A MULTIPLE ASSOCIATIVE COMPUTING MODEL TO SUPPORT THE EXECUTION OF
DATA PARALLEL BRANCHES USING THE MANAGER-WORKER PARADIGM

A dissertation submitted
to Kent State University in partial
fulfillment of the requirements for the
degree of Doctor of Philosophy

by
Wittaya Chantamas

December, 2009
Dissertation written by

Wittaya Chantamas

B.S., Kasetsart University, 1996
M.S., Syracuse University, 1999
Ph.D., Kent State University, 2009

Approved by

Johnnie W. Baker, Chair, Doctoral Dissertation Committee
Robert A. Walker, Members, Doctoral Dissertation Committee
Mikhail Nesterenko
Andrew Tonge
Mark Kershner

Accepted by

Robert A. Walker, Chair, Department of Computer Science
John Stalvey, Dean, College of Arts and Sciences
# TABLE OF CONTENTS

LIST OF FIGURES ...........................................................................................................viii

ACKNOWLEDGEMENTS ....................................................................................................xii

Chapter 1 Parallel Computing: Introduction ...............................................................1
  1.1 The Need for Parallel Computing .........................................................................1
  1.2 Research Motivation and Accomplishments .....................................................4
  1.3 Organization of Dissertation ..............................................................................6

Chapter 2 Preliminary .................................................................................................8
  2.1 Terminology .........................................................................................................8
    2.1.1 Flynn’s Taxonomy and Classification .............................................................8
      2.2.1.1 Single Instruction Stream Single Data Stream (SISD) .........................9
      2.1.1.2 Multiple Instruction Streams Single Data Stream (MISD) .................9
      2.1.1.3 Multiple Instruction Streams Multiple Data Streams (MIMD) ...........10
      2.1.1.4 Single Instruction Stream Multiple Data Streams (SIMD) ...............11
    2.1.2 Control and Data Parallelisms ....................................................................12
    2.1.3 Speedup ......................................................................................................13
2.1.4 Amdahl’s Law .............................................................13

2.2 Related Models of Computation and Architectures .............................................13

2.2.1 SIMD/MIMD Models and Architectures .........................................................14

2.2.2 Parallel Random Access Machine (PRAM) ..................................................16

2.2.3 The Associative Computing (ASC) Model .....................................................18

2.2.4 Reconfigurable Models ..............................................................19

2.2.4.1 Reconfigurable Mesh (RM) .................................................................20

2.2.4.2 Reconfigurable Multiple Bus Machine (RMBM) ....................................21

Chapter 3 A Multiple Associative Computing (MASC) Model ..............................24

3.1 Introduction .........................................................................................24

3.2 Components of the MASC Model ..........................................................25

3.3 Associative Properties and Operations .........................................................29

3.3.1 Global AND and OR Reductions ...............................................................30

3.3.2 Global MAX and MIN Reductions .............................................................30

3.3.3 An Associative Search Operation ..............................................................31

3.3.4 Any-Responder Operation ......................................................................31

3.3.5 Pick-One Operation ..............................................................................31
3.4 A MASC Model Using the Manager-Worker Instruction Stream Paradigm ..........32

3.5 Comparing the Power of a Basic MASC with a MASC with the Manager-worker
Paradigm ............................................................................................................34

Chapter 4 Comparing the Power of the Model with Other Computational Models ..........37

4.1 Introduction ................................................................................................37

4.2 Word Size ................................................................................................39

4.3 Simulations between MASC and B-RMBM ..................................................39

4.4 Simulation between MASC and S-RMBM ....................................................43

4.5 Simulation between MASC and F-RMBM, E-RMBM ....................................48

4.6 Relating MASC \(_c\) and MASC \(_{\log n}\) Using the PRAM Model ......................52

4.7 Extension of Relationships among MASC, PRAM, and RM .........................55

Chapter 5 A MASC Software Simulator Using the Manager Worker Paradigm ............59

5.1 Introduction ................................................................................................59

5.2 Previous Work .............................................................................................59

5.2.1 Previous Work on an ASC Language Compiler Extension ..........................60

5.2.2 Previous Work on a MASC Runtime System .............................................64

5.3 The MASC Simulator ....................................................................................65
5.3.1 Tasks and Task Pools .................................................................66

5.3.2 Simulating the Hardware Aspect of the MASC Model ..................67

5.3.3 Simulating the Software Aspect of the MASC Model ..................70

5.3.4 How the Simulator Determines the Precise Cycle Count .................71

Chapter 6 A MASC Floyd-Warshall All-pairs Shortest Path Algorithm ........76

6.1 Introduction ....................................................................................76

6.2 The Sequential Algorithm ..............................................................76

6.3 A MASC Floyd-Warshall Algorithm .................................................80

6.3.1 Expressing Parallelism for the Floyd-Warshall Algorithm .............81

6.3.2 A MASC Floyd-Warshall Algorithm ..........................................83

6.4 Simulation Results for the MASC Floyd-Warshall Algorithm ..........90

6.4.1 Problem Instances Used in the Simulator ....................................90

6.4.2 The Performance of the MASC Algorithm on the Simulator ..........92

6.4.3 The Performance Prediction of the MASC Algorithm on the Simulator ....96

6.5 Conclusion .....................................................................................98

Chapter 7 A MASC Quickhull Algorithm to Identify Points of a Convex Hull ........99

7.1 Introduction .....................................................................................99
LIST OF FIGURES

Figure 1: Changes in Intel CPU clock frequency .........................................................2

Figure 2.1: A diagram of a PRAM model .................................................................16

Figure 2.2: A diagram of an ASC model ...............................................................18

Figure 2.3: Structure of a 3x5 R-Mesh .................................................................20

Figure 2.4: A simple bus configuration of a 4x4 LR-Mesh ........................................21

Figure 2.5: A S-RMBM model with three processors .............................................22

Figure 3.1: A MASC model shown with all of its components ................................25

Figure 3.2: A broadcast and reduction network with its network processors attached ....27

Figure 3.3: A MASC model using the manager-worker paradigm .........................32

Figure 4.1: PE grouping to simulate a bus-access operation of the S-RMBM ..............44

Figure 4.2: Read step of simulating an S-RMBM on a MASC (p2 reads from bus g) ......47

Figure 4.3: Illustration of Step 4 -- (a) Step 4.1 (b) Step 4.2 (c) Step 4.3 (d) Step 4.4 ......50

Figure 5.1: The process of compiling and executing of a conventional ASC program ....60

Figure 5.2: The process of compiling an ASC program with MASC directives .............62
Figure 5.3: An ASC program with no MASC directive used ........................................62
Figure 5.4: The same ASC program as in Figure 5.3 with a MASC directive used ..........63
Figure 5.5: A simulation cycle of the manager instruction stream ...............................67
Figure 5.6: States of a Worker .........................................................................................69
Figure 6.1: Path p is a shortest path from vertex v_i to vertex v_j .................................78
Figure 6.2: The adjacency matrix of a 4-vertex input graph is divided into 4^2 elements ....81
Figure 6.3: Columnwise block-striped decomposition of an adjacency matrix ..............82
Figure 6.4: Rowwise block-striped decomposition of an adjacency matrix .................82
Figure 6.5: The R-MAT model shown with partitions a, b, c, and d .............................91
Figure 6.6: Results of the MASC implementation of the Floyd-Warshall algorithm .........92
Figure 6.7: Worker utilization in the simulator .................................................................93
Figure 6.8: The Floyd-Warshall algorithm execution time in steps by the MASC simulator on 8 to 64 vertex input R-MAT graphs .............................................................94
Figure 6.9: The speedup of the MASC model over an ASC model executing the Floyd-Warshall algorithm on 8 to 64 vertex input R-MAT graphs .................................95
Figure 7.1: A set of points S and its convex hull ..............................................................100
Figure 7.2: Point p is not extreme because it lies inside triangle (p1, p2, p3) .................100
Figure 7.3: Points w, r, and e subdivide the set S1 and eliminate all points in the yellow shaded triangle form further consideration .................................................................101

Figure 7.4: Points of set S in the plane are mapped to PEs (one point per PE) .......................105

Figure 7.5: After two extreme points, w and e, were found, PEs are partitioned into two partitions, one (a) with points lying above the line passing through w and e and another (b) with points lying below the line .................................................................106

Figure 7.6: No more than two workers can be used simultaneously for this recursive tree ….117

Figure 7.7: Results of the MASC implementation of the MASC algorithm .....................118

Figure 7.8: The MASC algorithm execution time in steps by the MASC simulator on 256 to 2048 convex hull points and 2 to 32 worker instruction streams .................................120

Figure 7.9: The speedup of the MASC model over an ASC model executing the algorithm on 256 to 2048 convex hull points and 2 to 32 worker instruction streams .......................121

Figure 7.10: Worker utilizations of each worker of a MASC(2048,1+8) finding 2048 extreme points .................................................................122

Figure 7.11: The MASC execution times in steps to identify 256 convex hull points on various iteration numbers .................................................................124

Figure 7.12: The case of 512 convex hull points on various iteration numbers ................125

Figure 7.13: The case of 1024 convex hull points on various iteration numbers ..............125

Figure 7.14: The case of 2048 convex hull points on various iteration numbers ..............126
Figure 8.1: A MASC model shown with hierarchical manager instruction streams ..........130

Figure A.1: The C Compilation Model .................................................................144
ACKNOWLEDGEMENTS

I would like to thank you my parent for their unconditional love to me, Mom and Yui for their support and encouragement given to me throughout the years, and all people that have helped me getting to where I am at today. The special thank you goes to Dr. Johnnie Baker, my advisor, for his generosity and kindness. Without his long hours working with me and his helpful suggestions, I cannot imagine how I can get this dissertation done. Finally, thanks are due to all of my dissertation committee members for their time and useful suggestions.
CHAPTER 1

Parallel Computing: Introduction

1.1 The Need for Parallel Computing

The very basic definition of sequential computers is computers that can perform one operation at a time regardless of how fast an operation can be performed. Unlike a parallel computer, which can perform many operations at once, it is a collection of processing elements that communicate and cooperate to simultaneously solve large problem fast.

We might not even aware that we already use parallel processing on a daily basis. The skill comes naturally considering these everyday examples. At home, we cook dinner in parallel. At a supermarket, customers check out items in parallel. Cars are built in parallel in an auto assembly line. A reason why we prefer to do these in parallel is to save time and, possibly, cost.

Some people understand that what a parallel computer can do is simply what a sequential computer can do (but somewhat faster). An analogy is given as if a car can travel from city A to city B in an hour, using ten cars does not lead to a faster trip. This is partly true. Some problems are called inherently sequential and cannot be executed any quicker in parallel. Fortunately, not all problems existing in this world are inherently sequential problems. Most problems can be executed and solved faster on a parallel computer than on a sequential computer.

Technologies and processor architectures played important roles in the past to provide an increase in performance of sequential computers. Eventually, the technology will reach a physical
limit for the speed at which they process data and this speed cannot be increased further through using of faster components. As shown in Figure 1, the processing capacity of CPUs (Central Processing Units) seemed to stop improving around 2004; any further improvement after that comes from adding more cores into a CPU die. To obtain an optimal performance out of today technology, parallel computing is a reasonable way to go. Whatever the performance a single processor is given at a time, higher performance can be achieved using parallel computing by utilizing many of such processors.

Figure 1: Changes in Intel CPU clock frequency (Source: http://www.numtech.com, column 13, The Economics of Grid Computing, March 2007).
Some problems are considered as grand challenge problems, which are identified by the US High Performance Computing and Communication Program [14] [38]. These problems usually are important scientific and engineering research problems such as global change, molecular biology, applied fluid dynamics, and ecosystem simulations. The characteristics of the problems are, first, they cannot be unsolvable, second, they are demonstrably hard to solve and require several orders-of-magnitude improvement in the capability required to solve them, and, third, the solutions to the problems must have a significant economic or social impact. Since these problems require enormous data storage and computing power hardly achievable by today sequential computers, simply, they are not viable currently for any sequential computer.

Some problems cannot even be completed without using parallel computers regardless of how much time one is willing to wait for an answer. These problems usually occur if the amount of data or their values are function of time, intermediate results effect future inputs, the input consists of many streams, or the effect of a computation cannot be reversed [1]. The speed of parallel computers does not actually play any role here. It is rather the ability “to be in more than one place at a time” of multiple processors of parallel computers that guarantee a successful completion of a computation.

In conclusion, parallel computing is a key in order to enable technology. It allows us to fulfill the ever-increasing demand for higher and higher performance at lower costs and sustain productivity in real-life applications [23].
1.2 Research Motivation and Accomplishments

There are some questions that a designer of a parallel computational model should consider before introducing a new computational model are the following:

- Does the model have simple and easy to follow description?
- Is the model realistic and buildable using today or very near future technology?
- Does the model support wide range of applications?
- Does the model have a simple style of programming while maintaining its respectable performance and predictability?

This research focuses on the study of a scalable associative computational model called MASC (acronym for Multiple Associative Computing) model that satisfies all of the above requirements. The MASC model is a multi-SIMD (multi Single Instruction Stream Multiple Data Streams) model that is a generalization of an ASC (Associative Computing) model designed to support multiple ASC threads by using control parallelism to substantially improve the low processor utilization often criticized in SIMDs (Single Instruction Stream Multiple Data Streams). SIMDs have been built to take advantages of data parallelism. They can process large amount of data concurrently to achieve massive scale parallelism. Unfortunately, SIMDs have some limitations. These limitations cause SIMD model to be viewed as a special purpose model that will only be used in specific applications. One of the SIMD limitations is the restricted control organization (more details about the SIMD model will be mentioned in Chapter 2). If PEs (Processing Elements) require different control patterns such as those in conditional structures, each pattern must be broadcast by the control unit separately. Based on the extensive survey of
massively parallel applications done by Fox [18], more than a half of applications were loosely synchronous or asynchronous. These applications require a more flexible control in order to execute efficiently. As a result, MIMDs (Multiple Instruction Streams Multiple Data Streams) have been viewed as the only practical option for general purpose massively parallel processing. The MASC model combines the advantages of both SIMD and MIMD models such as simple model description, inherent synchronous operation, and ease of programming and debugging of SIMDs while providing (up to a certain degree) flexible control flow support of MIMDs with small thread synchronization overhead. Moreover, the model is designed to support a large class of algorithms with better efficiency (in some cases) than strict SIMDs with small thread synchronization overhead.

Partial research work of this dissertation has been published as one journal paper [54], one conference paper [8], and one workshop paper [9]. Currently, previously unpublished work is submitted to a major conference. Key accomplishments in this research can be summarized as follows.

- This research provides a model description of a variation of the MASC model that uses the manager and worker instruction stream paradigm--one manager is used to control and coordinate workers in order to execute multiple ASC threads. The description of the interaction between instruction streams of the MASC model is omitted from the original model description [35].

- This research establishes theoretical results for the MASC model to justify choices made such as those for instruction streams, PEs, and networks in the model.

- This research establishes relationships between the MASC model and other computational models such as reconfigurable models and the PRAM (Parallel Random
Access Machine) model so that algorithm designed for other computational models can be compared and adapted to run on the MASC model with ease.

- The research provides an example of an implementation of a runtime system for the MASC model using the manager-worker paradigm. A software simulator for MASC used to demonstrate the performance of the model on two multithreaded algorithms. The first is a MASC Floyd-Warshall all-pairs shortest path algorithm using a static task assignment scheme. The second is a MASC Quickhull algorithm to identify points of a convex hull using a non-static task assignment scheme.

1.3 Organization of Dissertation

This dissertation comprises eight chapters as follows.

Chapters 1 and 2 provide a first look at the need for parallel computing and problems studied in this research. Additionally, they introduce related parallel models and terminology used throughout the dissertation.

Chapter 3 discusses the MASC model. The basic version of the MASC model is introduced first. A MASC model that uses the manager and worker instruction stream paradigm is introduced next. The relationship between these models based on computing power and the ability to implement such a model is described.

The power of a computational model is indicated by the efficiency with which it can simulate other computational models and algorithms it supports. Chapter 4 evaluates the power of a basic MASC model by establishing relationships between the MASC model and other popular
parallel models that have been studied previously such as reconfigurable models and the PRAM model.

Chapter 5 discusses an implementation of a software runtime system of a MASC model that uses the manager-worker instruction stream paradigm.

Chapter 6 provides the first example of two multithreaded algorithms. The example is a MASC Floyd-Warshall all-pairs shortest path algorithm using a static task assignment scheme. A problem statement, a MASC algorithm, and its execution results of the simulator are presented.

Chapter 7 provides the second example of two multithreaded algorithms. The example is a MASC Quickhull algorithm to identify points of a convex hull using a non-static task assignment scheme. A problem statement, a MASC algorithm, and its execution results of the simulator are presented.

Chapter 8 concludes the dissertation and provides a future research possibility.

Bibliography contains the bibliography of material used in this research.

Three appendices are included at the end of this dissertation. Appendix A provides a guide on how to program on a MASC simulator. Appendix B is a list of all functions available in the MASC library. Appendix C includes a table showing the worst case cost of each function in the MASC library.
CHAPTER 2

Preliminary

2.1 Terminology

This section introduces and discusses some terminology used throughout this dissertation. The well-known Flynn’s taxonomy and commonly used parallel computing terms are presented first. Then, related models of computation are introduced next.

2.1.1 Flynn’s Taxonomy and Classification

Michael J. Flynn introduced his taxonomy for parallel computer based on the multiplicity of the instruction (or control) and data streams available in the architecture [16] [17]. Flynn’s classification scheme is not specific enough and may cause many experts to some disagreement as to how to classify certain architectures [42]. A neural network machine and a SIMD/MIMD hybrid configuration machine cannot be clearly categorized into one of the four categories. Handler [20] suggested a refined scheme with more details than Flynn’s but, at the end, it is shown to leave much to be desired since some parallel computers such as CM-5 do not enter into the scheme at all [51]. However, Flynn’s taxonomy is still one of the simplest and best-known ways of categorizing parallel computers to date. The classification identified four classes of computers as follows.
2.2.1.1 Single Instruction Stream Single Data Stream (SISD)

This class of computers corresponds to regular von Neumann sequential computers. An application is run on a single processor under a control of an instruction controller. The computer can execute one instruction at a time on one data item at a time and have no parallelism in either the instruction or data stream. Some SISD machines may support multitasking. If time multiplexing of tasks is fast enough, SISD computers may look like or appear to support concurrency, but this is not true parallelism. An example of computers in this class is a conventional uniprocessor PC.

2.1.1.2 Multiple Instruction Streams Single Data Stream (MISD)

The characteristic of computers in this class is that multiple instructions operate on a single data stream. Not many architectures fall into this category. Some may consider a certain computer system used for fault tolerance to be in this classification because these systems operate with the same instructions redundantly on the same data stream, as known as task replication, and must agree on the computing result.

2.1.1.3 Multiple Instruction Streams Multiple Data Streams (MIMD)

Parallel machines have been built along the following two classifications, MIMD and SIMD. The characteristic of MIMD computers is that multiple autonomous processors execute different instructions on different data simultaneously. Processors of a MIMD are usually
complete stand-alone serial processors. Each of them has all components such as its control unit and local memory necessary for independent processing of a task. MIMDs have the flexible control flow. They are able to support a wide variety of control flow patterns in applications. The control of a MIMD computer is distributed. A control signal can be generated locally within the processor chip. Combining this with its local memory, pipelining, and cache, a processor instruction cycle can be made smaller than that of a SIMD. As a result, this leads to fast execution cycles of MIMD processors. Moreover, since MIMD processors usually are off-the-shelf serial processors, these processors benefit from all of the advanced technology (or allow the system to be updated to newer technology when available) from processor manufacturers. MIMDs have some drawback as follows. MIMD processors requires some coordination that requires expensive communication in a form of synchronization, barriers, or locks in order to preserve the precedence of applications such as maintaining the consistency of shared data. MIMD machines are usually large (in physical size) and require some forms of the communication network that may span over long distances. The most criticized drawback is that MIMDs are difficult to program, debug, and optimize. This is from the asynchronous nature or nondeterminism of MIMDs. The same program may be executed in many different ways because of the relative speeds of various processes. If a program is not properly synchronized, multiple executions of the same program may produce different outputs. In conclusion, MIMDs operate best on applications with infrequent communication operations between processors and large granularity. (Granularity is an approximate measure of amount of local execution between communication requests.) Therefore, a coarse grained application synchronizes less often than a fine grained application does.
2.1.1.4 Single Instruction Stream Multiple Data Streams (SIMD)

This classification contains most parallel computers have even been built. A computer in this class has one central control units (or program controller) that drives the program flow by fetching and decoding a single instruction stream from its local memory. Usually, scalar instructions are executed locally by the control unit. Parallel instructions are issued to a set of processing elements (PEs or processors) one instruction at a time for concurrent execution. PEs execute the issued instruction on their individual data items stored locally. PEs are simple, small, and cheap to build. Since the control unit does all of the fetching and decoding of instructions, PEs only need to execute parallel instructions. Each PE can be built containing simple arithmetic logic units (ALUs), digital circuits that perform arithmetic and logical operations, and its local memory to keep only data, not program. PEs execute data synchronously (in lock-step) after the control unit broadcasts instructions to them. Since only one control unit exists in a SIMD, there is no need for synchronization. The major advantage of SIMDs is their simplicity. Their programs are easy to program, debug, and optimize. One of the complaints by programmers about parallel computers is that they are difficult to program. This is not true in SIMDs. Because of the synchronous nature of SIMDs, the flow control of SIMDs is sequential and at exactly one place in the program text. There is no need to consider numerous process interactions. Additionally, there is only one program to write and debug and data are usually distributed among PEs naturally. Two major drawbacks of SIMDs are its low PE utilization and its restricted control organization. A simple if-then-else conditional structure may reduce average PE utilization to 50% because, during the execution of the then part, PEs needed to execute the else part must idle and vice versa. Nested conditional control structures further reduce PE utilization. Thus, a SIMD model requires the programmer to restrict the use of conditional control structures if there is a need to maintain
high processor utilization. SIMDs have restricted support of functional or control parallelism, which is described below. PEs of a SIMD must execute the same instruction and process the same data at the same time or sit-out that instruction. If PEs require different control patterns such as those in conditional structures, each pattern must be broadcast by the control unit separately.

2.1.2 Control and Data Parallelisms

Control parallelism (also sometime called function parallelism or task parallelism) represents parallel programming usually on MIMDs or computers consisting of a small number of processors. The computation is divided into unique tasks that can be carried out concurrently. Each task is assigned to a processor. Control parallelism emphasizes parallelization of tasks rather than parallelization of data. SIMDs are the ideal platform to execute data parallelism program. A single algorithm can be applied to numerous data items in a uniform pattern. In data parallelism, a data set is divided into small segments that are assigned to PEs for concurrent execution. The difference between the control parallelism and data parallelism is as follows. In control parallelism, the program is broken up into tasks and fed to different processors for concurrent execution. In data parallelism, a data set is broken up into smaller pieces and given to processors for concurrent execution. These forms of parallelization are not mutually exclusive. The majority of massively parallel algorithms utilize both data parallelism and (to a limited degree) control parallelism.
2.1.3 Speedup

The speedup of a parallel algorithm, denoted by $S(1,p)$, is defined as a ratio of the running time of the fastest known sequential algorithm, denoted by $t_1$, and the running time of a parallel algorithm, denoted by $t_p$, using $p$ processors. The formula for the speedup is $S(1,p) = t_1 / t_p$.

2.1.4 Amdahl’s Law

Let $f$ be the fraction of the operations in a computation that must be performed sequentially, where $0 \leq f \leq 1$. The maximum speedup $\psi$ achievable by a parallel computer with $p$ processors performing the computation is $\Psi \leq 1 / (f + (1 - f) / p)$.

As mentioned in Amdahl’s Law, the speedup achievable by $p$ processors is limited by an upper bound related to the number of operations that must be performed sequentially. The limit of the left hand side as $p$ approaches infinity of $1/f$ provides an upper bound to the maximal speedup possible, regardless of how many processors are used. This information is essential when designing an algorithm.

2.2 Related Models of Computation and Architectures

Since the MASC model supports control parallelism of data parallelism tasks (in the form of ASC threads), this section will discuss other models and architectures that have the ability to
execute a task in both SIMD and MIMD modes and some other models of computation that have been discussed throughout this dissertation.

2.2.1 SIMD/MIMD Models and Architectures

There are two types of SIMD/MIMD models and architectures. Multimode systems are capable of execute both SIMD and MIMD modes simultaneously. Mixed mode systems have to switch between the SIMD and MIMD modes of parallel processing.

The Image Understanding Architecture (IUA) developed by Hughes Research Labs and University of Massachusetts [62][63] is an example of a multimode system. The first and the second generations of the IUA have a three-level structure. The highest level consists of general purposed MIMD processors. The intermediate level is also a MIMD comprised of a mesh of processors for digital signal processing. The low level is a mesh of bit-serial SIMD processors. All levels of the IUA can be used simultaneously, if needed. Low-medium level image processing tasks are assigned to the low level SIMD processors. At the same time, high level image processing tasks are assigned to medium and high level MIMD processors.

The rest of this section will give examples of mixed mode SIMD-MIMD systems. The first example is a multifunctional partitionable large scale dynamically reconfigurable multimicroprocessor system for parallel image understanding called PASM. PASM was developed by Siegel at Purdue University [49][50]. PASM is considered a mixed mode system and incorporates over 1,000 PEs. PASM can be reconfigured among three dimensions of partitonability, parallelism mode, and connections among PEs.
The Superscalar SIMD is developed by Schimmel at Georgia Tech [48]. The Superscalar SIMD architecture extends the work of Siegel’s PASM and others on reconfigurable SIMD/MIMD architecture. The architecture is able to take most of the advantages of those machines via selective execution of a superscalar instruction stream while retaining most of the cost advantage of the SIMD architectural style. The Superscalar SIMD is somewhat resemble the MASC model to some degree. It can send k instruction streams to an array of PEs. Each PE listens to only one of k instruction streams. PEs can be dynamically partitioned by an instruction stream into two disjoint parent and child partitions. The original instruction stream retains the control of the parent partition and a new child instruction stream takes control of the child partition. Joining back of tasks happens after children tasks are finished.

The Texas Reconfigurable Array Computer (TRAC) is developed at the University of Texas by Lipovski [30]. TRAC is a dynamically partitionable mixed mode shared memory parallel machine. It has 16 processors connecting to 81 memory and I/O elements. The system is dynamically reconfigurable while running into anything from 1 to 16 partitioned. Partitions run independently from each other. The TRAC subsystems can operate in various types of parallel execution. A given task may fork into subtasks and run using asynchronous MIMD operations or synchronous parallelism with external control of startups and interrupts.

The Triton was developed at the University of Karlsruhe, Germany by Herter [22]. It consists of 260 nodes, scalable to 4096 nodes that are able to operate as either synchronous SIMD processors or MIMD processors. The Triton is a mixed-mode (SIMD/MIMD) parallel computer that supports high-level machine-independent programming languages and fast SIMD/MIMD mode switching.
2.2.2 Parallel Random Access Machine (PRAM)

The Parallel Random Access Machine (PRAM) model is an important model and is probably the best known parallel computing model. It is a natural parallel extension of the traditional Random Access machine (RAM) model for sequential computing and consists of multiple identical RAMs with a shared memory. Although a parallel computer has never been built that possesses all of the properties of the PRAM model, the PRAM continues to be an important model for parallel computation because it is a natural extension of RAM, the traditional model for sequential computing, and the fact that it provides ease to programming and analyzing.

![PRAM Diagram](image-url)

Figure 2.1: A diagram of a PRAM model.
As shown in Figure 2.1, a PRAM comprises a collection of identical processors and a shared memory consisting of unlimited number of memory cells [1][24]. Processors in PRAM can access the shared memory independently from each other and in constant time. The shared memory acts as a communication link between two processors, i.e., a communication between two processors takes one memory write and one memory read. Besides the shared memory, each processor also has a local memory. Generally, each processor executes the same step in a program synchronously. Each execution of any one instruction takes one processor cycle. Each processors cycles through three phases synchronously — read a memory element from an arbitrary location in memory into its registers, execute a program step by performing basic operation on the content of its registers, and write to a memory element from its registers to an arbitrary memory location.

There are a number of different ways a PRAM can gain access to the shared memory--exclusive read (ER) or concurrent read (CR) for reading from the memory elements and exclusive write (EW) or concurrent write (CW) for writing to the memory. For ER and EW, processors gain access to memory locations in a one-to-one fashion. For CR and CW, two or more processors can access the same memory location at the same time. For CW, some predefined conflict resolution rules resolve write conflicts. Well-known examples include Common (all processors writing to the same location write the same value is allowed), Collision (if multiple processors write to the same location, then a special collision symbol appears), Collision+ (if all concurrently writing processors write the same value, then it behaves like Common; otherwise it behaves like Collision), Arbitrary (an arbitrary writing processor succeeds in its write attempt), and Priority (the highest priority writing processor succeeds) [1][24].
2.2.3 The Associative Computing (ASC) Model

The Associative Computing (ASC) model as, shown in Figure 2.2, is based on the STARAN associative SIMD computer designed by Kenneth Batcher at Goodyear Aerospace in the early 1970's and its heavily Navy-utilized successor, the ASPRO.

![Figure 2.2: A diagram of an ASC model.](image)

The model supports massive parallelism using an associative SIMD style of computation. Duncan classified associative processors as a subclass of SIMD processors in 1990 [36]. The features that distinguish an associative computer such as an ASC model from SIMD computers are, first, the associative computer utilizes massively parallel searching by active memory logic so that searching can be used as an alternative to addressing. The ASC model does not assume an associative memory. In a true associative memory, every bit of memory has a comparator logic
associated with it. In the ASC model, all the bits of a word share the common comparator logic in a processor. Second, the associative computer maintains tightly coupled control via a responder/no responder signal that can be tested in constant time for the effective use of parallel-search feedback in program flow control. Third, the associative computer has a resolution hardware that can select an individual responder processor from a set of many processors in constant time. Last, the associative computer contains a broadcast network (a sequential control to array data path) that allows the broadcast of data to all array processors in parallel in constant time and a reduction network (an array to sequential control data path) that allows the transfer of data from selected responders to the instruction stream in constant time.

In addition to the above properties, the ASC model is able to perform MIN/MAX and logical reductions such as bitwise AND or bitwise OR in constant time [35][26]. The ASC model had be shown that its architecture easily supported a wide range of applications and a simple style of programming much closer to sequential programming than to traditional multiprocessor programming [3][32][56][57][12][25]. A language has also been designed for the ASC model and is called the ASC language [36].

2.2.4 Reconfigurable Models

A reconfigurable architecture is one that can alter functionalities of its components and structure connecting these components. This section summarizes properties of those reconfigurable models used in this dissertation.
2.2.4.1 Reconfigurable Mesh (RM)

A 2-D Reconfigurable Mesh (RM), as shown in Figure 2.3, is a basic mesh model enhanced by reconfigurable buses. Each processor has four ports, referred to as N, S, E, and W, that can be controlled locally, allowing disjoint buses to be established dynamically. With the ability of reconfiguring bus connections during algorithm execution, an RM can create different communication patterns based on the algorithm needs.

Figure 2.3: Structure of a 3x5 R-Mesh.

The RM model has been widely accepted as an extremely powerful model. A number of constant time algorithms have been discovered for this model [1][5][34][40] (assuming a constant time bus broadcast), while they require non-constant time on other models [1][40][59].
2.2.4.2 Reconfigurable Multiple Bus Machine (RMBM)

The Reconfigurable Multiple Bus Machine (RMBM) is a reconfigurable bus-based model proposed by Trahan et al. [55][58]. The RMBM model consists of a set of processors and a set of buses that are used for processor communication. Through the use of its local settings, a processor can manipulate its read and write connections to buses as well as manipulate bus structure. A processor can split a bus into segments or connect (fuse) one bus to another. Each processor controls one pack of switches for each bus, with up to five switches in each pack, and a fuse line that crosses all buses. Figure 2.5 shows an RMBM model but omits fuse lines and switches. Two switches control the connection of the bus to the processor’s reading port and writing port, respectively. Two switches are used to segment the bus at the points where they are located. One is located to the left of the reading port and the other is between the reading and...
writing ports. The last switch is used to connect the processor fuse line to the bus. The processor fuse line can connect multiple (not necessarily adjacent) buses together.

![Diagram](image)

**Figure 2.5:** A S-RMBM model with three processors.

Initially, all switches are reset, that is, buses are neither segmented nor fused and processor read/write ports are not connected to a bus. A processor can set or reset a switch in one time step but can only set or reset one switch at a time. A processor may connect its reading and writing ports to different buses. At any point of time, however, a processor can connect each port to only one bus. A value written on a bus reaches the other processors connected to the bus in the same step. We consider the concurrent read, concurrent write (CRCW) RMBM in this paper.

Depending on switches available in the processors, there are four versions of RMBM [55][58]:

- **B-RMBM:** A processor only has switches to connect reading and writing ports to a bus.
• **S-RMBM**: In addition to read/write switches, a processor can segment the bus.

• **F-RMBM**: In addition to read/write switches, a processor can fuse buses using its fuse line.

• **E-RMBM**: All five switches are functional for a processor, namely, a processor can connect to a bus for read/write, split a bus into segments, and fuse a bus to other buses via its fuse line. This is the strongest version of RMBM.
A Multiple Associative Computing (MASC) Model

3.1 Introduction

The MASC (Multiple Associative Computing) model is a generalization of the ASC (Associative Computing) model designed to support multiple ASC threads by using control parallelism to substantially improve the low processor utilization often criticized in SIMDs. It was proposed by Potter et al. [35] in 1994 and has been actively studied and researched at Kent State University since. Since the MASC model is a strictly synchronous model that supports SIMD computation, it is a multi-SIMD or MSIMD model, i.e., a SIMD enhanced with multiple instruction streams.

In contrast to a number of other parallel models and similar to the ASC model, the MASC model possesses certain constant time global properties, when the word size is assumed to be a constant, such as constant time broadcasting, constant time global reduction operations, and constant time associative search. These properties have enabled MASC not only to solve a wide range of problems effectively [3][12] but also problems in special areas such as real-time air traffic control in an extremely efficient manner, using worst case analysis to ensure that all deadlines are met [32]. An associative programming language called ASC language that supports only one instruction stream (IS) version of MASC (also) has been implemented on a number of
SIMD platforms [35][36]. Possible techniques for implementing the MASC model have been explored and related research work is still ongoing [8][45][60].

### 3.2 Components of the MASC Model

As shown in Figure 3.1, a MASC model consists of an array of processing elements (PEs), one or more instruction streams (ISs), broadcast and reduction networks (preferable one for each instruction stream), and an instruction stream network. The model may consist of a cell network, which is an optional. A MASC model with n PEs and j ISs is usually denoted by MASC(n, j).

**Figure 3.1:** A MASC model shown with all of its components.
All PEs are identical and are very simple, i.e., basically ALUs. Each PE, paired with its row of memory or local memory, is called a cell. The terms PEs and cells are used interchangeably in this research. Normally, a record of a set of data is stored in the memory of each cell. When the number of records is greater than the number of cells available, two or more records will be folded into one cell. Having more than one record stored in each cell is a common practice but will degrade the performance of the system. To simplify the discussion, each cell contains at most one record of data items is assumed. Each cell had a mask register, usually a 1 bit register. The mask register indicates whether that cell is a responder or not.

Historically, the instruction streams for SIMDs were called control units or front ends. Similar to a control unit of a SIMD computer, an instruction stream is a processor and able to fetch, decode, and broadcast instructions to its PEs. The number of ISs is expected to be considerably smaller than the number of PEs. Both ISs and PEs have unique numbers and each knows its number. To store a processor identification number, a word size of Θ(log n) is required for a parallel model with n PEs, so Θ(log n) is a frequent assumption for word size and bus widths in parallel models. An IS may broadcast a value to PEs grouped under it as follows. First, the IS activates PEs who need the data item. Then, the data item to be read is broadcast to those active PEs to first read and then store the broadcast item. An IS may want to read a value from a PE or PEs grouped under it. First, the IS uses associative operations to determine which PE (PEs) is (are) to be read from. A typical occurrence is for an IS to first have the active PEs make a search. If there is at least one responder, a Pick-One operation selects one of the responding PEs. Then, the IS reduces this PE data item to it by using the reduction network. Another method is for active PEs for an IS to do a Maximum/Minimum or AND/OR reduction over a value stored in a word. By using a reduction network connected to the IS, it automatically receives the reduction value. If needed, the IS can broadcast this reduction value back to some of the PEs.
A MASC model may have three types of networks, namely, a cell network for cell communications, an instruction stream network for instruction stream communications, and broadcast and reduction networks for communication between instruction streams and their sets of PEs.

![Diagram](image)

**Figure 3.2:** A broadcast and reduction network with its network processors attached.

The broadcast/reduction network is essential to the MASC model. It may be implemented using separate network circuits or sharing the same network circuit for both broadcasting and global reduction operations. In practice, as shown in Figure 3.2, the network can be constructed as a tree-structured set of resolver circuits. Each leaf represents a PE and all PEs are divided into groups of 4 to compose a basic 4-PE resolver building block. A resolver with \( n \) PEs can be built as a \( \lceil \log_4 n \rceil \) level tree of 4-PE resolvers. When a broadcast or a reduction operation is performed,
a bit takes at most \(2\lceil \log_4 n \rceil + 1 \) gate delays (or \(2\lceil \log_4 n \rceil - 1 \) gate delays without network processors) from any input to any output. This contrasts with a broadcast on a bus-based architecture that takes a linear order of gate delays with respect to the bus length and number of processors. Considering that the linear gate delay bus broadcast is generally assumed to be a constant time operation, the logarithmic gate delay MASC broadcast can certainly be treated as a constant time operation, even for impractically large numbers of processors \(n\) [26].

A network processor connects each PE to its 4-PE resolver circuit. Its function is forwarding data between its 4-PE resolver circuit and its PE. If there are \(j\) instruction streams in the system, a PE listens to only one of \(j\) instruction stream at a time. As the result, a PE connects to the network processor of the broadcast and reduction network of that instruction stream. If a PE does not connect to a broadcast and reduction network, it does not connect to the network processor of that network. In that case, any instruction or data broadcasted from an instruction stream arriving at that network processor will be discarded. When the instruction stream does a reduction, a default value (instead of a junk value from the PE, which is not listen to that instruction stream any way) will be forwarded back to its 4-PE resolver circuit, i.e., a default data value 1’s (0’s) is forwarded back for a global AND (OR) reduction.

With its broadcast/reduction network, the MASC model supports global and broadcast operations between an instruction stream and its active PEs in constant time (assuming the word size for these operations is a constant). These operations include the global reduction operations of OR and AND of binary values, the Maximum and Minimum of reduction operation integer values, and an associative search for a given search pattern. Following an associative search, an instruction stream can determine if any PE matches the search pattern (“any-responders”) and select an arbitrary PE from its responders (or “pick-one”) in constant time. These constant time
operations are called associative operations and support highly efficient database operations on records stored in identical locations across the PEs. The feasibility of these assumptions has been justified by Jin et al. [26]

Cell and instruction stream networks may be implemented by a bus, a tree, or other simple network to facilitate the communications among PEs and instruction streams. Usually a cell network is used for a massive PE data movement. Both networks may be considered as optional for the basic MASC model, as it is shown in chapter 4 that with or without them, the power of the MASC model does not change.

3.3 Associative Properties and Operations

The MASC model is a SIMD model with associative properties and some additional hardware features as already mentioned earlier. It supports massively parallelism using an associative, multiple SIMD style of computation. The additional hardware features enable the MASC model to broadcast data to PEs, do a global reduction such as AND/OR of Boolean values or MAX/MIN of integer values in constant time, and do an associative search for data in the memory by content rather than by location in constant time. In this section, key constant time associative operations will be presented.
3.3.1 Global AND and OR Reductions

The MASC model supports a constant time global bitwise AND and OR reduction operation. Data values from four PEs (or “up to four” default data values, 1’s or 0’s, from network processors) are the input into the 4-PE resolver circuit. The resolver circuit does an AND (OR) of these data values and output a value to its next resolver circuit. These 4-PE resolver circuits continue to propagate the data back up until an instruction stream gets the result. Since we treated the whole broadcast and reduction network’s gate delay as a constant time operation and each 4-PE resolver circuit does an operation (up to the word size) in constant time, the MASC model supports a constant time global bitwise AND and OR reduction operation.

3.3.2 Global MAX and MIN Reductions

The MASC model supports a constant time global MAX and MIN reduction operation. Data from four PEs (or “up to four” default data values, which is the known minimum or maximum for that data type, from network processors) are input into the 4-PE resolver circuit. The resolver circuit does a comparison of these data values and output a value to its next resolver circuit. These 4-PE resolver circuits continue to propagate the data back up until an instruction stream gets the result. Based on the same argument given earlier in 3.4.2, the MASC model supports a constant time global MAX and MIN reduction operation.
3.3.3 An Associative Search Operation

An associative search operation can be performed as follows. An instruction stream broadcasts instructions to its PEs to execute a conditional expression. If a PE satisfies the condition, it sets its mask bit to 1. It resets its mask bit to 0, otherwise. Since each step takes constant time, the associative search operation is a constant time operation.

3.3.4 Any-Responder Operation

An Any-Responder operation can be performed as follows usually after an associative search operation. An instruction stream does a global OR reduction on mask bits of its PEs. If the returned result of the reduction is 1, then there is a responder. There is no responder if the returned result is 0. Since a global OR reduction takes constant time, the Any-Responder operation is a constant time operation.

3.3.5 Pick-One Operation

A Pick-One operation can be performed as follows usually after the previous Any-Responder operation returned 1 as the result. An instruction stream does a global MAX (MIN) reduction on PE id of its PEs, whose mask bits are 1. The returned result is the id of a PE that will be selected. Later, the instruction stream may instruct that PE to reset its mask bit to 0 in order to avoid picking the PE again in the next round. Since a global MAX (MIN) reduction is a constant time operation, a Pick-One operation is also a constant time operation.
3.4 A MASC Model Using the Manager-Worker Instruction Stream Paradigm

The MASC model using the manager-worker paradigm is a model that combines the advantages of the SIMD and MIMD models as already mentioned in Chapter 1 and (in the SIMD/MIMD section) in Chapter 2. While Potter et al. [35] identified the MASC computational model's properties, these were high-level and did not specify the interactions between the instruction streams needed to support a MASC architecture.

Figure 3.3: A MASC model using the manager-worker paradigm.

One possible approach to supporting multiple instruction streams in the MASC model is to use the manager and worker paradigm to support multiple executions of branches in a data parallel program as shown in Figure 3.3. Two of the advantages of using the worker and manager
paradigm in the MASC model over other instruction stream control paradigms are its simplicity and its control overhead can be kept as constant or minimal. It is preferable to build a prototype or a run-time system using such paradigm since the expected number of instruction streams is considerably smaller than the number of PEs. Overworking the manager instruction stream and creating a bottleneck in the system can be avoided by using static task assignment scheme.

One instruction stream (usually the first instruction stream) is designated to be the manager instruction stream and the rest of the instruction streams are worker instruction streams. Duties of the manager instruction stream are managing tasks and coordinating worker instruction streams. The only function of a worker instruction stream is to execute the task given by the manager instruction stream. Each worker instruction stream works on a different task (as a part of the problem) and issues its own instruction sequence to its partition of PEs. Once an execution of the task has been started, there will be no communication between instruction streams until such execution is completed. The tasks created at a branch are totally independent of each other.

In this variant of the model, an instruction stream network is essential. A broadcast and reduction network connects the manager instruction stream to all of its worker instruction streams to support all instruction stream communications needed. One reason behind choosing a broadcast and reduction network as the instruction stream network is to keep the manager instruction stream’s operations to be constant time operations by using operations similar to those for PEs on instruction streams such as constant time global reductions, an associative search, Any-Responder, and Pick-One operations.

In addition to a mask register, each PE of a MASC(n, j) will have a log j-bit instruction stream selector register holding the ID of the current instruction stream that the PE listens to. In
order to support the maximum of \( k-1 \) tasks, a log \( k \)-bit register is required to keep a current task id, which indicates the current task the PE belongs.

### 3.5 Comparing the Power of a Basic MASC with a MASC with the Manager-Worker Paradigm

This section compares the power of a simplified MASC model without either cell network or instruction stream network with a MASC model enhanced with the manager-worker paradigm without its cell network. It is shown in Chapter 4 that these networks do not increase or decrease the power of the model.

**Lemma 3.1:** A simplified MASC model (without either cell or instruction stream networks) is equivalent in power to a MASC model using the manager-worker paradigm (without the cell network).

**Proof**

If a MASC model with the manager-worker paradigm has at least as much power as that of a simplified MASC model, it must be able to simulate every operation of the simplified MASC model in constant time. A MASC \((n, j+1)\) model with the manager-worker paradigm can simply simulate a simplified MASC \((n, j)\) model by performing every action the simplified MASC model
does by not using the manager instruction stream or the instruction stream broadcast and reduction network.

Conversely, if a simplified MASC model has the same power as that of MASC model with the manager-worker paradigm, it must be able to simulate every operation of the MASC model with the manager-worker paradigm in constant time. Since all other operations (except those that can be performed by the manager instruction stream) between the two models are the same, showing that a simplified MASC model can simulate any operation performed by the manager instruction stream in constant time is sufficient. Those operations consist of a broadcast of data or an instruction to worker instruction streams, a global reduction of AND/OR and MAX/MIN on data from worker instruction streams, an associative search on data from worker instruction streams, an Any-Responder on worker instruction streams, or a Pick-One on worker instruction streams.

A simplified MASC \((n+j, j)\) model is used to simulate a MASC \((n, j)\) model with the manager-worker paradigm. The first instruction stream (IS id 0) of the generic MASC model along with PEs (PE id n to PE id n+j-1) is used to simulate the manager instruction stream and its instruction stream broadcast and reduction network of the MASC model with the manager-worker paradigm. We will denote the \(i\)-th instruction stream by IS\((i)\) and the \(i\)-th PE by PE\((i)\). To simulate a broadcast of data or an instruction to worker instruction streams, at first, all PE\((i)\), with \(n \leq i \leq n+j-1\), are instructed to listen to IS\((0)\). Next, IS\((0)\) broadcasts a data (or an instruction) to these PEs using its broadcast and reduction network. Then, IS\((0)\) switches PE\((i)\) to listen to IS\((m)\), where \(m = i - n\). The instruction streams IS\((k)\), for \(1 \leq k \leq j-1\), read the data (or an instruction) from PE\((k+n)\) using their broadcast and reduction networks. Finally, each IS\((k)\) switches PE\((k+n)\) back to listen to IS\((0)\). Note that IS\((0)\) also serves as a worker IS, but it is not necessary for IS\((0)\)
to transfer this data item or instruction to itself. The simulation is completed and takes constant time.

To simulate a global reduction of AND/OR and MAX/MIN, an associative search, an Any-Responder, or a Pick-One on worker instruction streams, at first, the PE\(i\) for \(n \leq i \leq n+j\) are instructed to listen to IS\(m\), for \(m = i - n\). Instruction streams IS\(k\), where \(1 \leq k \leq j-1\), broadcast data (needed for a global reduction, a search, an Any-Responder, a Pick-One) to PE\(q\), where \(q = k + n\), using their broadcast and reduction network. Finally, each IS\(k\) switches those PEs back to listen to IS\(0\) and IS\(0\) performs the corresponding global action (i.e., global reduction, associative search, etc.) on these PEs. Following this global action, IS\(0\) must also take itself into consideration and adjust the result of the preceding global action appropriately. The simulation is completed and takes constant time.

The power relationship between the two models can be concluded as in lemma 3.1. □
CHAPTER 4

Comparing the Power of the MASC Model with Other Computational Models

4.1 Introduction

The power of a computational model is indicated by the efficiency with which it can simulate other computational models and algorithms it supports. To evaluate the power of a basic MASC model, simulation and comparison with other popular parallel models have been studied previously. A constant time simulation with high probability of PRAM (Parallel Random Access Machine) using MASC and a constant time simulation of MMB (Mesh with Multiple Buses) using MASC have been established [4][56]. Ulm and Baker [56] studied simulations between MASC and PRAM and found that a MASC without a cell network can simulate a Priority CRCW PRAM in O(1) time with high probability and a PRAM can simulate each step of a j IS MASC in O(j) time with high probability. This chapter is not directly focused on giving a MASC-PRAM simulation. The result, however, establishes O(1) time deterministic simulations between MASC and PRAM and concludes the fact that MASC and PRAM are actually in the same category in terms of their computation power.

In this chapter, simulations between a basic MASC model and reconfigurable bus-based models RMBM (Reconfigurable Multiple Bus Machine) are presented. The goal is to compare the power of the MASC model to that of the RM (Reconfigurable Mesh) model, as RM has been widely accepted as an extremely powerful model. Due to the power of RM and the fact that it is a
well-known model, it is a good idea to establish a relationship between the MASC model and RM model. This relationship could be a useful tool in evaluating the power of the MASC model. However, dissimilarities between the MASC and a general RM make direct simulations difficult. Instead, a bridge model—the RMBM that has been shown to be equally powerful to RM—is considered for a simulation, with the intent of establishing a relationship between MASC and RM. Relationships among different versions of RMBM, RM, and PRAM have been established [55][58]. Considering the similarities of RMBM and MASC models and taking advantage of the established relationships, simulations between MASC and RMBM are developed first. Then, the simulation extends results to establish further relationships among MASC, RM, and PRAM, as the strongest version of RMBM has the same power as RM and its weakest version has the same power as PRAM. This work provides a better understanding of the MASC model and useful information concerning its power.

The majority of work in this chapter is published as a journal article in the Journal of Parallel and Distributed Computing. The article is co-authored with Dr. Jerry L. Trahan, Dr. Mingxian Jin, and Dr. Johnnie W. Baker. The author made a contribution throughout the article and with a major contribution to simulations between segmenting-RMBM and MASC, PRAM and MASC, and MASC models with different word sizes.

The chapter is organized into seven sections. Section 4.2 gives a preliminary assumption about the word size. Sections 4.3 to 4.5 present simulations between MASC and different versions of RMBM, respectively. Section 4.6 compares two resulting MASC models when different word sizes are assumed. Section 4.7 extends the simulation results to categorize the power of MASC in relation to those of RM and PRAM.
4.2 Word Size

Originally, it is assumed that the word-size of a MASC model was constant. Let $\text{MASC}_v$ denote a MASC with word-size $v$ for associative operations, so $\text{MASC}_c$ (or $\text{MASC}_{\log n}$) denotes a MASC with constant $c$ (or log $n$) associative operation word length. In this chapter, a constant time simulation between $\text{MASC}_c$ and $\text{MASC}_{\log n}$ derived from a cycle of simulation results is presented to show that, in fact, $\text{MASC}_c$ and $\text{MASC}_{\log n}$ are equivalent in power.

To avoid any confusion on the word size assumption on the MASC model, sections 4.3 to 4.5 assume MASC can perform constant-time associative operations on $v$-bit words, where $v$ can be a constant or a function of $n$, i.e., $O(\log n)$.

4.3 Simulations between MASC and B-RMBM

The next three sections present simulations between the MASC model and different versions of the RMBM model. In this section, the first simulation starts with the weakest version, B-RMBM.

Simulating MASC with B-RMBM is fairly straightforward, as there are some similarities between the structures of the two models. Different B-RMBM resources, however, lead to slightly different results. Let $\text{MASC}_v(n, j)$ denote a MASC with $n$ PEs and $j$ ISs that can perform constant-time associative operations (i.e., a global OR/AND operation, a global maximum/minimum reduction operation, or an associative search) on $v$-bit wide data. Let $\text{B-RMBM}(n, m)$ denote a B-RMBM with $n$ processors and $m$ buses. To simulate each step of a $\text{MASC}_v(n, j)$, let’s start with a base B-RMBM with $n$ processors and $j$ buses using Common
concurrent write (CW). Next, let’s examine how different combinations of available processors, buses, and write resolution rules affect the simulation. Assume that \( j \leq n \). Let each of the first \( j \) processors, \( p_i \), on the B-RMBM simulate a unique MASC IS, \( IS_i \). The read/write ports of each \( p_i \) use bus \( i \) for IS, communications. Each of the overall \( n \) processors simulates one of the \( n \) MASC PEs. For each of these \( n \) processors, the bus to which its read/write ports connect is determined by the IS to which the corresponding simulated MASC PE listens.

Each processor of the B-RMBM can simulate the local operations of the corresponding processor of the MASC, so the MASC operations to examine closely are those involving the broadcast/reduction networks. These are OR/AND, maximum/minimum, associative search, Any-Responder, and Pick-One operations.

Bitwise OR/AND of up to \( n \) \( v \)-bit values: a B-RMBM can compute the OR or AND of \( n \) bits on one bus in constant time, so the base B-RMBM can execute this of one bit position at a time to get the result in \( O(v) \) time. Alternatively, a B-RMBM\( (n^2, \max\{n, jv\}) \), where \( v \leq n \), can compute this operation in \( O(\log v) \) time as follows. Let \( IS_i \) with set \( S_i \) of PEs compute a global OR. For each processor \( p_k \) of the corresponding set \( S'_i \) on the B-RMBM, \( p_k \) first distributes \( v_i \) copies of its data to \( v \) other processors \( p_{k,m} \), for \( 1 \leq m \leq v \). Each \( p_{k,m} \) extracts bit \( m \) of the data, then over all \( p_k \) in \( S'_i \), processors \( p_{k,m} \) compute the OR of their bits. Next, \( \log v \) levels of pairwise merges of results combines the \( v \) individual ORs into a final \( v \)-bit result.

Maximum/minimum of up to \( n \) \( v \)-bit values: the base B-RMBM can compute a maximum or minimum in a bit-serial manner by iterating broadcasting and concurrent write steps via the bus for each bit of the field in the order of the left (most significant) bit to the right (least significant) bit. This runs in \( O(v) \) time. This algorithm is based on a parallel search algorithm by Falkoff [13]. Alternatively, use a B-RMBM\( (n^2, n) \) to simulate a step of finding a maximum in
O(1) time, where \( j \leq n \), following a technique used by PRAMs [29][47] and RMs [34]. The idea is to compare all pairs of elements for an IS computing maximum and use AND operations to identify as the maximum value one that is greater than or equal to all other elements. Specifically, let \( p_{g,h} \), where \( 1 \leq g, h \leq n \), denote a processor of the B-RMBM. Broadcast all values being written and their corresponding IS indices, and let each \( p_{g,h} \) read value \( r_g \) and IS \( i_g \) as well as \( r_h \) and IS \( i_h \). If \( i_g = i_h \), then proceed; otherwise, do nothing. On bus \( g \), for all \( p_{g,h} \) with \( i_g = i_h \), AND the outcome of whether \( r_g \geq r_h \).

Associative search: when IS, performs an associative search with value \( t \) and set \( S_i \) of PEs, on the base B-RMBM, \( p_i \) writes \( t \) to bus \( i \), and the processors corresponding to \( S_i \) read and compare \( t \) to their local values.

Any-Responder: after an associative search for IS, on the base B-RMBM, any processor \( p_k \) with a match can write a 1 on bus \( i \). Recall that this B-RMBM uses Common CW. Processor \( p_i \) reads bus \( i \) to determine whether any of its processors had a match.

Pick-One: the base B-RMBM can compute a maximum (or minimum) on the IDs of matching processors in \( O(\log n) \) time to select one matching processor. As one alternative, a B-RMBM\((n^2, n)\) can compute a maximum (or minimum) in \( O(1) \) time as described above. As a second alternative, a B-RMBM\((n, j)\) with Arbitrary or Priority CW can select one matching processor in \( O(1) \) time. As a third alternative, a B-RMBM\((n \log n, O(n))\) using Common, Collision, or Collision+ CW can select one matching processor in \( O(1) \) time, using a PRAM technique due to Ragde [44]. The following theorem compiles these alternatives into a range of simulations of a MASC, by a B-RMBM.
**Theorem 4.1:** Each step of a MASC\(_v\)(n, j) with constant-time associative operations on \(v\)-bit data, can be simulated by:

(i) an Arbitrary or Priority CW B-RMBM(n, j) in \(O(v)\) time,

(ii) a Common, Collision, or Collision+ CW B-RMBM(n log n, O(n)) in \(O(v)\) time,

(iii) a Common, Collision, or Collision+ CW B-RMBM(n, j) in \(O(v + \log n)\) time, or

(iv) a B-RMBM(\(n^2\), max\{n, jv\}), where \(v \leq n\), in \(O(\log v)\) time.

**Remark:** For constant \(v\), the simulations of Theorem 1(i, iii, iv) run in \(O(1)\) time.

Next, let’s consider simulating a B-RMBM using a MASC model, specifically, a B-RMBM(n, m) with n processors and m buses on a MASC\(_v\)(n, j) with n PEs and j ISs. If \(j \geq m\), let each MASC PE simulate one of the n B-RMBM processors and let MASC IS\(_i\) simulate the \(i^{th}\) B-RMBM bus for \(1 \leq i \leq m\). Depending on how a B-RMBM processor sets the read (write) port to a bus in a particular step, a MASC PE listens (writes) to the corresponding simulating MASC IS. The MASC emulates concurrent reads from an RMBM bus by a broadcast by an IS. The MASC emulates concurrent writes to an RMBM bus by associative operations and the broadcast/reduction network. For example, for Collision CW on an RMBM bus, a MASC IS uses “Any-Responders” to make sure the set of PEs wishing to write is non-empty. If nonempty, use Pick-One to select one PE from the set of PEs wishing to write and temporarily deactivate this PE. Use Pick-One again to see if there is a second PE wishing to write. If a second PE is found, a special collision symbol will be broadcast to PEs wishing to read. If there is only one PE wishing to write, the data this PE holds will be broadcast to the PEs that wish to read. The MASC can also
emulate Common, Collision+, and Arbitrary CW in constant time. If the B-RMBM uses Priority CW, however, it takes an extra $O\left(\lceil (\log n)/v \rceil \right)$ time to locate the particular PE ID number required by the priority rule.

If $j < m$, let $IS_1$ simulate B-RMBM buses $b_1, b_{2j+1}, \ldots$; $IS_2$ simulate B-RMBM buses $b_2, b_{2j+2}, \ldots$; and $IS_j$ simulate B-RMBM buses $b_j, b_{2j}, b_{3j}, \ldots$. Since each IS simulates at most $\lceil m/j \rceil$ buses, the MASC executes the simulation described above (for the case $j \geq m$) $\lceil m/j \rceil$ times. Thus, Theorem 4.2 follows.

**Theorem 4.2:** Each step of a B-RMBM($n, m$) can be simulated by a MASC($n, j$) with constant-time associative operations on $v$-bit data in:

(i) $O(m/j)$ time, if the simulated B-RMBM uses Common, Collision, Collision+, or Arbitrary CW, or

(ii) $O\left(\frac{m}{j} \cdot \left\lceil \frac{\log n}{v} \right\rceil \right)$ time, if the simulated B-RMBM uses Priority CW.

Clearly, when $j = \Omega(m)$, it is $O(1)$ time for (i) and, when $j = \Omega(m)$ and $v$ is $\Omega(\log n)$, it is $O(1)$ time for (ii).

### 4.4 Simulation between MASC and S-RMBM

Since an S-RMBM is a B-RMBM enhanced by adding to processors the ability of splitting a bus into bus segments, an S-RMBM with one bus can perform a Pick-One operation
with exclusive writes. Consequently, an $O(v)$ time simulation of MASC using S-RMBM($n$, $j$) follows from Theorem 4.1.

Let’s consider simulation of an S-RMBM using a MASC with $v$-bit associative operations. For an S-RMBM($n$, $m$) with $n$ processors and $m$ buses, the simulation below takes $O((\log n)/v)$ time. When $v$ is $\Omega(\log n)$, this is constant time. The idea behind the simulation is that, since $n$ S-RMBM processors can read in one step, regardless of the number of distinct bus segments, the MASC simulates only the segments read by the S-RMBM processors. Let $S$ denote S-RMBM($n$, $m$) with $n$ processors and $m$ buses. We simulate $S$ using $Q$, a MASC($n^2$, $n$) consisting of $n^2$ PEs and $n$ ISs. Let $p_i$ denote processor $i$ of $S$, and let $\text{PE}_{i,j}$, for $1 \leq i, j \leq n$, denote a PE of $Q$, viewing the PEs as arranged in an $n \times n$ grid.

![Diagram of PE grouping](image_url)

**Figure 4.1:** PE grouping to simulate a bus-access operation of the S-RMBM: (a) Row grouping; (b) Column grouping.
Q will use two groupings to assign PEs to the ISs. Under the column grouping, Q assigns the n PEs in the j\textsuperscript{th} column to IS\textsubscript{j} (i.e., PE\textsubscript{i,j} for 1 \leq i \leq n) and under the row grouping, Q assigns the n PEs in the i\textsuperscript{th} row to IS\textsubscript{i} (i.e., PE\textsubscript{i,j} for 1 \leq j \leq n) (see Figure 4.1). The blacken PEs are main PEs in their groups. The idea behind the column grouping is to use an IS for each processor p\textsubscript{j} of S to broadcast any actions and value written by p\textsubscript{j}. The idea behind the row grouping is to use an IS for each processor p\textsubscript{i} of S to reduce the values written to the bus from which p\textsubscript{i} reads and determine the specific value read by p\textsubscript{i}. Observe PE\textsubscript{i,i} is in IS\textsubscript{i} under both groupings and is the main PE for simulating p\textsubscript{i}. In particular, PE\textsubscript{i,i} stores p\textsubscript{i}’s data and executes its operations. Each PE\textsubscript{i,j} in column j holds two m-bit arrays L\textsubscript{i,j} and R\textsubscript{i,j} in which entry k of L\textsubscript{i,j} (R\textsubscript{i,j}) indicates whether p\textsubscript{j} has set its left (right) segment switch on bus k. (We assume that each Q cell has at least O(m) bits of memory.) Initially, each bit of these arrays is 0, indicating that no S segment switch is set.

In each step, each processor of S performs any of the following operations: set or reset one segment switch on a bus; move its read or write connection from one bus to another; write; read; and perform local computation. The key for Q is simulating reads. The idea here is that, for each reading processor p\textsubscript{i} of S, an IS and its PEs in Q use arrays L\textsubscript{i,j} and R\textsubscript{i,j} to find the boundaries of the bus segment from which p\textsubscript{i} reads and then evaluate writes only on that bus segment.

Overall, Q simulates each operation in a step of S as follows.

Step 1: Segment switch change: assign PEs to ISs using the column grouping. For each j, if p\textsubscript{j} sets (resets) a switch on bus k, then IS\textsubscript{j} broadcasts the following to its PEs: the bus ID k, whether the action was to the left or to the right switch, and whether the action was to set or reset. Each PE\textsubscript{i,j} reads, then writes 1 (0) to bit k of array L\textsubscript{i,j} or R\textsubscript{i,j}, as appropriate.
Step 2: Move read or write connection: for each \( j \), if \( p_j \) moves its read connection from bus \( k \) to bus \( g \), then \( IS_j \) broadcasts this information to its PEs (i.e., \( PE_{i,j} \), for all \( i \)). \( Q \) acts similarly for moving write connections.

Step 3: Write: for each \( p_j \) that writes, \( IS_j \) broadcasts to its column of PEs the value from \( PE_{i,j} \) that \( p_j \) writes.

Step 4: Read: assign PEs to ISs using the row grouping. If \( p_i \) reads from bus \( g \), then determine the location of the closest set segment switch to the left of the reading port of \( p_i \) as follows. \( IS_i \) broadcasts the bus ID \( g \) to \( PE_{i,j} \), for all \( j \). Next, for each \( k \leq i \), if \( p_k \) has set a segment switch on bus \( g \) (that is, if entry \( g \) of \( L_{i,k} \) is 1), then \( PE_{i,k} \) sends \( k \) to \( IS_i \), which stores the maximum value written (Figure 11-a). Determine similarly the location of the closest set segment switch to the right of the reading port of \( p_i \) (Figure 11-b). Next, for each \( p_k \) that writes on bus \( g \) between the set segment switches, \( PE_{i,k} \) sends to \( IS_i \) the value written by \( p_k \) (Figure 11-c). In the event of concurrent writes on \( S \), \( IS_i \) computes the bus data using associative operations and Pick One appropriate to the write conflict resolution rule of \( S \).

Step 5: Local computation: each \( IS_i \) performs the location computation of \( p_i \).

Assuming bus width \( O(\max\{\log m, \log n\}) \), Steps 1 and 2 take constant time. Steps 3 and 5 take constant time. Using MASC associative operations to obtain the max/min value of processor IDs and resolve write conflicts in Step 4 takes \( O((\log n)/v) \) time. Therefore, the total time for the simulation is \( O((\log n)/v) \).

Notice that, the same constructed MASC can be used to simulate a B-RMBM(n, m), since the B-RMBM is a special case of the S-RMBM without any segment switch set on processors. Thus, Theorem 4.3 and Corollary 4.4 follow.
Figure 4.2: Read step of simulating an S-RMBM on a MASC (p2 reads from bus g).

**Theorem 4.3**: An S-RMBM(n, m) can be simulated by a MASCv(n², n) in $O((\log n)/v)$ time if the MASC can perform $v$-bit width associative operations in constant time.

**Corollary 4.4**: A B-RMBM(n, m) can be simulated by a MASCv(n², n) in $O((\log n)/v)$ time if the MASC can perform $v$-bit width associative operations in constant time.
4.5 Simulation between MASC and F-RMBM, E-RMBM

Since both F-RMBM and E-RMBM have more capabilities than the B-RMBM, obviously either F-RMBM(n, j) or E-RMBM(n, j) can simulate a step of a MASCv(n, j) in O(v) time, without setting fuse switches or segment switches.

Let’s discuss a simulating an F-RMBM using a MASC. The idea of the algorithm is for the MASC to compute the buses that can reach each fuse line by an approach similar to transitive closure by recursive doubling. Let Z denote an F-RMBM(n, m) with n processors and m buses and Q denote a MASC(n^2, n) with n^2 PEs and n ISs. The same row and column groupings as in Section 4.4 can be used to assign PEs to ISs in Q. Each PE_{i,j} holds two m-bit strings F_{i,j} and C_{i,j} in which each bit f of F_{i,j} indicates whether p_j has set its fuse switch on bus f and each bit c of C_{i,j} indicates whether buses that p_i has fused are connected to bus c. Initially, each bit of F_{i,j} and C_{i,j} is 0 indicating no switches are set. Note that a processor of Z can have multiple fuse switches in the set position at the same time but can have at most one read or write switch set to one bus at the same time.

In each step, each Z processor performs the same operations as stated in section 4.4. Q simulates each of these operations as follows (since steps 2, 3, and 5 are identical to those in section 4.4, the details are omitted here).

Step 1: Set/reset a fuse switch – Assign PEs to ISs using the column grouping. For each j, if p_j sets or resets the fuse switch on bus h, then IS_j broadcasts to its PEs (i.e., PE_{i,j}, for all i) the bus ID h and whether the action was set or reset. Each PE_{i,j} reads and updates string F_{i,j}.

Step 2: Move read or write connection

Step 3: Write
Step 4: Read - Assign PEs to ISs using the row grouping. If \( p_i \) reads from bus \( g \), then determine the group of fused buses that includes bus \( g \). \( Q \) performs the following steps to check common fused buses. All ISs have their PEs copy \( F_{i,j} \) to \( C_{i,j} \). Execute Steps 4.1-4.3 for \( \log m \) iterations to flag in \( C_i \), the buses connected to buses fused by \( p_i \). Figures 4.3 illustrates the operations in this step for a situation in which \( p_2 \) reads from bus \( g \) and \( m = 2 \).

Step 4.1: Reassign PEs to ISs using the row grouping. Each PE sets \( \text{flag}_{i,j} \) to 0. Each IS \( i \) broadcasts \( C_{i,i} \) to \( \text{PE}_{i,j} \). Each \( \text{PE}_{i,j} \) computes \( C_{i,i} \) AND \( C_{i,j} \). If at least one bit of the result is 1, then \( \text{PE}_{i,j} \) sets \( \text{flag}_{i,j} = 1 \). If \( \text{flag}_{i,j} = 0 \), then deactivate \( \text{PE}_{i,j} \).

Step 4.2: For \( k = 1 \) to \( \left\lceil \frac{m}{v} \right\rceil \), repeat the following. IS \( j \) computes the global OR of the \( k \text{th} \) block of \( v \) bits of \( C_{i,j} \) of all active \( \text{PE}_{i,j} \), and each \( \text{PE}_{i,i} \) stores the result in the \( k \text{th} \) block of \( v \) bits of \( C_{i,i} \).

Step 4.3: ISs activate all PEs deactivated in Step 4.1, if any. Reassign PEs to ISs using the column grouping. Each IS \( j \) broadcasts \( C_{j,j} \) to \( \text{PE}_{i,j} \) and \( \text{PE}_{i,j} \) copies this data to \( C_{i,j} \).

Step 4.4: Use the bus connectivity information generated in Steps 4.1-4.3 to collect and reduce data written to the buses connected to bus \( g \) from which a processor \( p_i \) reads.

Step 4.4: Reassign PEs to ISs using the row grouping. IS \( i \) broadcasts \( g \), the index of the bus from which \( p_i \) wishes to read, to all \( \text{PE}_{i,j} \). Each \( \text{PE}_{i,j} \) checks if bit \( g \) of \( C_{i,j} \) is 1. Use a Pick-One to select one \( \text{PE}_{i,h} \) satisfying this condition. IS \( i \) broadcasts \( C_{i,h} \) from this PE to all \( \text{PE}_{i,j} \). For \( \text{PE}_{i,j} \), if \( p_j \) writes to bus \( e \) and bit \( e \) of \( C_{i,h} \) is 1, then \( \text{PE}_{i,j} \) is active; otherwise inactive. (If no \( \text{PE}_{i,h} \) has bit \( g \) of \( C_{i,h} \) as 1, then IS \( i \) broadcasts this result to all \( \text{PE}_{i,j} \). For \( \text{PE}_{i,j} \), if \( p_j \) writes to bus \( g \), then \( \text{PE}_{i,j} \) is active; otherwise inactive.) Active processors write
and, in the event of concurrent writes on Z, IS, computes the bus data using associative operations and Pick-One appropriate to the write conflict resolution rule of Z.

Figure 4.3: Illustration of Step 4 -- (a) Step 4.1 (b) Step 4.2 (c) Step 4.3 (d) Step 4.4
Step 5: Local Computation

Thus, we have the following theorem.

**Theorem 4.5:** Each step of an F-RMBM(n, m) can be simulated by a MASC(n^2, n) in $O(\lceil m/v \rceil \log m)$ time, if the MASC can perform $v$-bit width associative operations in constant time.

**Proof**

At the start of each simulated step, each PE$_{i,i}$ holds $F_{i,i}$ with all the fuse switches set by $p_i$. Steps 1-3 distribute information on set fuse switches, the positions of read and write connections, and the value written by $p_i$ to all PE$_{i,j}$. Steps 4.1-4.3, over log $m$ iterations, calculate in $C_{i,i}$ the buses that can reach the fuse line of $p_i$ in a manner much like a recursive doubling approach to transitive closure. A PE$_{i,j}$ is active after Step 4.1 if its $C_{i,j}$ and $C_{i,i}$ have a 1 in a common position, that is, if the set of buses known to reach the fuse line of $p_i$ and the set of buses known to reach the fuse line of $p_j$ have a common element. If they have a common element, then step 4.2 combines the two sets of buses because they can all reach each other. Step 4.3 distributes the updated information to be in the proper places for the next iteration. At the start of iteration $q$, each $C_{i,i}$ holds the set of buses that can reach the fuse line of $p_i$ by paths via up to $2^{q-1}$ fuse lines, and at the end of iteration $q$, it holds the buses that can reach the fuse line of $p_i$ by paths via up to $2^q$ fuse lines. At the start of step 4.4, $C_{i,i}$ holds connectivity information stored according to fuse line index. Step 4.4 extracts connectivity information according to bus index, handles multiple writes, and delivers data that processors of Z would read.

Steps 1, 2, 3, and 5 run in constant time. The time complexity of the algorithm depends on Step 4. Let $v$ denote the data width on which an IS can perform an associative operation in
constant time, and let \( b \geq v \) denote the bus width. Assume \( b \) is at least \( \max\{\log m, \log n\} \). Each iteration of steps 4.1 and 4.3 runs in \( O(m/b) \) time, and each iteration of step 4.2 runs in \( O(\lceil m/v \rceil) \) time. The one execution of step 4.4 runs in \( O(m/b) \) time. Since \( b \geq v \), step 4 runs in \( O(\lceil m/v \rceil \log m) \) time, and this is also the time complexity of the overall algorithm.

Note: If the F-RMBM uses Priority CW, then each IS computes bus data for step 4.4 in \( O((\log n)/v) \) time; otherwise, each IS computes bus data in constant time.

To simulate the E-RMBM, we use a simulation of an E-RMBM on an F-RMBM.

**Theorem 4.6** Each step of an E-RMBM\((n, m)\) can be simulated by a MASC\(((n+2m)^2, 4nm)\) in \( O(\lceil m/v \rceil \log m) \) time, if the MASC can perform \( v \)-bit width associative operations in \( O(1) \) time.

**Proof**

An F-RMBM\((n+2m, 4nm)\) can simulate each step of an E-RMBM\((n, m)\) in \( O(1) \) time \[55\]. The result follows from Theorem 4.5.

4.6 Relating MASC\(_c\) and MASC\(_{\log n}\) Using the PRAM Model

So far, simulations in sections 4.3-4.5 have specified that the MASC can perform constant time associative operations on \( v \)-bit wide data, where \( v \) is a variable. If the MASC processes data in a bit-serial manner \[26\], then constant time operations would imply a constant value for \( v \). When \( v \) is a function of \( n \), for example, \( \Theta(\log n) \) in Theorems 4.2 and 4.3, then it appears that the assumption of constant time associative operations for a MASC\(_{\log n}\) is overly generous. In this section, however, we establish that a constant time simulation of MASC\(_{\log n}\) by
MASC_\text{c} exists when the number of PEs increases by a polynomial factor. Consequently, associative operations can be performed in constant time on MASC_{\log n} as well.

Next section presents a cycle of simulations linking MASC_\text{c} and MASC_{\log n}. To start, Lemma 4.7 describes a constant time simulation of a PRAM by a B-RMBM.

**Lemma 4.7**: Each step of a PRAM(n, s) can be simulated in O(1) time on a B-RMBM(n+s, s), where the PRAM and B-RMBM have the same concurrent or exclusive read and write capabilities [55].

Next, recall Theorem 4.2(i) with an O(m/j) time simulation of a B-RMBM by a MASC_\text{c}, where m is the number of buses on the B-RMBM and j is the number of ISs on the MASC_\text{c}. When j = \Theta(m), this is constant time. Clearly, a MASC_\text{c} can be simulated on a MASC_{\log n}.

Stringing these results together, a PRAM can be simulated on a B-RMBM which can be simulated on a MASC_\text{c} which can be simulated on a MASC_{\log n}. The next result establishes that a MASC_{\log n} can be simulated on a PRAM, closing the cycle.

**Lemma 4.8**: Each step of a MASC_{\log n}(n, j) can be simulated in O(1) time on a Common CRCW PRAM(n^2, n + j).

**Proof**

Let S denote a MASC_{\log n}(n, j) and construct a Common CRCW PRAM(n^2, n + j) R that will simulate each step of S in constant time. S can perform constant time associative operations
on log $n$-bit wide data; let the data width of $R$ be $\Theta(\log n)$ bits, a standard assumption for PRAMs. Each processor of $R$ can perform the local processor actions of the corresponding processor of $S$, so we just need to focus on the associative operations.

Let $p_{g,h}$ denote a processor of $R$, where $0 \leq g, h < n$. For $IS_i$ with $k$ PEs in the set $\{i_1, i_2, \ldots, i_k\}$, let $T(i)$ denote the team of $k^2$ processors of $R$ that will simulate the associative operation of $IS_i$; this team comprises processors $p_{g,h}$ such that $g, h \in \{i_1, i_2, \ldots, i_k\}$. Let $TS(i)$ denote the small team of $k$ processors of $R$ that will simulate some associative operations of $IS_i$; this team is the subset of $T(i)$ comprising processors $p_{g,g}$ such that $g \in \{i_1, i_2, \ldots, i_k\}$. $R$ has one shared memory cell per PE, denoted $cp_g$, for $0 \leq g < n$, and one shared memory cell per $IS$, denoted $cis_i$, for $0 \leq i < j$.

Each $IS$ of $S$ may simultaneously perform any associative or computational operation independent of other $IS$s in the same step. We now describe the simulation of the associative operations of $IS_i$ with $k \leq n$ PEs where each PE holds an operand.

- **OR, AND:** $TS(i)$ can compute the OR or AND of $k$ binary values in constant time using shared memory cell $cis_i$ with Common CW.

- **Maximum, Minimum:** $T(i)$ can compute the maximum or minimum of $k$ values, each up to $\log n$ bits wide, in constant time using Common CW. As in the proof of Theorem 4.1, the idea for maximum is to compare all pairs of elements and use AND operations to identify the maximum value [34][47].

- **Associative search:** $TS(i)$ can perform an associative search in constant time using shared memory cell $cis_i$ with CREW. Write the search value in $cis_i$, then each processor in $TS(i)$ can read $IS_i$’s value and compare that to its PE’s value.
• Any-Responder: TS(i) can perform an Any Responder operation in constant time using shared memory cell cis, with Common CW. TS(i) computes an OR where those processors that match (do not match) the search contribute 1 (0).

• Pick-One: T(i) can perform a Pick One operation in constant time by computing maximum or minimum among IDs of PEs that matched an associative search.

Observe that for any two ISs, IS_i and IS_f, T(i) and T(f) are disjoint and the shared memory cells that they use are disjoint, so R can independently and simultaneously perform the associative operations for all ISs.

From this cycle and other previously known results, Theorem 4.9 follows.

**Theorem 4.9:** Each of the following models can simulate a step of any other in constant time with at most a polynomial factor increase in hardware resources: PRAM, S-RMBM, B-RMBM, MASC_c, and MASC_{log n}.

Based on this theorem, the terms MASC_c and MASC_{log n} can be used interchangeably in most cases if no confusion is caused.

### 4.7 Extension of Relationships among MASC, PRAM, and RM

In [11][55][58], relative powers of the PRAM, RMBM, and RM models have been well studied. Related models can be placed into two groups, i.e.,

**Group 1:** B-RMBM, S-RMBM, PRAM
Group 2: F-RMBM, E-RMBM, RM, LRM.

All models in the same group have the same power. Any model in Group 2 is more powerful than any model in Group 1.

Based on these relationships and the simulation results that we have presented, we have the following observations. Note that all cases are CRCW.

1) Since constant time simulations between MASC and B-RMBM and constant time simulations between MASC and S-RMBM exist, MASC goes into the first group in terms of its power. MASC has the same power as PRAM.

2) The MASC used throughout the chapter is a MASC without cell and IS networks. Akl found that a PRAM can simulate any interconnection network as well as any combinational circuit in constant time [1]. From the observation (1) and the fact that cell and IS networks can be simulated by a Common CRCW PRAM in constant time, the MASC model used in this chapter is equivalent in power to a generic MASC model that has both cell and IS networks.

3) From the observation (2) and lemma 3.1 in chapter 3, we can conclude that a generic MASC model is also equivalent in power to a MASC model using the manager-worker paradigm.

4) When applying a special case of Heide and Pham’s model [21] to an MMB, regardless of relative values of number n of processors and m of buses, MMB needs $\Omega(\log \log n)$ time to compute the maximum. As MASC can do this in constant time, MMB does not belong in Group 1.
5) Since MASC has the same power as the models in Group 1, it is less powerful than RM, LRM, E-RMBM, and F-RMBM.

6) It can be shown that S-RMBM has the same power as BRM, a basic RM (the proof is omitted). Although our previous BRM simulation of MASC [4] takes non-constant time, a constant time BRM simulation using MASC is possible.

Therefore, our modified grouping to the above relationships is as follows.

Group 1: MASC, B-RMBM, S-RMBM, PRAM, BRM

Group 2: F-RMBM, E-RMBM, RM, LRM

Simulations between MASC and versions of RMBM--reconfigurable bus-based models--have been presented in this chapter. The MASC and the basic RMBM can simulate each other in constant time with appropriate assumptions. Also, MASC can simulate the segmenting RMBM in $O((\log n)/v)$ time and the fusing RMBM and extended RMBM in $O\left(\left\lceil \frac{m}{v} \right\rceil \log m \right)$ time. By taking advantage of established relationships between RMBM, RM, and PRAM, we have categorized the power of the MASC model in relation to those of RM and CRCW PRAM. The power of the MASC model is comparable to that of CRCW PRAM but less than that of RM. For some special cases of RM with restrictions, however, a MASC can perform a constant time simulation (e.g., simulations of MMB and BRM). Both RM and PRAM are very popular and important models for parallel computation; especially RM has been widely accepted as an
extremely powerful parallel computation model. Our research results provide a better understanding of the MASC model concerning its power.
CHAPTER 5

A MASC Software Simulator Using the Manager Worker Paradigm

5.1 Introduction

A software simulator of the MASC model using manager-worker paradigm (without a cell network) was written in C++. The MASC simulator is capable of executing MASC programs written in C++. The MASC library provides supports for the associative properties of the MASC model such as global MAX/MIN and AND/OR reductions, an associative search, and Any-Responder and Pick-One operations. The first objective of the MASC simulator was to provide a digital environment to test, debug, and evaluate MASC algorithms and programs. Another objective is to use the simulator as a tool to both evaluate the MASC model for a purpose of improving the model in the future.

5.2 Previous Work

A previous work on the ASC language backend compiler for a MASC runtime system is an experimental work done by Chantamas [8] as described in section 5.2.1. A user directive is used in a form of an ASC comment to enable concurrent of executions of branches. The backend compiler uses ASC intermediate code to generate MASC intermediate code. The MASC
intermediate code may be run on a MASC emulator later on. Unfortunately, the project halted and no MASC emulator has been developed for an ASC language.

5.2.1 Previous Work on an ASC Language Compiler Extension

The Associative Computing (ASC) language was developed during the 80’s to be an alternative to the SIMD machines. In the past, ASC programs were compiled and implemented on several parallel computers such as the STARAN, the ASPRO, the Wavetracer, and the Connection Machine [36]. The language was intended for an associative SIMD computer and is based on the SITDAC (Single Instruction Tabular Data Associative Computing) architecture model for associative computing given in [36]. Data in the SITDAC model are stored in a two-dimensional tabular data structure. An associative searching or processing can be done efficiently by using a massively parallel SIMD computer.

Figure 5.1: The process of compiling and executing of a conventional ASC program.
When compiling an ASC program using the ASC compiler, the compiler will generate an intermediate code. This intermediate code is intended to run on an associative SIMD, i.e., a STARAN, a traditional SIMD, or an ASC emulator which emulates a parallel associative computer with one instruction stream. Due to the introduction of the multiple instruction stream concept for the ASC model called the MASC model, a new compiler is needed to compile an ASC program and produces an intermediate code that can be executed on a MASC computer.

In the ASC language, the branches in a parallel if-then-else statement can be executed simultaneously provided two instruction streams are available. An if-then-else statement is defined as a parallel if-then-else statement if the result after the evaluation of the condition expression is a parallel result. If the result is a scalar result, the statement is a scalar if-then-else statement. Only the then part or the else part of a scalar if-then-else statement will be executed, but not both. The branches in a parallel case statement can also be executed in parallel, but this process is similar to that of the branches in a parallel if-then-else statement, and is not addressed here. Normally, in order to execute a parallel if-then-else statement in the ASC model, the instruction stream searches for responders that satisfy the condition expression. The responders are active and execute the statement in the then block. Then, the instruction stream does another search for non-responders to the previous search which now become active responders, and execute the statement in the else block. However, any processor that is inactive at the time the conditional statement is evaluated remains inactive during the execution of the if-then-else statement and does not execute either then or else block.
Figure 5.2: The process of compiling an ASC program with MASC directives.

Figure 5.3: An ASC program with no MASC directive used.
A user needs to place a MASC directive, /* .MASC fork */, before a parallel if-then-else statement in the ASC program to enable simultaneously executions of both then and else parts of the parallel if-then-else statement.

Figure 5.4: The same ASC program as in Figure 5.3 with a MASC directive used.

The directive tells the MASC backend compiler to generate exactly two tasks to be executed simultaneously, one from the then part and another from the else part. If there is another parallel if-then-else statement embedded within either the then or the else part, that parallel if-
then-else statement will not be effected by this .MASC fork directive. If the user wants the MASC backend compiler to generate two more tasks from that embedded if-then-else statement, another .MASC fork directive is required.

5.2.2 Previous Work on a MASC Runtime System

Although a previous work on the MASC model has been reported [46], this current work is different from the previous work in the following aspects.

First, a user has a tight control when to use multiple instruction streams to execute branches on program execution. There is always an overhead when using a multiple instruction streams to execute a branch, i.e., extra costs to fork and join tasks. If the overhead from using multiple instruction streams outweighs the benefit gained from using one instruction stream, a user has a choice to use only a single instruction stream to execute the branch.

Second, when using a non static ask assignment scheme, there is no requirement that a number of tasks generated for the model must be a direct proportion to the total number of instruction streams. Since the model uses a task pool technique, the number of tasks can be more (or less) than the number of instruction streams available. Tasks in the task pool will be assigned to an available worker instruction stream one at a time based on a scheduling policy selected (by the id priority) until there is no task available in the task pool or no available worker instruction stream left.

Third, when using a non static task assignment scheme, a worker instruction stream is free to be assigned a new task as soon as it finished the previous assigned task. There is no need
to wait for the last instruction streams generated from the same fork operation to finish its task before it can proceed to execute a new task.

Last, the testing of the conditional expression of a forked branch is done by a manager instruction stream. This approach simplifies the model and is easier to keep track.

5.3 The MASC Simulator

The MASC software simulator was written using a 32-bit version of the Microsoft Visual C++ 2008 Express Edition (as a Win32 console application). Once the MASC program has been compiled, it should be able to run natively on any 32-bit Windows platform. On a 64-bit Windows platform, the program will be run in an emulation layer since the 64-bit Windows Vista operating system is not capable of natively running 32-bit code [41]. The virtual address space of the simulator process is limited to 2GB for a 32-bit application. As the result, the number of ISs and PEs are limited to fixed numbers depending largely on how big each PE’s local memory is.

The simulator has two components. The first component simulates the hardware aspect of the model such as the instruction stream broadcast and reduction network, manager and worker instruction streams, broadcast and reduction networks, and PEs. The second component simulates the software aspect of the model such as the associative properties and the management of worker instruction streams and tasks.
5.3.1 Tasks and Task Pools

There are three types of tasks and another three types of task pools used by the simulator. Each one of the three types of tasks associates with one specific type of task pools. Worker tasks are parts of the user program and are associated with partitions of PEs. Worker tasks are executed by workers. Unassigned worker tasks are queued in the worker task pool. Fork tasks are control structure tasks needed for the flow of the program along with their associated PEs. These fork tasks are executed after been taken out from the fork pool by the manager during the FORK phase of the simulation cycle of the manager. During an execution of each fork task, the associated PEs will be partitioned into one or more partitions (depending on the condition statement of that fork task). Each new PE partition will be associated with a part of user program to create a new worker task (with a new task ID). Thus, one fork task creates one or more worker tasks. Join tasks, along with their associated PEs, are another control structure tasks needed for the flow of the program. Join tasks are executed after been taken out from the join pool by the manager during the JOIN phase of the simulation cycle of the manager. During an execution of each join task, the partitions of associated PEs will be combined (repartitioned) into one partition and assigned as a new fork task (with a new task ID). In most algorithms, join tasks can be embedded as parts of fork tasks and the JOIN phase of the simulation cycle of the manager may be skipped entirely. Thus, one join task, whose PEs are from one or more finished worker tasks, creates one new fork task.

Normally, tasks are predetermined and provided to the simulator in form of a task graph. A task graph tells the simulator the order and dependency of all tasks that will be created during the execution of the program. The manager checks the task graph to determine which new tasks
will be inserted next into which of the task pools during the course of the execution of the program.

5.3.2 Simulating the Hardware Aspect of the MASC Model

The manager instruction stream is assigned an instruction stream ID of 0. Each cycle of the simulation of the manager instruction stream may go through five phases as denoted by FFAJT (Finished-Fork-Assign-Join-Termination). Figure 5.5 shows the simulation cycle. During a simulation cycle, some phases may be skipped but at least one phase must be simulated.

![Figure 5.5: A simulation cycle of the manager instruction stream.](image)

**Finished (F):** This is the first phase of a manager instruction stream simulation cycle. The manager collects finished tasks from workers. The manager does an associative search on its workers to see if any of them has a finished task. If there is no responder, the manager moves on to the next phase. Otherwise, after the manager has obtained IDs of those tasks, it marks those
finished tasks as finished, adds children tasks of those finished parent tasks to a task pool, and broadcasts to those workers to change their states to from Finish to Ready.

**Fork (F):** This is the second phase of a manager instruction stream simulation cycle. The manager forks children tasks from prior finished parent tasks. The manager broadcasts an instruction (a parallel condition) to a group of PEs from a prior finished parent task in order to partition them into two children partitions. A new task ID is assigned to PEs of each child partition. The manager adds new created children tasks into a task pool.

**Assign (A):** This is the third phase of a manager instruction stream simulation cycle. The manager assigns new tasks from the task pool to workers. The manager does an associative search on its workers to see if any of them is ready. If there is no responder, the manager moves on to the next phase. Otherwise, it retrieves the first task from the task pool, switches, PEs whose task IDs match the id of the retrieved task, to listen to the ready worker, and broadcasts to the worker about the assigned task.

**Join (J):** This is the fourth phase of a manager instruction stream simulation cycle. The manager joins finished children tasks into one task. Normally, two finished sibling tasks are combined into a new child task. The manager broadcasts to PEs, whose task IDs match any IDs of the two finished sibling tasks, in order to update their IDs to that of the child task and adds the new task into a task pool.

**Termination (T):** This is the last phase of a manager instruction stream simulation cycle. The manager checks for a termination state. The program will be terminated if all of these conditions are true: the task pool is empty, no task is waiting to be forked or joined, and all workers are in Ready state.
Each worker is assigned an instruction stream ID from 1 to \( j-1 \) for a MASC(n, j). A worker has three states, Ready, Busy, and Finish as shown in Figure 5.6.

![Diagram of worker states](image)

**Figure 5.6: States of a Worker.**

**Ready:** This is the initial state for all workers. At this state, no PE is associated with a worker. The worker is idle and waits for the manager to assign it a task.

**Busy:** A worker changes its state from Ready to Busy after the manager has assigned it a task. At this state, a task—a set of instructions along with a group of PEs—is assigned to the worker. In a rare case, the group of PEs may be an empty set. Nevertheless, the worker executes the assigned task following the flow of the program.

**Finish:** After the worker finished an execution of the assigned task and switched its PEs back to the manager, it changes its state from Busy to Finish. At this state, its PEs are no longer
associated with the worker. After the manager has collected the finished task, the worker changes its state from Finish to Ready.

Each PE is assigned a PE ID from 0 to n-1 for a MASC(n, j). A data record in each PE comprises two parts; PE control data field used for system operations of a PE, and its data fields.

The network is simulated by software based on a network using 2-PE resolver circuits (instead of 4-PE resolver circuits). A simulation of a reduction is done at one 2-PE resolver circuit at a time. Intermediate results are passed up to their 2-PE resolver circuits until reaching the instruction stream.

5.3.3 Simulating the Software Aspect of the MASC Model

The software aspect of the MASC model is simulated using various C++ functions provided as follows and in the appendix B.

Global AND reduction of Boolean and integer data

Functions: BoolAndReduction and IntAndReduction.

Each function performs a global AND reduction of Boolean or integer data by using a broadcast and reduction network.

Input: Boolean or integer data to be reduced and a task id.
**Output:** A global AND reduction of the input Boolean or integer data at the root node of the reduction network.

### Global OR reduction of Boolean and integer data

**Functions:** BoolOrReduction and IntOrReduction.

Each function performs a global OR reduction of Boolean or integer data by using a broadcast and reduction network.

**Input:** Boolean or integer data to be reduced and a task id.

**Output:** A global OR reduction of the input Boolean or integer data at the root node of the reduction network.

### Global MAX reduction of integer and floating point data

**Functions:** IntMaxReduction and RealMaxReduction.

Each function performs a global MAX reduction of integer or floating point data by using a broadcast and reduction network.

**Input:** Integer or floating point data to be reduced and a task id.

**Output:** A global MAX reduction of the input integer or floating point data at the root node of the reduction network.
Global MIN reduction of integer and floating point data

**Functions:** IntMinReduction and RealMinReduction.

Each function performs a global MIN reduction of integer or floating point data by using a broadcast and reduction network.

**Input:** Integer or floating point data to be reduced and a task id.

**Output:** A global MIN reduction of the input integer or floating point data at the root node of the reduction network.

Associative search of Boolean, character, integer, floating point, and string data

**Functions:** BoolAssoSearch, CharAssoSearch, IntAssoSearch, RealAssoSearch, and StringAssoSearch.

Each function performs an associative search of Boolean, character, integer, floating point, or string data.

**Input:** Boolean, character, integer, floating point, or string data to be searched, a search key, and a task id.

**Output:** For each PE in the partition, if a search key is found, the mask register of the PE is set to 1 (true). Otherwise, the mask register is set to 0 (false).
**Any-Responder**

**Function:** AnyResponder.

This function looks for any responder among PEs in a partition by checking at their mask registers. This function should be performed after an associative search and uses a broadcast and reduction network.

**Input:** a task id.

**Output:** If at least one PE in the partition has a mask register set to 1 (true) (by utilize a global OR reduction of Boolean data in mask registers), a 1 (true) is reduced back to the root node of the reduction network. Otherwise, a 0 (false) is reduced back to the root node.

---

**Pick-One**

**Function:** PickOne.

This function picks one PE out of many responders. The function should be performed after a call of Any Responder returned 1 (true). A call of PickOne, when there is no responder, returns -1 as the id of a PE. Since a PE id starts from 0, a negative id should be treated as an error.

**Input:** a task id.

**Output:** If there is at least one responder, by performing a global MAX reduction of responders’ ids, the largest id of responders is returned. If there is no responder, a -1 is returned.
Reduce

Functions: ReduceBool, ReduceInt, ReduceReal, ReduceChar, and ReduceString.

By using a broadcast and reduction network, any specific data from a PE can be reduced back to its instruction stream.

Input: a PE id and data to be reduced.

Output: The specific data is returned.

A full list of functions in the MASC library with their worst case cost analysis is provided in appendices B and C at the end of this dissertation. A programmer, who wishes to program a MASC program, is encouraged to check out appendix A: a guide on how to program on a MASC simulator.

5.3.4 How the Simulator Determines the Precise Cycle Count

Each instruction executed by the simulator will be assigned a cost to it. All simple operations cost one step (or cycle) for the simulator to execute. Simple operations are those assignment, arithmetic, and logical operations with no more than two variables. These are examples of simple operations that cost one cycle each; \( A = B, \ C + D, \ E \ OR \ F, \) and \( G < H. \)

More complex operations cost more than one cycle for the simulator to execute. Normally, a complex operation consists of two or more simple operations. For an example, \( A = C + D \) costs two cycles; the first cycle is \( C+D \) and the second cycle is assigning \( C+D \) to \( A. \) Another
example, IF (A > 10) THEN ELSE costs two cycles; one for evaluating A > 10 and another for proceeding to either the THEN or ELSE parts.

The numbers of cycles of associative operations are determined by their worst case. Most operations cost more than one cycles in the simulator. These operations consist of many simple and smaller associative operations. The only two exceptions that cost only one cycle are broadcasting data using the broadcasting network and reducing data using the reduction network. Both operations deal with the hardware part of the model.

Let us look at one example, a Boolean AND reduction operation. A complete list of the cost of associative operations is shown in appendix C.

**Function Name: BoolAndReduction**

**Cost**

1. +1 for broadcasting from an IS to PEs
2. +4 for an IF statement with 3 comparisons in the conditional statement
3. +1 for an assignment inside the THEN/ELSE
4. +1 for the time needed for the value to propagate back to the IS

The total cost is 1+4+1+1 = 7 cycles.

When a user program is executed by the simulator, the simulator counts every cycle used by the program both by workers and the manager (the actual working cycles and the overhead cycles) and reports back when the execution is terminated.
CHAPTER 6

A MASC Floyd-Warshall All-pairs Shortest Path Algorithm

6.1 Introduction

This chapter describes a MASC implementation of the Floyd-Warshall algorithm. This algorithm is an example of a static task assignment scheme used by the manager. The important characteristics of an algorithm to be used a static task assignment scheme are, first, the computation time per task (a partition of PEs and instructions) is constant and, second, the number of tasks is static for a given problem size. Mapping of problem tasks in the algorithm to instruction streams are predetermined (cannot be changed during runtime) and done statically. Assignments of tasks to instruction streams can be done simultaneously using a constant number of broadcasts to PEs and workers by the manager. Up to n concurrent tasks can be assigned to n worker instruction streams at a time for an input graph G with n vertices. The task assignment cost remains constant regardless of the number of assigned tasks generated by the algorithm.

6.2 The Sequential Algorithm

The Floyd-Warshall algorithm is an algorithm to find shortest paths between every pairs of vertices in a weighted directed graph purposed by Floyd [15]. The algorithm is based on a theorem by Warshall [61], which described how to compute a transitive closure of Boolean
matrices. Floyd-Warshall algorithm solves the all-pairs shortest path problem by transforming a slightly modified adjacency matrix for the graph into a matrix whose \((i, j)\) entry contains the shortest distance from \(v_i\) to \(v_j\) for all pairs of vertices. The slightly modified adjacency matrix used in this process has the weight \(d(v_i, v_j)\) of the edge from vertices \(v_i\) to \(v_j\) in its \((i, j)\) entry. In addition, \(d(v_i, v_j)\) is set to 0 if \(i = j\) and set to \(\infty\) if there is no edge between \(v_i\) and \(v_j\) [10][15].

Let \(G = (V, E)\) be a directed graph with \(n\) vertices and \(m\) edges and assume that each edge \(e \in E\) is assigned a nonnegative integer weight. Then \(V = \{ v_1, v_2, ..., v_n \}\) and for each \(k\) with \(1 \leq k \leq n\), define \(V_k = \{ v_1, v_2, ..., v_k \}\). Each edge \(e \in E\) is assigned a nonnegative integer weight. The weight of a path in the graph is the sum of the weights of its edges. The Floyd-Warshall algorithm is based on the following observation. Let \(A_0 = (a_{ij}^0)\) denotes the adjacency matrix of the graph, with the following changes: reset \(a_{ij}^0 = 0\) when \(i = j\) and \(a_{ij}^0 = \infty\) when there is no edge from \(v_i\) to \(v_j\). For \(1 \leq k \leq n\), let \(A_k\) be the matrix \((a_{ij}^k)\) where the entry \(a_{ij}^k\) is the shortest distance from \(v_i\) to \(v_j\) that only has intermediate nodes from \(V_k\) along its path. Then the \((i, j)\) entry \(a_{ij}^n\) of \(A_n\) gives the length of the shortest path between \(v_i\) to \(v_j\). The Floyd-Warshall algorithm calculates the matrix \(A_n\) by successively converting the matrix \(A_{k-1}\) to \(A_k\) for \(k = 1, 2, ..., n\). This conversion process is described next.

Assume that matrices \(A^0, ..., A_{k-1}\) have been computed, where \(0 < k < 1\). Then \(A_k\) can be computed a follows. Note that the length of the shortest path \(p\) from \(v_i\) to \(v_j\) with all intermediate vertices from \(V_{k-1}\) is \(a_{ij}^{k-1}\). The shortest path from \(v_i\) to \(v_j\) using only intermediate vertices from \(V_k\) will contain the vertex \(v_k\) at most once. Moreover, if this path contains \(v_k\), it will consist of the shortest path \(p1\) from \(v_i\) to \(v_k\) consisting of only vertices from \(V_{k-1}\) combined with the shortest path \(p2\) from \(v_k\) to \(v_j\) consisting only of nodes from \(V_{k-1}\). Moreover the length of this new path through \(v_k\) will be \(a_{ik}^{k-1} + a_{kj}^{k-1}\). The other possibility is that the shortest path \(p\) from \(v_i\) to \(v_j\) does not
contain the vertex $v_k$, so its intermediate vertices are all from $V_{k+1}$ and its length remains the same, namely $a_{ij}^{k-1}$ as given by $A_{k-1}$. Either way, the shortest path from $v_i$ to $v_j$ is the shorter of the earlier path $p$ with intermediate vertices from $V_{k+1}$ or the path consisting of $p_1$ and $p_2$. This procedure is illustrated in Figure 6.1.

![Diagram](image)

Figure 6.1: Path $p$ is a shortest path from vertex $v_i$ to vertex $v_j$.

The Floyd-Warshall algorithm is based on this technique and is given below.

**The Floyd-Warshall Algorithm**

**Input:** An adjacency matrix of graph $G$ containing tentative distances, $d[v_0..v_{n-1}, v_0..v_{n-1}]$.

**Output:** The final adjacency matrix containing all-pairs shortest path between every pair of vertices
Algorithm

1. for k = 0 to n-1
2. for i=0 to n-1
3. for j=0 to n-1
4. Set d(v_i, v_j) to the minimum of d(v_i, v_j) and d(v_i, v_k)+d(v_k, v_j)
5. end for
6. end for
7. end for

The running time of the Floyd-Warshall algorithm as shown in Figure 6.2 is determined by the triply nested for loops of lines 1-3. Line 4 solves the problem sequentially by compute each tentative distance, individually, and takes O(1). The algorithm runs in Θ(n^3).

Many studies have been done on the all-pairs shortest path problem. Johnson purposed a faster algorithm to solve all-pairs shortest path problem on sparse graphs that runs in O(n^2 lg n + n|E|) [27]. Chan presented an improved Floyd-Warshall algorithm with the running time approaching O(n^3/log^2 n) for general real-weighted dense graphs without using fast matrix multiplication [7]. In terms of implementations, Katz implemented a shared memory cache efficient GPU Floyd-Warshall algorithm on a NVIDIA G80 GPU architecture using a CUDA API [28]. Han proposed a program generator for the Floyd-Warshall algorithm to produce highly optimized source code directly from a problem specification [19]. Han used techniques such
tilling, loop unrolling, and SIMD vectorization combined with a hill climbing search in the
program generator for a given platform and gains a speedup of 1.3 on a PC. Even though the
Floyd-Warshall algorithm is not the fastest algorithm, it is practical for up to moderate-size input
graphs since the code is tight and contains no elaborate data structures.

6.3 A MASC Floyd-Warshall Algorithm

In peta scale computing, graph-theoretic problems pose serious challenges on current
traditional parallel machines due to non-contiguous, concurrent access to global graph data
structures with low degrees of locality. There is a need of a non-traditional massive multithreaded
architecture as a platform to execute these types of large scale graph problems [31]. Given that
the MASC model is designed to support multiple ASC threads using control parallelism, the
MASC model seems to be a reasonable candidate for a platform to execute large scale graph
problems. Moreover, the MASC model is closer to the theoretical PRAM model than the widely
used shared memory symmetric multiprocessor system.

Grama et al. describe a parallel implementation of the Floyd-Warshall algorithm based on
a block checkerboard data decomposition [2]. The MASC implementation of the algorithm in this
chapter is based on an MPI version described by Quinn using row-wise and column-wise block-
striped decompositions [43]. The overall time complexity of Quinn’s version is \(O(n^3/p + n^2 \log p)\)
where \(n\) is the number of vertices and \(p\) is the number of processors. Our MASC version of the
algorithm has a running time of \(O(n)\) when a MASC\((n^2, n+1)\) is used.
6.3.1 Expressing Parallelism for the Floyd-Warshall Algorithm

Let us consider the input graph $G = (V,E)$ and its the matrix $A_0$ consisting of its first approximation of path length using edge lengths. The adjacency matrix has to be altered with path estimates between identical vertices set to 0 and estimates between two vertices not jointed by an edge set to $\infty$, which is denoted as MAX_INT. The input matrix is divided into $n^2$ elements. Let us assume that one PE is responsible for an element of the matrix as shown in Figure 6.2.

![Figure 6.2](image)

Figure 6.2: The adjacency matrix of a 4-vertex input graph is divided into $4^2$ elements.
PEs are partitioned into either row-wise or column-wise block-striped decomposition during the course of execution. In general, each PE partition is controlled by a worker, if there are enough workers available, as shown in Figures 6.3 and 6.4. If there are not enough workers available, two or more PE partitions may be assigned to a worker.

Figure 6.3: Columnwise block-striped decomposition of an adjacency matrix.

Figure 6.4: Rowwise block-striped decomposition of an adjacency matrix.
6.3.2 A MASC Floyd-Warshall Algorithm

The MASC Floyd-Warshall algorithm consists of an algorithm for the manager and an algorithm for workers. The first algorithm described is for workers. The manager’s algorithm will be described next.

The data record of each PE contains the following variables.

• A current best distance from vertex \( v_i \) to vertex \( v_j \) denoted as \( d \).

• Two intermediate current best distances, one from itself to vertex \( v_j \) and another from vertex \( v_i \) to itself, denoted as \( dC \) and \( dR \), respectively.

• PE’s row and column numbers denoted as \( pe\_row \) and \( pe\_column \), respectively.

• The IDs of workers that the PE has to listen to during column-wise and row-wise block-striped decompositions are denoted as \( column\_worker \) and \( row\_worker \), respectively.

Each instruction stream contains the following scalar variables.

• A temporary variable for a tentative distance denoted as \( tmp \).

• Variables indicate the row and column an instruction stream is currently working on denoted as \( row \) and \( column \), respectively.

• A variable indicates which block-striped decomposition (row-wise or column-wise) is currently being used is denoted using the Boolean variable \( is\_row \).
**An Algorithm for Workers of MASC(n^2, m+1)**

**Input:** An adjusted adjacency matrix $A_0$ of $G = (E, V)$ consisting of its first approximation of path length using edge lengths.

**Output:** The final adjacency matrix containing all-pairs shortest path between every pair of vertices

1. The worker changes its state from Ready to Busy.

2. Initialize data by setting tmp to 0 and i to 0.

3. If workers are working with the column-wise block-striped decomposition, do the following.

4. For each column $i > 0$ within the column stripe, do the following.
   
   a. Set the mask register of each PE to 1 to reactivate deactivated PEs.

   b. Pin point a PE whose $pe_row$ is $row$ and $pe_column$ is $this_column$ by executing the following instructions.

      i. Do an associative search for scalar $row$ on $pe_row$.

      ii. Note that, after an associative search, all non-responders PEs are deactivated.

      iii. Set the scalar $this_column$ to $i + (worker IS id - 1) * stripe_size$.

      iv. Do an associative search for the scalar $this_column$ on $pe_column$.

      v. Get the current best distance, $d$, from this PE by executing the following instructions.

     1. Do a Pick-One on the only active PE left.
2. Reduce the current best distance, \( d \), from the PE and copy to the scalar \( \text{tmp} \).

c. Set the mask register of each PE to 1 to reactivate deactivated PEs.

d. Broadcast the reduced current best distance to all other PEs in the same column by executing the following instructions.

   i. Do an associative search for the scalar \( \text{this\_column} \) on \( \text{pe\_column} \).

   ii. Broadcast the scalar \( \text{tmp} \) to all active PEs. Each PE copies \( \text{tmp} \) to \( dC \).

e. Increase \( i \) by 1.

f. If \( i < \text{stripe\_size} \), go back to 3 to execute the next column within the column stripe. Otherwise, the execution of this task is finished. Continue to 9.

5. If workers are working with the row-wise block-striped decomposition, do the following.

6. For each row \( i \) within the row stripe, do the following.

   a. Set the mask register of each PEs to 1 to reactivate deactivated PEs.

   b. Pin point a PE whose \( \text{pe\_row} \) is \( \text{this\_row} \) and \( \text{pe\_column} \) is \( \text{column} \) by executing the following instructions.

      i. Do an associative search for scalar \( \text{column} \) on \( \text{pe\_column} \).

      ii. Set the scalar \( \text{this\_row} \) to \( i + (\text{worker IS id - 1} \times \text{stripe\_size}) \).

      iii. Do an associative search for the scalar \( \text{this\_row} \) on \( \text{pe\_row} \).
iv. Get the current best distance, $d$, from this PE by executing the following instructions.

1. Do a Pick-One operation on the only active PE left.

2. Reduce the current best distance, $d$, from the PE and copy to $tmp$.

c. Set the mask register of each PE to 1 to reactivate deactivated PEs.

d. Broadcast the reduced current best distance to all other PEs in the same row by executing the following instructions.

   i. Do an associative search for the scalar $this_row$ on $pe_row$.

   ii. Broadcast the scalar $tmp$ to all active PEs. Each copies $tmp$ to $dR$.

e. Increase $i$ by 1.

f. If $i < stripe\_size$, go back to 5 to execute the next row in row stripe. Otherwise, continue.

7. Relaxing the current best distance of each PE by executing the following instructions.

   a. Broadcast to all PEs in this row stripe to set the mask register of each PE to 1 in order to reactivate deactivated PEs.

   b. If $d > dC + dR$, then the PE relaxes the current best distance by set $d$ to $dC + dR$.

8. The execution of this task is finished. Continue to 9.

9. When the worker is done with the execution of the assigned task, the worker broadcasts to its PEs to switch the instruction stream they are listening to the manager, whose ID is 0.
10. PEs copy 0 to their instruction stream selector register.

11. The worker changes its state from Busy to Finish.

Each line of the algorithm takes $O(1)$. The total running time of each worker task depends solely on the stripe size. If MASC($n^2,m+1$) is used, the total running time of a worker IS task is $O(|V|/m)$. If MASC($n^2, |V| +1$) is used, then the total running time of a worker IS task remains constant.

An Algorithm for the Manager of MASC($n^2$, m+1)

1. Initialize data by setting $row = column = 0$, $is\_row = false$, $max\_row = n$, $is\_num = m+1$, define $\infty$ as MAX_INT, and $iteration\_count = 0$.

2. /* Column-wise Block-striped Decomposition. */

3. FINISHED Phase:
   a. /* No OP */

4. FORK Phase:
   a. Remove a fork task from the fork pool.
   b. Perform a global reduction on workers to see if all of them are in Ready state.
   c. If that is true, continue. If not, go back to 4b.
   d. Copy task ID "row + column\_worker" to PEs task ID register.

5. ASSIGN Phase:
   a. Broadcast to all PEs to switch the instruction stream they are listening to their $column\_worker$.
   b. PEs copy $column\_worker$ to their instruction stream selector register.
c. Broadcast to all workers to work on their column-wise block-striped tasks, ID \((row + \text{worker ID})\) at row \(row\).

6. **JOIN Phase:**
   a. /* No OP */

7. **TERMINATION Phase:**
   a. If task, fork, and join pools are empty and all workers are Ready, the execution is terminated.

8. /* Row-wise Block-striped Decomposition. */

9. **FINISHED Phase:**
   a. Perform a global reduction on workers to see if all of them are in Finish state.
   b. If that is true, continue. If not, go back to 9a.
   c. Broadcast to all workers to change their states from Finish to Ready.
   d. Set \(is\_row\) to true.
   e. Increase \(row\) by 1.
   f. Add a fork task into a fork pool.

10. **FORK Phase:**
    a. Remove a fork task from the fork pool.
    b. Perform a global reduction on workers to see if all of them are in Ready state.
    c. If that is true, continue. If not, go back to 10b.
    d. Copy task ID “\(max\_row + column + row\_worker + is\_num\)” to PEs task ID register.
       \(is\_num\) is the total number of instruction streams.

11. **ASSIGN Phase:**
    a. Broadcast to all PEs to switch the instruction stream they are listening to their \(row\_worker\).
b. PEs copy row_worker to their instruction stream selector register.

c. Broadcast to all workers to work on their row-wise block-striped tasks at column column on task ID “max_row + is_num + column + worker IS ID”.

12. JOIN Phase:

a. /* No OP */

13. TERMINATION Phase:

a. If task, fork, and join pools are empty and all workers are Ready, the execution is terminated.

14. FINISHED phase:

a. Perform a global reduction on workers to see if all of them are in Finish state.

b. If that is true, continue. If not, go back to line 14a.

c. Broadcast to all workers to change their states from Finish to Ready.

d. Set is_row to false.

e. Increase column by 1.

f. Increase iteration_count by 1.

g. Add a fork task into a fork pool if column < max_row. Otherwise, no task is added to any pool.

h. If iteration_count ≥ n, go to TERMINATION phase (line 7). Otherwise, go back to FORK phase (line 4).

Each line in the manager algorithm takes O(1) regardless of the number of workers used. Since there are |V| = n iterations, the total running time of the manager algorithm is O(n).
When combining the running time of both algorithms, the final total running time of the MASC version of the Floyd-Warshall algorithm is $O(n^2|V|/m)$ for MASC($n^2$, $m + 1$). If MASC($n^2$, $|V| + 1$) is used, then the final total running time of the MASC Floyd-Warshall algorithm is $O(n)$.

The efficiency of the MASC algorithm can be computed from the efficiency formula $E(1,p) = t_1/t_p$, where $t_1$ is the worst case running time of a sequential algorithm, $t_p$ is the worst case running time of the $p$-processor parallel algorithm for the same problem.

In this algorithm, $t_1$ is $O(n^3)$ and $t_p$ is $O(n)$ when $p$ is (the number of PEs = $O(n^2)$ + the number of ISs = $O(n)$ + 1). Therefore, $E(1,p)$ is $O(n^3) / [O(n) * (O(n^2) + O(n) + 1)] = O(n^3) / [O(n^3) + O(n^2) + O(n)] = 1$. The parallel algorithm is cost optimal provided that the sequential algorithm is time optimal.

6.4 Simulation Results for the MASC Floyd-Warshall Algorithm

This section discusses simulation results after running the MASC Floyd-Warshall algorithm on the MASC simulator on various inputs.

6.4.1 Problem Instances Used in the Simulator

The parallel performance is evaluated using scale-free graphs (R-MAT) that represents unstructured data [6]. The R-MAT (recursive matrix) graph model is a graph generator that can quickly generate realistic graph (the distribution of vertex degrees obeys an inverse power law) with a few parameters such as those represent computer or social networks that can be used for a forecasting or simulations such as predicting how a computer virus will spread.
Parameters $a$, $b$, $c$, and $d$ are used to generate an R-MAT graph. These parameters represent probabilities of an edge falling into four partitions in the R-MAT model as shown in Figure 6.5. R-MAT graphs used in this chapter are generated with the following parameters; $a=0.17$, $b=0.55$, $c=0.18$, and $d=0.10$ where $a + b + c + d = 1$ in order to avoid clusters of loop-to-self edges.

![Diagram of R-MAT model](image)

Figure 6.5: The R-MAT model shown with partitions $a$, $b$, $c$, and $d$.

An input R-MAT graph is created by recursively subdividing the adjacency matrix into four equal-sized partitions and distributing edges within these partitions with unequal probabilities, $a$, $b$, $c$, and $d$. First, the adjacency matrix starts with an empty matrix whose entries are 0’s. Next, edges are dropped one at a time into one of the partitions of the matrix. Until the partition is a 1-by-1 partition, the chosen partition is subdivided into four smaller partitions repeatedly. The 1-by-1 partition is the cell occupied by the edge (or will be added one into the cell entry). Finally, the input R-MAT graph is cleaned up by eliminating self-loop edges; set $d(v_i, v_i) = 0$ and $d(v_i, v_j) = \infty$ if there is no edge from $v_i$ to $v_j$ (the entry is 0 if $v_i \neq v_j$).
6.4.2 The Performance of the MASC Algorithm on the Simulator

The running time results of the MASC Floyd-Warshall algorithm are shown in Figure 6.6. The algorithm is run using many configurations (from 1 worker to up to m workers) for each of m-vertex R-MAT graph data inputs. The stripe size is determined by the number of workers and $|V| = n$ for a graph $G = (V, E)$, i.e., for a MASC($n^2$, m+1), the stripe size is $n/m$ where $n$ and $m$ are powers of 2.

<table>
<thead>
<tr>
<th>MASC(PEs,ISs)</th>
<th>Stripe Size</th>
<th>Vertices</th>
<th>ASC Runtime*</th>
<th>MASC Runtime</th>
<th>Speedup over an ASC model</th>
<th>Worker IS Utilizations (each)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(64,1+2)</td>
<td>4</td>
<td>8</td>
<td>6,336</td>
<td>4,213</td>
<td>1.5039</td>
<td>0.7709</td>
</tr>
<tr>
<td>(64,1+4)</td>
<td>2</td>
<td>8</td>
<td>6,336</td>
<td>2,629</td>
<td>2.4100</td>
<td>0.6329</td>
</tr>
<tr>
<td>(64,1+8)</td>
<td>1</td>
<td>8</td>
<td>6,336</td>
<td>1,837</td>
<td>3.4491</td>
<td>0.4747</td>
</tr>
<tr>
<td>(256,1+2)</td>
<td>8</td>
<td>16</td>
<td>25,344</td>
<td>14,757</td>
<td>1.7174</td>
<td>0.8696</td>
</tr>
<tr>
<td>(256,1+4)</td>
<td>4</td>
<td>16</td>
<td>25,344</td>
<td>8,421</td>
<td>3.0096</td>
<td>0.7714</td>
</tr>
<tr>
<td>(256,1+8)</td>
<td>2</td>
<td>16</td>
<td>25,344</td>
<td>5,253</td>
<td>4.8247</td>
<td>0.6335</td>
</tr>
<tr>
<td>(256,1+16)</td>
<td>1</td>
<td>16</td>
<td>25,344</td>
<td>3,669</td>
<td>6.9076</td>
<td>0.4753</td>
</tr>
<tr>
<td>(1024,1+2)</td>
<td>16</td>
<td>32</td>
<td>101,376</td>
<td>54,853</td>
<td>1.8481</td>
<td>0.9299</td>
</tr>
<tr>
<td>(1024,1+4)</td>
<td>8</td>
<td>32</td>
<td>101,376</td>
<td>29,509</td>
<td>3.4354</td>
<td>0.8697</td>
</tr>
<tr>
<td>(1024,1+8)</td>
<td>4</td>
<td>32</td>
<td>101,376</td>
<td>16,837</td>
<td>6.0210</td>
<td>0.7716</td>
</tr>
<tr>
<td>(1024,1+16)</td>
<td>2</td>
<td>32</td>
<td>101,376</td>
<td>10,501</td>
<td>9.6539</td>
<td>0.6338</td>
</tr>
<tr>
<td>(1024,1+32)</td>
<td>1</td>
<td>32</td>
<td>101,376</td>
<td>7,337</td>
<td>13.8171</td>
<td>0.4757</td>
</tr>
<tr>
<td>(4096,1+2)</td>
<td>32</td>
<td>64</td>
<td>405,504</td>
<td>211,077</td>
<td>1.9211</td>
<td>0.9636</td>
</tr>
<tr>
<td>(4096,1+4)</td>
<td>16</td>
<td>64</td>
<td>405,504</td>
<td>109,701</td>
<td>3.6964</td>
<td>0.9299</td>
</tr>
<tr>
<td>(4096,1+8)</td>
<td>8</td>
<td>64</td>
<td>405,504</td>
<td>59,013</td>
<td>6.8714</td>
<td>0.8698</td>
</tr>
<tr>
<td>(4096,1+16)</td>
<td>4</td>
<td>64</td>
<td>405,504</td>
<td>33,669</td>
<td>12.0438</td>
<td>0.7717</td>
</tr>
<tr>
<td>(4096,1+32)</td>
<td>2</td>
<td>64</td>
<td>405,504</td>
<td>20,997</td>
<td>19.3125</td>
<td>0.6340</td>
</tr>
<tr>
<td>(4096,1+64)</td>
<td>1</td>
<td>64</td>
<td>405,504</td>
<td>14,661</td>
<td>27.6587</td>
<td>0.4758</td>
</tr>
</tbody>
</table>

* Calculated from steps required to execute the algorithm by a comparable ASC

Figure 6.6: Results of the MASC implementation of the Floyd-Warshall algorithm.

An ASC runtime is calculated from the minimum number of steps required to execute the algorithm by a comparable ASC model. The cost of associative operations on a comparable ASC
model is equal to that of the MASC model. A MASC runtime is a combination of the runtime required by the manager instruction stream to execute its execution cycles and the runtime required by the concurrent executions of assigned tasks by worker instruction streams.

![Worker IS Utilization](image)

Figure 6.7: Worker utilization in the simulator. Data input sets vary from 64 PEs and 2 workers to 4096 PEs and 64 workers.

Worker utilization is a ratio between a worker instruction stream’s busy time and the total running time of the simulator. A high worker utilization number usually means worker instruction streams spent more time executed assigned tasks than waiting for the manager instruction stream to complete its execution cycles. Figure 6.7 shows that using more workers for a data input set lowering the worker utilization (and also lowering the total execution time). Let us consider one
data input set, 4,096 PEs. Among the data input of 4,096 PEs, the manager overhead is fixed at about 7,685 steps (calculated from 14,661 * (1 - 0.4758)). The manager overhead of the (4096, 1+2) configuration is about 4% of the total execution time, which is the lowest ratio. While that of the (4096, 1+64) configuration is about 53% of the total execution time, which is the highest ratio.

![The Floyd-Warshall All-pairs Shortest Path algorithm execution time (steps) on the MASC simulator](image)

Figure 6.8: The Floyd-Warshall algorithm execution time in steps by the MASC simulator on 8 to 64 vertex input R-MAT graphs.

As one can see in Figure 6.8, a bigger MASC model (more worker instruction streams) executes the Floyd-Warshall algorithm (for a given input R-MAT graph) faster than a smaller
MASC model does. Unfortunately, the maximum number of usable worker instruction streams is limited to \(|V|\). In this implementation of the algorithm, using more than \(|V|\) worker instruction streams will not reduce the execution time since the maximum worker instruction stream tasks available is \(|V|\) tasks. When the problem size gets twice as big, i.e., from a 8-vertex R-MAT graph to a 16-vertex R-MAT graph, the execution time of an ASC quadruple, while the execution of a MASC does not quadruple. When using the maximum number of worker instruction streams allowed, the execution of a MASC increases about double. From this observation, one can conclude that a MASC model scales better to this Floyd-Warshall algorithm than an ASC model does.

Figure 6.9: The speedup of the MASC model over an ASC model executing the Floyd-Warshall algorithm on 8 to 64 vertex input R-MAT graphs.
Another observation is the MASC model benefits from a larger problem-size input graph. In Figure 6.9, when a graph size is double, the maximum speedup of a MASC over ASC continues about double, i.e., from 3.45 to 6.91, from 6.91 to 13.82, and from 13.82 to 27.66. However, for a fixed size graph, while the speedup graph starts very close to the perfect speedup, the more worker instruction streams used, the faster the tail of the graph begins to fall off from the perfect speedup graph. Lower worker utilization numbers can be used to explain this situation.

6.4.3 The Performance Prediction of the MASC Algorithm on the Simulator

Let assumed an input graph $G = (V, E)$ and a MASC($|V|^2$, 1+ $m$) where $1 < m \leq |V|$ is used. Based on the number of steps required by the MASC simulator. The prediction of both ASC and MASC runtime can be calculated as follows since the capability of the MASC simulator is limited to a maximum graph size of 64 vertices.

An ASC Runtime

= the number of iterations * ((the number of columns * the number of steps required to execute each column) + (the number of rows * the number of steps required to execute each row))

= $|V| * (47|V| + 52|V|)$

= $99|V|^2$

= $O(|V|^2)$. 
A MASC($|V|^2$, 1+ m) Runtime

= the number of iterations * [the number of steps to execute the manager instruction stream’s execution cycle + (column stripe size * the number of steps required to execute each column) + the number of steps to execute the manager instruction stream’s execution cycle + (row stripe size * the number of steps required to execute each row) + overhead]

= $|V|$ * [62 + 47*column stripe size + 5 + 62 + 52*row stripe size + 5]

= $|V|$ * [134 + 99*column stripe size]

= $|V|$ * [134 + 99($|V|$/m)]

= O($|V|$).

Example

If $|V| = 2^{10} = 1,024$ vertices, a number of PEs required is $2^{20} = 1,048,576$ PEs.

An ASC runtime is $99*(1,024)^2$

= 103,809,024 steps required to complete the algorithm.

A MASC($|V|^2$, 1+m) runtime

= $1,024*233 = 238,592$ required to complete the same algorithm if $m = |V|$.

The speedup of the MASC over ASC on the same algorithm and input is $103,809,024/238,592 = 435.09$. 
We can conclude that the maximum speedup of a MASC over ASC continues to grow if the input graph getting bigger. If the input size is very large, the MASC model can achieve a near perfect speedup while maintains a respectable running time.

6.5 Conclusion

This chapter shows the ability of the MASC model with the manager-worker instruction stream paradigm to address a graph problem such as all-pairs shortest path problem using a static task assignment scheme. When processing large-scale instances using multiple workers, the MASC Floyd-Warshall algorithm shows strong scaling with constant time overhead on this unstructured massively multithreaded problem. A key to this success is the use of a static task assignment scheme with multiple instruction streams.
CHAPTER 7

A MASC Quickhull Algorithm to Identify Points of a Convex Hull

7.1 Introduction

This chapter describes an implementation of a MASC Quickhull algorithm to identify extreme points of a convex hull. This algorithm is implemented using a non-static task assignment scheme. Mapping of problem tasks to instruction streams are not predetermined and will be done during runtime. One task can only be assigned to an instruction stream at a time (assuming that only one manager exists). The benefit of using a non-static task assignment scheme is its flexibility. The scheme can be used by the MASC simulator to execute problems, which cannot use static task assignment scheme.

7.2 The Sequential Algorithm

By the definition, a convex hull of a set of points $S$ is the smallest convex set containing $S$ as shown in Figure 7.1. Each point in $S$ is either on the boundary of the convex hull set or in its interior. If $S$ consists of finite set of points in the plane, one can make an analogy as follows. A large rubber band is stretched to surround all points. If pins are inserted at each point of $S$, then when the rubber band is released, it will assume the shape of the convex hull.
There are two steps required to find the convex hull of a finite set.

- First, extreme points need to be identified. A point $p$ of a convex set $S$ is an extreme point if no two points $a, b \in S$ exist such that $p$ lies on the open line segment $ab$. Moreover, point $p$ fails to be an extreme point of a plane convex hull set $S$ only if it lies in some triangle whose vertices are in $S$ but itself is not a vertex of the triangle. Figure 7.2 shows point $p$ is not an extreme point because it lies inside triangle $(p_1, p_2, p_3)$. 

Figure 7.1: A set of points $S$ and its convex hull.

Figure 7.2: Point $p$ is not extreme because it lies inside triangle $(p_1, p_2, p_3)$. 
Second, extreme points are ordered so that they form the successive corners of a convex polygon when traversal in order.

The MASC algorithm presented in this chapter does not order extreme points found. It deals with the first step of the requirement to find a convex hull of a finite set of points on the plane. As is typical for convex hull algorithms, this algorithm assumes that no two points have the same x and y coordinates. However, it is easy to extend this algorithm so it will handle the requirement, as well.

Figure 7.3: Points w, r, and e subdivide the set $S(1)$ and eliminate all points in the yellow shaded triangle form further consideration.

The Quickhull algorithm gets its inspiration from a sorting algorithm called Quicksort algorithm. Similar to Quicksort algorithm, for each round of execution, Quickhull algorithm
subdivides the original problem into two subproblems, each of which can be solved independently and simultaneously using a parallel computer. It is easy to implement the Quickhull algorithm. Initially, the N-point set S is partitioned into two subsets, each of which will contain one of two polygonal chains whose concatenation gives the convex hull polygon. The initial partition is determined by the line passing through the two points \( w \) (a.k.a. west) and \( e \) (a.k.a. east) with the smallest and largest \( x \) coordinates using \( y \)-coordinates as tie breakers. Let \( S(1) \) be the subset of points on or above the line through \( w \) and \( e \). \( S(2) \) is symmetrically defined as the subset of points on or below the same line.

Then, successive steps perform on set \( S(1) \) and \( S(2) \) as follows. Let only consider steps for \( S(1) \). Steps for \( S(2) \) are similar to those for \( S(1) \). A point \( r \in S(1) \) is determined such that the triangle \( (wre) \) has the maximum area among all triangles \( \{(wpe): p \in S(1)\} \) (using \( x \)-coordinates as tie breakers, if there are more than one maximum area triangles, the smallest \( x \) is chosen). Besides points \( w \) and \( e \), point \( r \) is guaranteed to be an extreme point and belonged to a convex hull. If one traces a line parallel to segment \( we \) and passing through \( r \), there is no points of \( S \) above the line. There may be some points of \( S \) on the line but \( r \) is the left most of them. Thus, \( r \) is cannot be expressed as the convex combination of two other points of \( S \) [37].

Next, two lines are constructed, line \( L1 \), which passes through \( w \) and \( r \), and line \( L2 \), which passes through \( e \) and \( r \). Each point of \( S(1) \) is tested with these lines. A point that lies inside the triangle \( (wre) \) will be eliminated from further consideration. Points of \( S(1) \) not to the right of \( L2 \) and lying on the left of \( L1 \) form a set \( S(1,1) \). Similarly, points not to the left of \( L1 \) and lying on the right of \( L2 \) form a set \( S(1,2) \) as shown in Figure 7.3. Both \( S(1,1) \) and \( S(1,2) \) are passed to the subsequent level of recursion.
The Main Algorithm to Identify Extreme Points of a Convex Hull

1. Set CH as an empty set.

2. Find two extreme points, \( w \) and \( e \), based on their \( x \)-coordinates. Use \( y \)-coordinates to as tie-breakers.

3. \( CH = \{ w, e \} \).

4. Let \( S(1) \) be the set of points of \( S \) on or above a line passing through \( w \) and \( e \).

5. Let \( S(2) \) be the set of points of \( S \) on or below a line passing through \( w \) and \( e \).

6. Call an algorithm to identify extreme points of the upper convex hull with input \( S(1) \), \( w \), and \( e \).

7. Call an algorithm to identify extreme points of the lower convex hull with input \( S(2) \), \( w \), and \( e \).

Let determine the running time of the main algorithm above. Let’s assume \(|S|\) is \( O(n) \). Lines 1 and 3 take \( O(1) \). Lines 2, 4, and 5 take \( O(n) \). Lines 6 and 7 are calls to two similar algorithms. Their running times will be given below.

Algorithm to Identify Extreme Points of the Upper Hull

**Input:** A set of points \( S(1) \) and points \( w \) and \( e \).

**Output:** A set of extreme points for the upper hull, \( CH \).
1. IF S(1) is \{w, e\}
   a. THEN set CH = \{w, e\} U CH.
   b. ELSE
      i. Find \( r \) such that the triangle \((wre)\) has the largest area among all triangles \(\{(wpe): p \in S(1)\}\) and set CH = \{\( r \)\} U CH.
      ii. Set S(1,1) as the set of points of S on or to the left of a line passing through \( w \) and \( r \).
      iii. Set S(1,2) as the set of points of S on or to the right of a line passing through \( r \) and \( e \).
      iv. Recursively call the algorithm on the input S(1,1) and points \( w \) and \( r \).
      v. Recursively call the algorithm on the input S(1,2) and points \( r \) and \( e \).

The running time the above algorithm is largely determined by lines 1, 1a, 1bi, 1bii, 1biii, and two recursive calls to itself. The running time of lines 1 and 1a is \( O(1) \) and that of lines 1bi to 1biii is \( O(n) \) if \(|S|\) is \( O(n) \). In the worst case, there are \( O(n) \) recursive calls in total. That gives the total running time of \( O(n^2) \). In an average case, even there are splits of extreme points to the left and the right as a ratio of 99-to-1, the depth of the recursion tree is \( O(\lg n) \) \[10\]. Thus, it yields the total running time of \( O(n \lg n) \).
When combine the running time of the above two algorithms, the worst case running time of a sequential algorithm to identify extreme points of a convex hull is \( O(n^2) \), which is similar to the that of the Quicksort algorithm. An average case running time remains \( O(n \log n) \).

### 7.3 A MASC Quickhull Algorithm to Identify Extreme Points of a Convex Hull

The ASC Quickhull algorithm purposed by Atwah [3] is modified to be used in the MASC model using the manager-worker paradigm. This MASC algorithm does not have a limitation that requires a certain number of instruction streams to be used. Any number of instruction streams can be used regardless to the number of convex hull points.

#### 7.3.1 Expressing Parallelism for the Algorithm

Let consider an input \( S \), a finite set of points in the plane. Each point is mapped to a PE, regardless to the order of points as shown in Figure 7.4. Each PE contains coordinates of the point along with other data needed for the parallel algorithm.

![Figure 7.4: Points of set S in the plane are mapped to PEs (one point per PE).](image-url)
At first, all PEs work in parallel and are controlled by one instruction stream. After points \( w \) and \( e \) are found, two new tasks are created by partitioning of PEs (one above the line passing through \( w \) and \( e \) and another below the line) as shown in Figure 7.5.

![Diagram of PEs and partitioning](image)

Figure 7.5: After two extreme points, \( w \) and \( e \), were found, PEs are partitioned into two partitions, one (a) with points lying above the line passing through \( w \) and \( e \) and another (b) with points lying below the line.

After one task has been finished, two more tasks can be created by partitioning PEs further if there are points left to be considered. These new tasks can be executed independently and simultaneously using multiple instruction streams.
7.3.2 An Implementation of the Algorithm Using the MASC Model

Similar to the algorithm given in the last chapter, the MASC Quickhull algorithm to identify extreme points of a convex hull consists of an algorithm for the manager and an algorithm for workers. The first algorithm described is for workers. The manager’s algorithm will be described next.

The data record of each PE contains the following variables.

- Variables to store coordinates \( x \) and \( y \) of the input point; points \( p \), \( q \), \( w \) (west), \( e \) (east); and three temporary points denoted as \( X \), \( Y \), \( X_p \), \( Y_p \), \( X_q \), \( Y_q \), \( X_e \), \( Y_e \), \( X_w \), \( Y_w \), \( X_tp \), \( Y_tp \), \( X_p2 \), \( Y_p2 \), \( X_q2 \), and \( Y_q2 \), respectively.

- A variable to store an area of a triangle denoted as \( area \).

- Four Boolean variables denoted as \( in_{\text{hull}} \), \( candidate \), \( is_{\text{above}} \), and \( is_{\text{west}} \), respectively.

- Three variables to store data to represent a line equation, in the form of \( Ax + By = C \), denoted as \( A \), \( B \), and \( C \), respectively.

- Four more variables to store data to compute an area of a triangle using Heron’s formula denoted as \( s \), \( a \), \( b \), and \( c \), respectively.

An Algorithm for Workers of MASC(n,m+1) to Partition Points into Two Halves

1. The worker changes its state from Ready to Busy.
2. /* Partition points into two halves, the upper and the lower. */

3. /* Find the west most point. */
   a. Broadcast to all PES to set their mask registers to true (1).
   b. Perform a MIN global reduction on x-coordinates of PEs. Get $X_{min}$.
   c. /* Need to pinpoint only one west most point. */
   d. Perform an associative search on PEs that have their x-coordinates equal to $X_{min}$.
   e. /* After an associative search, all non-responders PEs are deactivated. */
   f. Perform a MIN global reduction on y-coordinates of PEs. Get $Y_{min}$.
   g. Perform an associative search on active PEs that have their y-coordinates equal to $Y_{min}$.
   h. /* Get the west most point */
   i. Perform a Pick One operation on the only PE left since no two PEs has the same x and y coordinates.
   j. Add this PE as an extreme point (it is in a convex hull) and mark this PE as no longer to be considered.
   k. Reactivate all PEs (except those marked as no longer to be considered).
   l. Broadcast this west most point to PEs.

4. /* Find the east most point */
a. /* These steps are similar to finding the west most point. Use MAX global reductions instead of MIN reductions to get Xmax and Ymax. */

5. /* At this point, the west most and the east most are found and broadcasted to all other PEs. */

6. Each PE calculates a line equation in the form of \( y = (C - Ax) / B \) passing through the west most and the east most points.

7. Each PE checks if its point is above or below the line and set its variables accordingly.

8. A special case that a point is on the line is also handled. If no point is below (or above) the line, whatever points on the line are considered extreme points (they are in a convex hull) and are marked as no longer to be considered. Otherwise, consider them as below the line points.

9. The execution of this task is finished. Continue to 10.

10. When the worker is done with the execution of the assigned task, the worker broadcasts to its PEs to switch the instruction stream they are listening to the manager, whose ID is 0.

11. PEs copy 0 to their instruction stream selector register.

12. The worker changes its state from Busy to Finish.

Each line of the above algorithm takes constant time to be executed. After the algorithm has been finished, all PEs are partitioned into two halves by a line passing through \( w \) and \( e \) yielding the upper and lower halves. The running time of this worker task is \( O(1) \).
Algorithm for worker instruction streams of MASC(n,m+1) to find extreme points in the upper half

1. The worker changes its state from Ready to Busy.

2. This is a case that a worker may need to find (up to) k extreme points before marked this task as finished.
   a. If there is more extreme point to be found then
      i. /* Find extreme points in the upper half */
      ii. Broadcast to all PEs to set their mask registers to true (1).
      iii. Broadcast to all PEs to compute an area of the triangle (prq) assumed each PE has a point r, p, and q are the two given extreme points.
      iv. /* Area of a triangle is computed using Heron’s formula. */
      v. /* Area = SQRT ( s*(s-a)*(s-b)*(s-c) ) */
      vi. /* The distance between two points, (x1,y1) and (x2,y2), in 2D plane is Distance = SQRT( pow((x1-x2),2) + pow((y1-y2),2) ). */
      vii. Perform a MAX global reduction on areas of triangles. Get max_area.
      viii. /* Need to find a tie breaker among whose with the largest area, if any */
      ix. Perform an associative search on active PEs that have their areas equal to max_area.
x. Perform a MIN global reduction on areas of $x$-coordinates. Get $X_{\text{min}}$.

xi. Perform an associative search on active PEs that have their $x$-coordinates equal to $X_{\text{min}}$.

xii. /* Found an extreme point */

xiii. Perform a Pick-One operation. Add this PE as an extreme point (it is in a convex hull) and mark this PE as no longer to be considered.

xiv. /* Eliminate points that cannot be extreme points */

xv. Reactivate all PEs (except those marked as no longer to be considered).

xvi. Broadcast this new found extreme point to PEs called $p'$.

xvii. Each PE calculates two line equations in the form of $y = (C - Ax) / B$ passing through $p$ and $p'$ and $q$ and $p'$.

xviii. Each PE checks if its points is inside the triangle ($pp'q$). If yes, mark this PE as no longer to be considered.

xix. A special case that a point is on a line (either $pp'$ or $qp'$) is also handled. If no point is above the line, whatever points on the line are considered extreme points (they are in a convex hull) and are marked as no longer to be considered. Otherwise, mark PEs as no longer to be considered.

xx. /* Go back to find another extreme point, if needed */

3. If there is no more extreme point to be found or at least $k$ extreme points have been found, terminate this task and continue at 4. Otherwise, return to 2a.
4. When the worker is done with the execution of the assigned task, the worker broadcasts to its
   PEs to switch the instruction stream they are listening to the manager, whose ID is 0.
5. PEs copy 0 to their instruction stream selector register.
6. The worker changes its state from Busy to Finish.

The above algorithm for workers of MASC(n,m+1) finds extreme points in the upper half
of a convex hull. An algorithm to find extreme points in the lower half of the convex hull can be
obtained similarly. Each line of the above algorithm takes constant time to be executed. Each
extreme point can be found in O(1) time. If up to k extreme points, need to be found for a worker
task, the worst case running time of this algorithm is O(k).

An Algorithm for the Manager of MASC(n,m+1)

1. /* FINISHED Phase */
   a. Perform a global reduction on workers to see if at least one of them is in Finish state.
   b. If that is not true, go to FORK Phase. Otherwise, continue to the step.
   c. Perform a Pick-One operation on workers that are in Finish state and get this_worker.
   d. Reduce this_worker ‘s task ID back to the manager.
   e. Mark this task as finished.
   f. Get its children tasks, if any. Add them into a pool.
   g. Broadcast to this_worker to change its state from Finish to Ready.
   h. Go back to FINISHED Phase (1a).
2. /* FORK Phase */

   a. If there is no fork task in the fork pool, go to ASSIGN Phase. Otherwise, continue to the next step.

   b. Remove the first fork task from the fork pool.

   c. Partition PEs from the fork task into two partitions based on their variables. This creates two new children tasks from the fork task. These two tasks are added into the task pool.

   d. Go back to FORK Phase (2a).

3. /* ASSIGN Phase */

   a. If there is no task in the task pool, go to JOIN Phase. Otherwise, continue to the next step.

   b. Perform a global reduction on workers to see if at least one of them is in Ready state. If no worker is in the Ready state, go to JOIN Phase. Otherwise, continue to the next step.

   c. Remove the first task from the task pool and get this_task.

   d. Check PEs belonged to this_task, if there is at least one candidate PE (a PE that may be an extreme point), continue to the next line. Otherwise, go back to ASSIGN Phase (3a).

   e. Perform a Pick-One operation on workers that are in Ready state and get this_worker.
f. Broadcast to all PEs belonged to this_task and are candidates to switch the instruction stream they are listening to this_worker.

g. PEs copy this_worker to their instruction stream selector register.

h. Broadcast to this_worker instruction stream to work on this_task.

i. Go back to ASSIGN Phase (3a).

4. /* Join Phase */

    a. If there is no join task in the join pool, go to TERMINATION Phase. Otherwise, continue to the next step.

    b. Remove the first join task from the join pool.

    c. If this join task has not been completed yet, continue to the next step. Otherwise go back to JOIN phase (4a).

    d. Get its parents and its children tasks.

    e. If all parents have been finished, current task ID of PEs will be updated and its children tasks will be added into a pool.

    f. Mark this join task as completed.

    g. Go back to JOIN phase (4a).

5. /* TERMINATION Phase */

    a. If all workers are in Ready state and all pools are empty, then terminate the program.

    b. Otherwise, go back to FINISHED Phase (line 1a).
The total running time of the above manager algorithm can be calculated as follows.

- First, the execution time of each round of FINISHED Phase is $O(1)$. The maximum number of rounds in each FINISHED Phase is equal to the number of uncollected finished tasks, which equals to the number of worker ISs in the Finish state. For a MASC(n, m+1), the worst case is $O(m)$. Thus, the worst case of FINISHED Phase is $O(m)$.

- Similarly, the execution time of each round of FORK Phase is $O(1)$. The maximum number of rounds in each FINISHED Phase is equal to the number of fork tasks in the fork pool. Each finished task by a worker IS can produce at most one fork task. The maximum number of fork tasks in the fork pool is $O(m)$. Thus, the worst case of FORK Phase is $O(m)$.

- Next, the running time of ASSIGN Phase depends on the minimum number of tasks in the task pool and Ready workers. The worst case running time of ASSIGN Phase can be assumed as $O(m)$.

- Again, each finished task by a worker can produce at most one join task. The running time of this JOIN Phase is similar to that of FORK Phase, which is $O(m)$.

- Last, the running time of TERMINATION Phase is always $O(1)$.

In order to determine the manager overhead, the number of workers has to be factored in. In non-static task assignment problems like this algorithm, the number of workers is usually
(small or large) constant. The number of workers is not related to the number of PEs. The worst case running time of the manager’s algorithm can be considered as constant time or \(O(1)\) if the number of workers is \(O(1)\). In chapter 8, (the future work section), a future work that might reduce this total worst case running time is suggested.

When combine all execution phases of the manager, the total worst case running time is \(O(m)\) for MASC\((n, m+1)\). For an average case when \(O(\log n)\) out of \(O(n)\) points belong to a convex hull, the total average case running time for MASC\((n, m+1)\) is \(O[(\log \log n) (\log n)/m]\), where \(m \leq O(\log n)\).

Again, the efficiency of the MASC algorithm can be computed from the efficiency formula \(E(1,p) = t_1/pt_p\), where \(t_1\) is the worst case running time of a sequential algorithm, \(t_p\) is the worst case running time of the \(p\)-processor parallel algorithm for the same problem.

For an average case, in this algorithm, \(t_1\) is \(O(n \log n)\) and \(t_p\) is \(O[(\log \log n) (\log n)/m]\) when \(p\) is (the number of PEs = \(O(n) + \) the number of ISs = \(m + 1\)). In this case, \(m = O(1)\) will be used. Therefore, \(E(1,p) = O(n \log n) / [O[(\log \log n) (\log n)] \times (O(n) + O(1) + 1)] = O(n \log n) / [O(\log \log n)(\log n)(n) + O[(\log \log n) (\log n)] < 1. The parallel algorithm is not cost optimal provided the number of workers used in the algorithm is constant.

### 7.4 Simulation Results of the MASC Algorithm

This section discusses simulation results after running the MASC Quickhull algorithm to identify extreme points on the MASC simulator on various inputs.
7.4.1 Problem Instances Used in the Simulator

The parallel algorithm is tested using two sets of points. The first set is generated randomly to check for the correctness of the MASC algorithm. The second set of input, presented here to show the parallel performance, is points generated from a parabolic generator. All points in a parabolic are basically in a convex hull set. A recursive tree from these inputs will form balanced recursive trees. It is possible to use the same set of input in many runs of the simulator in order to measure the scalability of the MASC model by using as many as instruction streams as possible.

The results from these sets of input should be considered as the best running time for a given size input or an average case of the input set of \( N \) points if \( \log N \) points are in the parabolic. Using random points as the input may lead to a situation that limits the number of usable instruction streams. Consider the recursive tree in Figure 7.6. No more than two workers can be used simultaneously regardless of the size of the convex hull set.

Figure 7.6: No more than two workers can be used simultaneously for this recursive tree.
### 7.4.2 The Performance of the MASC Algorithm on the Simulator

The results shown in Figure 7.7 are taken from the best runs of the simulator on various extreme point and instruction stream sizes. The total running time of the algorithm is a combination of the thread synchronization overhead time (performed by the manager) and the execution time of concurrent tasks (performed by workers).

<table>
<thead>
<tr>
<th>MASC(PEs, ISs)</th>
<th>Number of Iteration</th>
<th>Points in Hull</th>
<th>ASC Runtime*</th>
<th>MASC Runtime</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>(256, 1+2)</td>
<td>16</td>
<td>256</td>
<td>47,992</td>
<td>28,341</td>
<td>1.6934</td>
</tr>
<tr>
<td>(512, 1+2)</td>
<td>16</td>
<td>512</td>
<td>96,120</td>
<td>53,758</td>
<td>1.7880</td>
</tr>
<tr>
<td>(1024, 1+2)</td>
<td>32</td>
<td>1024</td>
<td>192,376</td>
<td>103,695</td>
<td>1.8552</td>
</tr>
<tr>
<td>(2048, 1+2)</td>
<td>64</td>
<td>2048</td>
<td>378,684</td>
<td>201,436</td>
<td>1.8799</td>
</tr>
<tr>
<td>(256, 1+4)</td>
<td>8</td>
<td>256</td>
<td>47,992</td>
<td>18,010</td>
<td>2.6647</td>
</tr>
<tr>
<td>(512, 1+4)</td>
<td>16</td>
<td>512</td>
<td>96,120</td>
<td>31,402</td>
<td>3.0610</td>
</tr>
<tr>
<td>(1024, 1+4)</td>
<td>32</td>
<td>1024</td>
<td>192,376</td>
<td>58,721</td>
<td>3.2761</td>
</tr>
<tr>
<td>(2048, 1+4)</td>
<td>64</td>
<td>2048</td>
<td>378,684</td>
<td>109,198</td>
<td>3.4679</td>
</tr>
<tr>
<td>(256, 1+8)</td>
<td>8</td>
<td>256</td>
<td>47,992</td>
<td>13,825</td>
<td>3.4714</td>
</tr>
<tr>
<td>(512, 1+8)</td>
<td>8</td>
<td>512</td>
<td>96,120</td>
<td>22,350</td>
<td>4.3007</td>
</tr>
<tr>
<td>(1024, 1+8)</td>
<td>16</td>
<td>1024</td>
<td>192,376</td>
<td>37,094</td>
<td>5.1862</td>
</tr>
<tr>
<td>(2048, 1+8)</td>
<td>32</td>
<td>2048</td>
<td>378,684</td>
<td>65,836</td>
<td>5.7519</td>
</tr>
<tr>
<td>(256, 1+16)</td>
<td>8</td>
<td>256</td>
<td>47,992</td>
<td>12,529</td>
<td>3.8305</td>
</tr>
<tr>
<td>(512, 1+16)</td>
<td>8</td>
<td>512</td>
<td>96,120</td>
<td>18,948</td>
<td>5.0728</td>
</tr>
<tr>
<td>(1024, 1+16)</td>
<td>16</td>
<td>1024</td>
<td>192,376</td>
<td>28,832</td>
<td>6.6723</td>
</tr>
<tr>
<td>(2048, 1+16)</td>
<td>16</td>
<td>2048</td>
<td>378,684</td>
<td>46,086</td>
<td>8.2169</td>
</tr>
<tr>
<td>(256, 1+32)</td>
<td>8</td>
<td>256</td>
<td>47,992</td>
<td>12,529</td>
<td>3.8305</td>
</tr>
<tr>
<td>(512, 1+32)</td>
<td>8</td>
<td>512</td>
<td>96,120</td>
<td>18,705</td>
<td>5.1387</td>
</tr>
<tr>
<td>(1024, 1+32)</td>
<td>16</td>
<td>1024</td>
<td>192,376</td>
<td>27,050</td>
<td>7.1119</td>
</tr>
<tr>
<td>(2048, 1+32)</td>
<td>16</td>
<td>2048</td>
<td>378,684</td>
<td>40,092</td>
<td>9.4454</td>
</tr>
</tbody>
</table>

* on a comparable ASC model

Figure 7.7. Results of the MASC implementation of the MASC algorithm.
If the number of instruction streams is fixed, when the extreme point set is getting bigger, the total running time and speedup (when compared with that of the ASC) increase. The reason is that, for this problem with its parabolic input data sets, the number of concurrent tasks increases when the number of extreme points increases. Having more concurrent tasks means more workers can be used at the same time to work on the algorithm while maintaining a reasonable thread overhead to the total execution time ratio.

If the size of an extreme point set is fixed, using more workers reduces the total running time (up to a point until no more workers are needed). Let consider the case of the extreme point set of size 256. The speedup gain from using 2 workers to 4 workers is noticeable, i.e., from 1.69 to 2.66. The gaining from using 8 workers to 16 workers to the same the extreme point set is barely noticeable, i.e., from 3.47 to 3.83. Without a large enough extreme point set, using a smaller number of workers may suffice.

There is a reason why adding more instruction streams does not always reduce the total running time. For an example, for the extreme point set of size 256, using 16 or 32 workers to execute the algorithm gives the same total running time of 12,529 steps because there are not enough available tasks to be assigned to the all workers. This fact can be verified by checking their utilizations. Adding more workers to the system does not improve the total running time, in this case. Sometimes, even there are some available tasks to be assigned to the extra workers, the speedup gain or the reduction of the total running time may not be that significant.

Let us look at the case of the extreme point set size of 2048. Speedup gains from using more workers are quite noticeable, i.e., from 1.88 to 3.47, to 5.75, to 8.22, and to 9.45. As one can see from Figure 7.8, the total running time of a given extreme point size continues to decrease as
the number of workers increase (up to a point as mentioned previously). For a given number of workers, increasing the size of the extreme point set also increases the total running time.

Figure 7.8: The MASC algorithm execution time in steps by the MASC simulator on 256 to 2048 convex hull points and 2 to 32 worker instruction streams.
Figure 7.9: The speedup of the MASC model over an ASC model executing the algorithm on 256 to 2048 convex hull points and 2 to 32 worker instruction streams.

Similarly, as shown in Figure 7.9, the larger the extreme point set is, the higher the speedup the system can achieve up to a near perfect speedup (for the case of controlled sets of input). An uncontrolled set of input may give a different graph (either staying flat or gaining very slow). To achieve a near perfect speedup, the extreme point set must be large enough and the number of workers used is small, i.e., using 2 workers for the extreme point set of 4000 or more, based on the graph, may yield the speedup at least 1.9 when 2.0 is the perfect speedup.
In every data set used in the simulator, they all show similar graph as that in Figure 7.10. Worker utilizations of workers are not equal. Some workers are busier than the other. This is because the manager uses non-static task assignment. Tasks are assigned to workers one task to one worker at a time. An idle worker with the largest ID will be picked first by the manager to be assigned a task and this is shown by its highest utilization than the rest of workers. A larger ID worker usually has higher worker utilization than that of a smaller ID worker. The amount of the manager overhead cannot simply be determined by worker utilizations alone in this non-static task assignment scheme.
7.4.3 Speedup of the MASC Algorithm and Amdahl’s Law

Amdahl’s Law (mentioned in chapter 2) states that the speedup of a program using multiple processors in parallel computing is limited by the time needed for the sequential fraction of the program. In the MASC simulator, the execution time of the ASSIGN Phase by the manager can be considered as the sequential part of the program. Ready workers line up waiting for the manager to assign tasks to them, one task per worker. In this non-static task assignment problem, one task can be assigned to a worker at a time. To simplify the process, let consider the total execution time of a task is the time from when the manager assigns that task until the task has been finished by a worker. In this simulator, each task requires 48 steps for the manager to assign the task at the worst case. If a worker needs to find only one extreme point, the worst case execution time is 155 steps. The total execution time of this task is 48+155 = 203 steps (with 23.65% as the sequential fraction). The maximum achievable speedup for this algorithm is less than 1/0.24 = 4.17 regardless of how big the extreme point set is or how many workers are used.

There is a need to increase the speedup upper bound by decreasing the sequential fraction. The ASSIGN Phase of the manager is difficult to be optimized further. The easier way is to increase the parallel execution time by assigning a task to a worker to find up to k extreme points before considered the task as finished. In this case, the new total execution time of a task is now the time from the manager assigns that task until the task found up to k extreme points has been finished by a worker. That brings the worst case total time to 48 + (186k +26). If k is 8, then the total time is 48 + (186x8 + 26) = 1562 steps (with 3.07% as the sequential fraction). The maximum achievable speedup for this algorithm rises to 1/0.03 = 33.33 regardless of how big the extreme point set is or how many workers are used.
Figures 7.11 to 7.14 show execution times of the algorithm on various sizes of extreme point sets and the number of extreme points per task needed to be found (noted as iteration points in graphs).

Figure 7.11: The MASC execution times in steps to identify 256 convex hull points on various iteration numbers.
Figure 7.12: The case of 512 convex hull points on various iteration numbers.

Figure 7.13: The case of 1024 convex hull points on various iteration numbers.
Figure 7.14: The case of 2048 convex hull points on various iteration numbers.

From the above Figures 7.11-7.14, increasing iteration numbers does not always give the best running time. Bar graphs in those Figures give a letter U-like shape. Each starts at a high number of steps, drops lower until hitting the lowest number of steps, and then increases until staying flat. The lowest point in each graph strikes the balance between a reasonable sequential overhead fraction and the total number of tasks available in the system.
7.5 Conclusion

In this chapter, we have presented an implementation of a non-static task assignment problem “identifying extreme points of a convex hull set using the Quickhull technique. When processing large-scale instances using multiple workers, for an average case, the MASC Quickhull algorithm shows good scaling with low overhead. The model using the manager-worker paradigm is shown to be a flexible model. Even though one task can be assigned to a worker at a time, with a careful planning regarding to the task size, the sequential fraction of the program according to the Amdahl’s law can be kept low enough to not hinder the speedup gaining from using multiple instruction streams.
CHAPTER 8

Summary and Future Work

8.1 Summary

The MASC (Multiple ASsociative Computing) parallel model is a multi-SIMD model that uses control parallelism to coordinate the interactions of data parallel threads and supports “associative SIMD” (or ASC) execution of each of its threads. The ASC model is an enhanced SIMD that is more efficient and easier to program than SIMD. ASC is basically a SIMD parallel computer that has been enhanced (in hardware) so it can support several reductions and operations in constant time. These constant time operations not only simplify the programming and the process of evaluating the complexity of algorithms but are extremely useful in parallel database operations and for applications such as air traffic control (which involves large dynamic databases). Each of the concurrent ASC executions of a task is performed by one of the MASC instruction streams and the processors currently listening to this instruction stream (IS). In addition, the predictability of SIMD computers, which allows the worst case time to be calculated very accurately, is also an important feature of this model and is very important for real-time applications with critical deadlines. In fact, the ASC model was motivated by the STARAN associative SIMD parallel computer, which was designed by Kenneth Batcher and built by Goodyear Aerospace in the early 1970’s for the air traffic control problem. A second generation version of the STARAN (i.e., the ASPRO) was used extensively by the Navy for an air defense system type application.
The original definition of MASC in ACM Communications in 1992 and subsequent publications provided a detailed specification of all aspects of the model other than about how the instruction streams and the communications between them are supported and controlled. This research provides a manager-worker version of the model that provides detailed information about how these features can be supported. In addition, the techniques used in creating this version of the MASC model are extremely efficient and preserves the desirable properties of the ASC model, including predictability. In a sense, this implementation provided a completion of the MASC definition by showing an example of how the definition could be satisfied and producing a showcase example of a completion that can be used by other researchers in this area and is very efficient and satisfies all of the desired properties of ASC.

In addition to providing a theoretical completion of MASC model that is efficient and preserves desirable properties of ASC, a simulator for this manager-worker version of MASC is built to demonstrate the performance of MASC on various algorithms. Additionally, this simulator is able to provide the exact number of cycles the model requires to execute a program on a given data set. Finally, this simulator is actually a software prototype for the manager-worker version with sufficient software details to allow a computer engineer to convert this software prototype into a hardware prototype of the manager-worker version of MASC.

The software simulator is used to illustrate the performance of MASC on two MASC algorithms, namely the Floyd-Warshall all-pairs shortest path algorithm which used a static task assignment scheme that allows concurrent ASC parallel executions of the various tasks. Additionally, the MASC simulator is used to evaluate the execution of the MASC Quickhull algorithm and uses a non-static task assignment scheme. The simulator allows the user to evaluate the efficiency of MASC on executing the algorithm on data of varying sizes and with a
various number of instruction steams. Additionally, the user can determine what number of instruction streams are most efficient for a given data size. The executions of the algorithms by various number of instruction streams provides a lot of information, which provide very instructive graphs. In general, extremely large data sizes are needed to keep a relatively large number of instruction streams busy.

8.2 Future Research: Using Hierarchical Manager Instruction Streams

Figure 8.1: A MASC model shown with hierarchical manager instruction streams.
A technique presented in this section may not give any improvement on the total running time over that of an original MASC model for problems that can be applied a static task assignment technique. The improvement will be noticeable if the model with a large number of workers executing problems consisting of many concurrent tasks when only a non-static task assignment technique can be used. The technique will reduce the total time of thread synchronization overhead by a certain factor. The execution time for a task by a worker remains the same.

The technique used combines the ideas of using hierarchical manager instruction streams and an idea called work (or task) stealing frequency used in MIMD model to reduce the total thread overhead. The new MASC model used consists of hierarchical managers as shown in Figure 39. The manager (IS 0) at level 0 is considered as a lower leveled manager than those (ISs 1 and 2) at level 1. A manager (usually the highest leveled) that connects to workers directly maintains its own private task pool. A manager that does not connect to workers does not have a task pool. Instruction stream broadcast and reduction networks are used to connect instruction streams together. Managers are required to work their execution phases in lock step to one another (the root manager controls the synchronization).

Another idea used is task stealing. Task or work stealing is a common technique used for implementing load balancing, whose concept has been long studied among MIMD parallel computing crowd. Work stealing technique is used in a runtime scheduler to schedule fine-grained parallel tasks where one thread steals work from another thread. Normally, semantics of work stealing guarantee that each task is eventually executed once. Relaxed semantics allow repeated execution if the application can tolerate it. There has been many research done in the area of work-stealing such as these works by [33][52][53].
The term task stealing used in this section is not actually a stealing of a physical task. A stolen task is simply outsourced from one manager to a worker of another manager. The stolen task will be marked as assigned (to an outsider worker). When the execution of the stolen task is done and after PEs from the stolen task returned to the highest level owner manager, the lower leveled manager will broadcast to managers that the stolen task is finished. The highest leveled owner manager, then, continues as if the stolen task were executed by its worker. The idea will be implemented as an additional phase called STEAL Phase. It has two parts, return stolen tasks and steal tasks. The additional phase can be added into the original execution phases of a manager as FINISHED, FORK, ASSIGN, JOIN, STEAL, and TERMINATION.

Under a normal circumstance, a lower leveled manager instruction stream is allowed to steal a task from its higher leveled manager instruction stream and give the stolen task to another of its higher leveled manager instruction stream if both following two conditions are true. The first condition is all workers of a highest leveled manager, whose task will be stolen, are busy executing tasks and the manager’s work pool must not empty. The second condition is at least one worker of a highest leveled manager, who the stolen task will be given to, is in the ready state and the manager’s work pool must empty. If a depth-restricted policy is implemented, only managers in certain levels can steal tasks.

Algorithm Steal Phase (Return Stolen Tasks)

1. Each lowest leveled manager, Thief_manager, that is allowed to steal a task performs the following instructions.

2. Repeat steps 3 to 5 until this phase is indicated as terminated.
3. Thief_manager broadcasts to its higher managers and search for a highest leveled manager that received a stolen task and its worker had finished that stolen task. If there is a responder manager, it gets From_manager. From_manager switches PEs from the stolen task to Thief_manager. If there is no responder, this phase is terminated.

4. Thief_manager broadcasts to its higher managers and search for the owner of the stolen task. A responder manager is called To_manager.

5. Return a stolen task: Thief_manager switches PEs from the returned stolen task to To_manager and broadcasts to To_manager that its stolen task have been returned and finished. To_manager marks this stolen task as finished.

6. Go back to step 2.

Let's assumed that a MASC model with $m=qk$ workers (q highest leveled managers with k workers each) and a depth restriction of $r$ is used. Note that a depth restriction of $r$ means managers at level $r$ will steal tasks. Moreover, $r$ cannot be the highest level. Lines 3 to 5 of the above algorithm take $O(1)$. They will be executed up to $O(m/(r+1))$ times. The total running time of the algorithm is $O(m/(r+1))$.

**Algorithm Steal Phase (Steal Tasks)**

1. Each lowest leveled manager, Thief_manager, that is allowed to steal a task performs the following steps.

2. Repeat steps 3 to 5 until this phase is indicated as terminated.
3. Broadcast to its higher managers and search for a highest leveled manager that all of its workers are busy and its task pool is not empty. If there is a responder manager, continue to the next step and gets From_manager. Otherwise, cannot steal a task. This phase is terminated.

4. Broadcast to its higher managers and search for a highest leveled manager that at least one of its workers is ready and its task pool is empty. If there is a responder manager, continue to the next step and gets To_manager. Otherwise, cannot steal a task. This phase is terminated.

5. Steal a task: broadcast to From_manager and ask for a task. From_manager tells its PEs to switch to Thief_manager and marks the stolen task as being assigned (to an outsider worker). The Thief_manager switches PEs to To_manager and broadcasts to To_manager to add this stolen task to its task pool. This stolen task is assigned to one of To_manager’s ready workers.

6. Go back to step 2.

The total running time of the above algorithm is similar to that of the previous algorithm (Return Stolen Tasks), which is \( O(m/(r+1)) \).

Note that, if there is no depth restriction to steal a task, only the root is the one who steals tasks. The advantage is the root manager can steal a task from any worker and give the stolen task to any highest leveled manager. The disadvantage is only one task is allowed to be stolen at a time and the process of steal has to goes through more levels. If the depth of \( k \) is allowed to steal tasks, managers at level \( k \) are allowed to steal tasks. The advantage is there can be up to \( 2^k \) stolen
tasks at a time. The disadvantage is these managers can only steal tasks from and give stolen tasks to managers that are their descendants.

In conclusion, this technique has two obvious benefits. The first is this technique may reduce the total time of thread synchronization overhead by up to a factor of $\frac{m}{q}$ if there are $m$ workers in total and each highest leveled manager has only $q$ workers. The second is the number of workers per the highest leveled manager can be kept as constant. Thus, the complexity of thread synchronization overhead can be kept as constant as well.
BIBLIOGRAPHIES


[8] W. Chantamas, J. Baker, and M. Scherger, *Compiler Extension of the ASC Language to Support Multiple Instruction Streams in the MASC Model Using the Manager-worker*


[39]  G. Nudd, N. Francis, T. Atherton, D. Kerbyson, R. Packwood, and J. Vaudin,
Hierarchical Multiple-SIMD Architecture for Image Analysis, J. of Machine Vision and


[41]  M. Pietrek, Everything You Need to Know to Start Programming 64-bit Windows Systems,

051294-9.

[43]  M. J. Quinn. Parallel Programming in C with MPI and OpenMP, McGraw-Hill

(1992), 103-113.

[45]  K. Schaffer and R. A. Walker, A Prototype Multithreaded Associative SIMD Processor,
in Proc. of the 21st IPDPS (Workshop on APDCM), Long Beach, California, March 2007.

[46]  M. Scherger, J. Baker, and J. Potter, Multiple Instruction Stream Control for an
Associative Model of Parallel Computation, in Proc. of the 16th International Parallel and
Distributed Processing Symposium (Workshop in Massively Parallel Processing), April 2003.


APPENDIX A

A Guide on How to Program on a MASC Simulator

Two important files, MASC.h and MASC.cpp, contain the MASC software library and data declarations of PEs and ISs. The MASC simulator simulates the manager instruction stream first, starting from FINISHED to TERMINATION Phases, followed by each worker instruction stream executing an assigned task and PEs.

![Diagram](image_url)

Figure A.1: The C Compilation Model
These are steps required for a programmer to run the MASC simulator.

1. Modify MASC.h
   a. Indicate how many ISs (is_num) and PEs (pe_num) will be used in the
      simulator. The numbers of both PEs and worker ISs have to be powers of
      2. Usually, the practical number of PEs in the simulator is from a few
      PEs up to a few thousands PEs.
   b. Modify PE data record according to the problem. Parallel variables are
      in the form of arrays, i.e., data1[pe_num]. Scalar variables can be
      declared without using an array, i.e., saclar1.
   c. A user program may require modifications to manager functions such as
      FINISHED(), FORK(), ASSIGN(), JOIN(), and TERMINATION().
   d. A user program may need input-output functions to get inputs from input
      files and output PEs’ contents and other data to output files.

2. Modify MASC.cpp
   a. Initialize data in the Default constructor MASC::MASC().
   b. If a manager function is modified. An implementation of such function
      will be in this MASC.cpp file.
   c. New input-output files are implemented in this MASC.cpp.

3. Create a user program, i.e., my_program.cpp
a. Add #include "MASC.h" in the header to indicate this program will use the MASC library.

b. In the main(), declare a class MASC as MASC My_MASC;

c. Initialize data if needed and then call input functions to get data to PEs.

d. Get an initial task tree from a file, i.e., My_MASC.GetTaskTree ("my_tasks.txt");. This way, the manager will build a task tree based on the received data. Each input line of my_tasks.txt contains task type, child1 id, child2 id, parent1 id, and parent2 id. Task type is one of -1 is a worker task, 0 is a fork, and 1 is a join. -1 is used as a null task id.

e. This user program heavily uses switch statements to execute tasks. Figure 15 shows an example of how to use a switch statement in the user program.

f. Put instructions of the MASC algorithm for task k in case k of the switch statement.

g. Extra instructions are needed at each worker task

   i. Switch PEs back to the manager

   ii. Change the worker’s state from Busy to Finish

   iii. Update the worker's local clock.
The following part of a program shows a use of the switch statement to organize tasks into cases within the switch statement. The first manager task is always at case number “the maximum task” + 1. In this example, in the program, the maximum number of tasks is defined as a variable MAX_task. Below is a skeleton of a part of a MASC program.

```c
while (terminate == false) {
    switch (task_id) {
        case 0: // the first task, id 0
            // Put instructions for task 0 here.
            break;
        ...
        case k: // k is the last task id
            // Put instructions for task k here.
            break;
        // MANAGER TASKS
        case (MAX_task+0): // CHECK FOR A FINISHED TASK
            // call My_MASC.FINISHED();
            break;
    }
}
```
case (MAX_task+1): // FORK

// call My_MASC.Fork();
break;

case (MAX_task+2): // JOIN

// call My_MASC.JOIN();
break;

case (MAX_task+3): // ASSIGN A NEW TASK

// call My_MASC.ASSIGN();
break;

case (MAX_task+4): // CHECK FOR TERMINATED

// check the condition by calling My_MASC.TERMINATED();
break;

case (MAX_task+5): // SIMULATE ALL WORKERS

// Simulate one worker at a time
break;

default: terminate = true;

cout << "***ERROR, invalid task id."

endl;
} // end of switch

} // end of while not terminate
APPENDIX B

A List of MASC Functions

This appendix provides a list of functions available in the MASC library.

1. Functions for Task Managements

1.1 Getting a task tree

Function: GetTaskTree

This function gets input task data from a user provided text file in order to build a task tree used by manager IS.

Input: An input task data text file

Output: A type of each task (a worker task, a fork task, or a join task), its children tasks (if any), and its parent tasks (if any) are stored in an array.
1.2 Adding a task into a task pool

Function: AddTasktoPool

This function adds a task into one of these task pools (depending on a type of that task), a worker task pool, a fork pool, or a join pool.

Input: A task id

Output: The task is inserted at the end of the list of a pool.

2. Functions for Instruction Stream Managements

2.1 Checking a state of a worker

Function: AnyWorker

Manager IS, by utilize the IS broadcast and reduction network, is able to check if any worker instruction stream is in a certain state (ready, busy, or finish).

Input: A worker IS state (ready, busy, or finish)

Output: If there is at least one worker instruction stream that is in the same state, a 1 (true) is returned to the manager IS. Otherwise, a 0 (false) is returned.
2.2 Selecting one worker

**Function:** PickOneWorker

By utilize an IS broadcast and reduction network, manager IS can selects one worker out whose state matches the given state. This function performs a global MAX reduction of worker ids. To avoid any error, this function should be called after the previous Anyworker call of the same given state returned a 1 (true)

**Input:** A worker state (ready, busy, or finish)

**Output:** The largest id of a worker instruction stream whose state matches the given state is returned. If there is no worker IS’s state matches the given state, a -1 is returned as a worker IS id. This should indicate an error.

2.3 Reduce data from a worker

**Function:** ReduceWorkerTask

By using an IS broadcast and reduction network, any specific data from a worker IS can be reduced back to the manager IS. Note that, a communication between manager IS and a worker IS can be established only if the worker IS’s state is either ready or finish.

**Input:** An id of a worker IS

**Output:** A task id is returned if the worker instruction stream is currently having that task. Otherwise, a -1 is returned as a task id. This means there is no task associated with the worker IS.
2.4 Finished

**Function:** FINISHED

In this function, manager IS executes its Finished phase of its simulation cycle. If there is a finished task at a worker IS, the manager IS will collect that task. The collected task is marked as finished. Its children tasks may be added to a task pool as new tasks. This function may be modified to suit the executed program.

**Input:** None

**Output:** None

2.5 Fork

**Function:** FORK

In this function, manager IS executes its Fork phase of its simulation cycle by taking out a fork task from the fork pool. PEs will be partitioned and newly created tasks may be added to a task pool. This function may be modified to suit the executed program.

**Input:** None

**Output:** None
2.6 Assign

Function: ASSIGN

In this function, manager IS executes its Assign phase of its simulation cycle. A ready worker instruction stream is selected. A task in a worker task pool is assigned to that worker IS. This function may be modified to suit the executed program.

Input: None

Output: None

2.7 Join

Function: JOIN

In this function, manager IS executes its Join phase of its simulation cycle by taking out a join task from the join pool. PEs from two finished tasks will be combined into one partition. A new created task may be added to a task pool. This function may be modified to suit the executed program.

Input: None

Output: None
2.8 Termination

Function: TERMINATED.

In this function, manager IS executes its Termination phase of its simulation cycle. The manager IS checks to see if all worker ISs are ready and no task in a task pool. If the condition is met, the execution of the program is terminated. Otherwise, the simulation continues. This function may be modified to suit the executed program.

Input: None.

Output: None.
APPENDIX C

MASC Functions and Their Worst Case Cost

The following table shows the worst case cost of functions available in the MASC library.

<table>
<thead>
<tr>
<th>Functions</th>
<th>Operations</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>MASC</td>
<td>Initialization</td>
<td>0</td>
</tr>
<tr>
<td>Push</td>
<td>+2 if statement</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>+1 set inside the then/else</td>
<td></td>
</tr>
<tr>
<td></td>
<td>+1 set</td>
<td></td>
</tr>
<tr>
<td>AllSwitchIS</td>
<td>+1 broadcast to PEs</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>+2 if statement</td>
<td></td>
</tr>
<tr>
<td></td>
<td>+1 set inside the then</td>
<td></td>
</tr>
<tr>
<td>AllAssignTask</td>
<td>+1 broadcast to PEs</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>+2 if statement</td>
<td></td>
</tr>
<tr>
<td></td>
<td>+4 call Push inside the then</td>
<td></td>
</tr>
<tr>
<td>AllMRset</td>
<td>+1 broadcast to PEs</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>+2 if statement</td>
<td></td>
</tr>
<tr>
<td></td>
<td>+1 set inside the then</td>
<td></td>
</tr>
<tr>
<td>AllMRreset</td>
<td>+1 broadcast to PEs</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>+2 if statement</td>
<td></td>
</tr>
<tr>
<td></td>
<td>+1 set inside the then</td>
<td></td>
</tr>
<tr>
<td>MRset</td>
<td>+1 set</td>
<td>1</td>
</tr>
<tr>
<td>MRreset</td>
<td>+1 set</td>
<td>1</td>
</tr>
<tr>
<td>ItsPE</td>
<td>+1 a comparison of two items</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>+1 a comparison of two items</td>
<td></td>
</tr>
<tr>
<td></td>
<td>+1 a comparison of two items</td>
<td></td>
</tr>
<tr>
<td>BoolAndReduction</td>
<td>+1 broadcast to PEs</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>+4 if statement of 3 comparisons</td>
<td></td>
</tr>
<tr>
<td></td>
<td>+1 set inside the then/else</td>
<td></td>
</tr>
<tr>
<td></td>
<td>+1 value propagates back</td>
<td></td>
</tr>
<tr>
<td>IntAndReduction</td>
<td>+1 broadcast to PEs</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>+4 if statement of 3 comparisons</td>
<td></td>
</tr>
<tr>
<td></td>
<td>+1 set inside the then/else</td>
<td></td>
</tr>
<tr>
<td></td>
<td>+1 value propagates back</td>
<td></td>
</tr>
<tr>
<td>Operator</td>
<td>PEs Operations</td>
<td>Value Propagations</td>
</tr>
<tr>
<td>-------------------</td>
<td>--------------------------------</td>
<td>--------------------</td>
</tr>
<tr>
<td>BoolOrReduction</td>
<td>+1 broadcast to PEs</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>+4 if statement of 3 comparisons</td>
<td></td>
</tr>
<tr>
<td></td>
<td>+1 set inside the then/else</td>
<td></td>
</tr>
<tr>
<td></td>
<td>+1 value propagates back</td>
<td></td>
</tr>
<tr>
<td>IntOrReduction</td>
<td>+1 broadcast to PEs</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>+4 if statement of 3 comparisons</td>
<td></td>
</tr>
<tr>
<td></td>
<td>+1 set inside the then/else</td>
<td></td>
</tr>
<tr>
<td></td>
<td>+1 value propagates back</td>
<td></td>
</tr>
<tr>
<td>IntMaxReduction</td>
<td>+1 broadcast to PEs</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>+4 if statement of 3 comparisons</td>
<td></td>
</tr>
<tr>
<td></td>
<td>+1 set inside the then/else</td>
<td></td>
</tr>
<tr>
<td></td>
<td>+1 value propagates back</td>
<td></td>
</tr>
<tr>
<td>RealMaxReduction</td>
<td>+1 broadcast to PEs</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>+4 if statement of 3 comparisons</td>
<td></td>
</tr>
<tr>
<td></td>
<td>+1 set inside the then/else</td>
<td></td>
</tr>
<tr>
<td></td>
<td>+1 value propagates back</td>
<td></td>
</tr>
<tr>
<td>IntMinReduction</td>
<td>+1 broadcast to PEs</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>+4 if statement of 3 comparisons</td>
<td></td>
</tr>
<tr>
<td></td>
<td>+1 set inside the then/else</td>
<td></td>
</tr>
<tr>
<td></td>
<td>+1 value propagates back</td>
<td></td>
</tr>
<tr>
<td>RealMinReduction</td>
<td>+1 broadcast to PEs</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>+4 if statement of 3 comparisons</td>
<td></td>
</tr>
<tr>
<td></td>
<td>+1 set inside the then/else</td>
<td></td>
</tr>
<tr>
<td></td>
<td>+1 value propagates back</td>
<td></td>
</tr>
<tr>
<td>BoolAssoSearch</td>
<td>+1 broadcast to PEs</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>+4 if statement of 3 comparisons</td>
<td></td>
</tr>
<tr>
<td></td>
<td>+2 if statement inside the then</td>
<td></td>
</tr>
<tr>
<td></td>
<td>+1 set inside the then/else</td>
<td></td>
</tr>
<tr>
<td>CharAssoSearch</td>
<td>+1 broadcast to PEs</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>+4 if statement of 3 comparisons</td>
<td></td>
</tr>
<tr>
<td></td>
<td>+2 if statement inside the then</td>
<td></td>
</tr>
<tr>
<td></td>
<td>+1 set inside the then/else</td>
<td></td>
</tr>
<tr>
<td>IntAssoSearch</td>
<td>+1 broadcast to PEs</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>+4 if statement of 3 comparisons</td>
<td></td>
</tr>
<tr>
<td></td>
<td>+2 if statement inside the then</td>
<td></td>
</tr>
<tr>
<td></td>
<td>+1 set inside the then/else</td>
<td></td>
</tr>
<tr>
<td>RealAssoSearch</td>
<td>+1 broadcast to PEs</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>+4 if statement of 3 comparisons</td>
<td></td>
</tr>
<tr>
<td></td>
<td>+2 if statement inside the then</td>
<td></td>
</tr>
<tr>
<td></td>
<td>+1 set inside the then/else</td>
<td></td>
</tr>
<tr>
<td>StringAssoSearch</td>
<td>+1 broadcast to PEs</td>
<td>8</td>
</tr>
<tr>
<td>Function</td>
<td>Cost Breakdown</td>
<td>Cost</td>
</tr>
<tr>
<td>-------------------</td>
<td>-------------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>AnyResponder</td>
<td>+7 call BoolOrReduction function</td>
<td>7</td>
</tr>
<tr>
<td>PickOne</td>
<td>+1 broadcast to PEs, +4 if statement of 3 comparisons, +2 if statement, +1 set inside the then/else</td>
<td>10</td>
</tr>
<tr>
<td>ReduceBool</td>
<td>+1 broadcast to PEs, +2 if statement, +1 set inside the then/else, +1 value propagates back</td>
<td>5</td>
</tr>
<tr>
<td>ReduceInt</td>
<td>+1 broadcast to PEs, +2 if statement, +1 set inside the then/else, +1 value propagates back</td>
<td>5</td>
</tr>
<tr>
<td>ReduceReal</td>
<td>+1 broadcast to PEs, +2 if statement, +1 set inside the then/else, +1 value propagates back</td>
<td>5</td>
</tr>
<tr>
<td>ReduceChar</td>
<td>+1 broadcast to PEs, +2 if statement, +1 set inside the then/else, +1 value propagates back</td>
<td>5</td>
</tr>
<tr>
<td>ReduceString</td>
<td>+1 broadcast to PEs, +2 if statement, +1 set inside the then/else, +1 value propagates back</td>
<td>5</td>
</tr>
<tr>
<td>InsertTP</td>
<td>+2 if statement, +2 two sets inside the then/else, +2 two sets</td>
<td>6</td>
</tr>
<tr>
<td>RemoveTP</td>
<td>+2 if statement, +4 six sets inside the then</td>
<td>6</td>
</tr>
<tr>
<td>TPisEmpty</td>
<td>+1 set</td>
<td>1</td>
</tr>
<tr>
<td>InsertFP</td>
<td>+2 if statement, +2 two sets inside the then/else, +2 two sets</td>
<td>6</td>
</tr>
<tr>
<td>RemoveFP</td>
<td>+2 if statement</td>
<td>6</td>
</tr>
<tr>
<td>Method</td>
<td>Description</td>
<td>Complexity</td>
</tr>
<tr>
<td>-------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>------------</td>
</tr>
<tr>
<td>FPisNotEmpty</td>
<td>+1 set</td>
<td>1</td>
</tr>
<tr>
<td>InsertJP</td>
<td>+2 if statement&lt;br&gt;+2 two sets inside the then/else&lt;br&gt;+2 two sets</td>
<td>6</td>
</tr>
<tr>
<td>RemoveJP</td>
<td>+2 if statement&lt;br&gt;+4 six sets inside the then</td>
<td>6</td>
</tr>
<tr>
<td>JPisNotEmpty</td>
<td>+1 set</td>
<td>1</td>
</tr>
<tr>
<td>PeekForkPool</td>
<td>+2 if statement&lt;br&gt;+1 set inside the then/else</td>
<td>3</td>
</tr>
<tr>
<td>GetChild</td>
<td>+1 set</td>
<td>1</td>
</tr>
<tr>
<td>GetParent</td>
<td>+1 set</td>
<td>1</td>
</tr>
<tr>
<td>isFork</td>
<td>+2 if statement&lt;br&gt;+1 set inside the then/else</td>
<td>3</td>
</tr>
<tr>
<td>isJoin</td>
<td>+2 if statement&lt;br&gt;+1 set inside the then/else</td>
<td>3</td>
</tr>
<tr>
<td>isWorkersTask</td>
<td>+2 if statement&lt;br&gt;+1 set inside the then/else</td>
<td>3</td>
</tr>
<tr>
<td>AddTasktoPool</td>
<td>+3 call isFork/isJoin/isWorkersTask&lt;br&gt;+2 if statement&lt;br&gt;+6 call InsertTP/FP/JP</td>
<td>11</td>
</tr>
<tr>
<td>AnyWorker</td>
<td>+1 broadcast to Workers&lt;br&gt;+2 if statement&lt;br&gt;+2 one set inside the then/else and one comparison&lt;br&gt;+1 value propagates back</td>
<td>6</td>
</tr>
<tr>
<td>AllWorkerReady</td>
<td>+1 broadcast to Workers&lt;br&gt;+2 if statement&lt;br&gt;+2 one set inside the then/else and one comparison&lt;br&gt;+1 value propagates back</td>
<td>6</td>
</tr>
<tr>
<td>AllWorkerFinish</td>
<td>+1 broadcast to Workers&lt;br&gt;+2 if statement&lt;br&gt;+2 one set inside the then/else and one comparison&lt;br&gt;+1 value propagates back</td>
<td>6</td>
</tr>
<tr>
<td>PickOneWorker</td>
<td>+1 broadcast to Workers&lt;br&gt;+3 one if statement and one comparison&lt;br&gt;+1 set inside the then/else&lt;br&gt;+1 value propagates back</td>
<td>6</td>
</tr>
<tr>
<td>Method</td>
<td>Operations</td>
<td>Cost</td>
</tr>
<tr>
<td>--------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>------</td>
</tr>
</tbody>
</table>
| ReduceWorkerTask               | +1 broadcast to Workers  
+3 one if statement and one comparison  
+1 set inside the then/else  
+1 value propagates back       | 6    |
| IsBusy                         | +2 if statement                                                           | 2    |
| UpdateWorkerState              | +1 set                                                                    | 1    |
| UpdateWorkerTask               | +1 set                                                                    | 1    |
| FORK                           | +6 call RemoveFP  
+1 call GetChild  
+2 if statement  
+1 broadcast to PEs  
+6 if statement of 5 comparisons  
+4 call Push  
+11 call AddTaskToPool  
+1 call GetChild  
+2 if statement  
+1 broadcast to PEs  
+6 if statement of 5 comparisons  
+4 call Push  
+11 call AddTaskToPool  
+1 set | 57   |
| FORK_QUICKHULL                 | +6 call RemoveFP  
+3 if statement of two operations  
+5 five sets inside the then/else  
+1 broadcast to PEs  
+4 if statement of 3 comparisons  
+4 if statement of 3 comparisons  
+2 if statement  
+1 set inside the then/else  
+12 two calls of InsertTP | 38   |
| FORK_FLOYD                     | +6 call RemoveFP  
+6 call AllWorkerReady (may omit this)  
+2 if statement  
+1 broadcast to PEs  
+7 call Push and 3 assignments | 22 (16) |
| ASSIGN                         | +10 while statement of 3 comparisons & 2 Fn. Calls  
+6 call PickOneWorker  
+6 call RemoveTP  
+4 call AllSwitchIS | 28   |
<table>
<thead>
<tr>
<th>Event</th>
<th>Description</th>
<th>Complexity</th>
</tr>
</thead>
</table>
| ASSIGN_QUICKHULL   | +10 while statement of 3 comparisons & 2 Fn. Calls  
+6 call PickOneWorker  
+6 call RemoveTP  
+4 call AllMRset  
+8 call BoolAssoSearch  
+7 call AnyResponder  
+2 if statement  
+4 call AllSwitchIS  
+1 call UpdateWorkerTasks  
+1 call UpdateWorkerState | 49         |
| ASSIGN_FLOYD       | +2 if statement  
+1 broadcast to PEs  
+7 if statement with 6 comparisons and assignments in else  
  +1 set inside the then  
  +2 two broadcasts to workers  
+1 set | 14         |
| FINISHED           | +7 while statement of a comparison & 1 Fn. Call  
+6 call PickOneWorker  
+6 call ReduceWorker  
+1 set  
+1 call GetChild  
+2 if statement  
  +11 call AddTaskToPool inside then  
  +1 call UpdateWorkerTasks  
+1 call UpdateWorkerState | 36         |
| FINISHED_QUICKHULL | +7 while statement of a comparison & 1 Fn. Call  
+6 call PickOneWorker  
+6 call ReduceWorker  
+1 set  
+1 call GetChild  
+2 if statement  
  +6 call InsertFP  
+1 call UpdateWorkerTasks  
+1 call UpdateWorkerState | 28         |
| FINISHED_FLOYD     | +2 if statement  
+6 call AllWorkerFinish (may omit this)  
+2 two broadcasts to workers | 22 (16)    |
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>JOIN</td>
<td>+2 if statement +2 two sets inside the else +2 if statement +6 call InsertFP</td>
<td>54</td>
</tr>
<tr>
<td></td>
<td>+3 while statement of a comparison &amp; 1 Fn. Call +6 call RemoveJP +2 if statement +2 two calls of GetParent +1 call GetChild +1 set + 4 if statement of 3 comparisons +2 one comparison and 1 set +2 if statement +1 broadcast to PEs + 8 two if statements of 3 comparisons +8 two calls of Push +2 if statement +11 call AddTasktoPool +1 set</td>
<td></td>
</tr>
<tr>
<td>TERMINATED</td>
<td>+ 16 if statement of 7 comparisons &amp; 4 Fn. Calls +1 set inside the then/else</td>
<td>17</td>
</tr>
<tr>
<td>TERMINATED_FLOYD</td>
<td>+2 if statement +1 set</td>
<td>3</td>
</tr>
</tbody>
</table>