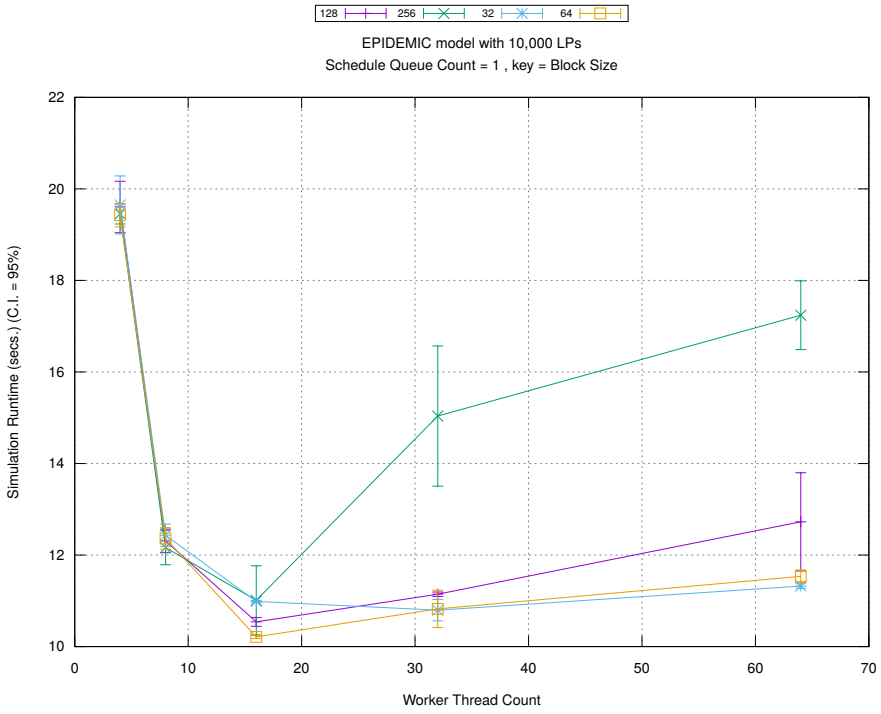


(c) Event Processing Rate (per second)

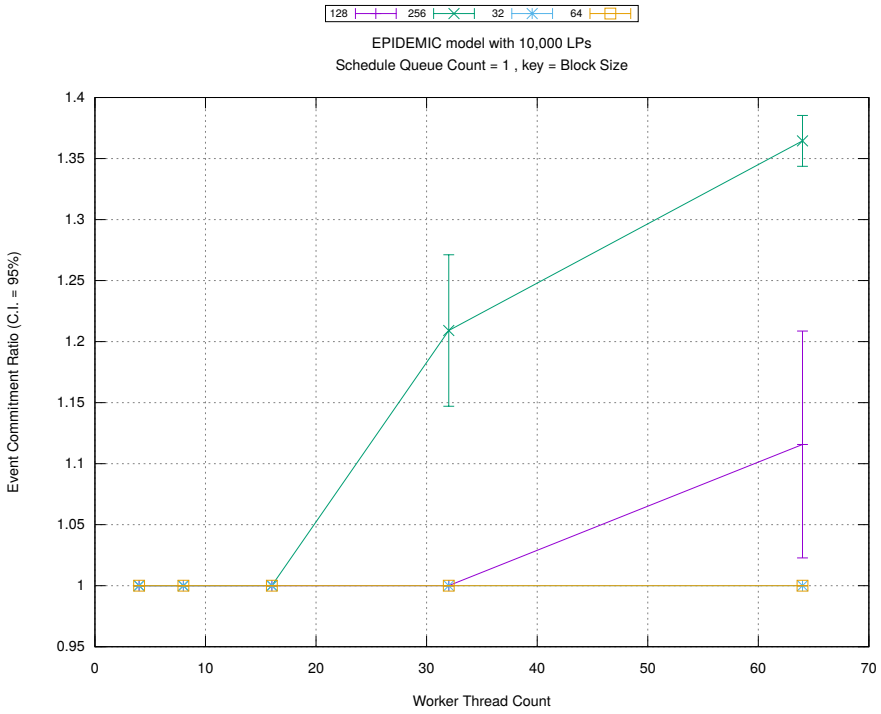
Figure A.167: epidemic 10k ba/plots/blocks/threads vs blocksize key count 16



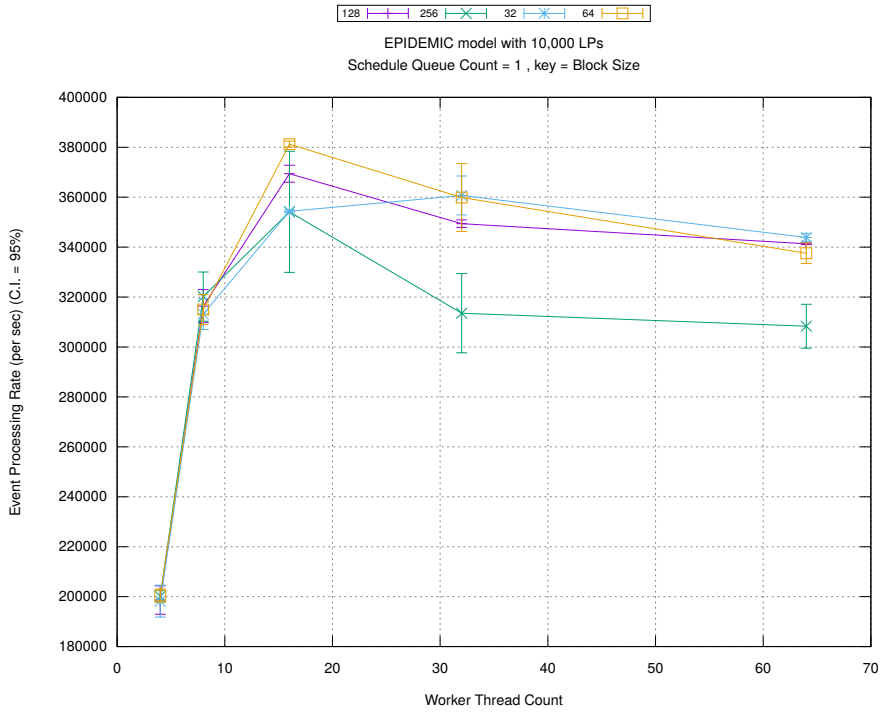
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

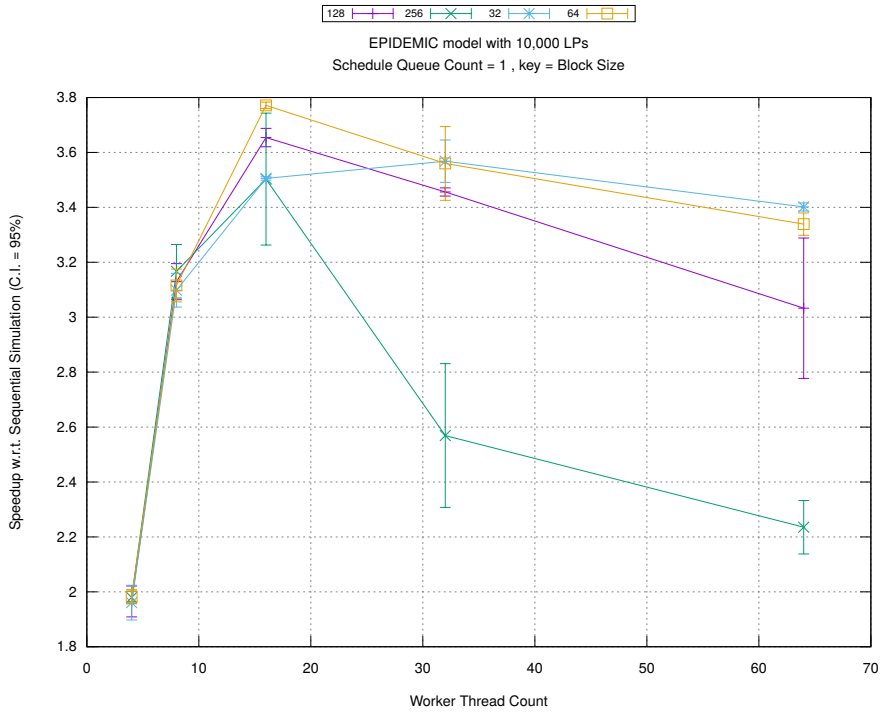


(b) Event Commitment Ratio

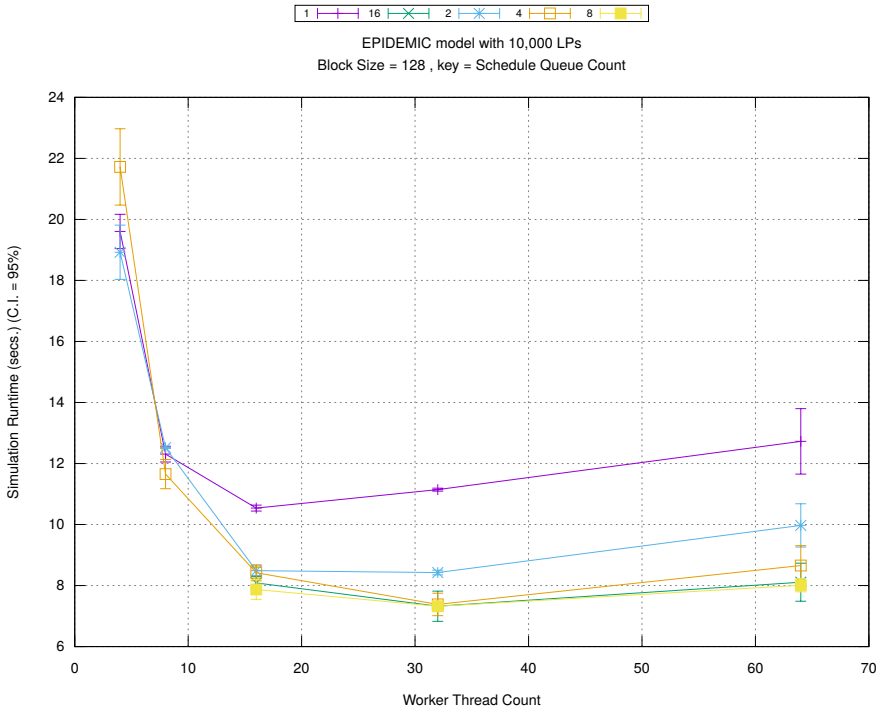


(c) Event Processing Rate (per second)

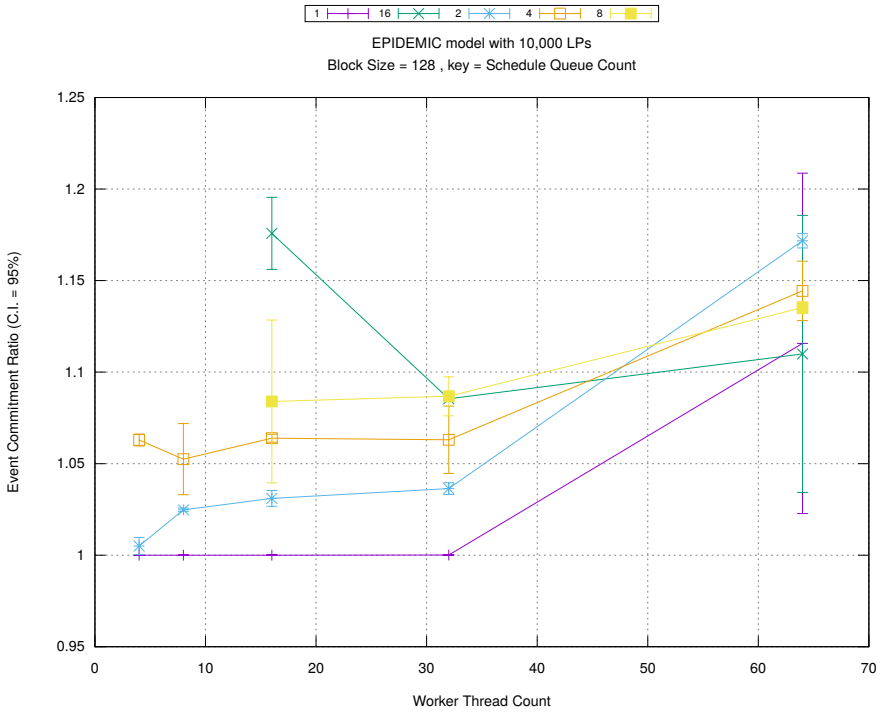
Figure A.168: epidemic 10k ba/plots/blocks/threads vs blocksize key count 1



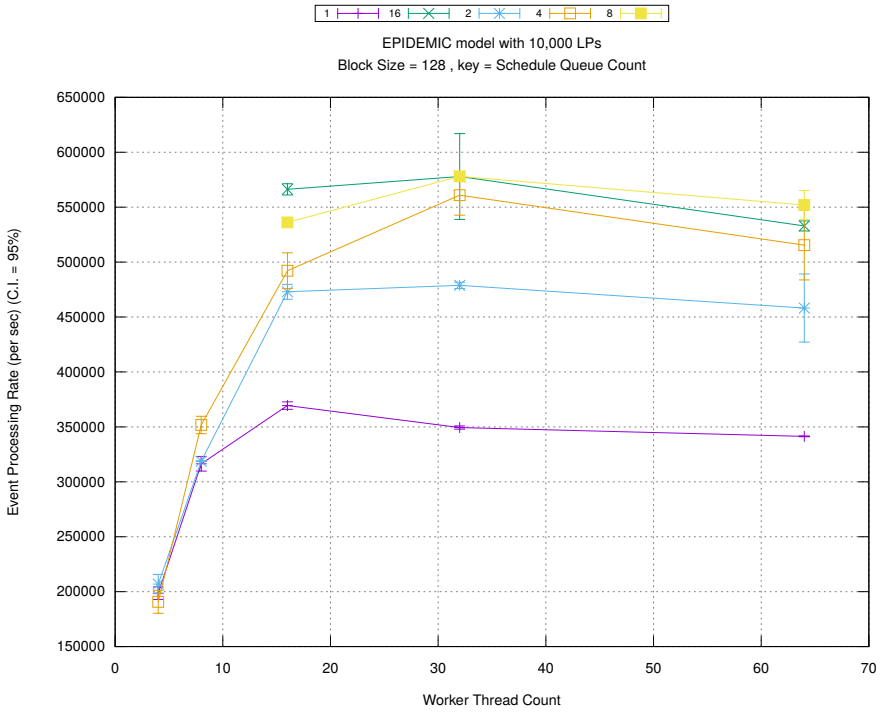
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

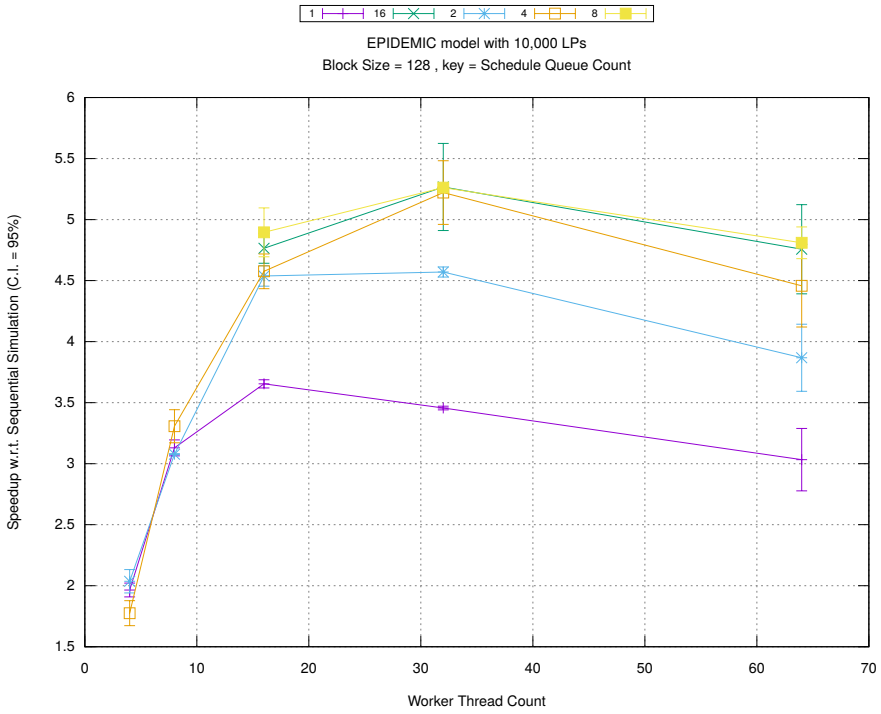


(b) Event Commitment Ratio

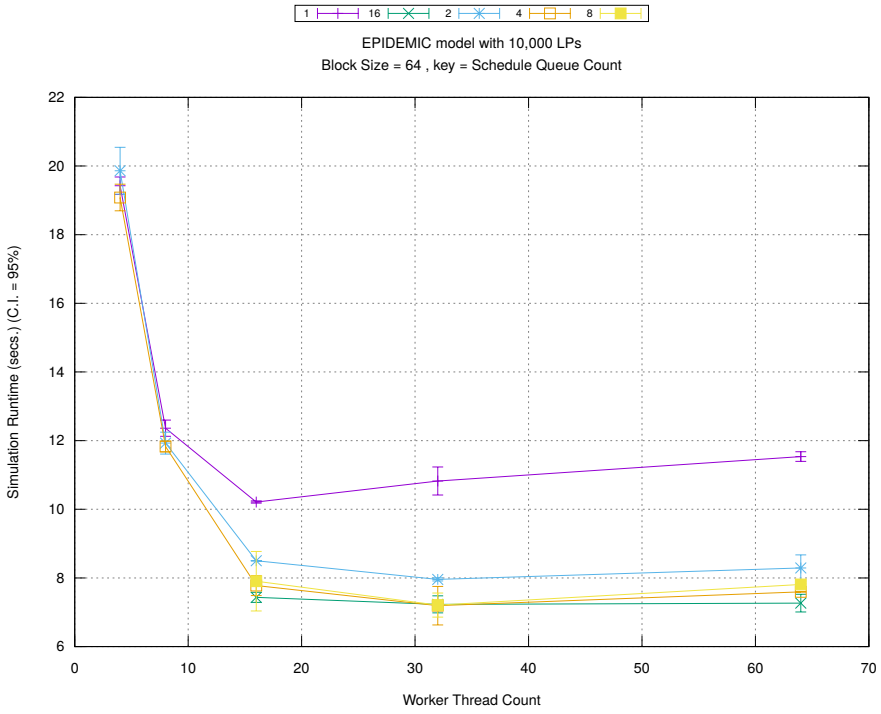


(c) Event Processing Rate (per second)

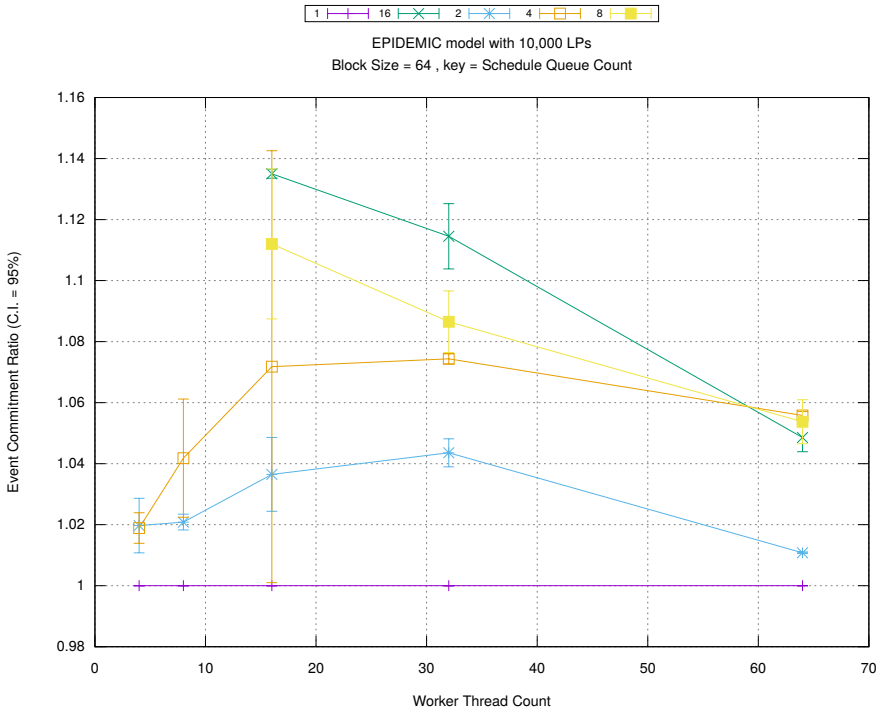
Figure A.169: epidemic 10k ba/plots/blocks/threads vs count key blocksize 128



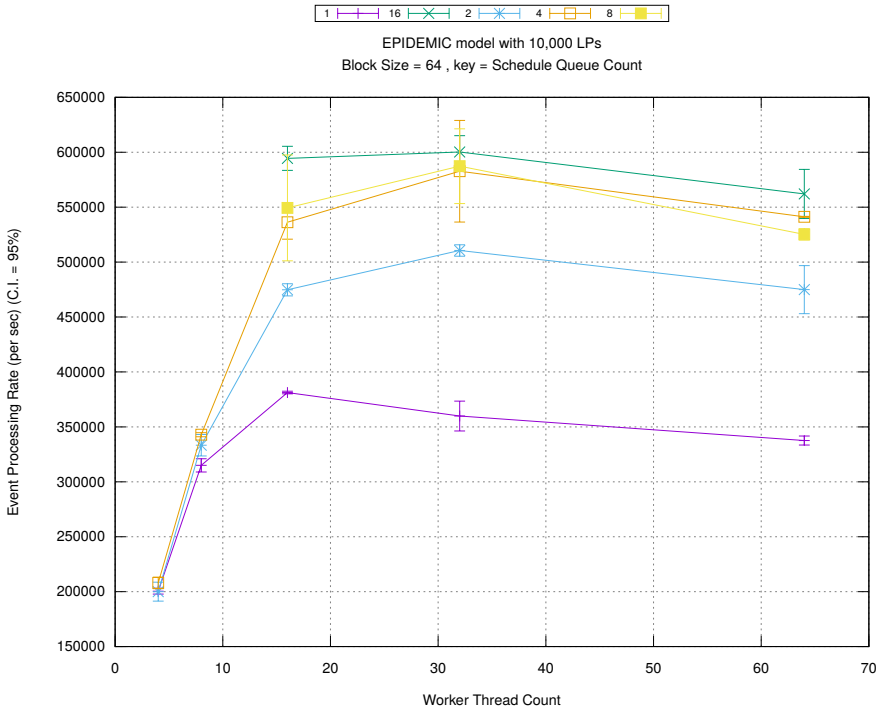
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

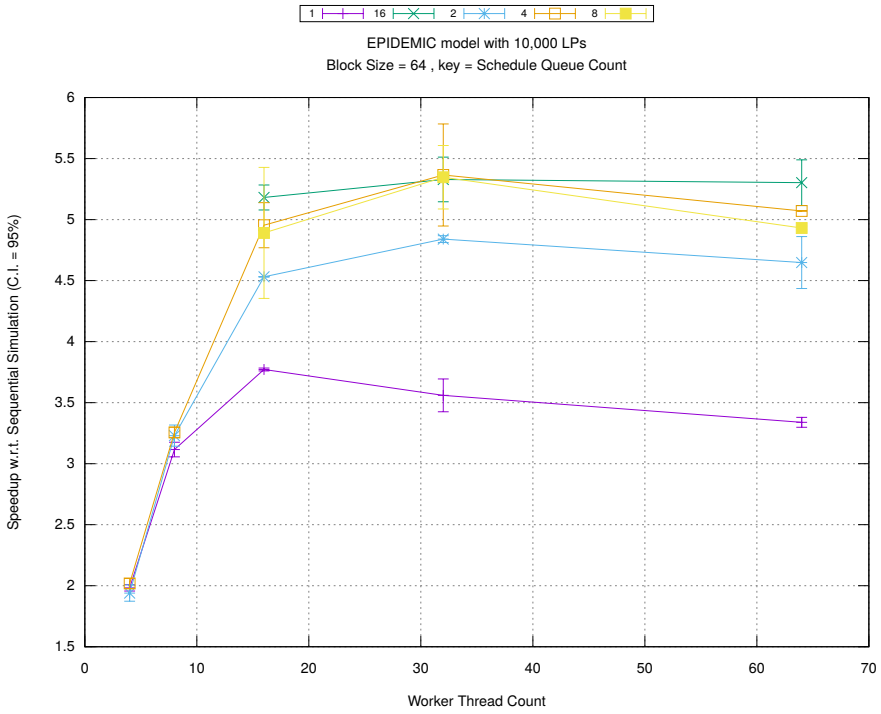


(b) Event Commitment Ratio



(c) Event Processing Rate (per second)

Figure A.170: epidemic 10k ba/plots/blocks/threads vs count key blocksize 64



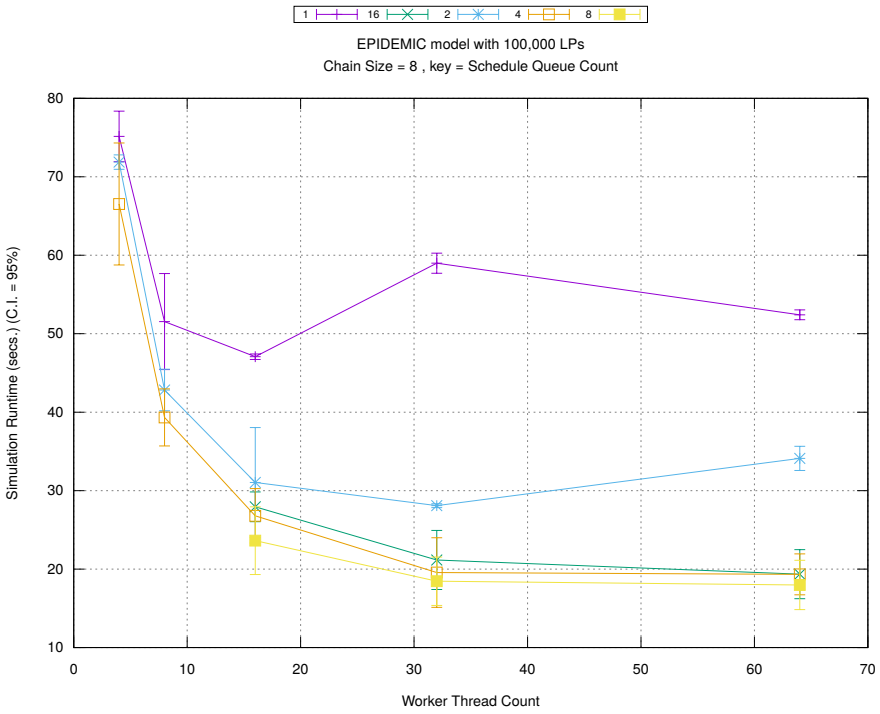
(d) Speedup w.r.t. Sequential Simulation

A.6 Epidemic Model with 100,000 LPs and Barabasi-Albert Network

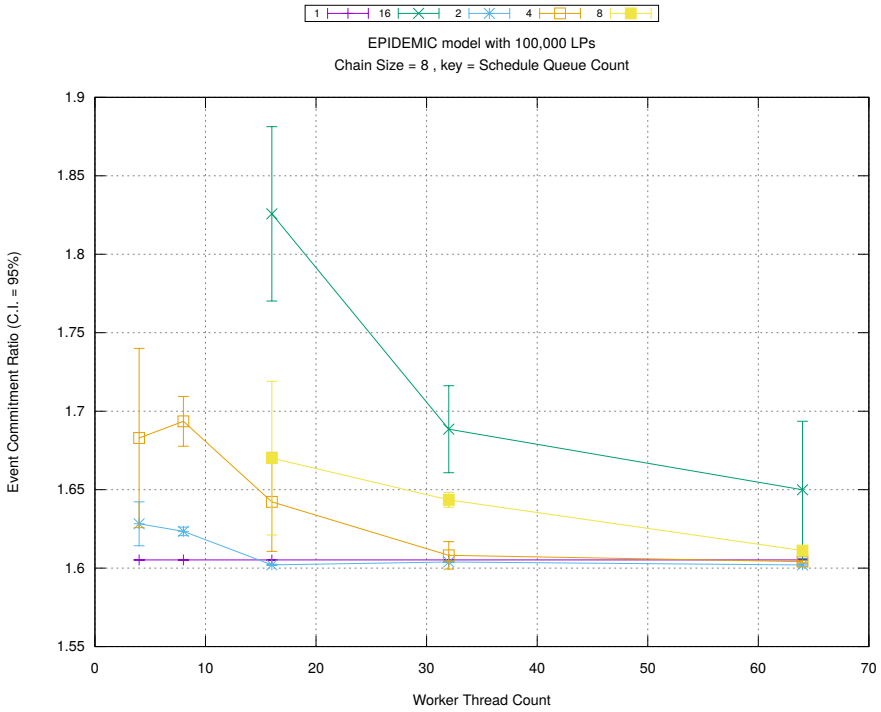
Table A.6 shows the configuration for this model.

Parameter	Values
Number of Intersections (or LPs)	100,000
Type of network connecting LPs	Barabasi-Albert [73]
Population Size	500,000
Simulation Time	6,000 timestamp units
Sequential Simulation Time for calculating modularity	200 timestamp units

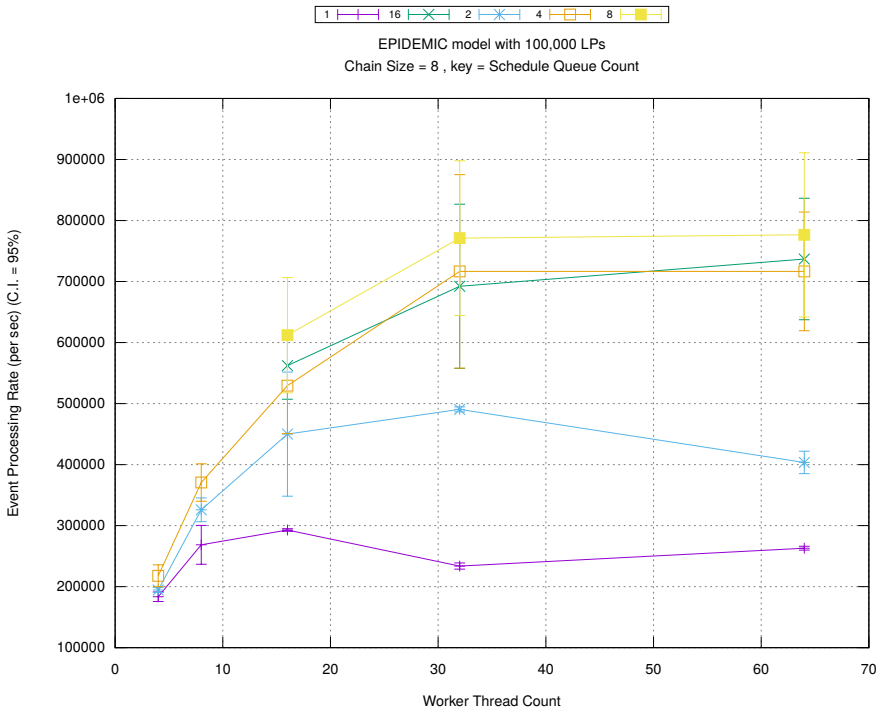
Table A.6: LARGE EPIDEMIC MODEL WITH BARABASI-ALBERT setup



(a) Simulation Runtime (in seconds)

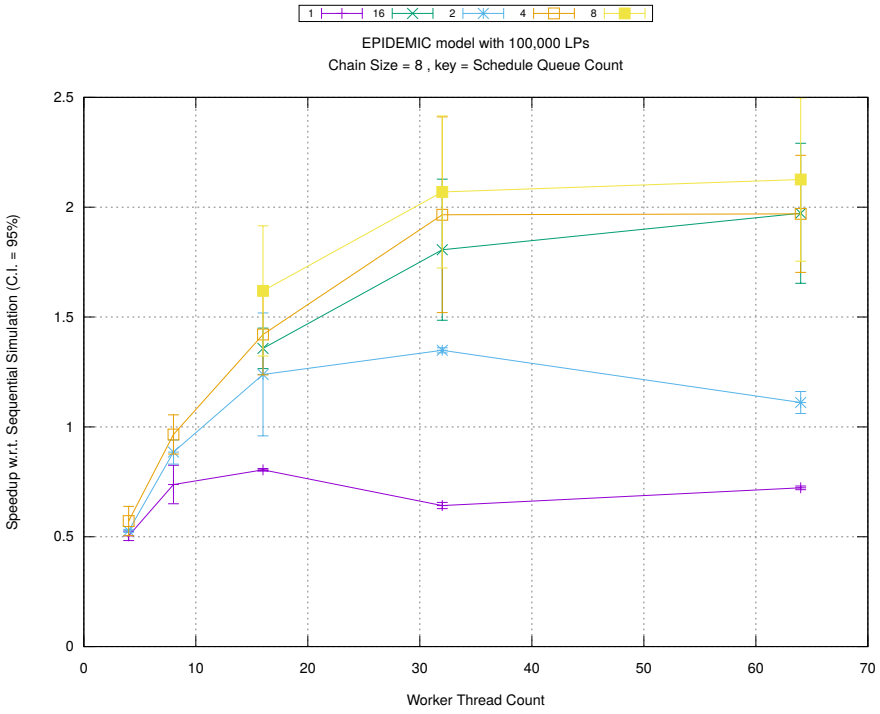


(b) Event Commitment Ratio

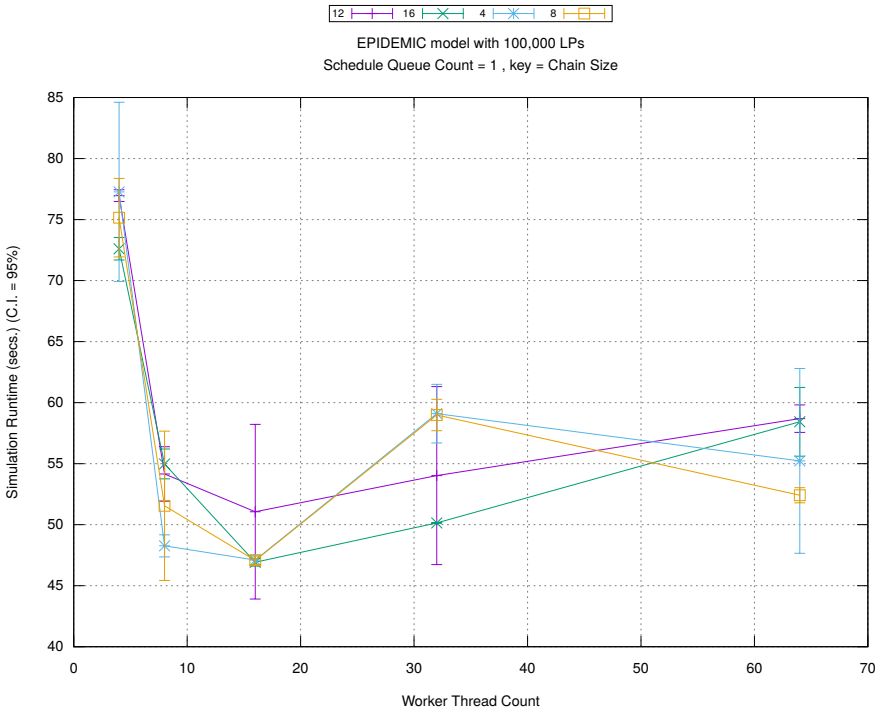


(c) Event Processing Rate (per second)

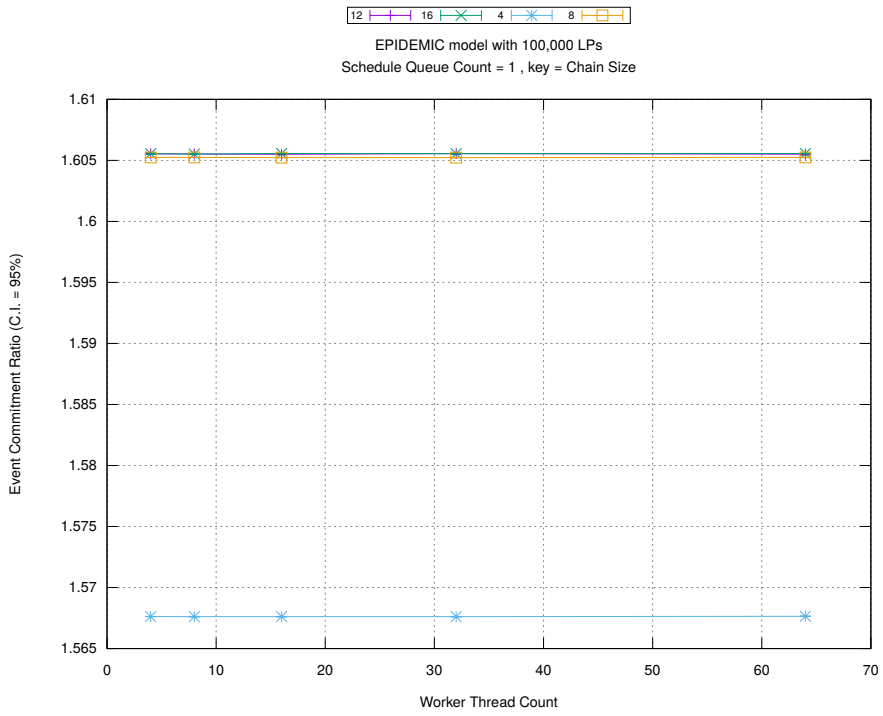
Figure A.171: epidemic 100k ba/plots/chains/threads vs count key chainsize 8



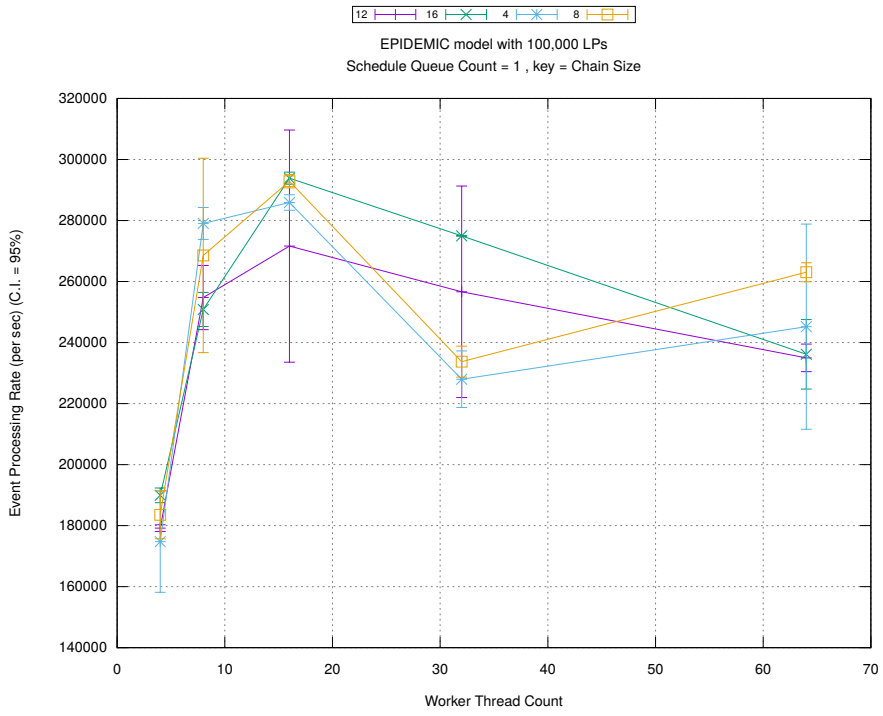
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

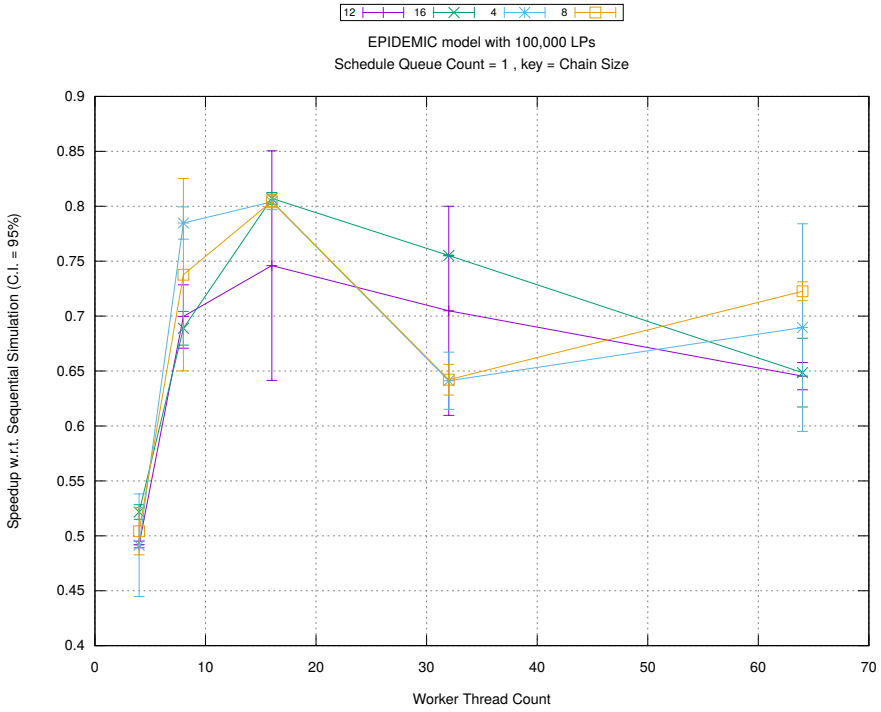


(b) Event Commitment Ratio

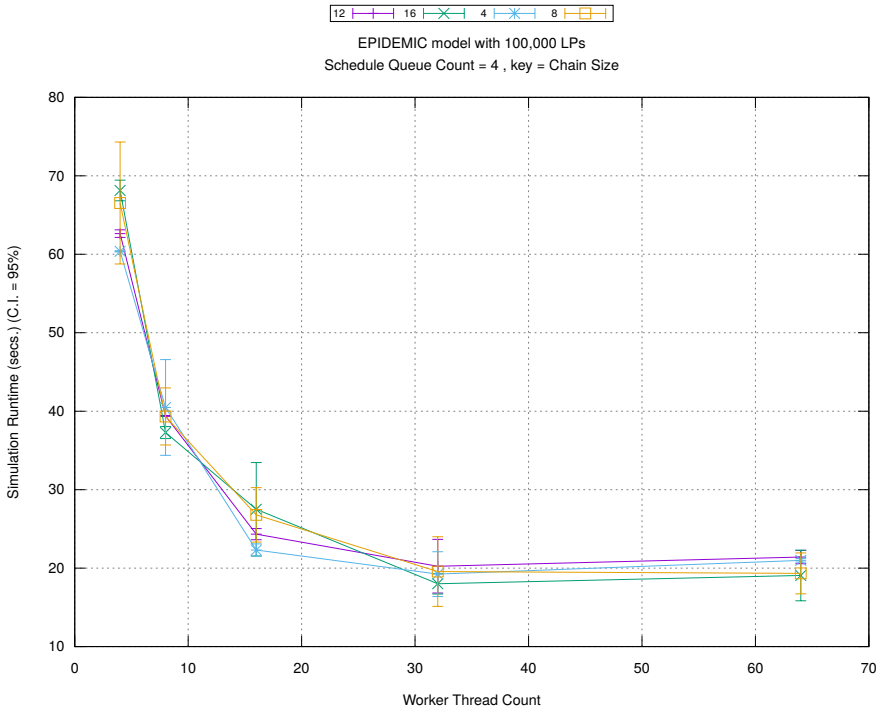


(c) Event Processing Rate (per second)

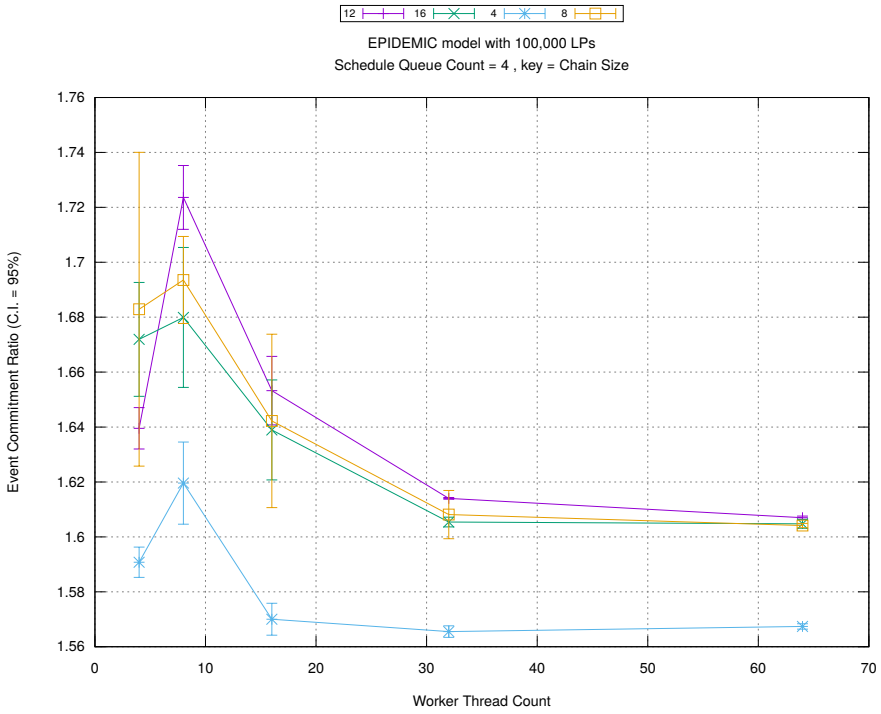
Figure A.172: epidemic 100k ba/plots/chains/threads vs chainsize key count 1



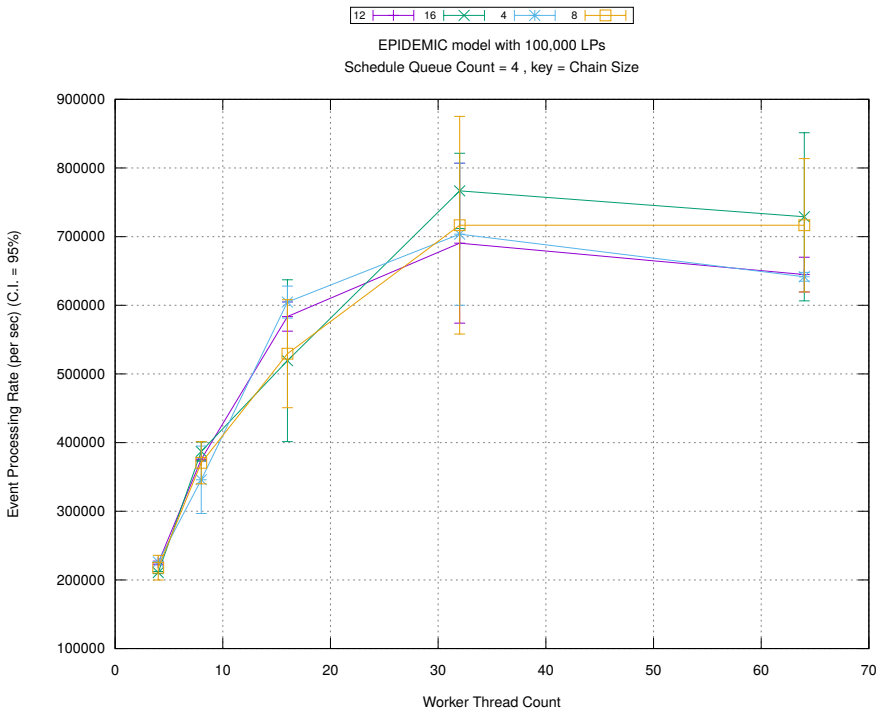
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

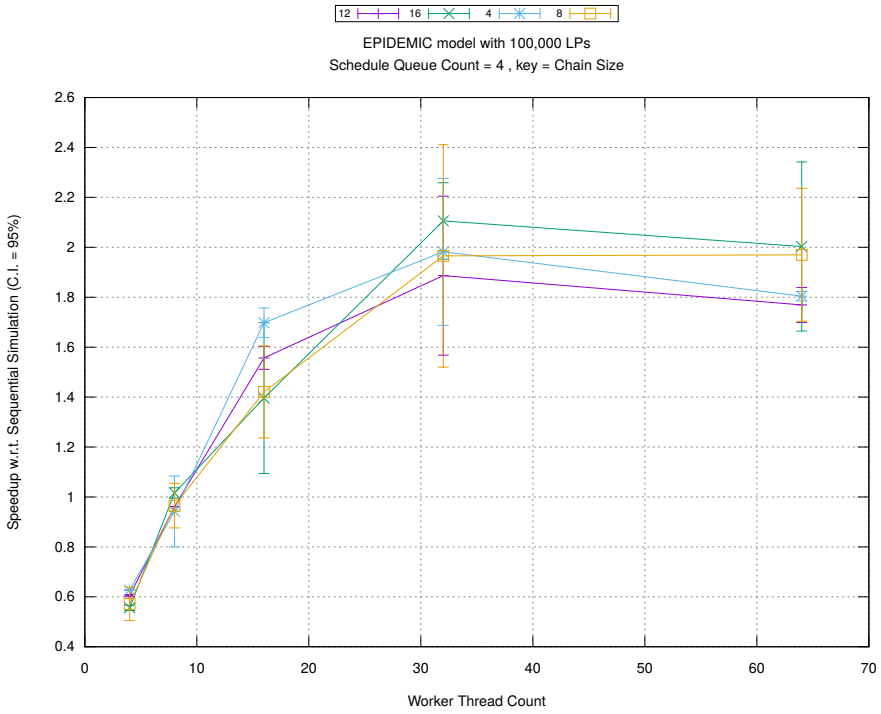


(b) Event Commitment Ratio

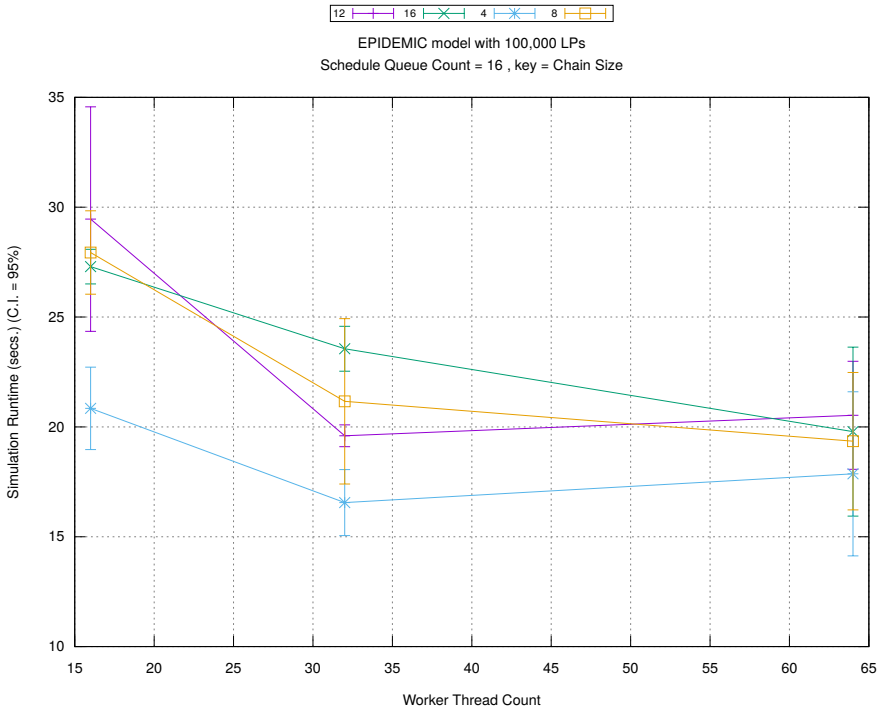


(c) Event Processing Rate (per second)

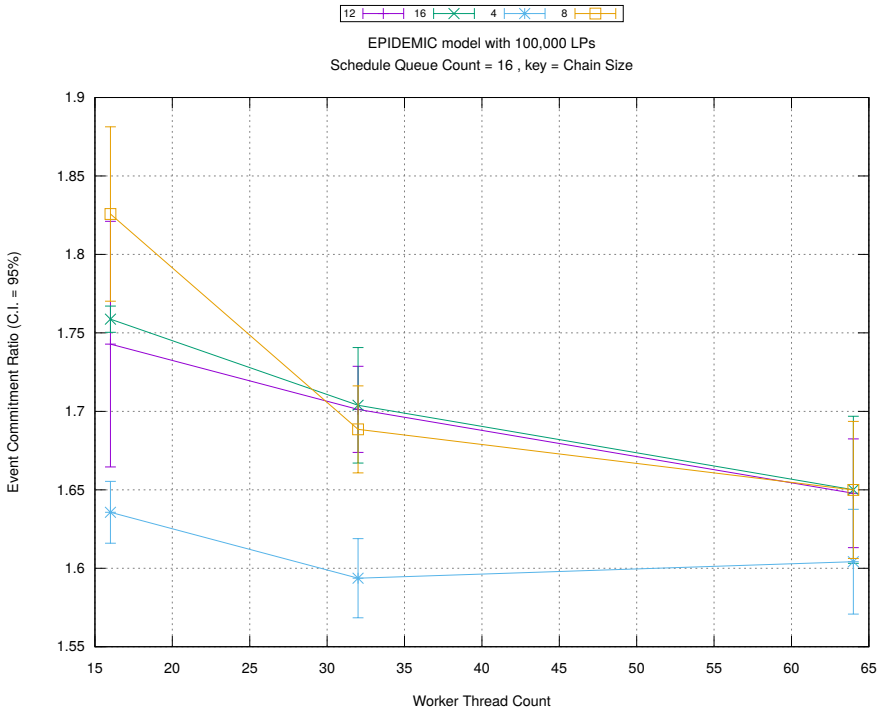
Figure A.173: epidemic 100k ba/plots/chains/threads vs chainsize key count 4



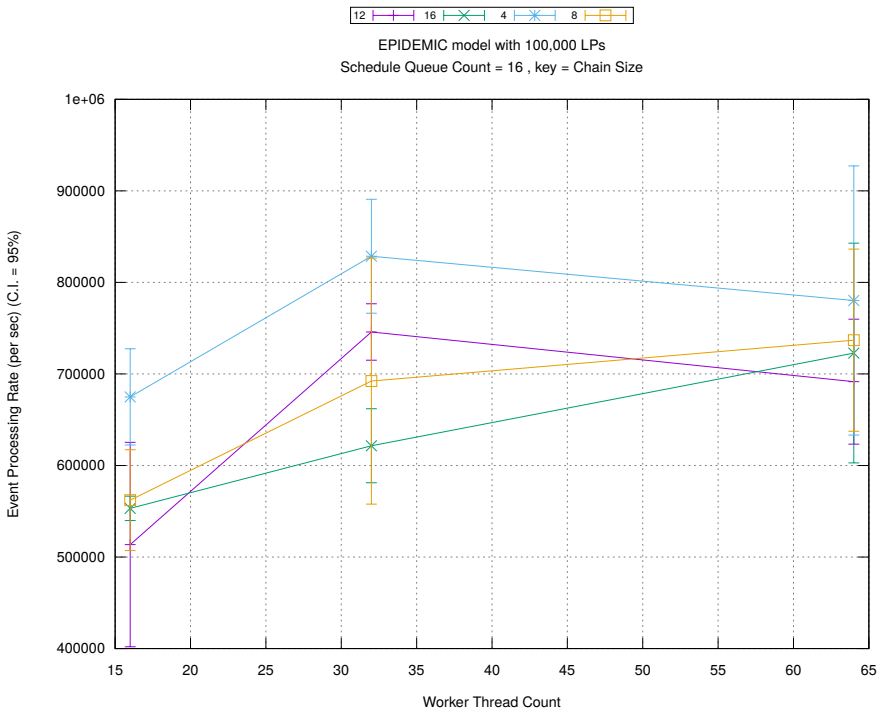
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

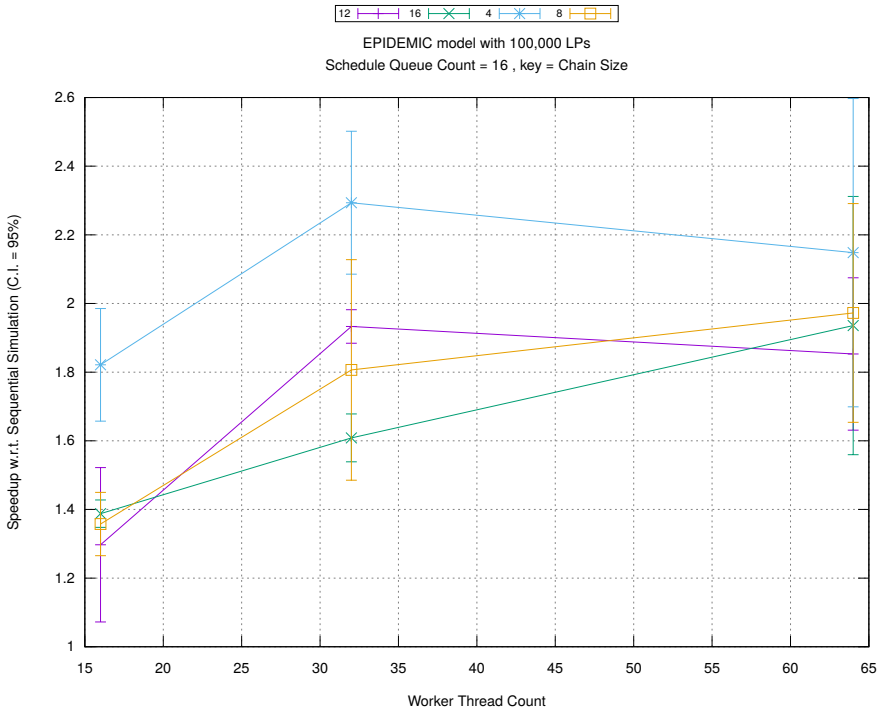


(b) Event Commitment Ratio

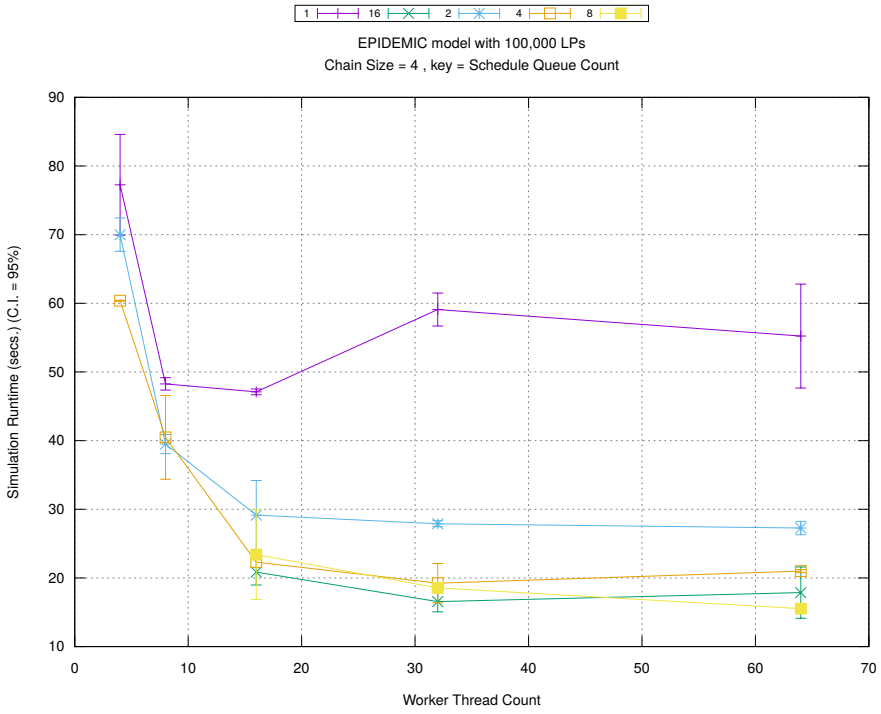


(c) Event Processing Rate (per second)

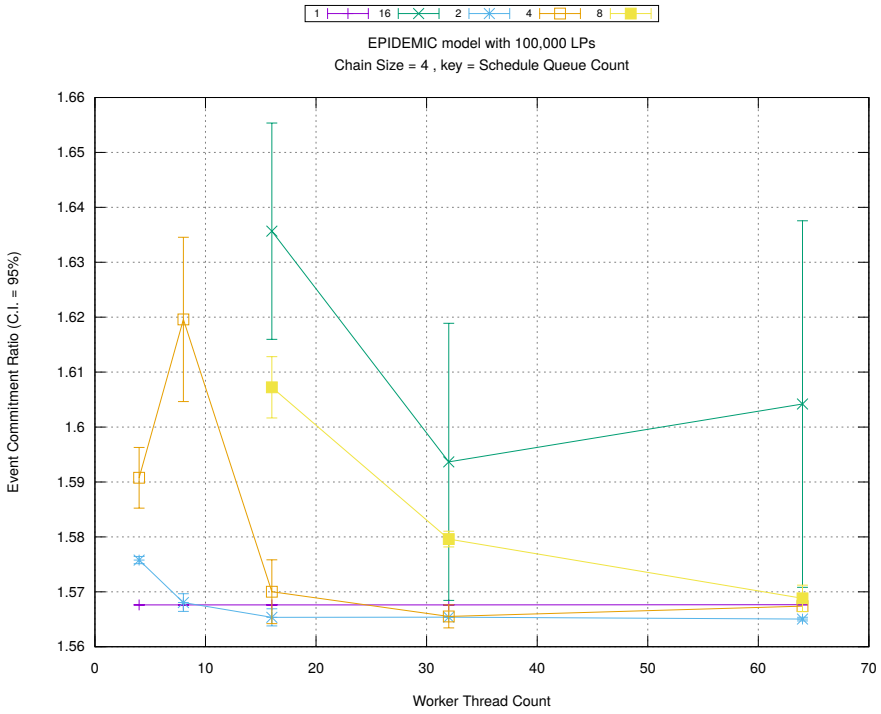
Figure A.174: epidemic 100k ba/plots/chains/threads vs chainsize key count 16



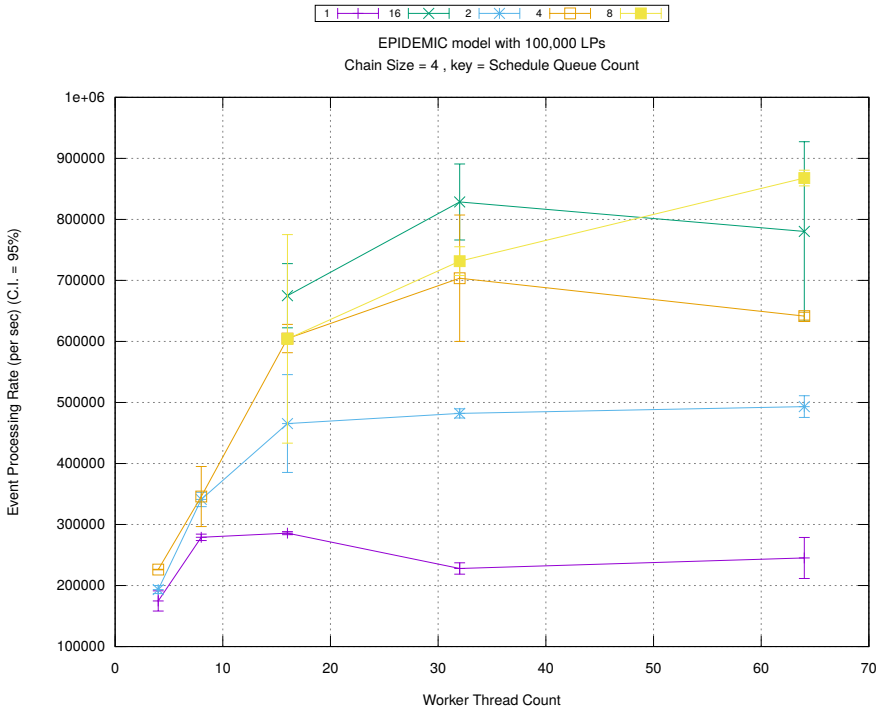
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

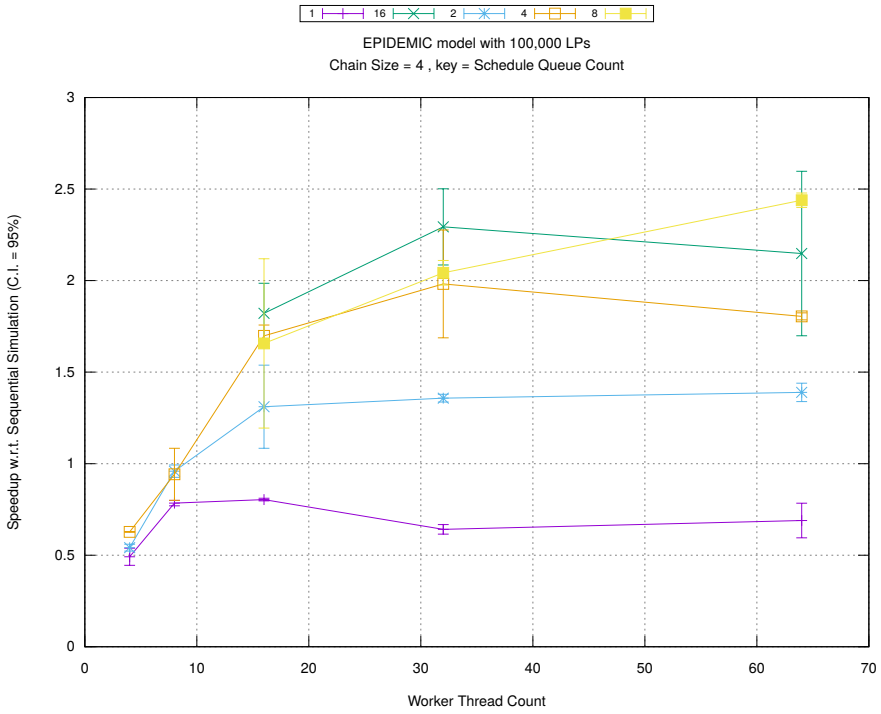


(b) Event Commitment Ratio

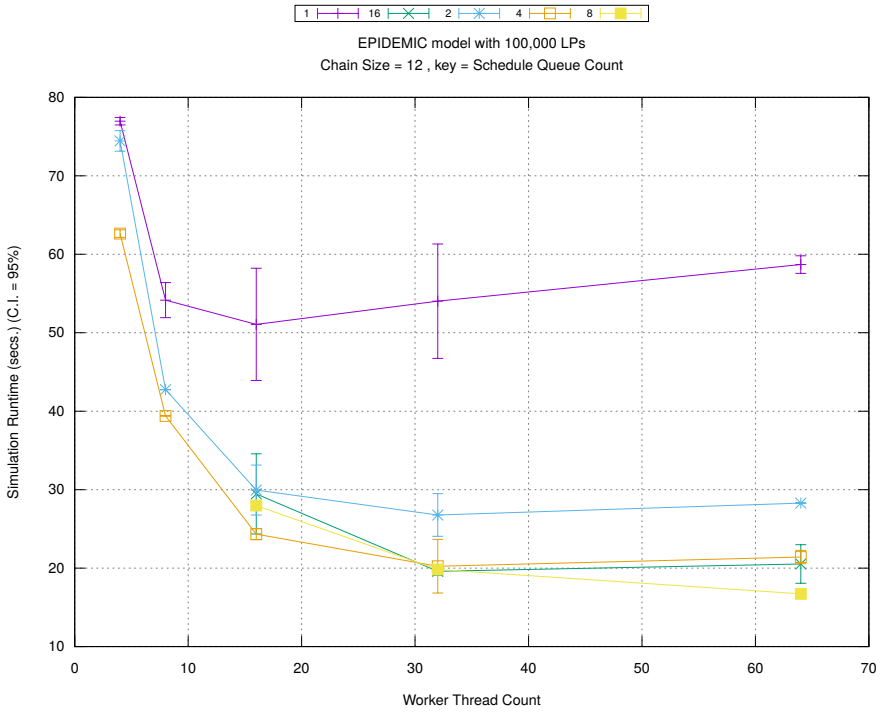


(c) Event Processing Rate (per second)

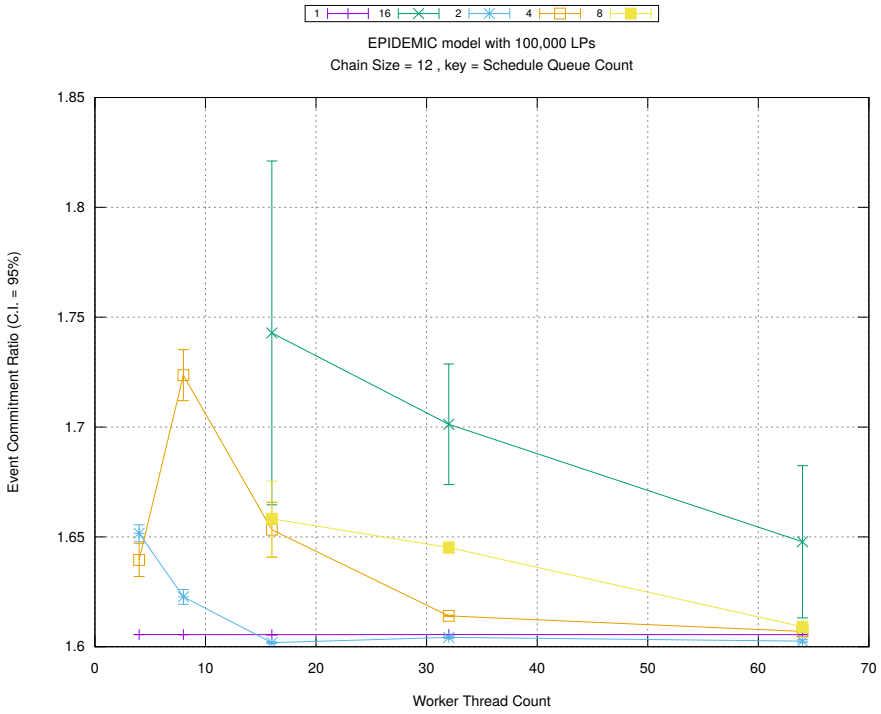
Figure A.175: epidemic 100k ba/plots/chains/threads vs count key chainsize 4



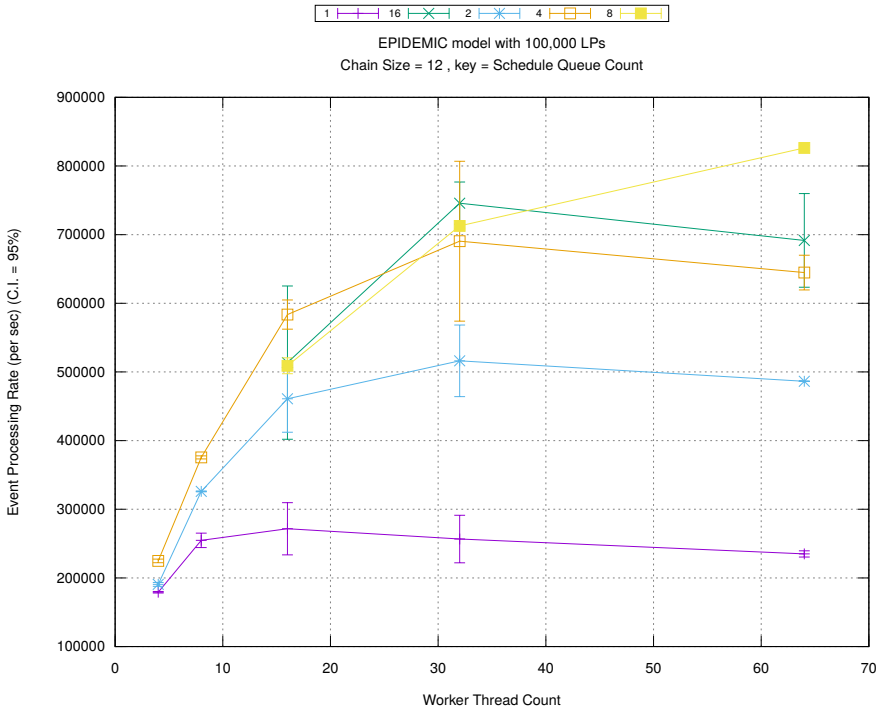
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

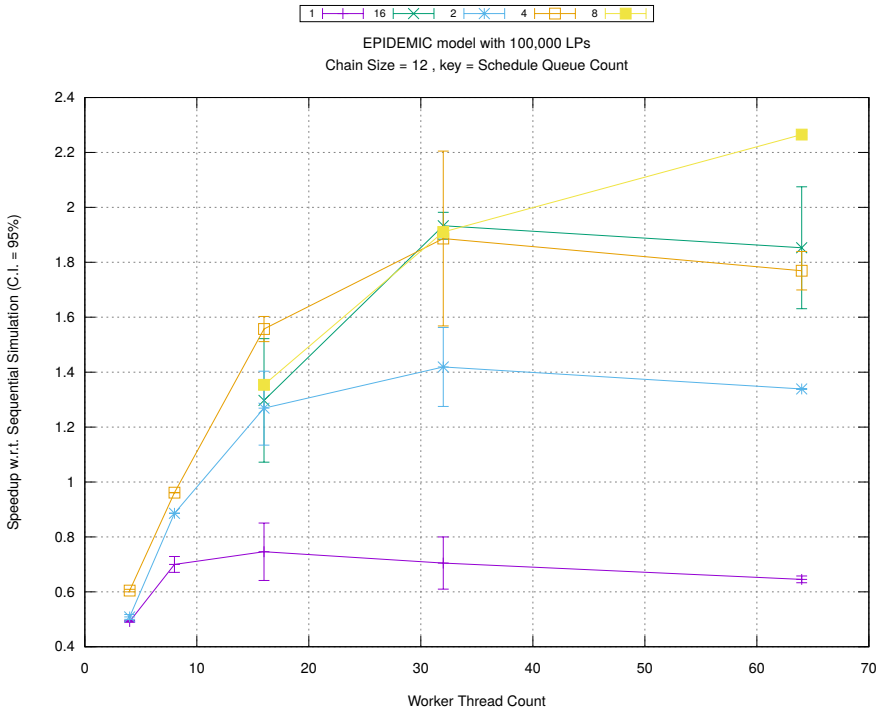


(b) Event Commitment Ratio

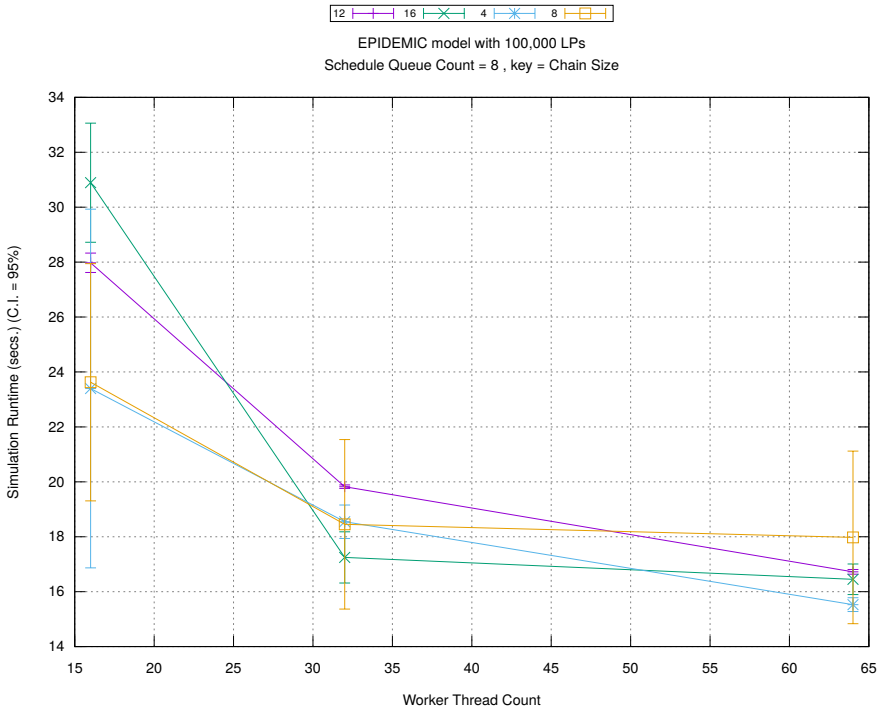


(c) Event Processing Rate (per second)

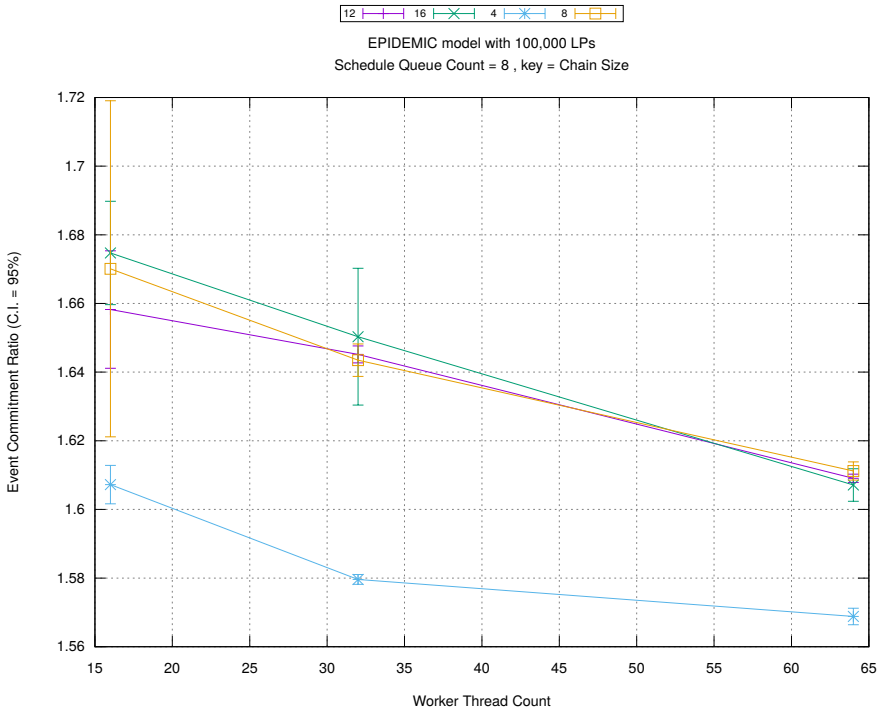
Figure A.176: epidemic 100k ba/plots/chains/threads vs count key chainsize 12



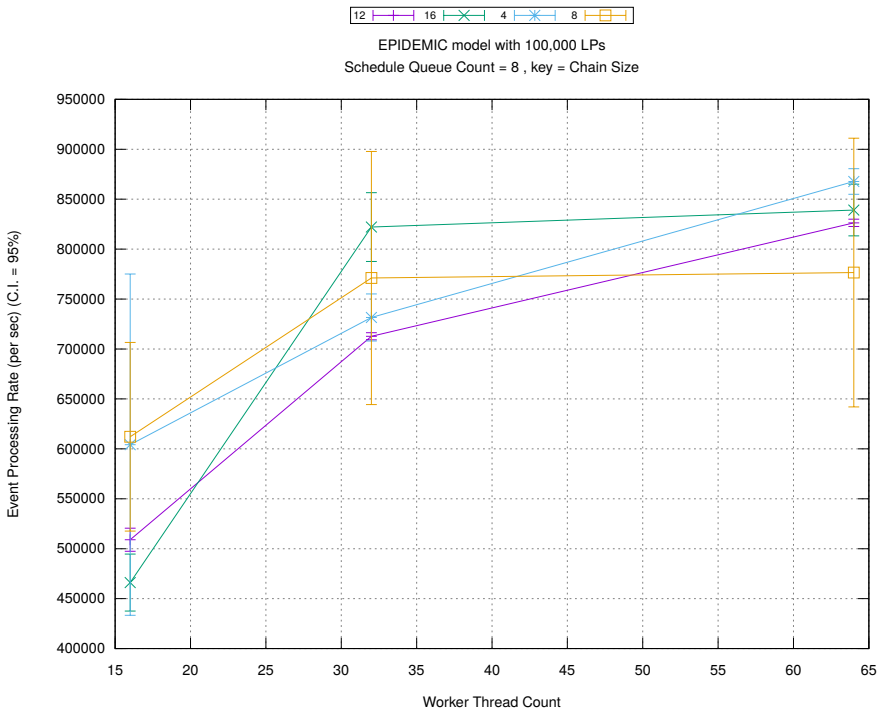
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

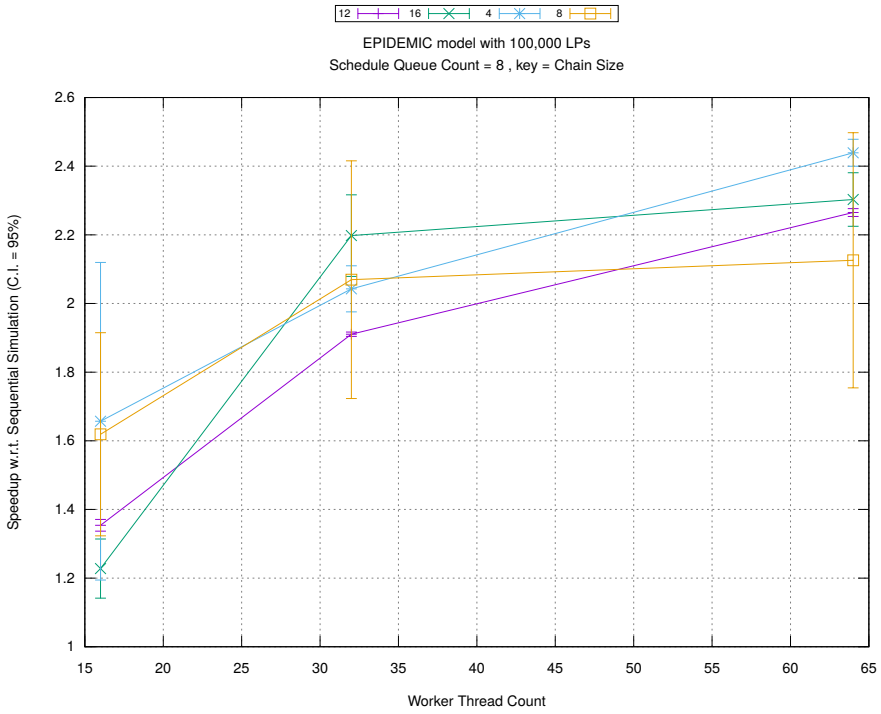


(b) Event Commitment Ratio

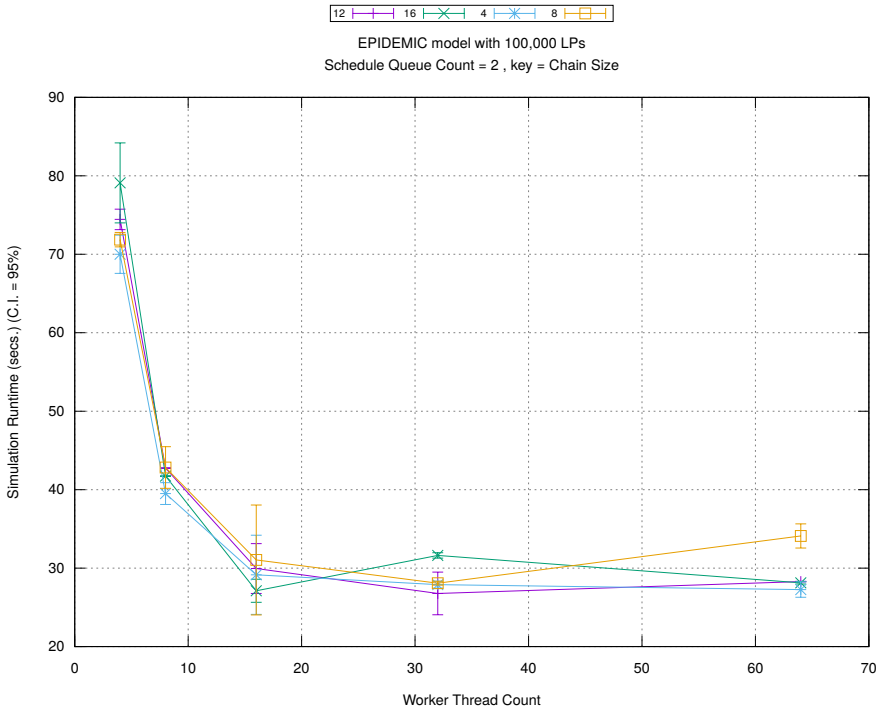


(c) Event Processing Rate (per second)

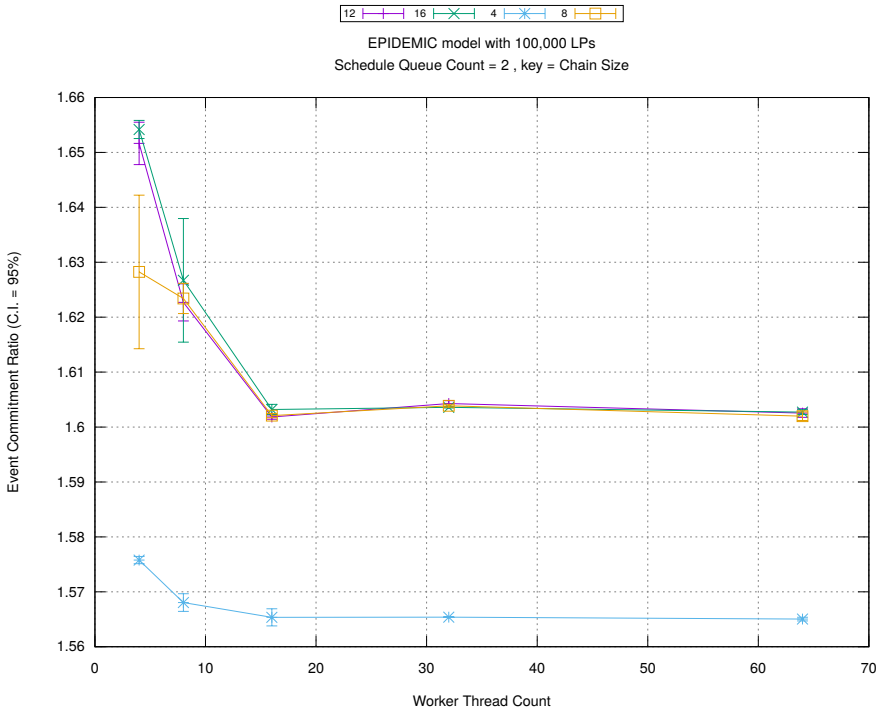
Figure A.177: epidemic 100k ba/plots/chains/threads vs chainsize key count 8



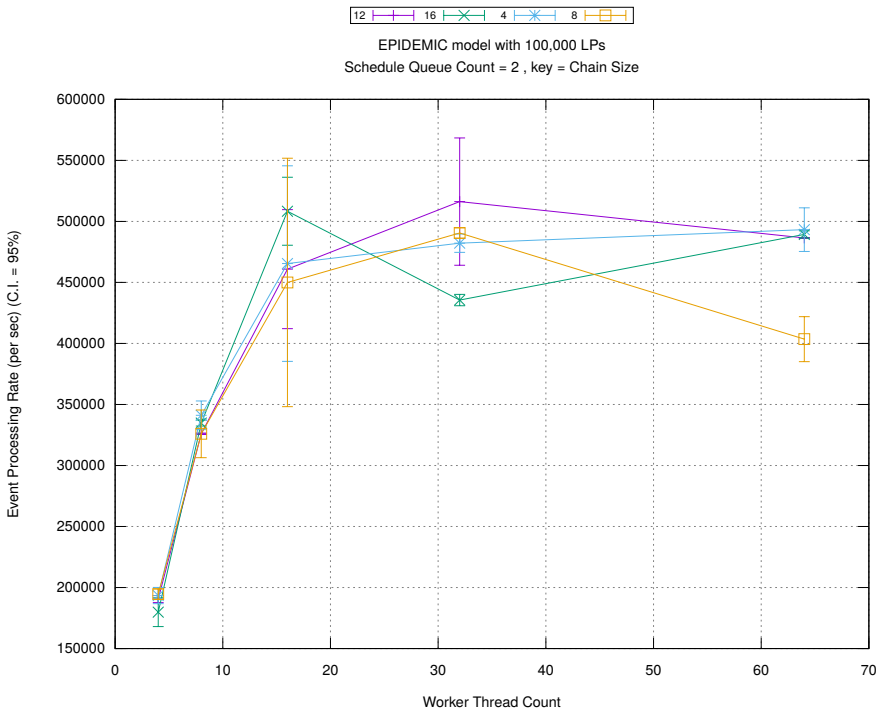
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

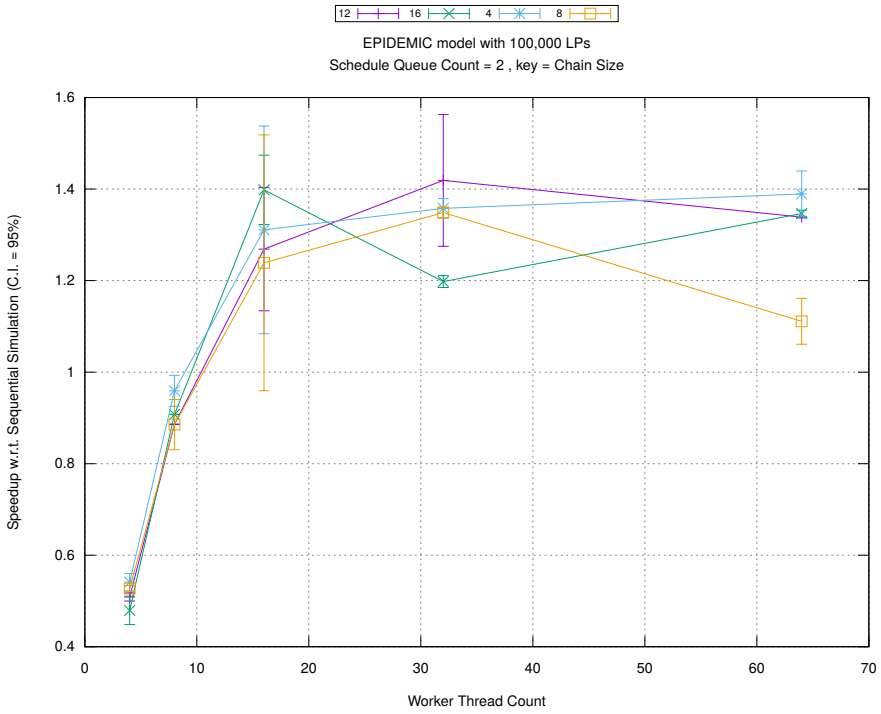


(b) Event Commitment Ratio

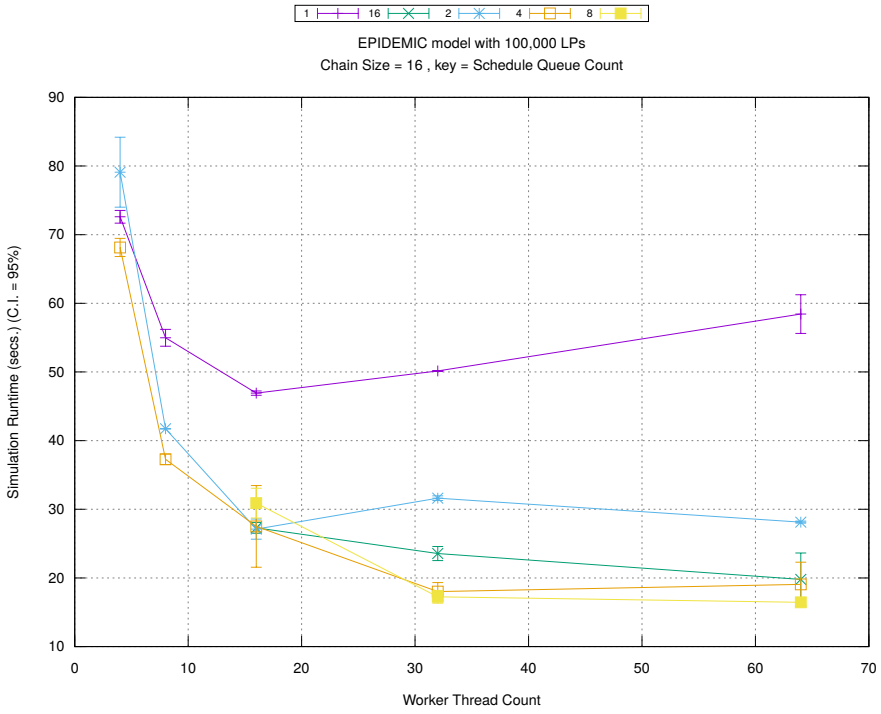


(c) Event Processing Rate (per second)

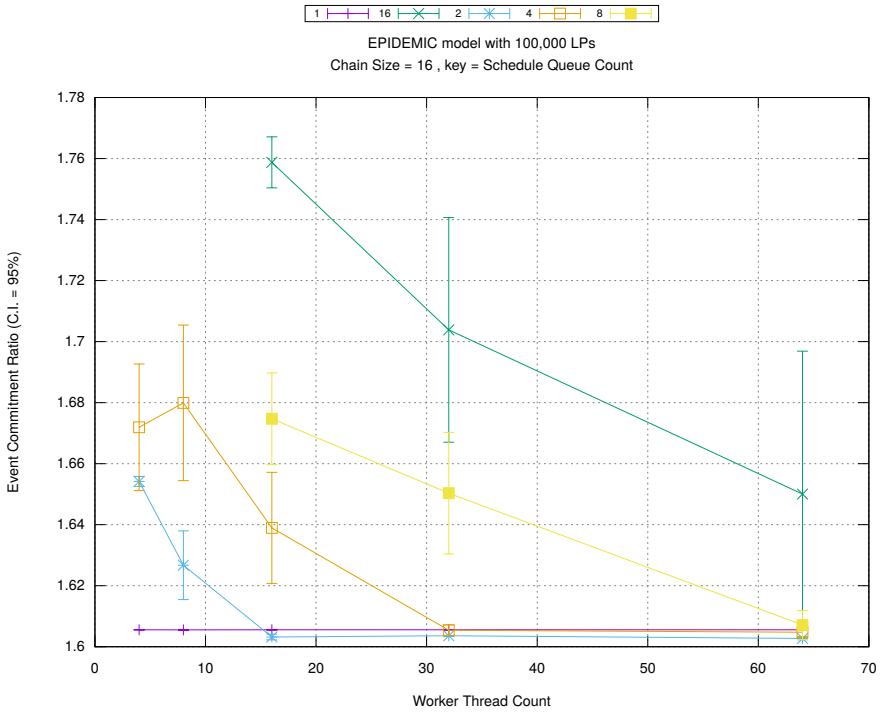
Figure A.178: epidemic 100k ba/plots/chains/threads vs chainsize key count 2



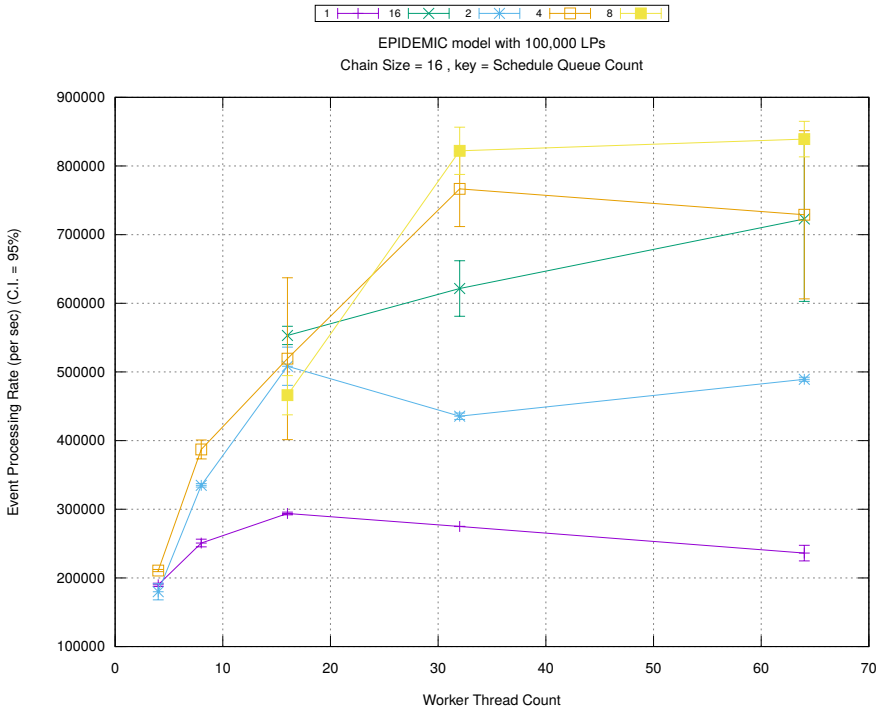
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

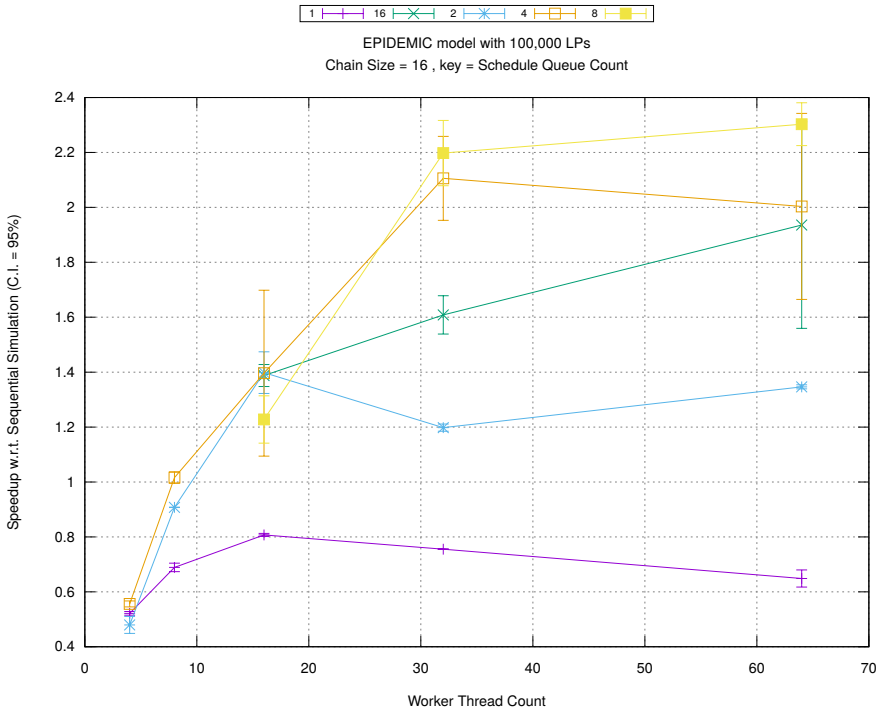


(b) Event Commitment Ratio

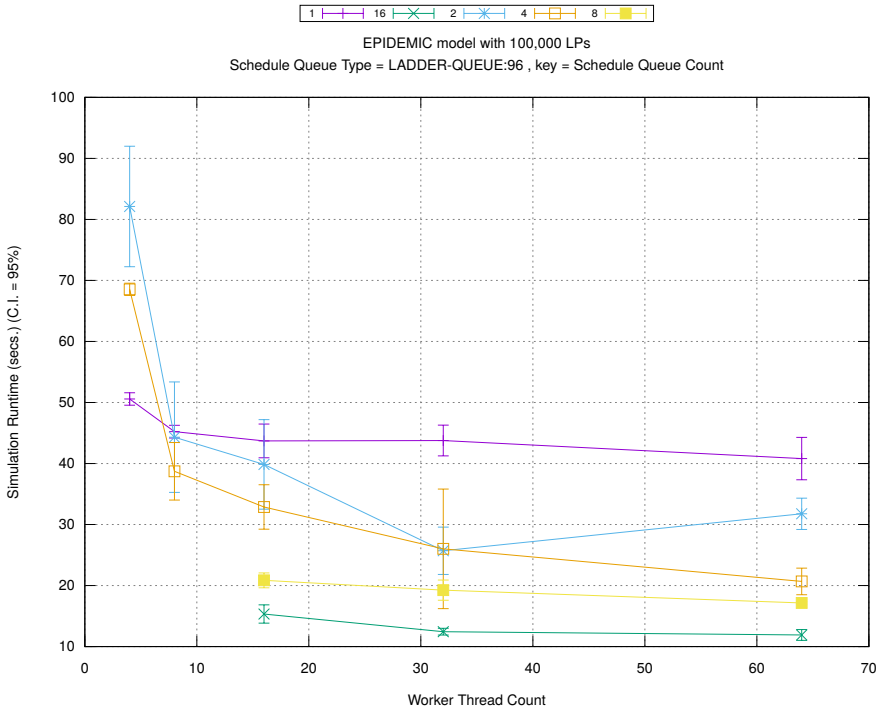


(c) Event Processing Rate (per second)

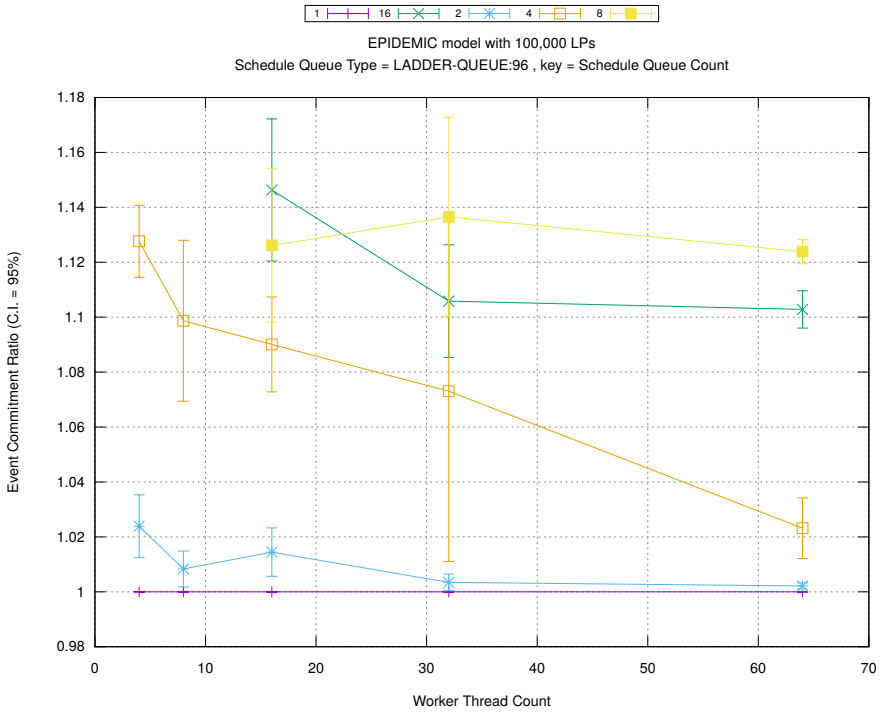
Figure A.179: epidemic 100k ba/plots/chains/threads vs count key chainsize 16



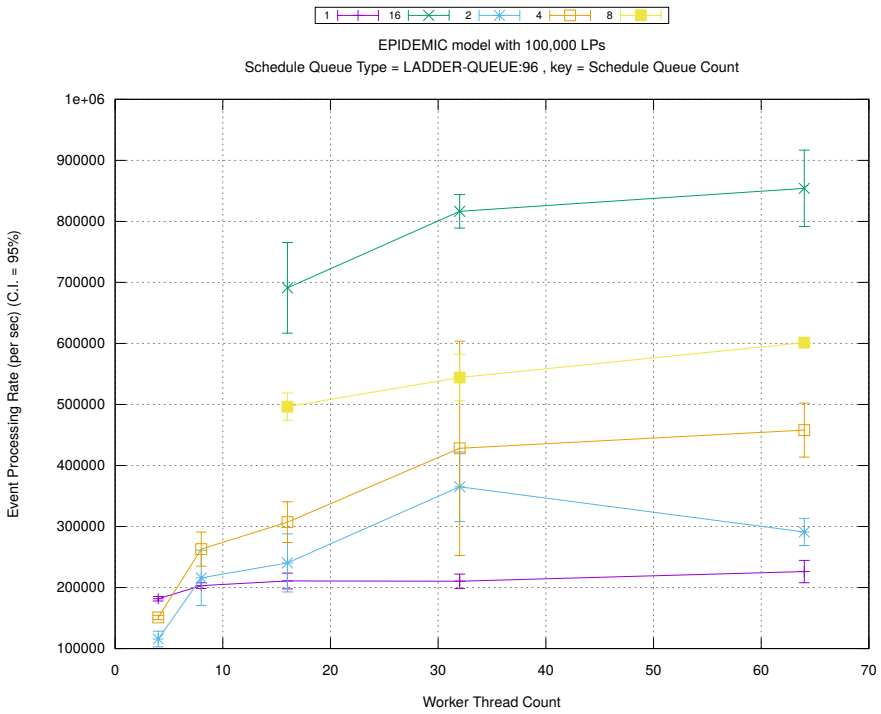
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

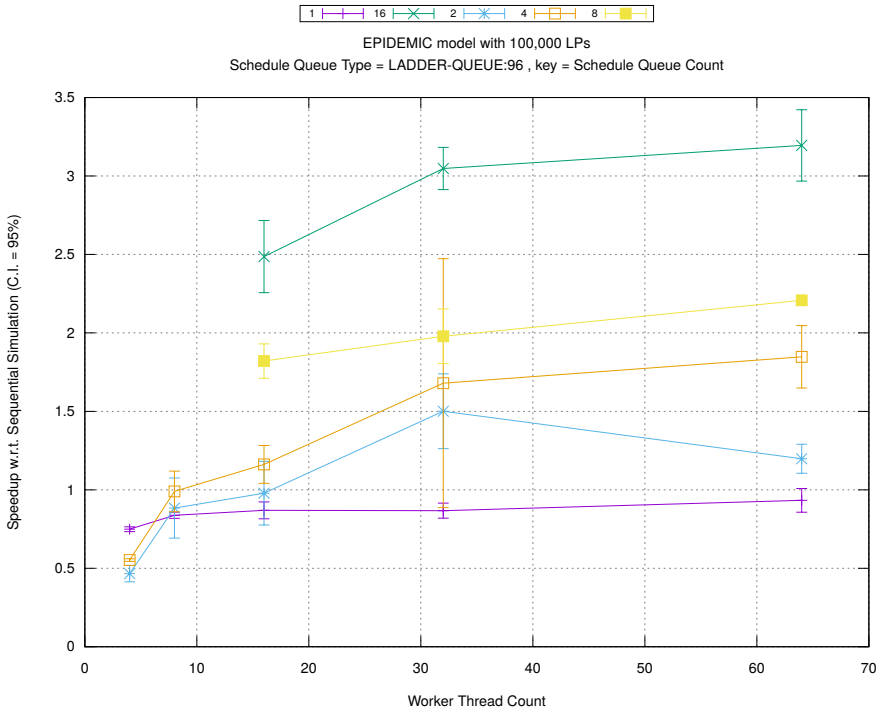


(b) Event Commitment Ratio

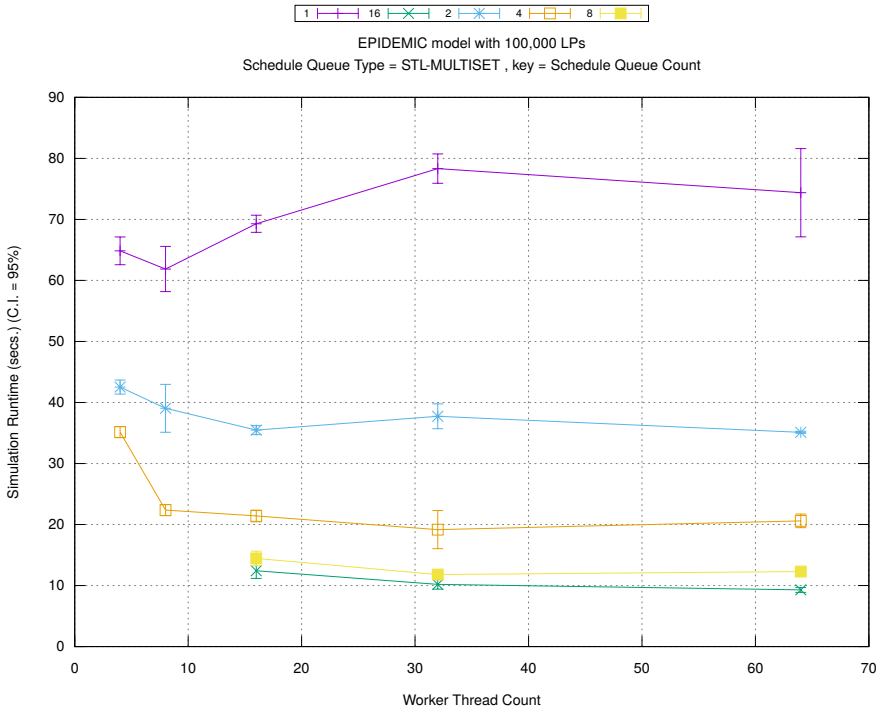


(c) Event Processing Rate (per second)

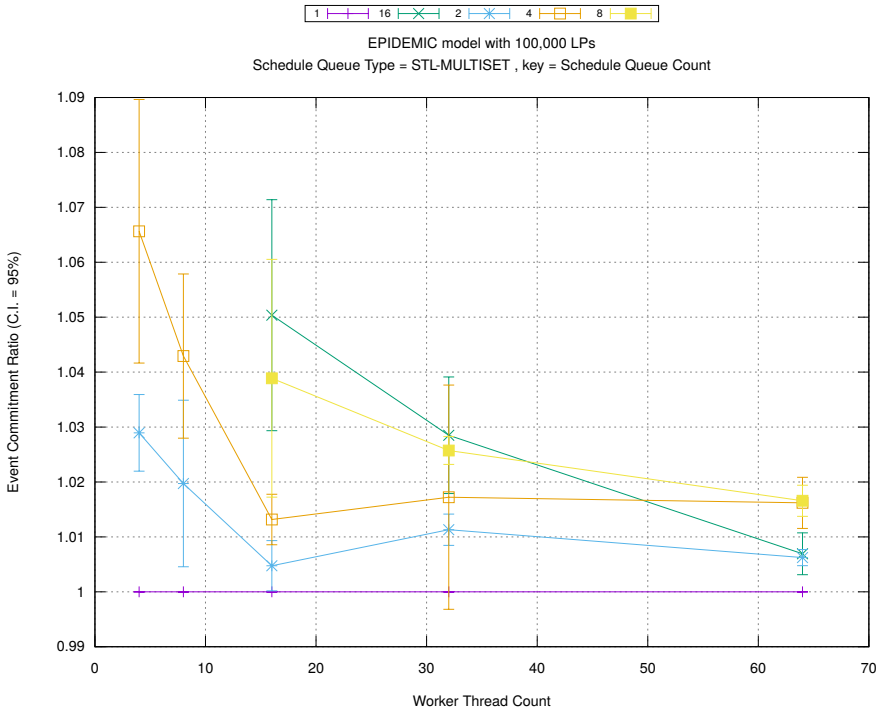
Figure A.180: epidemic 100k ba/plots/scheduleq/threads vs count key type ladder-queue 96



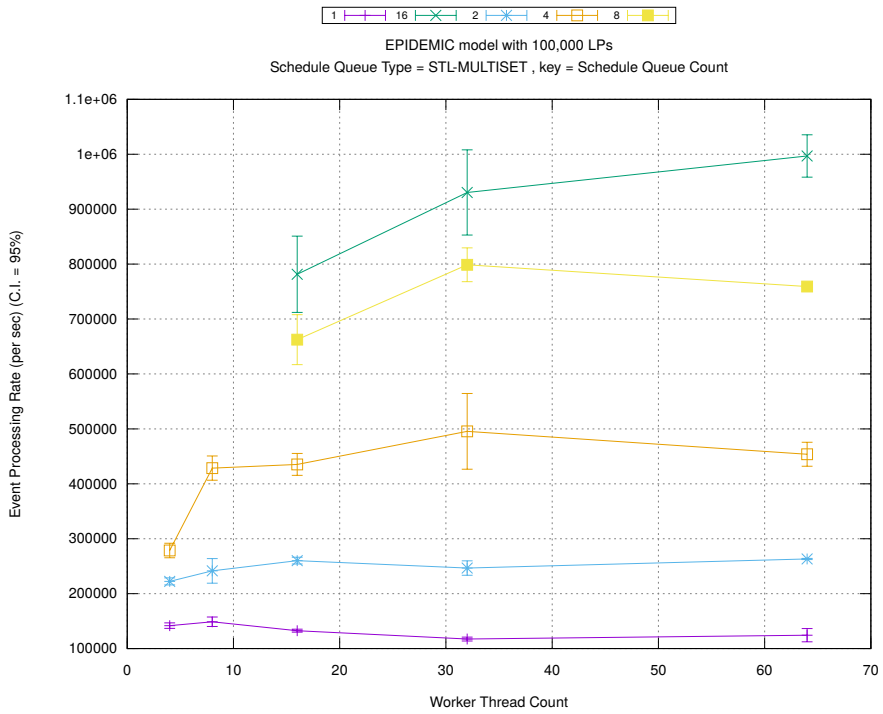
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

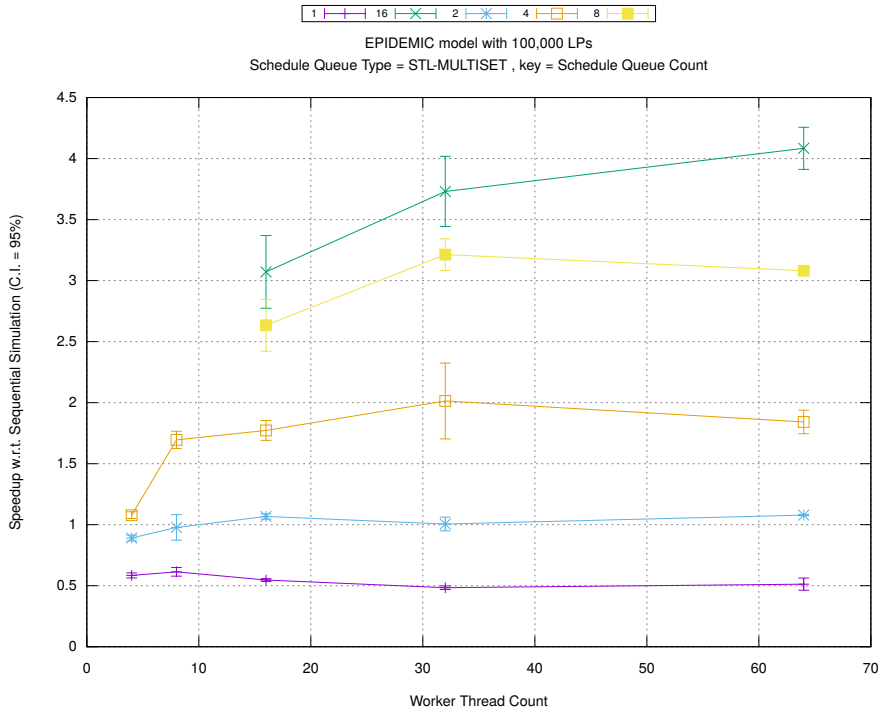


(b) Event Commitment Ratio

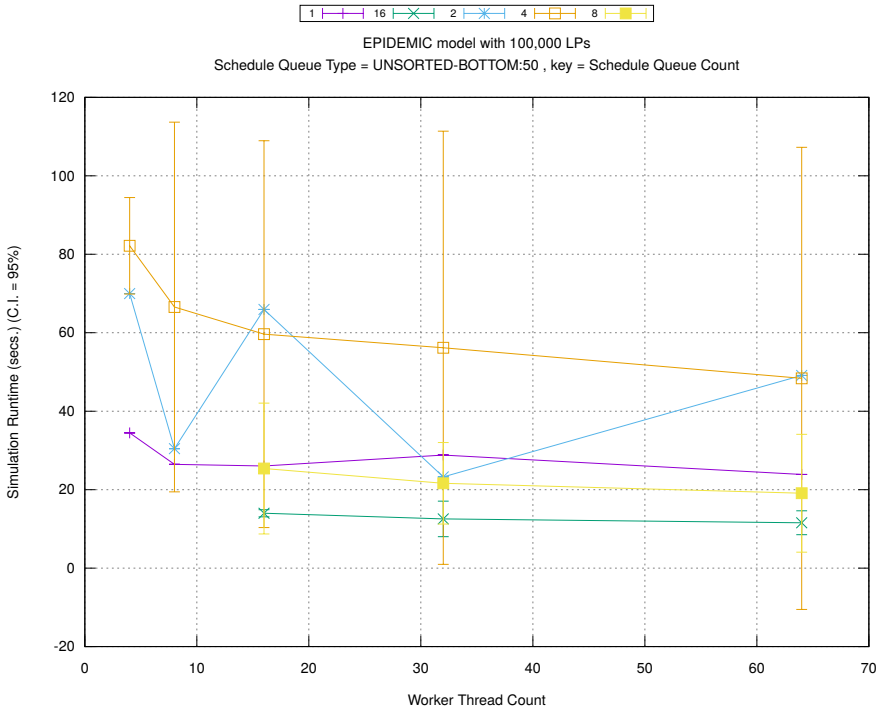


(c) Event Processing Rate (per second)

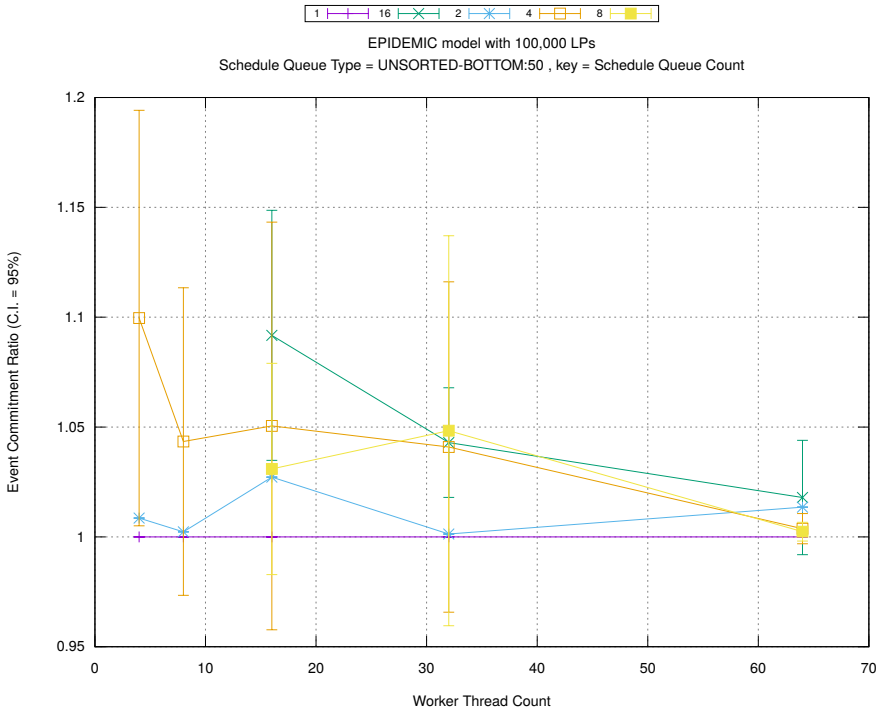
Figure A.181: epidemic 100k ba/plots/scheduleq/threads vs count key type stl-multiset



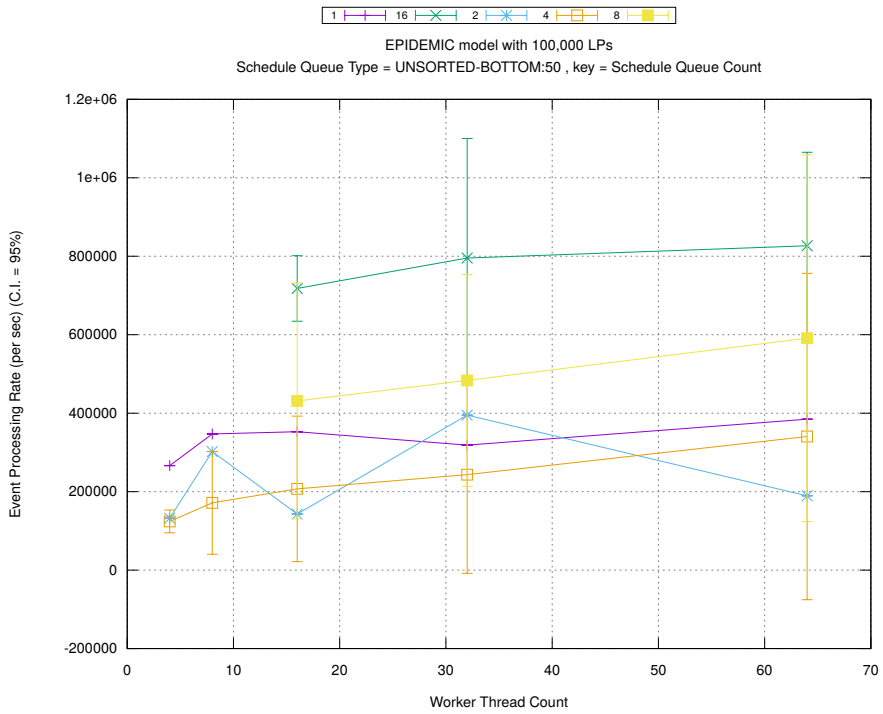
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

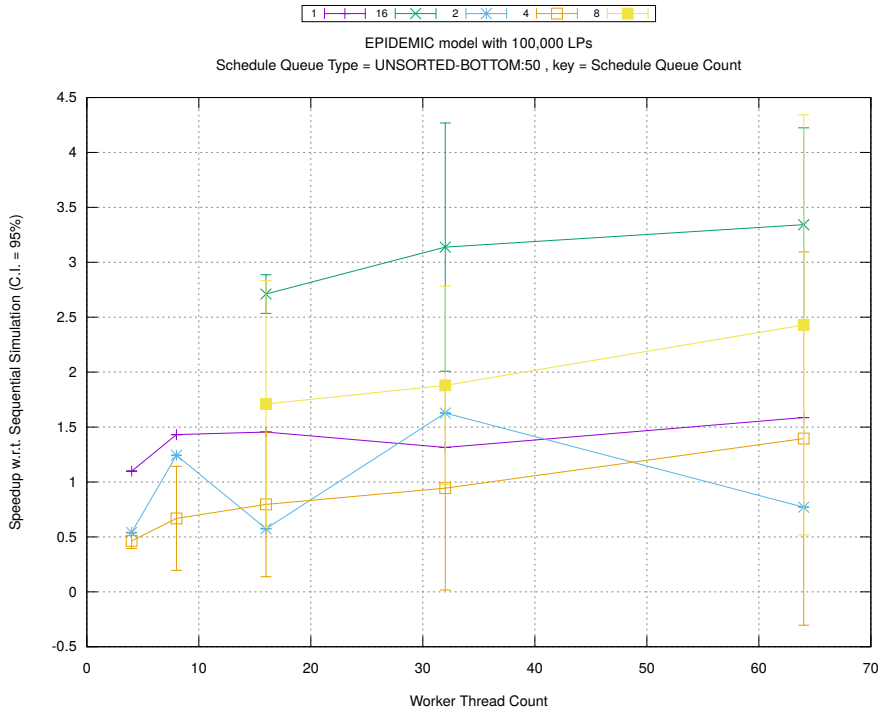


(b) Event Commitment Ratio

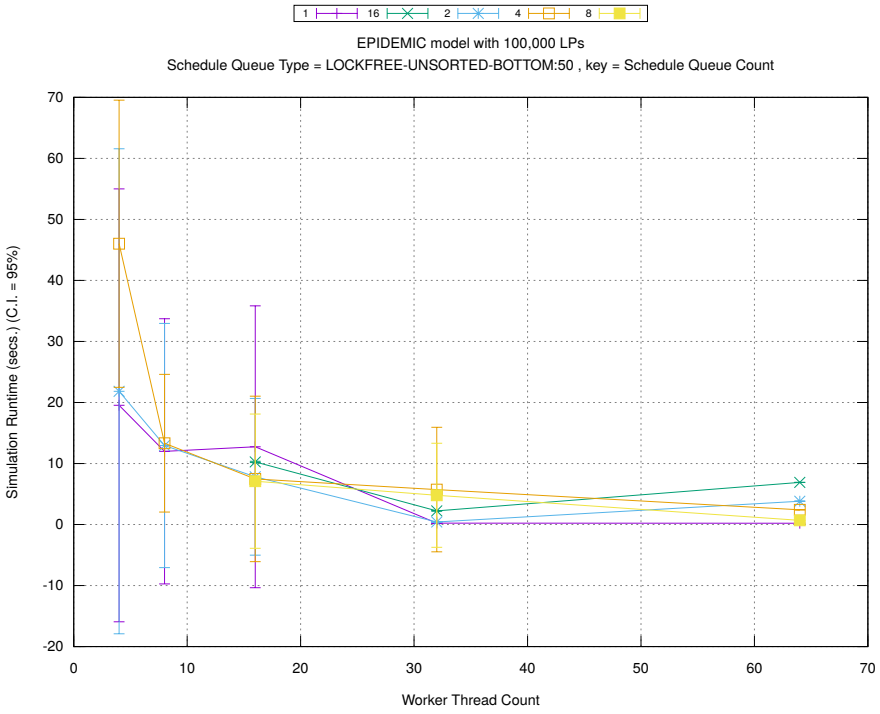


(c) Event Processing Rate (per second)

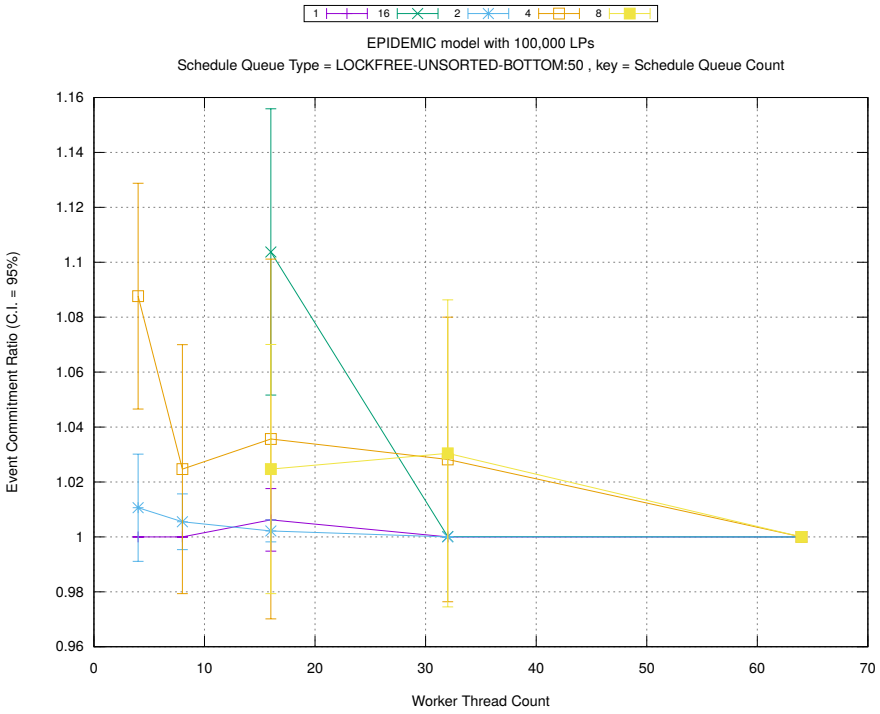
Figure A.182: epidemic 100k ba/plots/scheduleq/threads vs count key type unsorted-bottom 50



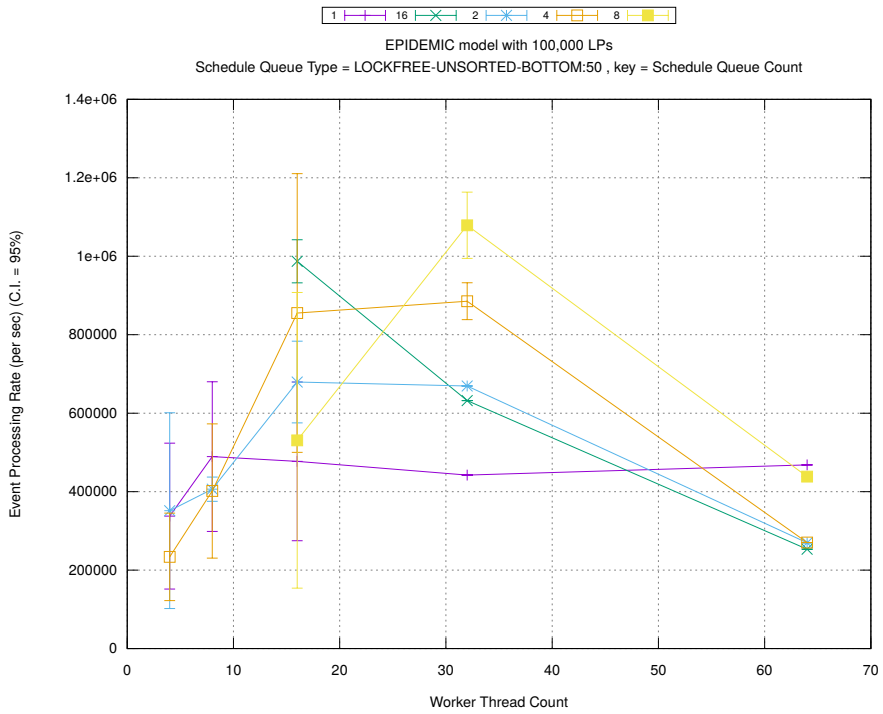
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

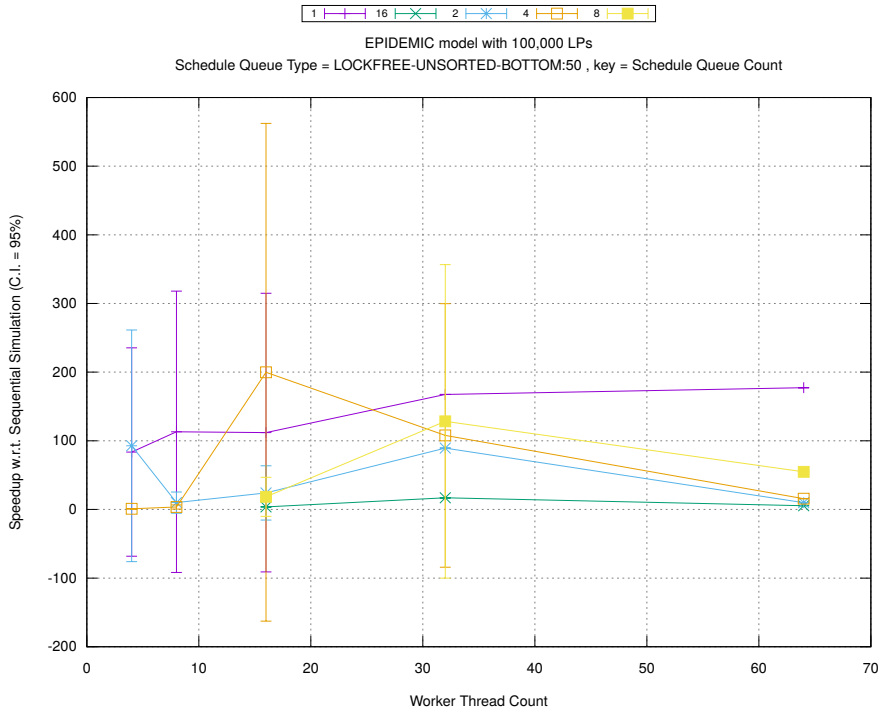


(b) Event Commitment Ratio

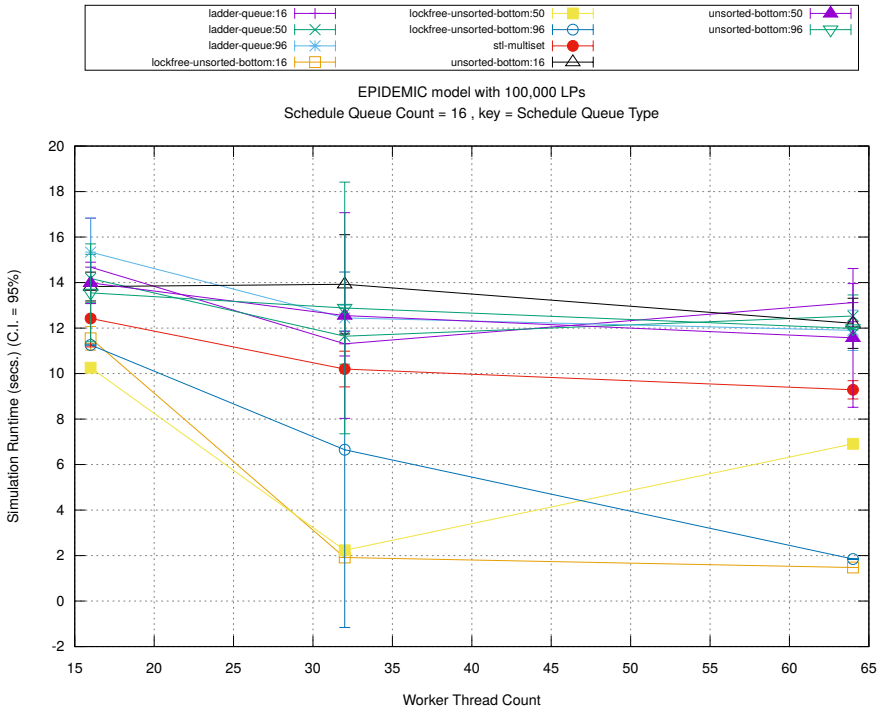


(c) Event Processing Rate (per second)

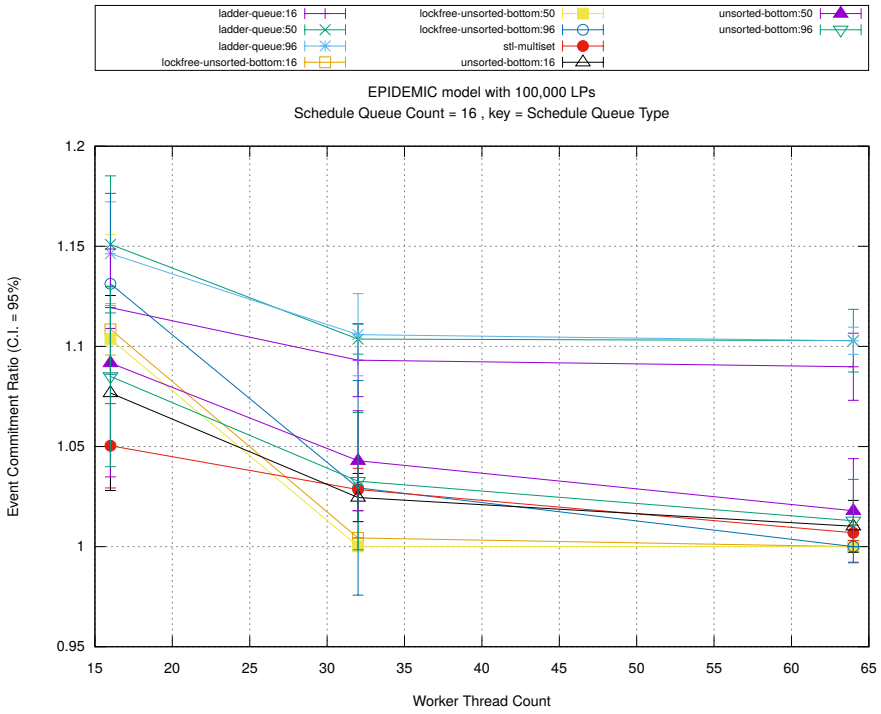
Figure A.183: epidemic 100k ba/plots/scheduleq/threads vs count key type lockfree-unsorted-bottom 50



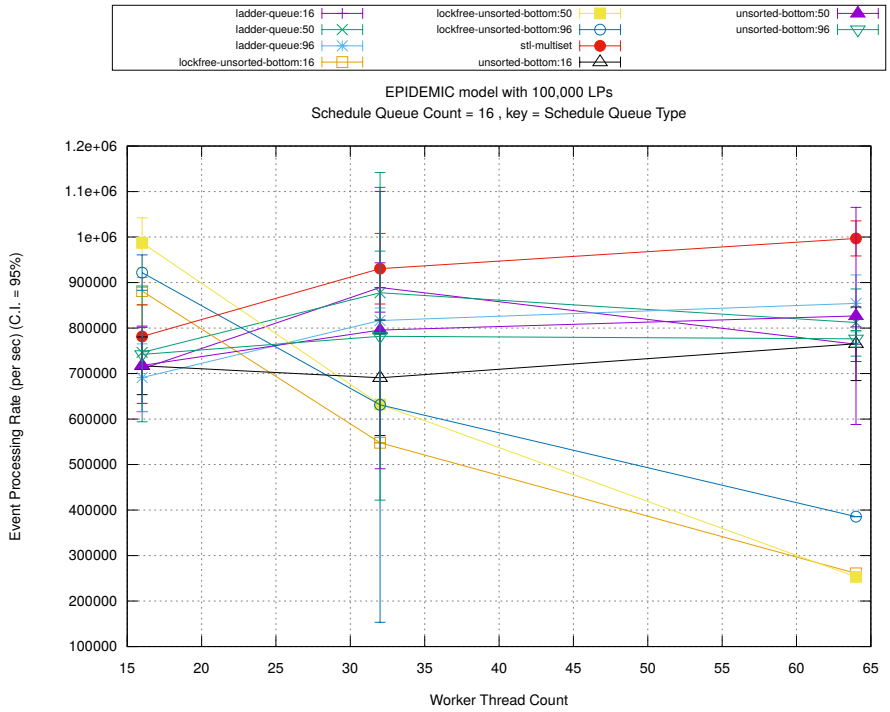
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

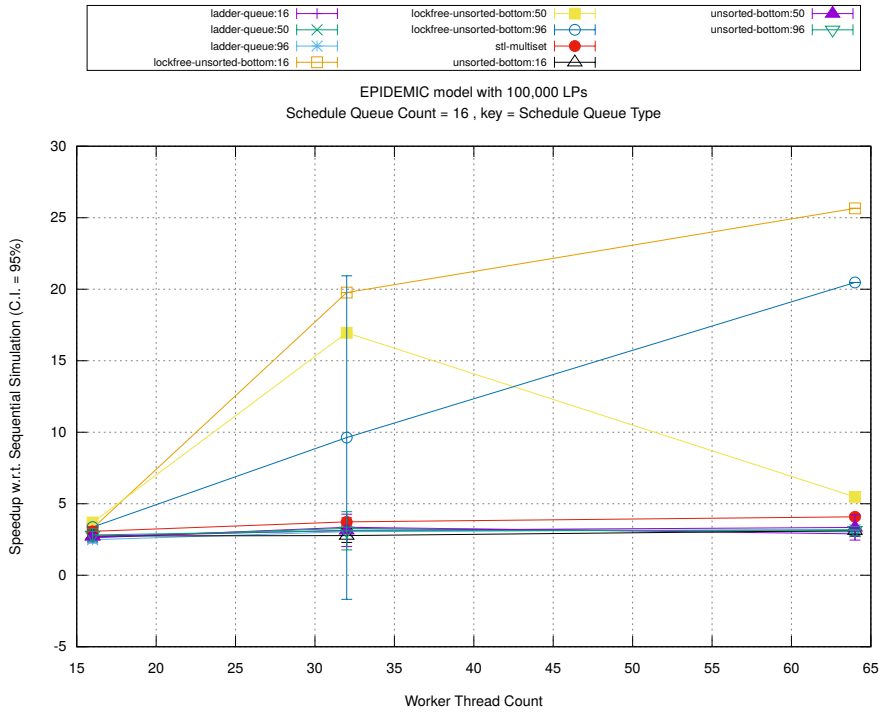


(b) Event Commitment Ratio

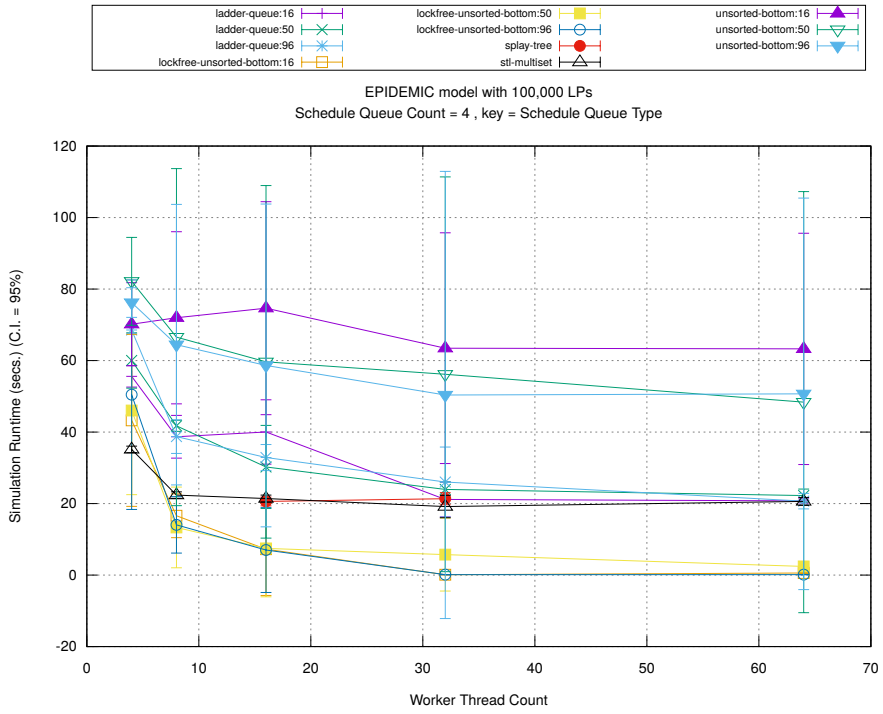


(c) Event Processing Rate (per second)

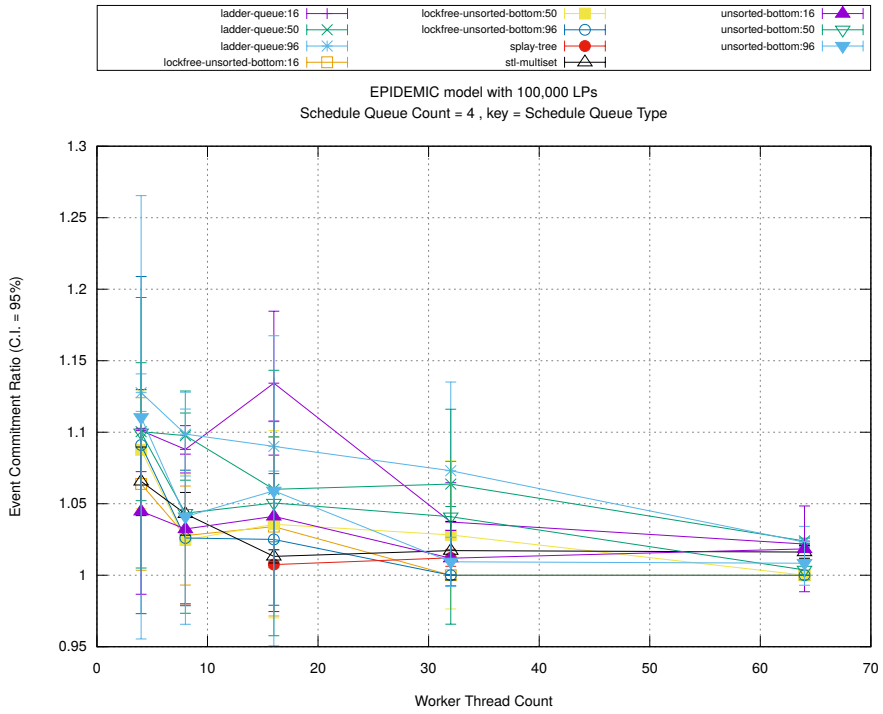
Figure A.184: epidemic 100k ba/plots/scheduleq/threads vs type key count 16



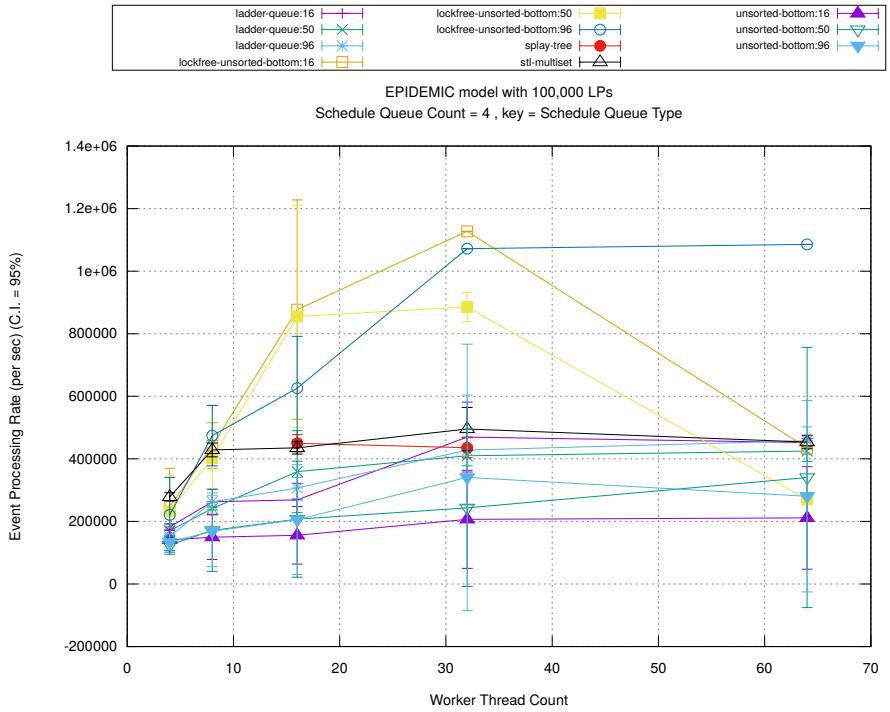
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

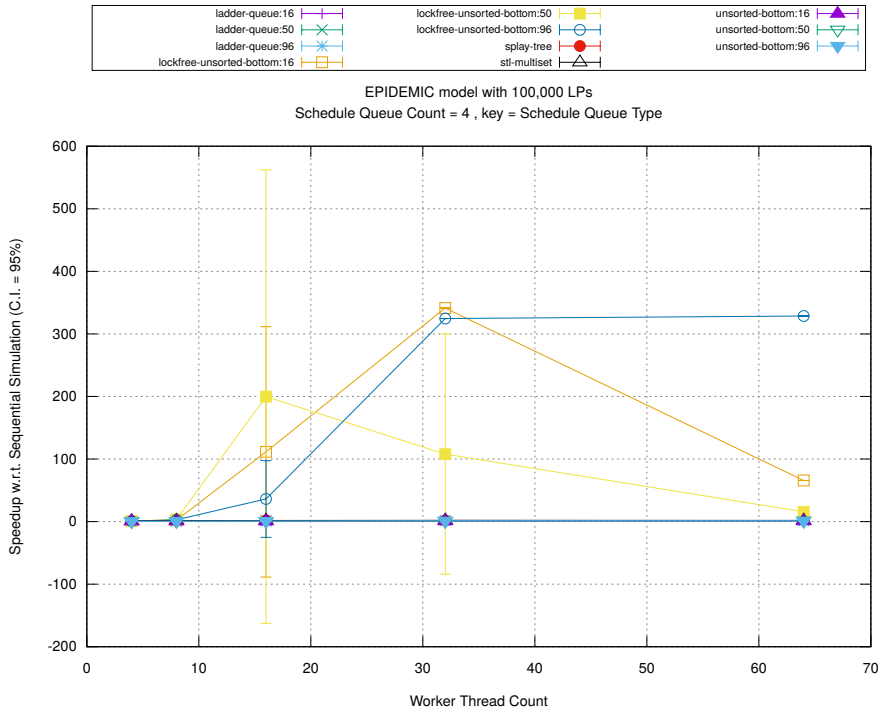


(b) Event Commitment Ratio

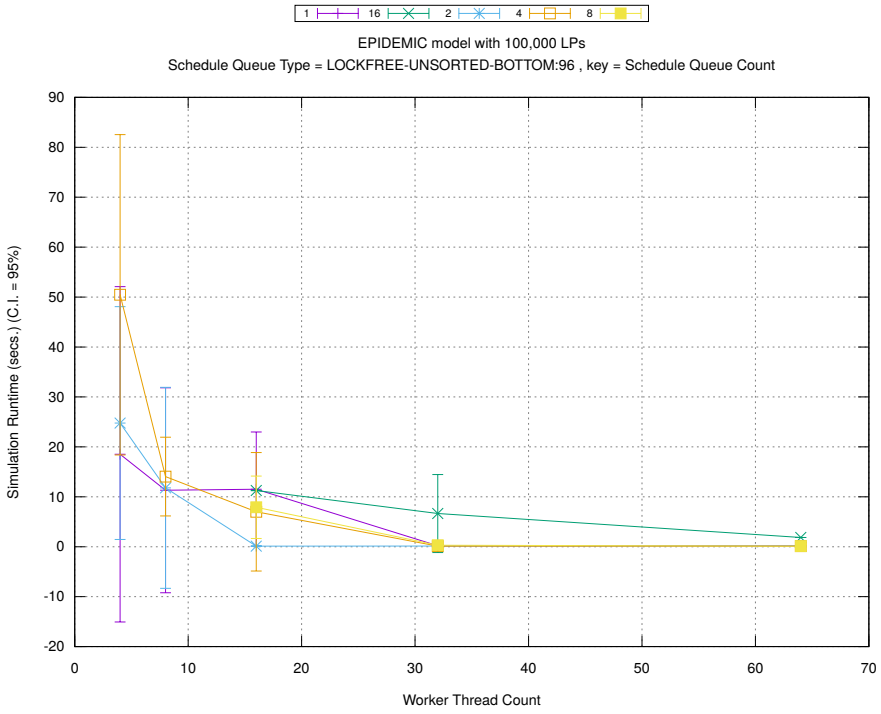


(c) Event Processing Rate (per second)

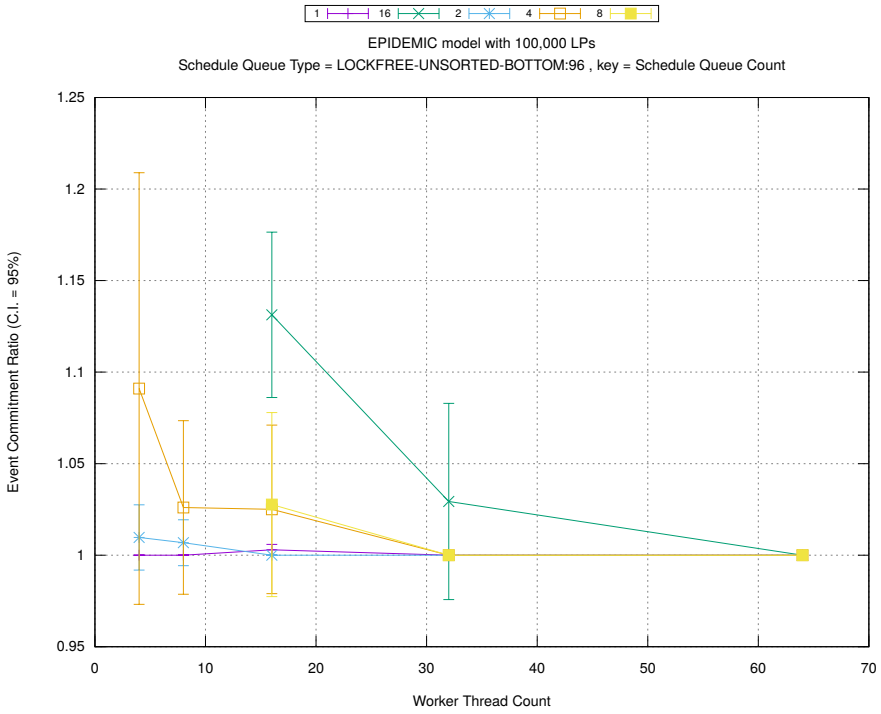
Figure A.185: epidemic 100k ba/plots/scheduleq/threads vs type key count 4



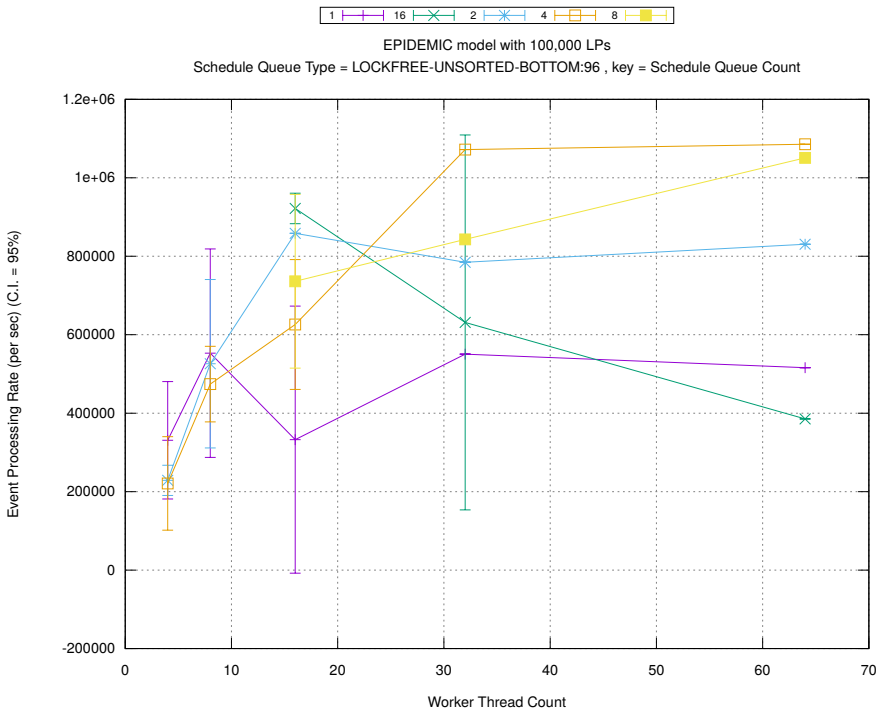
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

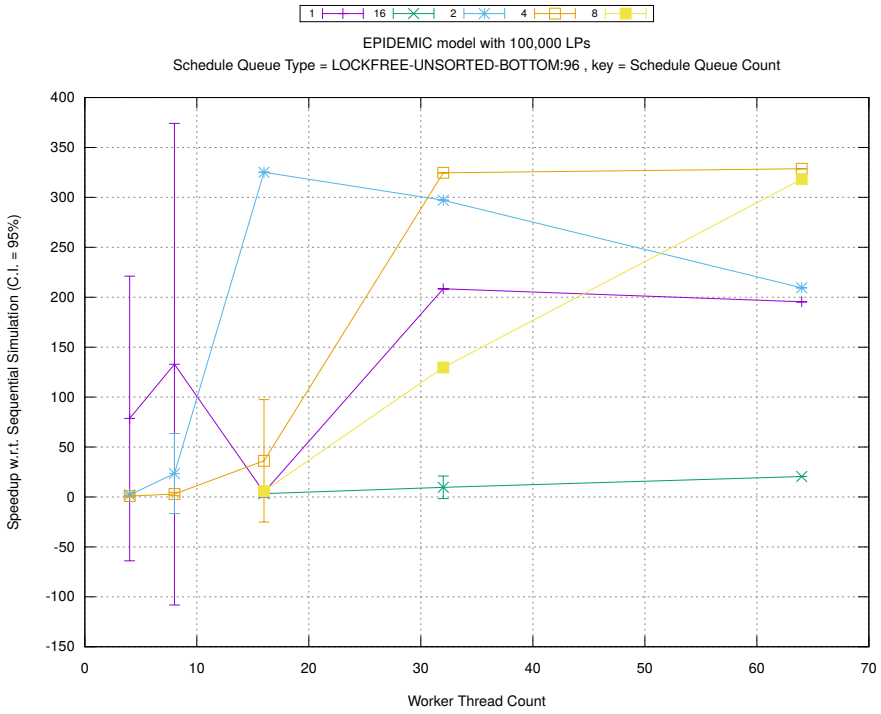


(b) Event Commitment Ratio

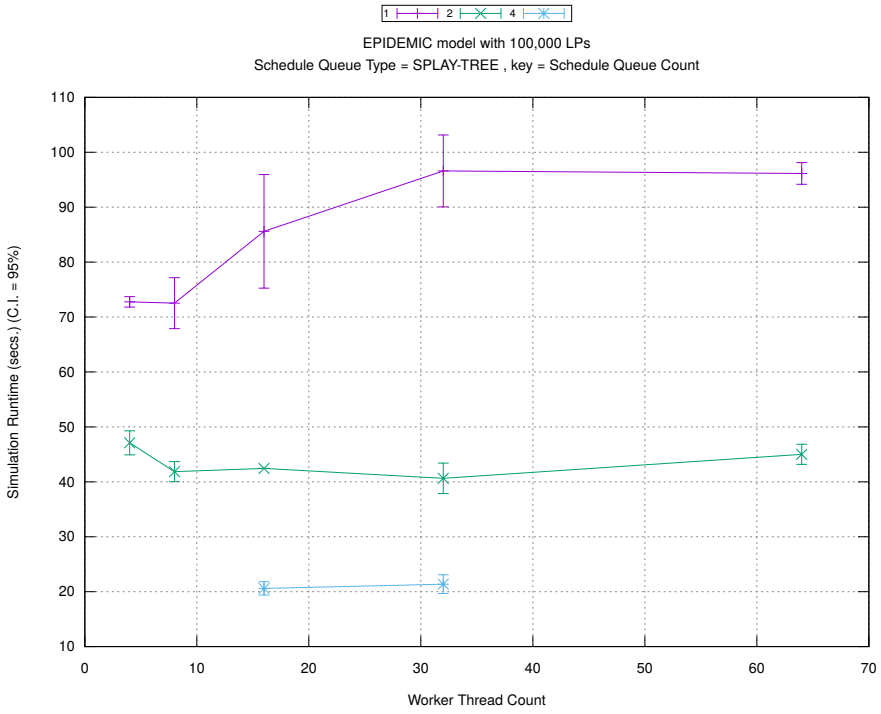


(c) Event Processing Rate (per second)

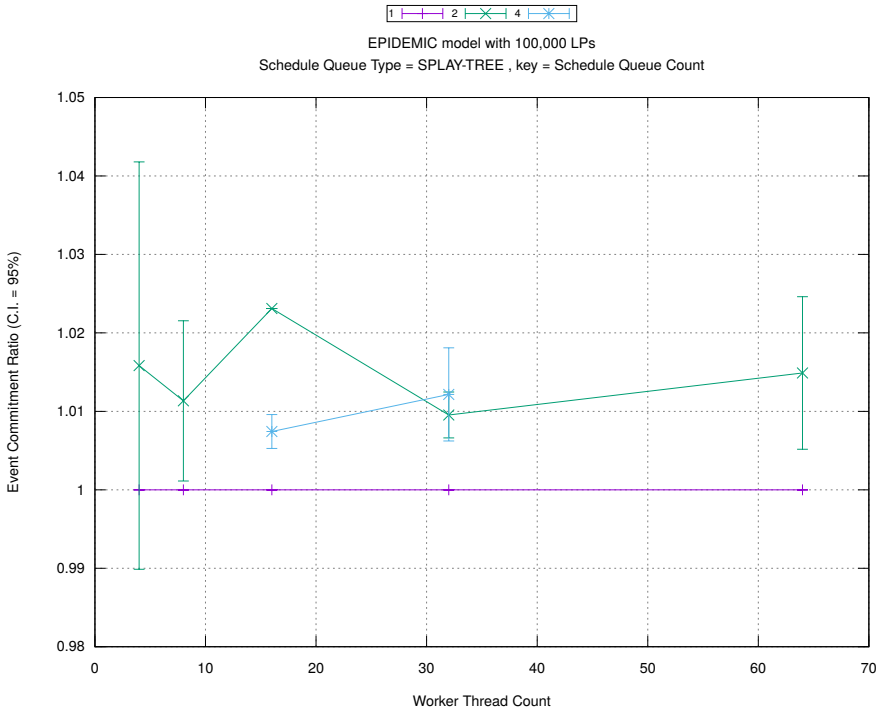
Figure A.186: epidemic 100k ba/plots/scheduleq/threads vs count key type lockfree-unsorted-bottom 96



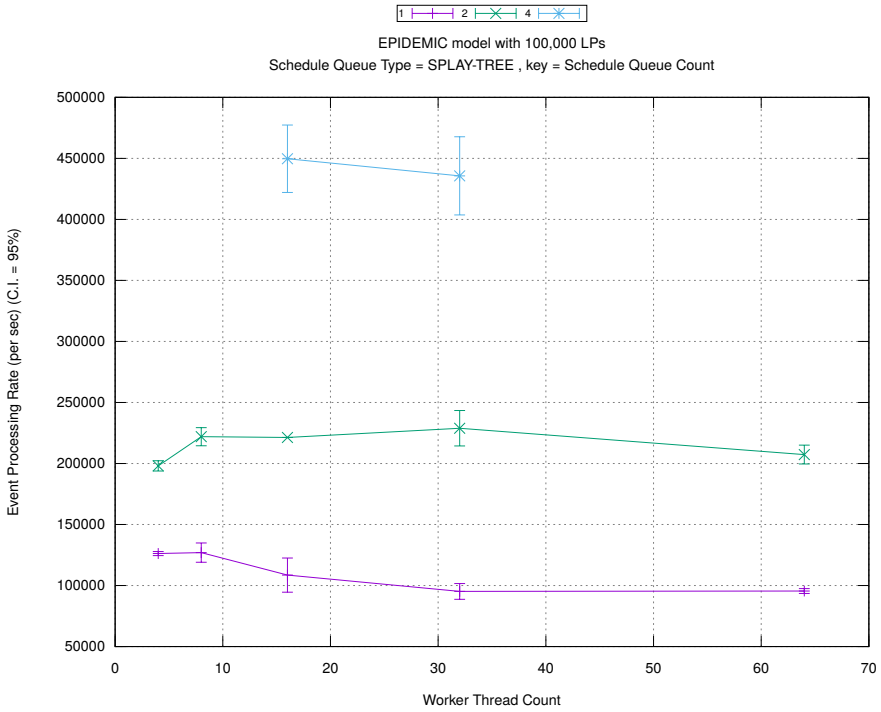
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

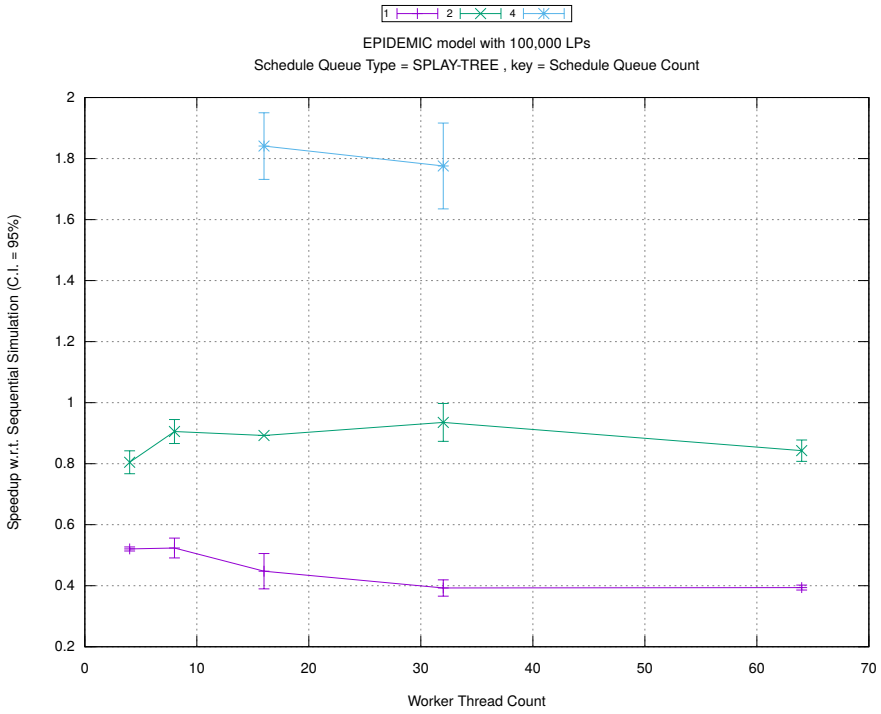


(b) Event Commitment Ratio

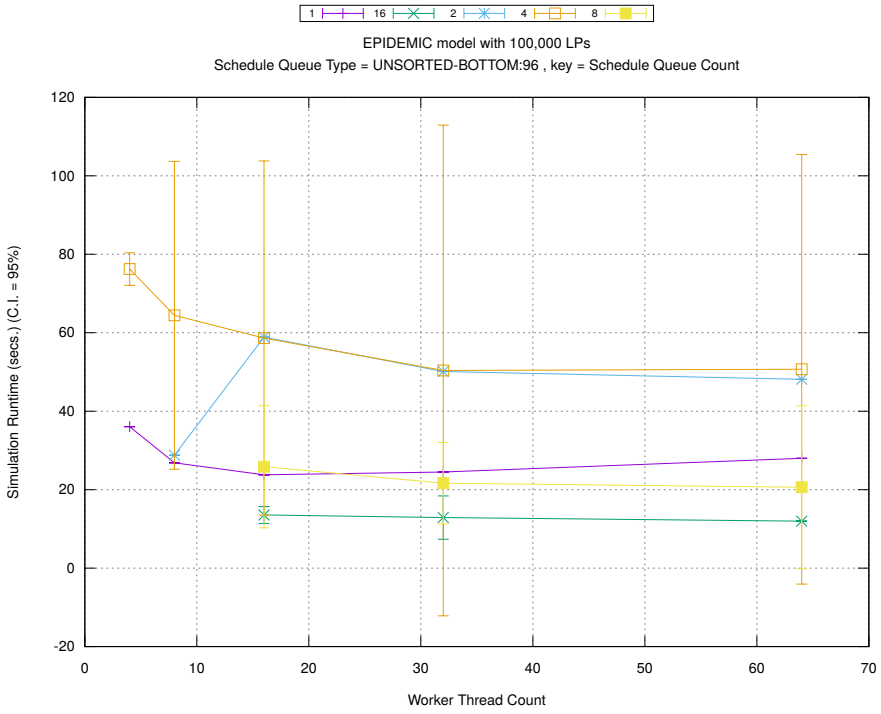


(c) Event Processing Rate (per second)

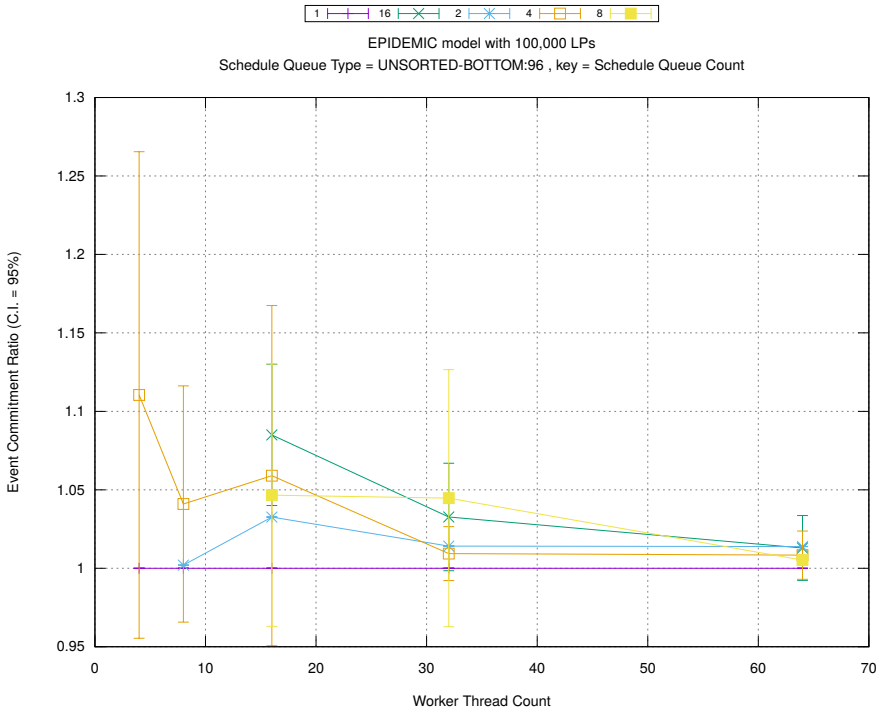
Figure A.187: epidemic 100k ba/plots/scheduleq/threads vs count key type splay-tree



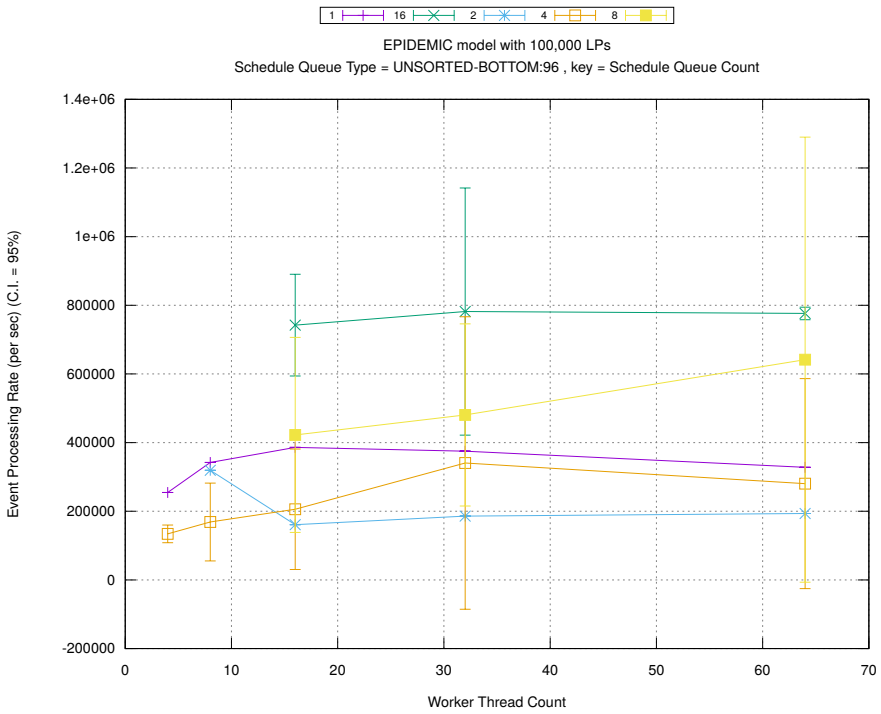
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

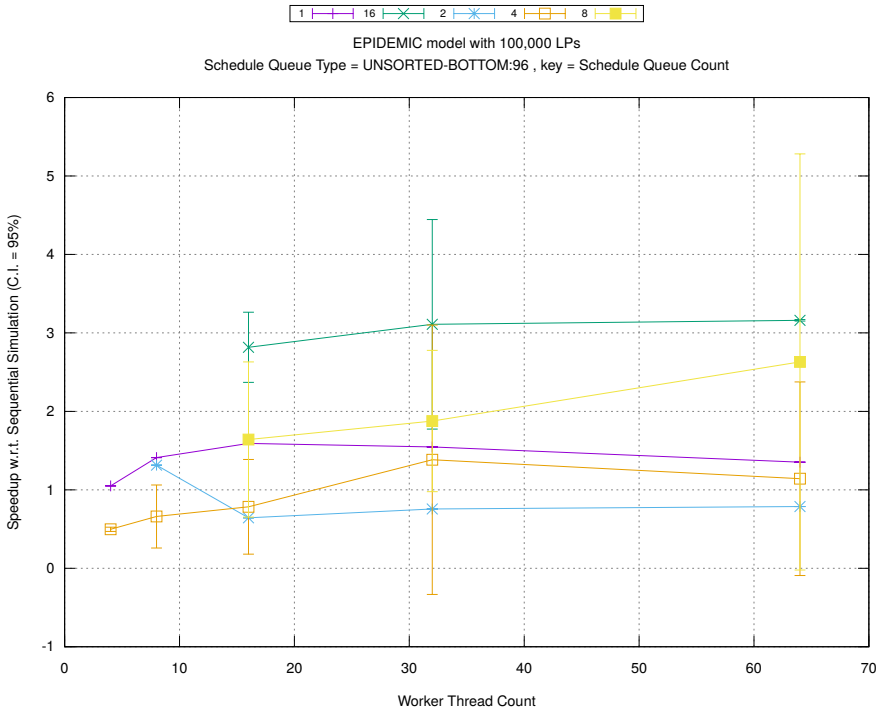


(b) Event Commitment Ratio

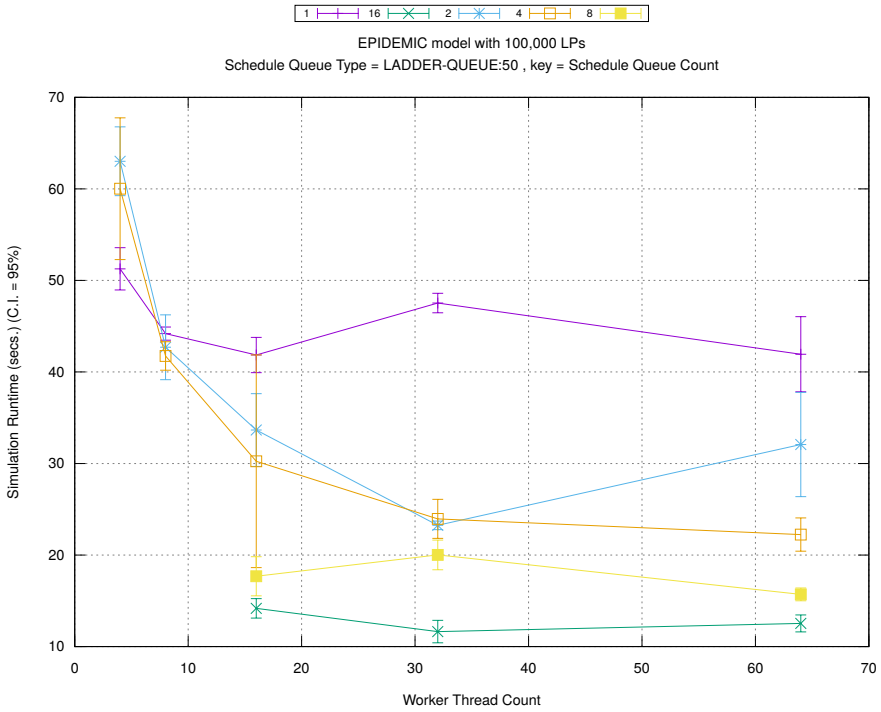


(c) Event Processing Rate (per second)

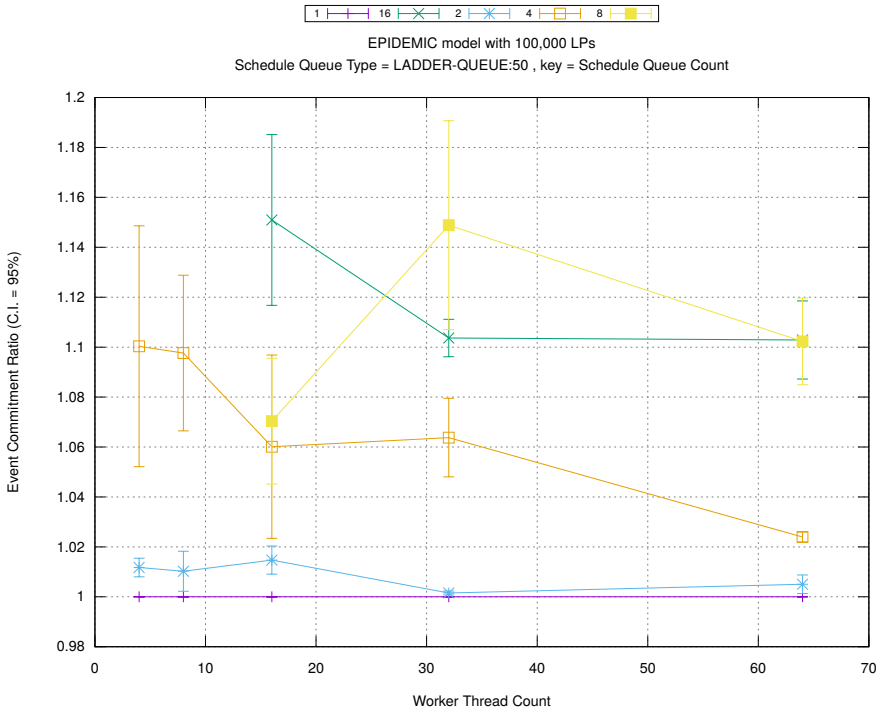
Figure A.188: epidemic 100k ba/plots/scheduleq/threads vs count key type unsorted-bottom 96



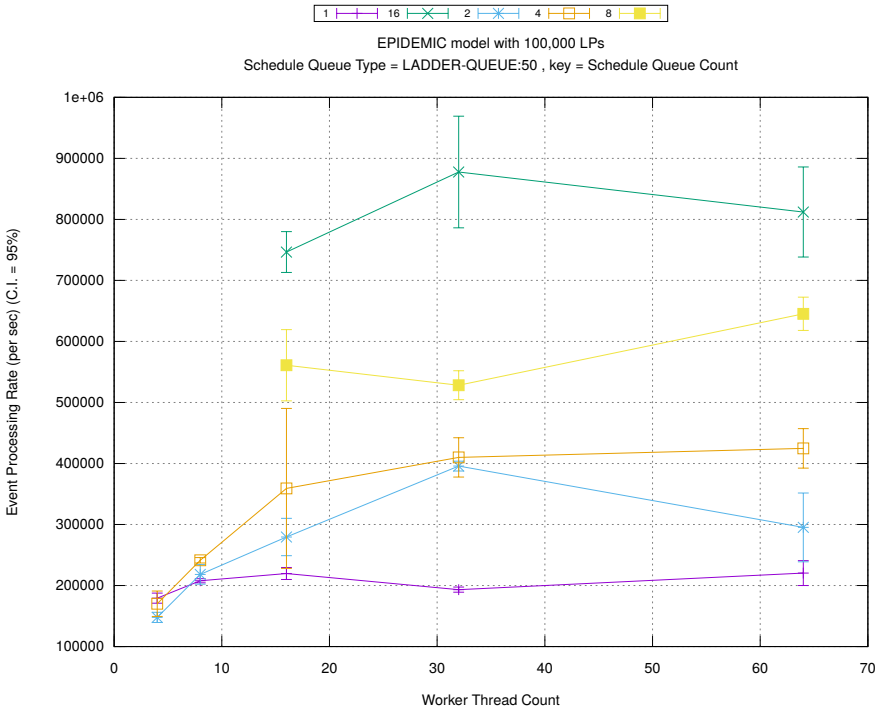
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

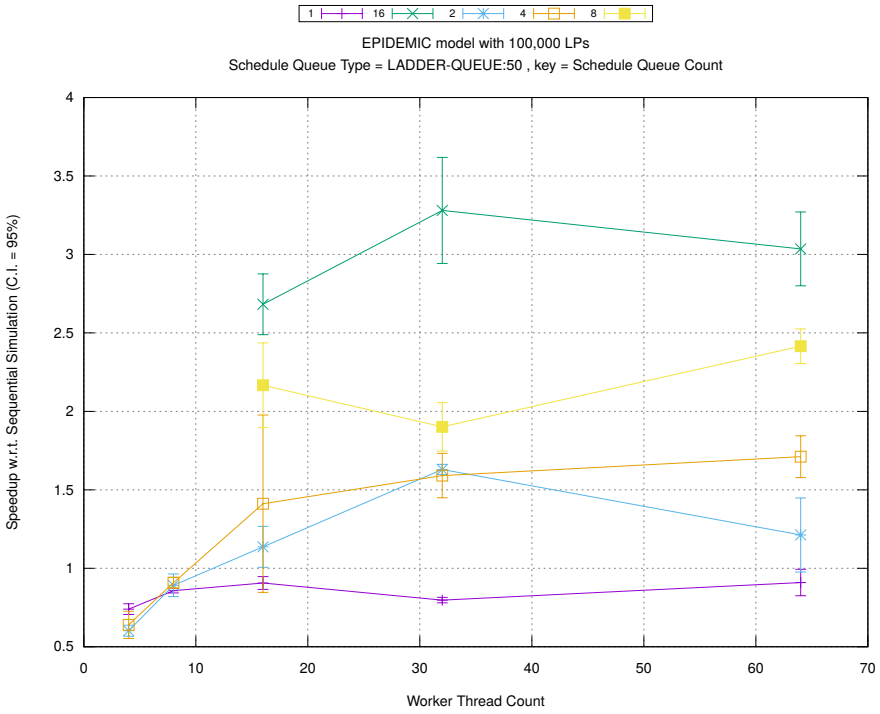


(b) Event Commitment Ratio

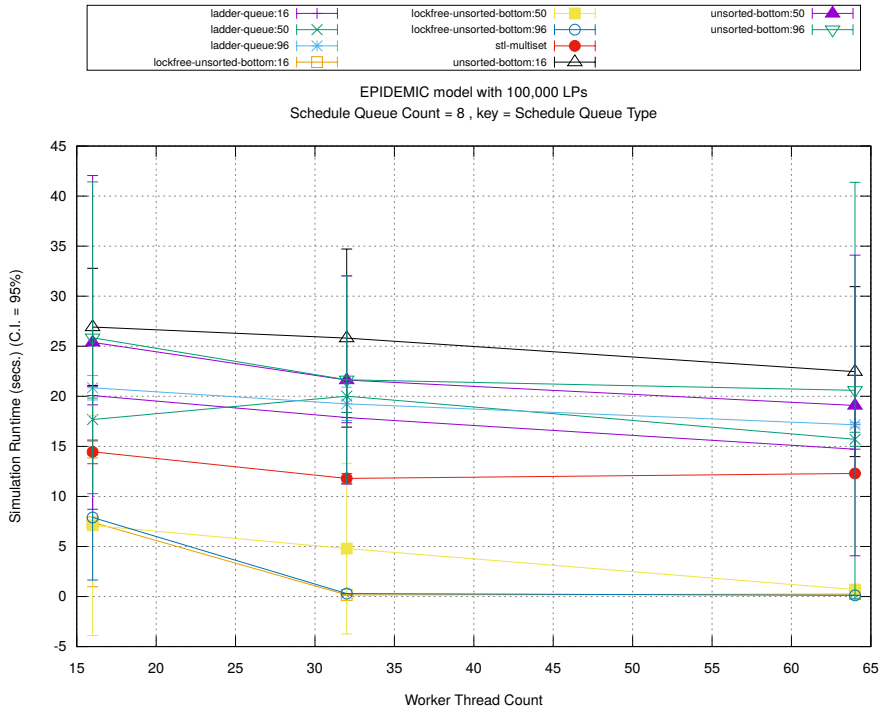


(c) Event Processing Rate (per second)

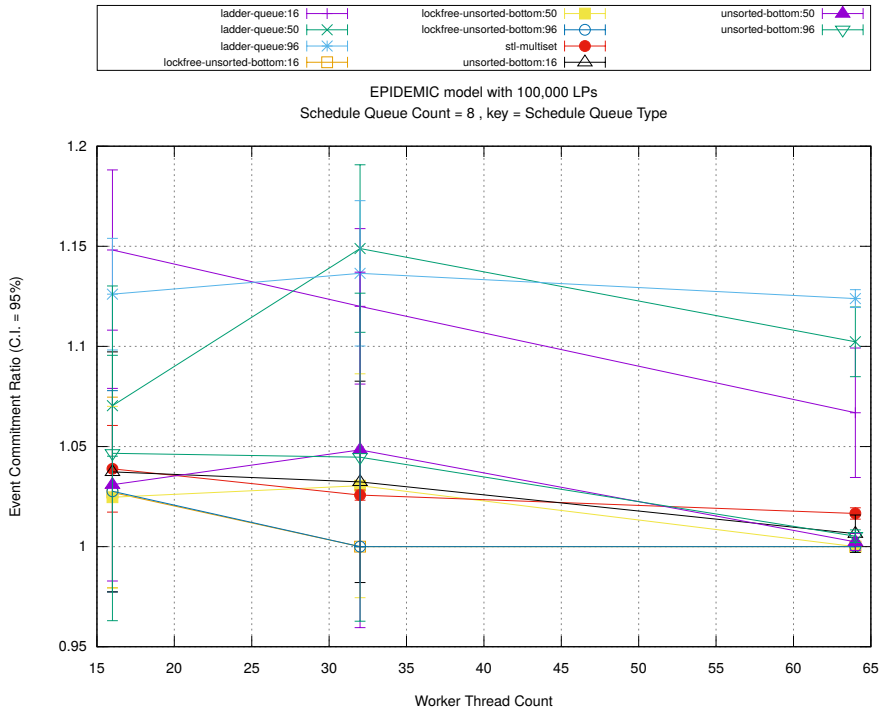
Figure A.189: epidemic 100k ba/plots/scheduleq/threads vs count key type ladder-queue 50



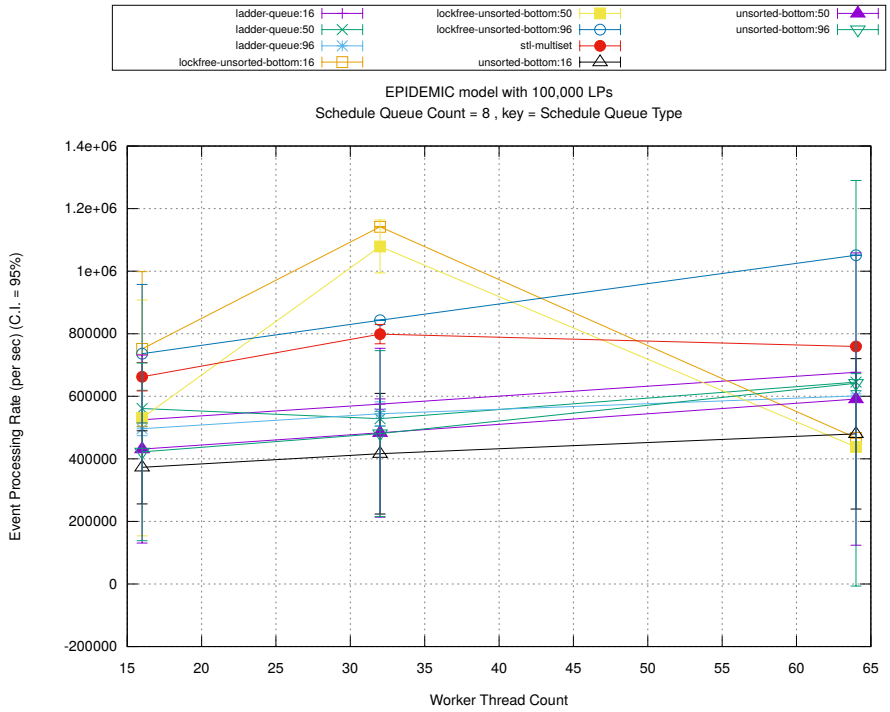
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

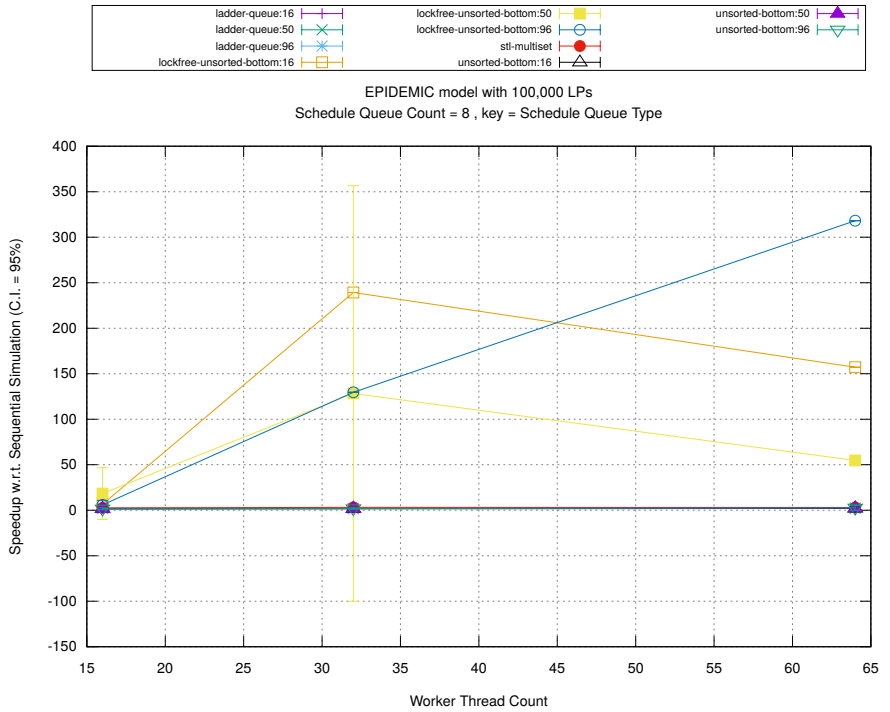


(b) Event Commitment Ratio

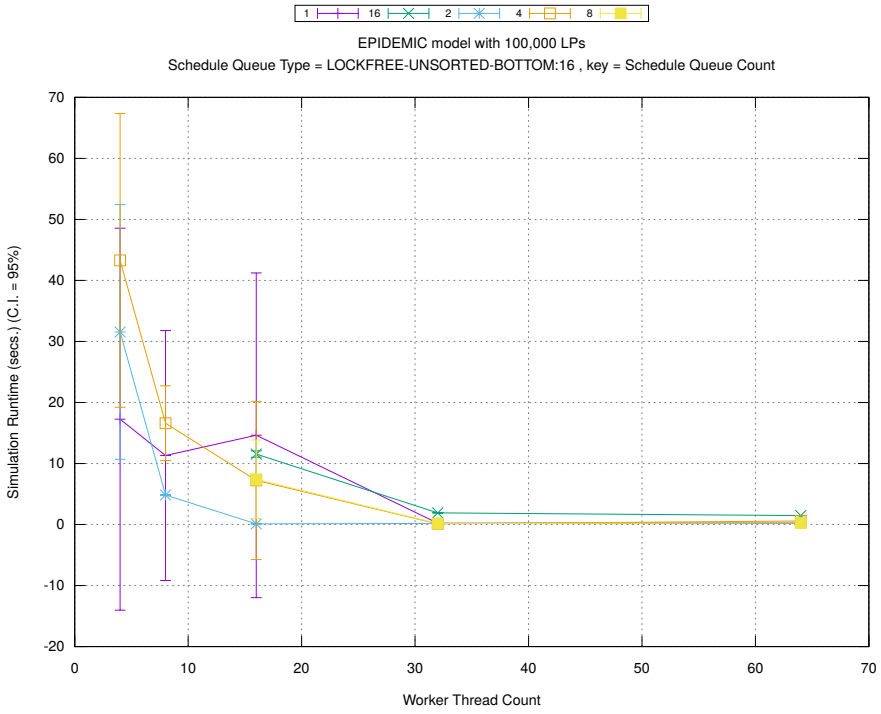


(c) Event Processing Rate (per second)

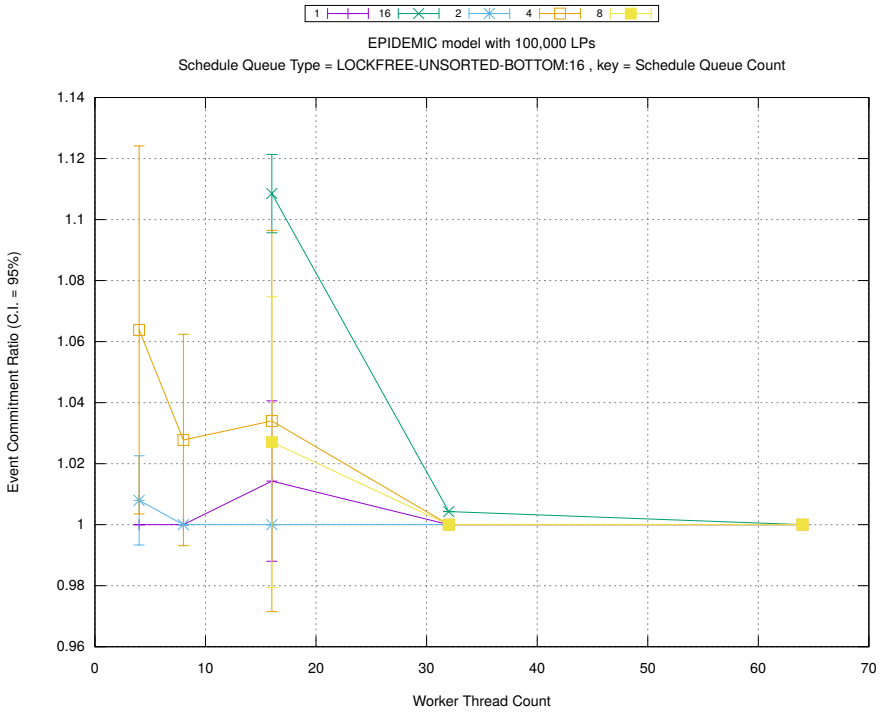
Figure A.190: epidemic 100k ba/plots/scheduleq/threads vs type key count 8



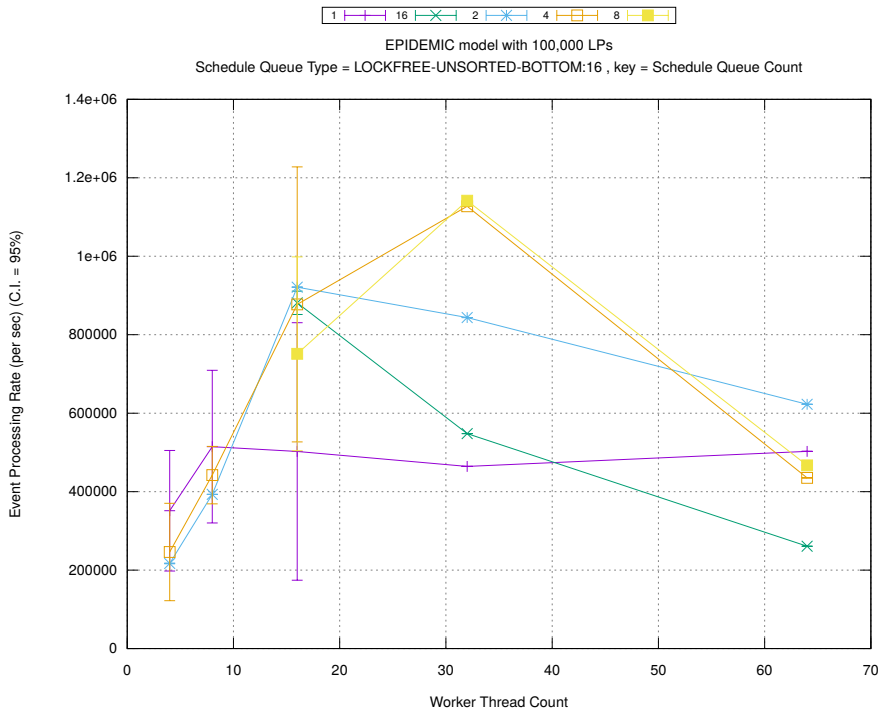
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

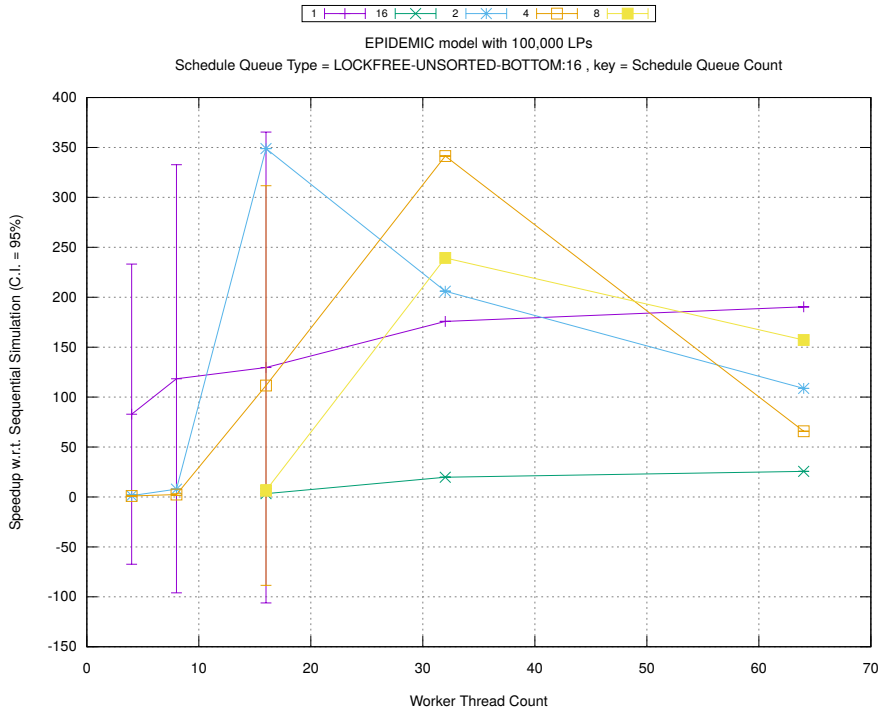


(b) Event Commitment Ratio

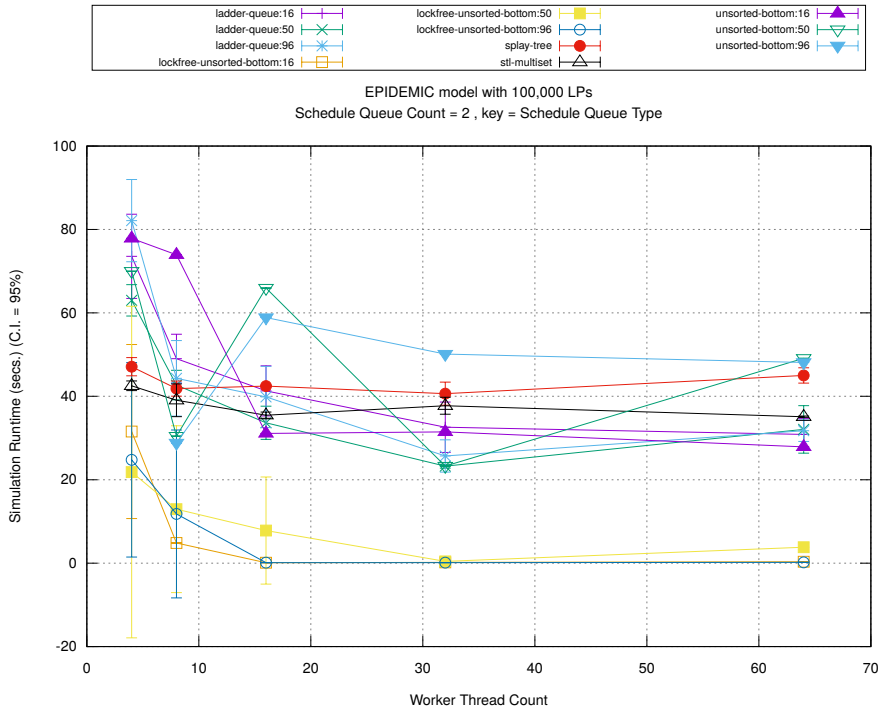


(c) Event Processing Rate (per second)

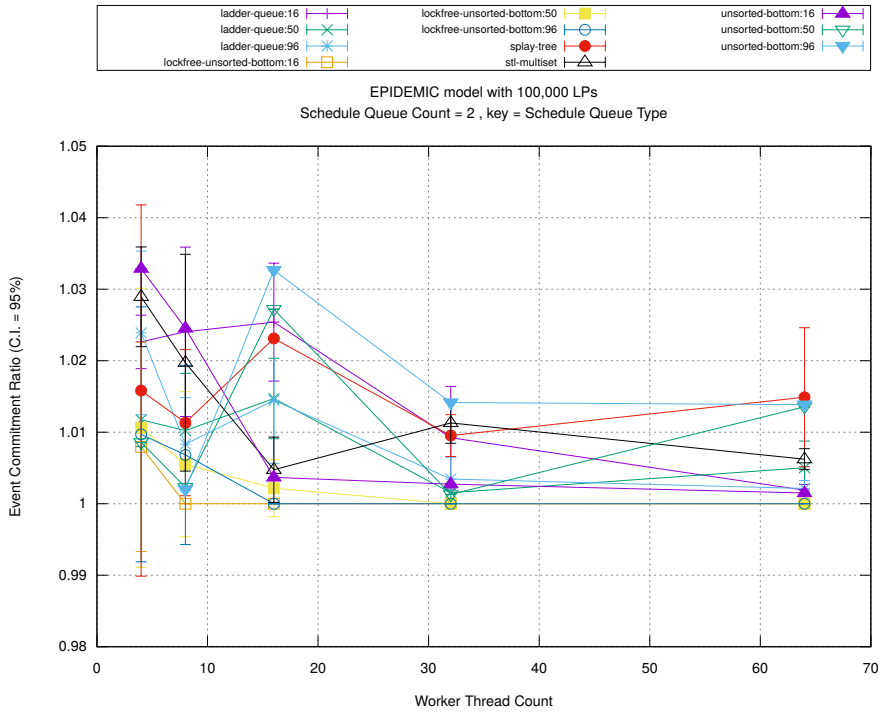
Figure A.191: epidemic 100k ba/plots/scheduleq/threads vs count key type lockfree-unsorted-bottom 16



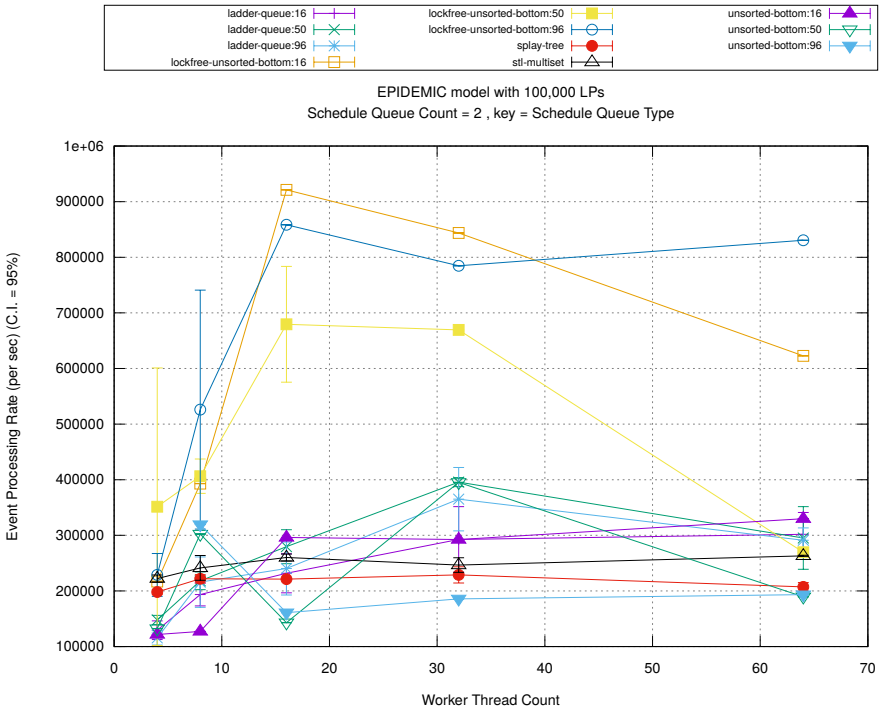
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

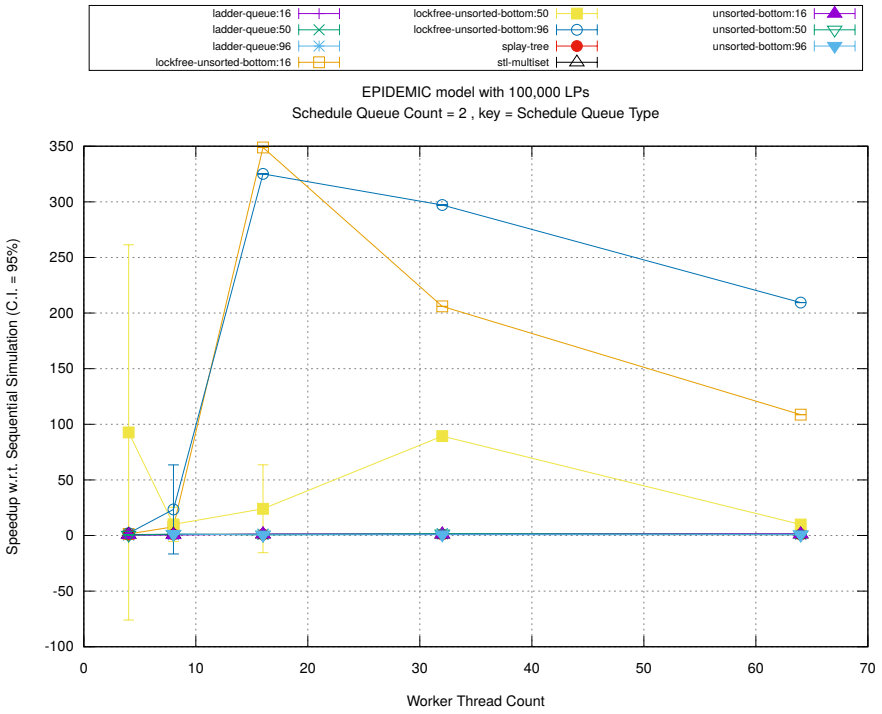


(b) Event Commitment Ratio

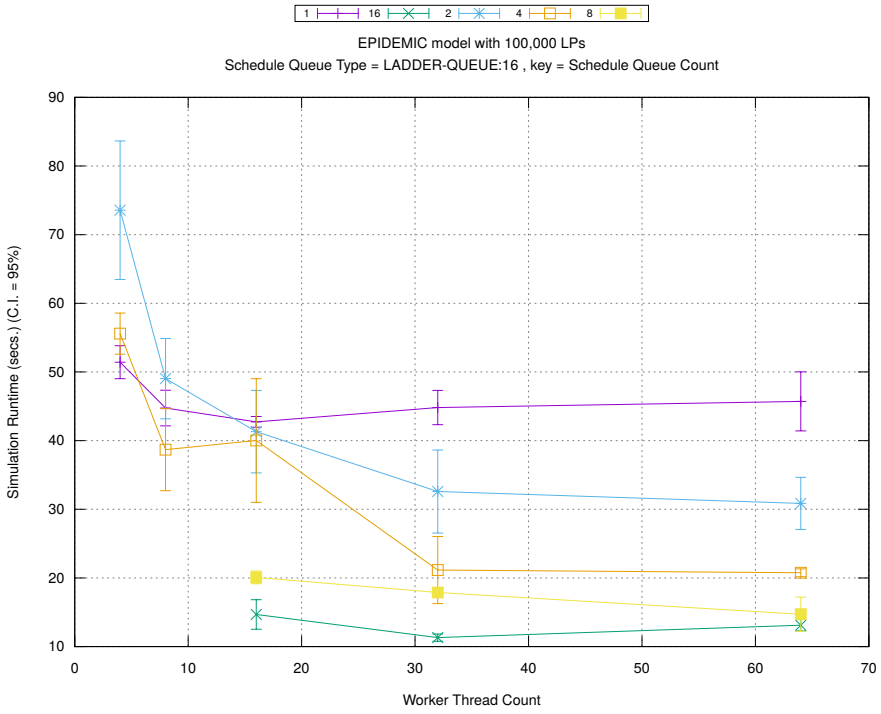


(c) Event Processing Rate (per second)

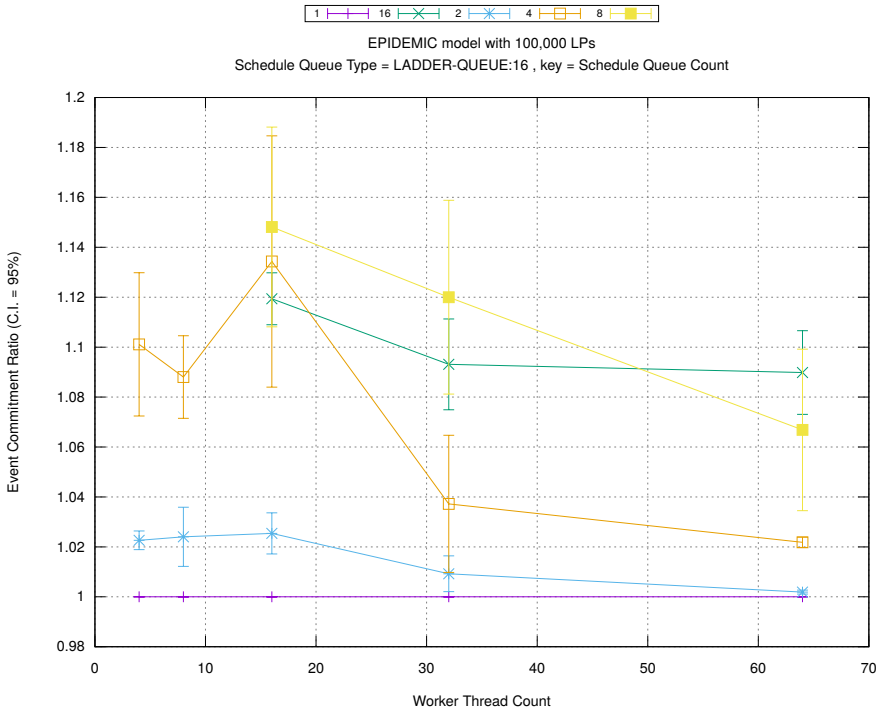
Figure A.192: epidemic 100k ba/plots/scheduleq/threads vs type key count 2



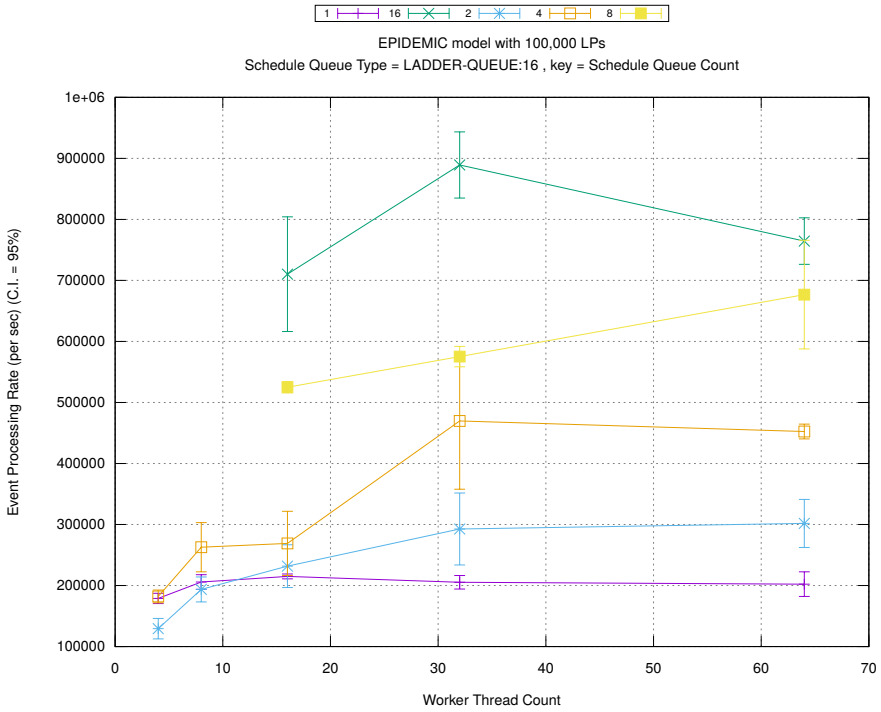
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

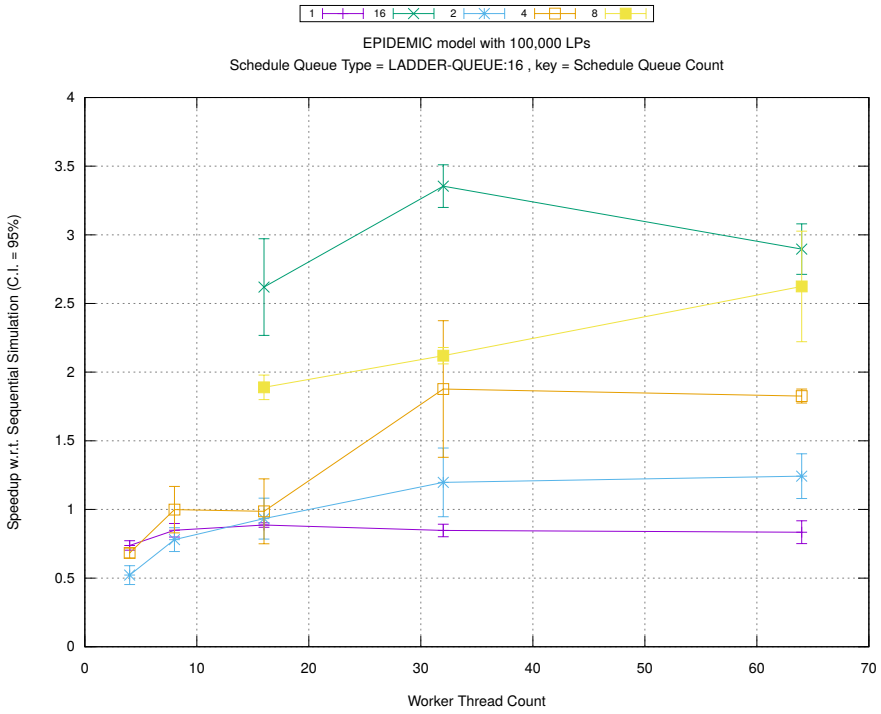


(b) Event Commitment Ratio

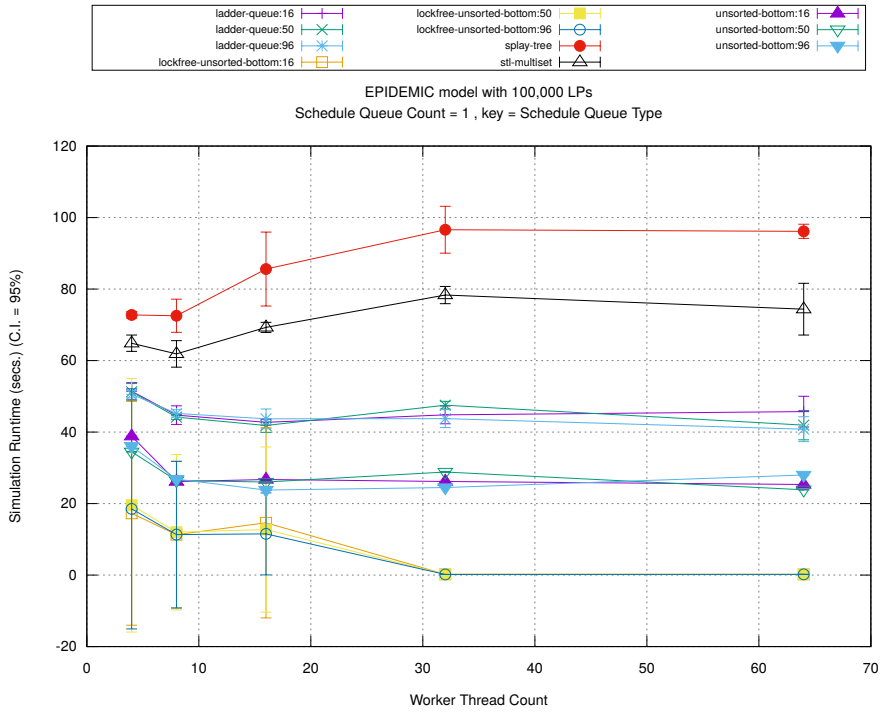


(c) Event Processing Rate (per second)

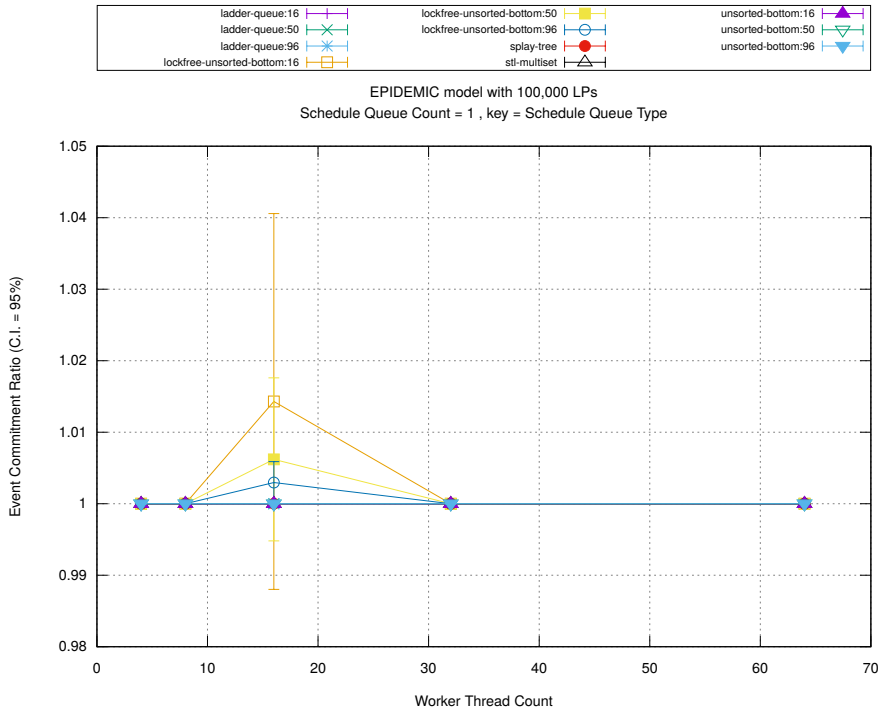
Figure A.193: epidemic 100k ba/plots/scheduleq/threads vs count key type ladder-queue 16



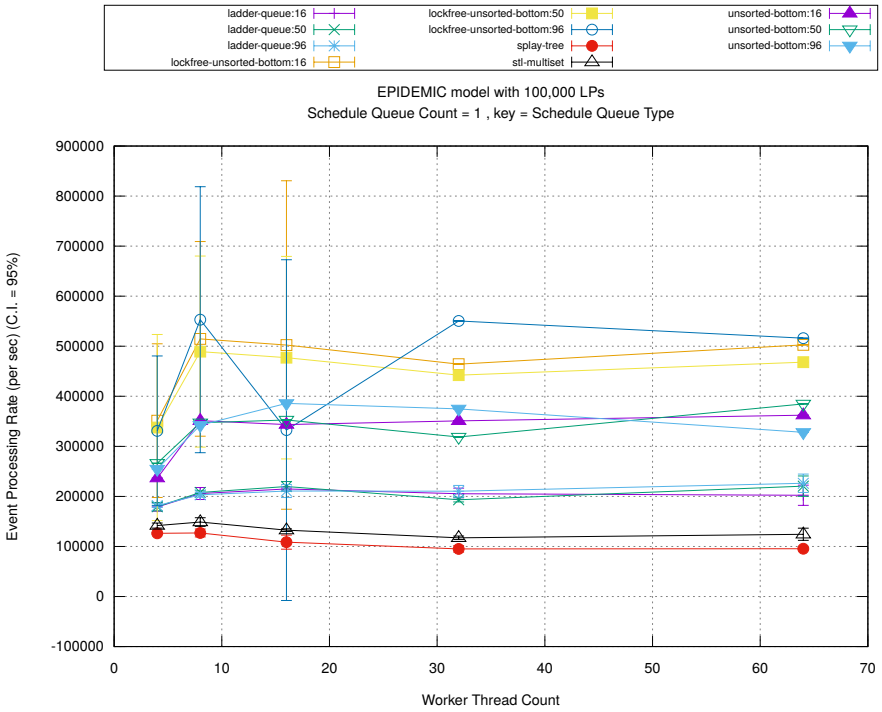
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

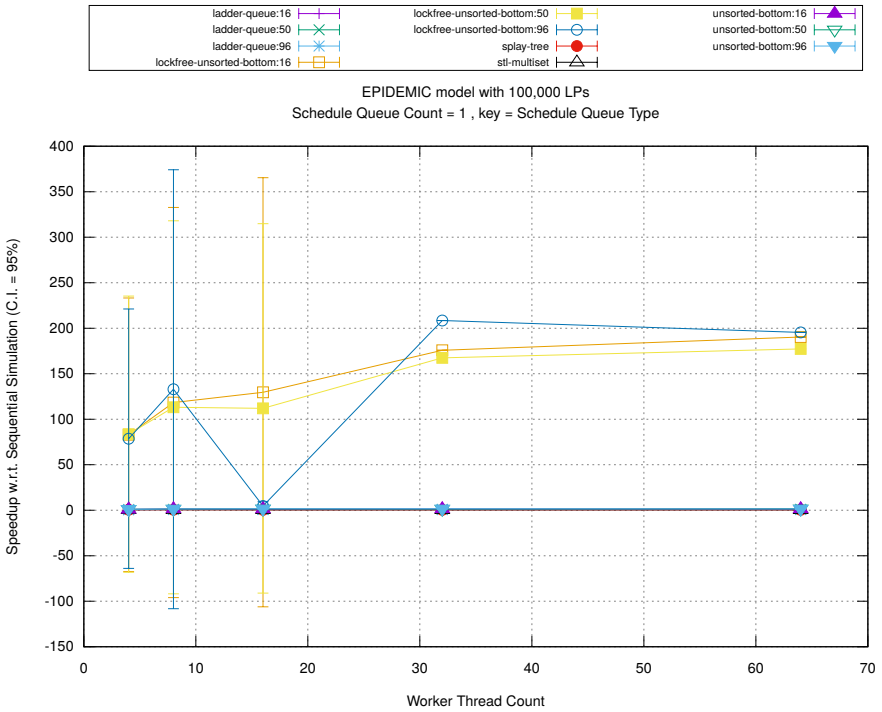


(b) Event Commitment Ratio

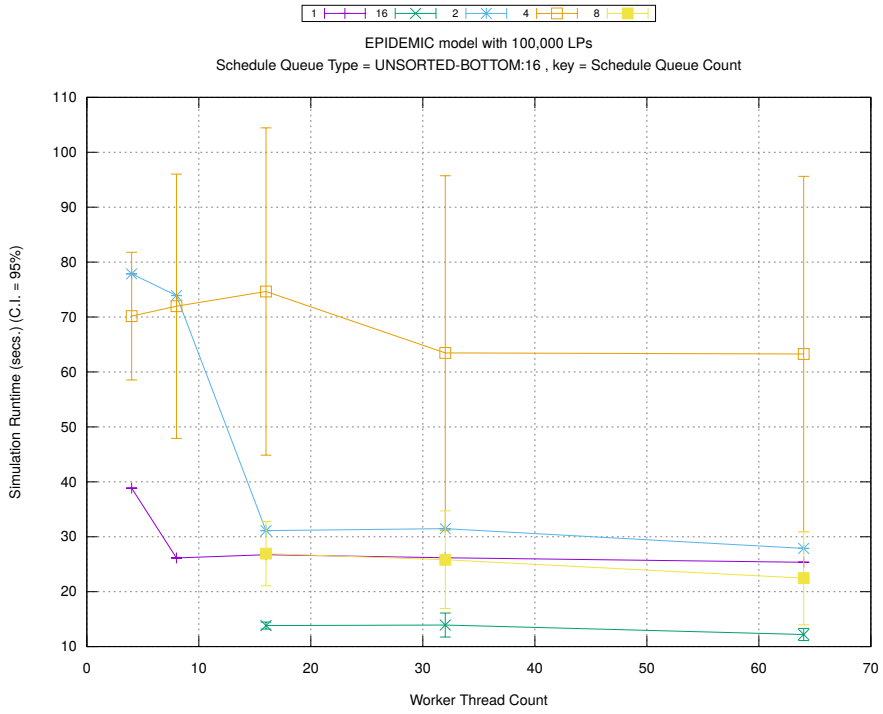


(c) Event Processing Rate (per second)

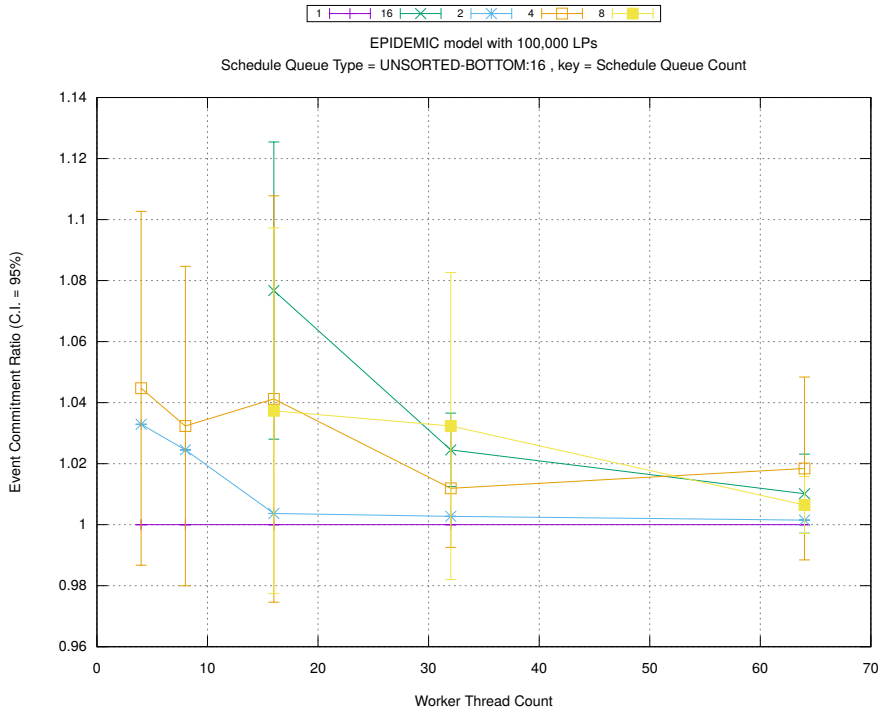
Figure A.194: epidemic 100k ba/plots/scheduleq/threads vs type key count 1



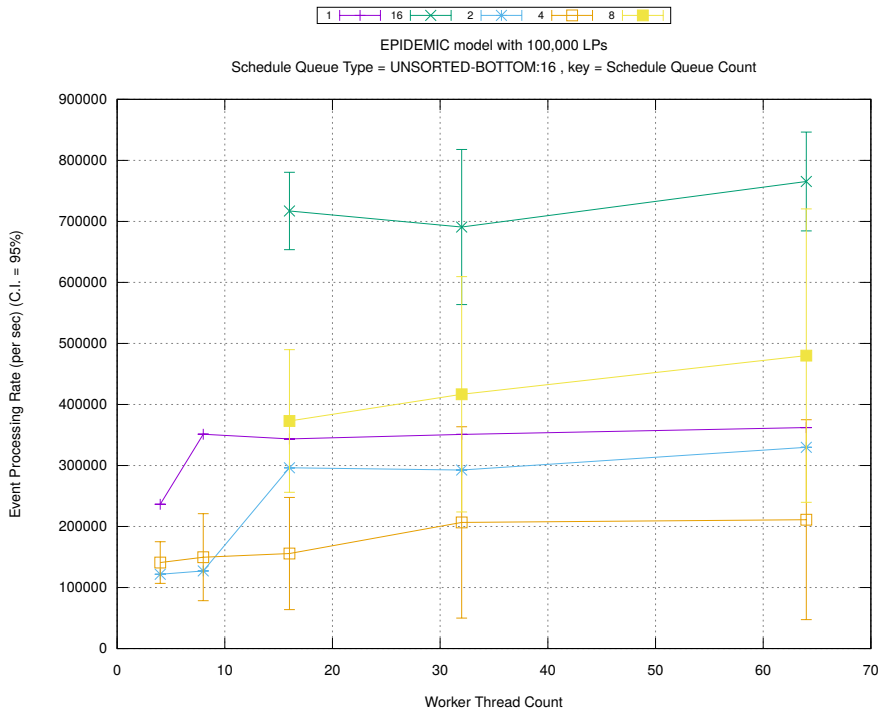
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

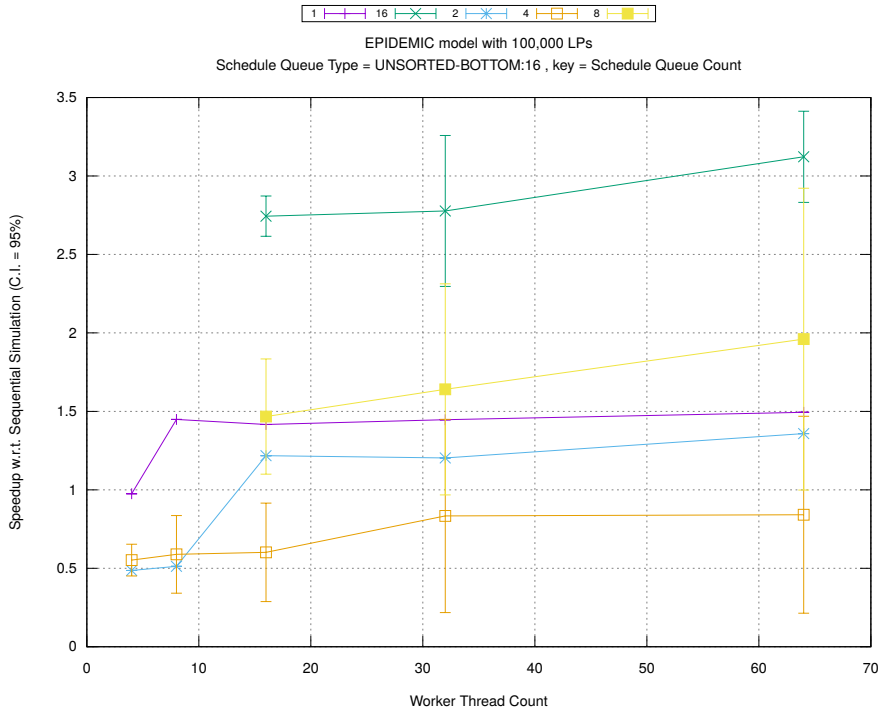


(b) Event Commitment Ratio

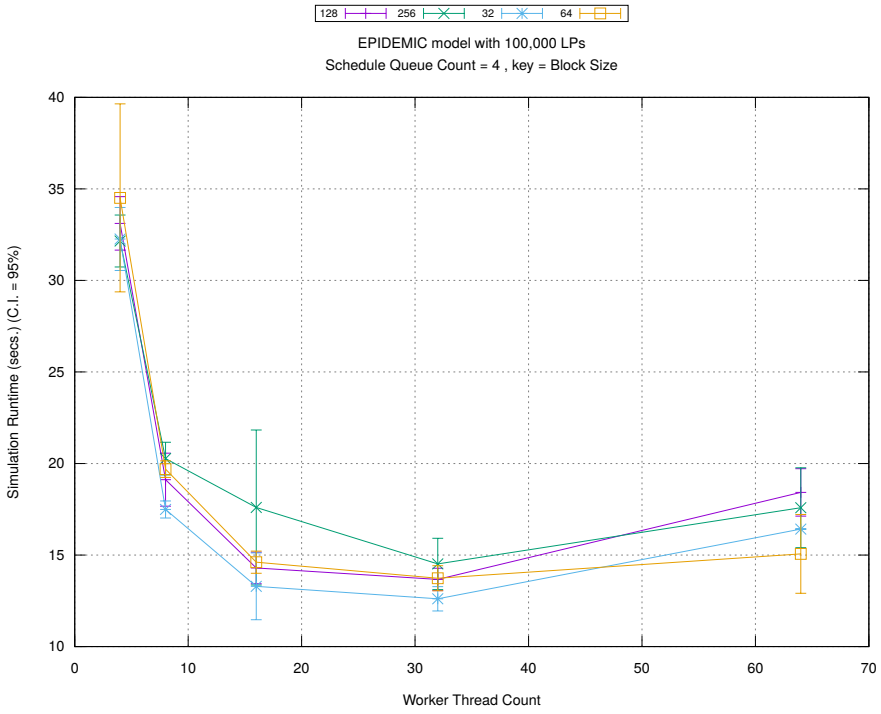


(c) Event Processing Rate (per second)

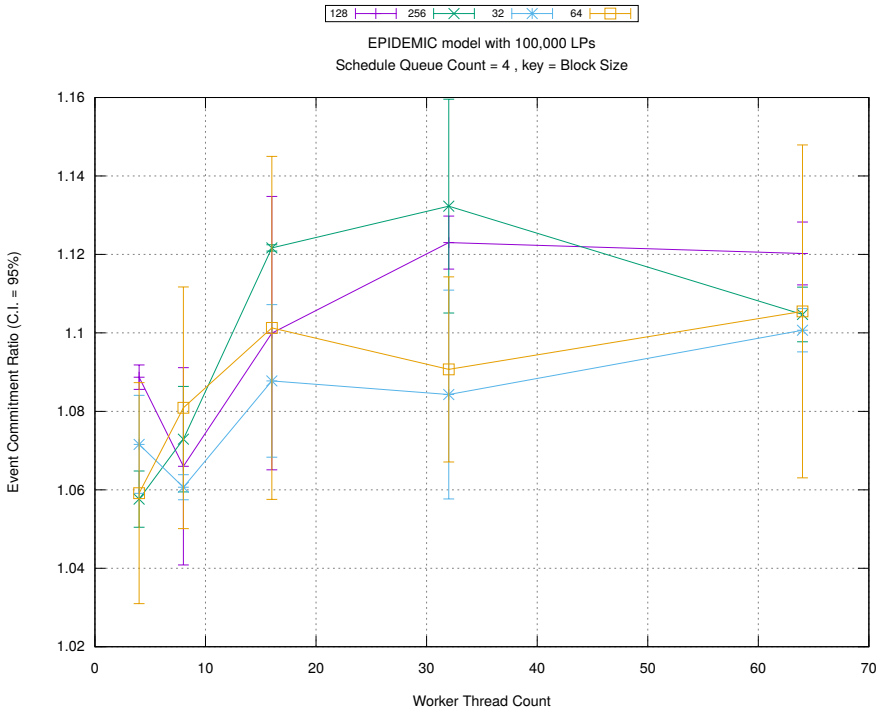
Figure A.195: epidemic 100k ba/plots/scheduleq/threads vs count key type unsorted-bottom 16



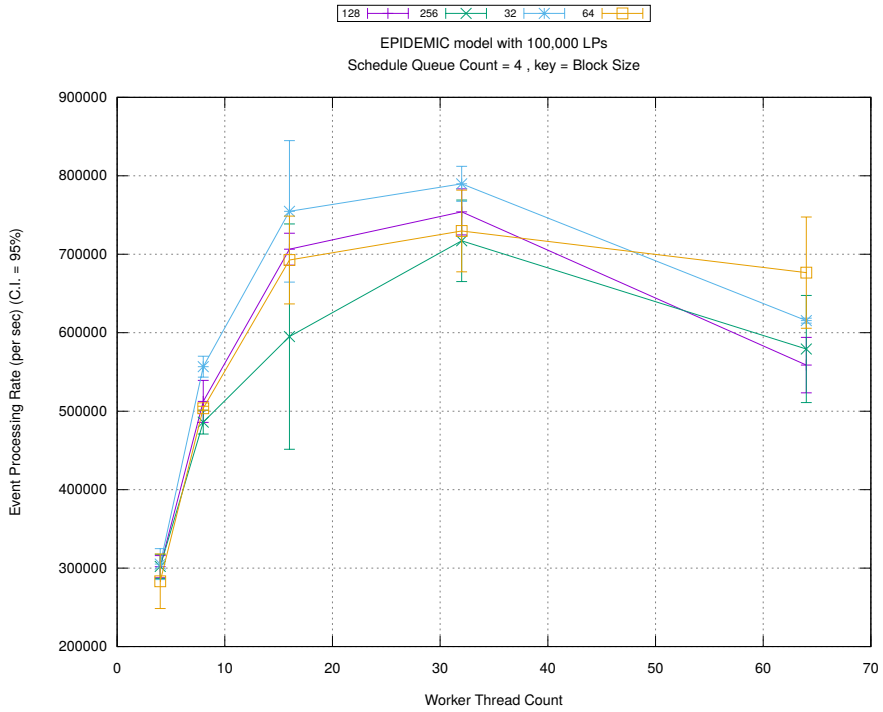
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

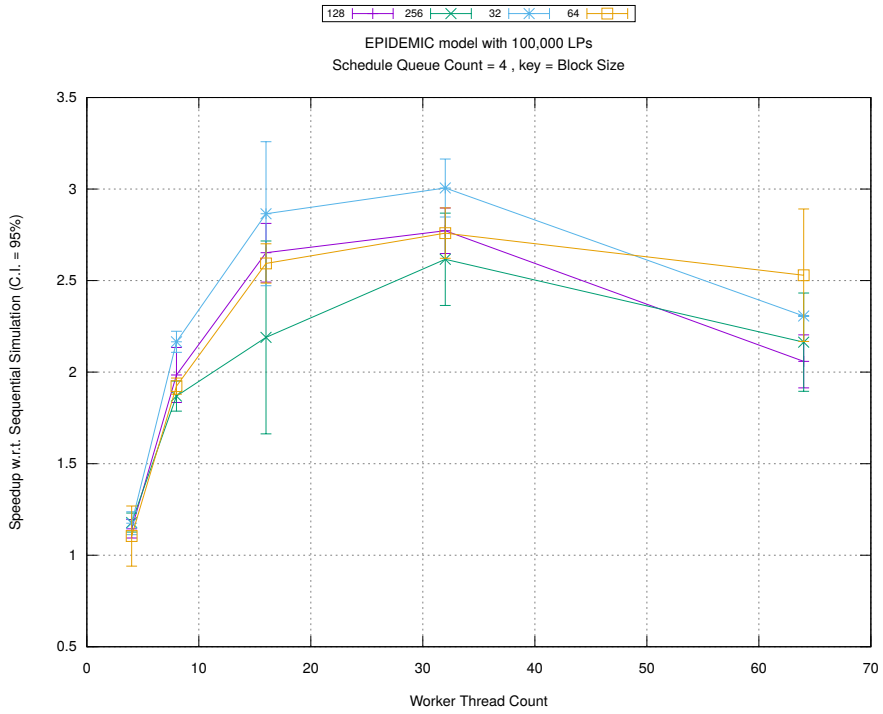


(b) Event Commitment Ratio

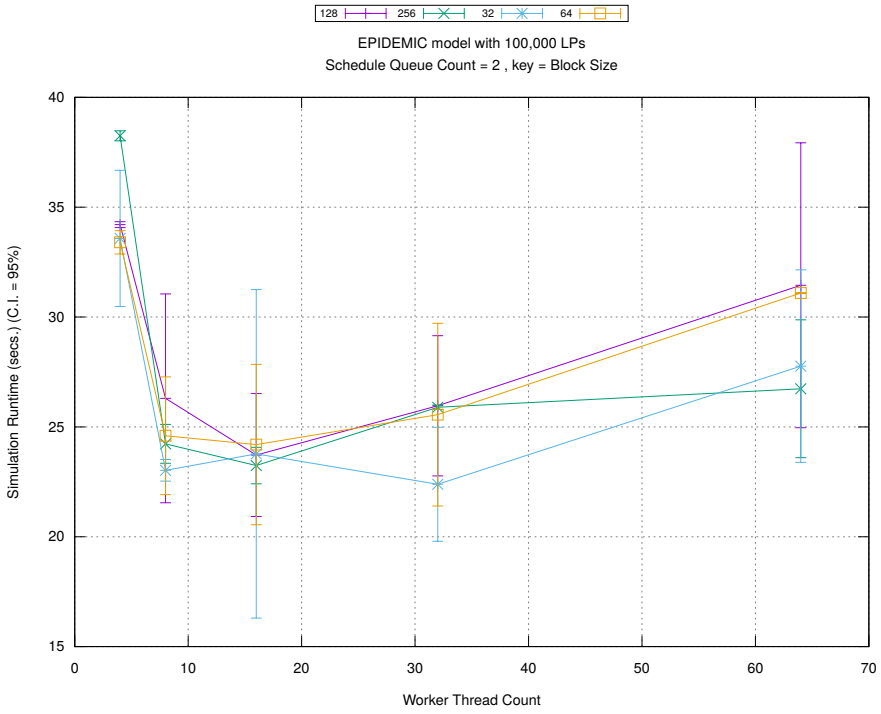


(c) Event Processing Rate (per second)

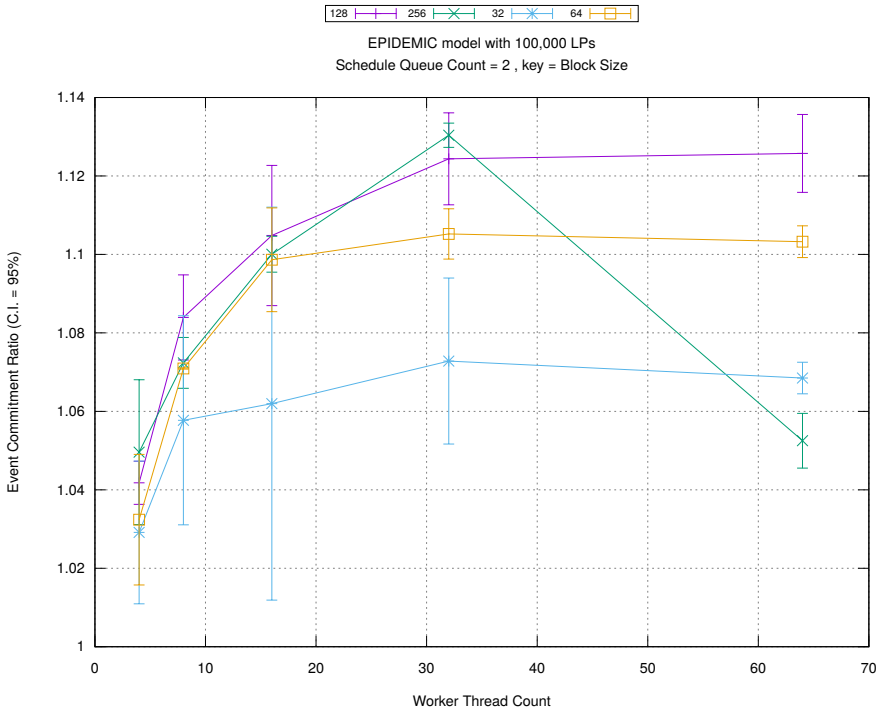
Figure A.196: epidemic 100k ba/plots/blocks/threads vs blocksize key count 4



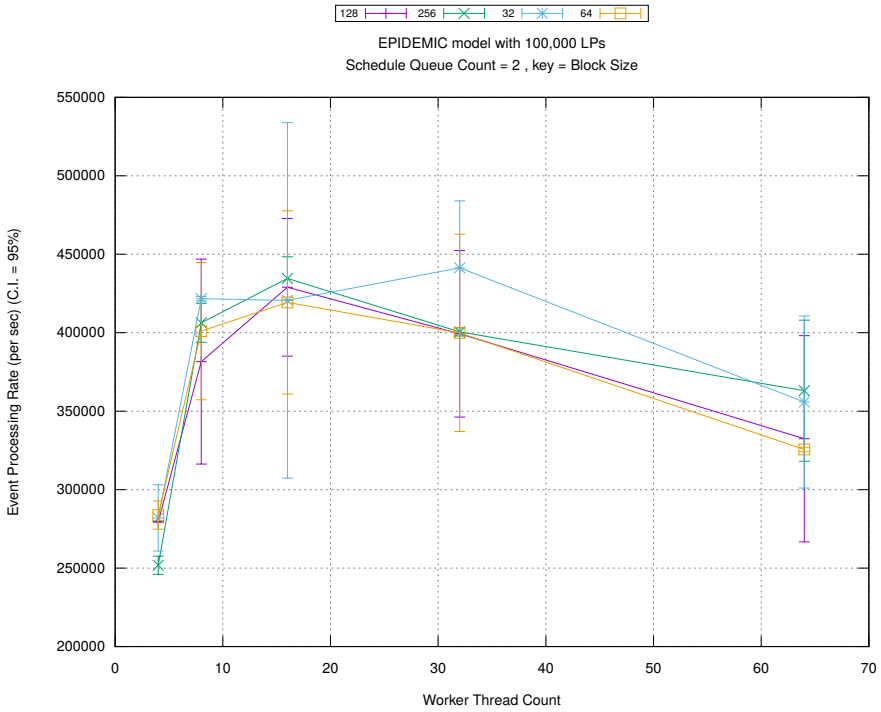
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

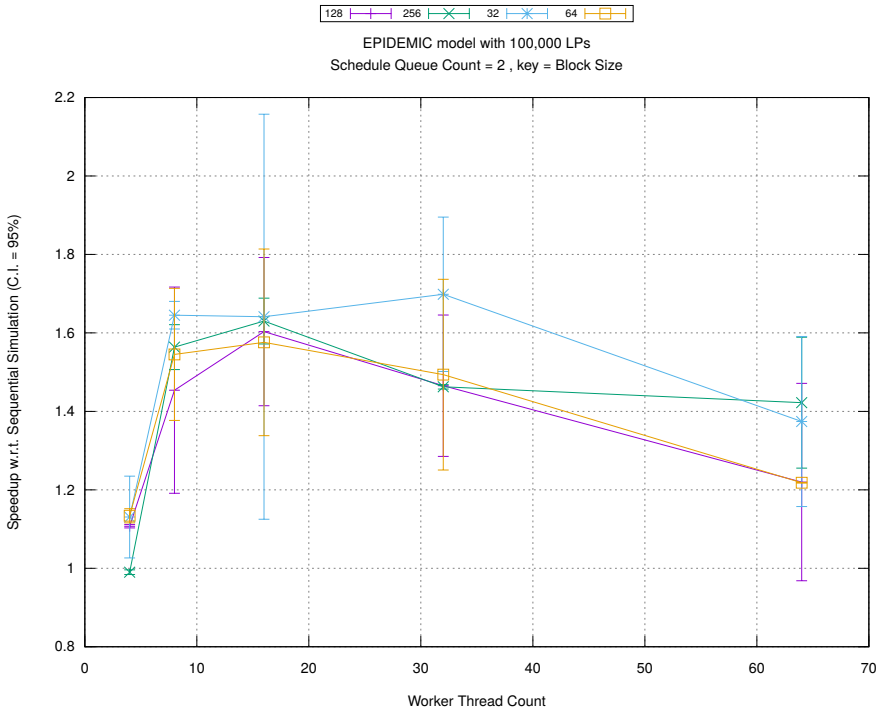


(b) Event Commitment Ratio

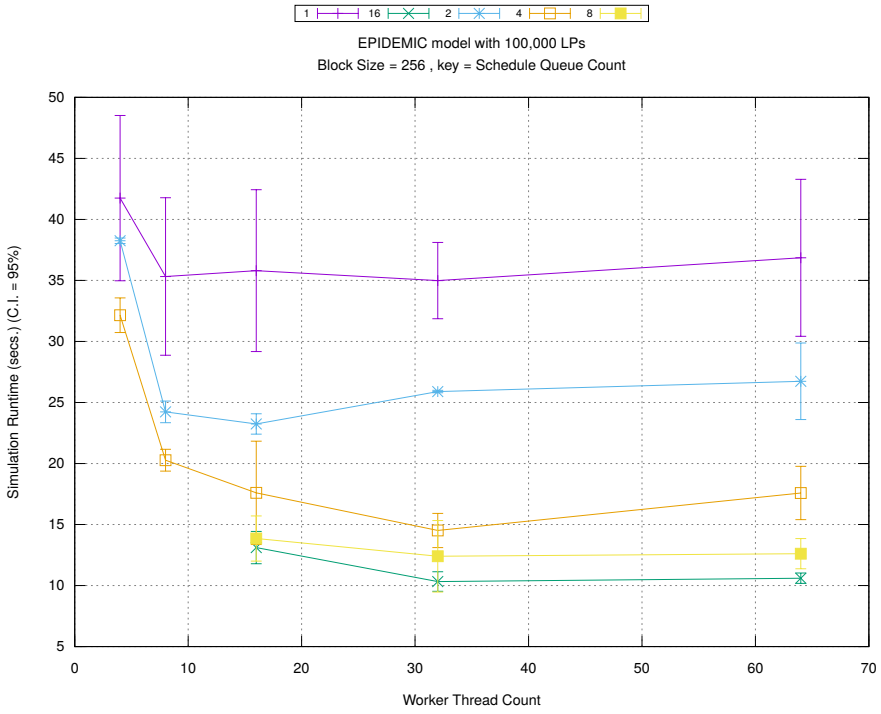


(c) Event Processing Rate (per second)

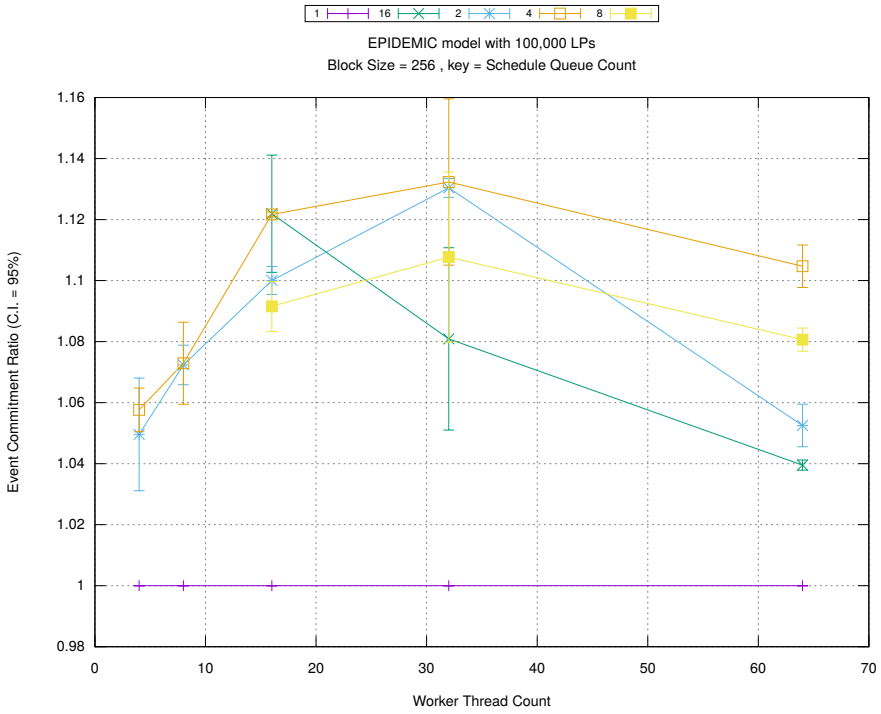
Figure A.197: epidemic 100k ba/plots/blocks/threads vs blocksize key count 2



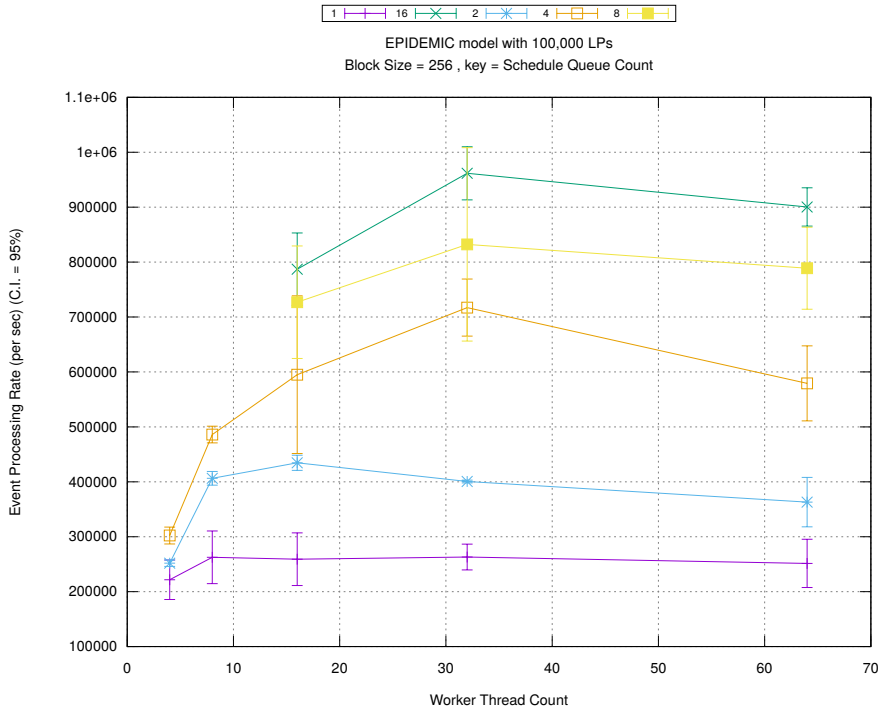
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

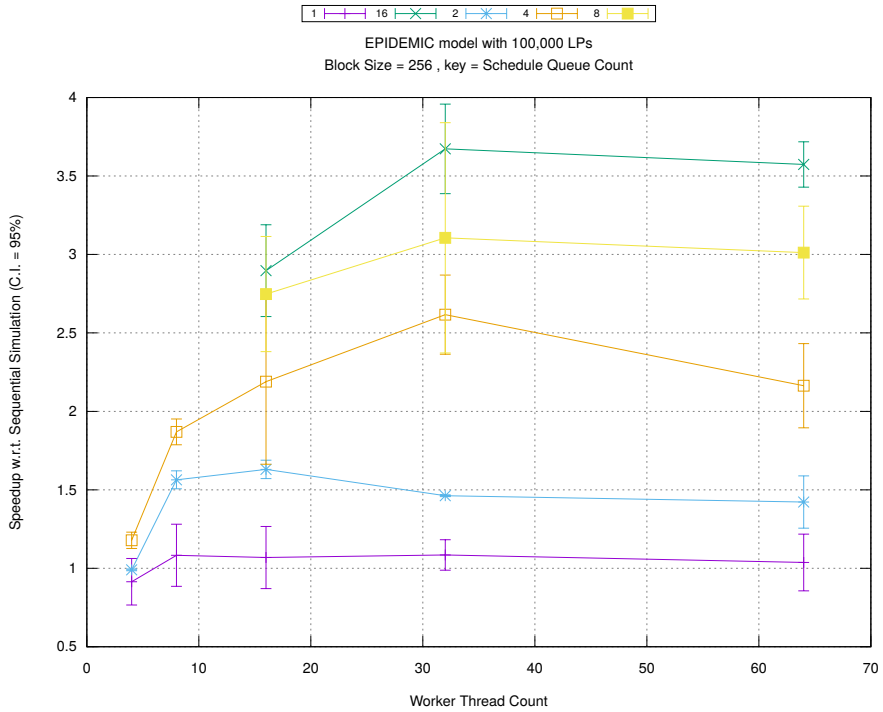


(b) Event Commitment Ratio

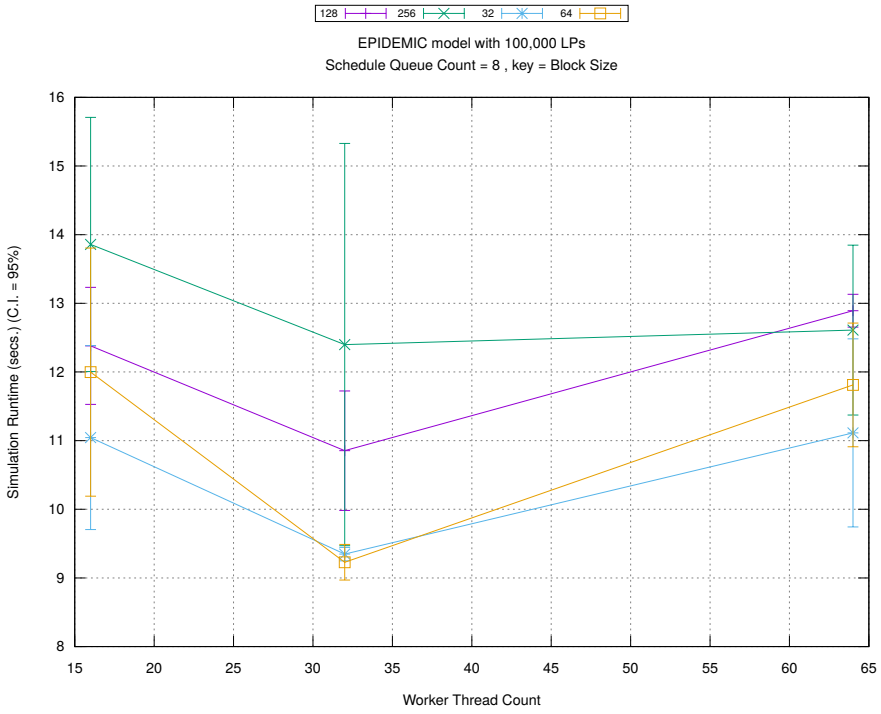


(c) Event Processing Rate (per second)

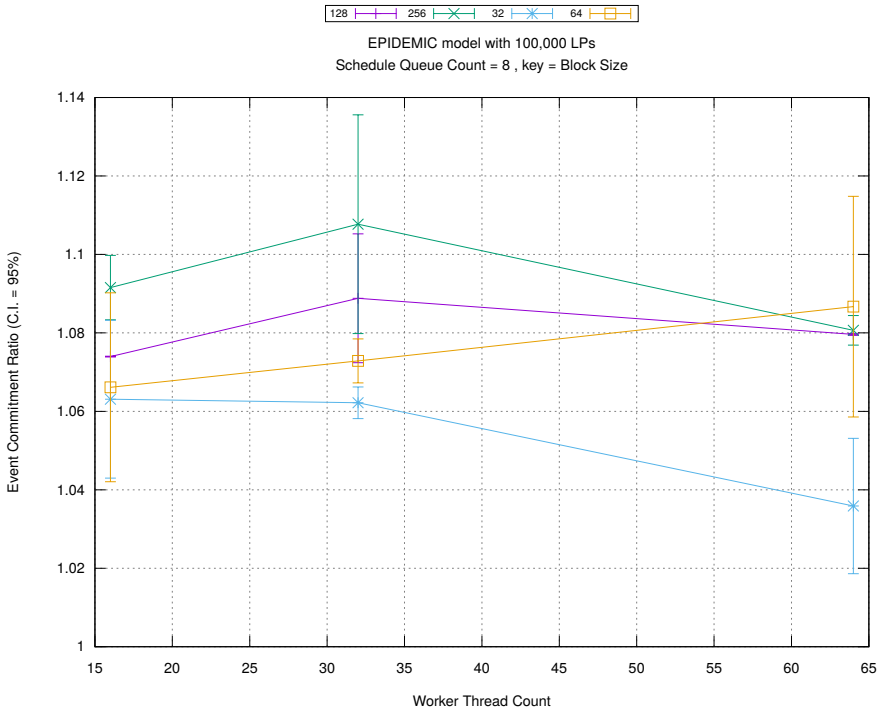
Figure A.198: epidemic 100k ba/plots/blocks/threads vs count key blocksize 256



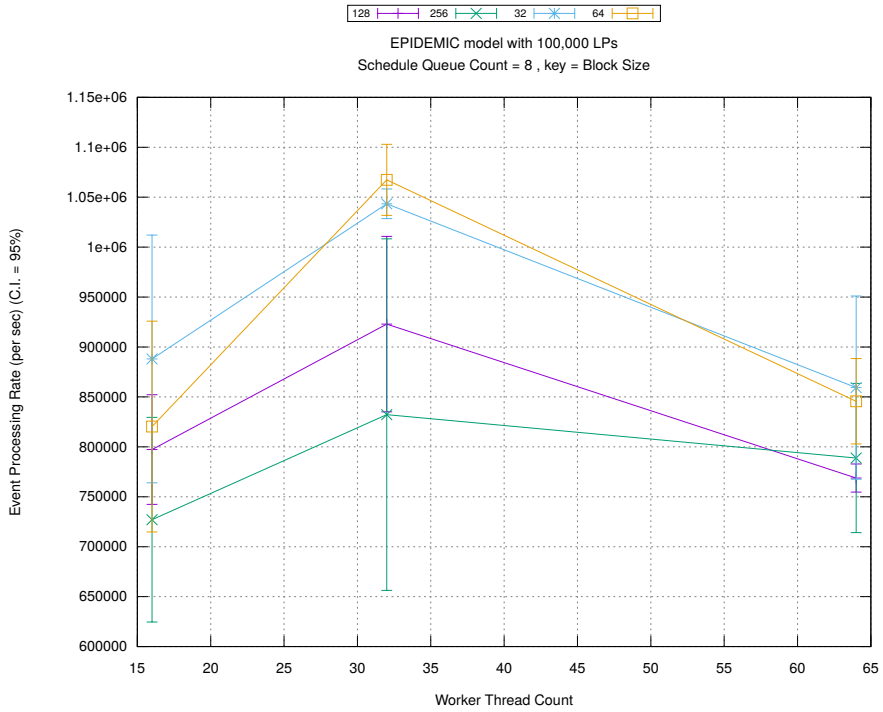
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

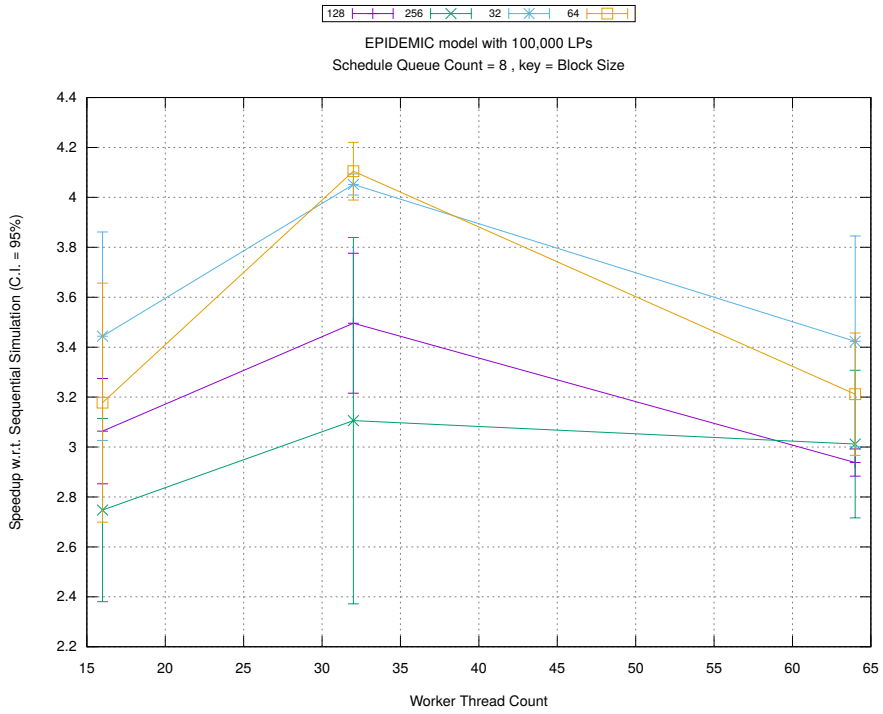


(b) Event Commitment Ratio

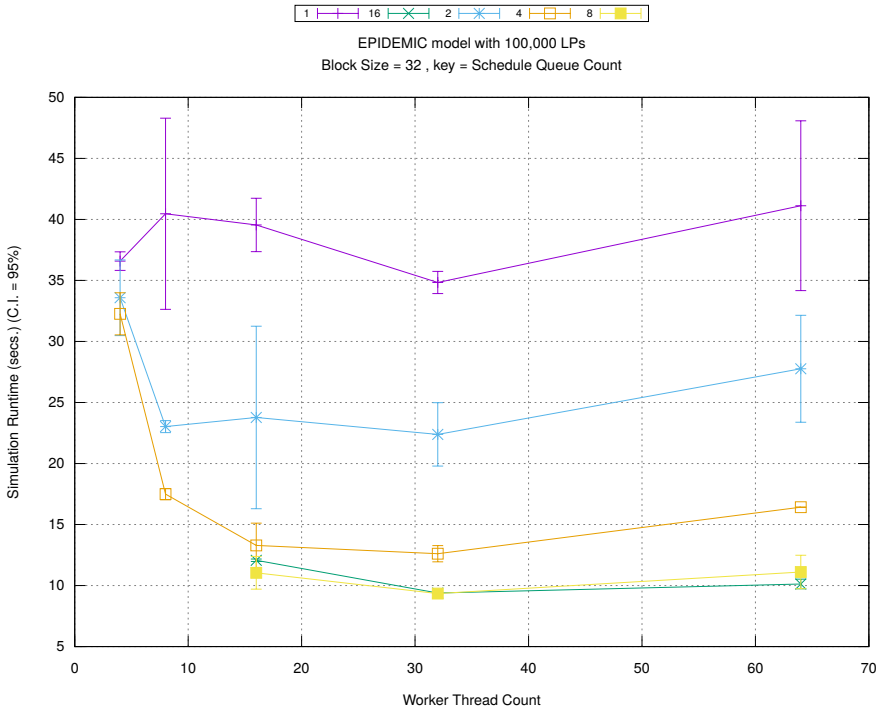


(c) Event Processing Rate (per second)

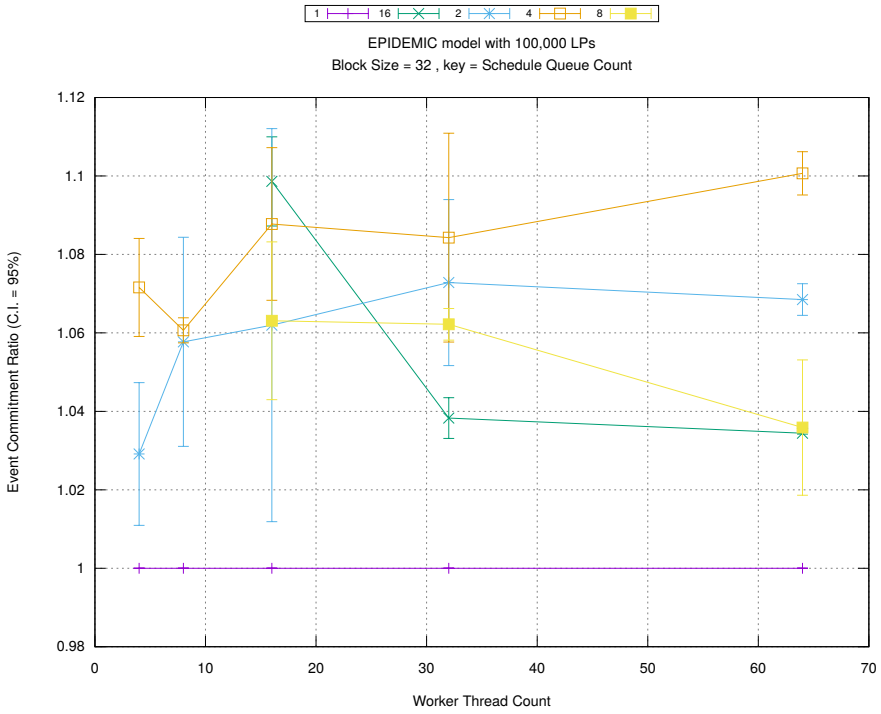
Figure A.199: epidemic 100k ba/plots/blocks/threads vs blocksize key count 8



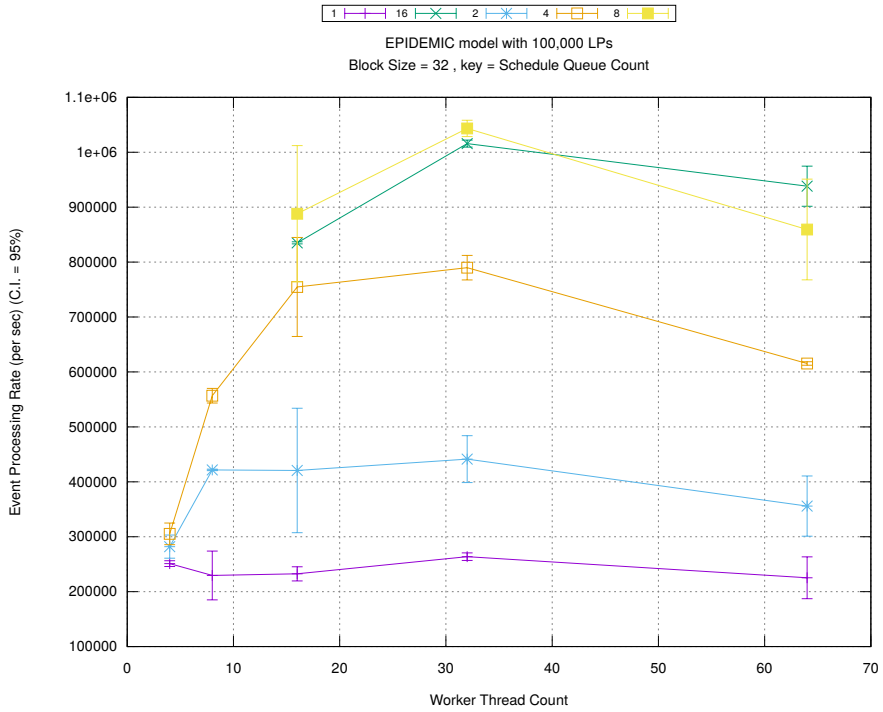
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

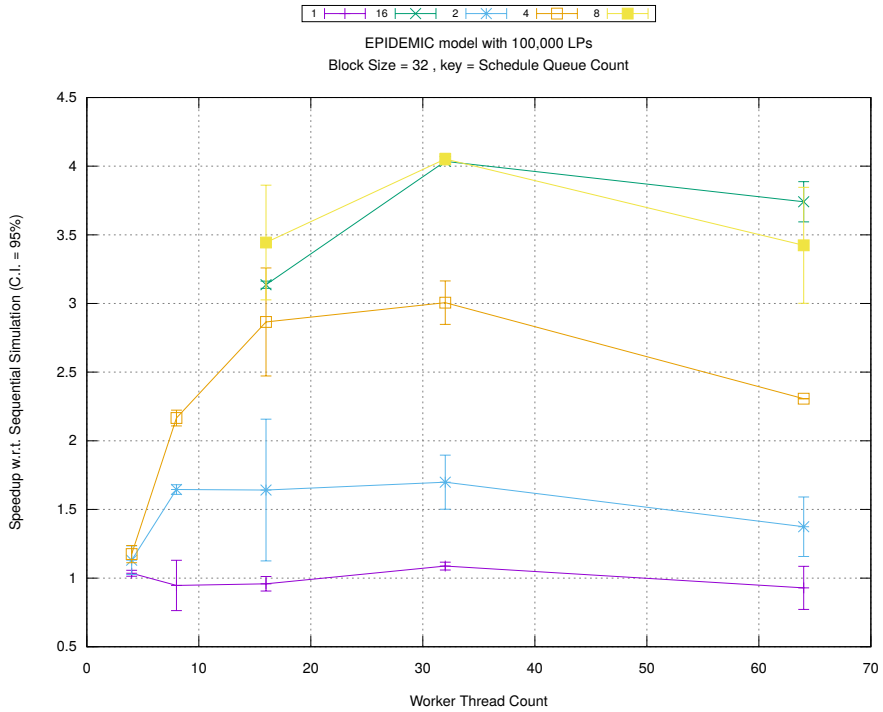


(b) Event Commitment Ratio

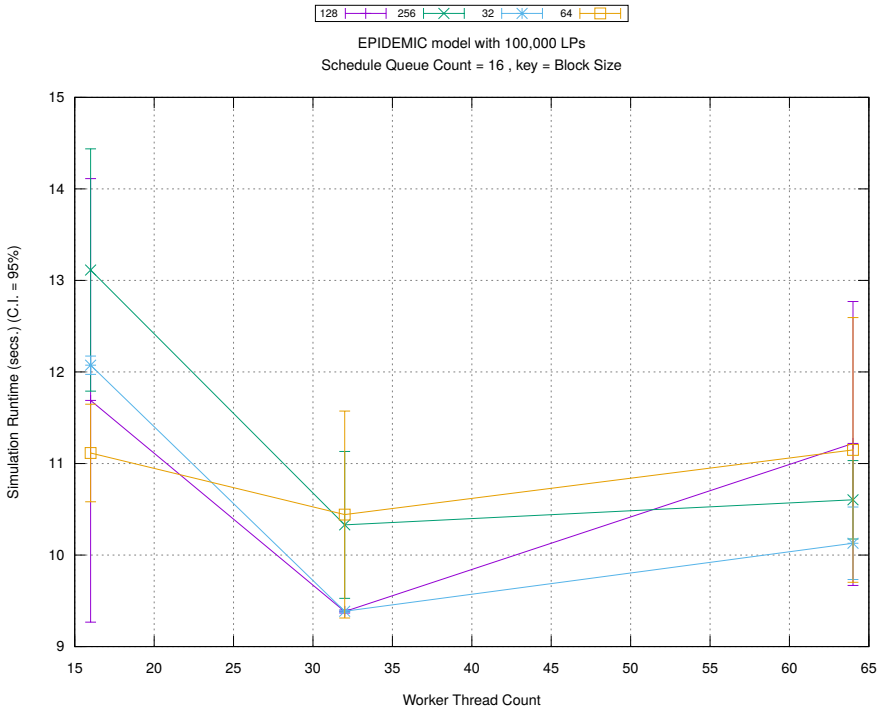


(c) Event Processing Rate (per second)

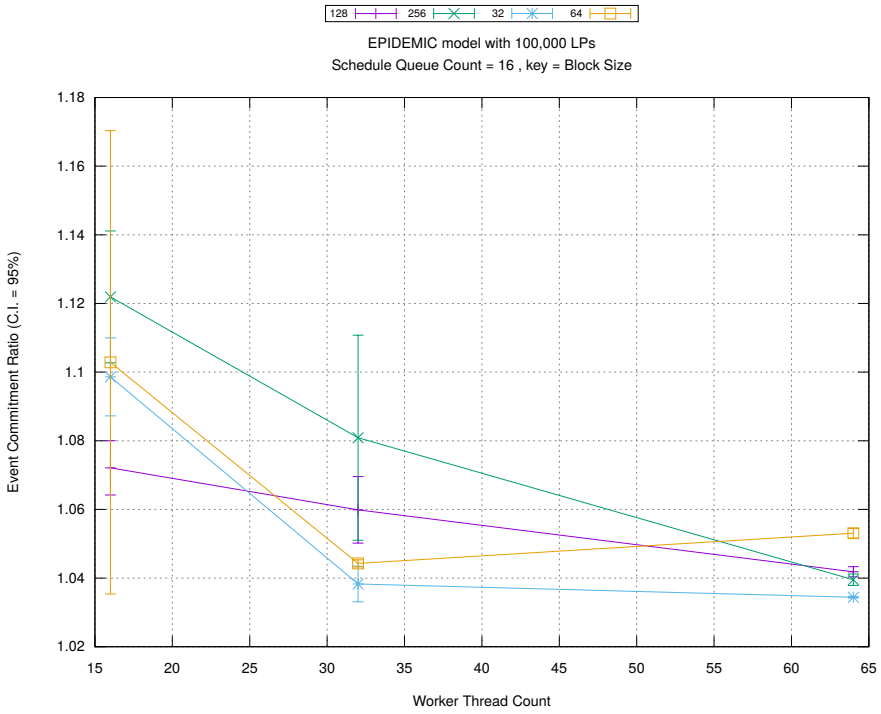
Figure A.200: epidemic 100k ba/plots/blocks/threads vs count key blocksize 32



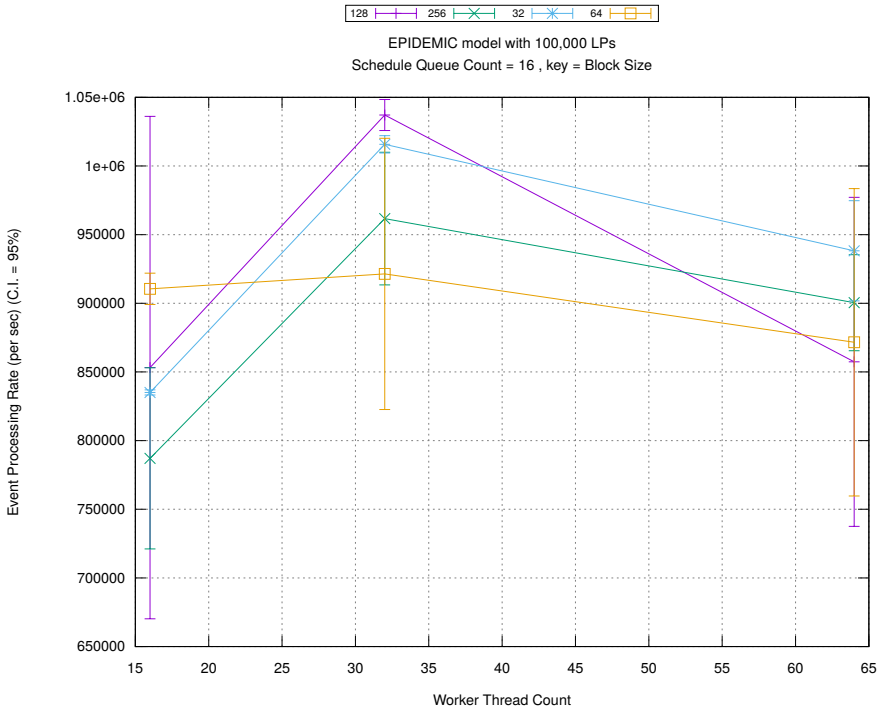
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

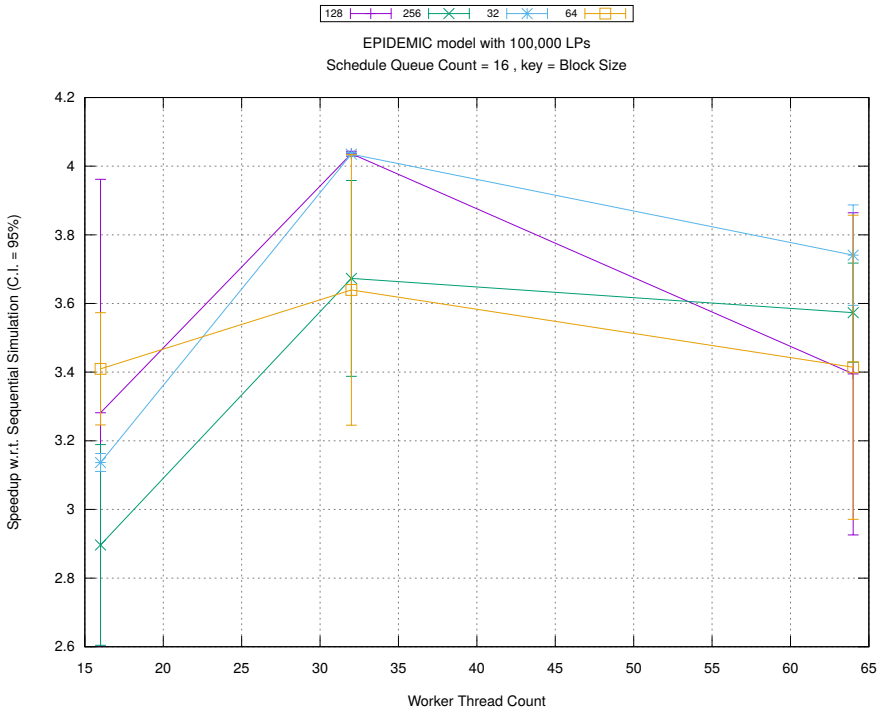


(b) Event Commitment Ratio

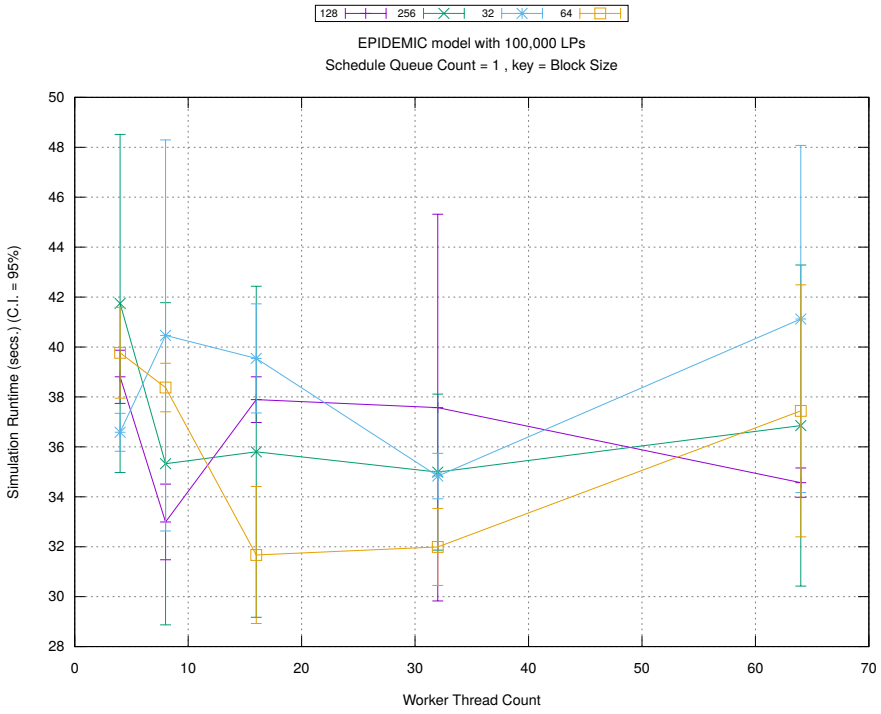


(c) Event Processing Rate (per second)

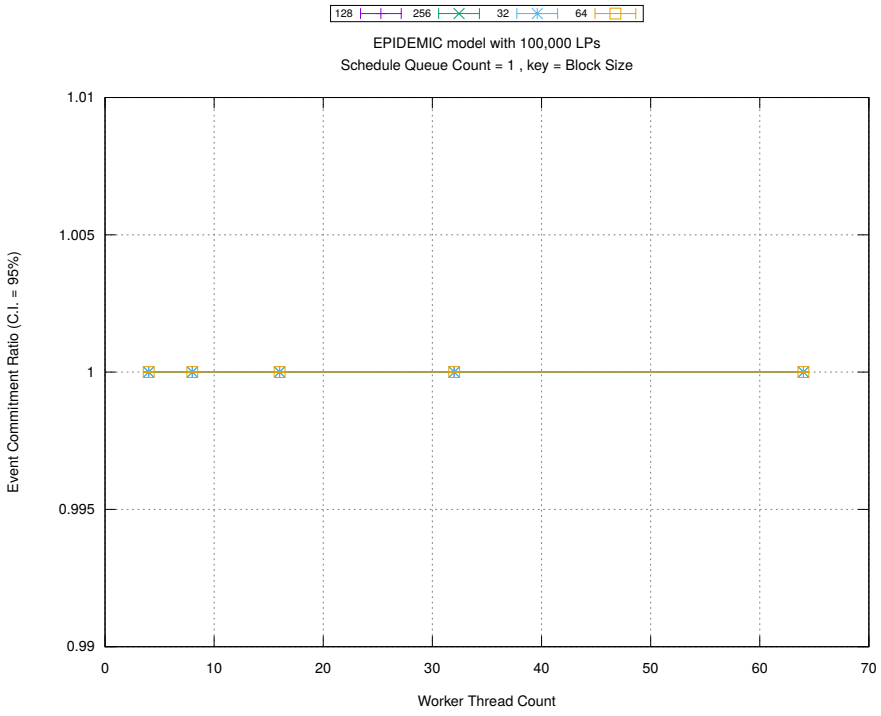
Figure A.201: epidemic 100k ba/plots/blocks/threads vs blocksize key count 16



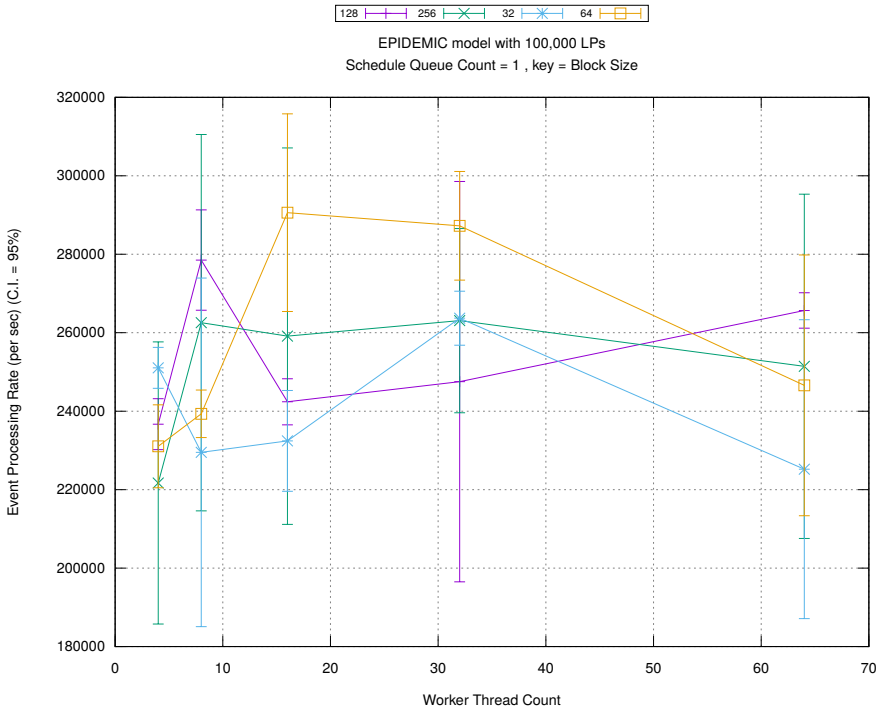
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

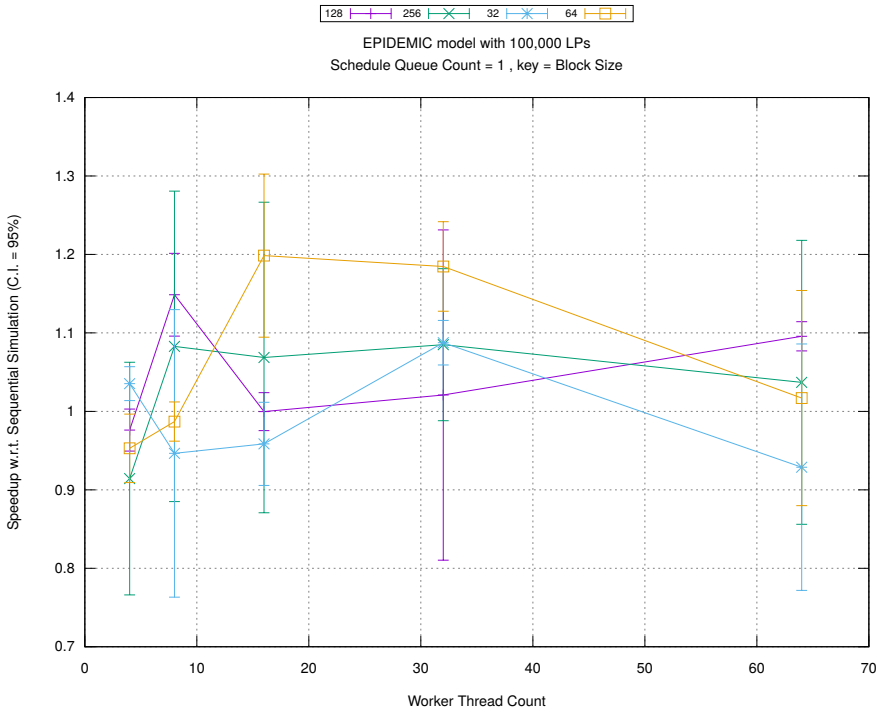


(b) Event Commitment Ratio

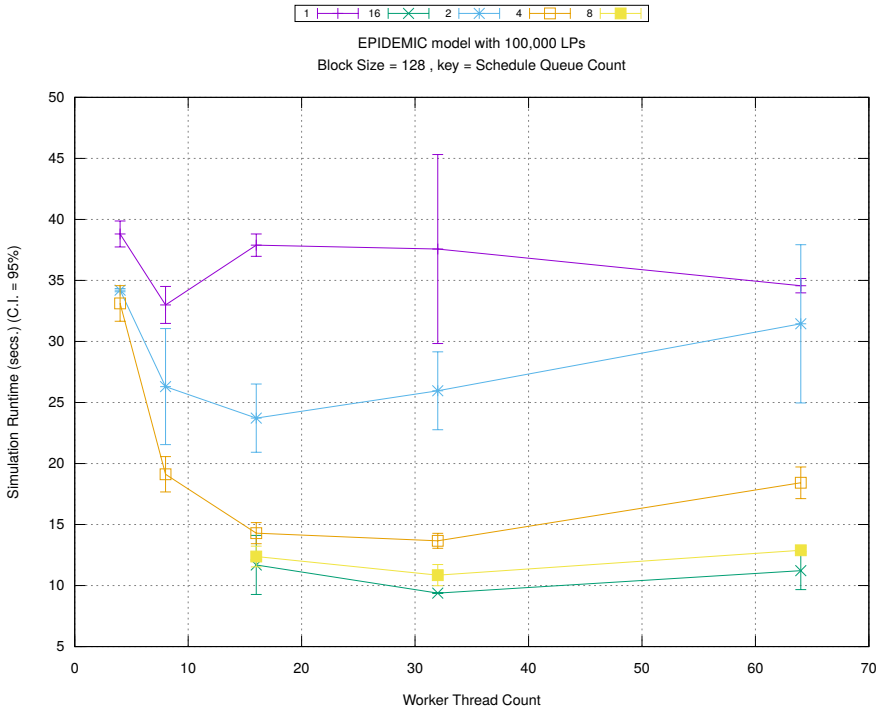


(c) Event Processing Rate (per second)

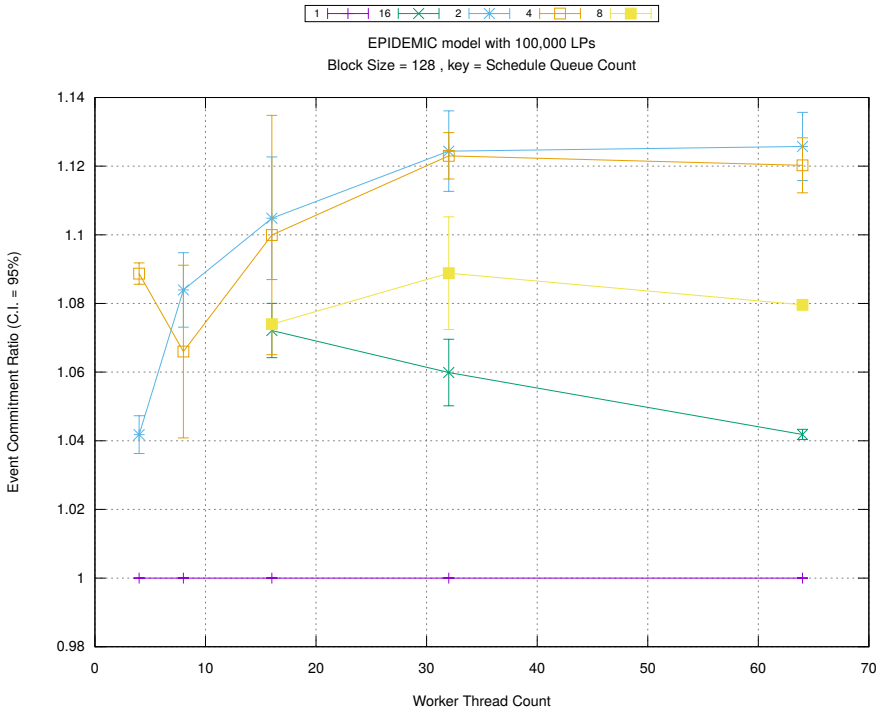
Figure A.202: epidemic 100k ba/plots/blocks/threads vs blocksize key count 1



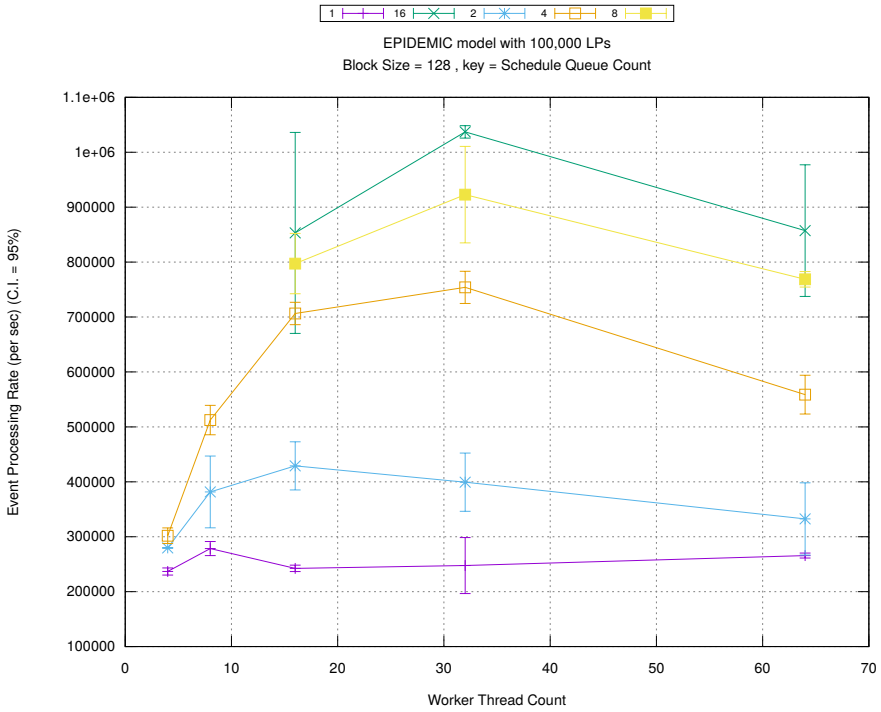
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

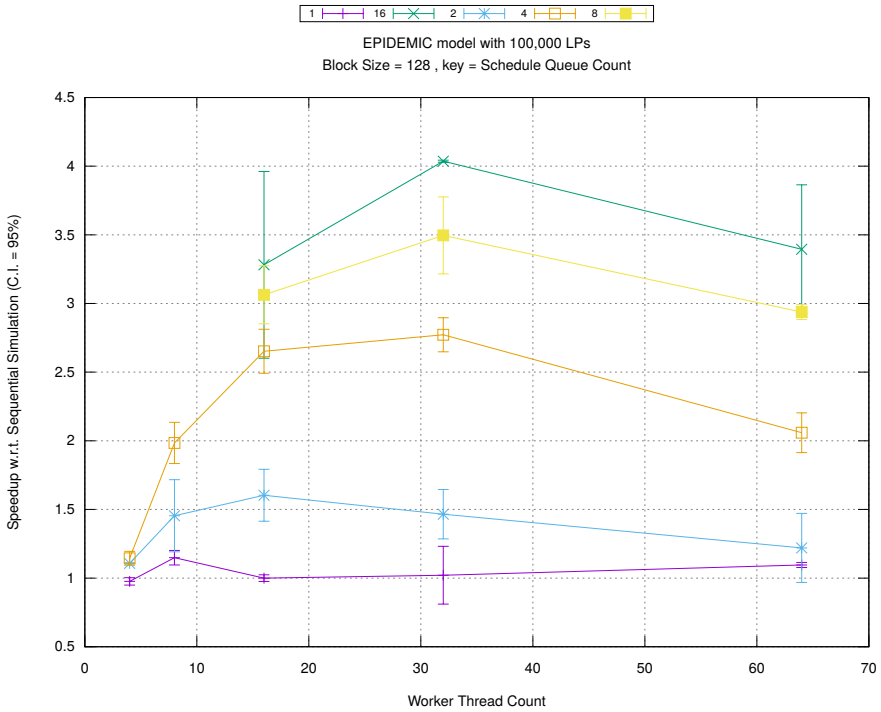


(b) Event Commitment Ratio

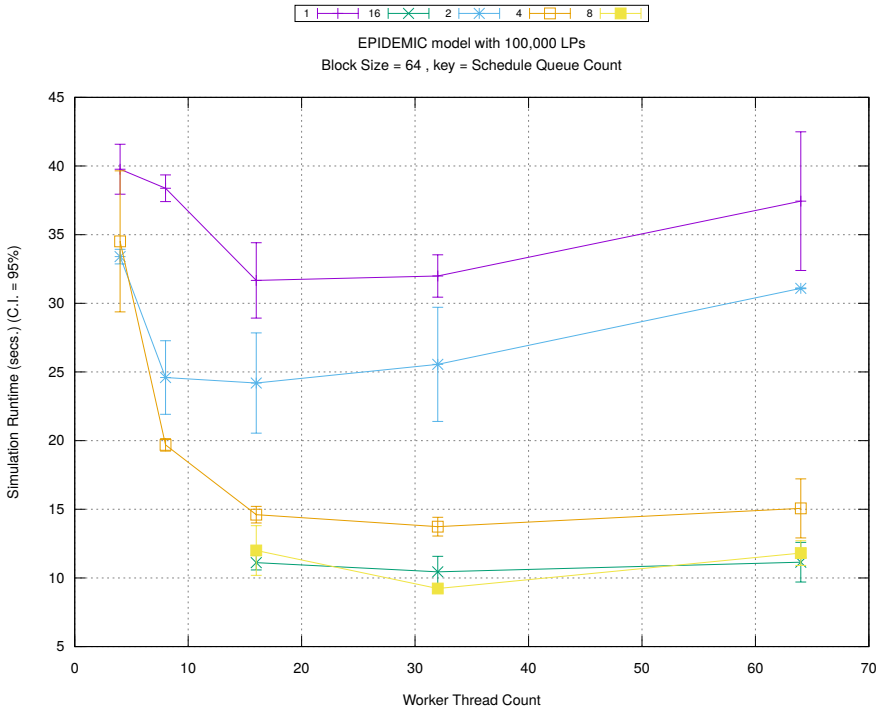


(c) Event Processing Rate (per second)

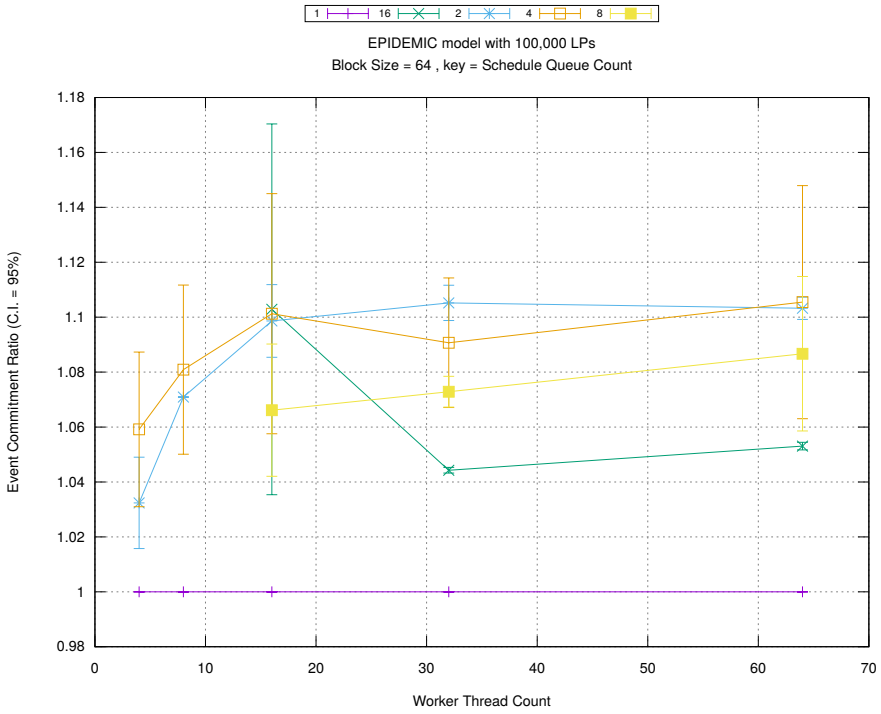
Figure A.203: epidemic 100k ba/plots/blocks/threads vs count key blocksize 128



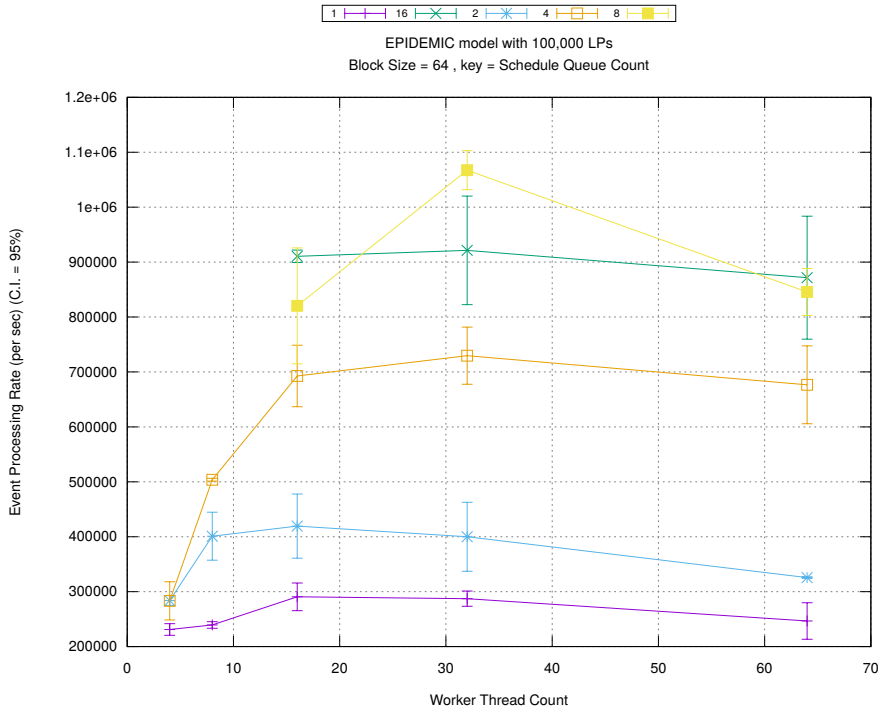
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

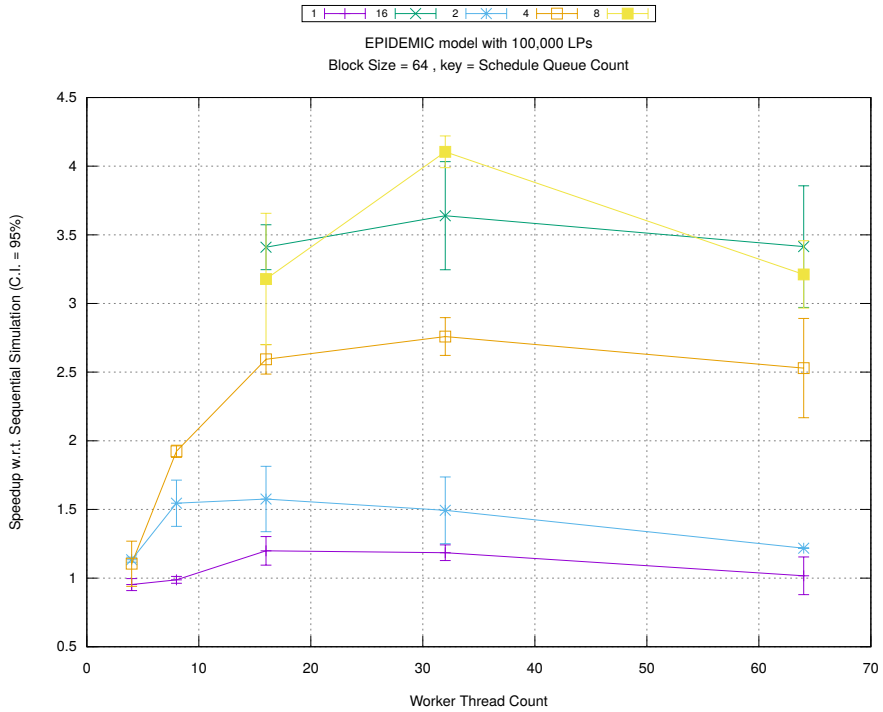


(b) Event Commitment Ratio



(c) Event Processing Rate (per second)

Figure A.204: epidemic 100k ba/plots/blocks/threads vs count key blocksize 64



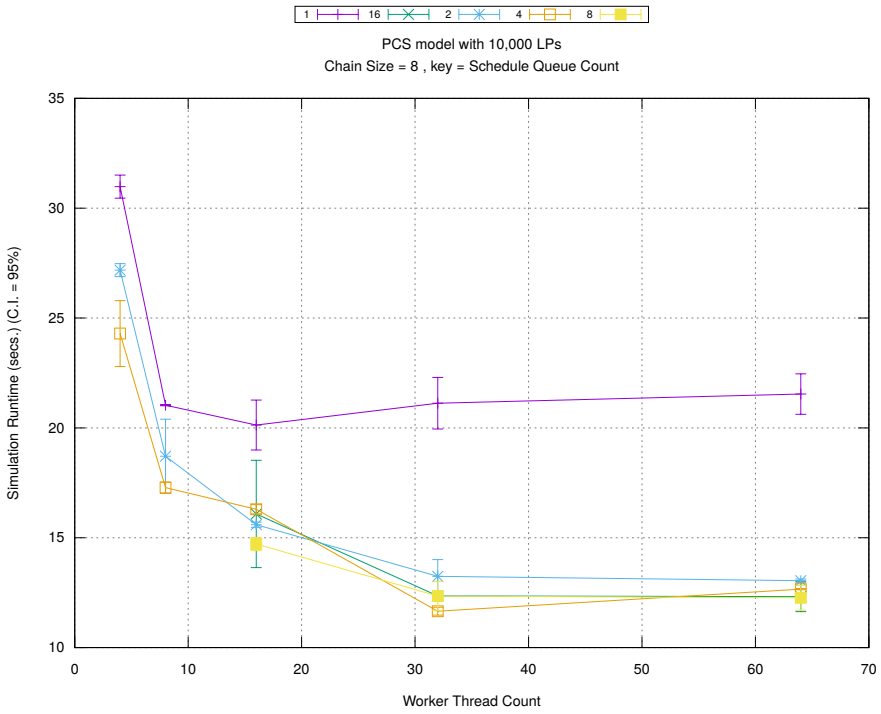
(d) Speedup w.r.t. Sequential Simulation

A.7 PCS Model with 10,000 LPs

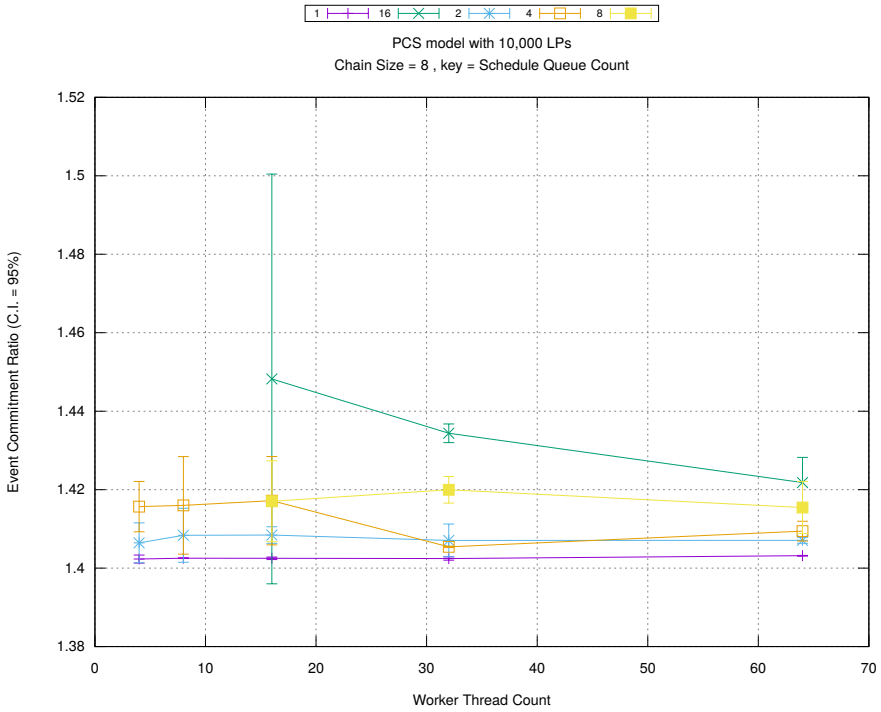
Table A.7 shows the configuration for this model.

Parameter	Values
Number of Intersections (or LPs)	10,000
Type of network connecting LPs	4-directional grid
Grid Size	100x100
Maximum number of channels	15
Mean call interval	200 timestamp units
Mean call duration	50 timestamp units
Mean move interval	100 timestamp units
Total number of portables	500,000
Simulation Time	500 timestamp units
Sequential Simulation Time for calculating modularity	350 timestamp units

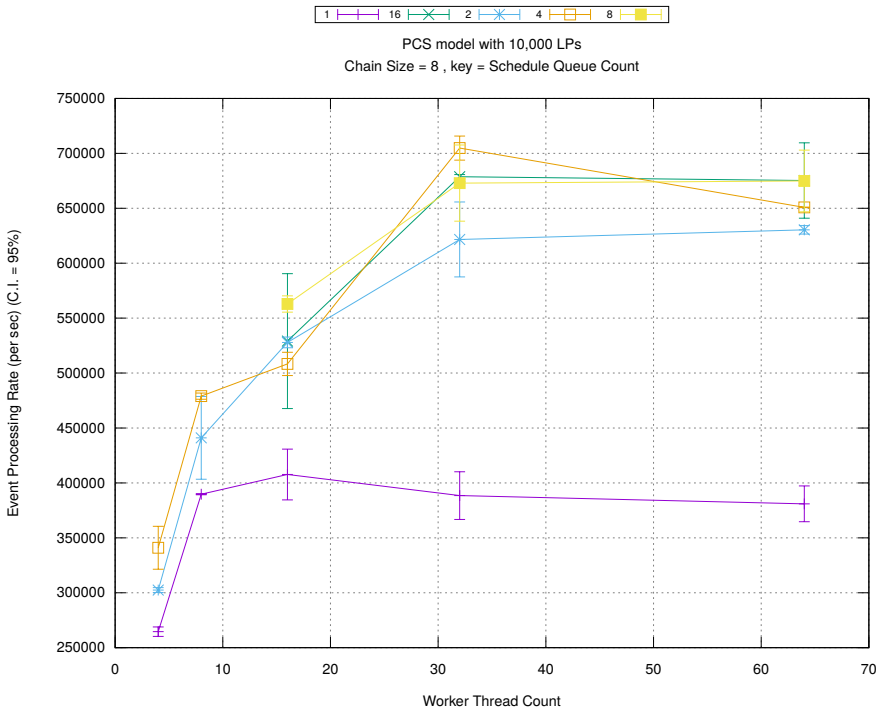
Table A.7: PCS MODEL setup



(a) Simulation Runtime (in seconds)

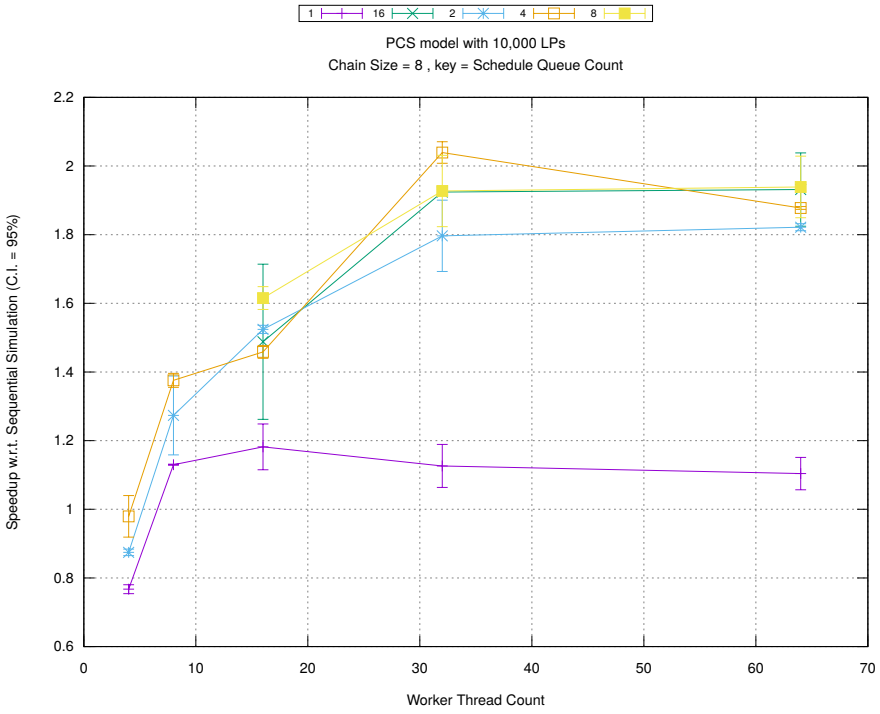


(b) Event Commitment Ratio

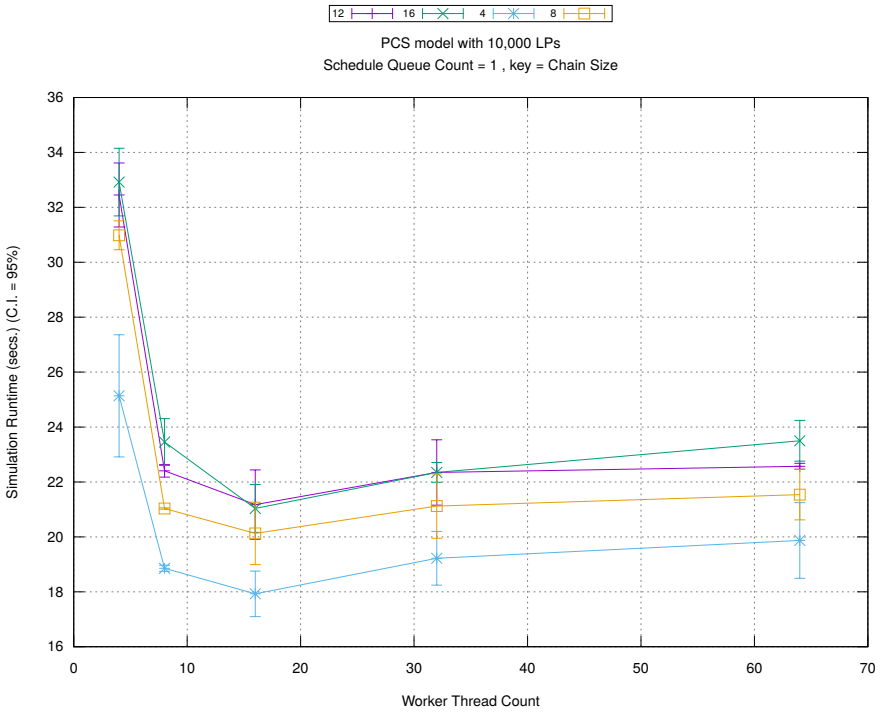


(c) Event Processing Rate (per second)

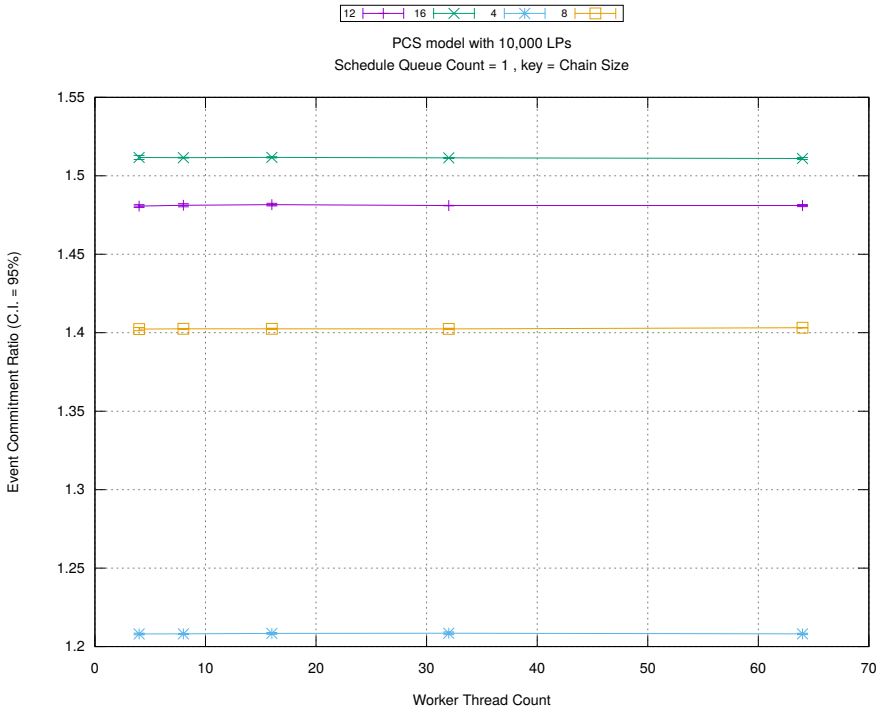
Figure A.205: pcs 10k/plots/chains/threads vs count key chainsize 8



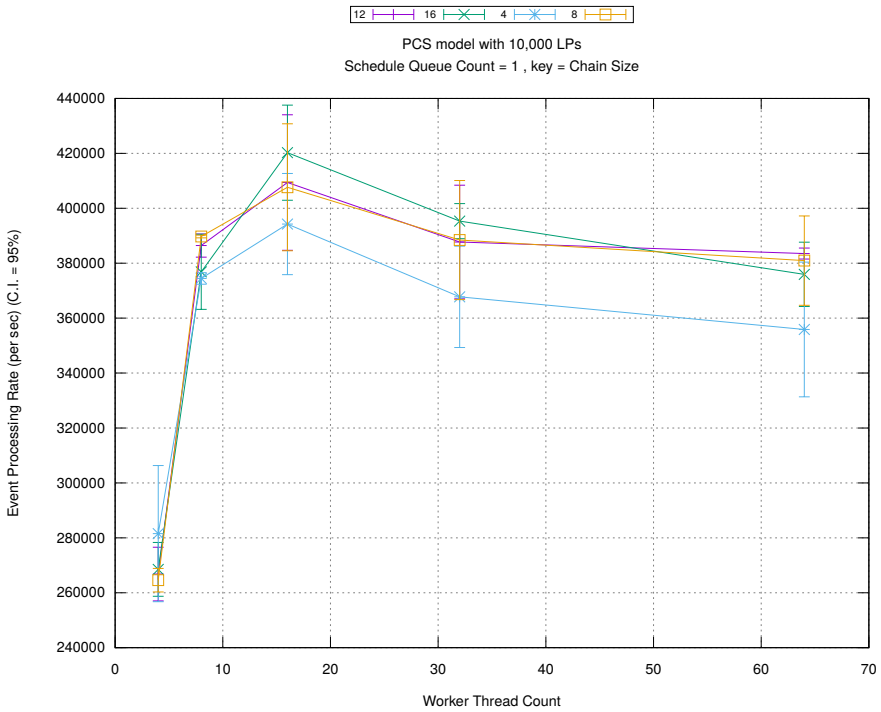
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

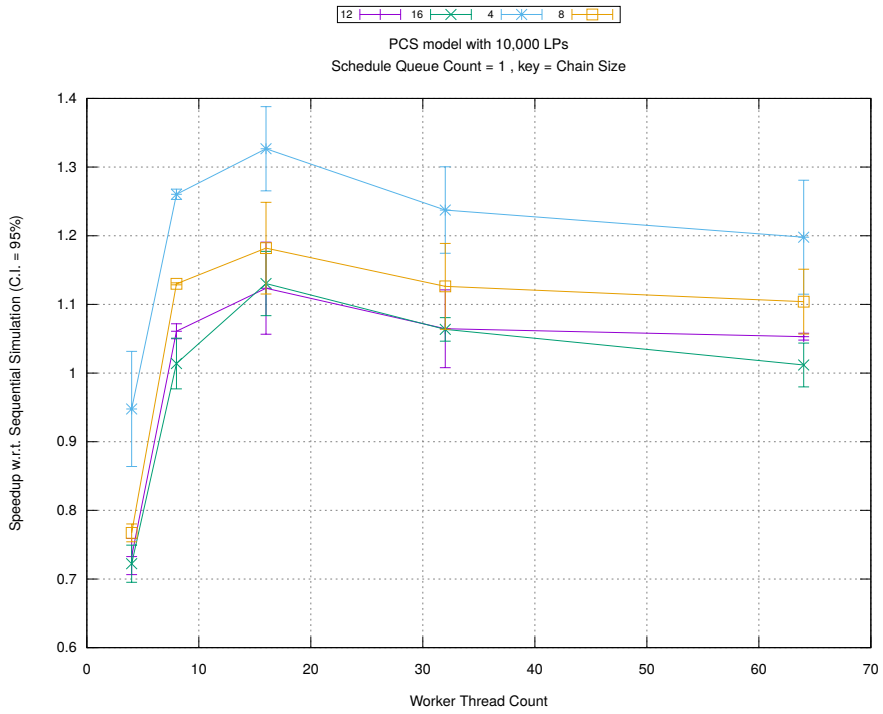


(b) Event Commitment Ratio

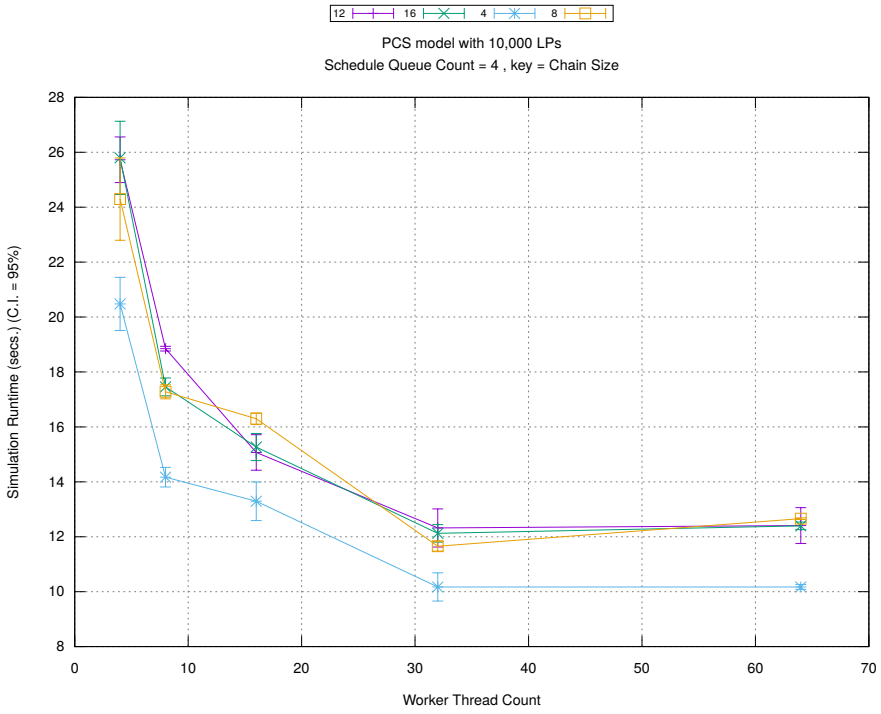


(c) Event Processing Rate (per second)

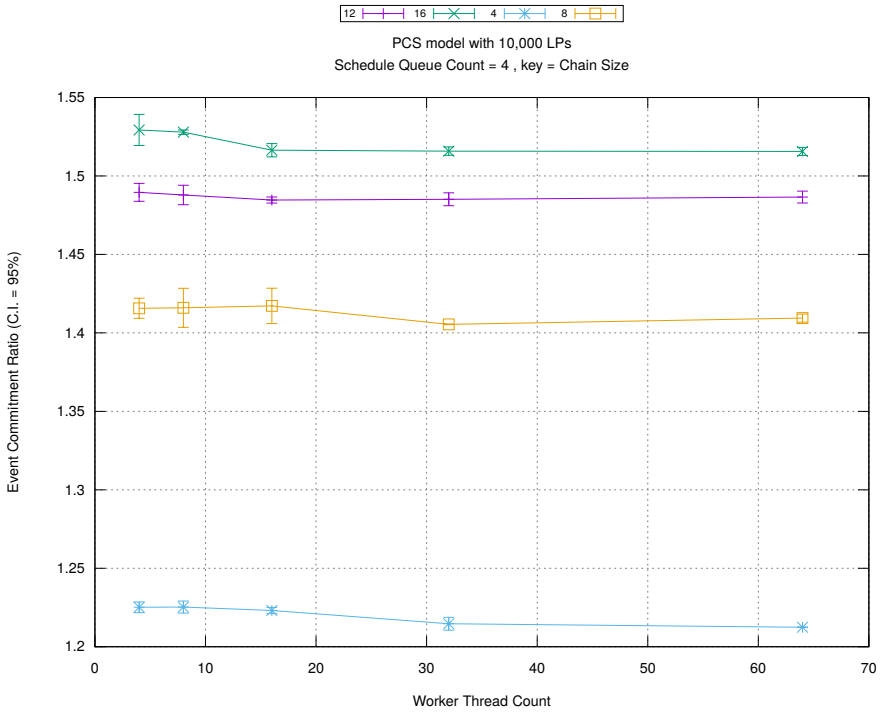
Figure A.206: pcs 10k/plots/chains/threads vs chainsize key count 1



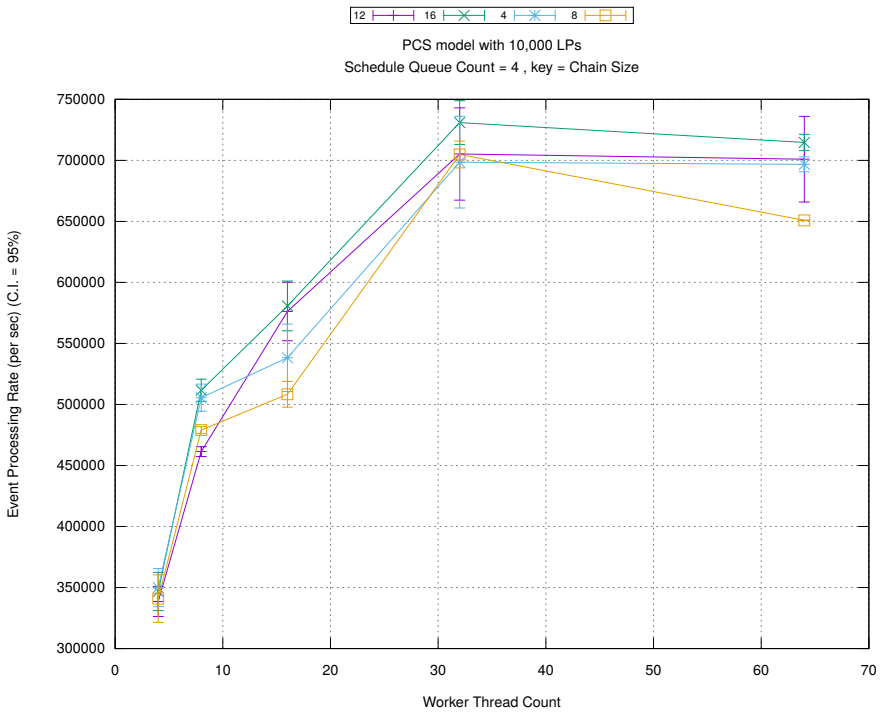
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

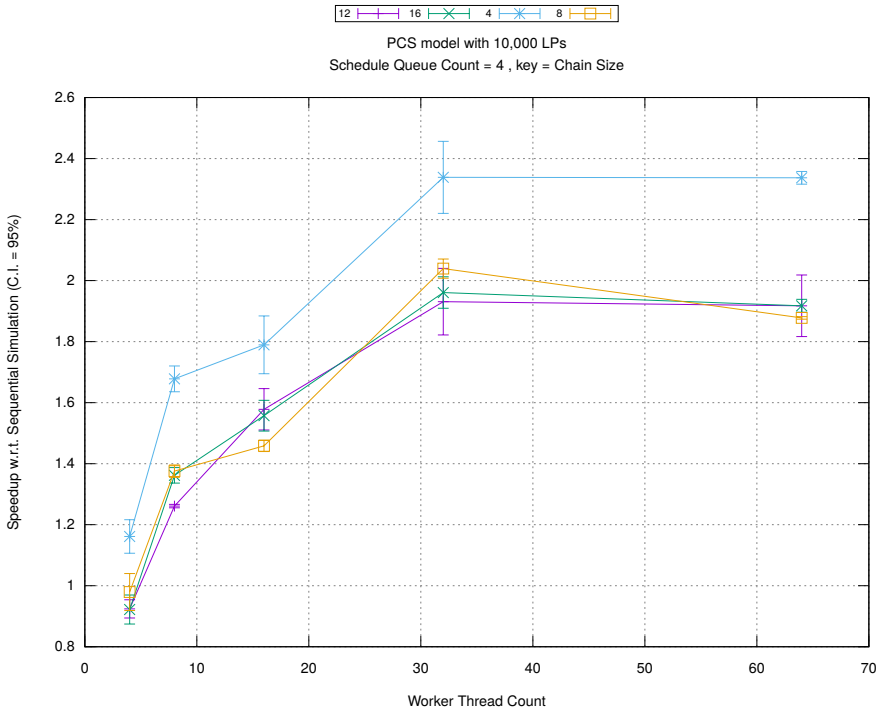


(b) Event Commitment Ratio

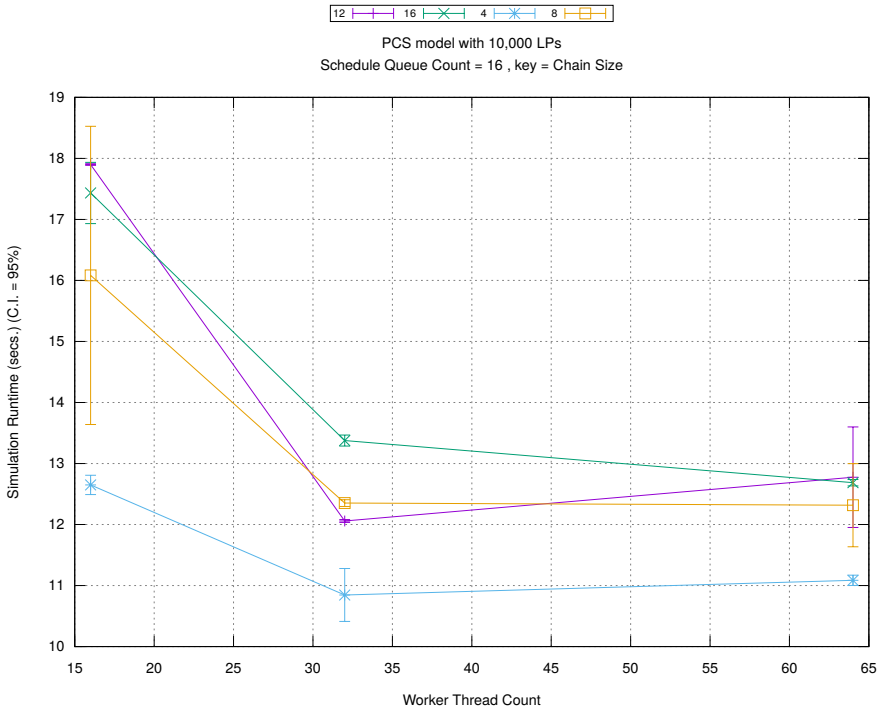


(c) Event Processing Rate (per second)

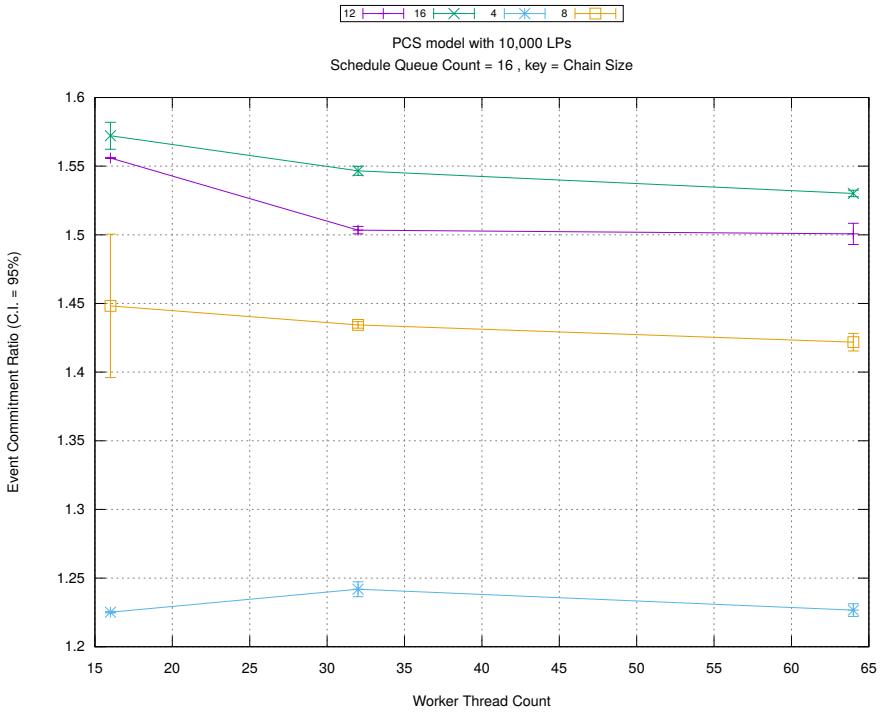
Figure A.207: pcs 10k/plots/chains/threads vs chainsize key count 4



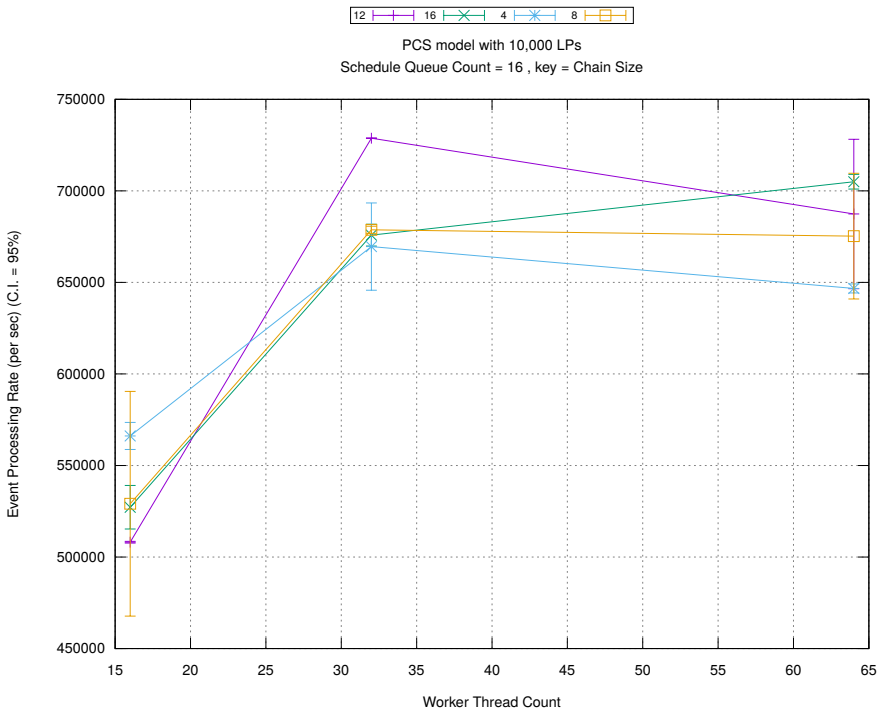
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

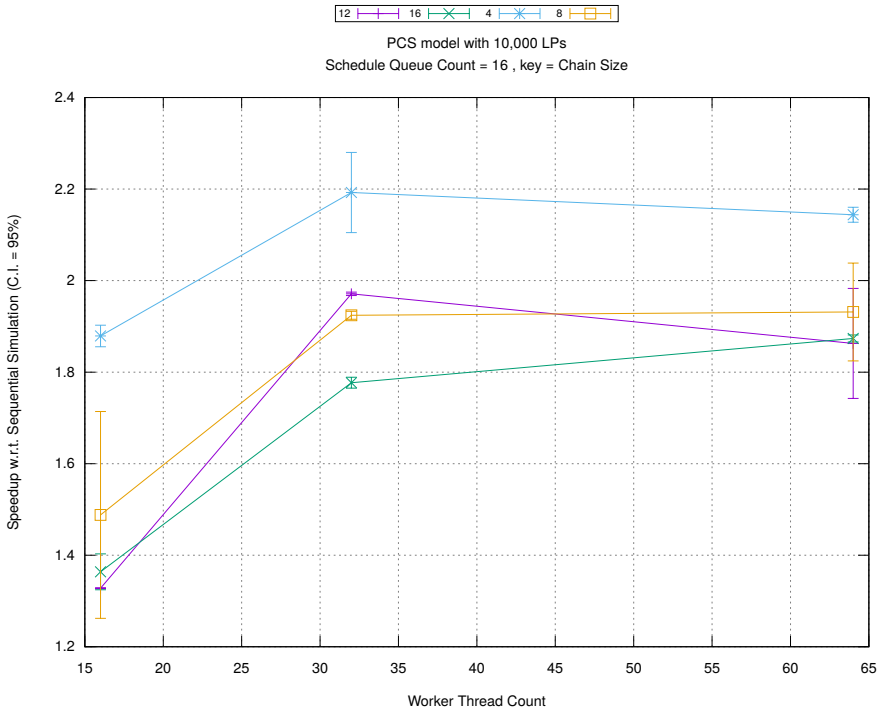


(b) Event Commitment Ratio

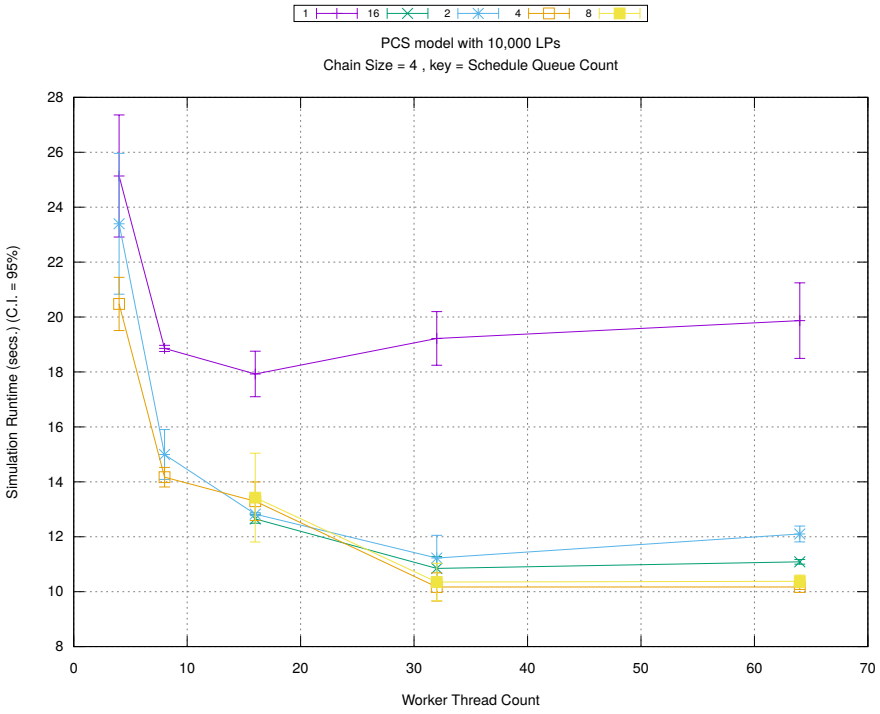


(c) Event Processing Rate (per second)

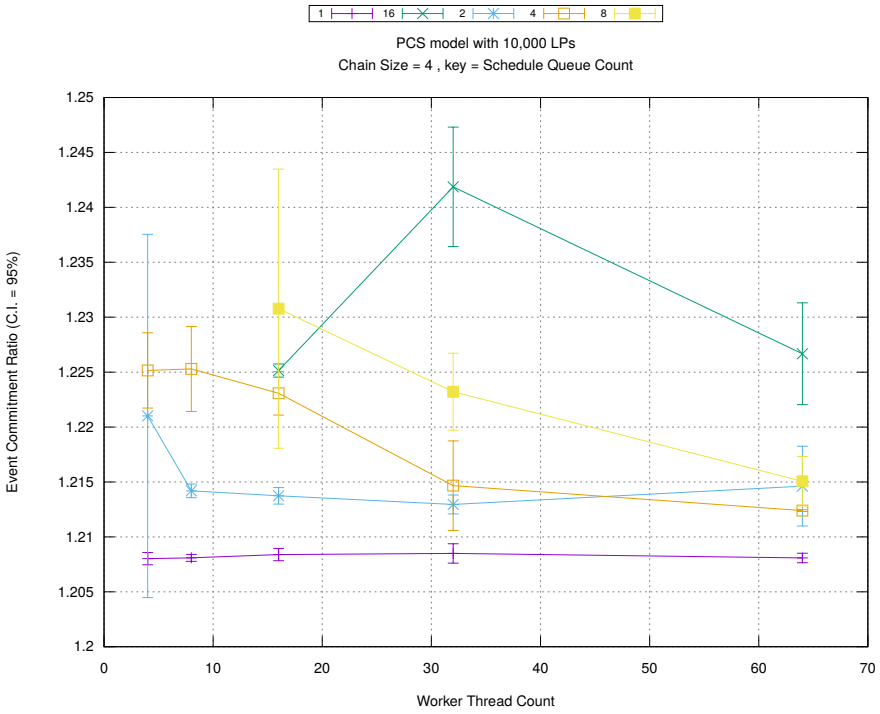
Figure A.208: pcs 10k/plots/chains/threads vs chainsize key count 16



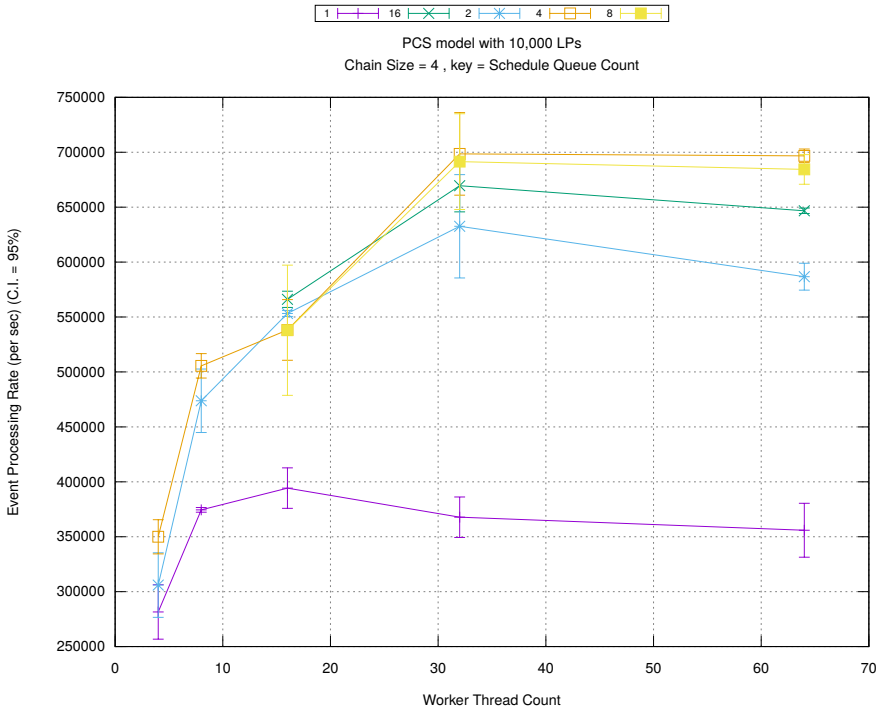
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

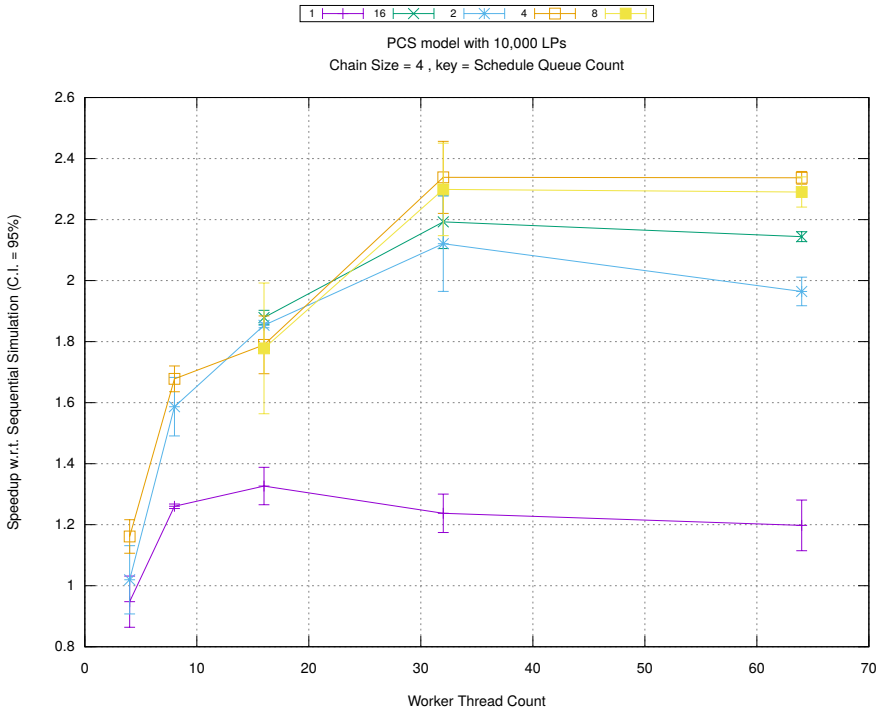


(b) Event Commitment Ratio

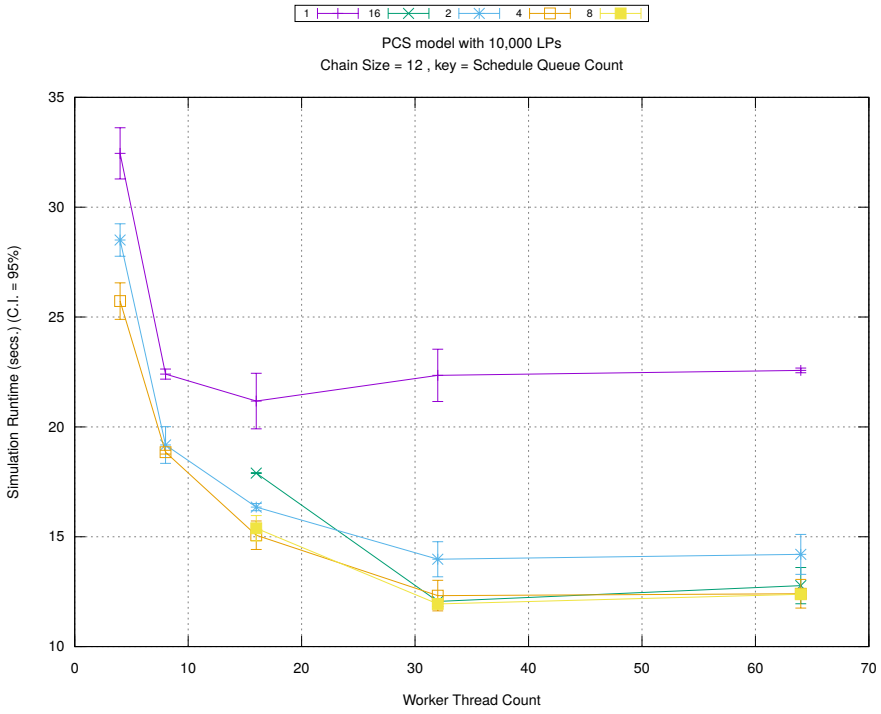


(c) Event Processing Rate (per second)

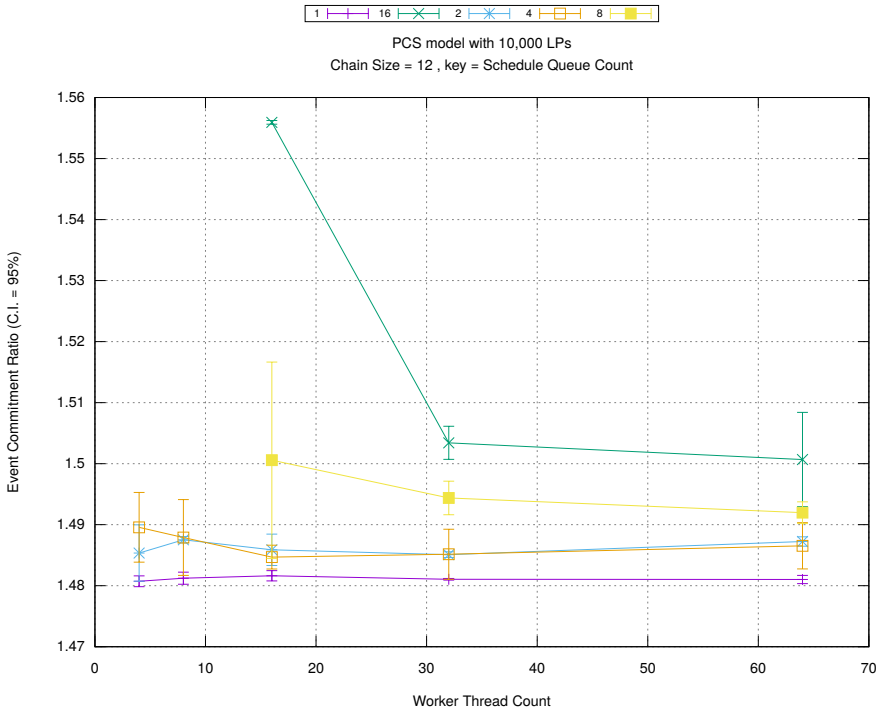
Figure A.209: pcs 10k/plots/chains/threads vs count key chainsize 4



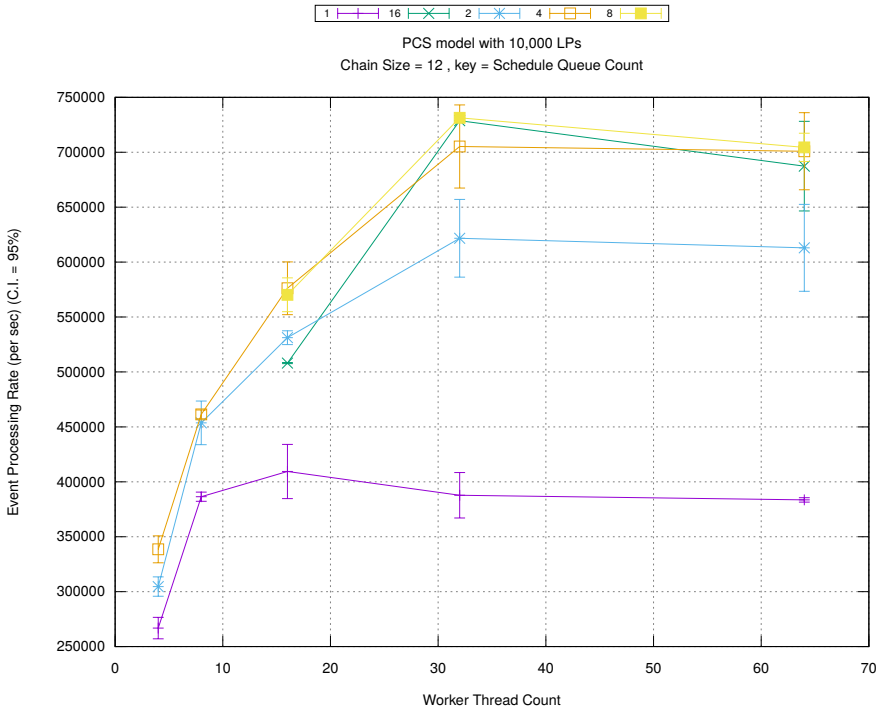
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

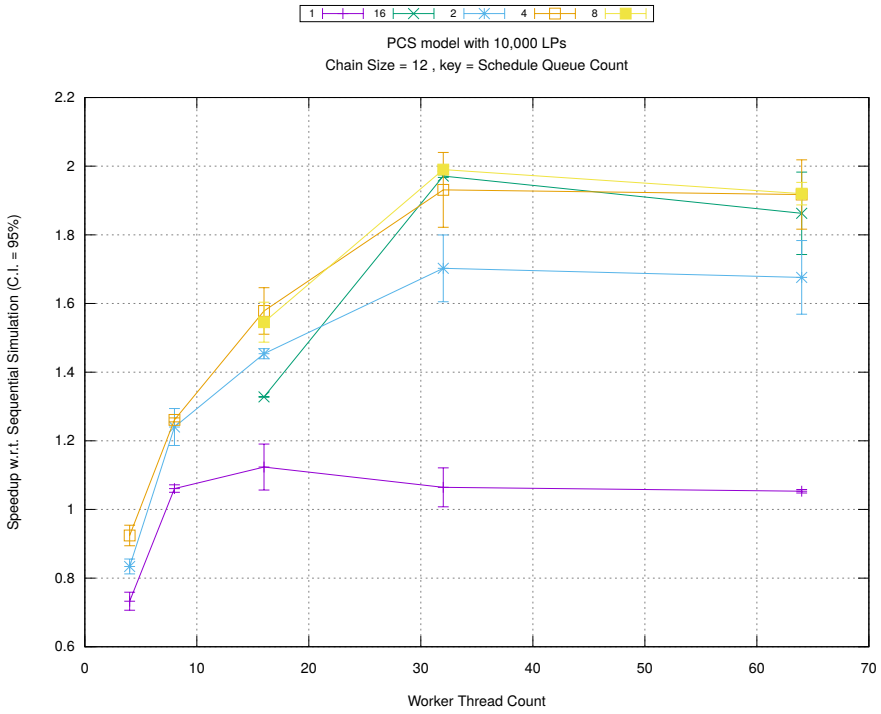


(b) Event Commitment Ratio

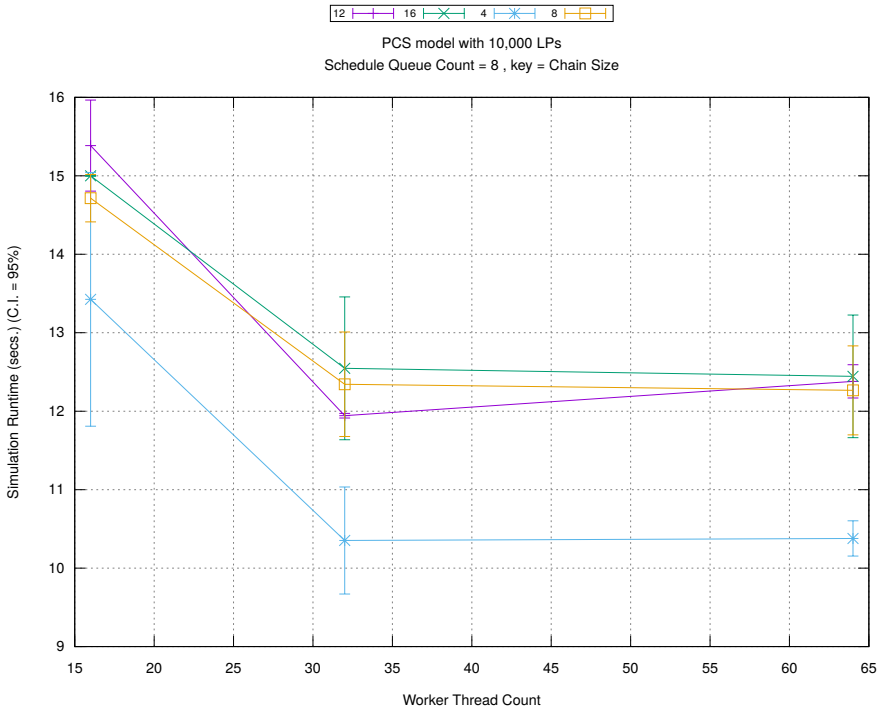


(c) Event Processing Rate (per second)

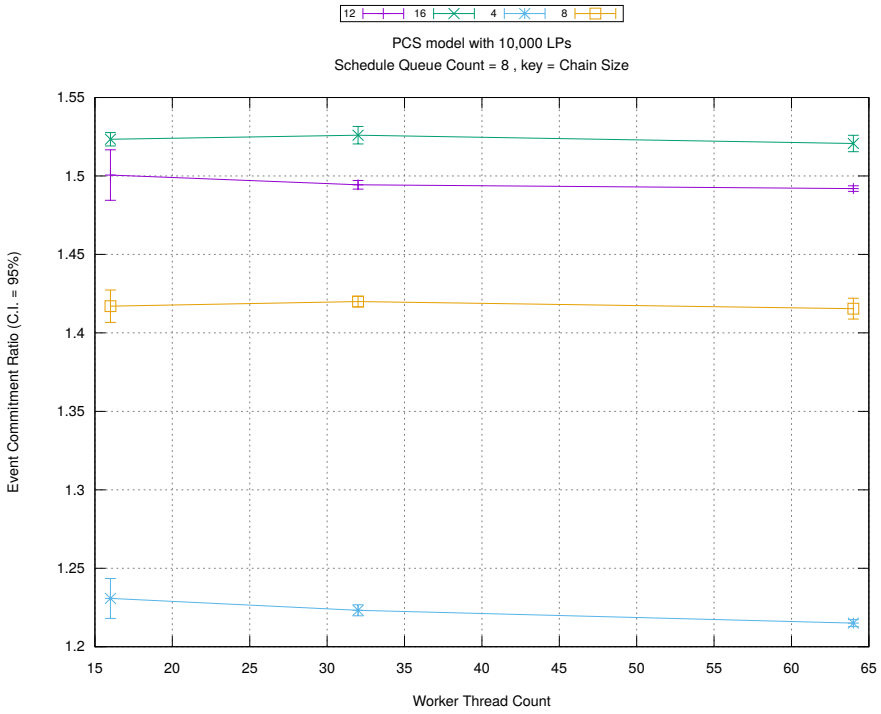
Figure A.210: pcs 10k/plots/chains/threads vs count key chainsize 12



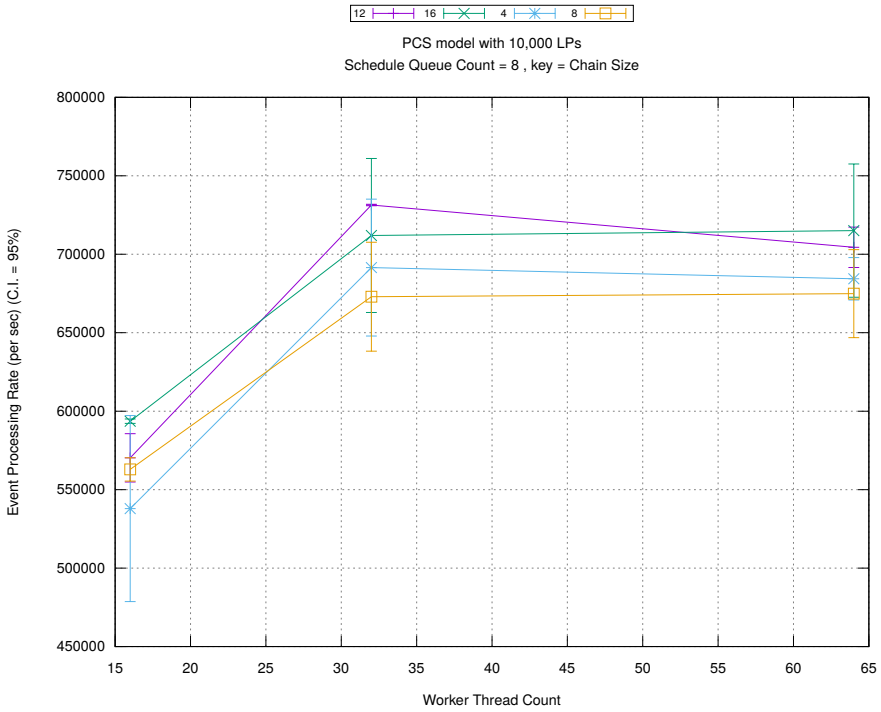
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

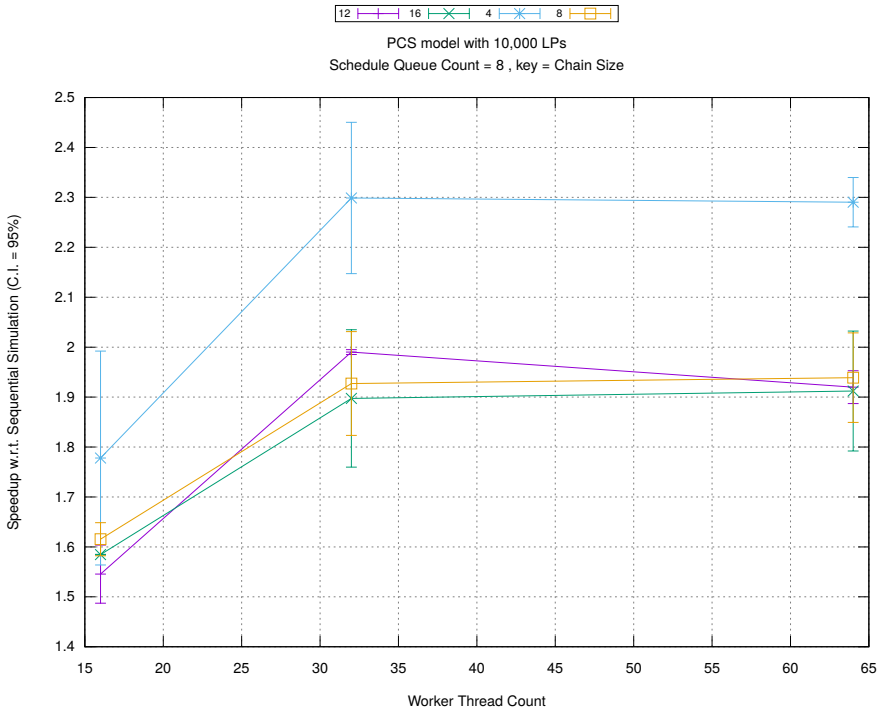


(b) Event Commitment Ratio

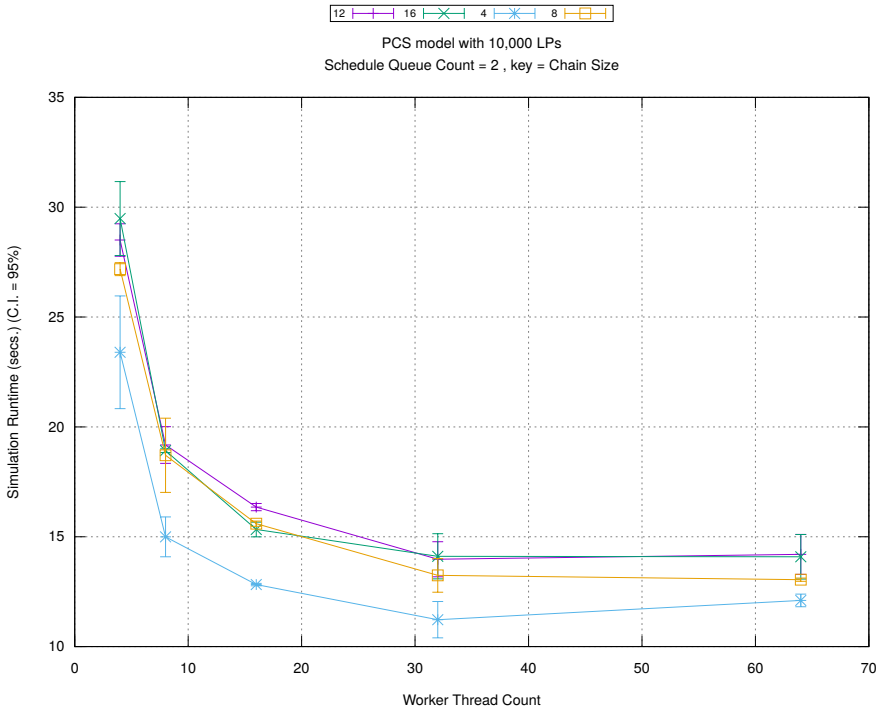


(c) Event Processing Rate (per second)

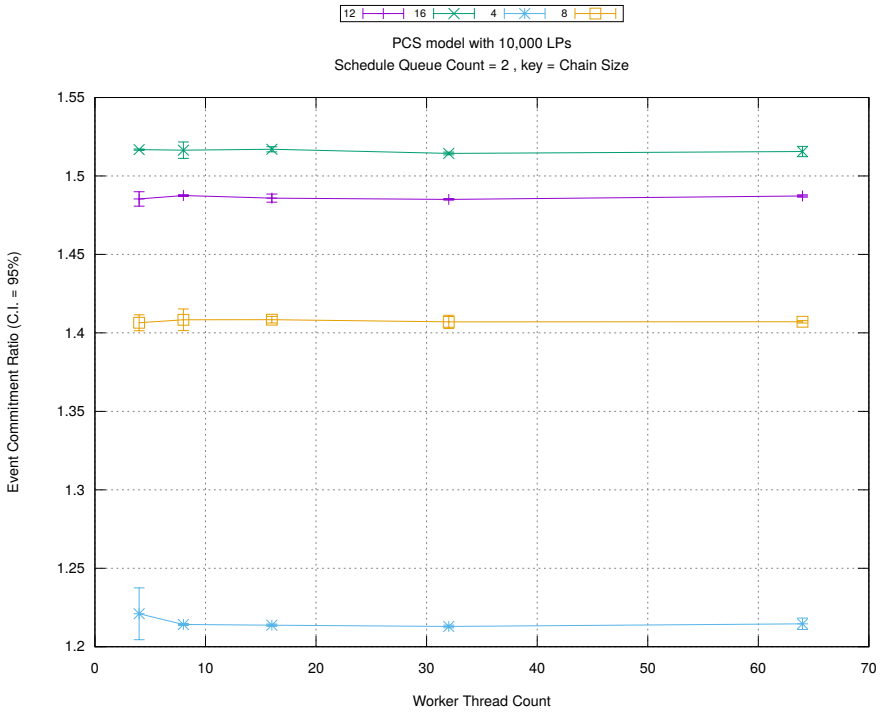
Figure A.211: pcs 10k/plots/chains/threads vs chainsize key count 8



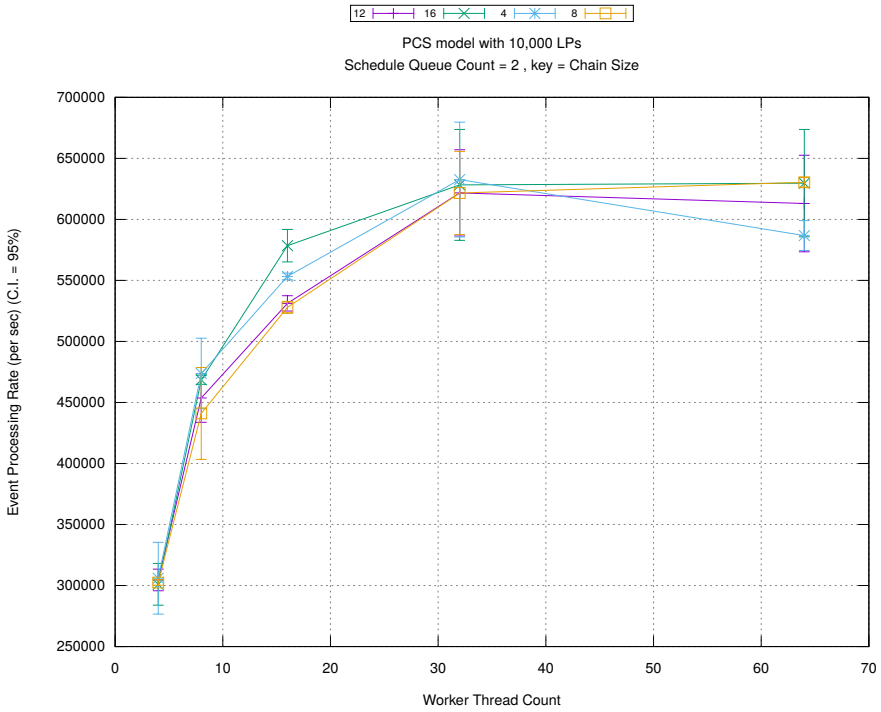
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

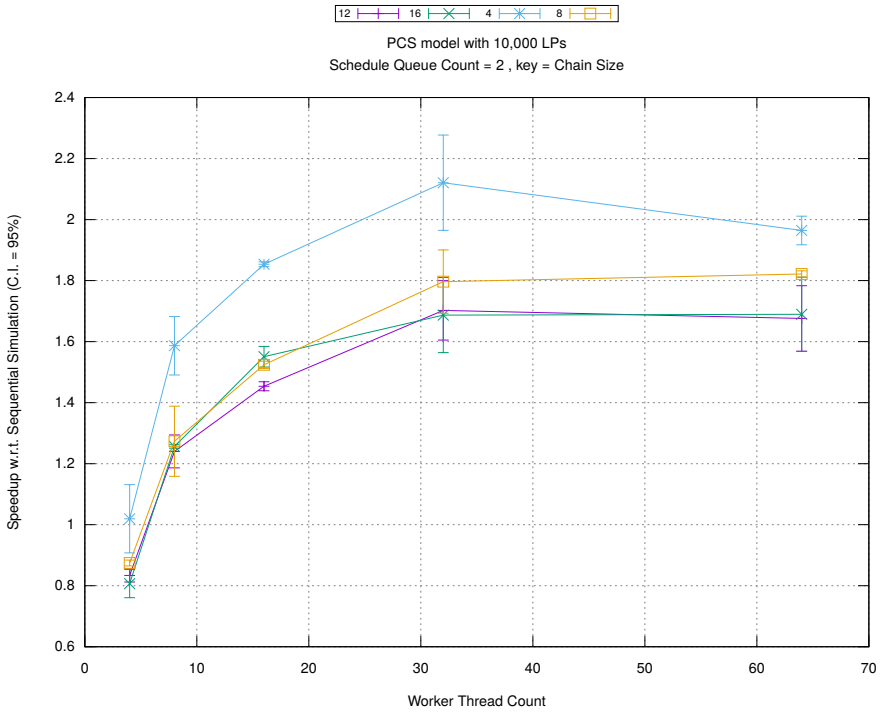


(b) Event Commitment Ratio

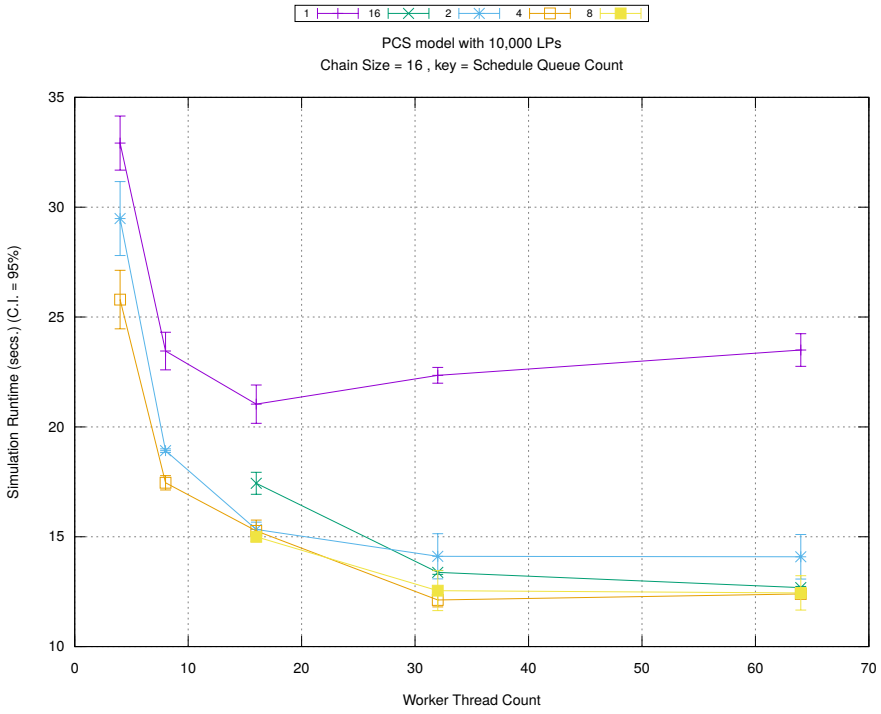


(c) Event Processing Rate (per second)

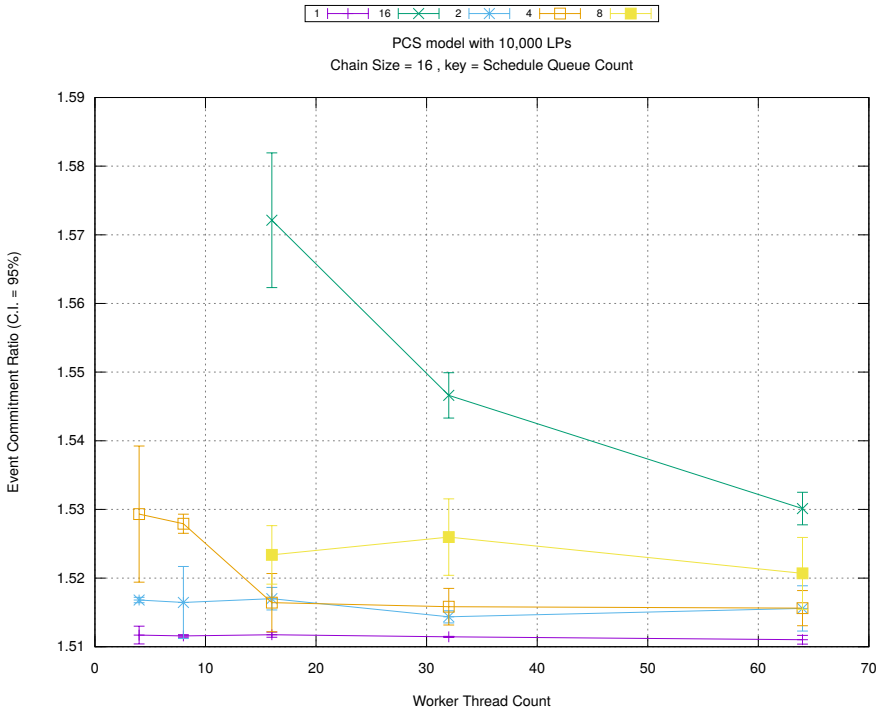
Figure A.212: pcs 10k/plots/chains/threads vs chainsize key count 2



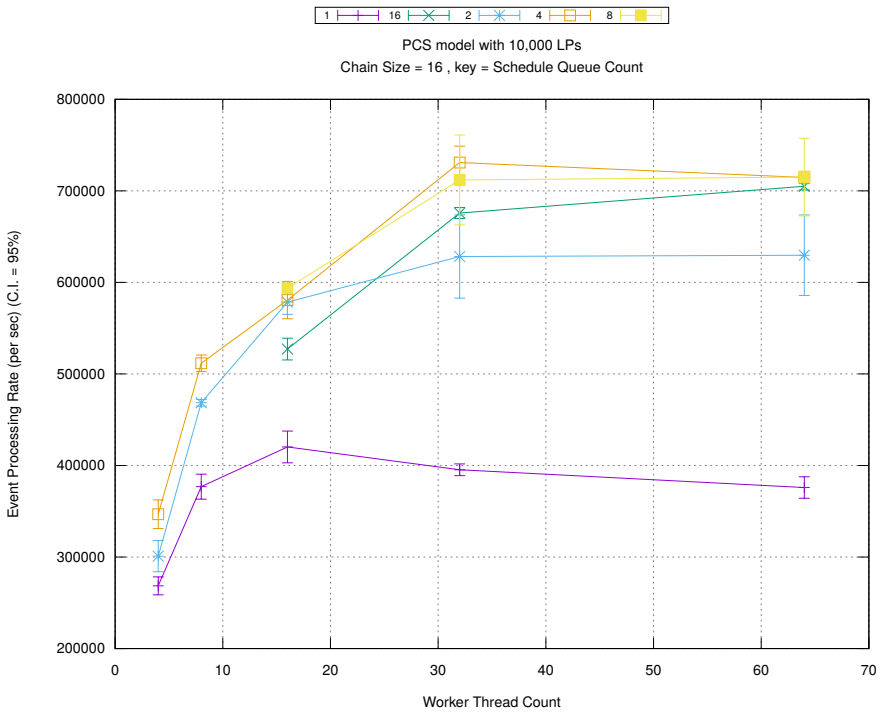
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

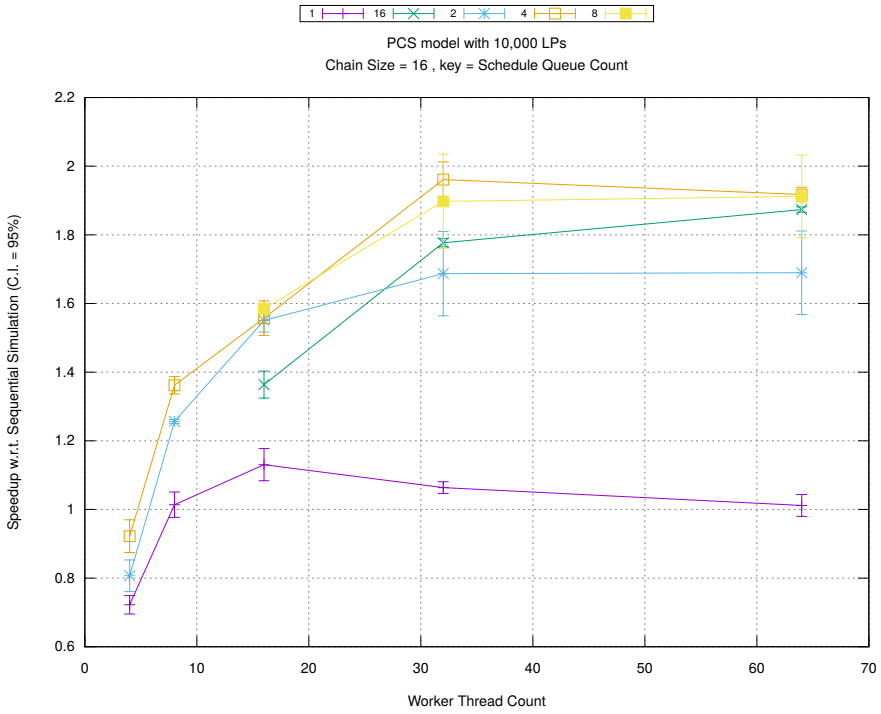


(b) Event Commitment Ratio

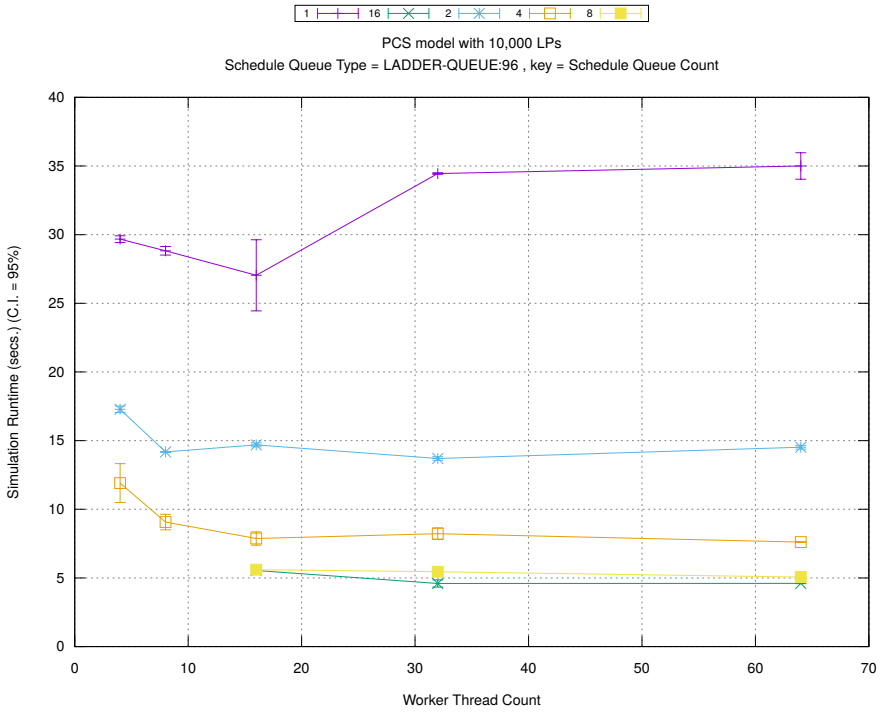


(c) Event Processing Rate (per second)

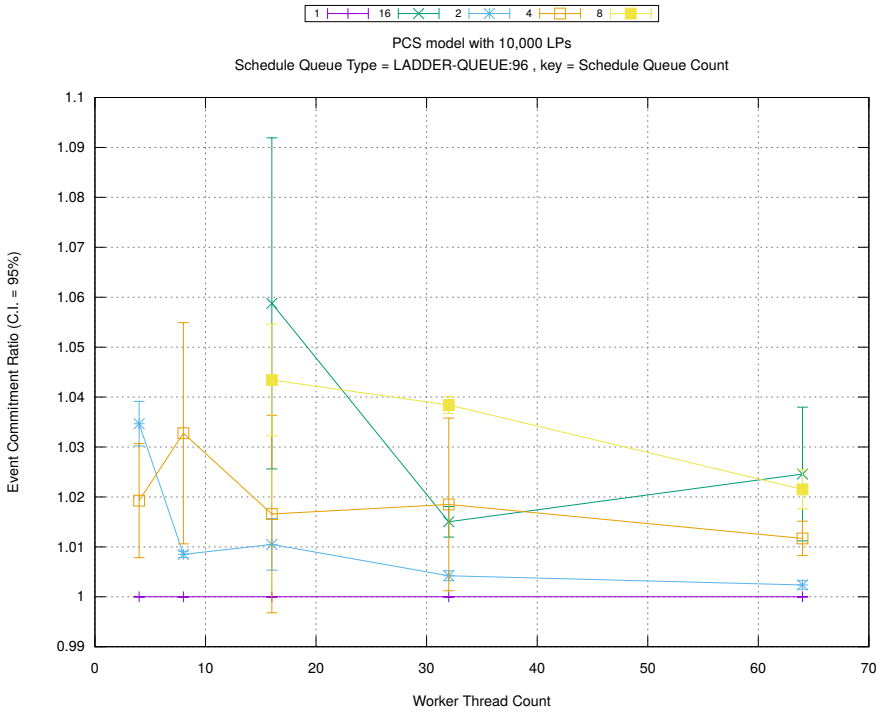
Figure A.213: pcs 10k/plots/chains/threads vs count key chainsize 16



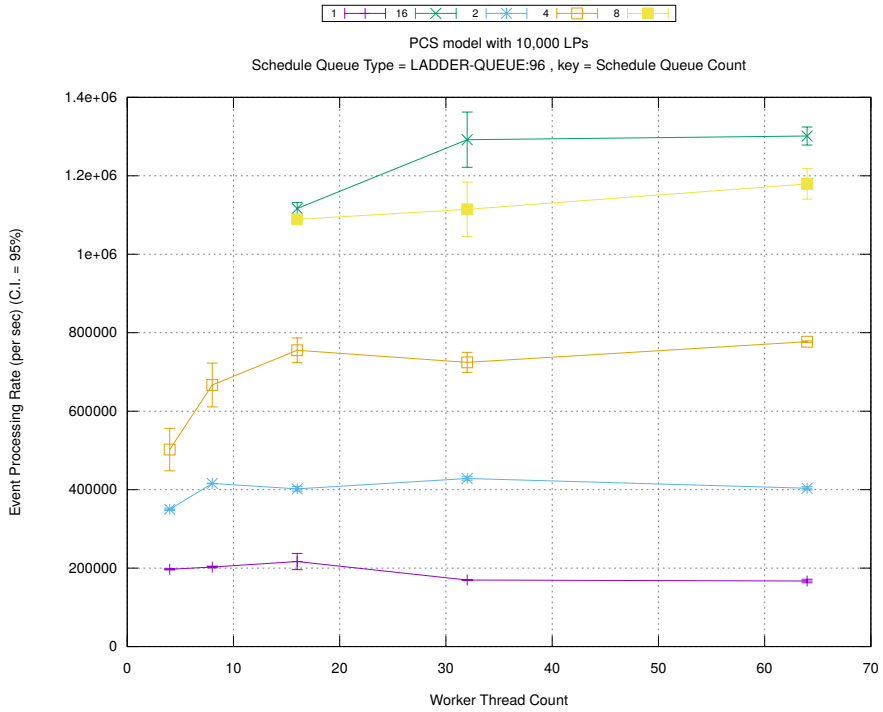
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

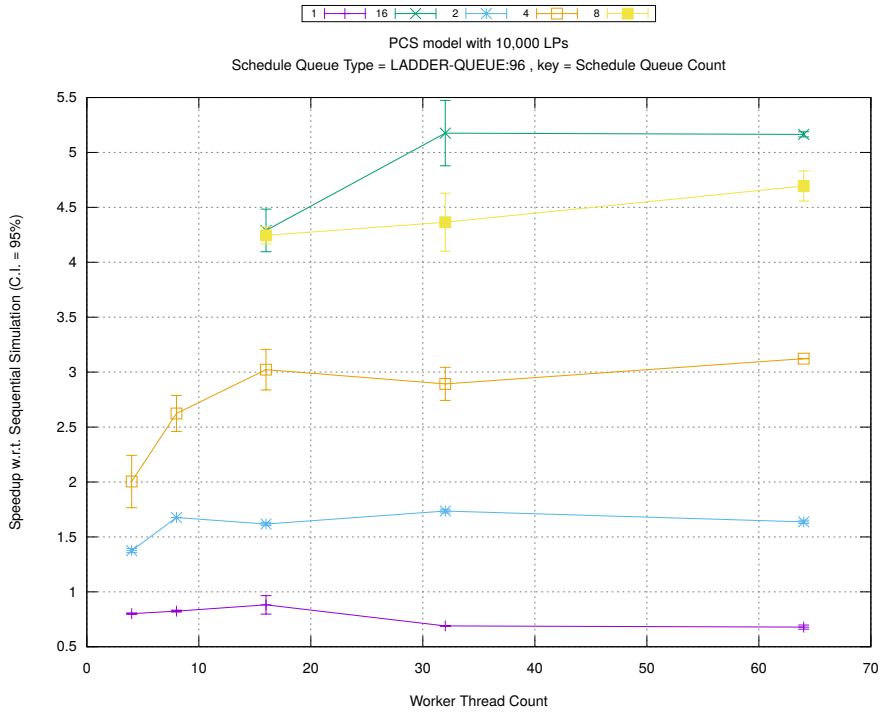


(b) Event Commitment Ratio

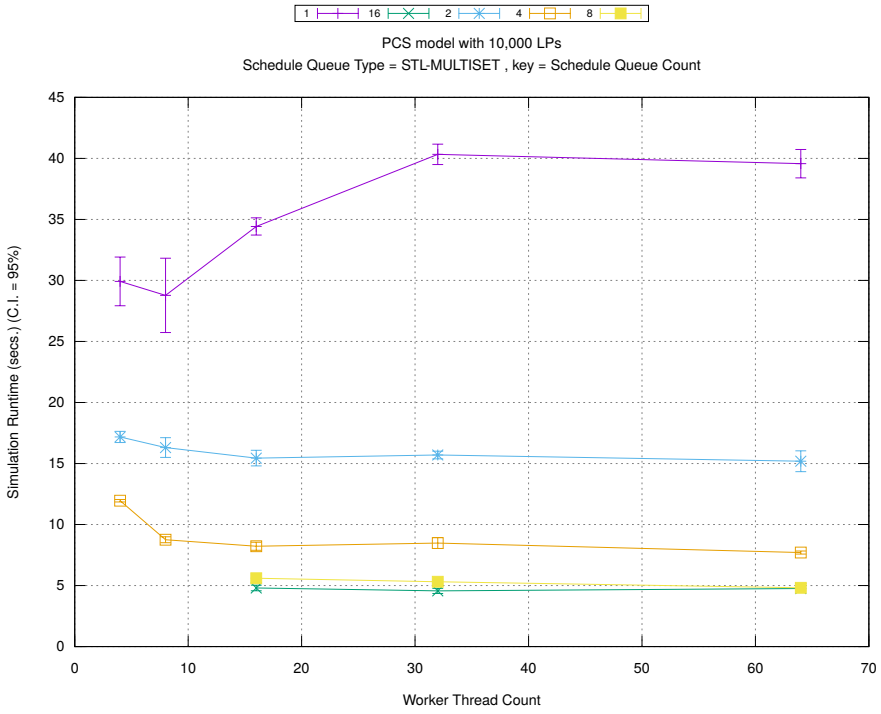


(c) Event Processing Rate (per second)

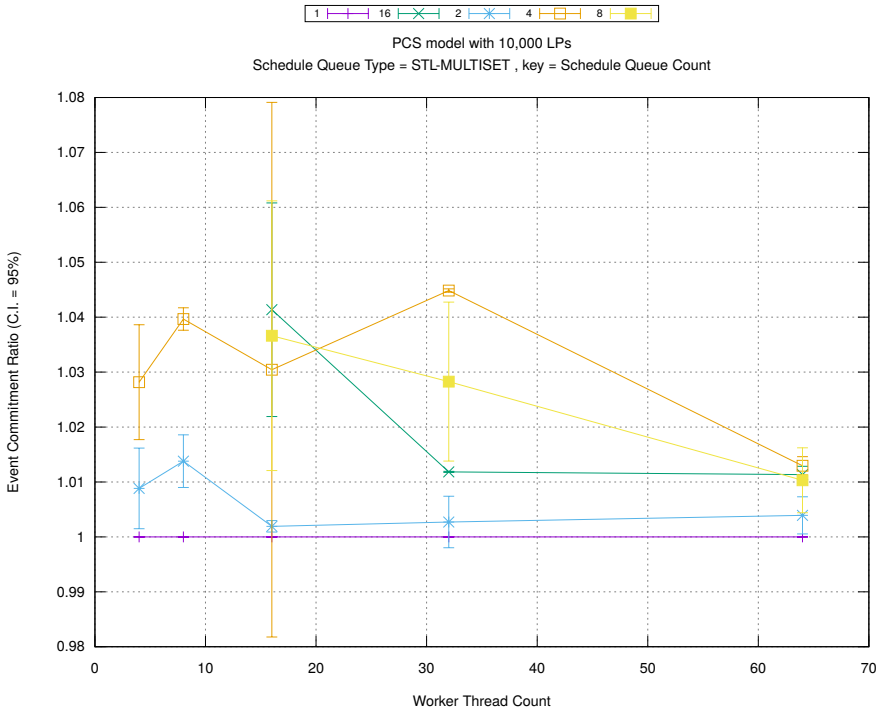
Figure A.214: pcs 10k/plots/scheduleq/threads vs count key type ladder-queue 96



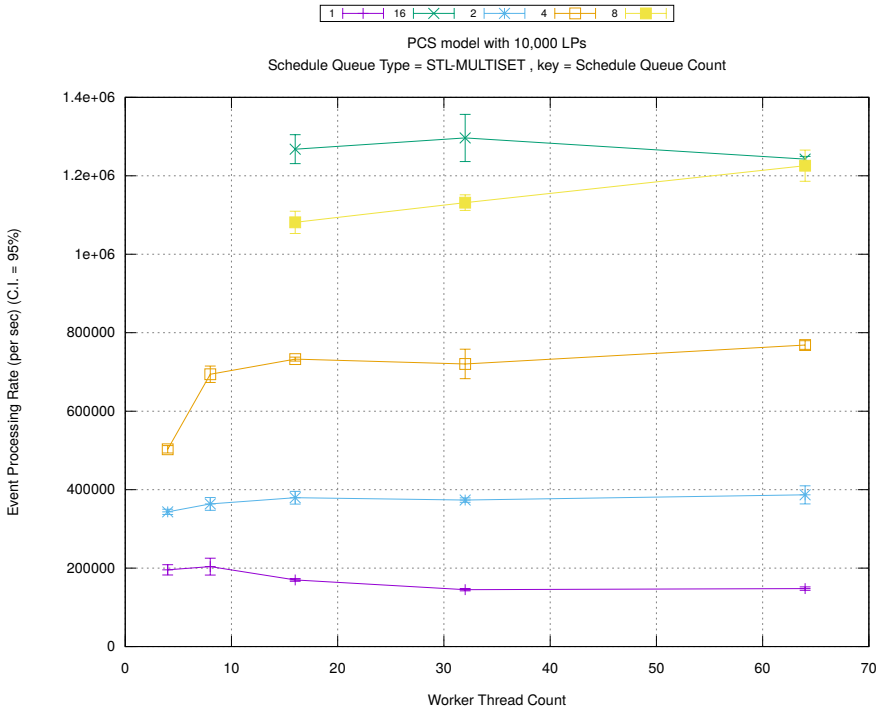
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

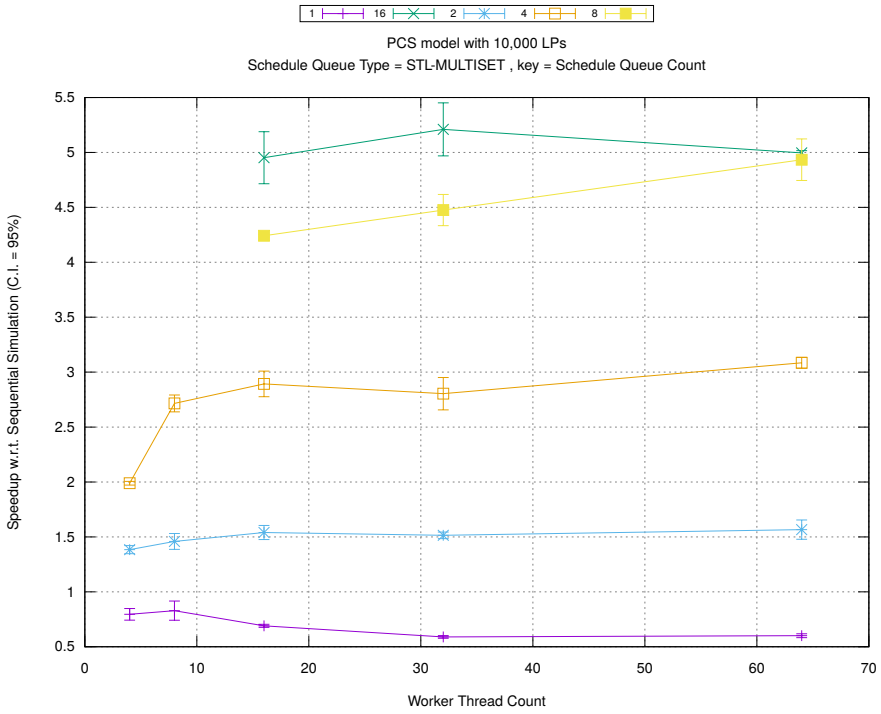


(b) Event Commitment Ratio

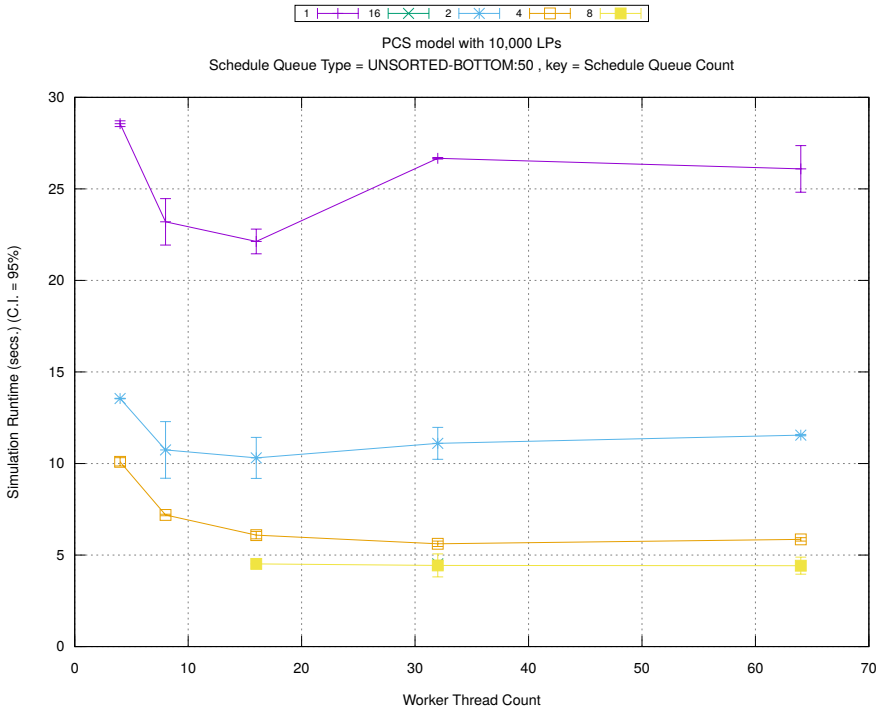


(c) Event Processing Rate (per second)

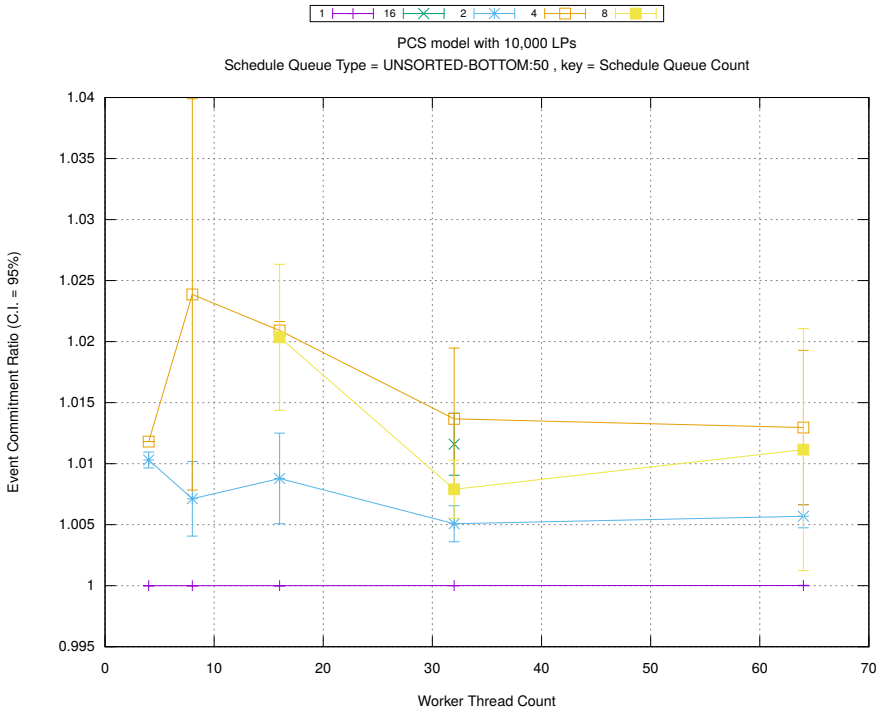
Figure A.215: pcs 10k/plots/scheduleq/threads vs count key type stl-multiset



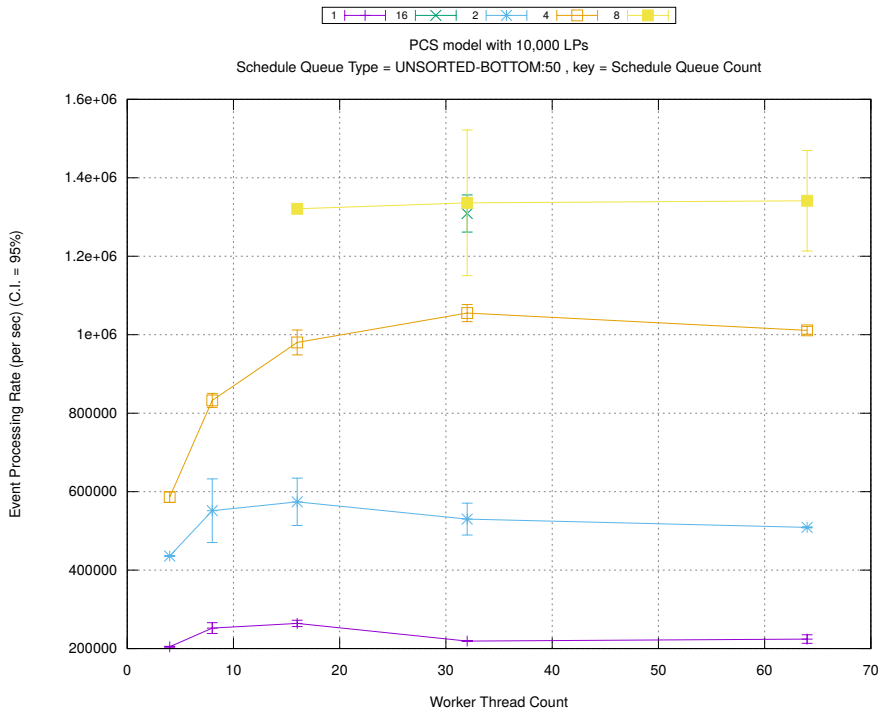
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

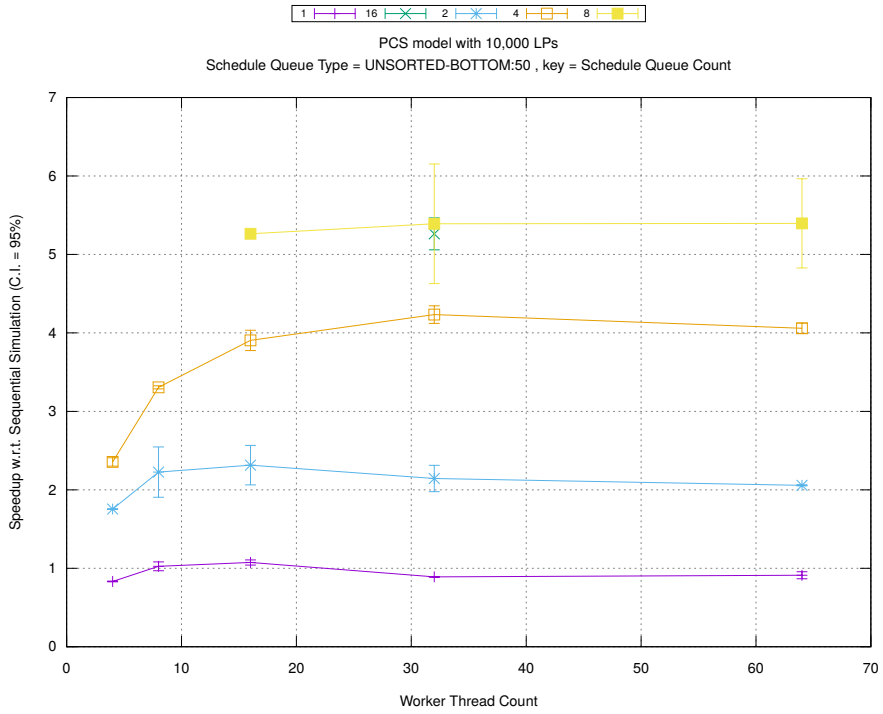


(b) Event Commitment Ratio

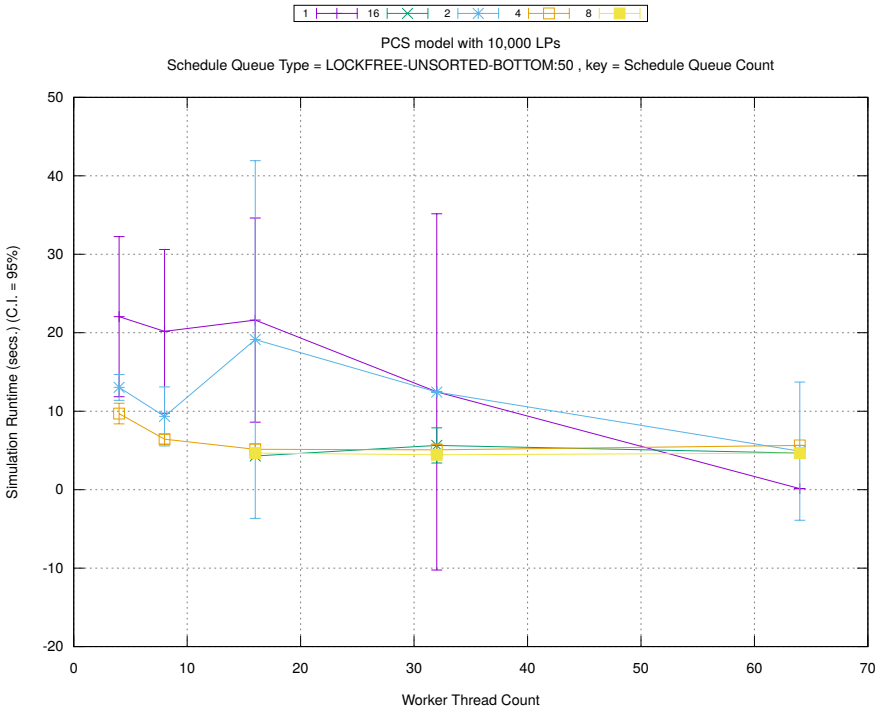


(c) Event Processing Rate (per second)

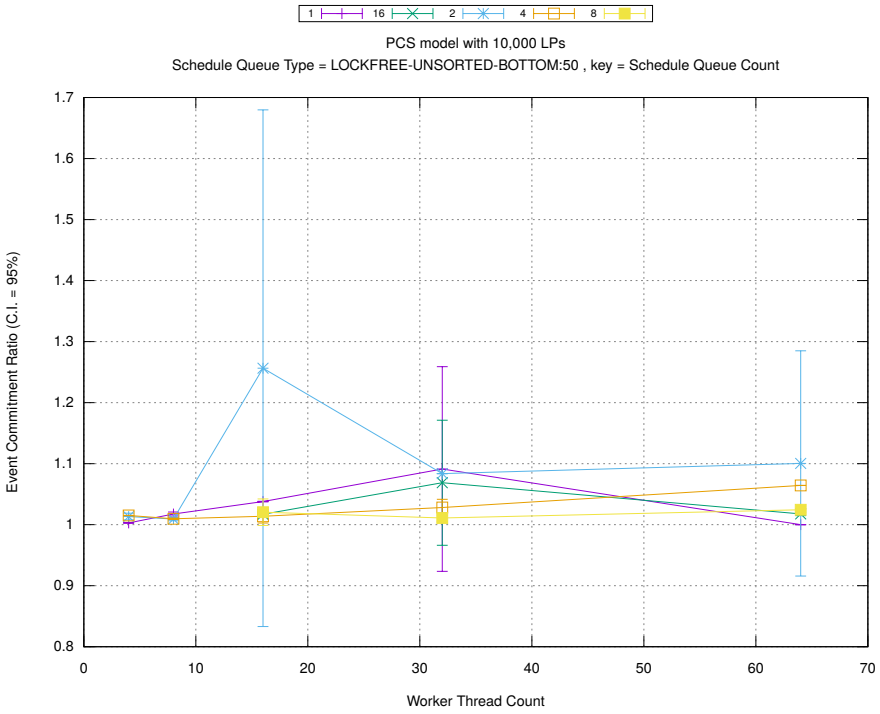
Figure A.216: pcs 10k/plots/scheduleq/threads vs count key type unsorted-bottom 50



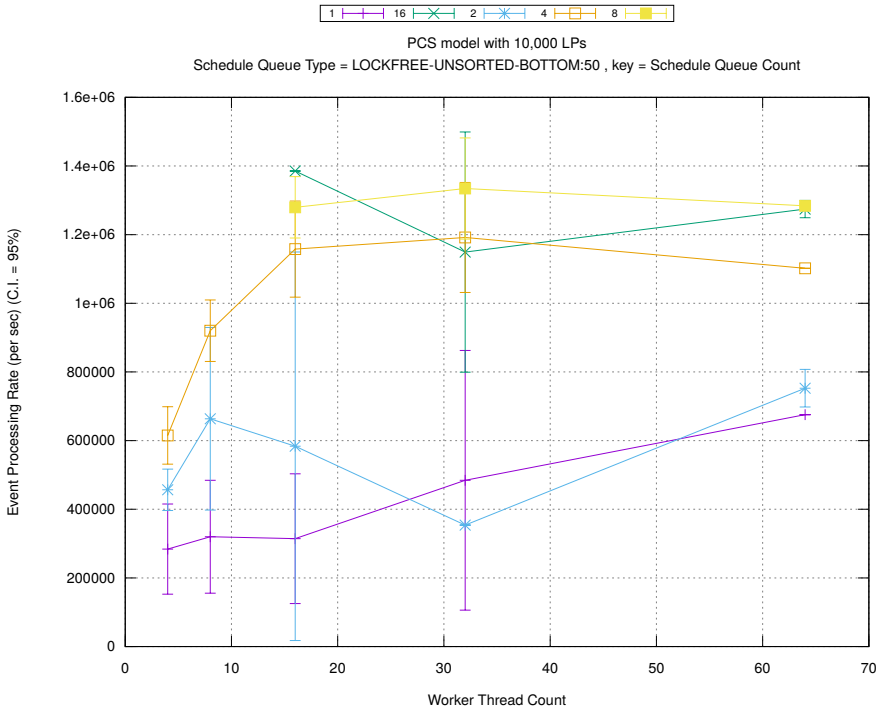
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

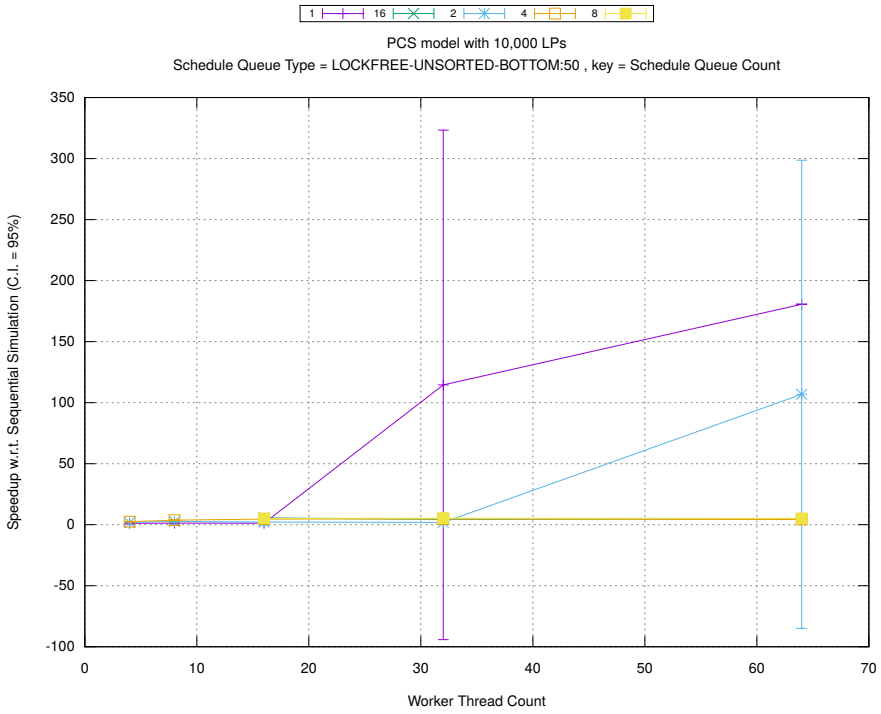


(b) Event Commitment Ratio

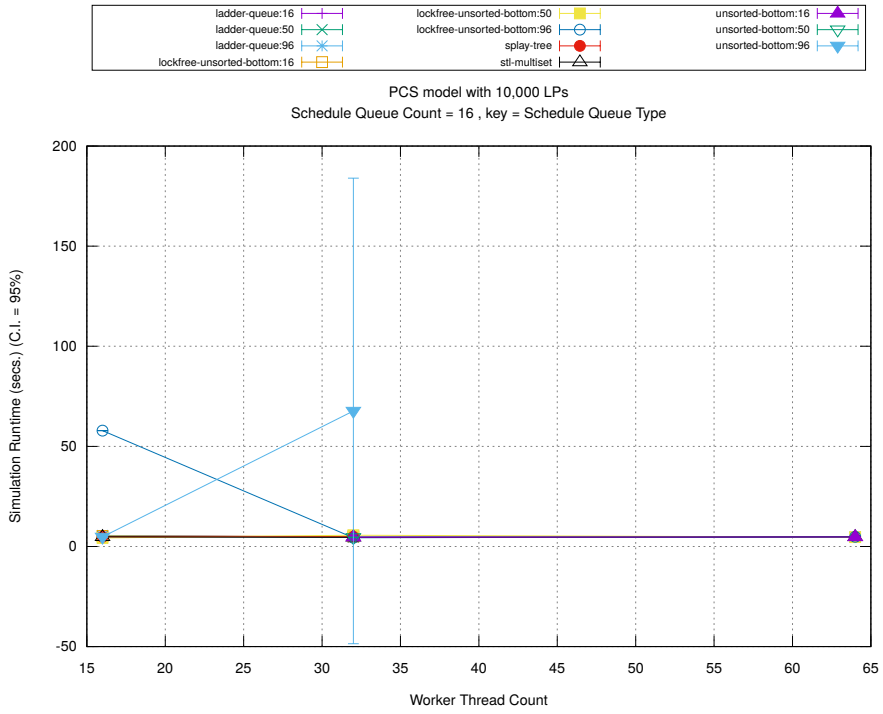


(c) Event Processing Rate (per second)

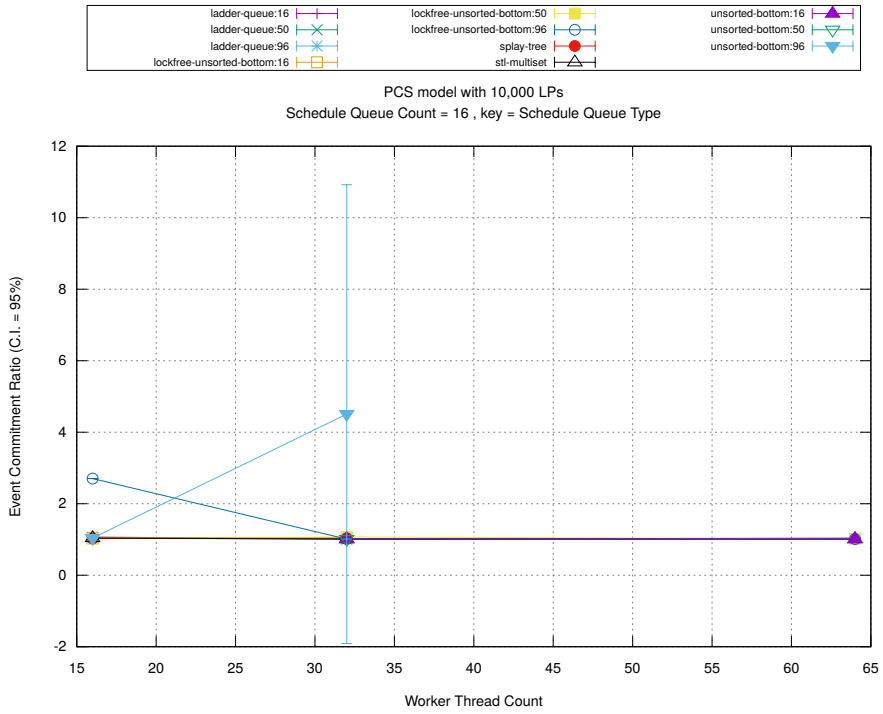
Figure A.217: pcs 10k/plots/scheduleq/threads vs count key type lockfree-unsorted-bottom 50



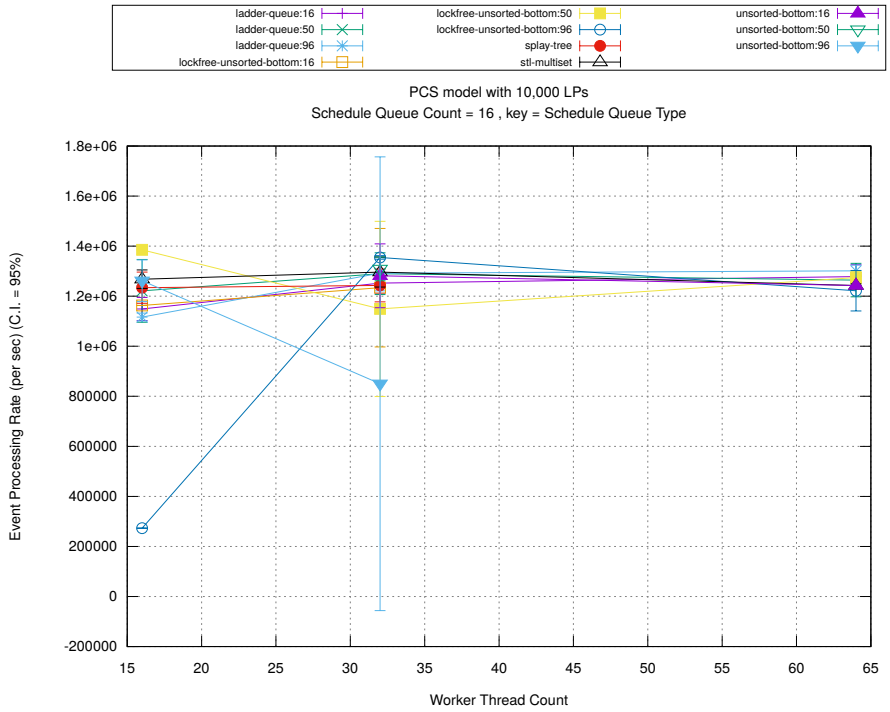
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

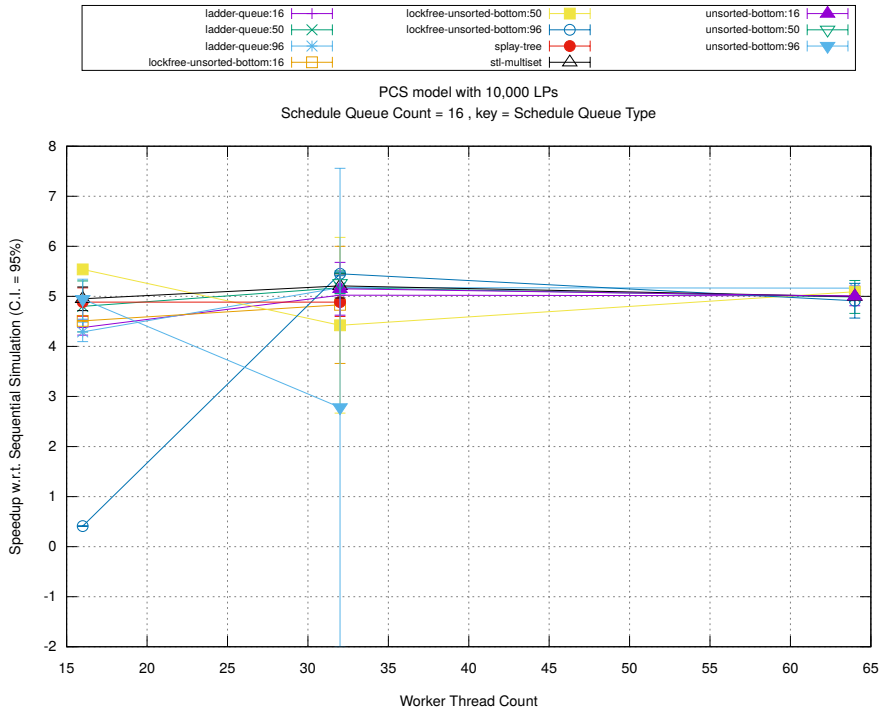


(b) Event Commitment Ratio

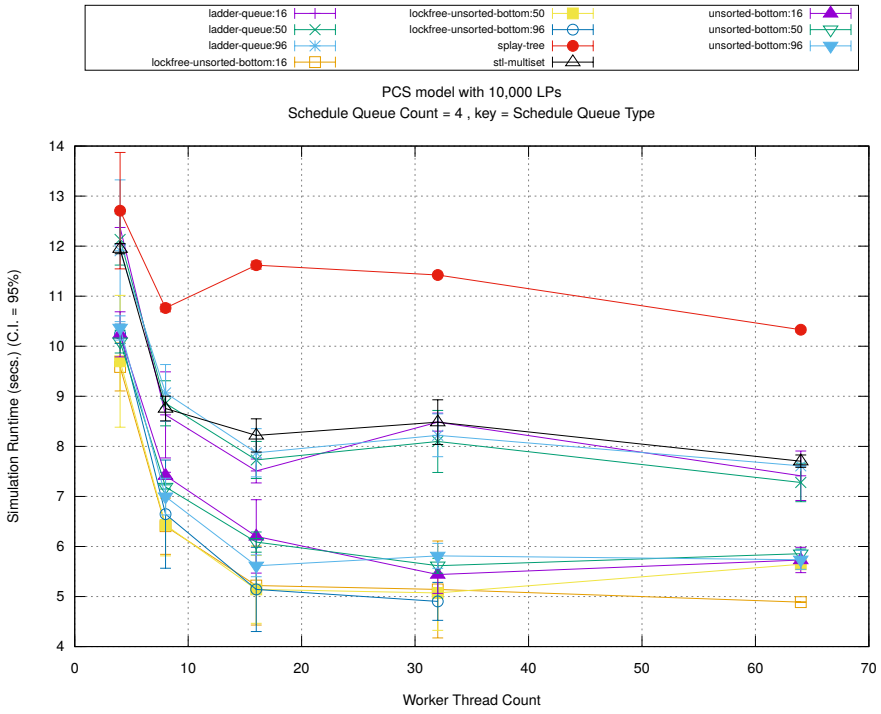


(c) Event Processing Rate (per second)

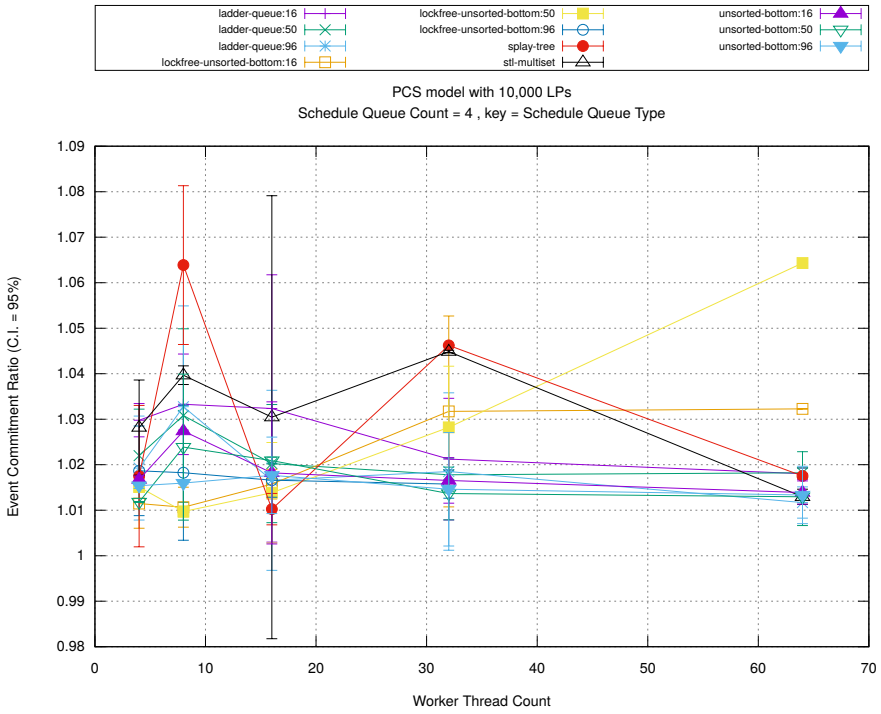
Figure A.218: pcs 10k/plots/scheduleq/threads vs type key count 16



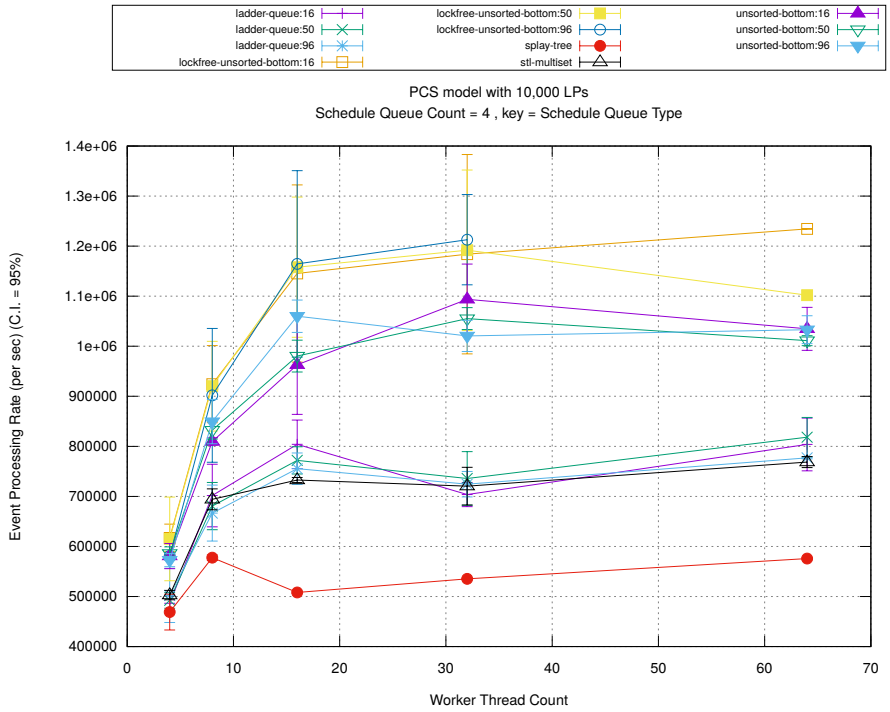
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

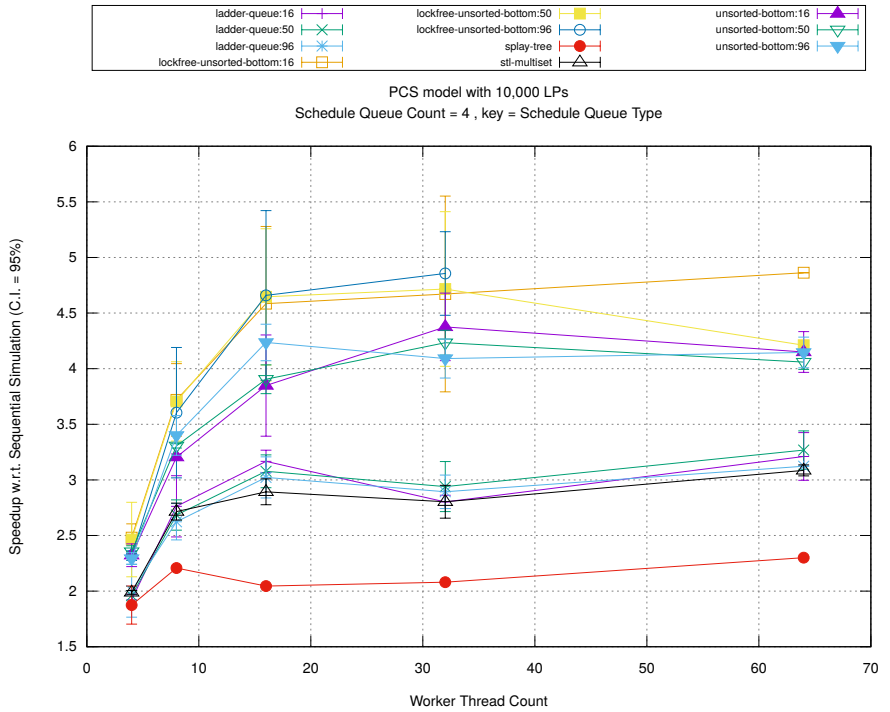


(b) Event Commitment Ratio

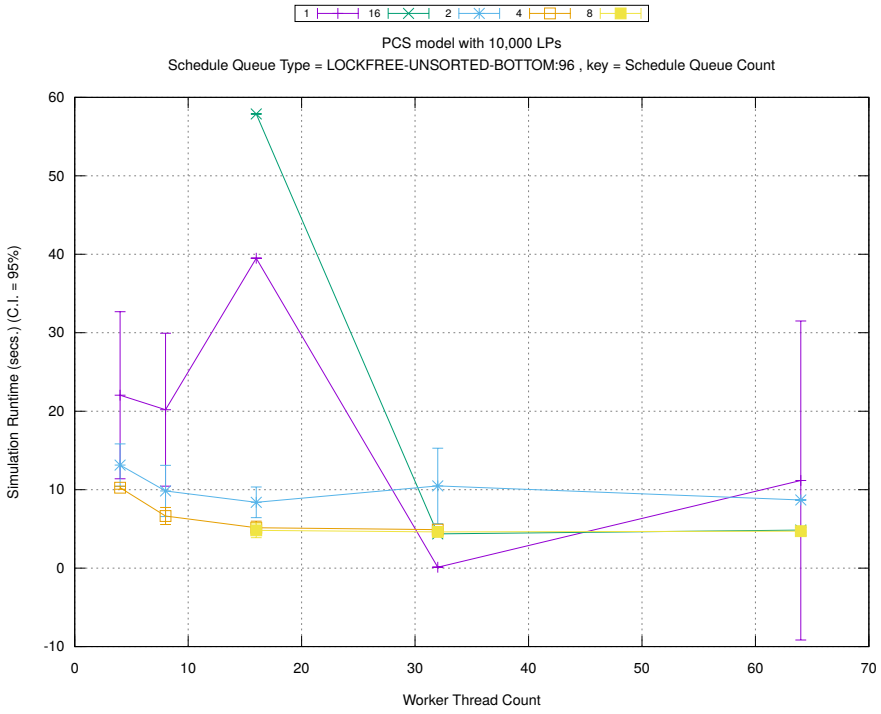


(c) Event Processing Rate (per second)

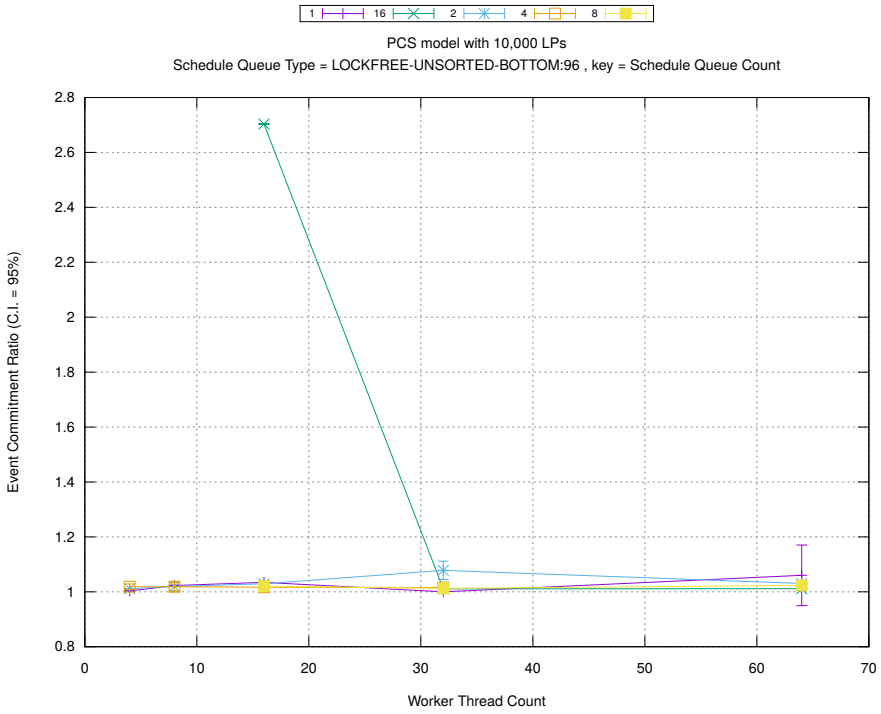
Figure A.219: pcs 10k/plots/scheduleq/threads vs type key count 4



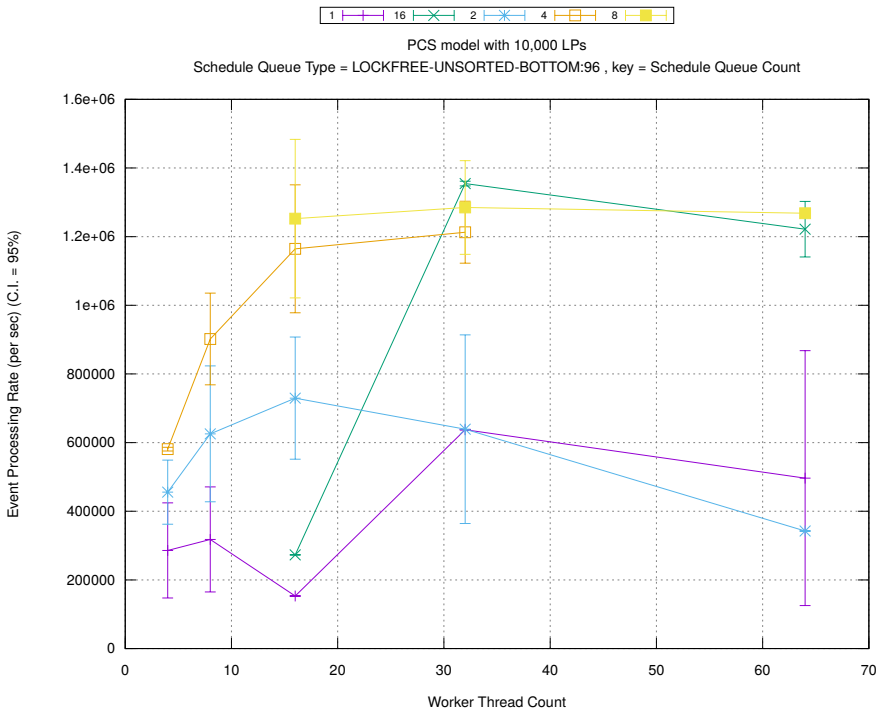
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

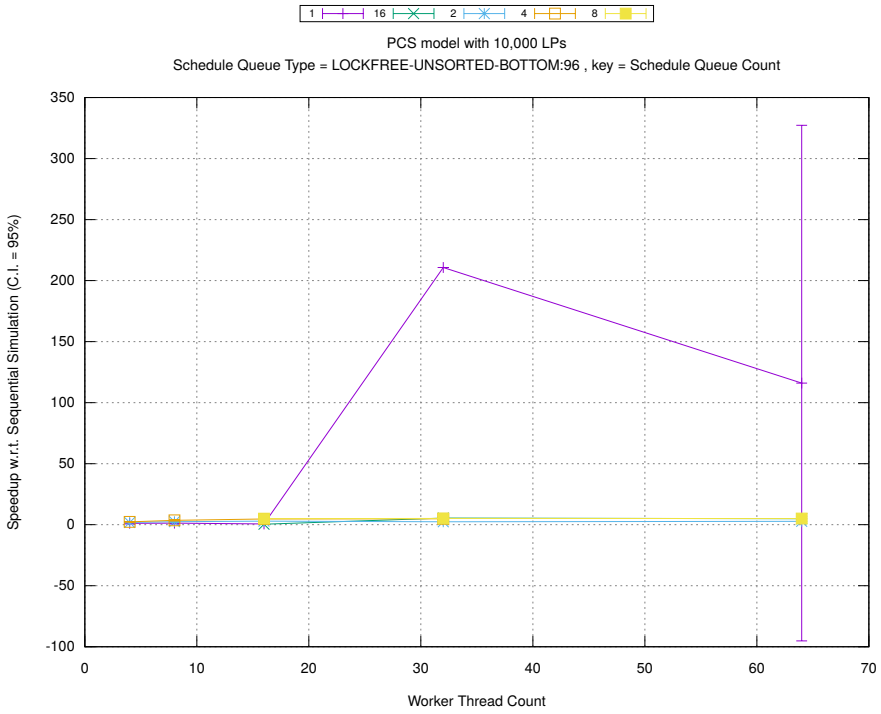


(b) Event Commitment Ratio

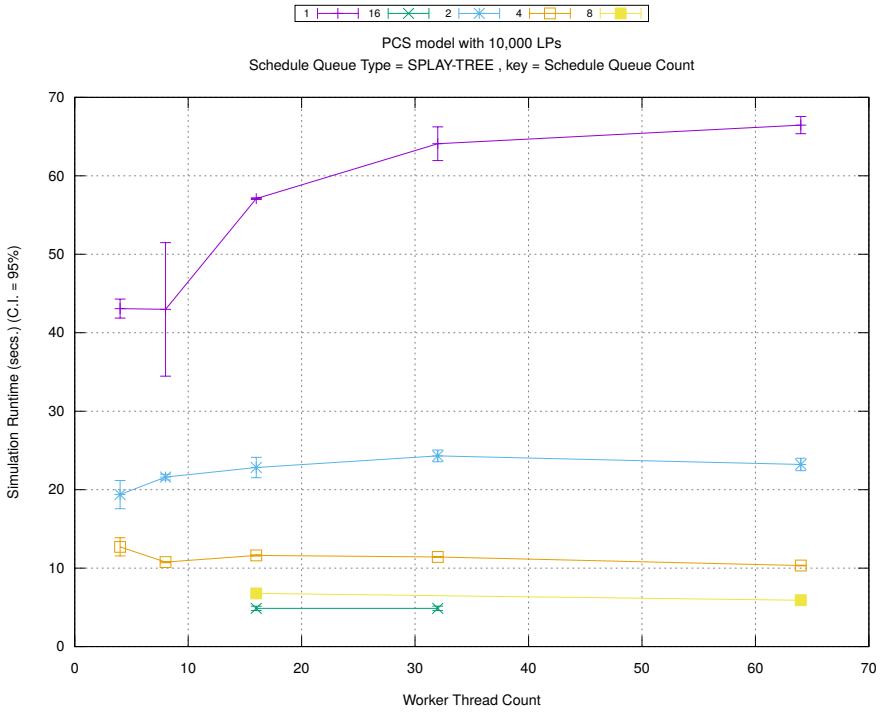


(c) Event Processing Rate (per second)

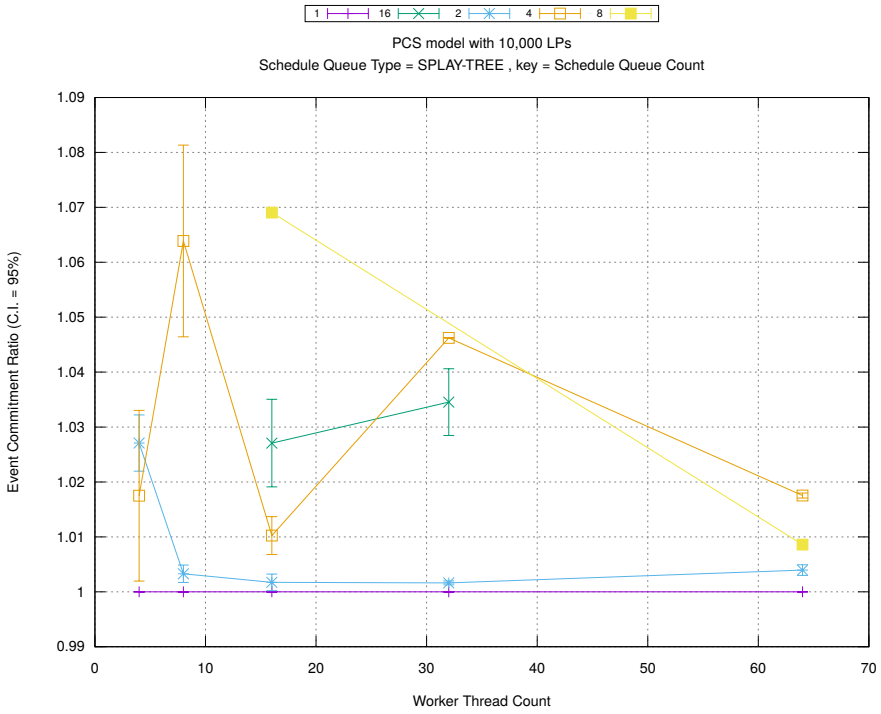
Figure A.220: pcs 10k/plots/scheduleq/threads vs count key type lockfree-unsorted-bottom 96



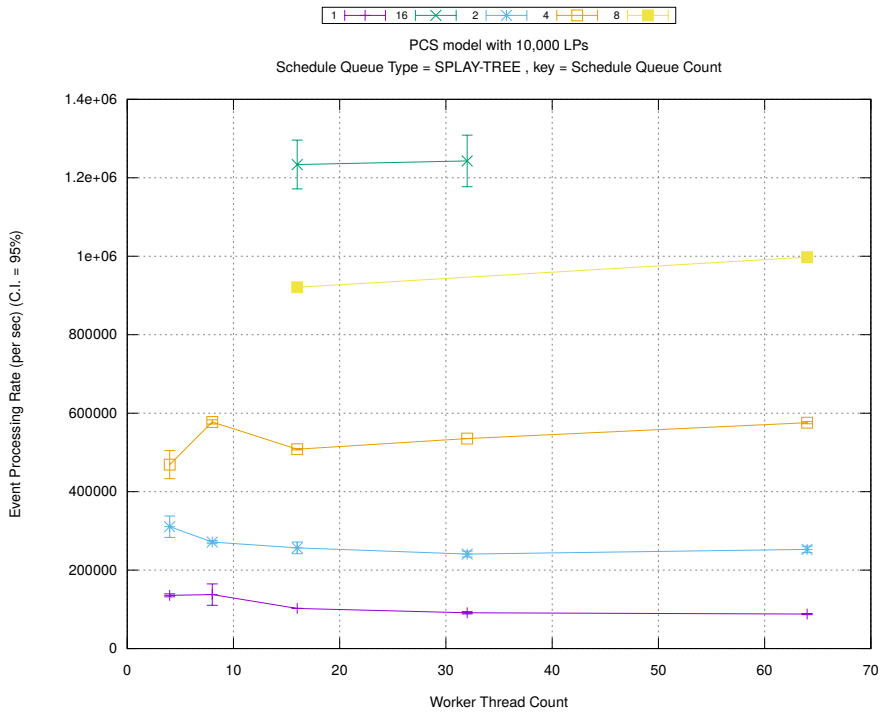
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

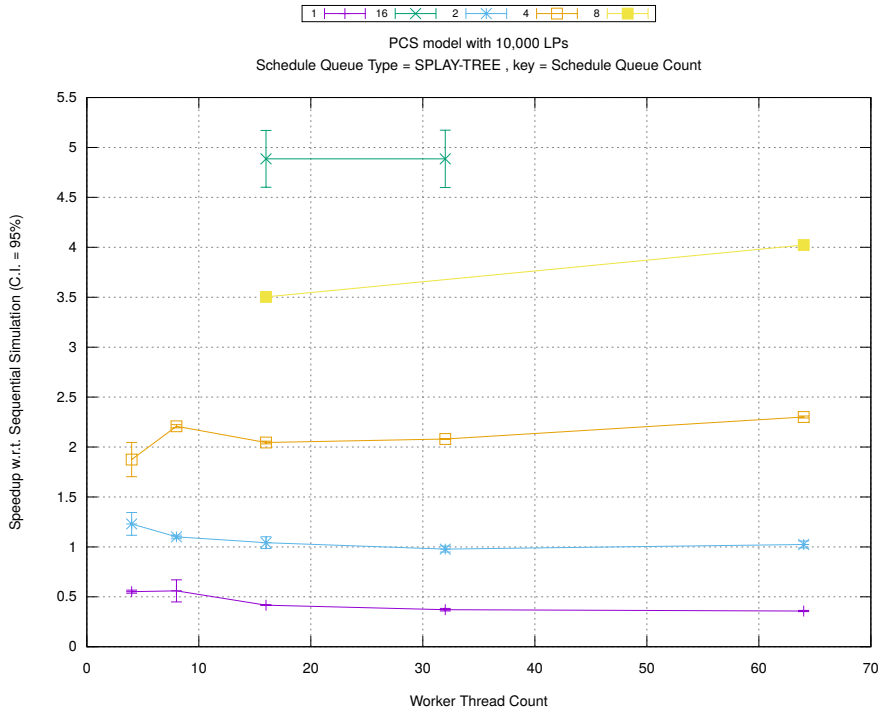


(b) Event Commitment Ratio

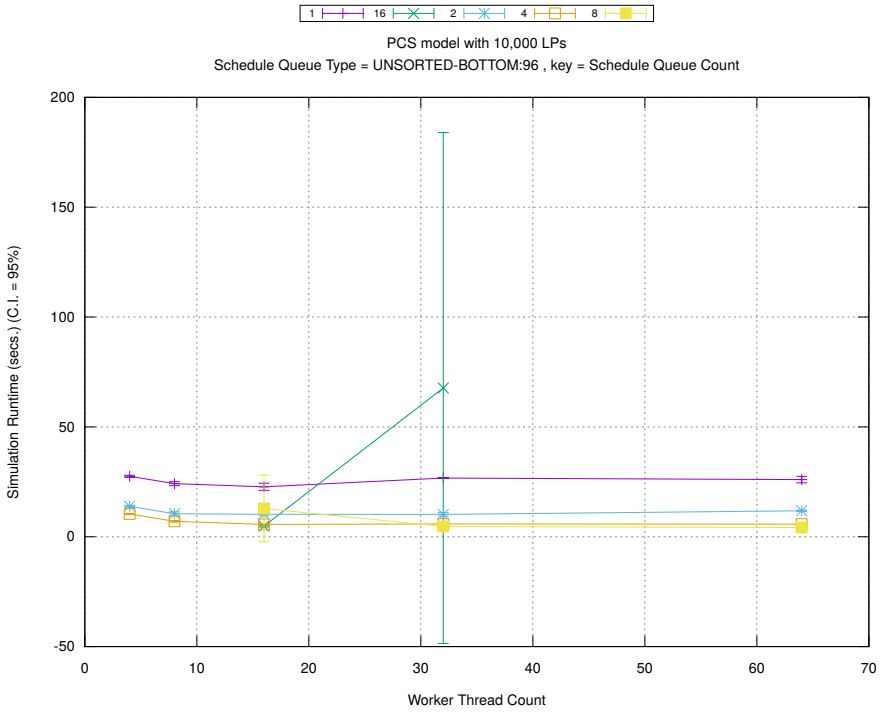


(c) Event Processing Rate (per second)

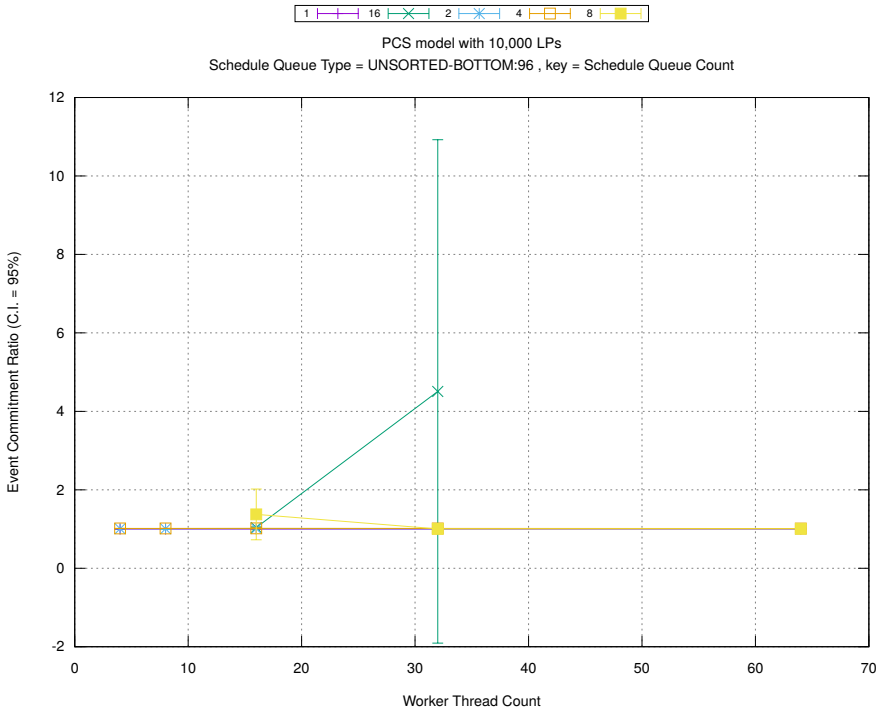
Figure A.221: pcs 10k/plots/scheduleq/threads vs count key type splay-tree



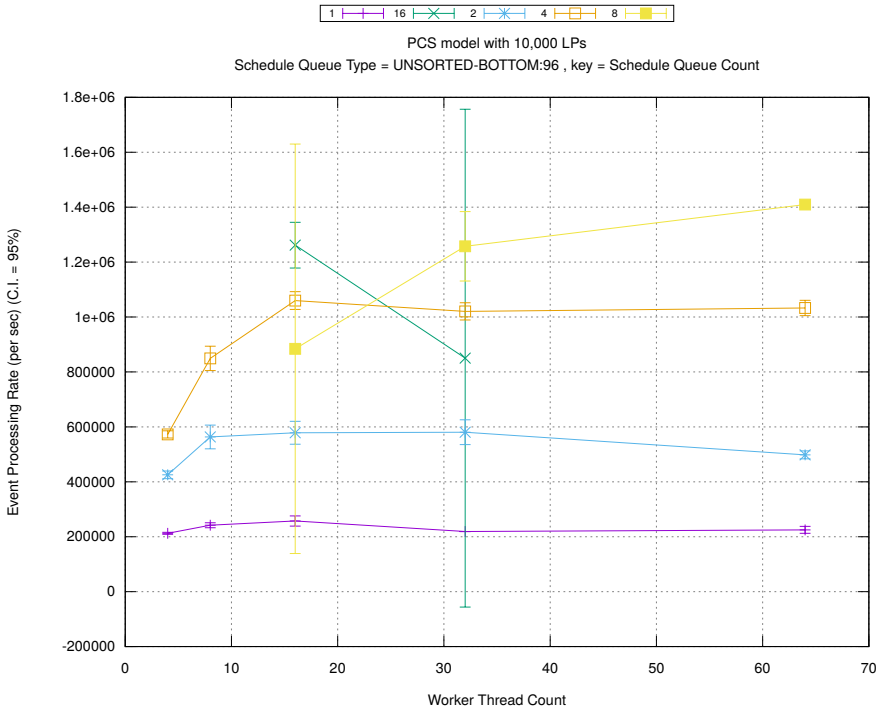
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

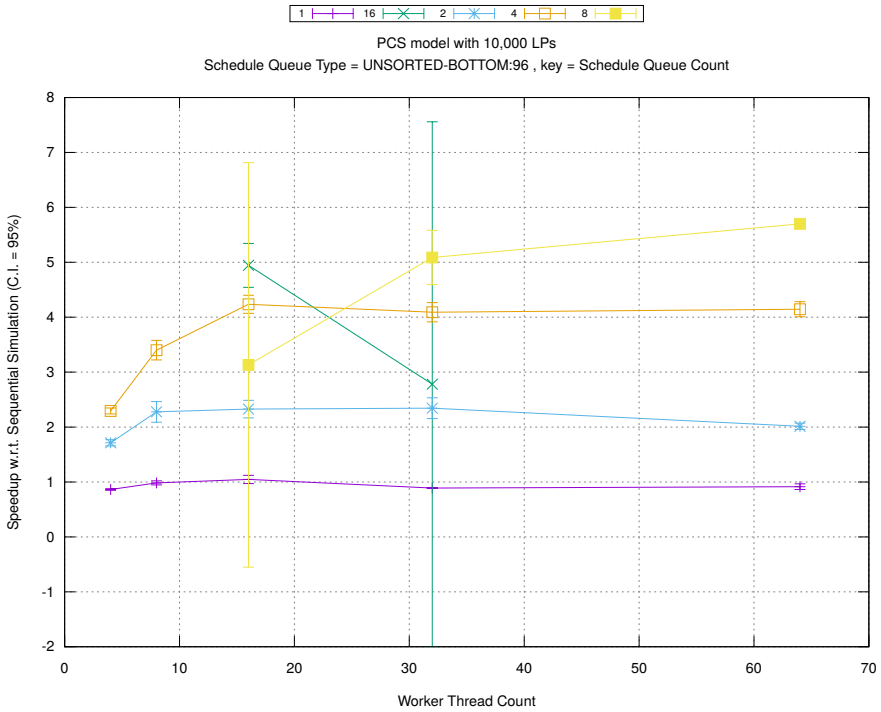


(b) Event Commitment Ratio

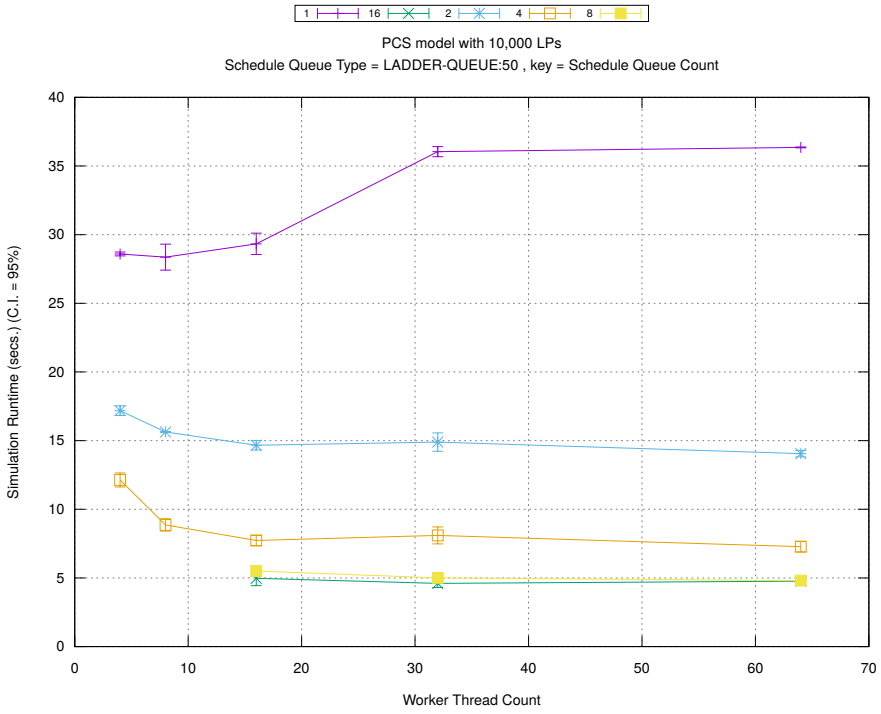


(c) Event Processing Rate (per second)

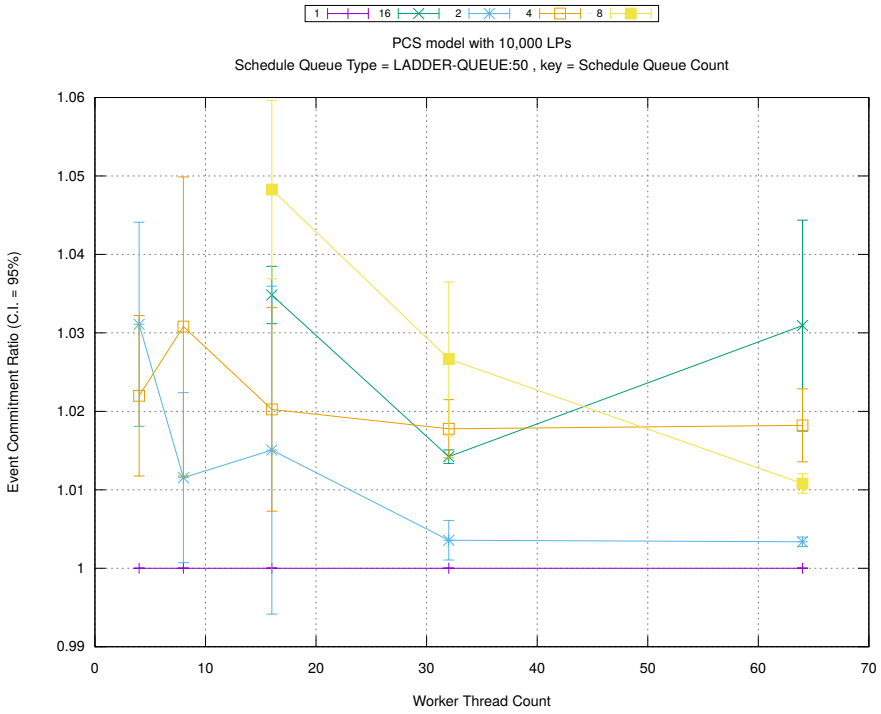
Figure A.222: pcs 10k/plots/scheduleq/threads vs count key type unsorted-bottom 96



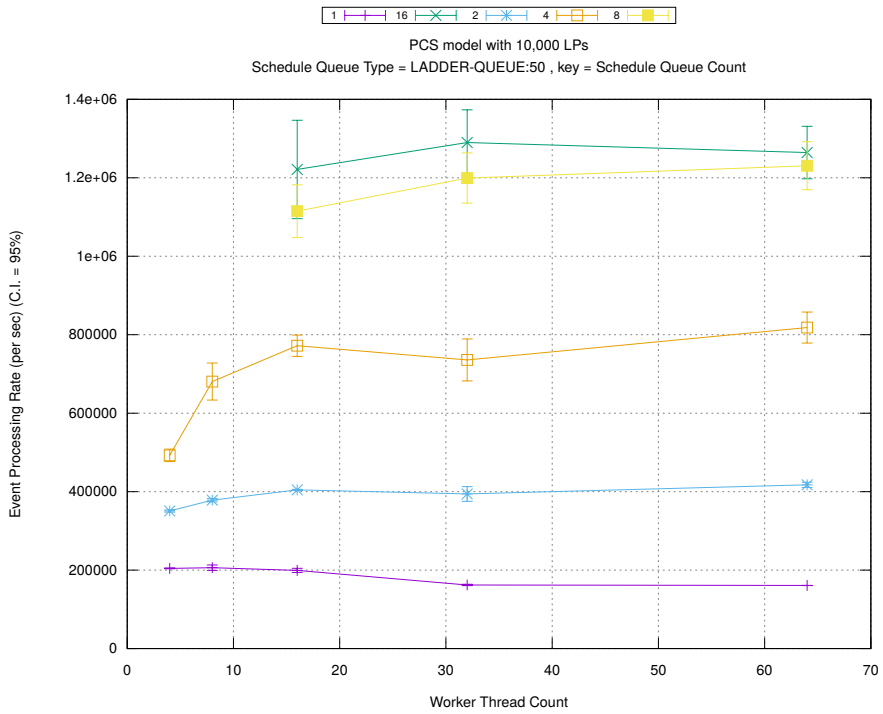
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

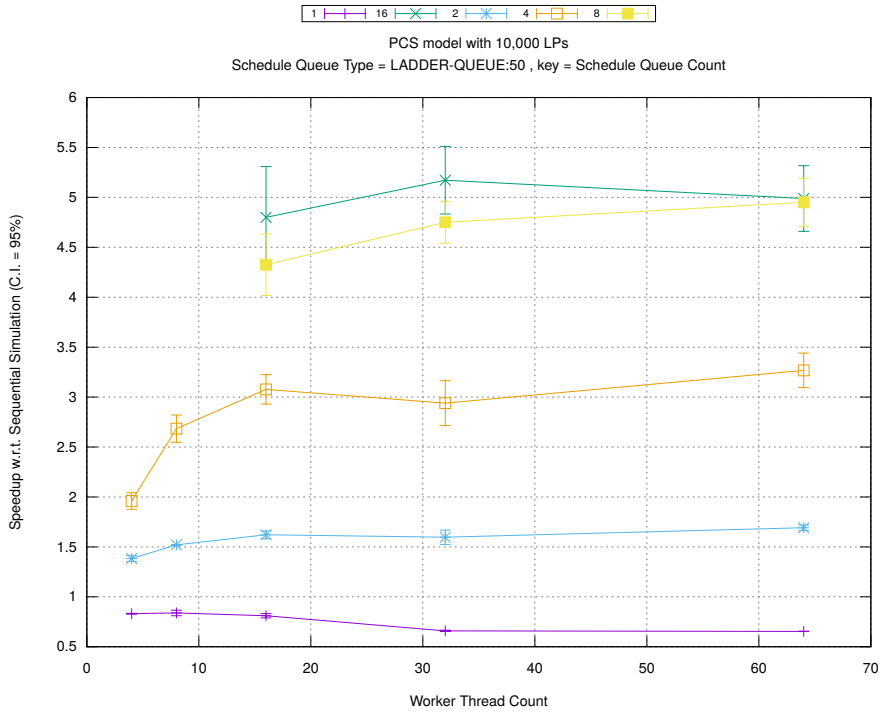


(b) Event Commitment Ratio

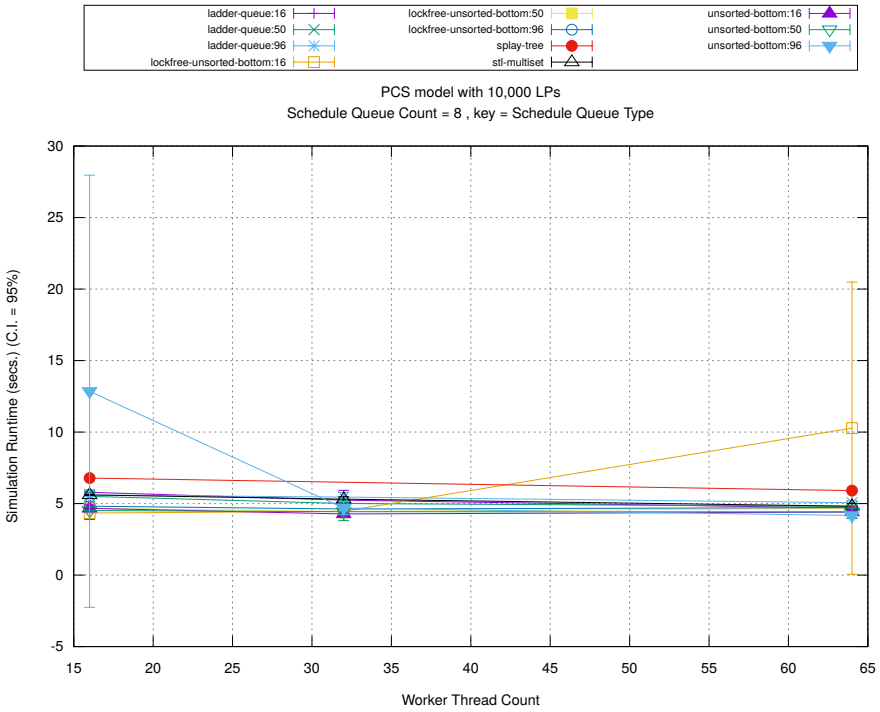


(c) Event Processing Rate (per second)

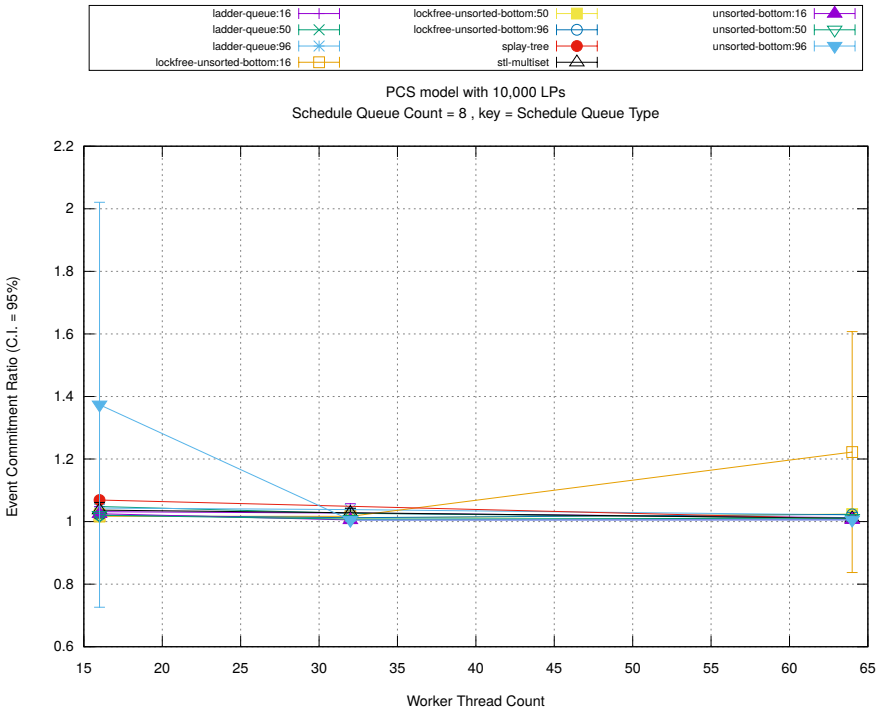
Figure A.223: pcs 10k/plots/scheduleq/threads vs count key type ladder-queue 50



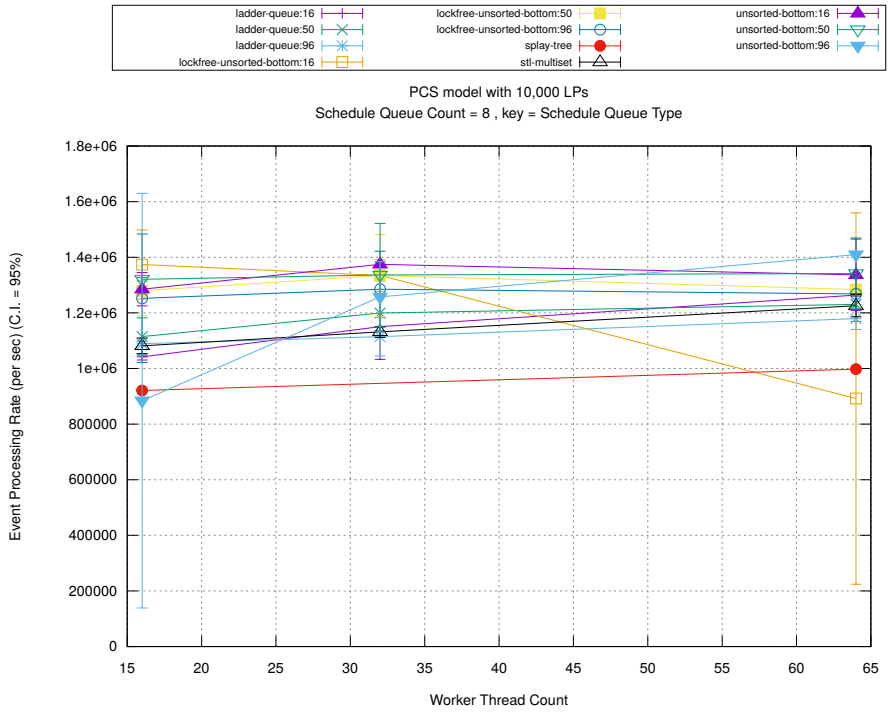
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

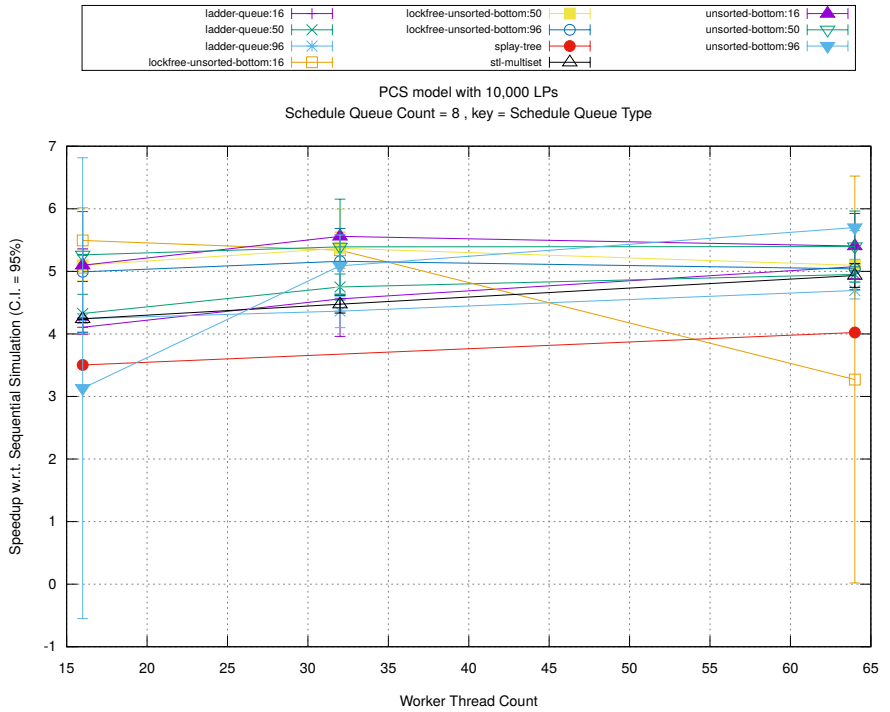


(b) Event Commitment Ratio

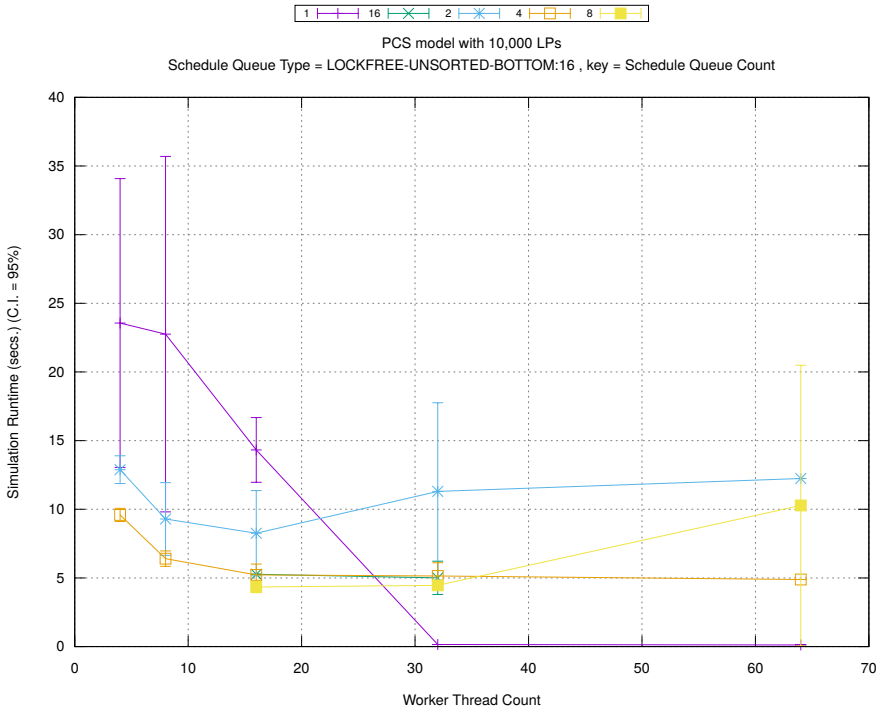


(c) Event Processing Rate (per second)

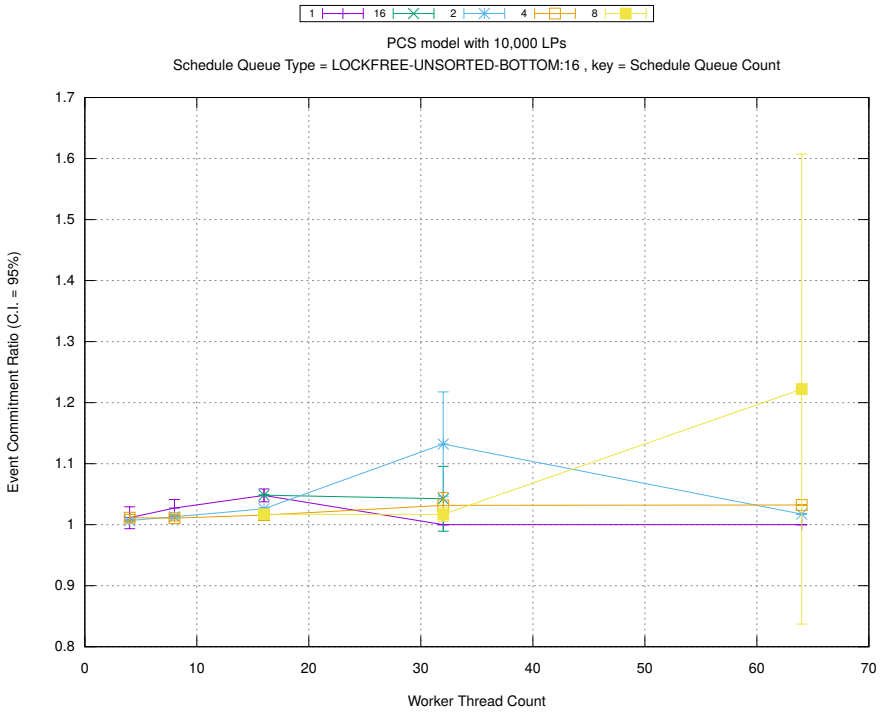
Figure A.224: pcs 10k/plots/scheduleq/threads vs type key count 8



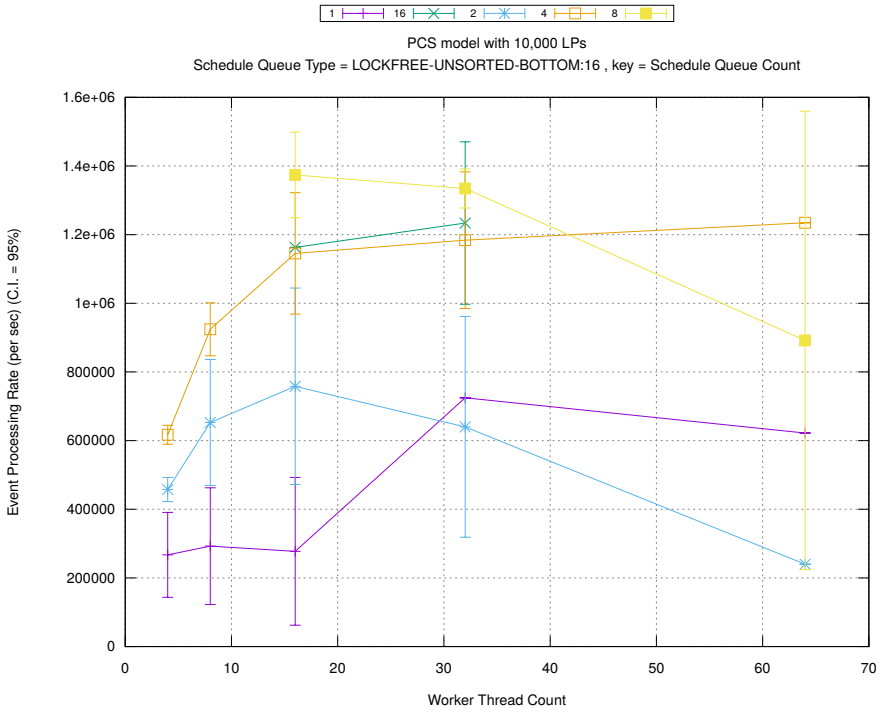
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

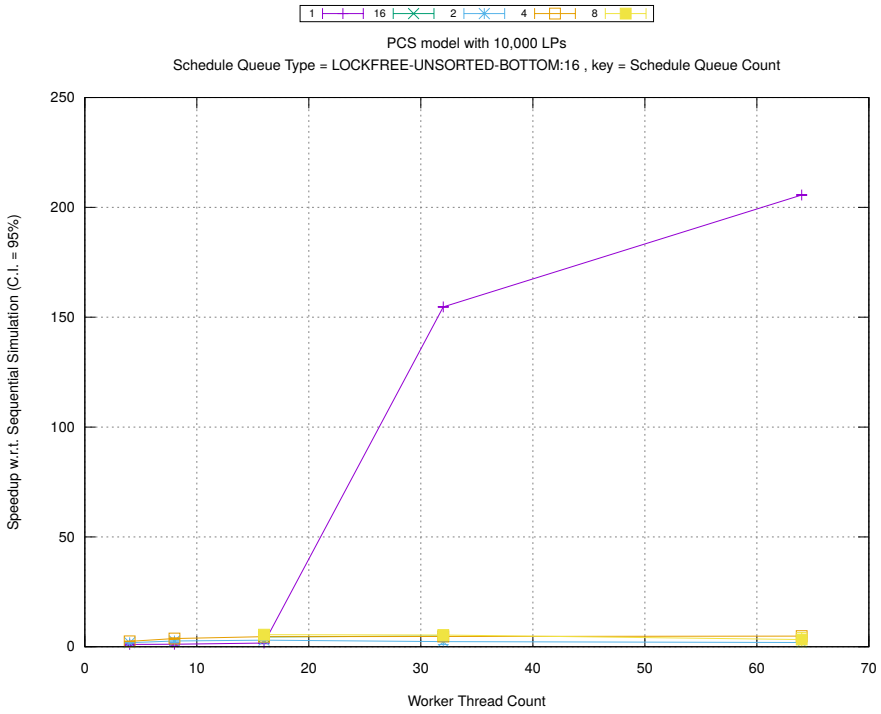


(b) Event Commitment Ratio

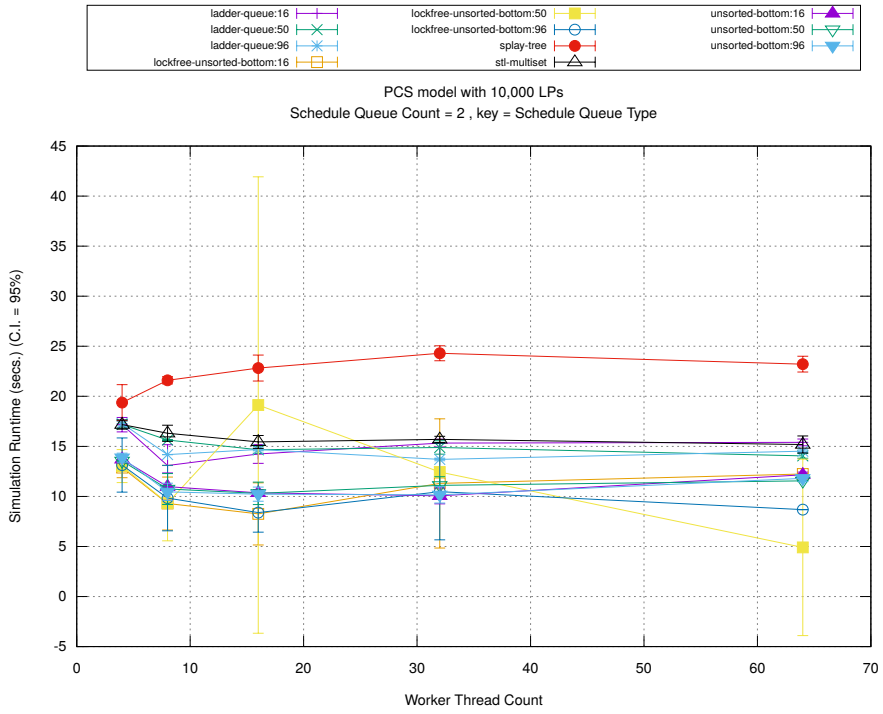


(c) Event Processing Rate (per second)

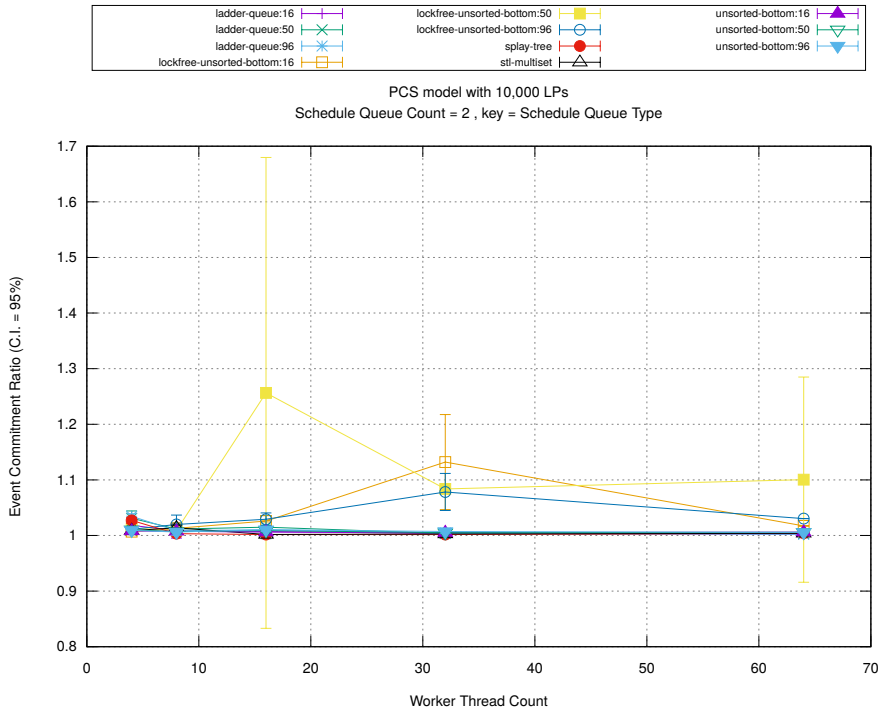
Figure A.225: pcs 10k/plots/scheduleq/threads vs count key type lockfree-unsorted-bottom 16



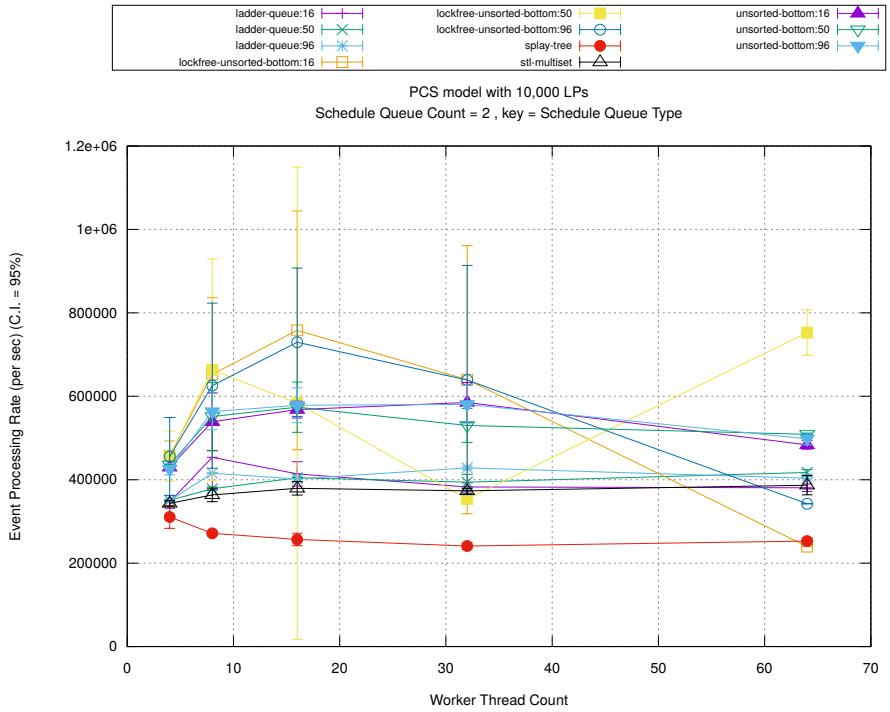
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

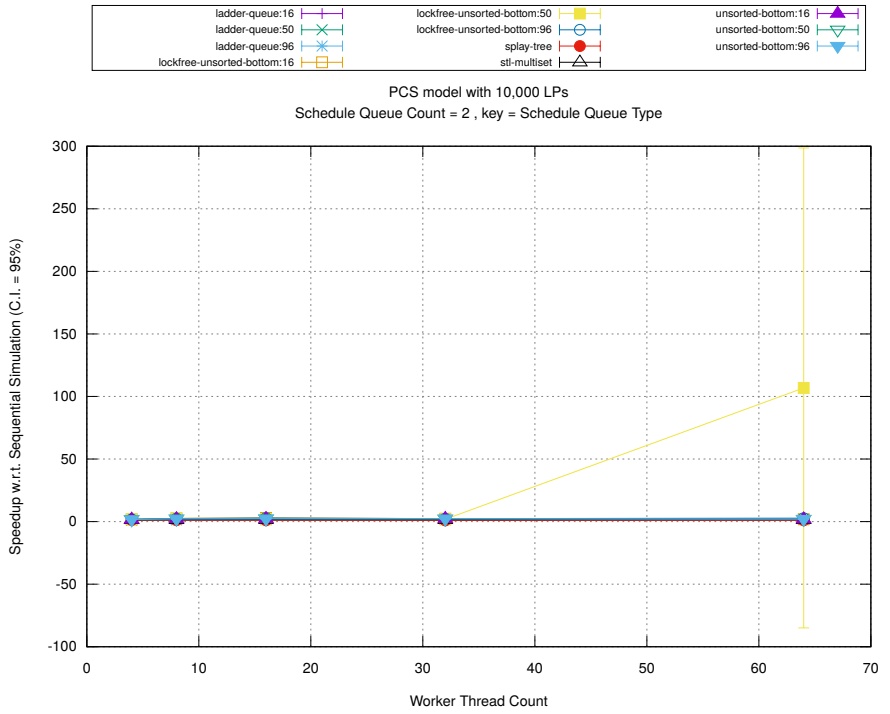


(b) Event Commitment Ratio

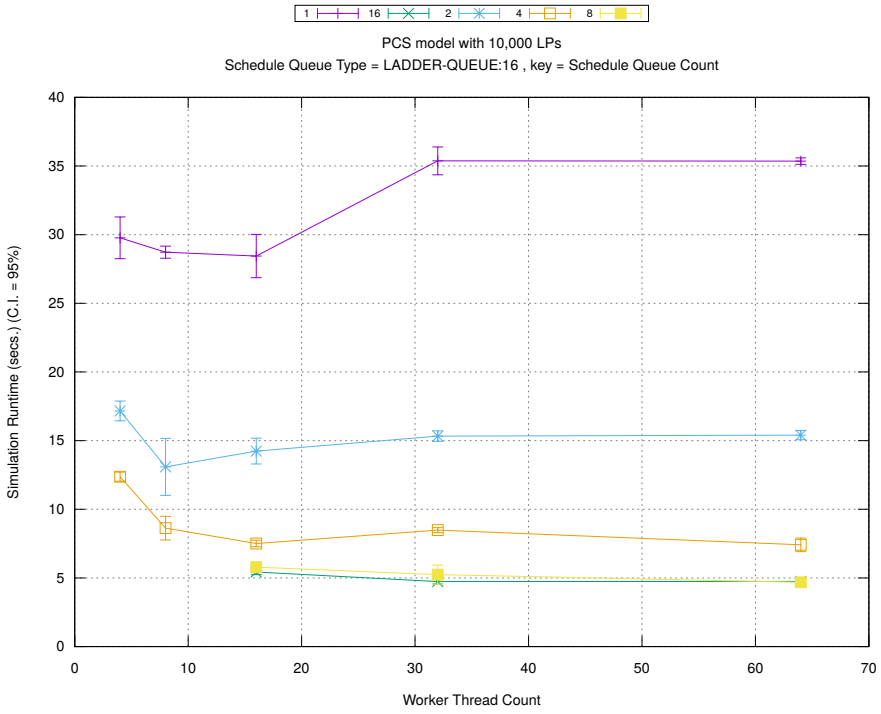


(c) Event Processing Rate (per second)

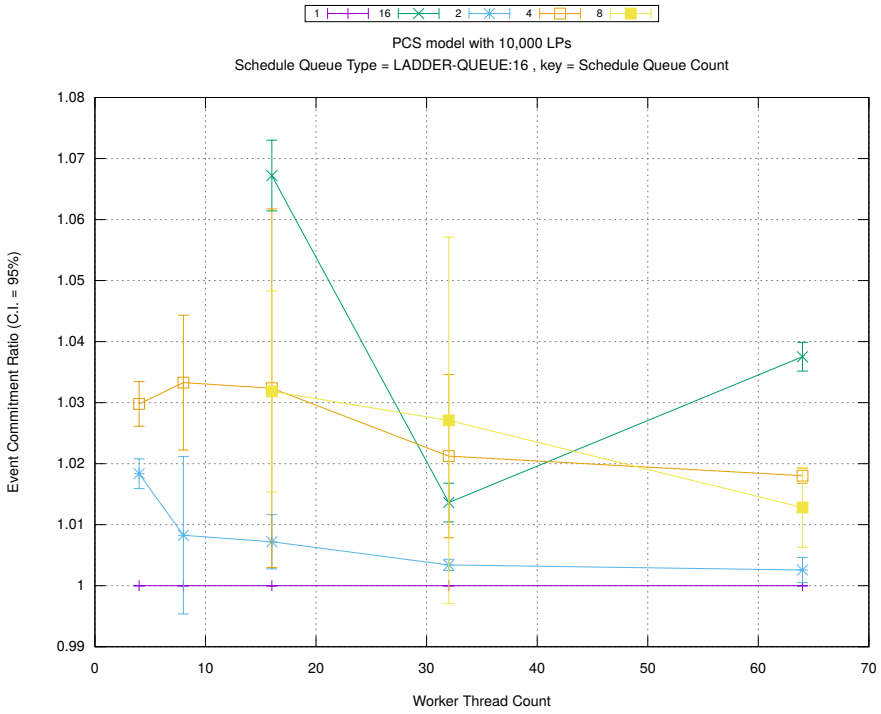
Figure A.226: pcs 10k/plots/scheduleq/threads vs type key count 2



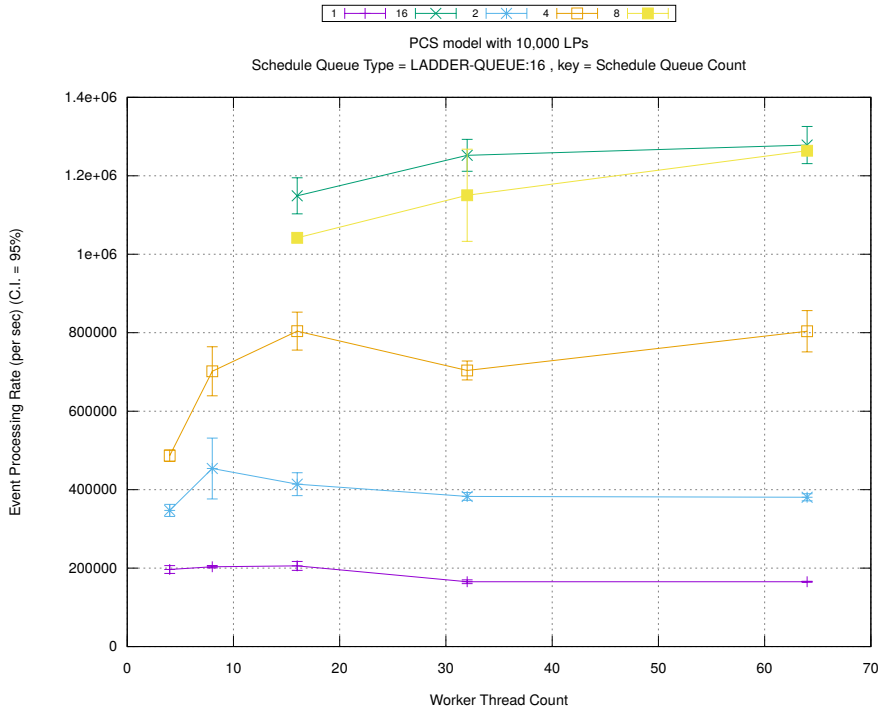
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

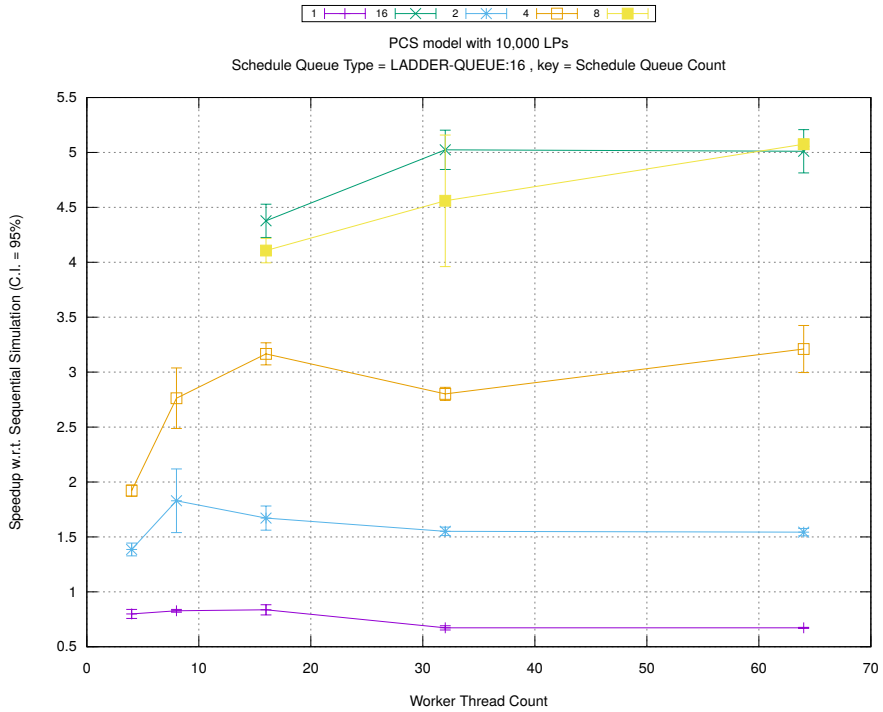


(b) Event Commitment Ratio

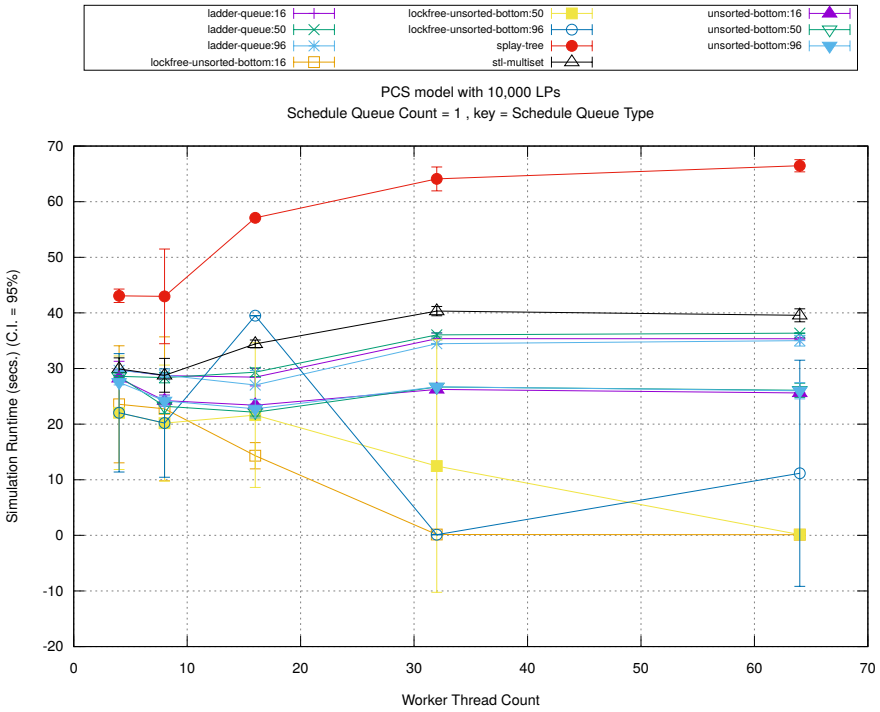


(c) Event Processing Rate (per second)

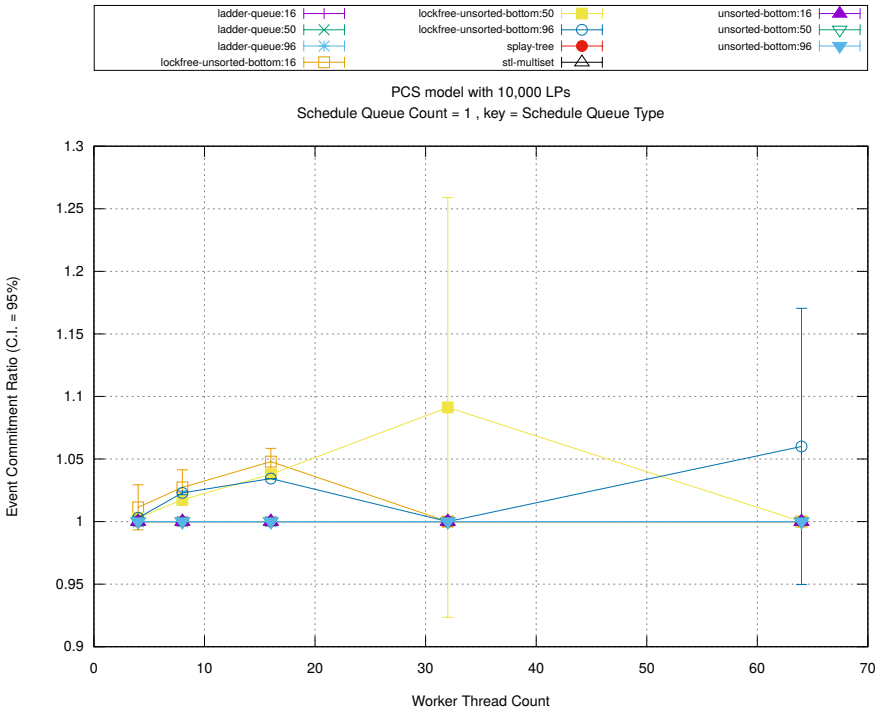
Figure A.227: pcs 10k/plots/scheduleq/threads vs count key type ladder-queue 16



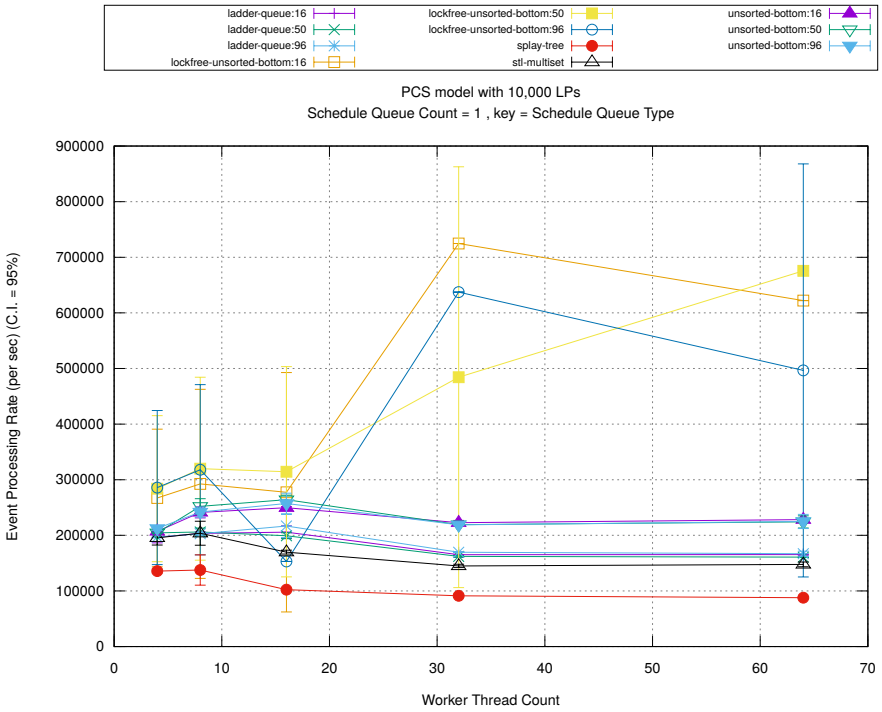
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

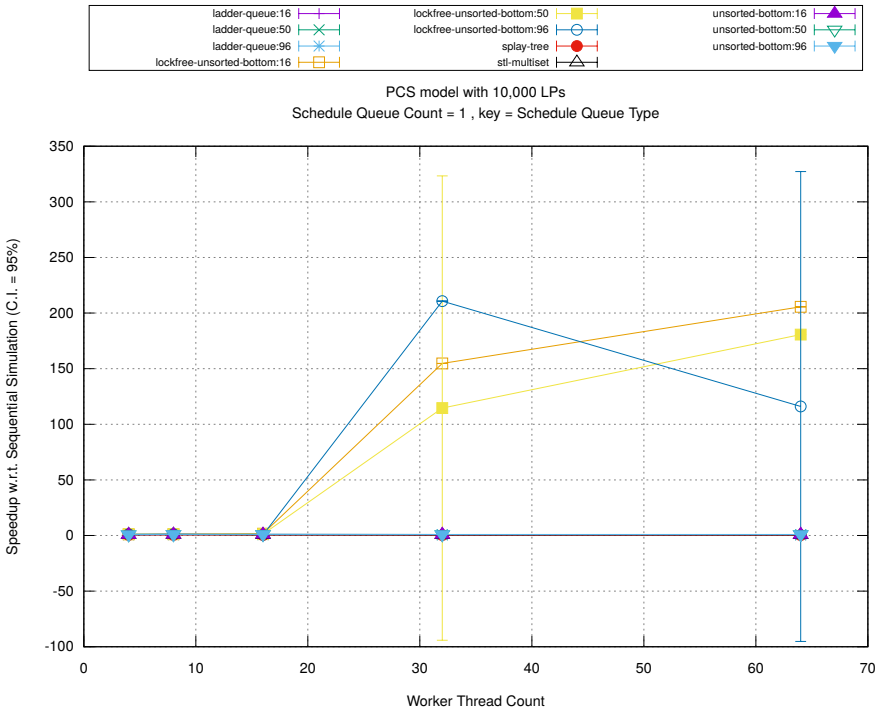


(b) Event Commitment Ratio

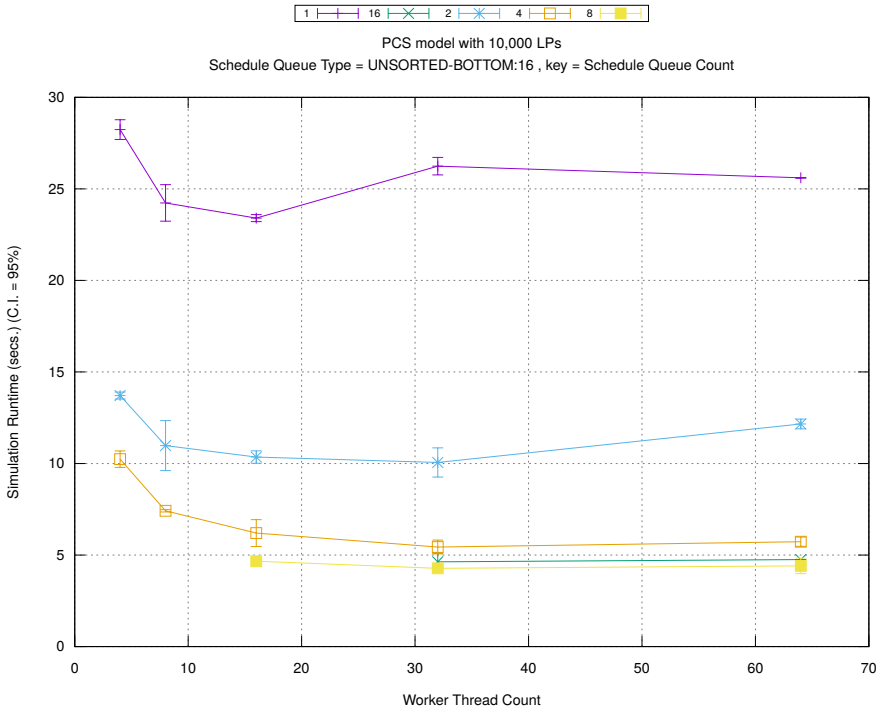


(c) Event Processing Rate (per second)

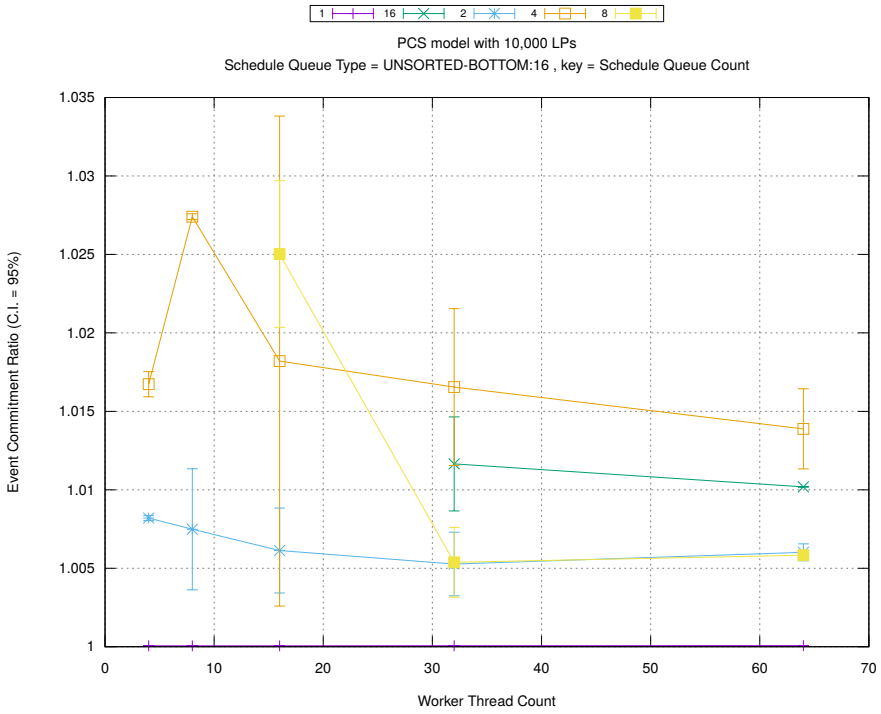
Figure A.228: pcs 10k/plots/scheduleq/threads vs type key count 1



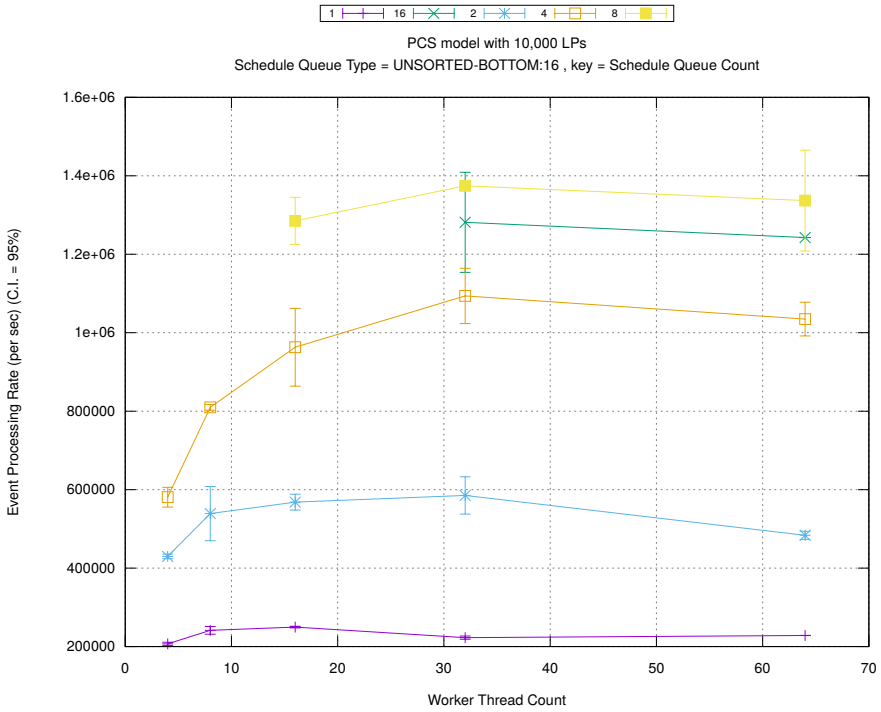
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

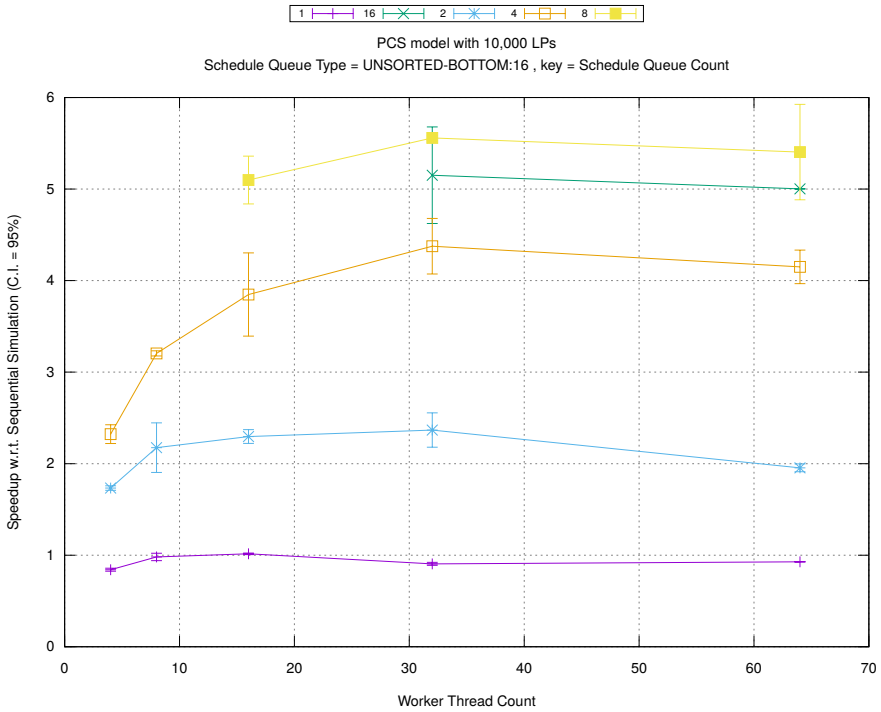


(b) Event Commitment Ratio

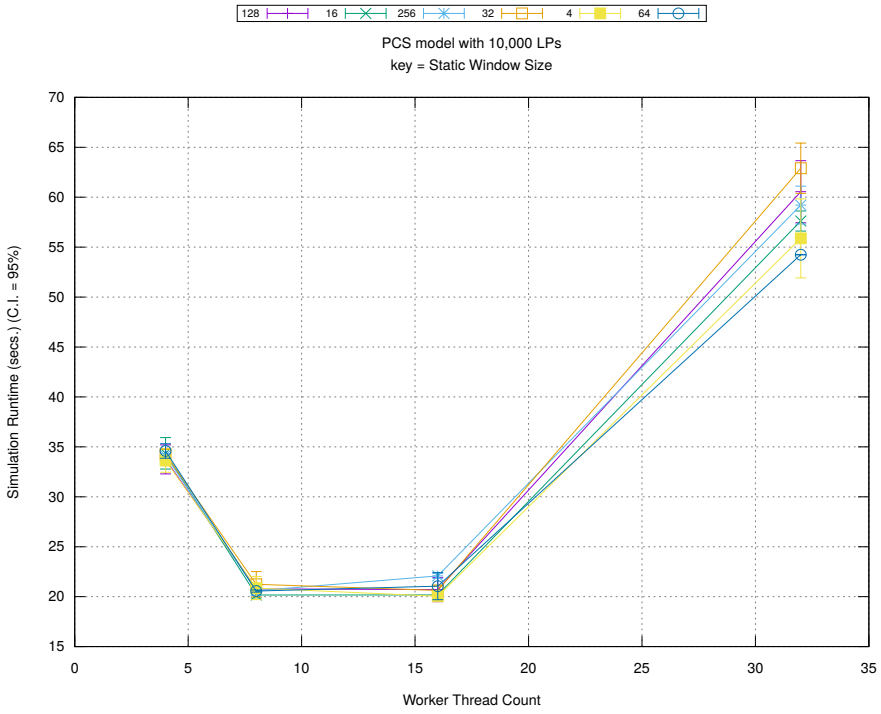


(c) Event Processing Rate (per second)

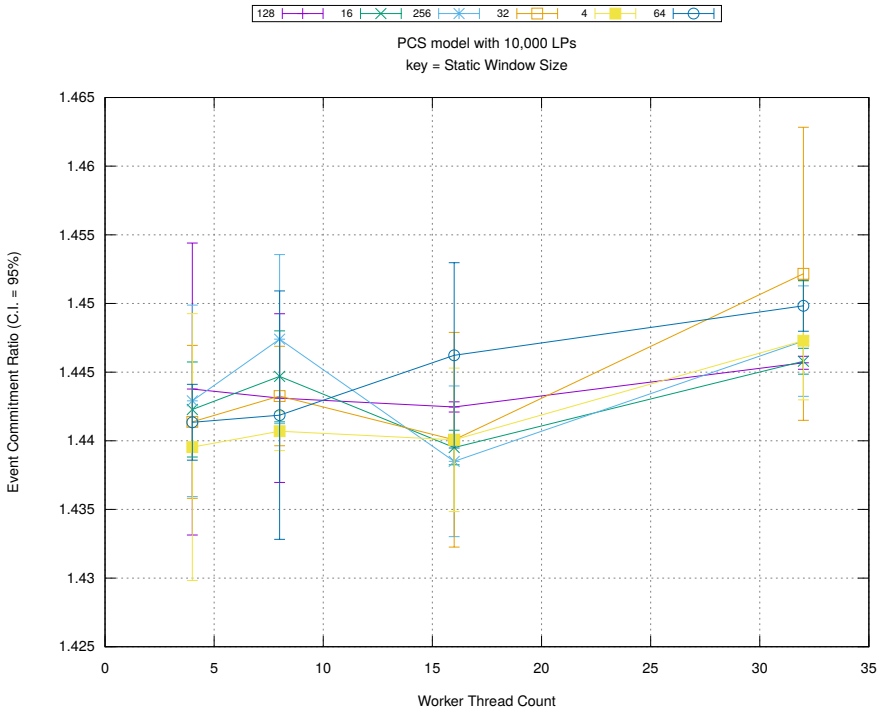
Figure A.229: pcs 10k/plots/scheduleq/threads vs count key type unsorted-bottom 16



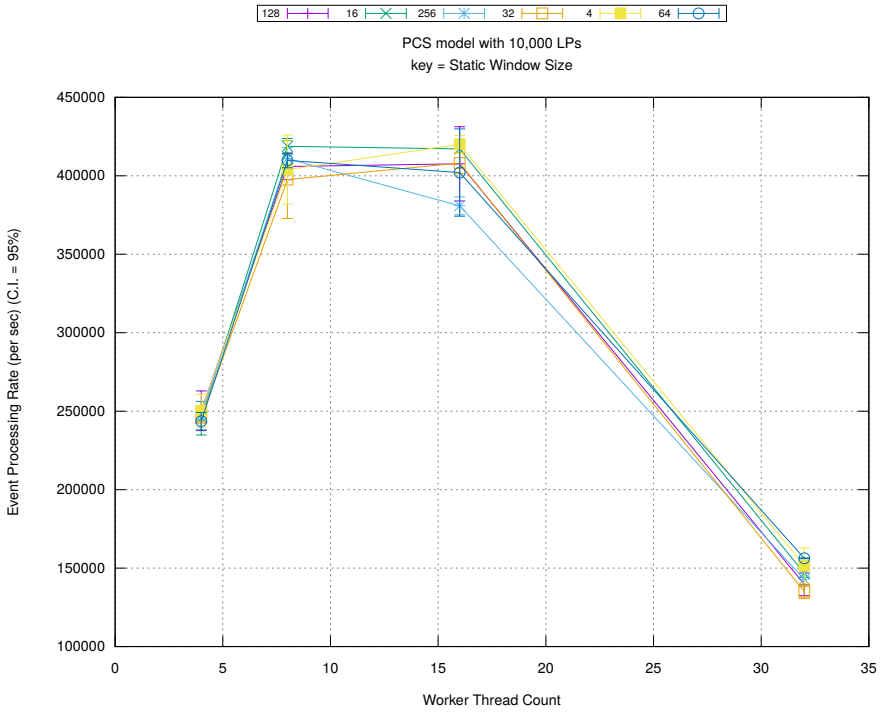
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

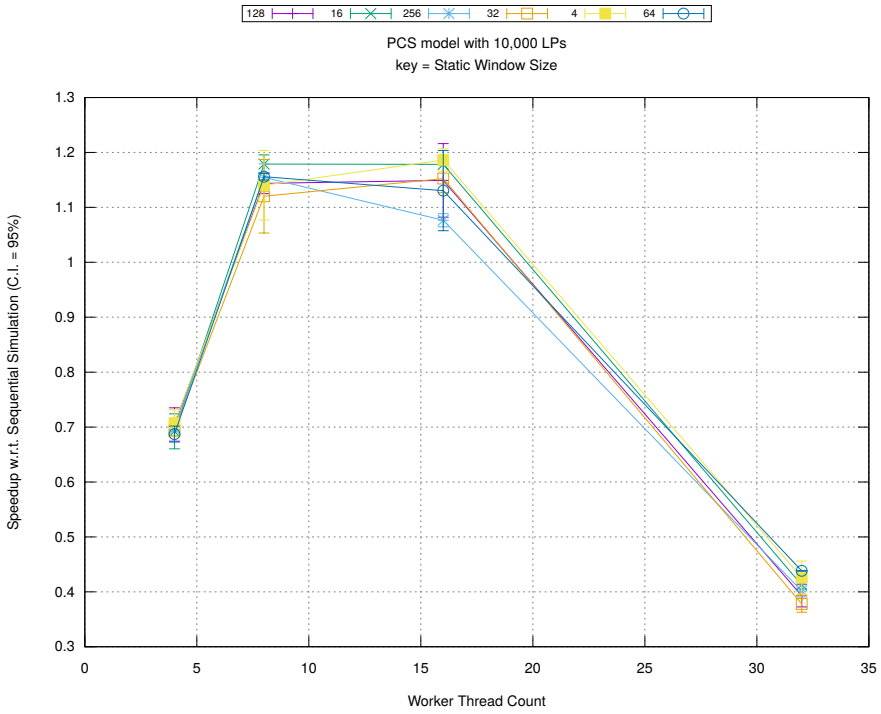


(b) Event Commitment Ratio

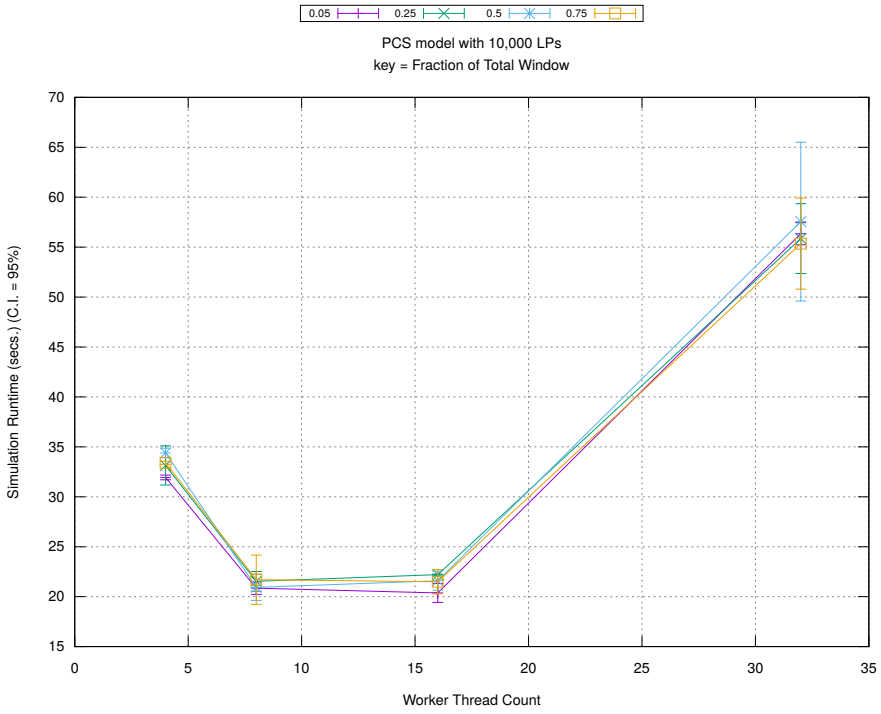


(c) Event Processing Rate (per second)

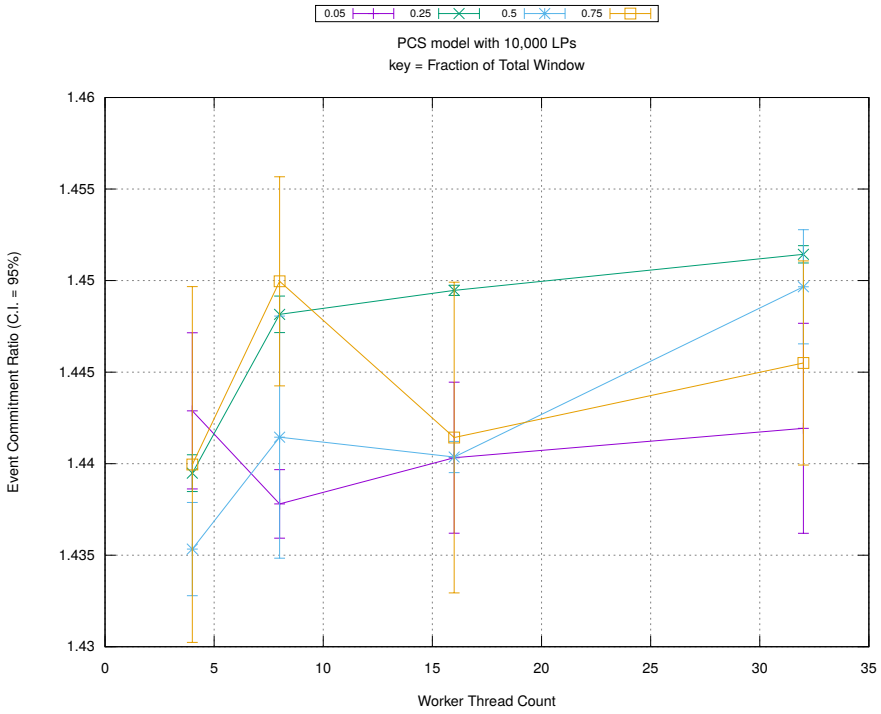
Figure A.230: pcs 10k/plots/bags/threads vs staticwindow key fractionwindow 1.0



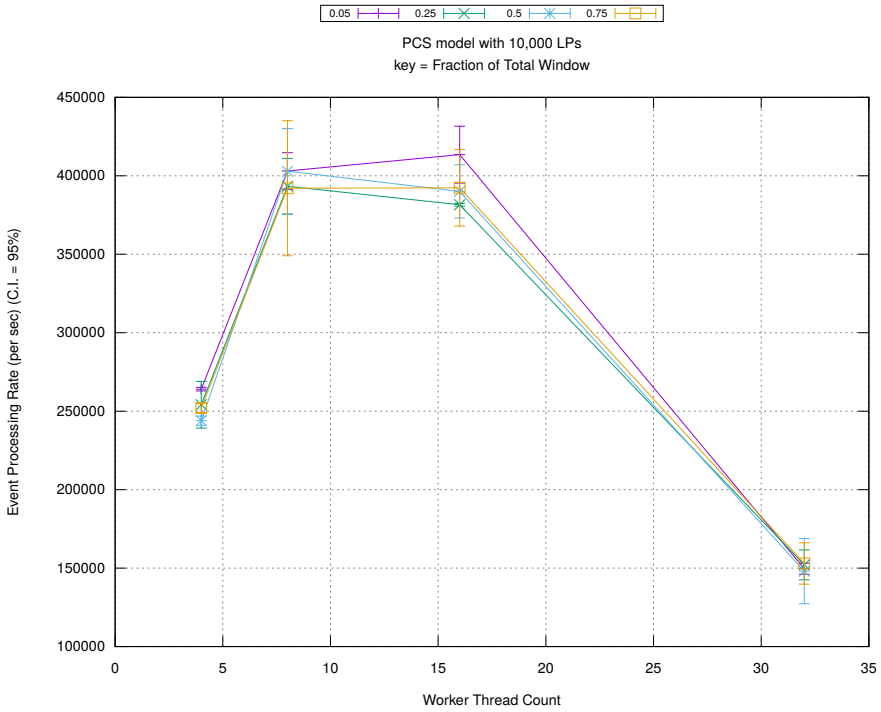
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

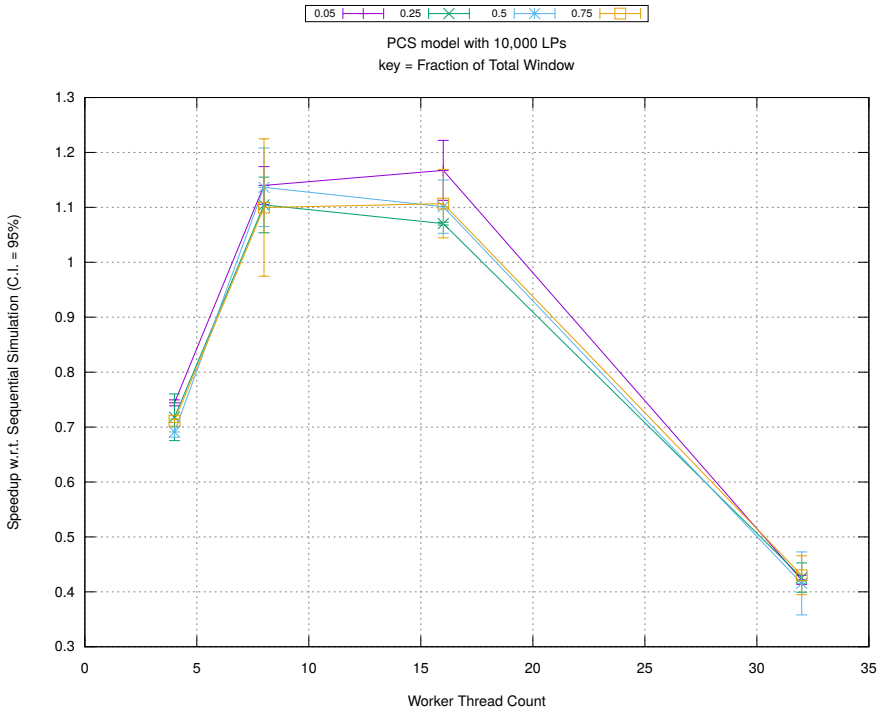


(b) Event Commitment Ratio

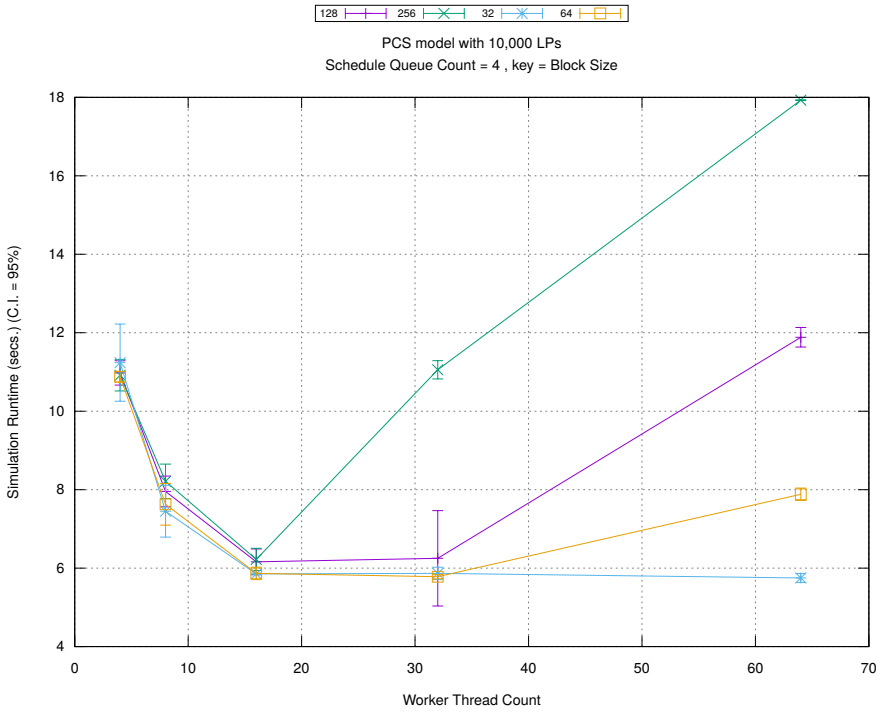


(c) Event Processing Rate (per second)

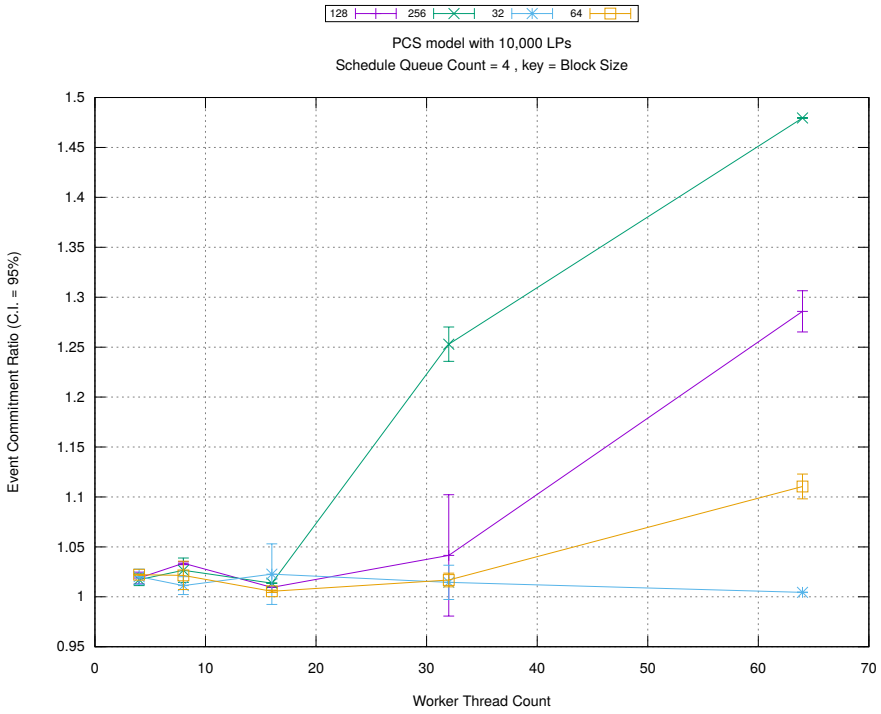
Figure A.231: pcs 10k/plots/bags/threads vs fractionwindow key staticwindow 0



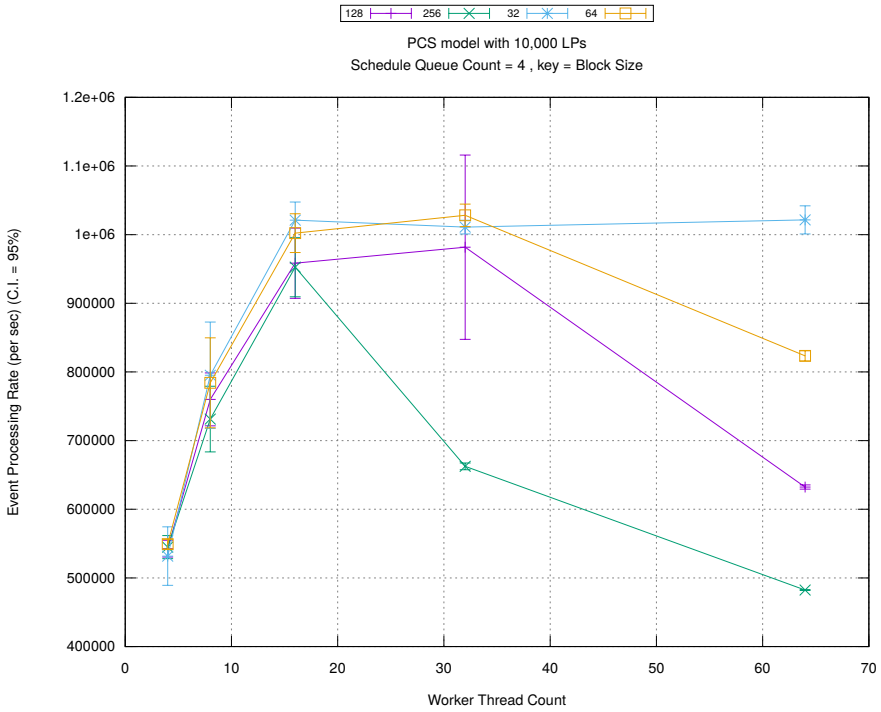
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

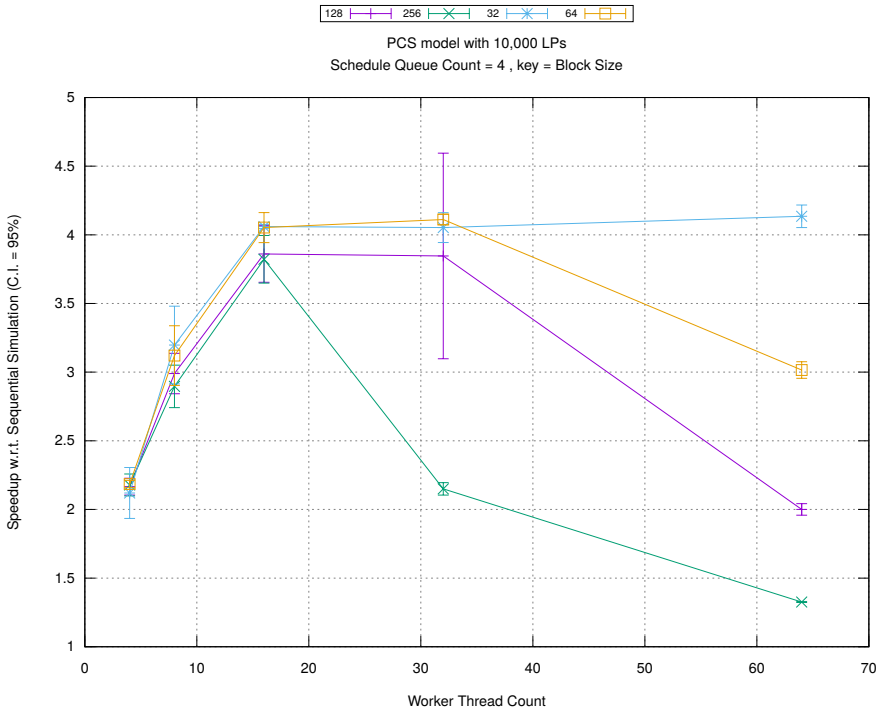


(b) Event Commitment Ratio

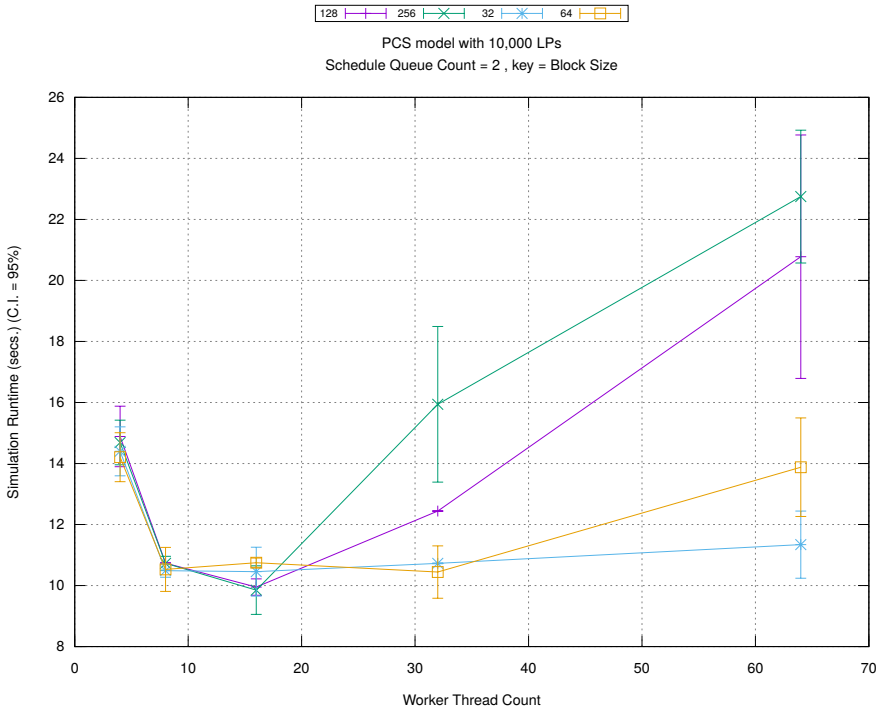


(c) Event Processing Rate (per second)

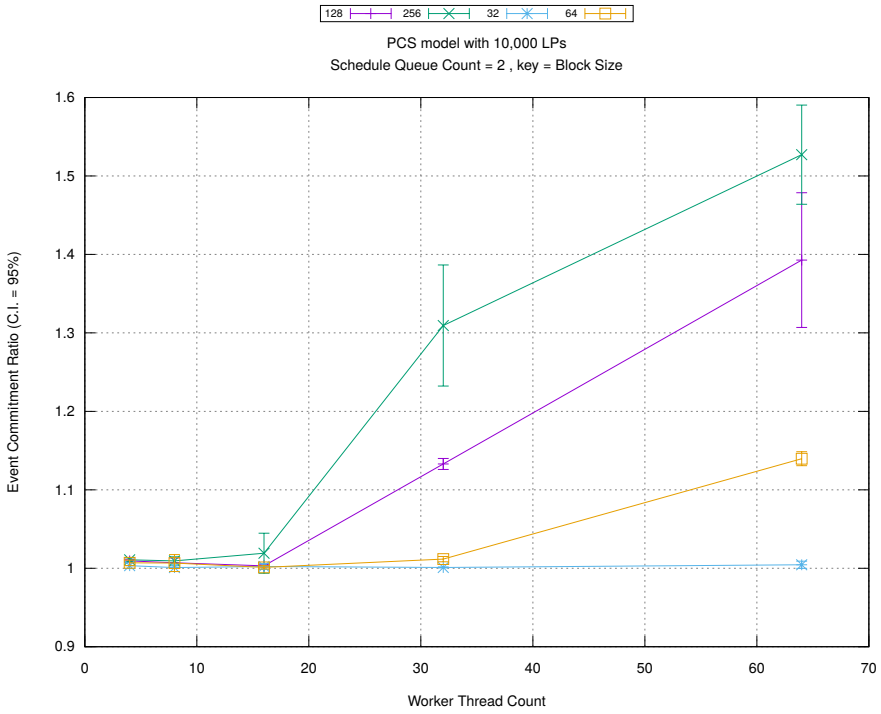
Figure A.232: pcs 10k/plots/blocks/threads vs blocksize key count 4



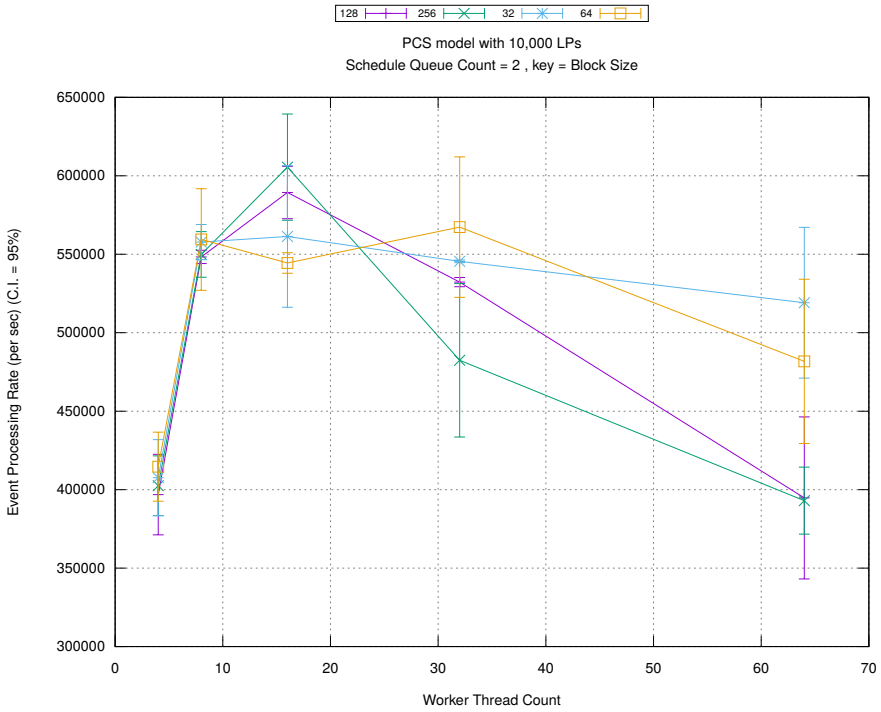
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

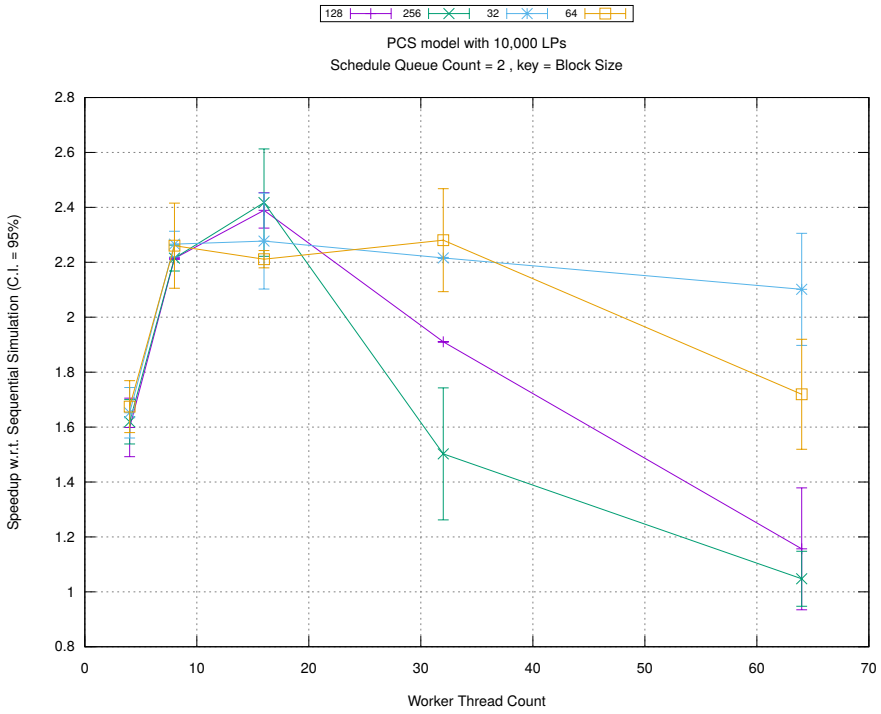


(b) Event Commitment Ratio

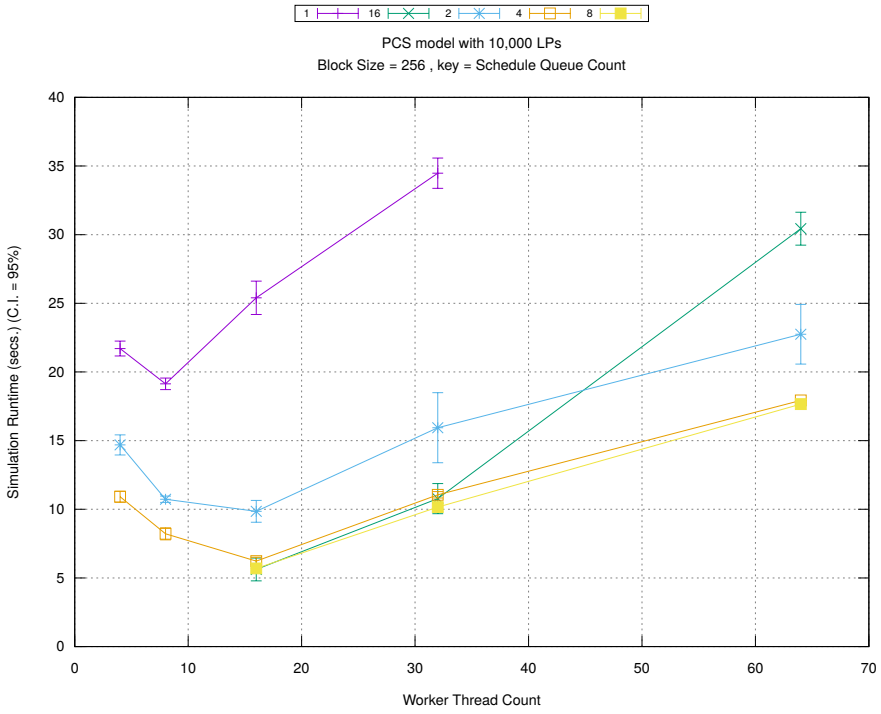


(c) Event Processing Rate (per second)

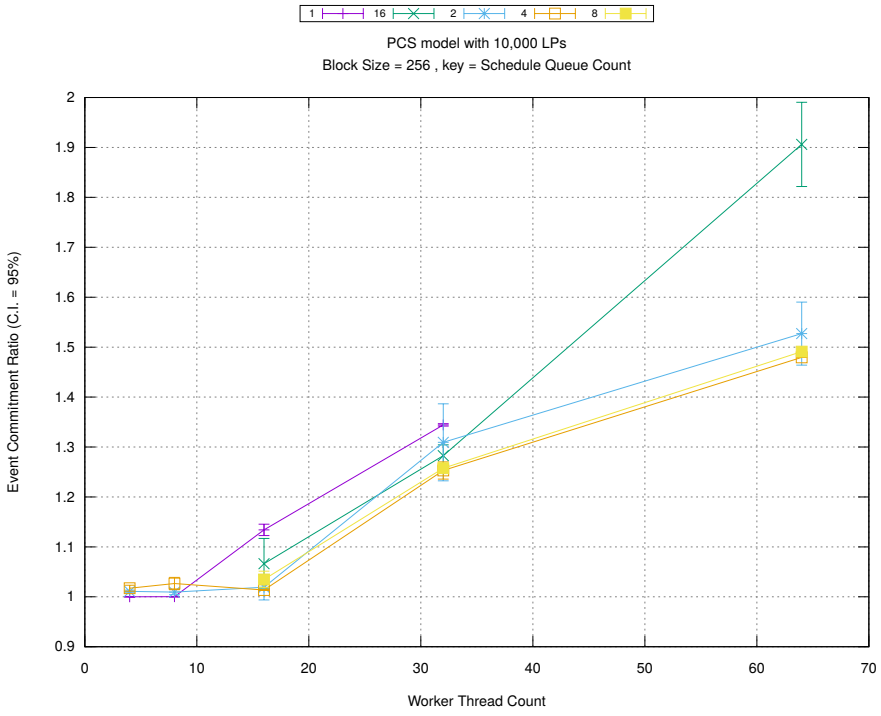
Figure A.233: pcs 10k/plots/blocks/threads vs blocksize key count 2



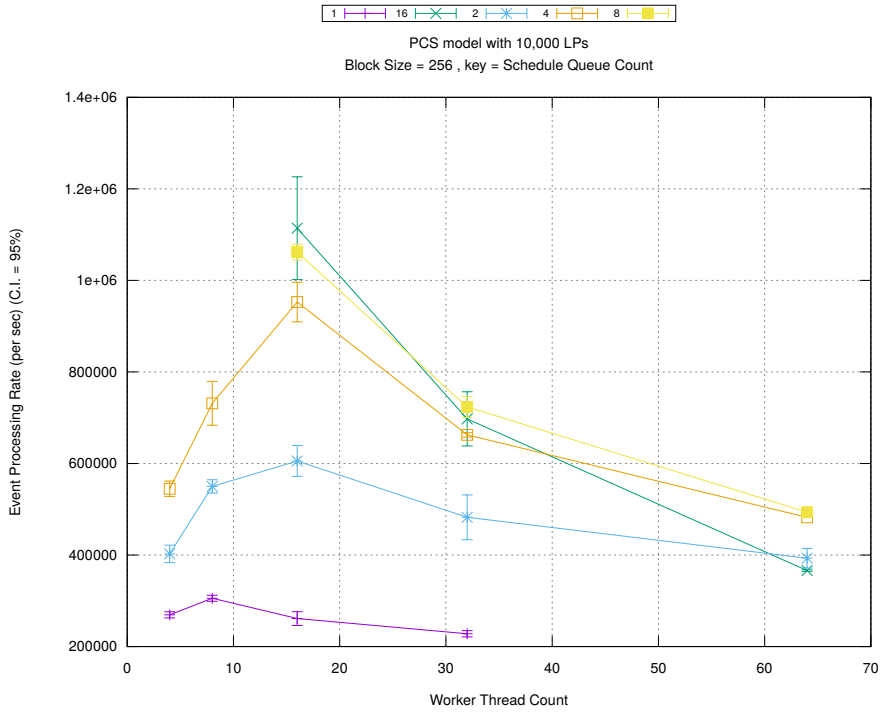
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

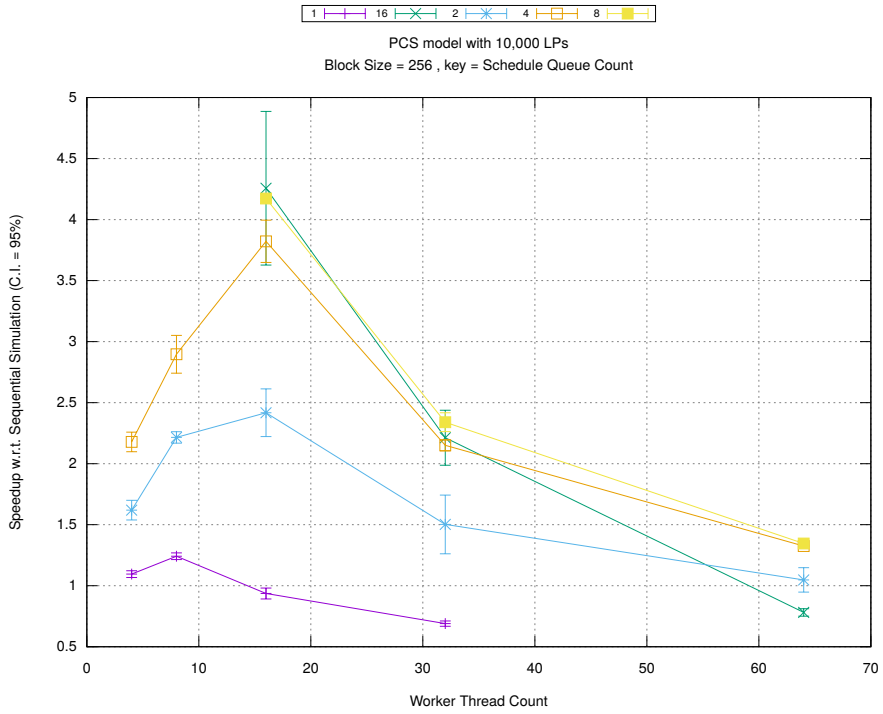


(b) Event Commitment Ratio

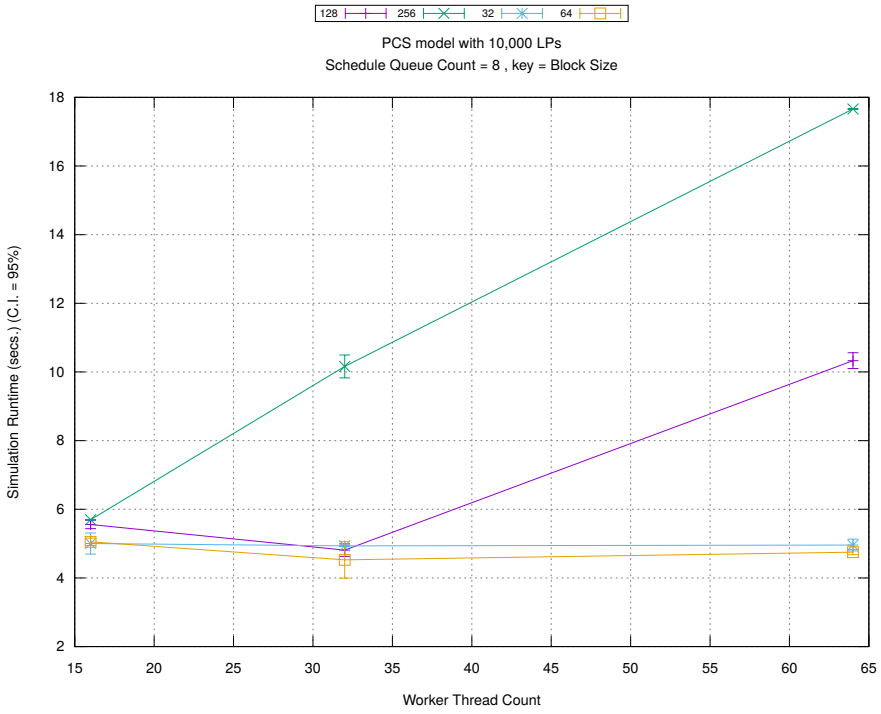


(c) Event Processing Rate (per second)

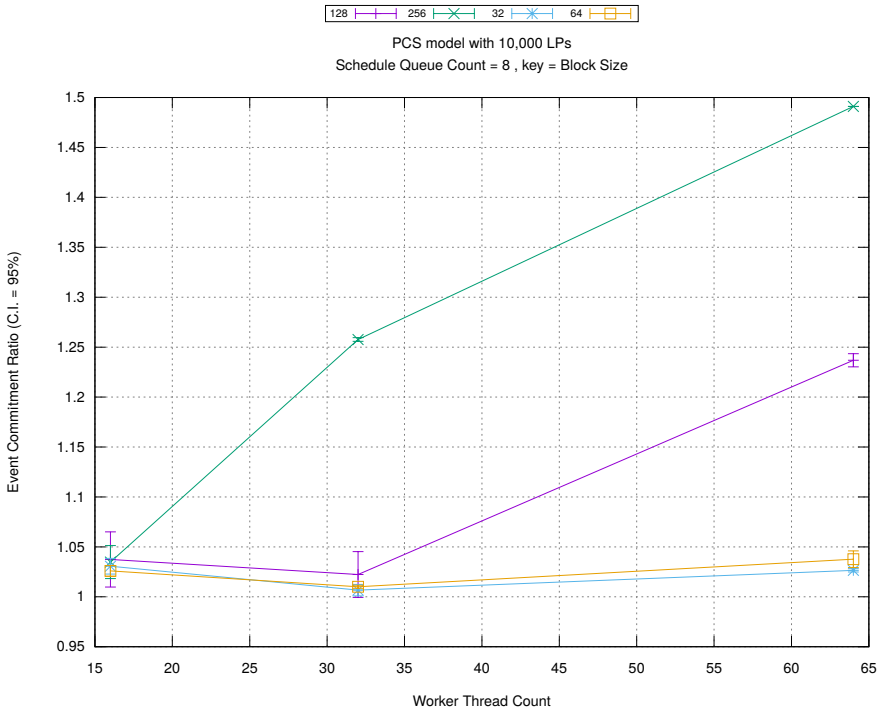
Figure A.234: pcs 10k/plots/blocks/threads vs count key blocksize 256



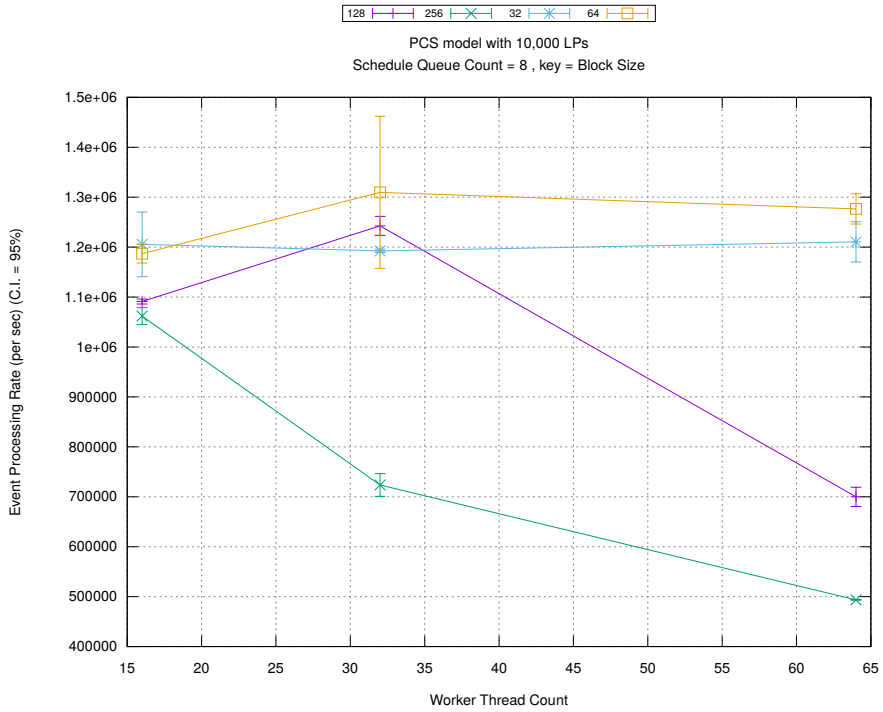
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

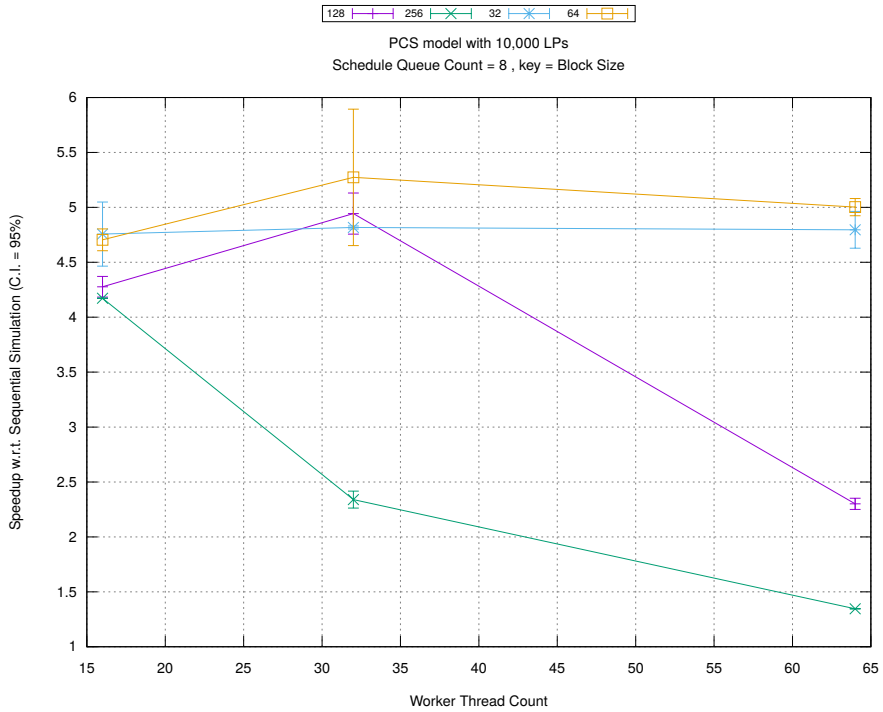


(b) Event Commitment Ratio

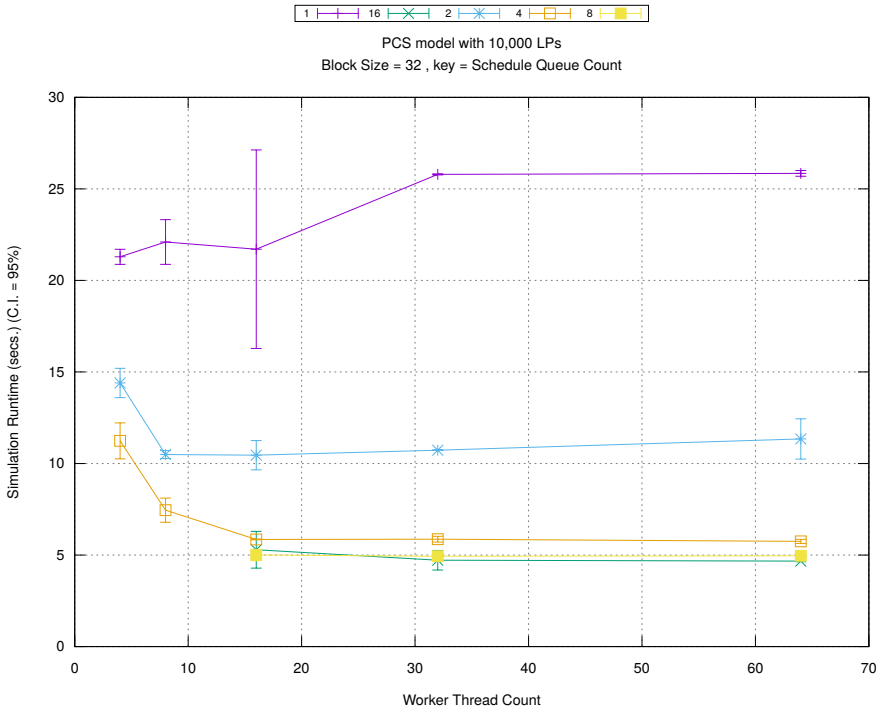


(c) Event Processing Rate (per second)

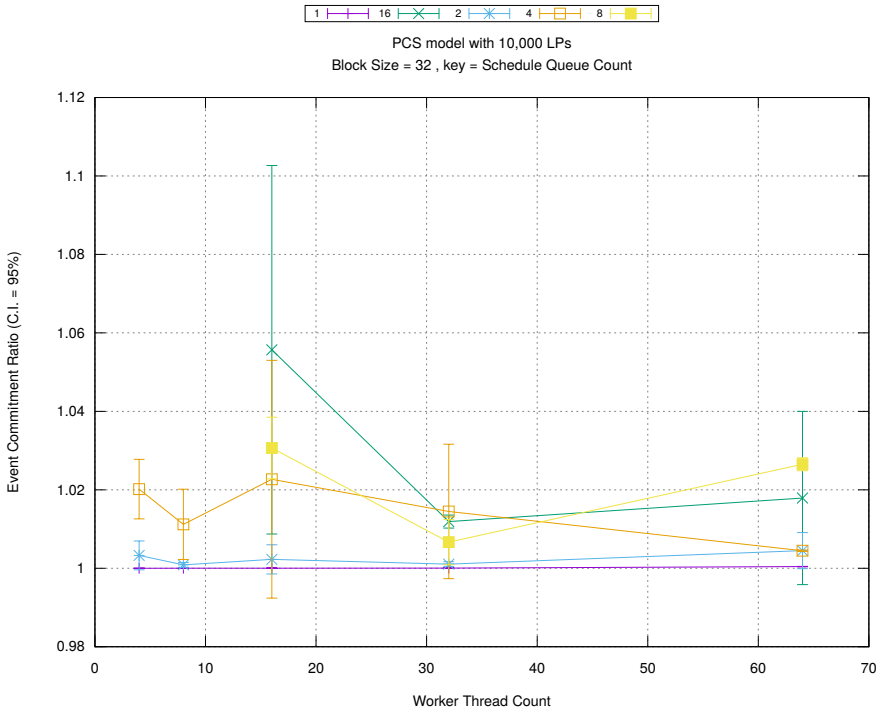
Figure A.235: pcs 10k/plots/blocks/threads vs blocksize key count 8



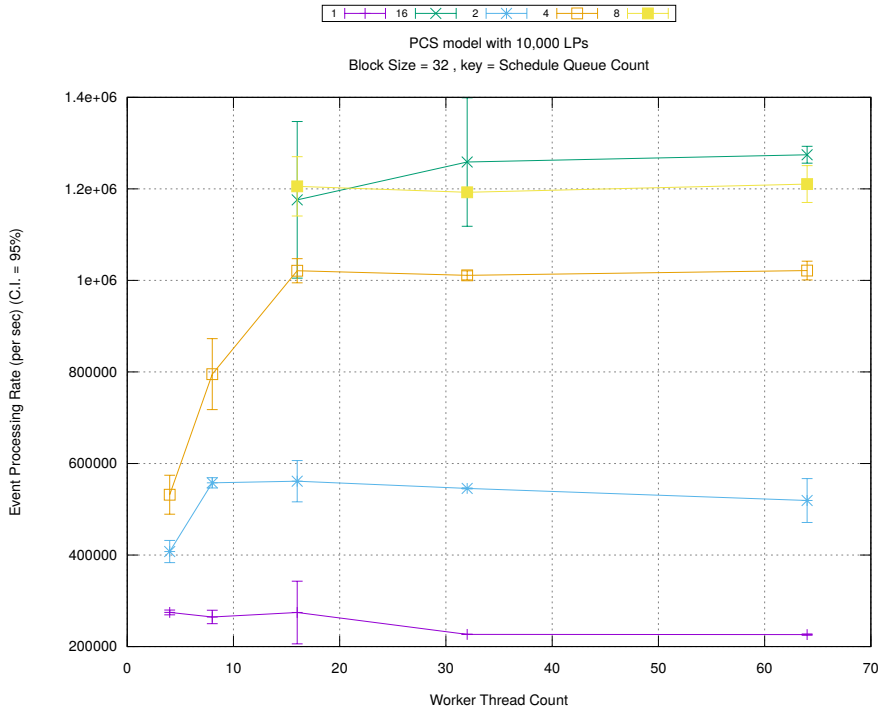
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

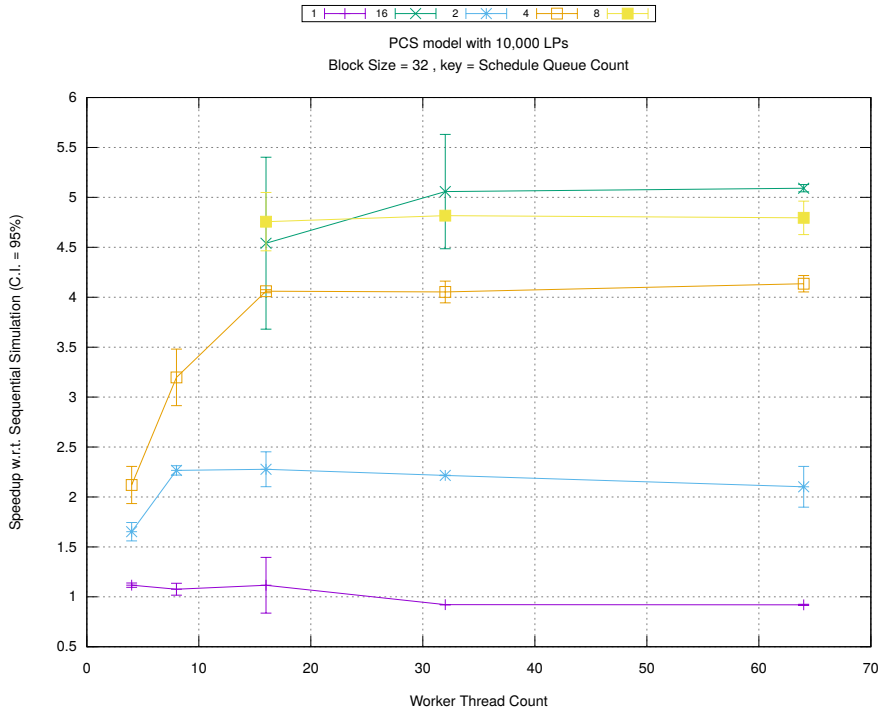


(b) Event Commitment Ratio

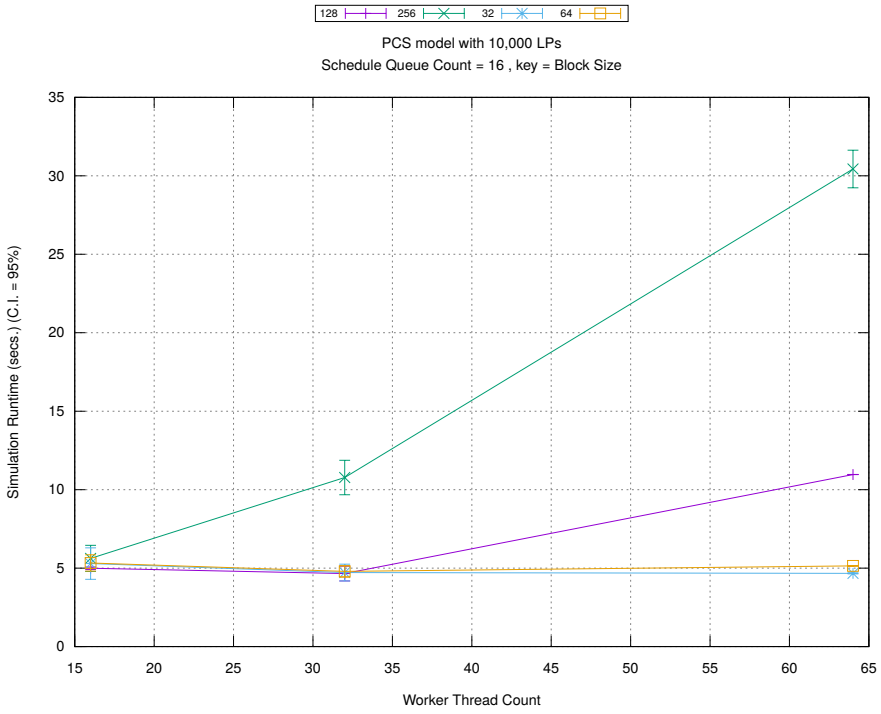


(c) Event Processing Rate (per second)

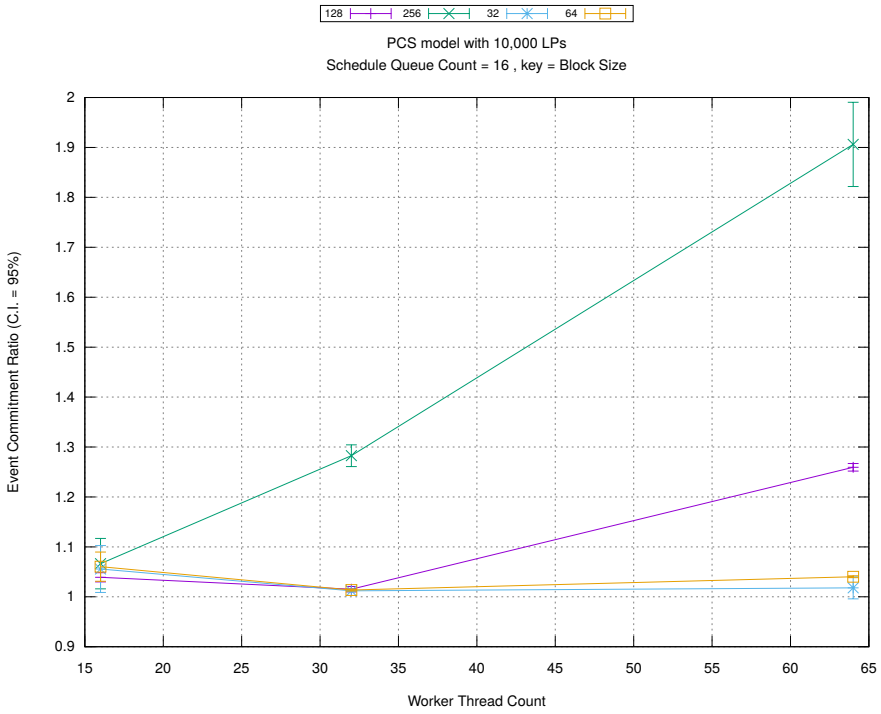
Figure A.236: pcs 10k/plots/blocks/threads vs count key blocksize 32



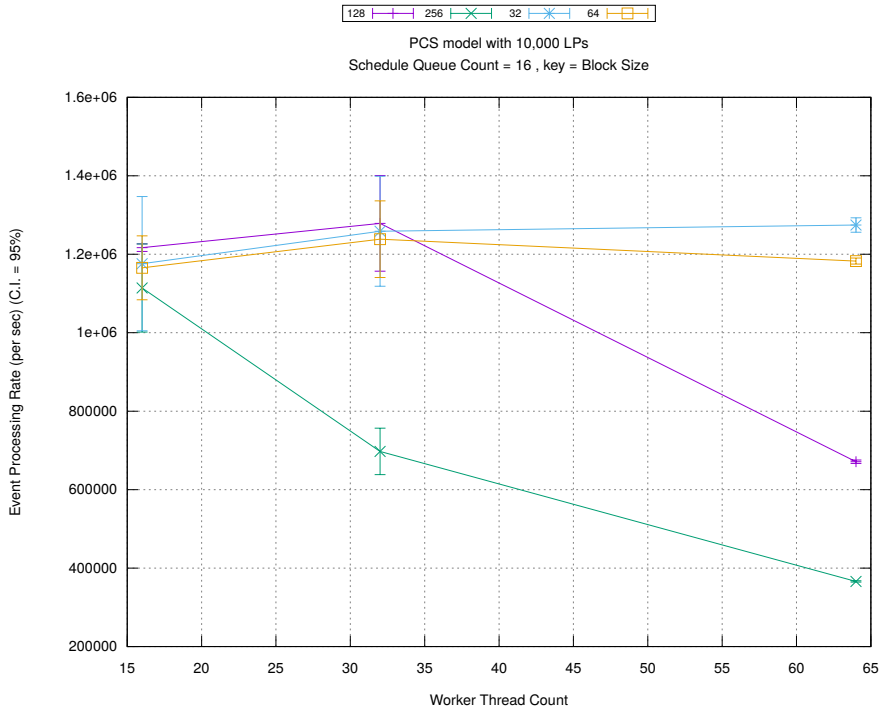
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

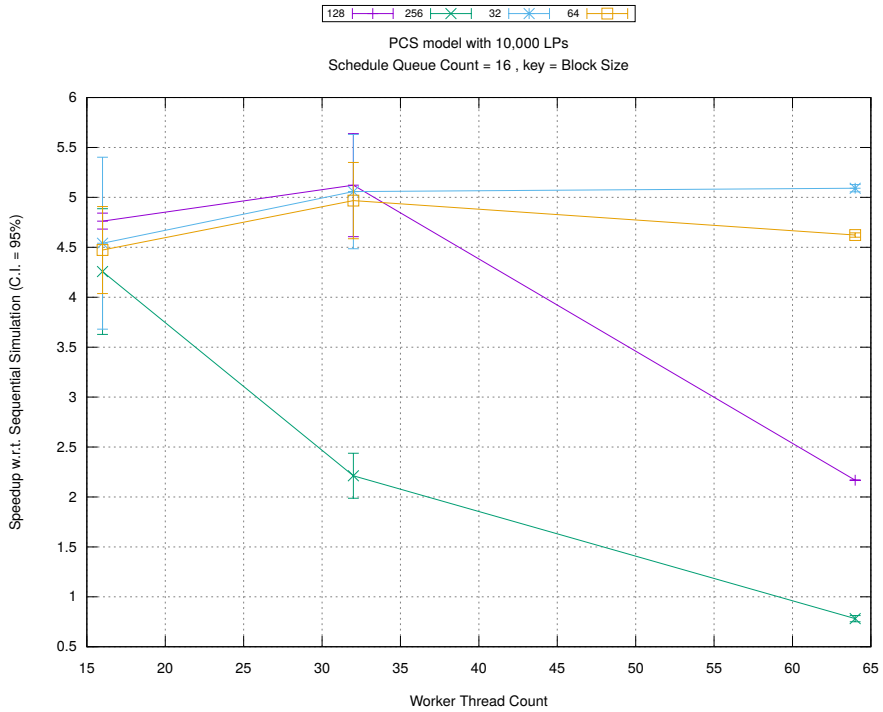


(b) Event Commitment Ratio

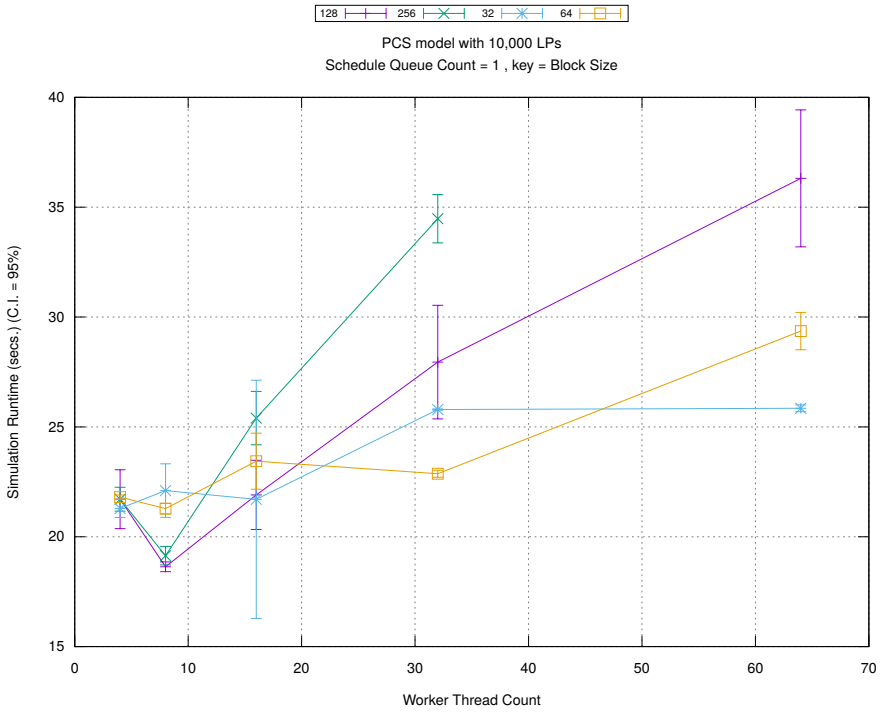


(c) Event Processing Rate (per second)

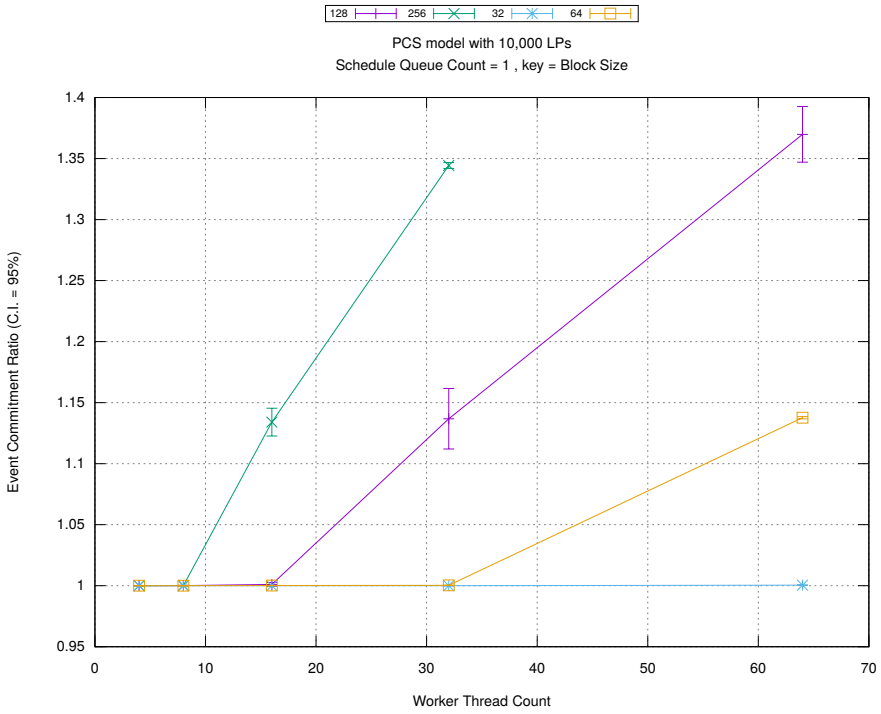
Figure A.237: pcs 10k/plots/blocks/threads vs blocksize key count 16



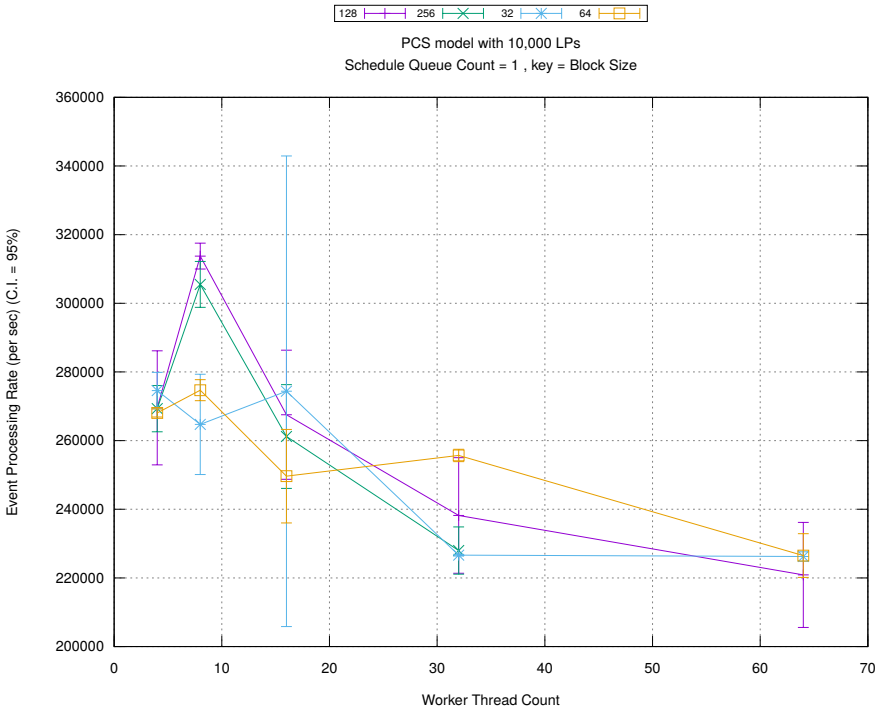
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

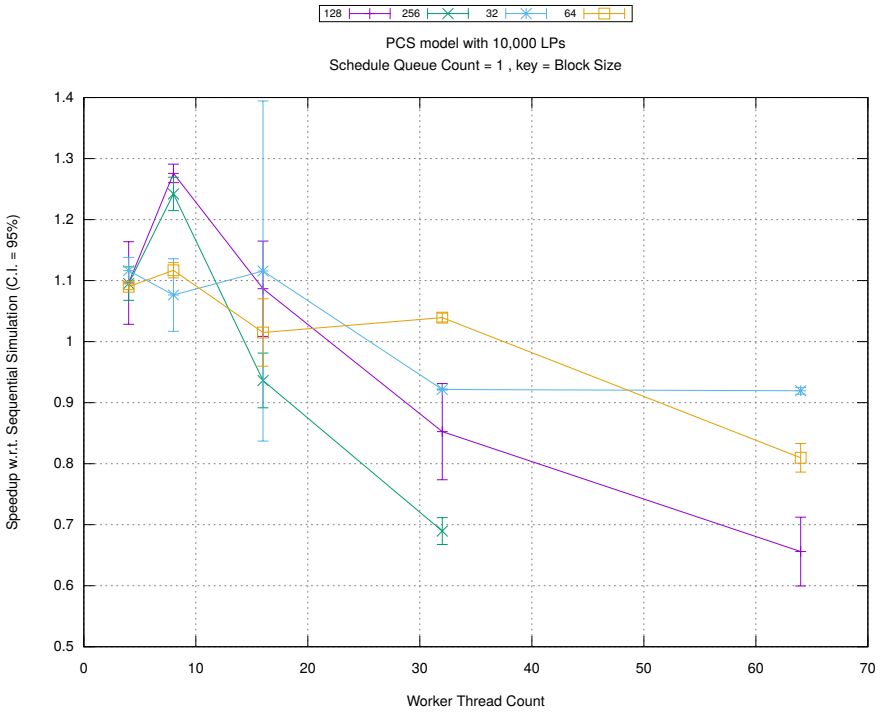


(b) Event Commitment Ratio

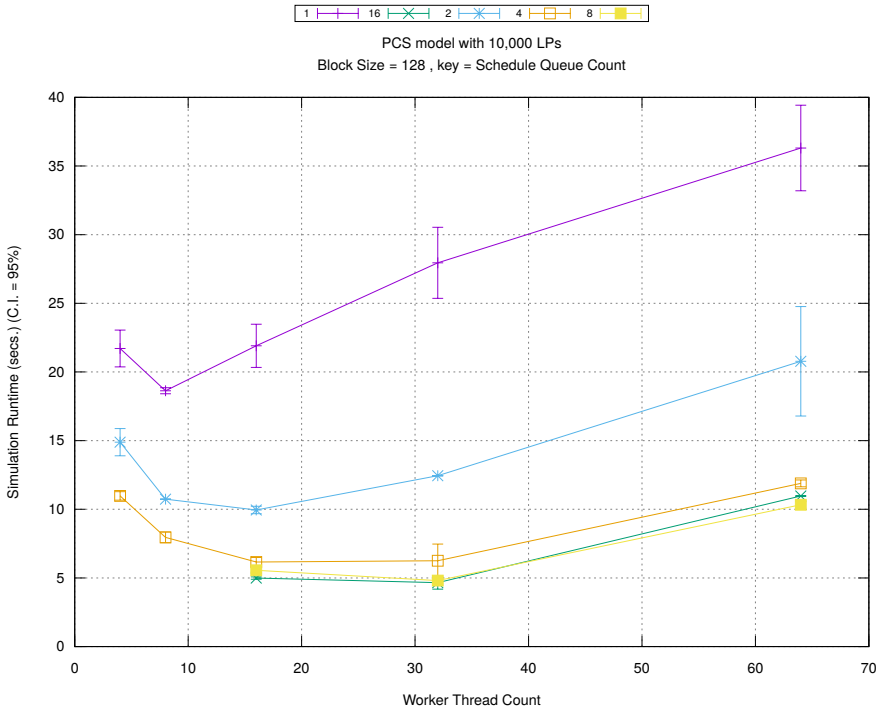


(c) Event Processing Rate (per second)

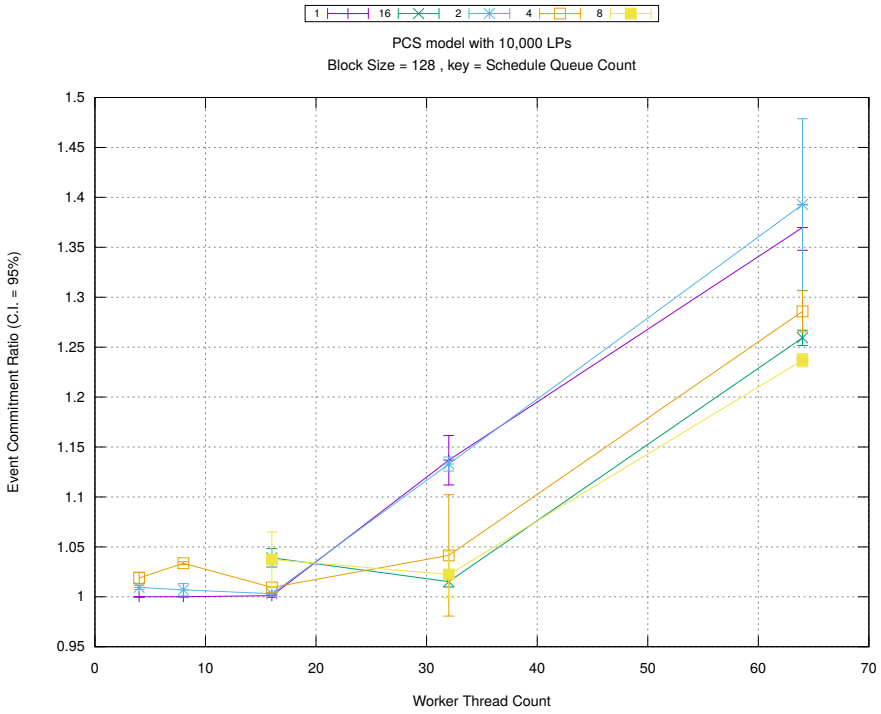
Figure A.238: pcs 10k/plots/blocks/threads vs blocksize key count 1



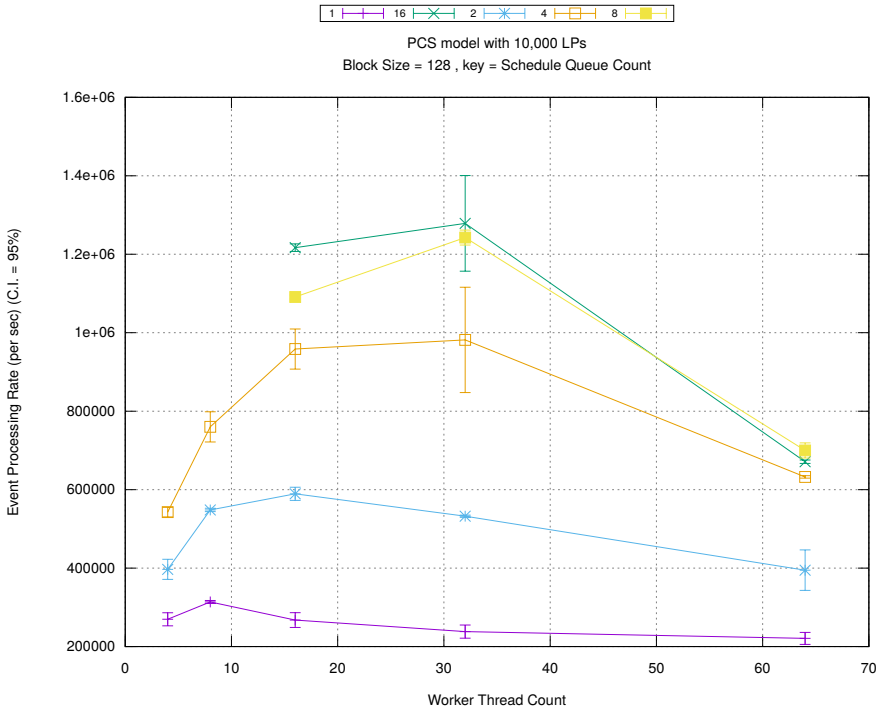
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

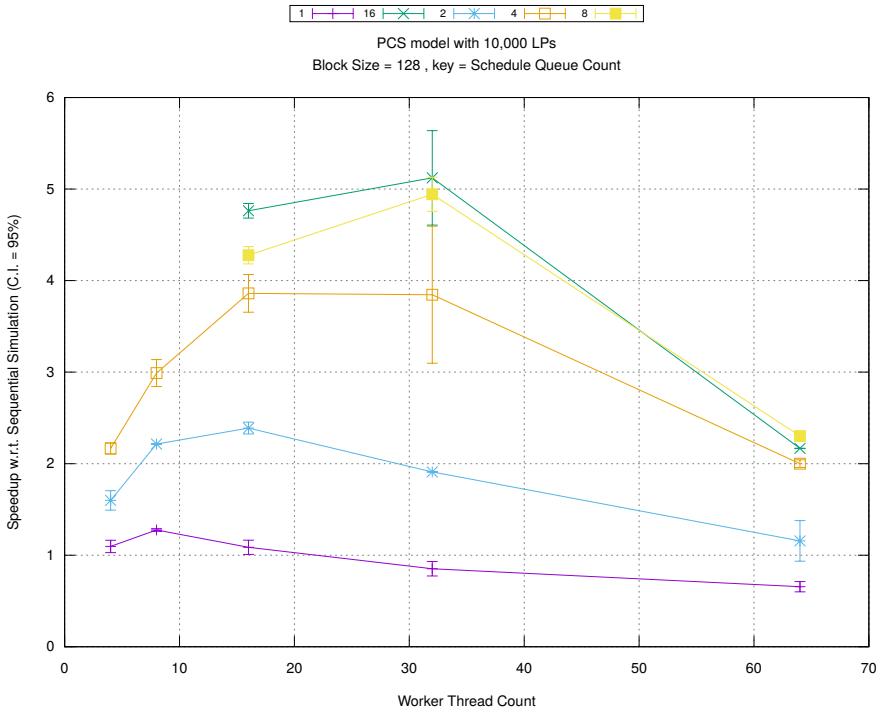


(b) Event Commitment Ratio

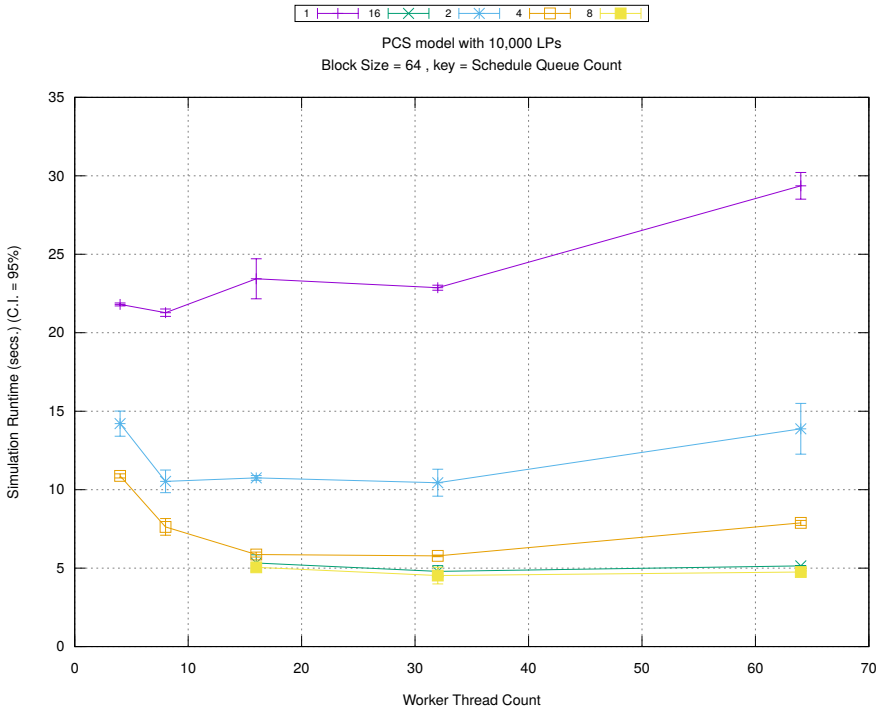


(c) Event Processing Rate (per second)

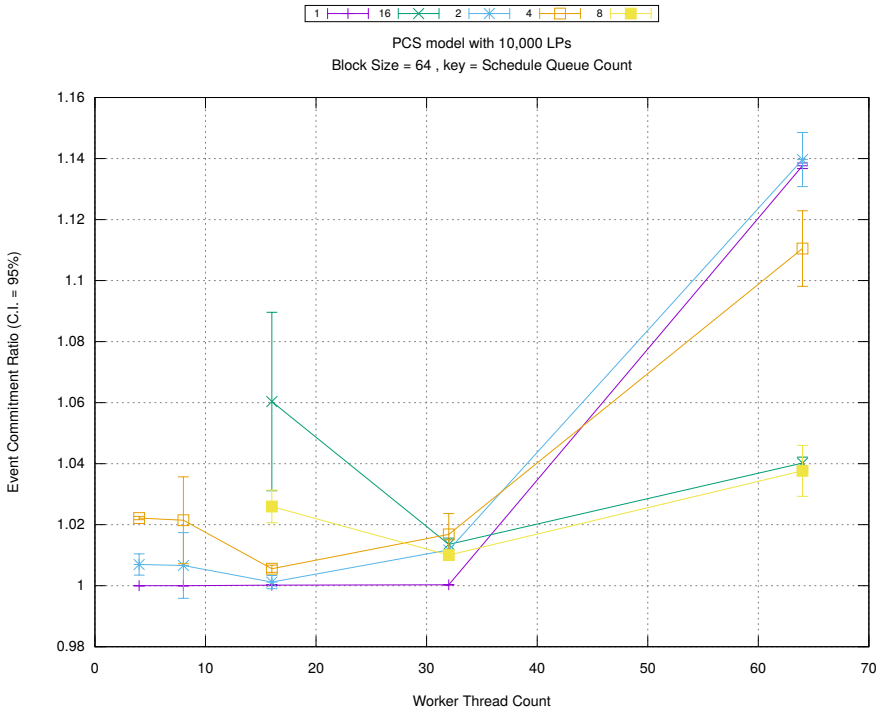
Figure A.239: pcs 10k/plots/blocks/threads vs count key blocksize 128



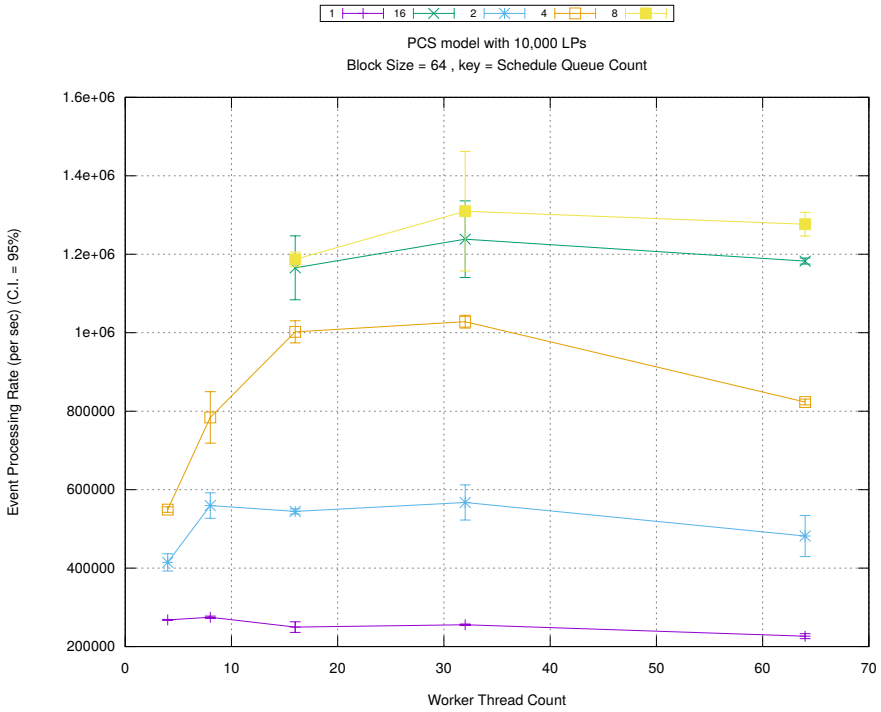
(d) Speedup w.r.t. Sequential Simulation



(a) Simulation Runtime (in seconds)

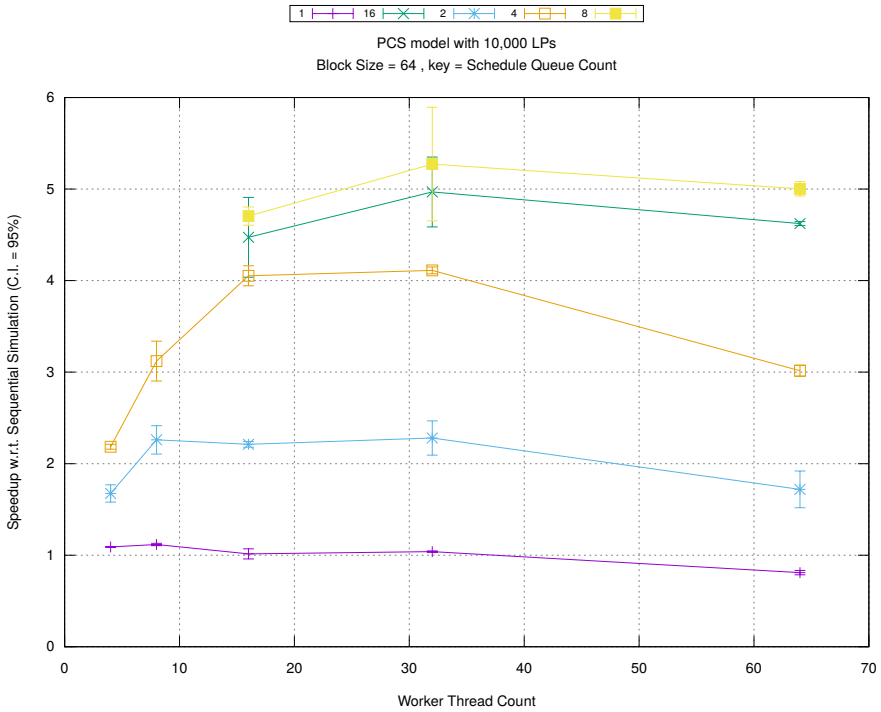


(b) Event Commitment Ratio



(c) Event Processing Rate (per second)

Figure A.240: pcs 10k/plots/blocks/threads vs count key blocksize 64



(d) Speedup w.r.t. Sequential Simulation