TIDEFLOW: A DATAFLOW-INSPIRED EXECUTION MODEL FOR HIGH PERFORMANCE COMPUTING PROGRAMS

by

Daniel A. Orozco

A dissertation submitted to the Faculty of the University of Delaware in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Electrical and Computer Engineering

Spring 2012

TIDEFLOW: A DATAFLOW-INSPIRED EXECUTION MODEL FOR HIGH PERFORMANCE COMPUTING PROGRAMS

by

Daniel A. Orozco

Approved: _____
Kenneth E. Barner, Ph.D.
Chair of the Department of Electrical and Computer Engineering

Approved: _____
Babatunde A Ogunnaike, Ph.D.
Interim Dean of the College of Engineering

Approved: _____
Charles G. Riordan, Ph.D.
Vice Provost for Graduate and Professional Education

I certify that I have read this dissertation and that in my opinion it meets the academic and professional standard required by the University as a dissertation for the degree of Doctor of Philosophy.

Signed: _____

Guang R. Gao, Ph.D.
Professor in charge of dissertation

I certify that I have read this dissertation and that in my opinion it meets the academic and professional standard required by the University as a dissertation for the degree of Doctor of Philosophy.

Signed: _____

Xiaoming Li, Ph.D.
Member of dissertation committee

I certify that I have read this dissertation and that in my opinion it meets the academic and professional standard required by the University as a dissertation for the degree of Doctor of Philosophy.

Signed: _____

Chengmo Yang, Ph.D.
Member of dissertation committee

I certify that I have read this dissertation and that in my opinion it meets the academic and professional standard required by the University as a dissertation for the degree of Doctor of Philosophy.

Signed: _____

Michela Taufer, Ph.D.
Member of dissertation committee

ACKNOWLEDGEMENTS

To my parents, because they showed me that there was no limit to what I could achieve.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

Traditional programming, execution and optimization techniques have been shown to be inadequate to exploit the features of computer processors with many cores.

In particular, previous research shows that traditional paradigms are insufficient to harness the opportunities of manycore processors: (1) traditional execution models do not provide constructs rich enough to express parallel programs, (2) traditional analysis tools allow little insight into the performance of parallel programs and (3) traditional programming and execution tools only offer awkward ways to execute parallel programs.

This thesis addresses those problems with the introduction of TIDeFlow, a parallel execution model aimed at efficient execution and development of High Performance Computing (HPC) programs in manycore processors.

The following are the main contributions of this thesis:

1. The formulation of a parallel execution model that is able to exploit the features present in manycore processors.
2. The development of several highly scalable algorithms and a technique to analyze their throughput.
3. The implementation of the TIDeFlow toolchain, including a programming model and a distributed runtime system.

Chapter 1

INTRODUCTION

The field of computing architecture has recently experienced a radical shift from serial execution to parallel execution. This change from a serial to a parallel paradigm in computing has created numerous problems that make difficult the development and execution of High Performance Computing (HPC) programs.

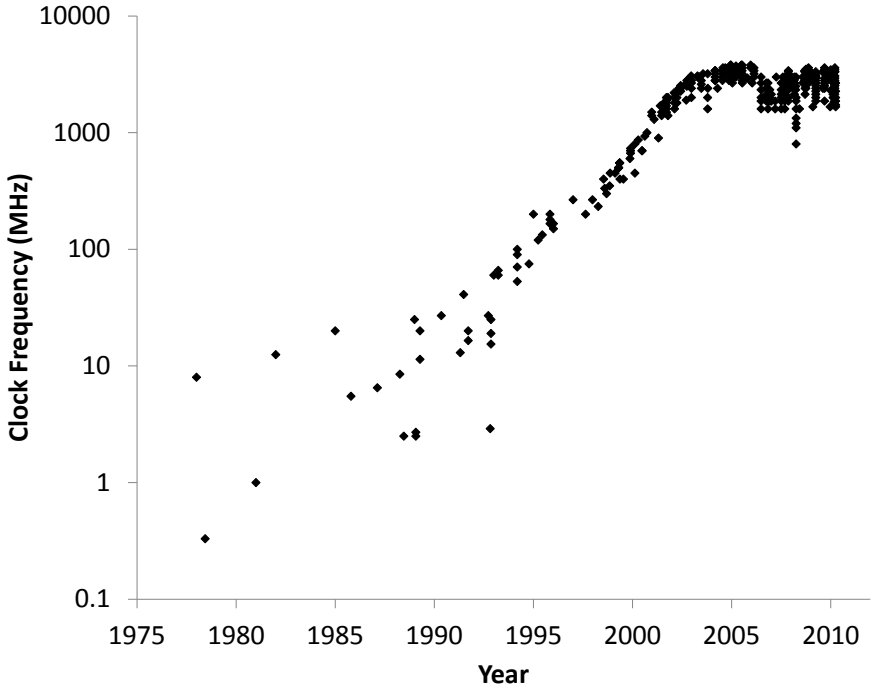
Serial programming and execution were predominant during the *Free Lunch* era. During that time–roughly 1970 to 2005–processor speed increased exponentially (Figure 1.1), which in turn translated into exponential increase of the performance of serial programs.

After 2005, however, processor chip manufacturers were forced to stop the exponential increase in the serial performance of processors due to production problems, power consumption, and cost, among other issues.

The multicore era of computing started when manufacturers placed several processors in a single chip in an effort to continue increasing the performance of processors. The multicore era was possible due to the continuing validity of Moore's Law [73], which granted the required number of transistors to build several processor cores per chip.

The possibilities of multicores were exploited, resulting in many commercial products such as Intel's multicore processors [95] nVidia's Graphical Processor Units [78], Sun's UltraSPARC T2 processor [99], IBM Cyclops-64 [35], Tilera's Tile processors [114] and many others. Of particular importance are the processors composed of many simple cores, such as Cyclops-64. These processors, known as *manycore* processors, rely on parallelism, rather than on individual processor speed, to obtain their performance.

The traditional computing paradigms, having been honed for serial processors and serial programs, did not offer adequate tools to exploit the opportunities offered

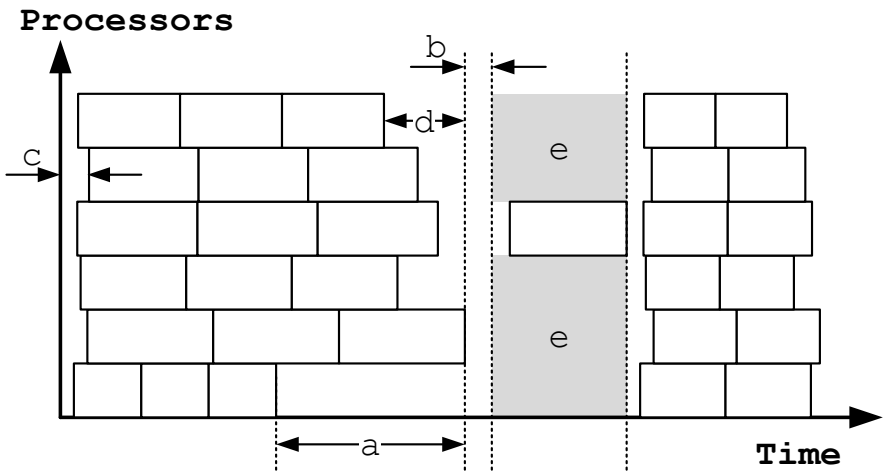


During the "Free lunch era", the speed of processors increased exponentially. The free lunch ended around 2005, when physical, architectural and practical constraints limited the performance of processors an average of 3GHz. Data from [69].

Figure 1.1: Improvement of processor speed over time.

by manycore processors. The following is a partial list of the problems of traditional paradigms:

- Traditional programming paradigms are a good choice to represent arbitrary programs, but they have not been optimized for *High Performance Computing* (HPC) programs, the kind of programs that are usually found in manycore processors.
- The programming tools were designed for serial programming, and they are rudimentary and inefficient to express parallel programs.
- Traditional programming models only offer awkward ways to express pipelining during execution.
- Traditional tools for analysis of parallel programs such as Amdahl's law [7] or the nonblocking properties [57] do not take into account the limitations of the processor architecture, and ultimately, the algorithms they produce fail to support fine-grained execution.

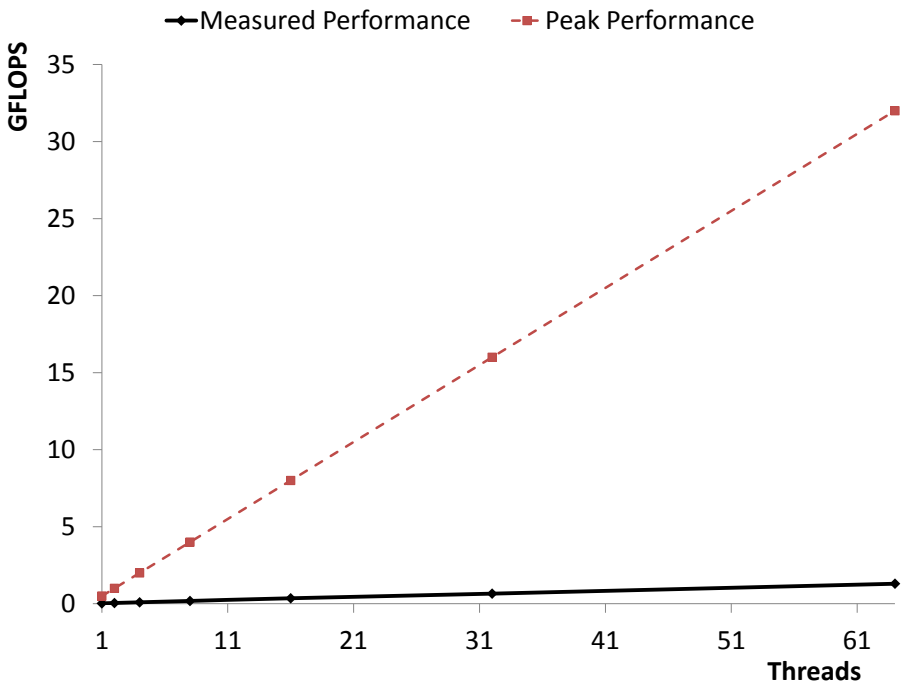


The traditional barrier/fork-join approaches are not very well suited for computation in manycore processors. The execution trace of the figure shows the wasted time due to the intrinsic limitations of traditional approaches: a) Time wasted due to poor scheduling, b) time wasted due to the overhead of global synchronization operations, c) time wasted due to thread management, d) time wasted due to the concept of a barrier itself and e) time wasted due to the poor parallelism of the fork-join model.

Figure 1.2: Sources of overhead in traditional fork-join approaches.

- Traditional system software is slow and centralized.
- Traditional static scheduling approaches fail to produce good results in many-core processors with fine-grained tasks [47], even in the favorable case of regular applications.
- Traditional execution models do not address the need for resilient or energy-efficient execution, which will be paramount to future parallel systems.

The comprehensive case study of matrix multiplication, conducted by Hu [59] and Garcia [50] provides excellent insight into the shortcomings of traditional programming in manycore processors. The example of matrix multiplication is excellent because (1) it is easy to understand, (2) a significant amount of work for matrix multiplication in manycores has been conducted [59, 50, 46, 47] and (3) the work in matrix multiplication put in evidence the limitations of of traditional programming and execution techniques when they are used in manycore architectures.



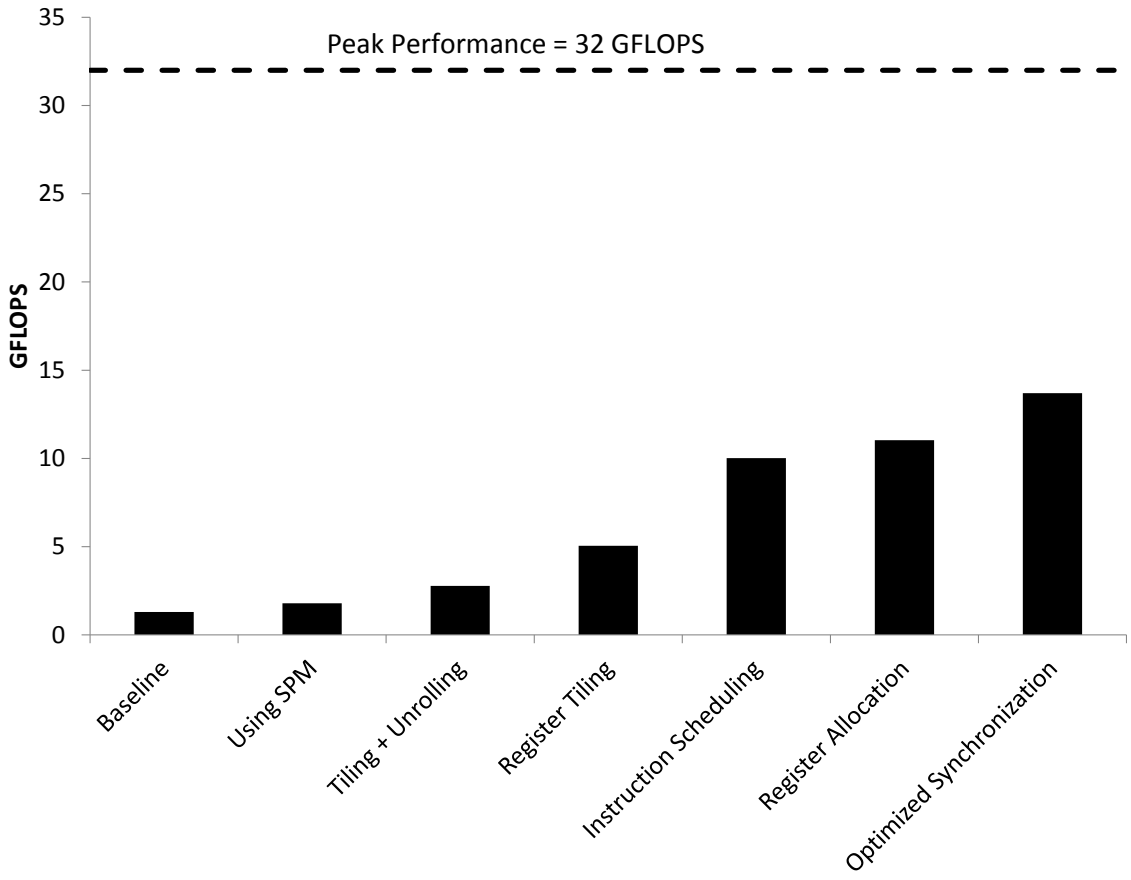
The figure shows the scalability and performance of Hu's implementation of matrix multiply for Cyclops-64 [59]. The performance obtained is very poor even though the matrices were already in on-chip memory.

Figure 1.3: Performance of a first implementation of matrix multiply for Cyclops-64.

The first attempt to develop a matrix multiplication program for Cyclops-64 was made by Hu et al. [59]. Hu's implementation used C and assembly language to write the program, TNT threads [31] (a library for threading similar to POSIX threads [20]) for thread management and traditional synchronization operations (such as barriers and mutexes) to manage synchronization.

The results of Hu's first attempts (Figure 1.3) were a disappointing 1.3 GFLOPS (out of a maximum 32 GFLOPS) using 64 threads. Even after carefully employing traditional static optimizations (Figure 1.4) he was only able to achieve 13.9 GFLOPS.

The poor performance of Hu's implementation gave important insight into the problem. Hu's efforts resulted in very poor performance despite a significant amount of work that included the use of assembly language for critical pieces of code and the



The use of traditional optimization techniques is not enough to achieve close-to-peak performance in parallel programs in manycore processors. Even though the optimizations were applied skilfully, the performance still remained less than half of the theoretical peak.

Figure 1.4: Effect of using traditional optimization techniques on manycore processors.

use of most traditional optimization techniques (Figure 1.4).

The shortcomings of Hu's approach were not due to lack of skill, but rather to the very nature of the tools available to him. His approach was hindered by the intrinsic limitations of the techniques used:

- Barriers, the main synchronization operation used, ensured that all threads worked at the *slowest* rate among them. Mutexes, the other method of synchronization used, did not allow good parallelism among threads.
- The thread-management library used (TNT [31]) followed a centralized approach and was ultimately slow when handling 160 threads.
- There was no simple way to describe a good strategy for task pipelining, causing numerous stalls during the execution.
- There was no way to schedule the computation dynamically, following the needs of the program, ultimately causing delays in the computation of the critical path.

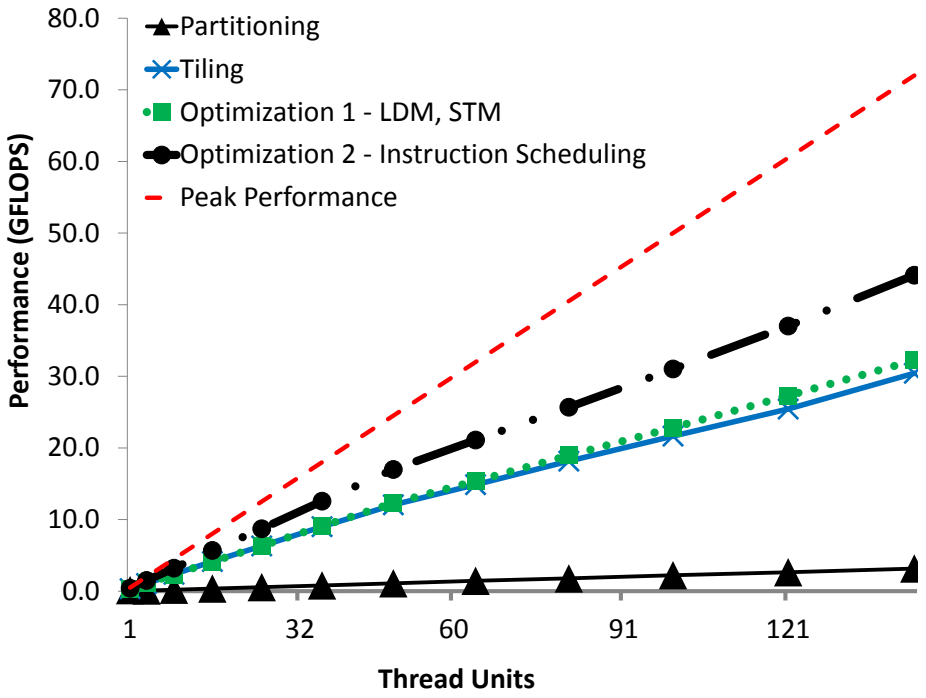
The work on matrix multiplication was continued by Garcia and was presented at the prestigious EuroPar conference [50]. Garcia strived to obtain better results and addressed some of the issues faced by Hu. In particular, Garcia improved Hu's implementation with architecture-specific optimizations and an architecture-aware tiling strategy.

Still, after months of work, Garcia only achieved a disappointing performance of 44.12 GFLOPS out of 80 GFLOPS possible (Figure 1.5), even in the favorable case when the operand matrices were already preloaded into on-chip memory.

The low payoff of Garcia's efforts, including months of optimizing assembly-written pieces of code, were the result of using traditional techniques for optimization. Garcia focused his efforts on selecting appropriate instructions and in performing code transformations such as instruction scheduling or register tiling.

Garcia's lack of success was prompted by the inadequacy of traditional optimization techniques, which he used skillfully. However, those traditional optimization techniques were not enough because they were inadequate for manycore processors:

- Explicit data movement operations were not carefully pipelined and scheduled with other tasks.



Hu's implementation [59] of matrix multiplication for Cyclops-64 was improved by Garcia [50] through the use of hardware-specific optimizations. The resulting program achieved a significantly better outcome, 44.12 GFLOPS out of 80 GFLOPS maximum. Although the results are good, the lack of dynamic scheduling and dynamic memory movement techniques limited the success of this approach.

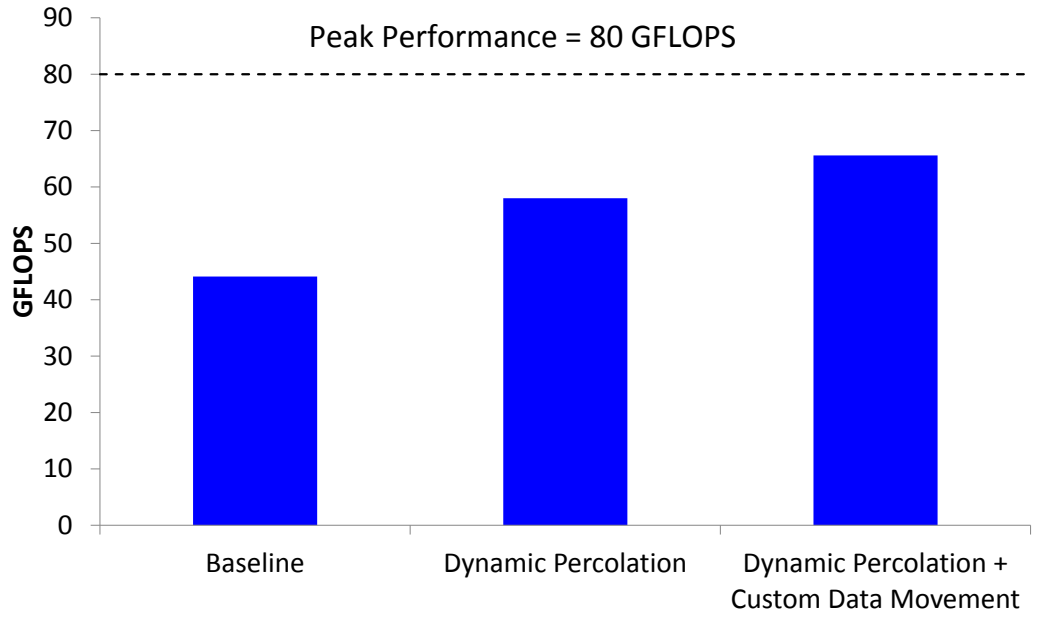
Figure 1.5: Matrix multiplication performance after hardware-specific optimizations.

- The synchronization operations available were slow and insufficient to express anything but trivial relationships in the parallel program.
- The static schedule used was unable to adequately manage the available resources.
- Memory bandwidth was not carefully allocated and used.

The combined experiences of Hu and Garcia ultimately confirm that current tools are insufficient to develop high performance computing programs for manycore architectures. In particular, their studies show several flaws that need to be addressed:

- The traditional programming models, such as C and its parallel libraries (such as POSIX threads [20]), do not provide tools to readily express the available parallelism, and they only provide awkward ways to exploit the available hardware.
- Data movement is done when needed, which is too late, resulting in frequent stalls of the program. Data movement should be done well before it is needed.
- During execution, there is not a good way to give preference to a task on the critical path over a task on a leaf path, resulting in further stalls when the tasks in the critical path are not executed early enough.
- The algorithms that form the building blocks of the scheduling and synchronization system were designed for little to no parallelism, and they exhibit a high overhead in highly parallel scenarios.
- A static schedule of resources (such as memory bandwidth) that has been computed a priory does not provide good enough results. The reason is that even in regular programs, the quantity and diversity of resources available create unpredictable variations during execution that do not fit static schedules. Also, static scheduling works poorly in manycores, even in cases that work for traditional architectures, such as regular applications. For example, in Garcia's first attempt, as much as 48% [50] of the total time was wasted as idle time because static scheduling failed to balance work properly.
- In most cases, execution control is centralized, resulting in high overhead when many processors make requests at the same time.

Garcia and I retook the work of Hu and Garcia [46, 47]. Armed by observations on the shortcomings of the approach, Garcia proceeded to (1) change the implementation to use a more parallel approach that resulted in little to no stalls, (2) employ a dynamic-scheduling approach to manage the available work, (3) design assembly-level



The performance of matrix multiply program was able to achieve close-to-peak performance once the necessities of manycore computing were addressed.

Figure 1.6: Performance of matrix multiply after using dynamic scheduling and dynamic percolation.

operations to support the required synchronization and dynamic scheduling and (4) design and implement a technique for dynamic memory prefetching (which we called *dynamic percolation* [46]).

Using dynamic scheduling [47] and dynamic percolation [46] allowed Garcia to expand the matrix multiplication algorithm to allow the more realistic case of matrix operands in DRAM and to avoid the stalls caused by static scheduling.

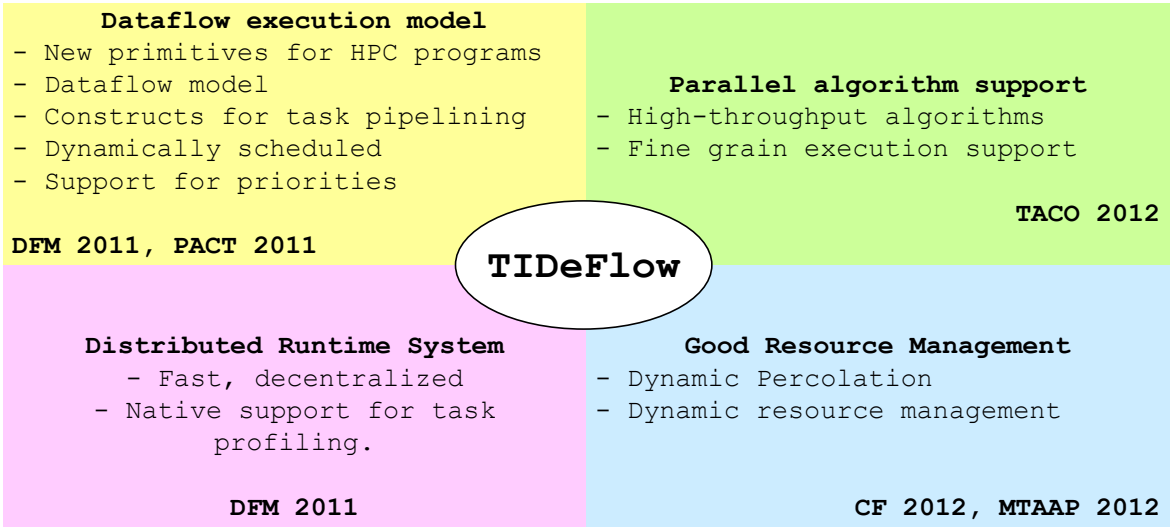
Ultimately, when the limitations of the traditional model were overcome, the performance of the matrix multiply program was 65.6 GFLOPS (Figure 1.6).

The experience gained through Hu's [59], Garcia's [50] and Garcia's and my [46, 47] work ultimately reveals the main issue that needs to be solved:

> *Traditional models are not adequate for computation of HPC programs in manycore architectures.*

If a new model is to address the problems of traditional models, it will have to be built from scratch to be able to incorporate all the features necessary for effective use on manycore processors.

The work advanced in this thesis, the Time Iterated Dependency Flow (TIDeFlow) model, is a new execution model that enables efficient execution and speedy development of HPC programs for manycore processors.



TIDeFlow is a parallel execution model that incorporates a variety of features that enable efficient, fine-grained execution in manycore systems.
The Figure shows the main features of TIDeFlow, which are described throughout the thesis. Emphasis is made on its execution model (Chapter 3), its support for fine-grained execution (Chapter 4) and its implementation (Chapter 5). The use of TIDeFlow for resource management is covered in other publications [47, 46].

Figure 1.7: TIDeFlow overview.

TIDeFlow incorporates all the features (Figure 1.7) identified by the study on matrix multiplication:

- It provides language support for common HPC constructs such as parallel loops.
- It supports a dataflow-style of computation able to represent arbitrary parallel programs.
- It has programming constructs for task pipelining and for dynamic resource management.

- Its implementation uses novel, high-performance, high-throughput, low-latency algorithms [85, 86].
- Its runtime system is decentralized, it supports fine-grained execution and it provides native support for task priorities and profiling.
- It provides dynamic scheduling to exploit the features of manycore processors.

With TIDeFlow, the time required to parallelize matrix multiply was reduced from 1 month (in Garcia's approach [50]) to a mere 1 week. The resulting program, explained in detail in Chapter 7, exhibited a performance close to that of the hand-optimized version by Garcia, at a fraction of the development cost.

The failures and the successes of the development and optimization of matrix multiply are intrinsically related to the nature of the architecture where the development was taking place. Because of its large number of processing units, Cyclops-64 posed new challenges and prompted new techniques that ultimately resulted in the development of TIDeFlow.

The use of Cyclops-64 was the necessary catalyst that led to the development of the TIDeFlow model in Chapter 7, the development of the ideas on throughput in Chapter 4 and the implementation of the distributed runtime system explained in Chapter 5.

The rich environment provided by Cyclops-64, where abundant amounts of hardware resources are present, is an excellent architecture to explain and test new techniques. With its large number of processing elements, Cyclops-64 constitutes an ideal environment to showcase the challenges faced by manycore architectures. For that reason, Cyclops-64 has been chosen to develop the models, the techniques and the experiments presented here. The choice of Cyclops-64 for this thesis in no way precludes the use of other architectures with TIDeFlow or the use of the conclusions and techniques reached here.

The successful use of TIDeFlow to develop matrix multiply for Cyclops-64 does not constitute a universal solution for all the problems in parallel execution. TIDeFlow focuses on execution of HPC programs on a single manycore processor, but it does not

solve all the problems that will be faced by exascale computing [17]. In particular, the TIDeFlow model presented here does not address the issue of hardware faults or energy consumption in large scale systems.

Resiliency is a feature of large computer systems that grant the ability to execute programs despite hardware faults. Resilient computing is of paramount importance to large scale computer systems in which hardware failures may be observed every few minutes. Execution models such as TIDeFlow may hold the key to unlocking successful implementations of resilient systems by allowing local, rather than global, fault recovery. Unfortunately, the work of TIDeFlow presented in this thesis has been crafted to address the issues related to the execution of programs in a single manycore processor, in which resiliency is not a pressing concern. Energy-efficient computation is another important design constraint for large computing systems. Instead of focusing on the computational rate of a program, an energy-efficient computation will try to minimize the total energy spent in the computation. TIDeFlow program optimization focuses on achieving a high performance rate rather than a low energy consumption, because the TIDeFlow system is designed to be run by a single manycore processor in which energy consumption is not an issue at the moment. However, Garcia, Gao and I have performed a preliminary study of the energy consumption of a manycore architecture.

The issues regarding resiliency and energy-efficient computation are described in greater detail in Chapter 9, where a detailed description of related open questions and their possible solutions are presented.

Synopsis

The rest of this dissertation covers in detail each one of the aspects involved during the development of TIDeFlow:

- Chapter 2 provides relevant background,
- Chapter 3 describes the TIDeFlow execution model,
- Chapter 4 advances the theory of intrinsic throughput of algorithms, along with some examples,

- Chapter 5 describes the implementation of the TIDeFlow toolchain,
- Chapter 6 develops useful programming techniques,
- Chapter 7 shows examples of TIDeFlow programs
- Chapter 8 provides some conclusions, and
- Chapter 9 presents open questions and future research directions.

Chapter 2

BACKGROUND

This chapter presents an overview of the history of computing.

This chapter describes the computing technology both from the point of view of computer architecture and from the point of view of how those architectures have been used to execute programs.

2.1 Computer Architectures

Computers have been a powerful tool for science and engineering since the moment of their invention. Since the beginning of computing, the usefulness of computers has increased with their computational power, leading scientists to be continuously on the lookout for more and more computational power and to create the field of High Performance Computing (HPC).

The desire to obtain high performance in programs can be seen in all ranges of computers, from the personal computers used by scientists to the supercomputers used by governments.

It would take thousands of pages and several years of study to provide a comprehensive description of every attempt made to improve computers since their invention. For that reason, this section aims to point out the most relevant highlights and their relationship to parallel computing and to HPC.

2.1.1 Serial Processor Systems

Executing programs using a single processor was immensely popular for consumer electronics and applications during the second half of the 20th century.

The reasons behind the success of single processors are manyfold. Compilers for them were readily available, the programming model was simple, uniprocessor machines were cheaper and so on.

One of the biggest reasons that prompted the success of computing using serial processors is the exponential improvement in clock frequency experienced between 1970 and 2004 (Figure 1.1). Such improvements in clock frequency [69] allowed programs to run faster as time progressed, greatly favoring programs that were written using serial programming languages. Additional factors also contributed to the success of serial processors [56]: the possibility to issue and complete multiple instructions per cycle (as in the Pentium 4 processor), the use of scoreboards to execute instructions out-of-order when no dependencies between instructions were detected (as done by the CDC 6600 [109]), the existence of multiple levels of cache (as in the Intel Core i7 processors [4]), the addition of vector arithmetic units (part of the Single Instruction Multiple Data features of the Intel processors [94]) and so on.

The effectiveness of uniprocessor computing for commercial applications is closely related to the quality of the tools available. The tools for serial programming languages such as C and Fortran were and still are excellent. A cycle was formed: serial paradigms were possible because of the good tools available, and good tools were available because of the popularity of serial programming models.

However, the increase in performance of uniprocessor computing has stopped almost completely during the past few years. Attempts to improve the speed of serial processor have met obstacles that have come to be known as the *performance walls*. The performance walls can broadly be described as the difficulties in increasing the processor frequency, the difficulties in reducing the power consumed by processors, the difficulties in finding parallelism within the instructions of a serial program and the difficulties in matching the speed of memory with the speed of processors.

Both the frequency wall and the power wall are related in that increases in the frequency at which processors operate will result in increases in the operating frequency of transistors. In particular, the amount of power used to change the state

of a transistor, known as the *dynamic power*, can be expressed as [56]:

$$P_{dynamic} = \frac{1}{2} C \times V^2 \times f$$

where $P_{dynamic}$ is the dynamic power required to switch the state of a transistor with a capacitive load C at an operating voltage of V and a frequency of f.

Processor systems today are already at the limit of their capabilities in both power and frequency. Frequency can not be increased any more because it will lead to an increase in power, and power can not be increased anymore because (1) using more power will result in an increased monetary cost to operate the processor and (2) because processors are already at the limit of their power dissipation abilities.

The memory wall refers to the disparity in clock frequency between processors and memory in general. This problem is the result of increased amounts of performance in processors that have not been matched with corresponding increases in memory performance. Although several techniques (caches, latency hiding, and instruction scheduling) have been used to reduce the impact of the memory wall, it remains a limiting factor in processor performance.

The Instruction Level Parallelism wall [56] has the difficulty of finding instructions that can be executed in parallel inside of a serial program. The two factors that make Instruction Level Parallelism (ILP) difficult are as follows: (1) the complexity of detecting if a group of instructions can be executed in parallel and (2) the limited amount of instructions that could be executed in parallel if they could be identified.

2.1.2 Shared Memory Systems

To increase the performance of programs, computer architects extended the concept of the traditional single processor system to include several processors. The addition of several processors to the same system allowed additional performance to be available. In these systems, known as *shared memory systems*, multiple processors use the same memory for computation and communication.

The popularity of shared memory systems can be partially attributed to the fact that the already-familiar serial programming languages could be extended to express some parallelism. Approaches such as POSIX Threads [21] and OpenMP [79, 32] quickly became popular and led to a wide range of parallel programs.

However, the introduction of parallelism in an environment with a shared memory space conflicted with traditional optimization operations. Optimizations that reordered instructions in serial programs no longer produced correct results in parallel programs that used shared memory. Nevertheless, the advantages in programmability and performance of shared memory system have made it popular to this day.

2.1.3 Distributed Memory Systems

Distributed Memory systems, where each processor has its own memory address space, is currently the most used model by supercomputing centers today. For example, in the November issue, all but one supercomputer on the TOP 500 list of the fastest supercomputers of the world [103] have a distributed memory architecture; however, 17.8% of them use techniques to emulate shared-memory behavior.

Interconnection networks are used as a way to support communication between processors using explicit messages. The interconnection network of a distributed system is a fundamental part of its architecture, and it greatly influences its usability and performance. Both proprietary and commercially available technology are used to build the interconnection network, with Gigabit Ethernet and Infiniband being the two most popular technologies [103].

The topology of a distributed memory system also plays an important role in the performance it can deliver. Although traditional topologies usually included processors tightly connected using several geometrical patterns such as grids, meshes, toruses and trees, the availability of the commodity hardware used to support the Internet has led to the creation of systems with lose topologies.

Programs for distributed memory machines are written in a variety of languages (Bal et al. have compiled a reasonably comprehensive list [11]); however, the most

common model remains the Message Passing Interface (MPI) specification, explained in detail in Section 2.2.8.

2.1.4 Multicore Systems

Multicore systems were a natural evolution of computer architecture that leveraged the existing tools used for shared memory systems and the advances in computer architecture that allowed additional transistors per processor chip.

Prompted by the difficulties involved in further increasing single processor performance, computer architects resorted to placing several processing cores per chip, creating a system similar to shared memory systems in that all processing elements can access main memory.

The similarities between shared memory systems and multicore systems allowed a speedy development of multicore technology. The same programming and execution paradigms could be used, and in most cases, programs could be ported directly from shared memory systems to multicore systems.

2.1.5 Manycore Systems

The recent developments on processor architecture are greatly influenced by previous approaches to parallel computing. Not all of these approaches were commercially successful, in part due to the steady increase of uniprocessor performance. Nevertheless, these approaches laid the foundations that would result in the development of manycore architectures.

As early as 1974, a fully parallel machine was proposed [37]. Other attempts followed in 1977 [97] with the development of a dataflow machine, in 1982 with the U-Interpreter initiative [9], and in 1986 with the Manchester Computer [111], among many others.

Such initiatives, although immensely successful in academia, nevertheless failed to become commercially mainstream. Many reasons helped the prevalence of serial processors and serial programming models over parallel processors with their respective

programming models, but among them all, the fact that serial processors became faster with each generation stands as the most important.

Indeed, the parallel computing revolution is a *forced* revolution [101]. Traditional increases in performance that could not be obtained through increase in individual performance of processors was sought through the use of more processors. Even the community of High Performance Computing (HPC), which had used parallel computers for decades but had written the large majority of their programs using MPI, faced new challenges when multiple processors were placed in the same chip.

The transition from a serial paradigm to a parallel paradigm is difficult: traditional programming languages such as C or Fortran were not designed to express parallelism and posed serious problems in portability. Extensions made to serial programming models to allow expression of parallelism (OpenMP, pThreads, and others) opened a new set of challenges such as correct synchronization of programs or correct modeling of memory operations.

The failure of processor chip manufacturers to continue increasing the frequency of individual processor cores has uplifted and revived the research started decades ago.

The field of computer architecture faced several challenges that led to the development of new processors characterized by large amounts of hardware parallelism inside a processor chip. First, the failure to increase the performance of single-processor computing forced the use of parallel architectures. For example, the Intel® Core™ 2 Duo processor E8600 operated at 3.33 GHz [27], almost the same speed of its single core predecessor, the Pentium 4 processor, which operated at 3.73 GHz [29]. Second, significant increases in transistor count resulted in only incremental increases in processor performance. For example, the Pentium III processor (1.4 GHz version) had 44 million transistors [28] and had a peak performance of 2.80 GFLOPS [26] while the Pentium 4 processor (Extreme Edition, HT, 3.73GHz) had 169 million transistors [29] but only had a peak performance of 14.93 GFLOPS [30] despite having a clock frequency 2.7 times larger. If both processors were to be run at the same speed, then, the performance difference would be only 2X. This increase in performance is due to

the architecture itself and comes at a cost of an increase of 3.8X in transistor count.

These factors led to a new paradigm in computer architecture: Performance would be obtained through the use of many simple processor cores instead of few high-performance processor cores. Processor components were simplified to reduce processor design time and to expose the architecture to the user. Virtualization was dropped when possible and hardware support for parallel execution was provided.

This new design philosophy led to the development of a new class of processors known as *manycore* processors. Representative examples of manycore processors include Tilera's TILE64 processor [5], Sun' UltraSPARC T2 processor [99] and IBM's Cyclops-64 [31].

The following sections describe the most relevant features of selected manycore systems. Special attention is devoted to IBM's Cyclops-64 because it is used to demonstrate the techniques presented by this thesis.

2.1.5.1 Tilera's Processors

Tilera is a company that produces manycore processors such as the 100-core TILE-Gx8100, or the 64-core TILE64 [5] processor. Tilera's processors retain many of the characteristics of traditional processors such as the use of caches or virtualization, but they also employ new techniques such as several dedicated networks for message passing between cores.

2.1.5.2 Sun UltraSPARC T2

Niagara 2, also know as UltraSPARC T2 [99, 65], is a processor by Sun Microsystems with 8 cores, each capable of executing 8 simultaneous threads. Niagara 2 is an intermediate step towards manycore architectures that features a full crossbar network that connects all cores to the memory.

2.1.5.3 IBM Cyclops64

This section presents an overview of Cyclops-64. The details of Cyclops-64 have been extensively covered by previous publications in which I appeared as an author

[116, 115, 82, 83, 86, 45, 80, 85, 47] and by publications done by others [36, 35, 31, 34, 33, 59].

Cyclops-64 (C64) is a manycore processor developed by IBM. Cyclops-64 distinguishes itself from other manycore processors because of its revolutionary design, its features for fine-grained synchronization and its large number of processor cores.

Cyclops-64 features a powerful on-chip interconnection network, hardware support for barriers and sleep-wakeup operations, a fully connected crossbar network that connects its 80 cores, and on-chip hardware for inter chip communication. One C64 chip has no automatic data cache and can address 1GB of DRAM memory.

The chip runs at 500 MHz, and each one of its 80 cores has two single-issue thread units that share a fully pipelined floating point unit. The peak rate of 80 GFLOPS can only be achieved if the floating points unit executes Multiply-Add instructions continuously.

Each thread unit in Cyclops-64 can issue one double precision floating point Multiply-Add instruction per cycle. The processor chip includes a high-bandwidth on-chip crossbar network with a total bandwidth of 384 GB/s. C64's memory hierarchy includes three levels of software-managed memory (DRAM, SRAM, ScratchPad), with the Scratch-Pad (SP) currently used to hold thread-specific data. Each hardware thread unit has a high-speed on-chip SRAM of 32KB (minus 2KB required for the system kernel) that can be used as a user-managed cache. Figure 2.2 shows the capacity, bandwidth and latency for the register file, the ScratchPad memory, the SRAM Memory and the DRAM memory.

One of the main features of the C64 chip is that *its memory controllers can execute atomic operations*. In C64, each memory controller contains its own Arithmetic and Logical Unit that allows the memory controller to execute integer and logical atomic operations *in memory* without the intervention of a processor or a thread unit.

Section 2.1.5.3 is based on several publications in which I appeared as an author [116, 115, 82, 83, 86, 45, 80, 85, 47]. Some paragraphs, figures and tables have been reproduced verbatim. Reproduced with permission.

Chip Features	
Processor Cores	80
On-Chip Interconnect	Full Crossbar
Frequency	500 MHz
Off-Chip Interconnect	Proprietary
Hardware Barriers	Yes
Hardware Mutex	Yes
Hardware Sleep - Wakeup	Yes
Addressable Memory	2GB
On-Chip Memory	User Managed, 5MB
Off-Chip Memory Banks	4 x 64 bits
Instruction Cache	320 KB
Processor Core Features	
Thread Units	2, single issue
Local Memory	32KB
Floating Point Unit	1, pipelined
Registers	64 x 64 bit

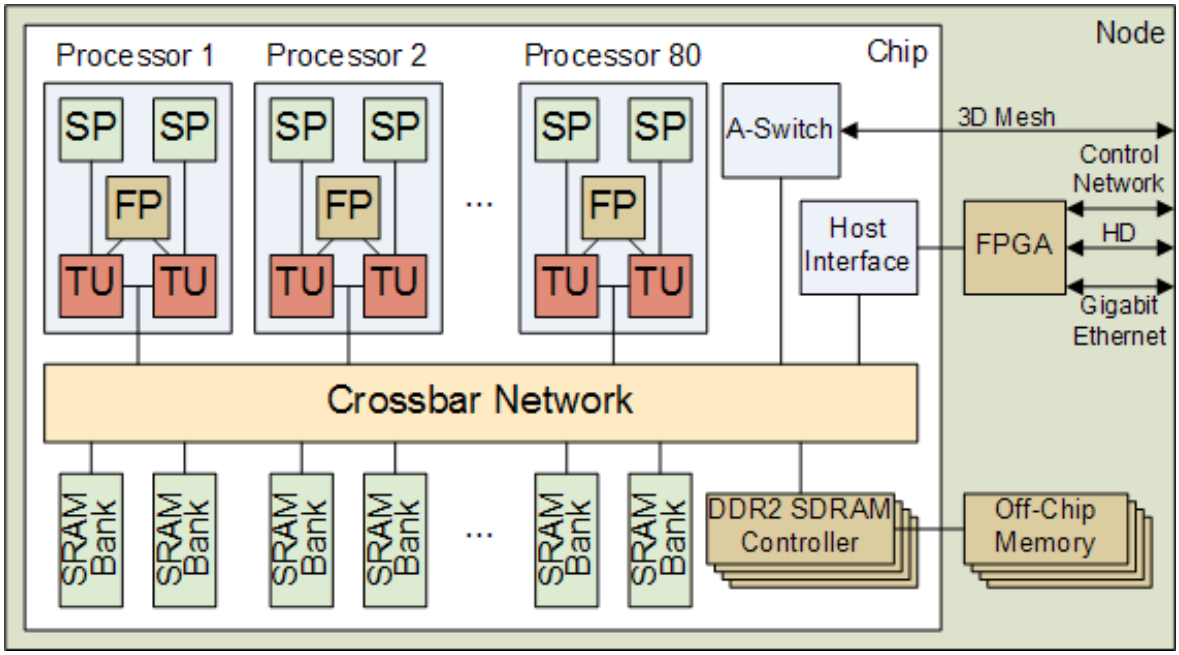
Table 2.1: Cyclops-64 Features

Atomic operations in C64 take 3 cycles to complete at the memory controller. All memory controllers in C64 have the capability to execute atomic operations.

Under the default configuration, C64 has 16KB of stack space (Scratch Pad Memory) for each thread unit, 2.5MB of shared on-chip memory, and 1GB of DRAM memory.

Table 2.1 summarizes the features of the Cyclops-64 many core chip.

One of the big challenges to achieve high performance for Cyclops-64 is the great amount of locality required to achieve the peak computational rate. The Floating Point Unit present in each core has a direct connection to the register file, allowing it to consume 2 double precision numbers and produce a third double precision number per cycle. On the other hand, the total off-chip memory bandwidth is only 4 x 64 bits or 4 double precision values per cycle. The Cyclops-64 chip can consume 160 double precision values in one cycle taking into account that each one of the 80 Floating point units can consume 2 double precision values per cycle. However, on average, only 4 double precision values are available from memory each cycle. In other words,



Cyclops-64 is a manycore architecture with 160 thread units. Its large number of thread units make it ideal to test parallel execution models. Figure taken from [115]. ©2011 IEEE.

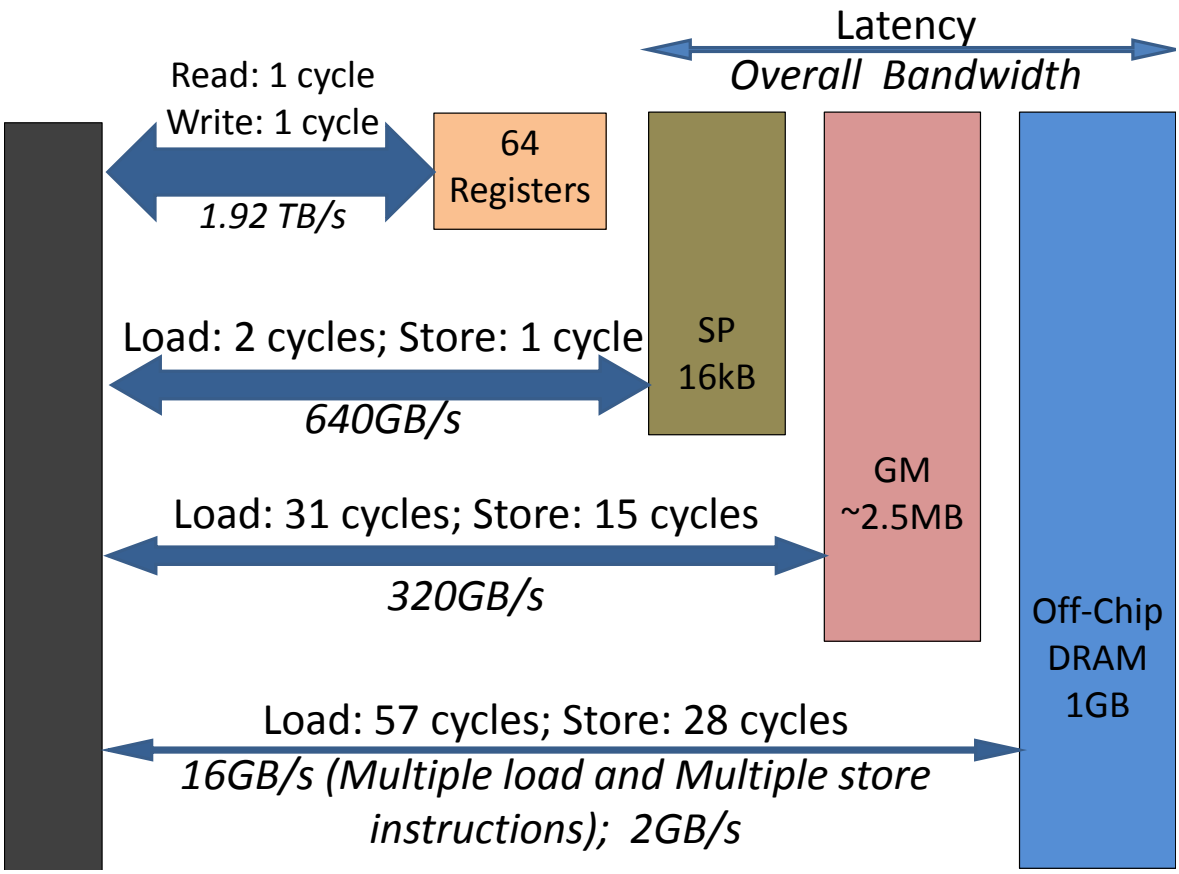
Figure 2.1: Cyclops-64 Architecture.

the overwhelming disparity between available bandwidth to off-chip memory and the floating point units from the register file means that at most only one out of $160/4 = 40$ instructions can be a DRAM memory operation, or the performance of programs will be bound by the available bandwidth.

Effectively using the on-chip memory in Cyclops-64 is key to achieving good performance. How to find the best possible use of on-chip memory is an important topic of research that will be pursued in this work.

The IBM Cyclops-64 system was originally developed by IBM as part of the Blue Gene project. Figure 2.1 shows the C64 processor features, including its 80 processing cores, its two hardware thread units per core, and its 64-bit floating point units.

Cyclops-64 provides hardware support for fast synchronization through a dedicated signal bus (SIGB). The signal bus broadcasts signals across processors through the use of a special register. As shown in Figure 2.3, reads to the signal bus register



Overview of the memory in Cyclops-64. Taken from [115]. ©2011 IEEE.

Figure 2.2: Cyclops-64 Memory Hierarchy.

result in the combined or-operator of all writes to signal bus registers of all threads. In this fashion, all threads read the same value, allowing synchronization decision to be done by inspection of particular bits in the resulting value read.

Read operations on the signal bus take the standard register latency to complete (1 cycle) and write operations take 10 cycles to propagate across the chip, allowing very fast primitives such as barrier operations and mutual exclusion.

A programmer interface for thread management operations has been developed, and it is available through the TiNy-Threads (TNT) [31] library. Table 2.2 shows a list of sample API calls provided by the TNT library.

The TNT API provides functions to suspend a thread and to awaken a sleeping thread. A suspend instruction temporarily stops execution in a non-preemptive way,

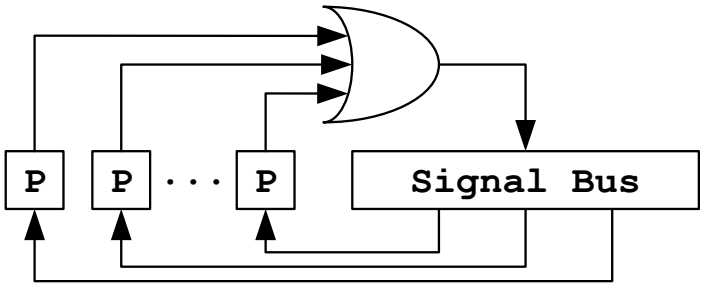


Figure 2.3: Cyclops-64 Signal Bus Design

and a signal instruction awakens the sleeping task. Using thread suspend and wake mechanisms in place of the busy-wait approach reduces memory bandwidth pressure because all waiting tasks can suspend themselves instead of spinning. The master can collect all the signals from waiting tasks and finally signal the suspended tasks to resume the execution.

The C64 chip provides an interesting hardware feature called the "wake-up bit". When a thread tries to wake up another thread, it sets the "wake-up bit" for that thread. This enables a thread to store a wake-up signal. Hence, if a thread tries to suspend itself after a wake-up signal is sent, it wakes up immediately and the suspend effectively becomes a no-op.

Name	**Description**
tnt_create()	Thread creation
tnt_join()	Thread Join
tnt_mutex_lock()	Mutex Lock
tnt_suspend()	Suspend current thread
tnt_awake (const tnt_desc_t)	Awaken a suspended thread
tnt_barrier_include (tnt_barrier_t *)	Join in the next barrier wait operation
tnt_barrier_exclude (tnt_barrier_t *)	Withdraw from the next barrier wait operation
tnt_barrier_wait (tnt_barrier_t *)	Wait until all threads arrive this point

Table 2.2: Cyclops64 TNT APIs for Hardware Synchronization Primitives

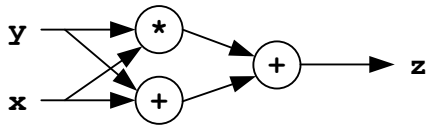


Figure 2.4: A static dataflow graph that computes $z = x + y + x * y$.

2.2 Previous Models for Execution, Concurrency and Programming

The following sections introduce previous models for execution, modeling and programming of parallel programs.

2.2.1 Dataflow

Dataflow is a program execution model proposed by Dennis [37], whose main idea is that operations can execute when their operands are available.

Dataflow programs are usually expressed as a graph where computation is represented as actor nodes, and data dependencies are expressed through directed edges in the graph. Data itself is represented as tokens located in edges. Figure 2.4 shows an example of a dataflow graph.

Dataflow is excellent at expressing parallelism in programs. Dataflow programs are well suited to highly parallel implementations. The graph description used, where actors in the graph execute (*fire*) when its operands are ready, lends itself to parallel implementations with distributed control.

Dataflow research in the past decades has resulted in a significant number of research publications, many variants and some machine implementations.

Some of the most important dataflow models include the initial model proposed by Dennis [37], Khan process networks [62], hybrid dataflow models such as the McGill dataflow architecture model and its argument fetching model [41], dataflow software pipelining [38], the MIT tagged token architecture [10], the EARTH model [104, 105, 106], the U-Interpreter [9] and others. A more comprehensive description of dataflow and its implementations can be found in the excellent survey of Najjar et al [75].

As the survey from Najjar shows, many advances in the dataflow field were done in the past. However, those advances were made while the computer architecture field produced predominantly uniprocessors with shared memory, vector processors or distributed memory processors. The recent introduction of many core technology has left many questions open, and if it is to survive, dataflow must evolve to be able to answer those questions and rise to the challenges of the manycore revolution.

The dataflow model, where parallelism is an intrinsic part of the model rather than an afterthought, promised excellent results in parallelism. However, the inertia acquired by the use of traditional programming models prevented dataflow from becoming commercially successful, even though research projects such as the Monsoon machine [91] or the U-Interpreter [52] showcased the capabilities of dataflow. On the other hand, the industry had, at the same time, developed thousands of software products written in sequential languages. The need to support those programs with little or no modification provided the economic fuel necessary to maintain the supremacy of serial paradigms over dataflow which required increased time and effort in order to be a viable alternative.

2.2.2 Petri Nets

Petri nets are "a graphical and mathematical modeling tool applicable to many systems" [74]. Petri nets are useful to describe and study parallel systems, asynchronous systems, stochastic systems and others.

The concept of Petri nets was first proposed by Carl Adam Petri's dissertation [92], and it has since enjoyed a significant amount of attention in the research world. Murata [74] presents a comprehensive survey of previous approaches to Petri nets.

A Petri net is a directed, weighted, bipartite graph N with an associated initial state called the *initial marking,* M_0. The graph consists of two kinds of nodes–*places* and *transitions*–and arcs that go from a place to a transition or from transitions to places [74]. In modeling, places represent conditions that need to be met, and transitions represent events. Usually, places are represented as circles while transitions

are represented as squares. The weight of an arc represents the number of equivalent parallel arcs between two nodes in the graph. Arcs with no weights are assumed to have a weight of 1. Each place has an associated marking (state) which is a nonnegative integer, representing a number of tokens. In practice, the tokens are represented graphically as dots inside the circle. The state of a program (its marking) is defined by a vector containing the marking of each place in the graph. The behavior of Petri net systems can be modeled through three simple rules: (1) A transition is enabled if each one of its input arcs comes from a place with a token. (2) Enabled transitions may (or may not) fire. (3) When firing a transition, one token is removed from the place that connects each input arc and produces one token on the places pointed out by each output arc. A more detailed explanation of the rules along with examples can be found in the survey by Murata [74].

Petri nets are preferable to dataflow in that they model resource sharing, which the original static dataflow does not. Extensions to the static dataflow model – such as queued dataflow – allow resource sharing, but they still lack in their synchronization capabilities. Although Petri nets are useful at modeling programs, they do not address the more general case of execution, synchronization and scheduling. The Petri net theory does not specify how to execute, synchronize or schedule programs.

2.2.3 EARTH

The Efficient Architecture for Running Threads (EARTH) [104, 105, 106, 117, 107] was a project aimed at an evolutionary approach to parallelism. One of the main features of EARTH is its intention to use commodity hardware to produce a dataflow implementation.

As in Dennis's dataflow, EARTH programs are expressed as graphs. However, two levels of parallelism, threaded procedures and fibers, were introduced to improve locality and to take advantage of the architecture features of commodity processors.

Threaded procedures are the minimum unit of execution that could be mapped

to a particular processor, usually executing the body of a function. Threaded procedures have state, a frame and stack. Execution of a threaded procedure is mapped to a particular hardware processor.

Fibers are the lowest level of execution. They are designed to provide fine-grained execution, and they typically consist of a few instructions. Fibers execute within the context of a thread: They can access its local variables and its frame. A fiber is a "sequentially-executed, non-preemptive, atomically-scheduled set of instructions" [104].

In EARTH, execution is controlled through synchronization signals. Fibers signal all of their data consumers when they finish execution. As in dataflow, fibers become enabled after receiving signals from each one of their dependencies.

The architecture model of EARTH is composed of nodes connected through an interconnection network. Each node is composed of one or more execution units, a synchronization unit, local memory and buffers for incoming and outgoing requests (Ready Queue and Event Queue buffers respectively). Scheduling and synchronization is handled by the synchronization unit. Requests for synchronization or scheduling of fibers are received in the Event Queue buffer either from other nodes or from the local execution units. When a fiber becomes active, it is placed in the Ready Queue where it is taken by one of the Processing Elements in the node. EARTH has been used successfully to execute parallel programs. For example, Chen [25] showed the advantages of using the EARTH model to compute a conjugate gradient benchmark, and Theobald [104] showed extensive experiments where EARTH was used to compute several well-known benchmarks.

Some of the main disadvantages of the EARTH model include its limited ability to express parallel loops as well as the centralized nature of the synchronization unit, which in a typical implementation, would consume a hardware processor.

2.2.4 ParalleX

ParalleX [42] is a parallel programming and execution model. The features of ParalleX include a global name space, rich semantics for parallelism, direct support for lightweight tasks, automatic latency hiding and low overhead for synchronization, data movement and load balancing.

ParalleX introduces the concept of Parcels, which is a message paradigm for asynchronous and distributed operation. Parcels are messages that include data as well as a specification for an action to be performed once the data has been received. Such action is commonly referred to as a "continuation".

The synchronization primitives of ParalleX take the form of Local Control Objects (LCOs). These objects, similar to dataflow synchronization primitives, futures or metathreads, can be used alone or as part of larger synchronization structures.

Other features of ParalleX include mechanisms for percolation (data prefetching), a memory model where cache coherency is specified on a local domain only, and a hierarchical view on processes.

ParalleX is related to TIDeFlow in that both attempt to specify a parallel execution model to address similar issues, but the implementation, specification and interfaces of ParalleX and TIDeFlow are significantly different.

2.2.5 Swarm

The Swift Adaptive Runtime Machine (SWARM) [3] is an execution and programming model by ET International.

In Swarm, programs are written as directed graphs where nodes are codelets and arcs represent the dependencies between codelets. Swarm features a programming model where graphs can be constructed dynamically while the program executes.

The conception of the TIDeFlow execution model is intrinsically related to that of Swarm. The current versions of Swarm and TIDeFlow were inspired by the same project at ET International, and both are able to target execution in manycore architectures.

The following differences exist between Swarm and TIDeFlow: (1) The implementation is different, (2) Swarm is able to express program graphs that change during execution while TIDeFlow does not, (3) TIDeFlow has constructs to represent parallel loops while Swarm does not and (4) TIDeFlow allows outer loop dependencies to be represented as a basic construct while Swarm does not.

2.2.6 POSIX Threads

The POSIX threads standard [21], commonly known as *pThreads*, describes the use of threads in serial programming languages. The pThreads standard provides library functions to perform thread management (creation, termination, synchronization) in shared memory systems.

pThreads provides a good interface for thread management, making it ideal as a building block for other implementations.

The pThread library relies heavily on the operating system to perform scheduling, synchronization and management of threads.

One of the main drawbacks of pThreads is that its shared memory model allows race conditions when multiple threads access the same memory location without proper synchronization [76]. Deadlocks may also happen due to programming errors [61].

2.2.7 OpenMP

OpenMP [79, 32] is another standard that specifies language extensions to serial programming languages such as C or Fortran to express parallelism in a program. In OpenMP, the programmer is responsible for expressing the available parallelism through pragmas. For example, there are pragmas to specify that particular loops can be executed in parallel, to specify parallel regions, parallel tasks, or reduction operations. An advantage of OpenMP programs is their compatibility with serial execution, since the pragmas that specify parallelism can always be ignored to construct an equivalent serial program. The ease of use of OpenMP along with its compatibility with serial program has undoubtedly contributed to its popularity.

One of the disadvantages of OpenMP is that it does not provide tools to express arbitrary constructs for parallelism or pipelining, and its model falls within the class of execution models where programs are described as a sequence of fork and join operations.

2.2.8 MPI

The Message Passing Interface (MPI) [53] is a specification for parallel computing where multiple processors with independent address spaces communicate through the interchange of messages. MPI belongs to a set of paradigms related to point-to-point communication between processes. Such set of paradigms were first advocated by Hoare [58] and consist of a collection of sequential programs that communicate through message passing. MPI remains today the tool of choice to develop supercomputer applications due to its implementation on top of serial languages such as C and Fortran.

However, MPI programs are difficult to write, and they are particularly susceptible to programmer errors that lead to deadlocks. Given its popularity in supercomputer systems, MPI programs are not intrinsically resistant to hardware failures, requiring significant programmer effort to be able to thrive in large computing systems.

2.2.9 X10

X10 [24, 112] is an initiative originally led by IBM to develop a programming language that would allow greater productivity. Although in general X10 allows computation in distributed memory systems, it is mostly used for shared memory systems.

In X10, parallelism can be expressed through the use of asynchronous tasks, which are executed asynchronously with the main execution. Results are obtained through a synchronization statement or through a *finish* construct.

X10 has led to other successful implementations of parallel languages, including Habanero-C [13], Habanero-Java [22] and others. The usability of languages like

Habanero has been greatly expanded with the addition of *phasers* [100], which allow point to point and collective synchronization between asynchronous tasks.

2.2.10 Cilk

Cilk [40, 1] is an execution model that has been very successful in the academic world due to its clean design and development. Cilk made contributions in several fields but in particular, its work-stealing [16] technique for scheduling has shown to be excellent to achieve load balancing across an arbitrary number of processors.

Cilk's scheduling techniques guarantee that the amount of memory used by any program executed with Cilk is bounded by the number of processors and the amount of memory used by a serial execution [15]. This important proof has contributed to make Cilk one of the references for parallel programming.

The Cilk model is particularly useful for recursive parallelism or embarrassingly parallel programs. Its programming model is simple, and it only requires the addition of a few keywords to the C language. However, the main drawback of Cilk programs is that they also fall within the category of the fork-join model, where arbitrary synchronization dependencies can not be expressed.

2.2.11 Intel's Concurrent Collections

Intel's Concurrent Collections (CnC) [18, 19, 23] is an execution model by Intel Corporation that facilitates writing and tuning C++ programs that execute in parallel.

CnC programs are typically expressed using two files. The first file describes a CnC program graph while the second file contains pieces of sequential code (*steps*) written in C++ and are meant to be executed sequentially. CnC program graphs are directed graphs where nodes are either *steps*, *items*, or *tags*.

Steps represent computation that must be performed. The code for the steps is given in a separate C++ file. Items represent data that is passed between steps. Tags are a way to represent control dependencies between steps.

Execution of a CnC program is determinate. The memory is managed automatically by the runtime system through hash tables. Steps execute when both their tags and their data become available.

CnC's technique to express parallel loops is similar to that of TIDeFlow. In CnC, parallel loops can be represented by a step with an associated range of tags that become available at the same time. The execution of CnC programs is also similar, providing the same program termination conditions.

An analogy between CnC tags and TIDeFlow time instances exists. Also, CnC steps are similar to TIDeFlow codelets.

Some differences between CnC and TIDeFlow remain: TIDeFlow has been designed to be a low-level runtime system where there is not an operating system. Memory is manually managed by TIDeFlow, and control and data dependencies are not distinguished in TIDeFlow.

2.2.12 StreamIt

StreamIt [108] is a programming language with its associated compiler that has been targeted to *streaming* systems. In the context of StreamIt, streaming programs are programs where a continuous data stream is processed to produce a continuous data stream. Streaming programs can be represented as a directed graph where nodes represent filters and arcs represent data channels between filters.

StreamIt aims to ease the programming of streaming applications as well as facilitating the mapping of those applications to several architectures, including commercial off-the-shelf processors, multicore processors and distributed memory systems such as clusters.

StreamIt is similar to TIDeFlow in that it targets regular and repeating computation where communication is explicit. StreamIt results in pipelined execution of tasks as in TIDeFlow.

Unlike TIDeFlow, StreamIt is designed to operate on a continuous stream of data. The TIDeFlow model suits well the stream computation model, but it also

addresses other models of computation.

StreamIt programming model is different to that of TIDeFlow. In StreamIt, programs are written entirely through a language interface that is similar to C.

Despite the differences, StreamIt is very similar to TIDeFlow in its intent and its techniques.

2.2.13 Intel's Thread Building Blocks

Intel's Thread Building Blocks (TBB) [93] is a parallel programming model for C++ based on threads. Execution of TBB programs rely on a template-based runtime library to take advantage of the features of multicore processors.

TBB aims to change programming paradigms [93], in particular TBB aims to "specify logical parallel structures instead of threads, emphasize data parallel programming and take advantage of concurrent collections and parallel algorithms" [2].

TBB provides additional keywords to the C++ language to specify parallel loops, mutual exclusion, atomic operations and task scheduling while supporting several of the primitive constructs used in higher level systems such as CnC or TIDeFlow. In fact, several implementations of CnC rely on Intel's TBB.

TBB, however, is not a full execution model, but rather a parallel programming model based on threads. In that sense, it is different to TIDeFlow or CnC.

2.2.14 Other Approaches

PLASMA (The Parallel Linear Algebra for Scalable Multi-core Architectures) [6] is a project that seeks to address the execution of linear algebra primitives in high performance computing environments that use multicore architectures. PLASMA uses a combination of algorithms and scheduling technique to solve linear algebra problems. Like TIDeFlow, PLASMA targets processors with many cores. However, PLASMA is not an execution model, but a library for linear algebra.

MAGMA (Matrix Algebra on GPU and Multicore Architectures) [6] also seeks to address execution of dense linear algebra problems similar to those of the LAPACK

[8] library. MAGMA is able to exploit heterogeneous systems containing multicores processors and Graphical Processor Units (GPUs). Computation of MAGMA operations use a dataflow graph that is dynamically scheduled to either a multicore system or a GPU. The main advantage of MAGMA is that it exploits the advantages of different algorithms for different frameworks. MAGMA is similar to TIDeFlow in that it also uses dataflow graphs for execution. It is different to TIDeFlow in that it (1) it is a library for linear algebra rather than an execution model and (2) it targets heterogeneous systems.

The Kernel for Adaptive, Asynchronous Parallel and Interactive Programming (XKaapi) [51] is a C++ library for parallel programming, specifically, for dataflow synchronization between threads. XKaapi's dataflow graphs are dynamic, unfolding at runtime. XKaapi has been designed for clusters of SMP machines. XKaapi is a library and not an execution model, like TIDeFlow.

Nanothreads [67] is a programming model whose goal is to manage a program's parallelism at the user level. The nanothreads model uses a Hierarchical Task Graph (HTG) to represent programs in a fashion similar to that of dataflow. Execution order is analogous to dataflow's lazy semantics, where an actor is enabled if its output is requested by another actor. Nanothreads is different to TIDeFlow in that it does not target parallel loops and it does not address time instances or task pipelining.

Other less known approaches that are based in dataflow, but are different to TIDeFlow in that they lack time instances, parallel loops or distributed control include Dataflow Process Networks [66], GRAPE [72], LUSTRE [55], SISAL [54], VAL [68], Lucid [113], Id [96], and others.

Chapter 3

THE TIDEFLOW PROGRAM EXECUTION MODEL

This chapter presents the Time Iterated Dependency Flow Model (TIDeFlow), a program execution model designed to address the challenges found in executing High Performance Computing programs in manycore processors.

The model proposed in this chapter has its roots in the experience with the implementation of matrix multiplication described in Chapter 1. The matrix multiplication experience showed the characteristics necessary for a successful execution model: (1) a dataflow-based representation of programs, (2) specialized constructs for parallel loops, (3) support for composability and (4) support for task pipelining at the programmer's level.

Each one of those requirements have been integrated in the TIDeFlow model. The model is based on queued dataflow, and it describes programs as graphs. Parallel loops are represented directly and easily by the nodes in the program graph. Full support for composability is provided, and task pipelining was supported automatically when the outer loop carried dependencies of a specified program.

This chapter describes each part of the execution model in detail. At the beginning, an overview of the model is presented, along with the reasons that drove its design. Then, the TIDeFlow program model is introduced. Examples are used to illustrate the role of each of its constituent parts: actors, arcs, tokens and programs. A discussion on the memory model is also presented, along with an explanation of the features that enable proper pipelining during execution.

This chapter is based on several of my previous publications [80, 88, 89] on the TIDeFlow model. Some paragraphs, figures and tables have been reproduced verbatim. Reproduced with permission.

3.1 An Overview of the TIDeFlow Model

The TIDeFlow model has been developed to simplify the execution and development of HPC programs. To succeed in its goal, TIDeFlow includes primitives that address the most common traits present in HPC programs such as Fast Fourier Transform (FFT) [90], Matrix Multiply (MM) [50] or Reverse Time Migration (RTM) [80]).

Of all the features common to HPC programs, perhaps the most relevant one is that in most of the computational loops found, *the innermost loop was parallel*. For example, in the matrix multiplication program presented by Garcia [48], most of the time is spent computing parallel loops . The code of RTM (experiments have been presented by Orozco et al. [80]) shows that most computational inner loops are parallel, and an analysis of the FFT algorithm shows that each computational step is fully parallel. Parallel loops are common in HPC programs because they can be readily used to express linear algebra operations, which are in turn at the heart of many scientific simulations.

The experience with the optimization of FFT, MM and RTM also shows that a highly-optimized version of an HPC program requires the ability to express complex dependence relationships between parts of the program. Unfortunately, the traditional constructs available are unable to express these relationships in a straightforward way, forcing the programmer to either resort to awkward and error-prone constructs or to build his/her own synchronization primitives from scratch.

The evidence presented shows that HPC programs have abundant parallel loops that depend on each other. For that reason, TIDeFlow provides constructs to express parallel loops, dependencies between parallel loops and multi-dimensional loops such as those commonly found in programs that simulate physical phenomena.

TIDeFlow addresses the necessity to support parallel loops by establishing parallel loops as one of the basic constructs in a program. The data, control or resource dependencies between loops are supported by describing programs as a graph, where nodes represent parallel loops, and arcs represent the dependencies between them.

TIDeFlow is a dataflow-base model. The decision to use dataflow as the foundation for TIDeFlow was based on the experience with the development of matrix multiply, where it became apparent that expressing arbitrary parallelism was required to produce an efficient implementation. TIDeFlow is similar to the dataflow model in that it uses actors, arcs and tokens to represent programs. TIDeFlow is different to dataflow, however, in that the user can control the generation of tokens or change the program graph while the program is executing.

Actors (Section 3.2) are used to represent parallel loops. Tokens (Section 3.3) are used as synchronization signals between parallel loops. Arcs (Section 3.3) express the dependence relations between parallel loops.

The mechanics of execution that govern TIDeFlow are very similar to those of dataflow [37]. An actor can execute (*fire*) when all its input dependencies have been met. In practice, this requirement is equivalent to a token being present in each of the actor's input arcs. When the actor fires, it consumes exactly one token from each input arc. When the actor finishes execution, it may (or may not) generate one output token per output arc.

Each part of a TIDeFlow program has an associated state to indicate whether it is executing, ready to be executed, or waiting for its dependencies to be met.

The following sections explain, in detail, the characteristics of a TIDeFlow program, with examples on how to express and execute programs.

3.2 Actors

The TIDeFlow execution model is based in the observation that *HPC programs are mostly composed of parallel loops*. For that reason, the parallel loops have been defined as the fundamental unit of computation, and they are represented by actors. Actors are similar to Macro Dataflow [98] in that they represent several sequential operations. Unlike macro dataflow, however, actors represent parallel loops rather than a serial piece of program.

The definition of each actor includes the name of a *codelet* that represents the instructions executed by each of the iterations of a parallel loop and an integer representing the number of iterations in the parallel loop.

The execution of actors is supported by their *actor state*, which contains information about the number of times the actor has been executed in the past and about its current eligibility for execution.

The following sections define actors as the basic computation unit and present the operational semantics of actors.

3.2.1 Actors Represent Parallel Loops

A careful study of several applications, including matrix multiply [50], Reverse Time Migration [85], and Fast Fourier Transform [80], reveals that most of the loops in these HPC programs are fully-parallel loops.

The loops in those programs are the result of computations that can be done in parallel, such as vector operations in matrix multiply, parallel updates in FFT or array updates in RTM. In other cases, parallel loops result from data prefetching (also called percolation [102, 46]) operations.

The philosophy of manycore processors is to achieve performance through parallelism. For that reason, it is natural to take advantage of the parallelism found in loops to supply work for processors. This new approach of using the inner iterations of loops as the basic computational unit was not possible before due to the memory and processing overhead of doing so. However, given the simplicity of hardware resources present in manycore processors, it is possible to conveniently express individual loop iterations as *tasks* that can be executed by processors.

Having individual loop iterations be the basic unit of computation is difficult, however. Previous execution models that have attempted to have single loop iterations executed as a standalone task have failed to provide implementations fast enough to support execution. These execution models include Static Dataflow [37], Dynamic Dataflow [10], Cilk [40], X10 [24, 112] and Habanero Java [22]. In Static Dataflow,

additional actors are required to control the execution of the loop iterations, and ultimately they restrict the available parallelism. Dynamic Dataflow made an advance on this issue by allowing a single actor to handle multiple tokens per arc, but it failed to completely solve the problem as the matching of tokens was still centralized. Cilk does not directly provide constructs for parallel loops, and it forces programs to rely on tree recursion to execute embarrassingly parallel loops. X10 and Habanero face problems that are similar to those of Cilk, where either parallelism is obtained through recursion, or it is limited by the ability of a single processor to create tasks for the parallel loop.

TIDeFlow avoids the problems of parallel loop execution by (1) defining the basic building blocks of a program as the parallel loop and (2) with a very fast, decentralized implementation of a runtime system that is responsible for the scheduling and assignment of tasks. This approach has the advantage that users have a powerful construct to represent most operations in HPC programs while simultaneously isolating the users from the difficulties and overhead of managing the tasks in a loop.

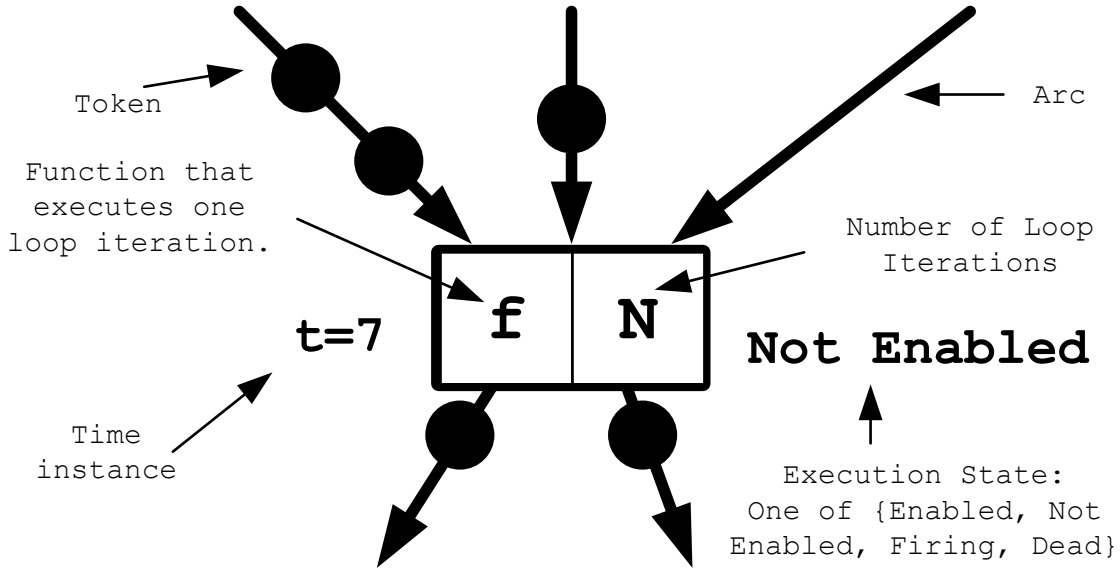
The basic construct used to represent parallel loops is the TIDeFlow *actor* (Figure 3.3). The parallel loop represented by an actor is expressed through the actor *properties*, which are constant: the number of iterations in the loop (N), a function f that contains the code to be executed by each iteration of the loop and a set of constant, user-defined values that are available to the user during execution.

Figure 3.1 shows the state and the properties of an actor in TIDeFlow. The user-defined data is not shown.

Figure 3.2 shows an excerpt of a code typical of HPC programs. The code of Figure 3.2 can be easily represented by a TIDeFlow actor as shown in Figure 3.3.

3.2.2 Execution of an Actor

The firing rules for an actor are taken from the dataflow model: An actor becomes enabled when there is a token in each one of its input arcs. When the actor fires, it consumes exactly one token from each one of its input arcs.



TIDeFlow actors represent parallel loops. The actor is described by a function representing the code of the parallel loop and by a number of loop iterations. To aid in the execution, an actor carries a state indicating how many times it has executed in the past (time instance) and whether or not it is ready to be executed (execution state). The firing rules of an actor are similar to those of dataflow: An actor becomes enabled when there is one token in each input arc.

Figure 3.1: Generic representation of a TIDeFlow actor.

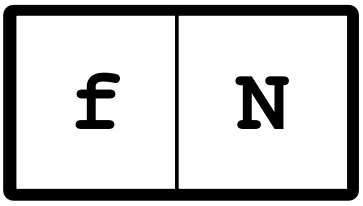
```
...
for t in 0 to T-1

  /* Parallel Loop */
  for i in 0 to N-1
    f(i,t);
  end for

end for
...
```

The basic unit of computation in TIDeFlow is the parallel loop. In the figure, the parallel loop is enclosed in an outer time loop. This configuration of parallel loop and outer time loop is frequent in HPC programs. A TIDeFlow actor is able to represent the parallel loop that iterates over `i`.

Figure 3.2: A parallel loop.



The actor of this figure is equivalent to the parallel loop of Figure 3.2. The code of the parallel loop is represented by function f. The number of iterations in the parallel loop are N. During execution, the runtime automatically creates parallel instances of the loop iterations and supplies the parameters to function f.

Figure 3.3: TIDeFlow actor.

TIDeFlow takes the stance that an actor with no input dependencies has all its dependencies automatically met because it has zero pending dependencies. The result of this policy is that actors with no input arcs are always eligible for execution unless they enter the *dead* state. A full description of the possible states of an actor is presented in Section 3.2.4.

An actor fires (executes) when it enters the *firing* state. When an actor fires, each one of the loop iterations represented by the actor are executed concurrently using the function f of the actor. When the actor fires, parallel, concurrent invocations of $f(i,t)$, $i = 0, ..., N-1$ (t is the time instance supplied by the runtime) are called. The actor finishes execution when *all* of the parallel invocations of the loop iterations end.

When an actor finishes execution (1) it generates a termination signal, (2) it may or may not generate output tokens and (3) it increases its time instance.

3.2.3 Signals Generated by Actors

An actor only generates signals when it finishes firing.

Termination Signals: When an actor finishes firing, exactly one termination signal is generated, and tokens in the output arcs may or may not be generated. The termination signals can be `CONTINUE`, `DISCONTINUE`, and `END`. `CONTINUE` indicates that the execution of the program should continue as normal, `DISCONTINUE` indicates that the actor should not be executed again, but the rest of the program should continue

and `END` indicates that the actor and all of its consumers should end. The kind of signals generated is decided by the user through the return value of the function f associated with the actor. An actor creates multiple instances of f when it fires. Given that f is written by the user, it is possible that not all instances of f created return the same signal. For that reason, the termination signal of an actor is defined as the return value of the instance that executes $f(0, t)$.

The termination signals can control whether or not an actor will execute again. Actors that emit a termination signal of `DISCONTINUE` or `END` do not execute again. This is accomplished by setting the actor's execution state to *dead*. A dead actor does not participate anymore in the computation.

Generated Tokens: The generation of tokens is subordinated to the termination signal. When `CONTINUE` is generated, exactly one token per output arc is generated and the token is tagged with the time instance with which the actor executed. `DISCONTINUE` is a termination signal that removes an actor from a graph along with its input and output dependency arcs. An actor that generates a `DISCONTINUE` signal is never fired again, and it does not need to produce any tokens because the actor and all its associated arcs have been removed from the graph. Finally, an `END` signal does not generate any output tokens in any output arcs.

The influence of the termination signals over the generation of tokens provides an indirect control over the execution of other actors. The reason is that an actor does not fire unless it has received a token in each one of its input arcs. If one or more of the tokens is not present, the actor will not fire. In that sense, once an actor stops producing tokens because it is dead, all actors downstream of it will eventually stop executing because they will not receive the tokens required to be enabled.

3.2.4 Actor States

The state held by actors has been designed to ease the execution and management of parallel HPC programs. The state of an actor is composed of:

- A time instance: An integer, initialized to zero, that increases its value by one whenever the actor finishes firing.

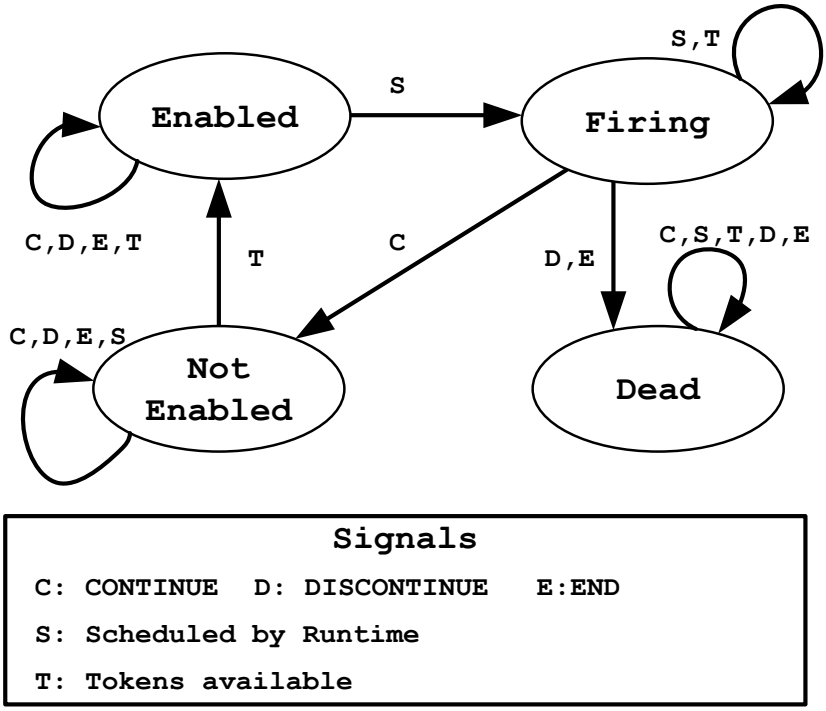


Figure 3.4: State Transitions for Actors.

- An execution state: An actor can be either *not enabled*, *enabled*, *fired* (executing), or *dead*.

Time instances tell how many times a particular actor has executed in the past. The time instance is useful because it has a one-to-one mapping to the iterators in the outer loops of an HPC program. For example, the time instance can be mapped to physical time in a program that simulates physical phenomena.

The time instance information is local to each parallel loop because parallel loops can execute asynchronously.

During the initialization of a program, the time instances of all actors are set to zero.

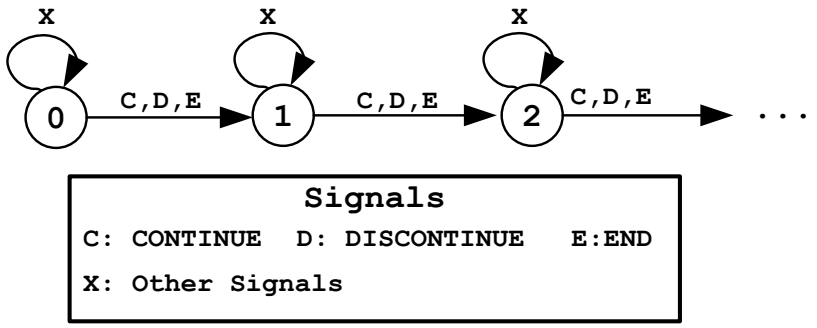


Figure 3.5: State Transitions for Time instances.

3.2.5 Actor Finite State Machine

The operational semantics that govern the execution of actors can be summarized with the state diagrams of Figures 3.4 and 3.5. The execution state of the actors is independent of the time instance of the actor. For that reason, separate Finite State Machine diagrams are given.

It should be noted that actors with no input dependencies can fire repeatedly if they produce a `CONTINUE` signal after they fire.

3.2.6 Examples

The following examples are a good way to illustrate the rules used to fire actors, to produce and to consume tokens.

3.2.6.1 A Hello World Example

Following the tradition of many programming languages, the introduction of the basic TIDeFlow constructs is done here by presenting a "Hello World" program.

Figures 3.6 and 3.7 show the two main components of a TIDeFlow program: A program graph (Figure 3.6) and a C file with code (Figure 3.7) for each one of the actors in the program. The code for each one of the actors in a program is called a *codelet*.

Figure 3.6: Program Graph for a Hello World program.

```
int64_t Hello( void * parameters, int it, int t )
{
  printf( "Hello World!\n" );
  return( END );
}
```

Figure 3.7: Codelets used in the Hello World program.

The TIDeFlow program of figure 3.6 will print the message `Hello, World!` and will end. Several observations about the program can be made:

- The `Hello` actor of Figure 3.6 is always enabled because it has not input dependencies.
- There is a total of *one* iteration of the `Hello` codelet in Figure 3.7. This is identified by the number in the actor.
- The codelet returns `END`. Causing the `Hello` actor to become dead after it executes.
- After the `Hello` actor becomes dead, the program ends because there are no actors that are enabled or running.
- The runtime system provides the actor with information about its state: The loop iteration (`it`), the time instance (`t`) and a pointer to user-defined parameters (`parameters`).

The output of the Hello World program is given in Figure 3.8.

An important observation of the Hello World program is that the user controls the termination of the program through the return value of the `Hello` codelet.

```
Hello, World!
```

Figure 3.8: Output of the Hello World program.

```
Hello, World!
Hello, World!
Hello, World!
Hello, World!
...
```

The sole actor in the Hello World program returns `CONTINUE`, making it elegible to be executed again. Because the actor has no input dependencies, it becomes enabled repeatedly, and executes several times.

Figure 3.9: Output of the Hello World program.

If the return value of the `Hello` codelet is changed to `CONTINUE`, the `Hello` codelet will become enabled again after each execution, repeatedly printing the `Hello, World!` message as shown in Figure 3.9

3.2.6.2 Iterations and Time Instances

In HPC programs, it is very frequent that the same piece of code is executed repeatedly. For example, in programs that simulate physical phenomena, frequently the same computation will be performed during each timestep, and it may even operate on the same data.

Even HPC applications that do not simulate physical phenomena have outer loops that are very similar to the "time loop" used in other physics programs. To cite an example, the matrix multiply program discussed in Figure 3.21 has an outer time loop that comprises the computation of a tile.

For simplicity, outer loops that encompass several parallel loops will be referred to as *time loops*.

The iterator information provided by the time loop is used by the application to make decisions during execution: It may determine the memory location that should be used, or it may modify the computation done. As such, this iterator must be included as part of the state of an actor, and it should be be available during execution. The time loop iterator, referred to as the *time instance*, is part of the state, and it is provided by the runtime as a parameter to the actor's function.

Figure 3.10: A TIDeFlow program graph.

```
int64_t Hello( void * parameters, int it, int t )
{
  printf( "it=%d, t=%d\n" );
  return( CONTINUE );
}
```

Figure 3.11: Codelets for the program of Figure 3.10.

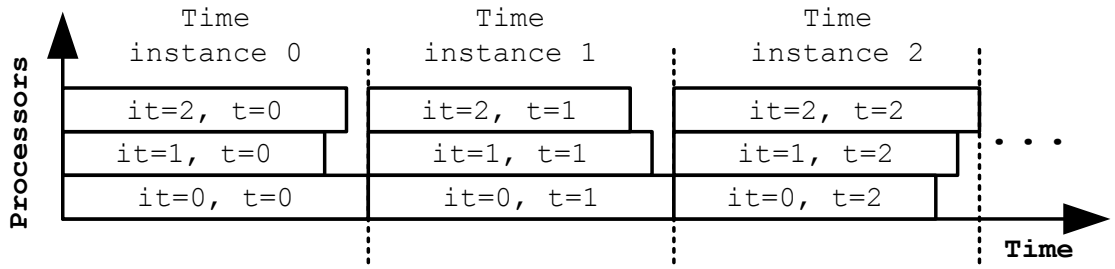
The concept of loop iterations and time instances is illustrated in Figures 3.10 through 3.12. The Hello World program has been modified to show the use of parallel loops. Figure 3.10 shows a TIDeFlow program graph where the actor `Hello` is executed three times in parallel. Figure 3.13 shows a possible output from the program, while Figure 3.12 shows an execution trace that matches the output of the example.

The sole actor of the program in Figure 3.10 is enabled by default, and it can fire repeatedly because it has no input arcs. When the actor fires, the runtime system will inspect how many parallel iterations are specified by the actor (3 in the case of this example), and it will create a task for each one of the iterations. The time instance – represented in the example as the variable `t` – is a part of the state of the actor, and it is passed to the function invocations by the runtime system during execution.

Figure 3.12 shows a possible trace of the execution. The figure shows that (1) 3 parallel loop iterations are created at each time instance, (2) execution of the actor finishes when *all* of the parallel loop iterations finish and (3) the time instance of the actor is advanced when the actor finishes execution and becomes enabled again.

3.2.6.3 Termination Signals

A slightly modified version of the Hello World program, shown in Figure 3.14 and its output (Figure 3.15), can be used to show how to use termination signals.



Execution trace of the program of Figures 3.10 and 3.11. Note that all parallel iterations in a time instance must finish before the time instance is advanced.

Figure 3.12: Execution trace of the Hello World program.

```
it = 1, t = 0
it = 2, t = 0
it = 0, t = 0
it = 2, t = 1
it = 1, t = 1
it = 0, t = 1
it = 0, t = 2
it = 1, t = 2
it = 2, t = 2
```

Output of a parallel Hello World program. Both the time instances and the loop iterations are used. Note that no ordering or synchronization between loop iterations within the same time instance are supported.

Figure 3.13: Output of the parallel Hello World program.

```
int64_t Hello( void * parameters, int it, int t )
{
  int t;

  printf( "Hello World at t = %d\n", t );

  if ( t == 4 )
    return( END );

  return( CONTINUE );
}
```

This example shows how to use termination signals to control the execution of the code: The time instance is used to decide whether or not to terminate the program by generating an `END` signal or to continue execution by generating a `CONTINUE` signal.

Figure 3.14: Use of signals from the user code to control execution.

```
Hello, World at t = 0
Hello, World at t = 1
Hello, World at t = 2
Hello, World at t = 3
Hello, World at t = 4
```

The output of the Hello World program of Figure 3.14 is shown here. The time instances and termination signals have been used to end the program at iteration $t = 4$.

Figure 3.15: Sample output of a TIDeFlow program that runs for a fixed number of iterations.

Note that the sole actor of the Hello World program is executed repeatedly because its codelet returns `CONTINUE` during execution. The program ends when the codelet returns *END* at time instance 4, changing the state of the sole actor to *dead* and ending the program.

Time instances are useful to distinguish time iterations in programs. They can be used to take decisions about the termination of programs, as in Figure 3.14.

3.3 Arcs and Tokens

When an actor finishes execution, it may signal other actors that depend on it by creating tokens. Tokens do not carry data, they only convey the meaning of a dependence met from one actor to another. Data is passed between actors through shared memory. This is similar to the EARTH model of computation [104].

3.3.1 Arcs Represent Dependencies Between Parallel Loops.

The arcs in TIDeFlow graphs provide a simple way to express dependencies between actors.

Dependencies in TIDeFlow typically represent data dependencies in producer-consumer scenarios but they may also represent resource or control dependencies.

Arcs are allowed to carry an unbounded number of tokens, although particular implementations can restrict the number of tokens to a certain maximum.

3.3.2 Representing Outer Loop Carried Dependencies

Although many inner loops in HPC programs are embarrassingly parallel, the iterations of outer loops can not, in general, be executed in parallel. The reason is that the outer loops of an HPC program usually express the high level relationships between computation stages in a program. In most cases, the outer loops in an HPC program capture the causality relationships between part of the program. They can represent a time-ordering, or a sequence of communication and computation and so on.

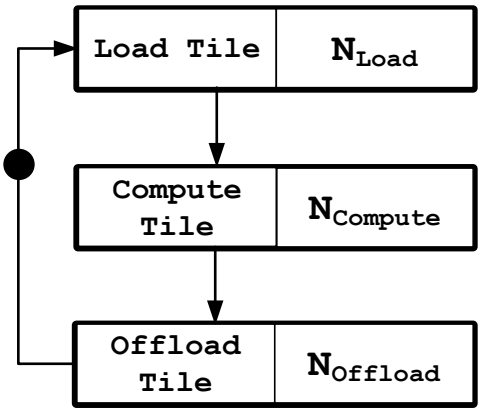


Figure 3.16: Dependencies between memory movement and computation in a tiled application.

Consider the case of an application where tiling has been employed to improve locality. When optimized, the tiling approach must be accompanied by matching memory movement and by successful memory management. In the simplest configuration, a buffer will be used. The buffer is used for computation with the aim of improving locality. Given the nature of on-chip memories in manycore processors, memory movement is explicit to and from the buffer. Figure 3.16 shows in detail the dependencies of such an approach. The computation section must finish before the results are offloaded to memory. Computation must wait for the results to be loaded to memory. However, there is a *loop carried dependency* between offloading of results and loading of the next data block: The *next* loading operation must wait until the results of the *previous* tile computation have finished. These outer loop carried dependencies are similar to traditional loop carried dependencies in that they refer to dependencies between different iterations of a loop. However, the outer loop carried dependencies not only express dependencies between individual memory accesses by the loops but also the conceptual (control, data, resource) dependencies between them. The dependency between offloading the data in a buffer and using the buffer again in the next iteration is represented by a backwards arc in the program. To allow execution of the first actor, the backwards edge is initialized with one token.

The dependencies of Figure 3.17 represent traditional data or resource dependencies in the traditional sense that an actor depends on the results immediately produced by another actor.

The concept of dependencies can be extended to allow dependencies between different time instances. These dependencies are referred to as *outer loop carried dependencies*. The name comes from the fact that there exists a loop carried dependency present in the outer loop. It should be pointed out that innermost loop carried dependencies are not possible in TIDeFlow because no data communication or synchronization is supported between loop iterations within the same parallel loop.

The dependence distance in outer loop carried dependencies is controlled by the number of tokens placed in the arc at the start of the program. In general, an outer loop carried dependency of distance k between an actor A and an actor B can be represented by placing k initial tokens in the arc that connects A and B. The additional number of tokens regulate the execution in such a manner that time instance $t + k$ of B will wait for a token produced by A at time instance t.

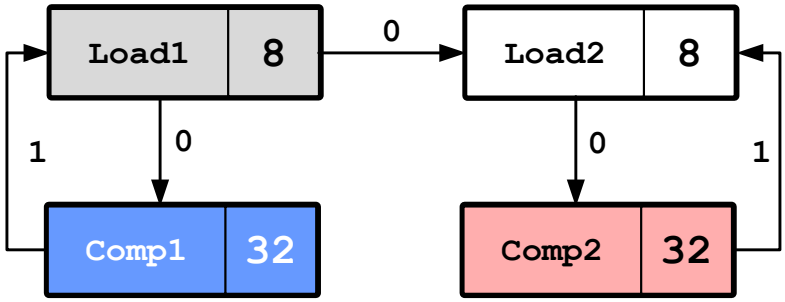
The examples in Section 3.3.3 are helpful in understanding the meaning and the use of outer loop dependencies.

3.3.3 Examples

3.3.3.1 Overlapping Communication and Computation

Tiling [60], the common construct for memory locality, can be used to illustrate the use of dependencies. When tiling is used in a program, some memory is loaded from lower levels of the memory hierarchy into on-chip memory. Once the memory has been loaded, the processors can work on it.

In the particular case of an application that uses tiling in C64, two buffers are used to overlap communication and computation: Computation can be done on one buffer while memory movement is done on the other. Figure 3.17 shows a C64 program that will optimize the use of memory bandwidth and processor resources. The number of loop iterations in the loader codelets is 8 because it takes exactly 8 threads



A simple TIDeFlow construct where some overlapping of communication and computation can be observed. `Load1` and `Load2` perform loads from main memory into on-chip memory. `Comp1` and `Comp2` perform computation on the data loaded. Note the dependencies: Computation must proceed only after a load. Two loads may not proceed in parallel because there is not enough memory bandwidth, hence the dependency between `Load1` and `Load2`. The initial tokens in the arcs between the computation and the load actors enforce a loop carried dependency of distance 1.

Figure 3.17: TIDeFlow construct for overlapping of communication and computation.

to saturate the memory bandwidth of C64 when all of them are performing off-chip memory operations. The dependency between the loaders and their respective compute actors are data dependencies; they indicate that computation can only proceed when the load has completed. Of particular interest is the dependency between the loader actors that indicates a resource dependency: Two loader actors can not execute at the same time because there is not enough available memory bandwidth.

Figure 3.18 shows a possible execution trace for the program of Figure 3.17. To simplify the trace example, it has been assumed that only 16 processors participate in the computation.

3.3.3.2 Using Outer Loop Carried Dependencies

The use of loop carried dependencies can be illustrated through an application where several buffers are available. Each buffer is used to hold some data that needs to be computed as a tile [60].

Processors can asynchronously do memory movement to put memory into the buffers. In that way, more than one buffer can be ready for execution, and more than

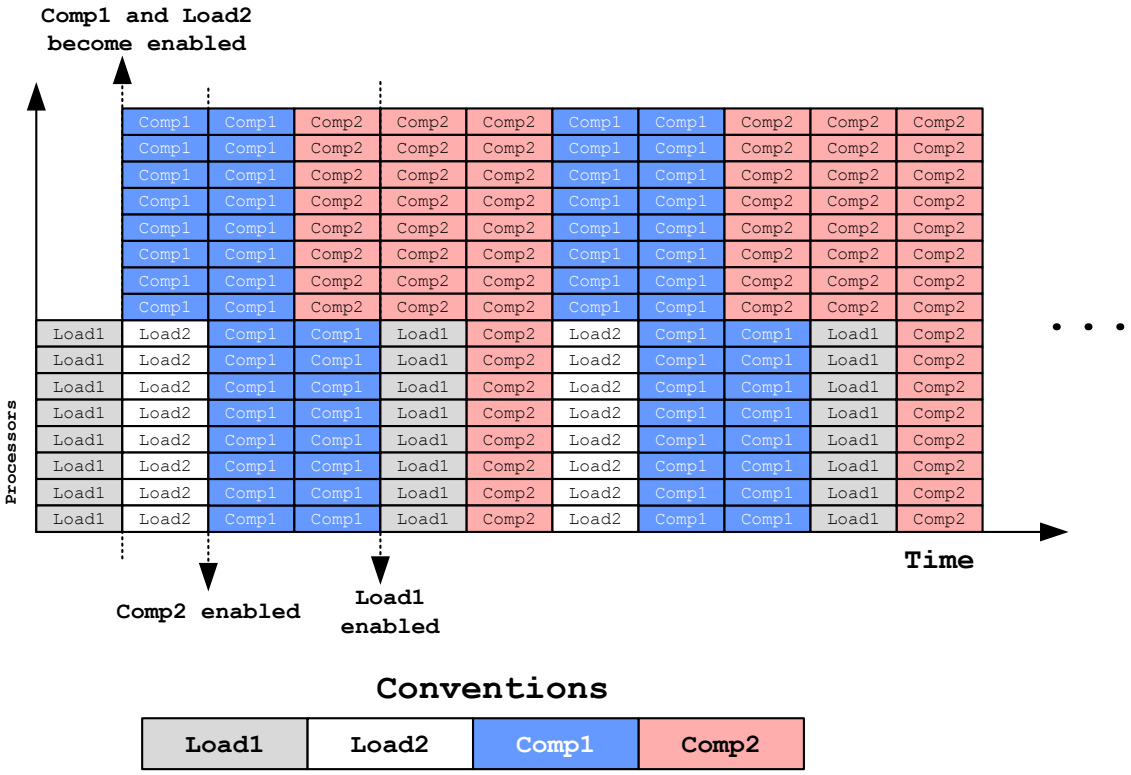


Figure 3.18: A possible execution trace for the program of Figure 3.17.

one memory transfer, either from main memory or to main memory, can be happening at the same time. This approach is useful when the computation of a tile takes an unpredictable amount of time, since it allows slow computation of some tiles to be amortized over several tiles.

Now, let's consider the dependencies between the operations. Three main operations are done: (1) Computation of a tile, (2) prefetching the data needed by a tile and (3) offloading the data computed by a tile.

The computation of a tile can proceed only after the data prefetching for that tile has completed. For that reason, there is a dependency between the memory movement (percolation) [46] required and the computation of the tile. In a similar way, offloading the data computed can only happen once the computation of the tile has finished, so there is a dependence from the computation step to the offloading step. However, there is not an immediate dependency between offloading of a tile and loading of the next

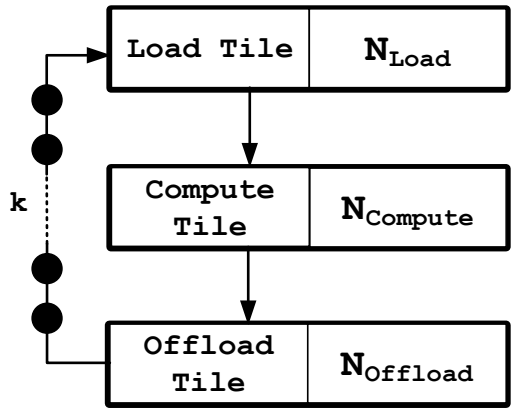


Figure 3.19: Dependencies between memory movement and computation in a tiled application.

tile. In fact, if there is a total of k buffers, a particular buffer is only reused after other $k-1$ buffers have been used.

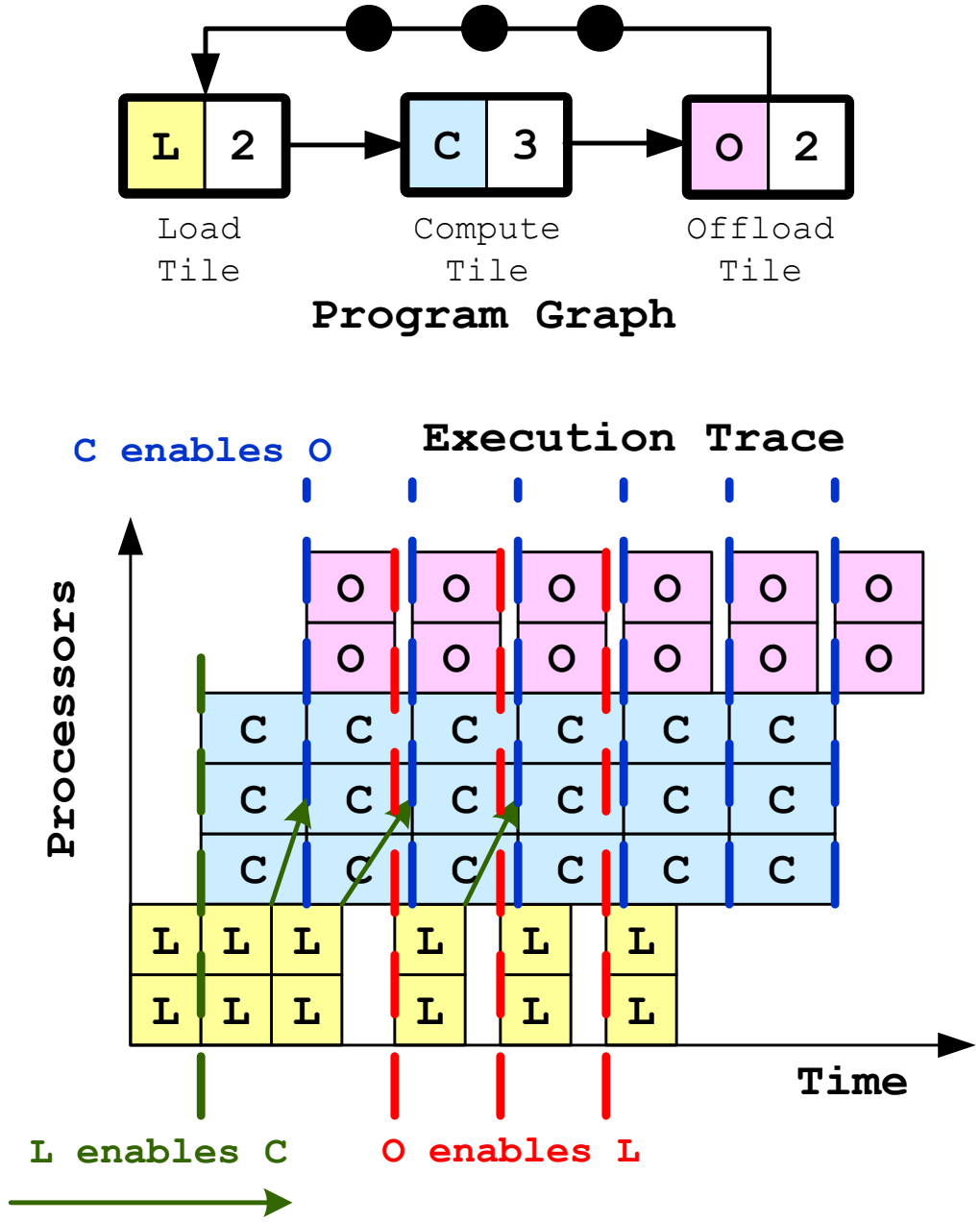
For this reason, there is an *outer loop carried dependency* between offloading a buffer and prefetching the same buffer with *dependence distance* of k. Figure 3.19 shows the dependency graph for this situation.

3.3.3.3 Expressing Pipelining Through Backedges

The program of Figure 3.20 presents an example of how backedges in programs cause pipelined execution.

The program represents a typical tiled computation where three buffers are available. First, buffers are loaded with data, then, computation is performed using the data loaded, followed by an unloading of the data.

As can be observed in the figure, the backedge between the `O` actor and the `L` actor restrains the speed at which `L` can execute, ultimately resulting in an optimal pipeline. The resulting pipelined execution was possible, in this case, because there were enough processors to execute the actors as they become available.



Execution trace of an N-Buffer tiled computation. The execution assumes that infinite processors are available and that all of them execute at the same speed. Note that the system naturally falls into optimal pipelining.

Figure 3.20: Execution trace of a pipelined program.

3.3.3.4 A Matrix Multiplication Kernel

Figures 3.21 and 3.22 show one of the ways to implement the inner kernel of matrix multiply presented by Garcia et al. [50]. A high level pseudocode for the computation of the tiles is given in Figure 3.21 and its matching TIDeFlow graph is shown in Figure 3.22.

The implementation uses two buffers in on-chip memory. A loop carried dependency of distance 2 between the `BufferLoader` actor and the `BufferOffloader` actor allows overlapping of communication and computation between the two buffers used in the program.

Because all the innermost parallel loops are embarrassingly parallel, they have been expressed as single TIDeFlow actors.

3.3.3.5 A Program Where Actors Execute Only Once

The use of the `DISCONTINUE` signal can be illustrated with the example of Figures 3.23 and 3.24.

The actors in the program of Figure 3.24 produce a *DISCONTINUE* signal when they finish execution, effectively removing the actors from the program graph. The act of removing some actors from a program may result in enabling other actors, as is the case in Figure 3.24. The overall effect is that a TIDeFlow program with an execution similar to that of a serial program can be achieved.

3.4 Composability

Smaller programs can be used to build larger programs. This powerful property allows modularity, and it gives the programmer useful abstractions that are not tied to implementation details such as the number of processors available.

Composability is achieved by allowing small programs to be seen as actors in the larger programs that use them. To follow the conventions, the actor that represents the small program is defined to have only $N = 1$ iterations because the small program

```
...
/* Globals */
int buffer;
...
for step in 1 to NumSteps

  /* Select which buffer to use */
  buffer = step % 2;

  /* Load buffer */
  for i in 0 to 7
   BufferLoader(i)
  end

  /* Innermost loop: Computes one tile */
  for i in 0 to 1023
     SubTile(i);
  end for

  /* Offload buffer */
  for i in 0 to 7
   BufferOffloader(i)
  end

end for
...
```

Pseudocode for a tiled, serial, matrix multiplication computation using two buffers. Although the code shown is serial, it could be parallelized and pipelined.

Figure 3.21: Serial code for matrix multiplication.

is only executed once when it becomes enabled. Additionaly, by definition, small programs return a `CONTINUE` signal when they complete.

TIDeFlow programs can only be constructed from existing programs. This design choice precludes the existence of recursive calls because a program can not be used as part of itself.

TIDeFlow programs enjoy the advantages of composability. For example, when a small program is used in two different parts of a larger program, no interference is generated. Each instance of a program generates its own local state that exists for the duration of the program only, and at the execution level they do not interact with

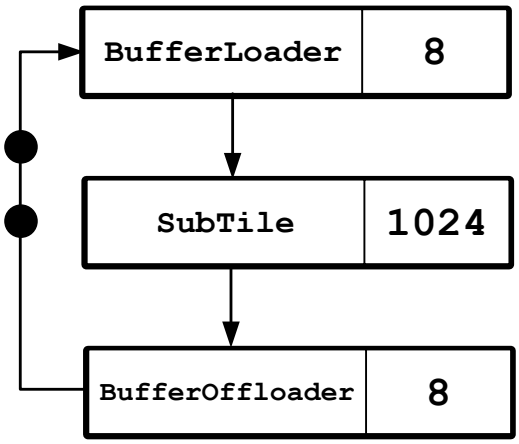


Figure 3.22: Representation of a tiled matrix multiplication using TIDeFlow.

```
...
x = malloc( ... );

for i in 0 to N-1
  x[i] = 0;

for i in 0 to N-1
  x[i]++;

free( x );
```

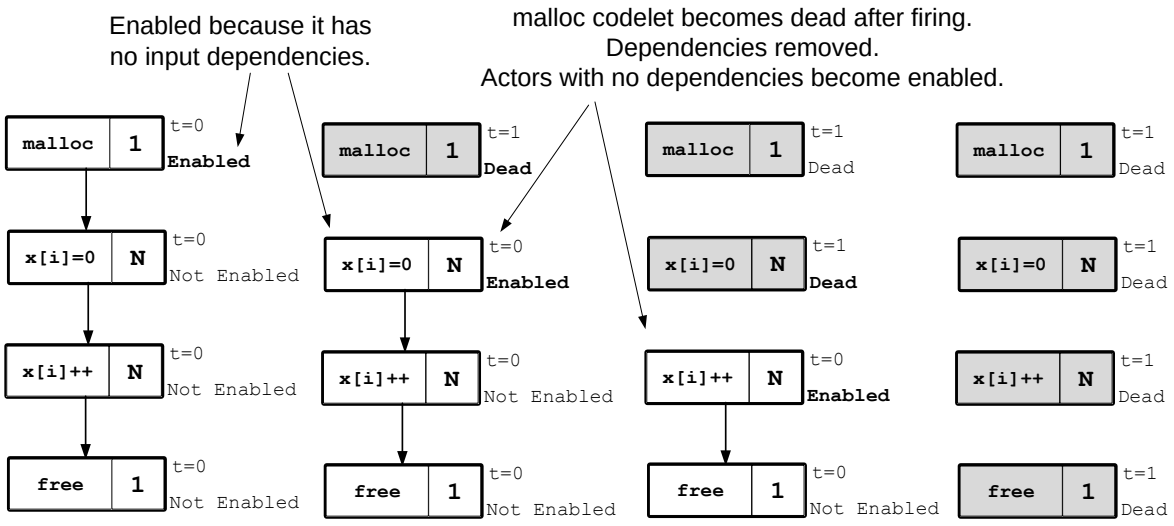
Figure 3.23: Example of a program with sequential statements.

other instances of the program.

The composability property is very powerful since it allows modularity and portability. Modularity allows the development of parallel programs that are used to perform common operations. Portability allows the use of the program in other systems regardless of the number of processors available. Examples of programs that can be reused include Matrix Multiplication, memory movement, Fast Fourier Transforms and so on.

The rules to execute TIDeFlow composable programs can be expressed in terms of the rules used to execute programs:

- All enabled actors may execute when a program begins execution.



`DISCONTINUE` can be used to support serial programs with no outer time loops. In the figure, all actors return `DISCONTINUE`, resulting in serial execution of the program.

Figure 3.24: TIDeFlow program graph for the the program of Figure 3.23.

- A program terminates when no actor is firing, and no more actors can become enabled in the future.

The rules for execution of an actor A that represents a program P are similar: The program P starts execution when the actor A fires. The actor A completes execution when the program P finishes.

The example of Figure 3.25 shows how to use programs as part of larger programs.

3.5 Task Pipelining

The presence of weighted loops in TIDeFlow program graphs allows for natural task pipelining during execution. Figure 3.20 shows that during execution, with infinite processors, the computation adjusts itself to the optimal pipelined schedule. This result had also been demonstrated by Dynamic Dataflow in the past.

The ability to express task pipelining constructs at the level of the program graph is a powerful feature that can significantly simplify the optimization of HPC programs. For example, in the matrix multiplication program optimized by Garcia [50],

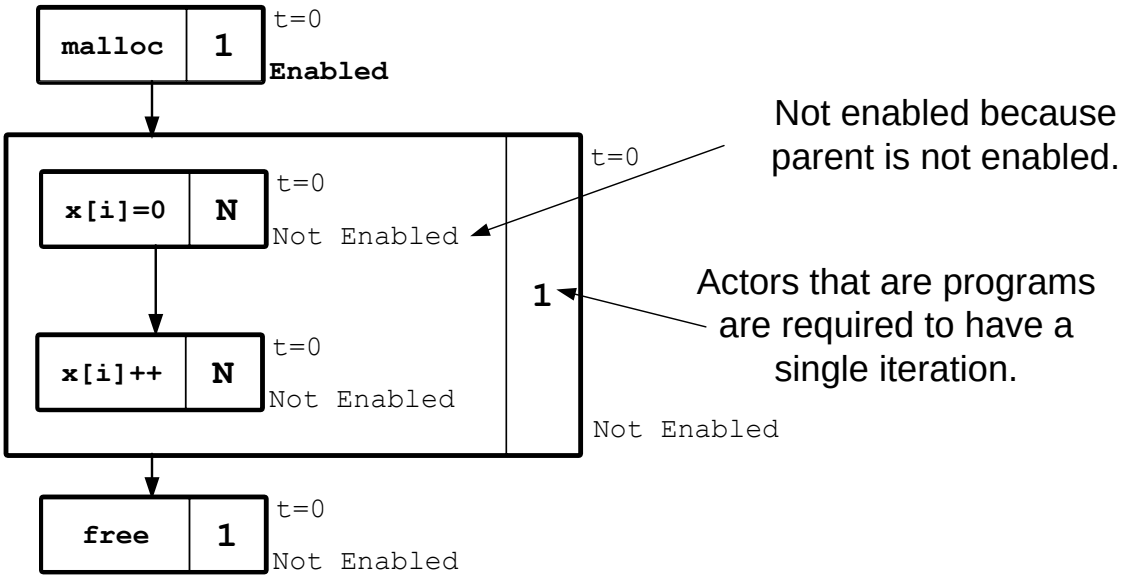


Figure 3.25: Example of how to use small TIDeFlow programs to construct larger ones.

a significant amount of time and effort was devoted to designing and implementing a good strategy for task pipelining. Garcia's efforts included the development of hand-made synchronization primitives as well as a synchronization plan that depended on experience gained from previous program traces. Garcia's approach, altough effective, is cumbersome, time consuming, and error prone. The details of Garcia's efforts can be found in several publications [46, 50, 47, 44].

An equivalent TIDeFlow program to compute the same matrix multiplication would not require such a titanic effort. The weighted arcs in the program graph can indicate the dependence relations between computational steps, leaving the job of scheduling them to the TIDeFlow runtime system.

Although most of the task pipelining duties are left to the runtime, a programmer can still benefit from execution traces and profiling to optimize programs in situations with limited amounts of resources. For example, the dependence relationships available in TIDeFlow can be used to allocate memory bandwith to processors as shown by the example of Section 6.2.

The use of priorities can further improve the effectiveness of pipelining during execution. Priorities are a mechanism through which the programmer can identify the tasks that are most likely to be in the critical path of execution. By identifying these tasks, the runtime system can schedule them whenever they become enabled, preventing stalls when possible. Correct results are still guaranteed because dependencies are still enforced.

Figures 3.17 and 3.18 show an excellent example of a situation where priorities can be modified to avoid stalls during execution. The tasks that form the critical path for the program of Figure 3.17 are the loader actors, which have been set to have high priority. During the execution (Figure 3.18), the loaders are executed as soon as possible, enabling the next set of computations and preventing stalls.

Experiments with the TIDeFlow system have shown that only two levels of priority (low and high) are enough to control the scheduling during execution.

3.6 Memory Model

This section strives to present an intuitive explanation of the TIDeFlow memory model. The memory model of TIDeFlow has been designed to provide useful constructs for programmers while at the same time allowing simple practical implementations in many-core systems.

Seeking simplicity of implementation and design, the TIDeFlow model uses shared memory as the main mechanism for data communication. This decision facilitates communication between actors at the expense of the necessity of additional rules to avoid race conditions.

The following rules form the core of the TIDeFlow memory model.

Rule 0: A TIDeFlow system has shared memory. All processors have access to all the shared memory. Processors can allocate and deallocate memory for their use or for use by other processors. Global variables are allowed.

Rule 0 specifies that TIDeFlow is a *shared memory system*. And all communication is done through memory.

Rule 1: Memory operations made by a loop iteration appear to complete in program order to the processor that issued them. No ordering is guaranteed between memory operations issued by different processors.

Execution of each one of the loop iterations that compose an actor appears serial. Rule 1 supports serial execution of individual loop iterations. Rule 1 does not provide any limitations between memory accesses made by two different processors.

Rule 1 does not specify what happens when loop iterations that belong to the same actor try to access the same memory location. Rule 1 assumes that actors represent parallel loops without data races and not other kinds of loops.

Note that the TIDeFlow model does not allow data sharing between iterations that belong in the same parallel loop. Attempts to share data or to build synchronization constructs may result in undefined behavior.

Rule 2: If there is a dependency from an actor A to an actor B, and A produces one token h at A's time instance k, then, when B consumes the token h, B observes all memory operations of A at time t as complete.

Rule 2 specifies that all memory operations from an actor will complete once the actor completes, and that the data produced will be available to other actors. Rule 2 is the main mechanism for orchestrating data sharing.

Rule 3: All memory operations in a program must have completed when the program ends.

Rule 3 supports composability. All memory operations of a program will complete once the program completes. This rule ensures that actors depending on data produced by a program (that was used as an actor) will have full access to all the memory produced by the program.

Chapter 4

A THROUGHPUT ANALYSIS TO SUPPORT PARALLEL EXECUTION IN MANYCORE PROCESSORS

The driving issues in concurrent algorithm development have radically changed from execution in an environment dominated by virtual parallelism to an environment dominated by massive hardware parallelism.

The change from virtual parallelism to hardware parallelism has decreased the importance of algorithm properties that specify whether or not the algorithm is non-blocking, lock-free, or wait free. Instead, the large number of processors in manycore systems obtain a greater benefit from higher levels of scalability rather than from theoretical properties about thread failure.

The following sections provide an argument in support of *throughput*, the maximum intrinsic rate of completion of parallel operations over traditional properties.

This chapter shows that the throughput of an operation is a function of the architecture used to implement the operation as well as the algorithm and technique used. This work is focused on the manycore-class of architectures. In order to provide a solid discussion, the explanations, examples, and conclusions presented use Cyclops-64. This is not a limitation since the principles of throughput analysis apply to any architecture. Cyclops-64 was used because it is a simple architecture where the effects of throughput can be illustrated easily, both in terms of the simplicity of the examples and their matching experiments. Future work on throughput will test the validity of throughput on other architectures, developing new examples and algorithms targeted to them.

The relevance of throughput in the particular case of queue algorithms is presented due to their importance to runtime system development. The exposition is done within the framework of queueing theory.

Several design guidelines are developed that result in the development of fast, concurrent queue algorithms, along with experimental information showing that intrinsic throughput is fundamentally linked to scalability of parallel programs. The chapter ends by presenting experimental evidence showing that a throughput-oriented approach does result in improved scalability for parallel programs in manycore processors.

4.1 The Importance of Throughput in Parallel Programs

This section explains the importance of throughput in algorithms, and the consequences of low throughput in parallel programs. A simple example, that shows the main issues present, is developed to illustrate the characteristics of the problems and how they can be analyzed.

Analysis of the scalability of different implementations of the parallel program reveals interesting results about the implementation of algorithms.

Finding the maximum scalability of the parallel count program requires, first, a definition of scalability, and second, an analysis of the parallel count program within the framework of that definition.

For the purposes of this discussion, scalability will be defined as the ratio of the maximum performance of a parallel program to the performance of a serial version of the program. It is of interest as well, to find the number of processors used to obtain the maximum scalability possible. Scalability can be defined as the ratio of the performance of a parallel program to a corresponding serial program. The parallel performance obtained (and the associated scalability that it allows) will be shown to be related to the implementation of the program and the features of the architecture used to run it.

The conclusion of this analysis is that the scalability of a program is directly related to the intrinsic ability of an operation to be executed in parallel. The intrinsic

rate at which an operation can be executed is the *throughput* of that operation.

The example of a parallel count program can be used to illustrate the meaning of throughput. In the parallel count program, an infinite number of processors attempt to increment a shared variable x. The only operation that processors make is the increment of x. When a particular processor increments x, it goes idle and does no other operations. In the parallel count program, all processors have shared access to variable x and the memory is sequentially consistent. The throughput of the program can be understood as the *rate* at which variable x is incremented.

The concept of throughput is important because it bears a direct relationship to the limits in the available parallelism of programs.

The throughput of a parallel operation is a constant value that does not depend on how many processors there are. The throughput of a parallel operation only depends on the algorithm used to implement the operation and the architecture used to execute the algorithm.

Throughput analysis is of particular importance to manycore systems. Manycore architectures are particularly sensitive to low throughput because an application with low throughput limits the maximum number of processors that can be used concurrently.

The problem can be easily illustrated with the critical case of concurrent queues in manycore processors. Queues are at the heart of the scheduling system of TIDeFlow and many other execution models, including Cilk, EARTH, Habanero and X10.

The problem with the throughput of concurrent queues is that if a queue has a throughput of μ operations per second in a system with P processors, it can serve a maximum of 1 queue operation every μ^{-1} cycles, which in turn, limits the processor request rate λ to at most:

$$\lambda^{-1} = \mu^{-1} P \tag{4.1}$$

Consider the case of C64 where 160 processor cores ($P = 160$) concurrently use an MS-Queue, and where each memory access to shared memory takes 30 cycles in the

best case ($m = 30$). Under those conditions, each individual processor core is limited to issue at most one queue request every 4800 cycles in the most optimistic scenario if low latency at the queue is desired.

Traditional queues severely limit the usability of queues as a basic parallel construct, since for many applications that use queues, the workload associated with a queue is already in the range of few thousand cycles: Our experiments show examples of two applications where each processor uses the queue, in average, every 10000 cycles. The limitation is given specifically by the product $\mu^{-1}P$. Eq. 4.1 provides another insight in the importance of throughput for extreme scale algorithms.

4.2 Queueing Theory and its Relationship to Throughput

Queueing theory is a mature field which can be used to analyze the behavior of systems where a number of agents attempt to obtain services from servers. An excellent introductory work on queueing theory can be found in Kleinrock's 1975 textbook [63].

Queueing theory is relevant to throughput analysis because programs with concurrent operations can be modeled as agents trying to access services. In the context of concurrent operations, processors can be modeled as agents and execution of a concurrent operation can be seen as a service. Processors are modeled as agents because they generate requests for shared resources whenever they perform concurrent operations. These shared resources can include algorithm and hardware constructs such as access to locks, memory and bandwidth. Resources are, in turn, provided by servers, which take the form of circuits, functional units or even small sections of programs.

The great advantage of using queueing theory is that it allows predictions about the performance of a system. It can model latencies, throughput, idle times and so on. In particular, results that apply to general situations have been obtained. These results provide intuition about the behavior of systems.

Queue algorithms are of paramount importance to scheduling systems. And as illustrated with the parallel count example, the ability of an algorithm to support many concurrent operations simultaneously directly depends on the throughput of

the algorithm. Queueing theory can support the development of queues by providing quantitative means of evaluating the effectiveness of particular queue implementations.

Queueing theory, for example, provides tools to predict the latency of operations in an agent-server system. An important conclusion that holds for the general case is that the latency of operations increases when the rate of incoming requests approaches the intrinsic throughput of the system.

The throughput of operations was not a determinant issue in the past because it was very rare that enough requests could be provided to saturate the intrinsic throughput of simple algorithms and operations.

However, current trends in computer architecture suggest that more and more processors will be used for computations, stressing the need for concurrent algorithms with throughputs that allow scaling to unprecedented numbers of processors.

The following sections provide definitions pertaining to queueing theory. These definitions are used when measuring the throughput of each algorithm presented in the examples.

Queueing Theory Conventions

The following conventions are useful to analyze the throughput of algorithms and operations:

- μ is the *throughput* of an algorithm or operation: It describes the maximum number of requests that can be serviced per unit of time.
- P is the number of processors attempting to access a service. Typically, the service consists of execution of a concurrent operation.
- r is the average amount of time taken between requests made by a particular processor.
- λ is the average request arrival rate as observed by the server. For example, in the parallel count program, λ is the observed requests for increments, per unit of time, made by all processors. When services are provided with low latency, λ can be approximated as $\lambda \approx P/r$
- $\rho = \lambda/\mu$ is the *utilization factor* of the queueing system.

- m is the average latency of a single memory operation in a processor.
- k is the amount of time (measured in cycles) that an atomic operation uses at a memory controller. This parameter arises from the fact that some processors, such as C64, have the ability to execute some operations at the memory controller without help from processors.
- z is the number of cycles a memory read or write uses the memory controller. In general $z < k$ because the memory controller needs to do less work to complete a normal write (or read) than to compute an atomic operation.
- O refers to a particular operation or algorithm.

By definition, μ, the throughput of an operation O, specifies the maximum number of operations of type O that can be completed per unit of time. μ sets a bound on the intrinsic parallelism on the operation: The operation will scale until the request rate λ reaches the throughput μ because the operation can not service more than μ requests per unit of time.

A queueing system is defined as stable if $\mu > \lambda$, or $\rho < 1$. When $\rho > 1$, requests entering the queueing system accumulate faster than they can be served and, in theory, latency increases to infinity. In practice, the system saturates, limiting the request rate to be $\lambda = \mu$, (or $\rho = 1$) and stabilizes the system by meeting requests with large latencies.

Analyzing with precision the relationship between latencies, throughput and request arrival rates requires knowledge of the probability distribution function of the arrival and service rates of the queueing system. In the context of this work, it suffices to say that in general, as λ approaches μ (and ρ approaches 1) the waiting time at the queue *increases*.

4.3 Techniques to Increase the Throughput of Parallel Operations

The *"inquire-then-update"* approach is one of the main throughput limitations in current queue implementations: In order to succeed, the queue must be locked during at least 2 memory roundtrips in the case of a locking implementation, or the queue

must remain unchanged during at least 2 memory roundtrips for implementations using Compare and Swap (See Section 4.4.3.2).

A surprising result from Sections 4.4.3.1 and 4.4.3.2 is that nonblocking implementations and lock-based implementations of queues in non-preemptive environments have throughputs that are in the same order of magnitude.

Instead, a stronger stance can be sought: Queue operations (enqueue, dequeue) should succeed *immediately* if they can succeed at all. The word immediately is used in the context of not requiring multiple operations, instead, the queue structure should be changed with only *one* memory operation. In this sense, processors trying to access the queue will directly write to the queue, without first reading the state of the queue. This important distinction allows a significantly greater queue throughput than the throughput provided by an inquire-then-update approach, because changes to the queue happen during the time the memory controller serves the memory operation *in memory* as opposed to happening over the course of several memory roundtrips.

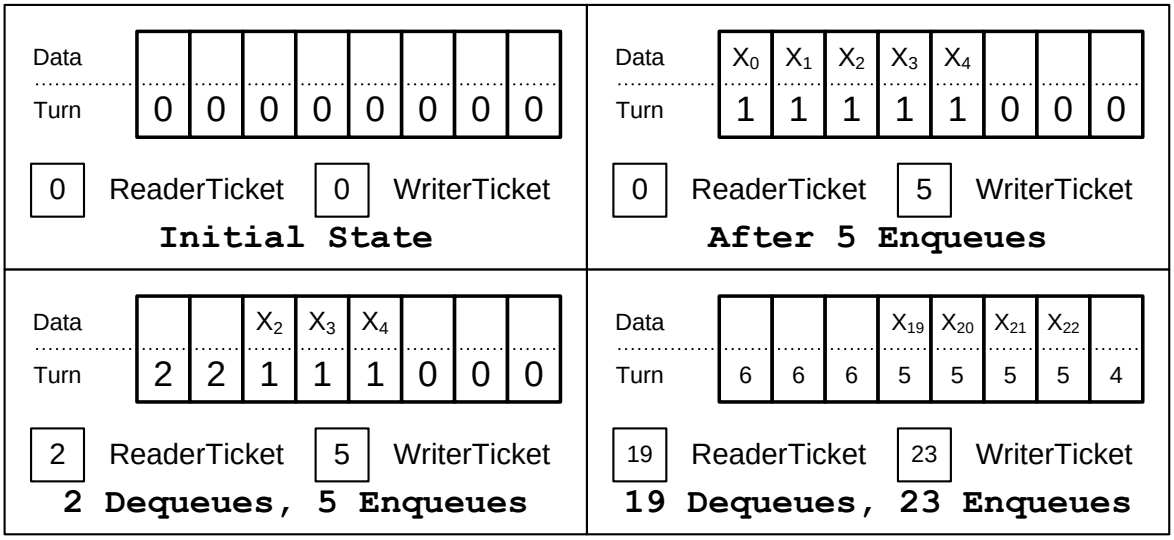
The inquire-then-update is avoided by constructing the queue as an array of queue elements, in which a positive integer can be associated to a position in the array. Processors performing enqueue or dequeue operations can claim positions in the array using a single atomic increment without exclusive access to the queue during a certain number of memory roundtrips.

The idea of using atomic in-memory increments can be successfully used to construct queue algorithms with significantly faster throughput. These algorithms, explained in detail in a previous publication [85], work by cleverly conducting most concurrent operations as in-memory atomic additions.

Figures 4.1 and 4.2 show the two main algorithms developed.

Circular Buffer Queue (CB-Queue), shown in Figure 4.1, demonstrates a first approach that illustrates the main idea of a throughput-oriented algorithm.

Examination of previous algorithms reveals that access to pointers result in inquire-then-update approaches, which will have very low throughput. As explained with the parallel count program, directly changing the data structures with the use of



This figure describes the data structure and the algorithm of the CB-Queue. Atomic in-memory operations on the `ReaderTicket` and `WriterTicket` variables allow claiming a position on the queue, by mapping the result of the increment to a position. This procedure happens both for reading and for writing. The `turn` variables associated with each data element allow synchronization when two threads are accessing the same queue element.

Figure 4.1: CB-Queue data structure and usage.

in-memory atomic increments result in the best throughput. As a solution, the pointers have been replaced by counters that represent each one of a set of preallocated elements, allowing mapping of integer values to elements in the queue.

The preallocated elements are associated with the numbers obtained from the counters in a round-robin fashion. To avoid conflicts when several processors try to access the same queue element, the variable *turn* has been introduced. The turn variable provides ordering between processors trying to access the queue, including ordering between enqueue and dequeue operations.

Unfortunately, the CB-Queue is limited in that operations in the CB-Queue *must* complete since it is difficult to undo an atomic increment. Once a processor executes an atomic increment to request a position in the queue for either enqueueing or dequeueing, there is no way to abort the operation. This poses problems because there is not a mechanism to know whether or not the queue is full or empty, and processors

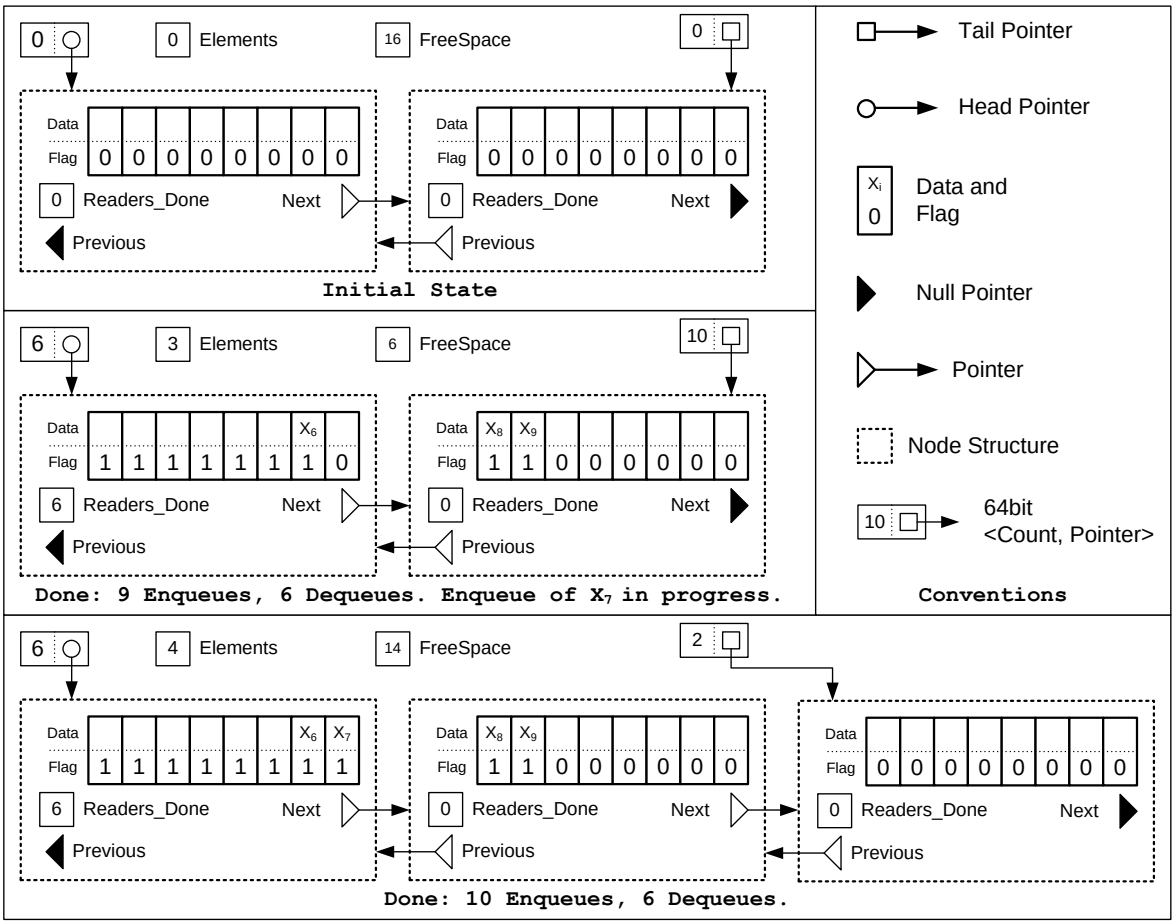
must attempt (and succeed) in their operations before the operation completes. This typically involves waiting until either enough space exists to complete an enqueue or until the queue becomes nonempty before completing a dequeue.

The HT-Queue (4.2) overcomes the limitations of the CB-Queue with the addition of new features (1) to support an unbounded number of elements, (2) to allow inquiring the status of the queue and (3) to avoid the presence of dangling pointers (*i.e.* pointers that are obsolete because the memory has been freed by another processor).

An unbounded number of elements is supported by the HT-Queue because the HT-Queue is constructed as a linked list of *nodes*. A node (Figure 4.2) is a data structure composed of (1) several queue items, each with a reader/writer synchronization flag, (2) pointers for the linked list and (3) an integer that counts how many reads to the node have been made. Among other things, the node structure amortizes the overhead of memory allocation because it holds several queue elements.

Inquiring about the status (*e.g.* empty) of the queue is supported by the HT-Queue algorithm with the addition of an element counter and a free space counter. Additionally, the *turn* variables associated with each queue element in the CB-Queue have been replaced with flags in the HT-Queue.

Dangling pointers are avoided in the HT-Queue because pointers are only dereferenced when it is guaranteed that the processor will successfully complete the operation for which the pointer is required. A simple idea is used to accomplish this: Obtaining a pointer and knowing whether or not the pointer is valid should be a single, atomic operation. For the HT-Queue, this is achieved by placing the (reader, writer) position counter and the (head, tail) pointer in the same 64 bit word. This serves a double purpose: It allows claiming a position in the array (with a 64 bit atomic increment) at the same time that the array pointer is read, and it allows the processor trying to claim the element to discover whether or not the queue is empty or full. Note that this technique also allows dereferencing the pointer *only* when it is guaranteed that there is available space for an enqueue or available queue elements for a dequeue. This is an important distinction that avoids the possibility of memory



The HT-Queue is a modified version of the CB-Queue that supports an unbounded number of elements. Pointers are used as in traditional queues, but the uncertainty of whether or not a pointer is stale has been removed by pairing the pointers with a counter that serves the double purpose of claiming a position in the queue and telling if the pointer is valid.

Figure 4.2: HT-Queue data structure and usage.

access exceptions, caused by a slow processor reading a pointer to a queue node that is about to be deallocated.

4.4 Examples

As it will be seen in the following sections, the throughput of an algorithm is intrinsically related to the features present in the architecture used to run it: Memory latencies, the ability to perform memory operations in memory, the presence of a shared bus or a crossbar, the number of memory banks and so on.

Cyclops-64 (C64) is used to explain the ideas about throughput and to show how to do a throughput-oriented design of an algorithm. C64 was chosen because it has a large number of thread units per chip, its architectural features are relatively easy to control and predict, there is no virtualization or preemption that introduces noise, the user can directly access the hardware, and it has features such as in-memory atomic operations.

4.4.1 Throughput of a Test and Set Lock

Consider a program δ composed of only two operations: (1) obtain a global lock using the test-and-set algorithm, and (2) release the global lock.

The intrinsic throughput of program δ is the number of processors that can *complete* program δ per unit of time. Note that the intrinsic throughput does not talk about the time taken by individual processors to complete the program. It talks about the number of completions per unit of time.

Possession of the lock is the bottleneck for the program. The number of programs that can complete per unit of time depends on the number of times that the lock can be obtained and released per unit of time.

Figure 4.3 shows that, from the point of view of the memory, acquiring the lock only takes *half a roundtrip*, because the lock is free (or owned by another processor) during the first half roundtrip of the test and set operation. Likewise, releasing the lock only costs half a roundtrip.

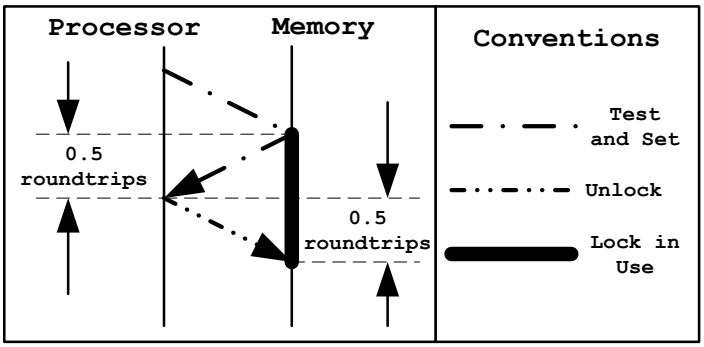


Figure 4.3: Time for ownership of lock.

Thus the throughput of program δ is $\mu_\delta = 1/m$ (m is defined as a memory roundtrip in Section 4.2).

4.4.2 Throughput of a Parallel Count Operation

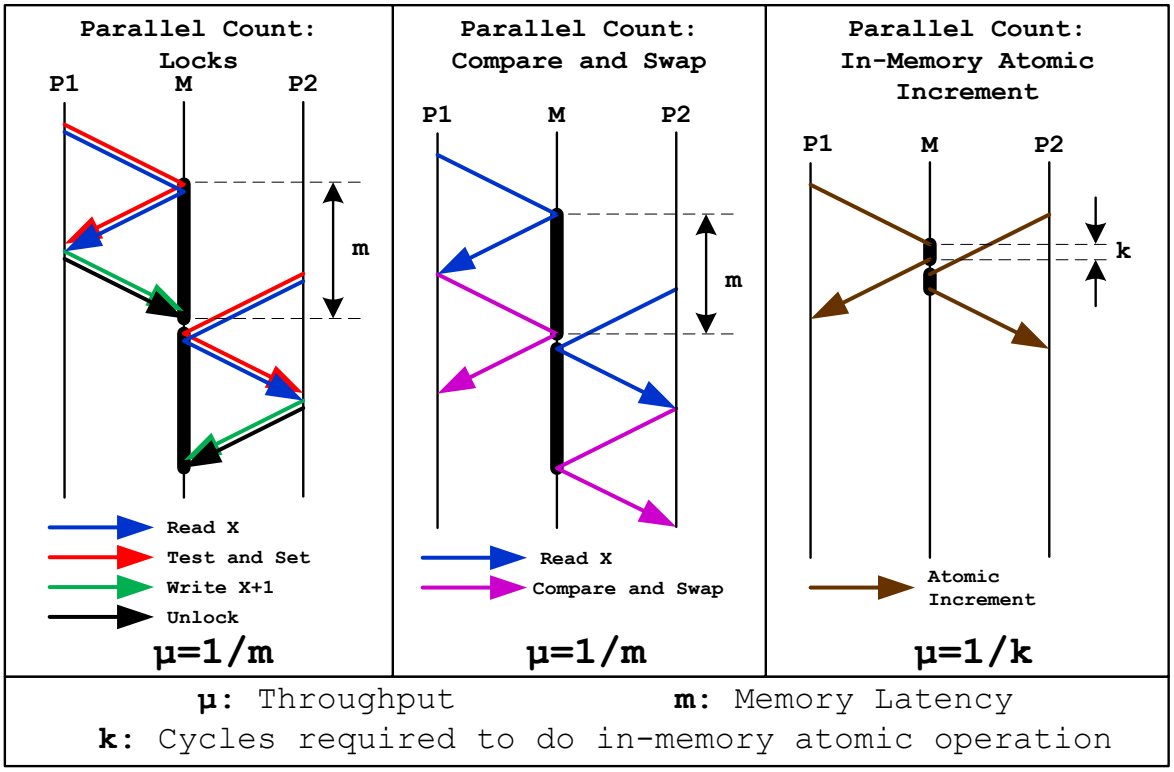
The throughput of a parallel count operation, or rather, the throughput of the *operation* of executing the count, can be analyzed by noting what the services are, what the agents are and what the limiting factors in the service itself are.

The throughput of the operation depends on many factors, including the particular implementation of the operation and the characteristics of the machine where it runs. For that reason, the Parallel Count Operation has a throughput that is dependent on its implementation.

4.4.2.1 Using Locks

In the lock-version of the parallel count operation, processors act as agents that try to access a shared resource. The shared resource that processors (agents) attempt to obtain is ownership of the lock.

As can be seen in Figure 4.4, the speed at which a lock can be acquired and released is limited. A processor must acquire the lock before releasing it. The amount of time that a lock remains in the possession of a particular processor is given by the ability of the processor to communicate with memory.



Timeline of events in the parallel count operation. The bottlenecks to fast counting are shown as thicker lines. Note that the use of locks results in the same performance as the use of Compare and Swap.

Figure 4.4: Throughputs of several implementations of a parallel count operation.

The lock is held for at least a memory roundtrip, even in the best case. Note that ownership of the lock starts at the moment that the compare and swap operation in memory becomes successful. However, the processor that issued the compare-and-swap operation does not yet have knowledge of its success, and it must wait for the response of the Compare-and-Swap, which takes half a memory roundtrip. In a similar fashion, releasing the lock is not an instantaneous process since the memory write that releases the lock must travel from the processor to the memory.

Note that the lock is held by a processor even before the processor is aware of it. Also, the lock may be owned by a processor even after the processor has issued a memory operation that will eventually release it.

The throughput of the parallel count operation is equal to the number of times

that the lock can be obtained and released by unit of time, which is once in the amount of time taken by a roundtrip to memory.

$$\mu = \frac{1}{m} \tag{4.2}$$

4.4.2.2 Using Compare-and-Swap

The throughput of the Compare-and-Swap implementation of the parallel count operation can be obtained by noting that the count must remain unchanged for at least a memory roundtrip. Otherwise, the Compare-and-Swap operation will not succeed.

For that reason, the throughput of the Compare-and-Swap implementation of the parallel count operation is:

$$\mu = \frac{1}{m} \tag{4.3}$$

4.4.2.3 Using In-Memory Atomic Increments

When in-memory atomic increments are used, the only factor limiting increment requests is the ability of the memory controller to execute atomic increments. Given that one in-memory atomic increment can be executed every k cycles, the throughput of the operation is:

$$\mu = \frac{1}{k} \tag{4.4}$$

4.4.3 Throughput of Common Queues

Queue algorithms are of particular importance to parallel algorithms. They are used in a variety of ways that span execution of irregular applications to runtime system support.

In particular, queues play an important role supporting scheduling and task management operations in runtime systems. For example, queues are the main scheduling mechanism of runtime systems such as Cilk [40], EARTH [104], Habanero-C [12], Habanero-Java [22] and TIDeFlow.

Because of the importance of queues when supporting highly concurrent systems, it is important that they possess the capability to operate efficiently in parallel programs. High intrinsic throughput is one of the requirements of queues that successfully support runtime systems.

The following subsections present an overview on current queues and their throughputs.

4.4.3.1 Single Lock Queue

The algorithm followed by processors in a single-lock queue implementation is (1) obtain a lock, (2) read the queue pointer, (3) update the queue structure, (4) release the lock. Figure 4.5 shows the data structure commonly used to implement this queue.

Completion of a queue operation takes at least 2 complete roundtrips to memory, even with optimal pipelining and scheduling (half a round trip to obtain the lock, 1 round trip to read the queue structure and half a round trip to update the queue and release the lock). Accordingly, the throughput of the single lock queue is:

$$\mu = \frac{1}{2m} \tag{4.5}$$

The analysis of the Single Lock queue and other subsequent analysis assumes that control flow instructions and other local instructions executed at the processor take very little time when compared to the memory latency.

Practical implementations of the Single Lock queue usually have a much lower throughput because optimal scheduling and pipelining are difficult due to library calls, or because thread preemption can not be disabled.

4.4.3.2 MS-Queue

The MS-Queue is a popular algorithm used in many commercial implementations, including the Java Concurrent Class. Its algorithm is described in detail in Michael and Scott's 1996 work [71].

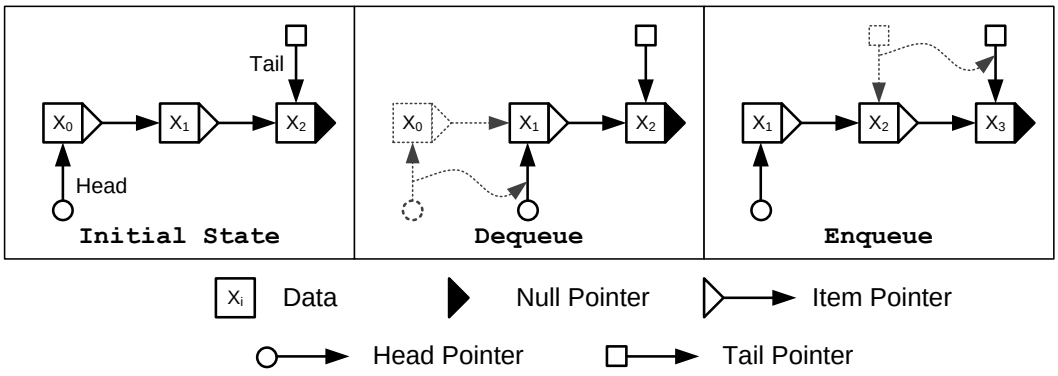


Figure 4.5: Traditional view of a queue.

The MS-Queue algorithm uses a data structure similar to that of Figure 4.5. Enqueues and Dequeues in the MS-Queue algorithm are based on successful execution of a Compare and Swap operation on the tail and the head pointers respectively. In general, a successful Compare and Swap operation on the MS-Queue requires that the memory location referred by the Compare and Swap remains constant for 2 memory roundtrips (half a memory roundtrip to read the initial pointer, one memory roundtrip to dereference the pointer, and half a memory roundtrip to complete the Compare and Swap operation).

The best throughput scenario (highest throughput) happens when the tail and head pointers are located in different memory banks, enjoying independent bandwidth and allowing simultaneous execution of Compare and Swap operations on the head and tail pointers. In that case, the total throughput is the throughput for enqueues plus the throughput for dequeues, and it is given by Eq. 4.6. 2 queue operations can be executed every 2 memory roundtrips.

$$\mu = \frac{2}{2m} = \frac{1}{m} \tag{4.6}$$

The throughput of this algorithm is better than the single-lock queue and it is not affected by thread preemption due to its non-blocking nature.

4.4.3.3 MC-Queue

The MC-Queue increases throughput by distributing queue requests over multiple traditional queues. The bottleneck is either a sequence of operations on a shared variable that keeps track of the element count or the aggregated throughput of all the traditional queues in the implementation. An enqueue-dequeue pair performs 2 atomic operations and one read on the shared variable limiting the throughput to 2 operations every $2k + z$ cycles. Enqueues and dequeues can complete in each individual queue after 3 roundtrips to memory limiting the throughput to 2 operations every $3m$ cycles. Eq. 4.7 presents the intrinsic throughput, the min is simplified under the assumption that the number of queues (G) is large enough.

$$\mu = \min\left(\frac{2}{2k+z}, \frac{2}{3m}G\right) = \frac{2}{2k+z} \tag{4.7}$$

4.4.3.4 Experiments

Several experiments were done to test the claims about the importance of throughput for parallelism and scalability in queues.

Experiments with microbenchmarks were conducted to test the effectiveness of the throughput models advanced. Then, experiments with the TIDeFlow system were conducted to test the importance of the throughput of queues in more involved situations. The microbenchmarks were written in assembly while the applications were written in C.

In all cases, the C64 processor architecture was used. The large number of processor units in C64 and the simplicity of its hardware make it useful to test the validity of the throughput models.

The highly accurate ET International's C64 simulator [35], instead of a real C64 processor, was used to gather all the data presented here because (1) despite the existence of real C64 chips, all of them are currently held by the U.S. Government and have not been released to the public and (2) some of the other queue techniques used as comparison, such as the MS-Queue [71] and the MC-Queue [70] require the use of Compare and Swap, which is not available on the C64 chip produced. The C64

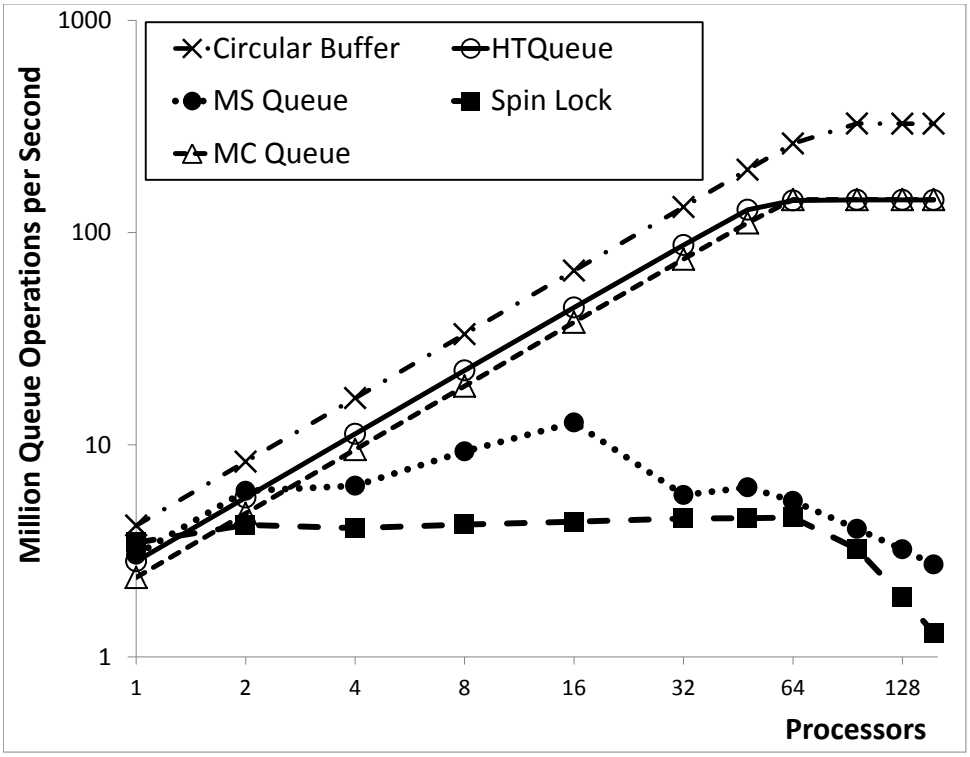
simulator was modified to include a CAS native instruction in its ISA. To make the comparison fair, the CAS instruction has the same implementation in the simulator as all other atomic operations: It has the same latency, it uses the same resources, it has its own opcode in the ISA, it is also executed in-memory and it generates the same contention at the memory controller. The compiler was modified accordingly to support the new opcode.

The throughput (Figure 4.6) and latency (Figure 4.7) of the queues analyzed in this thesis were measured using experiments where each processor performs a sequence of 75000 enqueue-dequeue pairs. During the interval measured, the processors do not execute anything other than the enqueue-dequeue pairs, and there is no waiting. Indirect effects such as system calls to `malloc` or `free` that may affect throughput were avoided. All memory is allocated before the experiment is run and deallocated after the experiment completes.

The measured throughput is defined as the total (aggregated) number of operations completed by all processors per unit of time. The latency reported is an average over all the individual queue latencies.

Table 4.1 shows that the theoretical predictions on throughput are confirmed by the experiments. The theoretical throughput for the MC-Queue, the CB-Queue and the HT-Queue matches very well the throughput measured. The reason is that the expression for the theoretical throughput is a function of C64's memory controller parameters (k and z) which are constant for C64. The theoretical throughput for the MS-Queue and the Single Lock queue does not match the throughput observed because the expressions for the theoretical throughput depend on the latency of individual memory operations, which is not a constant, and increases with the load of the system. This degradation of throughput can be seen in Figure 4.6: When the system is heavily loaded, the latency for individual memory operations increases, lowering the total system throughput.

The CB-Queue and the HT-Queue show significantly better intrinsic (maximum) throughput than the MS-Queue and the Single Lock implementation, and equal (in the



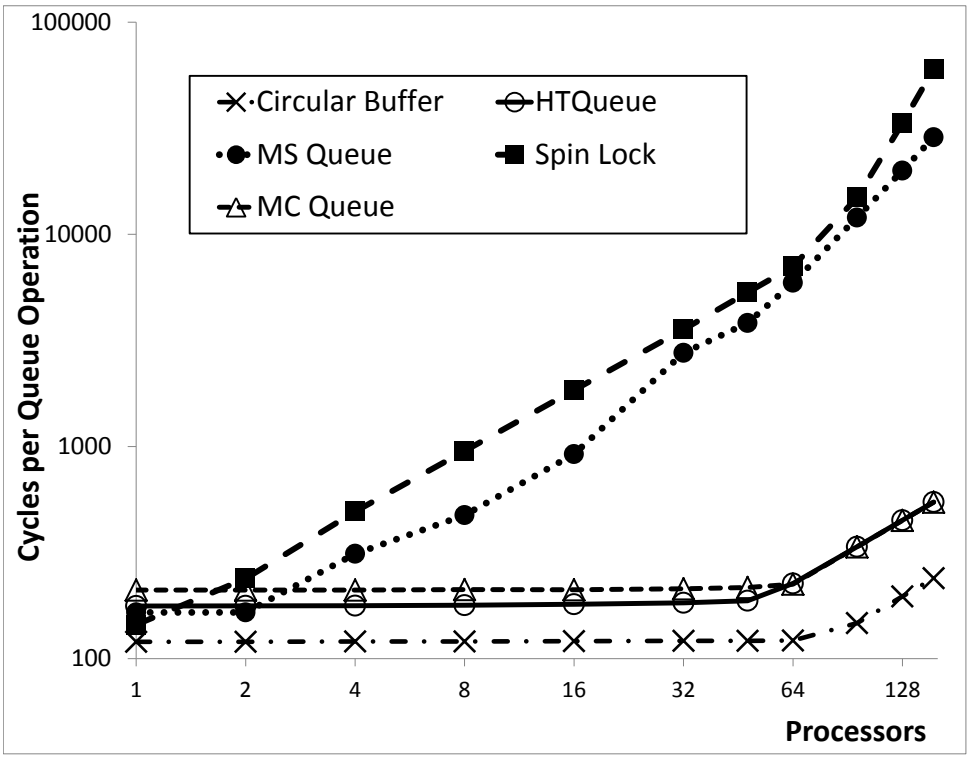
Note that the observed rate at which an operation is executed can not surpass its intrinsic throughput.

Figure 4.6: Observed throughput of several queue implementations.

Queue	μ **(Theoretical)**	μ **(Experimental)**
Single Lock	8.33	4.54
MS-Queue	16.7	12.7
MC-Queue	142.8	142.8
CB-Queue	333	326.0
HT-Queue	142.8	142.5

Units: Million Queue Operations per Second.

Table 4.1: Comparison of theoretical and experimental throughput for several queue implementations in C64.

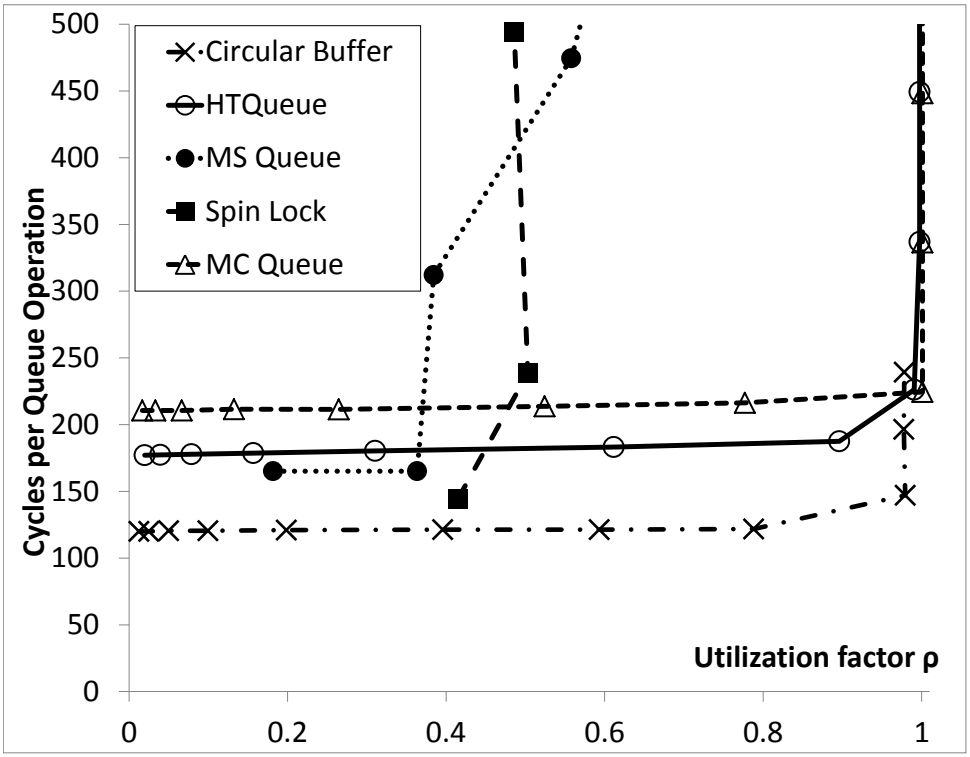


This figure shows the latency of a single queue operation. To obtain the latency, varying numbers of processors executed 75000 pairs of enqueue and dequeue operations. Note that the latency remains approximately constant until the intrinsic throughput of the queue is reached.

Figure 4.7: Latency of operations for several queue implementations in C64.

case of the HT-Queue) or greater (CB-Queue) throughput than the MC-Queue.

The behavior of latency with respect to the utilization factor ρ of the queue is shown in Figure 4.8. ρ has been calculated as the ratio of requests to theoretical throughput (Section 4.2) for each queue. The latency of the implementations designed following a high-throughput approach, when the queue is not saturated ($\rho < 1$), is better than the latency of all other implementations tested. When the queue becomes saturated ($\rho = 1$), the latencies of all queue implementations increase. Due to its high throughput, the CB-Queue can handle a large request rate before the latency increases due to saturation. The HT-Queue and the MC-Queue saturate similarly. Experimental



The throughput of an operation is the maximum rate at which the operation can be executed. The utilization factor ρ is an indication of how close a particular rate of use is to the limit. A good queue algorithm should maintain a low latency even when operating close to the maximum theoretical rate.

Figure 4.8: Latency vs. Utilization Factor for several queue implementations.

values for maximum request rates before the latency increases 5% are shown in Table 4.2.

The importance of queue throughput in larger applications was explored through experimentation with two applications supported by the TIDeFlow runtime.

In the experiments, each program is run with a modified version of the TIDeFlow runtime system. The different versions of the TIDeFlow system differ in the particular queue implementation used for its main scheduling queue. The five different queue algorithms used were: A simple queue that uses a lock, the MS-Queue (Section 4.4.3.2), the MC-Queue (Section 4.4.3.3), the CB-Queue (Figure 4.1) and the HT-Queue (Figure

Queue	**Max. Request Rate** (Million operations per second)
Single Lock	3.46
MS-Queue	6.06
MC-Queue	142.5
CB-Queue	262.68
HT-Queue	128.00

The maximum request rate before the latency increases by 5% is shown. The maximum request rate provides a practical limit for the concurrency of queues beyond which the performance starts degrading.

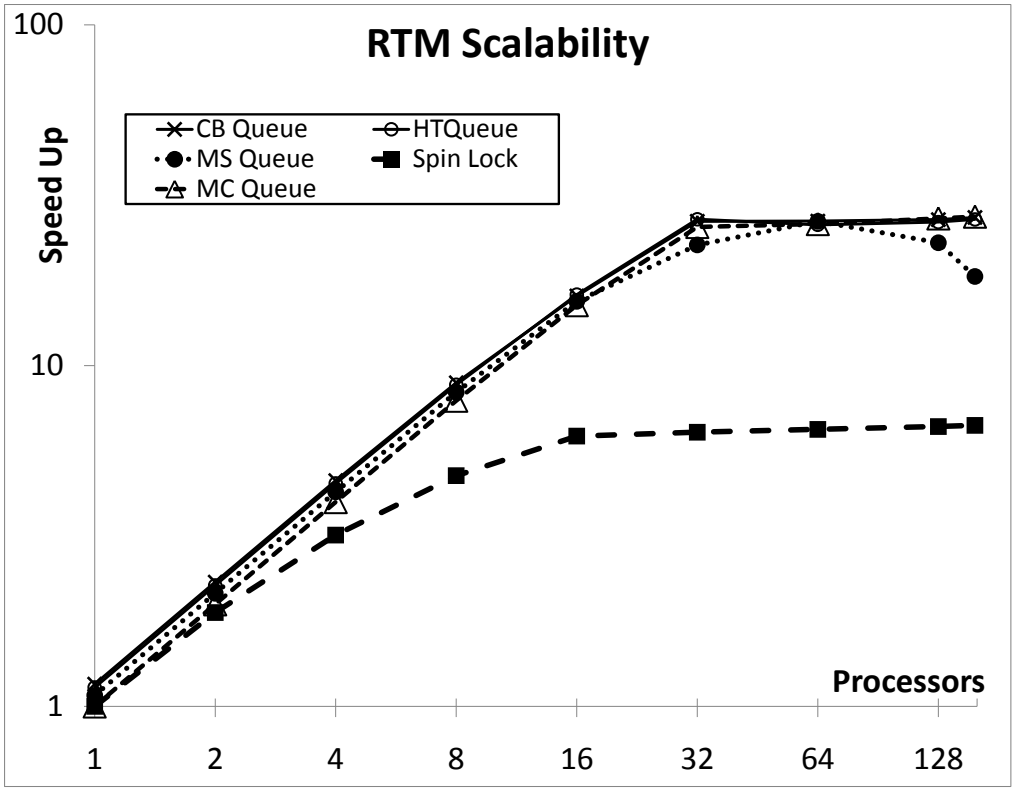
Table 4.2: Maximum rate of use of several queue implementations in C64.

4.2).

The applications used to test the impact of the throughput of each queue implementation were the following: A tiled version of a 3-Dimensional Reverse Time Migration (RTM) [14] used for oil exploration (8000 C code lines) with input size of $276 \times 276 \times 276$ and Blocked Matrix Multiply (MM) [49] (3000 C code lines) of size 5760×5760.

RTM consists of repeated point-wise multiplication and convolution of a set of 3-Dimensional input samples with a 3-Dimensional kernel. Good performance is achieved though multiple code transformations [77] that improve the locality during execution, resulting in moderate parallelism and abundant synchronization between tasks. The input samples reside in DRAM, and they are executed as tiles that fit in on-chip memory. Overlapping of communication (between DRAM and SRAM) and computation is done by having two tiles in on-chip memory. In the experiments conducted, four arrays of $276 \times 276 \times 276$ single precision numbers are used (approx 370MB in DRAM) with a kernel of size $13 \times 13 \times 13$. Tasks either compute a convolution between a data set of size $6 \times 1 \times 1$ and the kernel of size $13 \times 13 \times 13$, or they do memory movement between DRAM and SRAM.

In the case of Matrix Multiplication, three Double Precision matrices are in off-chip DRAM memory (approx. 800MB). The nature of the software-managed memory

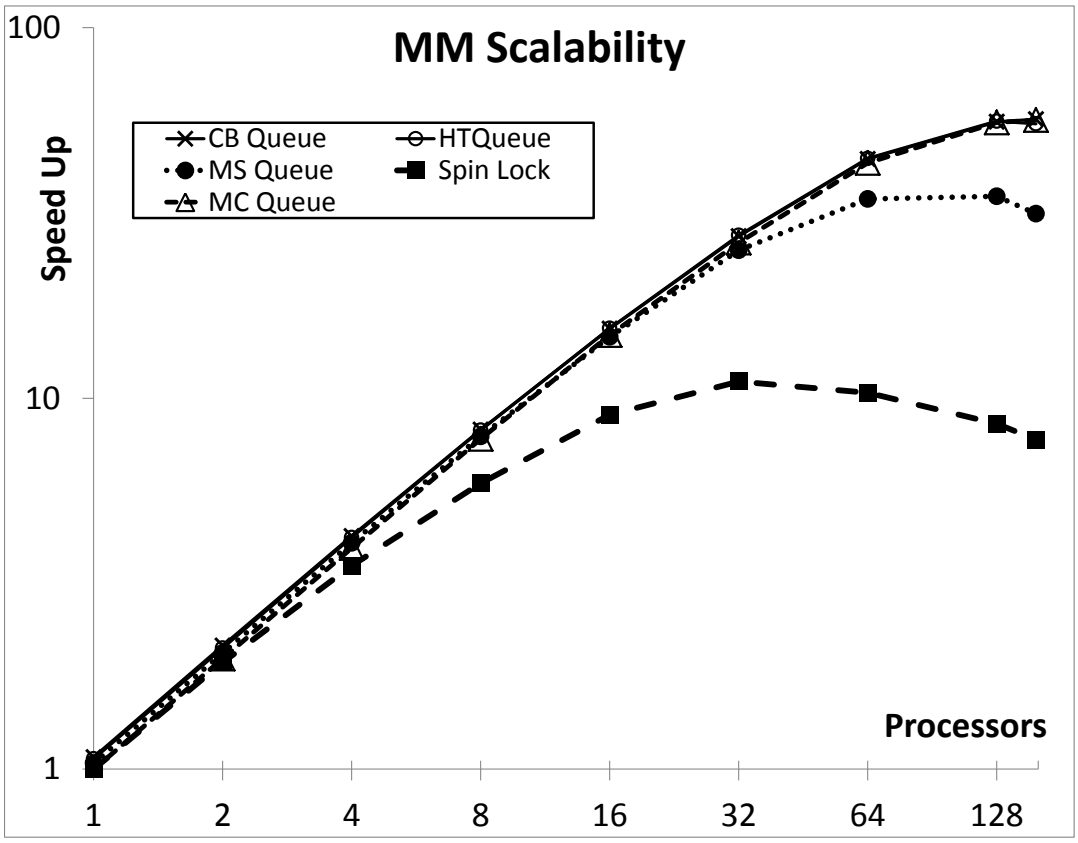


This figure shows the scalability of a Reverse Time Migration program. Although the RTM program used in each experiment is the same, the queue used to implement the scheduler was changed to show the effect of throughput on the performance of a parallel program. As observed in the figure, a high throughput scheduler is required to obtain high performance.

Figure 4.9: Influence of scheduler throughput on programs: RTM.

hierarchy of C64 requires a double buffering strategy where some threads move blocks of 192×192 between off-chip DRAM memory and on-chip SRAM memory while the other threads make computations of 6×6 tiles allocated in registers from the blocks in SRAM. The optimum ratio between data movement threads and computation threads, the optimum sizes of blocks and register tiles and other optimizations applied to this benchmark have been detailed in previous publications for C64 [43].

The impact of the queue choice in the overall application is shown in Figures 4.9 and 4.10. As seen in the figures, the choice of queue implementation does not play



In each experiment, the queue used to implement the scheduler is changed while the matrix multiplication program remained the same. As in Figure 4.9, a high throughput scheduler is required to obtain high performance.

Figure 4.10: Influence of scheduler throughput on programs: Matrix multiplication.

a critical role when few processors compete for access to the queue. When the number of processors is increased, however, the throughput available at the queue becomes a dominant factor in performance.

Reverse Time Migration does not posses enough parallelism, so the scalability of the program reaches a fixed point after few processors. However, the bottleneck of the program becomes intrinsic throughput, rather than available parallelism, when a queue implementation with low throughput is used (such as is the case of the Spinlock and the MS-Queue implementations).

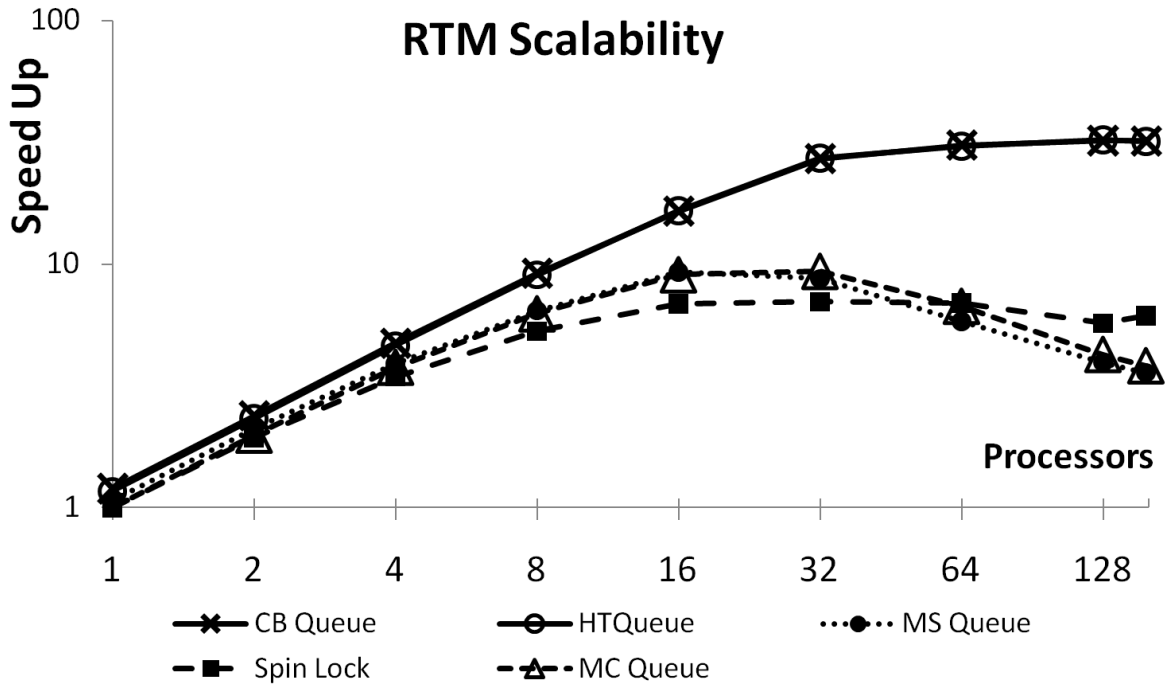
Parallelism is abundant in Matrix Multiply. However, despite the abundance of parallelism, a high level of scalability with respect to a serial implementation can only be achieved with high throughput. The results confirm this: whenever a queue algorithm with high throughput was used (HT-Queue, CB-Queue, MC-Queue), the program scaled better. When a slow-throughput implementation was used, the program did not scale well: the particular case of the spin-lock implementation actually decreased in performance and scalability when the system became congested.

Results of MM show that the proposed HT-Queue and the MC-Queue reach a similar maximum speed up: 55.9 for HT-Queue and 56.3 for MC-Queue given their similar theoretical throughput. CB-Queue performance is always slightly better than HT-Queue and MC-Queue, and its maximum speed up is 56.6. RTM shows similar results where the maximum speed up obtained is related to the throughput of the queue used.

The MC-Queue and the HT-Queue have a high throughput in themselves, but they require calls to malloc and free, which could become the bottleneck. The experiments of Figures 4.9 and 4.10 avoid this through the use of a high performance, distributed memory allocator based in the CB-Queue. A custom memory allocator was used to isolate possible throughput limitations imposed by the system.

Figures 4.11 and 4.12 further explore the influence of system calls in throughput and scalability. Instead of a custom, high-performance memory allocator, the standard memory allocator available for C64 was used.

The results of Figure 4.11 shows that the throughput of an operation is susceptible to side effects such as system calls or other sources of contention. The MC-Queue can be taken as an example to illustrate this point. The MC-Queue was shown to have an excellent throughput. However, the throughput obtained for the MC-Queue algorithm takes into account the algorithm itself and the architecture, and it assumes that an efficient memory allocator can be developed. However, as seen in Figure 4.11, a naive memory allocator will become the bottleneck and will ultimately limit the throughput of the operation.



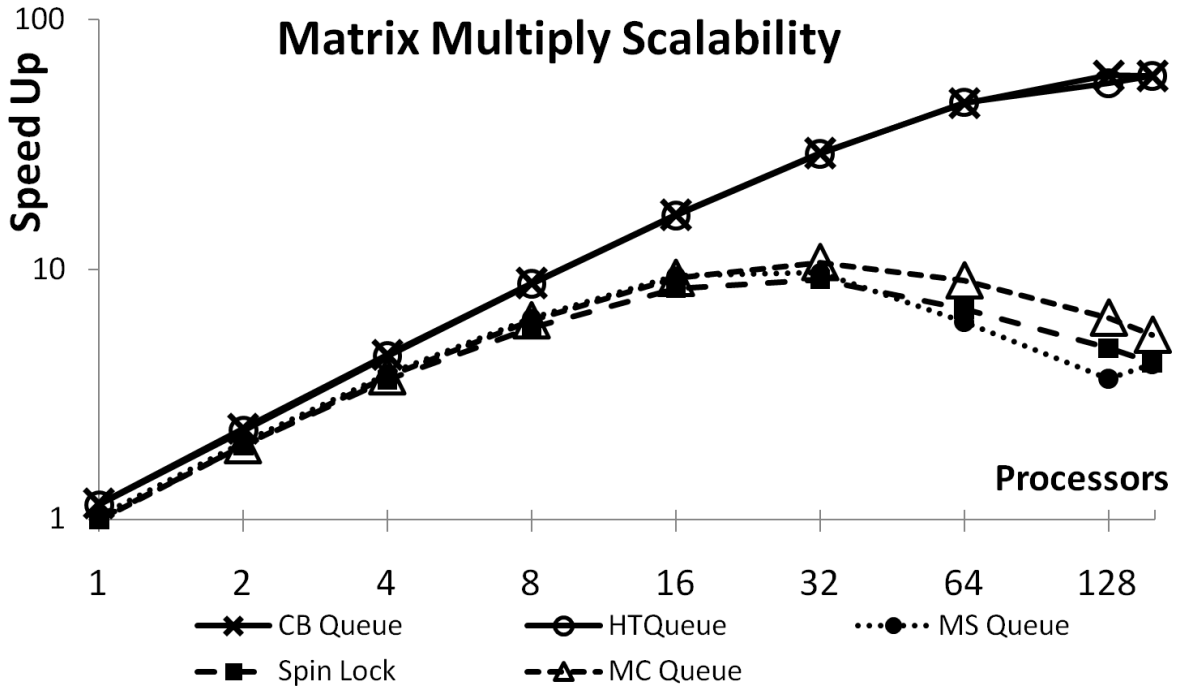
This Figure repeats the experiments of Figure 4.9, but the memory allocator used is a naive one.
The naive memory scheduler limited the throughput of the scheduler, decreasing the scalability of the overall program.

Figure 4.11: Influence of scheduler throughput on systems with a naive memory allocator: RTM.

The memory allocator of C64 uses a centralized, global lock, to ensure correctness between threads. As a result, its performance matches the performance of the queue implementation that uses a spin lock.

Experiments with Matrix Multiply and a naive memory allocator (Figure 4.12) show a similar trend. Even though enough parallelism is available, processors are not able to take advantage of it because the use of a naive memory allocator limits the throughput of the queue implementation.

Interestingly enough, the throughput (and the performance) of the CB-Queue and the HT-Queue is not affected significantly by the use of a naive memory allocator. This is due to the fact that the CB-Queue does not use memory allocation at all, and the HT-Queue allocates several queue items at the same time, reducing the number of



As in Figure 4.10, the experiments change the scheduler while the program remains the same. In this case, however, a naive memory scheduler was used. The poor throughput of the naive memory scheduler limited the scalability of the program.

Figure 4.12: Influence of scheduler throughput on systems with a naive memory allocator: Matrix multiplication.

memory allocation operations required.

The results of Figures 4.9 and 4.10 show that throughput plays an important role in parallel applications. In particular, high throughput is paramount to supporting high scalability.

4.4.4 Simplifying the Representation of Tasks to Increase Queue Throughput

The throughput of a particular operation can be increased when the algorithm used for the operation is changed. Both the data structure and the technique used to execute the operation can have a significant impact in the throughput of the algorithm. For example, an implementation of an operation that uses a combination of slow operations on a centralized data structure will have a lower throughput than another

implementation that uses fewer, faster operations on a distributed data structure.

The global queue used for the scheduling and representation of tasks in the TIDeFlow system constitutes an excellent example of a situation where the data structure used and the selection of operations used had a significant impact on the throughput of the scheduling system. The throughput of the queue is critical to support execution because all tasks that are are executed or that become enabled have to be enqueued and dequeued from the queue. Unfortunately, the process of enqueueing tasks to make them available to be executed takes a nontrivial amount of time and may result in unacceptable overhead in fine-grained programs.

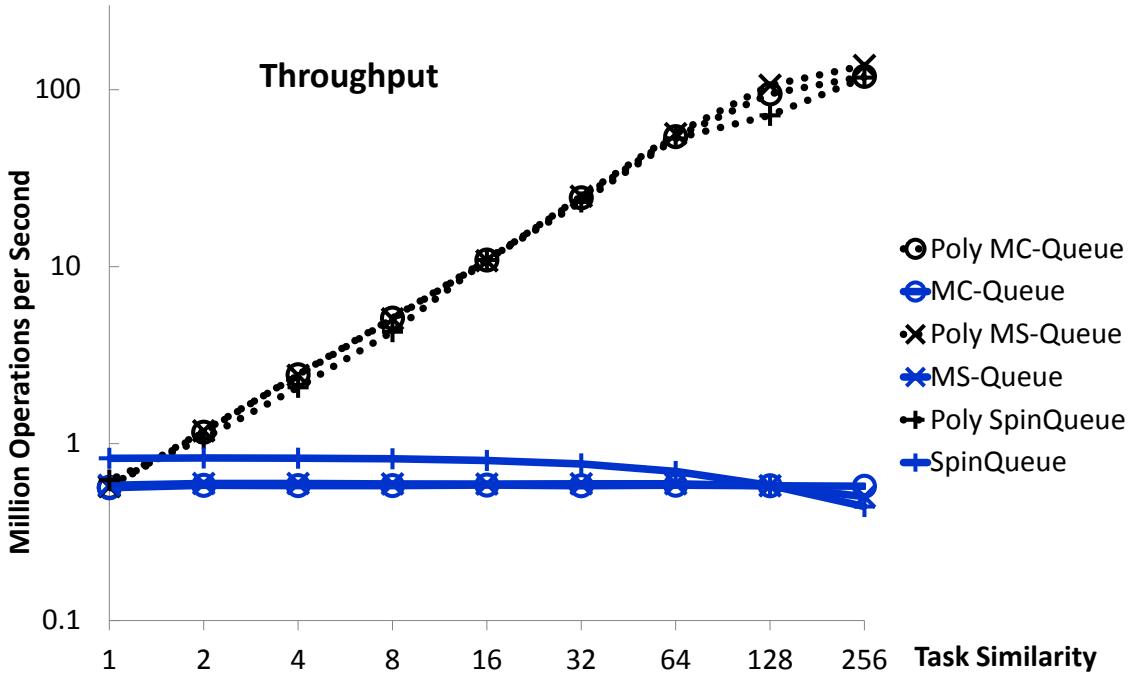
The analysis of the throughput of queues of Section 4.4.3 addresses the general case of arbitrary data items being placed in the queue. However, in the case of queues supporting runtime systems for HPC execution, it has been shown that there is significant similarity in the data placed in the queue [86]. This similarity can be exploited to produce modified versions of the algorithm where several queue items are compressed into a single item.

The name *polytask* has given to the compressed representation of several queue items representing tasks. The advantage of the use of polytasks is that in its compressed representation, only one queue operation is equivalent to several uncompressed queue operations. Figure 4.13 shows that the effective throughput runtime system queue is increased when a compressed representation for tasks is used.

In summary, the throughput of an operation can be affected when the operation itself is made more efficient. In the case presented in Figure 4.13, the throughput of accessing a queue was made more efficient through a better representation of the tasks generated. This same approach can be used to other concurrent operations.

4.5 Summary

This chapter presented throughput as a new property of concurrent operations. The throughput of a concurrent operation is the maximum rate at which a concurrent operation can be completed.



The choice of algorithm and data structure affect the throughput of an operation. The figure shows an experiment where the throughput of three queue algorithms is measured. Task similarity refers to the ability of data to be compressed. The "Poly" versions of the algorithms use data compression. The main conclusion from the data in the figure is that variations in the data structure used (such as compression) can lead to changes in the throughput of operations.

Figure 4.13: Throughput of queues with and without task compression.

The concept of throughput is of paramount importance for highly parallel processor architectures because it bears a direct relationship to the scalability of programs. Algorithms that are based on operations with low throughput are unable to scale because scalability requires that many operations can complete per unit of time.

An important characteristic of the throughput analysis is that *the intrinsic throughput of an operation is a constant* value that depends on the algorithm implementation and the architecture where it runs. It does not depend on how the algorithm is used, and it does not depend on other runtime factors such as whether or not other operations are being executed.

The intrinsic throughput of an operation does not guarantee that the operation will be able to complete at that rate. Rather, it establishes an upper bound on the rate of completion of the operation. Runtime factors such as the rate at which the operation is being attempted, or the existence of other conflicting operations may decrease the observed rate of completion of an operation.

Finally, the ideas on throughput presented in this chapter have used Cyclops-64 to motivate the discussion, develop the model, and provide the experiments. The ideas presented here will also apply to other architectures where the cost models may be different and the numeric value of the throughputs found may change. Nevertheless the principles of throughput will still be fully applicable.

Chapter 5

TIDEFLOW IMPLEMENTATION

The implementation of TIDeFlow was the next challenge that was faced after the development of the theoretical foundation for the execution model (Chapter 3) and the throughput analysis required to support highly concurrent execution (Chapter 4).

Several goals were at hand: The implementation should be simple, it should represent the model with fidelity, it should be easy to use and it should have a very high performance.

The first task that was addressed concerned the programming model. Although many alternatives were explored, including graphical environments such as the DOT programming language [39], it quickly became apparent that a programming model based on the C programming language was simple enough, it could be implemented in a few months and it would allow enough flexibility to represent a large number of programs.

The next problem was the development of the runtime system. A distributed runtime system was sought from the start. The desire to use a distributed runtime system arose from the large number of processing elements typically found in manycore architectures, where the use of a centralized system for synchronization was likely to incur in high overheads. Instead, the runtime system took the approach of separating the synchronization and the scheduling issues and developing algorithms for each one of them.

Synchronization and activation of tasks is done localy in TIDeFlow. This design decision resulted in distributed control during the execution and ensures that no one slow processor will stall all the computation on the system. The synchronization operations are supported by in-memory atomic operations that are performed

on synchronization variables that are local to the processors participating in the synchronization operation. The result is that each processor is able to enforce its own synchronization operations, and it is able to locally enable tasks that have met all of their input dependencies.

Scheduling, which happens concurrently, albeit separately, from synchronization, uses a centralized queue that can be accessed by all processors at the same time. The scheduling queue has been the subject of previous publications [86], and it serves as the main work pool where processors publicize and obtain work.

Compiling was another significant problem. A full compiler that would produce executable binaries would require a launcher, an assembler and all their associated tools. Instead, the TIDeFlow compiler takes the limited approach of placing the program in memory, from where it can be easily executed.

Finally, optimization of programs required the generation of program traces. The runtime system was developed to natively support profiling. To do so, the runtime system gathers information about the starting and ending times for each task along with information about which processor executed it.

The following sections describe the details of each part of the tools that form the TIDeFlow system.

5.1 TIDeFlow C Interface

TIDeFlow programs are described by a combination of graphs such as those of Figure 3.22 and codelet functions such as those of Figure 3.21.

A graphical interface can be used to represent TIDeFlow graphs. However, it is useful to have a pure C interface that allows construction of TIDeFlow graphs. When using the C interface, the user is responsible for initializing the TIDeFlow runtime system and for creation of TIDeFlow programs.

This section describes in detail the interface used to express and use TIDeFlow programs.

5.1.1 Initializing the TIDeFlow Runtime System

First the runtime system must be initialized, and a number of processors must be allocated to execute TIDeFlow programs.

Figure 5.1 shows the interface to initialize the TIDeFlow runtime.

```
void InitRuntime( int NumProcessors );
```

Figure 5.1: C interface to initialize the TIDeFlow runtime.

At initialization, the following tasks are completed:

- the specified number of processors are allocated,
- a CB-Queue, modified to support and manage TIDeFlow actors [86, 87] is allocated and
- all processors start polling the global queue to find work to execute.

5.1.2 Creation of TIDeFlow Programs

The steps to creating a TIDeFlow program are: (1) create a memory context for the program (2) add actors or other programs to the program graph, (3) add dependencies between actors and (4) provide static parameters to actors.

The following sections explain how to accomplish each one of this tasks in detail.

5.1.2.1 Creation of a Program Context

All TIDeFlow programs require a context that must be created. A TIDeFlow program can be identified by a pointer to its context. This pointer can also be used to include a program as part of a larger program.

Figure 5.2 shows the C interface to create a program context.

```
CodeletSet * = CreateCodeletSet( char * ProgramName );
```

Figure 5.2: C interface to create a TIDeFlow program.

```
int AddCodelet(
   CodeletSet *ProgramContext,
   void (*function)( void *, int ),
   int LoopIterations,
   char * ActorName
   );

void SetPriority(
   CodeletSet *ProgramContext,
   int ActorID,
   int Priority /* 0: High, 1: Low */
   );
```

Figure 5.3: C interface to add an actor to a TIDeFlow program.

5.1.2.2 Addition of Actors or Programs to a Context

Once a program context has been created, actors must be added to it using the interface shown in Figure 5.3. The interface allows specification of the function to be used and the number of loop iterations in the actor.

The interface returns an integer, that, along with the pointer to the context where the actor belongs, serves as an identifier for the actor.

Priorities for the execution of actors can also be specified through the interface. TIDeFlow supports two levels of priority (high and low) to aid the programmer in the scheduling and synchronization of tasks. During runtime, tasks with a high priority will always be scheduled first over tasks of low priority. The priority system if particularly useful to specify that actors in the critical path of a program should be executed first.

In addition to using codelets as actors, entire TIDeFlow programs can be used as an actor. The C interface to use a TIDeFlow program as part of another is shown in Figure 5.4

5.1.2.3 Addition of Dependencies Between Actors

Once all actors in a TIDeFlow program have been added, dependencies between them can be specified.

```
int = AddCodeletSet(
   CodeletSet *ProgramContext,
   CodeletSet *Program_To_Use,
   char * Name
   );
```

Figure 5.4: C interface to use TIDeFlow programs as part of larger programs.

```
void SetDependency(
   CodeletSet *ProgramContext,
   int SourceActor,
   int DestinationActor,
   int TimeOffset,
   char *DependencyName
   );
```

Figure 5.5: C interface to specify dependencies between actors.

Actors are identified by the integer returned when they were created. The interface to specify dependencies is given in Figure 5.5.

5.1.2.4 Providing Static Parameters to Actors

In TIDeFlow, communication between actors is done through shared memory and not through tokens. To accomplish this, actors are provided with pointers to locations in memory where they can consume and produce data.

A total of eight constant, 64-bit values, can be given to each actor using `SetStaticData` during the construction of the program. At runtime, actors can obtain any of these values through a call to `GetStaticData`. Their interfaces are provided in Figure 5.6.

5.1.3 Running TIDeFlow Programs

When a program has been properly set up and the runtime has been initialized, it can be given to the runtime for execution.

Figure 5.7 provides the interface to execute a program.

```
void SetStaticData(
   CodeletSet *ProgramContext,
   int ActorID,
   uint64_t Data[8]
   );

uint64_t GetStaticData(
   void * parameters, /* Provided by the runtime */
   int DataIndex /* An integer from 0 to 7 */
   );
```

Figure 5.6: C interface to set and get static data particular to an actor.

```
void SignalSet(
   CodeletSet *ProgramContext
   );
```

Figure 5.7: C interface to execute a program.

5.2 Intermediate Representation

Several desirable features were identified during the design cycle of TIDeFlow, including the ability to support future compiler optimizations, or the possibility to change the program at various stages of compilation. As a result, an intermediate representation was designed to represent the program during the early stages of compilation.

The intermediate representation of TIDeFlow programs uses a small data structure to represent each actor. A program is represented by an array-of-structures that contains the actors.

The design of the intermediate representation aims to express the graph as a collection of integers. For example, whenever a reference to an actor is used, its offset in the program's array-of-structures is used rather than a pointer. This same representation is used to describe arcs: Two integers are used to represent an arc, representing the starting and ending actor of the arc.

Representing TIDeFlow programs as a collection of integers has the great advantage of allowing simple duplication of a program (for composability of programs)

and portability to other architectures.

The intermediate representation is computed by the TIDeFlow toolchain.

5.3 Compilation and Execution of a TIDeFlow Program

The final stage of compilation of a TIDeFlow program takes into account the architecture in which it runs and leaves the program resident in memory. At this point, saving a compiled program to non-volatile storage is not supported. For that reason, the compiler and the launcher of a TIDeFlow program are integrated into the same tool.

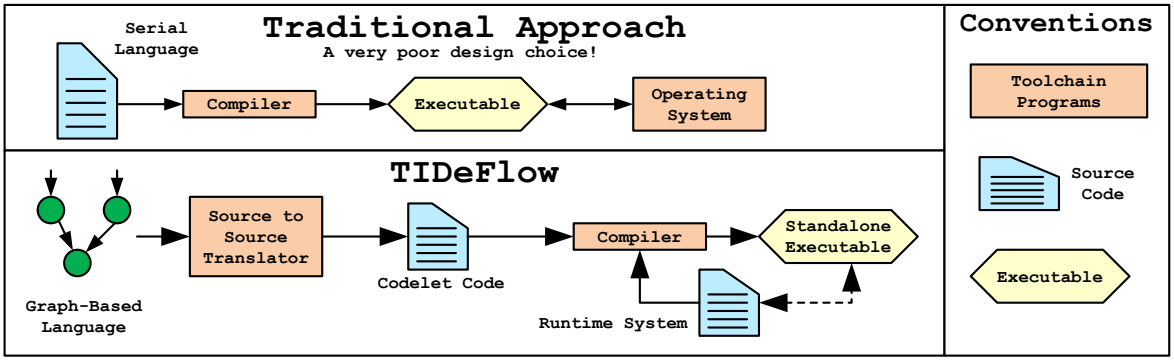
Compilation of a TIDeFlow program consists of translating the intermediate representation into an executable data structure where offsets in the Intermediate Representation structures result in pointers in the final program. Memory is allocated and initialized for the actors at the end of the compilation stage. The resulting executable data structure contains actors with their properties and their states, linked through pointers that follow the dependencies specified in the original program.

A TIDeFlow program is executed when a pointer to the program is passed to the runtime system. The runtime system scans the program and schedules all actors that have no dependencies for time instance zero. The execution continues until the runtime system detects that no more actors will be scheduled.

5.4 TIDeFlow Runtime System

The TIDeFlow runtime system supports the execution of programs by providing scheduling, synchronization, initialization and termination of programs.

The role of TIDeFlow's runtime system and its relationship to the toolchain is shown in Figure 5.8. TIDeFlow's runtime system has been designed to support execution in an environment without virtualization. The runtime system is directly embedded in the application binary and it is able to perform all task management operations.



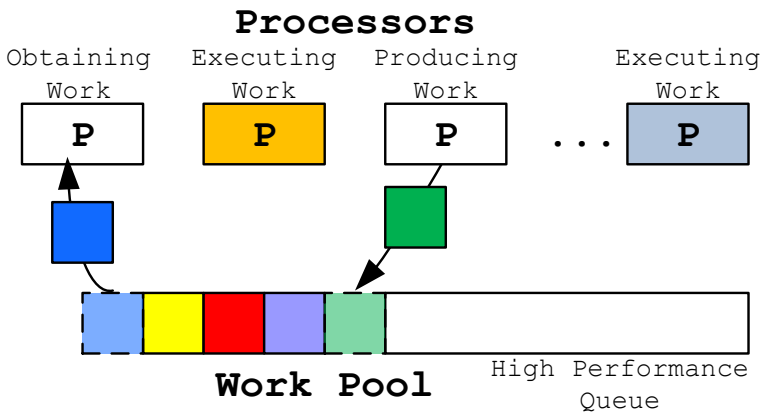
Comparison of TIDeFlow's approach to compilation and execution in comparison to a traditional approach. Note that TIDeFlow is able to run without the support of a traditional operating system. Instead, TIDeFlow's runtime system, embedded in the application binary, performs all task management operations.

Figure 5.8: TIDeFlow toolchain.

The implementation of TIDeFlow's runtime system presented several challenges that ultimately resulted in interesting advances and tools: A fully distributed runtime system, a programming language to describe program graphs, concurrent algorithms [86] and new ways to reason about performance models [85].

The basic unit of execution for scheduling and execution in the runtime system is the *task*. As explained, each one of the parallel iterations on an actor are represented by a single task in the runtime system. To allow immediate visibility of available work, all tasks that become enabled are written to a queue that can be accessed concurrently by all processors.

Perhaps the most important feature of the runtime system is that its control is *fully distributed*. There is no one process or thread or task in charge of the runtime system duties. Instead *each* processor concurrently (1) performs its own scheduling and (2) handles localized signals related to the actor being executed by the processor, including enabling other actors and writing them to the global task queue. The TIDeFlow runtime system is fully distributed with regard to the processors, because no one processor, thread, or task is responsible for scheduling, but it is still centralized from the point of view of the memory because the runtime system uses a single, global



TIDeFlow's Runtime System uses a high performance queue as the main mechanism for task management.

Figure 5.9: TIDeFlow scheduling queue.

queue.

Development of a decentralized runtime system required advances in concurrent algorithms and in the internal representation of actors and tasks. These advances were achieved by work in concurrent algorithms for runtime systems [84] and in task representation and management [86, 87]. The resulting high performance queue (Figures 5.9 and 4.1) was able to adequately support task management with very low overhead.

In the first study [85], it was found that the use of a global queue in a decentralized system is possible if the queue is designed to sustain a high throughput of operations in a concurrent environment. The study has resulted in a very efficient queue algorithm that can be used by processors to concurrently schedule their own work.

In the second study [86, 87], it was found that there is a high similarity in tasks from the same actor: They have the same properties and they execute the same function. Such similarity can be exploited to decrease the number of operations required to write those tasks in the centralized queue, greatly reducing the overhead of the runtime system.

The TIDeFlow runtime system, with the improvements developed in queue algorithms [80, 86, 85], has resulted in a very-high-performance decentralized system, with overheads that are much lower than the average duration of tasks, even for fine grained programs. The use of a centralized queue (Figure 5.9) for task management allows decentralized control, it renders load balancing irrelevant and it simplifies the design of the runtime system at large.

The examples shown in Chapter 7 show that the TIDeFlow runtime system is an excellent choice to support the execution of parallel programs on many-core architectures.

5.5 Parallel Program Traces

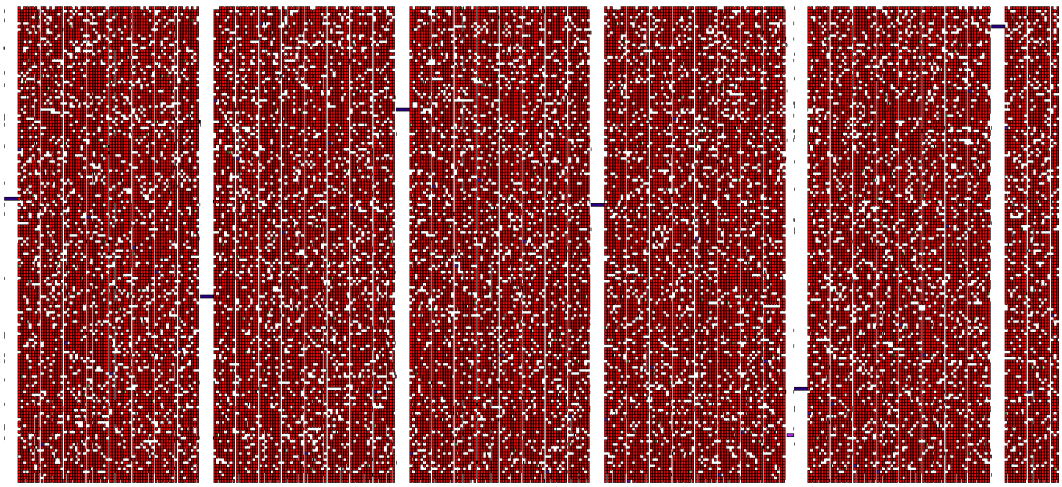
Program traces will be introduced before presenting the TIDeFlow programming environment. Parallel program traces are useful to understand programs. Several examples through the remainder of the chapter are complemented with program traces to improve their clarity.

Program traces are a powerful way to provide insight into the behavior of a parallel program. Through profiling and traces, a programmer can take decisions about parallelism, priorities, resource allocation and other things.

A program trace describes the activity that each processor executed at any point of time. The information reported for each processor includes the task that the processor was executing, and in some cases, the dependence relation between tasks.

TIDeFlow provides native support to create program traces. The runtime system provides the option of logging all events and producing a report file afterwards. The events observed by each processor are assembled together to produce a program trace of the TIDeFlow program executed. The logs of events at each processor are placed into a global queue that uses the CB-Queue algorithm, where they are read at the end of the execution and dumped to a file.

Figure 5.10 shows an example of a program trace. The trace, and its associated profile, were obtained from the execution of an early version of matrix multiply.



An example of a profile of the execution of a matrix multiply program. The horizontal dimension represents time, the vertical dimension represents processors and the color represents tasks. In this profile, it is easy to observe where processors are left idle.

Figure 5.10: Execution trace of matrix multiplication.

Profiles such as the ones in Figure 5.10 are useful because they show where the time is spent in the program. It shows the sources of overhead and it provides valuable insight for optimization.

Profiling is not enabled by default in TIDeFlow. To enable profiling, the TIDeFlow Runtime must be recompiled. A macro (`USEPROFILER`) allows control of whether or not profiling is enabled. The macro is usually passed specified from the command line when compiling a TIDeFlow program.

When a program finishes, the runtime system creates a file, called `profiler.dump`, in the local directory where the program was run. This file contains the profiling information.

A separate visualization tool is also provided with TIDeFlow to allow easy interpretation of the profiler plots. The visualization tool can read profiler files and provides an interactive environment where the programmer can zoom into parts of the program to better analyze its behavior.

The remaining sections use the concept of a program trace to illustrate the programming constructs of TIDeFlow.

Chapter 6

TIDEFLOW PROGRAMMING PRACTICES

The ability to effectively express a parallel program remains an elusive art. Several decades where the predominant paradigm was serial programming has greatly influenced the field of computer engineering.

Many features in programming languages and parallel architecture are the legacy of serial paradigms. For example, automatic data caches arose as a way to automatically improve the performance of serial programs. However, automatic data caches do not necessarily improve the performance of parallel programs, and in some cases, it decreases its performance when false sharing is present [110]. The use of pointers and global variables are a powerful way to use serial programs, but they result in race conditions when applied to parallel programs. Compiler optimizations that reorder instructions may provide excellent performance advantages for serial programs, but they may cause incorrect results in parallel programs.

The field of computer engineering faces a legacy of serial execution in an environment where parallel execution is required. This new environment needs a viable approach to parallel programming if it is to be used. Although some approaches to parallel programming have been undertaken (described in Chapter 2), no single approach has so far been able to address all the issues currently faced by parallel programming.

An excellent example of the challenges in parallel programming is the example of Garcia's implementation of Matrix Multiply [46, 44, 50, 47].

The matrix multiplication program developed by Garcia had as one of its primary objectives the demonstration of the capabilities of manycore architectures to

execute HPC programs. Garcia's efforts were directed toward achieving an implementation of a matrix multiplication library that would reach a performance close to the peak performance of the machine.

Garcia's task with matrix multiplication, although seemingly simple, was in reality a very difficult task. More than a year passed from the moment Garcia undertook his enterprise to the moment the results were published. The task of implementing matrix multiply was difficult because it required development of specialized kernels for the computation and the communication as well as new techniques for synchronization and scheduling of tasks.

Although the unavoidable task of developing specialized, assembly-written kernels for the computation tasks did require a significant amount of time, the synchronization and scheduling of tasks also consumed a significant amount of time and effort.

Garcia approached the problem of synchronization and scheduling first through the use of traditional parallel programming constructs. In particular, he attempted to use the TNT [31] library, a low-overhead, thread-management library very similar to pThreads. Despite the low overhead and the easy-to-use interface of TNT, it ultimately proved inadequate for the task. The inadequacy of TNT was partially due to the fact that it was designed to handle traditional fork-join approaches in coarse-grain programs. It quickly became apparent that a dataflow-like approach was needed to handle the available parallelism.

Garcia had to overcome several issues: The inadequacy of TNT to represent producer-consumer scenarios, the high overhead of TNT and the inability of TNT to manage resources. At the end, Garcia opted for having custom-designed mechanisms for synchronization, scheduling and resource management.

Synchronization was the first issue tackled by Garcia. Instead of using the TNT library primitives for synchronization, Garcia used –with success– an active-wait approach where atomic operations and a global variable was used for synchronization. The results allowed speedy synchronization with low overhead.

Scheduling and resource management were a significantly harder issue to tackle.

Right from the beginning of the optimization process, it became apparent that overlapping of communication and computation was necessary to achieve good performance. Furthermore, the different communication operations had to be orchestrated carefully because there was not available memory bandwidth to have more than 8 copying processors working at the same time.

Garcia's solution to the scheduling and resource management problem was to use counters to represent the tasks and to statically allocate resources to processors at different points in the program. These results proved effective but they required months of hard work and intense profiling.

To aid in the development of TIDeFlow, a careful study of Garcia's approaches with matrix multiply [46, 50, 44, 47] revealed many interesting issues in program development for manycore architectures.

The study confirms that successfully writing a high-performance program for a manycore architecture requires:

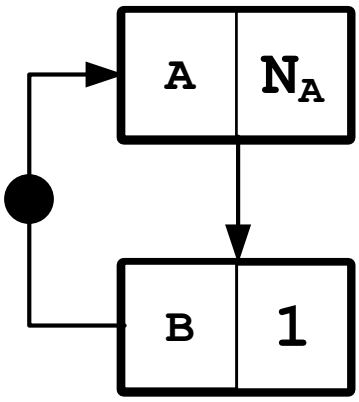
- a profiler able to produce program traces,
- a programming model with native support for parallel loops,
- constructs for complex producer-consumer relationships as well as data or resource dependencies,
- a priority system that enables the runtime system to select between tasks in the critical path and other, less important tasks and
- a mechanism for composability, where parallel constructs can be reused.

The TIDeFlow programming interface and the TIDeFlow runtime system provide mechanisms to address each one of those necessities.

The following sections present a detailed description of how each one of the necessities of efficient programming for manycore architectures was addressed along with a presentation of the interface provided to the user.

Experiments showing that TIDeFlow's constructs have very low overhead are shown as well.

At the end of this chapter, a summary of the features and their usability is presented.



The basic time loop construct. A is executed repeatedly. B controls the termination condition.

Figure 6.1: Time loop construct.

```
do{
  parallel loop A;
} while( B() == CONTINUE );
```

A pseudocode equivalent of the program of Figure 6.1. Parallel loop A is executed repeatedly. Actor B controls the termination of execution. Time instance variable not shown.

Figure 6.2: Serial code for a time loop.

This section presents common program constructs used by TIDeFlow programs.

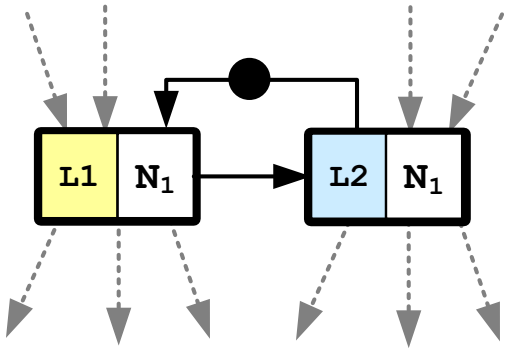
6.1 Time Loop

The objective of the time loop is to execute a part of a program repeatedly. The time loop is usually found in tiled applications or in simulations of physical phenomena where time is advanced.

Figure 6.1 shows the basic construct. The actor A will be executed repeatedly, and the termination condition will be controlled by actor B.

The program of Figure 6.1 holds an analogy to the program of Figure 6.2.

The basic time loop construct can be expanded to include more than just the actor A in the loop. A more complicated TIDeFlow graph where all the arcs have a



Construct for memory bandwidth allocation. The dependency from memory copy actor $L1$ to memory copy actor $L2$ guarantees ordering between $L1$ and $L2$. The single token in the dependency loop between $L1$ and $L2$ ensures that at most one of the actors is enabled at any given time. The effect is mutual exclusion between $L1$ and $L2$.

Figure 6.3: Mutual exclusion construct.

weight of zero, plus a backwards arc with a weight of 1 will result in more powerful time loops.

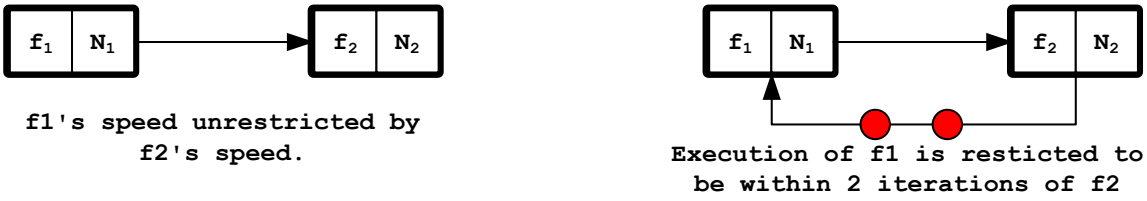
6.2 Bandwidth Allocation Between Loaders

Resource management in manycore architectures is of paramount importance to achieve high performance and low power execution.

The main construct to manage a resource, such as bandwidth, is to create a predefined order between them, as shown by Figure 6.3. The behavior achieved by the construct is mutual exclusion between $L1$ and $L2$. The computation actors, $C1$, $C2$ are shown to illustrate how to place other actors that may make use of the data loaded. A backedge completing a time loop is not shown in the figure, but is also common in programs that use this construct.

6.3 Restraining Execution Speed to Enforce Pipelining

There are some situations in which actors can fire repeatedly because they have no input dependencies or because their dependencies are produced quickly. However,



The figure shows an example of how to enforce pipelining with the addition of a backedge. The addition of the backedge restricts the execution of $f1$ to remain within two time instances of $f2$.

Figure 6.4: Construct to bound the relative execution of two actors.

there may be other slow actors that depend on the fast actor who are unable to match the faster rate of computation.

The problem in this situation, shown in the left of Figure 6.4, is that the computation will not settle into a pipelined execution, and it can result in poor utilization of resources such as memory bandwidth.

Fortunately, the computation progress of a fast actor can be constrained to be within a certain range of the progress of the slow actor with the addition of backedges. In the example of Figure 6.4, a backedge has been added (with two tokens) to ensure that the computation of $f1$ remains within two time instances of the computation of $f2$.

The addition of the backedge is a powerful tool to restrain the execution speed of certain actors, allowing the pipelined execution.

Chapter 7

EXAMPLES

This section presents the case of several programs where TIDeFlow has been used successfully to describe and execute programs for manycore architectures.

In all cases, the programs were developed for C64. Results, where available, are provided for each one of them.

7.1 Matrix Multiplication

The TIDeFlow implementation of Matrix Multiplication leveraged on the previous work by Garcia [46, 50]. The program developed used Garcia's highly optimized codelets, but replaced all the communication, synchronization and task management with TIDeFlow operations.

The program used for Matrix Multiply is shown in Figure 7.1. Several features are of interest:

- The program uses typical loop constructs such as the ones presented in Section 6.1,
- The program is modular. Extensive reuse of modules has been used.
- Although mutual exclusion is not specified between the loader elements in the program, at runtime, the sequence of operations naturally falls back to a pipeline of events as evidenced by the trace shown in Figure 7.2.

Execution of Matrix Multiply on C64 has resulted in a performance of 53 GFLOPS, out of 80 GFLOPS possible.

The experiment with Matrix Multiply is successful since it achieved a reasonably high performance with very little programming effort.

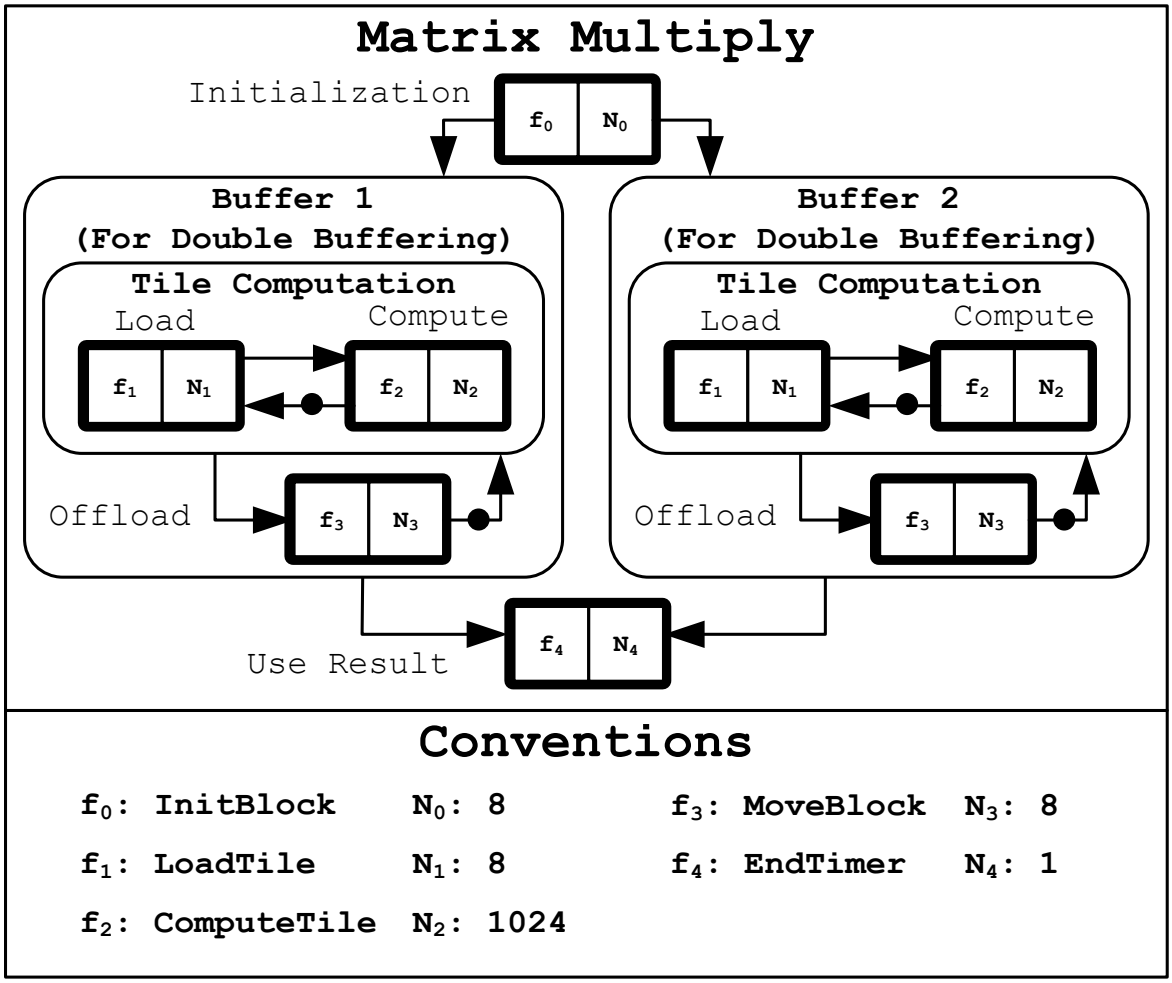


Figure 7.1: Example of a matrix multiplication program graph.

7.2 Reverse Time Migration

Reverse Time Migration [14] was interesting in its development due to its simple naive implementation and the many transformations required to optimize it.

The naive version of Reverse Time Migration is memory bound. Each floating point operation requires, approximately, one operand from DRAM memory. DRAM memory operations are the bottleneck because there are 80 floating point units but only 4 DRAM memory banks.

Data locality in the program was improved through a series of code transformations. Diamond Tiling was used at the outermost level [81, 82, 83], Skewed Tiling was used at the on-chip memory level, and register tiling was also used.

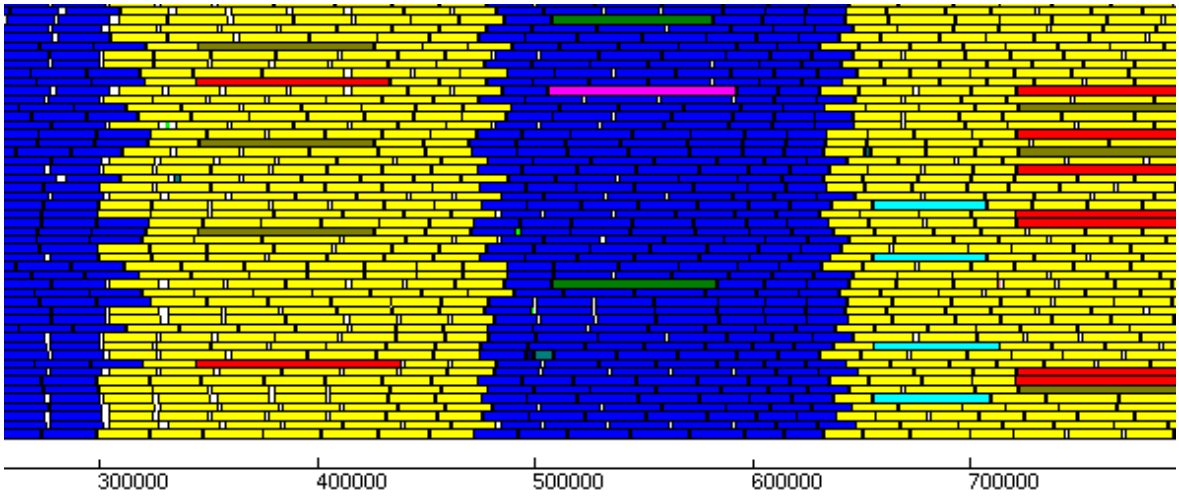


Figure 7.2: Execution trace of matrix multiplication.

To achieve proper pipelining, overlapping of communication and computation was done through the use of two buffers. Additional dependencies were used to limit the amount of parallelism that was exploited, and backedges were added to avoid overwriting of buffers.

The resulting RTM program (Figure 7.3) failed to approach peak performance in C64 due to the lack of highly specialized codelets to perform the computation. However, the experiment with RTM is a success in terms of the usability of TIDeFlow and its advantages for parallelism and task management.

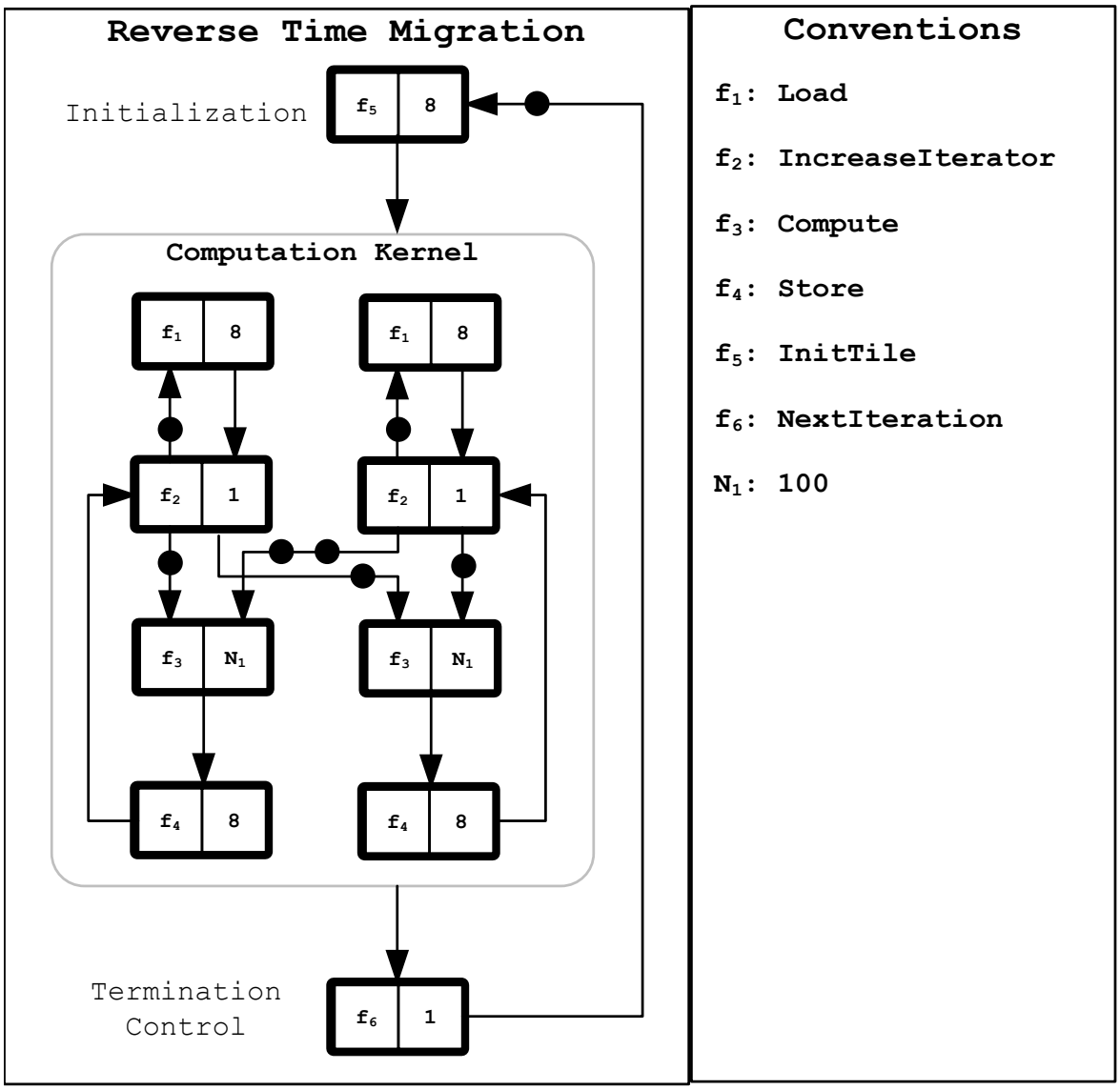


Figure 7.3: TIDeFlow program for Reverse Time Migration.

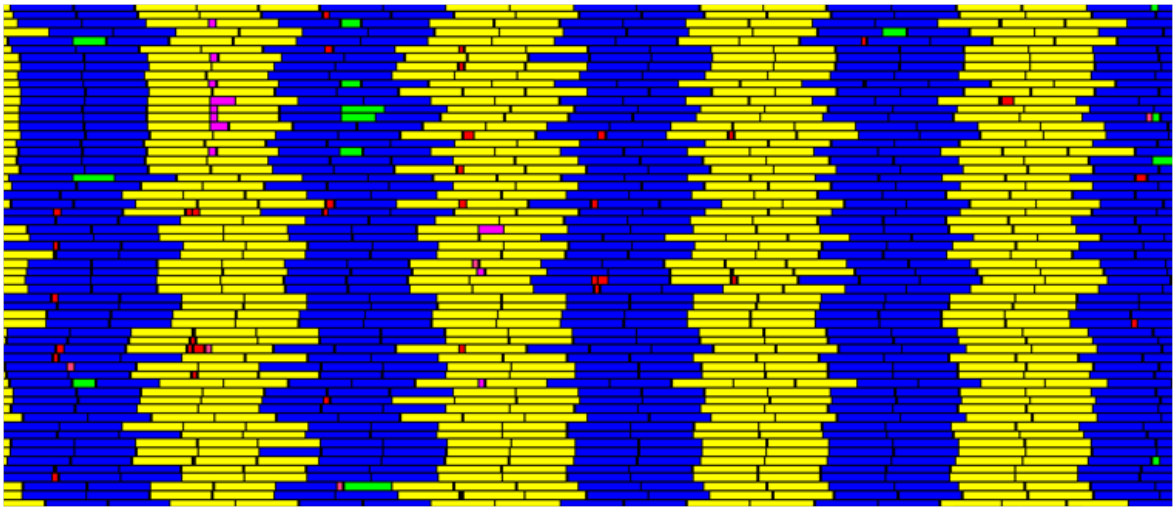


Figure 7.4: Execution trace of Reverse Time Migration.

Chapter 8

CONCLUSIONS

This thesis presented the TIDeFlow model and its implementation. TIDeFlow is a parallel execution model designed to express and execute HPC programs.

The contributions of this work are:

1. The description of a new execution model that extends the dataflow model of computation, including:
   - the proposition that parallel loops can be natively represented as single actors using weighted nodes,
   - the idea that loop carried dependencies can be represented as weighted arcs and
   - the ability to express task pipelining and overlapping of communication and computation.
2. The advancement of the idea of intrinsic throughput of algorithms, including
   - a systematic way to analyze the throughput of algorithms,
   - evidence supporting the importance of throughput for the scalability of parallel programs,
   - the development of several high-throughput algorithms that are useful to construct runtime systems and other highly parallel programs and
   - an analysis showing that algorithm properties such as the nonblocking property do no contribute to make an algorithm scalable or fast.
3. The implementation of the TIDeFlow system including
   - a design for decentralized control, where no one processor or thread is responsible for task management,
   - the development of a technique for representation of TIDeFlow programs that allows distributed execution of dataflow programs and
   - the development of a complete toolchain, including a compiler, a programming interface and a program launcher.

In this thesis, the TIDeFlow model has been formally defined: the operational semantics of execution have been described as Finite State Machines, a brief discussion of the memory model was presented, and the method by which to use weighted arcs to express loop carried dependencies has been described.

The experience of implementing the runtime system and executing the experiments shows that TIDeFlow is an effective way to develop and execute parallel programs. The advantages of using graphs to express parallelism were shown. They represent an improvement over the traditional synchronization primitives used in serial programs. It was also shown that TIDeFlow provides very good scalability and low overhead, partly due to its distributed runtime system and the new algorithms that have been developed for it.

Developing the programs presented in this thesis revealed good characteristics offered by TIDeFlow: (1) it was easy to express double buffering through reusing parts of the program and using a weighted arc to indicate a time dependency between the loader stages, in the fashion presented in Figure 6.3 (2) the dependencies expressed through weighted arcs resulted in good task pipelining during execution, (3) it was easier to express the dependencies through a graph rather than through other direct means such as conditional variables or MPI processes and (4) load balancing was done automatically by the runtime system. Those reasons demonstrate that TIDeFlow is a good choice for execution and development of HPC programs in manycore architectures.

This thesis also developed the concept of *intrinsic throughput* of algorithms, which was used to develop the CB-Queue and the HT-Queue in Chapter 4. Both algorithms have a large throughput and low latency. TIDeFlow, and in general, dataflow runtime systems and operating systems can benefit from the CB-Queue implementation. The HT-Queue serves as a viable replacement for traditional queues because it matches their functionality, and it exhibits excellent throughput and low latency.

The CB-Queue and the HT-Queue have been shown to have exceptional performance due to their very high throughput and very low latency. High throughput is a result of executing the critical parts of the queue operations *in memory* through the use

of atomic instructions as opposed to attempting inquire-then-update operations that are common to other implementations. The difference is important: An in-memory operation can complete in very few cycles, allowing more requests to be completed per unit of time than a read-modify-write approach, where at the very least, a roundtrip to memory plus some processor involvement is required for every access to the queue.

Chapter 9

FUTURE WORK

The work presented in this thesis made important advances to the execution of programs in manycore systems. Unfortunately, the work with TIDeFlow is by no means a silver bullet that solves all the problems currently faced by the field.

Not all problems of parallel execution have been addressed. In particular, important issues such as energy optimization and resiliency have not been addressed by this thesis, and in general, a satisfactory solution has not been found.

Additionally, the techniques and theories for the TIDeFlow model can be further improved in functionality and scope. Specifically, further research on throughput will be profitable, and further improvement of the TIDeFlow runtime system will allow a more flexible way to express programs.

The following sections describe possible research directions that will address some of the open questions that remain, or that will improve the current tools for parallel execution.

9.1 Open Questions

Resiliency to failure of individual processors in large systems is an important question that remains open. The traditional methods for resiliency that rely on global partitioning and synchronization of programs are not fast enough to be used with systems with millions of processor cores. The current concept of global partitioning and synchronization is also unsuited for dataflow-style of computations like TIDeFlow.

How to effectively execute dataflow-like programs in a resilient way is still an open question. The current approach to resiliency, where a global synchronization operation is followed by a global checkpoint, will not work because failures in exascale

supercomputer systems will happen more frequently than global checkpoints can be saved [64].

Several major accomplishments will have to be achieved. First, a technique for local checkpoints will have to be developed, in which checkpoints are done on the local state of a processor and not on the machine as a whole. Second, new local work sharing and redundant policies will have to be implemented to detect and recover from local failures. Third, resiliency techniques will have to decrease their dependence on hard drive systems and permanent storage because of its low bandwidth. Finally, a theoretical model for resiliency aimed at dataflow programs needs to be developed, since most current resiliency approaches assume that the computation uses message passing techniques such as MPI.

How to reduce the total energy required to execute a program is another important open problem in high performance computing. Current trends in power usage by supercomputers [17] suggest that the power required to run computers in the exascale era will be on the order of one gigawatt. Unfortunately, this amount of power is not practically obtainable due to its cost and the difficulty in obtaining it. New techniques will have to be developed to reduce the power used by processors, possibly through a combination of processors that are more power-efficient and algorithms that optimize energy consumption rather than computational performance.

Future execution models will need to include energy consumption as part of the optimization operations to be performed. Several changes will need to be made. Runtime systems and schedulers will need to be aware of the costs of executing particular operations, and they will need to orchestrate computations in a way that minimizes energy consumption. Programming models might be changed to support annotation of power-intensive operations, such as memory movement. Overall, the execution model used must provide mechanisms that support the optimization of energy rather than the optimization of computational speed.

The relationship between resource management and synchronization remains an open question. There is an ongoing dilemma on whether or not to have resource

management and synchronization be separate problems. This thesis took the path of separating resource management and synchronization into different processes. However, an open question remains: Is this the best approach? Can, and should, they be performed together? The approach of separating resource management and synchronization have worked appropriately for a single manycore architecture. However, it remains to be seen if such an approach will work for larger systems.

9.2 Improvements to Current Techniques

The throughput theory advanced in Chapter 4 provided the analytical foundation to find the maximum rate at which parallel operations can be executed. Unfortunately, the exposition of Chapter 4 is currently limited to analyze simple parallel operations, and it has only been tested for one particular kind of processor. Nevertheless, future work on throughput is likely to provide good payoffs that will serve to understand throughput for arbitrary architectures and its relationship to parallel programs.

One of the possible future research directions is exploring the relationships between throughput and algorithm latencies. Although some initial progress was accomplished by this thesis, a comprehensive theoretical model that relates throughput to other parameters in the system is still lacking.

Fully understanding the relationship between throughput and parallel execution will likely require a good statistical characterization of parallel programs, either as a whole or individually. This characterization may describe the statistical behavior of the scheduler, the speed of the synchronization operations, the average number of tasks that are enabled per unit of time and so on. Identifying which ones are the relevant parameters is a research question in itself.

The throughput experiments must be extended to include other architectures beyond Cyclops-64. So far the theoretical models for throughput have matched the experimental evidence found for Cyclops-64. However, it is still important to evaluate

the validity of the throughput idea in other architectures to see if it is still able to predict the rate of computation of parallel programs.

BIBLIOGRAPHY

[1] Cilk arts. http://www.cilk.com.

[2] Intel® threading building blocks. getting started guide. Technical report, intel.

[3] The swarm runtime system. http://www.etinternational.com/swarm/.

[4] 3rd generation intel ©core ™desktop processor family, product brief, 2012.

[5] Anant Agarwal. The tile processor: A 64-core multicore for embedded processing. In *Proceedings of HPEC Workshop*, 2007.

[6] Emmanuel Agullo, Jim Demmel, Jack Dongarra, Bilel Hadri, Jakub Kurzak, Julien Langou, Hatem Ltaief, Piotr Luszczek, and Stanimire Tomov. Numerical linear algebra on emerging architectures: The plasma and magma projects. volume 180, page 012037, 2009.

[7] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, AFIPS '67 (Spring), pages 483–485, New York, NY, USA, 1967. ACM.

[8] E. Anderson, Z. Bai, J. Dongarra, A. Greenbaum, A. McKenney, J. Du Croz, S. Hammerling, J. Demmel, C. Bischof, and D. Sorensen. Lapack: a portable linear algebra library for high-performance computers. In *Proceedings of the 1990 ACM/IEEE conference on Supercomputing*, Supercomputing '90, pages 2–11, Los Alamitos, CA, USA, 1990. IEEE Computer Society Press.

[9] A. Arvind and K. P. Gostelow. The u-interpreter. *Computer*, 15:42–49, February 1982.

[10] K. Arvind and Rishiyur S. Nikhil. Executing a program on the mit tagged-token dataflow architecture. *IEEE Trans. Comput.*, 39:300–318, March 1990.

[11] H.E. Bal, J.G. Steiner, and A.S. Tanenbaum. Programming languages for distributed computing systems. *ACM Computing Surveys (CSUR)*, 21(3):261–322, 1989.

[12] Rajkishore Barik, Zoran Budimlić, Vincent Cave, Sanjay Chatterjee, Yi Guo, David Peixotto, Raghavan Raman, Jun Shirako, Saugnak Tacsirlar, Yonghong Yan, Yisheng Zhao, and Vivek Sarkar. The habanero multicore software research project. In *OOPSLA '09: Proceeding of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*, pages 735–736, New York, NY, USA, 2009. ACM.

[13] Rajkishore Barik, Zoran Budimlic, Vincent Cave, Sanjay Chatterjee, Yi Guo, David Peixotto, Raghavan Raman, Jun Shirako, Sagnak Tasirlar, Yonghong Yan, Yisheng Zhao, and Vivek Sarkar. The habanero multicore software research project. In *Proceeding of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*, OOPSLA '09, pages 735–736, New York, NY, USA, 2009. ACM.

[14] Edip Baysal, Dan D. Kosloff, and John W. C. Sherwood. Reverse time migration. *Geophysics*, 48, 1983.

[15] R.D. Blumofe and C.E. Leiserson. Space-efficient scheduling of multithreaded computations. In *Proceedings of the twenty-fifth annual ACM symposium on Theory of computing*, pages 362–371. ACM, 1993.

[16] R.D. Blumofe and C.E. Leiserson. Scheduling multithreaded computations by work stealing. In *Foundations of Computer Science, 1994 Proceedings., 35th Annual Symposium on*, pages 356–368. IEEE, 1994.

[17] Shekhar Borkar. Major challenges to achieve exascale performance. In *Salishan Conference on High-Speed Computing*, 2009.

[18] Z. Budimlić, M. Burke, V. Cavé, K. Knobe, G. Lowney, R. Newton, J. Palsberg, D. Peixotto, V. Sarkar, F. Schlimbach, et al. Concurrent collections. *Scientific Programming*, 18(3):203–217, 2010.

[19] Z. Budimlic, A. Chandramowlishwaran, K. Knobe, G. Lowney, V. Sarkar, and L. Treggiari. Multi-core implementations of the concurrent collections programming model. In *CPC09: 14th International Workshop on Compilers for Parallel Computers*, 2009.

[20] David R. Butenhof. *Programming with POSIX threads*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.

[21] D.R. Butenhof. *Programming with POSIX threads*. Addison-Wesley Professional, 1997.

[22] V. Cavé, J. Zhao, J. Shirako, and V. Sarkar. Habanero-java: the new adventures of old x10. In *Proceedings of the 9th International Conference on the Principles and Practice of Programming in Java (PPPJ)*, 2011.

[23] A. Chandramowlishwaran, K. Knobe, and R. Vuduc. Performance evaluation of concurrent collections on high-performance multicore computing systems. In *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1–12. IEEE, 2010.

[24] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *Proceedings of the 20th annual ACM SIGPLAN conference on OOPSLA*, pages 519–538, New York, NY, USA, 2005. ACM.

[25] F. Chen, K.B. Theobald, and G.R. Gao. Implementing parallel conjugate gradient on the earth multithreaded architecture. In *Cluster Computing, 2004 IEEE International Conference on*, pages 459–469. IEEE, 2004.

[26] Intel®Corporation. Discontinued processors, intel microprocessor export compliance metrics.

[27] Intel®Corporation. Intel® core™ 2 duo desktop processors (desktop).

[28] Intel®Corporation. Intel®pentium ® 3 processors (desktop).

[29] Intel®Corporation. Intel®pentium ® 4 processors (desktop).

[30] Intel®Corporation. Intel®pentium ®processor 900 series.

[31] Juan del Cuvillo, Weirong Zhu, Ziang Hu, and Guang R. Gao. TiNy Threads: A Thread Virtual Machine for the Cyclops64 Cellular Architecture. In *IPDPS '05: Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05)*, page 265.2, Washington, DC, USA, 2005. IEEE Computer Society.

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

[33] J. del Cuvillo, W. Zhu, Z. Hu, and G. R. Gao. FAST: A Functionally Accurate Simulation Toolset for the Cyclops-64 Cellular Architecture. In *Proceedings of the 2005 Workshop on Modeling, Benchmarking, and Simulation (MoBS 2005)*, Madison, Wisconsin, June 2005.

[34] Juan del Cuvillo, Weirong Zhu, and Guang Gao. Landing openmp on cyclops-64: an efficient mapping of openmp to a many-core system-on-a-chip. In *CF '06: Proceedings of the 3rd conference on Computing frontiers*, pages 41–50, New York, NY, USA, 2006. ACM.

[35] Juan del Cuvillo, Weirong Zhu, Ziang Hu, and Guang Gao. Fast: A functionally accurate simulation toolset for the cyclops-64 cellular architecture. *CAPSL Technical Memo 062*, 2005.

[36] Juan del Cuvillo, Weirong Zhu, Ziang Hu, and Guang R. Gao. Toward a software infrastructure for the cyclops-64 cellular architecture. In *High-Performance Computing in an Advanced Collaborative Environment, 2006.*, page 9, May 2006.

[37] J. B. Dennis. First version of a data flow procedure language. In *Programming Symposium, Proceedings Colloque sur la Programmation*, pages 362–376, London, UK, 1974. Springer-Verlag.

[38] J.B. Dennis and G.R. Gao. An efficient pipelined dataflow processor architecture. In *Proceedings of the 1988 ACM/IEEE conference on Supercomputing*, pages 368–373. IEEE Computer Society Press, 1988.

[39] J. Ellson, E. Gansner, L. Koutsofios, S. North, and G. Woodhull. Graphvizopen source graph drawing tools. In *Graph Drawing*, pages 594–597. Springer, 2002.

[40] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the cilk-5 multithreaded language. In *Proceedings of the ACM SIGPLAN Conference on PLDI*, PLDI '98, pages 212–223, New York, NY, USA, 1998. ACM.

[41] G.R. Gao, HHJ Hum, and Y.B. Wong. Parallel function invocation in a dynamic argument-fetching dataflow architecture. In *Databases, Parallel Architectures and Their Applications,. PARBASE-90, International Conference on*, pages 112–116. IEEE, 1990.

[42] G.R. Gao, T. Sterling, R. Stevens, M. Hereld, and W. Zhu. Parallex: A study of a new parallel computation model. In *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, pages 1–6. IEEE, 2007.

[43] Elkin Garcia, Rishi Khan, Kelly Livingston, Ioannis Venetis, and Guang Gao. Dynamic percolation - mapping dense matrix multiplication on a many-core architecture. *CAPSL Technical Memo 098*, June 2010.

[44] Elkin Garcia, Rishi Khan, Kelly Livingston, Ioannis E. Venetis, and Guang R. Gao. Dynamic percolation: mapping dense matrix multiplication on a many-core architecture. Technical report, CAPSL Tech. memo 098, University of Delaware, 2010.

[45] Elkin Garcia, Daniel Orozco, and Guang R. Gao. Energy efficient tiling on a many-core architecture. *Fourth Workshop on Programmability Issues for Multi-Core Computers (MULTIPROG-2011)*, page 50, 2010.

[46] Elkin Garcia, Daniel Orozco, Rishi Khan, Ioannis E. Venetis, Kelly Livingston, and Guang R Gao. Dynamic percolation: A case of study on the shortcomings of traditional optimization in many-core architectures. *ACM International Conference on Computing Frontiers 2012 (CF'12)*, 2012.

[47] Elkin Garcia, Daniel Orozco, Robert Pavel, and Guang R. Gao. A discussion in favor of dynamic scheduling for regular applications in many-core architectures.

[48] Elkin Garcia, Daniel Orozco, Robert Pavel, and Guang R. Gao. Toward efficient fine-grained dynamic scheduling on many-core architectures. 2012.

[49] Elkin Garcia, Ioannis E. Venetis, Rishi Khan, and Guang Gao. Optimized Dense Matrix Multiplication on a Many-Core Architecture. In *Proceedings of the Sixteenth International Conference on Parallel Computing (Euro-Par 2010), Part II*, volume 6272 of *Lecture Notes in Computer Science*, pages 316–327, Ischia, Italy, August 2010. Springer.

[50] Elkin Garcia, Ioannis E. Venetis, Rishi Khan, and Guang R. Gao. Optimized dense matrix multiplication on a many-core architecture. *Euro-Par 2010-Parallel Processing*, pages 316–327, 2010.

[51] T. Gautier, X. Besseron, and L. Pigeon. Kaapi: A thread scheduling runtime system for data flow computations on cluster of multi-processors. In *Proceedings of the 2007 international workshop on Parallel symbolic computation*, pages 15–23. ACM, 2007.

[52] KP Gostelow et al. U-interpreter. *Computer;(United States)*, 2, 1982.

[53] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the mpi message passing interface standard. *Parallel computing*, 22(6):789–828, 1996.

[54] J.R. Gurd, C.C. Kirkham, and I. Watson. The manchester prototype dataflow computer. *Communications of the ACM*, 28(1):34–52, 1985.

[55] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data flow programming language lustre. *Proceedings of the IEEE*, 79(9):1305 –1320, sep 1991.

[56] J.L. Hennessy and D.A. Patterson. *Computer architecture: a quantitative approach*. Morgan Kaufmann Pub, 2011.

[57] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12:463–492, July 1990.

[58] C.A.R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978.

[59] Z. Hu, J. del Cuvillo, W. Zhu, and G. R. Gao. Optimization of Dense Matrix Multiplication on IBM Cyclops-64: Challenges and Experiences. In *12th International European Conference on Parallel Processing (Euro-Par 2006)*, pages 134–144, Dresden, Germany, August 2006.

[60] F. Irigoin and R. Triolet. Supernode partitioning. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 319–329. ACM, 1988.

[61] H. Jula, D. Tralamazza, C. Zamfir, and G. Candea. Deadlock immunity: Enabling systems to defend against deadlocks. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, pages 295–308. USENIX Association, 2008.

[62] G. Kahn. The semantics of a simple language for parallel programming. *proceedings of IFIP Congress74*, 74:471–475, 1974.

[63] Leonard Kleinrock. *Queueing Systems. Volume 1: Theory.* 1975.

[64] Peter Kogge, Keren Bergman, Shekhar Borkar, Dan Campbell, William Carlson, William Dally, Monty Denneau, Paul Franzon, William Harrod, Kerry Hill, Jon Hiller, Sherman Karp, Stephen Keckler, Dean Klein, Robert Lucas, Mark Richards, Al Scarpelli, Steven Scott, Allan Snavely, Thomas Sterling, R. Stanley Williams, and Katherine Yelick. Exascale computing study: Technology challenges in achieving exascale systems peter kogge, editor & study lead. 2008.

[65] P. Kongetira, K. Aingaran, and K. Olukotun. Niagara: a 32-way multithreaded sparc processor. *Micro, IEEE*, 25(2):21 – 29, march-april 2005.

[66] E.A. Lee and T.M. Parks. Dataflow process networks. *Proceedings of the IEEE*, 83(5):773–801, 1995.

[67] X. Martorell, J. Labarta, N. Navarro, and E. Ayguadé. Nano-threads library design, implementation and evaluation. *Dept. dArquitectura de Computadors-Universitat Politècnica de Catalunya. Technical Report: UPC-DAC-1995-33*, 1995.

[68] James R. McGraw. The val language: Description and analysis. *ACM Trans. Program. Lang. Syst.*, 4:44–82, January 1982.

[69] Paul E. McKenney. *Is Parallel Programming Hard, And, If So, What Can You Do About It?* Linux Technology Center, IBM Beaverton, 2011.

[70] John Mellor-Crummey. Concurrent queues: Practical fetch and phi algorithms. *Tech. Rep. 229, Dep. of CS, University of Rochester*, 1987.

[71] Maged M. Michael and Michael L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proc. of the 15th ACM symposium on Principles of distributed computing*, PODC '96, pages 267–275, New York, NY, USA, 1996. ACM.

[72] D. Minor and S. Rippa. Grape-an industrial distributed system for computer vision. In *Parallel and Distributed Processing Symposium, 2005. Proceedings. 19th IEEE International*, pages 8–pp. IEEE, 2005.

[73] G.E. Moore. Cramming more components onto integrated circuits. *Proceedings of the IEEE*, 86(1):82 –85, jan 1998.

[74] Tadao Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541 –580, apr 1989.

[75] W.A. Najjar, E.A. Lee, and G.R. Gao. Advances in the dataflow computational model. *Parallel Computing*, 25(13-14):1907–1929, 1999.

[76] R.H.B. Netzer and B.P. Miller. What are race conditions?: Some issues and formalizations. *ACM Letters on Programming Languages and Systems (LOPLAS)*, 1(1):74–88, 1992.

[77] Anthony Nguyen, Nadathur Satish, Jatin Chhugani, Changkyu Kim, and Pradeep Dubey. 3.5-d blocking optimization for stencil computations on modern cpus and gpus. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–13, Washington, DC, USA, 2010. IEEE Computer Society.

[78] C. Nvidia. Programming guide, 2008.

[79] OpenMP 3.1 specification.

[80] Daniel Orozco. Tideflow: A parallel execution model for high performance computing programs. *2011 International Conference on Parallel Architectures and Compilation Techniques*, page 211, 2011.

[81] Daniel Orozco and Guang R. Gao. Diamond tiling: A tiling framework for time-iterated scientific applications. Technical report, CAPSL Technical Memo 91. University of Delaware, 2009.

[82] Daniel Orozco and Guang R. Gao. Mapping the fdtd application to many-core chip architectures. In *Parallel Processing, 2009. ICPP'09. International Conference on*, pages 309–316. IEEE, 2009.

[83] Daniel Orozco, Elkin Garcia, and Guang R. Gao. Locality optimization of stencil applications using data dependency graphs. *Languages and Compilers for Parallel Computing*, pages 77–91, 2011.

[84] Daniel Orozco, Elkin Garcia, Rishi Khan, Kelly Livingston, and GR Gao. High throughput queue algorithms. Technical report, CAPSL Technical Memo 103, 2011.

[85] Daniel Orozco, Elkin Garcia, Rishi Khan, Kelly Livingston, and Guang R. Gao. Toward high-throughput algorithms on many-core architectures. *ACM Transactions on Architecture and Code Optimization (TACO)*, 8(4):49, 2012.

[86] Daniel Orozco, Elkin Garcia, Robert Pavel, Rishi Khan, and Guang R. Gao. Polytasks: A compressed task representation for hpc runtimes. *Proceedings of the 24th international conference on Languages and compilers for parallel computing, LCPC*, 11, 2011.

[87] Daniel Orozco, Elkin Garcia, Robert Pavel, Rishi Khan, and Guang R. Gao. Polytasks: A compressed task representation for hpc runtimes. *CAPSL Technical Memo 105*, 11, 2011.

[88] Daniel Orozco, Elkin Garcia, Robert Pavel, Rishi Khan, and Guang R. Gao. Tideflow: The time iterated dependency flow execution model. Technical report, CAPSL Technical Memo 107, 2011.

[89] Daniel Orozco, Elkin Garcia, Robert Pavel, Rishi Khan, and Guang R. Gao. Tideflow: The time iterated dependency flow execution model. In *2011 First Workshop on Data-Flow Execution Models for Extreme Scale Computing (DFM 2011)*, 2011.

[90] Daniel Orozco, Liping Xue, Murat Bolat, Xiaoming Li, and Guang R. Gao. Experience of optimizing fft on intel architectures. In *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, pages 1–8. IEEE, 2007.

[91] Gregory M. Papadopoulos and David E. Culler. Monsoon: an explicit token-store architecture. In *Proceedings of the 17th annual international symposium on Computer Architecture*, ISCA '90, pages 82–91, New York, NY, USA, 1990. ACM.

[92] Carl Adam Petri. Communication with automata: Volume 1 supplement 1. Technical report, DTIC Document, 1966.

[93] Chuck Pheatt. Intel©threading building blocks. *J. Comput. Sci. Coll.*, 23(4):298–298, April 2008.

[94] Srinivas K. Raman, Vladimir Pentkovski, and Jagannath Keshava. Implementing streaming simd extensions on the pentium iii processor. *IEEE Micro*, 20(4):47–57, July 2000.

[95] RM Ramanathan. Intel® multi-core processors. *Making the Move to Quad-Core and Beyond*, 2006.

[96] Version Rishiyur, Rishiyur S. Nikhil, and Rishiyur S. Nikhil. Id language reference manual, 1991.

[97] James Rumbaugh. A data flow multiprocessor. *IEEE Trans. Comput.*, 26:138–146, February 1977.

[98] Vivek Sarkar and John Hennessy. Partitioning parallel programs for macro-dataflow. In *Proceedings of the 1986 ACM conference on LISP and functional programming*, LFP '86, pages 202–211, New York, NY, USA, 1986. ACM.

[99] M. Shah, J. Barren, J. Brooks, R. Golla, G. Grohoski, N. Gura, R. Hetherington, P. Jordan, M. Luttrell, C. Olson, et al. Ultrasparc t2: A highly-treaded, power-efficient, sparc soc. In *Solid-State Circuits Conference, 2007. ASSCC'07. IEEE Asian*, pages 22–25. Ieee, 2007.

[100] Jun Shirako, David M. Peixotto, Vivek Sarkar, and William N. Scherer. Phasers: a unified deadlock-free construct for collective and point-to-point synchronization. In *Proceedings of the 22nd ICS*, pages 277–288, New York, NY, USA, 2008.

[101] Herb Sutter. Welcome to the parallel jungle!

[102] Guangming Tan, Vugranam Sreedhar, and Guang R. Gao. Just-in-time locality and percolation for optimizing irregular applications on a manycore architecture. *Languages and Compilers for Parallel Computing*, pages 331–342, 2008.

[103] The Top500 List. http://www.top500.org.

[104] Kevin Theobald. *EARTH: An Efficient Architecture for Running Threads*. PhD thesis, 1999.

[105] Kevin B. Theobald, Gagan Agrawal, Rishi Kumar, Gerd Heber, Guang R. Gao, Paul Stodghill, and Keshav Pingali. Landing cg on earth: a case study of fine-grained multithreading on an evolutionary path. In *Supercomputing '00: Proceedings of the 2000 ACM/IEEE conference on Supercomputing (CDROM)*, page 4, Washington, DC, USA, 2000. IEEE Computer Society.

[106] Kevin B. Theobald, Rishi Kumar, Gagan Agrawal, Gerd Heber, Ruppa K. Thulasiram, and Guang R. Gao. Developing a communication intensive application on the earth multithreaded architecture. In *Euro-Par 2000 Parallel Processing*, 2000.

[107] Kevin B. Theobald, Rishi Kumar, Gagan Agrawal, Gerd Heber, Ruppa K. Thulasiram, and Guang R. Gao. Developing a communication intensive application on the earth multithreaded architecture. In *Euro-Par 2000 Parallel Processing*, pages 625–637. Springer, 2000.

[108] William Thies, Michal Karczmarek, and Saman P. Amarasinghe. Streamit: A language for streaming applications. In *Proceedings of the 11th International Conference on Compiler Construction*, CC '02, pages 179–196, London, UK, UK, 2002. Springer-Verlag.

[109] James E. Thornton. The cdc 6600 project. *IEEE Ann. Hist. Comput.*, 2(4):338–348, October 1980.

[110] Josep Torrellas, Monica S. Lam, and John L. Hennessy. False sharing and spatial locality in multiprocessor caches. *Computers, IEEE Transactions on*, 43(6):651–663, 1994.

[111] Arthur H. Veen. Dataflow machine architecture. *ACM Comput. Surv.*, 18:365–396, December 1986.

[112] Vijay Saraswat and Nathaniel Nystrom. Report on the Experimental Language X10, Version 1.7. Technical report, September 2008.

[113] William W. Wadge and Edward A. Ashcroft. *LUCID, the dataflow programming language*. Academic Press Professional, Inc., San Diego, CA, USA, 1985.

[114] David Wentzlaff, Patrick Griffin, Henry Hoffmann, Liewei Bao, Bruce Edwards, Carl Ramey, Matthew Mattina, Chyi-Chang Miao, John F. Brown III, and Anant Agarwal. On-chip interconnection architecture of the tile processor. *IEEE Micro*, 27(5):15–31, 2007.

[115] Yonghong Yan, Sanjay Chatterjee, Daniel Orozco, Elkin Garcia, Zoran Budimlić, Jun Shirako, Robert Pavel, Guang R. Gao, and Vivek Sarkar. Hardware and software tradeoffs for task synchronization on manycore architectures. *Euro-Par 2011 Parallel Processing*, pages 112–123, 2011.

[116] Yonghong Yan, Sanjay Chatterjee, Daniel Orozco, Elkin Garcia, Jun Shirako, Zorzn Budimlic, Vivek Sarkar, and Guang R. Gao. Synchronization for dynamic task parallelism on manycore architectures. 2010.

[117] Weirong Zhu, Yanwei Niu, and Guang R. Gao. Performance portability on earth: a case study across several parallel architectures. *Cluster Computing*, 10(2):115–126, 2007.

Appendix A

COPYRIGHT INFORMATION

This thesis contains, in part, results, figures, tables and text written by me and published in scientific journals, conference proceedings or technical memos.

In some cases, the copyright for the figures, tables and text belongs to the publisher of a particular paper. Because those parts have been used in this thesis, I have obtained permission to reproduce parts of it.

This appendix contains the relevant details of the copy permissions obtained.

A.1 Permission from IEEE

The IEEE does not require authors of their papers to obtain a formal reuse license for their thesis.

Their formal policy states:

"*Requirements to be followed when using any portion (e.g., figure, graph, table, or textual material) of an IEEE copyrighted paper in a thesis:*

1. In the case of textual material (e.g., using short quotes or referring to the work within these papers) users must give full credit to the original source (author, paper, publication) followed by the IEEE copyright line ©2011 IEEE.
2. In the case of illustrations or tabular material, we require that the copyright line [Year of original publication] IEEE appear prominently with each reprinted figure and/or table.
3. If a substantial portion of the original paper is to be used, and if you are not the senior author, also obtain the senior authors approval.

Requirements to be followed when using an entire IEEE copyrighted paper in a thesis:

1. The following IEEE copyright/ credit notice should be placed prominently in the references: ©[year of original publication] IEEE. Reprinted, with permission, from [author names, paper title, IEEE publication title, and month/year of publication]
2. Only the accepted version of an IEEE copyrighted paper can be used when posting the paper or your thesis on-line.
3. In placing the thesis on the author's university website, please display the following message in a prominent place on the website: In reference to IEEE copyrighted material which is used with permission in this thesis, the IEEE does not endorse any of [university/educational entity's name goes here]'s products or services. Internal or personal use of this material is permitted. If interested in reprinting/republishing IEEE copyrighted material for advertising or promotional purposes or for creating new collective works for resale or redistribution, please go to
`http://www.ieee.org/publications_standards/publications/rights/rights_link.html`
to learn how to obtain a License from RightsLink.

If applicable, University Microfilms and/or ProQuest Library, or the Archives of Canada may supply single copies of the dissertation."

The copyright permission from IEEE applies to my works presented at the POHLL 2007 workshop [90], at the ICPP 2009 conference [82], at the DFM workshop [89] and at the PACT conference [80].

A.2 Permission from ACM

Permission has been obtained to reuse the paper on throughput presented in ACM Transactions on Architecture and Code Optimization [85], under license 2911931358693.

A.3 Permissions from Springer

Licenses have been obtained from Springer to reuse the content of my papers in this dissertation. The following are the licenses for each paper.

- EuroPar 2011 paper [115]: License Number 2878981113110.
- LCPC 2010 paper [83]: License Number 2878990333469.

A license for my LCPC 2011 paper [86] could not be obtained because it had not been officially published at the time this dissertation was written. A license for it will be obtained once the paper is published by Springer.

A.4 Papers I Own the Copyright to

Minor portions of the MULTIPROG 2011 paper [45] have been used in this thesis. As an author, I own the copyright for it.

The same holds true, at the time of writing this thesis, for CAPSL Technical Memos 091 [81], 094 [116] and 103 [84].

A.5 Copy of the Licensing Agreements

The following pages contain a copy of the licensing agreements for this work.

Title:	Mapping the FDTD Application to Many-Core Chip Architectures
Conference Proceedings:	Parallel Processing, 2009. ICPP '09. International Conference on
Author:	Orozco, D.;Guang Gao;
Publisher:	IEEE
Date:	22-25 Sept. 2009

Thesis / Dissertaion Reuse

The IEEE does not require individuals working on a thesis to obtain a formal reuse license, however, you may print out this statement to be used as a permission grant:

Requirements to be followed when using any portion (e.g., figure, graph, table, or textual material) of an IEEE copyrighted paper in a thesis:

1) In the case of textual material (e.g., using short quotes or referring to the work within these papers) users must give full credit to the original source (author, paper, publication) followed by the IEEE copyright line © 2011 IEEE.
2) In the case of illustrations or tabular material, we require that the copyright line © [Year of original publication] IEEE appear prominently with each reprinted figure and/or table.
3) If a substantial portion of the original paper is to be used, and if you are not the senior author, also obtain the senior author's approval.

Requirements to be followed when using an entire IEEE copyrighted paper in a thesis:

1) The following IEEE copyright/ credit notice should be placed prominently in the references: © [year of original publication] IEEE. Reprinted, with permission, from [author names, paper title, IEEE publication title, and month/year of publication]
2) Only the accepted version of an IEEE copyrighted paper can be used when posting the paper or your thesis on-line.
3) In placing the thesis on the author's university website, please display the following message in a prominent place on the website: In reference to IEEE copyrighted material which is used with permission in this thesis, the IEEE does not endorse any of [university/educational entity's name goes here]'s products or services. Internal or personal use of this material is permitted. If interested in reprinting/republishing IEEE copyrighted material for advertising or promotional purposes or for creating new collective works for resale or redistribution, please go to http://www.ieee.org/publications_standards/publications/rights/rights_link.html to learn how to obtain a License from RightsLink.

If applicable, University Microfilms and/or ProQuest Library, or the Archives of Canada may supply single copies of the dissertation.

ASSOCIATION FOR COMPUTING MACHINERY, INC. LICENSE TERMS AND CONDITIONS

May 18, 2012

This is a License Agreement between Daniel A Orozco ("You") and Association for Computing Machinery, Inc. ("Association for Computing Machinery, Inc.") provided by Copyright Clearance Center ("CCC"). The license consists of your order details, the terms and conditions provided by Association for Computing Machinery, Inc., and the payment terms and conditions.

License Number	2911931358693
License date	May 18, 2012
Licensed content publisher	Association for Computing Machinery, Inc.
Licensed content publication	ACM Transactions on Architecture and Code Optimization
Licensed content title	Toward high-throughput algorithms on many-core architectures
Licensed content author	Daniel Orozco, et al
Licensed content date	Jan 1, 2012
Volume number	8
Issue number	4
Type of Use	Thesis/Dissertation
Requestor type	Author of this ACM article
Is reuse in the author's own new work?	Yes
Format	Print and electronic
Portion	Full article
Will you be translating?	No
Order reference number	
Title of your thesis/dissertation	TIDEFLOW: A DATAFLOW-INSPIRED EXECUTION MODEL FOR HPC PROGRAMS
Expected completion date	May 2012
Estimated size (pages)	160
Billing Type	Credit Card
Credit card info	Master Card ending in 4810
Credit card expiration	10/2013
Total	8.00 USD
Terms and Conditions	

Rightslink Terms and Conditions for ACM Material

1. The publisher of this copyrighted material is Association for Computing Machinery, Inc. (ACM). By clicking "accept" in connection with completing this licensing transaction, you

agree that the following terms and conditions apply to this transaction (along with the Billing and Payment terms and conditions established by Copyright Clearance Center, Inc. ("CCC"), at the time that you opened your Rightslink account and that are available at any time at).

2. ACM reserves all rights not specifically granted in the combination of (i) the license details provided by you and accepted in the course of this licensing transaction, (ii) these terms and conditions and (iii) CCC's Billing and Payment terms and conditions.

3. ACM hereby grants to licensee a non-exclusive license to use or republish this ACM-copyrighted material* in secondary works (especially for commercial distribution) with the stipulation that consent of the lead author has been obtained independently. Unless otherwise stipulated in a license, grants are for one-time use in a single edition of the work, only with a maximum distribution equal to the number that you identified in the licensing process. Any additional form of republication must be specified according to the terms included at the time of licensing.

*Please note that ACM cannot grant republication or distribution licenses for embedded third-party material. You must confirm the ownership of figures, drawings and artwork prior to use.

4. Any form of republication or redistribution must be used within 180 days from the date stated on the license and any electronic posting is limited to a period of six months unless an extended term is selected during the licensing process. Separate subsidiary and subsequent republication licenses must be purchased to redistribute copyrighted material on an extranet. These licenses may be exercised anywhere in the world.

5. Licensee may not alter or modify the material in any manner (except that you may use, within the scope of the license granted, one or more excerpts from the copyrighted material, provided that the process of excerpting does not alter the meaning of the material or in any way reflect negatively on the publisher or any writer of the material).

6. Licensee must include the following copyright and permission notice in connection with any reproduction of the licensed material: "[Citation] © YEAR Association for Computing Machinery, Inc. Reprinted by permission." Include the article DOI as a link to the definitive version in the ACM Digital Library. Example: Charles, L. "How to Improve Digital Rights Management," Communications of the ACM, Vol. 51:12, © 2008 ACM, Inc. http://doi.acm.org/10.1145/nnnnnn.nnnnnn (where nnnnnn.nnnnnn is replaced by the actual number).

7. Translation of the material in any language requires an explicit license identified during the licensing process. Due to the error-prone nature of language translations, Licensee must include the following copyright and permission notice and disclaimer in connection with any reproduction of the licensed material in translation: "This translation is a derivative of ACM-copyrighted material. ACM did not prepare this translation and does not guarantee that it is an accurate copy of the originally published work. The original intellectual property contained in this work remains the property of ACM."

8. You may exercise the rights licensed immediately upon issuance of the license at the end of the licensing transaction, provided that you have disclosed complete and accurate details of

your proposed use. No license is finally effective unless and until full payment is received from you (either by CCC or ACM) as provided in CCC's Billing and Payment terms and conditions.

9. If full payment is not received within 90 days from the grant of license transaction, then any license preliminarily granted shall be deemed automatically revoked and shall be void as if never granted. Further, in the event that you breach any of these terms and conditions or any of CCC's Billing and Payment terms and conditions, the license is automatically revoked and shall be void as if never granted.

10. Use of materials as described in a revoked license, as well as any use of the materials beyond the scope of an unrevoked license, may constitute copyright infringement and publisher reserves the right to take any and all action to protect its copyright in the materials.

11. ACM makes no representations or warranties with respect to the licensed material and adopts on its own behalf the limitations and disclaimers established by CCC on its behalf in its Billing and Payment terms and conditions for this licensing transaction.

12. You hereby indemnify and agree to hold harmless ACM and CCC, and their respective officers, directors, employees and agents, from and against any and all claims arising out of your use of the licensed material other than as specifically authorized pursuant to this license.

13. This license is personal to the requestor and may not be sublicensed, assigned, or transferred by you to any other person without publisher's written permission.

14. This license may not be amended except in a writing signed by both parties (or, in the case of ACM, by CCC on its behalf).

15. ACM hereby objects to any terms contained in any purchase order, acknowledgment, check endorsement or other writing prepared by you, which terms are inconsistent with these terms and conditions or CCC's Billing and Payment terms and conditions. These terms and conditions, together with CCC's Billing and Payment terms and conditions (which are incorporated herein), comprise the entire agreement between you and ACM (and CCC) concerning this licensing transaction. In the event of any conflict between your obligations established by these terms and conditions and those established by CCC's Billing and Payment terms and conditions, these terms and conditions shall control.

16. This license transaction shall be governed by and construed in accordance with the laws of New York State. You hereby agree to submit to the jurisdiction of the federal and state courts located in New York for purposes of resolving any disputes that may arise in connection with this licensing transaction.

17. There are additional terms and conditions, established by Copyright Clearance Center, Inc. ("CCC") as the administrator of this licensing service that relate to billing and payment for licenses provided through this service. Those terms and conditions apply to each transaction as if they were restated here. As a user of this service, you agreed to those terms and conditions at the time that you established your account, and you may see them again at any time at http://myaccount.copyright.com

18. Thesis/Dissertation: This type of use requires only the minimum administrative fee. It is not a fee for permission. Further reuse of ACM content, by ProQuest/UMI or other document delivery providers, or in republication requires a separate permission license and fee. Commercial resellers of your dissertation containing this article must acquire a separate license.

Special Terms:

If you would like to pay for this license now, please remit this license along with your payment made payable to "COPYRIGHT CLEARANCE CENTER" otherwise you will be invoiced within 48 hours of the license date. Payment should be in the form of a check or money order referencing your account number and this invoice number RLNK500782141.
Once you receive your invoice for this order, you may pay your invoice by credit card. Please follow instructions provided at that time.

Make Payment To:
Copyright Clearance Center
Dept 001
P.O. Box 843006
Boston, MA 02284-3006

For suggestions or comments regarding this order, contact RightsLink Customer Support: customercare@copyright.com or +1-877-622-5543 (toll free in the US) or +1-978-646-2777.

Gratis licenses (referencing $0 in the Total field) are free. Please retain this printable license for your reference. No payment is required.

SPRINGER LICENSE
TERMS AND CONDITIONS

May 18, 2012

This is a License Agreement between Daniel A Orozco ("You") and Springer ("Springer") provided by Copyright Clearance Center ("CCC"). The license consists of your order details, the terms and conditions provided by Springer, and the payment terms and conditions.

All payments must be made in full to CCC. For payment instructions, please see information listed at the bottom of this form.

License Number	2878981113110
License date	Mar 30, 2012
Licensed content publisher	Springer
Licensed content publication	Springer eBook
Licensed content title	Hardware and Software Tradeoffs for Task Synchronization on Manycore Architectures
Licensed content author	Yonghong Yan
Licensed content date	Aug 18, 2011
Type of Use	Thesis/Dissertation
Portion	Full text
Number of copies	5000
Author of this Springer article	Yes and you are the sole author of the new work
Order reference number	None
Title of your thesis / dissertation	TIDEFLOW: A DATAFLOW-INSPIRED EXECUTION MODEL FOR HPC PROGRAMS
Expected completion date	May 2012
Estimated size(pages)	160
Total	**0.00 USD**
Terms and Conditions	

Introduction

The publisher for this copyrighted material is Springer Science + Business Media. By clicking "accept" in connection with completing this licensing transaction, you agree that the following terms and conditions apply to this transaction (along with the Billing and Payment terms and conditions established by Copyright Clearance Center, Inc. ("CCC"), at the time that you opened your Rightslink account and that are available at any time at http://myaccount.copyright.com).

Limited License

With reference to your request to reprint in your thesis material on which Springer Science and Business Media control the copyright, permission is granted, free of charge, for the use indicated in your enquiry.

Licenses are for one-time use only with a maximum distribution equal to the number that you identified in the licensing process.

This License includes use in an electronic form, provided its password protected or on the university's intranet or repository, including UMI (according to the definition at the Sherpa website: http://www.sherpa.ac.uk/romeo/). For any other electronic use, please contact Springer at (permissions.dordrecht@springer.com or permissions.heidelberg@springer.com).

The material can only be used for the purpose of defending your thesis, and with a maximum of 100 extra copies in paper.

Although Springer holds copyright to the material and is entitled to negotiate on rights, this license is only valid, provided permission is also obtained from the (co) author (address is given with the article/chapter) and provided it concerns original material which does not carry references to other sources (if material in question appears with credit to another source, authorization from that source is required as well).
Permission free of charge on this occasion does not prejudice any rights we might have to charge for reproduction of our copyrighted material in the future.
Altering/Modifying Material: Not Permitted
You may not alter or modify the material in any manner. Abbreviations, additions, deletions and/or any other alterations shall be made only with prior written authorization of the author(s) and/or Springer Science + Business Media. (Please contact Springer at (permissions.dordrecht@springer.com or permissions.heidelberg@springer.com)
Reservation of Rights
Springer Science + Business Media reserves all rights not specifically granted in the combination of (i) the license details provided by you and accepted in the course of this licensing transaction, (ii) these terms and conditions and (iii) CCC's Billing and Payment terms and conditions.
Copyright Notice:Disclaimer
You must include the following copyright and permission notice in connection with any reproduction of the licensed material: "Springer and the original publisher /journal title, volume, year of publication, page, chapter/article title, name(s) of author(s), figure number(s), original copyright notice) is given to the publication in which the material was originally published, by adding; with kind permission from Springer Science and Business Media"
Warranties: None
Example 1: Springer Science + Business Media makes no representations or warranties with respect to the licensed material.
Example 2: Springer Science + Business Media makes no representations or warranties with respect to the licensed material and adopts on its own behalf the limitations and disclaimers established by CCC on its behalf in its Billing and Payment terms and conditions for this licensing transaction.
Indemnity
You hereby indemnify and agree to hold harmless Springer Science + Business Media and CCC, and their respective officers, directors, employees and agents, from and against any and all claims arising out of your use of the licensed material other than as specifically authorized pursuant to this license.
No Transfer of License
This license is personal to you and may not be sublicensed, assigned, or transferred by you to any other person without Springer Science + Business Media's written permission.
No Amendment Except in Writing
This license may not be amended except in a writing signed by both parties (or, in the case of Springer Science + Business Media, by CCC on Springer Science + Business Media's behalf).
Objection to Contrary Terms
Springer Science + Business Media hereby objects to any terms contained in any purchase order, acknowledgment, check endorsement or other writing prepared by you, which terms are inconsistent with these terms and conditions or CCC's Billing and Payment terms and conditions. These terms and conditions, together with CCC's Billing and Payment terms and conditions (which are incorporated herein), comprise the entire agreement between you and Springer Science + Business Media (and CCC) concerning this licensing transaction. In the event of any conflict between your obligations established by these terms and conditions and those established by CCC's Billing and Payment terms and conditions, these terms and conditions shall control.
Jurisdiction
All disputes that may arise in connection with this present License, or the breach thereof, shall be settled exclusively by arbitration, to be held in The Netherlands, in accordance with Dutch law, and to be conducted under the Rules of the 'Netherlands Arbitrage Instituut' (Netherlands Institute of Arbitration).***OR:***
All disputes that may arise in connection with this present License, or the breach thereof, shall be settled exclusively by arbitration, to be held in the Federal Republic of Germany, in accordance with German law.
Other terms and conditions:
v1.3

If you would like to pay for this license now, please remit this license along with your payment made payable to "COPYRIGHT CLEARANCE CENTER" otherwise you will be invoiced within 48 hours of the license date. Payment should be in the form of a check or money order referencing your account number and this invoice number RLNK500751446.
Once you receive your invoice for this order, you may pay your invoice by credit card. Please follow instructions

provided at that time.

Make Payment To:
Copyright Clearance Center
Dept 001
P.O. Box 843006
Boston, MA 02284-3006

For suggestions or comments regarding this order, contact RightsLink Customer Support: customercare@copyright.com or +1-877-622-5543 (toll free in the US) or +1-978-646-2777.

Gratis licenses (referencing $0 in the Total field) are free. Please retain this printable license for your reference. No payment is required.

SPRINGER LICENSE
TERMS AND CONDITIONS

May 18, 2012

This is a License Agreement between Daniel A Orozco ("You") and Springer ("Springer") provided by Copyright Clearance Center ("CCC"). The license consists of your order details, the terms and conditions provided by Springer, and the payment terms and conditions.

All payments must be made in full to CCC. For payment instructions, please see information listed at the bottom of this form.

License Number	2878990333469
License date	Mar 30, 2012
Licensed content publisher	Springer
Licensed content publication	Springer eBook
Licensed content title	Locality Optimization of Stencil Applications Using Data Dependency Graphs
Licensed content author	Daniel Orozco
Licensed content date	Feb 24, 2011
Type of Use	Thesis/Dissertation
Portion	Full text
Number of copies	5000
Author of this Springer article	Yes and you are the sole author of the new work
Order reference number	None
Title of your thesis / dissertation	TIDEFLOW: A DATAFLOW-INSPIRED EXECUTION MODEL FOR HPC PROGRAMS
Expected completion date	May 2012
Estimated size(pages)	160
Total	**0.00 USD**
Terms and Conditions	

Introduction

The publisher for this copyrighted material is Springer Science + Business Media. By clicking "accept" in connection with completing this licensing transaction, you agree that the following terms and conditions apply to this transaction (along with the Billing and Payment terms and conditions established by Copyright Clearance Center, Inc. ("CCC"), at the time that you opened your Rightslink account and that are available at any time at http://myaccount.copyright.com).

Limited License

With reference to your request to reprint in your thesis material on which Springer Science and Business Media control the copyright, permission is granted, free of charge, for the use indicated in your enquiry.

Licenses are for one-time use only with a maximum distribution equal to the number that you identified in the licensing process.

This License includes use in an electronic form, provided its password protected or on the university's intranet or repository, including UMI (according to the definition at the Sherpa website: http://www.sherpa.ac.uk/romeo/). For any other electronic use, please contact Springer at (permissions.dordrecht@springer.com or permissions.heidelberg@springer.com).

The material can only be used for the purpose of defending your thesis, and with a maximum of 100 extra copies in paper.

Although Springer holds copyright to the material and is entitled to negotiate on rights, this license is only valid, provided permission is also obtained from the (co) author (address is given with the article/chapter) and provided it concerns original material which does not carry references to other sources (if material in question appears with credit to another source, authorization from that source is required as well).
Permission free of charge on this occasion does not prejudice any rights we might have to charge for reproduction of our copyrighted material in the future.
Altering/Modifying Material: Not Permitted
You may not alter or modify the material in any manner. Abbreviations, additions, deletions and/or any other alterations shall be made only with prior written authorization of the author(s) and/or Springer Science + Business Media. (Please contact Springer at (permissions.dordrecht@springer.com or permissions.heidelberg@springer.com)
Reservation of Rights
Springer Science + Business Media reserves all rights not specifically granted in the combination of (i) the license details provided by you and accepted in the course of this licensing transaction, (ii) these terms and conditions and (iii) CCC's Billing and Payment terms and conditions.
Copyright Notice:Disclaimer
You must include the following copyright and permission notice in connection with any reproduction of the licensed material: "Springer and the original publisher /journal title, volume, year of publication, page, chapter/article title, name(s) of author(s), figure number(s), original copyright notice) is given to the publication in which the material was originally published, by adding; with kind permission from Springer Science and Business Media"
Warranties: None
Example 1: Springer Science + Business Media makes no representations or warranties with respect to the licensed material.
Example 2: Springer Science + Business Media makes no representations or warranties with respect to the licensed material and adopts on its own behalf the limitations and disclaimers established by CCC on its behalf in its Billing and Payment terms and conditions for this licensing transaction.
Indemnity
You hereby indemnify and agree to hold harmless Springer Science + Business Media and CCC, and their respective officers, directors, employees and agents, from and against any and all claims arising out of your use of the licensed material other than as specifically authorized pursuant to this license.
No Transfer of License
This license is personal to you and may not be sublicensed, assigned, or transferred by you to any other person without Springer Science + Business Media's written permission.
No Amendment Except in Writing
This license may not be amended except in a writing signed by both parties (or, in the case of Springer Science + Business Media, by CCC on Springer Science + Business Media's behalf).
Objection to Contrary Terms
Springer Science + Business Media hereby objects to any terms contained in any purchase order, acknowledgment, check endorsement or other writing prepared by you, which terms are inconsistent with these terms and conditions or CCC's Billing and Payment terms and conditions. These terms and conditions, together with CCC's Billing and Payment terms and conditions (which are incorporated herein), comprise the entire agreement between you and Springer Science + Business Media (and CCC) concerning this licensing transaction. In the event of any conflict between your obligations established by these terms and conditions and those established by CCC's Billing and Payment terms and conditions, these terms and conditions shall control.
Jurisdiction
All disputes that may arise in connection with this present License, or the breach thereof, shall be settled exclusively by arbitration, to be held in The Netherlands, in accordance with Dutch law, and to be conducted under the Rules of the 'Netherlands Arbitrage Instituut' (Netherlands Institute of Arbitration).***OR:***
All disputes that may arise in connection with this present License, or the breach thereof, shall be settled exclusively by arbitration, to be held in the Federal Republic of Germany, in accordance with German law.
Other terms and conditions:
v1.3

If you would like to pay for this license now, please remit this license along with your payment made payable to "COPYRIGHT CLEARANCE CENTER" otherwise you will be invoiced within 48 hours of the license date. Payment should be in the form of a check or money order referencing your account number and this invoice number RLNK500751454.
Once you receive your invoice for this order, you may pay your invoice by credit card. Please follow instructions

provided at that time.

Make Payment To:
Copyright Clearance Center
Dept 001
P.O. Box 843006
Boston, MA 02284-3006

For suggestions or comments regarding this order, contact RightsLink Customer Support: customercare@copyright.com or +1-877-622-5543 (toll free in the US) or +1-978-646-2777.

Gratis licenses (referencing $0 in the Total field) are free. Please retain this printable license for your reference. No payment is required.