View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Columbia University Academic Commons

Data-Driven Programming Abstractions and Optimization for Multi-Core Platforms

Rebecca L. Collins

Submitted in partial fulfillment of the
requirements for the degree
of Doctor of Philosophy
in the Graduate School of Arts and Sciences

COLUMBIA UNIVERSITY

2011

©2011
Rebecca L. Collins
All Rights Reserved

ABSTRACT

Data-Driven Programming Abstractions and Optimization for Multi-Core Platforms

Rebecca L. Collins

Multi-core platforms have spread to all corners of the computing industry, and trends in design and power indicate that the shift to multi-core will become even widerspread in the future. As the number of cores on a chip rises, the complexity of memory systems and on-chip interconnects increases drastically. The programmer inherits this complexity in the form of new responsibilities for task decomposition, synchronization, and data movement within an application, which hitherto have been concealed by complex processing pipelines or deemed unimportant since tasks were largely executed sequentially. To some extent, the need for explicit parallel programming is inevitable, due to limits in the instruction-level parallelism that can be automatically extracted from a program. However, these challenges create a great opportunity for the development of new programming abstractions which hide the low-level architectural complexity while exposing intuitive high-level mechanisms for expressing parallelism.

Many models of parallel programming fall into the category of data-centric models, where the structure of an application depends on the role of data and communication in the relationships between tasks. The utilization of the inter-core communication networks and effective scaling to large data sets are decidedly important in designing efficient implementations of parallel applications. The questions of *how many low-level architectural details should be exposed to the programmer*, and *how much parallelism*

in an application a programmer should expose to the compiler remain open-ended, with different answers depending on the architecture and the application in question. I propose that the key to unlocking the capabilities of multi-core platforms is the development of abstractions and optimizations which match the patterns of data movement in applications with the inter-core communication capabilities of the platforms.

After a comparative analysis that confirms and stresses the importance of finding a good match between the programming abstraction, the application, and the architecture, this dissertation proposes two techniques that showcase the power of leveraging data dependency patterns in parallel performance optimizations. *Flexible Filters* dynamically balance load in stream programs by creating flexibility in the runtime data flow through the addition of redundant stream filters. This technique combines a static mapping with dynamic flow control to achieve light-weight, distributed and scalable throughput optimization. The properties of stream communication, i.e., FIFO pipes, enable flexible filters by exposing the backpressure dependencies between tasks. Next, I present *Huckleberry*, a novel recursive programming abstraction developed in order to allow programmers to expose data locality in divide-and-conquer algorithms at a high level of abstraction. Huckleberry automatically converts sequential recursive functions with explicit data partitioning into parallel implementations that can be ported across changes in the underlying architecture including the number of cores and the amount of on-chip memory. I then present a performance model for multi-core applications which provides an efficient means to evaluate the trade-offs between the computational and communication requirements of applications together with the hardware resources of a target multi-core architecture. The model encompasses all data-driven abstractions that can be reduced to a task graph representation and is extensible to performance techniques such as Flexible Filters that alter an application's original task graph. Flexible Filters and Huckleberry address the challenges of parallel programming on multi-core architectures by taking advantage of properties

specific to the stream and recursive paradigms, and the performance model creates a unifying framework based on the communication between tasks in parallel applications. Combined, these contributions demonstrate that specialization with respect to communication patterns enhances the ability of parallel programming abstractions and optimizations to harvest the power of multi-core platforms.

Table of Contents

1 Introduction **1**

1.1 Terminology . . . 2

1.2 Problem Statement . . . 3

1.3 Requirements . . . 5

1.4 Hypothesis . . . 6

1.5 Thesis Outline . . . 7

2 Background **10**

2.1 Multi-Core Platforms . . . 10

2.1.1 Heterogeneous Architectures: IBM/Sony/Toshiba Cell BE . . . 11

2.1.2 Graphics Processing Units: NVidia GeForce 8800 GTX . . . 13

2.1.3 Tiled Architectures: Tilera TILE64 . . . 15

2.1.4 General-Purpose Architectures: Intel Core i7 . . . 16

2.2 Parallel Programming Models . . . 17

2.2.1 Threads . . . 19

2.2.2 Message Passing . . . 20

2.2.3 Graphics Languages . . . 21

2.2.4 SPMD . . . 21

2.2.5 Stream Programming . . . 23

2.2.6 Recursive Parallel Programming . . . 24

2.2.7 Map-Reduce . . . 25

2.3 Programming Patterns 26

3 An Empirical Comparison of Two Multi-Core Architectures 29

3.1 Benchmark Applications 31

3.1.1 Option Pricing 31

3.1.2 Fast-Fourier Transform (FFT) 34

3.1.3 Bitonic Sort 35

3.1.4 Smith-Waterman Sequence Alignment 36

3.2 Experiments 38

3.2.1 Option Pricing 39

3.2.2 Fast-Fourier Transform (FFT) 42

3.2.3 Bitonic Sort 43

3.2.4 Smith-Waterman Sequence Alignment 45

3.2.5 Discussion 47

3.3 Related Works 49

3.4 Summary 50

4 Flexible Filters: Load Balancing through Backpressure in Streams 53

4.1 Flexible Filters 57

4.1.1 Pipeline-Aware Mapping 60

4.2 Implementation of Flex_Split and Flex_Merge 61

4.2.1 Multi-Channel Flexible Filters 64

4.2.2 Example 66

4.2.3 Practical Implementation Concerns 70

4.3 Experiments 71

4.3.1 Benchmarks 71

4.3.2 Results 76

4.3.3 Balance of Communication vs. Computation 79

4.3.4 Adapting to Data Dependent Flow 80

4.4 Related Works 81
4.5 Summary 83

5 Huckleberry: A Data Partition Abstraction 85
5.1 Huckleberry Programming Interface 89
5.1.1 Partition Library 90
5.1.2 Example 91
5.1.3 Optimized Local Code 93
5.2 Huckleberry Parallel Code Generator 94
5.2.1 Machine Model 94
5.2.2 Stages of Execution 95
5.2.3 Example: Traversing the R-Tree 100
5.3 Experiments 102
5.3.1 Scalability 104
5.3.2 Problem Granularity 106
5.3.3 Throughput and the Role of Local Memory 108
5.4 Related Works 108
5.5 Summary 110

6 A Performance Model for Multi-Core Applications 111
6.1 Petri Nets 113
6.2 Compositional Multi-Core Performance Model 114
6.2.1 Tasks 115
6.2.2 Task Composition 115
6.2.3 Architecture 117
6.2.4 Mutual Exclusion 118
6.2.5 Data Buffering in Pipeline Communication 118
6.2.6 Communication Latency 119
6.2.7 Flexibility 121

6.3 Generating a Task Graph from a Recursive Program 122
6.3.1 Off-chip Data Swaps 127
6.4 Experiments 128
6.4.1 Mutual Exclusion 128
6.4.2 Communication Latency 130
6.4.3 Flexibility 132
6.5 Composing Different Abstractions 135
6.6 Related Works 136
6.7 Summary/Future Avenues of Research 137

7 Conclusions 138

7.1 Contributions 138
7.2 Future Directions 140
7.2.1 Parallel Index Function 141

Bibliography 144

List of Figures

2.1 Cell BE architecture. 12

2.2 GeForce 8800 architecture. 14

3.1 Dependency structure of the binomial option pricing algorithm. . . . 34

3.2 FFT computation on eight input signals. 35

3.3 Structure of the bitonic sort algorithm. 36

3.4 Performance comparison of Monte Carlo simulations for Black-Scholes option pricing. 40

3.5 Binomial option pricing. 41

3.6 Performance comparison of single-precision 2-D FFT. 42

3.7 Performance comparison of bitonic sort. 44

3.8 Performance comparison of Smith-Waterman sequence alignment. . . 46

3.9 Performance comparison of Cell vs. NVIDIA 8800 GTX. The x-axis shows the spectrum of computation and communication patterns. Data points toward the left side are more computation bound; data points on the right are more communication bound. 47

3.10 Performance comparison of RapidMind vs. platform-specific SDKs. . 49

4.1 Stream graph of the Dedup benchmark application. 54

4.2 Histogram of execution times for Dedup's Compress filter. 54

4.3 Flexible filter design flow. 56

4.4 Example stream program structure. 57

4.5 Pipeline mapping. 58
4.6 Flexible filter mapping. 58
4.7 Example stream graph. 60
4.8 Relationship of *flex_split* and *flex_merge*. 61
4.9 Block diagram of a flexible filter with n output channels. 65
4.10 Two alternatives of a flexible filter with n input channels. 65
4.11 Flexible filter timelines. 67
4.12 Time-line when filter C has a granularity of two tokens per block. . . 69
4.13 Block diagrams of benchmarks used together with their mapping on the IBM Cell multi-core processor (the non-flexible case). 72
4.14 Profile of tasks for each benchmark. 73
4.15 Gedae trace tables of the VAR benchmark. A core's timeline is black when it is busy working on a task. Green and red marks show send and receive events. 76
4.16 Speedup as the relative cost of a bottleneck filter increases with respect to the cost of communication. 79
4.17 Histogram of workload per 114 cells, % targets/workload = 7/32μs. . 81

5.1 Wave-front dependency pattern. 86
5.2 Recursive quadrant dependency pattern. 87
5.3 Huckleberry design flow. 88
5.4 Patterns of recursively applied partition methods. 91
5.5 Nested dependencies between bitonic sort recursive functions from Algorithms 3 and 5 (programmer view). 94
5.6 Abstract machine model. 95
5.7 Stages in a Huckleberry-generated parallel application. 96
5.8 The R-Tree of bitonic sort recursive functions (code generator view), unrolled until the data partition size is two data blocks. 100
5.9 Subset of bitonic sort R-Tree visited by the locality wrapper. 101

5.10 One subset of the bitonic sort R-Tree visited by the concurrency wrapper, when there are two cores with a memory capacity of two blocks. 102
5.11 One subset of the bitonic sort R-Tree as visited by the concurrency wrapper, when there are four cores, each with a memory capacity of two blocks. 103
5.12 Scaling cores: Speedup when D and m_i are fixed and the number N of available cores scales up. 105
5.13 Scaling task granularity: Speedup when I is constant, but m_i is scaled down, forcing more cores to work on the problem. 106
5.14 Scaling data size: m_i remains fixed, while I scales up, normalized w.r.t. the highest throughput instance in that benchmark. 107

6.1 Example task graph. 115
6.2 Task graph represented as a Petri net. 116
6.3 Modeling mutual exclusion. 119
6.4 Modeling backpressure. 119
6.5 Modeling communication overhead. 120
6.6 Modeling *flex_split* and *flex_merge*. 121
6.7 Incorporating flexibility into the CMCP model. 122
6.8 Modeling *flex_split* and *flex_merge* with multiple channels. 123
6.9 Overall Petri net performance model representation of a stream program. 123
6.10 Task graph tree of the Smith-Waterman benchmark. 124
6.11 Row and column stripe mappings on two cores. 125
6.12 Different mappings on four cores. 126
6.13 Multiple stages of the Smith-Waterman benchmark. The feedback loop ensures that only one data set is active at a time (pipelining the stages is not the goal in this case). 127
6.14 Estimated vs. actual throughput testing mutual exclusion. 129
6.15 Estimated vs. actual throughput testing alternative mapping options. 130

6.16 Estimated vs. actual throughput for VAR, no communication latency. 131
6.17 Estimated vs. actual throughput for VAR. 132
6.18 Estimated vs. actual speedup across several benchmarks. 133
6.19 Estimated vs. actual throughput for CFAR. 133
6.20 Estimated vs. actual throughput for JPEG. 134
6.21 Estimated vs. actual latency of the CMCP model for Smith-Waterman on four cores. 134
6.22 Performance comparison of three mappings of a 4x4 task array on four cores. 135

7.1 Composition with SPMD task blocks. 142
7.2 Composition with reduction task blocks. 143
7.3 Composition with stream task blocks. 143

List of Tables

2.1 Cell BE details. 12

2.2 GeForce 8800 GTX details. 14

2.3 TILE64 details. 15

2.4 Core i7 details. 16

4.1 Baseline pipeline mapping timeline. 66

4.2 Summary of speedup results for benchmarks where one bottleneck filter is made flexible. 78

6.1 CMCP graph size compared to original task graph. 129

Acknowledgments

First, I would like to give my sincere thanks to my advisor, Luca Carloni, for his guidance and support through this journey. His amazing energy and optimism are an example I will forever endeavor to follow. I would also like to thank my committee members, Keren Bergman, Nadya Bliss, Stephen Edwards, and Martha Kim, for their time and helpful feedback.

The hardest part about not being a student anymore is that I have to leave the community that I have belonged to for the last five and a half years and move on to a new one. Many thanks to all the the members of the CSL who I have worked with, and who have been there to listen to my practice presentations (sometimes more than once) and offer feedback. In particular, I would like to thank Cheng-Hong Li, who worked closely with me on benchmarking multi-cores, Bharadwaj Vellore, who bravely became the first user and co-developer of Huckleberry, and Nishant Shah, who worked with me developing Stream applications for the flexible filter project. I would also like to thank Eugen Schenfeld, my IBM mentor in 2007, who set me up with training at Gedae, which was essential to the development of flexible filters. I am also very thankful for the opportunity to participate in WICS, which has been a wonderful source of moral support, professional development, and great friends.

Finally, and most of all, I would like to thank my family; my parents, for supporting me from the beginning in every endeavor that I have attempted, my mother-in-law for encouraging me to pursue a PhD, my husband Seva, for helping me keep things in perspective, and my daughter Helen, for being my biggest supporter and for pointing out that Petri nets are actually many-eyed monsters.

To Seva and Helen

Chapter 1

Introduction

Since the invention of the integrated circuit, the semiconductor industry has succeeded in doubling the number of transistors on a single chip approximately every two years, a trend known as Moore's law [111]. As more resources have been added to the chip, the processing core has become faster through rising clock rates, deeper pipelines, and larger caches. However, it is becoming apparent that chasing Moore's law into the future will not guarantee continued performance improvements as it has in the past [1; 105]. The strategies that traditionally led to faster cores now offer diminishing returns at the current scale of technology. For example, deep instruction pipelines that extract instruction-level parallelism (ILP) are one of the primary innovations that have been responsible for improved performance as the number of resources on a chip increases. However, after a certain depth, control flow interferes with the ability to overlap the execution of different instructions, hindering performance gains. Additional trends such as the complexity of design verification and power constraints also contribute obstacles to continuing to design faster cores. To sidestep these challenges, the focus for improving performance is shifting: rather than make one task run twice as quickly on a faster core, the goal instead is to run twice as many tasks in the same time on two cores [6].

Thus, in last ten years multi-core chip architectures have gained in popularity, and today multi-core platforms permeate the computer industry from high-performance supercomputers to personal computers and even embedded devices such as smartphones [8; 36; 92; 114; 124]. Duplicating a core (or cores) versus making a single core faster defers the problem of improving program performance to the software designer by exposing parallelism in the hardware, thus enabling more kinds of parallelism including task, data, and coarse-grained pipeline parallelism.

1.1 Terminology

This section defines terms used throughout the dissertation.

The terms *multi-core architecture*, *multi-core platform* and *multi-core system* are used somewhat interchangeably. I distinguish them as follows: a multi-core architecture is a chip which hosts more than one processing core; a multi-core platform, or multi-core system, is the multi-core chip together with off-chip resources on a single board (e.g., memory, or even other chips such as the CellBlade Server boards).

A *program*, or *application*, represents an algorithm in a programming language and is part of the implementation. A *parallel program* is a program separated into tasks that may be executed concurrently at runtime. The complete implementation of a parallel program also includes the mapping (or mapping strategy when the mapping is dynamic) of tasks to cores on the target architecture.

A *programming model*, or *programming abstraction*, is a simplification or template that a programmer can follow in order to create a parallel program. The interface between the programmer and the model may be implemented, for example, as a library or language.

A *task* is equivalent to a function; in other words, a sequence of operations that work together to accomplish a particular goal. A task may be hierarchical, comprising several *subtasks*. There is no hard requirement about how large a task should be and

when a task should be split up into subtasks. This decision is part of the overall set of design decisions for an application, and task granularity may be tuned to suit a specific target architecture. The process of parallelization aims to split an application into independent tasks that can be executed concurrently and that equally distribute the workload. I consider only finite tasks and sets of tasks.

1.2 Problem Statement

The move to multi-core chip design exposes the parallelism of hardware resources to the programmer rather than hiding it in deep instruction pipelines, leaving the responsibility to extract parallelism (beyond ILP) to the software. In order to deliver the performance promised by multi-core design, it is essential that software tools and techniques are developed that provide this capability.

Unfortunately, many challenges hamper the design of high-performance parallel code for multi-core architectures. While a linear sequence of tasks performed one after another may represent a sequential program, a parallel program includes more than one sequence of tasks that operate in parallel and at times must synchronize and share data. This change significantly increases the complexity of program design for communication, memory management, scheduling and algorithms.

Communication. Communication complexity includes ensuring correctness in inter-task synchronization (e.g., avoiding deadlocks and maintaining a correct ordering of events). The tedious job of managing lower-level synchronization primitives such as mutex locks and condition variables often proves prone to human errors. In addition to ensuring correctness with respect to communication events, the software must expose the communication structure between tasks and create a task schedule that balances communication and computation. Situations of imbalance and poor scheduling result in idle cores and lost performance.

Memory Management. Multi-core architectures often distribute memory across the chip together with the cores and expose many of the memory details to the software designer. In some cases, the hardware maintains cache-coherency among the distributed memory banks, while other cases promote flexibility in how programs use the memory by leaving coherency optional. In either case, non-uniform memory access across the chip seems likely in future architectures due to the increasing latency of sending a signal from one end of the chip to another with respect to the clock frequency. As with communication, the exposure of lower-level memory details (e.g., data alignment, direct memory access (DMA) operations, limited local memory capacity) to the programmer increases the difficulty of creating correct and efficient programs.

Scheduling. With many cores and many tasks, it is important to schedule the tasks across the cores so that the load is balanced and the dependencies between tasks do not cause some cores to be idle frequently. The mapping from program to multi-core architecture includes not only tasks to cores, but also the mapping of the communication between tasks to the on-chip communication infrastructure, and the mapping of data to the memory (in the case of a distributed memory architecture). Mappings which increase data locality between tasks minimize inter-core data movement and improve communication overhead. However, increased locality sometimes comes at the cost of decreased concurrency, and vice versa.

Algorithms. Algorithm design must reflect the new constraints, and also break an application up into tasks that can be executed concurrently. Furthermore, it is not enough to design an algorithm with a good theoretical complexity. Because of the additional design constraints, optimizations and design decisions chosen for the sequential version of an algorithm may not be ideal in a parallel setting. For example, even though Bitonic Sort's $O(n\ log^2 n)$ complexity is less than ideal for sorting, the algorithm is often used for parallel sorting because the order of its compare-and-swap

operations is not data-dependent, and tasks are evenly sized. Thus, algorithm design must exercise an awareness of the parallel resources on which software will run.

Additionally, extremely large data sets frequently drive the need for increased performance over what single core architectures can offer. Algorithms must scale, both in terms of data size and the number of cores. Often, some parallel algorithms perform better for smaller data sets while others perform better for larger data sets. For example, the Fast Fourier Transform libraries written for the Cell SDK switch between different algorithms depending on the input data size.

1.3 Requirements

Performance, scalability, and programmer productivity are the three necessary components of any multi-core programming tool. The search for better performance, e.g., more floating-point operations per second, motivates the design of multi-core chips. Second, each generation of microprocessors increases the number of cores on a single chip. While most of the commercially available multi-cores today host only tens of cores or fewer, industry forecasts anticipate that future designs will include hundreds of cores [6]. Parallel programs must be able to continue to offer improved performance and scale with the number of cores, or they will obsolesce in a very short time. Finally, the complexity of designing a parallel program cannot scale linearly with the number of cores, or it will quickly become unmanageable. In particular, as the number of cores approaches the hundreds and thousands, most programmers will be unable to explicitly manage hundreds of independent tasks. However, programming tools can reduce this complexity, e.g., by letting a programmer write one task that is automatically expanded to hundreds.

1.4 Hypothesis

My hypothesis is that the right level of abstraction can simplify the challenges of programming multi-core systems, while still providing an acceptable level of performance and scalability. A programming abstraction supplies a simplification or template that a programmer can follow in order to create a parallel program. An abstraction can be defined by (1) what it requires of the programmer, (2) what it exposes to the programmer about the underlying multi-core platform, and (3) what it hides. High-level abstractions, which hide more and expose less, are easier for the programmer, and thus improve design time and productivity, but offer less flexibility. In contrast, low-level abstractions, which hide less and expose more, provide a greater amount of flexibility and potentially better performance, but at the cost of the programmer's time. A good abstraction hides details that the compiler handles better than humans and exposes details that humans handle better than compilers to enable a productive collaboration that creates high-performance programs rapidly.

The question: *What can be reasonably asked of a programmer?* depends on the context of the application being written, and also on the experience and skill set of the programmer. In my experience as a programmer, I believe that reasonable expectations include an understanding of the structure and dependencies of an algorithm, what kind of data is used, and what role the data plays in the algorithm. Essentially, the programmer must master all of the application-dependent details. Details that are platform-dependent, such as explicit memory management and data alignment, are better left to compilers as well as issues that are common to all or many parallel programs such as mutual exclusion locks. Automated tools may also mitigate algorithm-specific aspects of the implementation when those aspects can be abstracted into patterns common to more than one algorithm, for example, a pattern for breaking an application up into independent subtasks. However, note that task granularity, i.e., the size of an application's subtasks, differs from the decomposition

pattern of an algorithm because task granularity depends on platform features such as the number of cores and their communication infrastructure.

1.5 Thesis Outline

This dissertation presents techniques to enhance throughput performance on applications for multi-core platforms. These approaches make it easier to write parallel programs and to optimize them once they are written. I use a variety of programming abstractions throughout the dissertation, and endeavor to uncover the needs of applications by including many benchmarks implemented on real systems with the tools that I have developed. What unifies the different programming techniques and abstractions is that they expose the role of data in applications. While this might seem like a very general statement, the utilization of the on-chip interconnect and memory systems is very important in creating efficient programs. Moreover, the chips themselves are collections of computational, communication and memory units. At some level (perhaps not the level of the programming abstraction, but as a later result of a compilation) every program is converted to a collection of computational tasks, communication events and pieces of data in order to be mapped to the chips. The different abstractions expose this in different ways to the programmer. My thesis is that *the key to unlocking the capabilities of multi-core platforms is the development of abstractions and optimizations which match the patterns of data movement in the applications with the inter-core communication capabilities of the platforms.*

The rest of this section summarizes the chapters of the dissertation.

Chapter 2 surveys the landscape of multi-core hardware platforms, parallel programming models and parallel application design patterns, and shows that there is great diversity in each space. An efficient implementation of a program on a multi-core platform requires a synergy between these three areas, and application design is challenging since there are so many choices.

Chapter 3 measures a suite of five benchmarks on two high-end multi-core platforms, the Cell Broadband Engine processor and the NVIDIA GeForce 8800 GTX GPU. The benchmark tests make comparisons along two lines: first, they compare the two architectures; and second, they compare the performance of low-level vs. high-level programming abstractions. In particular, I compare platform-specific toolkits to RapidMind, a portable single program multiple data (SPMD) language. No platform emerges the winner; instead, each demonstrates strengths in certain areas and is found to be more suitable for certain benchmarks. Similarly, the SPMD abstraction proves very effective for some benchmarks and platforms, but less beneficial for others.

Chapter 4 presents *flexible filters*, a load-balancing optimization technique for stream programs. Flexible filters utilize the programmability of the cores in order to improve throughput of individual bottleneck tasks by "borrowing" resources from neighbors in the stream. An application-independent implementation of flexible filters is empirically evaluated on several stream benchmarks. The strength of flexible filters is in their simplicity. Their basic function is very straightforward, and they do not introduce heavy-weight runtime systems. Rather, all runtime load-balancing decisions are distributed among the cores and based on pipeline backpressure which is already present to prevent buffer overflows.

Chapter 5 proposes *Huckleberry*, a novel recursive programming abstraction based on data partitioning. Huckleberry abstracts the problem of breaking a program up into independent tasks, and instead requires explicit data partitioning with the Huckleberry partition library. Unlike stream programs, where all dependencies are the result of the stream structure (i.e., waiting to send or receive data along a communication channel), in recursive Huckleberry programs dependencies are detected dynamically at runtime and data movement is the result of the dependencies that are detected. Huckleberry's parallel code generator automatically parallelizes and

distributes recursive tasks which can locally detect dependencies between each other and synchronize accordingly.

Chapter 6 proposes a unifying task graph framework for all data-driven programming abstractions together with a Petri net performance model that captures the behavior of programs. A task graph separates a program into tasks and the communication dependencies between those tasks.

Chapter 7 concludes and outlines future directions of possible research.

Chapter 2

Background

In order to effectively harness the resources of a multi-core system, there must be synergy among the architectural platform, the programming model and the application being executed. Great diversity characterizes all three of these key components. This chapter provides a survey of various platforms, programming models and benchmark applications in use today, several of which will be revisited in the experiments presented in later chapters.

2.1 Multi-Core Platforms

There are a wide range of multi-core platforms available both commercially and in research. These platforms vary based on the following architectural features:

- *Cores.* Homogeneous platforms replicate the same core across the chip, which reduces design time since architects only need to design the core once. Heterogeneous platforms specialize different cores to different types of tasks in order to optimize performance. The complexity of the cores is another facet of the design. Some platforms include a very large number of simple computing units, organizing them to accomplish complicated tasks; for example, GPUs and polymorphous platforms [35; 110].

- *Memory Model.* A multi-core platform's memory model may physically distribute level-1 caches and private scratch-pad memory banks to the cores, and the cores may share lower-level caches as well. Some multi-cores provide cache coherency while others leave coherency up to the software.
- *Interconnect.* The communication infrastructure on the chip is gaining importance in chip design as more computational units are added to the chip and need to communicate with each other. One example is a tiled packet-switched network; when one core wants to communicate with another, that core may route data across the chip through one of several possible paths depending on the current network traffic. Shared network resources maximize the use of overall bandwidth while reducing the space taken up by wires. There are many other interconnect possibilities. Compared to the communication networks of distributed parallel systems (e.g., clusters), on-chip interconnects have higher throughput and lower latency.
- *Purpose.* The degree to which a platform suits different application spaces also varies. At one extreme, general-purpose multi-cores can be used with the widest range of applications; however, multi-cores with a more narrow scope designed for a particular application space provide better performance in that space (e.g., Anton is specialized for large scale molecular dynamics simulations [114]).

The rest of this section examines four multi-core architectures representing heterogeneous architectures, graphics processing units, tiled architectures and general-purpose architectures.

2.1.1 Heterogeneous Architectures: IBM/Sony/Toshiba Cell BE

The Cell Broadband Engine architecture is a heterogeneous multi-core system-on-chip originally designed for high-performance embedded applications [71; 74; 98; 104]. Originally designed for the PlayStation 3 game console, Cell processors currently

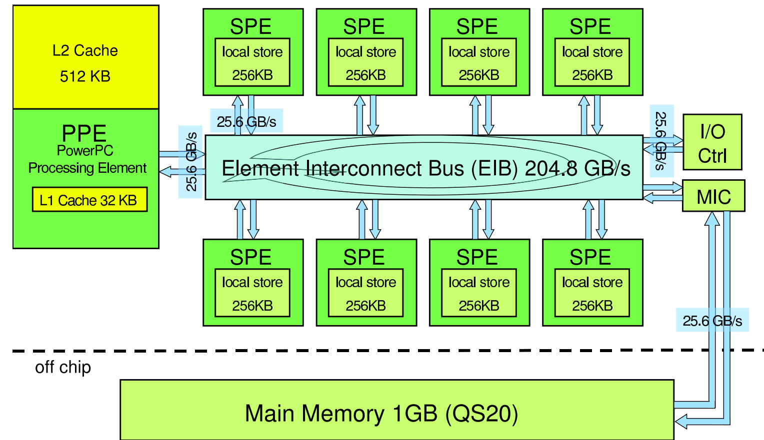


Figure 2.1: Cell BE architecture.

Cores	1 PowerPC, 8 SIMD
Memory Model	256KB private local memory per core; 512 L2 shared
Interconnect	4 circuit-switched rings
Purpose	gaming, high-performance computing

Table 2.1: Cell BE details.

compose two thirds of the processors in IBM Roadrunner, the fastest supercomputer in the Top500's list in 2008 [8]. Both IBM and Mercury Computer Systems developed high-performance servers to host Cell chips, which are used for a wide variety of applications, from physics to finance [2; 85; 118]. Cell also shares many architectural similarities with chips used in Anton, including: 128-bit registers, SIMD operations, local scratch-pad memories and DMA transfers [114].

The Cell BE features eight synergistic processing elements (SPE) and one dual-threaded 64-bit PowerPC processor (PPE). Heterogeneous architectures specialize different cores to different tasks. In this case, the Cell architecture specializes the PPE

for sequential code, and the SPEs for vector operations. Each SPE contains a *single instruction multiple data* (SIMD) processor operating on entries of 128-bit registers; an operation may, for instance, organize a register as a vector of four 32-bit integers. Each SPE core operates on a local store memory of $256KB$. Each core's local store holds both application data and code. The hardware does not supply cache coherency between the local stores; rather, software must manage data transfers between cores and to main memory via direct memory access (DMA) operations.

The Cell BE has a powerful on-chip network for inter-core communication, called the Element Interconnect Bus (EIB), made up of four circuit-switched rings [4; 79]. Two rings transfer data in one direction and two transfer data in the opposite direction, and the data arbiter only schedules data transfer circuits which take up half of a ring or less. Each ring supports up to three concurrent, non-overlapping data transfers. The EIB supports an on-chip communication bandwidth of over 200 GB/s, and the main memory uses an XDR RAM interface with a 25.6 GB/s bandwidth.

Much of the underlying architecture of Cell is exposed to the programmer, including the SIMD vector operations of the SPEs, the non-shared memory model, and the DMA data-transfer mechanisms. These features have made it notoriously difficult to program, but popular with programmers who seek extremely high performance because they can tune and optimize its resources at a very low level. I use the Cell in many of my experiments. In addition to being very flexible and tunable, the platform provides predictable performance results, in part because of the lack of hardware cache coherency operations and because the SPEs do not have complex branch prediction mechanisms.

2.1.2 Graphics Processing Units: NVidia GeForce 8800 GTX

Graphics Processing Units are specialized for graphics applications and have massive floating-point computation performance. Though originally intended for applications such as 3D image rendering, GPUs are now programmable and able to be applied

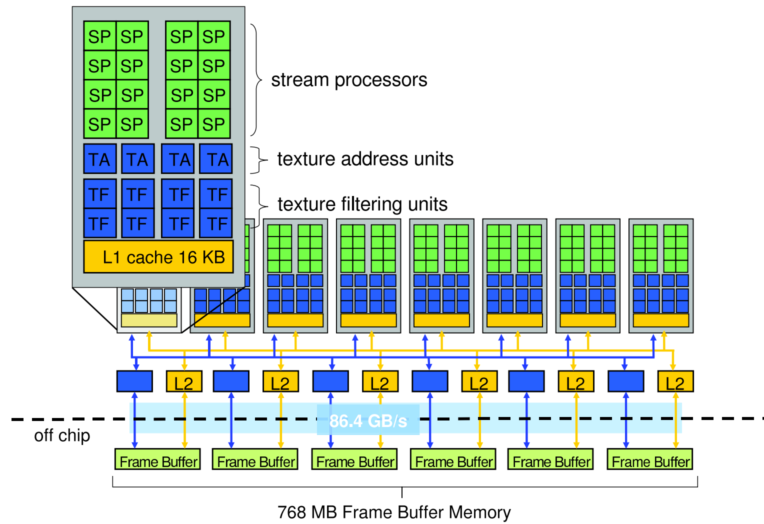


Figure 2.2: GeForce 8800 architecture.

Cores	128 “stream” processors clustered in 8 multiprocessors
Memory Model	16KB L1 per multiprocessor, L2 shared (size unknown)
Interconnect	communicate through shared memory
Purpose	graphics processing

Table 2.2: GeForce 8800 GTX details.

to any application that would benefit from their powerful computational abilities. Programming support (e.g., CUDA and OpenCL) is simultaneously being developed to open these platforms up to programmers who are not graphics specialists [33; 78], and general-purpose programming on GPUs (GPGPU) is gaining in popularity [84; 101]. The NVidia GeForce 8800 GTX is an example of a high-end programmable GPU with massive floating-point computation performance [35].

Figure 2.2 shows the high-level architecture of the GeForce 8800 GPU. It has 128 programmable processing units, called stream processors (SP), running at a clock rate of 1.5GHz. The GeForce 8800 architecture connects 768 MB of external memory

Cores	64, tiled homogeneous
Memory Model	local L1 and L2 caches, virtual L3 combines L2 caches (5MB on-chip cache total)
Interconnect	routed packet-switched network
Purpose	stream processing, networking, multimedia

Table 2.3: TILE64 details.

to the SPs by several links with an aggregated maximum bandwidth of 86.4 GB/sec. The architecture divides the SPs into 16 groups, called multiprocessors, each with 8 SPs. The SPs in one group execute instructions in a SIMD fashion, i.e., at every clock cycle they execute the same instruction on different data. If a branch instruction changes the fetch direction of some, but not all of the SPs of the same group, the execution of instructions of the two different basic blocks will be serialized (leading some SPs to stall). The SPs of the same multiprocessor communicate with each other through on-chip shared memory. Global communication between processing units across multiprocessor boundaries is only possible through a shared location in the external memory. This limits the performance of applications that present complex communication patterns.

The large number of floating point units on GPUs offer massive parallelism and enable unconventional parallelization strategies. For example, the GPU thread manager (not shown in Fig. 2.2) may aggressively speculate by forking a separate thread for each branch, and then running them in parallel until it is known which branch is the correct one.

2.1.3 Tiled Architectures: Tilera TILE64

Tilera's TILE64 architecture is an example of a tiled multi-core architecture. The Tilera Corporation descends from MIT's RAW processor project [36; 120]. Each tile

Cores	2-8, homogeneous
Memory Model	32KB L1 instr, 32KB L1 data, 256KB L2 per core; 4-8MB L3 shared
Interconnect	point-to-point
Purpose	general-purpose

Table 2.4: Core i7 details.

includes a processor, L1 and L2 caches, and a network switch. The tiled design and routed network interconnect simplify the design complexity (since the tiles which each include a core and network are replicas) and guarantee hardware scalability (since the percent of the chip devoted to the network scales linearly with the number of tiles). The RAW processor, an academic research platform, does not support hardware cache coherency, and can be programmed with C, Fortran and StreamIt [121]. The TILE64, in contrast, does provide hardware cache coherency and supports object-oriented C++ in addition to C. WaveScalar is another example of a tiled architecture [117].

2.1.4 General-Purpose Architectures: Intel Core i7

General-purpose multi-core architectures build on single core general-purpose architectures, and include all of the platform features that are customarily available in their single core counterparts, including deeply pipelined cores, and multi-level caches with hardware cache coherency [92; 93]. These architectures support widely used thread libraries such as POSIX threads. In addition, though they tend to have fewer cores in comparison with GPUs and tiled architectures, the cores they do have are more powerful, performing very well on the sequential parts of applications.

2.2 Parallel Programming Models

Programming models (also called programming abstractions) for multi-core platforms exhibit as much variation as the architectures. Parallel programming has been used for many years in supercomputers, clusters, and multithreaded processors. What is different about multi-cores is the high-performance inter-core communication available to them. Inter-core communication achieves extremely low latency and high throughput compared to more traditional parallel systems, and thus parallel programming must be reconsidered with respect to the new balance of communication and computation present in these newer systems. Programming models may be realized as either a separate programming language (such as StreamIt [121]) or as a library or extension on top of an existing language (often C [14; 29; 47; 90]). As noted in the Chapter 1, I distinguish a programming model based on (1) what it requires of the programmer, (2) what its abstraction exposes to the programmer, and (3) what it hides.

A programming model might hide or expose aspects of the underlying architecture such as the number of cores, the memory model, and properties of the interconnect. More conceptually, the model may either hide or expose different kinds of parallelism. *Data* parallelism applies the same operation to many different data instances concurrently with no dependencies between them. *Pipeline* parallelism splits an operation into a sequence of pipelined tasks which may work at the same time on different data as the data passes through the pipeline. Last, *task* parallelism splits an operation into a data flow of tasks which may include complex control flow such as splits, joins, and feedback loops. Two tasks may work separately on different data that will later be joined in the final result.

Generally, high-level abstractions optimize programmer productivity, since they enable faster design and debugging of an application. But lower-level abstractions optimize performance, since they expose more of the capabilities of the underlying resources and are more flexible. However, although low-level abstractions may pro-

vide the *potential* for the best performance, that potential will remain untapped if programmers cannot manage the burden of orchestrating all of the lower-level details of an architecture. Thus, higher levels of abstraction may alleviate the increasing complexity of multi-core architectures. And high-level abstractions do not necessarily preclude high performance. In particular, the restrictions of domain-specific abstractions correspond exactly to the capabilities and resources that are *not* needed within the target domain. As a bonus, if the communication structure of an application from another domain does match that of an application within the target domain, then it may work equally well in that setting (e.g., GPUs for non-graphics applications [101]). A given parallel programming model will often match up well to a particular hardware platform (or family of platforms). Likewise, models also match better to certain applications than others, so that an implementation of an application is a three-way match between the application, the programming model, and the hardware architecture.

Each section below describes a general programming model, including languages and libraries that utilize that model. I do not attempt to be fully inclusive of the extensive literature on this topic, but to hit the main points, with an emphasis on data-driven models. That is, models that capture the data movement within an application. The prevalence across many applications of a focus on inter-task dependencies, the balance of communication and computation, and throughput-based performance metrics underscores the importance of data movement. Focussing on how models handle data movement also highlights similarities in the different models.

This survey of parallel programming models will begin with lower-level parallel-programming abstractions: threads and message passing, which are well developed, having been used for many years on parallel platforms other than multi-cores. I categorize them as "low-level" because they expose tools for composing concurrent tasks (mutex locks, messages, etc.) without providing abstraction to the programmer outside of the memory model: threads typically imply shared memory, while message

passing typically implies distributed memory. Higher-level parallel abstractions are sometimes built on top of these models.

2.2.1 Threads

Many mainstream programming languages support threads, which provide one of the more expressive parallel programming models. Threads communicate through shared data structures, which abstract the underlying memory architecture from the programmer and work well for architectures with hardware cache coherency, but may not match well to other architectures. Threads traditionally require the programmer to explicitly program each parallel thread separately and to manage their synchronization, which can be cumbersome and error-prone. Synchronization primitives include locks, condition variables and semaphores. Many systems support POSIX threads (pthreads), the IEEE threads standard [19]. Several newer threading languages (Cilk, CUDA and Intel Threading Building Blocks) for multi-cores retain the parallel model of threads while abstracting away locks, synchronization, and other low-level details [14; 30; 66].

Cilk is a language specialized for dynamic, asynchronous parallelism [14]. The Cilk model expects the programmer to expose parallelism in an application while the compiler and run-time system manage performance details such as load balancing and managing the memory and inter-task communication. Cilk adds thread keywords such as *cilk*, *spawn* and *sink* to the C language. These keywords annotate C functions to expose parallelism. For example, the *cilk* keyword identifies parallelizable functions and the *spawn* keyword identifies subroutines that may potentially be forked off as separate threads. Removing all Cilk keywords produces a *C elision* of the code, i.e., a valid sequential C program. Thus, the programmer may annotate where the code may be forked as separate threads, but the runtime system manages the forking, joining and synchronization of threads.

The *Compute Unified Device Architecture (CUDA)* is a thread-based programming interface and environment developed by NVIDIA for general-purpose programming of its own GPUs [66]. The CUDA language abstracts the GPU hardware so that language features do not rely on a particular hardware configuration, and software may be easily ported to new GPU architectures. Programmers do not explicitly manage threads in a CUDA application; instead, they write with parallel operations through the CUDA API and the hardware thread manager handles the threading aspect of the program, which may reach thousands of threads. By handling the low level synchronization, the hardware guarantees that there are no deadlocks. For example, the SPMD CUDA operation $SAXPY$ performs $ax + y$ on two arrays x and y, with constant a. CUDA does expose tunability to programmers over the number of threads to be created and their division within and across the multiprocessor groups.

2.2.2 Message Passing

Message passing is another widespread general-purpose approach, where parallel tasks communicate through messages which can act like communication pipes; this matches well to distributed memory architectures. Like threads, message passing also requires the programmer to be responsible for explicitly programming the parallel tasks. The Message Passing Interface (MPI) is a widely accepted standard for message passing that supports both point-to-point and collective communication operations [68]. The Cell Software Development Kit (SDK) supports Cell-specific message passing operations, and work has been done to implement the MPI standard on the Cell BE [100]; however the standard MPI library is not readily available on all multi-core architectures, for instance, on GPUs.

SHIM provides a deterministic concurrency message passing model which has been implemented for both shared memory and distributed memory multi-cores [43; 125]. SHIM's model guarantees that communication events between tasks occur in a

deterministic order, eliminating data races and simplifying the debugging of concurrency errors such as deadlocks.

2.2.3 Graphics Languages

The Open Graphics Library (OpenGL) is a language specification for programming graphics hardware [112]. It is specialized to graphics functions, in particular, rendering into a framebuffer, and many of its function calls enable the programmer to draw different types of 2D and 3D objects such as lines and polygons.

The Open Computing Language (OpenCL) is a general-purpose language intended to help programmers take advantage of the impressive computing capabilities of GPUs for non-graphics applications [78]. Unlike CUDA, which only supports the family of NVIDIA GPUs, OpenCL provides an open standard, and supports a variety of heterogeneous systems, including non-GPU systems. In OpenCL, the programmer creates *kernels*, programs portable across OpenCL devices and *host programs* that run on a specific host. Before a kernel executes, the OpenCL runtime model defines the index space of a data set for each kernel instance, and breaks the index space up into separate *work-groups*. At runtime, the host executes a variety of operations, including kernel operations on work-groups, memory operations and synchronization operations. The flexibility of the programming model allows for different types of parallelism including data parallelism as well as task parallelism.

2.2.4 SPMD

Single Program Multiple Data (SPMD) programming exploits the data-parallelism of an application by applying the same code in parallel on separate data, for example, the elements of an array. A pure SPMD operation involves no inter-core communication. Therefore this style of parallel programming has been applied to many parallel systems that do not have low latency inter-core communication, including wide-area distributed ones like SETI@home. Several startup ventures have designed

commercial SPMD programming languages in recent years for multi-core platforms (e.g., RapidMind [90] and PeakStream, acquired by Intel and Google, respectively).

RapidMind grew out of Sh, a tool for programming GPUs intended to both unify shader programs with their host programs and provide a more general-purpose programming platform for GPUs than was previously available [89]. RapidMind provides C++ libraries that add a few new types: `Array`, `Value`, and `Program`. A programmer may invoke a RapidMind program with a RapidMind array as input, and the program will execute separately on each array element. For example, consider the following snippet of C++ code:

```
main {
   float a[N], b[N];
   // ... initialize a[] and b[] ...
   for(int i=0; i<N; i++) {
      a[i] *= b[i];
   }
}
```

The code above is rewritten in RapidMind by first creating a RapidMind `Program`, that can be called as a subroutine:

```
Program vector_mult = BEGIN {
   In<Value1f>a;
   In<Value1f>b;
   a = a*b;
}
```

where `Value1f` indicates that a and b are each floats (`Value4f` would indicate a vector of four floats). Next, `vector_mult` replaces the *for* loop, and `a[]` and `b[]` are defined with RapidMind types.

```
main {
   Array<1,Value1f> A(N);
   Array<1,Value1f> B(N);
   // ... initialize values ...
   A = vector_mult(A,B);
}
```

RapidMind automatically parallelizes and distributes the program over the target platform, hiding platform-specific thread management and data-transfer operations from the programmer. RapidMind also supports reduction functions, multi-dimensional `Array` types, and data views such as shifting or striping for manipulating the arrays.

2.2.5 Stream Programming

Stream processing captures the data flow model of computation, and applies to a wide range of applications including high-performance embedded applications, signal processing, image compression, and continuous database queries [20; 42; 113]. The stream processing abstraction decomposes an application into a sequence of data items (*tokens*) and a collection of tasks (referred to as *filters* or *kernels*) that operate upon the stream of tokens as they pass through them. Filters communicate with each other explicitly by exchanging the tokens through point-to-point communication channels. Stream programs expose pipeline, data and task parallelism. StreamIt, a language from the research community, and Gedae, a commercial language, are described below.

StreamIt is a stream language and compiler developed at MIT [121]. A StreamIt program comprises stream tasks, called *filters*, which accommodate a single input and a single output and use *push*, *pop*, and *peek* operations to interact with the input and output data streams. The language also provides control flow filters to allow for splits and joins in the stream. StreamIt has been used as a foundation for research in compiler analysis and optimizations [3; 58; 122]. Other research stream

projects include the Brook stream language and the Imagine stream processor and programming model [18; 76].

Gedae [54] is a commercial stream language specialized for embedded signal processing applications. Similar to StreamIt, a Gedae application also comprises stream tasks, called *blocks* in Gedae, which communicate with each other through communication pipes. Gedae supports blocks that have more than one input and output stream. During runtime, a typical Gedae block will fire, consuming a fixed number of data tokens from its input and producing a fixed number on its output. The language also provides support for nondeterministic and dynamic streams where the number of tokens consumed or produced may vary.

Chapter 4 explores the stream abstraction further with flexible filters, a load-balancing optimization method for stream programs.

2.2.6 Recursive Parallel Programming

Recursive parallel programming models utilize the divide-and-conquer aspect of recursive functions to break a problem up into concurrent tasks, leveraging the hierarchical nature of the memory hierarchy as well as locality in the application [12; 47; 80; 94]. Many parallel programming systems have used recursive models of parallelism because recursion concisely captures patterns of dependencies and exposes temporal and data locality and is a natural fit for exposing concurrency in a program. A recursive function corresponds to a hierarchical task graph with a tree topology, which conveniently separates an application up into a set of balanced tasks. Working on tasks under one branch of the task graph at a time promotes data locality between concurrently executing tasks. Furthermore, increasing or decreasing the depth of the tree can tune the task granularity by adjusting the size of leaf tasks.

Recursive parallel programming models for vector processors were developed in the late 1980s and early 1990s [12; 59]. Unlike multi-core architectures today, these systems presented relatively high inter-node communication costs (throughput around

1 Gbps and application-level latency at 40-100 microseconds), and the implementation of the model on such a system reflected these constraints. For example, the divide-and-conquer Algorithmic Skeleton is implemented with SPMD parallelization based on the *powerlist* data structure [24; 94].

More recent works that target multi-core architectures typically rely on compilers for parallelization. For example, compilers can parallelize divide-and-conquer programs by analyzing memory references to detect dependencies [64; 109]. Some parallel languages also explicitly expose divide-and-conquer patterns to the compiler. Cilk also includes support for recursion [14]. The Sequoia programming language uses hierarchical program design to leverage data locality in the memory hierarchy of parallel system [47; 80]. In Sequoia, different layers of the hierarchical tree are associated with different levels of memory. Sequoia isolates concurrent tasks so that they do not synchronize, but communicate through their parent task (which may be mapped to the same core).

Chapter 5 presents Huckleberry, a new recursive programming model developed as part of my research, and its library and code generator [29]. Huckleberry parallelization relies on explicit data partitioning through Huckleberry's partition library, but abstracts the decomposition of algorithms up into concurrent threads.

2.2.7 Map-Reduce

Map-Reduce separates an application into data-parallel *map* and *reduce* steps [39]. Google developed the Map-Reduce model for applications which process very large data sets. The map step breaks the data set up and distributes it among processing nodes performing the map operation. The map step produces intermediate <*key,value*> pairs and then passes them to the reduce step according to the keys. The reduce step merges all data with the same key together. The Map-Reduce model could also be used in multiple map and reduce stages. Phoenix implements Map-Reduce for shared-memory multi-core platforms [108]. Phoenix creates parallel tasks

for the map and reduce steps and manages dynamic load balancing across the cores. The stages of map and reduce execute separately. Phoenix includes built-in fault recovery for faulty tasks, which are detected through timeouts.

2.3 Programming Patterns

A parallel application is an application made up of more than one task such that some or all of the tasks are capable of being executed in parallel. Each parallel application exhibits a distinct pattern of dependencies among its tasks. Some works classify applications according to a taxonomy based on their computational requirements, communication dependencies, and memory access patterns [6; 24; 88]. These classifications are called programming patterns. For example, Asonovic *et al.*'s report, *A View from Berkeley*, identifies a number of programming patterns related to the structure of applications, called "dwarfs" in the report (for example, dense and sparse linear algebra, N-body methods, map-reduce, graph traversal, dynamic programming, etc.). In theory, all applications which fall into a category share common properties and constraints which might be exploited by tools specific to that category. Thus, the reuse of programming tools optimized for a specific class of applications may mitigate the task of designing parallel software. The taxonomy of patterns continues to expand as researchers consider more applications. Indeed, within five years of the original Berkeley report, the set of seven "dwarfs" grew to thirteen.

The problem of choosing a generalized application classification resembles that of choosing a programming model. However, an important distinction between the classification of an application and a programming abstraction is that the same application may be implemented using different programming models. For example, the tools presented in later chapters implement the bitonic sort benchmark both with the SPMD abstraction and with the recursive programming abstraction.

Several programming patterns from the Berkeley report are highlighted below:

- **Graph Traversal.** Graph traversal algorithms work with graph data structures and require indirect table lookups through memory pointers. Some examples of graph traversal applications are router lookup in a network, decision trees, and many graph theory algorithms such as shortest path, vertex cover, etc.

- **Dynamic Programming.** Dynamic programming problems often favor a bottom-up approach, which first solves subproblems, saving the results in a table, and builds up an overall solution from the subproblems. This approach befits problems that have the optimal substructure property, i.e., an optimal solution to a problem includes optimal solutions to all of its subproblems as well [32].

- **Dense Linear Algebra.** The class of dense linear algebra captures MATLAB-like functions, where data is densely packed into matrices or vectors. Regular data access patterns typically characterize dense linear algebra.

- **Sparse Linear Algebra.** Sparse linear algebra packs data sparsely into matrices and vectors with many zero value elements. The data is often then stored with compression over blocks of the matrix, and data is accessed through indexed loads and stores.

- **Spectral Methods.** Spectral methods include functions that operate in the spectral domain (e.g., FFT converts data to a sum of frequencies, and DCT converts data to a sum of cosine functions). Transformation to and from the spectral domain from space and time domains requires all-to-all communication, often organized in "butterfly" stages where data sharing is symmetric between two tasks.

- **MapReduce (originally Monte Carlo).** MapReduce parallel functions comprise many independent tasks and little or no inter-task communication. "Embarrassingly parallel" workloads fit into this pattern. This pattern captures the

applications supported by the SPMD model (Sec. 2.2.4), and also the MapReduce model (Sec. 2.2.7), since the tasks within a map or reduce step are independent from one another.

The categories above correspond to the overall classification of an application (or function). In an alternative approach to programming patterns, Mattson *et al.* identify recurring patterns at several stages along the design flow of a parallel application. The design space is separated into patterns for (1) finding concurrency, (2) algorithm structure, (3) supporting structures, and (4) implementation mechanisms. For example, the supporting structures design space includes the SPMD, master/worker, loop parallelism, fork/join, shared data, shared queue, and distributed array patterns.

Chapter 3

An Empirical Comparison of Two Multi-Core Architectures

Given the diverse range of multi-core architectures (Chapter 2), the question arises: which one is the best? It is unclear whether there can be a general-purpose multi-core architecture, i.e., a single architecture that is able to perform reasonably in all application domains. In particular, no architecture outperforms all others in gaming, image processing, dense linear programming, and so on. Although parallel benchmarks are being developed, they typically support different standards; e.g., the PARSEC benchmark suite employs pthreads and OpenMP, while SPEC MPI2007 requires MPI support [10; 96]. The lack of standardized programming tools at this point in time prevents a complete head-to-head comparison of two architectures. The reason there are no standardized programming tools across all platforms is both because the architectures are relatively new and quickly changing and also because tools are often designed with a particular architecture and application domain in mind. Because of these two issues – the diversity of multi-core architectures, and the lack of standardized tools – the evaluation of different platforms is analogous to comparing apples and oranges.

The diversity of on-chip interconnects further complicates the evaluation of performance on multi-core architectures. For example, the Cell architecture uses circuit switched rings to connect the cores with very high bandwidth, while NVIDIA GPUs use small shared memory as the only way for two cores to communicate on-chip. Given some application, the application's subtasks will have a dependency structure. The synchronizations from dependencies and communication between tasks needs to take place through the interconnect. Because application performance depends on the efficiency of communication over the network, not just the speed of computation for individual tasks, finding a good match between the network and the application dependencies is critical.

Mapping application dependencies down to the interconnect involves two steps. First, the programmer implements the algorithm with the abstraction provided by the programming model. Next, the compiler maps the programming abstraction representation of the application to the available hardware resources. Programming abstractions that mirror hardware support for different concurrency features are desirable. For example, a model with a shared memory abstraction may be implemented on either an architecture that provides hardware cache coherency or one that provides only scratchpad local memories. In either case, an inefficient mapping will result in lost performance, but an efficient mapping of shared memory will likely be easier for the compiler when hardware cache coherency is available.

Despite the challenges of comparing multi-core architectures and programming models, it is nonetheless beneficial to quantify performance so that it is possible to identify the ideal platform and programming abstraction for a specific application. Once a successful match between model and application has been made, other applications of the same programming pattern may also be matched to the model [6; 88].

This chapter presents experiments that compare several benchmarks on two leading multi-core processors: the Cell BE and NVIDIA GeForce 8800 GTX GPU (GeForce

8800), using several programming tools [25]. The benchmark suite includes applications which range from computation intensive (e.g., option pricing) to communication intensive (e.g., sorting). I compare RapidMind, a high-level portable SPMD language, to low-level architecture-specific software development kits in order to better understand how much performance (if any) is lost when switching from a low-level to a high-level programming abstraction. The comparison of the Cell versus the GeForce 8800 is a complementary goal since the programming tools must expose and utilize the features of the platforms. This chapter also provides a detailed description of several benchmarks and illustrates the possible range of dependency structures across multiple applications. The results imply that the Cell BE suits communication-rich applications, while the GeForce 8800 is stronger for computation-rich applications. Moreover, the SPMD abstraction works better with GPU architectures than with streaming architectures like the Cell.

3.1 Benchmark Applications

This section introduces five benchmark applications.

3.1.1 Option Pricing

An option is a right to sell or buy a financial asset at a predetermined price on a future date. The price of an option is determined by factors such as the current price of the underlying financial asset, that asset's volatility, and the timespan between the current and future date. An option itself can be traded and its price is the discounted profit made by exercising the option. Two kinds of options are described below: Black-Scholes options and binomial options. The main difference between the two is that time is continuous for the Black-Scholes model and discrete for binomial model, although in some cases, the two models converge (e.g., when there are no dividends). Black-Scholes options are typically used to price European options, where the option

to buy or sell may only be exercised on the expiration date; binomial options, on the other hand, are more commonly used to price American options, which may be exercised at any time up until the expiration date.

Black-Scholes Option Pricing. The Black-Scholes option pricing model assumes that the price fluctuations of the underlying financial asset can be modeled as geometrical Brownian motion [11; 91]:

$$dS_t = \mu S_t dt + v S_t dW_t \tag{3.1}$$

where S_t is the asset price at time t, W_t is a Wiener random process, and μ (drift) and v (volatility) are two constants. Based on Eq. 3.1, the asset price at time T is:

$$S_T = S_0 e^{((\mu - 0.5v^2)T + vT^{1/2}N(0,1))} \tag{3.2}$$

where $N(0,1)$ is a normally-distributed random number between 0 and 1.

Based on Eq. 3.2, Monte Carlo simulations can be applied to estimate the expected value of S_T at a future time T[1]. Each simulation consists of the following two steps:

1. generate a normally distributed random number x;
2. replace $N(0,1)$ in Eq. 3.2 with x to get one price S'_T.

The average of all the S'_T's obtained at Step 2 is an estimation of the expected value of S_T in Eq. 3.2. For the estimation of the expected S_T to converge, a sufficient number of simulations must be performed. Since all simulations are independent from each other, they can be exercised by parallel processes between which there is little communication. Hence, this is an *embarrassingly parallel* workload that can be tackled by distributing the various simulations evenly across the processing elements

[1] The standard Black-Scholes equation is actually closed-form, and does not require simulation. However, if any underlying assumptions to the equation are altered, then it may no longer be closed-form. Other financial benchmarks, such as VAR in Chapter 4, also require normally distributed random numbers.

of the multi-processor hardware platform. Such a workload represents one extreme on the spectrum of computation and communication patterns.

Binomial Option Pricing. Binomial option pricing is a discretized version of Black-Scholes option pricing. Like the Black-Scholes model, the binomial options model also assumes that the value of the underlying asset follows a Brownian motion. The binomial option pricing model is popular for American options as mentioned and also for other cases where no closed-form solution exists.

This is the algorithm for pricing an option with the binomial options model:

- Start at the leaves of a binomial tree. Each leaf has one possible value that the asset could have after n timesteps. It is assumed that the asset will only go up or down at a fixed rate during each timestep (let u be the upward factor and d be the downward factor), so there are $n + 1$ leaves.

- At the leaves, calculate the value of an option based on the projected value of the asset. If a leaf has a value of S from the last step, then the value of a call option (option to buy) at that leaf will be $S - X$, where X is the *strike price* (i.e., price at which the option is exercised).

- Once the value of the option at each of the leaves is determined, work backwards up the tree calculating each node's value based on the value of its two children. Each node has two children since the asset can either go up or down from each point in time.

- For each layer of the tree calculate the option price until the root of the tree is reached, which will be given the value of the option at time 0, the current time.

The complexity of this algorithm is $O(n^2)$ since a binomial tree is fully traversed; however, the space required for the calculation is just $O(n)$ since older layers of the tree can be overwritten as the newer layers are calculated. The dependency structure is illustrated in Fig. 3.1.

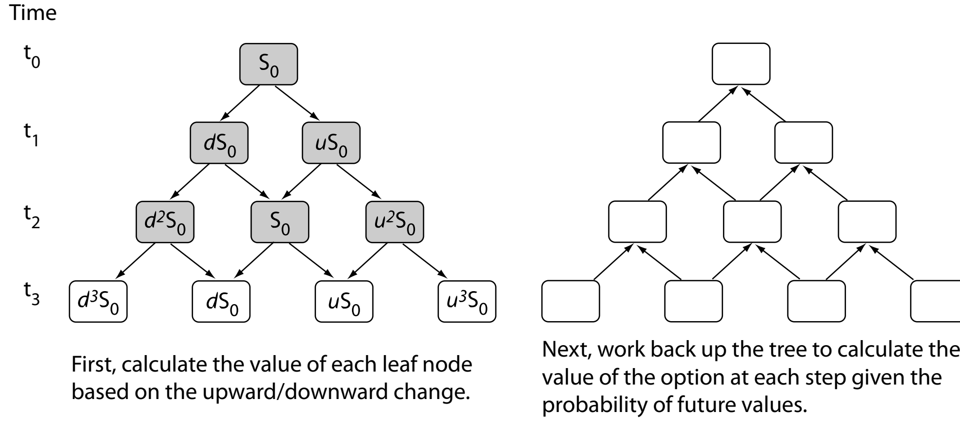


Figure 3.1: Dependency structure of the binomial option pricing algorithm.

3.1.2 Fast-Fourier Transform (FFT)

Fast Fourier Transform (FFT) is a divide-and-conquer algorithm to compute the discrete-time Fourier transform (DFT), which converts discrete signals from the time domain to the frequency domain. Given an input of N discrete signals ($x_1, x_2, \ldots, x_N$), the DFT ($X_1, X_2, \ldots, X_N$) is defined as follows:

$$X_k = \sum_{n=0}^{N-1} x_n \, e^{-\frac{2\pi i}{N} nk}, \quad k = 0, \ldots, N-1$$

While a naïve implementation of this convolution requires $O(n^2)$ floating point operations, the FFT requires only $O(n \log n)$ of them.

Figure 3.2 illustrates a simple implementation of FFT on eight signals. The input signals are fed into the left side, and the transformed signals are produced as output from the right. The FFT is divided into three stages, and each stage has four "butterfly" operations, each of which is applied to two signals (this is the so called radix-2 FFT). Since the butterfly operations of a single stage are independent, they can be executed in parallel. The FFT belongs to the class of *spectral methods* [6].

The FFT benchmark requires both intensive computational and communication support from the hardware execution platform. From the computational aspect, a butterfly operation takes in two inputs and a complex number $\omega(k)$, called the

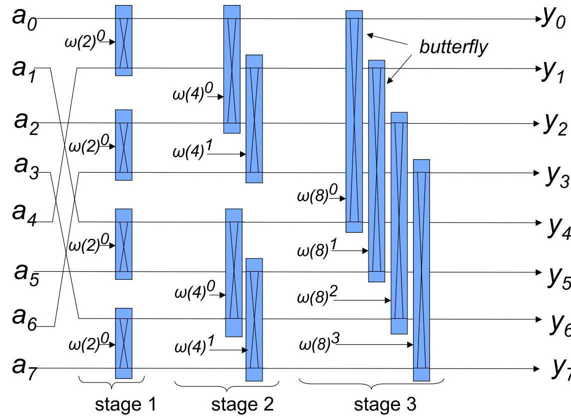


Figure 3.2: FFT computation on eight input signals.

"twiddle factor", which is the k-th root of unity, and performs two sets of floating point additions and one set of multiplications. The calculation of each twiddle factor requires two trigonometric function calls. The communication aspect of FFT is also very challenging since the inter-core communication pattern changes with each stage of the algorithm.

3.1.3 Bitonic Sort

Bitonic sort is a popular $O(n\ log^2 n)$ sorting algorithm for parallel architectures. Although its complexity is less optimal than $O(n\ log\ n)$ sorting algorithms such as merge sort, bitonic sort is desirable because the order of its compare-and-swap operations is not dependent on their outcome.

The bitonic sort algorithm uses a divide-and-conquer approach. To sort a list of elements, the list is broken into two evenly-sized pieces. The two pieces are sorted in opposite directions and then merged together with the $(O(n))$ bitonic merge operation. Bitonic merge works as follows: assuming there are two lists sorted in opposite directions, first compare-and-swap the first element of the first list to the first element of the second list; next, compare-and-swap the second element of the first list to the second element of the second list; etc. Figure 3.3 shows the steps of the bitonic sort

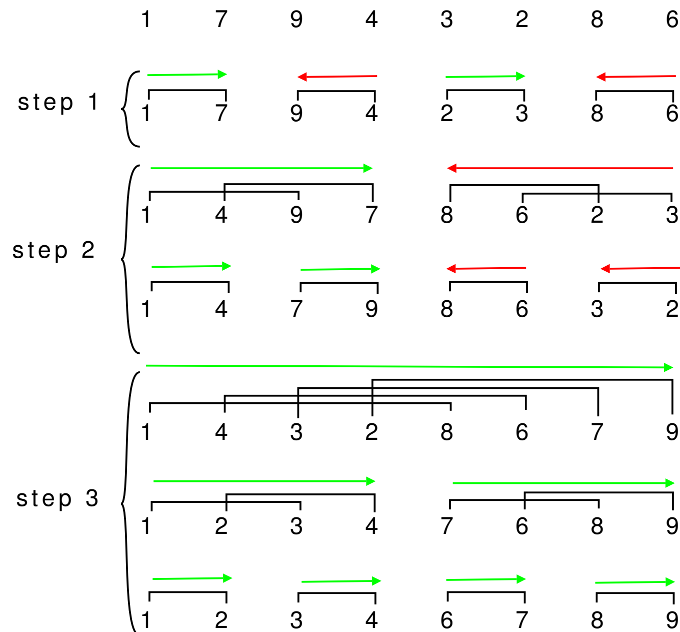


Figure 3.3: Structure of the bitonic sort algorithm.

algorithm sorting a list of eight integers. In Step 1, every other set of two elements is sorted in ascending order, and the other sets are sorted in descending order. In Step 2, every other set of four elements is sorted in ascending order, and so on.

Sorting, in general, is a task that requires very little computation, but intensive data movement. Bitonic sort can be implemented in a variety of ways. A recursive implementation is intuitive from the divide-and-conquer definition of the algorithm; however, non-recursive iterative implementations are typically used to avoid the overhead of recursive function calls.

3.1.4 Smith-Waterman Sequence Alignment

Smith-Waterman sequence alignment is a bioinformatics application that determines the similarity of two sequences, such as protein sequences [60; 115]. The goal is

to compute a score given a pair of sequences, and this is done by building a two-dimensional matrix, where the columns correspond to the characters of one sequence and the rows correspond to the characters of the other sequence.

Given one character from one sequence and one character from the other sequence, a score matrix determines the similarity of the two characters. If they have a high similarity, they get a high score. Otherwise they get a low or negative score. The overall sequence alignment score is built up from the individual character similarity scores. Gaps are allowed; for example, comparing sequences `abbcdabab` and `ababab`, the following is one possible alignment between them. This alignment skips `bcd` in the first sequence.

```
abbcdabab
ab---abab
```

It is also possible to align sequences where not every character matches. For example, comparing `abbcdabab` and `abaabab` below, the third character does not match.

```
abbcdabab
aba--abab
```

I consider the case where there is a fixed negative cost for starting a gap, and a fixed cost for extending the gap by one character. There are other schemes for sequence alignment where the gap cost follows a function, but these cases in general are less tractable.

To compute the score for an alignment, the algorithm fills the matrix starting at the upper left corner, $m[0][0]$. The value for each remaining element, $m[i][j]$, depends on $m[i-1][j]$,$m[i][j-1]$, and $m[i-1][j-1]$, and the similarity score for the ith element of the first sequence and the jth element of the second sequence.

Sequential algorithms typically traverse the matrix one row at a time or one column at a time. However, dependencies between elements of the same row and column inhibit this approach for parallel algorithms. The most straightforward way

to parallelize the execution of this algorithm is to compute the diagonals one at a time.

3.2 Experiments

This section presents the experimental comparisons of the Cell BE versus the GeForce 8800, and of RapidMind versus low-level programming tools.

Two Cell platforms are included in the experiments: the IBM Cell blade QS20 equipped with two Cell BE multi-core processors, and the PlayStation 3 with one Cell processor, of which six SPEs are enabled.

The programming tools used are summarized in the following categories:

- High-level abstractions: RapidMind supports both the GPU and Cell, and is a high-level SPMD abstraction;
- Mid-level abstraction: CUDA is an NVIDIA GPU-specific thread language;
- Low-level abstraction: The Cell SDK provides a highly tunable Cell-specific programming tool, and OpenGL provides a graphics programming language for GPUs.

One goal of these experiments is to understand how much performance is lost when moving from processor-specific software development kits to a high-level portable multi-core development platform, which attempts to be a standardized programming and benchmarking tool across multi-core platforms. The second goal is to compare the performance of Cell and the GeForce 8800, and understand which architecture better suits different application domains (outside of the domains for which they are specially designed – i.e., gaming and graphics, respectively).

Most of the low-level benchmarks are drawn from publicly available hand-tuned implementations, and represent very good performance for each respective platform. In contrast, most of the high-level RapidMind implementations were developed by

non-experts over the duration of a summer internship. One exception is the bitonic sort on Cell RapidMind implementation, which is provided by RapidMind. Note that although the base RapidMind language is portable, RapidMind implementations for the Cell and the GeForce 8800 often differ to take advantage of the native SIMD instructions on both architectures. Furthermore, certain language features are only available on one of the two architectures, and so some of the benchmarks are not available through RapidMind on both architectures. The results highlight some of the challenges and lessons learned from implementing the benchmarks using the SPMD abstraction. The mid-level CUDA benchmarks are a mix of both publicly available optimized benchmarks, and in-house implementations.

3.2.1 Option Pricing

Black-Scholes Option Pricing. The experiments for Black-Scholes option pricing include three distinct implementations of Monte Carlo simulations. The three implementations compute the same pricing of an option, but differ in the way of generating and transforming random numbers. The first two approaches adopt the Mersenne-Twister random number generator [87], but use different methods for transforming uniformly distributed random numbers to normally distributed ones. The third approach uses the Hammersley sequence, a low discrepancy sequence, instead of pseudo-random numbers[2].

Figure 3.4 reports the run time of each implementation for two hundred million simulations on both the Cell blade and the GeForce 8800. Comparing the performance of hardware using platform-specific SDKs, the GeForce 8800 (using CUDA) outperforms the Cell blade (using Cell SDK) in all three variations of Monte Carlo simulations by a wide margin. Note that using the Hammersley sequence on the GeForce 8800 boosts the performance significantly, compared to performances of the other two approaches based on pseudo-random numbers. This is because the low-

[2]This is referred to as "quasi Monte Carlo".

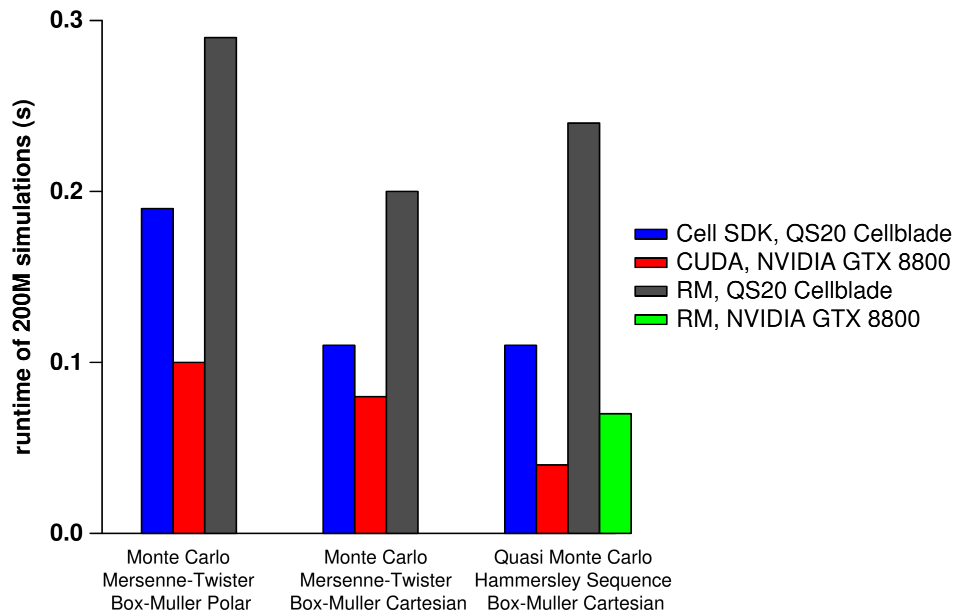


Figure 3.4: Performance comparison of Monte Carlo simulations for Black-Scholes option pricing.

latency texture memory of GPUs can be exploited to store the read-only lookup table required by the Hammersley sequence algorithm. However, using the Hammersley sequence is no more advantageous on the Cell processor, which does not provide such specialized hardware.

Comparing the performance of RapidMind and platform-specific SDKs on the same hardware, the SDK versions run faster than their corresponding RapidMind versions both on the Cell and on the GeForce 8800. Note that on the GeForce 8800, however, the RapidMind implementation of Mersenne-Twister random number generator cannot run on GPUs, because the algorithm reads and writes a local array, which is not supported by the latest RapidMind backend (version 2.1) of GPUs at the time of this study[3].

[3]This restriction is lifted in RapidMind's Cell backend.

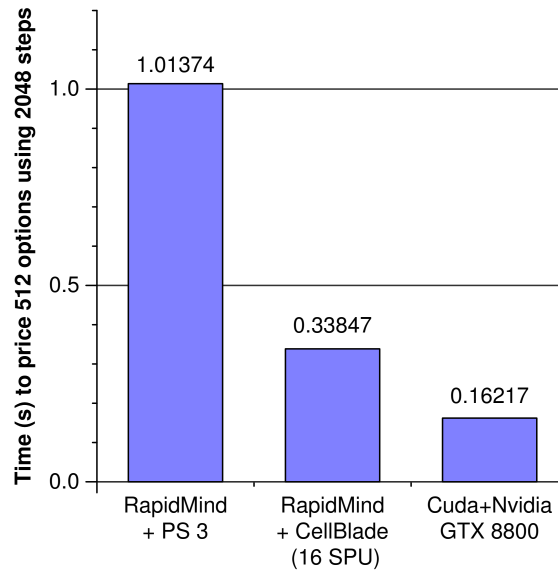


Figure 3.5: Binomial option pricing.

Binomial Option Pricing. I tested a parallel implementation of the binomial option pricing algorithm and found that parallelizing the pricing of a single option with the binomial option pricing algorithm using the task decomposition shown in Fig. 3.1 for a practical number of steps (e.g., 2048) did not result in a good speedup. The results in Fig. 3.5 instead price many options in parallel, each one on a separate processor, like the results for the Black-Scholes option pricing experiments above; also with an embarrassingly parallel communication pattern. Like the Mersenne-Twister implementation for Black-Scholes option pricing, this benchmark requires local arrays stored on each core, which were not supported by RapidMind on the GPU. In addition, a hand implementation on the Cell SDK was not available at the time of this study. Therefore, Fig. 3.5 only includes RapidMind implementations on the Cell and a CUDA implementation on the GeForce 8800. Between sixteen SPE cores on a CellBlade server and six SPE cores on a PS3, the RapidMind implementation scales linearly with the number of processing units. Compared to the CUDA Nvidia GPU implementation, RapidMind with two Cells on a Cellblade was about half the speed.

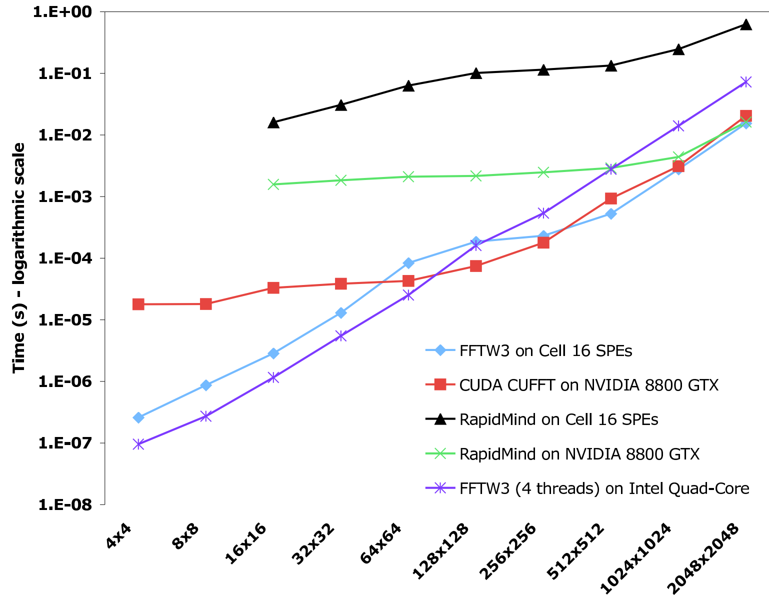


Figure 3.6: Performance comparison of single-precision 2-D FFT.

In this application, which has no inter-process communication, the GPU demonstrates more raw processing power than the CellBlade.

3.2.2 Fast-Fourier Transform (FFT)

Figure 3.6 reports the run time of the two-dimensional FFT on various architectures. Cell's low-level implementation comes from the one of the most popular FFT libraries, FFTW, whose core computation optimally combines several straight lines of code fragments called codelets written in platform natives [50]. CUFFT, the CUDA FFT library, provides a similar interface to FFTW on the GeForce 8800 [34]. RapidMind implementations on the Cell and the GeForce 8800 are also included. The run time measurements are performed by interfacing the above libraries with the "benchFFT" environment, an extensible FFT benchmark program [49].

The fastest FFT-performing architecture varies depending on input sizes. For inputs smaller than 64×64 2-D arrays, the FFTW library on Cell runs faster than the CUFFT on GeForce 8800. For the inputs with sizes in the interval [64×64,

256×256], CUFFT/GeForce 8800 outperforms FFTW/Cell. Finally, for input size beyond 256×256, the FFTW/Cell is slightly faster than CUFFT/GeForce 8800.

The results of the FFT performance can be analyzed in the context of the computation and communication aspect of the FFT algorithm. The FFT algorithm requires not only intensive floating-point computations but also frequent data communications between processing elements. On most 2-D FFT instances Cell's flexible on-chip communication fabric overcomes its floating-point computation disadvantage with respect to GeForce 8800, which does not provide direct links for inter-multiprocessor communications. Therefore, for large inputs the Cell edges GeForce 8800, even though it has less floating-point computation capability.

The RapidMind implementations have lower performance compared to their SDK counterparts on both platforms. Compared to FFTW/Cell BE, this is not surprising because RapidMind's programming model does not support direct communications between concurrent processes, thus the powerful Cell on-chip ring is not utilized. On the other hand, RapidMind's limited communication model is actually based on GPU hardware. Therefore on large data inputs RapidMind's performance is comparable to CUFFT/GeForce 8800.

This set of benchmarks also includes the FFT run time on a general-purpose Intel CPU (Intel Kentsfield quad-core clocked at 2.6GHz). For input data smaller than 128×128, the quad-core CPU has better FFT performance than Cell and GeForce 8800, which incur the overhead of distributing the work to processing cores, including the time investment of "forking" and "joining" parallel processes on the processing units.

3.2.3 Bitonic Sort

The RapidMind implementation of bitonic sort is an iterative loop-based solution, where the innermost loops are replaced with data-parallel RapidMind program calls. The low-level implementations of bitonic sort are CellSort [56] on the Cell QS20 and

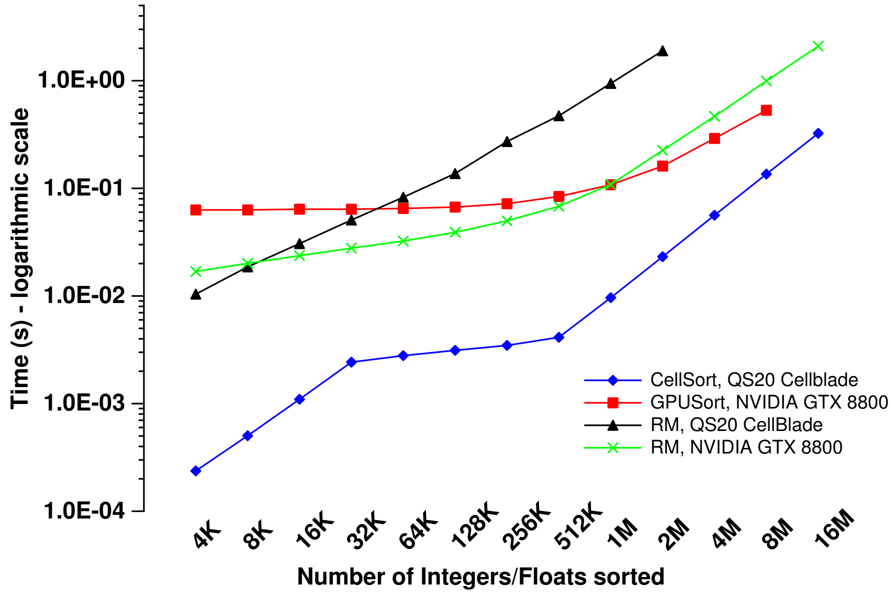


Figure 3.7: Performance comparison of bitonic sort.

GPUSort [61] on the GeForce 8800. The results for the different software implementations and hardware platforms are shown in Figure 3.7. The curves end at different input data sizes because the different implementations do not support the same maximum sizes. The RapidMind and CellSort implementations sorted integers, while the GPUSort implementation sorted floats (GPUs tend to handle floats more efficiently, while Cell handles both equally).

Most striking is that CellSort, the Cell SDK implementation for Cell, has the best performance by a large margin. This reflects Cell's high bandwidth intercommunication network and large local stores. A second observation is that the performance with RapidMind comes much closer to the performance of a hand implementation on the GPU backend than it does on the Cell Backend. In fact, for smaller cases (<256K elements), RapidMind performs better than GPUSort. These results reflect both a better GPU backend in RapidMind (RapidMind, developed from Sh [89] has supported GPUs longer), and also the fact that an SPMD programming model is probably not ideal for Bitonic Sort.

The gap in performance between the best performing implementations on the Cell and GPU platforms narrows as the data size increases. For example, sorting 32K elements on the GeForce 8800 is about 25 times slower, but sorting 8M elements is only 4 times slower. With larger data sets, the amount of data being sorted exceeds the capacity of the on-chip memory, and more off-chip data movement is required, evening out the capabilities of the platforms somewhat since their off-chip bandwidth is more evenly matched than that of their on-chip interconnects. And although GPUSort is not optimized for the smaller data sets as some of the other implementations are, it is the most scalable GPU implementation.

Looking closer at the memory performance, the curve of CellSort's performance has three distinct sections: 4K-32K, 32K-512K, and >512K. In the first section, the data is small enough to fit into the local store of a single SPE, and sorting is handled locally. In the second section, the data is too large for a single SPE's local store, but small enough to fit into the combined local stores, so off-chip communication is not necessary during the sorting (except in the case of 512K where data must be transferred between the two Cell chips). In the last section, the problem size is too large to fit onto the chips and so data must be swapped in and out of main memory throughout the sort. In these larger problem sizes, the GeForce 8800 begins to catch up because it has higher off-chip memory bandwidth than the Cell.

3.2.4 Smith-Waterman Sequence Alignment

Several difficulties were encountered with the Smith-Waterman benchmark. A RapidMind implementation based on the method mentioned in Sec. 3.2.4 of computing diagonals in parallel turned out to be much slower than an optimized sequential code. One performance issue is that although the same RapidMind program is called for each diagonal, the length of each diagonal changes, and from the performance logs, RapidMind requires a new online-compile for every other diagonal size. A second embarrassingly parallel implementation, which computes an alignment separately for

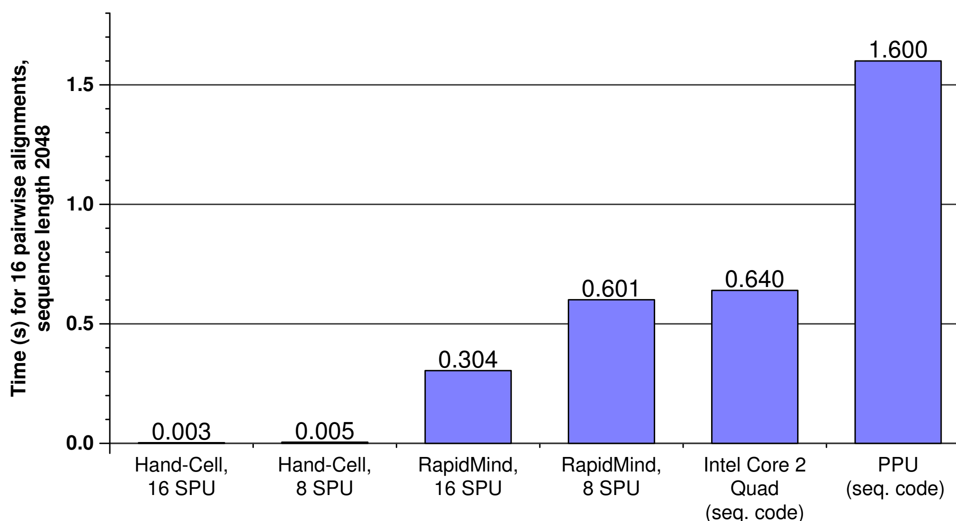


Figure 3.8: Performance comparison of Smith-Waterman sequence alignment.

each processing unit is used instead for the experiments presented in Fig. 3.8. This implementation cannot handle very large sequences, but could be useful in implementing sequence database search tools such as FASTA [102]. Like the binomial option pricing RapidMind program, the implementation of Smith-Waterman was not supported on GPUs due to the use of local arrays. At the time of the study, optimized GPU implementations were not available, though some have been recently implemented [86]. While these challenges prevented a complete comparison of Smith-Waterman in this study, they inspired the Huckleberry project, which is covered in Chapter 5.

Figure 3.8 shows the time to complete 16 alignments on the Cell BE, with both a hand-optimized Cell SDK implementation and a RapidMind implementation. The hand-optimized implementation performs roughly 1000x faster than the RapidMind implementation. Note that the RapidMind implementation is not vectorized, and could potentially have a speedup equal approximately to 4 with vectorization. With RapidMind there was actually no speedup for just 16 alignments because the startup overhead is great, so the numbers reported are the average based on 1024 parallel alignments. However, even though the performance of RapidMind implementation on Cell does not approach that of the hand-coded implementation, it outperforms a

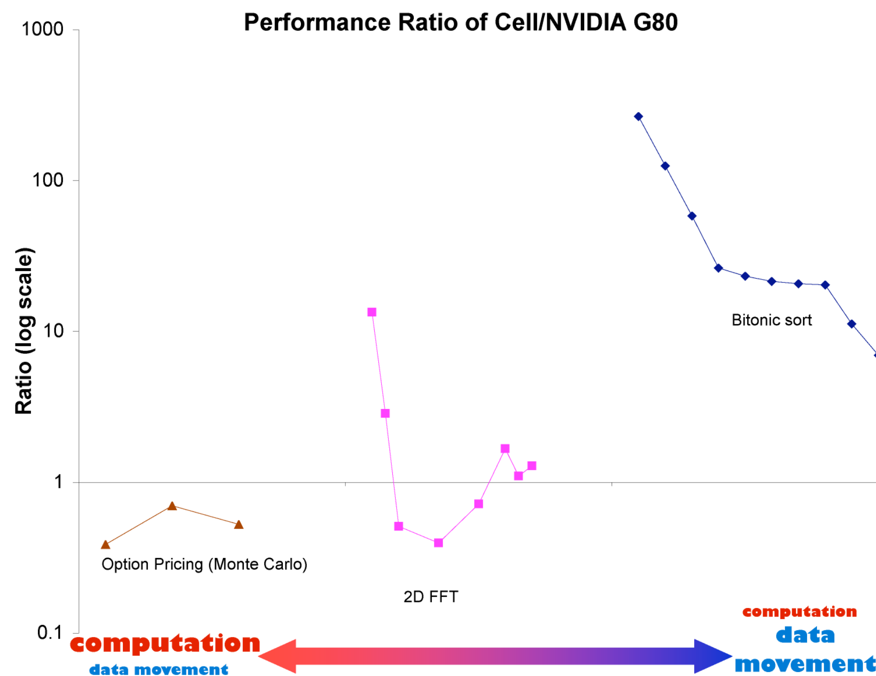


Figure 3.9: Performance comparison of Cell vs. NVIDIA 8800 GTX. The x-axis shows the spectrum of computation and communication patterns. Data points toward the left side are more computation bound; data points on the right are more communication bound.

sequential implementation on the Intel Core 2 and the Cell PPU processor. Because the application is embarrassingly parallel, the ability to harness more cores will set apart the parallel implementations as the architectures scale up.

3.2.5 Discussion

This section summarizes the experiment results and qualitatively evaluates the impact of the computation and communication aspects of the two hardware platforms and the benchmark programs.

Figure 3.9 highlights the relative performance of the Cell blade (16 SPEs) and the

GeForce 8800 using platform-specific SDKs across the spectrum of different computation and communication patterns. If the ratio is above 1, the Cell blade is faster. The general trend is that the GeForce 8800 has an edge on computation bound workloads; in contrast, Cell performs better on communication intensive applications. For example, the GeForce 8800 runs faster if the application is computation bound, like Monte Carlo methods. On the other hand, Cell is faster than GeForce 8800 on applications like FFT on large data inputs and bitonic sort, both of which require intensive data communications.

In particular, data movement is the limiting factor in the performance of bitonic sort. Thus the memory capacity and communication network of a given multi-core architecture play an important role for this application. The Cell's EIB gives the Cell a great advantage since cores can transfer data between each other very quickly. However as the problem size scales up and data must be swapped in and out of the off-chip memory, the bandwidth to main memory becomes a limiting factor.

The relative performance of RapidMind programs and their platform-specific SDK counterparts are reported in Figure 3.10. If a RapidMind program runs faster, its relative ratio is larger than one. Except for the bitonic sort OpenGL implementations on GeForce 8800, RapidMind's performance cannot compete with the SDK-based implementations yet. At best RapidMind program are slightly faster (only in one problem instance), but at worst they are orders of magnitude slower than the corresponding SDK versions.

RapidMind's SPMD programming model accounts for part of this performance gap. The SPMD model fits applications like Monte Carlo methods well. However, it provides no inter-process communications. These restrictions profoundly limit the performance of applications like FFT and sorting where data movements between parallel processes are as important as computation itself. This limitation is especially visible when these applications are implemented on the Cell using RapidMind, which does not exercise the Cell's high-bandwidth inter-SPE communication links.

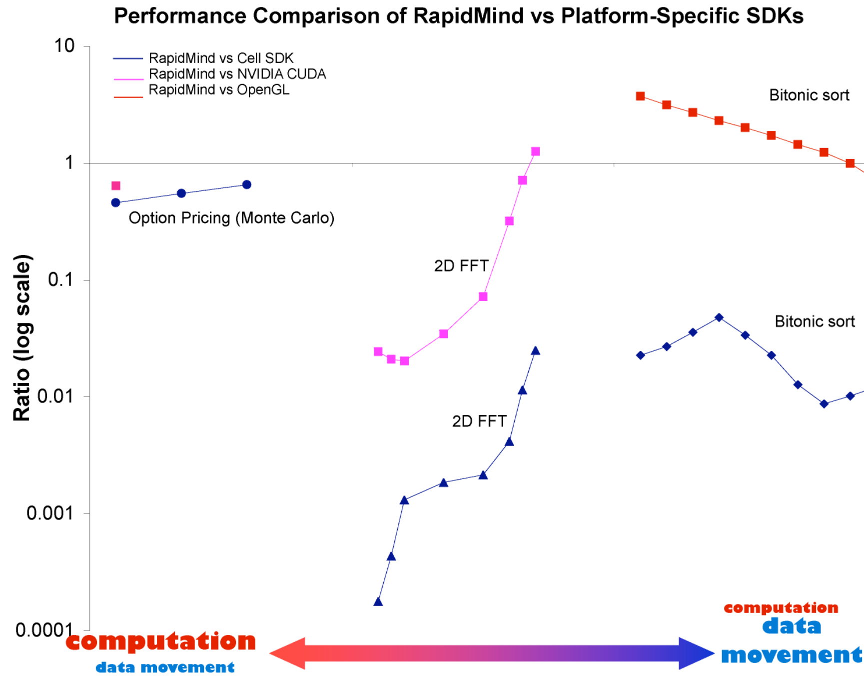


Figure 3.10: Performance comparison of RapidMind vs. platform-specific SDKs.

3.3 Related Works

Comparing multi-core platforms has become a hot research topic recently [21; 62; 106; 123]. Other studies confirm the observation that there is no one "best" architecture for all problems, but that each architecture has its own unique strengths.

Accelerating benchmark applications on multi-core architectures is also an interesting and active area of research. Some of the missing hand-optimized implementations of the benchmark applications in this study have since been developed on various platforms. CUDASW implements the Smith-Waterman algorithm with CUDA for CUDA-enabled GPUs [83]. Farrar introduces a striped SIMD multithreaded implementation of Smith-Waterman that supports both general-purpose multi-core architectures and the Cell BE architecture [45; 46]. SWPS3 extends Farrar's work with performance improvements [119]. Ganesan *et al.* accelerate the binomial option pric-

ing algorithm with a GPU architecture. Their implementation alters the algorithm to calculate multiple timesteps in parallel since the dependencies between two timesteps can be broken if values from a previous time step are available [52]. Wynnyk and Magdon-Ismail leverage reconfigurable hardware to accelerate single instances of binomial options [126].

3.4 Summary

In this chapter, five benchmarks are compared on two multi-core architectures and over a range of low-level to high-level programming abstractions. From this empirical survey, several observations can be made:

- Programming abstractions may be more successful with some applications than they are with others. For example, the SPMD abstraction is a good match for embarrassingly parallel and computation-rich benchmarks but not as good of a match for communication-rich benchmarks like bitonic sort.

- Programming abstractions may be more successful with some architectures than they are with others. GPU architectures feature a large number of floating point units and a small capacity for inter-core communication. The SPMD abstraction fits GPU architectures well since it focusses on tasks that do not have inter-dependencies. However, in applications where inter-dependencies play a great role, such as binomial option pricing and Smith-Waterman sequence alignment, parallel speedup of a single application instance proved extremely elusive given only data-parallelism. Both of these benchmarks were slower as parallel implementations than as sequential implementations. The best parallelization strategy found in these cases was to keep single problem instances sequential and increase the throughput by processing many instances concurrently with embarrassingly parallel data-parallelism.

- The ability of a programming model to scale up to many cores and very large data sets is more important than the performance of the model over small data sets and fewer cores. Although the RapidMind programs were sometimes much slower than the hand-optimized parallel programs, especially on the Cell architecture, they scale well with the number of cores. Hence, the RapidMind programs will be portable to new generations of the architectures. Furthermore, in some cases, the RapidMind implementation was available when a hand-optimized implementation was not.

The RapidMind programming language uses an SPMD abstraction, and these experiments provide a picture of how well the SPMD abstraction fits with various benchmarks and architectures. In the next chapters, I propose two new techniques that work with two different programming abstractions, the stream abstraction and the recursive parallel abstraction.

In Chapter 4, I present *flexible filters*, a distributed load-balancing technique for stream programs. While the SPMD abstraction extracts data parallelism from applications, the stream abstraction extracts both pipeline and task parallelism from applications. Likewise, while the SPMD abstraction allows for no communication between tasks, the stream abstraction allows complex communication between tasks. Through their contrasts, these approaches complement each other.

In Chapter 5, I propose Huckleberry, a recursive parallel abstraction based on data partitioning. As noted earlier, the difficulties encountered while creating an SPMD implementation of the Smith-Waterman benchmark inspire the Huckleberry project. In particular, an SPMD implementation of the Smith-Waterman algorithm computes a SPMD calculation over the diagonals of the score matrix. The changing diagonal sizes in the matrix cause performance degradation in this study. Moreover, the SPMD abstraction does not capture locality between diagonals in the matrix, and thus requires more off-chip data swapping than would be necessary otherwise. For example, if the a diagonal of a large data set does not fit entirely into on-chip

memory, the SPMD solution would compute that diagonal entirely before moving on to the next, never reusing the computed solutions before they are swapped off chip. Huckleberry addresses these issues with a recursive data-partitioning approach. The Huckleberry implementation of Smith-Waterman recursively divides the score matrix into quadrants so that concurrent tasks share a uniform size, and leverage locality within their local neighborhood to minimize off-chip data swapping.

Chapter 4

Flexible Filters: Load Balancing through Backpressure in Streams

Stream processing is a promising model for programming multi-core platforms that is applicable to a wide range of applications including high-performance embedded applications, signal processing, image compression, and continuous database queries [18; 20; 63; 76; 81; 90; 113; 121]. The stream processing paradigm decomposes an application into a sequence of data items (*tokens*) and a collection of tasks (referred to as *filters* or *kernels*) that operate upon the stream of tokens as they "flow" through. Filters communicate with each other explicitly by exchanging tokens through point-to-point communication channels. This model exposes the inherent locality and concurrency of the application and enables the realization of efficient implementations based on mapping the filters onto parallel processor architectures. Given a stream program and a target architecture, the filters of the stream program are mapped to the cores of the architecture, and the communication channels to the communication substructure of that architecture, including mapping input and output buffers to the (possibly distributed) memory and the communication itself to underlying communication protocols such as message passing. In general, it is a challenge to achieve an optimal mapping that maximizes the program performance given data dependencies

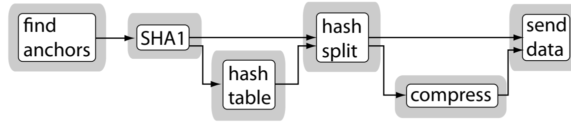


Figure 4.1: Stream graph of the Dedup benchmark application.

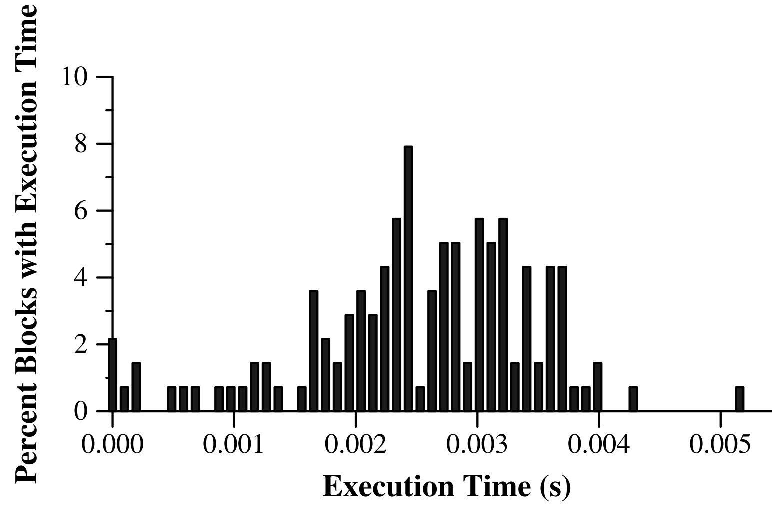


Figure 4.2: Histogram of execution times for Dedup's Compress filter.

among the filters and the available hardware resources (processing cores, memories, and interconnect). Moreover, the execution time of a software task is often variable, making mapping more difficult since the relative cost of filters with respect to each other is not constant. Consider the Dedup benchmark, a parallel compression application [10], that can be implemented as a stream program with six main filters as illustrated in Fig. 4.1, which shows the corresponding *stream graph*. A data-dependent execution time characterizes the compress filter, illustrated by a histogram in Fig. 4.2. The execution time to compress a block varies by up to 0.005 seconds in the sample of blocks. The histogram shows the distribution of execution times of different blocks in the sample.

This chapter presents *flexible filters* as a technique to balance stream programs on distributed-memory multi-core platforms that combines static mapping of the stream

program filters with dynamic load balancing of their execution [27; 28]. The goal is to increase the overall processing throughput of the stream program by reducing the impact of *bottleneck filters* running on particular cores. A filter can cause a bottleneck because either (a) its algorithmic characteristics make it disproportionately expensive to run on a given core with respect to the other filters running on neighboring cores or (b) at run time it may go through phases where it has to process a larger number of tokens per unit of time. When a filter becomes a bottleneck, its neighboring upstream or downstream filters, or both, may start suffering a loss of throughput and, ultimately, this affects the data processing throughput of the overall stream. If a slow computation creates a bottleneck by delaying the production of new tokens, the downstream filters may become idle due to the lack of inputs. Alternatively, a filter can also be a bottleneck if it cannot keep up with the data production of upstream filters. If this is the case, the input buffers of its processing core start filling up. This ultimately leads to the emission of *backpressure* signals between the the cores running the bottleneck filter and its upstream neighbors, forcing the upstream filters to become idle to avoid a loss of data from buffer overflows.

The basic idea of flexible filters is precisely to take advantage of the available cycles on these neighboring cores and use them to dynamically accelerate the execution of bottleneck filters. In other words potential bottleneck filters can be balanced by making their mapping to the underlying architecture "flexible" so that for certain periods of time they can run simultaneously on more than one processing core to execute different substreams of the data stream.

Figure 4.3 illustrates my proposed design flow to guide the application of flexible filters. Profiling or modeling of the stream application on the target architecture may identify bottleneck filters. Based on this profiling, the graph is modified to include redundant copies of the flexible filter as well as auxiliary code which leverages the backpressure mechanism to dynamically activate the execution of the additional copies of the bottleneck filters when necessary, while preserving the correct ordering

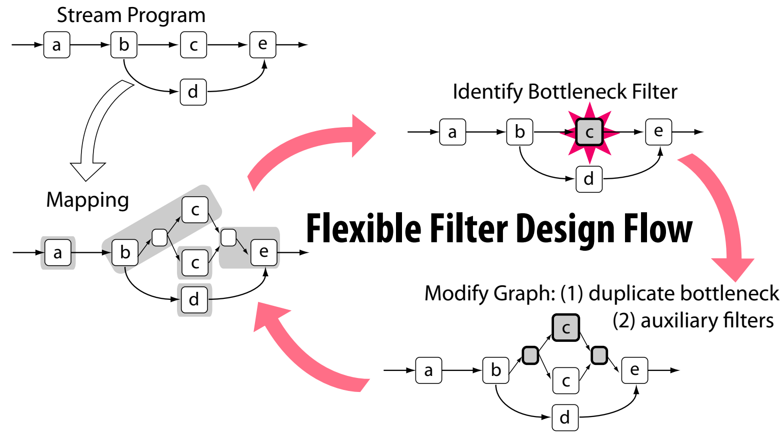


Figure 4.3: Flexible filter design flow.

of the tokens in the data stream. Finally, a mapping assigns the set of filters to the cores of the architecture. The design flows in a cyclical process since the first program profile depends on a mapping of the application, and the modification to the original stream graph may give rise to new bottlenecks.

In summary, the main contributions of this chapter are the implementation and evaluation of flexible filters on a suite of five benchmarks, including an example with a data dependent processing flow. Later, Chapter 6 proposes task graphs as a general representation for all data-driven multi-core programs. Stream programs have a one-to-one correspondence with task graphs (e.g., the correspondence of filters to tasks), and therefore load balancing techniques for stream programs, including flexible filters, may also be applicable to more general task graphs. Here, however, I focus only on stream programs. The rest of the chapter is organized as follows: Section 4.1 describes the mechanics of adding flexibility to a stream graph, and Section 4.2 describes the implementation of *flex_split* and *flex_merge*. Section 4.3 presents experimental results obtained with the application of flexible filters to several real world benchmarks. My experiments show that flexible filters achieve speedup over a wide variety of application domains and in cases where the execution time of filters varies during runtime.

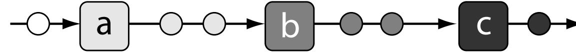


Figure 4.4: Example stream program structure.

4.1 Flexible Filters

The presence of a bottleneck filter may limit performance by preventing utilization of the full capabilities of an architecture. This section first defines throughput as a performance metric and then presents a small example to illustrate how performance can be lost because of a bottleneck filter and how the incorporation of flexibility into a program corrects for this loss.

In order to implement a stream program on a multi-core architecture each of its filters must be mapped to at least one core. A core may host several filters and rely on a scheduler to time-multiplex the core's resources among the filters. The performance of a given implementation can be measured by its *maximum sustainable throughput (MST)*, i.e., the maximum rate at which data tokens can be processed under the assumption that the environment is always willing to produce new tokens and never requires the system to stall through a backpressure signal. In an *ideal* multi-core architecture, (1) the overhead of inter-core communication and intra-core context switching is negligible and (2) each core has unlimited local memory. An ideal mapping of filters would result in a runtime execution where no core ever stalls and the MST scales linearly with the number of cores.

Consider the simple example of a generic stream program whose structure is shown in Fig. 4.4: it consists of three filters a, b, and c with data tokens traveling between them on communication channels (a, b) and (b, c). If the filters have *execution times*[1]

[1]The execution time of a filter is the time necessary to execute it on a given core as a stand-alone task. In a heterogeneous multi-core architecture the same filter would have different execution times when executed on different programmable cores. However, this example considers only homogeneous architectures.

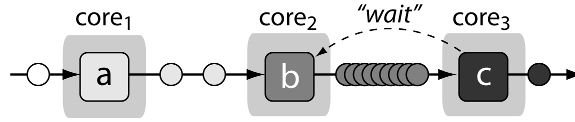


Figure 4.5: Pipeline mapping.

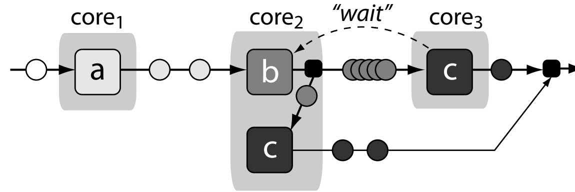


Figure 4.6: Flexible filter mapping.

$L_a = 2$, $L_b = 2$, and $L_c = 3$, respectively, then the ideal MST (i.e., assuming that no core is idle, and performance scales linearly) is $\frac{\#\ cores}{L_a+L_b+L_c} = \frac{3}{7} = 0.429$.

Fig. 4.5 illustrates a simple pipeline mapping: each filter is mapped to a separate core. Using the same execution times as above, this mapping delivers an MST equal to $\frac{1}{3} = 0.333$, lower than the ideal 0.429 because filter c can process a new data token only every three time steps, limiting the performance of the program. Once the buffers between $core_2$ and $core_3$ (where b and c are located, respectively) fill up, $core_3$ requests $core_2$ to stall occasionally through the emission of a backpressure signal (and backpressure continues to propagate upstream).

However, suppose that $core_2$ can also execute filter c. Then, instead of stalling, $core_2$ can "work ahead" on the data tokens in its buffers. Now the rate at which data tokens are processed by filter c increases, and the load on $core_3$ decreases, and so the system runs faster. Thus, load balancing based on flexible filters duplicates bottleneck filters and maps the duplicate copies together with upstream filters. For instance, as shown in Fig. 4.6, adding flexibility to the stream program from Fig. 4.5 makes it possible to alleviate the bottleneck caused by filter c on $core_3$. The new mapping duplicates filter c on $core_2$ so that $core_2$ can share the load of filter c. Performance

gained will be discussed with examples later in this section and tested experimentally in Section 4.3.

Besides increasing the code footprint in $core_2$ with respect to the pipeline mapping, flexible filters also add some complexity to the program because now the data stream is split and merged around $core_3$. The two *auxiliary filters*, *flex_split* and *flex_merge* accomplish the split and merge steps. These filters, which are represented as small black boxes in Fig. 4.6, can be added to the stream program without changing any of the original filters. Flexible filters provide a notion of semantic preservation whereby the ordering of tokens is preserved in the final output of the program, and lossless channels guarantee that no token is dropped so that the resulting output data stream is unaltered when some filters are made flexible in the execution. To be eligible for flexibility, a filter must be stateless; i.e., given an input token x, a stateless filter will produce the same output token regardless of what tokens came before x. The filters surrounding a flexible filter may be stateful, and in some cases it is possible to break a stateful bottleneck filter up so that the most computationally expensive part is stateless.

The flexible filter solution combines a static mapping of stream tasks with dynamic runtime flow so that the flow may be redirected at runtime around bottlenecks as allowed by flexibility in the static mapping. Note that in this example, a static load-balancing split and join could achieve the same speedup as flexible filters if each core always had the same execution time. Previous works have shown that static splits and joins of the data flow can be used to balance the workload of cores and improve performance [58]. The decision of where to insert splits and joins and to what extent a job should be split is left to the compiler. Hence, it is a static optimization choice. However, the dynamic load balancing of flexible filters has advantages over static load balancing in cases where the application or environment do not allow for constant execution times; e.g., when bottlenecks are data dependent, and when the

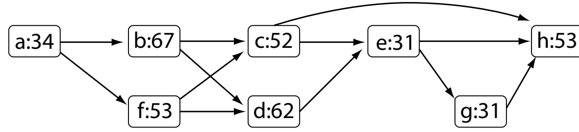


Figure 4.7: Example stream graph.

sharing of resources in the system changes dynamically due to contention with other applications.

Flexible filters differ from previous load-balancing approaches because backpressure alone drives load balancing, and data dependencies across the filters in the stream program guide the task reassignment to idle cores rather than random reassignment. The approach does not require centralized control, and sends no extra messages among cores beyond backpressure messages, which are already present to prevent the communication buffers from overflowing. Since the runtime load drives load balancing, flexible filters can be used not only to optimize the implementation of programs whose filters have constantly unbalanced computational loads but also to adjust temporary imbalances due to spikes of activity, e.g., detecting "bargains" in real-time streaming stock tick data [55].

4.1.1 Pipeline-Aware Mapping

The throughput of a stream program reflects the mapping of that program to the underlying architecture. Data flow dependencies distinguish stream programs from general-purpose parallel programs with respect to mapping because of the buffering requirements between neighboring stream filters.

Consider the stream graph in Fig. 4.7, which is a pseudo-randomly generated task graph by the Task Graph For Free (TGFF) tool [41]. The figure annotates each filter in the example with the execution time of that filter's computation. If we assume that the number of available cores, N, equals six then some of the filters must share a core with each other. If the costs of buffering and communication were omitted,

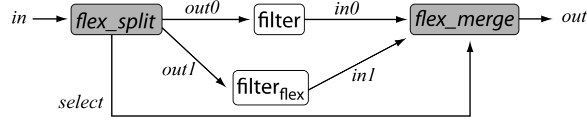


Figure 4.8: Relationship of *flex_split* and *flex_merge*.

the optimal mapping would co-map filters a with e and c with g (with e and g interchangeable since they have the same execution time). This mapping minimizes the maximum workload assigned to any core. A pipeline-aware mapping algorithm might instead co-map filter a with f and e with g. The schedule of a pipeline-aware mapping does not minimize the maximum workload of an individual core, but it favors co-mapping filters that neighbor each other in the graph. Co-mapping neighboring filters allows the output buffers of the upstream filter and the input buffers of the downstream filter to be implemented as the same buffer in the local memory of a core, while co-mapping filters that are not neighbors requires separate buffers.

4.2 Implementation of Flex_Split and Flex_Merge

The programmer identifies potential bottleneck filters through profiling or other program analysis. Then the original stream program is transformed into a flexible stream program by duplicating these filters and by adding pairs of *flex_split* and *flex_merge* auxiliary filters around the flexible duplicates. Fig. 4.8 shows the connections among the filters newly added to the graph. *Flex_split* and *flex_merge* can be provided by an application-independent library because they do not depend on application-specific details. Furthermore, *flex_split* and *flex_merge* do not require modification of the original stream filters. The addition of these filters to the stream graph is the same regardless of the eventual mapping of filters to cores. However, the mapping determines the direction of flexibility, i.e., whether flexibility is achieved by pushing extra load upstream or downstream.

Algorithm 1 flex_split

[Input: stream in; Output: streams $out0, out1, select$]

```
pop data block b from in
n0 ← avail(out0)
n1 ← |b| − n0
for i = 0 to n0 − 1 do
   push 0 to select
end for
for i = n0 to |b| − 1 do
   push 1 to select
end for
push n0 tokens from b to out0
push n1 tokens from b to out1
```

Implementations of *flex_split* and *flex_merge* work with *data blocks*, where a data block is a substream of data tokens. Each data block may consist of many data tokens, and the blocks, like tokens, form a stream and follow an ordering that depends on their place in the bigger stream. One difference between data tokens and data blocks with respect to scheduling the flow of data is that it is possible to break a data block up into several pieces and process them in parallel. A data block is the input unit for *flex_split* and the output unit for *flex_merge*. The divisibility of data blocks is one factor that enables load balancing with flexible filters. But data blocks can only contain a finite number of data tokens and cannot be divided into arbitrarily-sized fractions. Lower granularity (fewer tokens per block) can limit the benefits of flexibility in the data stream because it puts more constraints on the possible data flow.

Flex_split (pseudocode shown in Algorithm 1) dynamically reuses the backpressure information on the current capacity of the downlink input buffers to manage load balancing by dividing the data stream between $out0$ and $out1$. Specifically, it checks

Algorithm 2 flex_merge

[Input: streams $in0, in1, select$; Output: stream out]

```
pop i from select
if i is 0 then
  pop token t from in0
else
  pop token t from in1
end if
push t to out;
```

how much space is available on the buffering queue for the primary copy of the flexible filter, f, and divides the data stream by sending as much data to f's primary copy as it can (stream $out0$) and then sending any leftover data to the flexible copy (stream $out1$). *Flex_split* also produces a *select* bitstream that contains information on how to reconstruct the correct ordering of the stream. *Flex_merge* (pseudocode in Algorithm 2) takes the input streams $in0$ and $in1$ from both of f's copies along with the *select* bitstream, which comes directly from *flex_split*. The *select* bitstream indicates which of f's copies has the next data token, thus allowing *flex_merge* to reassemble the stream into its original order.

Backpressure plays a key role in the implementation of flexible filters. Before a core can send data downstream, it must ensure the availability of adequate buffering space for the data in the receiving core. A typical handshake protocol guarantees that buffers do not overflow and proceeds through a sequence of phases: it starts with the sending core placing a request to send data; then, the receiving core sends back an acknowledgement with information on how much data it can receive (backpressure); and finally the sending core sends the data. In practice, the various phases can be overlapped to further improve performance by adding sufficient memory space.

If a flexible filter is inherently slower than the all of the other filters, then the imbalance will cause the input buffering queue of its primary copy to be full often,

and *flex_split* will redirect the data flow to f's secondary copy at regular intervals. Instead, if f experiences only occasional spikes of activity that cause it to slow down – or if f's upstream neighbor occasionally creates extra data tokens on its output – the flow of data will usually behave as if there is no redundant flexible filter present, and *flex_split* will intervene sporadically when a spike arises.

Finally, notice that instead of having a distinct bit for every token, a compressed format may reduce the *select* bitstream to counts of how many of the next tokens go to $out0$ and then how many go to $out1$. In practice, if the data tokens are vectors or other large data structures, using a distinct *select* bit for each token does not take up a significant portion of memory.

4.2.1 Multi-Channel Flexible Filters

The preceding discussion of *flex_split* and *flex_merge* assumes that the flexible filter has exactly one input and one output channel. Filters with multiple input and output channels may also be flexible, but *flex_split* and *flex_merge* are inserted differently into the graph, and in the case of a filter with more than one input channel, *flex_split* requires modification.

For a filter with several output channels but only one input channel, no modification to *flex_split* or *flex_merge* is necessary: the flexible stream graph simply inserts a separate *flex_merge* for every output channel, and copies the *select* bits of *flex_split* to each copy of *flex_merge*, as illustrated in Fig. 4.9. Because each copy of the flexible filter produces data tokens to its output channels in the same order, the same *select* bitstream is correct for every *flex_merge*.

Adding flexibility around several input channels poses a greater challenge. Duplicating *flex_split* in the same way that *flex_merge* is duplicated for the multiple-output case does not result in a correct implementation because *flex_split* splits the data stream and builds the *select* bitstream based on how much queue space is available downstream. If multiple copies of *flex_split* check for queue space separately at slightly

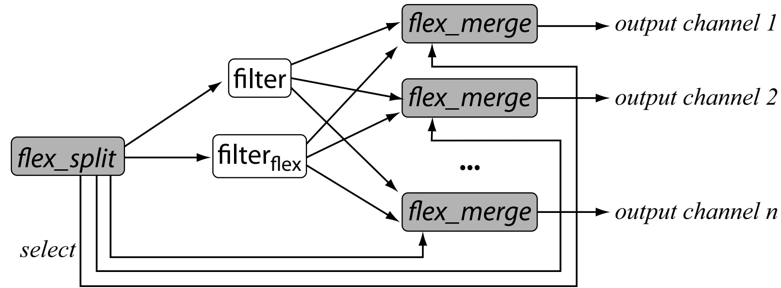


Figure 4.9: Block diagram of a flexible filter with n output channels.

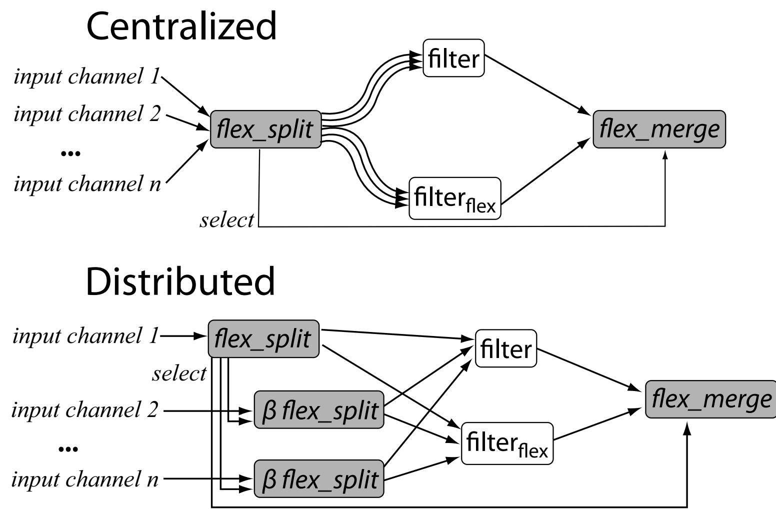


Figure 4.10: Two alternatives of a flexible filter with n input channels.

different times they may get different answers, and thus the data tokens in the input streams would be mismatched. Fig. 4.10 illustrates two possible solutions. One possible solution is to create one centralized *flex_split* that monitors all of the input queues for the copies of the flexible filter, and then splits the data stream in a way that is consistent across all input streams. The downside of this approach is that it may create a bottleneck in processing. Another approach is to introduce a second version of the *flex_split* implementation, denoted β*flex_split*. The original *flex_split* is used for the first channel, and then instead of building new *select* bitstreams, β*flex_split* filters reuse the original *flex_split*'s *select* stream and wait for sufficient space on their out-

Time Steps		t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}	t_{11}	t_{12}
core$_0$	Step	$a_{0,0}$	$a_{0,1}$	$a_{1,0}$	$a_{1,1}$	$a_{2,0}$	$a_{2,1}$	$a_{3,0}$	$a_{3,1}$	$a_{4,0}$	$a_{4,1}$	$a_{5,0}$	$a_{5,1}$	$a_{6,0}$
	Block(s)	0	0	1	1	2	2	3	3	4	4	5	5	6
core$_1$	Step			$b_{0,0}$	$b_{0,1}$	$b_{1,0}$	$b_{1,1}$	$b_{2,0}$	$b_{2,1}$	$b_{3,0}$	$b_{3,1}$	$b_{4,0}$	$b_{4,1}$	$b_{5,0}$
	Block(s)			0	0	1	1	2	2	3	3	4	4	4,5
core$_2$	Step					$c_{0,0}$	$c_{0,1}$	$c_{0,2}$	$c_{1,0}$	$c_{1,1}$	$c_{1,2}$	$c_{2,0}$	$c_{2,1}$	$c_{2,2}$
	Block(s)					0	0	0,1	1	1,2	1,2	2,3	2,3	2,3

Time Steps		t_{13}	t_{14}	t_{15}	t_{16}	t_{17}	t_{18}	t_{19}	t_{20}	t_{21}	t_{22}	t_{23}	t_{24}	t_{25}
core$_0$	Step	$a_{6,1}$	$a_{7,0}$	$a_{7,1}$	$a_{8,0}$	$a_{8,1}$	$a_{9,0}$	$a_{9,1}$	$a_{10,0}$	$a_{10,1}$	$a_{11,0}$	$a_{11,1}$	-	$a_{12,0}$
	Block(s)	6	7	7	8	8	8,9	9	9,10	9,10	10,11	10,11	10,11	11,12
core$_1$	Step	$b_{5,1}$	$b_{6,0}$	$b_{6,1}$	$b_{7,0}$	$b_{7,1}$	-	$b_{8,0}$	$b_{8,1}$	-	$b_{9,0}$	$b_{9,1}$	-	$b_{10,0}$
	Block(s)	5	5,6	5,6	6,7	6,7	6,7	7,8	7,8	7,8	8,9	8,9	8,9	9,10
core$_2$	Step	$c_{3,0}$	$c_{3,1}$	$c_{3,2}$	$c_{4,0}$	$c_{4,1}$	$c_{4,2}$	$c_{5,0}$	$c_{5,1}$	$c_{5,2}$	$c_{6,0}$	$c_{6,1}$	$c_{6,2}$	$c_{7,0}$
	Block(s)	3,4	3,4	3,4	4,5	4,5	4,5	5,6	5,6	5,6	6,7	6,7	6,7	7,8

Table 4.1: Baseline pipeline mapping timeline.

put queues before proceeding. This approach avoids forcing all of the input channels through a bottleneck, but may result in extra stalling by the new *βflex_split* filters.

4.2.2 Example

I now walk through the execution of a stream program at runtime when flexibility is invoked to balance the load. Table 4.1 shows the timeline for the example shown in Fig. 4.5, using the same example execution times that were used in Sec. 4.1 ($L_a = 2$, $L_b = 2$, and $L_c = 3$). The table shows both the current step being executed on each core, and the contents of the core's local buffering memory.

In Table 4.1, each filter completes processing a block i in timesteps equivalent to that filter's execution time. For example, filter a whose execution time is two, computes a block i in two timesteps, denoted $a_{i,0}$ and $a_{i,1}$, respectively. Since filter c has an execution time of three, it must compute blocks in three timesteps ($c_{i,0}$, $c_{i,1}$, and $c_{i,2}$).

In Table 4.1, even though the filters' latencies are not equal, the buffer capacity allows the faster filters to work ahead initially. However, at time step t_{18}, core$_2$ must stall. At this timestep, core$_2$'s memory contains Blocks 6 and 7, and even though core$_1$ is ready to pass Block 8, core$_3$ holds Blocks 4 and 5 and will not be ready to

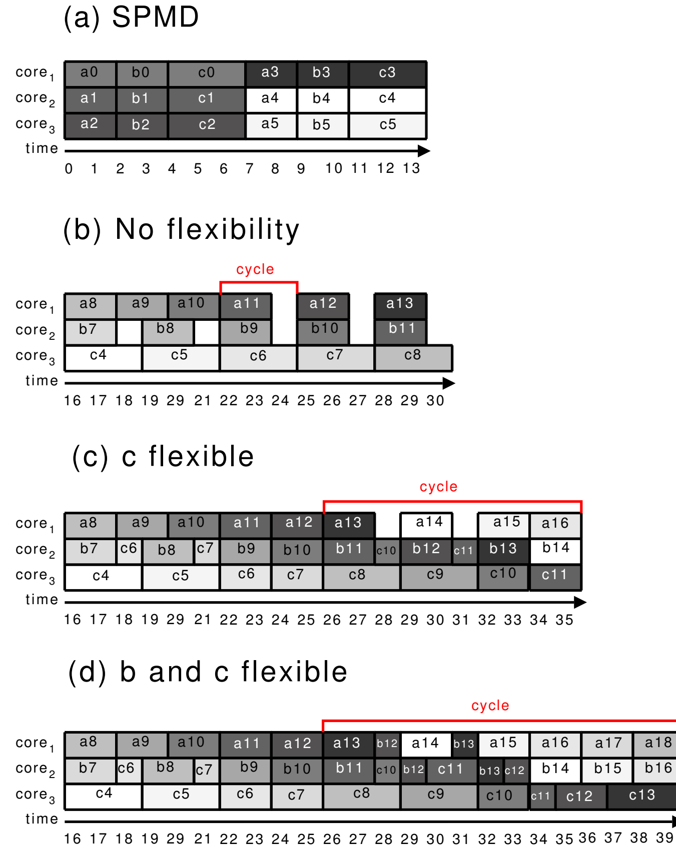


Figure 4.11: Flexible filter timelines.

take the next block from $core_2$ until it is done processing Block 4. Therefore, $core_2$ must wait until $core_3$ is ready to accept the next block before it can make space in its memory for Block 8. The state of the system is the same at time steps t_{22} and t_{25} in terms of the state of each core with respect to the blocks in that core's memory. In fact, the system begins to cycle through a pattern of states, in this case the pattern from t_{22} to t_{24}. During one cycle, this implementation completes one block every three cycles, confirming that the MST is $\frac{1}{3}$, as calculated in Sec. 4.1. Note that if the filter latencies are unbalanced, stalling will occur no matter how much buffering space is available on the cores: additional memory simply extends the time that it takes to initially fill up the buffers.

Fig. 4.11 summarizes timelines for several mappings in a more abbreviated format that does not include the current memory state. For each case other than SPMD,

the timelines start at t_{16} using the same state of t_{16} in Table 4.1 and continue until a cyclic pattern emerges. (For the SPMD mapping, the entire stream program is duplicated separately to each core, with no intercommunication, and data blocks are distributed round-robin to the cores.) Fig. 4.11(b) depicts the same timelines as in Tables 4.1. Fig. 4.11(c) shows the timeline for a flexible-filter mapping where filter c is made flexible and is mapped to $core_2$ and $core_3$ (same as Fig. 4.6). The cyclic pattern for this mapping begins at time step t_{26} and continues until t_{35}. Fig. 4.11(d) shows an alternative flexible-filter mapping where both filters b and c have been made flexible. In particular, filter c is again mapped to $core_2$ and $core_3$, while filter b is mapped to $core_1$ and $core_2$. Here, the pattern goes from time step t_{26} to t_{39}.

When no filters are flexible, the MST (0.333) is degraded by 22% compared to the ideal throughput, 0.429. When only filter c is flexible, the MST is increased to $\frac{4}{10} = 0.400$ (only 7% degradation). When both filter b and c are flexible, the MST reaches its ideal limit of 0.429, thus matching the MST of the SPMD mapping, but without requiring a complete copy of the stream program on every core. From another perspective, the *speedup* gained when going from a non-flexible pipeline mapping to a flexible pipeline mapping is 1.29. A speedup of 2.0 is the maximum possible in any case where only one duplicate copy of a flexible filter is made. Flex_split could be extended to a three, or four-way split to take advantages of other available cores. However, a few caveats on higher degree splits should be kept in mind. (1) In an architecture like the Cell, where application code and data occupy the same memory, there may not be room for additional code and data buffers to accommodate the flexible filter, even if a core is not as busy with computation; e.g., a hash table filter is not compute-intensive, but it is memory-intensive, and the less space available for building the table, the more times the downstream and possibly compute-intensive filters must execute. (2) The pipeline nature of stream applications forces dependencies between the buffers of different filters and adding extra channels within the stream may place higher buffering burdens on those parts of the stream graph.

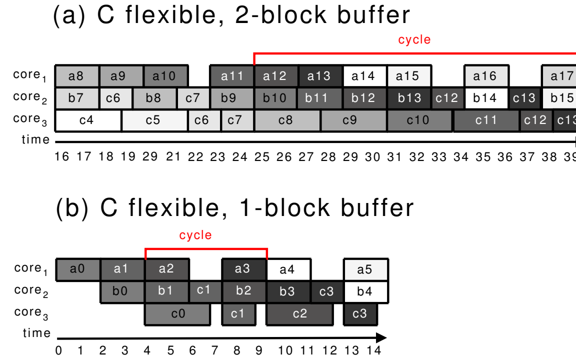


Figure 4.12: Time-line when filter C has a granularity of two tokens per block.

4.2.2.1 Granularity of Firing Constraints and Buffer Size

The previous examples have assumed that it is always possible to break one of c's data blocks up into thirds and b's data blocks up into halves. Suppose, however, that the local data memory of each core only holds a block of two tokens for c. Since data tokens are the minimum amount of data that a filter can fire on, it is now only possible to break one of c's data blocks up into two pieces. Fig. 4.12 repeats the mapping from Fig. 4.11(c) to show the timeline when c has this constraint. There are two cases shown. In Fig. 4.12(a), I assume buffers of size two just like in the previous examples, while in Fig. 4.12(b) I assume that the buffer has capacity for one data block only. At t_7 in Fig. 4.12(b), core$_3$ must wait for Block 1 until core$_2$ is ready to send it. Similarly, core$_1$ must also wait to send Block 2 to core$_2$. When buffers have enough capacity for two blocks, the MST is $\frac{6}{15} = 0.4$, which is the same as the MST when we did not have the additional granularity constraint. However, when the buffers only hold one block, the MST is degraded to $\frac{2}{5.5} = 0.364$. This example shows that the local buffering memory plays a critical role in insulating performance from granularity constraints.

4.2.3 Practical Implementation Concerns

Streaming programming languages typically abstract away the backpressure mechanism that is implemented at the lower level of the inter-core communication stack [54; 121]. Hence, programmers need not worry about the current state of the buffers between stream functions and can focus on the computational aspects of the algorithm and data manipulation through higher-level functions such as *push* and *pop*. At the same time, the underlying message-passing API functions that support the handshake communication protocol and backpressure mechanism between communicating cores, and that are often specific to the target architecture, may also be made available to allow performance optimizations. The implementation of *flex_split* and *flex_merge* relies on such functions. In particular, the *flex_split* implementation given in Algorithm 1 uses the *avail()* function that returns how much buffering space is available in the next core's buffer. If the programmer does not use *avail()* to check the buffering availability of its output channels at runtime then the filter will automatically stall whenever there is not sufficient space for the data to be sent on any of its output channels. Instead, using *avail()* to check the available space on a channel allows the programmer to dynamically send only the right amount of data to that channel and then proceed to the next instruction without stalling the filter. For instance, to avoid stalling when there is not enough space to send the entire block to f's primary copy, *flex_split* sends exactly the amount of data equal to $avail(out0)$ to $out0$. Then, the rest of the data is sent to f's secondary copy without calling *avail()* on this channel but relying instead on the underlying backpressure mechanism to regulate the stream $out1$. In my experience, relying on the implicit backpressure of the channel instead of explicitly checking *avail()* on $out1$ tends to produce better results, possibly because the leftover portion of the output stream can move forward faster to the filter's secondary copy in the presence of a temporary input buffering shortage.

4.3 Experiments

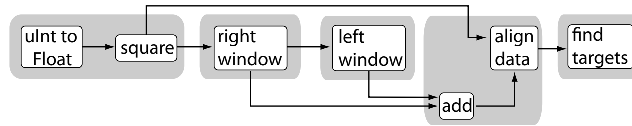
All of the experiments were performed on a Sony PlayStation3 (PS3) which hosts one Cell BE processor [98]. Because PS3 systems enable only six of the Cell's eight SPEs, this is the maximum number of cores used in my experiments. Gedae, a data flow language, is utilized to program the Cell. Gedae provides an abstraction of the communication layer for my implementation by handling low-level details like direct-memory access (DMA) alignment and double buffering while exposing point-to-point communication channels between filters [54]. Gedae's API contains functions to implement the communication channels, including the *avail()* function (mentioned previously in Section 4.2) that gives information on how much space is available in the input and output buffers.

4.3.1 Benchmarks

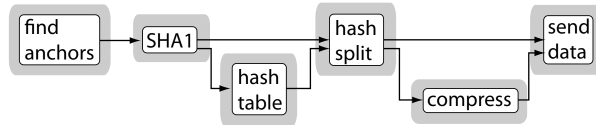
My experiments implement several benchmarks with Gedae and then test them with flexible filters. Fig. 4.13 shows block diagrams of the filters of each benchmark together with how they are mapped to SPE cores of the Cell, and Fig. 4.14 shows profile information for the filters. The remainder of this section briefly describes the benchmarks.

Constant False Alarm Rate Detection (CFAR). CFAR is a signal processing benchmark from the HPEC benchmark suite that identifies targets in a stream of incoming data given a noisy background. It does so by using an adjustable threshold value that changes based on the background noise so that the false alarm rate is constant [67]. CFAR evaluates each token in the stream by comparing it to a sliding window of tokens before and after that token in the stream. The token under evaluation at a given point in time is called the "cell under test". The filters of CFAR are:

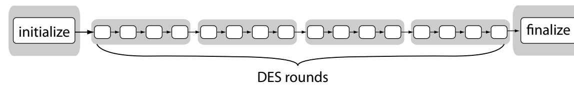
- *uInt to Float*: converts unsigned integer to float;



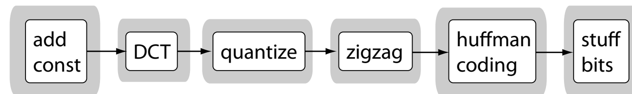
(a) CFAR block diagram.



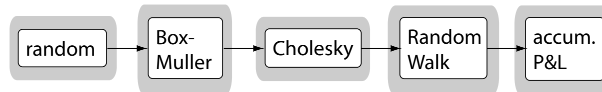
(b) Dedup block diagram.



(c) DES block diagram.



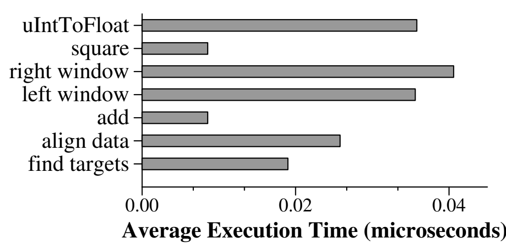
(d) JPEG block diagram.



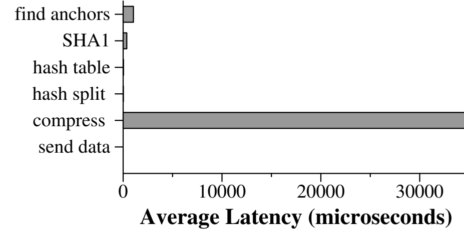
(e) VAR block diagram.

Figure 4.13: Block diagrams of benchmarks used together with their mapping on the IBM Cell multi-core processor (the non-flexible case).

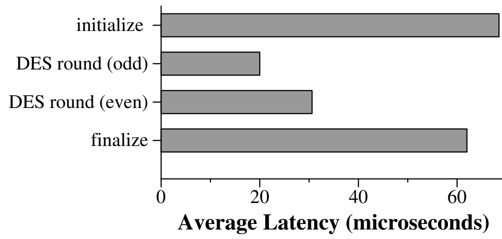
- *square*: mathematical square of float;
- *right window*: maintains a sum of a sliding window of values in the stream;
- *left window*: same as the right window, but positioned after the *cell under test*;
- *add*: sums the values from the right and left windows;
- *align data*: stores and saves incoming data in order to line up the left and right window with the cell-under-test;



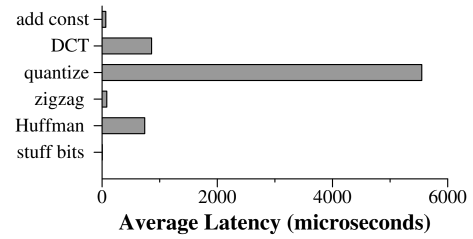
(a) CFAR Profile.



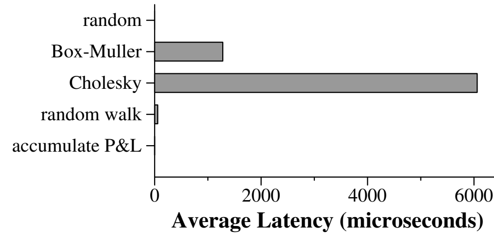
(b) Dedup Profile.



(c) DES Profile.



(d) JPEG Profile.



(e) VAR Profile.

Figure 4.14: Profile of tasks for each benchmark.

- *find targets*: identifies targets in the stream based on how the next cell compares with the right and left windows around it.

Dedup compression. Dedup is a pipeline compression algorithm from the PARSEC benchmark suite. Dedup breaks a file up into blocks according to the Rabin fingerprinting method, and then compresses the file on a block-by-block basis, checking the hash of each block beforehand and only compressing duplicate blocks once [10]. The filters of Dedup are:

- *find anchors*: separates the data stream into blocks according to anchors which are based on Rabin fingerprints;
- *SHA1*: computes SHA1 hash of a data block;
- *hash table*: maintains a hash table of SHA1 hashes encountered so far;
- *hash split*: splits the data stream according to whether the hash of the next block was found in the table;
- *compress*: compresses a data block;
- *send data*: for each block, copies only the hash if the block has already been encountered, otherwise, copies the hash and the compressed data block.

Data Encryption Standard (DES). DES is a block cipher security algorithm [31]. The 16 DES subkeys are generated in advance. The filters of DES are:

- *initialize*: performs an initial permutation on the data;
- *DES round (odd)*: computes one round of des with one of the subkeys;
- *DES round (even)*: the "even" rounds are identical to the "odd" rounds; however, in experimentation, their execution times consistently differed as illustrated in Fig. 4.14(c);
- *finalize*: performs a final permutation on the data.

JPEG Encoder. This benchmark implements the baseline grayscale JPEG encoder [57]. The data stream is broken into 8x8 pixel blocks, where each pixel is a 256 grayscale value. The filters of JPEG are:

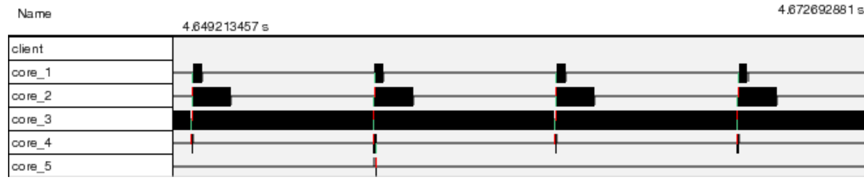
- *add const*: adds a constant to the incoming data pixels (converts [0,255] range to [-128,127]);
- *DCT*: the discrete cosine transform of a pixel block separates the data into a sum of cosine functions;
- *quantize*: lossy compression step that reduces the amount of the least significant data;

- *zigzag*: reorders 2-D block into 1-D sequence according to a zigzag trace through the array;
- *Huffman coding*: replaces data stream with values from Huffman coding tables designed specially to replace the JPEC AC and DC coefficients;
- *stuff bits*: searches for 8-bit aligned 0xff values in the stream and inserts 0x00 after them (0xff bits are reserved for code words in the JPEG standard).

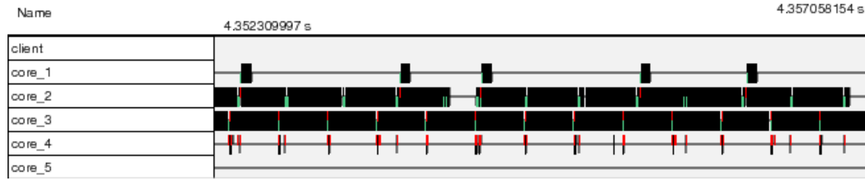
Value-at-Risk (VAR). VAR is a finance benchmark that calculates the value at risk for a portfolio of stocks (or other assets) averaged over a number of random walks over a discrete number of timesteps, assuming that the stocks change at each timestep according to a random correlated set of moves[2]. The filters of VAR are:

- *random*: generates random numbers with uniform distribution;
- *Box-Muller*: transforms a set of uniformly distributed random numbers to a set of normally distributed random numbers with mean $(\mu) = 0$, and variance $(\sigma^2) = 1$;
- *Cholesky*: transforms a set of normally distributed random numbers to a correlated set of random numbers using a lower triangular matrix generated by Cholesky decomposition of a correlation matrix (i.e., starts with correlation matrix C and decomposes it into $C = U * U^t$);
- *Random Walk*: using a stream of random float values as input, takes a random walk for each stock/asset in a portfolio. The random walk lasts a specified number of steps, and then the filter outputs aggregate P&L (profit & loss) values for the portfolio;
- *accumulate P&L*: aggregates and sorts P&L values from previous step to determine VAR.

[2]In light of the recent economic recession, a more popular approach recently has been to estimate VAR as the average of the worst seven days over the previous year, but one hopes that optimism will return.



(a) Trace table of VAR without flexibility.



(b) Trace table of VAR with flexibility.

Figure 4.15: Gedae trace tables of the VAR benchmark. A core's timeline is black when it is busy working on a task. Green and red marks show send and receive events.

4.3.2 Results

Flexible filters provide speedup to an application whenever there is a bottleneck filter as long as the relative cost of communication to computation is not too high. However, when computational tasks are relatively inexpensive compared to communication, the additional overhead from adding flexibility outweighs its performance benefits. The performance gained depends on the relative latencies of filters to each other, and not on the particular operation of each filter. The Dedup, JPEG and VAR benchmarks all include one filter that is significantly more computationally expensive than the others.

Figures 4.15(a) and 4.15(b) show a Gedae trace table for the VAR benchmark before and after the Cholesky filter has been made flexible to show how a bottleneck filter can slow down all of the other cores. In the trace table, the black rectangles show when a core is busy working on a task, the smaller red and green rectangles show the sequence of send and receive events. In Fig. 4.15(a), Core 3, which is assigned the Cholesky filter, is always working, while the other cores spend most of their time

waiting for data to arrive. In Fig. 4.15(b), Core 2 assists Core 3 and reduces its load. Notice that the timespan is actually different in the two timelines. When flexibility is added, I found it often necessary to reduce the overall data block granularity in order to achieve optimal speedup. Therefore compute tasks are broken up more frequently by send and receive events in Fig. 4.15(b).

Table 4.2 reports the speedup gained by making the bottleneck filter flexible in benchmarks where I observed a bottleneck. (The CFAR benchmark is a data-dependent example, whose performance is discussed in Section 4.3.4.) The DES benchmark is an example where there is no bottleneck. Even though there is some variation in the latencies of DES filters, once the eighteen filters are mapped to the six cores of the PS3, the load becomes fairly balanced and adding flexibility to one or more of the filters does not benefit performance. However, many of these benchmarks do include a "bottleneck" filter, and are amenable to the addition of flexibility. In some cases, a better implementation may alleviate the bottleneck. However, in practice, software is often designed using pre-existing libraries. This was the case for the Dedup benchmark in these experiments. The Gedae implementation invoked the same libraries as the original Dedup benchmark[3]. When a filter is made flexible (one redundant copy) as described here, potential speedup is limited to twice the original parallel performance (the overall parallel speedup may be higher from pipelining). For example, flexibility does achieve a full 2.0 speedup for the Dedup benchmark. However, it is often the case that a full 2x speedup is not achieved when flexibility is added even though the bottleneck filter is much more expensive than its neighbors (e.g., the JPEG benchmark). The overhead of communication and changes in data block granularity required by flexible filters are additional costs of flexibility that can impact the performance speedup gained. Section 4.3.3 explores the balance of com-

[3]The hash table library required a minor modification in one of its constants so that the table would be guaranteed to fit within an SPE's local memory.

Benchmark	Input Data	Speedup
CFAR	% targets/workload	
	7.3/16μs	1.45
	7.3/32μs	1.39
	7.3/63μs	1.47
	1.3/16μs	0.82
	1.3/32μs	1.06
	1.3/63μs	1.27
Dedup	Rabin block/max chunk size	
	4096/512	2.00
DES	–	(no speedup)
JPEG	image width x height	
	128x128*	1.31
	256x256*	1.16
	512x512*	1.25
VAR	stocks/walks/timesteps	
	16/1024/1024	0.98
	32/1024/1024	1.34
	64/1024/1024	1.56
	96/1024/1024	1.54
	128/1024/1024	1.55
	160/1024/1024	1.81

*individual benchmark images, each with different content

Table 4.2: Summary of speedup results for benchmarks where one bottleneck filter is made flexible.

munication and computation with respect to the speedup gained by adding flexibility to an application.

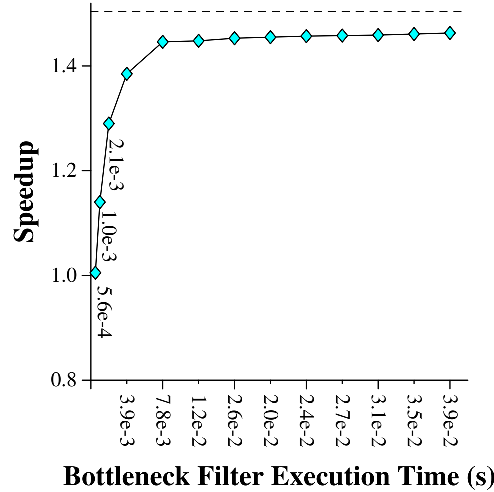


Figure 4.16: Speedup as the relative cost of a bottleneck filter increases with respect to the cost of communication.

4.3.3 Balance of Communication vs. Computation

Adding flexibility to a stream filter typically adds more communication overhead compared to the original pipeline implementation because *flex_split* and *flex_merge* require additional buffers, which reduce the space per buffer available on the cores. This translates to a lower granularity (i.e., fewer tokens per data block) in the data stream. There is also additional data movement between the buffers of the filter and *flex_split* and *flex_merge* even when some are co-located on the same core. The experiments in Fig. 4.16 synthetically vary the latency of the Cholesky and BoxMuller filters in a subset of the VAR benchmark. By increasing the execution time of the filters while the relative ratio between them stays the same, these tests examine how the speedup changes as the relative cost of communication and computation changes. The flexible copy of Cholesky is mapped to the same core as BoxMuller, and the latency of Cholesky is approximately 3 times that of BoxMuller so that the optimal speedup in any case possible is about 50% (shown with a dashed line in Fig. 4.16). The speedup approaches 50% as the computation cost of the filters becomes very large, overshadowing the cost of communication. On the other end

of the spectrum, speedup drops off as the execution time of the "bottleneck filter", Cholesky, is reduced. When the execution time of the Cholesky filter is less than 560 microseconds, no improvement is observed. The point at which speedup tapering-off occurs in this benchmark is a result not only of the execution latency of the bottleneck filter, but also of the data block size (vector of 128 floats) and task granularity (3 data blocks in the flexible case, 50 in the non-flexible case). In other words, other applications will experience a trade-off point in a different location depending on their own data structures and granularity.

4.3.4 Adapting to Data Dependent Flow

One of the strengths of flexible filters is that they can adapt load dynamically at runtime when there are data dependent spikes of activity that may cause temporary bottlenecks in the flow of execution. The CFAR benchmark provides an example to explore data dependent flow volume. As shown in Fig. 4.14(a) all filters of CFAR have a relatively lightweight execution time with respect to the communication overhead, and initially the program did not benefit from the addition of flexibility. In particular, the *find targets* filter in the original implementation does no additional work after a target is detected, and so has relatively constant execution time regardless of the content of the data stream. However, in practice it is possible that once a target is found, additional processing such as target classification and tracking is needed [99]. To capture this fact, in the CFAR experiments reported in Table 4.2 I add an additional synthetic workload to *find targets* every time a target is detected. Since the location of targets is data dependent and may not be uniformly distributed in the stream, the workload of *find targets* may change dynamically, and spikes in the number of targets detected could cause bottlenecks. The percent of targets detected is varied by adjusting the sensitivity threshold for the input data sets provided by the HPEC challenge. Fig. 4.17 plots a histogram of the time it takes to process a data block of 114 cells, where 7% of the cells are targets, and an additional workload of

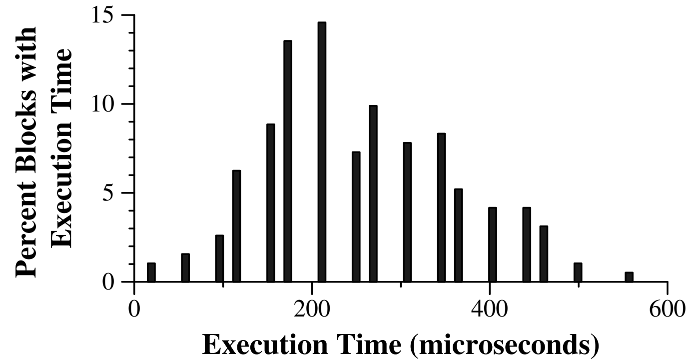


Figure 4.17: Histogram of workload per 114 cells, % targets/workload = 7/32μs.

32μs is added for each target. The speedup gained by applying flexible filters in this case depends both on the percentage of data tokens that require extra processing and on the amount of extra work required. While it would likely be possible to achieve similar results for any one instance of CFAR with a static stream split and enough buffering, the strength of flexible filters is that the same implementation will adapt to changing load in the same stream without modification (i.e., a stream that switches from one distribution of execution times to another).

4.4 Related Works

Flexible filters balance load using a version of work stealing for stream programs. Work stealing is a technique used in a variety of parallel systems to balance load by allowing idle cores to "steal" tasks from busy cores [9; 51; 75]. Most work stealing techniques go through stages of load evaluation, reassignment, and task migration; and their "victim" processors (from whom tasks will be stolen) are selected randomly. In contrast, flexible filters do not steal randomly, but use the knowledge that neighbors of a bottleneck filter will be idle because they depend on this filter to continue processing data tokens. Items are never migrated between buffering queues of different processors; instead, when queues become full new items are redirected elsewhere. With flexible filters, tasks are not "stolen" per-se but rather the data flow is re-routed

when a bottleneck arises. Whichever filters have been mapped with flexibility determine the available routes for data during runtime. Flexible filters are specialized to pipeline data flow because the pipeline stream dependencies narrow down good candidates for redundant-code placement by exposing which tasks will become idle when another becomes a bottleneck.

Load balancing approaches specific to stream programs can be categorized depending on whether the stream models rely on data parallelism or pipeline parallelism (in practice both approaches can be used simultaneously [58]). In data parallel stream systems, there can be many producers that feed many consumers, and there may be many instances of producer and consumer functions [5; 108]. Load balancing is achieved by routing data to different instances of consumers based on their current load and productivity. On the other hand, in pipeline-parallel stream systems, the data may need to flow through a series of pipelined filters where each filter can be viewed as a producer and consumer of input and output data. The order of filters constrains the order in which tasks may be executed.

Flexible filters are a solution for load balancing in pipeline-parallel stream programs. Many related works for balancing the load of pipeline parallel stream programs involve a central control and/or phases where the compute nodes collect statistics which are used by the control to direct reorganization [48; 113; 127]. The number of filters is designed to outnumber the cores, and load balancing is typically achieved by moving filters from nodes with heavy loads to nodes with lighter loads, similar to work stealing. Flexible filters simulate filter migration by duplicating some filters on the cores and invoking duplicates when the load becomes unbalanced.

Chen *et al.* perform load balancing for stream programs by compiling several alternative filter mappings [22]. During run-time, the system can "context-switch" between the alternatives based on the properties of the data. Flexible filters, on the other hand, dynamically adapt to the current flow behavior of the system. In the Diamond system developed by Huston *et al.*, data tokens are forwarded based on

threshold values in the input and output queues [70]. Load balancing with flexible filters similarly is an outcome of the state of the queues, but the difference is that flexible filters balance load based on backpressure. Moreover, DIAMOND is optimized for distributed search which relaxes several constraints of stream programs – namely that the filters need not be executed in a particular order because they are used to eliminate unwanted data (rather than transform the data) and that data can be processed in any order.

Many stream programming languages, such as StreamIt include *split* and *join* nodes in their supporting library that are used to transform the stream programs [48; 58; 121]. *Split* and *join* nodes in StreamIt can be used in two ways. First, the programmer may use them while writing a new stream program. Second, the StreamIt compiler may introduce *split* and *join* nodes to optimize the program by increasing data parallelism. This accomplishes static load balancing because the data flow is split at run-time regardless of the loads on the various cores. In contrast, the Flexible-Filter *flex_split* and *flex_merge* filters described in Section 4.2 are not intended for use when building a stream program, but are application-independent library filters that are introduced at a later stage when flexibility is added. Dynamic load balancing in my approach is based only on the insertion of *flex_split* and *flex_merge*. These are statically added during compilation but achieve dynamic load balancing via the backpressure mechanism applied to the dataflow.

4.5 Summary

Stream processing is a promising paradigm for programming multi-core systems for high-performance embedded applications. Flexible filters combine static mapping of stream program tasks with dynamic load balancing of their execution in order to improve system-level processing throughput of the program when it is executed on a distributed-memory multi-core system as well as the local (core-level) memory uti-

lization. The flexible filters technique is scalable because it is based on distributed point-to-point handshake signals exchanged between neighboring cores. Flexibility may be applied to any stateless filter without any modification to the filter itself, and only altering the overall stream program with the addition of the application-independent auxiliary filters *flex_split* and *flex_merge* around the filter and its flexible duplicate. The experiments in this chapter apply flexible filters to five stream benchmarks, and achieve performance speedup higher than 30% in most cases.

In the next chapter, a recursive parallel programing abstraction called Huckleberry is presented. Data flow is very obvious in a stream program because it is a direct result of the program structure. In Huckleberry recursive programs, data partitioning is exposed, but data dependencies are hidden from the programmer. Instead, the data dependencies are handled automatically at runtime. Chapter 6 presents a performance model for programming abstractions which frames all programming models as task graphs.

Chapter 5

Huckleberry: A Data Partition Abstraction

This chapter presents Huckleberry, a novel tool for automatically generating parallel implementations for multi-core platforms from sequential recursive divide-and-conquer programs. Explicit data partitioning is the cornerstone of Huckleberry's programming abstraction. With regard to the three questions that define a programming model – (1) what is expected of the programmer? (2) what does the model expose? and (3) what does the model hide? – Huckleberry expects the programmer to explicitly partition data using Huckleberry's partition library; the model allows for mutually recursive functions with inter-task dependencies within the recursive functions, and hides parallel programming concepts from the programmer including individual threads, synchronization and message passing. Huckleberry's high level of abstraction lessens the challenge of programming distributed memory architectures and extracting data parallelism from applications. My preliminary implementation of Huckleberry focusses on distributed memory architectures since Huckleberry includes data distribution techniques; however, the concepts of Huckleberry are not limited only to distributed memory architectures.

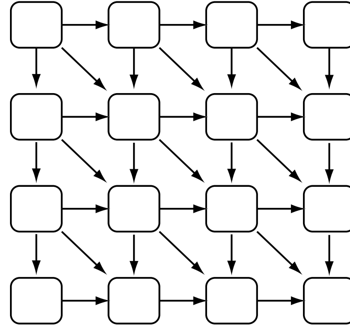


Figure 5.1: Wave-front dependency pattern.

Programming distributed memory multi-core platforms has distinct challenges: First, hardware may not provide cache-coherency, meaning that a parallel program must be designed so that tasks *and data* are explicitly distributed to the cores. A parallel implementation must schedule tasks and data to maximize concurrency and minimize unnecessary (expensive) off-chip data-swapping and also account for non-uniform access among the cores to the distributed memory banks. In addition, the implementation must coordinate the on-chip communication network with the system memory and task scheduling. Finally, the number of cores is likely to scale much higher in future generations of chips, thus requiring new algorithms that avoid single-point bottlenecks. Chips with more cores, whether they are distributed memory architectures or not, are more likely to experience more drastic non-uniform memory access latencies due to the increased relative latency of communication across the chip as feature sizes decrease and clock rates increase.

Recursive parallel models typically extract data-parallelism from an application by recursively divide-and-conquering a problem into many smaller pieces that may all be executed in parallel. Unlike the data-parallelism extracted by SPMD programs, recursive data-parallelism may include additional constraints, such as inter-task dependencies. For example, Fig. 5.1 illustrates an array with a wavefront dependency pattern. Data-parallelism is available in the diagonals (where no two tasks are dependent on one another); however, the number of elements in a diagonal changes with

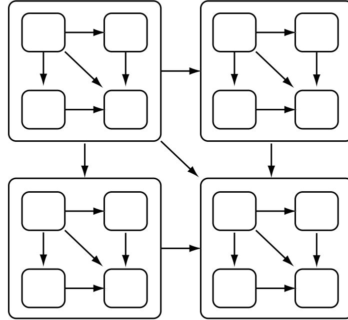


Figure 5.2: Recursive quadrant dependency pattern.

each step, making an iterative SPMD solution cumbersome. And if the size of the data set is large, and a diagonal is too large to fit on the chip all at once, data locality between diagonals is lost in off-chip data swapping.

Huckleberry's recursive programming model highlights the temporal and spacial locality of data use among tasks to promote schedules that distribute data nearby to the tasks that will use it and minimize data swapping. Huckleberry manages data swaps between cores with distributed synchronization to avoid a bottleneck in processing. Recursive parallelism references data in a top-down fashion, so that different chunks of the problem are separated into pieces that will fit on the chip. Fig. 5.2 shows a recursive quadrant pattern which can be recursively applied to the wave-front array from Fig. 5.1 in order to manage the varying diagonal sizes with a simple hierarchical pattern. Notice that each quadrant is a smaller version of the overall problem and could be distributed as a unit onto the chip to preserve data locality between tasks.

Huckleberry differs from other recursive parallel programming models because it completely abstracts the decomposition of concurrent tasks so that a programmer need only to focus on the data partitioning, and the code generator transparently expands a Huckleberry recursive function into a parallel task graph. Recursive algorithms are used by Huckleberry's code generator not only to automatically divide a problem up into smaller tasks, but also to derive lower-level parts of the implemen-

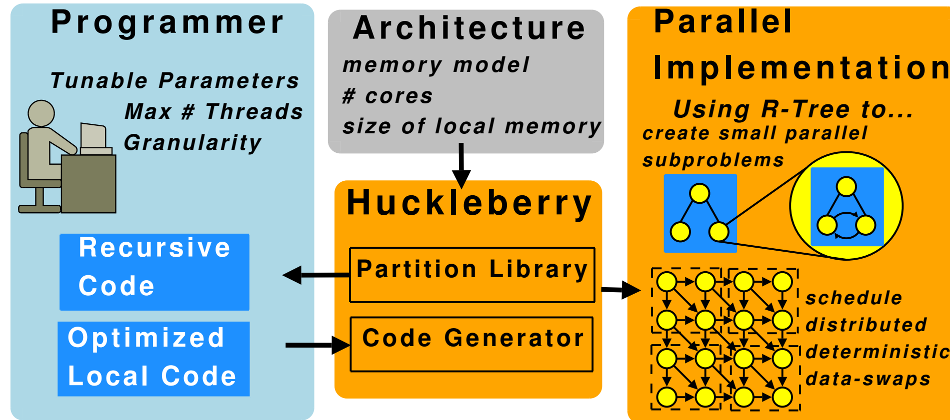


Figure 5.3: Huckleberry design flow.

tation, such as data distribution and inter-core synchronization mechanisms. Huckleberry's recursive parallel model supports interaction between nested calls including data-dependencies between the calls and mutually recursive functions that alter the nested data access pattern. It does so by leveraging the powerful mechanisms for inter-core communication and synchronization that are typically provided by on-chip networks.

Fig. 5.3 illustrates the Huckleberry design flow. The programmer provides one or more *divide-and-conquer* recursive functions that employ Huckleberry's Partition Library application programming interface (API). These functions will be referred to as *user-input functions* throughout this chapter. The code generator takes user-input functions together with specifications of the underlying architecture and returns a parallel implementation. The recursive task graph resulting from the combined recursive functions of an application, called an *R-Tree*, is used at runtime in the parallel implementation to: (1) break the problem up into subproblems small enough to fit onto the chip, (2) distribute data across the cores, and (3) coordinate data swaps when necessary between the cores. Huckleberry also gives the programmer the flexibility of specifying which core is responsible for which task (through its *parallel-index function* f_{pi}) and of using optimized local code that runs at the leaves of the

R-Tree (i.e., on individual cores). Decentralization is a central idea to the Huckleberry approach, and is achieved by allowing cores to calculate for themselves which tasks they are responsible for, and when and with whom they should swap data. By keeping as much communication contained on a chip as possible, Huckleberry takes advantage of the performance edge delivered by high-speed on-chip networks. The first release of Huckleberry supports multi-core platforms based on the Cell BE processor and generates parallel code for a variety of benchmarks. The experiments presented adjust code generation parameters to uncover the distinct demands that each benchmark places on the system's resources.

5.1 Huckleberry Programming Interface

Huckleberry is based on the C programming language, supplemented by Huckleberry's partition API. The constraints on the programmer are as follows: foremost, only recursive divide-and-conquer functions where the divide step can be determined before the compute steps are supported (i.e., the partitioning of the data does not depend on the data; however, there can be *data dependencies* between branches where several steps of the algorithm alter the same data.). If a user-input function has this property, as it is the case for bitonic sort that is introduced as an example later in this section, the programmer can modify it to be accelerated by Huckleberry simply by wrapping all of the function's parameters with the API's `Partition` data structures. In return, Huckleberry abstracts away the details of implementing a parallel algorithm. The programmer does not need to separate the algorithm into independent tasks or consider architectural details like the number of cores or the size of the local memory. Huckleberry supports mutually recursive functions that invoke one another, and multidimensional data in user-defined types.

5.1.1 Partition Library

The `Partition` API is the centerpiece of the Huckleberry partition library. User-input functions must use `partitions` for all of their parameters. Partitions support generic data structures, but annotate the actual data with meta-information about the data; for example, array-based metadata includes data type, dimensions, and where a partition's data begins and ends within each dimension. The partition API includes the following functions:

- `create_partition()` creates and fills a partition;
- `free_partition()` frees the memory of a partition;
- `left_half()` copies a partition's metadata into a new partition, altering the new partition to only include the original partition's left half;
- `right_half()` is the inverse of `left_half()`;
- `copy_element()` copies an element of an array into a new unit partition (a small data structure that is not divided but is passed down the R-Tree intact);
- `update_int()` updates an integer unit partition;
- `mydata_intersects()` returns true if any part of two partition sets intersect and false otherwise;
- `mydata_contains()` compares two sets of partitions, a local and global set; returns true if the local set entirely contains the global set, and false otherwise;
- `partition_size()` calculates the size of a partition based on the number of dimensions in a partition and the begin and end boundaries of each partition.

The partition library provides functions that perform operations on partitions to reduce their size for the divide step of divide-and-conquer functions, including adjusting their data pointers and keeping track of the original boundaries and where the new

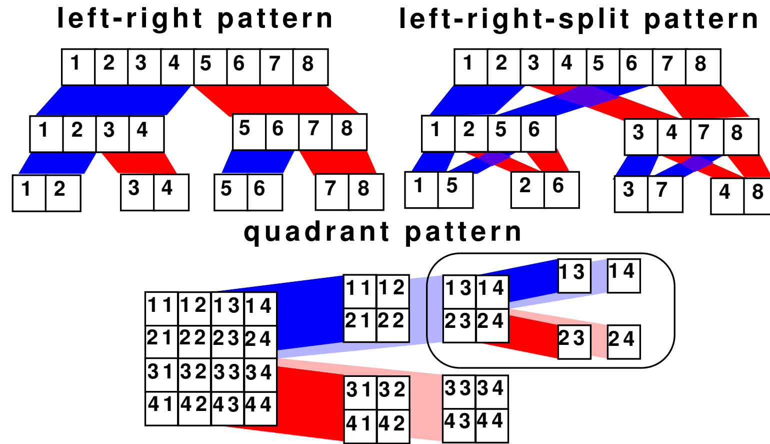


Figure 5.4: Patterns of recursively applied partition methods.

reduced partition lies within them. My initial implementation of Huckleberry supports data that is arranged in arrays of one or more dimensions. Data structures such as trees could also be supported using the same concepts. For example, `left_half()` and `right_half()` may take the left and right subtrees of the tree structure (assigning the root node to one of the halves).

Repeatedly applying the partition library methods to data results in a *partition pattern*. Fig. 5.4 shows three patterns that break larger data structures down into smaller pieces in a divide-and-conquer fashion. The patterns use the same library methods, but in different combinations. For example, the *left-right* partition pattern shows how data is broken down if `left_half()` and `right_half()` are used to partition data into halves once for each branch, while the *left-right-split* pattern uses `left_half()` and `right_half()` twice per branch.

5.1.2 Example

Bitonic sort is a divide-and-conquer algorithm where a list of elements is sorted by first sorting its two halves in opposite directions, and then merging the two halves together [32; 56]. While having a complexity of $O(n\ log^2\ n)$, which is slightly less

Algorithm 3 `sort`(Partition *list*, Partition *dir*)

$idir \leftarrow$ extract_int(dir)

$left \leftarrow$ left_half($list$)

$right \leftarrow$ right_half($list$)

sort($left, dir$)

update_int($dir, idir * -1$) // *change directions for the next half*

sort($right, dir$)

update_int($dir, idir * -1$)

merge($left, right, dir$)

sort_merge_only($left, dir$)

sort_merge_only($right, dir$)

Algorithm 4 `sort_merge_only`(Partition *list*, Partition *dir*)

$idir \leftarrow$ extract_int(dir)

$left \leftarrow$ left_half($list$)

$right \leftarrow$ right_half($list$)

merge($left, right, dir$)

sort_merge_only($left, dir$)

sort_merge_only($right, dir$)

Algorithm 5 `merge`(Partition *left*, Partition *right*, Partition *dir*)

$left_of_left \leftarrow$ left_half($left$)

$right_of_left \leftarrow$ right_half($left$)

$left_of_right \leftarrow$ left_half($right$)

$right_of_right \leftarrow$ right_half($right$)

merge($left_of_left, left_of_right$, dir)

merge($right_of_left, right_of_right$, dir)

efficient than $O(n\ log\ n)$ sorting algorithms like *Merge Sort* or average-case *Quick Sort*, bitonic sort is a popular parallel sorting algorithm because the order of its compare-and-swap operations is not data dependent.

Algorithms 3, 4 and 5 show a recursive implementation of bitonic sort written with the Huckleberry API which consists of three mutually-recursive functions. The `dir` partition provides the direction that the list should be sorted in, and is a unit

partition. This example demonstrates how the programmer may statically express data partitioning at a high level of abstraction, while the generated parallel code adapts partitions dynamically to runtime parameters (input size, available memory, etc.). Notice that the functions lack exit conditions. Huckleberry's code generator inserts function wrappers around recursive functions which manage the exit condition based on the runtime size of the `Partition` data compared with the available memory in the underlying architecture. At the leaves of the R-Tree, exit conditions (or a non-recursive optimized local code, discussed next) are necessary.

5.1.3 Optimized Local Code

The programmer may provide an optimized implementation of the user-input functions to be used once a partition is small enough to fit in a single core's local memory. It is called *local* code because it is executed when all of the relevant data is in a core's local memory space. Since local code is executed sequentially, it can be optimized using standard sequential coding techniques which may be specifically designed for the hardware in use (e.g., vectorized code for Cell's SPEs). Optimized local code is essential for good overall performance because it will be repeated many times during execution. In the bitonic sort example, for each instance of `sort()` that is called on a platform with 16 cores, `local_sort()` is called 1 time, `local_merge()` 10 times, and `local_sort.mo()` (abbreviated from `sort_merge_only()`) 4 times for each core (and 16, 160 and 64 times on all cores together). Thus, performance improvements in the local code can translate into significant overall improvements. The bitonic sort benchmark experiences more than a 10x speedup when switching between recursive unoptimized local code and (non-recursive) optimized local code. However, the dramatic performance benefits of a highly optimized local code only impact what happens on a single core. Huckleberry is able to take good single-core performance and extend it to a complete multi-core implementation, bridging together many instances of the single-core code.

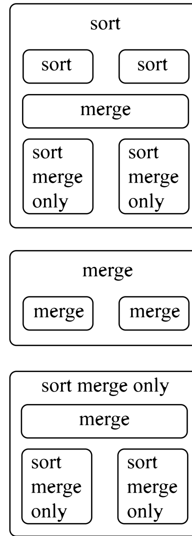


Figure 5.5: Nested dependencies between bitonic sort recursive functions from Algorithms 3 and 5 (programmer view).

5.2 Huckleberry Parallel Code Generator

The Huckleberry code generator creates a parallel implementation that follows a hierarchical call tree at runtime by recursively unrolling the nested structure provided by the programmer. For example, bitonic sort in Sec. 5.1.2 is made up of several recursive functions that interact with each other in the nested dependency structure illustrated in Fig. 5.5. In this section, after describing the machine model used for code generation I describe how a user-input function is parallelized from the perspective of the thread of execution at runtime.

5.2.1 Machine Model

A distributed memory multi-core system is made up of a set of N cores, each of which is associated with a local memory whose capacity is denoted m_i for core c_i. There may also be an *organizer core (OC)* dedicated to sequential and administrative tasks or other special purpose cores. The capacity m_i reflects the available memory for

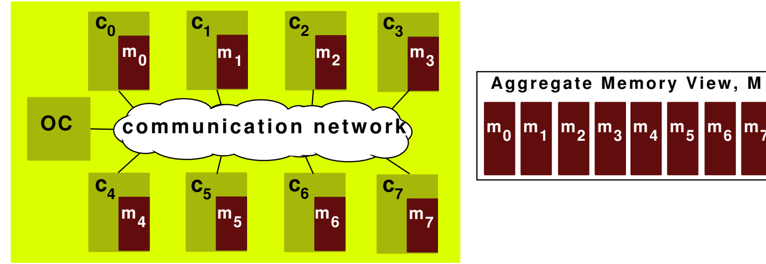


Figure 5.6: Abstract machine model.

data once space for the application code and temporary buffers has been accounted for. The *aggregate local memory*, denoted $M = \bigcup_{\forall i} m_i$, is the sum total of the local memories of the cores. The value of m_i is processed by Huckleberry as an input parameter to generate the parallel code and can be varied to change the granularity of the parallel execution; m_i can also vary depending on the particular application. As in typical sequential programming models, the notion of data is kept separate from memory. I denotes the program's input data set while D denotes the working data set that is stored in the aggregate local memory M at any given time during the execution of the program. Finally, d_i denotes the subset of D that is stored in the local memory space m_i. Fig. 5.6 illustrates how the machine model can be used to derive an abstraction of the Cell processor, with 8 SPE vector cores, and one PowerPC core which serves as the OC.

5.2.2 Stages of Execution

The code generated by Huckleberry creates a flow of execution that passes through three major stages (Fig. 5.7). Each stage addresses a different aspect of the parallel implementation: the *locality* stage determines which subproblem should be executed next (when the overall input data size is too large for the chip; the *distribute data* stage distributes data to the chip; and the *concurrency* stage is executed concurrently by cores. All stages are generated by refactoring the user-input functions with wrapper

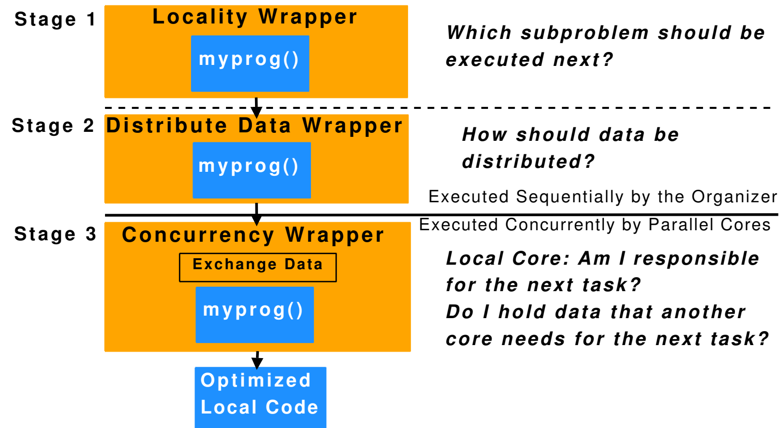


Figure 5.7: Stages in a Huckleberry-generated parallel application.

functions that make different scheduling decisions, which are implemented as follows: (1) the user provides a user-input function, `myprog()`; (2) the code generator inserts a wrapper by replacing calls for `myprog()` with `wrapper_myprog()`, including within the body of `myprog()` itself; (3) `wrapper_myprog()` performs bookkeeping steps and then calls `myprog()`. Interleaving calls to `myprog()` and `wrapper_myprog()` in this way has the effect of executing some extra code around each of the instances of `myprog()`. For each of the three stages, there is a wrapper and a separate copy of the original recursive function. The current implementation of Huckleberry assumes that the input recursive functions correctly partition data so that each branch covers a proper subset of the data of its parent. Partition set size is used in the exit conditions of the wrappers.

5.2.2.1 Locality Wrapper

The *locality* wrapper stage decides what part of the problem should be executed next when the initial input problem size is too large for the aggregate local memory M of the cores. This step is executed only by the organizing core, and it is executed sequentially to preserve data dependencies in the recursive program while maximizing data locality.

Algorithm 6 `loc_wrapper_myprog`(*data*). In the bitonic sort example, *data* for `loc_wrapper_sort()` includes `list` and `dir`.

if $|data| \leq M$ **then**
  call dd_wrapper_myprog()
else
  loc_myprog(*data*)
end if

Algorithm 7 `loc_merge`(Partition *left*, Partition *right*, Partition *dir*)

$left_of_left \leftarrow$ left_half($left$)
$right_of_left \leftarrow$ right_half($left$)
$left_of_right \leftarrow$ left_half($right$)
$right_of_right \leftarrow$ right_half($right$)
loc_wrapper_merge($left_of_left, left_of_right$, *dir*)
loc_wrapper_merge($right_of_left, right_of_right$, *dir*)

The steps of the locality wrapper are shown in Alg. 6. Note that the wrapper is application-independent, and will look the same for any program `myprog()`. Alg. 7 illustrates how the locality wrapper is wrapped into the `merge()` function from Alg. 5 with the `merge()` renamed to `loc_merge()`. The replica of `merge()` is prefixed with "`loc_`" in order to distinguish it from the replicas called by the distribute data and concurrency wrappers (prefixed with "`dd_`" and "`con_`", respectively). Each call to `loc_merge()` is wrapped with a call to `loc_wrapper_merge()`, which checks that the problem size is small enough for M by iterating through a list of the input partition parameters and calculating their size based on the `partition_size()` subroutine from the partition library. When the exit condition is met (i.e., the size *is* small enough), the locality wrapper calls the next stage, the *distribute data* wrapper. The assumption that branches cover a proper subset ensures the problem size is reduced with each step.

Algorithm 8 `dd_wrapper_myprog` (*data*, *depth*). Let m be the size of data assigned leaf nodes.

if $|data| \leq m$ **then**
  $i \leftarrow f_{pi}(id_seq)$
  if *data* has not been already sent **then**
    send *data* to core i
  end if
else
  dd_myprog(*data*, *depth*)
end if
$id_seq[depth] \leftarrow id_seq[depth] + 1$

5.2.2.2 Distribute Data Wrapper

The *distribute data* wrapper distinguishes between individual cores and their neighbors. The *distribute data* stage starts with a problem that will fit in the aggregate local memory of the N cores, and breaks the problem up into N pieces based on the application's R-Tree. For example, a divide-and-conquer function that divides its input in two ways can be represented as a binary tree, whose leaves correspond to instances of the function that reach the exit case. Each node in the tree is uniquely assigned to a specific core that is determined by calling the *parallel-index function* (f_{pi}) on the node's position in the tree. F_{pi} operates on two parameters: depth and sibling order id (e.g., left child 0, right child 1) of a node and its parents in the R-Tree.

The *distribute data* wrapper (shown in Alg. 8) does several things. First, it keeps track of the current sibling id at each level of the tree with an array *id_seq*[] and the current depth. $Id_seq[depth]$ is incremented every time `dd_wrapper_myprog()` is called. `Dd_wrapper_myprog()` includes depth as an input parameter; for example, `dd_wrapper_merge`($left_of_left$, $left_of_right$, dir) becomes `dd_wrapper_merge`($left_of_left$, $left_of_right$, dir, $depth + 1$). The minimum possible depth is determined by the size of each core's local available memory. However, granularity can be tuned by traversing deeper into the R-Tree to reach smaller leaf tasks. With smaller leaf tasks, a core

is assigned more smaller tasks vs. fewer larger tasks. With coarser task granularity, dependencies may reduce parallel speedup by forcing some tasks to wait for others. Finer task granularity also may have finer grained dependencies so that greater concurrency is possible. Second, the wrapper applies f_{pi} to determine where to send the next data. Last, it keeps track of which data it has already sent to the cores. Data may be revisited several times in the R-Tree, but it only needs to be transferred to the chip once.

5.2.2.3 Concurrency Wrapper

The *concurrency* wrapper is similar to the *distribute data* wrapper because it starts with the same data, uses the same f_{pi} function, and handles *depth* and *id_seq*[] in a similar fashion. However, while the *distribute data* wrapper is executed once on the OC, the *concurrency* wrapper is executed in parallel on each core over the global data set that is shared among the cores. Each core is aware of its own *rank* in the group. In addition to identifying which core is responsible for each task, the *concurrency* wrapper also organizes synchronization among the cores and data swapping. Data swapping is needed when one core must read or modify data that has already been modified by another core, i.e., there is a data dependency between tasks assigned to different cores.

The *concurrency* wrapper handles context switches by recalculating the correct depth depending on the data size and resetting the values of *id_seq*[] to 0 for elements beyond the new depth. The steps of the *concurrency* wrapper are shown in Alg. 9. A *push* handshake protocol organizes data swaps: a core that needs data simply waits to proceed until another core sends data, and the sending core will not send data until it has reached the same node in the R-Tree as the first core. The downside of this protocol is that some concurrency may be lost because the sending core does not send data as soon as it is available.

Algorithm 9 `con_wrapper_myprog` (*data*,*depth*).

context switch if necessary
if $|data| \leq m$ **then**
  $i \leftarrow f_{pi}(id_seq)$
  if i is $rank$ **then**
    if mydata_contains($data$) **then**
      wait for $data$
    end if
    local_myprog($data$) $//not\ to\ be\ confused\ with\ loc_myprog()$
  else if mydata_intersects($data$) **then**
    send $data$ to core i
  end if
else
  con_myprog($data$, $depth$)
end if
$id_seq[depth] \leftarrow id_seq[depth] + 1$

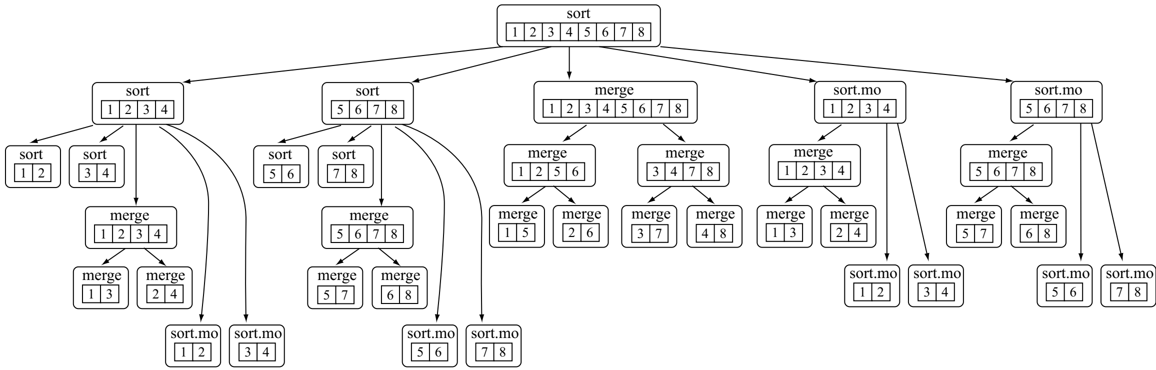


Figure 5.8: The R-Tree of bitonic sort recursive functions (code generator view), unrolled until the data partition size is two data blocks.

5.2.3 Example: Traversing the R-Tree

Fig. 5.8 illustrates the hierarchical call tree, or R-Tree, constructed for bitonic sort from its three functions `sort()`, `merge()`, and `sort.mo()`. In the locality stage, the

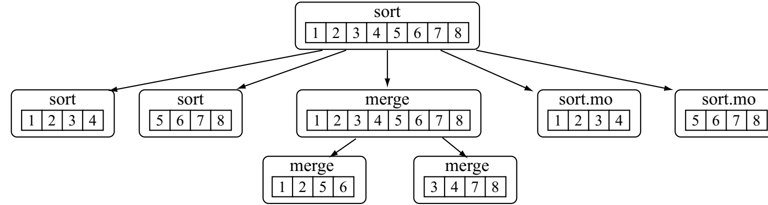


Figure 5.9: Subset of bitonic sort R-Tree visited by the locality wrapper.

OC initially traverses the R-Tree until it reaches a point where the data size is less than or equal to M. Suppose that there are two cores ($N = 2$), and each core can hold two data blocks ($m_i = 2$, and $M = 4$). Fig. 5.9 shows the nodes of the tree from Fig. 5.8 which would be visited during the locality stage. In each branch, the locality wrapper stops when the data partition is reduced to four data blocks (two blocks $\times$ two cores). Notice that when the tree stops at a `sort()` branch with four data blocks, although `sort()` has a `merge()` sub-branch that also has four data blocks, the locality wrapper does not consider the `merge()` sub-branch separately. Instead, the concurrency stage will manage the `merge()` sub-branch since the data used by that sub-branch will already have been distributed.

Next, the OC continues in the distribute data stage. The distribute data stage picks up in the R-Tree where the locality stage stops, and continues in the R-Tree until it reaches data partitions that are small enough to be distributed to individual cores. To distribute data, it is not necessary to traverse the entire tree; it is possible to stop the traversal when all of the data has been distributed once. The distribute data and concurrency stages use f_{pi} to determine which core is responsible for which data and tasks. As mentioned, it is possible to tune the granularity smaller so that the OC traverses deeper and distributes multiple smaller data partitions to the cores.

Up until this point, execution has taken place sequentially, but once the data has been distributed, the cores continue in parallel in the concurrency stage. Fig. 5.10 illustrates the subtree of `sort()` visited in the concurrency wrapper. This example assumes that there are two cores, each of which can hold two data blocks (and has

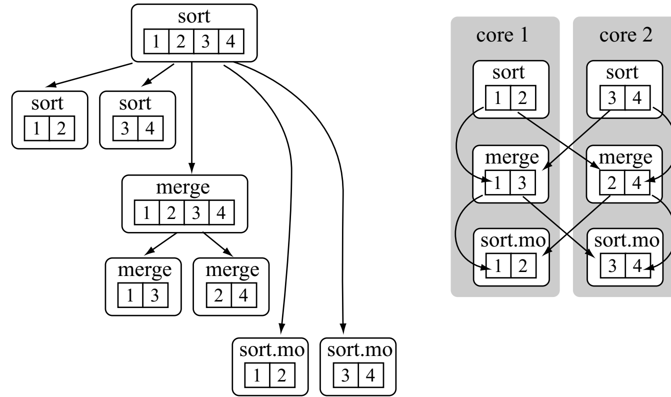


Figure 5.10: One subset of the bitonic sort R-Tree visited by the concurrency wrapper, when there are two cores with a memory capacity of two blocks.

sufficient buffering to swap one block with another core). Thus, the concurrency wrapper is called when the data partition has been reduced to four data blocks. Because different recursive functions may use different partition patterns, the divide-and-conquer pattern may be disrupted when switching between different recursive functions as is the case when switching between `sort()` and `merge()`. Each core in this example manages three tasks, and data swaps between the tasks as necessary. With only two cores, after these steps are completed, the data blocks 1-4 must be swapped off chip with data blocks 5-8 to complete the next steps. If four cores were available instead, more tasks could be completed in one concurrency stage as shown in Fig. 5.11. Execution returns to the locality stage tree once the complete subtree has been executed in the concurrency stage tree.

5.3 Experiments

I evaluated the initial implementation of Huckleberry on the QS20 Cell Blade [98] because of its flexibility and computational power and because it represents the class of distributed-memory multi-core platforms. Each QS20 features two Cell BE processors

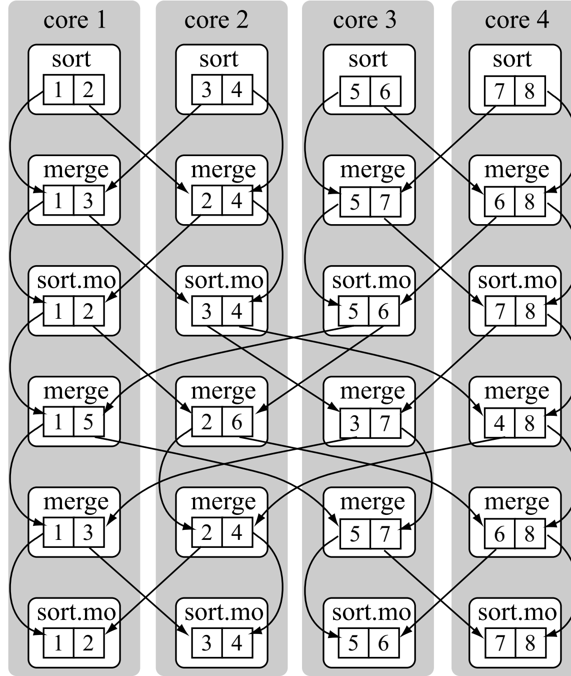


Figure 5.11: One subset of the bitonic sort R-Tree as visited by the concurrency wrapper, when there are four cores, each with a memory capacity of two blocks.

together with 1GB of XDRAM. In these experiments, Huckleberry derives parallel implementations targeting the QS20 for four benchmarks (revisited from Chapter 3):

1. *Smith-Waterman Sequence Alignment* is a dynamic programming algorithm which computes a similarity score between two sequences such as DNA sequences [60; 115]. The algorithm fills in a matrix m starting from the top-left corner such that $m[i, j]$ is assigned a value which is a function of $m[i-1, j]$, $m[i, j-1]$, and $m[i-1, j-1]$. The Smith-Waterman benchmark is implemented in Huckleberry using a combination of the *quadrant* pattern on its two-dimensional matrix data and the *left-right* pattern on its one-dimensional sequence data.

2. *Black-Scholes* is an algorithm for stock-option pricing. The algorithm calculates the price of a stock option given information such as the current price, time

period, volatility, interest rate, etc; and can be calculated with a closed form equation. Black-Scholes is implemented using the *left-right* pattern to distribute data.

3. *One-Dimensional FFT* is implemented based on the 'four-step' method [7] with the bit-reversal algorithm [72]. The input array is a 2D matrix, and the FFT is computed by performing smaller FFTs on the matrix rows and columns. For the FFT the *left-right* pattern is used.

4. *Bitonic sort* is implemented as described in Section 5.1, using both the *left-right* and *left-right-split* partition patterns.

All these benchmarks are amenable to a divide-and-conquer specification, but they are different in nature and stress our approach in different ways. These experiments focus on evaluating the overhead and trade-offs of communication rather than on optimizing local single-core code to get the best performance.

5.3.1 Scalability

Fig. 5.12 shows the performance speedup for large problem sizes as the number of cores is increased. These problem sizes require multiple stages in the *locality wrapper*. The Black-Scholes benchmark performs almost ideally, which is expected, since the benchmark is an example of an "embarrassingly parallel" program. This demonstrates that the overhead of partitioning and distributing the problem in the absence of inter-core communication is very low. The overhead of inter-core data passing and synchronization is more difficult to quantify with respect to alternative implementations; however, using double-buffering to hide the overhead is a potential solution in both cases, though it is not implement here. Two curves for bitonic sort are shown; the bitonic sort benchmark achieves slightly more than a 5x speedup with 16 cores for the smaller problem size (128K integers), but the speedup degrades as the problem size increases. The size of local memory likely plays a role in the parallel speedup

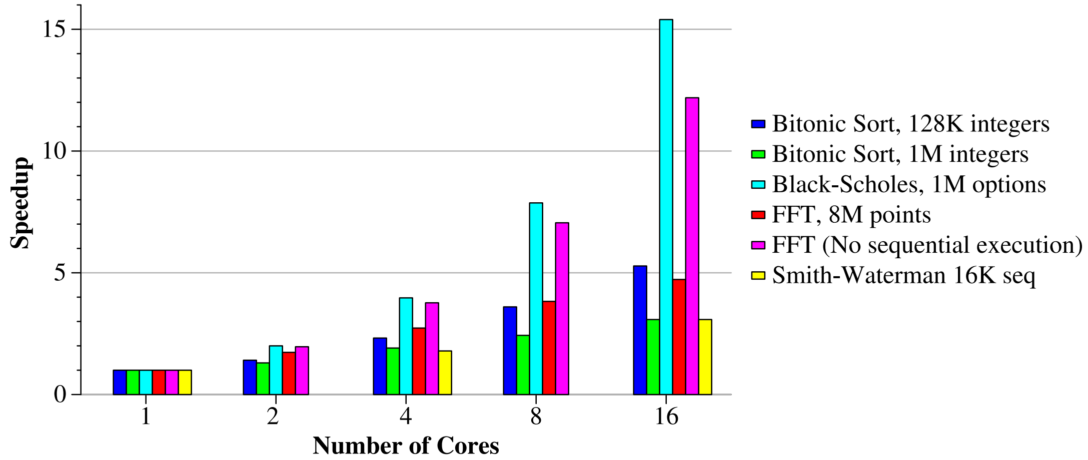


Figure 5.12: Scaling cores: Speedup when D and m_i are fixed and the number N of available cores scales up.

in this case because a hand-coded recursive implementation was able to compute larger problems on a single core than the Huckleberry-generated implementation, and achieved a speedup closer to 7x with larger problem sizes. For the Smith-Waterman benchmark, the data dependencies of the algorithm limit speedup. Namely, imposing the dependencies of high levels of the hierarchy onto lower levels causes some cores to wait for data exchanges longer than is necessary. This behavior may be improved by changing the data swap protocol. The FFT benchmark achieves a speed-up of 5x, though notably, increasing from 4 to 8 and from 8 to 16 cores does not significantly improve performance; as per Amdahl's law, matrix transpose and multiplication operations are performed sequentially on the OC in our implementation, even though performance is near ideal when the sequential operations are excluded. Algorithmic optimizations and the use of huge page sizes on the Cell may be a possible solution to reduce the time consumed by these operations, as suggested by Chow *et al.* [23].

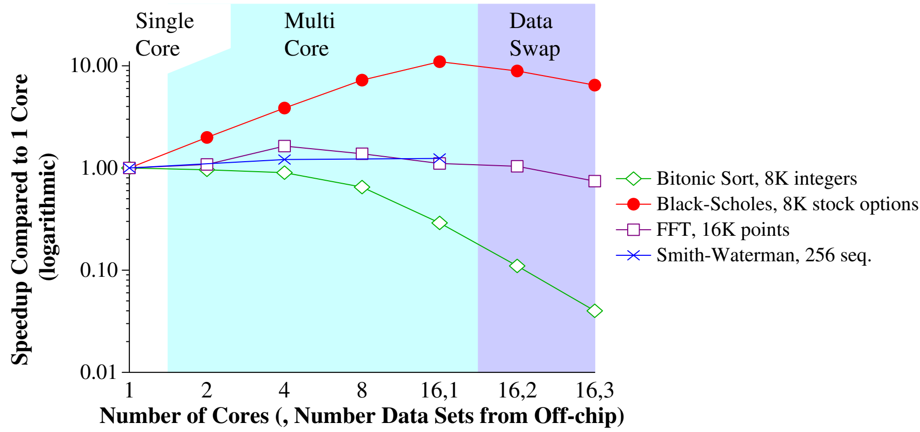


Figure 5.13: Scaling task granularity: Speedup when I is constant, but m_i is scaled down, forcing more cores to work on the problem.

5.3.2 Problem Granularity

Fig. 5.13 shows how performance changes as the problem granularity becomes finer. The problem size is fixed and is small enough to fit in the local memory of a single core, but more cores are recruited for their additional computational power. For example, the Black-Scholes curve corresponds to pricing 8K stock option values. With one core, all options are calculated by this core; with four cores, each core calculates $\frac{8K}{4} = 2K$ options. Cases *16,2* and *16,3* correspond to each core calculating 256 and 128 options, respectively.

The curves are highlighted in three groups. In the first, data is small enough to fit on a single core; in the second, data is small enough to fit entirely in the aggregate local memory space; in the third, data is swapped on and off the chip. Breakpoints between groups occur at different places for the benchmarks. For example, bitonic sort requires many inter-core data exchanges. During data exchanges, temporary buffers take up some of the local memory space, and limit the size of the input data that can be assigned to a single core.

The benchmarks perform strikingly differently. The Black-Scholes benchmark improves almost linearly as more cores are utilized (note the logarithmic y-axis).

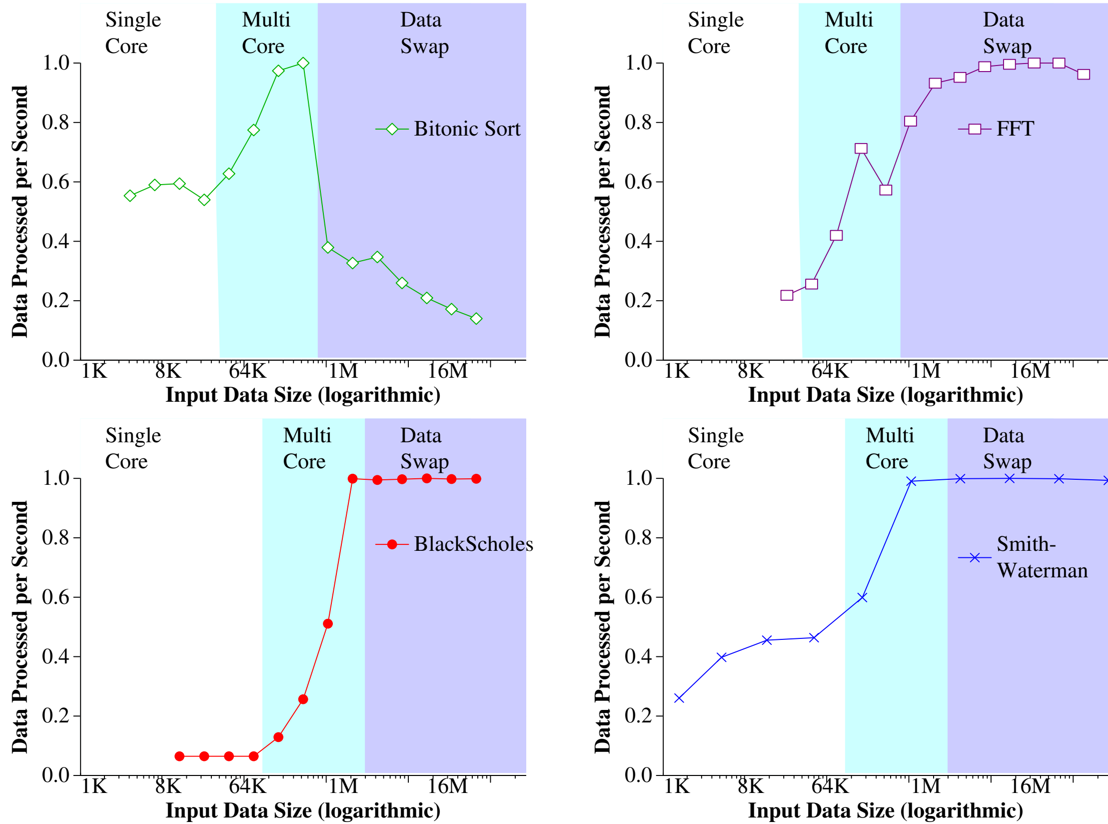


Figure 5.14: Scaling data size: m_i remains fixed, while I scales up, normalized w.r.t. the highest throughput instance in that benchmark.

However, data swapping eventually becomes a bottleneck as the problem granularity is reduced. The Smith-Waterman benchmark's speedup improves slightly as N is increased to 16, but is relatively flat. Performance of the FFT benchmark first improves and then degrades as granularity is increased, while the bitonic sort benchmark performs best when the entire problem is handled on one core. For all benchmarks, the cost of additional off-chip data swapping outweighs the benefits of increased concurrency.

5.3.3 Throughput and the Role of Local Memory

Fig. 5.14 plots the performance of the benchmarks as the input data size scales up, but granularity is fixed. The results in Fig. 5.14 show why bitonic sort performs better with smaller data (128K integers versus 1M integers) in Fig. 5.12. For input sizes that do not require data swapping, the benchmark throughput increases with the input size, but once data is large enough to require data swapping, the throughput drastically decreases. For the other three benchmarks throughput stays steady as the input size increases beyond what will fit on a chip. Since data swapping is not the bottleneck for these benchmarks, a good balance of communication and computation has been achieved.

5.4 Related Works

With respect to related works in recursive parallelism, Huckleberry is distinct because of its programming abstraction and the interaction of its data partitioning library with parallel code generator. The parallel implementation generated by Huckleberry reuses the programmer's user-input recursive functions not only to break a problem up into concurrent tasks, but also to break a data set up into pieces that will fit on-chip in one stage, and to schedule synchronization between concurrent tasks. Nested dependencies and context switches between mutually recursive functions are detected at runtime and managed with distributed control.

Huckleberry follows a number of works in recursive parallel programming. All take advantage of the divide-and-conquer hierarchy to decompose an algorithm into parallel tasks. The Sequoia programming language uses hierarchical program design to leverage data locality in the memory hierarchy of a parallel system, and also supports the Cell architecture [47; 80]. In Sequoia, different layers of the hierarchical tree are associated with different levels of memory. Concurrent tasks are isolated and do not synchronize, but communicate through their parent task (which may be

mapped to the same core). The Sequoia compiler plays a role in optimizing the parallel implementation. The parallel implementation generated by Huckleberry similarly focusses on the memory hierarchy; however, the Huckleberry compiler performs no optimizations, but is paired with a distributed application-independent runtime library that has access to the aggregate memory view and partition metadata on local cores. Compilers have also been used to parallelize divide-and-conquer programs in other parallel architectures by analyzing memory references to detect dependencies [64; 109]. Cilk is an expressive general-purpose C-based parallel programing language that includes support for recursion [14]. Cilk does not abstract parallelism from the programmer to the same extent that Huckleberry does; the programmer must expose parallelism in applications through the use of thread keywords such as *spawn* and *sync*. Huckleberry, in contrast, requires the equivalent of data partitioning keywords (i.e., library functions) instead of thread keywords.

NESL is a nested parallel programming languages [12]. Huckleberry is novel with respect to NESL because data passing and inter-core synchronization are determined at runtime via a distributed decision making process which is fully integrated with the distributed tasks. Algorithmic Skeletons capture abstract communication patterns of parallel programs, and are intended to be developed separately from the algorithmic specification of an application by system and application experts, respectively [59]. The divide-and-conquer skeleton, which supports the parallelization of recursive programs, is implemented with SPMD parallelization based on the *powerlist* data structure [94]. Huckleberry's implementation does not separate the recursive algorithm from the application's communication pattern, but instead models the communication after *partition patterns*.

5.5 Summary

As multi-core systems of the future scale up to large numbers of cores, there is a need for tools that can abstract away the process of separating a program into parallel tasks. My goal with Huckleberry is to create such a tool for recursive divide-and-conquer programs where the programming abstraction is based on data partitioning. My experiments demonstrate Huckleberry's ability to automatically generate parallel implementations from sequential recursive functions. I find that the speedup available from parallelization provided by Huckleberry is affected by the interaction of data dependencies and workload.

What does Huckleberry's recursive model have in common with stream graphs? A stream graph is a fairly low-level parallel abstraction. The programmer identifies the tasks and how they communicate, and (possibly after compiler optimizations) the tasks are mapped directly to cores. The Huckleberry recursive model, on the other hand is high-level, and the programmer does not identify individual tasks. However, at run-time, the program is broken up into a task graph where some of the tasks may execute in parallel. Both programming models share a common task graph form at run time, and this task graph can be used as a common framework. The next chapter describes a model for the task graphs of multi-core applications which simplifies the search for a good mapping to the hardware platform.

Chapter 6

A Performance Model for Multi-Core Applications

The trend towards multi-core chip design anticipates performance gains proportional to the increasing number of cores. In fact, it is very difficult to achieve linear speedup as the number of cores increases. If there is any imbalance between the tasks that two cores are executing, then one core will become idle, and whenever a core is idle, the potential performance gain promised by its presence is lost. Secondly, when there are many more tasks than cores, these tasks must be co-mapped. It is then a challenge to choose a good partitioning of the tasks among the cores that balances the execution times and the communication and memory needs of the tasks. Dependencies between tasks can also limit their ability to be executed in parallel, further reducing performance. Thus, finding a mapping of an application is an essential part of designing a balanced parallel implementation of that application. Mapping applications to multi-cores has been studied in a number of different contexts [13; 81], and in general is a hard problem [53; 95].

When exploring the search space for a good mapping, the performance for new mappings can be evaluated through either experimentation or simulation. Empirically implementing and testing an application on a target multi-core platform can

be very time consuming. The implementation is challenging for the reasons mentioned in Chapter 1, for example, the correct usage of synchronization primitives, and low-level memory management such as direct memory access operations and data alignment. Moreover, many multi-core platforms lack sophisticated debugging tools, which further increases the time to design and test an application. For these reasons, a performance model may be preferable over actual experiments to facilitate rapid design-space exploration of different filter mappings.

In this chapter, I present the compositional multi-core performance (CMCP) model for multi-core applications. The proposed model provides a representation of a program that brings together the algorithmic and hardware dependencies. The CMCP model captures the physical constraints of a parallel architecture including mutual exclusion, buffer capacity, communication latency, and off-chip data swapping as well as the properties of the application such as the execution time and composition of tasks. The model is *compositional* because the different constraints (e.g., mutual exclusion, etc.) are captured in modular constructs which are modelled separately, but can be composed together to create a comprehensive representation of an application. Furthermore, it is extensible to performance optimizations such as flexible filters.

The CMCP model is based on the Petri net model of computation, which captures the data-driven nature of the programming abstractions and applications presented in this thesis. Programs are represented as a set of tasks together with a network of the data movement between them.

Sec. 6.1 briefly outlines the Petri net model of computation and Sec. 6.2 describes in detail the CMCP model and how it can be modularly built up to represent different aspects of a parallel implementation of an application. Next, in Sec. 6.4, I present an experimental evaluation of the CMCP model compared to actual performance data for the benchmarks presented in Chapters 4 and 5 for Flexible Filters and Huckleberry.

6.1 Petri Nets

Petri nets are a mathematical model of computation [103]. Originally invented by Carl Petri for describing chemical compound production nets, Petri nets have been used over the years for modelling and performance evaluation of many computer systems, including manufacturing, hardware design, and communication protocols, and more recently systems-on-a-chips and distributed software systems [26; 40; 44; 73]. Petri nets are a convenient representation for evaluating concurrent systems because it is possible to analyze a Petri net and determine its behavioral properties such as graph liveness, boundedness, and reachability [97]. After a brief description of Petri nets, this section shows how multi-core application task graphs can be modelled using Petri nets.

Petri nets are directed bipartite graphs with two kinds of nodes, *transitions* and *places*. Transitions may only be connected to places, and vice versa. A place is a container for tokens and starts with an initial *marking*, denoting how many tokens it initially contains. A transition is a node that connects places to each other, and is able to *fire*, consuming tokens from each of the places on its incoming arcs, and producing tokens on each of its outgoing arcs. A transition is only enabled to fire when all of the places on its incoming arcs have a sufficient number of tokens. Each arc is assigned a weight corresponding to the number of tokens that is produced or consumed during one firing event. The rate of production/consumption of tokens by the actors of a Petri net can be used to model the data-processing throughput of the components of a computing system.

Throughput and latency are the two performance metrics considered for the performance model. Throughput is a measure of how much work a system can complete per unit of time, and latency is the overall time from start to finish that is necessary to complete a unit of work. Latency is estimated from a throughput calculation by modifying the graph to have a feedback loop between the sinks and sources (this

feedback loop ensures that only one unit of work is active at a time, and thus latency can be calculated as $\frac{1}{throughput}$).

Formally, a Petri net is defined as a five-tuple: $PN = (P, T, F, W, M_0)$, where

- P is the set of places;
- T is the set of transitions (with $P \cap T = \emptyset$);
- F is the set of arcs $(P \times T) \cup (T \times P)$;
- W is the set of weights for each arc;
- M_0 is the initial marking of each place; each $m \in M_0$ is a non-negative integer.

The original definition of Petri nets does not place any restrictions on the timing of when a transition fires. In particular, transitions may wait an arbitrary amount of time before firing after having become enabled, and there is no synchronization between the firings of different transitions. Different studies of Petri nets have introduced timing semantics into the model in order to model different systems (e.g., timed Petri nets and stochastic Petri nets [17; 65; 107]).

6.2 Compositional Multi-Core Performance Model

I introduce the following two refinements over general Petri nets for the performance model. First, firing semantics are discretized over time steps, and all transitions enabled at a given time step will fire. Second, when nondeterminism is present in the graph (i.e., more than one transition could consume tokens from the same place, but the firing of one transition would preclude that of others), the transitions follow a round-robin priority schedule.

The proposed performance model is expressive enough to capture various aspects of mapping a task graph onto an architecture, including intertask dependencies, different task mappings, communication latency, and buffer sizing. Individual Petri net

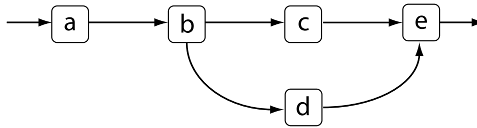


Figure 6.1: Example task graph.

constructs are introduced to capture each of these aspects, and these constructs are composed into a comprehensive Petri net representation of the program. Figures 6.1 through 6.9 step through the construction of an instance of the CMCP model from an example task graph in order to illustrate the combination of these aspects. Fig. 6.1 illustrates a simple example task graph consisting of five computational tasks which are composed with pipelined inter-task communication.

6.2.1 Tasks

A *task* is equivalent to a sequential sub-program; in other words, a sequence of operations that work together to accomplish a particular goal. A task may be hierarchical, comprising several *subtasks*. Each task corresponds to a transition in the CMCP graph (transitions are shown as rectangles, while places are large empty circles, and tokens are small filled-in circles within the places). The execution time, e, of a transition corresponds to the measured or estimated execution time of the task on the target hardware, and when a transition fires, it will not produce tokens until e time steps have passed.

6.2.2 Task Composition

Composition captures the dependencies between tasks and sets of tasks. For example:

- *pipelined composition*, $f \circ g$: there is a data dependency between f and g such that the input consumed by g depends on the output produced by f.

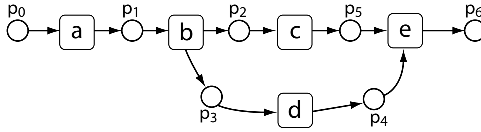


Figure 6.2: Task graph represented as a Petri net.

- *commutative composition*, $f + g$: there are no dependencies between f and g, and f and g can be executed in any order, or concurrently;
- *transactional composition*, $f \oplus g$: f and g can be executed in any order, but they must be executed atomically (i.e., f must complete before g or vice versa). Thus, there is an implied data dependency between f and g since otherwise their order would not matter;
- *functional composition*: dynamic and/or data dependent rules about the ordering of tasks;

The examples in this section, which are based on experiments from previous chapters, primarily employ pipelined and commutative composition. Pipelined communication paths and dependencies are illustrated as directed edges in the task graph, and places with an initial marking of zero are inserted between transitions. Commutative composition involves no dependencies between tasks, and corresponds to a lack of dependency arcs in the task graph.

The following five-tuple expresses the task graph shown in Fig. 6.1 as a Petri net, $PN = (P, T, F, W, M_0)$, where $P = \{p_0,\ p_1,\ p_2,\ p_3,\ p_4,\ p_5,\ p_6\}$, $T = \{a,\ b,\ c,\ d,\ e\}$, $F = \{(p_0, a),\ (a, p_1),\ (p_1, b),\ (b, p_2),\ (b, p_3),\ (p_2, c),\ (c, p_5),\ (p_5, e),\ (p_3, d),\ (d, p_4),\ (p_4, e),$ $(e, p_6)\ \}$, $W = \{1,\ 1,\ 1,\ 1,\ 1,\ 1,\ 1\}$, and $M_0 = \{0,\ 0,\ 0,\ 0,\ 0,\ 0,\ 0\}$. Fig. 6.2 illustrates the full set of places and arcs in the Petri net representation of this example. In the case of feedback loops within the task graph, it is necessary to insert tokens so that the Petri net can make progress. A minimum of one token must be present in

the initial marking of every cycle in the original graph, but more may be inserted depending on the initialization of tasks in the real program.

6.2.3 Architecture

The CMCP model combines a general Petri net representation of an application task graph with an abstraction of the target multi-core architecture. Multi-core architectures are characterized by their (1) cores, (2) communication infrastructure, and (3) memory model.

The CMCP model includes a set of cores C. The experiments in this chapter utilize uniform models for the communication infrastructure and memory model. Equation 6.1 calculates a uniform communication latency between cores as a function of the message size.

$$latency = \alpha + \beta * message\ size \quad (6.1)$$

Profiling of the target architecture determines the α and β factors. The CMCP model assigns each core c_i the same memory capacity μ.

The CMCP model allows for extensions to more complex models of non-uniform communication latency and memory distribution, for example, by modelling non-uniform communication latency as a function of the identities of the sender and receivers as well as the message size. The memory capacity of each core may be adjusted on a per-core basis to reflect uneven memory distribution; e.g., the Cell architecture maps application code and data to the same local memory and thus the size of the application code reduces the available memory to a core.

The resources of a multi-core architecture interact with application properties and dependencies during runtime. The CMCP model captures various interactions including mutual exclusion, data buffering and backpressure constraints by introducing additional transitions and places to the Petri Net which represent the resources and constraints of the architecture.

Formally, the CMCP model extends a general Petri net model to an eight-tuple: $CMCP = (P, T, F, W, M_0, E, C, Map)$, where

- $E : T \longrightarrow \mathbb{N}$ is the set of execution times for each transition;
- $C = \{c_1, c_2, ..., c_N\}$ is a set of N cores;
- $Map : T \longrightarrow C$ is a mapping of each transition to a core.

The remainder of this section addresses different aspects of the program's implementation and how they are captured in the performance model.

6.2.4 Mutual Exclusion

To capture the fact that each core only works on one task at a time, mutual exclusion (mutex) constructs are added for all tasks co-mapped to the same core. A mutex construct consists of a place initialized with one mutex token, such that a task must consume this token in order to fire, and will not return the token until it is done executing. For each core c_i, where the set $\{t{:}map(t) = c_i\}$ has more than one element, an additional place is added to P with arcs to and from all t with $map(t) = c_i$. For example, Fig. 6.3 adds a mutex construct between tasks a and b, and the CMCP model is updated with the following changes: add $C = \{c_1, c_2, c_3, c_4\}$, and $map(a) = map(b) = c_1$, $map(c) = c_2$, $map(d) = c_3$, $map(e) = c_4$, $P = P \cup \{p_7\}$, and $F = F \cup \{(a, p_7), (p_7, a), (b, p_7), (p_7, b)\}$, with the weight of all new arcs set to 1, and the initial marking of the new place $m_0(p_7) = 1$. A round-robin priority scheme is enforced when more than one filter is simultaneously enabled but requires the same mutex token.

6.2.5 Data Buffering in Pipeline Communication

When two neighboring pipelined tasks are co-mapped on the same core, data may be passed between these tasks via in-place buffering (i.e., in the core's local memory)

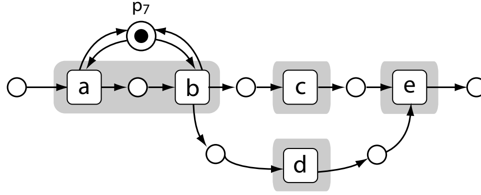


Figure 6.3: Modeling mutual exclusion.

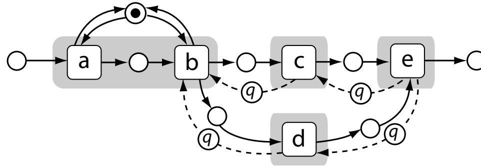


Figure 6.4: Modeling backpressure.

and does not incur additional storage cost. Consequently, there is never backpressure due to buffering between co-mapped neighbors. However, when neighboring tasks are not co-mapped, a data buffer of finite size is maintained between them. Backpressure, corresponding to the available buffer space, is modeled by adding a backward arc/place pair between tasks. The place is initialized with q tokens for a buffer of q size. The CMCP captures data buffering in a pipeline communication by the addition of backpressure arcs. For each pair of transitions t_1 and t_2, if there exists p such that (t_1, p), $(p, t_2) \in F$ and $map(t_1) \neq map(t_2)$, a new place p_{back} is added in the opposite direction: $P = P \cup \{p_{back}\}$, $F = F \cup \{(t_2, p_{back}), (p_{back}, t_2)\}$, the weight of the new arcs is set to q, and $m_0(p_{back}) = q$. The figures of this chapter follow the convention of representing backpressure edges as dashed lines. In Fig. 6.4, four backpressure edges are added among tasks b, c, d, and e. In this example, all backedges have the same number of tokens (q), though uniform buffer sizing is not mandatory in general.

6.2.6 Communication Latency

The cost of communication is incorporated into the performance model with additional transitions, shown as darkened rectangles in Fig. 6.5. These transitions add

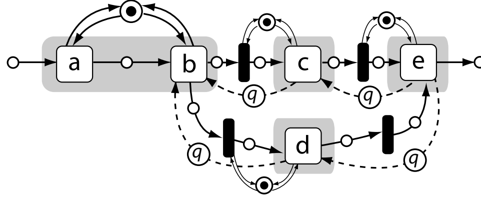


Figure 6.5: Modeling communication overhead.

additional latency based on the size of data being passed and on the execution time of the tasks that follow them. The cost of pipeline communication may be hidden when the data movement is overlapped with computation. This is known as *double buffering* and is a popular technique to optimize the execution of stream programs on multi-core architectures. However, if the execution time of a task is relatively low compared to the latency of communication, the communication overhead may not be hidden. Notice that Fig. 6.5 imposes mutual exclusion constraints on the communication transitions. The latency of these transitions corresponds to the communication overhead which is not hidden through double buffering. Multiple incoming communication streams do typically overlap, such as the two incoming communication transitions of filter e. In this case, it is not necessary to add a mutex loop to the second communication transition. To avoid cluttering the figures in the next pages, mutex loops to communication transitions are not drawn. Notice that the backpressure arcs bypass the communication overhead. This reflects the difference in latency between sending a block of data and a control message. Depending on the hardware platform, the properties of communication latency and how it changes with data size may vary.

Formally, for the same set of transitions considered when adding buffering constraints (i.e., each pair of transitions t_1 and t_2, if there exists p such that (t_1, p), $(p, t_2) \in F$ and $map(t_1) \neq map(t_2)$), a new transition t_{lat} and corresponding place p_{lat} are added between the transitions, thus altering the existing arcs. The CMCP model is updated with these changes: $P = P \cup \{p_{lat}\}$, $T = T \cup \{t_{lat}\}$, $F = F \cup \{(p, t_{lat}),$

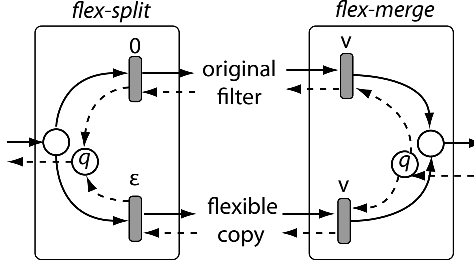


Figure 6.6: Modeling *flex_split* and *flex_merge*.

(t_{lat}, p_{lat}), $(p_{lat}, t_2)\} \cup \{(p, t_2)\}^c$ (i.e., replacing the arc (p, t_2) with arcs through p_{lat} and t_{lat}). W is updated so that all new arcs have the same weight as (t_1, p). The initial marking M_0 for new places is zero: $m_0(p_{lat}) = 0$. The execution time of the communication transitions, $e(t_{lat})$, is set according to the communication latency of the architecture. The new transitions are co-mapped to the same core as t_2 ($map(t_{lat}) = map(t_2)$), and the mutual exclusion constructs are added accordingly (unless t_2 already has incoming arcs from communication latency transitions, like filter e in the example above).

6.2.7 Flexibility

Flexible filters (which are discussed in Chapter 4) are modelled with the *flex_split* and *flex_merge* structures shown in Fig. 6.6 and incorporated into the CMCP graph as shown in Fig. 6.7, which illustrates a mapping where c is flexible, and b and c_{flex} are co-mapped. (For clarity, places with an initial marking of zero are omitted from the rest of the figures in this chapter.) The construction of *flex_split* has two transitions (shown shaded gray in Fig. 6.7), one with an execution time of zero, and one with a small positive execution time, ϵ. The difference enforces a permanent priority of the original copy of c over c_{flex}. The implementation of *flex_merge* buffers incoming data from c and c_{flex} separately (discussed in Section 4.2). The performance model abstracts this into a shared backpressure place, where both tasks may consume tokens

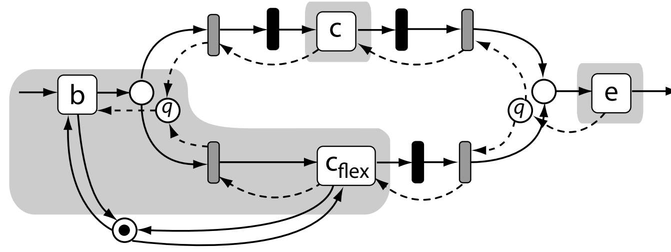


Figure 6.7: Incorporating flexibility into the CMCP model.

from the buffer. The overhead in latency and communication cost of *flex_split* and *flex_merge* is added to the *flex_merge* transitions (labeled v in Fig. 6.6).

In this example, the flexible filter c only has one input and one output channel. To model multiple input channels, the nondeterministic places in the *flex_split* structure are replicated. This results in a synchronization between the input channels which is necessary in the stream program since the same tokens must be matched up in order to produce the same output results regardless of whether c is flexible or not. Like the experimental implementation, the construction of *flex_merge* across multiple output channels is simply a replication of the *flex_merge* structure for each output channel. No synchronization is necessary for the output channels. Fig. 6.8 illustrates the constructs for flexibility across multiple input and output channels.

Fig. 6.9 depicts the overall CMCP model representation of the program from Fig. 6.1, including mutual exclusion, buffering, communication overhead, and flexibility.

6.3 Generating a Task Graph from a Recursive Program

The CMCP model builds on a task graph representation of applications. However, higher-level programming abstractions may not be based on task graphs (i.e., they may not require that the programmer specify an application as a set of tasks).

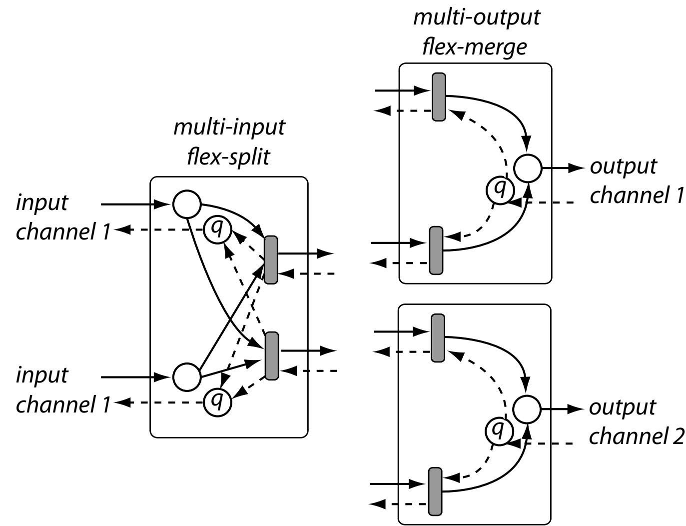


Figure 6.8: Modeling *flex_split* and *flex_merge* with multiple channels.

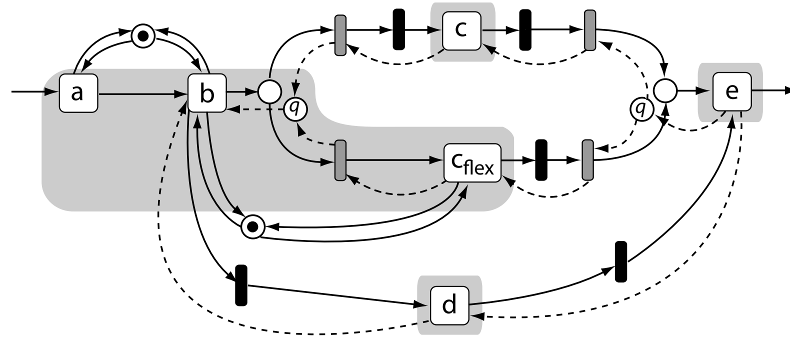


Figure 6.9: Overall Petri net performance model representation of a stream program.

Nonetheless, during runtime the parallel implementation of even a high-level abstracted program partitions the application among the cores and moves data to and from different parts of memory. The runtime expression of a program corresponds to a task graph representation where parts of the program that run on different cores correspond to different tasks, and the movement of data between the memory of two cores corresponds to a dependency between the tasks executing on those cores. Before presenting experiments on the CMCP model, this section details the conversion of

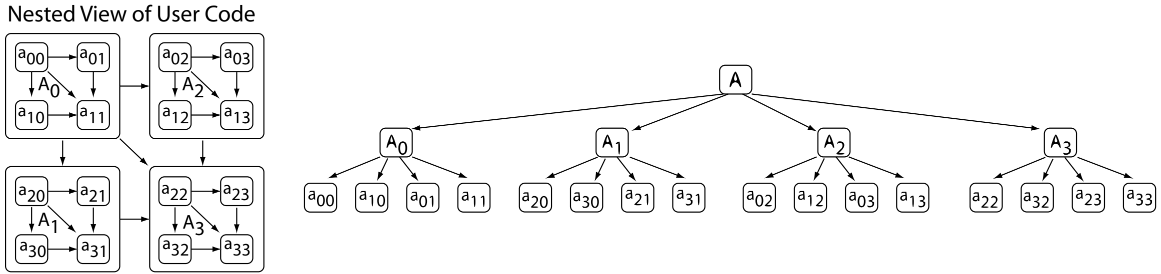


Figure 6.10: Task graph tree of the Smith-Waterman benchmark.

a high-level abstraction into a task graph, in particular, the construction of a task graph from a Huckleberry program.

The task graph from a recursive Huckleberry program is not obtained directly from user code, as is the case for stream programs where filters translate directly to tasks, because the user code does not identify separate tasks. Instead, the task graph is generated from the R-Tree of the program. Fig. 6.10 shows the R-Tree for the Smith-Waterman sequence alignment benchmark. Recall that the Smith-Waterman benchmark works over an array of data with a wave-front dependency pattern, and it is implemented recursively using a quadrant partition pattern (i.e., breaking the array up into quadrants). In the Huckleberry implementation of the Smith-Waterman algorithm, the data of the array is assigned to different cores, and each core will separately traverse the tree shown in Fig. 6.10, executing only the leaf nodes that correspond to data assigned to that core. However, the mapping of tasks to cores impacts the structure of the task graph. The example below illustrates how the dependencies between tasks are altered.

Each core traverses the tree in depth-first order. Thus every core will visit the leafs in the same order: a_{00}, a_{10}, a_{01}, a_{11}, a_{20}, a_{30}, etc. Consider the two mappings shown in Fig. 6.11 for only tasks a_{00}, a_{10}, a_{01} and a_{11}. In the column-stripe task distribution across two cores, a_{00} and a_{10} are assigned to core_0 and a_{01} and a_{11} are assigned to core_1, At runtime, both cores perform a depth-first search of the R-Tree,

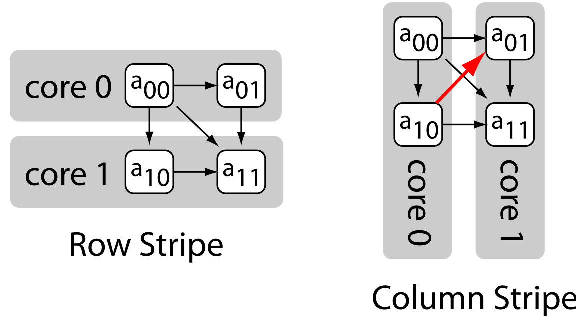


Figure 6.11: Row and column stripe mappings on two cores.

executing tasks for which they are responsible and passing data as necessary to other cores for tasks for which they are not. This is a list of the steps that occur at runtime (for simplicity each task takes one timestep and data swaps are instantaneous):

1. timestep$_0$: core$_0$ executes a_{00}. core$_1$ visits a_{00} and a_{10} in the tree and detects it is not responsible for those tasks, and that it does not have any data to pass. core$_1$ then reaches a_{01} and waits for data produced by a_{00}.

2. timestep$_1$: core$_0$ executes a_{10}.

3. timestep$_2$: core$_0$ visits a_{01} and detects that data must be sent to core$_1$. After receiving this data, core$_1$ may now execute a_{01} while core$_0$ also sends data for a_{11} to core$_1$.

4. timestep$_3$: core$_1$ executes a_{11}.

No speedup is gained in the column-stripe mapping even though a_{10} and a_{01} could be executed in parallel. The row stripe mapping takes advantage of this and can achieve a speedup of $\frac{4}{3} = 1.33$ since core$_1$ is able to execute a_{01} starting in timestep$_1$. Dependencies caused by the recursive call tree are called *spurious dependencies*.

From a performance model perspective the question is not necessarily how to avoid the interactions between the mapping and constraints imposed by the partition pattern, but how to capture them in the model. My approach to detect spurious dependencies is as follows: for each core, a task list is created including tasks assigned to

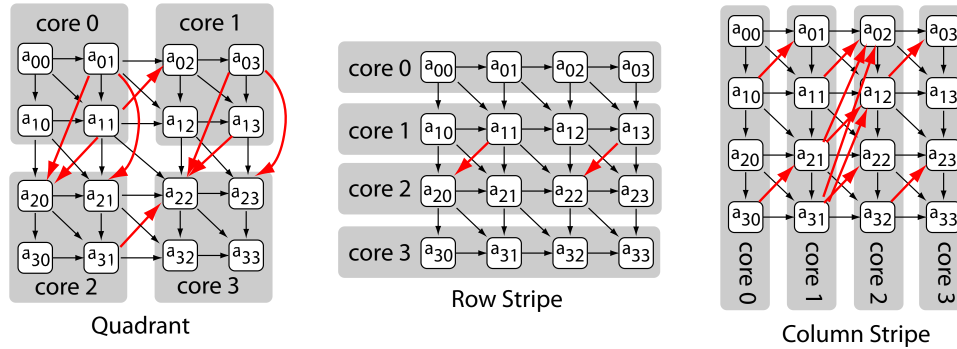


Figure 6.12: Different mappings on four cores.

that core as well as communication arcs for which that core is the source, maintaining the order of tasks that is created by a depth-first search of the tree. For example, in the column stripe distribution, core$_0$ is assigned a task list: { a_{00}, $(a_{00} \rightarrow a_{10})$, a_{10}, $(a_{00} \rightarrow a_{01})$, $(a_{00} \rightarrow a_{11})$, $(a_{10} \rightarrow a_{11})$}. Since the communication events corresponding to arcs with a_{00} as a source occur after a_{10} has been visited in the tree, the list is altered to reflect the latest task executed on core$_0$ before that communication event takes place. These are the dependencies used by the CMCP model. The list is updated as: { a_{00}, $(a_{00} \rightarrow a_{10})$, a_{10}, $(a_{10} \rightarrow a_{01})$, $(a_{10} \rightarrow a_{11})$, $(a_{10} \rightarrow a_{11})$}. This list can be created automatically by traversing the recursive user-input functions (not necessarily on a multi-core platform).

Unfortunately, for the Smith-Waterman application it is not possible to fix this issue in the recursive kernel by adjusting parameters such as the granularity (for example, creating a 16-way divide-and-conquer recursive kernel rather than a 4-way divide-and-conquer algorithm), because no matter how many tasks are explicitly defined, they are visited in sequential order in a depth-first search of the call tree. And more spurious dependencies arise as the system scales to include more cores. Fig. 6.12 shows the spurious dependencies when sixteen tasks are distributed across four cores in three alternative mappings. To mitigate performance losses caused by spurious dependencies, it is important to choose a good mapping (e.g. row striping over column striping in the examples above).

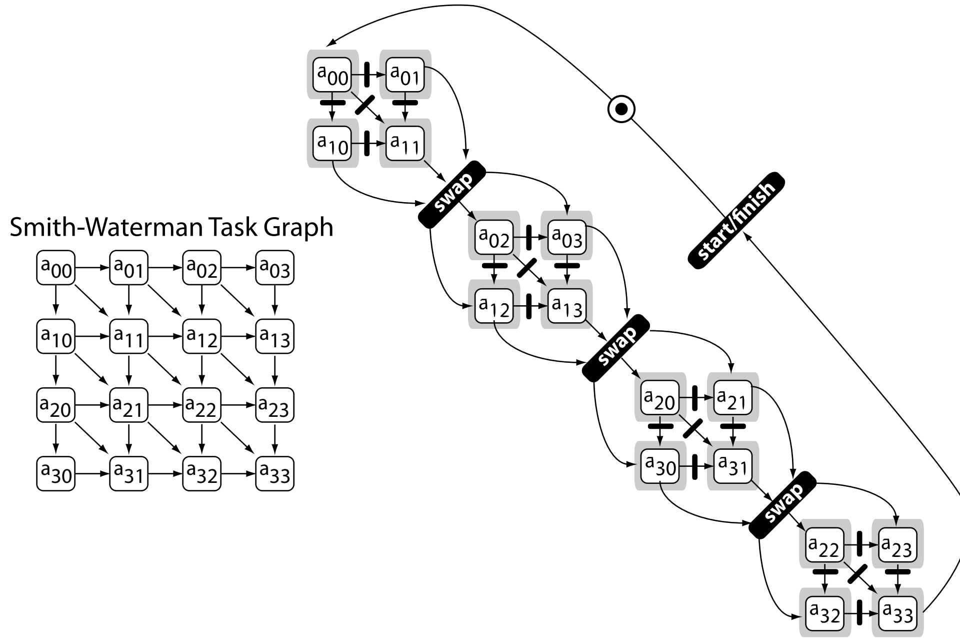


Figure 6.13: Multiple stages of the Smith-Waterman benchmark. The feedback loop ensures that only one data set is active at a time (pipelining the stages is not the goal in this case).

6.3.1 Off-chip Data Swaps

So far, the CMCP model covers the overhead costs of on-chip communication, but not off-chip communication, like the data swap stages described in the discussion of Huckleberry in Chapter 5. Huckleberry programs compute in several stages, so that off-chip communication is synchronized between the cores. Fig. 6.13 illustrates the task graph for the Smith-Waterman benchmark over a four-core architecture where the input data size is too large to fit into the local memory available on chip.

Stages of execution are separated by synchronization transitions that represent the off-chip data swapping and synchronization of the cores. The execution time and granularity of these transitions is set according to the latency and throughput of the platform's off-chip link. Because Huckleberry programs operate on a single data set at

a time rather than pipelining multiple data sets, an additional start/finish transition creates a barrier between data sets.

6.4 Experiments

In this section, I evaluate the proposed CMCP model to understand how closely it approximates actual experiment data collected. The experiments draw from the benchmarks of both the Flexible Filters and Huckleberry projects, as well as synthetic benchmarks generated with the Task-Graphs For Free (TGFF) tool [41] and implemented using the Gedae stream language. The hardware platform for the empirical part of these experiments is the PlayStation3 with a Cell BE processor with six enabled SPE cores.

One advantage of a CMCP model is the ability to explore a wider space of design decisions than can be feasibly tested empirically. A case study of the Smith-Waterman benchmark is also included that tests the interaction of different task mappings with the spurious dependencies created by its recursive partition pattern.

Performance on instances of the CMCP model is estimated using a simulator that tracks the movement of tokens around the graph as transitions become enabled and fire. As described in Sec. 6.2, the CMCP adds new transitions and arcs to the task graphs of applications. Table 6.1 reports the total number of transitions and places in the experiments for the VAR and Smith-Waterman benchmarks; the VAR benchmark is shown with and without flexibility added; and the Smith-Waterman information uses the row-stripe and column-stripe mapping from Fig. 6.12 on 1K sequences which do not require multiple data-swaps.

6.4.1 Mutual Exclusion

Fig. 6.14 illustrates how effectively the CMCP model captures mutual exclusion. The benchmarks tested are synthetic stream graphs generated by TGFF and implemented

Benchmark	(orig) tasks	(orig) arcs	transitions	places
VAR	5	4	9	11
VAR-flexible	5	4	15	21
Smith-Waterman (row-stripe)	17	38	41	87
Smith-Waterman (col-stripe)	17	38	49	111

Table 6.1: CMCP graph size compared to original task graph.

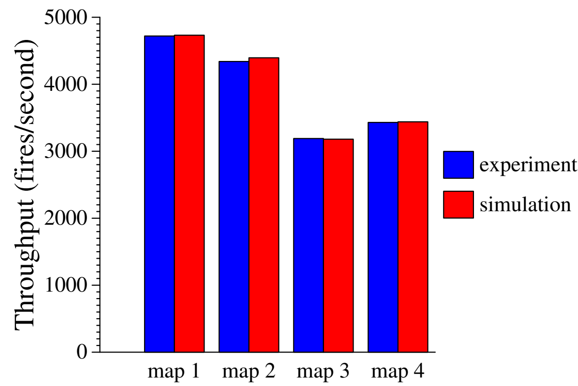


Figure 6.14: Estimated vs. actual throughput testing mutual exclusion.

with Gedae for the Cell processor. The mappings tested are illustrated in Fig. 6.15. TGFF provides relative execution times of the different tasks. For this experiment, they are set large enough so that the communication latency is completely absorbed into double buffering in the communication channels. The communication buffers between the pipelined stream tasks are also set large enough so that backpressure does not impact performance and the model of mutual exclusion is highlighted by the results. For all mappings, the CMCP model's simulated throughput came within 2% of the throughput of the experimental model.

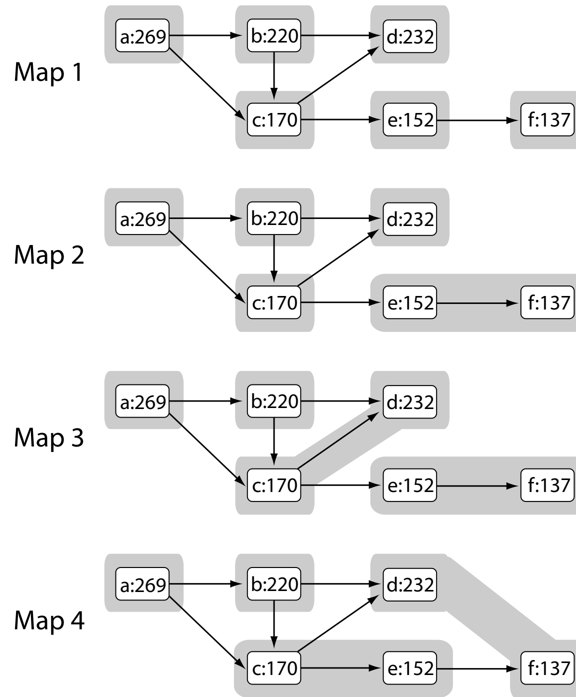


Figure 6.15: Estimated vs. actual throughput testing alternative mapping options.

6.4.2 Communication Latency

The experiments presented in this chapter are all conducted on a Cell BE processor. Based on the Cell BE profiling experiments of Kistler *et al.* [79], the latency of a DMA data transfer between cores is estimated with the following equation:

$$latency(nanoseconds) \quad = \quad 91 + 0.03939 * message\ size\ (bytes) \tag{6.2}$$

The actual latency added by a communication operation in a pipeline communication, accounting for double buffering, may be calculated with Equation 6.3, where the computational operation has execution time t:

$$pipeline\ latency \quad = \quad max(latency(message\ size) - t, 0) \tag{6.3}$$

Equations 6.2 and 6.3 provide a lower bound for the latency of programs running on the Cell written with the Cell SDK. In practice, I observed larger latencies in the

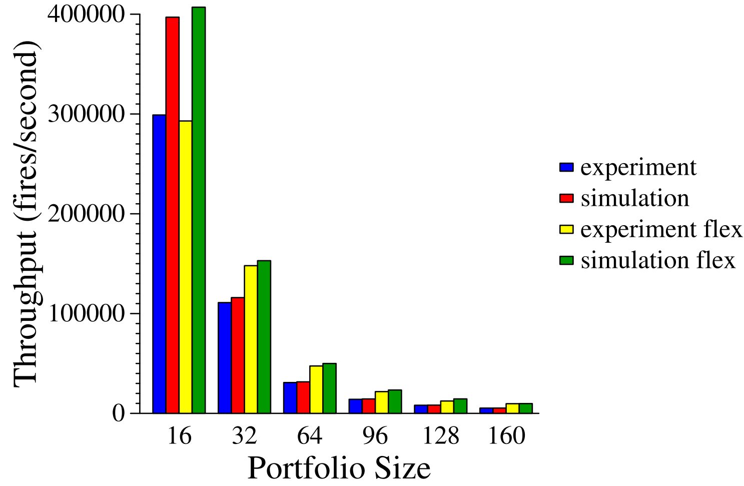


Figure 6.16: Estimated vs. actual throughput for VAR, no communication latency.

benchmarks run on the Cell on top of Gedae, which adds more runtime operations. Latency is better approximated for Gedae applications using Equation 6.4.

$$latency(microseconds) = 56 + 0.15 * message\ size\ (bytes) \tag{6.4}$$

Note that the time units are in microseconds instead of nanoseconds. It is likely that there are some computational operations taking place in the Gedae runtime environment that account for the increase, and there may also be some inefficiencies in the implementation of my benchmarks (e.g. how data is packed into 128-bit chunks). Although it seems like a much larger latency, in most experiments computational tasks dominate, and communication latency does not significantly impact performance.

Profile data collected from the Flexible Filter benchmarks reported in Sec. 4.3.1 populate the CMCP model for the following sets of experiments (with the granularity remaining fixed when flexibility is added in the model).

Fig. 6.16 shows the comparison of the estimated and measured throughput for all of the VAR benchmark's input data sets when communication latency is not included in the model. The discrepancy between the simulator and experiments is largest for the smallest portfolio size which is equal to 16.

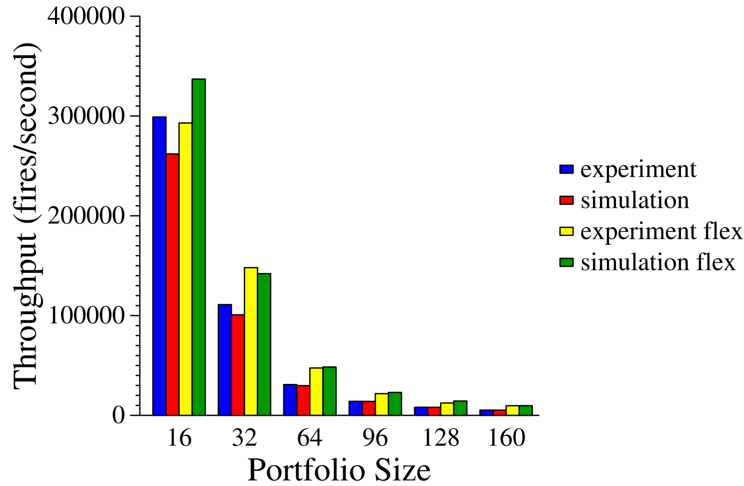


Figure 6.17: Estimated vs. actual throughput for VAR.

Fig. 6.17 plots the same data when communication latency is included in the CMCP model, using the communication latency estimate from Equation 6.4. Most of the simulations are fairly accurate, with greater accuracy in the data points which correspond to larger portfolio sizes. The benchmark requires steps that grow with the square of the portfolio size. Thus, with the smaller portfolio sizes, the cost of communication plays a greater role.

6.4.3 Flexibility

Fig. 6.18 shows how well the simulator predicts trends of speedup when flexibility is added to a stream graph for the CFAR, JPEG and VAR benchmarks. In most cases the simulation accurately captures trends in performance gains when flexibility is added to a benchmark. The CFAR benchmark results demonstrate the largest differences when comparing the simulation and experiments. CFAR presents challenges to simulation since it is a data-dependent benchmark and its filters have a fairly light workload, and thus communication plays a bigger role.

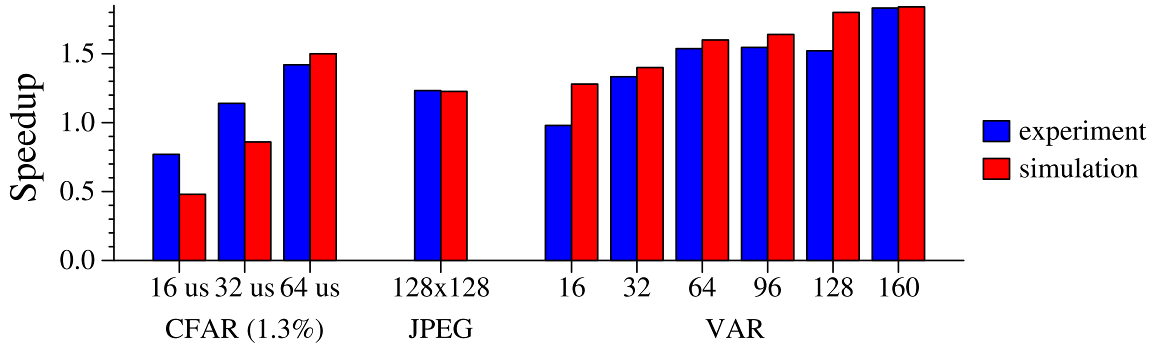


Figure 6.18: Estimated vs. actual speedup across several benchmarks.

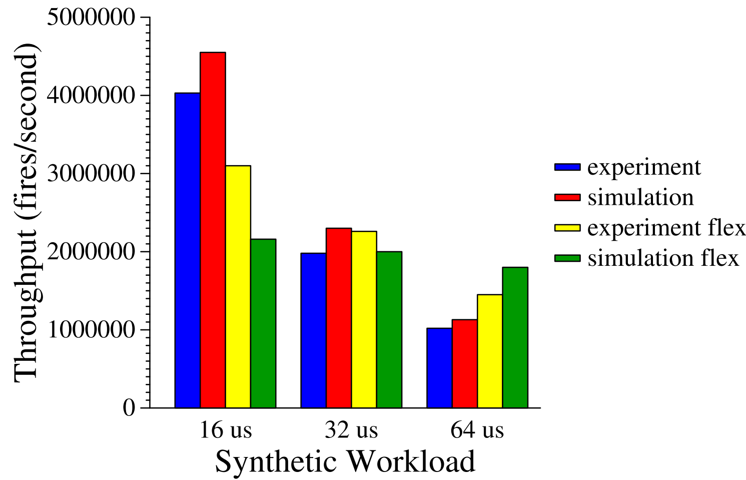


Figure 6.19: Estimated vs. actual throughput for CFAR.

6.4.3.1 Recursive Task Graphs and Data Swaps

Fig. 6.21 compares the latency predicted by the CMCP model with the latency measured experimentally on four cores for the Smith-Waterman algorithm, using a row-stripe data distribution among the cores with the recursive call tree structure from Fig. 6.10. In all of the data sets, the task granularity is a 256x256 character array, which corresponds to 64KB of data, and has a profiled execution time of 4280 ms. Each core is assigned four tasks, such that the data set with sequence length 1024 fits entirely onto the chip, but larger sets require numerous stages of data swapping.

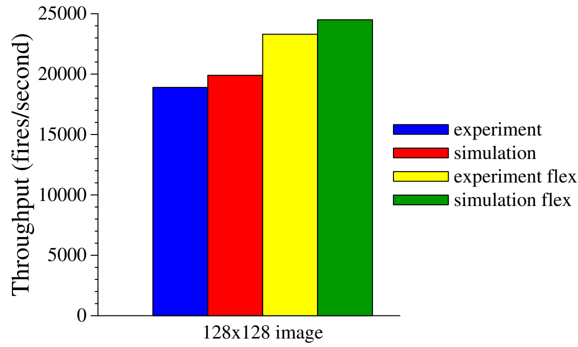


Figure 6.20: Estimated vs. actual throughput for JPEG.

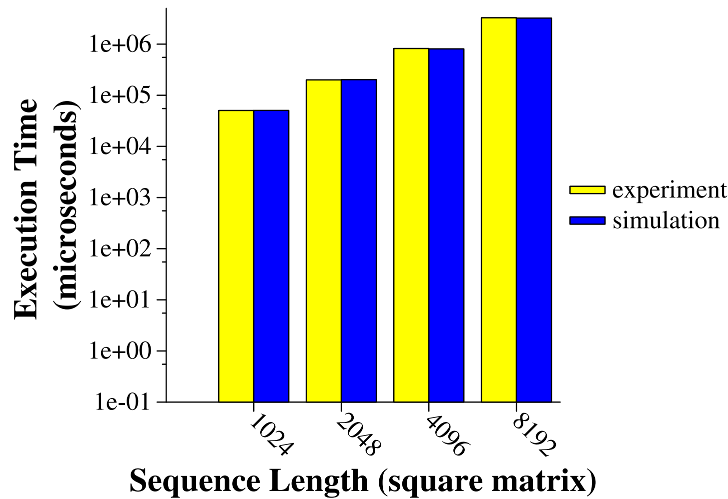


Figure 6.21: Estimated vs. actual latency of the CMCP model for Smith-Waterman on four cores.

The CMCP model comes closest to the experimental results when each off-chip data swap (cumulative over all four cores) is set to cost an additional 1200 ms. The 64KB array is filled in on the cores, and then sent off-chip and saved for computing the final alignment later. Meanwhile, only a single row or column (256 bytes) must be passed between cores (approx. 100 ns, according to Equation 6.3).

When the input size increased beyond 4K, the variability of the results increased significantly because the CMCP model does not take into account the L2 cache be-

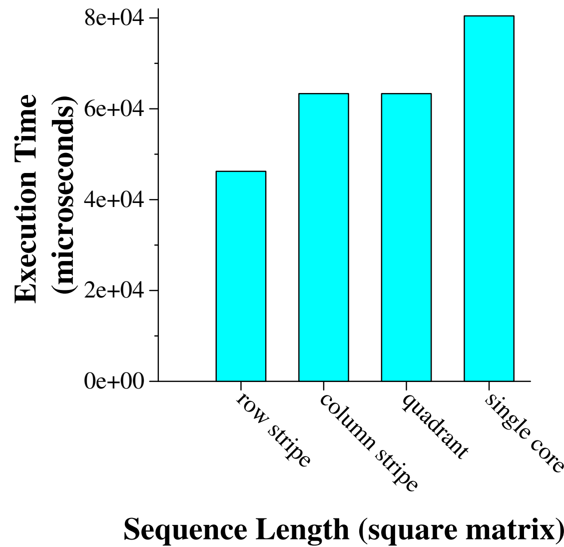


Figure 6.22: Performance comparison of three mappings of a 4x4 task array on four cores.

havior. In order to adjust for this variability and capture the ideal L2 cache behavior, the experimental results listed are best-case times over 100 trials.

Fig. 6.22 explores the performance of three different task mappings for performing a 1k x 1k sequence alignment, taking into account different spurious dependencies for each mapping. The estimate for a single core execution is also included. With the row stripe mapping performing almost 30% faster than the column and quadrant mappings, it is clear that the data and task distribution can have a significant impact on the performance of a Huckleberry program, and the CMCP model can aid in quickly narrowing the space down to the best mappings.

6.5 Composing Different Abstractions

With the examples of stream abstractions and high-level recursive abstractions, I demonstrate that the CMCP model can capture general multi-core applications. By providing a common intermediate form among different levels of programming ab-

stractions, it also offers a common framework for composing different abstractions within the same application. A single application may include some functions which are best parallelized with one abstraction and other functions which are a better suited to other abstractions. Because of Amdahl's law, the best performance won't be achieved if portions of an application which could be parallelized are left sequential. This includes the communication of an application, which if possible should be distributed among the parallel communication resources just as the tasks are distributed among the parallel cores.

6.6 Related Works

Synchronous Dataflow (SDF) represents a special case of dataflow with fixed rates of data production and consumption [82]. An extensive body of literature exists on throughput analysis of SDFs, and more recently in resource trade-offs on multi-processor systems [95; 116]. Static analysis can reveal the Minimum Cycle Mean, which in turn can be used to derive the Maximum Sustainable Throughput of a data flow graph [37; 77]. Previously developed techniques can be leveraged to calculate the throughput in the CMCP model presented in this chapter: when a graph is deterministic, throughput can be calculated with Karp's algorithm in $\bigcirc(|V||E|)$ (used with Marked Graphs, a special case of Petri nets which are deterministic) [38; 77]. When the graph is dynamic or nondeterministic, throughput can be estimated through simulation over a number of time steps.

The CMCP model is based on a combination of previous works, each of which address a subset of the combined view presented here (either the co-mapping of tasks on the same core, or the behavior of FIFO buffers between tasks, but not both). Bonfietti *et al.* modify the SDF to include multi-core mapping by adding additional arcs to the SDF to create a cycle between filters mapped to the same core, such that the cycle has only one token since only one filter at a time can execute on the core [15;

16]. The mutex loops added to the CMCP model also ensure that only one filter at a time will execute on a core, but in contrast allows arbitrary order of execution of the filters and different execution rates of the filters, depending on when they are enabled. Hölzenspies *et al.* add backedges and tokens to the SDF to create buffer space constraints [69].

Compared to other models, the CMCP model provides a holistic view of a parallel application running on a multi-core platform, taking into account the algorithmic aspects of the application as well as the resource constraints of the platform, and interactions between the two at runtime. The CMCP model is also extensible to include performance optimizations and to represent high level programming abstractions, as demonstrated by the inclusion of Flexible Filters and Huckleberry. Finally, it provides a unifying intermediate framework among different levels of programming abstractions.

6.7 Summary/Future Avenues of Research

The CMCP model performs well in matching the absolute throughput and performance trends for different mappings and flexibility assignments. In future work, more research to understand how buffering and granularity impacts performance in the system would be necessary, both experimentally and in the CMCP model. In my experiments, buffer sizes and granularity for each task were selected manually through trial and error in both Gedae programs and Huckleberry programs. An algorithm for determining the best buffer and granularity through the performance model could simplify and automate this design stage.

Chapter 7

Conclusions

The goal of my research has been to discover and understand the parallelism that is present in applications, and how it is captured by parallel programming abstractions and implemented on multi-core architectures. I have made in-depth examinations of the single program multiple data (SPMD), stream, and recursive parallel programming abstractions in the context of numerous benchmarks (bitonic sort, fast Fourier transform, option pricing, Smith-Waterman sequence alignment, JPEG image compression, constant false alarm-rate detection, value-at-risk, dedup image compression, and the data encryption standard). Overall, no abstraction is able to capture all applications well. And this is to be expected, since each application is characterized by its own dependency structure, and each abstraction makes assumptions about the properties of the dependency structure of applications. In some cases, they fit, while in others they do not.

7.1 Contributions

In summary, the contributions of this thesis include the following:

- A **quantitative comparison** of the Cell BE architecture vs. the NVIDIA GeForce 8800 GPU and of the SPMD programming abstraction vs. low-level

platform-specific SDKs. This study highlights the advantages of the Cell on data-centric benchmarks and the advantages of the GeForce 8800 on computation-centric benchmarks. As a high-level abstraction, it is expected that some performance is lost when moving from low-level SDKs to the SPMD abstraction. The gap in performance between hand-optimized code and SPMD code is much smaller for the GeForce 8800 than it is for the Cell across all benchmarks.

- **Flexible filters**, a lightweight, distributed, load-balancing throughput-optimization method for programs written in the stream programming abstraction. Flexible filters can significantly improve the performance of stream programs. They are especially effective in cases where one filter has a relatively high execution latency compared to other filters in the program. Since this approach automatically adapts the data flow to the filter latencies, it can reduce the need to break large filters up by hand. Further, load balancing is determined solely by backpressure signals and can be applied both to systems with static filter latencies and systems with dynamically-varying latencies. I implement flexible filters on top of the Gedae stream language and demonstrate speedup of at least 30% over a non-flexible parallel pipeline stream implementation in the majority of the benchmarks tested.

- **Huckleberry**, a data partitioning abstraction, including the design and implementation of the Huckleberry partition library and code generation tool. Huckleberry provides an intuitive abstraction for parallelizing recursive divide-and-conquer programs, and is evaluated with a suite of benchmarks. Huckleberry demonstrates automatic parallelization of these benchmarks up to sixteen cores, and the flexibility to adapt to different dependency structures in the benchmarks and different available resources.

- A **performance model and simulator**, which unifies data-driven programming abstractions, including SPMD, stream, and recursive abstractions, to-

gether at the task graph level of a parallel implementation. The model is able to capture properties of the application, such as its dependency structure and task execution times together with properties of the architecture, such as the number of cores, communication overhead, and buffering constraints, as well as interactions between the two (e.g., spurious dependencies).

7.2 Future Directions

Many high-level parallel languages and libraries have been discussed that can help the programmer in writing parallel programs and in extracting performance from multi-core platforms. A popular approach to mitigate the complexity of designing parallel algorithms is to select a programming pattern that matches the algorithm, and then use a model that is specialized for that particular pattern [6; 24]. However, many programs would be best represented as a heterogeneous composition of several patterns. In this section, I briefly outline a potential avenue for future work that generalizes Huckleberry's *parallel index (PI) function*, f_{pi}, to enable the composition of heterogeneous parallel functions; that is, parallel functions that do not share the same parallel pattern.

The challenge in composing heterogeneous parallel functions is how to find a common frame of reference. The intermediate task graph representation presented for the CMCP Model (Chapter 6) for parallel programs provides not only a framework for modelling performance, but also for composing different types of parallel functions. Within the CMCP model, all programs are represented as a set of concurrent tasks, together with communication connections between those tasks, leaving the nature of communication open-ended. This discussion of composition starts from the point where all functions, regardless of their parallel model, are represented as tasks in a task graph.

A trivial composition of programming models can always be implemented, where each separate parallel function is completed before the next is started, with data collected at the end of each function and redistributed for the next function. In this way, the functions act as separate applications. However, the trivial solution does not leverage on-chip communication capabilities. In addition, as the bandwidth on and off chip fails to keep up with the memory capacity on chip, repeated scatter/gather steps will become a bottleneck. I believe that in order to design a distributed composition of two or more parallel models, two things are key:

- *data partitioning*: that is, a data set is defined not only by its data structure, but also by the partitioning of the data set across a distributed memory architecture;
- *boundary tasks*: given a parallel function, which is represented as a general task graph, the boundary tasks are those that form a gate between that function and other parallel functions. For example, in an SPMD function, all tasks are boundary tasks, but in a stream function, there is a distinction: only tasks that have channels to the outside environment (i.e., typically source and sink nodes) are boundary tasks, while all other tasks are internal.

The rest of this section outlines at a high level how a general f_{pi} can enable the composition of parallel programming models.

7.2.1 Parallel Index Function

A parallel index function matches tasks and data to an index, which corresponds to a unique core identifier. At any point in time, each core works on some task and some data while the other cores work on other tasks and data. A task, executed on a core, is finite, and between executing tasks, a core must decide which task to do next, and whether it should initiate a change in the data mapping in the system (either by sending data from its local store elsewhere or by moving data into its local store). A *task block* is a set of tasks that can be considered one instance of a parallel

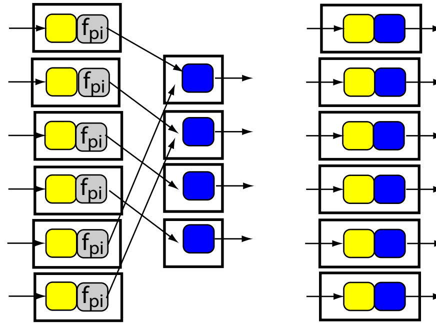


Figure 7.1: Composition with SPMD task blocks.

model (e.g., one *forall* loop from an SPMD program). The boundary tasks of a task block communicate outside of the block through a parallel index function. For the discussion, it is assumed that the composition of two task blocks entails the transition of data between two (possibly overlapping) sets of cores.

Even when two task blocks can be represented with the same model, they may use different parameters and structures within that model. Explicit composition in these cases may be needed just as if the composition were between heterogeneous models. When a boundary task has completed its execution and is holding some data, the parallel index function decides where the data should be sent next, and this is based on information such as data dependent keys, array indexing, etc. Composition with four parallel programming models is considered below.

SPMD Composition. Fig. 7.1 depicts the composition of two SPMD task blocks, shown in yellow and blue. Since every SPMD task pushes its results independently, each task is a boundary task and must be accompanied by an instance of f_{pi}. In the first instance shown in Fig 7.1, the second task block has fewer tasks, and so f_{pi} must manage the redistribution of data. A very simple static implementation might be $f_{pi}(index) = index\%4$. Notice that since data is redirected in this case regardless of the value of data tokens, f_{pi} takes no input parameters other than the core's $index$. Alternatively, f_{pi} may also take into account the data-index in addition

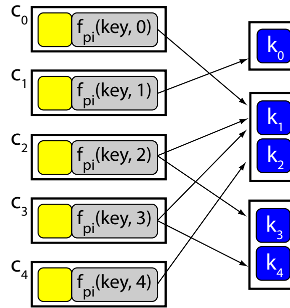


Figure 7.2: Composition with reduction task blocks.

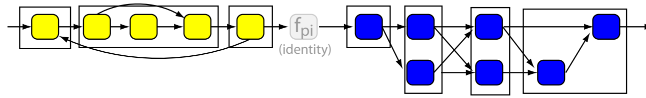


Figure 7.3: Composition with stream task blocks.

to the core index. In the second instance, where both task blocks have the same number of tasks and one of each is mapped to each core, f_{pi} is the identity function and may be omitted.

Reduction Composition. Reduction composition is like the reduce step from MapReduce, and represents data dependent composition [39]. In the composition of two reduction task blocks, f_{pi} requires the *key* value of data tokens as an input parameter, and possibly the current core *index* depending on whether tasks are replicated, and will remap data according to how the reduce tasks are mapped to cores.

Stream Composition. The stream model is perhaps the most composable of parallel models. Since stream programs are constructed by connecting independent tasks via their input and output channels, every subset of a stream program is also a stream program, and two streams can be connected simply through their input and output channels. Because composition in the stream model always uses pipeline composition, f_{pi} is the identity function for stream-to-stream composition.

Recursion Composition. Rather than being driven by the identity of the core, like SPMD composition, or by the task mapping, like reduction composition, recursion

composition is driven by the local data partition assigned to a core. Boundary tasks are identified as described in the bitonic sort example of Section 5.2.2.2.

Heterogeneous Composition. In order to compose heterogeneous programming models, parallel index functions similar to the cases above may be reused. For example, f_{pi} for SPMD to reduction would remain the same as reduction to reduction, and vice versa. One difference in the heterogeneous composition compared to homogeneous composition is how stream functions interface with non-stream functions. Streams tend to have a single or very few boundary tasks compared to other models, which could potentially give rise to bottlenecks, for example, if all of the tasks of an SPMD function simultaneously try to send data to the single source task of a stream function. This case may call for the addition of hierarchical reduction and expansion steps.

Bibliography

[1] V. Agarwal, M. S. Hrishikesh, S. W. Keckler, and D. Burger. Clock rate versus IPC: The end of the road for conventional microarchitectures. In *Proc. Annual Intl. Symposium on Computer Architecture*, pages 248–259, 2000.

[2] V. Agarwal, L.-K. Liu, and D. A. Bader. Financial modeling on the cell broadband engine. In *IEEE Intl. Parallel & Distributed Processing Symposium*, pages 1–12, April 2008.

[3] S. Agrawal, W. Thies, and S. Amarasinghe. Optimizing stream programs using linear state space analysis. In *Int. Conf. on Compilers, Architecture, and Synthesis for Embedded Systems*, September 2005.

[4] T. W. Ainsworth and T. M. Pinkston. Characterizing the Cell EIB on-chip network. *IEEE Micro*, 27(5):6–14, 2007.

[5] 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 *Proc. of the Sixth Workshop on I/O in Parallel and Distributed Systems*, pages 10–22, May 1999.

[6] K. Asanovic, R. Bodik, B. C. Catanzaro, J. 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. Techni-

cal Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, December 2006.

[7] D. H. Bailey. FFTs in external of hierarchical memory. In *Proc. of ACM/IEEE Conf. on Supercomputing*, pages 234–242, November 1989.

[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 *Proc. of ACM/IEEE Conf. on Supercomputing*, pages 1–11, November 2008.

[9] M. A. Bender and M. O. Rabin. Online scheduling of parallel programs on heterogeneous systems with applications to Cilk. *Theory of Computing Systems*, 35(3):289–304, 2002.

[10] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC benchmark suite: Characterization and architectural implications. In *Proc. of the Intl. Conf. on Parallel Architectures and Compilation Techniques*, October 2008.

[11] F. Black and M. Scholes. The pricing of options and corporate liabilities. *Journal of Political Economy*, 81(3):637–654, 1973.

[12] G. E. Blelloch. Programming parallel algorithms. *Communications of the ACM*, 39(3):85–97, March 1996.

[13] N. Bliss. Addressing the multicore trend with automatic parallelization. *Lincoln Laboratory Journal*, 17(1):187–198, 2007.

[14] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. *J. of Parallel and Distributed Computing*, 37(1):55–69, August 1996.

[15] A. Bonfietti, L. Benini, M. Lombardi, and M. Milano. An efficient and complete approach for throughput-maximal SDF allocation and scheduling on multi-core

platforms. In *Proc. of the Conf. on Design, Automation and Test in Europe*, March 2010.

[16] A. Bonfietti, M. Lombardi, M. Milano, and L. Benini. Throughput constraint for synchronous data flow graphs. In *Proc. of the Intl. Conf. on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, pages 26–40, May 2009.

[17] G. Bucci, L. Sassoli, and E. Vicario. Correctness verification and performance analysis of real time systems using stochastic preemptive time petri nets. *IEEE Transactions on Software Engineering*, 31(11):913–927, 2005.

[18] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan. Brook for GPUs: Stream computing on graphics hardware. In *Intl. Conf. on Computer Graphics and Interactive Techniques (SIGGRAPH)*, pages 777–786, August 2004.

[19] D. R. Butenhof. *Programming with POSIX Threads (1st ed.).* Addison-Wesley, 1997.

[20] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. Madden, V. Raman, F. Reiss, and M. Shah. TelegraphCQ: Continuous dataflow processing for an uncertain world. In *Conf. on Innovative Data Systems Research*, pages 668–668, January 2003.

[21] S. Che, J. Li, J. W. Sheaffer, K. Skadron, and J. Lach. Accelerating compute-intensive applications with GPUs and FPGAs. In *Proc. of the Symposium on Application Specific Processors*, pages 101–107, 2008.

[22] J. Chen, M. I. Gordon, W. Thies, M. Zwicker, K. Pulli, and F. Durand. A reconfigurable architecture for load-balanced rendering. In *Proc. of the SIGGRAPH/ EUROGRAPHICS Conf. on Graphics Hardware*, pages 71–80, July 2005.

[23] A. Chow, G. Fossum, and D. Brokenshire. A programming example: Large FFT on the Cell Broadband Engine. Technical report, IBM, May 2005.

[24] M. Cole. Bringing skeletons out of the closet: A pragmatic manifesto for skeletal parallel programming. *Parallel Computing*, 30(3):389–406, March 2004.

[25] R. Collins, C.-H. Li, L. P. Carloni, K. J. Nowka, and E. Schenfeld. An experimental analysis of general purpose computing with commodity data-parallel multicore processors. Technical Report RC25070, IBM T. J. Watson Research Center, October 2010.

[26] R. L. Collins and L. P. Carloni. Topology-based performance analysis and optimization of latency-insensitive systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 27(12):2277–2290, December 2008.

[27] R. L. Collins and L. P. Carloni. Flexible filters: Load balancing through back-pressure for stream programs. In *Proc. of the Intl. Conf. on Embedded Software*, October 2009.

[28] R. L. Collins and L. P. Carloni. Flexible filters for high-performance embedded computing. In *High Performance Embedded Computing Workshop*, September 2010.

[29] R. L. Collins, B. Vellore, and L. P. Carloni. Recursion-driven parallel code generation for multi-core platforms. In *Proc. of the Conf. on Design, Automation and Test in Europe*, pages 190–195, March 2010.

[30] G. Contreras and M. Martonosi. Characterizing and improving the performance of intel threading building blocks. In *IEEE Intl. Symposium on Workload Characterization*, pages 57–66, September 2008.

[31] D. Coppersmith. The data encryption standard (des) and its strength against attacks. *IBM J. of Research and Development*, 38(3):243–250, 1994.

[32] T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson. *Introduction to Algorithms*. McGraw-Hill, 2001.

[33] NVIDIA Corporation. NVIDIA CUDA Toolkit. [Online]. Available: `http://www.nvidia.com/cuda`.

[34] NVIDIA Corporation. CUDA CUFFT library, June 2007. [Online]. Available: `http://developer.download.nvidia.com/compute/cuda/1_1/CUFFT_Library_1.1.pdf`.

[35] NVIDIA Corporation. NVIDIA GeForce 8800 GTX, 2007. [Online]. Available: `http://www.nvidia.com/page/8800_tech_briefs.html`.

[36] Tilera Corporation. Tile64 processor product brief, 2007. [Online]. Available: `http://www.tilera.com/products/processors/TILE64`.

[37] A. Dasdan and R. K. Gupta. Faster maximum and minimum mean cycle algorithms for system performance analysis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 17(10):889–899, October 1998.

[38] A. Dasdan and Rajesh K. Gupta. Faster maximum and minimum mean cycle algorithms for system-perofrmance analysis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 17(10):889–899, 1998.

[39] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *Symposium on Operating System Design and Implementation*, San Francisco, CA, December 2004.

[40] A. A. Desrochers and R. Y. Al-Jaar. *Applications of petri nets in manufacturing systems: modeling, control, and performance analysis*. IEEE Press, 1995.

[41] R.P. Dick, D.L. Rhodes, and W. Wolf. TGFF task graphs for free. In *The Sixth International Workshop on Hardware/Software Co-Design (CODES)*, March 1998.

[42] M. Drake, H. Hoffmann, R. Rabbah, and S. Amarasinghe. Mpeg-2 decoding in a stream programming language. In *IEEE Intl. Parallel & Distributed Processing Symposium*, April 2006.

[43] S. A. Edwards, N. Vasudevan, and O. Tardieu. Programming shared memory multiprocessors with deterministic message-passing concurrency: Compiling SHIM to pthreads. In *Proc. of the Conf. on Design, Automation and Test in Europe*, pages 1498–1503, March 2008.

[44] H. Ehrig, W. Reisig, G. Rozenberg, and H. Weber, editors. *Petri Net Technology for Communication-Based Systems - Advances in Petri Nets*, volume 2472 of *Lecture Notes in Computer Science*, 2003.

[45] M. Farrar. Optimizing smith-waterman for the cell broadband engine. [Online]. Available `http://farrar.michael.googlepages.com/SW-CellBE.pdf`.

[46] M. Farrar. Striped smith–waterman speeds database searches six times over other simd implementations. *Bioinformatics*, 23(2):156–161, 2007.

[47] K. Fatahalian, T. J. Knight, M. Houston, M. Erez, D. R. Horn, L. Leem, J. Y. Park, M. Ren, A. Aiken, W. J. Dally, and P. Hanrahan. Sequoia: Programming the memory hierarchy. In *Proc. of ACM/IEEE Conf. on Supercomputing*, November 2006.

[48] Eric T. Fellheimer. Dynamic load-balancing of StreamIt cluster computations. Master's thesis, Massachusetts Institute of Technology, 2006. Department of Electrical Engineering and Computer Science.

[49] M. Frigo and S. G. Johnson. benchFFT. [Online]. Available: `http://www.fftw.org/benchfft/`.

[50] M. Frigo and S. G. Johnson. The design and implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231, 2005.

[51] M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the Cilk-5 multithreaded language. In *Proc. of the SIGPLAN Conference on Program Language Design and Implementation*, pages 212–223, June 1998.

[52] N. Ganesan, R. D. Chamberlain, and J. Buhler. Accelerating options pricing calculations via parallelization along time-axis on a GPU. In *Proceedings of the First Symposium on Application Acceleration and High Performance Computing*, June 2009.

[53] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, 1979.

[54] Gedae. [Online]. Available: `http://www.gedae.com`.

[55] B. Gedik, H. Andrade, K.-L. Wu, P. S. Yu, and M. Doo. Spade: The system S declarative stream processing engine. In *Proc. of ACM SIGMOD Intl. Conf. on Management of Data*, pages 1123–1134, 2008.

[56] B. Gedik, R. Bordawekar, and P. Yu. CellSort: High performance sorting on the Cell processor. In *Very Large Data Bases Conf.*, September 2007.

[57] R. C. Gonzalez and R. E. Woods. *Digital Image Processing*. Addison-Wesley Longman Publ. Co., Inc., Boston, MA, 2001.

[58] M. I. Gordon, W. Thie, and S. Amarasinghe. Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. *SIGOPS Operating Systems Review*, 40(5):151–162, 2006.

[59] S. Gorlatch. Programming with divide-and-conquer skeletons: A case study of FFT. *J. Supercomput.*, 12(1-2):85–97, Jan./Feb. 1998.

[60] O. Gotoh. An improved algorithm for matching biological sequences. *J. Mol. Biol.*, 162(3):705–708, December 1982.

[61] N. K. Govindaraju, J. Gray, R. Kumar, and D. Manocha. GPUTeraSort: High performance graphics coprocessor sorting for large database management. In *ACM SIGMOD Intl. Conf. on Management of Data*, June 2006.

[62] C. Grozea, Z. Bankovic, and P. Laskov, editors. *FPGA vs. Multi-core CPUs vs. GPUs: Hands-On Experience with a Sorting Application*, volume 6310/2011 of *Lecture Notes in Computer Science*, 2011.

[63] J. Gummaraju and M. Rosenblum. Stream programming on general-purpose processors. In *Proc. of the Intl. Symp. on Microarchitecture (MICRO)*, pages 343–354, November 2005.

[64] M. Gupta, S. Mukhopadhyay, and N. Sinha. Automatic parallelization of recursive procedures. *Intl. J. of Parallel Programming*, 28(6):537–562, 2000.

[65] P. J. Haas and G. S. Shedler. Stochastic petri net representation of discrete event simulations. *IEEE Transactions on Software Engineering*, 15(4):381–393, 1989.

[66] T. R. Halfhill. Parallel processing with CUDA. *Microprocessor Report*, January 2008.

[67] R. Haney, T. Meuse, J. Kepner, and J. Lebak. The HPEC challenge benchmark suite. In *The High-Performance Embedded Computing Workshop*, September 2005.

[68] R. Hempel. The MPI standard for message passing. In *High–Performance Computing and Networking*, volume 797 of *Lecture Notes in Computer Science*, pages 247–252, 1994.

[69] P. K. F. Hölzenspies, G. J. M. Smit, and J. Kuper. Mapping streaming applications on a reconfigurable mpsoc platform at run-time. In *Proceedings of the International Symposium on System-on-Chip (SoC 2007), Tampere, Finland*, pages 74–77, November 2007.

[70] L. Huston, A. Nizhner, P. Pillai, R. Sukthankar, P. Steenkiste, and J. Zhang. Dynamic load balancing for distributed search. In *Proc. of the Intl. Symposium on High Performance Distributed Computing*, pages 157–166, July 2005.

[71] C. R. Johns and D. A. Brokenshire. Introduction to the Cell broadband engine architecture. *IBM J. Res. Develop.*, 51(5):521–528, September 2007.

[72] D. Jones. Decimation-in-time (DIT) Radix-2 FFT, September 2006. [Online]. Available `http://cnx.org/content/m12016/1.7/`.

[73] G. Juanole, B. Algayres, and J. Dufau. On communication protocol modelling and design. In *Advances in Petri Nets 1984*, volume 188 of *Lecture Notes in Computer Science*, pages 267–287, 1985.

[74] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy. Introduction to the Cell multiprocessor. *IBM J. Res. Develop.*, 49(4-5):589–604, September 2005.

[75] P. Kakulavarapu, O. Maquelin, J. N. Amaral, and G. R. Gao. Dynamic load balancers for a multithreaded multiprocessor system. *Parallel Processing Letters*, 11(1):169–184, 2001.

[76] U. J. Kapasi, S. Rixner, W. J. Dally, B. Khailany, J. H. Ahn, P. Mattson, and J. D. Owens. Programmable stream processors. *IEEE Computer*, 36(8):54–62, August 2003.

[77] R. M. Karp. A characterization of the minimum cycle mean in a digraph. *Discrete Mathematics, Discrete Math*, 23(3):309–311, September 1978.

[78] Khronos OpenCL Working Group. *The OpenCL Specification, Version: 1.1, Document Revision: 36*, September 2010.

[79] M. Kistler, M. Perrone, and F. Petrini. Cell multiprocessor communication network: Built for speed. *IEEE Micro*, 26(3):10–23, 2006.

[80] T. J. Knight, J. Y. Park, M. Ren, M. Houston, M. Erez, K. Fatahalian, A. Aiken, W. J. Dally, and P. Hanrahan. Compilation for explicitly managed memory hierarchies. In *Symposium on Principles and Practice of Parallel Programming*, pages 226–236. ACM, March 2007.

[81] M. Kudlur and S. Mahlke. Orchestrating the execution of stream programs on multicore platforms. *ACM SIGPLAN Notices*, 43(6):114–124, 2008.

[82] E.A. Lee and D.G. Messerschmitt. Synchronous data flow. *IEEE*, 75(9):1235–1245, September 1987.

[83] Y. Liu, D. Maskell, and B. Schmidt. CUDASW++: Optimizing Smith-Waterman sequence database searches for cuda-enabled graphics processing units. *BMC Research Notes*, 2(1), 2009.

[84] D. Luebke and G. Humphreys. How GPUs Work. *IEEE Computer*, 40(2):96–100, February 2007.

[85] E. Luttmann, D. L. Ensign, V. Vaidyanathan, M. Houston, N. Rimon, J. Oland, G. Jayachandran, M. Friedrichs, and V. S. Pande. Accelerating molecular dy-

namic simulation on the Cell processor and Playstation 3. *J. of Computational Chemistry*, 30:268–274, 2009.

[86] S. Manavski and G. Valle. CUDA compatible GPU cards as efficient hardware accelerators for Smith-Waterman sequence alignment. *BMC Bioinformatics*, 9(Suppl 2), 2008.

[87] M. Matsumoto and T. Nishimura. Mersenne Twister: A 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Transactions on Modeling and Computer Simulation*, 8(1):3–30, 1998.

[88] T. Mattson, B. Sanders, and B. Massingill. *Patterns for parallel programming.* Addison-Wesley Professional, 2004.

[89] M. McCool, S. Du Toit, T. Popa, B. Chan, and K. Moule. Shader algebra. *ACM Transactions on Graphics*, 23(3):787–795, 2004.

[90] M. D. McCool. Data-parallel programming on the Cell BE and the GPU using the RapidMind development platform. In *GSPx Multicore Applications Conf.*, October 2006.

[91] R. C. Merton. Theory of rational option pricing. *Bell Journal of Economics and Management Science*, 4(1):141–183, 1973.

[92] Intel Microarchitecture. Intel core i7-800 processor series and the intel core i5-700 processor series based on intel microarchitecture (nehalem). [Online]. Available `http://www.intel.com/products/processor/corei7/index.htm`.

[93] Sun Microsystems. Ultrasparc architecture 2005. [Online]. Available: `http://www.opensparc.net`.

[94] J. Misra. Powerlist: A structure for parallel recursion. *ACM Trans. Program. Lang. Syst.*, 16(6):1737–1767, November 1994.

[95] O. Moreira, J-D. Mol, M. Bekooij, and J. van Meerbergen. Multiprocessor resource allocation for hard-real-time streaming with a dynamic job-mix. In *Proc. of the IEEE Real Time on Embedded Technology and Applications Symposium*, pages 332–341, 2005.

[96] M. S. Müller, M. van Waveren, R. Lieberman, B. Whitney, H. Saito, K. Kumaran, J. Baron, W. C. Brantley, C. Parrott, T. Elken, H. Feng, and C. Ponder. SPEC MPI2007 - An application benchmark suite for parallel systems using MPI. *Concurrency and Computation: Practice and Experience*, 22(2):191–205, 2010.

[97] T. Murata. Petri Nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, April 1989.

[98] A. K. Nanda, J. R. Moulic., R. E. Hanson, G. Goldrian, M. N. Day, D. B. D'Arnora, and S. Kesavarapu. Cell/B.E. blades: Building blocks for scalable, real-time, interactive, and digital media servers. *IBM J. of Research and Development*, 51(5):573–582, September 2007.

[99] L. M. Novak, G. J. Owirka, W. S. Brower, and A. L. Weaver. The automatic target-recognition system in SAIP. *The Lincoln Laboratory Journal*, 10(2):187–202, 1997.

[100] M. Ohara, H. Inoue, Y. Sohda, H. Komatsu, and T. Nakatani. MPI microtask for programming the Cell broadband engine processor. *Online Game Technology*, 45(1):85–102, March 2006.

[101] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. E. Lefohn, and T. J. Purcell. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, 26(1):80–113, March 2007.

[102] W. R. Pearson and D. J. Lipman. Improved tools for biological sequence comparison. *Proc. of the National Academy of Sciences of the United States of America*, 85(8):2444–2448, April 1988.

[103] Carl A. Petri. *Kommunikation mit Automaten ("communication with automata")*. PhD thesis, Darmstadt University of Technology, 1962.

[104] D. Pham, S. Asano, M. Bolliger, M. N. Day, H. P. Hofstee, C. Johns, J. Kahle, A. Kameyama, J. Keaty, Y. Masubuchi, M. Riley, D. Stasiak, M. Suzuoki, M. Wang, J. Warnock, S. Weitzel, D. Wendel, T. Yamazaki, and K. Yazawa. The design and implementation of a first-generation CELL processor. In *ISSCC*, pages 184–185, February 2005.

[105] Timothy Mark Pinkston and Jeonghee Shin. Trends toward on-chip networked microsystems. *Intl. Journal of High Performance Computing and Networking*, 3(1):3–18, 2005.

[106] F. Pratas, P. Trancoso, A. Stamatakis, and L. Sousa. Fine-grain parallelism using multi-core, Cell/BE, and GPU systems: Accelerating the phylogenetic likelihood function. In *International Conference on Parallel Processing*, pages 9–17, September 2009.

[107] Chander Ramchandani. *Analysis Of Asynchronous Concurrent Systems by Timed Petri Nets*. PhD thesis, Massachusetts Institute of Technology, 1974.

[108] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis. Evaluating MapReduce for multi-core and multiprocessor systems. In *Proc. of the Symposium on High Performance Computer Architecture*, February 2007.

[109] R. Rugina and M. Rinard. Automatic parallelization of divide and conquer algorithms. In *Symposium on Principles and Practice of Parallel Programming*, pages 72–83, May 1999.

[110] K. Sankaralingam, R. Nagarajan, H. Liu, C. Kim, J. Huh, D. Burger, S. W. Keckler, and C. R. Moore. Exploiting ILP, TLP, and DLP with the polymorphous TRIPS architecture. *SIGARCH Computer Architecture News*, 31(2):422–433, 2003.

[111] R. R. Schaller. Moore's law: Past, present and future. *IEEE Spectrum*, 34(6):52–59, June 1997.

[112] M. Segal and K. Akeley. The OpenGL graphics system: A specification (version 4.0 (core profile)), March 2010.

[113] M. A. Shah, J. M. Hellerstein, S. Chandrasekaran, and M. J. Franklin. Flux: An adaptive partitioning operator for continuous query systems. In *Intl. Conf. on Data Engineering*, pages 25–36, March 2003.

[114] D. E. Shaw, M. M. Deneroff, R. O. Dror, J. S. Kuskin, R. H. Larson, J. K. Salmon, C. Young, B. Batson, K. J. Bowers, J. C. Chao, M. P. Eastwood, J. Gagliardo, J. P. Grossman, C. R. Ho, D. J. Ierardi, I. Kolossváry, J. L. Klepeis, T. Layman, C. McLeavey, M. A. Moraes, R. Mueller, E. C. Priest, Y. Shan, J. Spengler, M. Theobald, B. Towles, and S. C. Wang. Anton, a special-purpose machine for molecular dynamics simulation. *Communications of the ACM*, 51(7):91–97, July 2008.

[115] T. F. Smith and M. S. Waterman. Identification of common molecular subsequences. *J. Molecular Biology*, 147(1):195–197, March 1981.

[116] S. Stuijk, M. Geilen, and T. Basten. Exploring trade-offs in buffer requirements and throughput constraints for synchronous dataflow graphs. In *Proc. of the Design Automation Conf.*, pages 899–904, 2006.

[117] S. Swanson, A. Schwerin, M. Mercaldi, A. Petersen, A. Putnam, K. Michelson, M. Oskin, and S. J. Eggers. The wavescalar architecture. *ACM Trans. Comput. Syst.*, 25(2):4, 2007.

[118] Mercury Computer Systems. 25u/42u dual cell-based blade 2 system. [Online]. Available: `http://www.mc.com/microsites/cell`.

[119] A. Szalkowski, C. Ledergerber, P. Kraehenbuehl, and C. Dessimoz. SWPS3 - Fast multi-threaded vectorized Smith-Waterman for IBM Cell/B.E. and x86/sse2. *BMC Research Notes*, 1(1), 2008.

[120] M. B. Taylor, W. Lee, J. Miller, D. Wentzlaff, I. Bratt, B. Greenwald, H. Hoffmann, P. Johnson, J. Kim, J. Psota, A. Saraf, N. Shnidman, V. Strumpen, M. Frank, S. Amarasinghe, and A. Agarwal. Evaluation of the Raw microprocessor: An exposed-wire-delay architecture for ILP and streams. In *Proc. Annual Intl. Symposium on Computer Architecture*, 2004.

[121] W. Thies, M. Karczmarek, M. Gordon, D. Maze, J. Wong, H. Hoffmann, M. Brown, and S. Amarasinghe. StreamIt: A compiler for streaming applications. Technical report, MIT-LCS Technical Memo TM-622, Cambridge, MA, December 2001.

[122] W. Thies, M. Karczmarek, J. Sermulins, R. Rabbah, and S. Amarasinghe. Teleport messaging for distributed stream programs. In *Symposium on Principles and Practice of Parallel Programming*, June 2005.

[123] D. B. Thomas, L. Howes, and W. Luk. A comparison of CPUs, GPUs, FPGAs, and massively parallel processor arrays for random number generation. In *Proceeding of the ACM/SIGDA international symposium on Field programmable gate arrays*, pages 63–72, 2009.

[124] C. H. van Berkel. Multi-core for mobile phones. In *Proc. of the Conf. on Design, Automation and Test in Europe*, pages 1260 –1265, April 2009.

[125] N. Vasudevan and S. A. Edwards. Celling SHIM: Compiling deterministic concurrency to a heterogeneous multicore. In *Proc. of the ACM Symposium on Applied Computing*, pages 1626–1631, March 2009.

[126] C. Wynnyk and M. Magdon-Ismail. Pricing the american option using reconfigurable hardware. In *Intl. Conf. on Computational Science and Engineering*, pages 532 – 536, August 2009.

[127] Y. Xing, S. Zdonik, and J.-H. Hwang. Dynamic load distribution in the Borealis stream processor. In *Intl. Conf. on Data Engineering*, pages 791–802, April 2005.