Accelerating Component-Based Dataflow Middleware with Adaptivity and Heterogeneity

Dissertation

Presented in Partial Fulfillment of the Requirements for the Degree Doctor of Philosophy in the Graduate School of The Ohio State University

By

Timothy D. R. Hartley, M.S.

Graduate Program in Electrical and Computer Engineering

The Ohio State University

2011

Dissertation Committee:

Professor Ümit V. Çatalyürek, Advisor

Professor Füsun Özgüner

Professor Charles A. Klein

Abstract

This dissertation presents research into the development of high performance dataflow middleware and applications on heterogeneous, distributed-memory supercomputers. We present coarse-grained state-of-the-art ad-hoc techniques for optimizing the performance of real-world, data-intensive applications in biomedical image analysis and radar signal analysis on clusters of computational nodes equipped with multi-core microprocessors and accelerator processors, such as the Cell Broadband Engine and graphics processing units. Studying the performance of these applications gives valuable insights into the relevant parameters to tune for achieving efficiency, because being large-scale, data-intensive scientific applications, they are representative of what researchers in these fields will need to conduct innovative science. Our approaches shows that multi-core processors and accelerators can be used cooperatively to achieve application performance which may be many orders of magnitude above naïve reference implementations. Additionally, a fine-grained programming framework and runtime system for the development of dataflow applications for accelerator processors such as the Cell is presented, along with an experimental study showing our framework leverages all of the peak performance associated with such architectures, at a fraction of the cognitive cost to developers. Then, we present an adaptive technique for automating the coarse-grained ad-hoc optimizations we developed for tuning the decomposition of application data and tasks for parallel execution on distributed, heterogeneous processors. We show that our technique is able to achieve high performance, while significantly reducing the burden placed on the developer to manually tune the relevant parameters of distributed dataflow applications. We evaluate the performance of our technique on three real-world applications, and show that it performs favorably compared to three state-of-the-art distributed programming frameworks. By bringing our adaptive dataflow middleware to bear on supporting alternative programming paradigms, we show our technique is flexible and has wide applicability.

To Annie and Charlotte

Acknowledgments

Graduate school is hardly a solo endeavor, and I am indebted to the following people for their support.

I would like to thank the Dayton Area Graduate Studies Institute and Dr. Elizabeth Downie for providing support for me to be able to focus on my research. The flexibility a fellowship affords was deeply appreciated.

I am grateful to my collaborators Charles Berdanier, AFRL, and Ahmed Fasih for their help drilling radar signal analysis into my head, what little it would hold. I would like to thank Dr. George Teodoro and Professor Renato Ferreira for their collaborations. I would like to thank Professor Manuel Ujaldón and Antonio Ruiz for helping me get interested in GPUs.

I would like to thank Professor Füsun Özgüner for her invaluable support and her committee membership. I am also grateful to Professor Charles Klein for his committee membership, and I wish him well in his impending retirement!

I would especially like to thank Dr. Erik Saule for his constant advice, support, on-point technical criticisms, and his friendship. I owe thanks also to all of the other members of the HPC lab.

I will never be able to thank Professor Ümit Çatalyürek enough for his Herculean support in getting me to this point. There were too many times to count where an encouraging talk lifted me out of yet another motivation black hole. His advice, support, kindness, and friendship will always be truly appreciated.

I'd like to thank my daughter, Charlotte Madeline Lindgren Hartley for her amazing attitude, intelligence, and sense of humor. She has changed my life.

Lastly I'd like to thank Dr. Annie Rose Lindgren for being the best partner anyone could ever hope for. For her steadfast support, advice, patience, and good humor in the face of a difficult situation (dealing with a graduate student husband while being a talented research scientist and an incredible mother), I will forever be in her debt.

Vita

January 16, 1980	Born - High Wycombe, UK
2002	B.S. Electrical and Computer Engineering, New Mexico State University
2005-2011	Graduate Research Associate, The Ohio State University.
2006	M.S. Electrical and Computer Engineering, The Ohio State University
2009-2011	Air Force Research Laboratory / Dayton Area Graduate Studies Ohio Student-Faculty Fellow

Publications

T. D. R. Hartley, U. V. Catalyurek, A. Ruiz, F. Igual, R. Mayo, and M. Ujaldon. Biomedical image analysis on a cooperative cluster of gpus and multicores. In *Proceedings of the 22nd Annual International Conference on Supercomputing, ICS 2008*, pages 15–25, 2008.

F. Igual, R. Mayo, T. D. R. Hartley, U. V. Catalyurek, A. Ruiz, and M. Ujaldón. Optimizing co-occurrence matrices on graphics processors using sparse representations. In *Proceedings of the 9th International Workshop on State-of-the-Art in Scientific and Parallel Computing (PARA '08)*, 2008.

H. G. Ozer, D. Bozdag, T. Camerlengo, J. Wu, Y.-W. Huang, T. Hartley, J. D. Parvin, T. Huang, U. V. Catalyurek, and K. Huang. A comprehensive analysis workflow for genome-wide screening data from chip-sequencing experiments. In *Proceedings of 1st International Conference on Bioinformatics and Computational Biology*, volume 5462 of *Lecture Notes in Computer Science*, pages 320–330. Springer, April 2009.

T. D. R. Hartley and U. V. Catalyurek. A component-based framework for the cell broadband engine. In *Proceedings of 23rd International Parallel and Distributed Processing Symposium, The 18th Heterogeneous Computing Workshop (HCW 2009)*, May 2009.

T. D. R. Hartley, A. R. Fasih, C. A. Berdanier, F. Özgüner, and U. V. Catalyurek. Investigating the use of GPU-accelerated nodes for SAR image formation. In *Proceedings of the IEEE International Conference on Cluster Computing, Workshop on Parallel Programming on Accelerator Clusters (PPAC)*, 2009.

U.V. Catalyurek, T. Hartley, O. Sertel, M. Ujaldon, A. Ruiz, J. Saltz, and M. Gurcan. Processing of large-scale biomedical images on a cluster of multi-core cpus and gpus. In W. Gentzsch, L. Grandinetti, and G. Joubert, editors, *High Performance and Large Scale Computing*, volume 18, pages 341–364. IOS Press, 2009.

F. Igual, R. Mayo, T. D. R. Hartley, U. Catalyurek, A. Ruiz, and M. Ujaldon. Exploring the gpu for enhancing parallelism on color and texture analysis. In *Proceedings of the 2009 International Conference on Parallel Computing (ParCo 2009)*, 2009.

U. V. Catalyurek, R. Ferreira, T. D. R. Hartley, R. Sachetto, and G. Teodoro. Dataflow frameworks for emerging heterogeneous architectures and its application to biomedicine. In *Scientific Computing with Multicore and Accelerators*. Chapman and Hall / CRC Press, 2010.

T. D. R. Hartley, E. Saule, and U. V. Catalyurek. Automatic dataflow application tuning for heterogeneous systems. In *Proceedings of The 17th International Conference on High Performance Computing (HiPC 2010)*, 2010.

F. Igual, R. Mayo, T. D. R. Hartley, U. Catalyurek, A. Ruiz, and M. Ujaldon. Color and texture analysis on emerging parallel architectures. *International Journal of High Performance Computing Applications*, to appear.

G. Teodoro, T. D. R. Hartley, U. V. Catalyurek, and R. Ferreira. Run-time optimizations for replicated dataflows on heterogeneous environments. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing (HPDC)*, 2010.

G. Teodoro, T. D. R. Hartley, U. V. Catalyurek, and R. Ferreira. Optimizing dataflow applications on heterogeneous environments. *Cluster Computing*, to appear.

T. D. R. Hartley, E. Saule, and Ü. V. Çatalyürek. Evaluating support for heterogeneous computing in high-level distributed programming frameworks. In *Proceedings of The Twentieth International Conference on Parallel Architectures and Compilation Techniques (PACT 2011)*, 2011, under review.

T. D. R. Hartley, E. Saule, and Ü. V. Çatalyürek. Improving performance of adaptive component-based dataflow middleware. *Parallel Computing*, 2011, under review.

Fields of Study

Major Field: Electrical and Computer Engineering

Table of Contents

List of Tables

List of Figures

Chapter 1: Introduction

1.1 Motivation

In recent years, a growing falloff of the performance of microprocessors compared to Moore's Law (which states that the number of transistors on a chip will double roughly every eighteen months) has created a new opportunity for specialized, high-performance accelerator architectures such as the Cell Broadband Engine (CBE) [50]. The same trend of microprocessor performance rolloff, predominantly caused by increasing power consumption, has forced microprocessor manufacturers to increasingly rely on on-die, CPU-level parallelism for increased performance [79]. Further, during the same time period, Graphics Processing Unit (GPU) manufacturers have increased programmability and computational throughput such that GPUs are an attractive substitute for standard CPU architectures in certain extremely parallel and computationally demanding applications [81].

These new architectures have very high computational capability, but applications often need to be restructured to take advantage of the high performance on tap when using emerging architectures. Traditional techniques - pipelining of instruction execution, superscalar instruction execution, out-of-order instruction execution, as well as the introduction of SIMD operations to commodity microprocessors - have provided sequential programs enough performance [113] to make the added benefit of parallel computation not worth the additional complexity in application development. However, with the increasing parallelism inherent in new architectures, the available performance for certain applications [81, 51, 53, 112, 118, 93, 92] makes the increased development cost acceptable, since making full use of these high-performance architectures leads to cost-savings in other areas, such as lead time to market, reduced latency in decision-making, or power savings due to reduced computation time. Additionally, certain applications absolutely require extremely fast computation times to even be feasible such as certain types of medical image analysis [20, 96] and radar signal processing [56]. If sequential versions of certain computerized image analysis applications were the only ones available, no clinical practice, research laboratory, or military concern could use them, due to their excessively large execution times.

The modern supercomputers which can support these types of applications are built by using fast networks to connect large clusters of commodity microprocessors and emerging architecture-based nodes [8]. These systems are becoming more difficult to program because of the increasing levels of hierarchy and heterogeneity present in modern supercomputer designs due to the result of system upgrades, or more recently, the result of explicit initial design decisions. Current practical software solutions have not kept pace with these programming challenges, placing most of the burden on the developer [4]. To address this issue, this dissertation shows that component-based dataflow middleware allows developers of high-performance scientific applications to leverage all of the peak computational capability inherent in heterogeneous, distributed supercomputers, without requiring heavy use of low-level programming tools requiring expert knowledge. Further, efficient solutions to tune and schedule component-based applications for these systems are few and far between, even in state-of-the-art middleware runtime systems. Therefore, this dissertation presents an adaptive framework and runtime system for developing and executing component-based dataflow applications on heterogeneous, distributed supercomputers. We show our framework is able to generalize our ad-hoc optimization approaches and achieve comparable performance to manually tuned implementations in a number of large-scale, real-world applications.

The rest of this introduction consists of an overview of the state-of-the-art of component-based programming frameworks, followed by a presentation of autotuning research, and of the programming models and runtime systems commonly used to develop applications for supercomputers. Following these sections, we present our contributions.

1.2 State of the Art

1.2.1 Component-Based Framework Overview

The central thesis of component-based programming is that there are many advantages to describing and implementing complex applications by way of components - distinct tasks with well-defined interfaces. By describing these components and the explicit data connections between them, many advantages are conferred. Applications are decomposed along natural task boundaries, according to the application domain. Therefore, component-based application design is an intuitive process, with explicit demarcation of task responsibilities. Further, the communication patterns are also explicit, as each component includes in its description its input data requirements and outputs. Since many applications are comprised of a series of serial processing or analysis steps, explicitly defining a task graph to represent these processing steps is a natural method for breaking up and writing the program.

Beyond the implementation benefits, component-based programming also enables some runtime benefits, which come at no additional cost to the developer. Applications composed of a number of individual tasks can be executed on parallel and distributed computing resources and gain extra performance over those run on strictly sequential machines. Additionally, provided the interfaces exposed by a task to the rest of the application match, different implementations of tasks, possibly on different processor architectures can co-exist in the same application deployment, allowing developers to take full advantage of modern, heterogeneous supercomputers.

Filter-stream programming is an instance of component-based programming supported by DataCutter, a component-based middleware tool. The filter-stream programming model [13] (a specific implementation of the dataflow programming model [33]) implements computations as a set of components, referred to as *filters*, that exchange data through *logical streams*. A *stream* denotes a uni-directional data flow from some filters (i.e., the producers) to others (i.e., the consumers). Data flows along these *streams* in untyped *databuffers* so as to minimize various system overheads. A *layout* is a filter ontology which describes the set of application tasks, streams, and the connections required for the computation. A *placement* is one instance of a *layout* with actual filter copy to physical processor mappings. A *filter* can be *replicable*, if it is stateless; for instance, if a filter's output for a given *databuffer* does not depend on the ones it processed previously, it is stateless and replicable. Traditionally, the filter-stream programming model is used through a middleware runtime system, which sits on top of the operating system(s) of the involved computational nodes and provides the dataflow interface abstraction to the application. Figure 1.1 shows an example filter-stream layout and placement.

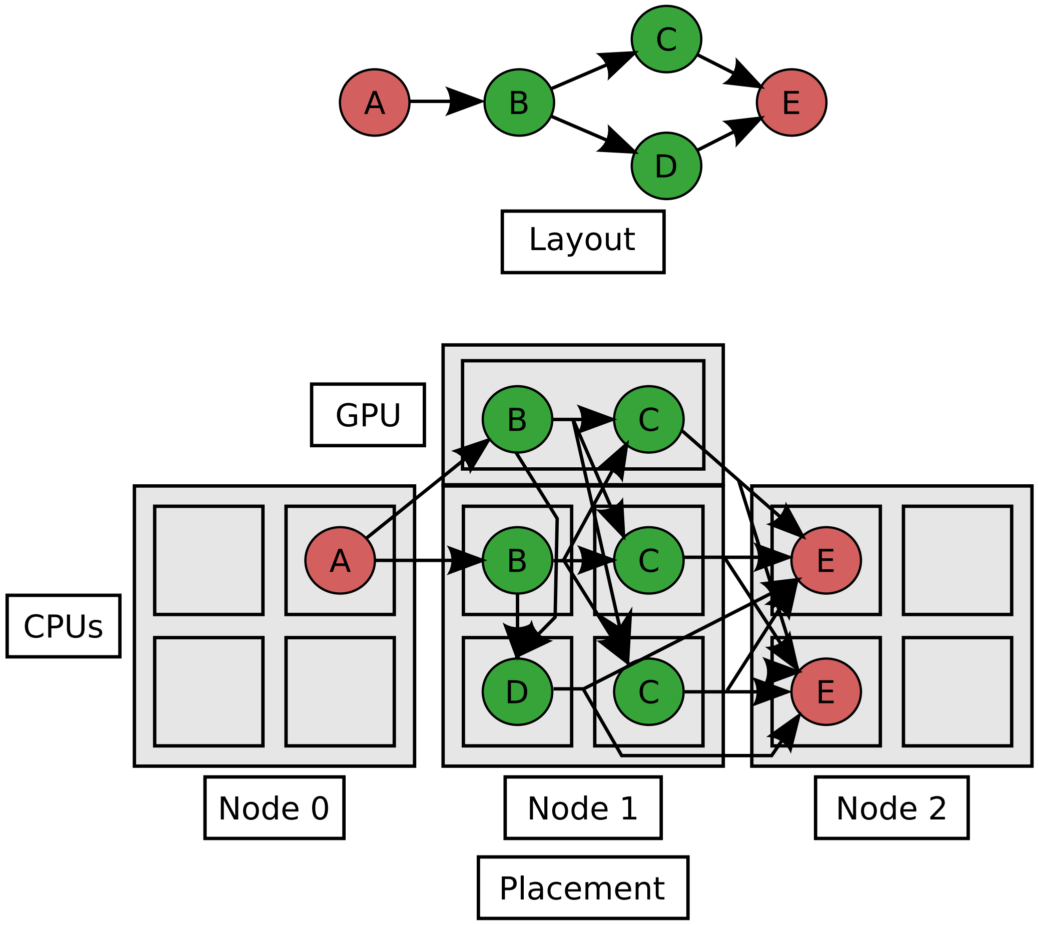


Figure 1.1: Example Filter-stream Application Layout and Placement

In the filter-stream programming model, filters perform all of the computation in the application. Data flows through streams and into filters, and undergoes transformations inside the filters. Filter functions can transform data in a number of ways, ranging from one-to-one databuffer functions, to data join functions (where one or more databuffers are required from more than one stream), to data split functions (where more than one databuffer is created based on a single input databuffer). In the filter-stream model, application developers are only responsible for writing the filter functions and determining the filter and stream layout. The filter-stream model is ideal for programming for heterogeneous processor types, because the architecture-specific details are hidden inside the filter function; provided the same data structure interface is used by two implementations of the same filter function targeted at two different architectures, they can be used interchangeably.

1.2.2 Autotuning

Autotuning (automatic performance testing and optimization) is a field where a feedback loop is introduced in the software development process or when the application is executed. This feedback loop is used to allow some adaptive process to occur, such that execution times from test runs of a sequential or a parallel application can be used to choose application parameters such as data partitioning, number of concurrent processing threads, size of computational kernel, size of cache footprint, extent of loop unrolling, amount of data prefetching. It is an active field, especially in the realm of multicore computing, as traditionally compiled application performance has begun to lag behind the architectures' capabilities [4, 66].

Atune-IL [94] is a language-independent framework for instrumenting parallel applications for an automatic tuning process. The types of tuning include number of threads, data chunk size, and algorithm choice. Developers introduce pragmas into their code, and then build their executable with a special compiler. Then, by using a search algorithm, the framework automatically adapts the instrumented parameters for best performance.

Tiwari et al. present a method for efficiently searching for the highest-performing compiler transformation for an arbitrary set of nested loop iterations [109]. By combining an updated Parallel Rank Ordering algorithm [102] algorithm and a library for generating complex, but correct loop transformations, CHiLL [25], the authors rival the performance of other auto-tuned numerical libraries.

FIBER [68, 67] is a framework for automatically tuning - at install time, before execution, and at runtime - numerical software, by testing which algorithms and loop unrolling parameters achieve the highest performance.

Several researchers at UC Berkeley have conducted extensive analyses of optimizations required to generate high-performance stencil kernels on several state-of-the-art multicore architectures such as standard multi-core CPUs, highly-threaded multi-core CPUs, the Cell Broadband Engine, and GPUs [29, 65]. By first developing architecture-agnostic, application-specific optimization techniques, as well as a framework for automatic optimization, the authors are able to quickly search a large space of optimizations, to find and present the important tradeoffs involved with the area of regular numeric kernals.

POET [115] is a language for describing and applying complex loop optimizations to numerical kernels. It is intended to be used as a communication method between successive optimization steps, and as a generator for different combinations of parameters for optimized kernels.

SPRIAL, Signal Processing Implementation Research for Adaptable Libraries is a framework for autotuning DSP software [90].

ATLAS [111] uses compile-time performance tests to tune its implementations of linear algebra operations. FFTW [42] tunes data structures and algorithms at runtime to efficiently perform FFT operations.

Networking techniques for congestion control and QOS, such as TCP Vegas [18] have been well-known for some time as adaptive methods to improve performance in a distributed setting.

1.2.3 Parallel and Distributed Programming Frameworks

Since our work targets hierarchical, heterogeneous, distributed supercomputers, we will discuss programming models, middleware, and libraries to ease programming for parallel systems, as well as systems with heterogeneous processors or networks. The axes along which we will compare the frameworks are their overall programming model, their supported level of parallelism, their explicit support for heterogeneity, and their load balance capabilities. Figure 1.2 and Table 1.1 show the overall picture.

Our list of frameworks falls into three categories of parallelism, off-load, single-node, and multi-node. The off-load frameworks focus on developing high-performance kernels for graphics processors. While the development of kernels for accelerator devices is outside the scope of this dissertation, GPUs are in and of themselves parallel architectures, and their effective use requires many of the same steps as does more traditional parallel programming: data and task decomposition, parallel algorithm design, etc. Brook [19] was the first framework for developing general purpose computation on GPUs, predating CUDA by several years. However, with the release of the Nvidia CUDA SDK [26], the level of computational power in GPUs had increased steadily, meaning that the commodity status and the increased programmability of

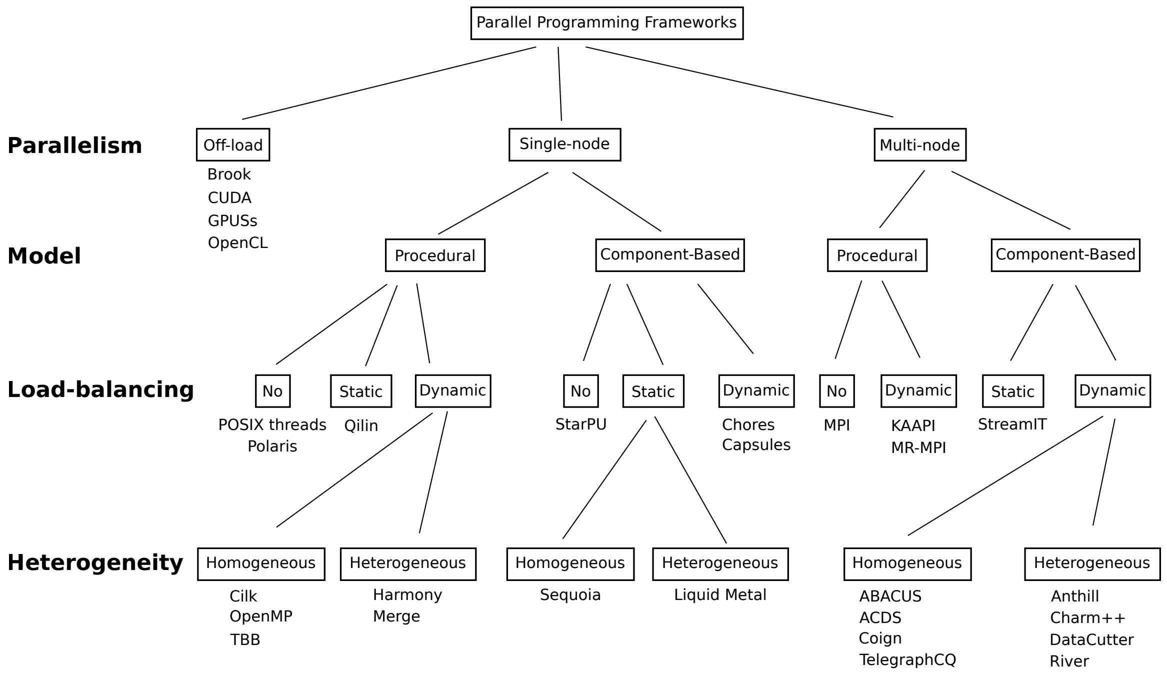


Figure 1.2: Taxonomy of Parallel and Distributed Programming Frameworks

GPUs made for very fertile research ground. Later, higher-level frameworks such as GPUSs [6] were developed for writing applications to leverage multiple GPUs more easily. Finally, the industry has standardized an approach to writing applications for GPUs from all vendors, OpenCL [80].

Though our research focuses on parallel computing in the physically and logically distributed-memory paradigm, the array of frameworks which target single nodes can still be inspirational. These frameworks can be classified along many different axes, but among the most important characteristics is the framework's similarity to sequential programming. Here we call sequential-like programming frameworks *procedural*, while the opposing style is called *component-based*. Component-based frameworks encourage (or require) developers to create multiple, concurrent components and an explicit plan for how these components will interact. Well-known procedural frameworks such as POSIX threads [61] and OpenMP [27] are mature technologies, which are mainstays of high-performance scientific computing. Cilk [16] uses a similar procedural-style programming paradigm, although it adds a work-stealing load-balancing framework, and some theoretical guarantees of asymptotically optimal performance, in certain circumstances. TBB [105] is somewhat of a shared-memory Swiss army knife, in that it provides a framework for exploiting loop iteration parallelism, task parallelism with work-stealing load-balancing, and pipeline parallelism. Finally, frameworks like Qilin [75], Harmony [34], and Merge [74] leverage explicit support for heterogeneous processor types to enable developers to write efficient applications.

High-level parallel and distributed programming frameworks for multi-node systems lower the bar to programming efficient applications for modern HPC systems. As can be seen from Figure 1.2, while single-node frameworks can be found in both the procedural and component-based categories in equal measure, multi-node frameworks are more often component-based than not. Be that as it may, the vast majority of high-performance scientific application software is undoubtedly written in MPI [77]. Before the message-passing standard was developed, there were many competing vendor-specific frameworks, each tied to the manufacturer's specific architecture. While programming efficient applications with an explicit message-passing interface is challenging, developing a parallelizing compiler capable of producing an

Table 1.1: Parallel and Distributed Programming Frameworks

Framework	Programming Model	Parallelism	CPU/GPU	Load Balancing
ABACUS [1]	Component-Based	Multi-node	Homogeneous CPU	Dynamic
ACDS [62]	Component-Based	Multi-node	Homogeneous CPU	Dynamic
Anthill [40]	Component-Based	Multi-node	CPU/GPU	Dynamic
Brook [19]	Procedural	Off-load	GPU	No
CUDA [26]	Procedural	Off-load	GPU	No
Capsules [76]	Component-Based	Single-node	Homogeneous CPU	Dynamic
Charm++ [64]	Component-Based	Mixed	CPU/Accelerators	Dynamic
Chores [36]	Component-Based	Single-node	Homogeneous CPU	Dynamic
Cilk [16]	Procedural	Single-node	Homogeneous CPU	Dynamic
Coign [59]	Component-Based	Multi-node	Homogeneous CPU	Dynamic
DataCutter [13]	Component-Based	Multi-node	CPU/GPU	Dynamic
GPUSs [6]	Procedural	Off-load	GPU	No
Harmony [34]	Procedural	Single-node	CPU/GPU	Dynamic
KAAPI [45]	Procedural	Multi-node	Homogeneous CPU	Dynamic

Continued on next page

Table 1.1 – continued from previous page

Framework	**Model**	**Parallelism**	**Heterogeneity**	**Load Balancing**
Liquid Metal [58]	Component-Based	Single-node	CPU/FPGA	Static
MIT's StreamIT [108]	Component-Based	Multi-node	Homogeneous CPU	Static
MPI [77]	Procedural	Multi-node	Homogeneous CPU	No
MR-MPI [88]	Procedural	Multi-node	Homogeneous CPU	Dynamic
Merge [74]	Procedural	Single-node	CPU/GPU	Dynamic
OpenCL [80]	Procedural	Off-load	GPU	No
OpenMP [27]	Procedural	Single-node	Homogeneous CPU	Dynamic
POSIX threads [61]	Procedural	Single-node	Homogeneous CPU	No
Polaris [15]	Procedural	Single-node	Homogeneous CPU	No
Qilin [75]	Procedural	Single-node	CPU/GPU	Static
River [3]	Component-Based	Multi-node	Heterogeneous CPU	Dynamic
Sequoia [38]	Component-Based	Single-node	Homogeneous CPU	Static
StarPU [5]	Component-Based	Single-node	Homogeneous CPU	No
TBB [105]	Mixed	Single-node	Homogeneous CPU	Dynamic
TelegraphCQ [23]	Component-Based	Multi-node	Homogeneous CPU	Dynamic

efficient parallel version of an arbitrary sequential program is essentially impossible. Thus, the great variety of MPI-based parallel program. However, recently MapReduce [32], another procedural parallel programming framework for multi-node systems, has made a significant impact on developers of parallel programs. By using a simplified application task and data model, MapReduce allows users to write functions which operate on portions of an implicitly-referenced distributed hashtable. Further, MapReduce allows application-oblivious parallel communication. As such, developers of MapReduce programs do not need to write any parallel-specific code.

Component-based distributed programming frameworks attempt to ease the developer's task of managing the often conflicting concerns of processor utilization, network efficiency, and load balance. MIT's StreamIT project [108] provides a framework and compiler to easily produce high-performance streaming dataflow applications. While StreamIT performs heuristic dataflow graph operations which give a best-effort static load balance, many multi-node distributed programming frameworks offer online, dynamic load balancing. This final group of frameworks is still further divided by their explicit support for heterogeneity. The legacy frameworks ABACUS [1], ACDS [62], Coign [59], and TelegraphCQ [23] all focus on homogeneous clusters of SMP systems. KAAPI [45], a newer framework based on the Athapascan-1 language [43], also targets homogeneous clusters, although the authors are currently completely rewriting the codebase from the ground up to explicitly support accelerators. Anthill [40], Charm++ [64], and DataCutter [13] also support heterogeneous CPUs and GPUs with encapsulation and abstraction. Finally, River [3] supports heterogeneous data production and consumption rates, in that producer/consumer relationships are efficiently handled over the network by their flow rate. Thus, any node characteristic that would alter the flow rate is explicitly handled.

1.3 Contributions

The *dataflow* programming paradigm [33] and in particular the *filter-stream* paradigm are uniquely suited to designing large-scale, data-intensive scientific applications for complex, heterogeneous, distributed computer systems. The simple abstraction exposed by dataflow programming and supported by dataflow runtime middleware systems, such as DataCutter [13] and Anthill [40], provides developers with an excellent solution for dealing with the complexities of modern scientific application design for distributed, heterogeneous supercomputers. Thus, this dissertation presents research into developing high-performance, component-based dataflow applications for distributed, heterogeneous supercomputers.

In Part I, we focus on tuning the performance of large-scale applications with ad-hoc approaches. Although the values we select for the tunable parameters are necessarily application-specific, the first part of this dissertation acts as a blueprint for the development of high-performance dataflow applications in the presence of heterogeneity. Chapter 2 presents a high-performance implementation of a real-world biomedical image analysis application for a cluster of GPU-equipped nodes. This application is very data-intensive, with single image sizes reaching nearly 40 GB in the most demanding case. We use DataCutter and CUDA to leverage all of the parallelism inherent in the architecture. DataCutter handles all of the inter-node communication, and the intra-node CPU-GPU task parallelism, while our CUDA kernel implementations leverage both SIMD and ILP with intelligent thread and block layouts. By analyzing the application bottlenecks and tuning the data and task granularity statically, we are able to ensure that the downstream image analysis tasks are not starved for work when analyzing very large data sets.

Chapter 3 presents an investigation of the important parameters to consider for radar signal analysis applications on distributed, heterogeneous platforms. The particular application used as a case-study is synthetic aperture radar (SAR) imaging, which is a compute-intensive operation. In this chapter, we only make use of the GPUs for the computation, since the GPU implementation is much more efficient than the CPU implementation, and there is no task parallelism to speak of in the application. However, we show that there can be drastic differences in the performance of distributed applications due to different work-space partitioning decisions.

In Part II, we focus on middleware approaches to help developers leverage high-performance supercomputers with accelerator architectures to their fullest extent. Chapter 4 presents a design of light-weight dataflow middleware for accelerators, called DataCutter-Lite, and an initial implementation for the Cell Broadband Engine. Accelerator architectures, due to their design decisions to be higher-performing for certain styles of computation than standard CPUs, are often much more difficult to program to perform efficiently. Thus, we show that by using a fine-grained component-based dataflow programming paradigm, we can efficiently leverage all of the power inherent in accelerator devices, without forcing developers to be experts in parallel algorithms, multithreaded programming, architectural-specific constraints (which can be severe on accelerators), as well as their own research domain.

In Chapter 5, we present an adaptive technique for tuning the databuffer size for large-scale dataflow applications and a load-balancing technique based on modeling heterogeneous dataflow task processing rates and automatic workspace partitioning. Most distributed programming paradigms leave to the developer the task of finding the right data decomposition granularity for their application. Further, the same application can have more than one optimal databuffer size, since the system configuration (including processor types, network topology, system size) can affect the choice. By using a simple programming interface, developers can write dataflow applications that are automatically tuned for the best performance, where the load is nearly perfectly balanced across all of the available computational resources. We experimentally validate the effectiveness of our approach on two real applications and two synthetic applications.

Chapter 6 presents a significant extension of our adaptive technique to improve scalability to much larger system configurations. We introduce a new work-stealing layer which handles all of the inter-node work distribution. Also, we present a new storage layer which intelligently manages the application-specific input and output data, which is important for making efficient use of the network, particularly in data-intensive applications. We compare against three state-of-the-art distributed programming frameworks, and our technique shows quite favorable performance.

Chapter 7 presents preliminary research into efficiently supporting alternative programming models for applications or users which do not need the full breadth of component-based dataflow programming. In particular, we look at supporting the popular, simple programming framework MapReduce in our adaptive dataflow runtime middleware. By leveraging the efficiency brought in by our middleware, we are able to effectively use distributed resources, while reducing the programming effort.

Part I: Ad-Hoc High-Performance Dataflow Application Programming for Heterogeneous Systems

Chapter 2: Biomedical Image Analysis on a Cooperative Cluster of GPUs and Multicores

2.1 Introduction

Biomedical applications are becoming a major research focus due to the large possible benefit to the public welfare as well as to the scientific community. In particular, imaging applications are emerging as a new opportunity for innovation at the meeting point between medicine and computer science. These applications are challenging for several reasons. The practical concerns of providing sufficient solution quality and timeliness, and the performance-centric difficulties of using disruptive architectures for maximum application throughput provide many opportunities for novel solutions to complex problems.

The steady increase in the computational power of modern computational resources has made a tremendous impact on medical imaging technology. The recent availability of whole slide digital scanners has made research on pathological image analysis possible, by enabling quantitative analysis tools to decrease the evaluation time pathologists spend for each slide. The same research on biomedical image analysis also attempts to reduce the variation in decision-making processes among different pathologists or institutions.

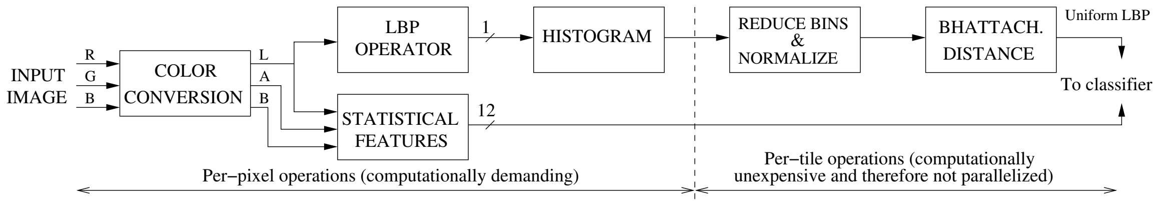


Figure 2.1: Flow chart for the stroma classification algorithm.

The analysis of pathology images is particularly challenging due to the large size of the data. Since uncompressed image sizes can be 30 gigabytes for one slice of tissue, typical datasets for case studies can easily stretch to terabyte scale. Further, the computation required to analyze these images can be extensive; analysis can often take hours on a single CPU. Several research studies on different cancer types have been conducted to develop computational methods within this field [37, 52, 69, 70, 86, 96, 103]. Most of these approaches only tested randomly selected image tiles, while some [85] have recently extended their methods to processing whole-slide images, but without discussing the computational burden. One of the most recent results [20] involves a parallelization technique, but the focus of the study was not strictly one of execution time performance, and processing a relatively small image took almost half an hour on a 16-node configuration, which is still impractical for clinical application.

With respect to improving the processing time of scientific applications, the newest versions of programmable Graphics Processing Units (GPU) provide an ideal platform, since they allow extremely high floating point arithmetic throughput for applications which fit their architectural idiosyncrasies [81]. This fact has attracted many researchers and encouraged the use of GPUs in many fields [49] including data mining [51], image segmentation and clustering [53], numerical methods for finite element computations used in 3D interactive simulations [112], and nuclear, gas dispersion and heat shimmering simulations [118]. In an earlier work [93, 92], we have also leveraged GPU processing power and introduced techniques based on shaders and Cg to accelerate the feature extraction by a factor of 321x versus a Matlab version and 45x versus an equivalent C++ version running on a single CPU.

GPU manufacturers have responded to the wide acceptance and use of GPUs in general-purpose computing with the introduction of CUDA (Compute Unified Device Architecture) [26], a more general programming interface, and Tesla [107], a new computational node with multiple GPUs, reaching near supercomputer performance levels starting at $1500. Recent announcements from Nvidia (GeForce 9 Series) and ATI (FireStream) [41] have also responded to the scientific computing community's widespread call for double-precision, floating-point arithmetic which does not incur a large performance penalty compared to single-precision.

In this work, our goal is the efficient execution of large-scale biomedical image analysis applications on a cooperative cluster of GPUs and multi-core CPUs. The advent of multi-core CPUs means that more computation can occur on a single computer than ever before. Traditional SMPs are now becoming hybridized, such that multiple multi-core CPUs are resident in a single compute node. Furthermore, new architectures can support more than one GPU card per node. Hence, our target hardware architecture is a cluster of compute nodes with multiple multi-core CPUs and multiple GPUs, and this work presents methods to make full use of all of the computational power such a hardware system offers.

On the software side, our cooperative approach is enabled by software libraries and middleware which ease both the GPU programming and the parallelization computation at many granularities. Nvidia's CUDA provides easier access to the high computational performance available in modern era GPUs. Additionally, it provides capabilities beyond that of other programming methods with respect to applications which do not entirely fit into the more traditional graphics processing paradigm. DataCutter [13] is a powerful middleware tool for data-parallel application decomposition, transparent task replication, and task graph execution. Its use here allows us to leverage all of the parallelism inherent in the hardware architecture. By analyzing the application bottlenecks appropriately, we are able to hide much of the latency incurred by the hardware when analyzing very large data sets. Additionally, DataCutter enables the overlap of computation and communication, which are still fundamental issues in the GPU and multi-core era.

The rest of the chapter is organized as follows. Section 2.2 presents the neuroblastoma image analysis application which provides the testbed for this new paradigm of cooperation between multi-core CPUs and GPUs. The specifications and properties of an example state-of-the-art, multi-socket, multi-core, multi-GPU cluster are presented in Section 2.3. Section 2.4 focuses on the specifics of the GPU programming with CUDA and that of the parallelization strategies. The experimental results are presented in section 2.5, and section 2.6 summarizes this chapter's results.

2.2 Pathological Image Analysis

Neuroblastoma is a cancer of the sympathetic nervous system which mostly affects children. The diagnosis of the disease is currently based on visual examination under a microscope of tissue slides by expert pathologists. These tissue slides are classified into different groups depending on the differentiation grade of the neuroblasts, among other issues [98]. Manual examination by pathologists is an error-prone and very time consuming process, and may lead to inter- and intra-reader variations. Therefore, together with our collaborators, we are developing a computerized pathological image analysis system [20, 52, 93, 92, 96] to assist in the determination of the classification of the neuroblastoma tumor type by automatically characterizing stroma regions.

2.2.1 The algorithm

Figure 2.1 shows the flowchart of the image analysis algorithm for the classification of stromal development in neuroblastoma images [93, 92]. The image analysis occurs in four stages:

Phase 1: Color conversion. The RGB input image is converted into the LA*B* color space, which provides color perceptual uniformity and enables the use of Euclidean distance in feature calculation [84].

Phase 2: Statistical features. Four second-order statistical features are extracted from each color channel: contrast, correlation, homogeneity and energy [110]. An intermediate data structure used during the calculation of those twelve features is the co-occurrence matrix [30], which represents how often a pixel with the intensity value i occurs in a specific spatial relationship to a pixel with the intensity value j (see Figure 2.2). The size of the co-occurrence matrix has a major impact on the workload, but only a marginal influence on the algorithm's classification accuracy [92]. Therefore we have selected a $4{\times}4$ co-occurrence matrix for our experiments, which yields the fastest execution times.

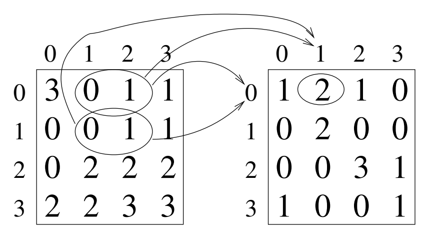


Figure 2.2: The computation of a co-occurrence matrix (right) from a 4x4 image (left) where pixel intensities are shown.

Phase 3: LBP operator. The local binary pattern feature (LBP) is extracted from the L (luminance) channel to become the thirteenth feature for texture analysis. Widely used in many applications such as facial expression recognition and content based image retrieval [104], LBP is a rotationally invariant operator defined within a 3×3 neighborhood of each pixel, where the eight neighbors are examined to see if their intensity is greater than that of center pixel p. The results form a binary number $b_1b_2b_3b_4b_5b_6b_7b_8$ where $b_i = 0$ if the intensity of the ith neighbor is less than or equal to p and 1 otherwise (see Figure 2.3).

Phase 4: Histogram. Since LBP feature values are in the range $0 \rightarrow 255$, a histogram is constructed with 256 bins for the entire image, with each bin accumulating the number of pixels which have that LBP feature value. These bins are then reduced into ten canonical classes and normalized between 0 and 1 to become the components of a ten-dimensional vector. The Bhattacharyya distance [91] is then measured between the LBP feature vector and a $(1, 1, 1, 1, 1, 1, 1, 1, 1, 1)$ vector to constitute the uniform LBP feature value used for the stroma classification. The subsequent classifier is a computationally inexpensive process, so we will not mention it further.

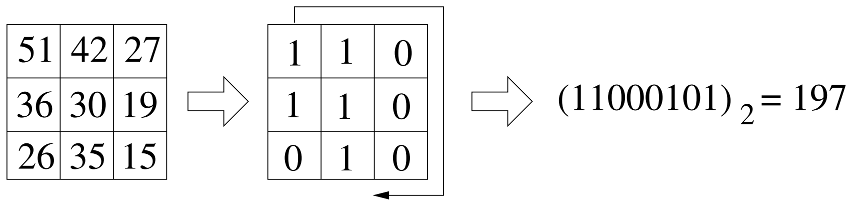


Figure 2.3: The LBP operator on a 3x3 grid.

2.2.2 Major challenges

Input tissue samples are digitized at 40x magnification and stored in TIFF files using JPEG compression and the RGB color format. Each whole-slide image has a resolution of 110K×80K pixels in the most demanding case, which poses two major challenges:

- A single uncompressed image's size is well over the memory capacity of a single GPU. Additionally, providing balanced parallelization and effective use of cluster resources like disk I/O requires a smaller data granularity. In order to relieve memory usage and allow for an efficient parallelization of the computation across nodes, we decompose the image into 1K x 1K tiles.
- The algorithm takes several hours to run on a single CPU (see Table 2.5). This motivates us to study alternative platforms for its parallel execution. Our selection of multi-socket and multi-core CPUs combined with high-end GPUs will give us the opportunity for assessing scalability on both sides.

Tiling and parallelism are in fact tightly coupled, since the first strategy favors the latter. On a multi-processor system, each processor may independently perform the image analysis task on a subset of tiles and return a label indicating whether the particular image tile is stroma rich or stroma poor. Finally, labels are gathered on a central host and the classification map for the whole-slide image is constructed.

Apart from its implementation on a multi-processor system, our particular image analysis application is of great interest for evaluating the memory hierarchy and computational power of graphics architectures, because it meets a diverse number of features. For example, color conversion is a typical streaming operation with no data reuse. The LBP operator, on the other hand, exhibits a large degree of data reuse and locality, which typically favors more traditional, cache-based systems. Statistical features (through a co-occurrence matrix) and histogram construction both use extensive indirect array accesses such as those characterizing irregular computing and reduction operations often found in linear algebra. These two phases present undesirable features for both the CPU and the GPU. In our earlier work, we have investigated these in the context of Cg programming [92], while here we look at the trade-offs in the context of CUDA and a parallel implementation using cooperative cluster of multi-core, multi-socket CPUs and multiple GPUs.

2.3 GPU Cluster Testbed

In this section we will present the specifications and properties of a typical, state-of-the-art, multi-socket, multi-core, multi-GPU cluster. For this purpose we picked the Ohio Supercomputer Center's BALE cluster [7]. The BALE cluster is comprised of a total of 71 Linux nodes, but our main focus is the newly added 16 visualization nodes, equipped with two dual-core AMD Opteron 2218 CPUs and two Nvidia Quadro FX 5600 GPUs each. The interconnection network for the BALE cluster is Infiniband. Figure 2.4 illustrates the architecture of the visualization nodes, and Table 2.1 shows

Table 2.1: Hardware features of the CPU and GPU used on each node of the testbed cluster. The GFLOPS listed are for 32-bit, single-precision floating-point arithmetic.

Hardware feature	CPU from AMD	GPU from Nvidia
Board	ASUS KFN32-D SLI	Quadro FX 5600
Processor model	Opteron X2 2218	G80
Processor speed	2.6 GHz	600/1350 MHz
Number of sockets	2	2
Number of cores	2	128
Peak processing	2 x 8.8 GFLOPS	2 x 330 GFLOPS
Memory size	8 GB	2 x 1.5 GB
Memory bus width	2 x 64 bits	2 x 384 bits
Memory speed	667 MHz	1600 MHz
Memory bandwidth	2 x 10.8 GB/s.	2 x 76.8 GB/s

the specifications of the CPU and GPU processors. Below we will go over some important specifications and features of the CPUs and GPUs used in our system.

On the CPU side, each node consists of two Opteron X2 2218 chips, which are dual-core processors running at 2.6 GHz. Each core in the system has a pair of 64 KB 2-way set associative L1 caches for holding data and instructions, and a 1 MB 4-way set associative L2 cache which is not shared among cores but are cache coherent. Each socket includes its own dual-channel DDR2-667 memory controller as well as a single HyperTransport link to access the other socket's cache and memory. Each socket can thus deliver 10.6 GB/s for an aggregate memory bandwidth of 21.3 GB/s to each node's 8 GB of 667 MHz DDR2. The peak double-precision performance is 4.4 GFLOPS per core, 8.8 GFLOPS per socket, 17.6 GFLOPS per node. The peak single-precision performance per node is 35.2 GFLOPS, providing an aggregate single-precision performance of 563.2 GFLOPS for the 16 visualization nodes.

On the GPU side, each node has two Nvidia Quadro FX 5600 GPUs based on the G80 architecture. From a graphics viewpoint, the G80 can be seen as a 4-stage

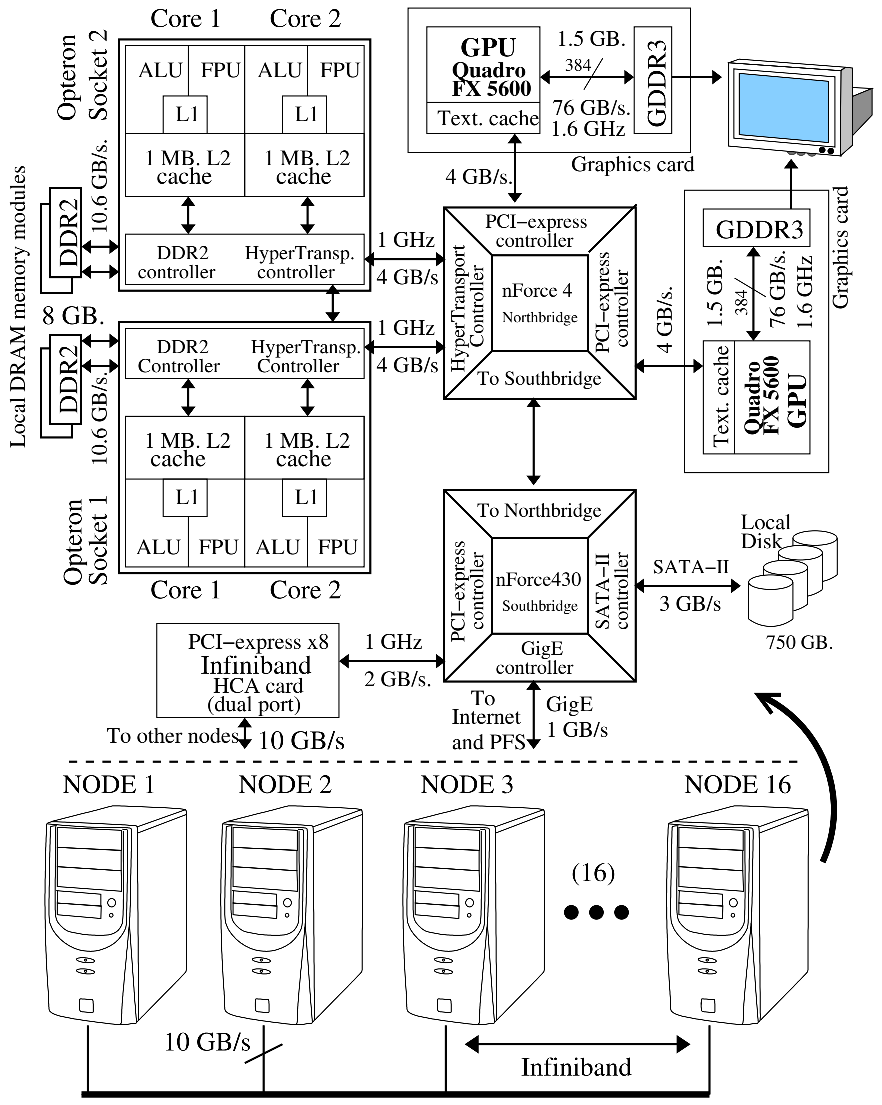


Figure 2.4: The BALE supercomputer at a glance [55].

graphics pipeline for shading, texturing, rasterizing and coloring. As a parallel architecture, however, the G80 becomes a SIMD processor endowed with 128 cores, and CUDA is the programming interface to use it for general purpose computing. From the CUDA perspective, cores are organized into 16 multi-processors, each having a set of 32-bit registers, constants and texture caches, and a 16 KB memory shared by the eight cores inside each multi-processor. In any given cycle, each core executes the same instruction on different data, and communication between multi-processors is performed through global memory (see Figure 2.6).

2.4 Tools and GPU implementation

In order to give an example of the cooperative relationship between GPUs and multi-core CPUs, this section shows the details of the CUDA [26] implementation of the image analysis algorithm as well as the multi-node, multi-core and multi-GPU parallelization using DataCutter middleware [13]. We have used Matlab [20], C++ and Cg implementations [92] as a baseline for our performance comparisons. Programming tools and paradigms are summarized in Figure 2.5.

2.4.1 CUDA

The CUDA programming interface consists of a set of C language library functions, and the CUDA-specific compiler generates the executable code for the GPU. Since CUDA is designed for generic computing, it does not suffer from excessive constraints when accessing memory (as Cg [21] does), but memory access times do vary for different memory types.

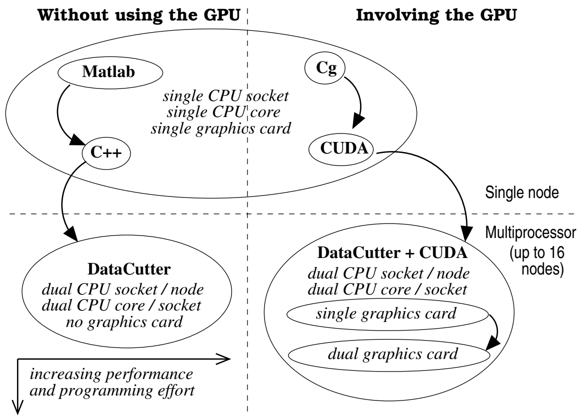


Figure 2.5: The programming approaches and their effect on performance, both single node and parallel.

Computation Paradigm

CUDA exposes a model to the developer consisting of a collection of threads running in parallel. There are several other elements involved in the execution which bear mentioning, since they are unique to graphics processor programming with CUDA:

- A program is decomposed into **blocks** that run *logically* in parallel (physically only if there are resources available). A block is a group of threads that is mapped to a single multi-processor, where they can share 16 Kbytes of memory (see Figure 2.6). All threads of concurrent blocks on a single multi-processor divide the resources available equally amongst themselves. The data is also divided amongst all of the threads in a SIMD fashion with a decomposition explicitly managed by the developer.
- A **warp** is a collection of threads that can actually run concurrently (with no time sharing) on all of the multi-processors. The size of the warp (32 threads

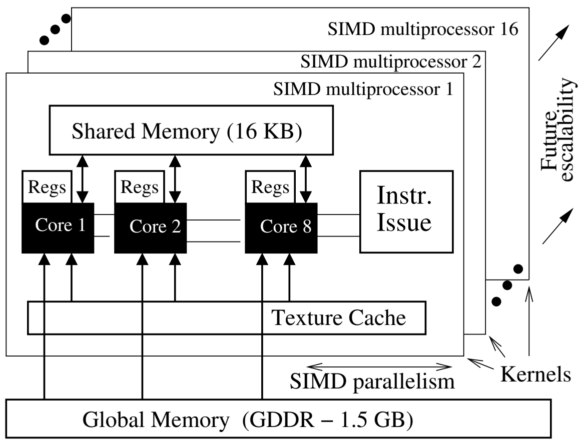


Figure 2.6: The CUDA hardware interface.

on the G80) is less than the total number of available cores (128 on the G80) due to memory access limitations. The developer has the freedom to determine the number of threads to be executed (up to a limit intrinsic to CUDA), but if there are more threads than the warp size, they are time-shared on the actual hardware resources.

- A **kernel** is the actual code to be executed by each thread; the executable is shared among all of the threads in the system. Conditional execution of different operations on each multi-processor can be achieved based on a unique thread ID.

The CUDA documentation states that a single block should contain 128-256 threads to maximize execution efficiency, with a CUDA-imposed maximum of 512. Other hardware and software limitations are listed in Table 2.2, where we have ranked them according to their impact on the developer's implementation and overall performance based on our own experience.

Table 2.2: Major hardware and software limitations with CUDA. Constraints are listed for the G80 GPU and categorized according to their difficulty of optimization and impact on the overall performance.

Parameter	Limitation	Impact
Multi-Processors per GPU	16	Low
Processors / Multi-Processor	8	Low
Threads / Warp	32	Low
Thread Blocks / Multi-Processor	8	Medium
Threads / Block	512	Medium
Threads / Multi-Processor	768	High
32-bit registers / Multi-Processor	8192	High
Shared Memory / Multi-Processor	16 Kbytes	High

Memory and registers

In the CUDA model, all of the threads can access all of the GPU memory, but, as expected, there is a performance boost when each thread reads data resident in the shared memory area, particularly when the data resides in several different memory banks (each bank can only support one memory access at a time, and therefore simultaneous accesses are serialized, hurting parallelism). The use of up to 16 KB of shared memory is explicit within a thread, which allows the developer to solve bank conflicts wisely. However, this type of optimization is often very difficult, but can also be very rewarding. Execution times may decrease by as much as 10x for vector operations and latency hiding may increase by up to 2.5x [39].

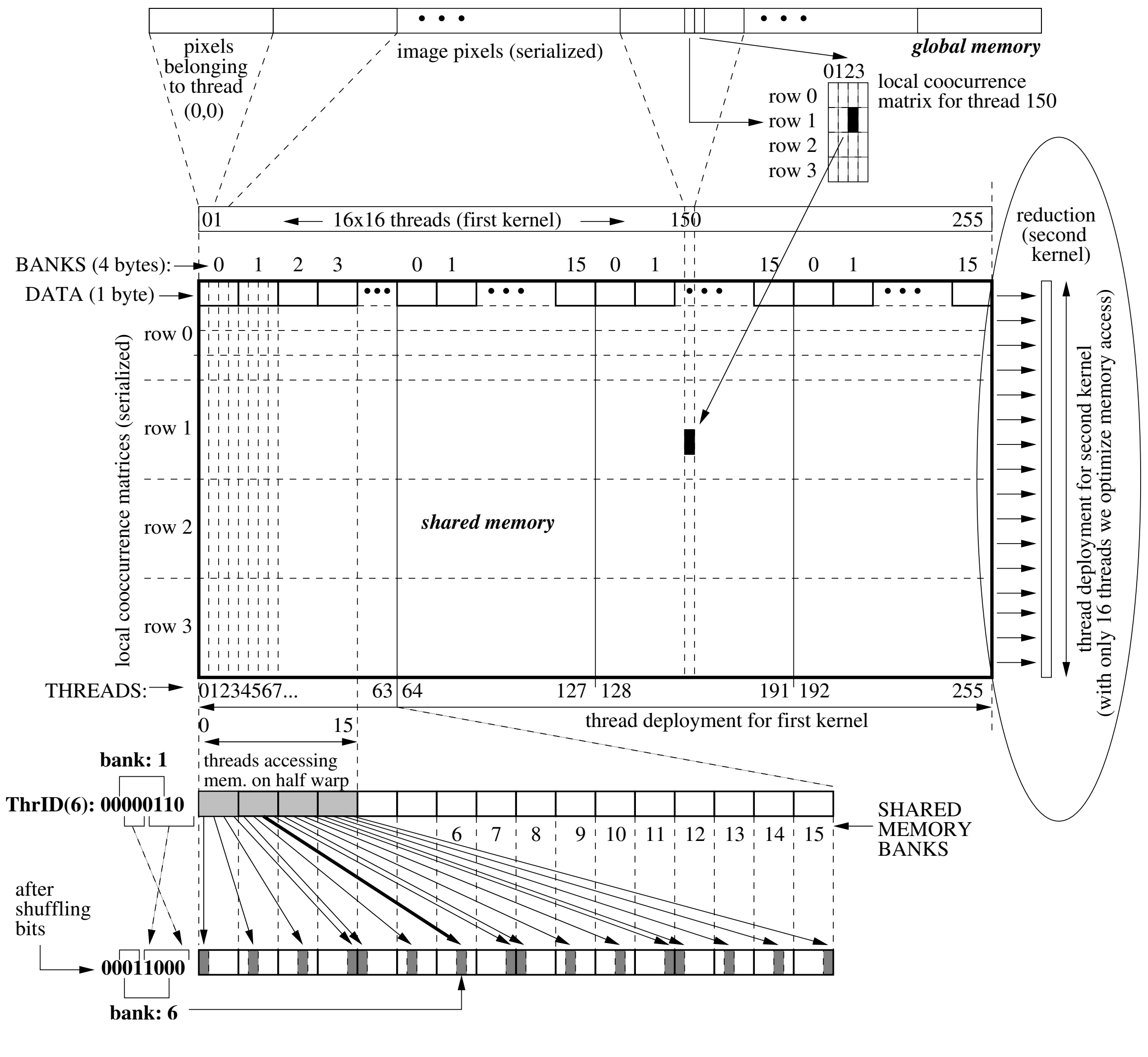
When developing applications for GPUs with CUDA, the management of registers becomes important as a limiting factor for the amount of parallelism we can exploit. Each multi-processor contains 8,192 registers which will be split evenly among all the threads of the blocks assigned to that multi-processor. Hence, the number of registers needed in the computation will affect the number of threads able to be executed simultaneously, given the constraints outlined in Table 2.2. For example, if a kernel (and therefore a thread) consumes 16 registers, only 512 threads can be assigned to a single multi-processor, and this can be achieved by using 1 block with 512 threads, 2 blocks of 256 threads, and so on.

Implementation of Image Analysis Code

We have used a typical CUDA development cycle, which we will describe briefly. First, the code was compiled using the CUDA compiler and a special flag that outputs the hardware resources (memory and registers) consumed by the kernel. Using these values, we were able to analytically determine the number of threads and blocks that were needed to use a multi-processor with maximum efficiency. If a satisfactory efficiency could not be achieved, the code would need revision to reduce the memory footprint.

Due to the high floating point computation performance of the GPU, memory access becomes the bottleneck in many parts of our application. The input image tile (1K×1K×3 bytes) is much larger than the size of the multi-processor shared memory (16 KB), so we prioritize data structures like partial co-occurrence matrices (used in phase 2) and partial histograms (phase 4). However, in phase 1, even though the tile pixels are swept over without being reused, the execution time was lower when using shared memory (2.32 ms versus 2.77 ms - see Table 2.3). Also, in phase 3, the input pixels were moved to shared memory because the calculation of the LBP feature shows high data reuse.

In order to illustrate the progression of a typical CUDA implementation, we will discuss the specific optimizations applied to each phase of the image analysis application below.



global memory
image pixels (serialized)
pixels belonging to thread (0,0)
0123
local coocurrence matrix for thread 150
row 0
row 1
row 2
row 3
01
16x16 threads (first kernel)
150
255
reduction (second kernel)
BANKS (4 bytes):
0 1 2 3 0 1 15 0 1 15 0 1 15
DATA (1 byte)
local cooccurrence matrices (serialized)
shared memory
thread deployment for second kernel (with only 16 threads we optimize memory access)
THREADS:
01234567... 63 64 127 128 191 192 255
thread deployment for first kernel
0 15
threads accessing mem. on half warp
bank: 1
ThrID(6): 00000110
SHARED MEMORY BANKS
6 7 8 9 10 11 12 13 14 15
after shuffling bits
00011000
bank: 6

Figure 2.7: Phase 2: Local co-occurrence matrices in CUDA. Assigning banks in shared memory to each thread to avoid conflicts when computing co-occurrence matrices.

Table 2.3: Major CUDA optimizations in the image analysis application. Execution times correspond to an isolated 1Kx1K tile on a single GPU. Phase 1 shows the times for CPU to GPU communication and actual GPU computation. Phases 2 and 3 show only GPU computation (no communication is required). Phase 4 shows the times for GPU to CPU communication and actual GPU computation. The fifth column in the table refers to conflicts arising when several threads simultaneously access the same bank of shared memory.

Analysis Phase	Code tag	Description / Optimizations	In shared memory	Conflicts solved	Execution time (millisecs.)		
					Comm.	Comput.	Total
1:	1.10	Baseline version: Using float3 per color channel	All in global memory		8.49	3.71	12.20
RGB to	1.11	Coalescing (Alpha channel inserted) on float3	All in global memory		10.79	2.44	13.23
LA*B*	1.12	Replacing float3 by uchar (256 threads/block)	All in global memory		2.98	2.77	5.75
color	1.20, 1.30	**Using shared memory (256 threads/block)**	**Pixels**	**Unneeded**	**2.98**	**2.32**	**5.30**
conversion	1.40	Using between 169 and 192 threads/block	Pixels	Unneeded	2.98	2.43	5.41
2:	2.10	Baseline version: Using global memory	All in global memory		None	15.40	15.40
Statistical	2.20	Using shared memory for co-occurrence matrices	Co-oc. mat.	No	None	4.48	4.48
features	2.30	**Solving conflicts on shared memory banks**	**Co-oc. mat.**	**Yes**	**None**	**2.58**	**2.58**
3:	3.10	Baseline version: Special threads on grid borders	All in global memory		None	2.29	2.29
LBP	3.20, 3.30	**Blocks of 16x16 threads, 14x14 computing**	**Pixels**	**Unneeded**	**None**	**1.82**	**1.82**
operator	3.40	Blocks of 8x8 threads, 6x6 of them computing	Pixels	Unneeded	None	2.31	2.31
4.	4.10	Baseline version: Using global memory	All in global memory		4.02	2.08	6.10
Histogram	4.20	Using shared memory for local histograms	Local hist.	No	0.31	0.61	0.92
	4.30	**Solving inter-warp conflicts in mem. banks**	**Local hist.**	**Inter-warp**	**0.31**	**0.59**	**0.90**
	Σ(*.10)	Baseline version	All in global memory		12.51	23.48	35.99
Total	Σ(*.20)	Involving shared memory	Enabled	No	3.29	9.23	12.52
	Σ(*.30)	**Optimal version**	**Enabled**	**Most**	**3.29**	**7.31**	**10.60**

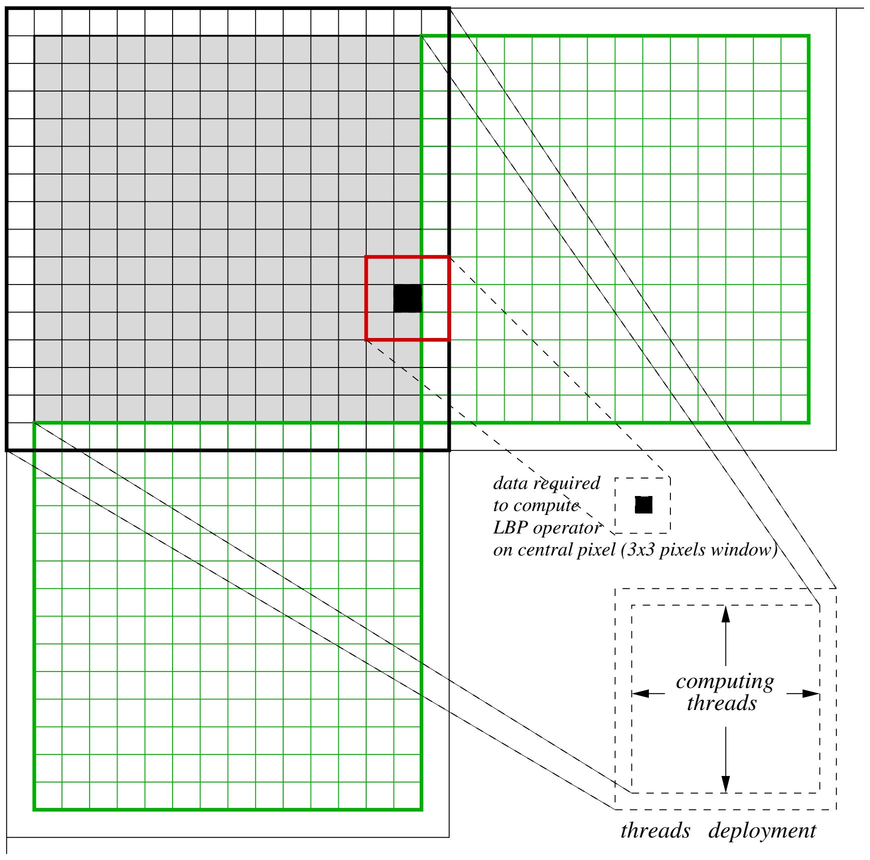


Figure 2.8: Phase 3: LBP operator in CUDA. Each thread block operates on an image tile which has a 1-pixel-wide border surrounding it, such that every pixel in the entire image (except the extreme edges) have LBP values calculated for them.

Phase 1: Color conversion As a departure point, we started using 24-bit `float3` data types for each color channel. However, extra performance can be obtained by padding the RGB input to match the expected data width of 32-bits, which simplifies all subsequent optimizations involving shared memory. This is called data *coalescing*, and for this phase it saved 35% of the computation time at the expense of communication time (see code 1.11 in Table 2.3). Next, it was found that 8-bit `uchar` data types were sufficient for the precision of the application. As expected, this reduces the communication time by nearly a factor of 4.

We then used the special CUDA compilation flag to find that the color conversion kernel requires 13 registers and 1064 bytes of shared memory, leading to a maximum processor occupancy of 75% when allocating between 176 and 192 threads. However, we chose to allocate 256 threads instead, trading processor occupancy (from 75% down to 67%) for better load balance, since 256 is a multiple of 32 (maximum threads per warp) and a divisor of 768 (maximum threads per multi-processor) and 1024 (maximum pixels per image). The result was that the execution times improved slightly (see code 1.30 versus 1.40 in Table 2.3 - version 1.40 reports the minimum time obtained for all threads/block cases between 169 and 192, which turns out to be 169 threads). Unfortunately, maximum performance here is limited because each thread needs more than 10 registers (11 to be precise), which, as discussed earlier, prevents us from reaching the maximum occupancy of 768 threads within a multiprocessor. The optimal execution time for this phase was 2.98 ms for pixel transfer and 2.32 ms for computing the color conversion, as reflected in Table 2.3.

Phase 2: Statistical features This kernel requires 9 registers and 4132 bytes of shared memory, which allowed us to allocate 3 parallel blocks of 256 threads. This perfect usage of all 768 threads filled all of the G80 hardware resources for a 100% occupancy factor. Pixels are equally distributed among threads and local co-occurrence matrices are simultaneously computed within them. Finally, partial results are accumulated through a reduction operator.

It was found to be challenging to compute co-occurrence matrices in shared memory while avoiding conflicts accessing its 16 banks. With a grid of 256 threads arranged in a 16x16 grid, the naive thread deployment would force the 32 threads in a warp to access only 8 shared-memory banks, which would severely limit parallelism and performance. We found that by intelligently shuffling the active threads combined into a warp, the local matrix computation and the subsequent global matrix aggregation operation can proceed without forcing threads to wait for bank access (see Figure 2.7). This complex optimization solves all conflicts when accessing memory banks, reducing the execution time to 2.58 ms from 4.48 ms. Without using shared memory, a straightforward implementation consumes 15.40 ms instead (see Table 2.3).

Phase 3: LBP operator The computation of the LBP feature entails a convolution with a 3x3 mask, followed by a binary to decimal conversion (see Figure 2.3). Each thread requires 10 registers and each block of threads uses 296 bytes of shared memory. Due to the memory usage characteristics, we were able to allocate 256 threads in a 16 x 16 grid to reach 100% occupancy on the G80. Each thread reads a pixel from global memory and stores it in a shared memory data structure. Those threads located on the border of the grid are unable to compute, since they do not have access to their neighbor data (see Figure 2.8); they exit the kernel at this stage. The LBP for the border regions will be computed by the next block of threads, since we overlap the thread layout by two rows and two columns of pixels each time. By using this strategy, we incur 23% idle cycles and the associated redundant memory accesses, but the computation is more homogeneous and the threads are more lightweight. As compared to the heterogeneous version, where there are no idle cycles and no memory redundancy, the homogeneous version is faster by 1.25x, leading to the lowest execution time of 1.82 ms.

Since the LBP kernel is a very regular computation, we can select different sizes of thread deployments, provided a square grid is used. Therefore, we investigated the effect of different thread/block ratios. We gathered values for grids of threads of sizes 20x20 (2.02 ms), 18x18 (1.90 ms), 16x16 (1.82 ms - optimal), 14x14 (1.89 ms), 12x12 (2.00 ms), 10x10 (2.16 ms), and 8x8 (2.31 ms - reported in Table 2.3 as version 3.40). A grid of 16x16 threads is a popular choice among expert programmers to maximize performance in CUDA; that said, our intention was more to quantify the penalty we may incur by making an inappropriate choice for thread allocation.

Phase 4: Histogram The implementation of this phase is based on a histogram kernel included with the CUDA library [89], where the global reduction is delegated to the CPU. We deemed this an appropriate solution, since the histogram is only computed once per tile, and does not incur a large execution time cost. The execution time for this last phase is 0.9 ms.

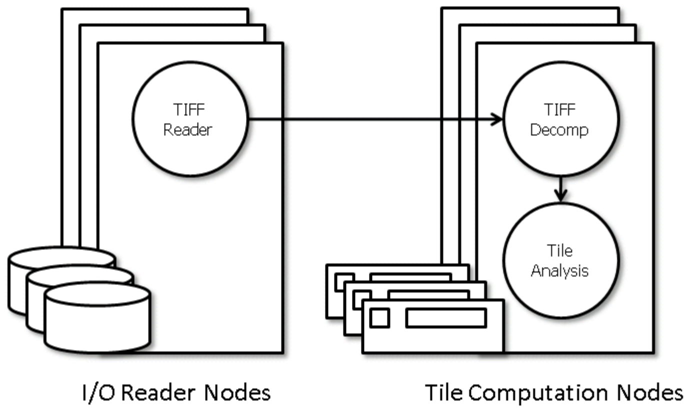


Figure 2.9: DataCutter layout of the image analysis application.

Table 2.3 summarizes all of the major optimizations performed within CUDA on each phase along with the execution times obtained. Since the current CUDA programming model does not allow the developer to assign different kernels to different multi-processors, the four phases are sequentially executed and the whole GPU is used during each phase independently. Overall, by using CUDA to implement the image analysis, we were able to reduce the execution time by a factor of 3-5 over the times obtained by Cg (see Table 2.5 for times on an entire image), by an additional factor of 3 when enabling shared memory, and by an extra 20% when solving memory conflicts as much as possible for an optimal execution.

2.4.2 Implementation with DataCutter

We employed DataCutter middleware for the parallelization of the testbed image analysis computation, which is presented in Section 1.2.1 in detail.

Our image analysis application is easily divided into three stages, each of which is implemented as a single DataCutter filter type. These are: a *TIFF-Reader* filter that reads binary TIFF tiles, a *TIFF-Decompressor* filter that decompresses TIFF tiles and produces RGB images, and a *Tile-Analysis* filter where the real image analysis computation is carried out. A layout which includes one each of these three filter types constitutes a complete image analysis task graph, while multiple copies of each filter type allows for quick parallelization and efficient execution. Figure 2.9 shows the general layout of the system. One or more reader nodes (with one *TIFF-Reader* filter each) read the binary TIFF image tiles from the disk and write them to the stream leading to the *TIFF-Decompressor*. One or more *TIFF-Decompresser* filters are able to coexist on the same node and simply take the binary TIFF file and decompress it into the appropriate RGBA tuples. This decompressed image data is then written to the input stream to the *Image-Analysis* filter on the same node. When the *Tile-Analysis* filter makes use of the GPU, the ability for multiple *Tiff-Decompressor* filters to feed data to the GPU allows for full utilization of the GPU's throughput. Placing both the *TIFF-Decompressor* and the *Image-Analysis* filters on the same node has the benefit of saving memory bandwidth by leaving the largest representation of each TIFF tile in place and simply transferring the pointer.

The component-based programming model of DataCutter allows us to easily develop and deploy image analysis applications that utilize GPUs as co-processors. By simply replacing the C++ *Tile-Analysis* filters with filters using the GPU, we can quickly develop a parallel code that efficiently executes on a multi-socket, multi-core, multi-GPU cluster. The layout, the two *TIFF-Reader* and *TIFF-Decompressor* filters, and the entire parallelization system are reusable.

Table 2.4: Properties of the three slides used in our experiments.

Name	Resolution in pixels	Number of 1Kx1K tiles
`SMALL`	32,980 x 66,426	33 x 65 = 2,145
`MEDIUM`	76,543 x 63,024	75 x 62 = 4,659
`LARGE`	109,110 x 80,828	107 x 79 = 8,453

Table 2.5: Execution times for the image analysis application
Execution times for the image analysis application for different programming methods and hardware platforms. These times do not include disk I/O or decompression overheads. The image tile size is 1Kx1K pixels. The co-occurrence matrix size is 4x4. The values in the CUDA column represent running on one and two GPUs.

	On the CPU		On the GPU	
Image size	Matlab	C++	Cg	CUDA
`SMALL`	2h 57’ 29”	43’ 40”	1’ 02”	27’ 14”
`MEDIUM`	6h 25’ 45”	1h 34’ 51”	2’ 08”	0’ 58”
`LARGE`	11h 39’ 28”	2h 51’ 23”	3’ 54”	1’ 48”

2.5 Experimental Results

Our experiments were performed on the BALE cluster (see Section 2.3) using the 16 visualization nodes and an additional six compute nodes as reader nodes, in order to provide enough disk bandwidth to avoid the disk I/O bottleneck. Further, these reader nodes had their system file caches preloaded with several discarded experiments to ensure extremely high I/O from the upstream, *TIFF-READER* filters. We feel this is a suitable experimental setup to use; any production cluster designed to analyze this kind of image data with very high performance with multiple GPUs and a fast interconnect can reasonably be said to have parallel disks providing high I/O bandwidth.

In our experiments, we have used three different digitized pathology images. Table 2.4 summarizes their features.

The first set of experiments shows the single node CPU and GPU performance for the various implementations of the image analysis algorithm. Figure 2.10 shows the execution time and overhead time of each implementation when analyzing the `SMALL` image. The first four stacked bars represent CPU-only implementations, while the last six stacked bars bring one or two GPUs into the fold. Those bars with labels beginning with 'DC' are results from those implementations using DataCutter and those without the 'DC' label are the basic, serial implementations.

The execution time in Figure 2.10 (shown by the lower, darker portion of each bar) is due solely to the actual image analysis, while the overhead (shown by the upper, lighter portion of each bar) is caused by disk I/O, TIFF decompression, remote process invocation, and network latencies, where applicable. In this experiment, the most important thing to note is the two to three orders of magnitude of performance speedup when moving from the CPU-based solutions to the GPU-based solutions. While a large amount of performance is gained by moving away from Matlab, the C++ implementations are still far slower than those which are GPU-based. Additionally, the DataCutter versions of the GPU-based image analysis algorithms are able to shorten the execution time for the entire image versus the non-DataCutter versions, since the decoupled, multi-threaded nature of DataCutter allows the image analysis to overlap with the TIFF tile decompression and the disk I/O. Unfortunately, the CUDA implementation of the image analysis algorithm is fast enough to cause the

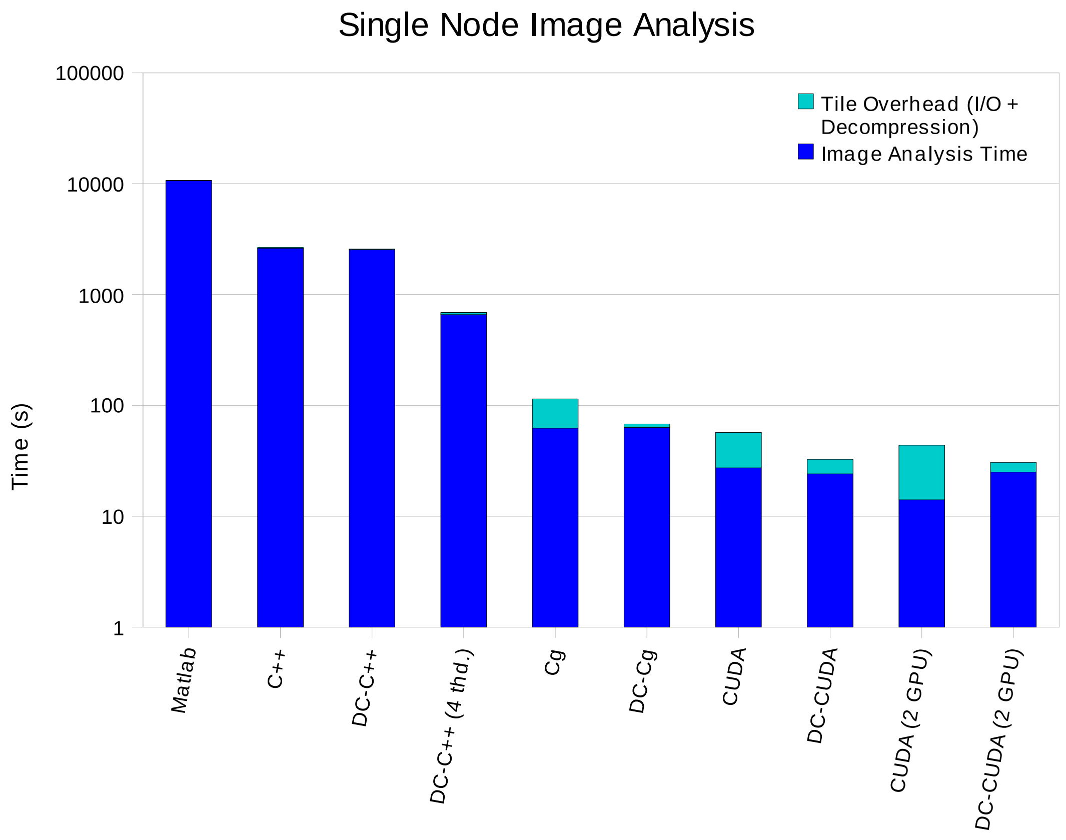


Figure 2.10: Execution time comparison of all implementations of the image analysis codes running on a single node using `SMALL` image.

TIFF tile decompression stage to become a bottleneck when 2 GPUs are used. This stalling, incurred by the GPUs waiting for tiles to analyze, prevents both GPUs from being fully utilized. Since four C++ threads shows a clear advantage over running a single thread, all future C++ results will be comprised of the DataCutter version with four tile analysis threads per node.

Figure 2.11 shows the performance comparison of the GPU-based implementations of the analysis routine for our three images. The main point to take from this chart is that there is a linear relationship between the execution time and the overall size

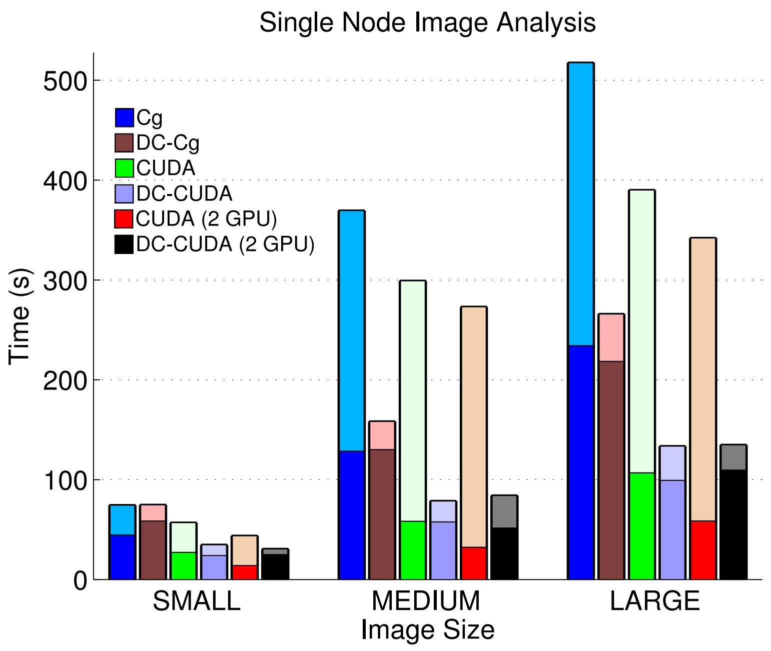


Figure 2.11: Execution time comparison of GPU and DataCutter implementations running on a single node using all three input images.

of the image under analysis in all of the implementations. Unfortunately, even when analyzing large images, making full use of 2 GPUs is hindered by the associated overheads; this being the case, we will not show 2 GPU results for the remainder of the experiments.

Figure 2.12 shows the scalability of our solution with respect to the number of nodes. The numbers of nodes involved in the image analysis are $1, 2, 4, 8, 12, 16$, and are represented by the bars in the figure from left to right, six in each color group. As in Figure 2.11, the lower, darker-colored portion of each bar represents the image analysis time, while the upper, lighter-colored portion of each bar shows the aggregated overhead. This type of image analysis computation scales extremely well, resulting in image analysis execution times which decrease nearly linearly with

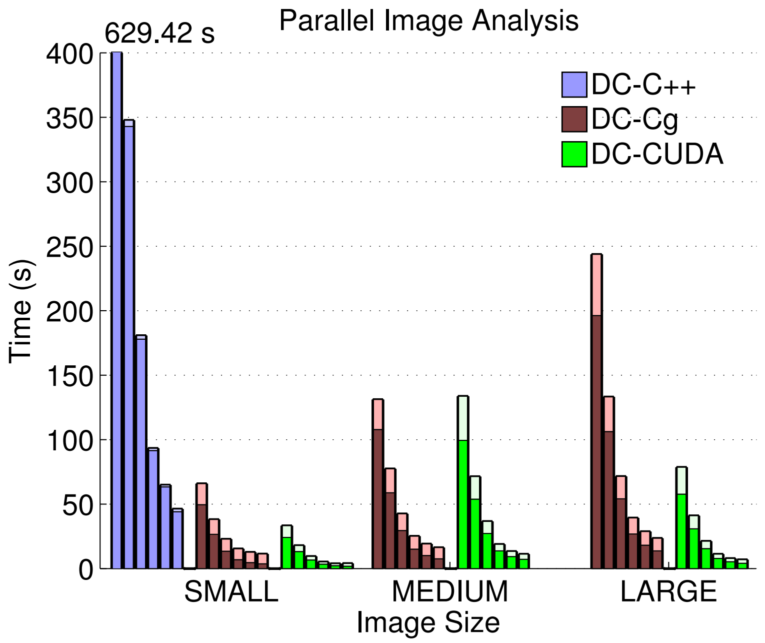


Figure 2.12: Parallel execution times of C++, Cg, and CUDA based DataCutter implementations using three input images while varying the number of nodes from 1 to 16.

the number of nodes. Further, the total analysis times for the DataCutter/CUDA implementation are under four seconds for the `SMALL` image, under seven seconds for the `MEDIUM` image, and just over eleven seconds for the `LARGE` image, when running on sixteen nodes. Compared with the single node CPU Matlab computation time of nearly three hours for the `SMALL` image and almost twelve hours for the `LARGE` image, this represents a tangible benefit of increased productivity. In the interest of increased chart legibility, we have cropped the single node C++ result. Its values are 629.42 seconds of image analysis time and 29.7 seconds of overhead. Additionally, since the main focus of this chapter is the GPU results, and since the C++ result is shown to scale well in the worst-case (because it incurs the lowest proportional overhead), we

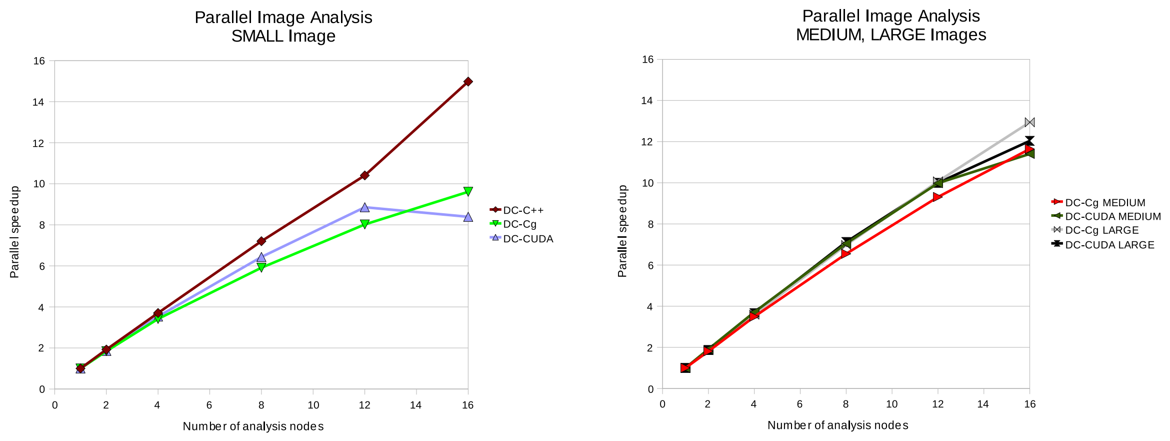


Figure 2.13: Parallel speedup results. Within DataCutter (DC) versions, we compare on the left a CPU-only case with two GPU-assisted ones for the `SMALL` image. On the right, we contrast the two GPU versions for the `MEDIUM` and `LARGE` images.

have chosen to remove it from the figures showing results for the `MEDIUM` and `LARGE` images.

Figure 2.13 shows the parallel speedup of the execution time versus the number of nodes. As seen in the figures, there is nearly linear speedup since the tiles are able to be decompressed and processed entirely independently of each other. However, due to the small execution times in the GPU-based implementations, the various overheads (comprised of remote process startup, network, and TIFF decompression latencies) begin to become comparable in overall time to the total time spent per node processing the image tiles. For instance, on sixteen nodes, the CUDA implementation requires at most 1.80 seconds of computation to compute $2,145$ tiles. However, despite concurrently running three *TIFF-Decompressor* filters, the decompression time alone for each node's allotment of 135 tiles could range from 0.3 seconds to 1.3 seconds.

Nearly linear speedup could be achieved in a production environment, however, since it is reasonable to assume that remote process invocation would only occur once for many images which are to be analyzed. Under these server-like circumstances, only the I/O system and network latencies would comprise the system overheads.

2.6 Summary

In this chapter, we have presented design trade-offs and a performance evaluation of a sample biomedical image analysis application running on a cooperative cluster of CPUs and GPUs.

By implementing algorithms on GPUs using CUDA and using DataCutter to parallelize the computation within and across nodes, we establish a solid heterogeneous and cooperative multi-processor platform where all the granularities of parallelism inherent in the architecture and in the application are fully exploited: multi-node (using DataCutter for data partitioning across nodes), SMP and thread-level (using DataCutter to fully utilize the available on-node and on-chip hardware resources), SIMD (using CUDA to fully populate the 128 stream processors of the GPU with work), and finally, ILP (Instruction Level Parallelism, by setting up blocks of computational threads within the GPU execution).

Our experimental results show great success for our techniques, first by decreasing the execution time on a single CPU/GPU node by using different intra-node optimizations, and then extending those performance gains to inter-node parallelism for a scalable multi-processor execution. When analyzing the largest test image and including overheads, on the 16 node cluster configuration, the single GPU DataCutter-CUDA implementation is 31.3 times faster than the serial CUDA implementation.

By using two GPUs per node, the single-node time to process the image is under one minute, if you ignore the overheads associated with disk I/O and tile decompression, proving that the CUDA method is extremely powerful. Additionally, the use of DataCutter to overlap the computation with disk I/O and tile decompression helps the GPU stay as busy as possible. This results in up to 12.94 speedup on 16 nodes using GPU-based DataCutter implementations.

GPUs are highly scalable and are evolving towards general-purpose architectures [49]; we envision biomedical image processing as one of the most exciting fields able to benefit from the use of GPUs. Additionally, new tools like CUDA [26] may assist non-computer scientists with a more friendly interface for adapting biomedical applications to GPUs. This computational power may then be combined with DataCutter to parallelize the computation across clusters of GPUs as outlined in this chapter to provide real-time response to clinicians of all types.

Chapter 3: Investigating the Use of GPU-Accelerated Nodes for SAR Image Formation

3.1 Introduction

Due to the rapid growth of the computational capacity of Graphics Processing Units (GPUs) over the past decade, researchers are increasingly using these emerging architectures to accelerate high performance applications. In fields such as data mining [51], image segmentation and clustering [53], numerical methods for finite element computations used in 3D interactive simulations [112], nuclear, gas dispersion and heat shimmering simulations [118], and biomedical imaging [55], GPUs have been used to speed up operations which are time-consuming on standard processors, dramatically affecting the overall execution times of the final applications.

Synthetic Aperture Radar (SAR) is a computationally intensive technique which can be used for, among other things, creating 2-D and 3-D images from radar signals gathered by a moving platform such as an aircraft. By combining signals gathered from multiple points in space (multiple angles of azimuth and elevation), higher resolution images can be constructed without needing a larger physical antenna or radar array. The computational burden increases with the image size and the amount of input, and so techniques for accelerating the processing of the input radar signals and generating the output images are necessary to be able to process the large amount of data in real time. Even if real time processing is not the goal, the sheer volume of data which SAR platforms can gather necessitates fast processing to enable fast decision making.

This chapter investigates the use of a cluster of GPU-equipped processing nodes to perform SAR image formation by backprojection. We discuss the particulars of the backprojection algorithm and briefly present an overview of computed tomography in Section 3.2. We present the software technologies used to implement the algorithm on the target system in Section 3.3, and the algorithm design space and parallelization decisions in Section 3.4. We finish by presenting our experimental results in Section 3.5, summarizing our findings, and discussing some future work.

3.2 Overview of Computed Tomography

In this section, we provide an introduction to tomographic imaging, the principle behind x-ray computer-aided tomography (CAT), magnetic resonance imaging (MRI), and synthetic aperture radar (SAR) imaging. A tomographic system involves a sensor capable of taking one-dimensional line projections through a two- or three-dimensional scene, and then reconstructing this underlying scene from a collection of line projections taken from different aspect angles.

The mathematics of line projections are given by the Radon transform. The specific way that this transform enters each of these modalities is slightly different, due to differences in their respective sensors, but the common element is this: the two-dimensional scene is collapsed into a one-dimensional projection by means of a dense set of line integrals penetrating the scene (in three dimensions, the projection

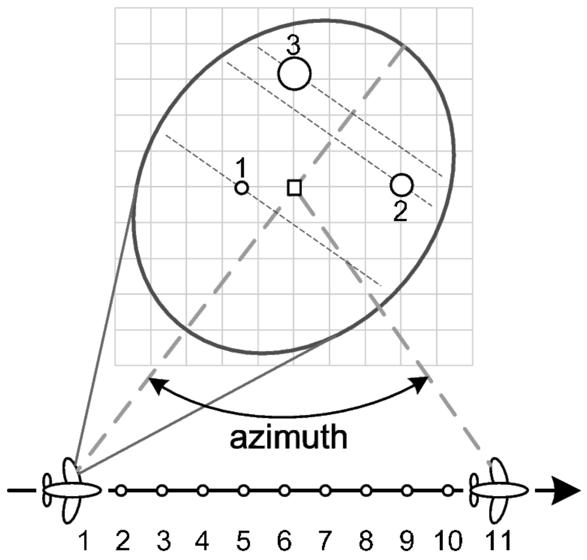


Figure 3.1: Schematic demonstrating operation of the tomographic principle Schematic demonstrating operation of the tomographic principle. The scene consists of the three targets of different amplitudes (circles), and produces the range profiles shown in Figure 3.2. Note that the flightpath may be circular or straight.

is obtained by a set of slice integrals). This line integral is sampled and stored as a one-dimensional data vector, and tagged with the location of the sensor when the projection was taken.

Figure 3.1 presents a diagram of a SAR antenna, which is mounted on an aircraft and pointed at a scene of interest on the ground. The antenna broadcasts a very short radio pulse (lasting microseconds) at the scene and records any reflections. It is assumed that there is negligible aircraft motion during this process; the aircraft moves and the process is repeated at a pulse repetition frequency of several thousands per second. With a single pulse, the scene cannot be reconstructed: although the one-dimensional range profile gives good range resolution, cross-range resolution

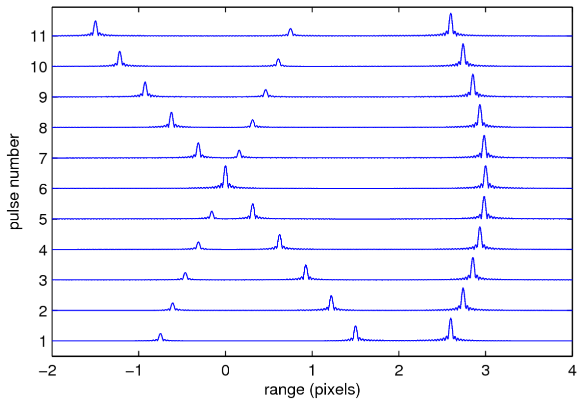


Figure 3.2: Airborne sensor-gathered line projections
Line projections obtained by a sensor flying the large aperture of Figure 3.1. Each range profile contains the linear contributions of all three scatterers. Note the merging and crossing of the two left-most scatterers at pulse 6 due to them possessing identical range displacement.

is non-existent because two reflectors equidistant from the sensor would be indistinguishable. However, by combining many one-dimensional pulse returns collected over a large azimuth extent, multi-dimensional reconstruction of the scene becomes feasible. Azimuth functions as the second dimension, variation along which suffices for a two-dimensional reconstruction. Height or elevation angle diversity, is required for three-dimensional reconstruction. Complete details of SAR reconstruction are provided in [63].

Mathematically, it can be shown that a multi-dimensional Fourier transform of a continuous function is equivalent to the one-dimensional Fourier transform of that function's Radon transform (along each projection). The discrete version of this relationship is used by a large class of tomographic reconstruction algorithms which use a fast Fourier transform (FFT). Because most FFT implementations require a rectangular grid, whereas projections are usually collected along a radial grid, polar-to-rectangular interpolation is an important pre-processing step. Such polar-formatting algorithms are attractive because the central step of multi-dimensional FFT is $\mathcal{O}(N^n \log N)$, where n is the dimensionality of the scene.

Another class of of popular algorithms is filtered backprojection (also known as convolved backprojection) which, put simply, reverses the action of the Radon transform. An image is initialized to zero; then for each projection (shown in Figure 3.2), every pixel that may have contributed to an element of the sampled projection vector is incremented by that element. For any given sample of a projection vector, the pixels that could have contributed to its value when the line integral was taken correspond to those pixels that are equidistant from the radar at the given azimuth and elevation angles. Because each pixel must query a single point along each projection, backprojection has $\mathcal{O}(N^{n+1})$ complexity, where n again is the dimensionality of the scene. It is called "filtered" backprojection because each projection is given a frequency weighting to adjust for a larger number of pixels clustering around the center of the image, due to a radial acquisition mode. This clustering is also visible in Figure 3.1: whether the aircraft flies past a scene or circles it, the radially-sampled range profiles obtained sample the center of the scene more finely than the edges.

In theory, both algorithms produce equivalent outputs, and frequently, commercial tomographic reconstruction systems are locked into one or the other. Experts on both sides have compiled lengthy lists of pros and cons, for various modalities, imaging scenarios, and scene sizes over successive generations of computer capabilities, because the image outputs of the two algorithms do differ. We sidestep the controversy by noting that while both algorithms have been demonstrated to scale well to distributed systems, backprojection very easily allows an image to be formed on a previously-obtained digital elevation map (DEM). For this major reason, we have chosen to implement backprojection for a GPU cluster processing environment.

SAR differs from CAT or MRI reconstruction in that the underlying scene being Radon-transformed is a complex-valued electromagnetic reflectivity function, encoding attenuation and absorption properties of the materials, rather than a real-valued x-ray absorption function or hydrogen energy release map. What this practically means is that with coherent SAR processing, which backprojection accomplishes, a modern system can produce fully legible image with less than 5° of azimuth. (MRI and CAT systems typically need 180° to reconstruct a two-dimensional slice.)

For mapping or surveillance applications, SAR is valuable because it has range-independent resolution, operates day and night, and is to a large degree impervious to weather conditions. Many deployments successfully deploy it with camera or LIDAR sensors to accomplish many varied tasks, but perhaps the most common is forming an image.

Having described the computational complexity of the backprojection algorithm, we next describe our choice of software engineering frameworks to write parallel implementations.

3.3 Software Support

This section describes the software tools and libraries used during the development of our radar signal analysis application. To parallelize the computation across multiple nodes, we have chosen DataCutter, a component-based programming framework and runtime engine. With DataCutter, we are able to easily make use of multicore processors and accelerators, decompose the input and output domains, and efficiently execute the overall application in and end-to-end fashion. To program the GPU, we have chosen to use Nvidia's CUDA programming environment and hardware solution. With CUDA, we are able to efficiently make use of the GPU's vast computational throughput as well as integrate seamlessly with DataCutter for parallelization across a full GPU cluster. DataCutter is presented in Section 1.2.1 in detail, while CUDA is presented in Section 2.4.1.

3.4 Implementation Details

Having given a high-level overview of tomographic reconstruction in Section 3.2 and having presented the software solutions we will leverage to accelerate image reconstruction, we delve into the backprojection algorithm in further detail.

We first present this caveat: before being used for imaging, each one-dimensional projection vector must be pre-processed. This involves windowing (to adjust the mainlobe-sidelobe tradeoff) and filtered for frequency deweighting. The former step involves element-wise multiplication of each projection vector by the window of choice. As SAR data is typically sampled and stored in the Fourier domain, filtering involves multiplying each projection by a filter frequency response and inverse Fourier bringing each projection to the spatial domain (via an inverse FFT with $\mathcal{O}(N \log N)$ complexity). As mentioned above, the filter in question adjusts for the fact that projections have a coordinate origin, and are thus more densely sampled in some areas than others, due to the radial nature of acquisition. The most common filter, the Ram-Lak filter, is simply a ramp filter with a frequency response equal to $|f|$, for frequency f.

In medical imaging, the computational burden of pre-processing has to be accounted for, and it can frequently be a critical factor (e.g., in the Cell Broadband Engine implementation of [95]). Traditionally, too much data is sampled by a SAR system to be either processed on board the aircraft or wirelessly broadcast to a ground station, making imaging non-real-time. For this reason, we chose to not parallelize the one-time pre-processing: we simply perform windowing and filtering on a central CPU, and off-load the actual backprojection to the acceleration system. We will discuss the advantages and trade offs of each acceleration method both here and in Section 3.5 where we present our experimental results.

3.4.1 Backprojection with DataCutter

In backprojection, each one-dimensional projection has a contribution for each given pixel. A number of algebraic operations have to be performed to obtain the index for each projection; then, that element of the projection vector must be fetched and incremented to the pixel. Therefore, one may assign subsets of projections to cluster nodes and partition the input, or one may assign sub-image tiles of the output image to cluster nodes, and partition the output.

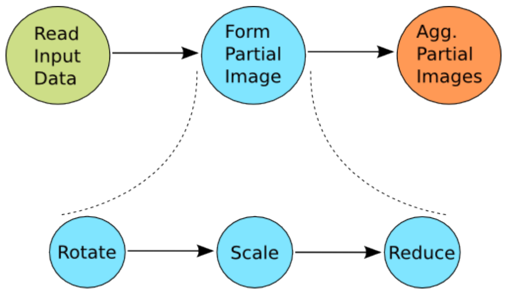


Figure 3.3: SAR Imaging Pipeline

Figure 3.3 shows the basic processing pipeline. The major tasks are *Read Input Data*, *Form Partial Image*, and *Aggregate Partial Images*. For processing pipelines with no task parallelism, the second stage will consist of only one processing element, and as such the partial image formed during this stage will in fact be the final image. However, in more complex processing pipelines (specific instances of tasks mapped to CPUs, GPUs or Cell processors) with task parallelism, the work to be performed is partitioned amongst all of the *Form Partial Image* pipeline tasks.

With SAR image formation, the work to be performed is the calculation of each input vector's contribution to each pixel in the output image. As such, both the input and the output can be partitioned amongst the imaging pipeline tasks. Figures 3.4 and 3.5 show small examples of partitioning the input and output of copied imaging pipeline tasks, respectively. When partitioning the input, the amount of computation which each imaging task performs is reduced to C/N where N is the number of imaging tasks and C is the total amount of computation required to form the total final image. When the input is partitioned, the output data size stays constant among all of the imaging tasks.

When partitioning the output in SAR image formation, the input data set size stays constant (it is broadcasted by the Read task to all of the downstream imaging tasks). Then, each imaging task only computes the input data set's contribution to a subset of the pixels of the output image.

Although more complex hybrid solutions to the decomposition of the input and output data can be developed [72], their performance is typically close to that of the output-partitioned scheme when the output size is large. Also since the input-partitioned and output-partitioned parallelization schemes represent the extremes of

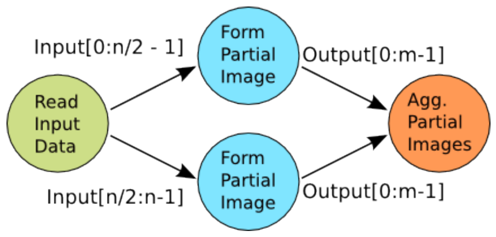


Figure 3.4: SAR Imaging Input Partitioning

the SAR image formation pipeline design space and constitutes a good base cases for comparisons, in this work we only develop and present these two parallelization schemes.

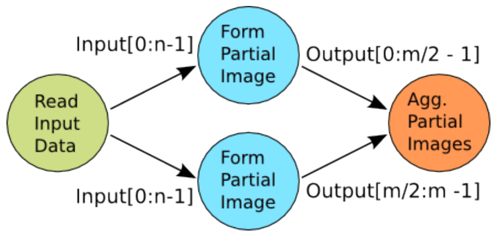


Figure 3.5: SAR Imaging Output Partitioning

3.4.2 Backprojection with GPU

Our CUDA backprojection implementation partitions the output image subset assigned to the kernel; this could be the whole range of output pixels or just a subset, depending on the output partitioning conducted across the image formation tasks. During the remainder of this section, we will assume the simple case where the pipeline includes only one imaging task. CUDA blocks correspond to rectangular sub-images, and CUDA threads correspond to individual pixels. This partitioning is appealing because we can take advantage of texture caching if we store the projections as a texture. Another advantage of using texture memory is the hardware for linear interpolation of textures [95]. This is beneficial because interpolation is required to choose an intermediate value between two samples of a given projection during the image formation, and the GPU hardware provides this for free.

As the cached-texture memory is read-only, it cannot be used for storing portions of the output image. Therefore, individual CUDA blocks allocate small image tiles in shared memory, and each member thread increments its assigned tile pixel by independently computing the index of each projection. After all the projections have been queried and the sub-image completed, it is copied back to global memory in a fully-coalesced write operation.

3.4.3 Combining DataCutter and CUDA

Since DataCutter is a component-based programming framework, it is ideally suited to leveraging other programming frameworks to ease implementation within components. Further, DataCutter allows encapsulation all of the low-level details of making use of accelerator architectures such as GPUs. Provided the interfaces exposed to the other components stay consistent, the implementations of each of the components are quite flexible.

Our combined DataCutter/CUDA backprojection algorithm used DataCutter's ability to return pointers to specific portions of data buffers which are the quantum of data the runtime engine handles. These pointers are then simply passed to a function which transfers data to the GPU and executes the CUDA kernel. By preallocating outgoing data buffers, we are able to transfer the results of the GPU computation directly to the outgoing data buffer, rather than requiring an extra copy operation.

3.5 Application Experiments

This section details the experiments we conducted to investigate the performance of using GPU clusters for SAR image formation. Following the description of the computer hardware, we discuss the dataset we used for our experiments. Then, a presentation of our results and a discussion of the interesting points follows.

All of the CPU and GPU experiments were conducted on the Ohio Supercomputer Center's (OSC) BALE Visualization GPU cluster [7]. The BALE GPU nodes consist of two dual-core 2.6 GHz AMD Opteron processors, 8 GB of main memory, Nvidia Quadro 5600 graphics cards with 1.5 GB of memory, and Infiniband network cards.

For our experiments, we used four nodes, which provides 16 cores and four GPUs. The CPUs have a peak performance of 17.6 GFLOPS in single-precision arithmetic, while the GPUs have a peak performance of 330 GFLOPS. We restrict our discussion of hardware specifications to single-precision floating-point arithmetic, because our application's implementations only make use of single-precision data types.

The Air Force Research Lab's Sensor Data Management System (SDMS) web site has released certain data sets to the public, one of which is the GOTCHA[1] data set. The GOTCHA data set consists of SAR phase history data collected with a 640 MHz bandwidth. We use a single elevation angle and up to 11° of azimuth coverage. The imaged area is that of a parking lot, and is populated with various cars and a few construction vehicles.

[1]https://www.sdms.afrl.af.mil/datasets/gotcha/index.php

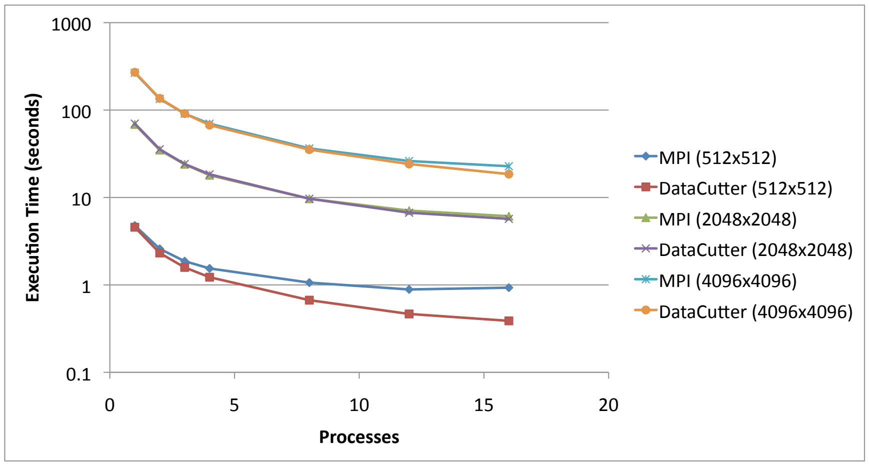


Figure 3.6: Execution times of C/MPI and DataCutter backprojection implementations with 1° of input data and varying image sizes.

In our first set of experiments, we tested the scalability of our basic DataCutter-based parallelization scheme, and compared it with an existing, simple C/MPI parallel implementation of the backprojection algorithm [48]. Figure 3.6 shows the CPU-only results, where we show the scalability of these two implementations while varying the number of processes. As the figures show, our DataCutter implementation is as efficient at using the parallel machines as the straight MPI version, and as such, will introduce no unwanted overheads when transitioning to the parallel GPU implementation. Indeed, due to the streaming nature of DataCutter, and the staggered start times of the processing nodes receiving messages from the initial Read filter, the aggregation filter does not act as a bottleneck in the application, leading to better scalability as the number of nodes increases.

Our next set of experiments present the performance gains that can be achieved by GPU parallelization of backprojection code. Figure 3.7 shows the execution times of a single GPU running the backprojection algorithm on 1° of azimuth data with two different implementations. The figure shows that DataCutter introduces a slight overhead to single GPU executions, which is to be expected for a pipelined parallel code written under the assumption either input or output data will be partitioned. Necessary additional steps (the aggregation of output sub-images for the output partitioning case, for instance) need to be executed even when only one GPU will perform the image formation. Also note the exponential growth in execution times when the output image size is increased from 2048x2048 to 4096x4096. This calls for a multi-node/multi-GPU parallelism, especially for larger image sizes.

The comparison between parallel CPU and GPU implementations is striking in Figure 3.8; as expected, the GPU implementations are significantly faster than CPU implementations. For example, running the same 1° of azimuth data using C/MPI code takes about 4.7 seconds for 512x512 image size, whereas it only takes 0.15 seconds using the GPU, hence resulting in just over 31x speedup. The performance gap increases with increasing image size; for 2048x2024 image size, the GPU's speedup over CPU is about 55x, while for 4096x4096 images, the speedup climbs to about 58x.

Figure 3.9 shows the largest-scale multi-GPU results. Here, we use 11° of data in order to highlight the issues we can solve within the multi-GPU domain. The results show that the use of additional GPUs can help to further reduce the execution time. Using 4 GPUs, we achieved up 3.45 speedup with this particular combination of input and output sizes. Also note that especially on the larger image size, the output-partitioned parallelization scheme has slightly better performance. For example, on 4096x4096 image size we achieve 3.05 speedup using input partitioning and 3.45 speedup using output partitioning. This is undoubtedly due to the fact that when the output image is partitioned, the GPU need only calculate a subset of the output

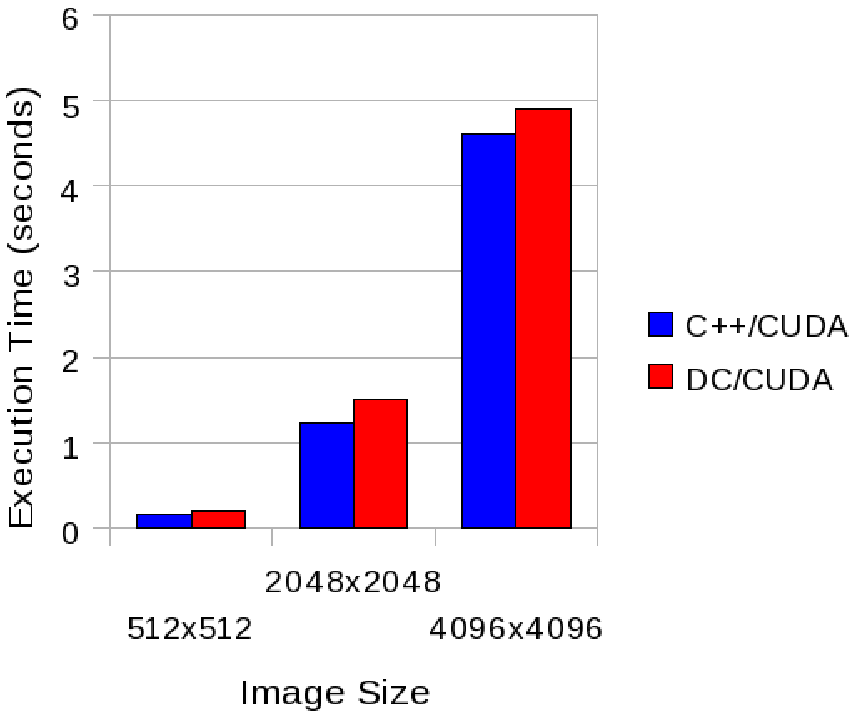


Figure 3.7: Execution times of single GPU implementations with 1° of input data.

image, and must only copy that subset from the GPU back to the host's memory. Whereas, when the input is partitioned, the whole image needs to be transfered between host and GPU. This is known to be a slow operation, due to the relatively anemic bandwidth of the PCI Express bus, to which GPUs are connected.

Our last set of experiments, depicted in Figure 3.10, shows the effect of varying the number of degrees of input azimuth data while using 4 GPUs. Since the number of projections in each azimuth degree is roughly the same, there is a linear increase in processing time for each degree of input data added to the image formation. As expected, the slopes of the lines are different for each output image size, because for

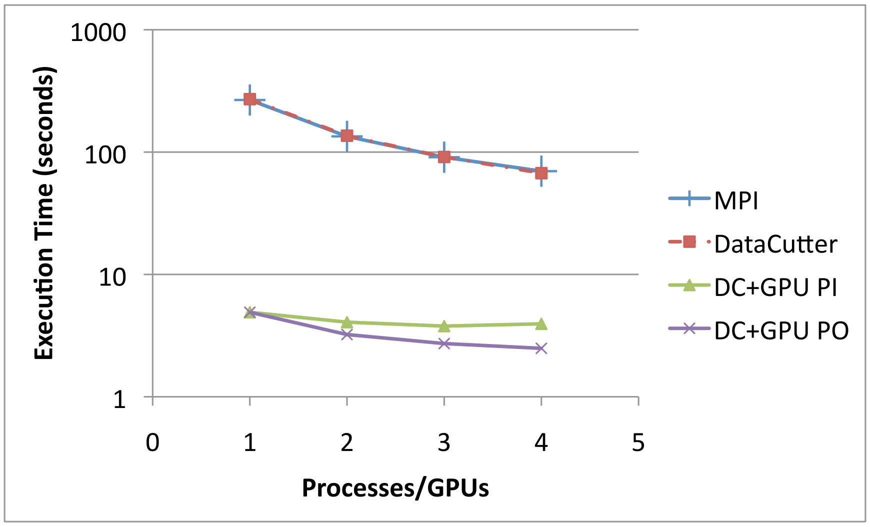


Figure 3.8: Execution times of CPU and GPU implementations with 1° of input data. The data series suffices ‘PI’ and ‘PO’ denote those implementations where the input or output is partitioned amongst the GPUs, respectively.

each input projection, the amount of computation is highly dependent on the number of pixels in the output image.

3.6 Summary

In this chapter we have presented a method for performing 2-D image formation from SAR on a cluster of GPUs. By using DataCutter for the internode parallelization and CUDA for the GPU programming, we have shown that our solution is efficient at making use of the computational resources. Further, by making use of 4 GPU-equipped processing nodes to perform the 2-D backprojection computation, we can get (versus a single CPU core executing a relatively simple backprojection implementation) 29.9x speedup on a 512x512 image, 92.1x speedup on a 2048x2048 image, and 109.9x speedup on a 4096x4096 image.

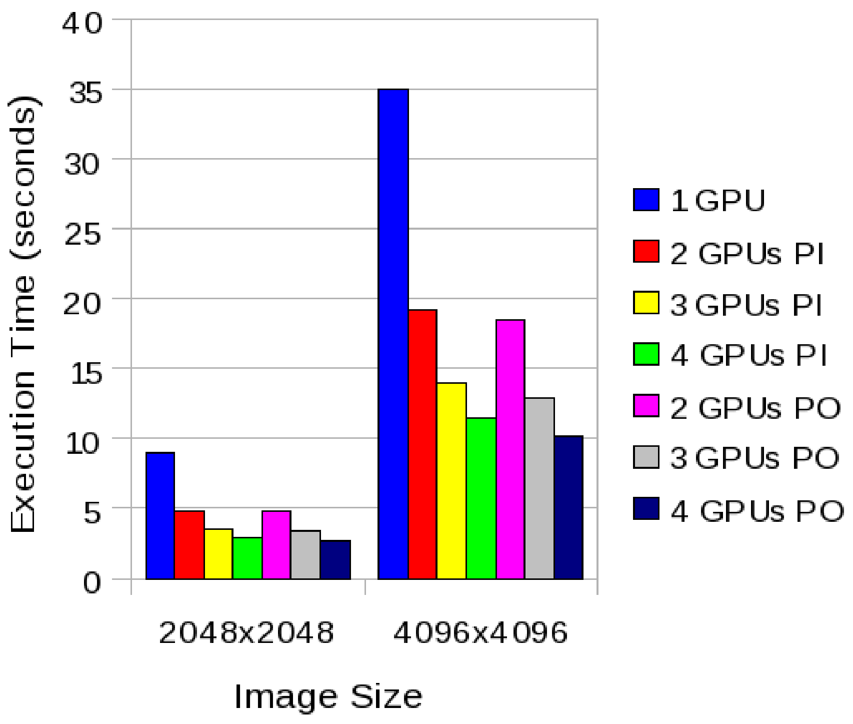


Figure 3.9: Execution times of DataCutter/GPU implementation running on up to 4 GPUs with 11° of input data. The data series suffices 'PI' and 'PO' denote those implementations where the input or output is partitioned amongst the GPUs, respectively.

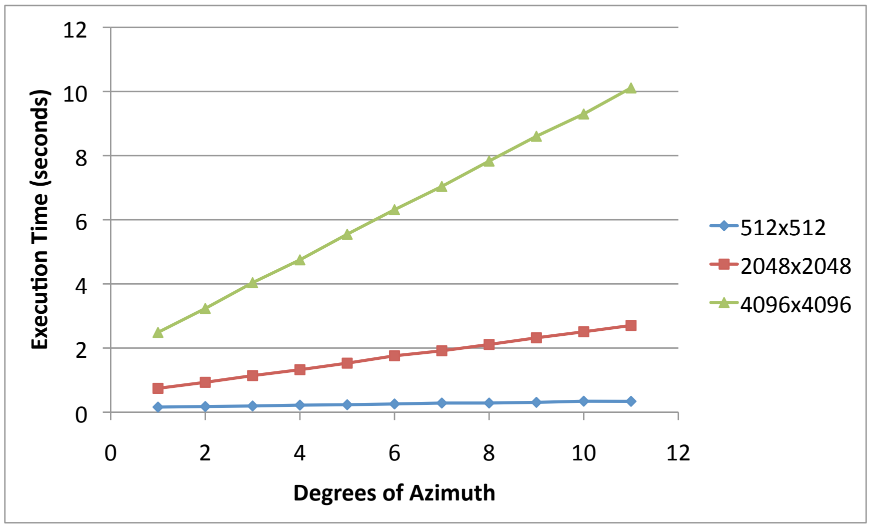


Figure 3.10: Execution times of DataCutter/GPU implementation running on 4 GPUs while varying the input data set size.

Part II: High-Performance Dataflow Middleware for Heterogeneous Computing

Chapter 4: A Component-Based Framework for the Cell Broadband Engine

4.1 Introduction

Designing parallel and distributed programs for efficient execution on large, complex supercomputers is a challenging task. Experts in fields such as physics, image and signal analysis, and biology are ill-equipped to design applications to take full advantage of the hierarchical, heterogeneous, distributed cluster supercomputers which are quickly becoming the norm for high-performance computing resources. Microprocessor architectures like the Cell Broadband Engine Architecture and Graphics Processing Units have many unique characteristics constraining their use, not to mention their own, difficult-to-learn and harder-to-master application programming interfaces. This chapter presents DataCutter-Lite, a fine-grained filter-stream programming library and runtime system which enables the simple design of filter-stream applications for modern multicore processors. We show that by using DataCutter-Lite, developers can leverage modern, heterogeneous multicore processors with good efficiency and productivity.

In order to fully use the computational power of modern multicore processors, developers must be well-versed in:

Table 4.1: High Performance Computing (HPC) techniques and DataCutter-Lite's approach

HPC Technique	**DataCutter-Lite's Method**
General HPC techniques (data blocking, communication overlap)	Non-blocking buffer writes and appropriate buffer sizing
Parallel programming and distributed address-space programming	Application-specific task decomposition and componentization
Specific architectural idiosyncrasies (messaging libraries, etc.)	Efficient lower-level libraries enabling consistent higher-level interface

- Parallel programming techniques, such as data decomposition, master-slave and pipelined computations.
- Parallel algorithms, if basic operations such as search and sort comprise a large part of the computation the developer wishes to perform.
- A threading library, such as POSIX threads.
- Architecture-specific constraints, such as memory size, or the myriad of concerns associated with the Cell Broadband Engine:
  - The Memory Flow Controller, used to transfer data in the system.
  - Techniques to deal with the small memories of the Synergistic Processor Elements, such as double-buffering.
  - Distributed memory programming models.
- The developer's own domain of expertise.

Conversely, the filter-stream programming paradigm is simple to learn, since the application implementation's components match natural divisions in the application task structure. Developers gain a large amount of flexibility and power by explicitly describing these application tasks. By defining an application programming interface and runtime engine for filter-stream programming on multicore processors, we can bridge the gap between the developers who have need of their high computational power and the breadth of knowledge and experience required to write efficient applications for them. Table 4.1 presents the high-performance computing techniques necessary for efficient use of multicore processors and our approach to incorporating them into the DataCutter-Lite runtime system and software library. Filter-stream programming meets the needs of performance-seeking, non-computer-scientist programmers for a number of reasons:

- A filter-stream programming paradigm is appropriate for many diverse types of applications, and at many data granularities. The use of this paradigm eliminates the need to use multiple complex parallel and distributed programming techniques. Further, since filter-stream programming inherently allows applications to run on parallel and distributed systems, it is appropriate for the large-scale, complex requirements of many data-intensive applications.

- Filter-stream programming is inherently component-based, and as such, the development of standard algorithms is a natural method to achieve high productivity.

- No knowledge of threading or message-passing libraries is required for filter-stream programming; all of the lower-level threading and message-passing functions are handled by the runtime system.

- All architecture-specific constraints are abstracted away, leaving the application developer with a clean interface and programming semantic. However, should some later developer with good knowledge of the internal workings of the architecture wish to make some modifications, the system is compatible with all of the standard optimizations that High Performance Computing gurus make.

As a case study, we have implemented DataCutter-Lite on the Cell Broadband Engine. The Cell Broadband Engine is an excellent example of modern multicore processors, with its heterogeneous nature, its high performance communication bus, and high throughput processing capabilities. Other examples of modern multicore processors are the current top-end products from AMD and Intel with deep cache hierarchies and the forthcoming Larrabee processor from Intel. Larrabee will have 16 - 32 vector processors and will initially be marketed strictly as a graphics processor. These types of microprocessors are not considered in this chapter. The Cell Broadband Engine is an excellent microcosm of the future of multiprocessors, and as such is an excellent test case for our techniques. As future work, we will extend these results to more traditional architectures.

Due to the large potential benefit of modern multicore processors, many research projects have developed techniques to ease the burden associated with writing applications for them. BlockLib [2], IBM's Accelerated Library Framework (ALF) [60], Charm++ [71], CellSs [9], and Sequoia [38] are examples of programming frameworks for the Cell Broadband Engine which use various block-based methods for handling the memory hierarchy of the Cell in order to get good application performance. The Cell Messaging Layer [83] and MPI microtasks [78] are examples of frameworks designed to provide an MPI-like set of semantics for SPE communication. We offer an alternative here, the CBE Intercore Messaging Library, which provides the underlying basis for our streaming programming framework. GLIMPSES [99] provides a method to improve the efficiency of SPE programs by providing profiling information for code optimization. Some streaming frameworks even exist for the Cell Broadband Engine [116], although they offer an interface more appropriate for use with a stream-language compiler and not for application development. Hence, while other programming frameworks exist for the Cell Broadband Engine, DataCutter-Lite is the only active research project which aims to develop a comprehensive, hierarchical middleware system for the development of large-scale, efficient filter-stream programs.

The rest of this chapter is organized as follows. Section 4.2 discusses the overall DataCutter-Lite architecture, while Section 4.3 discusses the Cell Broadband Engine Intercore Messaging Library, which enables DataCutter-Lite to abstract some of the specific architecture's details. Section 4.4 presents some of the optimizations involved with the design of a filter-stream runtime-engine for the CBE, and Section 4.5 presents the results of some of the test-case applications. Section 4.6 summarizes this chapter and presents some open research directions.

4.2 Filter-Stream Programming for Heterogeneous, Hierarchical Clusters

Section 1.2.1 introduces the DataCutter middleware that is used as insipration for DataCutter-Lite.

4.2.1 DataCutter-Lite Architecture and Programming Model

While DataCutter is capable of handling all levels of granularity for an application, from multiple, distributed-address space clusters to SMP nodes, the introduction of heterogeneous and massively parallel microprocessor architectures necessitated a new runtime engine. DataCutter-Lite (DCL) operates only within a single node, but is optimized for modern multicore processors. By using DCL, application developers will able to efficiently make use of modern multicore processors without being expert computer scientists fluent in the newest cutting-edge parallel programming techniques. Additionally, since DCL is a component-based framework, developers can realize a higher degree of productivity over programming the application directly for the architecture in question.

DCL's architecture has two separate pieces, the runtime engine and the application programming interface (API). The API is comprised of a small number of easy-to-use functions to support application set up and execution. Figure 4.1 gives an overview of the functions in the main API. While the concepts of layout and placement are distinct, since our work is still in the development stage, we have chosen to combine them into a single initialization function.

The first implementation of DCL is designed for the Cell Broadband Engine (CBE). The overall architecture is comprised of three software layers. Figure 4.2 shows the runtime system and its connections to the various system components. The stepped connections to the various system components are meant to express the notion that while applications can be designed solely to use the DCL API, there is nothing stopping expert developers from mixing function calls from all levels of the

```
// Library Initialization Functions
void setup_app(Placement *)
void init_dcl()
// Communication Functions
DCLBuffer * create_buffer(stream, size)
int stream_write(stream, DCLBuffer *)
int stream_close(stream)
```

Figure 4.1: DataCutter-Lite Library Application Interface

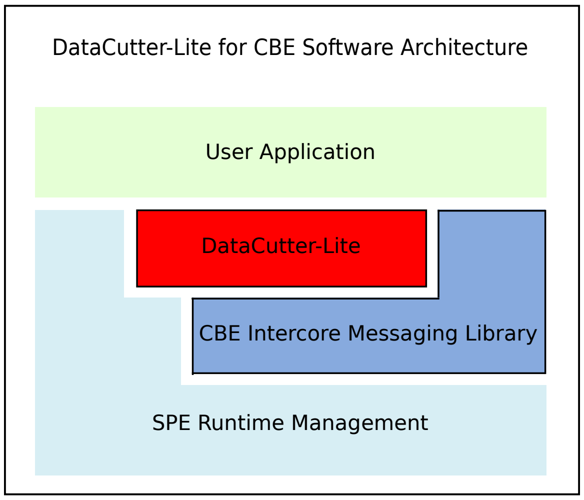

DataCutter-Lite for CBE Software Architecture
User Application
DataCutter-Lite
CBE Intercore Messaging Library
SPE Runtime Management

Figure 4.2: Overall DataCutter-Lite System on the CBE

software hierarchy if they so choose. The two portions of the system architecture figure surrounded in bold lines, the DCL implementation and the CBE Intercore Messaging Library represent our contributions.

The lowest layer of software is IBM's SPE Runtime Management Library Version 2.2 (libspe2), which is part of IBM's Software Development Kit (SDK) for Multicore Acceleration. The libspe2 library is the interface the developer uses to gain access

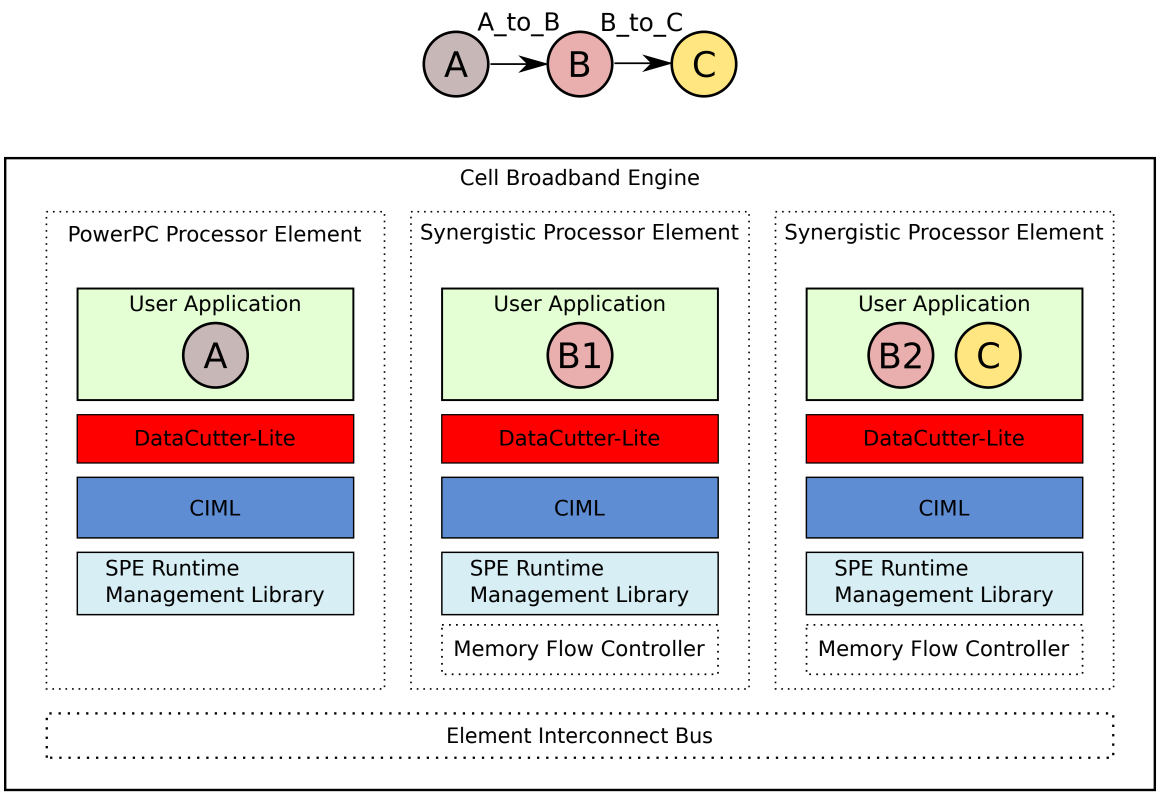


Figure 4.3: Sample DataCutter-Lite application and example mapping onto the Cell.

to the Synergistic Processing Elements (SPE), which represent the majority of the computing power of the CBE.

The next layer of software is the CBE Intercore Messaging Library (CIML). This library is a two-sided communications library whose interface and communication model mimic that of MPI, including blocking and non-blocking send and receive primitives. CIML is detailed in Section 4.3.

The last layer of software is the DCL runtime system, which is initialized by a function call from the user's main PowerPC Processor Element (PPE) program, as in Figure 4.1. The initialization function creates threads to manage the SPEs, begins the execution of the SPE programs, and initializes the CIML, to allow two-sided communication between the system processors. At this point, each of the PPE and SPE processors enters a runtime daemon thread which listens for data buffers. Upon receiving a buffer, the daemon consults the filter placement and calls the correct filter's processing function. This event-driven model is well-suited to the CBE, since the SPEs are single-threaded processors and have no capability to switch thread contexts like more traditional processors. All of the underlying details involved with determining where to send the data, the actual transfer of the data and the function call is all handled by the runtime system. Also, the developer need not concern themselves with the cleanup of data buffers, since the runtime engine keeps track of all of the buffers that are created, sent, and received. When a filter returns from the processing function, the buffer passed as input is freed. Similarly, when a buffer is written to a stream it is assumed that it is handed off to runtime system. All of the buffers are created in a configurable heap area, so as to alleviate the burden of explicit memory management.

Figure 4.3 shows a small example application composed of three pipelined filters, A, B, and C and one possible filter placement. To help explain some of the details of the runtime engine, we have decided to create a placement with one copy each of the A and C filters and two copies of the B filter. Buffers written to the 'A_to_B' stream are handed off to the DCL daemon on the PPE. The daemon then determines where to send the buffer, either copy B1 or B2 of the B filter type. This determination is made by a configurable stream sink protocol. Common protocols are round-robin, random, or broadcast. Future versions of DCL will include a demand-driven protocol where filters which sink data faster from a stream will receive more buffers. Buffers written

```
// PPE main()
// Set up Matrices A, B, pointers
// a_ptr, b_ptr, constants
int main(int argc, char ** argv) {
 init_dcl();

 for (i = 0; i < NUM_ROWS; i++) {
  DCLBuffer * buffer = create_buffer("raw_data", BUF_SIZE);

  append_array(buffer, a_ptr, NUM_COLS * sizeof(float));
  append_array(buffer, b_ptr, NUM_COLS * sizeof(float));

  stream_write(buffer);
  // increment pointers a_ptr, b_ptr
 }
 finish_dcl();
 return 0;
}
```

Figure 4.4: DataCutter-Lite Example PPE Code

to the 'B_to_C' stream are likewise handed off to the DCL daemon, but since in the placement only one copy of the C Filter type exists, there is only one destination for these buffers. While buffers written by B1 have to travel through DCL, CIML and libspe2 libraries and through the Memory Flow Controller (MFC) and Element Interconnect Bus (EIB) CBE processor elements, buffers written by B2 will be directly handed off to filter C by DCL by simply passing the pointer of it.

Figures 4.4, 4.5 and 4.6 show some example code for an application which uses one SPE to add together two matrices. The `setup_application()` function in Figure 4.5 is defined by the developer to tell the runtime engine where to place filters and how to connect the data streams. After the `init_dcl()` initialization function in Figure 4.4

```
// PPE setup and filter code
// Called by init_dcl()
void setup_application(Placement * p) {
 Filter * console  = get_console(p);
 Filter * fadded = place_ppu_filter(p, "added_data");
 Filter * fadder = place_filter(p, 0, "add_values");

 Stream * sraw = add_stream(p, "raw_data");
 add_source(p, sraw, console);
 add_sink(p, sraw, fadder);

 Stream * sadded = add_stream(p, "added_matrix");
 add_source(p, sadded, fadder);
 add_sink(p, sadded, fadded);
}

// When receving a buffer from SPE
void added_data(DCLBuffer * buffer) {
  // Deal with added matrix data
}
```

Figure 4.5: DataCutter-Lite Example PPE Code

returns, the `main()` function can do work to begin the computation. Note how simple the SPE code is (see Figure 4.6); it has none of the complicated operations normally associated with programming the CBE. However, the code is fully multi-buffered, as CIML (see Section 4.3) allows for the overlap of computation with communication.

4.2.2 DataCutter for Distributed Multicore Programming

DataCutter and DCL are parts of a burgeoning component-based middleware framework designed to provide efficient dataflow execution on modern hierarchical, heterogeneous cluster supercomputers. Our ultimate goal is to develop a comprehensive, flexible API such that DataCutter will handle coarse-grain dataflow over the

```
// SPE code: Set up constants
void add_values(DCLBuffer * buffer) {
 DCLBuffer * out_buffer =
           create_buffer("added_matrix", BUF_SIZE);

 float * a = (float *) get_extract_pointer(buffer);
 float * b = (float *) get_extract_pointer(buffer);
 float * c = (float *) get_data_pointer(out_buffer);

 for (i = 0; i < NUM_COLS; i++)
  c[i] = a[i] + b[i];

 stream_write(out_buffer);
}
```

Figure 4.6: DataCutter-Lite Example SPE Code

LAN/WAN network and DCL will handle fine-grain dataflow within a single node. While we draw a distinction between these two projects since DCL is presented in this chapter, the end goal is for that distinction between DataCutter implementations to disappear. DCL instances within a node will act like filters to DataCutter to achieve seamless coarse-to-fine grain integration and interoperability.

Figure 4.7 shows what a mixed DataCutter and DCL application might look like. At the largest application granularity, whole sets of data are considered. Raw datasets from large-scale simulations, whole patient files, or whole experiments to be run might be examples of this coarse-grained data. This data will be partitioned using Data-Cutter to run on whole clusters. At the smallest granularity, DCL can be used to leverage multicore processors to analyze the fine-grained data, such as individual simulation timesteps, single data point analysis, or single pixel operations. In Section 4.5

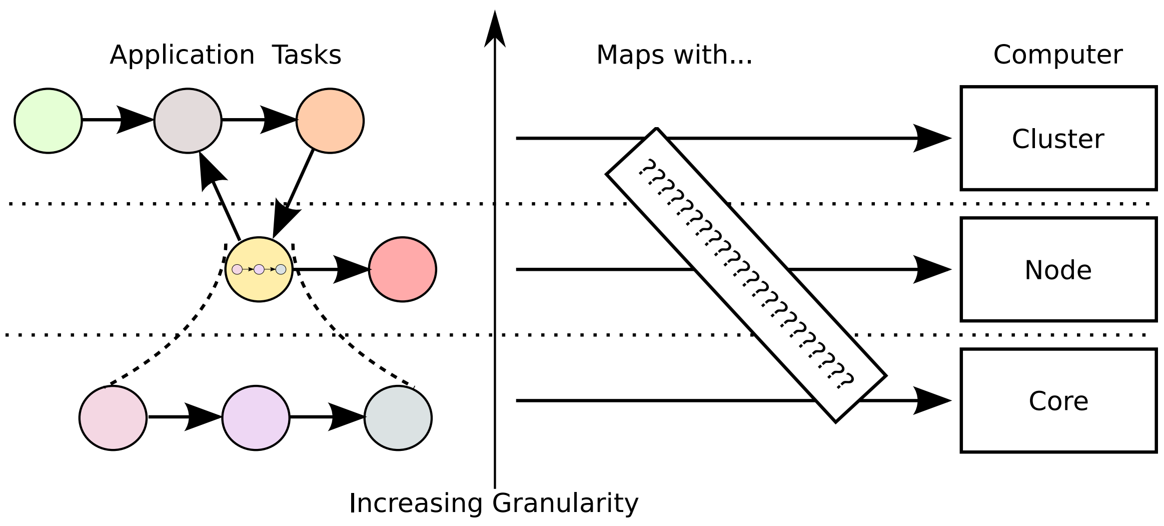


Figure 4.7: Software and Hardware Granularities

we discuss a real-world biomedical image analysis application implementation with a mixed DataCutter and DCL paradigm.

4.3 CBE Intercore Messaging Library

We have designed and implemented a two-sided communication library for the CBE processor: The CBE Intercore Messaging Library (CIML). CIML makes use of the libspe2 interface, and begins to abstract away some of the architecture-specific details of the communication channel. That is, a developer using function calls in CIML will not need any knowledge of the MFC, which is the functional unit associated with a Synergistic Processor Unit (SPU) enabling it to access main memory through Direct Memory Access (DMA) commands. Each SPU has an associated MFC, and each MFC can queue 16 DMA commands before blocking the SPU's execution of instructions. It is through this method the SPEs can overlap communication with useful computation.

CIML mimics the spirit and usage patterns of MPI with both blocking and non-blocking send and receive function calls. Additionally, CIML allows direct SPE-SPE data transfers, without passing through the main memory cache hierarchy or involving the PPE. IBM's libspe2 allows the developer to map the SPEs' local store space into main memory. DMAs involving addresses in these memory-mapped locations do not go through main memory, and are therefore not subject to the 25 GB/s main-memory bandwidth limit. This being the case, multiple pairwise SPE-SPE communications can occur simultaneously, and these transfers are only subject to SPEs' end-point throughput of 25 GB/s. (Each SPE has simultaneous send and receive bandwidths of 25 GB/s.)

Table 4.2 shows the API for SPE-SPE communication in CIML. The B and NB suffices on the function names are intended to convey the fact that the send and receive calls deal with blocking and non-blocking communications, respectively. As such, the function calls sendB and recB will block until the communication is complete. The function calls suffixed with NB will return as quickly as possible to allow for more computation to continue while the communication completes. The final PPE-SPE communication API is similar - but not identical - to the SPE-SPE API. The reasoning behind a slightly different API for PPE-SPE communication is discussed in Section 4.4.

Figure 4.8 shows the communication bandwidth for CIML, while Figure 4.9 shows the latencies involved in ping-pong communication. (Incidentally, Figure 4.8, Figure 4.9, Figure 4.10, Figure 4.11, and Figure 4.12 show the results of the final, optimized version of CIML; discussions about optimizations and their effects are included in Section 4.4.) In this experiment we have a single source SPE and we have varied

Table 4.2: CBE Intercore Messaging Library Application Programming Interface

Send Functions	Receive Functions
sendB(dest, src_ptr, size)	recB(src, dest_ptr)
sendNB(dest, src_ptr, size)	recNB(src, dest_ptr)
send_completeNB(dest, src_ptr)	rec_completeNB(src, dest_ptr)
send_complete_allNB(dest)	rec_complete_allNB(src)
	int probeNB(src)

the number of destination SPEs. The results are an average over 100 iterations for each message size from 1 byte to 16 KB; also, the results are aggregate and do not use any hardware-based broadcast mechanism. Since CIML is a two-sided library, there is some extra overhead associated with communications. However, at the larger message sizes, a single SPE-SPE communication channel gets over 80% of the possible bandwidth with 4 KB messages, and over 90% with 8 KB messages. By using more than one SPE, maximum bandwidth can be achieved even with shorter messages.

Figures 4.10, 4.11 and 4.12 show the bandwidth and latency measurements for PPE-SPE communication. Again, these results are an aggregate value, and averaged over 100 iterations. The most striking thing about these charts is the large decrease in communication bandwidth versus the SPE-SPE communication. In order to provide a two-sided communication interface, some information must be transferred from the main memory to the local store of the SPE, and the CBE is constrained in its PPE-SPE communications by a number of architectural idiosyncrasies. These results are actually the result of several rounds of optimizations, some of which are discussed in the next section.

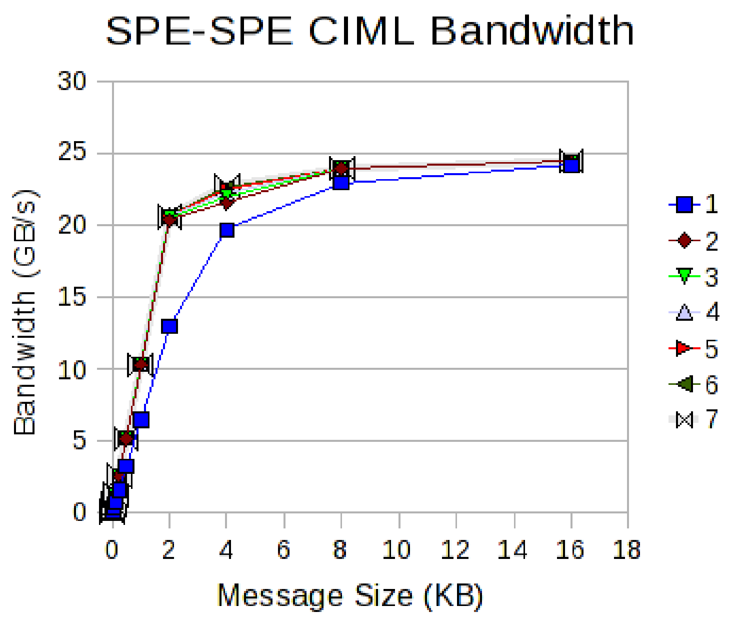


Figure 4.8: SPE-SPE Communication Bandwidth Results

4.4 DataCutter-Lite for CBE Optimizations

This section presents some of the optimizations made on the DCL for CBE runtime engine and on CIML. The first two optimizations are applied due to constraints or capabilities of the CBE, while the last two are made for more traditional reasons like allowing communication and computation overlap, or avoiding deadlock.

4.4.1 High-Bandwidth SPE-SPE Two-Sided Communication

In any two-sided communication, some information must be transferred from the sender to the receiver, and vice versa. Since SPEs in the CBE only have 256 KB of local store memory with which to store the executable and all data, our communication protocol must take this into account. In order to allow high-bandwidth SPE-SPE communication, we have chosen to implement a sender-initiated, pull-based protocol.

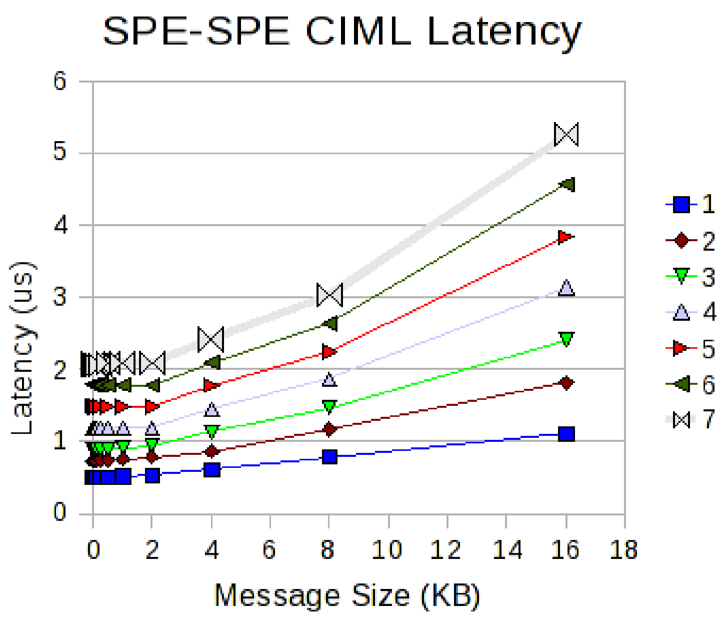


Figure 4.9: SPE-SPE Communication Latency Results

While the authors of [83] achieve good results with a receiver-initiated protocol, in the streaming paradigm no write is ever blocking, meaning that the cost of having a sender wait for the receiver to transfer destination data is too high.

Therefore, in our protocol, with a 'put,' the sender transfers to the receiver a header packet containing the source address and size of the message. The receiver polls its local header queue, waiting for a message header. When the header is received, the message transfer can be initiated with a 'get', and the data can be used once the transfer is complete. A 'message-received' header is also sent back to the sender. The sender is not involved in the operation, except for sending the header to the receiver. Since DMAs are non-blocking (as long as the DMA command queue is not full), the sender can then proceed to other computation. Figure 4.13 shows the effect of using

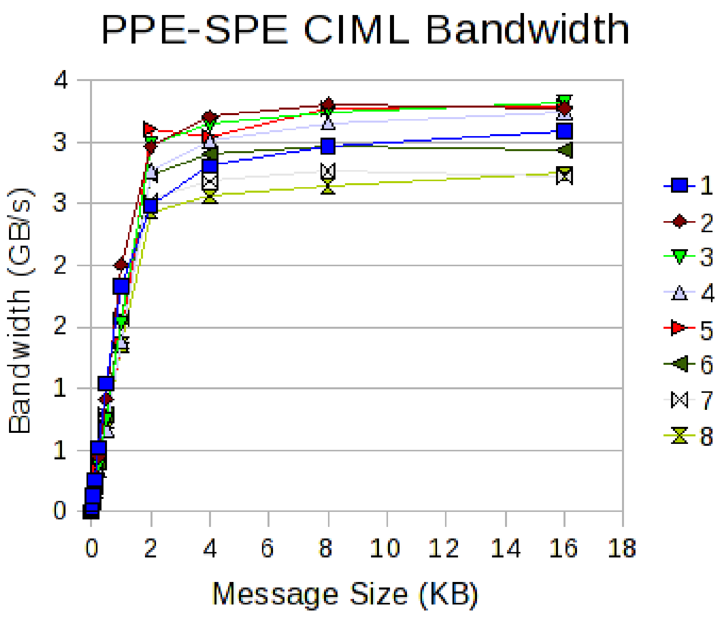


Figure 4.10: PPE-SPE Communication Bandwidth Results

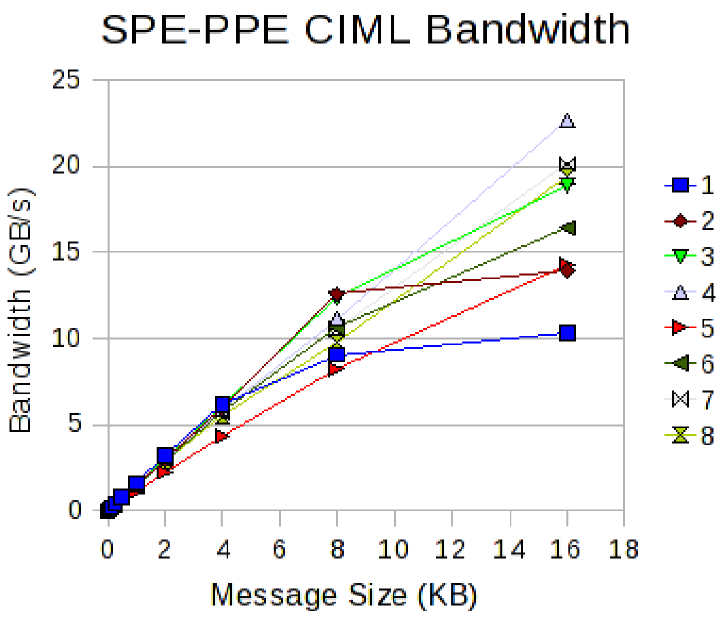


Figure 4.11: SPE-PPE Communication Performance Results

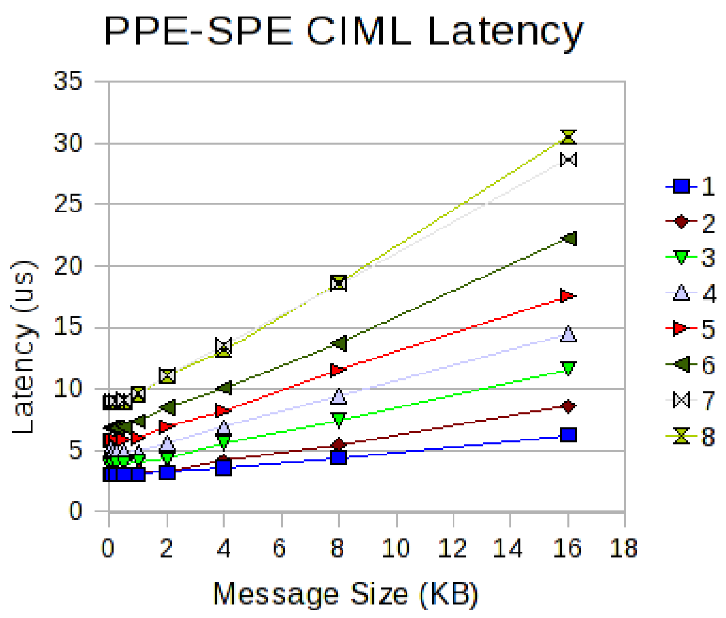


Figure 4.12: SPE-PPE Communication Latency Results

a sender-initiated, pull-based protocol versus a sender-initiated, push-based protocol. The results were taken with a single sender and a single receiver.

4.4.2 Pure Pull-based PPE-SPE Communication

Unfortunately, while a sender-initiated, pull-based scheme works well on the SPEs, when the sender of the message is the PPE, the smaller DMA command queue size available to the PPE harms the communication bandwidth. The MFC in the SPE has a DMA command queue of size 16 for DMA commands initiated by the SPU. The PPE only has a queue of size 8 for DMA commands dealing with that SPE. This asymmetry means that a carefully designed pull-based PPE-SPE communication library is more appropriate than one in which the sender transfers the message header to a known location in the receiver's memory space. That is, when the SPE is attempting to

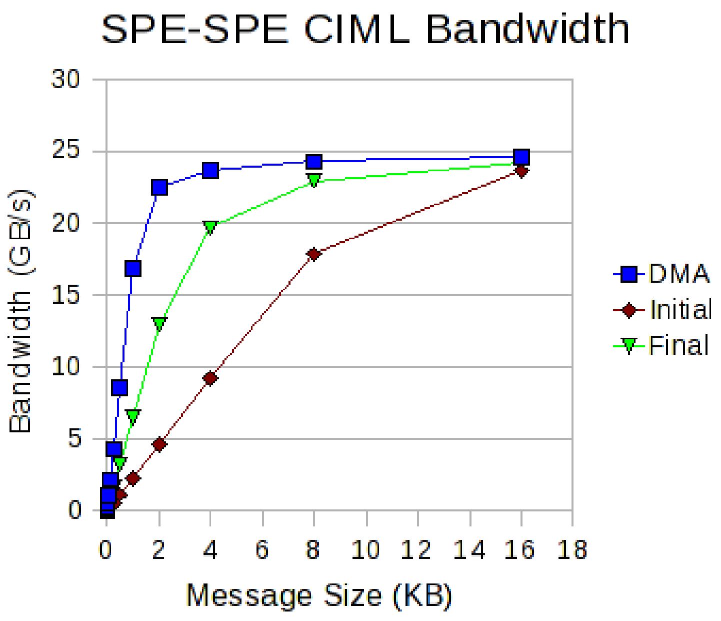


Figure 4.13: SPE-SPE Communication Bandwidth Results

read from the PPE, it must first transfer the message header from the PPE; the PPE's responsibility is simply to write to its own message header area. To increase the bandwidth, we transfer the entire set of message headers. When the PPE is writing multiple messages to the same SPE without expecting a response, the SPE can successfully use this local cache of the message headers to initiate plenty of DMA commands. As such, CIML implements this type of PPE-SPE communication, in favor of mimicking the 'more' two-sided approach used in SPE-SPE communication. Figures 4.14 and 4.15 show the results of changing the PPE-SPE communication method from that mimicking the SPE-SPE method to a pure pull-based method. As above, the figures show the results of a single sender and a single receiver. Therefore, the final values match the single-threaded results from Section 4.3. The PPE-SPE transfer from main memory is still hindered by architectural characteristics of the

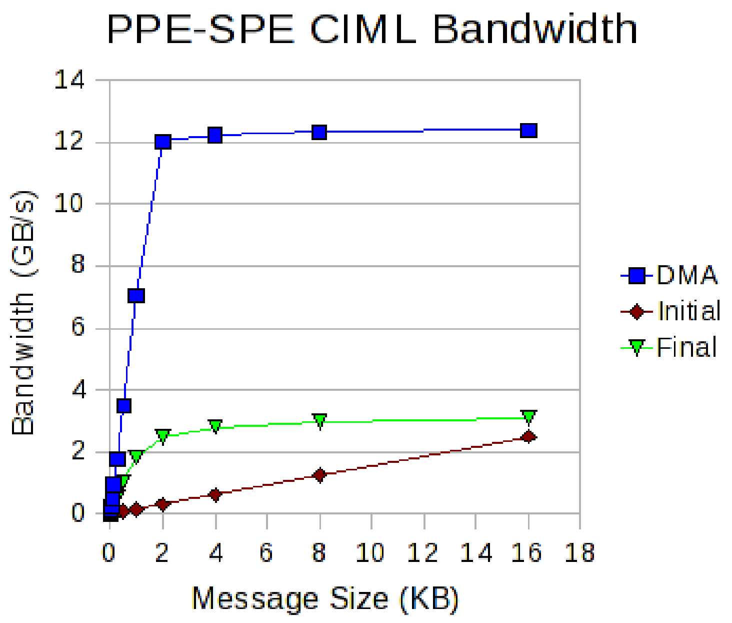


Figure 4.14: PPE-SPE Communication Bandwidth Results

CBE, and as such suffers from a more anemic transfer rate than SPE-PPE data transfers to main memory. Also, even DMAs reach only half of the maximum main memory bandwidth of 25 GB/s during the SPE-PPE transfer.

4.4.3 Buffer Prefetching

A standard high-performance computing technique is to overlap communication with computation. This is eminently possible with the CBE, since instructions to place DMA commands into the SPE DMA command queue are issued quickly, and are non-blocking when the DMA command queue is not full. Therefore, DCL uses prefetches buffers when calling filters' processing functions. In the simplest case, this allows automatic double-buffering of data for use in streaming operations.

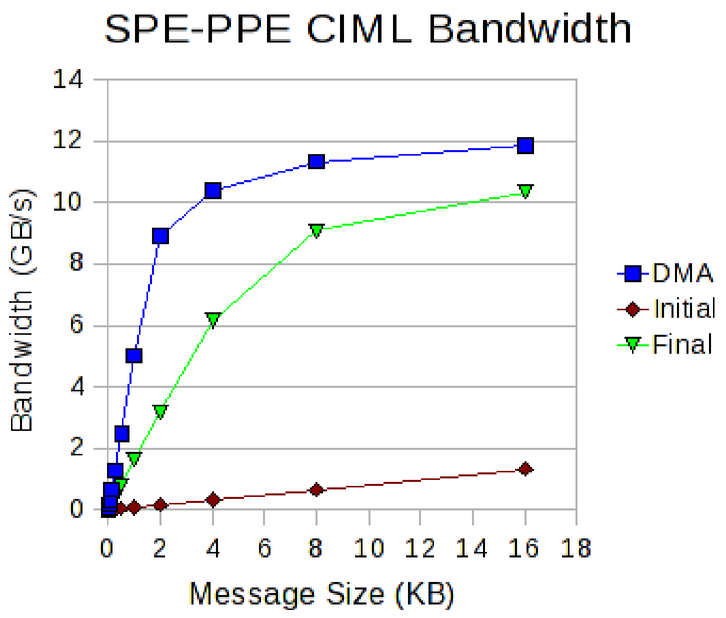


Figure 4.15: SPE-PPE Communication Bandwidth Results

4.4.4 Fine-grained Buffer Arrival Blocking

When DMA commands are issued for message transfers in the CBE, each command can be assigned a 5-bit tag id. While each SPE's filters are intended to be independent of one another, the buffer heap and the DMA command queue tags are shared among all of the filters running on one SPE. By keeping track of which buffer matches which DMA tag, CIML is able to provide fine-grained buffer receipt or send completion blocking for each filter. The practical effect is that multiple filters running on the same SPE do not step on each other's toes when sending and receiving complex patterns of messages. When a filter needs to create a buffer which is too large for the total remaining heap, the runtime engine can wait for the oldest remaining message transfer to complete. Once that message transfer is complete, the heap space can be freed, giving room to the new buffer. If CIML did not keep track of these DMA tag/buffer associations, the entire DMA command queue would have to be flushed in order to create enough heap space for new buffers. This would cause unwanted latency and bandwidth degradation.

4.5 Application Experiments

In order to evaluate the performance of the CBE DataCutter-Lite implementation and to evaluate the programming paradigm, we have developed three applications with a range of characteristics. The experiments were performed on the Ohio Supercomputer Center's Glenn e1350 Blade Center.

The first application we developed to evaluate the DCL implementation and the programming API is a simple matrix addition application. Since the computation involved with this application is extremely small, this code shows a large Communication-to-Computation-Ratio (CCR). We have used IBM's Accelerated Library Framework (ALF) matrix addition example [60], in order to obtain a good baseline comparison for our DCL-based implementation.

The second of our example applications is a simple color-space transformation to be performed on an image. The application simply transforms each pixel in an image from the RGB color space to the LAB color space. These types of computations, while simple, are fairly time consuming. As such, this example application will feature a small CCR value. As a baseline comparison, we have used a custom-implemented color-space transformation application which uses only IBM's SDK for the CBE.

The last of these applications is a real biomedical image analysis application [55]. The input to the overall application is a tissue slide image digitized at high resolution.

Each RGB pixel is converted to the LAB color space and some statistics are calculated on a per-tile basis. (We reused the color-space transformation code presented earlier.) The luminance channel is then taken from the LAB image and a local binary pattern (LBP) feature is calculated. The four statistics per image channel and the LBP feature comprise a feature vector which is used in a classification stage in order to determine the properties of the image tile. To compare with our DCL-based implementation, we have developed a custom CBE implementation of the image analysis application. This implementation uses one main loop which reads the RGB image tiles from a socket and performs the functions on each tile. The operations are slightly decoupled, meaning that the RGB-to-LAB color space transformation and the LBP feature calculation must be scheduled separately on the SPEs. The main loop in the application acts as this task scheduler.

The first two applications, the matrix addition and the color-space transformation, are simple kernels that represent the widest range of CCR values which real applications might exhibit. Most real applications are built from components like these and hence their performance can be easily predicted by looking at how the individual components behave.

The DCL versions of all three applications and their baseline counterparts are inherently composed of the same independent tasks. As such, there is no algorithmic method used to reduce the amount of work for one of the implementations versus the other. All of the execution times shown in the charts for the baseline versions of the applications are the best times obtained by hand-tuning the application performance. Similarly, optimizations were made to the DCL runtime engine in order to allow the

DCL versions of the three applications to achieve the best performance results. Further, optimizations were made to the DCL versions of the applications themselves. On the PPE, the optimizations were mainly made to relieve the main memory bandwidth requirements. For the SPE code, the most important optimization to be made is choosing the size of the data buffers; incorrect choices limit the amount of communication and computation overlap which is achievable. These optimizations were made in an ad-hoc manner, and their in-depth discussion is beyond the scope of this chapter. In our future work, we intend to develop automatic techniques such that these performance optimizations are made without the developer's intervention.

Figure 4.16 shows the execution times for the two matrix addition implementations. The input matrices are 1024 x 512 in size. DCL's execution times are greater than those of IBM's ALF implementation, ranging from 8% higher on 6 SPEs to 91% higher execution time on one SPE. The higher execution times are due to a couple of reasons. First, the construction of serialized data buffers is an operation which the ALF implementation does not need. Further, since the DCL matrix addition program is very simple, the highest throughput DMA buffer size of 16 KB is not used (DCL application simply transfers one row at a time), whereas the ALF implementation uses this DMA buffer size. Both of these issues can be solved with some extra effort, but the simple implementation is meant to serve as a baseline number, and is the worst the DCL method is likely to give. Further, the ALF implementation uses many cryptic function calls to set up the task graph. The DCL implementation merely requires the use of a handful of functions.

Figure 4.17 shows the execution times and parallel speedups for the color-space transformation performed on 32 image tiles. Since the overheads of reading the images

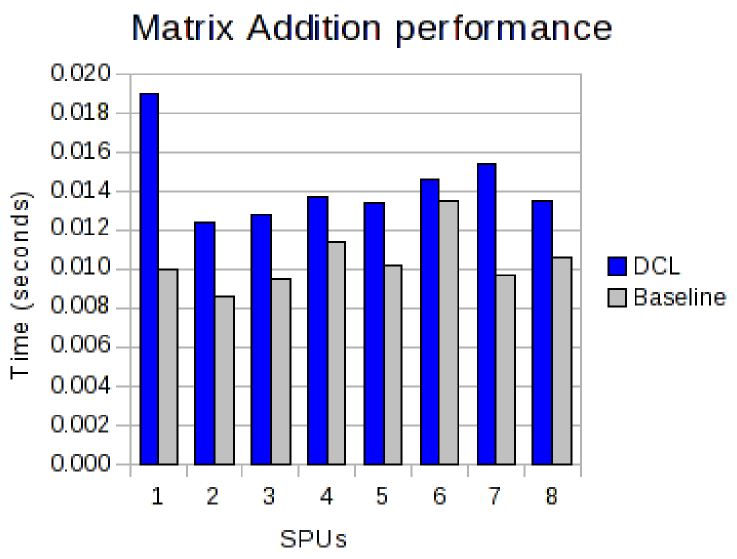
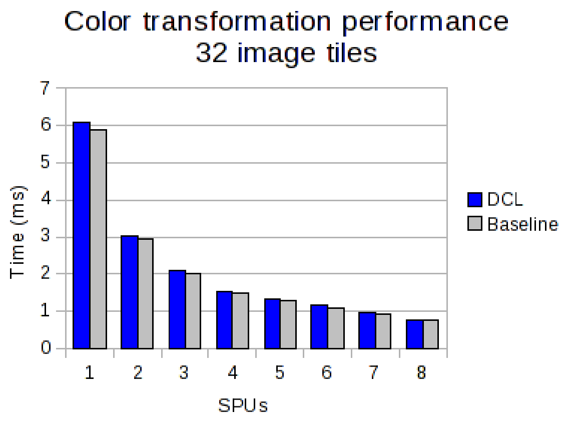


Figure 4.16: Execution times for Matrix Addition



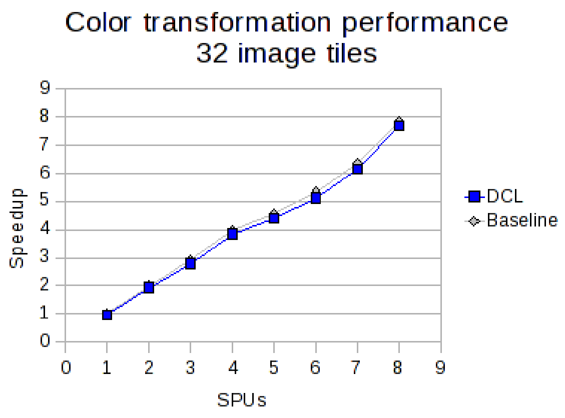
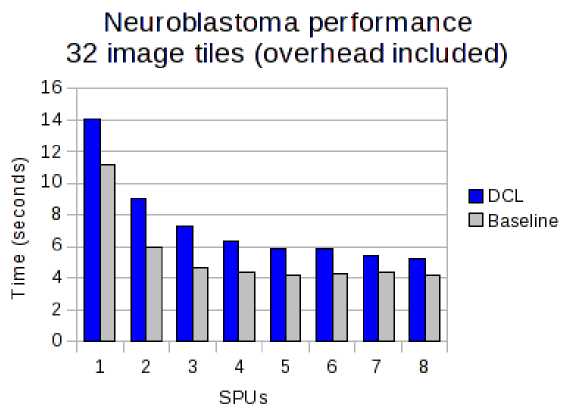


Figure 4.17: Execution times and speedups for color transformation for 32 image tiles



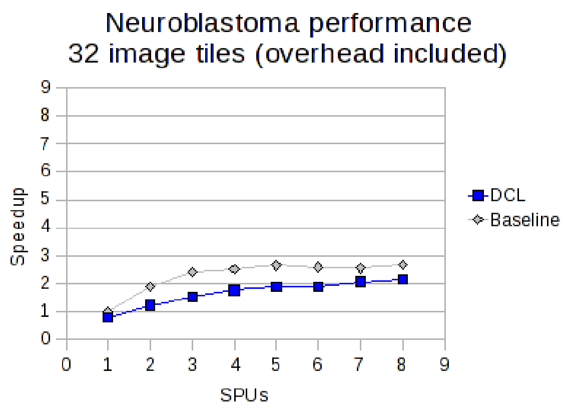


Figure 4.18: Execution times and speedups for biomedical image analysis application for 32 image tiles - overheads included

from the disk are disregarded here, the speed up is nearly linear, reaching a value of 7.9 for the baseline version and 7.7 for the DCL version. Without intimate knowledge of how the IBM libspe2 library schedules the logical SPE contexts onto physical SPE resources, it is hard to postulate a reason why a knee exists in the speedup curves after 4 SPEs. However, we expect that up to 4 SPEs, a degree of regularity is maintained in the placement on the physical resources - and of the communication pattern, there being 4 independent message channels in the ring bus. From 4-7 SPEs, this regularity is necessarily disturbed. When 8 SPEs are used, this regularity returns to a reasonable degree, even though the communication bus is most highly loaded at this configuration.

Figures 4.18, 4.19 and 4.21 show the results of the experiments on the full biomedical image analysis application with various overheads included and excluded, respectively. When end-to-end applications are considered, even the most efficient algorithm implementation is subject to such concerns as disk latency, and upstream data overheads. In this case, the upstream data overhead is the decompression of the TIFF images to be calculated. When excluding these overheads, we see a similar pattern of near-linear speedup for the DCL implementation. Unfortunately, the baseline version of the application actually begins to suffer when the number of SPEs used rises above 5. This is likely due to the extra scheduling overhead and memory bandwidth used in the baseline's implementation, since it decouples the two major stages of the operation and schedules them separately. The DCL implementation simply writes one buffer as output from the first stage to the input of the second stage. Since this buffer stays in the SPE, it saves main memory bandwidth, and since the second stage is triggered by the runtime system resident on the SPE, the PPE is not involved in the scheduling operation, saving time.

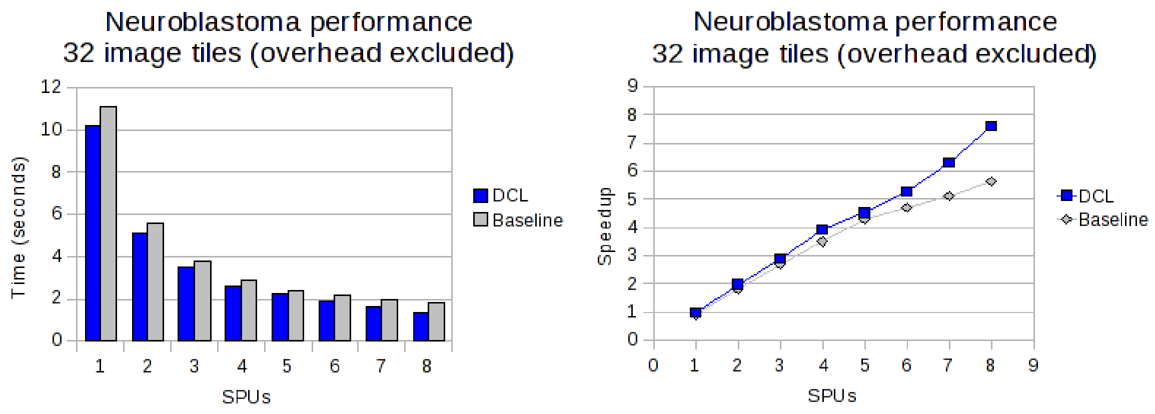


Figure 4.19: Execution times and speedups for biomedical image analysis application for 32 image tiles - overheads excluded

When the TIFF decompression overheads are considered, the application performance decreases, particularly when more SPEs are involved in the computation. The

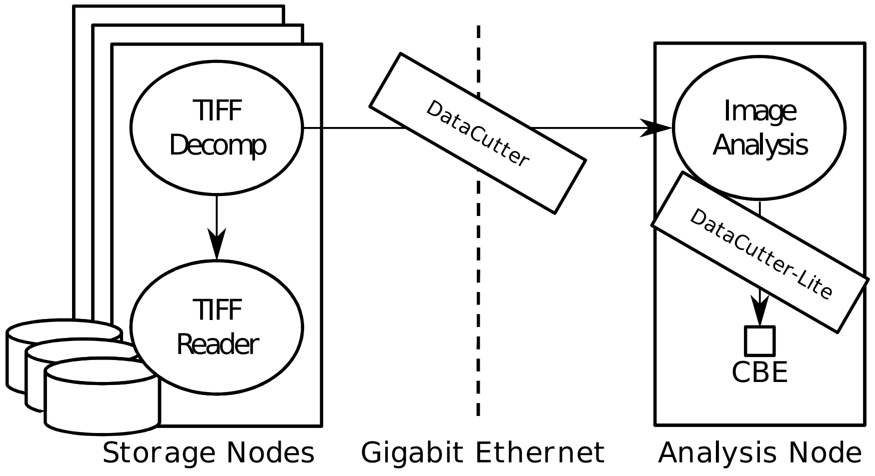
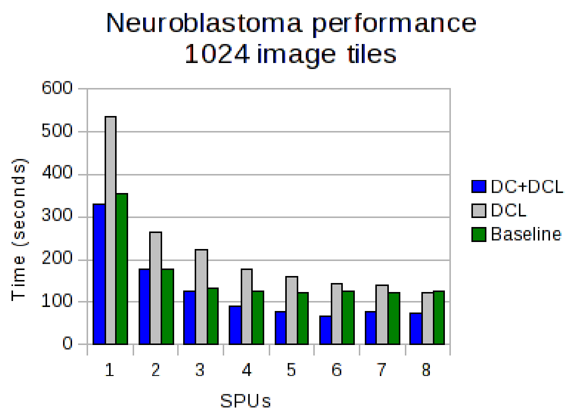


Figure 4.20: DataCutter and DataCutter-Lite mixed implementation

best speedups achieved are 2.2 for the DCL version and 2.7 for the baseline version. The DCL version does not read the decompressed TIFFs from an incoming socket, since this operation would require the use of mutexes in order to share the socket, and we have avoided this type of programming, since it is incompatible with our goal of designing a runtime system devoid of these types of details. As such, each TIFF is decompressed in the same thread which calls the DCL routines to analyze the image.

To solve the problem of insufficient TIFF decompression bandwidth, we have implemented a mixed DataCutter+DCL version of the image analysis application. Figure 4.20 shows the layout of the integration of DataCutter for internode communications and DCL for intranode executions. As such, Figure 4.21 shows the results when DataCutter is used to distribute the TIFF tile decompression stage among several computational nodes, and one CBE processor is used to analyze the tiles with its 8 SPEs. Unfortunately, OSC's CBE blades are currently only configured to run with Gigabit Ethernet, which limits the amount of help distributed nodes can give.



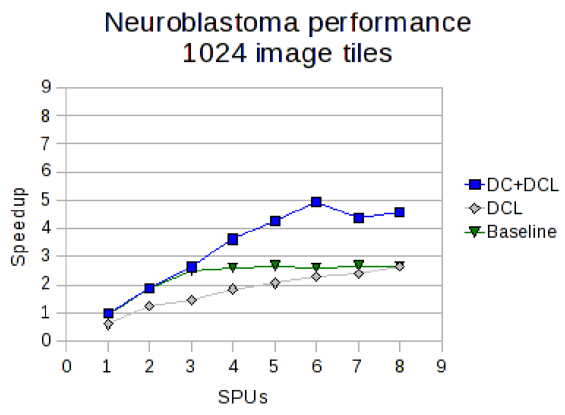


Figure 4.21: Execution times and speedups for biomedical image analysis application for 1024 image tiles

4.6 Summary

This work presented DataCutter-Lite (DCL), a fine-grained, component-based, filter-stream programming library and runtime engine. DCL is meant to allow application developers access to the new high-performance, multicore microprocessors available in the marketplace. We showed that the runtime engine is able to support high-bandwidth communications among the processor's cores without burdening the developer with low-level, architecture-specific instruction syntax. We showed that for applications with high communication to computation ratios, DCL does not incur an additional overhead. For applications with low CCR values, we showed that DCL scales as well as custom application implementations.

Chapter 5: Automatic Dataflow Application Tuning for Heterogeneous Systems

5.1 Introduction

The high-performance computing world is in the midst of a major paradigm shift. The increased on-die parallelism of multicore processors, and the increased programming flexibility of current high-computational throughput accelerator devices such as graphics processing units (GPUs) combine to make modern clustered supercomputers into hierarchical, heterogeneous systems [82, 55, 28]. Designing applications for such complex, heterogeneous systems is challenging, and few solutions are offered to these challenges by traditional parallel and distributed programming technologies such as pthreads and MPI. However, even with these challenges, recent research has shown that using GPUs and CPUs in concert can yield better performance than by simply using the accelerators [75, 106].

The *dataflow* programming paradigm [33] and in particular the *filter-stream* paradigm are uniquely suited to designing large-scale, data-intensive scientific applications for complex, heterogeneous, distributed computer systems. The simple abstraction exposed by dataflow programming and supported by dataflow runtime middleware systems, such as DataCutter [13] and Anthill [40], provides developers with an excellent solution for dealing with the complexities of modern scientific application design for distributed, heterogeneous supercomputers.

Most distributed programming paradigms (including filter-stream) leave to the developer the task of finding the right data decomposition granularity for their application. Since filter-stream programs operate on data in discrete chunks called *databuffers*, and since static analysis of distributed applications is difficult, the size of the databuffer must be tuned empirically, not only to find the "best" databuffer size for each processor type, but also to balance the competing demands of cache locality, network throughput, network latency, and overlap of communication with computation. Since different dataflow applications have distinct task layouts, communication patterns, and amounts of computation, this tuning step must be performed for every application. Further, the same application can have more than one optimal databuffer size, since the system configuration (including processor types, network topology, system size) can affect the choice. Using a sub-optimal databuffer size can significantly increase the overall application's execution time. Therefore, to help developers extract the best performance from their filter-stream applications, we should allow the middleware to tune the databuffer size automatically.

Traditional load-balancing techniques for dataflow applications include demand-driven solutions [13, 40] and work-stealing solutions [16], where dataflow tasks with higher processing rates request data more quickly, and process more of the databuffers in the application. However, with heterogeneous processors and a static databuffer size, the load can become unbalanced at the end of the application's execution, due to the longer execution times of slower processors. This situation is exacerbated when certain processors, such as accelerators, are much faster than others, such as CPUs. Worse, different system or application configurations can change the balance between network usage, databuffer queue length, and communication overlap.

To solve these problems, we present an adaptive technique for tuning the databuffer size for large-scale dataflow applications and a load-balancing technique based on modeling heterogeneous dataflow task processing rates and automatic workspace partitioning. By using a simple programming interface, developers can write dataflow applications that are automatically tuned for the best performance, where the load is nearly perfectly balanced across all of the available computational resources.

This adaptive technique has been implemented in the DataCutter [13] middleware, a runtime system supporting the dataflow programming abstraction. We experimentally validate the effectiveness of our approach on two real applications and two synthetic applications.

The rest of the chapter is organized into five sections. Section 5.2 discusses the differences between our intent and previous works. Section 5.3 presents the targeted application model in more formal terms. In Section 5.4, we present our algorithms for adaptive databuffer size tuning and automatic workspace partitioning and load balancing. Section 5.5 shows the results of experiments conducted with two microbenchmark applications, and two real-world, large-scale dataflow applications. In Section 5.6 we present our summary and discuss future work.

5.2 Related Work

There is a wealth of research related to our problem. ATLAS [111] uses compile-time performance tests to tune its implementations of linear algebra operations.

FFTW [42] tunes data structures and algorithms at runtime to efficiently perform FFT operations. Some researchers have discussed adaptive techniques for improving the performance of MPI collective communication operations [87]. Thus, automatic tuning for a variety of fine-grained kernels and for general communication patterns is not new; however, our approach begins to develop a runtime framework which can automatically tune and load balance general dataflow applications, provided they fit the application model.

Research into mapping and scheduling of static or dynamic programs onto heterogeneous resources also informs this work [10]. However, our work differs from the research in this field because the focus there is on the scheduling of tasks under various constraints, such as heterogeneous execution times and hardware capabilities. These algorithms do not attempt to subdivide tasks or data in any way; the applications are considered immutable input.

The Divisible Load Scheduling (DLS) model [44] is the closest related theoretical work. In DLS, a large amount of data has to be processed in parallel by multiple processors. The main concerns are "what fraction of the data should a processor process?" and "how to schedule the data communications?" Models in this area usually consider large time intervals so that any discrete effect in the computation can be omitted. Therefore, the models often consider the platform to be static and the buffer of the processors to be infinite. The beginning and ending of the computation are also frequently neglected. However, newer work such as [46] introduces heuristic solutions for scheduling communications and computation, including the collection of the results of the computation. Our work differs from this work in two main respects. DLS requires accurate knowledge of the system's computation and communication rates for scheduling. Our intent is to provide a real system for dataflow application development, without requiring developers to perform any tedious parameter discovery themselves. DLS also focuses on the steady-state, whereas real-world applications can perform all of their work without reaching any steady-state condition.

5.3 Application and System Model

Section 1.2.1 introduces the DataCutter middleware we use to implement our adaptive technique. In this chapter, we assume that the workload is finitely divisible, subject to minimum and maximum databuffer size constraints, which stem from application-specific constraints and hardware constraints such as memory size. Since the range of acceptable databuffer sizes can be large, and can dramatically affect the overall execution time of the application, the databuffer size is an important parameter to choose carefully (this will be shown in Section 5.5). We also assume that there is a monotonically increasing execution time for increasing databuffer sizes for all processor types. The simplest workload is 1-dimensional, and represents a simple reduction operation on a 1-d vector of input data. An image analysis operation which operates on 2-d images and produces a feature vector represents a 3-d work area. The work area definition simply needs to match the dataflow application's requirements.

This work focuses on appropriately choosing the databuffer size. Therefore we restrict our study to the following case. The application has a single input filter and a single output filter, respectively called the source and the sink. We assume that all the internal filters (i.e., all but the source and the sink) are replicable. The machine executing the internal filters is composed of a set P of processors which can be completely heterogeneous. The application is mapped to the processors so that each processor executes one copy of all the internal filters and no communication between the processors occur. Therefore, a databuffer will be processed exclusively by one processor (we will use the terms “filter” and “processor” interchangeably).

This model is not that restrictive. First, all the filters of an application may not be replicable, but the part of the layout which is replicable fits this model. Second, it might be interesting to exchange databuffers between the processors that execute the internal filters. But this is likely to saturate the communication network. Moreover, this constraint is common in scheduling algorithms that decide the placement of replicable filters (see [100]). Finally, this type of model fits a large number of applications such as scientific image analysis, audio/video compression, and numerical solvers.

5.4 Adaptive Algorithm For Work Partitioning

In this section, we will describe our adaptive databuffer size tuning and load balancing algorithm. There are two parts to our method. The first part is a performance model which tracks each databuffer's execution time. By continually tracking the execution time of databuffers in the system, we can choose the databuffer with the fastest processing rate. The second part of the method is a dataflow work partitioner. By using the performance model, the work partitioner streams databuffers to the downstream processors with the aim of balancing the load.

The programming interface for the databuffer processing rate performance model contains only three functions and does not use application-specific information, allowing it to be used with any application. Calls to *add_data_point*() add new data to the performance model. Meanwhile, $get\_next\_buffer\_size()$ returns the next suggested databuffer size and $best\_processing\_rate()$ estimates the processing rate for a processor. We use a simple prediction technique. For a given databuffer size, only the latest information is kept in the performance model. The fastest buffer size is the one that maximizes the processing rate (buffer size divided by computation time). As a bootstrap step, this prediction technique will initially suggest every possible buffer size at least once. Therefore, the first calls to the $get\_next\_buffer\_size()$ function sweep the whole range of possible values for the buffer size. After this bootstrap period, the performance model will return the buffer size which gives the highest expected processing rate.

The range of possible databuffer sizes returned by $get\_next\_buffer\_size()$ during bootstrap is given by a databuffer sizing function. We have used a simple function, which doubles the size of the dimensions of the previously returned tile, until the maximum size is reached. Thus, the developer only needs to provide the minimum and maximum databuffer sizes. These values are affected by the specific processing requirements of the application, as well as the memory and network constraints of the target hardware system. These values are not overly burdensome to request. Additionally, users can provide their own databuffer sizing function, if their application's work area partitioning requirements are not met by the default function.

We experimented with using a linear least-squares regression to provide a linear model for databuffer processing rates, but found that in the face of varying system load, the linear equation produced by the regression was unsuitable for use in this context. In particular, during the bootstrap stage of the dynamic load balancing algorithm, unstable estimated execution times could dramatically damage the overall execution time of the application, by suggesting sub-optimal databuffer sizes.

Figure 5.1 shows the main Adaptive Partitioning Controller (APC) pseudocode. APC relies on estimated processing rates for all of the processors in the system. To bootstrap, APC starts by sending to every processor a single, minimum-sized databuffer to initialize the performance model.

Then, using a simple round robin scheme, APC sends databuffers to each processor having less than T time units of work queued. After bootstrap, the size of the databuffer sent to a processor is the one with the fastest expected processing rate. APC allocates work to each processor such that the system achieves perfect load balance. When the remaining work is low, and the databuffer size suggested by the performance model is too large, APC will send a smaller databuffer (even if the databuffer sizing function is still in the bootstrap phase). By doing so, we prioritize load balance over processing rate at the end of the application's execution. Frequently (exactly after receiving a reply from all the processors), the amount of work each processor should be given is updated using $PARTITION()$; this allows APC to react quickly to changing system conditions.

The initial workload is in *queue*. For our targeted application type, the queued workload represents the amount of work to be performed by the application. Since APC is oblivious to application specifics, it is up to the developer to choose a work space which can be interpreted in the application domain. For instance, in a 2-d image analysis application, the queued workload would consist of a set of (r, c) pairs representing several images of a certain number of rows and columns. APC will tile these (r, c) pairs into (r, c, x, y) tuples, which represent contiguous subtiles of a certain number of rows and columns, and starting at a certain (x, y) point in the original image. The analysis application will then process these image subtiles.

Currently, APC supports divisible workloads of the type discussed above. APC automatically tiles these work areas by initially dividing the entire workarea into the maximum allowed tile size; smaller subtiles are made out of larger tiles hierarchically. For example, the applications discussed in Section 5.5 use tiles of size 32 x 32 up to 1024 x 1024, in powers of 2.

```
1:  function APC($T, queue$)
2:      for all processor $p \in P$ do
3:          $size$ = get_next_buffer_size($p$)
4:          $tile$ = $queue$.fetch($size$)
5:          send_work($p$, $tile$)
6:      loop $|P|$ times
7:          receive ($p$, $tile$, $timing$, $block = true$)
8:          add_data_point($p$, $size$, $timing$)
9:      $replied = P$
10:     $p = 0$
11:     while !$queue$.empty() do
12:         if  $replied = P$ then
13:             $alloc\_work$ = PARTITION($queued\_work$, $queue$)
14:             $replied = \emptyset$
15:         $block = (\forall p' \in P, \frac{queued\_work[p']}{\text{best\_processing\_rate}(p')} > T)$
16:         if receive ($p'$, $tile$, $timing$, $block$) then
17:             add_data_point($p'$,$size$, $timing$)
18:             $replied = replied \cup \{p'\}$
19:             $queued\_work[p']$ -= $size$
20:         if $\frac{queued\_work[p]}{\text{best\_processing\_rate}(p)} < T$ then
21:             $size$ = min ($alloc\_work[p]$, get_next_buffer_size($p$))
22:             $tile$ = queue.fetch($size$)
23:             send_work($p$, $tile$)
24:             $queued\_work[p]$ += $size$
25:             $alloc\_work[p]$ -= $size$
26:     $p = (p + 1) \bmod |P|$
```

Figure 5.1: Adaptive Partitioning Controller Pseudocode

Figure 5.2 shows pseudocode for the $PARTITION()$ function which provides the amount of work each processor should be allocated. The function calculates the estimated amount of time each processor's queued work will take to complete; it uses this information to allocate to each processor enough work to last max_time. It then distributes the remaining work to the processors proportionally to their processing rate. If $PARTITION()$ runs out of work before all of the processors are given sufficient work to match the longest running processor, then that longest running processor will set the application end time anyway; by not actually queuing more than T time units of work to any processor in Figure 6.3, we ensure the load imbalance is no worse than T.

```
1: function PARTITION($queued\_work$, $work$)
2:     $procrate = \sum_{p \in P}$ best_processing_rate($p$)
3:     for all processor $p \in P$ do
4:         $queue\_time[p] = \frac{queued\_work[p]}{\text{best\_processing\_rate}(p)}$
5:     $max\_time = \max_{p \in P} \{queue\_time[p]\}$
6:     for all processor $p \in P$ do
7:         $alloc[p] = \min\{(max\_time - queue\_time[p]) \times$
   best_processing_rate$(p), work\}$
8:         $work = work - alloc[p]$
9:     for all processor $p \in P$ do
10:        $alloc[p] += \frac{work \times \text{best\_processing\_rate}(p)}{procrate}$
11:    return $alloc$
```

Figure 5.2: Adaptive Dataflow Work Partitioner Pseudocode

5.5 Application Experiments

5.5.1 Experimental setting

To validate the performance of our adaptive databuffer tuning and load balancing algorithm, we used two microbenchmark applications and two real-world data-intensive applications. All these applications match our application model (see Section 5.3).

Our first real-world application is a biomedical image analysis (BIA) application [97]. In this application, highly magnified digital images of specially prepared biopsy tissue samples are processed using cooccurrence matrices and operators called linear binary patterns to determine the texture of the tissue found in the biopsy sample. The texture determination affects the prognosis for the patient. This is a time-consuming, error-prone process for human slide readers to perform, and the goal of this application is to reduce error and increase analysis throughput. The digitized images can be over 100K x 100K pixels, and many slides are often analyzed as part of a patient's study. The processing requirements are a good match for the GPU's manycore architecture and high memory bandwidth; hence, speedups of up to 45x as compared to a reference CPU implementation can be obtained. Our dataflow implementation includes a set of storage nodes whose sole job is to store the full decompressed image in memory and serve subtiles to requesting filters. We replicate the stored image data over several "frontend" nodes to provide sufficient memory and network bandwidth to serve the "backend" processing nodes.

Our second real-world application is a synthetic aperture radar (SAR) imaging application [56]. By processing the input samples through a radar backprojection step, we can create images from input radar return data. The simple backprojection algorithm used here is a triple loop where each input vector of radar return data is applied to each pixel of the output image. Because backprojection is a pleasingly parallel application, and because the filter execution times are more dependent on the number of output pixels created, we partition the output space into tiles. The inputs to the application are small, and therefore we broadcast them at the beginning of the application execution. The computation style is very well-suited to the GPU, and so we see enormous speedups (up to 175x) when compared to a simple, single CPU core implementation of the algorithm.

The two microbenchmark applications are intended to give our application-space more coverage in terms of different processing rate ratios between the CPU cores and the GPUs. Since some processing functions are better suited than others to specialized acceleration processors like GPUs, there can a wide range of speedup values for GPU code, as compared to a CPU core. Thus, we wanted to ensure that our techniques work for a wide range of speedup values. Therefore, our microbenchmark codes fill in the smaller end of the spectrum, where the GPU is only 5x and 10x faster than one CPU core, respectively. These slower speedup values are found in applications which require complex data structures, which have a large amount of conditional code execution, or which have an irregular data access pattern. Examples of applications with lower speedup ratios which fall within our application target area can be found in [47] and [101]. These applications would get the most benefit from using both CPUs and GPUs.

Both of the microbenchmark applications are based on the real-world SAR application. To provide a realistic benchmark, we have kept the input data and output data requirements the same as SAR. Further, the replacement GPU filter calculates its processing time for each databuffer by using a constant 10 millisecond wait time as well as a proportional processing time according to the work area described by the databuffer. No code actually runs on the GPU, and the output data produced by the replacement GPU filter is not intended to be correct, as compared to the SAR application. Inside the busy wait "GPU" filter, we simply used a call to `usleep()`.

Table 5.1 shows the single CPU core and single GPU execution times for each of the four applications and for each of the databuffer sizes used in our testing.

Table 5.1: Single tile execution times in milliseconds for four applications

App.	**Tile size**	**CPU core**	**GPU**	**Speedup**
5:1	32 x 32	206	52	4.0
μbench-	64 x 64	840	174	4.8
mark #1	128 x 128	3 306	666	5.0
	256 x 256	**13 045**	2 633	5.0
	512 x 512	52 234	**10 497**	5.0
10:1	32 x 32	205	31	6.5
μbench-	64 x 64	831	92	8.9
mark #2	128 x 128	3 306	338	9.8
	256 x 256	**13 059**	1 322	9.9
	512 x 512	52 339	**5 254**	10.0
Bio-	32 x 32	0.37	1.58	0.2
medical	64 x 64	1.31	1.86	0.7
Image	128 x 128	5.23	2.07	2.5
Analysis	256 x 256	**20.22**	2.64	7.7
	512 x 512	81.00	3.80	21.3
	1024 x 1024	354.25	**7.82**	45.3
SAR	32 x 32	205	18	11.1
Imaging	64 x 64	820	19	42.1
	128 x 128	3 256	36	88.7
	256 x 256	**13 023**	91	142.5
	512 x 512	52 449	**324**	161.6

Our experimental testbed was the Ohio Supercomputer Center's Glenn cluster, which has recently been upgraded with Tesla Quadroplex 2200 S4 rack-mounted GPUs. Each node in our cluster has dual-socket quad-core AMD Opteron 2380 processors running at 2.5 GHz, 24 GB of system memory, and two Quadro FX 5800 GPUs, each with 4 GB of memory. The nodes are connected with 20 Gps Infiniband. The operating system kernel is Linux 2.6.18 with glibc 2.5. We used GCC 4.1.2 with -O3 optimizations. The GPU programming framework is CUDA toolkit 2.3.

The Adaptive Partitioning Controller (APC) has been implemented in DataCutter [13]. We compare APC against the Demand-Driven (DD) controller built into DataCutter which uses a user-configurable static databuffer size and controls the rate at which databuffers are sent to each processing filter by how quickly they are processed there. The tile sizes chosen for the experiments are the ones reported in Table 5.1. This table presents the computation time in milliseconds for both a single CPU core and a single GPU as well as the speedup of the GPU over one CPU core. The time shown in bold font is the tile size that achieve the highest processing processing rate for each processor type in each application. It is worth noting that on the microbenchmarks and the SAR imaging application, the choice of the tile size for the CPU impacts the processing rate by less than 1%. The case of the BIA application is different. Indeed, the CPU's processing rates for tiles from 32x32 to 512x512 are similar, whereas using 1024x1024 tiles is 10% slower. Meanwhile, the choice of the tile size is always crucial when running filters on the GPU.

5.5.2 Results and analysis

The running time of the experiments are reported in Figures 5.3, 5.4, 5.5, and 5.6. Each figure corresponds to one application and presents execution times using different distribution policies while varying the system configuration. Depending on the figures, the execution times of policies which did not fit into the range of the y-axis for the chart are omitted to improve the figure clarity. The execution times shown in the figures are averages of 5 runs. The average standard deviation for the two microbenchmarks is less than 1%, while the maximum is over 5% (usually due to one or two experiment configurations with widely variable results). SAR has an average standard deviation of 1.5% and a maximum value of 6.4%. For BIA, the average standard deviation is 3.0% and the maximum value (except for one very variable system configuration) is 6.4%. For the two microbenchmarks, we analyzed 10 1K x 1K images. For the BIA application, we analyzed 20 25K x 25K images, while the backprojection application analyzed 100 2K x 2K images.

Figures 5.3 and 5.4 show the results of our two microbenchmarks, with a 5:1 and 10:1 GPU to CPU core processing rate ratio, respectively. The first thing to notice is that the optimal databuffer size is different for different system configurations. For both microbenchmark applications, the fastest execution times for the homogeneous GPU-only configurations are found when using the largest tile size. When adding CPU filters to the system configuration, we would expect the application execution time to decrease. This is not the case for the 512 x 512 tile size, because the runtime of a 512 x 512 tile on the CPU takes over 50 seconds (see Table 5.1). Therefore, we must use a smaller tile size to improve the load balance. APC handles this automatically, always coming very near the fastest DD configuration's execution time. For a few system configurations, APC even beats all of the DD results.

APC is competitive with all of the demand driven scheme's different configurations. A well-tuned demand driven scheduling policy is very effective at executing dataflow applications in homogeneous cases. Therefore, the extra initial bootstrap overhead and the work area fragmentation of APC makes it difficult to beat those results. However, in all cases, APC comes very close to the best DD result. For both of the microbenchmarks, 10 images do not represent a large amount of work for the system; there are only 40 of the largest allowed 512 x 512 databuffers. While this may be a reasonable amount of work in the application domain, this represents the worst-case scenario for an adaptive system. There is not much time to perform the load balancing before the entire application exits.

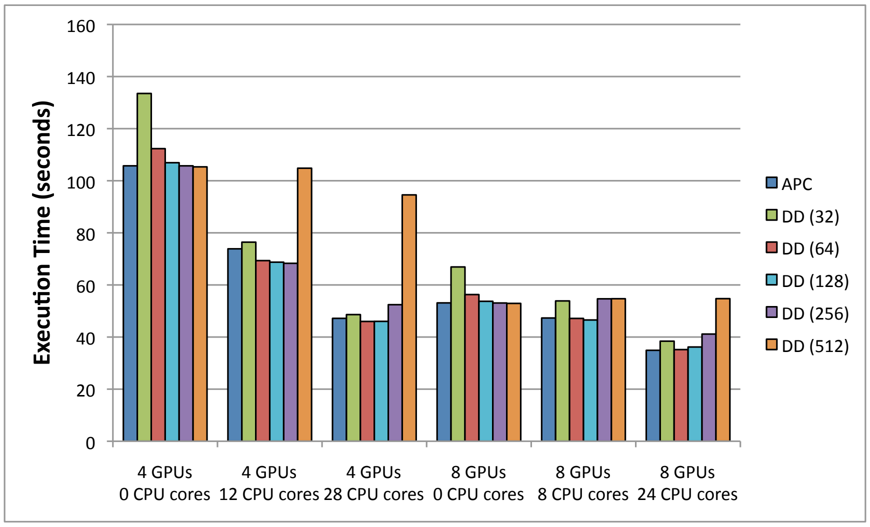


Figure 5.3: Microbenchmark #1 with 5:1 GPU:CPU core speedup; 4 Nodes; 10 1K x 1K Images

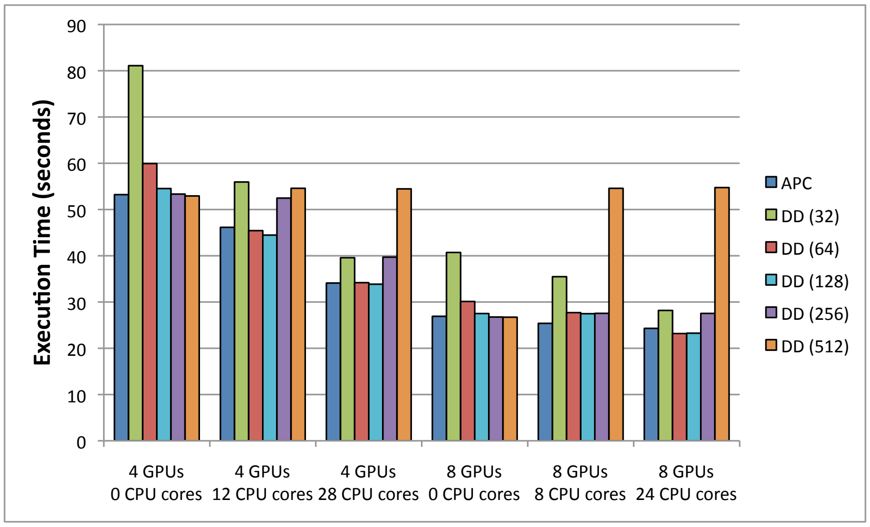


Figure 5.4: Microbenchmark #2 with 10:1 GPU:CPU core speedup; 4 Nodes; 10 1K x 1K Images

Figure 5.5 shows the results of running the BIA application on 4 nodes with various system configurations. The system configurations listed are only the nodes devoted to processing; as discussed previously, we use 4 nodes for decompressed image storage, in order to provide sufficient memory and network bandwidth. For this experiment we analyzed 20 25K x 25K images. Smaller tile sizes than 512 are possible, but these configurations give extremely poor performance due to memory and network bandwidth contention, and as such are not shown (even though the single-tile CPU fastest processing rate is for the 256 x 256 tile size).

The values shown in Table 5.1 show that the CPU and GPU have different optimal tile sizes; the CPU reaches its maximum processing rate at a tile size of 256 x 256, while the GPU's best performance is at 1024 x 1024. Therefore, a static choice of tile sizes is insufficient. Certainly, using two tile sizes would be possible, and would give each processor type its preferred tile size. However, using an incorrect tile size for the

GPU could increase the computation time by a significant amount, as can be seen from Table 5.1. Further, statically partitioning the entire work area into different groups, from which the two tile sizes could be partitioned would be insufficient if the system configuration were to change. Our APC algorithm shows that by conducting this databuffer tuning and load balancing at runtime, we can usually beat even the optimal demand-driven scheme.

Also, in BIA, the demand-driven scheme is unable to beat the performance of the homogeneous GPU-only case by adding CPU cores. However, while the extra CPU cores do not improve the application performance much while using APC, APC achieves better performance than demand-driven schemes for the 4 GPU 12 CPU core and 8 GPU 8 CPU core configurations. Additionally, for those system configurations where the optimal DD performance beats APC, it is usually by less than 1%. The small improvement for APC when adding CPU cores is due to the high performance for this application on the GPU; when the GPU is 40x faster than a single CPU core, 40 additional CPU cores are needed to double the application's performance, even while neglecting the additional input partitioning and output aggregation overhead.

As you can see from Figure 5.6 and from Table 5.1, the huge speedup of the GPU as compared to a CPU core in the SAR image formation based on backprojection means that extra CPU cores do not improve the execution time much. Even in the face of this challenging situation, APC works well. The demand driven scheme shows a poor load balance, due to the fact that it is oblivious to the amount of time each databuffer takes to process in different processor types. While the overhead for adding many CPU cores is high in APC as compared to the GPU-only homogeneous cases, its performance degrades far more gracefully than the demand driven scheduling policy.

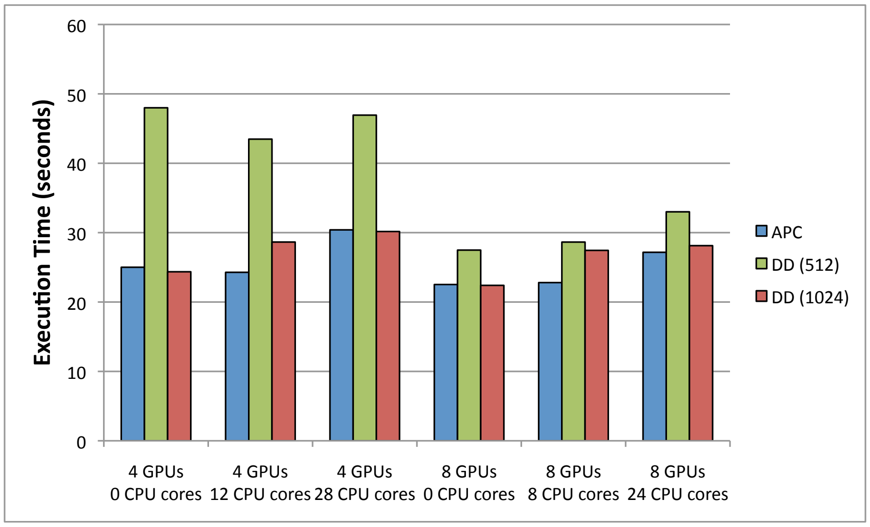


Figure 5.5: BIA: Biomedical Image Analysis; 4 Nodes; 20 25K x 25K Images Analyzed

Tables 5.2 and 5.3 respectively show the number of tiles processed of each size, and the total work area processed by each tile of each size for APC on the SAR application. Table 5.2 shows the number of tiles processed by each filter type (in total across all 4 nodes); we can see the total number of tiles processed by the system increase when we add CPU cores in each configuration. Meanwhile, Table 5.3 shows the amount of work processed per tile size, also broken down by filter type. Here, we can see than in all system configurations, the GPU processes more than 90% of the total work area. However, we can see that even though our overall execution time does not degrade significantly when adding CPU cores in a non-optimal fashion (see Figure 5.6), we are actually scheduling work to the CPU cores in such a fashion as to not damage the execution time badly. In fact, it is more likely the fragmentation of the work area into smaller tile sizes that harms the overall execution time.

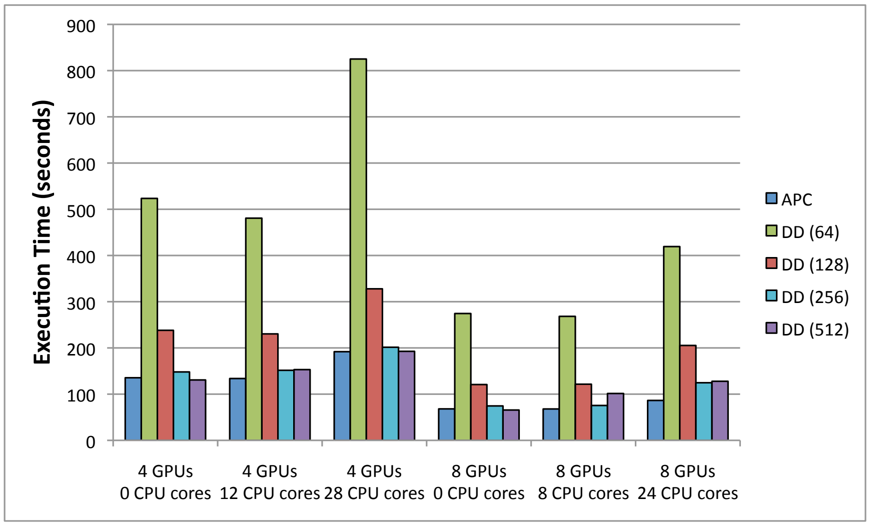


Figure 5.6: SAR: Synthetic Aperture Radar Image Formation; 4 Nodes; 100 2K x 2K Images Formed

To efficiently compare the various demand-driven configurations and APC, we use *performance profiles* [35], a visual tool for comparing various *methods* over a large set of *test cases* with respect to some *metric*. The metric we use in the following profile plots is the application execution time.

Each performance profile plot (in Figure 5.7) shows the probability that a specific demand-driven configuration or APC gives results within some value τ multiple of the best result reached by all of the load-balancing schemes. The higher the probability of small τ values, the more preferable the method is. For example, the curve DD128 includes the point (2, 0.7) which means that on 70% of the tested cases, the execution time of the DD128 scheme was no more than 2 times slower than the best (fastest) obtained by any method.

The DD-dev line represents what a developer would likely choose as the static databuffer size for use in a demand-driven scheme after examining Figures 5.3 - 5.6.

Table 5.2: Aggregate number of tiles computed by each processor type while varying the system configuration

Proc. Type &	Number of GPUs / Number of CPU cores					
Tile size	4/0	4/12	4/28	8/0	8/8	8/24
GPU 32x32	12	8	9	24	21	30
GPU 64x64	13	5	17	26	19	9
GPU 128x128	12	10	13	8	25	24
GPU 256x256	4	15	16	24	14	140
GPU 512x512	1598	1564	1512	1593	1584	1527
CPU 32x32	0	12	39	0	27	62
CPU 64x64	0	22	99	0	25	60
CPU 128x128	0	182	307	0	73	265
CPU 256x256	0	19	92	0	22	74
CPU 512x512	0	15	39	0	0	0
Sum	1639	1852	2143	1675	1810	2191

Each application would get its own tile size, but the same tile size would be used for all system configurations. For the two microbenchmark applications, we chose 128x128 as the tile size. BIA uses the 1024x1024 tile and SAR uses 512x512.

What the performance profile shows us, is that of all of the load-balancing schemes, APC is the most likely to give desirable results. None of the other statically sized demand-driven configurations gives reasonable results across all of the system configurations and applications, bolstering our claim that tuning the databuffer size for each application is necessary. However, APC can schedule application executions within 1.1 times the fastest result more than 90% of the time and within 1.25 times the fastest result found 100% of the time, meaning that developers can now leave the tuning of their applications to the runtime system itself, and still be assured of good performance.

Table 5.3: Aggregate work area computed by each processor type while varying the system configuration

	Number of GPUs / Number of CPU cores					
	4/0	4/12	4/28	8/0	8/8	8/24
GPU	100%	98%	94.8%	100%	99.3%	95%
CPU	0	2%	5.2%	0	0.7%	5%

5.5.3 Sum up of experimental results

This section highlights the important points from the experimental results.

Using CPU cores and GPUs cooperatively can be useful. For both of the microbenchmark applications, with their comparatively low speedup values, the addition of CPU cores to help the GPU perform the computation lowered the overall application execution time.

Static work partitioning leads to a poor load balance. Because of the wide variability in the processing rates for GPUs and CPUs, a static work partitioning and tile size will yield a load imbalance. For instance, on the microbenchmarks and SAR, the load balance is terrible when the optimal GPU tile size is used.

Static work partitioning requires a difficult tuning phase. To determine the correct tile size to use for a specific dataflow application, a specific amount of work, and a specific system configuration, developers must conduct a search across all of the possible tile sizes. Unfortunately, this tile size is not necessarily consistent across system configurations, forcing developers to search again if the system configuration changes (based on system availability, hardware upgrades, or code changes). Further,

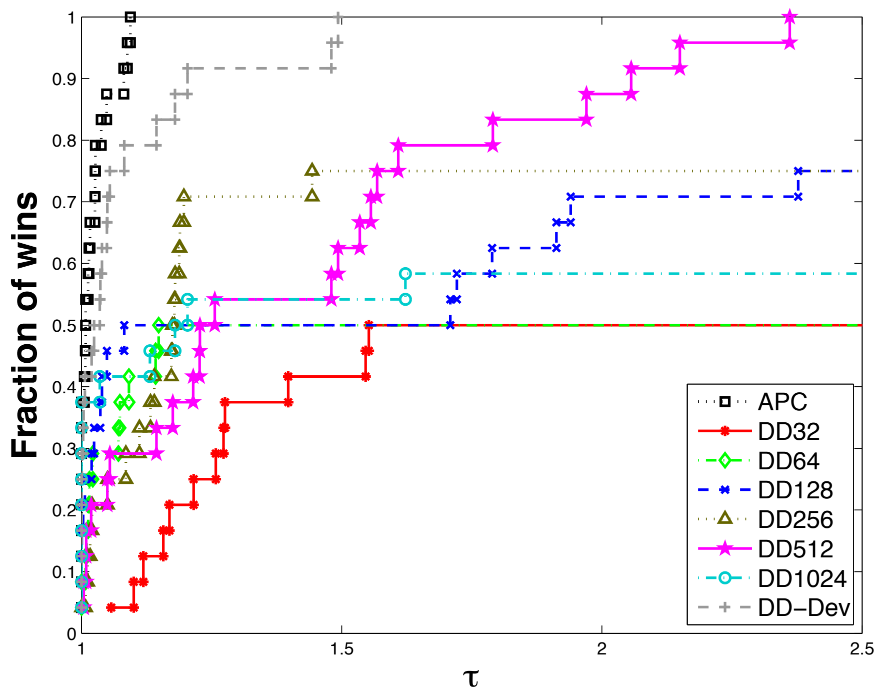


Figure 5.7: Performance profile of APC and DD with a fixed tile size for all the system configurations for the four applications

there are too many interactions at work to allow a static analysis of the dataflow application to work well.

To address these issues, we propose *APC* which *does not require tuning parameters*. Moreover, *APC leads to an efficient execution.* The performance profiles show that APC will overall achieve better runtimes than demand-driven schemes manually tuned by the application developer.

APC performs reasonably on the worst-case scenario. SAR is really a nightmare application for cooperatively using CPUs and GPUs. The application performs so well on the GPU so as to make any possible benefit from additional CPU cores be canceled out by the extra overhead required to feed them with work. APC performs up to 31% better than the fastest demand-driven scheme, and no more than 4% worse across all of the system configurations.

APC conducts a more efficient execution than any static work partitioner. For applications with difficult-to-predict tile speedup ratios, such as BIA, APC is able to handle the extra complexity. Because APC is able to select a different tile size for the each architecture, it is able give each processor tiles which maximize their processing rates. This allows APC to give the best overall processing rate among the three 4 GPU configurations for APC.

5.6 Summary

In this chapter, we have presented our method for adaptively tuning the databuffer size and dynamically balancing the load in dataflow applications. By conducting the workspace partitioning in the dataflow runtime system itself, we can adaptively tune the databuffer size and balance the load among heterogeneous processors with drastically different processing rates. This algorithm solves two problems with the dataflow programming paradigm, one of which was previously ignored or left to the developer to accomplish with a tedious tuning process.

Chapter 6: Improving Performance of Adaptive Component-Based Dataflow Middleware

6.1 Introduction

The high-performance computing world has undergone a major paradigm shift. Supercomputers are now inherently hierarchical and heterogeneous [82, 55, 28] due to the increasing degree of on-die parallelism in microprocessors, as well as the ever-increasing programming flexibility afforded by current accelerator devices such as graphics processing units (GPUs). Indeed, as of April, 2011, all of the five fastest supercomputers in the world use multi-core processors[2], while three of the five also use Nvidia GPUs. However, while the raw performance of supercomputers has continued its steady increase, programmability of such systems is suffering; indeed, programming such systems can be a very tedious task even for expert programmers. The heterogeneity of the systems in terms of processing capability and network infrastructure make their efficient use a major technical challenge. However, even with these challenges, recent research has shown that cooperatively using CPUs and GPUs is an important goal [55, 75, 106].

[2]According to the Top 500 `http://www.top500.org/`

Traditionally, applications have been developed with various low-level software tools for high-performance computing (HPC) systems. MPI provides a portable, high-performance message passing semantic for distributed-memory HPC applications, with certain implementations providing support for high-bandwidth networks. With the release of Nvidia's Compute Unified Device Architecture (CUDA)[3], and later with the standardization of OpenCL[4], GPUs were widely capable of performing general purpose computation, without requiring developers to learn to overload graphics APIs such as OpenGL. Multi-core, multi-processor machines are commonly used through POSIX threads, OpenMP, and more recently shared-memory frameworks such as Intel's Threading Building Blocks[5]. These tools are important and useful to the expert programmer, but designing applications solely with these low-level building blocks remains a complex task.

To lower the bar to programming efficient applications for modern HPC systems, high-level programming frameworks must provide a simple but powerful programming API, must easily support processor heterogeneity, and must provide automatic load-balancing. Unfortunately, most distributed programming frameworks leave to the developer the task of finding the right data and task granularity for their application. Since distributed applications targeted to heterogeneous systems have many variables (such as processor speeds, node configuration, and network topology) which affect their runtime performance, the size of the data partitions must be tuned empirically, not only to find the "best" size for each processor type, but also to balance the competing demands of cache locality, network throughput, network latency, and

[3]http://developer.nvidia.com/object/cuda.html

[4]http://www.khronos.org/opencl/

[5]http://threadingbuildingblocks.org/

overlap of communication with computation. Using a sub-optimal partitioning can significantly increase the overall application's execution time. Therefore, to help developers extract the best performance from their applications, we should allow the middleware to partition the data automatically.

The *Adaptive Partitioner Controller* (APC), presented in [57] and Chapter 5, simultaneously balanced the load and tuned the data partition size for applications running on distributed, heterogeneous systems; it was implemented in the Data-Cutter middleware [13]. However, APC has some shortcomings. Using a single APC controller enables a global view of the system, but this centralized mechanism suffers from scalability issues for large number of processors. Using multiple APC controllers achieves higher scalability, but each APC performs its decision independently, leading to potentially high load imbalance. Additionally, by design, APC solely focuses on efficiently using the processors, neglecting the network. In this chapter, we address the shortcomings of APC by shifting the focus of APC+ to take both processor and network performance into account. One controller runs on each node and is only responsible for scheduling tasks on the local, on-node processors. It still enables good performance modeling and utilization of the local processing units, while global load balance is achieved by a work-stealing mechanism. Finally, the data transfers are explicitly taken into account by the use of a dedicated storage layer, which enables high network performance. These improvements allow APC+ to offer enhanced scalability and utility.

In this paper, we will compare our technique with other state-of-the-art high-level programming frameworks in terms of their programming API, their support of modern multi-core and accelerator processors, their load-balancing technique, and the level to which data partitioning impacts the performance of runtime applications. We have chosen to use three general-purpose distributed programming frameworks as point of reference by which to judge our technique: DataCutter [13] with Demand-Driven load-balancing, a coarse-grain dataflow framework, KAAPI [45], an asynchronous task execution framework based on the Athapascan-1 language [43], and MR-MPI [88], a MapReduce framework using an MPI communication back-end.

We will focus on three applications, whose computation occurs in a finitely-divisible workspace, namely Synthetic Aperture Radar [56], Biomedical Image Analysis [97], and Black-Scholes[14]. This model is common in real-world applications and exposes the simplest model of parallelism. Each application has different characteristics: they differ in initial data distribution, task execution time profiles, and CPU/GPU speedups. To further understand the performance of the different middleware, we designed an application where we can tune the ratio between the size of the input and the amount of computation.

The rest of this paper is organized as follows. First, we describe the three comparison programming frameworks and their specific instances in Section 6.3, while Section 6.4 presents our proposed adaptive application programming framework. Section 6.5 describes the three test applications used in the experiments. The programming APIs of the frameworks and important application optimizations are discussed in Section 6.6. Then we will present an experimental study in Section 6.7, and finally conclude in Section 6.8.

6.2 Related Work

Software runtime systems which adapt the work partitioning between heterogeneous processors on a single node are closely related to our work. Qilin [75] conducts a static work partitioning between a multicore CPU and a GPU by initially executing training runs with various work sizes for each processor type, while Harmony [34] adaptively subdivides the application tasks to the CPU and a Field Programmable Gate Array at runtime. Both of these systems use the CPU and an accelerator in concert to achieve faster processing than by using either alone. GPUSs [6] is an OpenMP-like programming framework which automatically creates parallel code for multiple GPUs from annotated C code. By using knowledge about the data locations, tasks can be scheduled to run on the correct GPU without excessive data movement. StarPU [5] is a single-node dataflow programming framework and runtime engine which uses work-stealing to balance the workload. All of these systems only target single computational nodes, whereas our target is to handle distributed resources.

The field of middleware runtime systems has a long history of developing techniques for balancing the load of distributed applications at runtime. River [3] uses a distributed queue and a simple load balancing technique for producer-consumer application task relationships. Anthill [106] features algorithms for scheduling dataflow applications where the amount of work is not known prior to runtime. Coign [59] focuses on partitioning the dataflow graph to perform a static runtime data decomposition, whereas Cilk [16], Charm++ [64], ACDS [62], and TelegraphCQ [23] perform automatic load balancing as well as application partitioning. Capsules [76] allows developers to tune their application's task and data granularity with a simple parameterization scheme. More recently, MapReduce frameworks for heterogeneous systems such as Merge [74] have been designed. In these systems, a manual data-granularity tuning step must be performed to achieve the best application performance. Our technique automatically performs this tuning step at runtime. Thus, our work is complementary to the research in this field, and indeed, could be directly implemented in many of these dataflow runtime systems.

The family of Partitioned Global Address Space (PGAS) languages such as Titanium [114], X10 [24], and Chapel [22] are powerful languages for programming high-performance distributed applications. However, all of these languages leave the important load-balancing problem up to developers to solve, albeit with a concise programming methodology.

There are many other instances of runtime engines we could have chosen to include in our comparison. Cilk [16] is an obvious choice, due to its powerful work-stealing runtime engine, and although a distributed version of Cilk, called Cilk-NOW [17] (Network Of Workstations) is referenced in the literature, it is not available for use. The most famous publicly available MapReduce implementation is Hadoop[6], but its focus is inapplicable for our work; Hadoop is an enterprise-grade framework designed for out-of-core operation and fault-tolerance, and relies heavily on a parallel file system which cannot be bypassed. Charm++ [64] is an object-oriented distributed programming framework which associates computation to specific object instances, and migrates these objects to balance the computational load. However, adding Charm++ to our discussion would not increase our coverage of the distributed runtime engine space significantly beyond what we already present. While Charm++

[6]`http://hadoop.apache.org/`

provides a number of load-balancing techniques, including a measurement-based technique, a manual tuning step to determine the optimal data domain decomposition is required. Further, well-tuned demand-driven applications are already close to optimally scheduled [57].

Additional related work in autotuning can be found in Chapter 5.

6.3 Distributed Programming Frameworks

6.3.1 DataCutter: Component-Based Dataflow

DataCutter is discussed in detail in Section 1.2.1, although we will restate important points about the load-balancing mechanism and support for heterogeneity here, for completeness.

Load-balancing mechanism

The default DataCutter load-balancing technique is Demand-Driven databuffer distribution. It balances the load between a number of data consumers, according to their processing rate. By only sending databuffers to each consumer when they request more work, consumers which work faster will get more work to process.

The Demand-Driven mechanism is initialized by each producer sending one databuffer downstream to each data consumer. Data consumers will send a request for more data to a producer when they remove a databuffer from their incoming queue. Consumers will request data from the very upstream filter which produced the consumed databuffer. One can initially send more than one databuffer in order to hide the latency of the send-demand/reply-data handshake in cases where the processing time of one databuffer is insufficient. To further allow intra-node load-balancing in

DataCutter, copies of a filter allocated on the same node use a shared queue, and pop databuffers from this queue, instead of a private queue.

Support for heterogeneity

The filter-stream model is ideal for programming for heterogeneous processor types, because the architecture-specific details are hidden inside the filter. Provided the same data interface is used by two filters, they can be used interchangeably. Filter implementations can be specialized for dedicated architectures such as the Cell processor [54], GPUs [55] or SMPs.

6.3.2 KAAPI: Asynchronous Task Execution

KAAPI [45] is a programming framework tightly integrating the Athapascan-1 [43] language and a runtime system for the development of distributed applications. The Athapascan-1 programming language, through the use of C++ templates, allows users to describe the exact data dependencies between tasks in a dataflow graph. The use of commands to asynchronously spawn tasks at runtime and the use of explicit *shared variables*, which describe the data flow, allows the application's task graph to be created at runtime and to be unfolded in a dynamic, and data-dependent fashion.

In order to develop an efficient schedule of how to map this dynamically-created dataflow task graph onto distributed resources, Asynchronous Task Execution frameworks in general, and the KAAPI runtime system in particular, construct at runtime a lexicographic ordering of the tasks to be completed, and use the data dependency information from the user program to determine which tasks are ready to execute. The middleware distributes the *ready* tasks onto the processors automatically using a work-stealing strategy.

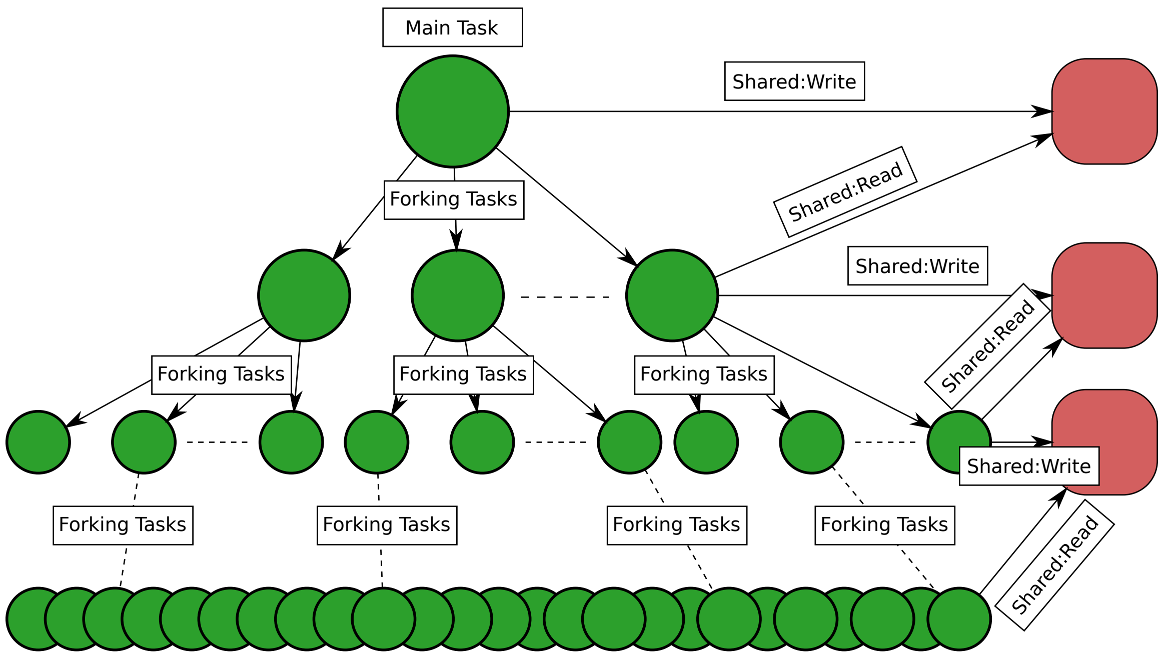


Figure 6.1: Example Asynchronous Task Forking and Shared Data Access

Figure 6.1 shows a general example of the power of KAAPI's runtime execution of asynchronously forked tasks, and the implicit ordering of task executions by shared data access. Since developers explicitly define how all shared data objects are accessed (in terms of read, write, or read/write semantics), the runtime engine can keep track of which tasks are ready to execute.

Load-balancing mechanism

KAAPI's load balancing is achieved through work-stealing [16]. Each processor involved in the computation has its own thread which manages its stack of tasks to complete. When a processor runs out of tasks to complete, it will attempt to steal tasks from a random processor. The victim processor will then yield some portion of its tasks to the thief processor. Provided that tasks and the processors' processing speeds are homogeneous, this will achieve excellent load balance. If processing speeds are proportional (heterogeneous, but with a constant scaling factor), then this work-stealing scheme still achieves theoretically optimal load balance [11].

Support for heterogeneity

KAAPI's support of GPUs is essentially left to the developer. Because KAAPI tasks can be stolen at any time before their execution, there is, by design, no affinity between tasks and any particular computational unit. Therefore, developers must turn to mechanisms to either control access to the GPU through a mutex, or create some affinity between one particular KAAPI thread and the GPU device.

We would like to remark that KAAPI is currently being completely rewritten as XKAAPI [12], which will include native support for GPUs. However, XKAAPI is still in development and currently does not support distributed memory architectures.

6.3.3 MR-MPI: MapReduce over MPI

MapReduce [31] is a programming framework for distributed applications emphasizing simplicity and accessibility for non-parallel programmers. Developers define sequential primitives which operate on portions of an implicitly-referenced distributed hash table which resides locally on nodes involved in the computation. All parallel communication details are hidden from the user by API calls and occur simply to reorganize the distribution of the hash table in the distributed environment. By providing a simple interface, even novice parallel programmers can leverage parallel computational resources and gain benefits in terms of speed of application execution or scale of problem size.

MR-MPI [88] is a C++ MapReduce implementation using MPI as the distributed communication library. The data in MR-MPI are stored in the internal hash table either as Key-Value pairs or Key-MultiValues tuples and are processed by five main functions. Key-Value pairs are produced by the *map()* function through a user-defined callback function. The *aggregate()* function distributes Key-Value pairs to processors and guarantees that pairs with the same key are on the same processor. Key-Value pairs with identical keys are merged into a single Key-MultiValue tuple by the *convert()* function. Each Key-MultiValue tuple is transformed into (zero, one, or more) Key-Value pairs by the *reduce()* function based on a user-defined function. Data are redistributed on less processors (typically a single one) using the *gather()* function. MR-MPI's API is easy to understand and its core API fits into a single page.

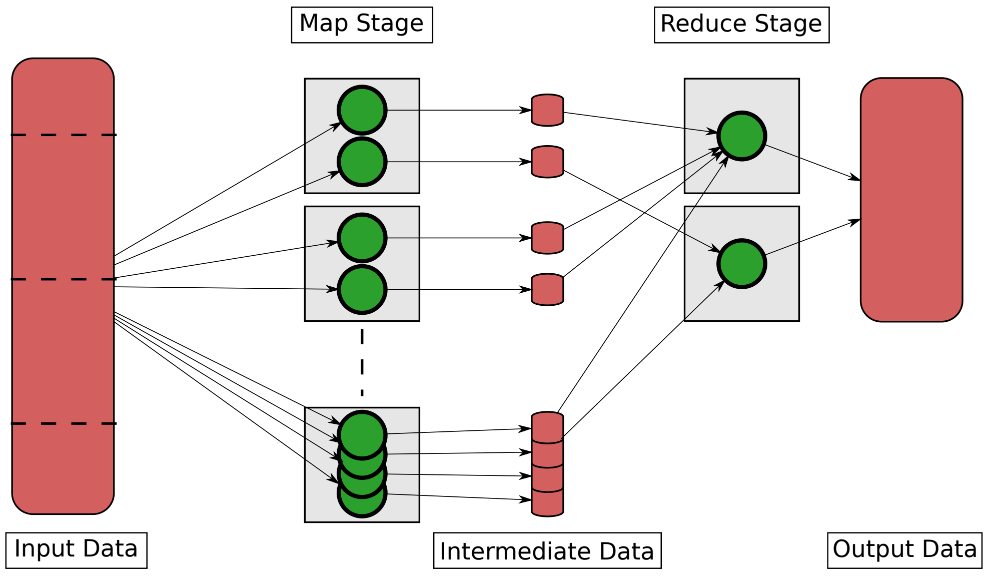


Figure 6.2: Example MapReduce Execution

Figure 6.2 shows a simple example of a MapReduce application running on a distributed cluster of machines. The simplest MapReduce applications only need to define one Map stage, and one Reduce stage. Most MapReduce frameworks (including MR-MPI) have some built-in support for managing file-based input data, to ease the burden of partitioning and distributing multiple files, possibly of different lengths, and possibly resident in a non-uniform manner across the set of computational nodes.

Map tasks, therefore, take as input some subset of the globally defined input data space. The Map stage has two goals: to put the input data into the implicitly-held hash map data structure, and to ensure that the size of the key-set is sufficiently large so as a good hash-based load-balancing of the downstream reduce tasks. The number of reduce tasks may not match the number of map tasks; the number can be higher or lower (the latter is reflected in Figure 6.2). The Reduce stage's goal is to operate on the intermediate results in the distributed hash map and determine some final result, or another intermediate result to be passed to a further MapReduce cycle, for more complex applications.

Load-balancing mechanism

The user is not required to write any code related to partitioning, communicating, sorting, or balancing the application data. Rather, users simply write sequential functions which will operate on portions of their data, and use calls to the MR-MPI runtime system to perform all of the parallel communication. Internally, the load-balance is achieved by partitioning the key space into equal chunks. If the internal hash table's keys are complex structures, users should supply a custom hash function, which ensures good key distribution over the processors. If applications have unpredictable task execution times, MR-MPI also has a master/slave load balancing mechanism, which can improve load balance, depending on the specific application and system configuration.

Support for heterogeneity

MR-MPI's support for GPUs is entirely up to the developer to manage. Because MR-MPI is closely linked with its MPI back-end, we can simply use the MPI rank to determine which processes should use the GPU and which should use CPU threads. If cooperative CPU / GPU is desired, then a custom hash function needs to be supplied which gives the GPU ranks a larger share of the intermediate data values. For the best load balance, developers can profile their application kernels, and give the GPUs the correct proportion of the total work, such that they finish at the same time as the CPU threads.

6.4 APC+: Adaptive Component-based Dataflow

Because parallelism, hierarchy, and heterogeneity conspire to reduce the performance of applications developed for modern supercomputers, high-level programming frameworks must explicitly handle these concepts. Since parallelism implies data and computation partitioning, and since a poor choice can adversely affect the overall execution time, the partition size is an important parameter to choose carefully [57]. All of the previously discussed distributed programming frameworks require an explicit decision to be made as to the partition size. The range of acceptable partition sizes can be large for complex parallel applications, and thus, the process to find the optimal value for the partition size can be quite tedious. Thus, this section describes our enhanced Adaptive Partitioner Controller, APC+, a technique for using a feedback loop to automatically tune the data and computation partition sizes for efficient processor utilization, efficient network utilization, and good overall load balance.

6.4.1 APC+ Algorithm

Built on top of DataCutter, APC+ gains all of the benefits of a high-performance dataflow runtime engine. APC+ automatically tunes the databuffer size and application data partitioning, which helps schedule an efficient application execution. There are four parts to our method:

- *Performance Model* [57]

  The performance model tracks each databuffer's execution time for each processor in the system. By continually tracking the execution time of databuffers in the system, we can dynamically find the databuffer size with the fastest processing rate.

- *Work Partitioner* [57]

  By using user-supplied knowledge of the application's workspace, and the information about the system's characteristics from the performance model, the work partitioner streams databuffers to the processors with the aim of balancing the load.

- *Distributed Work-Stealing Layer*

  With APC+, each node of the system has a controller filter running, which only provides work to the local processing units. It handles all of the performance modeling of the local processors, and manages the partitioning of its own queue of work. All work sharing between the nodes is conducted by a new work-stealing layer, which tightly integrates with the performance modeler and the work partitioner.

- *Storage Layer*

  While the performance model focuses on keeping each processor in the system operating at peak efficiency, and the work partitioner and work-stealing layer ensure good load balance across all of the nodes, the storage layer's goal is to optimize the use of the network interface. By streaming work to remote nodes, and by transferring data in large chunks, the storage layer ensures that the network is used as efficiently as possible. It is important to note that the work-stealing layer simply balances the *assignment* of work to nodes, while the storage layer actually moves the application's input data to the appropriate node. So stealing large work assignment is actually a light-weight operation, and the actual data is streamed only as fast as the network can handle.

In preliminary work [57], we proposed an adaptive databuffer tuning and work partitioning technique, which featured the first two parts, the Performance Model and the Work Partitioner. Here, we present improvements to the technique, the Distributed Work-Stealing and Storage Layers, which make our framework more network aware. Providing large-scale scalability with a single APC filter was impractical, because APC is only indirectly network-aware. Indeed, APC does not make any distinction between communication time and computation time. The main reason is that the network interface is difficult to model effectively within a high-performance dataflow runtime system. Since distributed dataflow runtime systems are not inherently synchronous, it is impossible to instantaneously determine the execution time of any one data transfer, when there are multiple pending data transfers. Therefore, APC cannot accurately predict the data transfer times in a dynamic system, preventing good load balance among the downstream processors. Further, more than one APC must be used when there are upstream data sources on multiple nodes (e.g., some data is read from disk in blocks and sent into the system for processing). Each APC must be able to send computation to all the processing filters to be able to utilize the distributed system fully. However, each APC is oblivious to its peers, and each of them will send too-small tiles to each processing filter, because they do not have a global picture of the amount of work flowing into the system. This unfortunate consequence induces high network overhead, sub-optimal processing filter efficiency, and high load imbalance.

To solve these problems, APC+ features a work-stealing layer which handles all distributed-memory work allocation and a storage layer which optimizes the use of the network interface. This twofold approach has many benefits: 1) It allows APC+ filters to transfer input data in sizes which are efficient for the network interface, 2) it decouples the transfer of data and the processing of the data, and 3) it simplifies dataflow application executions which have distributed input data (data can first be processed locally, without forcing a remote operation). Since work is shared at the node level, instead of the filter level, work can be transferred in much larger partitions. Further, the *assignment* of work to a thief APC+ filter is a light-weight operation, and is separate from the transfer of the application data. This distinction allows the storage layer to stream the data and avoid overloading the network interface, while still allowing for rapid load-balancing among the nodes. To be sure, component-based dataflow with demand-driven load balancing also simplifies distributed work sharing, but the approach adopted by APC+ integrates knowledge about where work is produced and where and how fast work is consumed.

APC+ targets applications whose *work area* is finitely divisible, subject to minimum and maximum databuffer size constraints stemming from application-specific and hardware constraints such as memory size. We assume that there is a monotonically increasing execution time for increasing databuffer sizes for all processor types. APC+ can handle any type of computation that can be expressed as an n-dimensional work area. For instance, the simplest workload is 1-dimensional, and represents a simple reduction or transformation operation on a 1-d vector of input data. Or, an image analysis operation which operates on 2-d images and produces a feature vector represents a 3-d work area. The work area definition simply needs to match the dataflow application's requirements.

Figures 6.3 and 6.4 show simple example placements of APC and APC+. The major difference is that APC is a centralized system, using only a single filter to manage all of the processors in the entire distributed computational resource, whereas APC+ is a decentralized system, using an APC+ filter copy on each node, which is only responsible for driving the processors resident on that same node.

Load-balancing mechanism

As already discussed, APC+ contains four elements which integrate tightly to form APC+'s automatic dataflow application tuning and work partitioning. APC+ operates on *WorkTiles*, which are n-dimensional work area abstractions and are generic enough to allow APC+ to automatically partition the overall application's work area

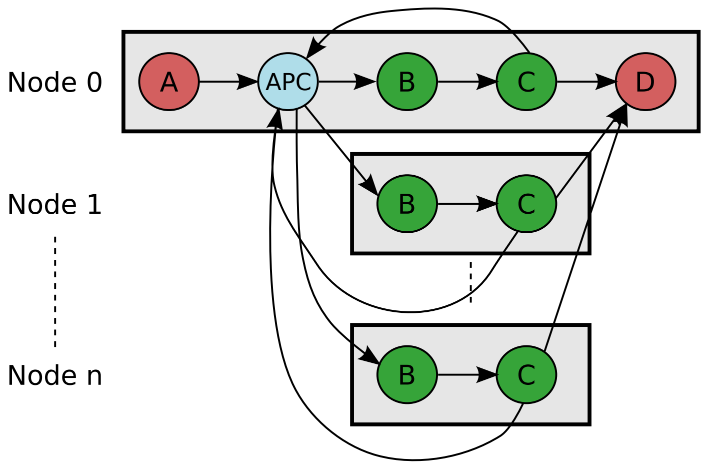


Figure 6.3: Example APC placement

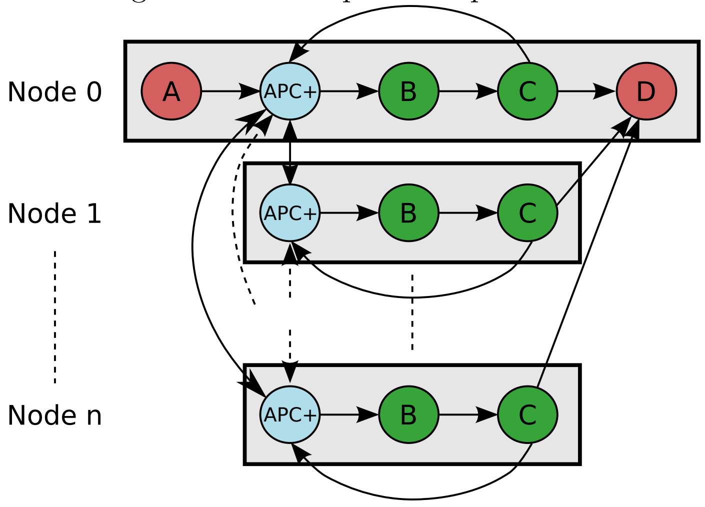


Figure 6.4: Example APC+ placement

and submit work to downstream processors. Applications simply send a WorkTile to an APC+ to alert it of work to be performed. Since APC+ is oblivious to application specifics, it is up to the developer to choose a work area which can be interpreted in the application domain. For instance, in a 2-d image analysis application, the work area would consist of a set of WorkTiles pairs representing several images of a certain number of rows and columns. APC+ will recursively break these WorkTiles down into smaller WorkTiles, representing contiguous subtiles of a certain number of rows and columns. The analysis application will then process these image subtiles.

```
function APC_SETUP($min\_tile, max\_tile, threshold$)
    $queue$ = get_more_work($steal = queue.empty()$)
    $exit\_bootstrap$ = false
    while $min\_tile < max\_tile$ and !$exit\_bootstrap$ do
        for all processor $p \in P$ do
            $min\_tile$ = get_next_buffer_size($p$)
            send_work($queue, min\_tile, p$)
        $exit\_bootstrap$ = true
        loop $|P|$ times
            receive ($p$, $tile$, $timing$, $block = true$)
            add_data_point($p$, $size$, $timing$)
            if $timing < threshold/2$ then
                $exit\_bootstrap$ = false
    $min\_tile$ = parallel_reduce($min\_tile$, MAX)
    $completed\_work = 0$
    $replied = P$
    $p = 0$
    $global\_end\_time = 0$
    $agg\_proc\_rate = 0$
```

Figure 6.5: Adaptive Partitioning Controller (APC+) Setup Pseudocode

Figures 6.5 and 6.6 show the main APC+ pseudocode. *APC_SETUP* (in Figure 6.5) runs first, once, then the *APC_LOOP* (in Figure 6.6) is entered. One APC+ copy will run on each node in the application's placement. Each APC+ relies on its own performance model for estimated processing rate information for all of the processors to which it is responsible. To bootstrap the performance model, APC+ looks for the largest minimum tile size for which a minimum time threshold is reached.

```
1: function APC_LOOP($min\_tile, max\_tile, threshold$)
2:     while !shutdown($queue$) do
3:         $queue$ = get_more_work($steal = queue.empty(), work\_received, agg\_proc\_rate$)
4:         $queue$ = receive_steal_requests($queue, agg\_proc\_rate$)
5:         if $|replied| = |P|$ or $work\_received$ then
6:             $alloc\_work$ = PARTITION($queued\_work$, $queue, agg\_proc\_rate$)
7:             $replied = \emptyset$
8:             $global\_end\_time$ = end_time($work\_received, completed\_work, agg\_proc\_rate$)
9:         $block = (\forall p' \in P, \frac{queued\_work[p']}{\text{best\_processing\_rate}(p')} > threshold)$
10:        if receive($p'$, $tile$, $timing$, $block$) then
11:            add_data_point($p'$,$size$, $timing$)
12:            $replied = replied \cup \{p'\}$
13:            $queued\_work[p']$ -= $size$
14:            $completed\_work$ += $size$
15:        if $\frac{queued\_work[p]}{\text{best\_processing\_rate}(p)} < threshold$ then
16:            $size$ = min ($alloc\_work[p]$,
   get_next_buffer_size($p, global\_end\_time, queued\_work[p]$))
17:            send_work($queue, size, p$)
18:            $queued\_work[p]$ += $size$
19:            $alloc\_work[p]$ -= $size$
20:        $p = (p + 1) \bmod |P|$
```

Figure 6.6: Adaptive Partitioning Controller (APC+) Main Loop Pseudocode

This helps to alleviate the overhead of partitioning the work area into tiles which are too small. Once this largest minimum tile size is found, the information is exchanged among all of the distributed APC+ copies. This introduces some synchronization, unfortunately, but the cost is quite small as compared to the extra databuffer packing/unpacking, network, and work area fragmentation overheads associated with choosing a minimum tile size which is too small.

Following bootstrapping, APC+ enters its main processing loop, shown in Figure 6.6, and the overall execution of the application begins. WorkTile sizes are increased exponentially until the maximum-sized WorkTile is reached, at which point the WorkTile size which gives the fastest processing rate for a particular processor will be preferred.

In the situation where an APC+ filter has an upstream data source, the main loop will perform a non-blocking read on it each time through the loop, because having accurate knowledge of the global amount of work in the system is key to ensuring a good load balance. If there is no more work to be queued to downstream processors, the APC+ filter will attempt to steal some work from a randomly selected peer. If, when asked, a peer returns that it has no work, that peer is marked such that it is not asked for work again. If all the peers are marked, then a shutdown process is attempted. In shutdown, each peer is asked whether its upstream is closed, and if all of the upstreams (including its own) are closed, then the filter will complete its shutdown. Otherwise, all of the peers with active upstream ports will be marked as eligible to steal from, and the shutdown attempt will be canceled.

Our stealing algorithm is quite simple, although we do leverage the aggregate processing rate of all of the node's processors to let APC+ filters steal only as much data as they are likely to be able to process. Therefore, the APC+ filter (the thief) making the steal request to a peer (the victim) includes its own aggregate processing rate in the steal request. Then, the victim calculates the correct amount of work to give to the thief. That is, the victim will give the thief work such that both APC+ filters will complete their work at the same time. The victim APC+ filter will send the work to the thief APC+ in as large WorkTiles as possible, since this reduces the network and processing overhead of sending many small WorkTiles.

Periodically, the APC+ filters will exchange the size of any additional work received from an upstream data source, the amount of completed work, and the aggregate fastest processing rate information for its processors. This enables the entire system to have some global knowledge of the estimated end time. When WorkTiles are queued to processors, this global estimated end time is used to provide another opportunity for reducing the size of excessively long-running WorkTiles, to prevent load imbalance.

Using a simple round robin scheme, APC+ sends WorkTiles to each processor having less than T time units of work queued (T defaults to 100 ms). Once the entire range of possible WorkTile sizes has been tried for a particular processor, APC+ switches to sending the WorkTiles which are of a size to gives the highest expected processing rate. APC+ allocates work to each processor such that they finishing executing their queue at the same time. When the remaining work is low, and the WorkTile size suggested by the performance model is too large, APC+ will send a smaller WorkTile (even if the WorkTile sizing function is still in the bootstrap phase). By doing so, we prioritize load balance over processing rate at the end of the application's execution. Frequently (exactly after receiving a reply from all the processors), the amount of work each processor should be given is updated using $PARTITION()$; this allows APC+ to react quickly to changing system conditions.

The *shutdown()* function uses a master-slave method to reach global consensus that there is no more work to be completed by anyone. Once all of the upstream data sources are closed, and each APC+ has no more work to complete, rank 0 will send a shutdown message to all of the peer APC+ filters, and the application will exit.

Figure 6.7 shows pseudocode for the $PARTITION()$ function which provides the amount of work each processor should be allocated. The function calculates the estimated amount of time each processor's queued work will take to complete; it uses this information to allocate to each processor enough work to last max_time. It then distributes the remaining work to the processors proportionally to their processing rate. If $PARTITION()$ runs out of work before all of the processors are given sufficient work to match the longest running processor, then that longest running processor will set the application end time anyway; by not actually queuing more than T time units of work to any processor in Figure 6.3, we ensure the load imbalance is no worse than T.

```
1:  function PARTITION($queued\_work$, $work$)
2:      $procrate = \sum_{p \in P}$ best_processing_rate($p$)
3:      for all processor $p \in P$ do
4:          $queue\_time[p] = \frac{queued\_work[p]}{\text{best\_processing\_rate}(p)}$
5:      $max\_time = \max_{p \in P} \{queue\_time[p]\}$
6:      for all processor $p \in P$ do
7:          $alloc[p] = \min\{(max\_time - queue\_time[p]) \times$
    $\text{best\_processing\_rate}(p), work\}$
8:          $work = work - alloc[p]$
9:      for all processor $p \in P$ do
10:         $alloc[p] += \frac{work \times \text{best\_processing\_rate}(p)}{procrate}$
11:     return $alloc$
```

Figure 6.7: Adaptive Dataflow Work Partitioner

Storage Layer

The new storage layer in APC+ allows the work-stealing mechanism to work efficiently and quickly to balance the application's workload, even in situations where a large input data source is only resident on one node of the system. Like APC+, the storage layer is distributed over all of the nodes involved in the processing of the application. The light-weight WorkTiles used by APC+ keep track of where the application data is actually stored (in which node's storage layer), so that remote nodes can request the data directly from the source. The actual application input data can be arbitrarily large (as it is application-specific), and the storage layer serves WorkTiles which contain the data in a demand-driven fashion. When a thief APC+ steals some work from a victim, the victim alerts its local storage which WorkTiles have been stolen. All of the WorkTiles which have been sent to the thief are put into a transfer queue, and the first WorkTile is sent immediately to the thief's storage filter. Upon receipt of the first WorkTile by the thief's storage filter, two things occur: 1) an acknowledgement is sent back to the victim storage filter, and the next WorkTile is sent; and 2) the thief's storage filter alerts the thief APC+ that work has arrived which can be queued to the processors.

Notice that an APC+ filter running on node 0 might steal work from node 1, while the actual input data is is physically resident on node 2 (and is queued to be transferred to node 1). Since the WorkTiles themselves keep track of where the application-specific data is, the transfer queues can simply be canceled and rerouted to node 0 by sending the appropriate messages to node 2. Since the APC+ work-stealing mechanism prioritizes stealing large WorkTiles, and because the storage layer only streams data as quickly as the network can transfer, APC+ exhibits improved network performance, when compared to APC.

The storage layer places no limit on the number of times a WorkTile can be transferred. However, since the work-stealing operation is very lightweight, in most cases the storage layer will not have to transfer WorkTiles over the network more than once. Compute-intensive applications may have more redundant WorkTile transfers than data-intensive applications, because the data transfers will occur faster than the computation. However, the work-stealing layer ameliorates this problem by stealing WorkTiles according to the relative aggregate processing rates of the victim and the thief, such that the correct amount of WorkTiles which balances the load is stolen, and scheduled for transfer.

Support for heterogeneity

APC+'s support for heterogeneity inherits all of DataCutter's built-in support for heterogeneity, specifically the ability to abstract away architectural details from the application decomposition point of view. However, unlike the other distributed programming frameworks presented here, APC+ partitions the application's work area dynamically, allowing each processor type to work on WorkTile sizes for which it is the most efficient, providing efficient performance, and good load balance.

6.5 Application Descriptions

To test the performance of the three distributed programming frameworks, we have chosen three applications with different characteristics. Each application has a different initial data distribution, task execution time profiles, and CPU/GPU speedups. This helps to stress the runtime systems tested in our experiments.

Our first application is a **Synthetic Aperture Radar (SAR)** imaging application [56]. Radar return signals from multiple viewing angles of the same scene are combined together to create an image. The simple backprojection algorithm used here is a triple loop where each input vector of radar return data is applied to each pixel of the output image. The inputs to the application are small, so they are broadcasted at the beginning of the application execution. Because the application tasks are independent, and because the task execution times are more dependent on the number of output pixels created than on the number of inputs, we partition the output space into tiles. After processing, the final image is aggregated back on one node, and written to disk. Our output image sizes range from 4K x 8K on one node up to 32K x 32K on 32 nodes. Our CPU implementation is only of reference quality, but its further optimization is outside the scope of this paper. The computation style is very well-suited to the GPU, and we see large speedups (see Section 6.7 for all CPU and GPU execution time specifics for our platform). The lack of an efficient CPU implementation in specific libraries is common with GPU implementations of applications [73]. This is not a middleware issue and runtime systems should be able to handle it.

Our second application is a **Biomedical Image Analysis (BIA)** application [97]. In this application, highly magnified digital images of specially prepared biopsy tissue samples are processed using cooccurrence matrices and operators called linear binary patterns to determine the texture of the tissue found in the biopsy sample. The texture determination affects the prognosis for the patient. This is a time-consuming, error-prone process for human slide readers to perform, and the goal of this application is to reduce error and increase analysis throughput. The digitized images can be over 100K x 100K pixels, and many slides are often analyzed as part of a patient's study. In our experiments we have used image sizes ranging from 16K x 32K pixels for a single node to 128K x 128K pixels for 32 nodes. For our experiments, the input image begins already decompressed in the memory of a single node (or two nodes, if it would not fit into a single node's main memory). This input image is tiled and sent to computational nodes for processing, and a feature vector for each tile of 13 single-precision floats is returned. The processing requirements are a good match for the GPU's many-core architecture and high memory bandwidth; hence, high speedups can also be seen, as compared to a single-threaded reference CPU implementation can be obtained.

Our last application is **Black-Scholes**, from the Nvidia CUDA SDK. It is a stock market option pricing application which is designed to model the way options are sold on the market, and to predict when buying and selling these options would be advantageous. Each option consists of three 32-bit floating point values, and the output calculated by the application is two 32-bit floating point values, which represent the "call" and "put" prices of the option. To make this simple calculation data-intensive, our input dataset includes 2^{27} options per node to calculate. On 32 nodes, this means that there are 4 billion options to process. Options are processed in tiles. Because the computation is relatively simple, this application has a high communication to computation ratio. This application is very I/O-bound. Therefore to provide the maximum disk bandwidth, we partition the input data set over the local disks of all the nodes involved in the experiment, as part of an untimed setup phase. Thus, when using 2 or more computational nodes, the input data may be read concurrently from one disk of each of these nodes. The output of the computation is aggregated on a single node, although not written to disk.

6.6 API and Optimizations

This section briefly discusses the APIs for DataCutter, KAAPI, MR-MPI, and APC+ and some specific optimizations which are required in each framework to achieve good runtime application performance. While the basic application-specific functions are the same, the glue code to establish the task graph, orchestrate the application's execution, and setup and tear down the frameworks is quite different.

6.6.1 DataCutter

The DataCutter *main()* function is the most verbose of the three programming frameworks, because it is the most explicit. For the best performance, developers should control where instances of filters are placed and executed, and the exact set of interconnections of streams in between those filters. When filters are replicable, the API provides syntactic shortcuts to create "transparent" copies of filters, and place them on specific nodes.

The *main()* function instantiates a DCLayout object, adds all of the filters in the application's task graph, connects the filters' endpoints, and executes the application.

DCFilter code, on the other hand, is about as verbose as comparable tasks in the other frameworks. Filters often implement a while() loop internally, such that they can process multiple databuffers from upstream filters.

DataCutter's GPU support entails simply writing a specific DCFilter class implementation where the GPU kernel is called in place of the CPU function. Cooperative

CPU / GPU applications are simple to develop, also, since the setup stage can instantiate GPU-based implementations of filters alongside CPU-based filters.

DataCutter has configurable memory limits, such that applications will not page to disk. Additionally, DataCutter overlaps the communication of databuffers and filter computation by using a daemon thread and non-blocking MPI calls. Further, developers can use knowledge about their application to maximize performance. Computationally intensive applications should only instantiate as many filters per node as there are computational elements, while data intensive applications should ensure that only one filter per disk controller is instantiated on one node, to avoid disk-thrashing.

6.6.2 KAAPI

In KAAPI, developers use Athapascan-1 to explicitly describe the shared data dependencies between tasks. These shared variables can be given explicit read and/or write access controls, which defines the task graph ordering which will be scheduled by the KAAPI work-stealing runtime engine.

The *main()* function in KAAPI programs is quite simple; first it initializes the runtime system, and calls a special *ForkMain()* function to tell KAAPI where to start the first task. Once inside this main task, the real application processing begins, typically by spawning more tasks. Child tasks are all placed in the same stack as the thread which created them; other processors only can do useful work by stealing tasks from a victim processor. When a task is stolen by a remote processor, its parameters are copied across the network. Because the load balancing is done using point-to-point stealing operations, one task may be copied multiple times from one machine to another one. On non-data intensive applications such as state space expansions, the data typically is small, leading to a small network overhead. However, a data intensive application will see gigabytes of data being copied around without being processed.

To avoid such wasting of network resources and induced network overhead, our implementations make use of KAAPI's remote iterators construct. Remote iterators are a separate method for accessing remote data as opposed to using Athapascan-1's shared variable constructs. In addition to optimizing the network usage, remote iterators are initialized in a known memory space, for instance the same node where the application was started, which is convenient to aggregate the output data on a single node. By using a special *Fetch()* command, tasks gain access to shared data, even when running on remote nodes.

We use remote iterators because they cause less data copies, but unfortunately this breaks Athapascan-1's method of determining correct ordering of tasks (because there are no explicit Shared dependencies left). We used SyncGuard constructs to solve that problem. SyncGuards are a syntactic method for ensuring that all tasks forked inside a frame of reference are completed before execution of the parent thread can continue.

While the notion of remote iterators is not general to asynchronous task programming paradigms, their use in KAAPI is extremely significant in its effect on the application execution time. By only requiring data to be copied once a task has actually begun executing, users can dramatically speed up the section of their applications which focus on forking tasks (thereby partitioning the data domain or search space), and not require essentially useless copying of data during this phase.

Unlike DataCutter and MR-MPI (discussed in Section 6.6.3), Athapascan-1 and KAAPI do not allow developers to “pin” a task to an arbitrary node. Therefore, it is difficult to efficiently read from multiple disks in applications such as Black-Scholes, since there is no way to know from which node a particular file will be read. Thus, we can either copy all of the input data files to each of the nodes in the environment ahead of time, very inefficiently, or use a parallel file-system of some type. In place of a heavy-weight parallel global file-system, we have used simple sshfs remote directory mounts.

KAAPI's GPU support takes the form of a simple if/else statement inside the task. Developers must determine which threads are allowed to use the GPU themselves, but once chosen, it is simple to choose the correct path of execution for those threads.

6.6.3 MR-MPI

MR-MPI applications are by far the simplest of the three frameworks. They are initialized simply by setting up the MPI environment and creating a MR-MPI MapReduce object. Then the MapReduce primitives can be called. Finally, during the map and reduce stages of the application's processing, we simply pass function pointers to the MR-MPI library's function calls.

A defining feature of MapReduce frameworks is organizing intermediate data as a distributed hash table. Therefore we need to define key and value structures which the MR-MPI engine handles as byte arrays, but this is not especially onerous.

If MR-MPI's internal Key-Value or Key-MultiValue hash tables ever exceed one “page” of memory in one MPI process (as given by the *memsize* setting), then temporary disk files are used, on a per-processor basis. When using more than one MPI process per node (as in system configurations with multi-core processors), the local disk can become a severe performance bottleneck. Therefore, for applications which can support such a partitioning and global merge, it makes sense to divide the total application's computation into working sets, such that the working set fits into the memory of the machine. Thankfully, MR-MPI's *memsize* setting can be set to a different value for each MPI process, allowing working set merges to a single node to take place in-memory also. This is an important optimization to give MR-MPI applications efficient execution time performance.

Like KAAPI, MR-MPI's GPU support takes the form of a simple if/else statement inside the sequential *map()* or *reduce()* callback function. While determining which rank should use the GPU is a task outside MR-MPI's purview, once this is done, users can simply use that flag to choose the correct path of execution.

6.6.4 APC+

As alluded to in Section 6.4, the user-level programming API for APC+ is quite straightforward. Users simply write upstream data sources which produce WorkTiles with application-specific data attached to them. Processing filters simply need to be able to understand WorkTiles in the application-specific context, and parse the user data. Finally, users must provide a function to split the application-specific data for APC+ to use when it partitions WorkTiles in an abstract manner. Then, one APC+ and one storage filter, and as many processing filters as there are processing cores need to be instantiated on each node in the DataCutter placement. All of the runtime orchestration of task execution and data partitioning is handled automatically.

6.7 Application Experiments

6.7.1 Experimental Setting

Our experiments were run on *Owens*, the new GPU cluster in the Department of Biomedical Informatics at The Ohio State University. For our experiments, we used 32 computational nodes, each with dual Intel Xeon E5520 Quad-core CPUs (with 2-way Simultaneous Multithreading), 48 GB of memory, and 500 GB of scratch disks. The nodes are equipped with Nvidia C2050 Fermi GPUs, each with 3 GB of memory, and are interconnected with 20 Gbps InfiniBand. Our experiments were run on CentOS with the 2.6.18-194.8.1.el5 Linux kernel, and compiled with GCC 4.1.2 using the -O3 optimization flag. Our GPU code was compiled with CUDA 3.1. While DataCutter and MR-MPI can use an InfiniBand-aware MPI, such as MVAPICH, KAAPI is not InfiniBand-aware. Therefore, to provide a meaningful comparison, we have run all of our DataCutter and MR-MPI experiments using MVAPICH2 1.5 over TCP (TCPoIB) as well as InfiniBand (IB). Our experimental results show the difference between the IB and TCPoIB runtimes as an explicit overhead. As a point of reference, for our cluster, using IB nets 800 MB/s point-to-point bandwidth vs 110 MB/s for TCPoIB.

In all our experiments, we performed a parameter sweep to select the tile size and thread configuration that lead to the best performance separately for each runtime system, except for APC+ (since it requires no developer tuning). We tuned each application for the three remaining runtime systems by choosing the configuration which gave the best performance for the majority of different numbers of nodes. Detailed discussions of why a specific tile size and thread configuration are chosen over a different configuration are out of the scope of this chapter; rather, we will simply be satisfied that the configurations for each runtime system are as optimized as they can be, and consider the application kernels to be black boxes, except for the tunable parameters they expose.

Table 6.1 shows the execution times using only one CPU thread and using only one GPU of all three applications for the tiles sizes which were found to give the fastest performance, according to our parameter sweep.

Application	**Tile Size**	**1 CPU thread**	**GPU**	**Speedup**
SAR	128	3,354.1	21.5	156.1
	512	53,792.3	182.5	294.7
BIA	512	213.8	8.2	26.0
	1024	857.6	22.3	38.4
	2048	3,433.6	96.0	35.8
Black-Scholes	64K	19.6	0.8	24.4
	128K	34.7	1.4	25.3

Table 6.1: Execution times (in milliseconds) for SAR, BIA, and Black-Scholes for various tile sizes

The next sections present the results of the experiments for our three applications. To best show the scalability of each system, our experiments are weak scalability experiments: we increase the amount of work in commensurate amounts when we increase the numbers of nodes. The volume of data per node for each application is given in Section 6.5. To aid in the clarity of the charts, we have broken down the execution times into several categories. There are five main categories which are present in the charts of all three applications, PROC, L-IMB, OVER, BOOT, and TCP. The PROC portion of each bar represents the minimum processing time of all of the threads, while L-IMB is the load imbalance of the system (calculated by subtracting the minimum processing time of all of the threads from the maximum).

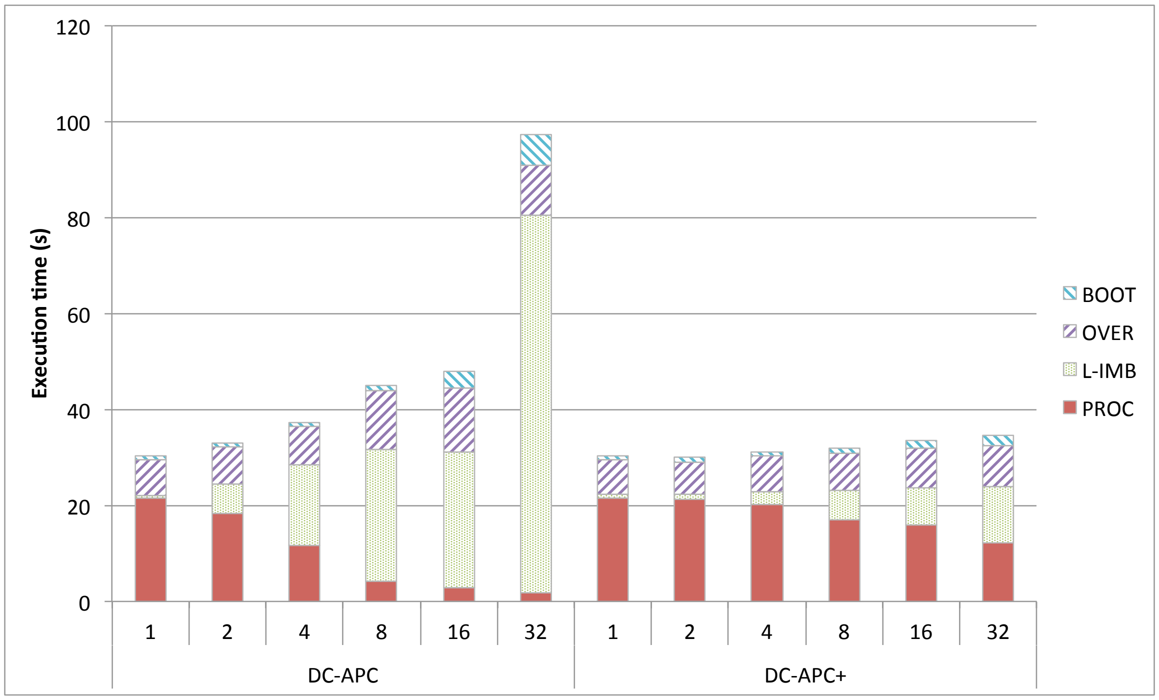


Figure 6.8: SAR: CPU-GPU, APC vs APC+

The OVER category is a catch-all category, and includes communication, and other runtime engine overheads. Finally, BOOT represents the amount of time to bootstrap the runtime engine itself, including remote process invocation, etc, and TCP represents the difference in the total runtimes of the TCPoIB and IB experiments for DataCutter and MR-MPI. Note, therefore, that the breakdown of the overall TCPoIB execution time into the sub-categories may be different from that of IB. On each chart, the y-axis gives the runtime of the configuration in seconds, while the x-axis gives the runtime system used and the number of nodes it ran on.

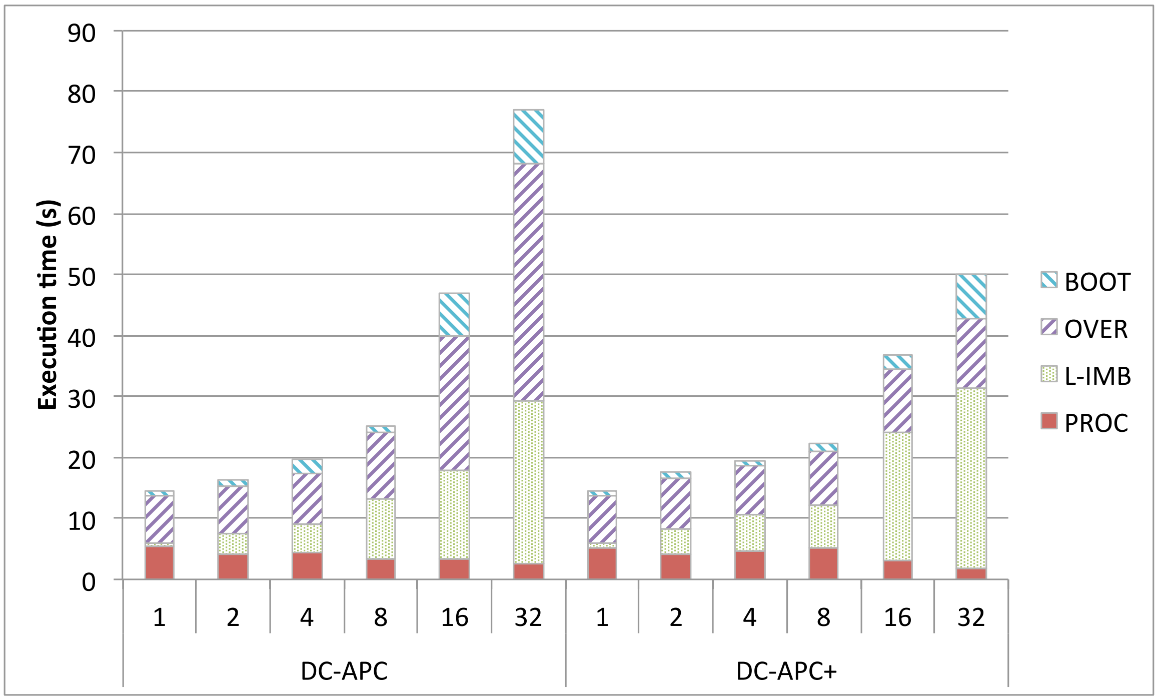


Figure 6.9: BIA: CPU-GPU, APC vs APC+

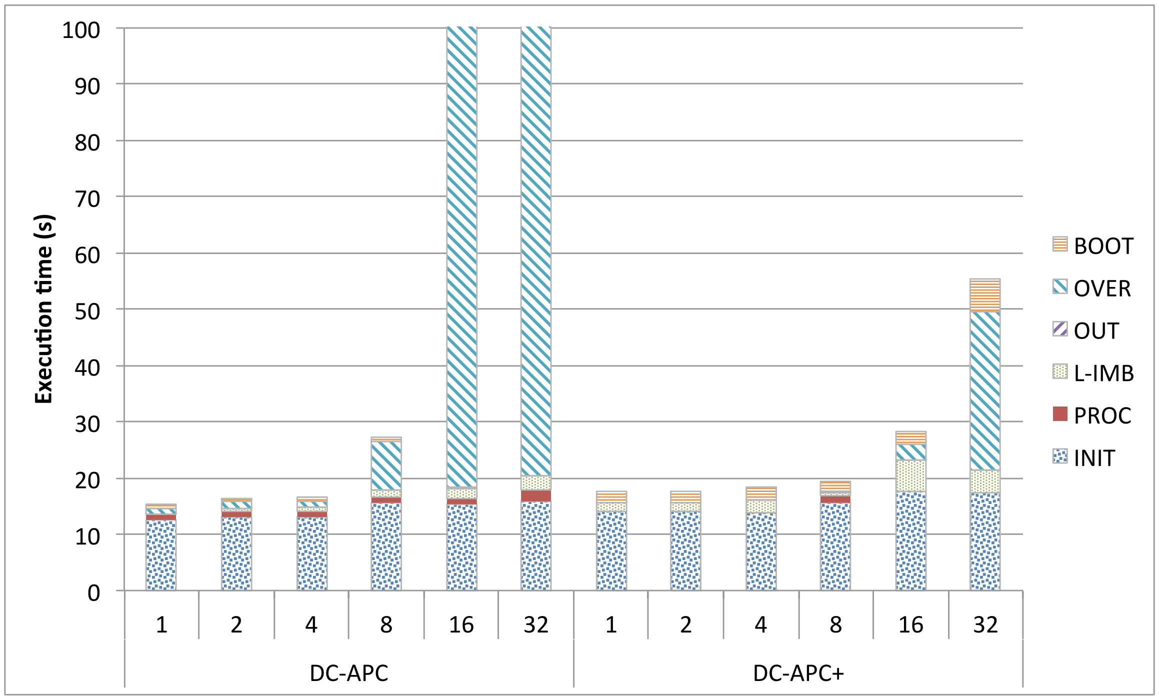


Figure 6.10: Black-Scholes: CPU-GPU, APC vs APC+

6.7.2 APC and APC+ Comparison

This section briefly compares the performance of APC and APC+ while running experiments on our three applications over the IB network. Figures 6.8, 6.9, and 6.10 show the CPU-GPU results for APC and APC+ for SAR, BIA, and Black-Scholes, respectively. Recall that the experiments show weak scalability; linear scaling is achieved when the execution times are flat, because extra work and extra computational nodes are provided in equal measure. We have chosen to restrict this comparison to the CPU-GPU results, to avoid belaboring the point: scalability on system configurations with higher numbers of nodes is poorly affected by using a centralized work scheduler.

In SAR (Figure 6.8), there is a large initial work area (although with relatively small input data), and the different processor types have drastically different processing rates types. APC suffers in this instance because while it tries to balance the load, even a single too-long tile being executed on a single CPU can cause serious load imbalance. As one can see, APC+'s execution times are nearly flat, showing near linear scalability up to 32 nodes. Interestingly, APC+ also shows some load imbalance, although this is a direct side effect of the dramatic GPU speedup; APC+ keeps the CPUs slightly starved, in order to keep one too-long-running task from setting the overall execution time.

In BIA (Figure 6.9), a large input image starts on one node (or two nodes for the 32 node experiment), which causes a major bottleneck in the network endpoint, which is an impediment to effectively distributing the input data to the whole cluster. Worse, the smallest tile size used in our experiments (32 x 32 pixels) leads to a very small 3 KB databuffer, which results in poor IB performance. This runtime effect of the network contention is visible both in the overhead category and as a side-effect in load imbalance. While executing with 16 nodes, a 64K x 128K image is 24 GB, meaning that it takes over 28 seconds merely to transfer an equal 1/16 portion of the image to each remote node.

Black-Scholes (Figure 6.10) has distributed input data streaming into the system from every disk, which can lead to a situation where a centralized controller, responsible for every node in the system, can never increase the databuffer size sufficiently to reach high-performing tiles. In the case of Black-Scholes, all of the computation time *can* be hidden by reading the input data off of the disk, but when the tile size is too small, the network becomes a major bottleneck, especially for the aggregation down to one node. As one can see from Figure 6.10, APC+ also has issues with the one node aggregation on 32 nodes, but as we will see later in this section, this is simply caused by endpoint contention on the aggregation node.

6.7.3 Synthetic Aperture Radar

Figures 6.11 and 6.12 present the experimental results for the SAR imaging application for all four programming frameworks for, respectively, CPU-only and CPU-GPU configurations. For the CPU-only experiments, APC+ uses CPU 15 threads per node, because one thread is taken up by the APC+ functions. The remaining frameworks all use 16 threads. Please note that, the results shown for DC-DD, KAAPI and MR-MPI use tile size that have been manually tuned to obtain the best performance at the most scales. DC-DD performs best with tiles of size 128 x 128, while KAAPI and MR-MPI perform best with 512 x 512 tiles. For the CPU-GPU experiments,

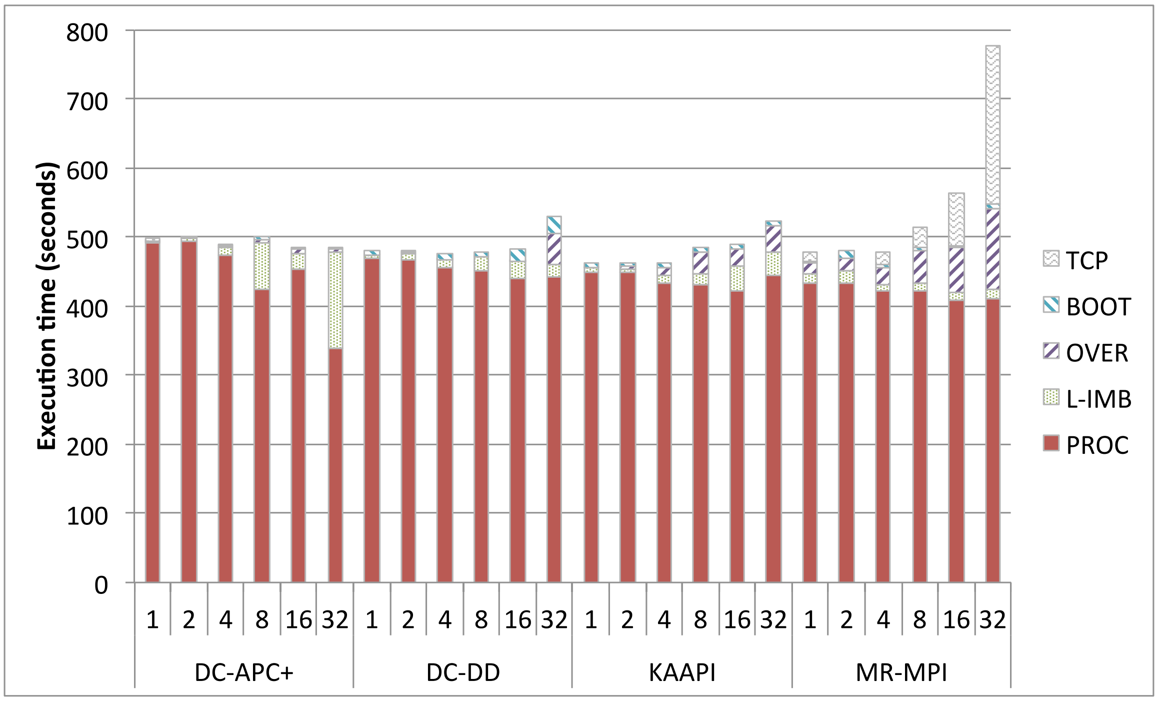


Figure 6.11: SAR: CPU-only

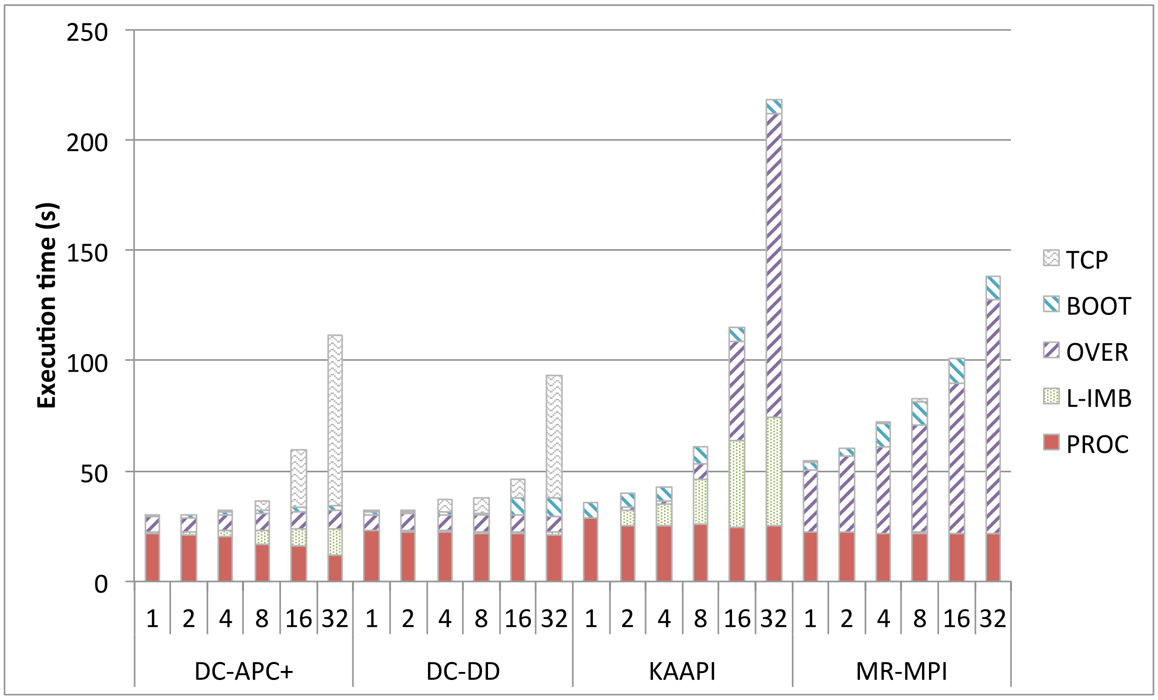


Figure 6.12: SAR: CPU-GPU

APC+ uses 14 CPU threads and the GPU, while the rest of the CPU-GPU configurations for SAR use the GPU and no CPU threads, reflecting the large GPU speedup. DC-DD, KAAPI and MR-MPI obtain the maximum performance when a tile size of 512 x 512 is selected.

This application is very computationally intensive, and the CPU kernel implementation is fairly inefficient, accounting for the excellent CPU scalability for APC+, DC-DD, and KAAPI up to 32 nodes, and for MR-MPI when using pure IB visible on Figure 6.11. The slowdown for MR-MPI is related to the relatively large image size formed by the application on 32 nodes. A 32K x 32K complex-valued single precision image is 8 GB, which takes some time to transfer over the inefficient TCPoIB protocol. Since this communication is not overlapped at all by MR-MPI, it is visible as overhead in the TCP category. Each of the frameworks reports some degree of load imbalance. However, it is always less than the execution time of a single tile.

Our GPU filter implementation for SAR, on the other hand, is quite efficient, leading to much faster execution times, and regrettably, poor scalability for all of the frameworks with TCPoIB (which can be seen on Figure 6.12). Notice that APC+ and DC-DD are able to keep up with the increasing computational demand and scheduling complexity when using IB, however. The large image size on higher node configurations causes large execution time penalties, in the TCP category for APC+ and DC-DD, and in the OVER category for KAAPI and MR-MPI.

KAAPI is the only framework which shows significant load imbalance in the GPU case when using more than 4 nodes. This is caused by the requirement to gather the output image onto a single node before exiting the application. (Recall that the image is not written to disk, but simply gathered on one node.) The gathering of the output is implemented using remote iterators initialized on the node where the data need to be gathered. The data transfers involving remote iterators are not overlapped with communication. Therefore, the node which owns the data executes significantly more computations than the other ones, leading to a large load imbalance. Unfortunately, this load imbalance also causes most of the time accounted for in OVER. When a thread on the node that owns the data no longer has work in its queue, it tries to steal work from a remote thread. However, the network stack and hardware are congested by the remote iterator transfer, making the total time of each stealing operation more than one second on 16 processors. This idles the node that owns the data for a significant amount of time at each stealing operation. Further, it is important to notice that KAAPI's remote iterators can be read-only or read-write but not write-only. Therefore, to write the output data for each tile, the uninitialized tile data must be read first, doubling the total amount of data transfer and worsening the network congestion in Figure 6.12.

Most of the time spent in MR-MPI is accounted for in OVER. This can be decomposed into two parts, a constant part and a part proportional to the number of processors. The constant overhead consists of the requirement to simply distribute the small input data, and the requests to each node for what portion of the output image they have to calculate. Indeed, these two steps take over 24 seconds to complete. DC-DD and KAAPI allow asynchronous task execution and the processing can start immediately. However, MR-MPI is synchronous and must wait for all the data transfer to finish. The variable part comes from the aggregation of the data to one node, which must occur for all three middleware systems. However, the synchronized behavior of MR-MPI forces the communications to be serialized after the computation, worsening the network contention.

6.7.4 Biomedical Image Analysis

Figures 6.13 and 6.14 present the experimental results for the biomedical image analysis (BIA) application for, respectively, CPU-only and CPU-GPU configurations. Again, APC+ uses 15 CPU threads during the CPU-only experiment, and 14 CPU threads and the GPU during heterogeneous configurations. All the other three frameworks' CPU-only experiments are fastest when using 16 threads per node, when running without the GPU. DC-DD and KAAPI use a 1024 x 1024 tile size, while MR-MPI use a 512 x 512 tile size. In the mixed CPU-GPU experiments, DC-DD and MR-MPI are fastest when only using the GPU, while KAAPI is able to make use of 7 CPU threads as well as the GPU, with a 512 x 512 tile size. DC-DD uses a 2048 x 2048 tile size, while KAAPI and MR-MPI use a 1024 x 1024 tile size.

Unfortunately, unlike the SAR application, none of the runtime systems scale linearly beyond 4 nodes when using TCPoIB, even on the CPU implementations. Further, APC+, DC-DD and KAAPI all suffer from an increasing degree of load imbalance on high numbers of nodes. A bottleneck must account for the loss of scalability. In this case, it is the network interface contention on the "storage" node (or storage nodes in the case of 32 nodes), where the input data is resident. Since there are 16 threads per CPU processing tiles, and 31 nodes in the system making requests of the network endpoint, at 857 milliseconds per tile, this would require 1820 MB/s network endpoint bandwidth. The cluster has 20 Gbps InfiniBand, which gives an effective bandwidth around 800 MB/s. MR-MPI does not show any load imbalance: since the work is statically partitioned, all the processors performs the same number of operations.

However, we see that DC-APC+ is able to scale near linearly using the IB interface while running with CPU threads only. Unfortunately, the load imbalance gets the better of it on high numbers of nodes when using a heterogeneous system. Since the CPU runtimes for even large tiles are relatively small for BIA, a well-tuned Demand-Driven execution will be hard to beat outright, but APC+ is able to reach nearly the same performance level, without the laborious tuning step.

The major slowdown of KAAPI is again due to KAAPI's inability to overlap communication with computation. All of the latency is visible in the L-IMB execution time category, and is visible in the OVER time category, while DC-DD does a good job of overlapping communication with computation thanks to the dataflow paradigm.

The MR-MPI runtime system's performance is not linearly scaling for a variety of reasons. First, since MR-MPI cannot overlap communication with computation, all of the data transfer time is seen directly in the OVER category. When using 32 nodes, our image size is 128K x 128K pixels, with three color channels, leading to a 48 GB image. Second, to stay within the memory constraints of the node, we need to use working sets which fit inside the memory of a single MPI process. We can asymmetrically allocate memory to the MPI processes which share a node, such that the process responsible for loading and distributing the input data can get the lion's share of the data, but MR-MPI allocates 7+ pages of memory (according to the documentation) for parallel communications, requiring the working set of the

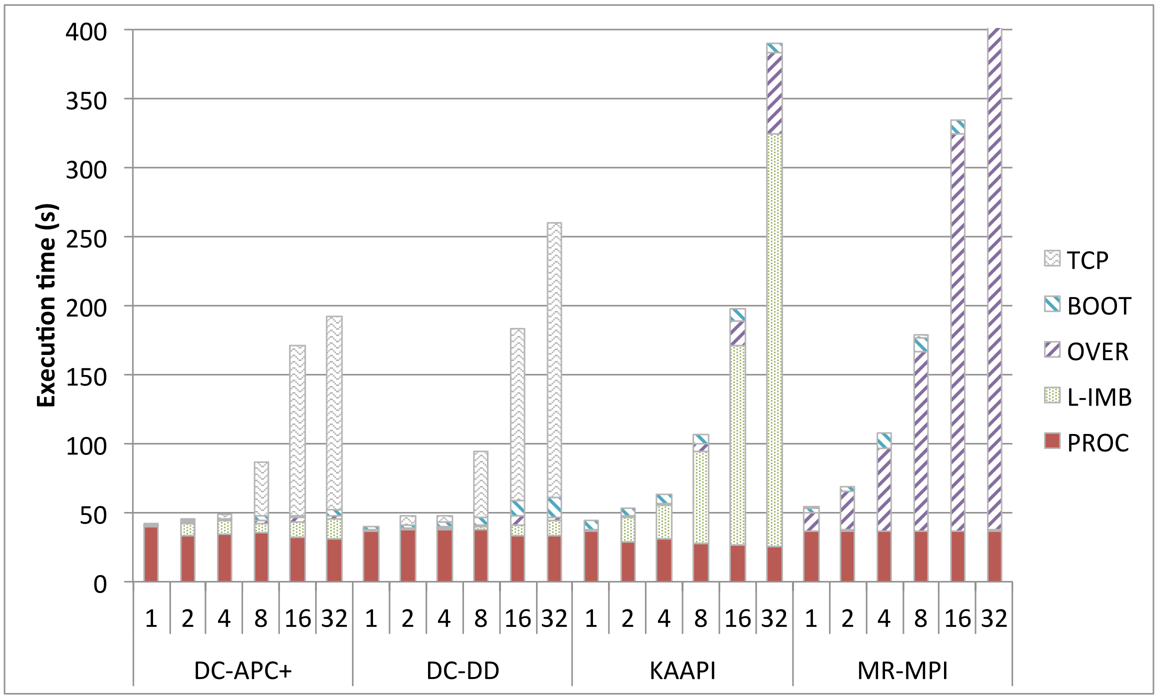


Figure 6.13: Biomedical Image Analysis: CPU-only

application to be artificially small. Thus any overheads to set up and tear down these parallel communications are paid more than necessary.

6.7.5 Black-Scholes

Figures 6.15 and 6.16 show the CPU-only and the CPU-GPU results for the Black-Scholes application, respectively. All of the CPU-only implementations use 8 CPU threads, and all of the CPU-GPU implementations use 1 GPU thread only, except for DC-APC+, which uses 7 CPU threads and the GPU. DC-DD's CPU-only and KAAPI's GPU implementations are fastest at most scales with tiles of 128K options, while the rest of the implementations are fastest at most scales with 64K option tiles.

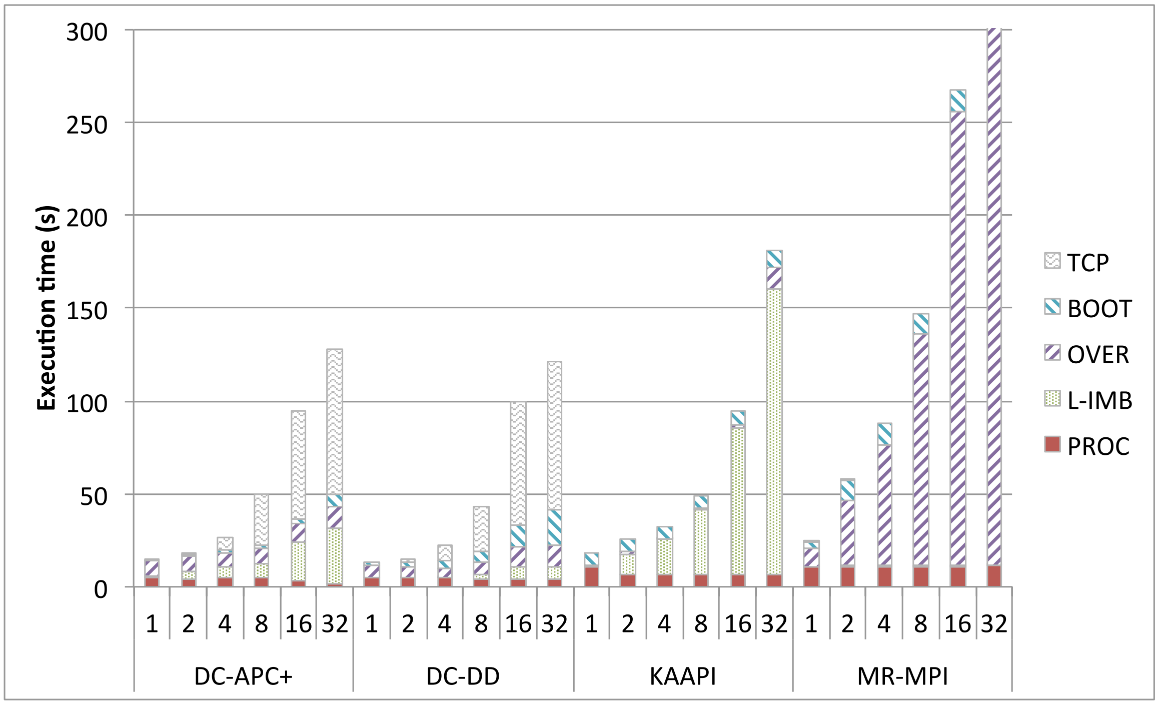


Figure 6.14: Biomedical Image Analysis: CPU-GPU

Like many real-world applications, the Black-Scholes application is bound entirely by disk read bandwidth, because the amount of computation per option is relatively small. Thus, even though our implementation streams data in tiles to the processing tasks from the scratch disk of the nodes, there will be some amount of disk access time which is not overlapped with computation. The INIT time presents this value: it represents the maximum of all of the time each read task takes to read the options from disk and store them in memory minus the maximum processing time of a processing thread. The OUT portion of each bar refers to the non-overlapped time to aggregate the output data on one node (but not actually written to a file - the application is already disk-bound, we do not need to belabor the point).

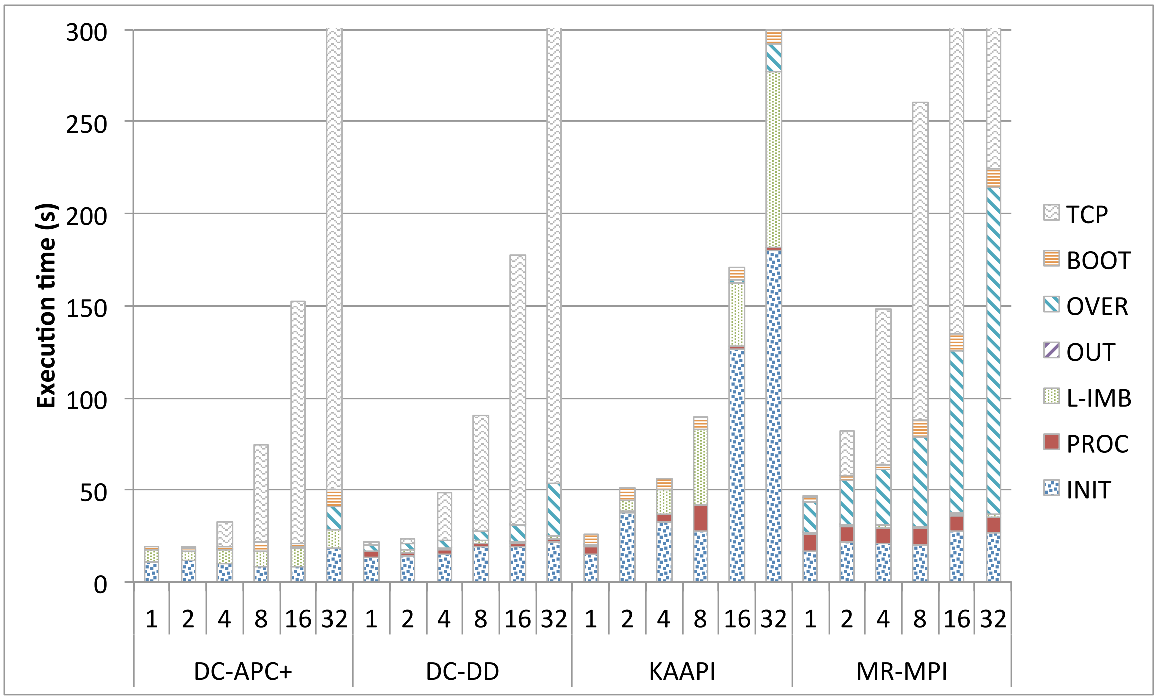


Figure 6.15: Black-Scholes: CPU-only

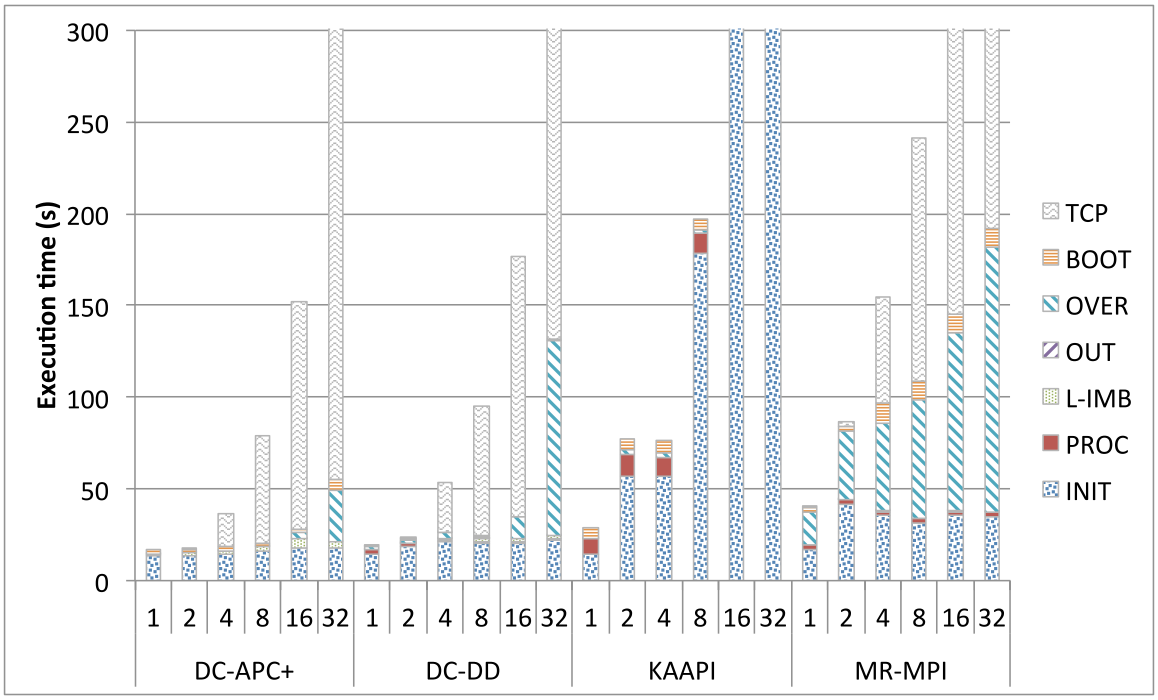


Figure 6.16: Black-Scholes: CPU-GPU

There are several notable trends in Figures 6.15 and 6.16, which we will discuss in turn. The most obvious fact is that all of the runtime systems perform terribly when using TCPoIB. The INIT time for DC-APC+, DC-DD and MR-MPI stays constant when the number of nodes is increased, meaning that while the size of the disk-based input data is increased, the disk read bandwidth is also increased, and the INIT execution time component is not worsened. However, perfect scalability is not achievable in Black-Scholes, since all the results are gathered on one node, and this operation is limited by that node's network bandwidth.

DC-APC+ using IB scales the best, followed by DC-DD. When using IB, DC-APC+ slows down by a factor of 2.6 when increasing to 32 nodes. DC-DD slows down by 2.7 times, KAAPI is 11.4 times slower on 32 nodes, and MR-MPI only slows down by 4.9 times, although it is over twice as slow on one node as DC-APC+. When using TCPoIB, however, KAAPI scales better, beating DC-DD's performance on 8, 16, and 32 nodes. While not surprising, efficient use of a high bandwidth network is important. DC-DD's Demand-Driven policy induces communication from each producing filter to each processing filter. Since the data are present on each node, there are one producing filter per node and eight (in the CPU-only case) or one (in the GPU-only case) processing filters on each node. Clearly, all to all communications take place which overstress the network system when using TCPoIB.

The most noticeable difference between KAAPI and the other runtime systems is the growing INIT time. Indeed, KAAPI is not able to place data input read tasks on the same node where the data resides; we therefore incur a large latency and disk and network contention penalty by reading the data across the network through a parallel file system of any type. The decrease in non-overlapped INIT time beyond 2 nodes is due to there being less disk contention and more aggregate disk bandwidth than on 2 nodes. KAAPI's load imbalance is once again due to the lack of communication and computation overlapping.

MR-MPI's poor scalability performance is caused by not overlapping communication and computation, and by MapReduce's load-balancing technique. On 32 nodes, while using IB, MR-MPI spends 57.8 seconds communicating input data to the processing tasks, and 92.8 seconds gathering the output data to a single node. When using TCPoIB, MR-MPI spends 302.3 seconds distributing the input data, and 607.0 seconds gathering the output data to a single node. INIT takes more time in MR-MPI than in DC-DD because of the synchronization of MR-MPI that prevents the overlapping of reading the files and of their processing. An overhead similar to the one shown on BIA is present in Black-Scholes as well.

6.7.6 Tunable CCR

Finally, Figure 6.17 shows a strong scalability experiment where we choose a static system configuration, and vary the relative amounts of communication and computation. Our tunable CCR application is based on Black-Scholes: the amount of computation is increased by executing several iterations of the Black-Scholes kernels for each options. If the number of iterations is set to 1, the tunable CCR application and Black-Scholes are exactly the same. The chart shows the 16-node CPU speedups (calculated by dividing each framework's 16-node time at each iteration point, by the fastest 1-node execution time for that iteration point) as a function of the number iterations on the Black-Scholes kernel per option. As such, for low values of number

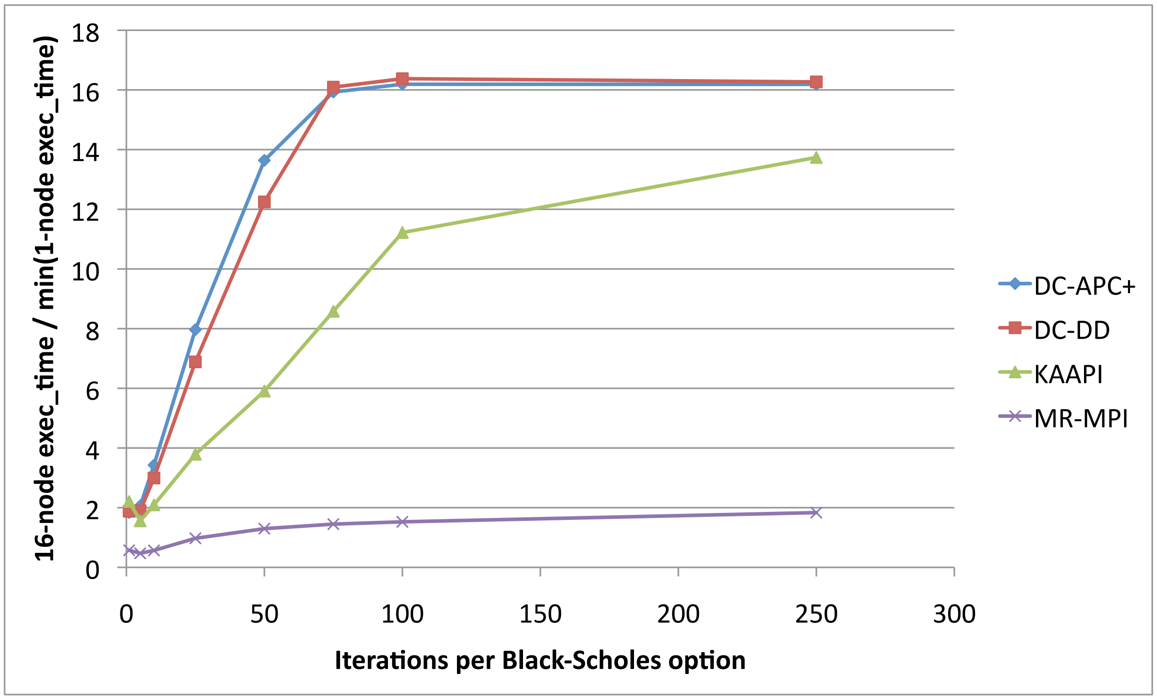


Figure 6.17: Tunable CCR Application: 16 nodes, CPU-only

of iterations, the application is heavily data-intensive, and as the number of iterations increases, the application becomes compute-intensive.

While it appears that there is some super-linear speedup for DC-APC+ and DC-DD for 75 iterations and above, this is merely a consequence of the additional disk bandwidth provided the application when running on 16 nodes. The execution time on 16 nodes is never more than 3% faster than the numerically derived best-case execution time extrapolated from the single-node execution time for the fastest framework.

When the number of iterations is 1, DC-APC+, DC-DD and KAAPI have a low speedup of 2 due to the long time spent aggregating the results on one node. When the number of iterations increases, the time spent to aggregate becomes smaller (in relative terms) and the speedup increases. DC-DD and DC-APC+ achieve linear speedup when the number of iteration is greater than 75. KAAPI scalability does not increase as well, since it achieves a speedup less than 14 for all of the numbers of iterations less than 250. We believe KAAPI's speedup would likely reach 16 if we had run larger experiments.

MR-MPI does not show great speedup. Its speedup increases when the number of iterations increases, although even when the kernel of Black-Scholes is executed 250 times per option, the speedup of MR-MPI stays less than 2. This is caused by several factors. First, by examining Figure 6.15 we see that MR-MPI begins considerably slower on 16 nodes than the fastest framework. When running on TCPoIB, the MR-MPI 16-node execution time is 470.0 seconds, while DC-APC+ runs in 152.5 seconds. Thus, MR-MPI is at an immediate 3x deficit. Additionally, MR-MPI serializes the reads from disk and the processing of the options. Thus, any increases in the processing time of the options are immediately visible. The other frameworks are able to hide the increase in processing time until it exceeds the disk read time, at which point the overall execution time begins to increase.

6.7.7 Discussion of Experimental Results

Because DataCutter gives the developer great flexibility in the implementation of the application, it can leverage all of the capabilities of HPC systems to achieve excellent performance in homogeneous settings. However, the Demand-Driven policy shows some issues. First, all to all communications decrease network efficiency, particularly at larger system scales. Further, it is oblivious to the heterogeneity of the downstream filter which will cause a large load imbalance when accelerators have particularly efficient filter implementations.

KAAPI/Athapascan-1 focuses more on the description of the computation than on the data it manipulates. While this focus leads to close-to-ideal performance on a shared memory machine, it also penalizes the performance of distributed data-intensive applications. For instance, the lack of support for pinning tasks in KAAPI made the implementation of reading distributed input data difficult on the Black-Scholes application, leading to poor performance. KAAPI only transfers data during work-stealing operations or when fetching remote iterators. However, neither of these two communication modes enables the overlapping of communication and computation. It leads to poor scalability when the communication times are close to the computation times such as on the BIA application. These issues could certainly be solved by a simple modification of the API and middleware.

Writing applications using MR-MPI's API is quite straightforward, but (by design) does not allow exact management of the communication. For instance, the lack of communication and computation overlap induces unnecessary overhead that leads to bad scalability. Notice that the MapReduce programming model does not require synchronization for all operations, and specific calls could be implemented to provide pipelined execution, overlapping communication and computation. These improvements would certainly lead to much better scalability.

Meanwhile, APC+ provides performance which is close to the best-tuned Demand-Driven implementation, without the tedious manual tuning step. It provides efficient use of processors by tuning databuffer size for best processing rate, while still maintaining good distributed load balance for heterogeneous systems. It provides enhanced scalability and a simple programming semantic, for applications which fit the targeted model.

To sum up, we find that the following properties must be included in a middleware to perform well on data-intensive applications: Communication and computation overlapping must be available for a middleware to perform gracefully on application where the time to transfer the data over the network is close the time spent to process them. Providing some method to express affinity between tasks and nodes is necessary to perform I/O operations efficiently. Controlling the amount of computation performed by each processing element is crucial in obtain good load balance on heterogeneous architectures. Finally, all to all communication must be avoided to optimize the network efficiency.

6.8 Summary

Writing a distributed application for high performance computing systems such as clusters and the grid is becoming more and more complex. The tremendous number of parameters involved in their design makes their programming and tuning a tedious task. Numerous programming models and middleware frameworks have been proposed to lower the programming complexity, while still achieving good parallel performance. We compared three programming models and runtime frameworks by their ease of programming and actual performance through the implementation of three distributed applications within the simplest class of parallel applications: the execution of independent, nearly identical tasks. We found that even a manually tuned application cannot make use of all of the peak performance which is available in the system, often due to the lack of communication and computation overlapping, all-to-all communication, localization-oblivious data consumption, and the execution of long and non-preemptable computations.

Dataflow programming frameworks, and DataCutter in particular, provide communication and computation overlapping and enable a localized consumption of data. We have developed APC+ to remove the a lengthy manual tuning process, by monitoring processor performance and adaptively partitioning the application workarea. It achieves good load balance through the use of a work-stealing engine, while still providing efficient network usage by using a dedicated storage layer. We experimentally showed that APC+ is able to meet or best the performance of the well-tuned Demand-Driven configuration, without any manual tuning process. Further, we also compare against well-tuned KAAPI and MR-MPI implementations, and show that APC+ consistently has shorter execution times than those frameworks.

Chapter 7: Efficiently Supporting Programming Models with Adaptive Dataflow

7.1 Introduction

The application model for our adaptive component-based dataflow middleware presented in Chapters 5 and 6 is expressive and powerful. Unfortunately, this expressiveness comes at a cost of increased verbosity, particularly for simple tasks. Thus, this chapter explores some methods for using our middleware to support alternative programming models, while still leveraging our high performing adaptive techniques. In particular, we focus on two specific programming paradigms which allow us to encapsulate most of the verbosity of the full adaptive technique, and to present a minimal API. These two programming paradigms are *Parallel-For* and *MapReduce*. These two case studies merely scratch the surface for possibilities of supporting alternative paradigms, but since they are widely used as a simple way to leverage parallelism in user applications, they represent a logical first foray.

7.2 Parallel-For

OpenMP [27] and Intel's Threading Building Blocks (TBB) [105] both make use of the simple *Parallel-For* loop construct. Parallel-For is a simple semantic programmers can use to denote for loops as eligible for parallelization, for improved execution time performance on shared-memory systems. Through a procedural construct, parallel-for is able to glean from the developer what the parallelization requirements are: which variables should be shared, and which private; how to distribute the loop iterations, in chunks, or one-by-one; and whether any synchronization is required between iterations.

7.2.1 Standard Application Programming Interface

Figure 7.1 shows a simple parallel for loop implementation of a vector sum. The *#pragma omp* preprocessor directive tells the preprocessor that some OpenMP code is to follow. It is the use of the *private()* portion of the directive which partitions the loop iterations, along with the *#pragma omp for* directive, since *i* is the loop index.

```
#pragma omp parallel default(none) shared(n,x,y,z) private(i)
{
  #pragma omp for
  for (i=0; i<n; i++)
    z[i] = x[i] + y[i];
}
```

Figure 7.1: Simple OpenMP Parallel-For Example

Figure 7.1 shows the power of OpenMP for concisely representing a developer's wish to perform some data-parallel computation. However, there are some drawbacks to OpenMP's simple interface.

First, the chunking of loop iterations is still a best-effort exercise. Some researchers have proposed an adaptive loop scheduler which optimizes the number of threads to use on multi-core nodes with simultaneous multithreading enabled [117]. Unfortunately, while this technique will choose the highest performing configuration, including what number of threads to use and the which of the dynamic algorithms for choosing loop iteration chunk sizes, it still does not complete the feedback loop with a focus on using execution time behavior to intelligently scheduling loop iterations with an exact load-balance. Thus, we use our dataflow middleware to model the performance behavior to tune the chunk size and achieve a very small load imbalance.

Further, OpenMP leaves the task of supporting heterogeneity to the developer. Certainly, developers can rely on the OpenMP thread id and *if()* statements to control which thread uses the GPU, but GPUs may operate best with a completely different scale of loop iterations than CPU cores, due to their different data communication and kernel invocation overheads. Thus, we also rely on our dataflow middleware's explicit support for the heterogeneity GPUs bring to efficiently use modern computational resources.

7.2.2 Interface Improvements and Runtime Optimizations

Figure 7.2 shows a preliminary API for an adaptive component-based parallel-for engine. Undoubtedly better compiler tricks can be brought to bear on the API, and the unfortunate requirement to use a callback function instead of an inline scope can be improved. Still, the syntax is not overly cumbersome, and indeed, encapsulation of this type for complex processing functions can be useful for good software engineering practices.

```
void processing_callback(DCPF * dcpf)
{
  int num_iters;
  double x;
  float y;
  ComplexObject p;
  dcpr->unpack_args("idfz", &num_iters, &x, &y, &z);

  // <snip application code>
}

int main(int argc, char ** argv)
{
  // <snip application code>
  // Initialize the parallel for engine
  DCPF * dcpr = DCPF::get_parallel_for_engine();
  // Using a simple function call and var_args, users can
  // begin a parallel for loop

  int num_iters;
  double x;
  float y;
  ComplexObject p; // must extend DCSerializable
  dcpr->pack_args("idfz", num_iters, x, y, &z);
  dcpr->parallel_for(&procesing_callback);
  // <snip application code>
}
```

Figure 7.2: Homogeneous Adaptive Component-Based Parallel-For

Additionally, heterogeneity can be utilized by simply alerting the parallel-for engine that a GPU is available, and is to be used. Figure 7.3 shows a snippet of an application where a single GPU is to be used, in addition to however many cores the system exposes. It is important to note that often, GPUs will have dramatically different constant overheads (related to kernel invocation times and data transfer times) as compared to CPU cores. This is important, because the best-effort systems used by OpenMP are likely to fail at using GPUs effectively for scheduling parallel-for loop iterations. Also note that the figure shows ellipses in the *get_parallel_for_engine()* and *parallel_for()* calls, meant to show that heterogeneity is not limited to two processor types.

7.3 MapReduce

Section 6.3.3 discusses the MapReduce programming paradigm in general, and the MR-MPI middleware in particular. In Section 6.7.7 we discuss some of the shortcomings of MR-MPI. However, we will repeat a brief introduction to the MapReduce paradigm and discuss its shortcomings here for completeness.

MapReduce is a programming framework which focuses on hiding the complexity of parallel programming from developers. MapReduce runtime systems store all application-specific data in a hash table which is distributed among the processors by using application-agnostic primitives. Developers write sequential functions which operate on portions of the hash table which are located locally. All parallel communication details are hidden from the user. Thus, by providing a simple interface, even novice parallel programmers can leverage parallel computational resources.

```
void gpu_processing_callback(DCPF * dcpf)
{
  int num_iters;
  double x;
  float y;
  ComplexObject p;
  dcpr->unpack_args("idfz", &num_iters, &x, &y, &p);

  // <snip application code>
}

void cpu_processing_callback(DCPF * dcpf)
{
  // <snip application code>
}

int main(int argc, char ** argv)
{
  // <snip application code>
  // Initialize the parallel for engine
  DCPF * dcpr = DCPF::get_parallel_for_engine(int num_gpus,
                                               GpuType, ...);
  // Using a simple function call and var_args, users can
  // begin a parallel for loop

  int num_iters;
  double x;
  float y;
  ComplexObject p; // must extend DCSerializable
  dcpr->pack_args("idfz", num_iters, x, y, &p);
  dcpr->parallel_for(&cpu_procesing_callback,
                     &gpu_processing_callback, ...);
  // <snip application code>
}
```

Figure 7.3: Heterogeneous Adaptive Component-Based Parallel-For

7.3.1 Application Programming Interface

Figure 7.4 shows the MR-MPI main function implementing Black-Scholes (see Chapter 6). The main API documentation fits onto a single page; it is quite simple to understand and use, even for novice users. The MR-MPI code is exceedingly simple for a powerful distributed programming framework. It initializes by setting up the MPI environment and creating an MR-MPI MapReduce object. Then the MapReduce primitives can be called. Finally, during the map and reduce stages of the application's processing, we simply pass function pointers to the MR-MPI library's function calls. The calls to *aggregate()*, *clone()*, and *gather()* are calls to manipulate the hash table, both in terms of distribution over the processing nodes (*aggregate()* and *gather()*), and in terms of simple hash table transformations (*clone()*) to give the correct format to the correct function.

However, in Chapter 6 we identified that synchronization seriously damages the performance of the MR-MPI middleware, particularly on data-intensive applications. Synchronizing the parallel execution in between each call to *map()* or or *reduce()* is unnecessary in many applications, since typically all of the synchronization can be implicit in the reorganization of the implicitly held hash table of application-specific data. Additionally, we find that implementations of MapReduce, such as MR-MPI rely on static partitioning of work tasks, which is non-ideal in the face of the kind of heterogeneity GPUs provide. Therefore, we have implemented a subset of the MapReduce API as a case-study in using our adaptive dataflow middleware to show that its high performance can be brought to bear on simple programming paradigms, which hiding the full complexity of general component-based dataflow programming.

```
void init_black_scholes(int itask, MAPREDUCE_NS::KeyValue *kv,
                         void *ptr);

void run_black_scholes(char *key, int keybytes,
      char *multivalue, int nvalues, int *valuebytes,
      MAPREDUCE_NS::KeyValue *kv, void *ptr)
{
  // GPU support works with a simple if() statement: Need to
  // put MPI rank, num_gpus per node and processes_per_node
  // into global variables
  if (rank % procsses_per_node < num_gpus) {
    // run GPU code
  } else {
    // run CPU code
  }
}

void output_black_scholes(char *key, int keybytes,
      char *multivalue, int nvalues, int *valuebytes,
      MAPREDUCE_NS::KeyValue *kv, void *ptr);

int main(int argc, char ** argv)
{
 int rank;
 MPI_Init(&argc, &argv);
 MPI_Comm_rank(MPI_COMM_WORLD, &rank);
 MapReduce * mr = new MapReduce(MPI_COMM_WORLD);

 mr->map(master_file, &init_black_scholes, &options_per_file);
 mr->aggregate();
 mr->clone();
 mr->reduce(&run_black_scholes, NULL);
 mr->gather(1);
 mr->clone();
 mr->reduce(&output_black_scholes, argv[2]);

 MPI_Finalize();
 return 0;
}
```

Figure 7.4: Black-Scholes MR-MPI main function

7.3.2 Interface Improvements and Runtime Optimizations

Consider the classic MapReduce "Hello World" example, Wordcount, shown in Figure 7.5. Conceptually, the application is quite simple, and the MapReduce programming API keeps the application simple.

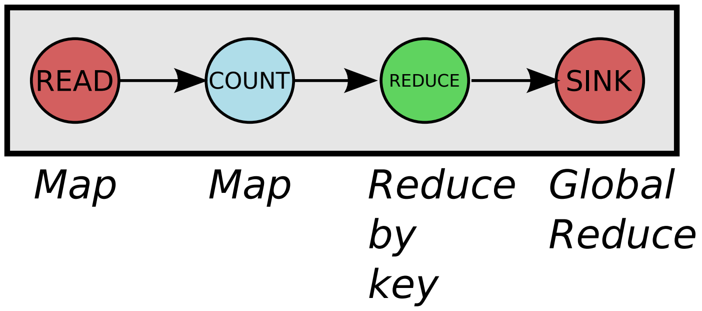


Figure 7.5: MapReduce Wordcount Application Layout

Figure 7.6 shows our implementation of MapReduce using our adaptive dataflow middleware, called DataCutter MapReduce (DCMR). At the highest level, a DCMR control filter manages the execution of the overall application. DCMR acts as the upstream data source for APC, and as such it has a great deal of control over when work is available to be scheduled.

Because the DataCutter layout will always be the same (except for transparent copies on multiple nodes, which is straightforward), the user of DCMR no longer needs to provide a lengthy setup function, or indeed, to know anything about how the DataCutter MapReduce, or dataflow programs work in general. With a simple API, consisting of one callback function from the DCMR controller, users schedule their tasks. By using a task queue instead of blocking task calls, DCMR can schedule

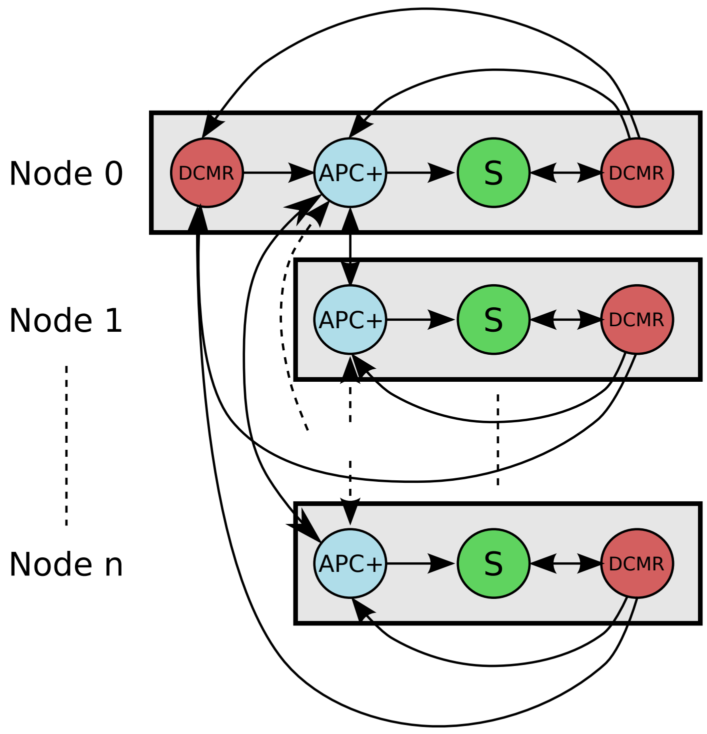


Figure 7.6: DataCutter MapReduce

the events in the queue with more freedom, allowing task and pipeline parallelism, both important keys to attaining high performance on distributed architectures.

Figure 7.7 shows the DCMR API setup function. A handle to a DCMR object is passed into the function, such that users can register *MRCommands* with a call to *add_mr_command()*. MRComands are an encapsulation of a map-reduce task, complete with the type of task *Map* or *Reduce*, the allowed parallelism *Dist*, *OnePerNode*, or *One*, and the callback function, also provided by the user. Dist means that full pipeline and task parallelism is allowed, while OnePerNode and One restrict the allowed task parallelism to one task running per node (possibly for disk access), or one total (for tasks such as writing to disk at the end of a full reduction of the results).

Figures 7.8, 7.9, and 7.10 show the map and reduce function callbacks which together implement the WordCount application shown in Figure 7.5.

```
// Controller setup callback function
void setup_dcmr_controller(DCMR * dcmr)
{
  dcmr->add_mr_command(MRCommand(Map, OnePerNode, &source);
  dcmr->add_mr_command(MRCommand(Map, Dist, &count);
  dcmr->add_mr_command(MRCommand(Reduce, OnePerNode,
    &reduce_count));
  dcmr->add_mr_command(MRCommand(Reduce, One, &sink));
}
```

Figure 7.7: DataCutter MapReduce Application Programming Interface: Setup

```
// source function, to create lists of words from
// the files on disk
void source(DCMR * dcmr, WorkTile * tile);
{
  const char * str = (const char *) tile->user_data;
  vector<string> files = dcmpi_file_lines_to_vector(str);

  long num_tiles = 0;
  ListOfWords l;
  for (int fx = 0; fx < files.size(); fx++) {
    ifstream in(files[fx].c_str(), ios::in);
    while (in.good() && !in.eof()) {
      string s;
      in >> s;

      l.add(s);
      if (l.size() > THRESHOLD) {
        WorkTile * out_tile = new WorkTile;
        out_tile->user_data = l;
        dcmr->add("count", l);
        l.clear();
      }
    }
  }
  if(l.size() > 0) {
    WorkTile * out_tile = new WorkTile;
    out_tile->user_data = l;
    dcmr->add("count", l);
    l.clear();
  }
}
```

Figure 7.8: DataCutter MapReduce Application Programming Interface: Source

```
// count function, to change a list of words to
// partial results of string, count pairs
void count(DCMR * dcmr, WorkTile * tile);
{
  ListOfWords * l = (ListOfWords *) tile->user_data;

  CountedWords wordmap;
  string s;
  for (list<char *>::iterator it = l.words.begin();
    it != l.words.end(); it++)
  {
    s.assign(*it);
    if (wordmap.words.count(s) == 0)
      wordmap.words[s] = 1;
    else
      wordmap.words[s]++;
  }
  WorkTile * out_tile = new WorkTile;
  out_tile->user_data = wordmap;

  dcmr->add("reduce_count", out_tile);
}
```

Figure 7.9: DataCutter MapReduce Application Programming Interface: Count Words

```
// reduce_count function, to generate final <string, count>
// pairs
void reduce_count(DCMR * dcmr, WorkTile * tile)
{
  // Get the num_maps out of the user_data in the WorkTile
  // <snip>

  for (int i = 0; i < num_maps; i++) {
    CountedWords counted_words;
    buffer->unpack("z", &counted_words);

    // Iterate over the local counted_words map, and insert
    // into the final wordmap map
    for (map<string, unsigned int>::iterator it =
        counted_words.words.begin();
        it != counted_words.words.end(); it++) {
      string s = it->first;
      unsigned int num = it->second;

      if (wordmap.words.count(s) == 0)
        wordmap.words[s] = num;
      else
        wordmap.words[s] += num;
    }
  }
  // Insert the final word-count map into a WorkTile, and
  // add that to the DCMR object
  WorkTile * out_tile = new WorkTile;
  out_tile->user_data = wordmap;
  dcmr->add("sink", out_tile);
}
```

Figure 7.10: DataCutter MapReduce Application Programming Interface: Reduce Count

7.4 Preliminary Experimental Results

Our experiments were run on the *Owens* cluster at the Department of Biomedical Informatics at The Ohio State University. For our experiments, we used one computational node with dual Intel Xeon E5520 Quad-core CPUs (with 2-way Simultaneous Multithreading), 48 GB of memory, and 500 GB of scratch disk space. Our experiments were run on CentOS with the 2.6.18-194.8.1.el5 Linux kernel, and compiled with GCC 4.1.2 using the -O3 optimization flag.

To evaluate the performance of our adaptive parallel-for engine, we will use a simple vector addition mini-application, which performs the vector addition $\vec{z} = \vec{x} + \vec{y}$ on single-precision floating-point numbers in a loop. This is a simple computation, so we will perform the function on vectors of size 2^{20} and greater. For comparison, we have also implemented the mini-application in OpenMP (OMP-PF), as shown in Figure 7.1.

Figure 7.11 shows the performance of the vector addition application. This experiment shows the performance of our adaptive solution (DC-PF in the chart) and of the OpenMP solution (OMP-PF in the chart), when increasing the size of the vectors. Both axes are in logarithmic scale, since we increase the number of loop iterations by 2x for each new data point. The chart shows that the performance of our adaptive solution is roughly 2x slower than OpenMP until about 512 million vector additions. Beyond this point, a well-tuned OpenMP implementation is no faster than our adaptive solution. Part of the slow-down is related to our adaptive technique's bootstrap initialization, where it models the application on the processors at runtime to determine the best chunk size to use. In our experiments, we performed a manual search to determine the best chunk size for OpenMP's dynamic scheduler to be 2^{20} loop iterations.

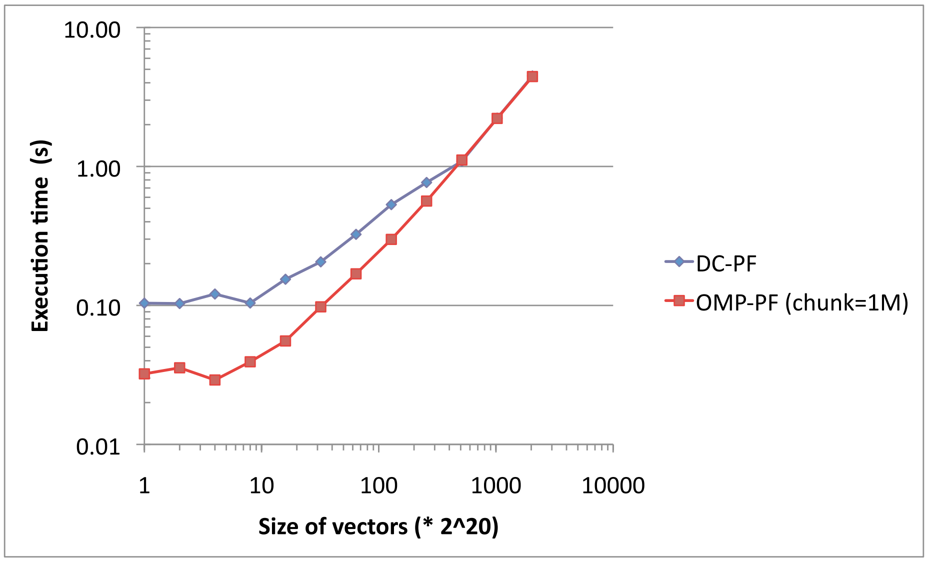


Figure 7.11: DataCutter Parallel-For Vector Addition Single Node Performance

To evaluate the performance of our adaptive MapReduce framework, we have implemented the simple word count shown in Section 7.3.2. As a point of comparison, we have implemented the same application in MR-MPI, a MapReduce implementation on top of MPI, which is presented in Section 6.3.3. Figure 7.12 shows the strong scalability results of our DCMR application when counting the words in 230 MB of text. The chart only shows application results for a single node. However, this will be the most challenging situation, since MR-MPI's distributed performance has been shown to suffer from synchronization overheads in section 6.7.

While it is clear from the chart that this application's performance is limited by the disk bandwidth available on one node (there is no speedup beyond 4 nodes), our

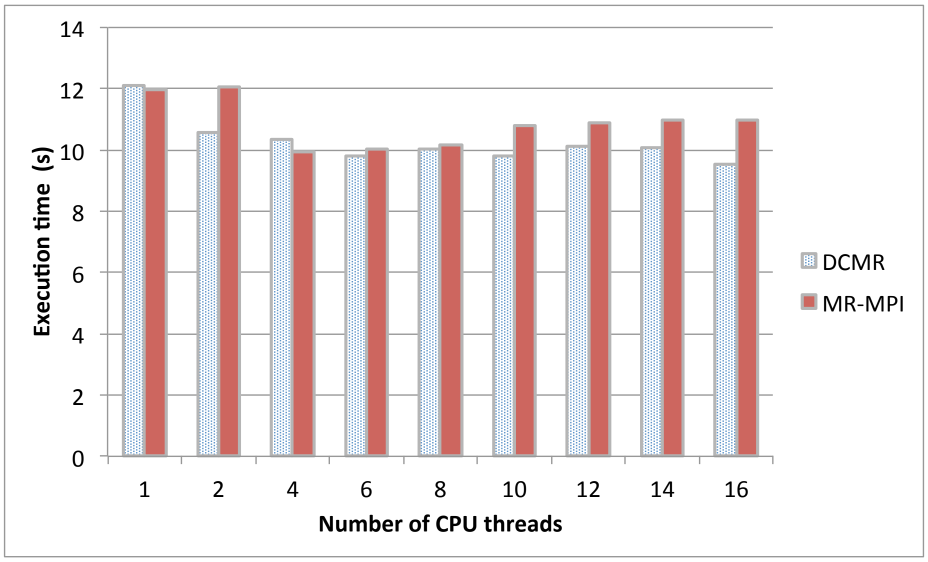


Figure 7.12: DataCutter MapReduce WordCount Single Node Performance

adaptive solution (DCMR in the chart) is able to beat an optimized solution based on MR-MPI. The average performance increase (for numbers of threads greater than 1) is 6.2%.

When using a single thread, the execution time of DCMR is worse than that of MR-MPI by 1.5%. Because only one processing thread is active, DCMR is not able to overlap any of the processing of the successive Map tasks. Further, there is an overhead to DCMR in terms of bootstrapping the adaptive runtime engine. On numbers of threads greater than two, data and task parallelism is leveraged, and the runtime system overhead is able to be better amortized and hidden. MR-MPI has a slight 1.1% increase in runtime when moving from one thread to two threads. This is caused by MR-MPI's synchronization, and its inability to overlap the shuffling of the implicitly-held hashmap and useful computation. The level of data parallelism available to the system when using two threads is insufficient to amortize this increase in overhead. However, when using more threads, MR-MPI is able to beat its single thread execution time.

7.5 Summary

This chapter shows the flexibility of our adaptive dataflow middleware in terms of supporting alternative programming paradigms. We show that we are able to support competitively simple application programming interfaces while maintaining high performance.

The Parallel-For programming interface we develop for our adaptive application tuning and load balancing technique is flexible and expressive, and includes support for heterogeneity. We show its performance is competitive with a well-tuned OpenMP implementation, even for kernels which have very little computation.

The MapReduce programming interface we develop with our adaptive technique is able to beat the fastest MapReduce implementation we have found on a classic MapReduce application, counting frequent words in a corpus of text, by 6%, on average.

Chapter 8: Conclusions and Future Work

This dissertation presented research into high-performance component-based dataflow applications and middleware to support their development. Part I focused on ad-hoc approaches to application development for heterogeneous computational resources, using two real-world scientific applications as case-studies, biomedical image analysis and synthetic aperture radar image formation. Part II focused on effectively using emerging accelerator architectures in a component-based dataflow middleware framework, both by implementing a fine-grained version of the middleware for heterogeneous supercomputer systems-on-a-chip such as the Cell Broadband Engine, and by introducing the notion of autotuning runtime performance via feedback and integrated load balance. We then presented techniques for efficiently supporting popular, simple programming paradigms such as Parallel-For and MapReduce.

The overarching intent of this research has been to ease the developer's task of implementing high-performance scientific applications in the challenging domain of hierarchical, heterogeneous, distributed supercomputers. In particular, domain expert scientists are unlikely to be well-versed in the myriad skills required to efficiently leverage complex modern supercomputer architectures, although they are often the very researchers for whom these massive systems are constructed. Modern 100,000+

core supercomputers are designed solely to run important, enormous simulations of models of real-world phenomena.

Clearly there is promise to the idea of fine-grained filter-stream programming on modern multicore processors. The work completed with DataCutter-Lite (DCL) represents a solid first step towards the future goals of programming libraries meant to provide a robust component-based dataflow framework for programming large supercomputers comprised of duplicated nodes with multiple multicore processors. Future work in this area will be in multiple directions. First, the CBE-specific implementation presented in this chapter will be used as a test-bed for further optimizations.

The next set of goals will be implementing DCL for more traditional multicore microprocessors. This will involve work into minimizing the overheads involved at every level of the software stack. However, by paying careful attention to these overheads - as well as processor traits such as cache hierarchy behavior and size - we intend to create a runtime engine and programming library competitive other options.

In parallel with both the CBE-specific DCL work and the more traditional multicore microprocessor version, work will continue to integrate DCL into DataCutter proper. This will allow a seamless application development experience, from coarse-grained dataflow at the grid or cluster level to fine-grained dataflow at the node level. Further end-to-end application optimizations would then be possible, since the layout of the entire application's filters will be known a priori. The filter stages can be appropriately sized and optimally placed for maximum application bandwidth at all granularities of dataflow.

The most long-term goals associated with DCL involve the development of algorithms to automate the creation of transparent filters for increased bandwidth, along with the placement of the instantiated filters onto physical resources. These decisions, along with the scheduling of the transmission of data buffers and the tasks to compute can be automated, alleviate the stress placed on the developer to optimize these application characteristics through trial and error.

In terms of future work on coarse-grained adaptive dataflow middleware, our algorithms need to target more irregular applications, such as sparse matrix-vector multiplication, string search, iterative solvers, data-mining, or applications where the automatic partitioning step has less of a direct effect on databuffer processing times. Additionally, the scalability of our algorithm needs to be increased by designing a hierarchical load balancing system, which will improve scalability on systems with tens or hundreds of thousands of processing cores. We envision this research as being the first step towards a dataflow runtime system capable of optimally mapping, scheduling, and tuning general, complex dataflow applications for large heterogeneous systems. We believe an adaptive tuning framework such as the one presented here will be instrumental in designing such a system. Additionally, we believe our system will be able to support myriad application programming interfaces, such that we can provide middleware which meets the needs of a large variety of scientific application developers.

The degree of performance being designed into modern HPC resources and the increasing degree of complexity involved with programming them to perform efficiently means that even expert developers are hard-pressed to make complex applications run well on the wide variety of systems. Adaptive, dynamic solutions are of vital importance to ensuring that research scientists in a number of important fields are able to meet their application's goals.

Bibliography

[1] The ABACUS project. *http://www.cs.cmu.edu/∼amiri/abacus.html.*

[2] M. Älind, M. V. Eriksson, and C. W. Kessler. BlockLib: a skeleton library for cell broadband engine. In *IWMSE '08: Proceedings of the 1st international workshop on Multicore software engineering*, pages 7–14, New York, NY, USA, 2008. ACM.

[3] R. H. Arpaci-Dusseau, E. Anderson, N. Treuhaft, D. E. Culler, J. M. Hellerstein, D. Patterson, and K. Yelick. Cluster i/o with river: making the fast case common. In *IOPADS '99: Proceedings of the sixth workshop on I/O in parallel and distributed systems*, pages 10–22, New York, NY, USA, 1999. ACM Press.

[4] K. Asanovic, R. Bodik, B. Christopher, C. Joseph, J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick. The Landscape of Parallel Computing Research: A View from Berkeley. Technical report, University of California at Berkeley, 2006.

[5] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier. StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. In *Euro-Par '09: Proceedings of the 15th International Euro-Par Conference on Parallel Processing*, pages 863–874, 2009.

[6] E. Ayguadé, R. M. Badia, F. D. Igual, J. Labarta, R. Mayo, and E. S. Quintana-Ortí. An Extension of the StarSs Programming Model for Platforms with Multiple GPUs. In *Euro-Par '09: Proceedings of the 15th International Euro-Par Conference on Parallel Processing*, pages 851–862, 2009.

[7] The BALE cluster at the ohio supercomputer center. http://www.osc.edu/supercomputing/hardware.

[8] K. J. Barker, K. Davis, A. Hoisie, D. J. Kerbyson, M. Lang, S. Pakin, and J. C. Sancho. Entering the petaflop era: the architecture and performance of roadrunner. In *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pages 1–11, Piscataway, NJ, USA, 2008. IEEE Press.

[9] P. Bellens, J. M. Pérez, R. M. Badia, and J. Labarta. CellSs: a programming model for the Cell BE architecture. In *Proceedings of the ACM/IEEE Conference on Supercomputing (SC10)*, page 86, 2006.

[10] M. Bender and C. Phillips. Scheduling DAGs on Asynchronous Processors. In *Proceedings of the Symposium on Parallelism in Algorithms and Architectures (SPAA 2007)*, 2007.

[11] M. A. Bender and M. O. Rabin. Scheduling cilk multithreaded parallel programs on processors of different speeds. In *Proceedings of the Symposium on Parallelism in Algorithms and Architectures (SPAA 2000)*, pages 13–21, 2000.

[12] X. Besseron, C. Laferriére, D. Traoré, and T. Gautier. X-kaapi : Une nouvelle implémentation extrême du vol de travail pour des algorithmes à grain fin. In *19èmes Rencontres Francophones Du Parallélisme (RenPar'19)*, Sept. 2009.

[13] M. D. Beynon, T. Kurc, U. Catalyurek, C. Chang, A. Sussman, and J. Saltz. Distributed processing of very large datasets with DataCutter. *Parallel Computing*, 27(11):1457–1478, Oct. 2001.

[14] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The parsec benchmark suite: characterization and architectural implications. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, PACT '08, pages 72–81, New York, NY, USA, 2008. ACM.

[15] W. Blume, R. Doallo, R. Eigenmann, J. Grout, J. Hoeflinger, and T. Lawrence. Parallel programming with polaris. *Computer*, 29(12):78–82, Dec 1996.

[16] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: an efficient multithreaded runtime system. *SIGPLAN Notices*, 30(8):207–216, 1995.

[17] R. D. Blumofe and P. A. Lisiecki. Adaptive and reliable parallel computing on networks of workstations. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, 1997.

[18] L. Brakmo and L. Peterson. Tcp vegas: end to end congestion avoidance on a global internet. *Selected Areas in Communications, IEEE Journal on*, 13(8):1465 –1480, oct 1995.

[19] Brook+. http://ati.amd.com/technology/streamcomputing/AMD-Brookplus.pdf.

[20] B. B. Cambazoglu, O. Sertel, J. Kong, J. Saltz, M. N. Gurcan, and U. V. Catalyurek. Efficient processing of pathological images using the grid:

Computer-aided prognosis of neuroblastoma. In *Proceedings of the International Workshop on Challenges of Large Applications in Distributed Environments (CLADE 07)*, 2007.

[21] The Cg language. Home page maintained by Nvidia. http://developer.nvidia.com/page/cg_main.html.

[22] B. Chamberlain, D. Callahan, and H. Zima. Parallel Programmability and the Chapel Language. *International Journal of High Performance Computing Applications*, 21(3):291–312, 2007.

[23] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. R. Madden, F. Reiss, and M. A. Shah. TelegraphCQ: continuous dataflow processing. In *Proceedings of the ACM International conference on Management of data (SIGMOD '03)*, pages 668–668, 2003.

[24] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *OOPSLA '05: Proceedings of conference on Object-oriented programming, systems, languages, and applications*, pages 519–538, 2005.

[25] C. Chen, J. Chame, and M. Hall. CHiLL: A Framework for Composing High-Level Loop Transformations. Technical report, University of Southern California, 2008.

[26] CUDA. Home page maintained by Nvidia. http://developer.nvidia.com/object/cuda.html.

[27] L. Dagum and R. Menon. Openmp: an industry standard api for shared-memory programming. *Computational Science and Engineering, IEEE*, 5(1):46–55, Jan-Mar 1998.

[28] A. Danalis, G. Marin, C. McCurdy, J. S. Meredith, P. C. Roth, K. Spafford, V. Tipparaju, and J. S. Vetter. The Scalable Heterogeneous Computing (SHOC) benchmark suite. In *GPGPU '10: Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, pages 63–74, 2010.

[29] K. Datta, M. Murphy, V. Volkov, S. Williams, J. Carter, L. Oliker, D. Patterson, J. Shalf, and K. Yelick. Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, SC '08, pages 4:1–4:12, Piscataway, NJ, USA, 2008. IEEE Press.

[30] L. Davis, S. Johns, and J. Aggarwal. Texture analysis using using generalized co-occurrence matrices. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 3:251–259, 1979.

[31] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. In *OSDI'04: Proceedings of the conference on Symposium on Operating Systems Design & Implementation*, 2004.

[32] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, January 2008.

[33] J. B. Dennis. Data flow supercomputers. *Computer*, 13(11):48–56, 1980.

[34] G. F. Diamos and S. Yalamanchili. Harmony: an execution model and runtime for heterogeneous many core systems. In *HPDC '08: Proceedings of the 17th international symposium on High performance distributed computing*, pages 197–200, 2008.

[35] E. D. Dolan and J. J. Moré. Benchmarking optimization software with performance profiles. *Math. Prog.*, 91(2):201–213, 2002.

[36] D. L. Eager and J. Jahorjan. Chores: enhanced run-time support for shared-memory parallel computing. *ACM Transactions on Computer Systems*, 11:1–32, February 1993.

[37] A. N. Esgiar, R. N. G. Naguib, B. S. Sharif, M. K. Bennet, and A. Murray. Microscopic image analysis for quantitative measurement and feature identification of normal and cancerous colonic mucosa. *IEEE Transactions on Information Technology in Biomedicine*, 2:197–203, 1998.

[38] K. Fatahalian, D. R. Horn, T. J. Knight, L. Leem, M. Houston, J. Y. Park, M. Erez, M. Ren, A. Aiken, W. J. Dally, and P. Hanrahan. Sequoia: programming the memory hierarchy. In *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 83, New York, NY, USA, 2006. ACM.

[39] M. Fatica, D. Luebke, I. Buck, D. Owens, M. Harris, J. Stone, C. Phillips, and D. B. CUDA tutorial at supercomputing 2007, November 2007.

[40] R. Ferreira, W. Meira Jr., D. Guedes, L. Drummond, B. Coutinho, G. Teodoro, T. Tavares, R. Araujo, and G. Ferreira. Anthill:a scalable run-time environment for data mining applications. In *Symposium on Computer Architecture and High-Performance Computing (SBAC-PAD)*, 2005.

[41] Amd stream computing. http://ati.amd.com/technology/streamcomputing/ index.html.

[42] M. Frigo and S. G. Johnson. The design and implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231, 2005. Special issue on "Program Generation, Optimization, and Platform Adaptation".

[43] F. Galilee, G. Cavalheiro, J.-L. Roch, and M. Doreille. Athapascan-1: On-line building data flow graph in a parallel language. In *Proceedings of Parallel Architectures and Compilation Techniques*, pages 88–95, Oct. 1998.

[44] M. Gallet, Y. Robert, and F. Vivien. Divisible Load Scheduling. In Y. Robert and F. Vivien, editors, *Introduction to Scheduling*. CRC Press, 2009.

[45] T. Gautier, X. Besseron, and L. Pigeon. KAAPI: A thread scheduling runtime system for data flow computations on cluster of multi-processors. In *PASCO '07: Proceedings of the international workshop on Parallel symbolic computation*, pages 15–23, 2007.

[46] A. Ghatpande, H. Nakazato, H. Watanabe, and O. Beaumont. Divisible load scheduling with result collection on heterogeneous systems. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS 2008)*, pages 1 –8, 14-18 2008.

[47] D. Göddeke and R. Strzodka. Cyclic reduction tridiagonal solvers on GPUs applied to mixed precision multigrid. *IEEE Transactions on Parallel and Distributed Systems*, PP(99), Mar. 2010.

[48] L. A. Gorham, U. K. Majumder, P. Buxa, M. J. Backues, and A. C. Lindgren. Implementation and analysis of a fast backprojection algorithm. In *Society of Photo-Optical Instrumentation Engineers (SPIE) Conference Series*, volume 6237 of *Society of Photo-Optical Instrumentation Engineers (SPIE) Conference Series*, June 2006.

[49] GPGPU. General-purpose computation using graphics hardware. http://www.gpgpu.org.

[50] M. Gschwind. Chip multiprocessing and the cell broadband engine. In *CF '06: Proceedings of the 3rd conference on Computing frontiers*, pages 1–8, New York, NY, USA, 2006. ACM.

[51] S. Guha, S. Krisnan, and S. Venkatasubramanian. Data visualization and mining using the gpu. In *Data Visualization and Mining Using the GPU, Tutorial at 11th ACM International Conference on Knowledge Discovery and Data Mining (KDD 2005)*, 2005.

[52] M. Gurcan, J. Kong, O. Sertel, B. Cambazoglu, J. Saltz, and U. Catalyurek. Computerized pathological image analysis for neuroblastoma prognosis. In *2007 AMIA Annual Symposium*, 2007.

[53] M. Hadwiger, C. Langer, H. Scharsach, and K. Buhler. State of the art report on gpu-based segmentation. Technical Report TR-VRVIS-2004-17, VRVis Research Center, Vienna, Austria, 2004.

[54] T. D. R. Hartley and U. V. Catalyurek. A component-based framework for the cell broadband engine. In *HCW '09: Proceedings of the Heterogeneity in Computing Workshop at IPDPS*, 2009.

[55] T. D. R. Hartley, Ü. V. Çatalyürek, A. Ruiz, F. Igual, R. Mayo, and M. Ujaldon. Biomedical image analysis on a cooperative cluster of GPUs and multicores. In *ICS'08, Proceedings of the 22nd Annual International Conference on Supercomputing*, pages 15–25, 2008.

[56] T. D. R. Hartley, A. R. Fasih, C. A. Berdanier, F. Ozguner, and U. V. Catalyurek. Investigating the use of GPU-accelerated nodes for SAR image formation. In *Proceedings of the International Conference on Cluster Computing, Workshop on Parallel Programming on Accelerator Clusters (PPAC)*, 2009.

[57] T. D. R. Hartley, E. Saule, and U. V. Catalyurek. Automatic dataflow application tuning for heterogeneous systems. In *Proceedings of The 17th International Conference on High Performance Computing (HiPC 2010)*, 2010.

[58] S. Huang, A. Hormati, D. Bacon, and R. Rabbah. Liquid metal: Object-oriented programming across the hardware/software boundary. In J. Vitek, editor, *ECOOP 2008 Object-Oriented Programming*, volume 5142 of *Lecture Notes in Computer Science*, pages 76–103. Springer Berlin / Heidelberg, 2008.

[59] G. C. Hunt and M. L. Scott. The Coign Automatic distributed partitioning system. In *OSDI '99: Proceedings of the symposium on Operating Systems Design and Implementation*, pages 187–200, 1999.

[60] IBM. Accelerated Library Framework. http://www-01.ibm.com/chips/techlib/techlib.nsf/techdocs/41838EDB5A15CCCD002573530063D465.

[61] IEEE. Threads extension for portable operating systems (draft 6), february 1992. p1003.4a/d6.

[62] C. Isert and K. Schwan. ACDS: Adapting Computational Data Streams for High Performance. In *IEE International Parallel and Distributed Processing Symposium (IPDPS 2000)*, pages 641–646, May 2000.

[63] C. Jakowatz, D. Wahl, P. Eichel, and D. Ghiglia. *Spotlight-Mode Synthetic Aperture Radar: A Signal Processing Approach.* Springer, New York, 1996.

[64] L. V. Kale and S. Krishnan. CHARM++: a portable concurrent object oriented system based on C++. In *OOPSLA '93: Proceedings of the eighth annual conference on Object-oriented programming systems, languages, and applications*, pages 91–108, 1993.

[65] S. Kamil, C. Chan, L. Oliker, J. Shalf, and S. Williams. An auto-tuning framework for parallel multicore stencil computations. In *Proceedings of the IEEE International Parallel Distributed Processing Sypmosium (IPDPS 2010)*, pages 1 –12, april 2010.

[66] T. Karcher, C. Schaefer, and V. Pankratius. Auto-tuning support for many-core applications: perspectives for operating systems and compilers. *SIGOPS Operating System Review*, 43:96–97, April 2009.

[67] T. Katagiri, K. Kise, H. Honda, and T. Yuba. Fiber: A generalized framework for auto-tuning software. In A. Veidenbaum, K. Joe, H. Amano, and H. Aiso, editors, *High Performance Computing*, volume 2858 of *Lecture Notes in Computer Science*, pages 146–159. Springer Berlin / Heidelberg, 2003.

[68] T. Katagiri, K. Kise, H. Honda, and T. Yuba. Effect of auto-tuning with user's knowledge for numerical software. In *Proceedings of the 1st Conference on Computing Frontiers*, CF '04, pages 12–25, New York, NY, USA, 2004. ACM.

[69] K. J. Khouzani and H. S. Zadeh. Multiwavelet grading of pathological images of prostate. *IEEE Transactions on Biomedical Engineering*, 50(6):697–704, 2003.

[70] J. Kong, H. Shimada, K. Boyer, J. Saltz, and M. Gurcan. Image analysis for automated assessment of grade of neuroblastic differentiation. In *International Symposium on Biomedical Imaging (IEEE ISBI07)*, 2007.

[71] D. M. Kunzman, G. Zheng, E. J. Bohm, J. C. Phillips, and L. V. Kalé. Poster reception - Charm++ simplifies coding for the cell processor. In *SC*, 2006.

[72] T. Kurc, F. Lee, G. Agrawal, U. Catalyurek, R. Ferreira, and J. Saltz. Optimizing reduction computations in a distributed environment. In *ACM/IEEE SC2003*, Phoenix, AZ, November 2003.

[73] V. W. Lee, C. Kim, J. Chhugani, M. Deisher, D. Kim, A. D. Nguyen, N. Satish, M. Smelyanskiy, S. Chennupaty, P. Hammarlund, R. Singhal, and P. Dubey. Debunking the 100X GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU. In *Proceedings of the 37th annual international symposium on Computer architecture (ISCA)*, pages 451–460, 2010.

[74] M. D. Linderman, J. D. Collins, H. Wang, and T. H. Meng. Merge: a programming model for heterogeneous multi-core systems. *SIGPLAN Not.*, 43(3):287–296, 2008.

[75] C.-K. Luk, S. Hong, and H. Kim. Qilin: Exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In *42nd International Sym. on Microarchitecture (MICRO)*, 2009.

[76] H. A. Mandviwala, U. Ramachandran, and K. Knobe. Capsules: Expressing composable computations in a parallel programming model. In *LCPC*, pages 276–291, 2007.

[77] The Message Passing Interface (MPI).

[78] M. Ohara, H. Inoue, Y. Sohda, H. Komatsu, and T. Nakatani. MPI microtask for programming the Cell Broadband Engine processor. *IBM Syst. J.*, 45(1):85–102, 2006.

[79] K. Olukotun, B. A. Nayfeh, L. Hammond, K. Wilson, and K. Chang. The case for a single-chip multiprocessor. *SIGOPS Oper. Syst. Rev.*, 30(5):2–11, 1996.

[80] OpenCL. http://www.khronos.org/opencl/.

[81] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Kruger, A. E. Lefohn, and T. J. Purcell. A survey of general-purpose computation on graphics hardware. *Journal of Computer Graphics Forum*, 26:21–51, 2007.

[82] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Kruger, A. E. Lefohn, and T. J. Purcell. A survey of general-purpose computation on graphics hardware. *Journal of Computer Graphics Forum*, 26:21–51, 2007.

[83] S. Pakin. Receiver-initiated message passing over RDMA networks. In *IPDPS*, pages 1–12, 2008.

[84] G. Paschos. Perceptually uniform color spaces for color texture analysis: An empirical evaluation. *IEEE Transactions on Image Processing*, 10:932–937, 2001.

[85] S. Petushi, F. U. Garcia, M. Habe, C. Katsinis, and A. Tozeren. Large-scale computations on histology images reveal grade-differentiating parameters for breast cancer. *BMC Medical Imaging*, 6(14), 2006.

[86] S. Petushi, C. Katsinis, C. Coward, F. Garcia, and A. Tozeren. Automated identification of microstructures on histology slides. In *IEEE International Symposium on Biomedical Imaging (ISBI'04)*, 2004.

[87] J. Pjeivac-Grbovi, G. Bosilca, G. Fagg, T. Angskun, and J. Dongarra. Decision trees and mpi collective algorithm selection problem. In A.-M. Kermarrec, L. Boug, and T. Priol, editors, *Euro-Par 2007 Parallel Processing*, volume 4641 of *Lecture Notes in Computer Science*, pages 107–117. Springer Berlin / Heidelberg, 2007.

[88] S. J. Plimpton and K. D. Devine. Mapreduce in MPI for large-scale graph algorithms. *(to appear in a special issue of) Parallel Computing*, 2011.

[89] V. Podlozhnyuk. Histogram calculation in CUDA, 2007.

[90] M. Puschel, J. Moura, J. Johnson, D. Padua, M. Veloso, B. Singer, J. Xiong, F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R. Johnson, and N. Rizzolo. Spiral: Code generation for dsp transforms. *Proceedings of the IEEE*, 93(2):232 –275, Feb. 2005.

[91] S. Ray. On a theoretical property of the bhattacharyya coefficient as a feature evaluation criterion. *Pattern Recognition Letters*, pages 315–319, 1989.

[92] A. Ruiz, O. Sertel, M. Ujaldón, U. Catalyurek, J. Saltz, and M. Gurcan. Pathological image analysis using the gpu: Stroma classification for neuroblastoma. In *IEEE International Conference on BioInformatics and BioMedicine (BIBM'07)*, November, 2007.

[93] A. Ruiz, O. Sertel, M. Ujaldon, U. Catalyurek, J. Saltz, and M. N. Gurcan. Stroma classification for neuroblastoma on graphics processors. *Journal of International Journal of Data Mining and Bioinformatics*, 3:280–298, June 2009.

[94] C. Schaefer, V. Pankratius, and W. Tichy. Atune-il: An instrumentation language for auto-tuning parallel applications. In H. Sips, D. Epema, and H.-X. Lin, editors, *Euro-Par 2009 Parallel Processing*, volume 5704 of *Lecture Notes in Computer Science*, pages 9–20. Springer Berlin / Heidelberg, 2009.

[95] H. Scherl, B. Keck, M. Kowarschik, and J. Hornegger. Fast GPU-based CT reconstruction using the Common Unified Device Architecture (CUDA). *Nuclear Science Symposium Conference Record, 2007. NSS '07. IEEE*, 6:4464–4466, 26 2007-Nov. 3 2007.

[96] O. Sertel, J. Kong, H. Shimada, U. Catalyurek, J. Saltz, and M. Gurcan. Computer-aided prognosis of neuroblastoma: classification of stromal development on whole-slide images. In *SPIE Medical Imaging*, San Diego, California, 2008.

[97] O. Sertel, J. Kong, H. Shimada, U. V. Catalyurek, J. H. Saltz, and M. N. Gurcan. Computer-aided prognosis of neuroblastoma on whole-slide images: Classification of stromal development. *Pattern Recognition*, 42(6):1093–1103, 2009.

[98] H. Shimada, I. M. Ambros, L. P. Dehner, J. Hata, V. V. Joshi, B. Roald, D. O. Stram, R. B. Gerbing, J. N. Lukens, K. K. Matthay, and R. P. Gastlebery. The international neuroblastoma pathology classification (the Shimada system). *Cancer*, 86(2):364–372, 1999.

[99] J. Sreeram and S. Pande. GLIMPSES: A profiling tool for rapid spe code prototyping. In *Workshop on New Horizons in Compilers (Held in Conjunction with IEEE International Conference on High Performance Computing (HiPC 2007))*, 2007.

[100] J. Subhlok and G. Vondran. Optimal latency-throughput tradeoffs for data parallel pipelines. In *SPAA '96: Proceedings of the eighth annual ACM symposium on Parallel algorithms and architectures*, pages 62–71, 1996.

[101] L. G. Szafaryn, K. Skadron, and J. J. Saucerman. Experiences Accelerating MATLAB Systems Biology Applications. In *Proceedings of the Workshop on Biomedicine in Computing: Systems, Architectures, and Circuits (BiC)*, 2009.

[102] V. Tabatabaee, A. Tiwari, and J. K. Hollingsworth. Parallel parameter tuning for applications with performance variability. In *Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, SC '05, pages 57–, Washington, DC, USA, 2005. IEEE Computer Society.

[103] M. A. Tahir and A. Bouridane. Novel round-robin tabu search algorithm for prostate cancer classification and diagnosis using multispectral imagery. *IEEE Transactions on Information Technology in Biomedicine*, 20:782–791, 2006.

[104] V. Takala, T. Ahanen, and M. Pietikainen. Block-based methods for image retrieval using local binary patterns. *Lecture Notes in Computer Science*, 3540:882–891, 2005.

[105] Intel Threading Building Blocks 2.0 for Open Source. http://threadingbuildingblocks.org/.

[106] G. Teodoro, T. D. R. Hartley, U. Catalyurek, and R. Ferreira. Run-time optimizations for replicated dataflows on heterogeneous environments. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing (HPDC)*, 2010.

[107] Nvidia Tesla GPU computing solutions for HPC. http://www.nvidia.com/object/tesla_computing_solutions.html, 2008 (accessed, January, 1st).

[108] W. Thies, M. Karczmarek, and S. Amarasinghe. Streamit: A language for streaming applications. In R. Horspool, editor, *Compiler Construction*, volume 2304 of *Lecture Notes in Computer Science*, pages 49–84. Springer Berlin / Heidelberg, 2002.

[109] A. Tiwari, C. Chen, J. Chame, M. Hall, and J. Hollingsworth. A scalable auto-tuning framework for compiler optimization. In *Proceedings of the IEEE International Parallel Distributed Processing Symposium (IPDPS 2009)*, pages 1 –12, may 2009.

[110] M. Tuceryan and A. K. Jain. *Texture Analysis, in The Handbook of Pattern Recognition and Computer Vision (2nd Ed.)*. World Scientific Publishing Co, 1998.

[111] R. C. Whaley and J. J. Dongarra. Automatically tuned linear algebra software. In *Proceedings of the 1998 ACM/IEEE conference on Supercomputing*, pages 1–27, 1998.

[112] W. Wu and P. Heng. A hybrid condensed finite element model with gpu acceleration for interactive 3d soft tissue cutting: Research articles. *Computer Animation and Virtual Worlds*, 15(3-4):219–227, 2004.

[113] W. A. Wulf and S. A. McKee. Hitting the memory wall: implications of the obvious. *SIGARCH Computer Archit. News*, 23(1):20–24, 1995.

[114] K. A. Yelick, L. Semenzato, G. Pike, C. Miyamoto, B. Liblit, A. Krishnamurthy, P. N. Hilfinger, S. L. Graham, D. Gay, P. Colella, and A. Aiken. Titanium: A high-performance java dialect. *Concurrency - Practice and Experience*, 10(11-13):825–836, 1998.

[115] Q. Yi, K. Seymour, H. You, R. Vuduc, and D. Quinlan. Poet: Parameterized optimizations for empirical tuning. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS 2007)*, pages 1 –8, March 2007.

[116] D. Zhang, Q. J. Li, R. Rabbah, and S. Amarasinghe. A lightweight streaming layer for multicore execution. *SIGARCH Computer Archit. News*, 36(2):18–27, 2008.

[117] Y. Zhang, M. Burcea, V. Cheng, R. Ho, and M. Voss. An adaptive openmp loop scheduler for hyperthreaded smps. In *In Proceedings of the International Conference on Parallel and Distributed Computing Systems (PDCS 2004)*, 2004.

[118] Y. Zhao, Y. Han, Z. Fan, F. Qiu, Y.-C. Kuo, A. Kaufman, and K. Mueller. Visual simulation of heat shimmering and mirage. *IEEE Transactions on Visualization and Computer Graphics*, 13(1):179 –189, Jan.-Feb. 2007.