ABSTRACT

Title of dissertation:	EASY PRAM-BASED HIGH-PERFORMANCE PARALLEL PROGRAMMING
	Fady A. Ghanim, Doctor of Philosophy, 2017
Dissertation directed by:	Professor Rajeev Barua Department of Electrical and Computer Engineering

Parallel machines have become more widely used. Unfortunately parallel programming technologies have advanced at a much slower pace except for regular programs. For irregular programs, this advancement is inhibited by high synchronization costs, non-loop parallelism, non-array data structures, recursively expressed parallelism and parallelism that is too fine-grained to be exploitable.

This work introduced ICE, a new parallel programming language that is easy-to-program, since: (i) ICE is a synchronous, lock-step language; (ii) for a PRAM algorithm its ICE program amounts to directly transcribing it; and (iii) the PRAM algorithmic theory offers unique wealth of parallel algorithms and techniques. This work suggests that ICE be a part of an ecosystem consisting of the XMT architecture, the PRAM algorithmic model, and ICE itself, that together deliver on the twin goal of easy programming and efficient parallelization of irregular programs. The XMT architecture, developed at UMD, can exploit fine-grained parallelism in irregular programs. This work also presents the ICE compiler which translates the ICE language into the multithreaded XMTC language; the significance of this is that multi-threading is a feature shared by practically all current scalable parallel programming languages. As one indication of ease of programming, it was observed a reduction in code size in 11 out of 16 benchmarks vs. XMTC. For these programs, the average reduction in number of lines of code was 35.53% when compared to hand optimized XMTC The remaining 4 benchmarks had the same code size. The ICE compiler achieved comparable run-time to XMTC with a 0.48% average gain for ICE across all benchmarks.

EASY PRAM-BASED HIGH-PERFORMANCE PARALLEL PROGRAMMING

by

Fady Ahmad Abdalrahim Ghanim

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2017

Dedication

﴿ وَقَالُوا الْحَمْدُ لِلَّهِ الَّذِي هَدَانَا لِهَذا وَمَا كُنَّا لِنَهْتَدِيَ لَوْلَا أَنْ هَدَانَا اللَّهُ ﴾- الأعراف ٤٣

Quran 7:43

﴿ نَرْفَعُ دَرَجَاتٍ مَنْ نَشَاءُ وَفَوْقَ كُلِّ ذِي عِلْمٍ عَلِيمٌ ﴾- يوسف ٧٦

Quran 12:76

I dedicate this work to my mother.

Acknowledgments

First and foremost, I start by thanking *God* for all the achievements discussed herein.

Second, I owe my gratitude to all the people who have made this dissertation possible and because of whom my graduate experience has been one that I will cherish forever.

I'd like to thank my advisor, *Professor Rajeev Barua* for giving me an invaluable opportunity to work on challenging and extremely interesting projects over the past years. He has always made himself available for help and advice and there has never been an occasion when I've knocked on his door and he didn't give me time. Even during busy times, he'd always patiently listen to the long-winded -incoherent at times- explanation of my raw ideas, then help me summarize them and advise me on ways to improve and build upon them. It is a pleasure to work with and learn from such an extraordinary individual.

I would also like to thank my co-advisor, Professor Uzi Vishkin. Without his extraordinary theoretical ideas and expertise in parallel algorithms, this dissertation would have been impossible. Thanks are due to Professor Bruce Jacob, Professor Manoj Franklin and Professor Alan Sussman for agreeing to serve on my PhD defense committee and for sparing their invaluable time reviewing the manuscript.

I owe my deepest and undying gratitude to my *mother* who have always stood by me and guided me through my entire life. Her wisdom and her sound advice was a light that pulled me through in darkest times. Ever since my conception she's been carrying me, whether physically or in her heart and thoughts. Words cannot express the amount of gratitude I owe her. Also, I want to thank my *little nephews and nieces* who cried their hearts out when I first left home to do my studies. I want to thank my brothers and the rest of my family for supporting me throughout the entire period.

Also, I'd like to express my gratitude to my friend *Kelly Flanagan*. Kelly has always been there for me at the time of need. While doing this work, I have gone through trying times and personal ordeals. Kelly, with kindness and support, helped me through it all. I am lucky to have such a great person in my life. Kelly's a lifesaver.

I would like to acknowledge financial support from the Fulbright program and for providing me with this opportunity.

It is impossible to remember all, and I apologize to those I've inadvertently left out. Lastly, thank you all!

Contents

List of Tables

List of Figures

List of Abbreviations

CFG	Control Flow Graph
CRCW	Concurrent Read Concurrent Write
DA	Dependence Analysis pass
ERCW	Exclusive Read Concurrent Write
EREW	Exclusive Read Exclusive Write
ICE	Immediate Concurrent Execution
IOS	Independence of Order Semantics
MIV	Multiple Induction variable
MTCU	Master Thread COntrol Unit
PRAM	Parallel Random Access Machine
PS	Prefix-Sum
RDIV	Restricted Double Index Variable
SIV	Single Induction Variable
SPMD	Single Program Multiple Data
SSA	Single Static Assignment
TCU	Thread Control Unit
TID	Thread ID
WD	Work-Depth
XMT	eXplicit Multi-Threading
XMTC	eXplicit Multi-Threading C language
ZIV	Zero Induction Variable

Chapter 1: Introduction

Since 2005, practically all computers have become (multi-core) parallel machines. The field of parallel computing has made tremendous strides in exploiting parallelism for performance. However, it is also increasingly recognized that its trajectory is short of its general-purpose potential.

Parallel machines require partitioning the task at hand into subtasks (threads) to be run concurrently for minimizing: (i) memory accesses beyond local (cache) memories, and (ii) communication and synchronization among subtasks. Other programmers responsibilities include locking, which can be tricky for fine-grained multi-threading needed for scaling, work distribution and scheduling and handling concurrent access to data structures. While parallel programming languages and parallel machines differ on how much of the partitioning is the programmers responsibility, they all expect a significant effort from the programmer for producing an efficient multi-threaded program. Establishing correctness of these programs is yet another challenge, as asynchrony may increase the number of reachable states exponentially.

The theory of general-purpose parallel algorithms assumes an abstract computation model (known as PRAM for parallel random-access machine, or model) that stands in sharp contrast to these hardships; each time step involves a plurality of operations, all operation performed synchronously in unit time and may include access to a large shared memory. This PRAM computation model abstracts away opportunities for using local memories, and minimizing computation or synchronization, locking, work distribution, scheduling and, in fact, any concept of threads. Also, for PRAM practically every problem has a parallel algorithm. This makes it both desirable and much easier to specify PRAM parallel algorithms, and the question that started out this thesis has been: but, at what performance penalty? As explained next, our surprising result is that it is feasible to avoid any performance penalty.

Coupled with prior work, this thesis establishes the following result: (i) it is feasible to get competitive speedups while essentially using PRAM algorithms as-is for programming a parallel computer system; furthermore (ii) these speedups are on par with multi-threaded code optimized to minimize non-local memory accesses, communication and synchronization. Establishing feasibility of using such abstract (and much simpler) PRAM programming whose performance is on par with the best manually optimized programs is a specific new contribution of the current paper.

The prior work of our research group has anticipated the above hardships. To preempt as many of them as we deemed feasible, our starting point for the design of a many-core architecture framework called XMT was the rich theory of parallel algorithms, known as PRAM (for parallel random-access machine or model) developed in the 1980s and early 1990s. XMT made big strides toward overcoming claims by many that it would be impossible in practice to support effectively PRAM algorithms [e.g., [1]]. Its premise (in prior work) has been that it must be the programmer who will produce a multi-threaded program: [2] outlines a programmers workflow for advancing from a PRAM algorithm to an XMT multi-threaded program. Namely, the programmer is still responsible for producing a multi-threaded program with improved locality and reduced communication and synchronization. Hardware support that XMT provides made this effort easier than for commercial machines, which paid off. This workflow allowed better speedups and demonstrated easier learning of parallel programming. Since our prior work remained wedded to programmer-provided multi-threading, it characterized XMT programming as PRAM-like, as opposed to just PRAM.

This new work is fundamentally different. It shows for the first time that the threading-free synchronous parallel algorithms taught in PRAM textbooks can be used as-is for programming without performance penalty. Namely, it is feasible to reduce multi-threading to a compiler target, altogether freeing the cognition of the programmer from multithreading. In fact, This thesis shows that the programmer can essentially use the pseudo-code used in textbooks for describing synchronous parallel algorithm as-is; this elevates XMT from supporting PRAM-like programs to supporting PRAM programs. Note that the new result is exceeding our research groups expectations at the beginning of the XMT project: it was expected that the programmer will need to make an extra effort for explicating PRAM parallelism as multi-threaded parallelism; indeed, the name of XMT, explicit multi-threading, reflects the original expectation. As can be seen from the example, XMT gets only part of the way to fine-grained multi-threading, but not to lock-step PRAM programming.

ICE allows the same intuitive abstraction that made it easy to reason about and program in serial. Namely, any instruction available for execution can execute immediately. In serial, a program provides the instructions to be executed in the next time step. This made serial programs behave as rudimentary inductive steps from the start of program to its final result. Similarly, ICE describes time-steps of serial or concurrent parallel instructions that execute immediately each time-step (inductively), while falling back to serial execution for the serial portion of the code. In unifying serial and parallel code, ICE can be thought of as the natural extension of the serial model.

1.1 Contributions

In this work we make the following contributions:

(1) A new programming language called ICE that allows the programmer to express the ICE abstraction easily and directly.

(2) To enable this much higher-level programming language, this thesis introduces a new compiler component that automatically translates the ICE program into an efficient XMTC program.

(3) The ICE compiler produces XMTC code that achieves comparable performance to hand-optimized XMTC programs for a given PRAM algorithm, while requiring much less effort than the hand-written optimized XMTC program to write and implement.

(4) Extended the compiler to translate nested parallel programs using the ICE language into hand-optimized XMTC code

The significance of these contributions is:

(1) The work in this thesis enables programmers to write lock-step PRAM algorithms as programs, implement them 'as-is', and execute them over threaded machines without sacrificing performance

(2) ICE requires programmers only to specify the parallelism available in an algorithm. This reduces the effort required to write parallel code significantly, and makes

it much easier for programmers to learn how to write parallel programs.

(3) The ICE compiler is the first compiler that translates from a language based on the lock-step execution model, into a programming language that follows the threaded execution model.

Chapter 2: Background

2.1 Overview

The ICE language is intended to make programming in parallel easier by allowing programmers to write parallel programs based on PRAM algorithms. The ICE language compiler translates programs written in ICE into the XMTC high level language, and is executed over the XMT processor. ICE helps programmers by saving them the effort in converting their programs from PRAM algorithms into high performance XMTC programs. This chapter will go over all the background necessary for understanding the basis of ICE and appreciating the accomplishment achieved in this thesis.

2.2 The PRAM Algorithmic Models

Since ICE is grounded in the Parallel Random Access Machine (PRAM) model, understanding PRAM is crucial to the understanding of ICE. Developed in the 1980s and early 1990s, PRAM is the parallel equivalent of the standard random access machine model used for serial algorithmic theory. PRAM is used by algorithms designers to model the algorithmic performance of parallel algorithms.

PRAM is intended to abstract shared memory machines; PRAM assumes p parallel processors each of which have symmetric access time to shared memory, and has own private memory. There is no limit on the number of processors or the amount of shared memory in the system. Each unit time, a processor can read, write from shared memory or complete a computation simultaneously with other processors as shown in figure 2.1. This results in conflicts while accessing same shared memory locations, for that reason certain rules have been established to resolve these conflicts. Some of the most common rules are:

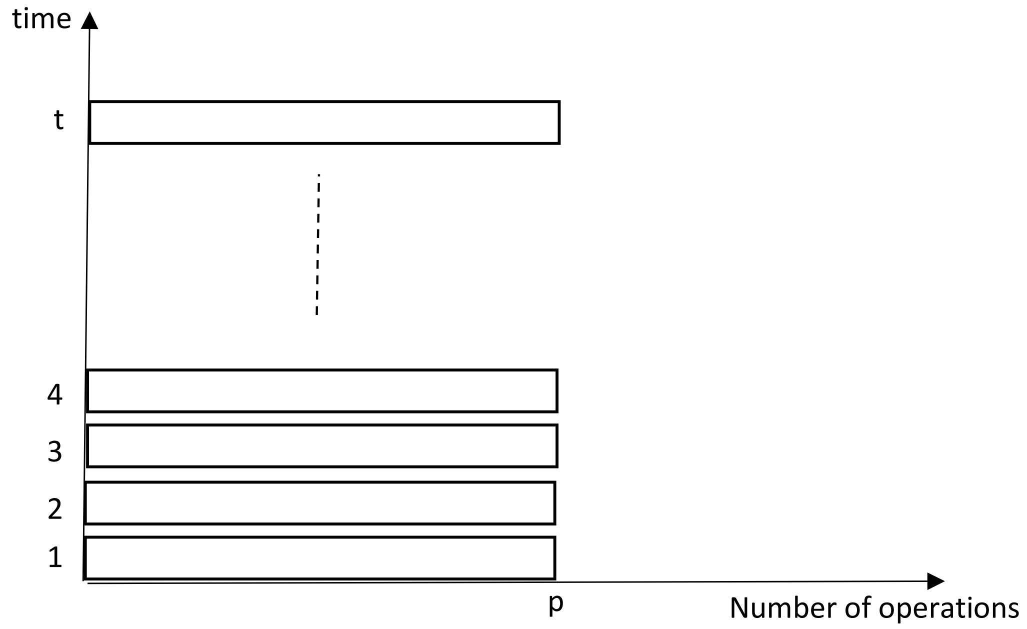


Figure 2.1: *Representation of the standard PRAM mode. Only p operation are executed at each t time step*

- **Exclusive-Read Exclusive-Write (EREW)** Only one processor can access a memory location at unit time, for both reads and writes.

- **Concurrent read exclusive-write (CREW)** processors can read same memory location simultaneously at unit time, however only one processor can write to it.

- **Concurrent read Concurrent write (CRCW)** Processors can read and write simultaneously at unit time. Reads always happen before writes. There are multiple rules for determining which write to memory succeeds. Some of these rules are:

(i) *Priority CRCW*: The processor with the smallest ID wins and its write is the one that succeeds.

(ii) *Common CRCW*: Does not allow concurrent writes unless all processors are attempting to write same value to same memory location.

(iii) *Arbitrary CRCW*: Any processor among all the processors trying to write to the same memory location succeed. ICE follows this is the convention.

The PRAM model presents algorithms as a sequence of time units, each of which contains p operations being executed as one instruction per one processor. As such PRAM only allows p operations to be executed concurrently, as shown in figure 2.1. This presentation is called the standard PRAM mode. To express operations that are performed in parallel, the `pardo` (parallel do) programming construct is used.

```
for $P_i, 1 <= i <= n$ pardo
    perform some operation
```

The standard PRAM model has a few disadvantages especially when compared to other forms of presenting PRAM algorithms:

(i) It does not reveal how changing the number of processing units will effect the speed of algorithm execution.

(ii) Writing a parallel algorithm requires specifying a level of detail that might be unnecessary, due to that algorithms are based on the number of processors available on the platform to be used.

2.3 The Work-Depth Model for PRAM Algorithms

The Work-Depth (WD) model is an alternative presentation model for PRAM algorithms. In this model, algorithms are represented as a sequence of time units where each unit contains a group of operation to be executed concurrently as shown in figure 2.2. The main difference between this model and the standard PRAM model, is that the number of concurrent operation to be executed at each time step in WD model is not bound by the number of available processing units as in the standard PRAM model. Rules for resolving conflicts such as arbitrary, priority and common CRCW are defined in WD model similar to their definition in standard PRAM.

Performance of a parallel algorithm of size n that is given in WD mode is measured in terms of worst case running time $T(n)$ or total number of operations or work $W(n)$.

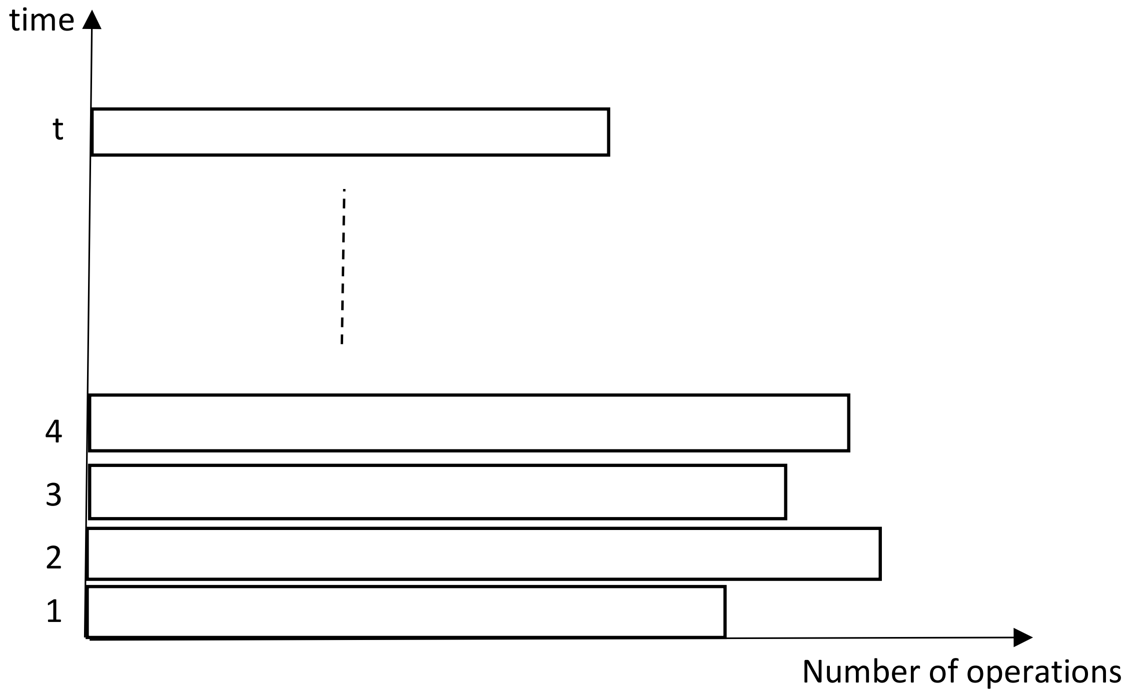


Figure 2.2: *Representation of the Work-Depth mode. Execute as many operation as needed at each t time step*

2.4 The many-core XMT Architecture

A brief review of some basic concepts of the eXplicitMulti-Threading (XMT) on chip general-purpose computer architecture [3, 4, 5] follows.

The XMT architecture was designed to capitalize on the huge on-chip resources becoming available as a result of modern fabrication technologies. The primary goal of XMT has been to take advantage of parallelism to improve the performance of single-tasks. Since taking advantage of the huge body of knowledge available within PRAM is the goal behind XMT, it was designed with PRAM algorithms in mind.

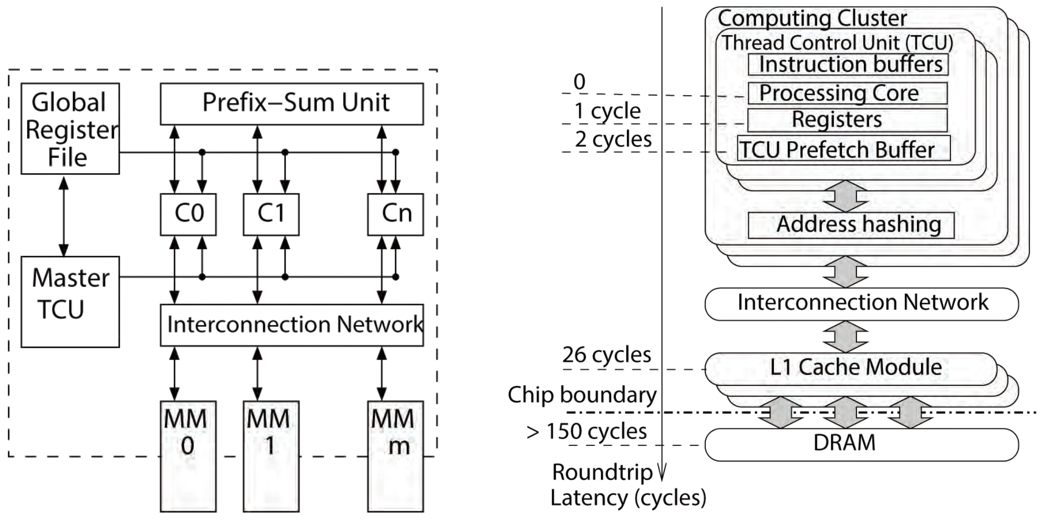
The XMT framework uses an arbitrary CRCW (concurrent read concurrent write) SPMD (single program multiple data) programming model. An arbitrary number of virtual threads, initiated by a `spawn` instruction and terminated by a `join` instruction, share the same code [6]. The arbitrary CRCW aspect dictates that concurrent writes to the same memory location result in an arbitrary one committing. No assumption needs to be made beforehand about which will succeed. An algorithm designed with this property in mind permits each thread to progress at its own speed from its initiating `spawn` to its terminating, `join`, without ever having to wait for other threads; that is, it exhibits "independence of order semantics" (IOS) [5, 6], such that no thread busy-waits for another thread. See Figure 2.4(b).

The **XMT processor**, shown in Figure 2.3a, implements the above programming model efficiently. It includes a master thread control unit (MTCU), processing clusters (C0...Cn) each comprising several thread control units (TCUs), a high-bandwidth low-latency interconnection network (an essential component presented in [7, 8]) between clusters and memory modules, a global register file (GRF), a prefix-sum unit explained below, and memory modules (MMs) each comprising on-chip cache and off-chip memory, with several MMs possibly sharing a memory controller. The MTCU has a standard private data cache, used only in serial mode, and a standard instruction cache, while sharing the MMs with all the TCUs.

Since the **prefix-sum (ps) operation** described above is a central component of XMT, it must be executed very efficiently. The hardware implementation of the prefix-sum unit [6, 9] can accept binary input from multiple TCUs and the execution time does not depend on the number of TCUs that are sending requests to it. It enables constant time, low-overhead coordination between tasks, a key requirement for implementing efficient

fine-grained parallelism. As such, It also can be used for implementing efficient and scalable inter-thread synchronization, by arbitrating an ordering between the threads.

The XMT programming model allows programmers to specify an arbitrary degree of parallelism in their code. Clearly, real hardware has finite execution resources, so in general all threads cannot execute simultaneously. A **hardware scheduler**, which extends the stored-program-plus-program-counter apparatus at the MTCU to the TCUs [9], allocates the individual virtual threads to the physical thread control units (TCU). It relies heavily on hardware support and the prefix-sum unit, which allow for a very efficient implementation. Before the parallel execution starts, two global registers (grLO and grHI) are initialized with the thread IDs of the first and last thread. Then the spawn instruction signals the beginning of the parallel execution, which wakes up the TCUs in a way that each TCU knows immediately its thread ID (TID) and makes the MTCU broadcast the parallel code to the TCUs. When a TCU completes a thread and is ready to execute another thread, it performs a ps operation on grLO with an increment of 1 to get its new thread ID (TID). The hardware combines these ps operations and all requesting TCUs receive their TIDs in a few cycles. Then each TCU executes the powerful `chkid` instruction that compares its TID against grHI: if $TID \leq grHI$ the TID is valid and the TCU proceeds to execute the thread; otherwise it blocks until either the TID becomes valid (grHI got incremented), or all TCUs are blocked signaling the end of the parallel execution. The end of the parallel section is marked by the join instruction.



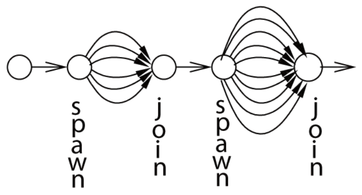
(a) **(b)**

Figure 2.3: XMT hardware. ***(a)*** *Block diagram.* ***(b)*** *Memory Hierarchy in parallel mode. The left side of (b) shows the estimated latency to each memory hierarchy level from the processing core for a 1024 TCU configuration (64 clusters × 16 TCUs). Some elements are omitted for simplicity, such as the Master TCU, which operates in serial mode, the global register file and the prefix-sum unit.*

Figure 2.3b gives an overview of the **XMT memory hierarchy** while operating in parallel mode. The XMT TCUs/clusters has no private caches, since scalable cache coherence protocols are very complicated for hardware implementation and inefficient for certain types of memory access patterns, typically for fine-grained parallelism. For the fine-grained parallelism, the cache coherent private cache is also not efficient in terms of power, due to the large granularity of the data movement between caches, extra cache coherence message exchange and complicated hardware. The downside of our

```
int A[N],B[N], base=0;
spawn(0,N-1){
 int inc=1;
 if (A[$]!=0){
  ps(inc,base);
  B[inc]=A[$];
 } join
}
```

(a)



(b)

Figure 2.4: *XMT Programming.* ***(a)*** *Array Compaction example. Array A's non-zero elements are copied into B. The order is not necessarily preserved. After executing* `ps(inc,base)`*, the* `base` *variable is increased by* `inc` *and the* `inc` *variable gets the original value of* `base`*, as an* **atomic** *operation.* ***(b)*** *The XMT execution model: switching between serial and parallel modes.*

approach is the relative long latency in memory accesses that require round trip to shared parallel cache through an interconnection network. Several techniques have been designed to reduce this latency or overlap with computation, most notably prefetching customized for XMT [10].

A first commitment to silicon of XMT is reported in [3, 11]: a 64-processor, 75MHz computer prototype based on FPGA technology was built at the University of Maryland (UMD)[1]. This milestone contributes towards advancing the perception of PRAM implementability from impossible to available.

[1]Following an international naming contest with close to 6000 submissions, the name Paraleap was given to the prototype.

2.5 The XMTC language

To deepen the understanding of the XMT architecture, an examination of how it is programmed is required. Architectures that support parallel execution has programming frameworks that encourage the programmer to express all the parallelism available in the applications, and XMT is no different. In XMT, a scheduler allocates parallel threads to the physical thread control units, and the execution consists of alternating sequences of serial and parallel code. To allow such paradigm, XMT uses a programming language, XMTC, which was designed to provide programmers with low level operations supported by the hardware, as well as easy mapping of the structures of the PRAM algorithms.

The **XMTC high-level language** is a modest extension of standard C language detailed in [12]. Figure 2.5 shows the general syntax of the XMTC language, while figure 2.4(a) presents an example of XMTC code. The XMTC language extends the C programming language by adding few keywords to allow programmers to write parallel code and access the XMT processor's different features. The most notable are

(i) **The** `spawn` **(*low*,*high*) statement**. Accepts the number of threads to create as its parameter, as a `spawn` creates $(high-low+1)$ threads. Similar to the XMT spawn instruction, it specifies a code region that is to be executed in parallel by the created threads, and also serves as a directive to XMT to "spawn" the threads. The number of threads created is independent from the number of TCUs available in an XMT processor, often exceeding it significantly. The threads are usually short and the execution switches frequently between serial and parallel modes, as pictured in Figure 2.4(b). The threads terminate at an implicit `join` at the closing bracket of the `spawn` block. All tasks must complete before proceeding beyond the spawn block.

(ii) **Thread-id** `$`. is a reserved identifier inside the parallel region, and evaluates to a thread's unique ID, which allows the SPMD programming style of XMTC. Uses the $ to designate thread ID, which takes integer values within the range $low \leq \$ \leq high$

(iii) **Prefix-sum statements** `ps/psm` $(base, increment)$. Defines an atomic prefix-sum operation similar in function to the atomic Fetch-and-Add of the NYU Ultracomputer [13]. It gives programmers access to XMT's powerful prefix sum unit. The way prefix sum operates is by taking a $base$ and an $increment$ as parameters, and value of $increment$ is atomically added to $base$, and the original value of $base$ is returned in variable used for $increment$. The $increment$ has to be a thread private variable that is usually allocated in a TCU's local register. There are two versions of prefix sum available.

- `ps` $(base, increment)$ which takes only global XMT registers as the $base$ parameters, and the $increment$ variable can only have a value of either 1 or 0. Uses XMT's prefix-sum hardware, which combines multiple concurrent requests and execute all fo them in constant time.
- `psm` $(base, increment)$ It takes memory locations as the $base$ parameter, and the $increment$ value can be any integer value. More expensive than the `ps` variant, since it requires a round trip to memory. Multiple `psm` requests arriving simultaneously to cache will be queued.

```
⋮
serial code
shared variables declaration
⋮
spawn (low; high) {
   ⋮
   private variables declaration
   threaded parallel code
   ⋮
} //implicit join
```

Figure 2.5: *XMTC language syntax.* `low` *and* `high` *represent the IDs of the first and last threads. The* `join` *is implicit in the closing bracket of the* `spawn` *block. variable declared outside the spawn block are shared while variables declared inside are thread private.*

The XMT language follows a fork-join model for the creation and termination of threads. It also follows a convention where shared variables are declared outside the `spawn` block, while thread private variables are declared inside the `spawn` block.

To summarize how parallel programming in XMT/XMTC works; a parallel region is delineated by `spawn` and `join` statements, as shown figure 2.4(a). Every thread executing the parallel code is assigned a unique thread ID, designated $. The threads proceed with independent control, synchronizing at the implicit `join` which terminates the threads. The `spawn` statement takes as parameters the IDs of the first and last thread to create. Synchronization is achieved through the prefix-sum (ps)and join commands.

Figure 2.4(a) demonstrates XMTC's power by showing how it can be used to assign a unique index in array B when compacting an array A. The order is not necessarily preserved. the ps operation is used to acquire the next available location in array B, where the non-zero elements of array A will be stored. This example exhibits how each thread progresses at its own pace due to XMT's *Independence of Semantics* IOS property which is accomplished by using the ps operation to obtain next available location in array B. Since ps provides answers in constant time, threads can execute at their own pace without having to wait for one another.

Nested parallelism in XMTC. The XMTC Language allows programmers to nest spawn regions to create nested parallel code. However, when writing nested parallel code using the XMTC language, attention must be paid to how thread IDs are handled in this situation. The XMTC language has only one way to access a thread's ID (*i.e.*, \$), and has no variation of it for situations when nested is involved. In those situations the \$ will act as an identifier for the thread

IDs of the innermost `spawn` block. This can be handled by an XMTC programmer by manually creating a local variable to store the thread ID of a `spawn` block before starting a new `spawn` block nested inside of it.

2.6 The Programmer Workflow for Writing PRAM-Based Programs

The ICE language provides programmers with savings in translating their programs from a PRAM algorithm, to an efficient XMT parallel program. The steps programmers should take to translate an algorithm into a parallel program is known as *The Programmer Workflow*. This methodology links a PRAM algorithm to the XMT platform toolchain (*i.e.*, compiler + hardware implementation). A discussion of the effort involved in this translation is crucial for understanding an important benefit of ICE.

The XMT platform provides programmers with a workflow for deriving efficient programs from PRAM algorithms. It also allows them to reason about these algorithm's correctness and execution time [14]. This programmer workflow provides a ”recipe” to programmers in converting PRAM algorithms to high-performance XMTC programs. It also provides steps programmers can take to incrementally optimize the pro-

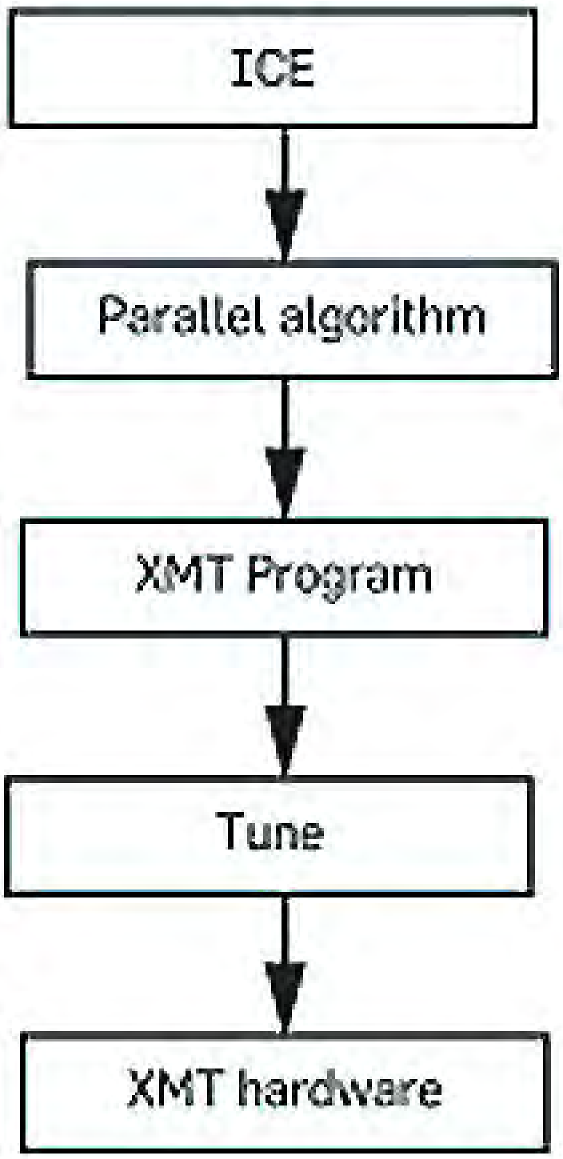


Figure 2.6: *The program flow for translating PRAM algorithm into XMT algorithm.*

grams' performance without having to redesign the original algorithm, which allows them to avoid many of the parallel programming pitfalls.

The steps involved in the programmer workflow for developing an XMT implementation for a PRAM algorithm are as follows.

- Starting from a specific problem, a design stage for an algorithm will produce a sequence of steps each has a set of concurrent operation that should execute in parallel, forming a high level WD description of the parallel algorithm of interest.
- This draft is further refined and optimized for work and depth to form a sequence of rounds each consisting of concurrent operations. It also specifies the steps required to advance in a given step. These rounds form a formal work-depth description of the algorithm.
- The programmer writes their program by translating the WD description into an SPMD program using the XMTC programming language, and fine tunes the program for best performance.
- The program is then compiled into an XMT executable binary optimized for the best performance

Figure 2.6 gives an overview of the discussed steps involved in the programmer workflow. The figure starts from a PRAM algorithm which programmers will use to write a high-level work-depth description of the algorithm. Then programmers translate the work-depth model into its equivalent XMTC code, and then it is further fine tuned for best performance. This XMTC code is then compiled using the XMTC compiler and executed using the XMT hardware.

2.7 Advantages of the XMT Platform

2.7.1 The Performance of the XMT Platform

ICE uses the XMT platform due to XMT's ability at exploiting parallelism in irregular programs which is a result of designing XMT with PRAM algorithms in mind. XMT retains good performance for serial and regular parallel programs as well. Below is a list of experiments that show XMT's performance as compared to other commodity architectures. All XMT's speedups listed below were achieved over the best serial implementation on the state-of-the-art vendor's platform; hence they represent real improvements in processing time.

- **Graph Connectivity** 1024-core XMT processor achieves a speedup of 99.8X, while the NVidia GTX480 had a speedup of 27.1X for graph connectivity [15].

- **Graph Biconnectivity** 1024-core XMT achieves speedups up to 33X, while GPU/CPU hybrid achieved only a 4X speedup [15].

- **Graph Triconnectivity** 1024-core XMT got a speedup of 129X against serial on a core i7 920 processor [16].

- **Finding maximum flow** The best speed up for this algorithm on a hybrid NVedia Fermi GPU/CPU was 2.5X [17]. In contrast, a speedup of 108X was attained on a 1000-core XMT that uses the same silicon area as the GPU [18].

- **Burrows-Wheeler transform - BZIP2** XMT reaches up to 13X/25X Speedup for de/compression [19]. In comparison, there was a slowdown of 2.8 for compression and a speedup of 1.1 for decompression on GPU.

- **2-D FFT** XMT reached 20.4X speed up, whereas a 16-core AMD opteron got less than 4X [20].

- **Gate-level Simulation Benchmark Suite** XMT obtained 100X speedups versus serial for [21].

2.7.2 The XMT Teachability and Ease of Use

Programmer's productivity and ease of programming are central focal points for the XMT platform. A Platform that is easy to learn is a necessary condition for it to be an easy to program platform, thus, demonstrating how XMT is easy to teach and learn has been one of the central objectives of the project.

- Since 2007, more than 300 high-school students have been taught to program the XMT platform, including two exemplary cases: Montgomery Blair High School, Silver Spring MD, and Thomas Jefferson High School for Science and Technology, Alexandria VA. At Thomas Jefferson, Torbert [22] has incorporated XMT into their classes, and advocates using it in the education of Computer Science. Tolbert reports that, when compared to MPI, The XMT platform spurred students creativity to invent their own personal programs to solve a variety of problem instead of chasing the same canonical solution as was the case with MPI.

- In a study supported by DARPA HPCS program, it was shown that the development time of XMTC is about half that of MPI, under circumstances favoring MPI [23].

- A joint experiment between the University of Illinois and the University of Maryland compared programming in both OpenMP and XMTC [24]. This study included 42 students who were asked to achieve speedups for BFS using OpenMP over an 8-core SMP, and using the XMT platform. The students could not achieve a speed up higher than 1 when they used OpenMP. However, they were able to obtain 7x to 25x speedups on the 64-TCU XMT FPGA. In addition, the PRAM/XMT part of the joint course were able to write more advanced algorithms as compared to the OpenMP part.

Chapter 3: The ICE Programming Language

3.1 Overview

This thesis presents methods to write synchronous parallel code based on PRAM algorithms 'as-is' and obtain comparable performance when executed. This chapter will go over the details of the ICE programming language and its different features. The chapter will discuss the ICE syntax and the lock-step model that ICE follows, as well as the advantages that ICE gains by following that model.

3.2 The ICE language

The Immediate Concurrent Execution (ICE) language, a modest extension of the C programming language, is a parallel programming language that is intended for ease of programming and development of high-performance parallel programs based on the PRAM algorithmic model. ICE is based on the WD-model of PRAM that describes a sequence of time steps, each containing multiple concurrent operations. ICE enables that through following the lock-step execution model. ICE requires the programmer to simply express all available parallelism and nothing else. ICE is unique in being the only language that can take the PRAM lock-step and translates it to a threaded program.

A lock-step parallel programming model is one where each statement in a parallel loop has all its iterations appear to execute exactly in the same cycle to the programmer. This appearance is enforced by the ICE compiler (discussed in 4), usually without enforcing same-cycle execution in hardware. Figure 2.2 shows the lock-step nature of ICE. Lock-step execution contrasts with the threaded execution followed by virtually all parallel programming languages, where all the functional units execute independent threads of control which proceed at their own unpredictable pace, and where synchronization with other threads only happens if the program explicitly requests it. Lock-step execution is common in hardware – for example in VLIWs or GPUs. However we are not aware of a higher level programming language meant for general purpose, irregular programming that uses lock-step execution.

The advantages of lock-step programming are that many explicit synchronizations become unnecessary and in-place concurrent updates to aggregate data structures becomes possible without introducing non-determinism (these will be explained in section 3.5)

ICE provides a shortcut to programmers following the programmer workflow discussed in 2.6, which started from an ICE abstraction of an algorithm and advanced to threaded implementation. ICE allows programmers immediately write programs using the ICE abstraction of an algorithm 'as-is'.

The intention behind designing ICE is not advocating a lock-step model of hardware execution. Indeed lock-step parallel hardware has mostly been explored in the past in the context of SIMD machines, which have not met wide success. SIMD machines can only exploit vector and dense-matrix parallel codes well, but so can MIMD machines. This work is primarily motivated by parallelism in irregular programs in graph-traversal and divide-and-conquer algorithms, which do not parallelize well on any existing parallel machine (either traditional MIMD multi-cores or SIMD). Hence

It does not further consider translation of ICE to either of those machine types; instead we focus on translating ICE to XMT code.

ICE is not meant to replace any of the current programming models either, it is meant to work along side them instead. ICE is generally better suited for applications based on PRAM algorithms when compared to threaded model. In contrast, the threaded model is likely to be better suited to task parallel applications. ICE is orthogonal to the threaded model, and can be included along side a language like XMTC, thus enabling programmers to choose and mix either lock-step (ICE) or threaded (XMTC) models, whichever is more natural for each parallel section. Hence the same program can have some parallel loops implemented in XMTC, and others in ICE.

3.3 The Syntax of the ICE Language

The ICE language enable programmers to write parallel programs using lock-step execution model. The ICE language extends the C programming language by introducing new keyword, `pardo` , to allow programmers to specify where parallelism lies in the algorithm they intend to implement.

`pardo` is inspired by the construct pardo , short for PARallel DO, used in many PRAM texts [25, 26, 27]. `pardo` creates a number of concurrent virtual lock-step parallel contexts.

Figure 3.1 provides the ICE syntax, and the generic structure of the new `pardo` keyword. The `pardo` keyword requires programmers to specify four parameters; a parallel context control variable used to refer to a parallel context ID, the ID of the first parallel context `low`, the ID of the last parallel context`high`, and the stride `step`. A `pardo` creates $(high - low)/step + 1$ parallel contexts that execute the instructions specified inside the `pardo` region based on the lock-step model. Concurrent writes performed by multiple parallel contexts to the same memory location are handled using arbitrary CRCW. ICE follow the convention of having parallel context local variables declared inside the parallel region, while shared variables are declared in serial regions.

The ICE language allows programmers to specify nested parallelism by using the `pardo` keyword from within a `pardo` region. Each parallel context created by the outer `pardo` creates multiple parallel contexts as specified by the inner `pardo` . Lockstep execution extends to the newly created parallel contexts. As such, they advance synchronously with

```
⋮
serial code
shared variables declaration
⋮
pardo (pid = low; high; step) {
    ⋮
    private variables declaration
    lockstep parallel code
    ⋮
}
```

Figure 3.1: *ICE language Syntax.*

all parallel contexts of the same level created by any 'parent' parallel contexts.

Variable locality for nested ICE follows the same principle that was discussed earlier, namely; variables declared inside a `pardo` region are private to each individual context, and variables declared outside a `pardo` region are shared between all the parallel contexts created by that `pardo` . The implication of this is that variables that are private to a context, are shared between all parallel contexts created by that context, not across all contexts of the same level of nesting.

Table 3.1 provides a comparison between the syntax of the lock-stepped `pardo` and the threaded `spawn` . ICE and

Table 3.1: *Comparison of the `pardo` and `spawn` constructs.*

	`pardo` **(lock-step)**	`spawn` **(threaded)**
Syntax	`pardo` (CID=LB;UB;ST)	`spawn` (LB,UB)
Contexts Num. N	$(UB - LB)/ST + 1$	$UB - LB + 1$
First—last IDs	LB — $LB + ST \times N$	LB — UB
Stride	ST	1
MYPID	CID (user defined)	$
Execution Model	Each instruction is executed over all parallel contexts before the next one is initiated.	Instructions within a thread progress at their own pace.
Synchronization	After every Instruction	`join` or `(ps)`

XMTC follow the same convention of how local and shared variables are declared.

3.4 Example Showcasing the ICE Language

To see the features and advantages of the ICE programming language, consider the example in figure 3.2. Fig 3.2(a) shows the problem specification for *pointer jump-*

ing, a well-known and useful task in tree and graph algorithms. The input shown consists of array S containing the specification of a forest of trees, and array W containing an initial weight at each node. The output "flattens" the tree by directly pointing each node to the root of its tree, and computes the sum of weights (or distance) from the node to the parent in the input tree. The example shows a specific assignment of weights which will compute the distance to the root in the output; however, any input assignment of weights can be chosen. The pointer jumping algorithm is widely used, for example in the *disjoint-set* (union-find) data structure for efficiently maintaining sets and supporting set union and find operations. In the case that the input trees degenerate to linked lists, computing the output becomes the well-known *list ranking* problem, a key component of many algorithms, so called because it calculates the position (or rank) of every element in a linked list. List ranking also computes a *prefix sum* operation for any input weights in W, and stores the result in W. List ranking is also used to solve many problems on trees and graphs via an Euler tour technique (Tarjan and Vishkin [28]).

Problem

Given a linked list with n elements, find for every elements its distance from the last element.

Input

- *Array S(1...n): S(i)* contains the index of the successor of element i. The successor of the last element is the element itself.
- W(1...*n*): *W*(*i*) contains the weight of element *i*. Initially *W(i)=0 for the last element in the list and W(i)=1 for all other elements.*

Output

- *S(i) is the index of the last element of the list.*

W(i) is the distance of element i from this last element.

(a) Problem specification

```
void pointer_jump (int S[n], int W[n], int n) {
  pardo (unsigned i = 0; n-1;1) {
     while (S[i] != S[S[i]]) {
         W[i] = W[i] + W[S[i]];
         S[i] = S[S[i]];
     }
  }
}
```

(b) ICE program

```
psBaseReg flag;  // number of threads that require
                    another loop iteration
void pointer_jump (int S[n], int W[n], int n) {
   int W_tmp[n];
   int S_tmp[n];
   do {
      spawn (0, n-1) {
          if (S[$] != S[S[$]]) {
             W_tmp[$] = W[$] + W[S[$]];
             S_tmp[$] = S[S[$]];
          } else {
             W_tmp[$] = W[$];
             S_tmp[$] = S[$];
          }
      }
      flag = 0;
      spawn (0, n-1) {
          if (S_tmp[$] != S_tmp[S_tmp[$]]) {
             int i = 1;
             ps(i, flag);
             W[$] = W_tmp[$] + W_tmp[S_tmp[$]];
             S[$] = S_tmp[S_tmp[$]];
          } else {
             W[$] = W_tmp[$];
             S[$] = S_tmp[$];
          }
      }
   } while (flag != 0);
}
```

(c) XMTC program

```
void pointer_jump (int S[n], int W[n], int n) {
   int W_tmp[n];
   int S_tmp[n];
   int *W_rd = W, *W_wt = W_tmp;
   int *S_rd = S, *S_wt = S_tmp;
   int *tmp_ptr;
   int crs_size = n/P + ((n%P) > 0);
   int flag = 1;
   while (flag != 0) {
      flag = 0;
      #pragma omp parallel num_threads(P) {
         #pragma omp parallel for reduction(+,flag) schedule(static, crs_size)
            for (int i = 0; i < n; i++) {
               if (S[i] != S[S[i]]) {
                  int x = 1;
                  flag += x;
                  W_wt [i] = W_rd[i] + W_rd[S_rd[i]];
                  S_wt [i] = S_rd[S_rd[i]];
               } else {
                  W_wt[i] = W_rd[i];
                  S_wt[i] = S_rd[i];
               }
            }
      }
      tmp_ptr = W_rd;      W_rd = W_wt;         W_wt = tmp_ptr;
      tmp_ptr = S_rd;      S_rd = S_wt;         S_wt = tmp_ptr;
}
```

(d) OpenMP Program

Figure 3.2: *Pointer jumping example showing simplicity of ICE code. (a) provides a description of the pointer jumping problem. This problem is then solved using ICE programming language (b), XMTC programming language (c), and OpenMP (d)*

Figure 3.2(b) shows the ICE code to solve the pointer jumping problem defined in figure 3.2(a). From this figure it can be seen how the ICE programming language is loosely based on the PRAM algorithmic model. It has serial regions, and parallel regions inside `pardo` constructs. As can be seen in the figure, the ICE code for pointer jumping is indeed very simple. The in-place updates of S and W are possible because of the lock-step nature of execution, where, for example, the right-hand side (RHS) of the first statement in the loop $(W(i) + W(S(i)))$ is read and computed on all the parallel units before the LHS writes the new value of W(i).

Although the code in figure 3.2 uses arrays to implement trees, pointer jumping can be implemented in ICE with structures and pointers just as easily. The code will be conceptually similar.

Figure 3.2(c) shows the XMTC code to solve the same pointer jumping problem. XMTC, as discussed in 2.5, has parallel constructs such as `spawn` (x,y) which starts $y - x + 1$ parallel threads numbered x to y. Since the parallel threads are independent, they proceed at their own pace, synchronizing only at the implicit join at the end of the spawn block, or at the prefix sum (**ps**) operation shown.

As can be seen in the figure, the XMTC code is longer and much more complicated than the ICE code. *The underlying cause is the unpredictable pace of parallel threads, which prohibits in-place updates*, such as those of arrays S and W. Instead the threaded code must use temporaries S_temp and W_temp. In the first part of the code, the program reads from the original arrays and writes to the temporary arrays; roles are reversed in the second part. The first and second part alternate until the computation is completed. Additional synchronization is needed to count the number of incomplete threads remaining in the $flag$ variable; counting is done using the prefix sum (ps) construct described in section 2.5[1].

Figure 3.2(d) shows an OpenMP code to solve the same pointer jumping problem. The OpenMP code in figure 3.2(d) essentially executes similarly to the XMTC version. However, there are two main differences. 1. The ps operation in XMTC version is replaced by a reduction operation in OpenMP. 2. Unlike the XMTC version, the loop was not unrolled in the OpenMP version. Instead, two sets of pointers were used to alternate the source and destination of copying between the

[1]The ps operation could have been avoided by multiple writes of true to a boolean variable called threads-remaining in the loop, but that would create a hot-spot in memory. The XMT ps operation uses registers, avoiding the hot spot.

original and temporary S and W arrays. It is imporant to understand that implementations in figures 3.2(c) and (d) are fully interchangeable between XMTC and OpenMP. Namely, the implementations will work very similarly regardless of the platform used. However, when implemented on a similar platform, the implementation in figure 3.2(c) will have a slight performance advantage over the implementation in figure 3.2(d), while the later is slightly shorter and easier to write.

The ICE compiler translates lock-step ICE programs into multi-threaded parallel software. Methods for this translation will be discussed in section 4.3. This compiler translates ICE programs to XMTC; thus leveraging XMT's ability to execute irregular programs efficiently.

*The code in figure 3.2(c) gives good speedup on XMT despite the code being very fine grained (*i.e.*, having short parallel sections)*. Pointer jumping on XMT gives a speedup of 50X over the best serial algorithm on an Intel Core i7-920. Despite the many more cores of XMT, these are comparable in area since XMT cores are extremely lightweight. This is despite the parallel version performing *O(n log n)* work while the serial algorithm is *O(n)*, so the parallel code is not work efficient compared to the serial algorithm. Parallel code on traditional multi-cores is unlikely to get any speedup at all because of the high cost barriers, poor load balancing in coarsened versions (where multiple parallel sections are combined into longer-running parallel sections to reduce barrier costs), and the work-inefficiency of the parallel algorithm.

The example in fig 3.2 shows many of the strengths of the ICE programming model, which will be discussed in section 3.5.

3.5 The ICE language Advantages

The ICE language was designed to be easy to use, and to leverage the scientific wealth in PRAM algorithms. The ICE language follows the lock-step model which allows it to have many advantages when compared to the threaded model followed by many other parallel programming languages. These advantages are:

- **Easier translation from PRAM algorithms.** The ICE abstraction has been used as the first stage in the design and analysis of PRAM algorithms. PRAM algorithms readily fit into the ICE programming model whereas extra effort is needed to fit them into a threaded model. This is illustrated by the great difference between figures 3.2(b) and

(c) - manually translating the first to the second can be a significant effort. Thus ICE makes parallel programming easier, fulfilling one of our primary goals.

- **No need to think about synchronization.** Thinking about synchronization is a major contributor to making parallel programming difficult. ICE greatly reduces this problem by assuming a maximum degree of synchronization: there is an implied barrier between every statement in a parallel region. Thus in ICE, synchronization comes "for free" in terms of programmer effort. This is unlike threaded languages, where the programmer must decide when synchronization is needed and when it is not, and explicitly request it when needed. This is illustrated by figure 3.2(c), where the programmer has to decide the location of synchronization at the end of spawn blocks, and the locations of any needed **ps** operations. In ICE the compiler would decide when synchronization is needed when translating to a threaded model, relieving the programmer of that burden.

- **No need to introduce intermediate variables.** The lack of assumed synchronization in threaded programming models often results in the need for intermediate variables to avoid race conditions. For example, the in-place parallel update of a data structure often requires copying to an intermediate data structure to avoid race conditions. The code in figure 3.2(c) has *S_temp, W_temp*, and *flag* as intermediate variables. This duplicating and copying of data structures must be managed by the programmer, increasing his or her burden. In contrast the ICE programming model makes intermediate variables unnecessary in most cases because of its assumed lock-step synchronization, making parallel programming easier (with a convention such as: 'all reads complete before writes'). Of course, such intermediate variables may be re-introduced by the compiler when it translates ICE to a threaded language. In effect, the ICE compiler takes over the management of intermediate variables instead of the programmer.

- **Avoids unintended race conditions.** Threaded programming models allow the programmer to express unintended race conditions. To eliminate them, the programmer must be proactive. In particular, the programmer must know about and rely on consistency models.

  The threaded code in figure 3.3 illustrates race conditions and consistency models. It shows a classic exam-

P1: A = 0;	P2: B = 0;
…	…
A = 1;	B = 1;
L1: if (B == 0) …	L1: if (A == 0) …

Figure 3.3: *Threaded code with race condition*

ple from Hennessy and Patterson textbook (HP:AQA,4th ed,pp 243), where two parallel processes P1 and P2 running on different cores are shown side-by-side. Assume both cores have locations A and B cached with initial value 0. The question here is: is it possible for the `if` statements in both threads to evaluate to true? At first glance it seems impossible. Hypothetically, if writes are seen on remote processors immediately, then it is not possible for both to evaluate to true. However writes are often delayed on real machines; hence real machines use consistency models to define allowed behaviors. For example on machines with sequential consistency both `if` statements cannot evaluate to true. However some computers implement weaker consistency models for efficiency, where the anomalous behavior is allowed. Unfortunately threaded programming models usually expose the consistency model to the programmer, who must understand it well to avoid introducing unintended race conditions.

In contrast the lock-step parallel behavior of the ICE programming model makes it impossible for the programmer to express such unintended race conditions. With ICE, a programmer never needs to consider race conditions or consistency models. Instead the compiler manages race conditions when translating ICE code to threaded code in a platform-specific way; thus relieving the programmer of this burden.

- **No need to think about scheduling or coarsening.** Although declarative (pre-scheduling) threaded models such as XMTC have been proposed, several threaded models in common use such as MPI and *pthreads* are post-scheduling, thus requiring the programmer to manually schedule available parallelism into N threads, where N is the number of hardware contexts available on the target hardware[2] The programmer is also responsible for coarsening if the available parallelism exceeds N. In contrast ICE is a declarative programming model where

[2]The number of hardware contexts is the number of threads that the hardware can actually run at any one instant. This equals the number of cores $\times$ the hyper-threading factor for multi-cores, and equals the number of TCUs on XMT.

the programmer simply expresses all available parallelism without regard to the number of hardware contexts, or the scheduling of the code to those contexts. Scheduling and coarsening is performed automatically by the compiler and/or run-time system. This significantly reduces the burden on the programmer since he or she no longer needs to do scheduling/coarsening, and it also makes the code more portable across XMT computers with different numbers of hardware contexts. This feature is already available in the XMTC compiler, and the ICE compiler takes advantage of that.

- **No need to think about data decomposition or locality.** The uniform-memory access (UMA) design of XMT relies on a high bandwidth low latency interconnection network between TCUs and shared memory. Thus all functional units see the same latency to all regions of memory, except for registers local to their TCUs, and prefetch buffers. This led to a situation (supported by our experimental results) where the XMTC programmer does not need to consider data decomposition or locality as a first-order consideration. The XMTC programmer's workflow [2, 29] instructs to first produce a handwritten ICE-like algorithm with no data decomposition or locality; the programmer is then expected to develop XMTC code in which every `spawn` command comprises its own scoping (cf. Figure 2.4(a)); this scoping allows designation of local variables that, in turn, the XMTC compiler translates into local registers. For ICE, even this designation of local variables will be left to the (ICE) compiler. Since not needing decomposition is inherited from XMT, both ICE and XMTC have this advantage, but not threaded programming models for NUMA machines.

Given the advantages above, ICE represents a significant leap in the ease of programming compared to threaded programming models. In addition, execution on XMT will deliver unmatched speedups for irregular programs written in ICE.

Chapter 4: The ICE Language Compiler

4.1 Overview

In this thesis, an ICE compiler was built to translate ICE programs to threaded XMTC programs. The output XMTC code is compiled, using the existing relatively mature and well-studied XMTC compiler, into an executable XMT binary. This chapter will go over the challenges in producing correct translation, difficulties in optimizing the translation, and the complete structure of the ICE compiler.

4.2 Translating ICE to XMTC

This section will focus on the main challenges and the problems that may arise while building a new compiler that translates from ICE, a language following the lock-step model into XMTC, a threaded language.

4.2.1 Splitting a `pardo` Region into Multiple `spawn` blocks

To translate ICE programs to XMTC programs, the `pardo` region is split into multiple `spawn` regions. Replacing every `pardo` with `spawn` will not work since the former requires lock-step execution, but the latter (regular multi-threading) does not ensure it. We saw this in figure 3.2. Splitting occurs at points where a barrier is required. In XMT there is no way to implement barriers except through using the `join` instruction. A `join` is introduced by terminating a `spawn` region and starting a new one, effectively splitting the `pardo` . This solution ensures that there will be no violation of the data dependencies (true or anti-dependence) between the memory accesses within the `pardo` region. This method's downside is that the parallelism granularity is reduced, but its degree is maintained.

To ensure correctness, the order of reads and writes must be maintained. Thus when translating ICE to XMTC, it is required that a `pardo` be split into multiple `spawn` blocks wherever the `pardo` contains both a read and a write to a data object accessed by at least two different parallel contexts. This ensures that a memory access is completed by all parallel contexts, before any context starts with the next memory access. This splitting is performed by introducing a barrier between the read and the write. Two cases are possible: anti-dependence where a write to a data object are done after a read (e.g. W and S in figure 3.2(b)), and true dependence where a read is performed after a write. Both cases require splitting the `pardo` region into two successive `spawn` regions. However, in the anti-dependence case, we also need to introduce a (compiler-inserted) temporary, to which we perform the writes instead in the first `spawn` region, and copy them back in the second.

4.2.2 Communication of Intermediate Information Among `spawn` Blocks

Splitting `pardo` regions is likely to introduce many challanges for maintaining correct operation of the translated ICE code. Since different `spawn` blocks, preventing correct progression of the data and control flow of the program

4.2.2.1 Handling Data Flow Across `pardo` Region Splits

Splitting `pardo` regions is likely to be a problem to the data flow within a `pardo` block. Processors perform computations by reading source data from memory which is then processed to produce the final results that are stored into memory again. During computations, the intermediate data is performed over, and kept in a processor's registers. Adding barriers between sources' loads and a results' stores prevent correct data flow, due to the inability to communicate a parallel context's intermediate data between one `spawn` block and the next.

This is resolved by 'demoting' the registers and recording their contents onto memory locations. Hence, for each intermediate value that is still being used in later `spawn` blocks, The ICE compiler creates an array of n elements, where n is the number of parallel contexts. Each of these element has the same data type, size, and holds the same value as that of the intermediate value being recorded. Then, each intermediate value is saved onto memory before the split, and retrieved after the split to be used in subsequence `spawn` blocks.

4.2.2.2 Handling Control Flow Across `pardo` Region Splits

Splitting `pardo` regions may cause complications for the program's control flow. There are two cases when this can happen:

(1) When a `pardo` region contains a conditional branch where one of its directions requires a barrier as in figure 4.1.

```
pardo (i = 0; n; 1) {
  if (i < 50) {
    A[i+1] = c[i];
    c[i] = A[i] + 1;
  }
}
```

(a) Ice code

```
char cond[n+1];
spawn(0,n) {
  unsigned i = $;
  cond[i] = i< 50;
  if (i < 50)
    A[i+1] = c[i];
}

spawn(0,n) {
unsigned i = $;
  if (cond[i])
    c[i] = A[i] + 1;
}
```

(b) XMTC translation

Figure 4.1: *(a) A `pardo` with a conditional branch. (b) Its XMTC translation.*

(2) When a `pardo` region contains a serial loop within which a barrier is needed. This causes a problem when expressing the `continue` and `break` statements, and the serial loop's back edge as in figure 3.2(b).

To maintain correctness, a parallel context must preserve its intended control flow, which is not easily possible in these cases since XMT disallows branching between `spawn` blocks. To that end, branch decisions are communicated across splits by recording the branch state for each context into memory, and retrieve it when needed. Hence, for the first case when a branch condition is evaluated as in figure 4.1(b), we record the result to memory (temporary array `cond`) and retrieve it in any later `spawn` that is on either direction of the conditional branch. A similar solution is used for the second case where the serial loop is taken outside the parallel region and is executed by the MTCU, the loop condition becomes a flag indicative of the existence of threads that have not completed execution yet, and the original loop termination condition becomes a normal branch and is treated as in the branch case. An example of this is the `do-while` loop in figure 3.2(b)(c) where the serial loop is taken outside the `spawn` block, the terminating condition now is $(flag! = 0)$ instead of $(S(i) == S(S(i)))$. `flag` is incremented by threads which still have work to do, using the `ps` operation. Furthermore, temporary arrays are used to record when a context executes a `continue` or `break`. Resultant `spawn` blocks from this loop split will check if the context have executed either, and will act accordingly.

4.3 Optimizing The Translated ICE Code

Splitting a `pardo` region into multiple `spawn` blocks can degrade performance due to the overhead of creating and managing more threads, and due to using memory to commu-

```
pardo (int i = 0; n; 1) {
  A[i+1] = c[i]; \\A1
  c[i] = A[i] + 1; \\A2
  B[i-1] = d[i]; \\B1
  d[i] = B[i] + i; \\B2
}
```

(a) Code in ICE

```
spawn(0,n) {
    unsigned i = $;
    A[i+1] = c[i]; \\A1
}

spawn(0,n) {
    unsigned i = $;
    c[i] = A[i] + 1; \\A2
    B[i-1] = d[i]; \\B1
}

spawn(0,n) {
    unsigned i = $;
    d[i] = B[i] + i; \\B2
}
```

(b) Equivalent code in XMTC

```
spawn(0,n) {
    unsigned i = $;
    A[i+1] = c[i]; \\A1
    B[i-1] = d[i]; \\B1
}

spawn(0,n) {
    unsigned i = $;
    c[i] = A[i] + 1; \\A2
    d[i] = B[i] + i; \\B2
}
```

(c) Optimized XMTC

Figure 4.2: *Rescheduling memory accesses. Statement A2 is dependent on statement A1, and statement B2 is dependent on statement B1. statements A are independent from statements B*

nicate information between the different `spawn` blocks which increases the degradation even further. This is exacerbated when the number of splits is high. Hence it is crucial to avoid splitting whenever possible, and to mitigate the effects of the unavoidable splits.

Splitting a `pardo` can be avoided if we can prove that a memory location is exclusively accessed only by a one parallel context. In this case, the splitting becomes unnecessary and a direct conversion from a `pardo` to a `spawn` will be possible. One example of this is when a parallel context with ID 'i' always reads and writes to $A[i]$; hence we know that no two contexts access the same memory location. This means that no race conditions are possible; hence no splitting is needed.

Optimization for anti-dependence case within serial loops in `pardo` When the anti-dependence is within a loop in a `pardo` region (as in figure 3.2 example), we can get better performance by unrolling the `pardo` once, and then transforming the two loops that result so that the first loop updates temporary data structures that are clones of the original data structures, and the second loop does the opposite. An example of this is seen in figure 3.2(c). Thereafter the `pardo` region is split to place the two loops in different `spawn` blocks in the XMTC output. Other elements in the figure such as `ps` operation and `'flag'` will be discussed in detail shortly.

4.3.1 Clustering of Memory Instructions

In an optimization for unavoidable splits, we rearrange memory accesses within a `pardo` into **clusters** to minimize the number of splits needed. Each cluster represents a `spawn` block. These clusters consist of a group of memory accesses that are independent from one another across the different parallel contexts. When a `pardo` region is split into multiple `spawn` blocks, often there are more splits than necessary. We see an example of this in figure 4.2(a), where there is a dependence between statements A1 and A2, and another between B1 and B2, but none exist between the A and B statements. Without optimization we will end up with three spawns after the splitting as in figure 4.2(b). However, by rearranging and grouping independent memory accesses as in figure 4.2(c) and only then doing the splitting, we end up with two spawns. We call this rescheduling scheme **clustering**.

The clustering algorithm is a list scheduling algorithm. Figure 4.3 shows the algorithm used. The compiler builds a dependence graph in which we capture all data (flow or 'loop-carried'[1]) and control dependencies between all the memory accesses. Then we start building one cluster at a time by scheduling all 'ready-to-fire' nodes in the current cluster (lines 28 - 34). A node is 'ready-to-fire' if it satisfies the conditions in the lines (13 - 25). In simple terms, when

[1]Even though the execution order within a `pardo` is different from that of a loop, the term 'loop carried dependence' is being used to refer to the parallel contexts cross dependence between different memory access in the `pardo` block

```
$\boldsymbol{M}$: $\mathit{set\ of\ all\ memory\ accesses}$
$\boldsymbol{CL_i} = \{\boldsymbol{m} \in \boldsymbol{M} : \boldsymbol{m}$ is a member of cluster $\mathbf{i}\}$
$\boldsymbol{NM} = \{\boldsymbol{m} \in \boldsymbol{M} : \boldsymbol{m}$ is not a member of any cluster$\}$

For an $\boldsymbol{m} \in \boldsymbol{NM}$:
$\boldsymbol{L_m} = \{\boldsymbol{m_L} \in \boldsymbol{M} :$ loop carried dependence between $\boldsymbol{m_L}$ $\mathit{and}$ $\boldsymbol{m}\}$
$\boldsymbol{F_m} = \{\boldsymbol{m_F} \in \boldsymbol{M} : \boldsymbol{m}$ is Data flow dependent on $\boldsymbol{m_F}\}$
$\boldsymbol{C_m} = \{\boldsymbol{m_C} \in \boldsymbol{M} : \boldsymbol{m}$ is control dependent on value of $\boldsymbol{m_C}\}$.
$\boldsymbol{LP_m} = \{\boldsymbol{m_{LP}} \in \boldsymbol{M} : \boldsymbol{m}$ exist in a different loop from $\boldsymbol{m_{LP}}\}$
$\boldsymbol{NL_m} = \boldsymbol{L_m} \cap \boldsymbol{NM}$
$\boldsymbol{NF_m} = \boldsymbol{F_m} \cap \boldsymbol{NM}$
$\boldsymbol{NC_m} = \boldsymbol{C_m} \cap \boldsymbol{NM}$
$\boldsymbol{NLP_m} = \boldsymbol{LP_m} \cap \boldsymbol{NM}$

**Define Procedure** ConflictsWith ( $m, CL$ ) :
    **if** $NL_m \neq \Phi$ **then**
        **return** true
    **if** $L_m \cap CL \neq \Phi$ then
        **return** true
    **if** $LP_m \cap CL \neq \Phi$ then
        **return** true
    **for** $m_F \in NF_m$ **do**
        **if** ConflictsWith ($\boldsymbol{m_F}$ , $CL$ ) **then**
            **return** true
    **for** $m_C \in NC_m$ **do**
        **if** ConflictsWith ($\boldsymbol{m_C}$ , $CL$ ) **then**
            **return** true
    **return** false

**Define Procedure** cluster:
    **Def:** integer i = 0
    **While** ($NM \neq \Phi$) **do**
        define new cluster $\boldsymbol{CL_i}$
        **for** $m \in NM$ **do**
            **if** ConflictsWith (m, $CL_i$) **then**
                skip m
            **else**
                Add m to $\boldsymbol{CL_i}$
        i = i + 1
```

Figure 4.3: *The clustering algorithm.*

the compiler considers a memory access to be added to cluster *i*, that memory reference and all the unscheduled data flow and control memory accesses it depends on must not have a 'loop carried' dependence with any member of that cluster. The clustering algorithm has a complexity of O(nl), where n is the number of instructions that access memory, and l is the number of resulting clusters.

4.3.2 Reducing the Number of Temporaries

The ICE compiler attempts to minimize the amount of intermediate information communicated across the different `pardo` region splits, such as branch directions, loop states, and intermediate data. This information is stored to and retrieved from memory, which can cause performance degradation. So in order to achieve maximum performance, avoidable memory accesses must be eliminated or promoted to local variables inside the `spawn` blocks that resulted from the splitting where possible. Alternatively, communicated information must be aggregated such that it can be stored and retrieved in the least number of accesses possible. For that reason, 1. We take clustering a step further. Memory accesses scheduled to an earlier cluster are moved to a later clusters if these clusters contain members dependent on the memory accesses and it is legal to do so. For a move to be legal, a memory access must satisfy all the conditions in the lines (13 - 25) in figure 4.3 for the target cluster, and all clusters in between the target cluster and the memory access original cluster. 2. We use bit vectors to record the branch directions for split pardo regions, where each branch decision along the tree gets a single bit.

4.3.3 Fixing Control Flow after Clustering

The clustering process will result in reordering of memory accesses which can potentially distribute instruction of a basic block across two clusters or more. This causes two major problems: 1- Complicate and disorganize the control flow of a pardo region. Instructions that have the same parent basic block can be scattered across multiple (potentially) nonconsecutive spawn blocks, and will likely be preceded or followed by other instructions that belong to other basic blocks. More on this in subsection 4.4.3 2- A bigger problem is that it prevents the transformation of a serial loop within a pardo region, discussed in subsection 4.3 above, in which a split serial loop within a pardo block is replaced by a serial loop outside the resulting spawn blocks. Since, after clustering, the instructions belonging to that serial loop are likely to get mixed with instructions from other basic blocks that are not part of the serial loop.

We solve this problem by creating an empty replica of the Control Flow Graph (CFG) of the `pardo` region in all resultant `spawn` blocks. As such, every basic block inside the `pardo` will have an empty copy of it inside every resulting `spawn` block. Then a copy of the branch terminating the original basic block will be placed in each of the replicated basic blocks. This allows us to maintain the correctness of the control flow more easily, and allows a direct and uncomplicated placement of the memory accesses in their respective `spawn` blocks. Basically, a memory access is simply moved from the original parent basic block inside the `pardo` block, to the parent block's replica inside the `spawn` block where it belongs. Furthermore, we can still use memory to communicate control direction as discussed in section 4.2 above, however it now must be performed in every `spawn` block.

There are two exceptions where a basic block is not replicated:

1 If the basic block is a target of a conditional branch whose condition cannot be calculated at a specific spawn block yet because it depends on a memory access(es) that have been placed at a later spawn block. While the condition is not ready, the conditional branch will be replaced with a direct branch to the first common immediate post-dominator basic block of the conditional branch's targets.

2 If the basic block belongs to a serial loop inside a pardo block. Since, as was discussed in section 4.2, we achieve the back edge of the loop by creating a serial loop outside the spawn blocks and replace the loop with branches inside of it, the basic blocks from the loop cannot exist along basic blocks from outside it, since that means that these other basic blocks will execute every time the loop is executed. Instead, during clustering we make sure that a cluster is not shared between multiple loops (lines 17 - 18 of figure 4.3). As such, a split serial loop will be clustered into a set of consecutive spawn blocks.

4.4 The ICE Compiler Structure

The ICE compiler uses a modified Clang frontend and the LLVM compiler infrastructure to perform source-to-source translation of ICE program into XMTC program. Thereafter the XMTC code is compiled using the existing gcc-based XMTC compiler [12]. Clang was modified by adding the `pardo` ' keyword, and implementing the parsing of the `pardo` and the relevant IR code generation. Even though multiple LLVM passes were also implemented to accomplish all the various steps required to convert the lock-step semantics into threaded code, native LLVM passes were used wherever possible.

4.4.1 preliminary Code Optimization

The LLVM compiler stack is designed for serial threaded code executed by a single processor, making it incompatible with lock-stepped parallel code. Since the available compiler transformations do not take into account many of the properties of parallel code (e.g. differentiating between shared vs local variables or serial vs parallel contexts), Some steps were required to maintain the correctness of the ICE code when using native LLVM passes. For example, the beginning and end of a `pardo` block are marked when generating IR from source. Also, each parallel section is outlined into its own function, giving it a different context from its surrounding code. Furthermore, only the following native LLVM transformations that are guaranteed to not modify the memory ordering were used (listed by order of usage):

(1) **Control Flow Graph Simplification** `CFGSimplify` pass which removes all empty and extraneous basic blocks. This helps in making many of the passes the compiler uses more efficient in reasoning about control and data flow of the program[2].

(2) **Memory to register promotion** `mem2reg` pass which transforms the code into SSA (Static Single Assignment) form making subsequent optimizations much easier.

(3) **Instruction combine** `InstCombine` pass to combine instructions into simpler forms whenever possible. This helps in removing all extra instructions thus making

[2]There are many passes that are harmed by usage of this pass as well. However, since none of these passes are used in the ICE compiler, usage of CFGSimplify will only benefit the compilation process.

the code more efficient. It also helps in reducing the amount of information communicated across `pardo` splits

(4) **Global Value Numbering (GVN)** pass which finds all redundant instructions and remove them.

At this stage, the clustering and scheduling of `pardo` block instructions is performed, Which involve multiple stages to build the dependency graph and perform the clustering algorithm. It also involves all the steps taken to reduce the information communicated across splits. This is explored below.

4.4.2 Building the Dependency Graph

The clustering algorithm relies on dependencies between memory references to decide which `spawn` block each memory access will be assigned to. For that purpose, the ICE compiler builds a dependency graph that takes into consideration only the dependencies that may affect the correctness or performance of the translation from ICE as discussed in 4.2. The compiler relies on the Dependence Analysis (DA) pass in building that dependence graph.

The Dependence Analysis pass is a native LLVM analysis pass that uses certain dependency tests to determine whether a dependence exists between a pair of memory accesses, and if it exists, it attempts to provide as much information as possible about the dependence. This analysis pass builds an internal dependency graph based on mutual dependencies between memory references, and can be queried about the dependency relationship between two memory accesses, responding with one of three states; dependent (flow, output, anti), independent, or confused.

The dependence analysis pass checks for the conditions necessary to apply the suitable dependence test, and then applies that test. The pass performs each of the following dependence tests: the Zero Index Variable (ZIV) test, the Single Index Variable (SIV) strong and weak tests, the Restricted Double Index Variable (RDIV) test, and one of the Multiple Index Variable (MIV) tests. For readers interested in knowing the cases where each test applies, or the methodology each test uses to disprove dependence between a memory reference pair, consult Appendix A.

When the dependence pass is queried about a memory reference pair, it will first determine which dependence test is most suitable to prove independence between the reference pair, and then apply all the suitable tests. If independence cannot be proved, the pass will provide information about distance and direction vectors if the test used is capable of finding that kind of information. This information can be very useful for optimizing passes as well as for auto-parallelizing compilers, especially in case of nested serial loops.

In order to be able to use the dependence analysis pass, the ICE compiler creates fake loops based on the `pardo` regions. This is done for two reasons; **first**, by creating fake serial loops out of parallel `pardo` regions, the compiler is effectively checking for the problems bound to arise when a `pardo` region is translated as is to a single `spawn` block. Dependencies that are found between different iterations of the serial loop will translate as dependencies between the different threads created by the `spawn` . **Second**, This pass is intended for the dependence analysis within serial loops and structures, and is unable to recognize parallel loops including any lock-step loops. As a result, the dependence analysis pass is not going to be able to recognize the dependencies within a `pardo` region correctly, and it will not find any loop carried dependence. For that reason, making a parallel `pardo` region resemble a serial loop will allow the pass to provide all dependency information needed to build the dependence graph used by the clustering algorithm. The fake serial loop is created as follows:

(1) The compiler adds two new empty basic blocks, one for the loop header which is inserted right after the `pardo` region header, and another basic block that will replace the `pardo` block trailer, and will be the last basic block to execute in every iteration of the loop. For purposes of this explanation, this basic block will be referred to as the ”loop trailer”. After this is completed, the loop header should dominate all `pardo` basic blocks, and the loop trailer should post-dominate all the `pardo` basic blocks.

(2) The compiler adds the code required for checking the loop condition and places it into the loop header basic block. This check makes sure the loop index variable does not exceed the $high$ parallel context ID in the `pardo` statement. If the check fails, the loop execution terminates, and execution is set to continue at the `pardo` region's trailer basic block. The compiler also adds instructions to the loop trailer, for incrementing the loop

index by $step$ and an unconditional branch to the loop header.

(3) The loop index variable is initialized to low parallel context ID outside the loop.

Once the dependency graph is completed, the fake serial loop structure is removed, and all the changes are reversed. After the dependency graph completion, the compiler executes the clustering pass. Once clustering has completed and clusters are set, the compiler will split the `pardo` block, replicate the CFG as discussed in 4.3.3, and move all memory references and intermediate instructions to their respective clusters.

4.4.3 Maintaining Correctness of LLVM's SSA Form

The LLVM compiler Infrastructure Intermediate representation (IR) uses the Static Single Assignment (SSA) to represent the intermediate operations, and to maintain the def-use and use-def chains within the compiled module. One of the required properties of SSA representation is that *a Definition must dominate all its Uses*, with the only exception to this rule being if the user instruction is a phi-node. As a result of clustering and the subsequent replication of the CFG, followed by the placement of instructions into their parent basic block's replica within the cluster they belong to, it is often the case that the domination property of the SSA representation is broken. Figure 4.4 shows an example of this problem.

This problem is resolved through cloning the offending definition into memory in the cluster where it occurs, then reading the clone in the user instruction's cluster, right before the user instruction. Unfortunately, This solution may potentially result in cloning too many intermediate definitions into memory.

Each definition cloned will cost at least two memory accesses; `store` instruction for making the clone copy and a `load` instruction every time the cloned value is used by an instruction. Hence, in order to minimize the overhead of memory accesses, clones must not be performed unless it is necessary. As such the compiler checks for opportunities of cheaper options that enables the recalculation of the value instead of creating a clone of it. The recalculation is deemed cheaper if at most it costs less than a load and a

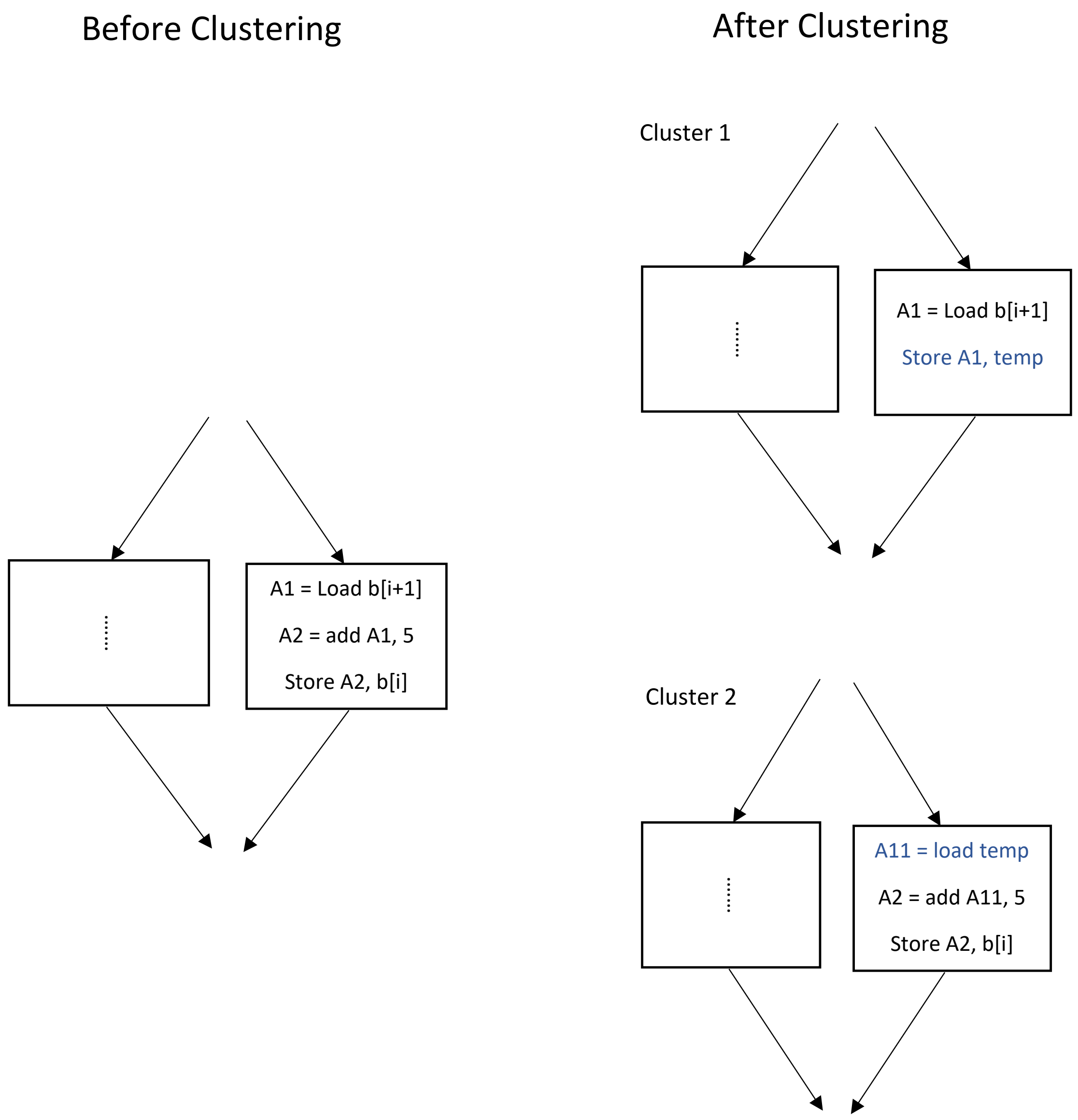


Figure 4.4: *Example showing placement of instructions into their respective clusters, before and after clustering, and the CFG replication process. The store depends on the load across multiple parallel contexts. The placement of instructions* $A1$ *and* $A2$ *into clusters 1 and 2 respectively, results in breaking the SSA dominance property. To resolve this problem,* $A1$ *is cloned to* $temp$ *in cluster 1, and* $temp$ *is retrieved to be used by* $A2$ *in cluster 2*

store. In that sense, unless the value to be cloned is (i) the result of a computation involving at least three loads from memory, or (ii) it is the result of the computation of two loads and the clone is going to be used in more than a single cluster, or (iii) it uses the results of memory read that is conflicted with other memory accesses within the cluster (as is the case in figure 4.4), then the value is not cloned and is recalculated instead. The compiler keeps track of the values that it cloned, so that later on after optimizing the control flow graph, the compiler will check which ones are not necessary anymore and remove them.

Following this, the compiler executes the *CFGSimplify* pass to remove all the unnecessary or empty basic blocks that resulted from replicating the CFG. After that, the compiler executes another round of the LLVM passes used for the preliminary optimization stage to clean up and remove all the extra variables or memory clones. After that the compiler will check to make sure that only the necessary clones are left. If the LLVM passes seem to have missed a clone that is not needed anymore, the compiler will remove it.

4.4.4 Transforming the LLVM IR into XMTC code

Finally, the compiler translates the LLVM IR to XMTC high level code using our XMTC backend. The XMTC backend is a modified version of LLVM native C Backend with added support to generate high-level XMTC code. Here the compiler does the splitting of `pardo` regions into `spawn` blocks based on the results of the clustering pass. Also, in this stage the compiler splits loops and conditionals as discussed earlier, create all arrays for communicating intermediate data, and any other steps required for generating correct XMTC code. After the XMTC code is produced, it is compiled using the existing gcc-based XMTC compiler [12] to produce binaries for the XMT FPGA and XMT cycle accurate simulator.

4.5 Support for Nested Parallelism

Correct translation of nested ICE code into nested XMTC code is similar to non nested ICE code in that it requires splitting the `pardo` region into multiple `spawn` regions. However, splitting an inner `pardo` region requires that all outer `pardo` regions containing it to be split as well. Translating nested ICE code by only splitting the inner `pardo` region without splitting any of the outer `pardo` regions will create multiple `spawn` blocks contained within one enclosing `spawn` block. Each 'parent' thread created by the outer `spawn` will in turn execute its instance of the inner `spawn` calls at its own pace, creating multiple tasks that are synchronous only with threads created by same `spawn` call instance. So, a 'parent' thread may potentially complete the execution of multiple inner `spawn` calls before any is executed by other 'parent' threads. Thus, the parallel contexts created by a nested `pardo` will not synchronize with other nested parallel contexts on same level of nesting, consequently, breaking the lock-step execution semantics of ICE.

Translating the nested ICE code will face many of the same problems and use many of the solutions used while translating non-nested ICE code with minor differences. Similar to the non-nested case, splitting an inner nested `pardo` will introduce many of the data flow and control flow problems discussed in section 4.2.2. To resolve this, temporaries will need to be created to communicate the intermediate data computations and control direction of the nested parallel contexts. However, since a split within the inner `pardo` requires that we split its enclosing pardo regions as well, the temporaries created should account for all the parallel context created across all enclosing parallel contexts on all levlels of nesting. Hence, the ICE compiler creates multi-dimensional array temporaries with as many dimensions as the levels of nesting.

Optimizing for the nested ICE code is performed in a very similar manner to the non-nested case as well. The clustering algorithm is still used to minimize the number of splits involved, as well as the optimizations used to reduce the number of temporaries. Control flow after clustering is still replicated. However, there are two changes that are added for nesting:

(1) **Building the Data Flow Graph** The change required was in extending the algorithm used to recognize the pardo nesting, which allowed the data dependency graph to capture the data flow across nested ICE code as well. Since the clustering algorithm relies solely on the dependency graph, the clustering algorithm does not require other changes.

(2) **Handling serial loops occurring within a nested** pardo **block** similar to how serial loops were handled in the

non-nested ICE code, serial loops are taken outside the `pardo` region containing it. Also, since all enclosing `pardo` blocks need to be split, the serial loop is also taken outside all enclosing `pardo` blocks and is executed by the MTCU, the loop condition becomes a flag indicative of the existence of threads that have not completed execution yet, and the original loop termination condition becomes a normal branch within the `pardo` where the loop used to reside and is treated as in the regular branch case.

Chapter 5: Evaluating The ICE Language: Results and Analysis

This chapter presents the results of the experiments of writing and compiling ICE programs, and compare the results to programs written in the XMTC language. First, this chapter examines the difference in ease of programming between ICE and XMTC by showing a comparison of the number of lines of code needed to implement the same algorithms. Then, it examines the translation accuracy of the ICE compiler, by comparing the ICE to XMTC translation produced by the compiler, to the hand-optimized XMTC in terms of the number of required `spawn` blocks and temporaries used. Finally, This chapter lists and examines the performance of ICE programs for the used benchmarks, and compares to that of the manually-optimized XMTC.

5.1 Environment and Methodology

Since ICE is a new language with no standardized benchmarks, a suite of 16 benchmarks based on common PRAM algorithms was developed to be used for the experiments. This benchmark suite contains benchmarks that can be classified as nested and non-nested algorithms, or regular and irregular programs. A list of the the non-nested benchmarks in the new benchmark suite is available in table 5.1, and the nested benchmarks can be seen in table 5.2. four of the five nested benchmarks had a non-nested version of them. In exception of BFS, the non-nested versions of the algorithms were essentially flattened versions of the nested benchmarks. The fifth benchmark (*i.e.*, topological sort) does not have a non-nested counterpart in the benchmark suite. A detailed description of each of the algorithms the benchmarks were based on can be found in [25, 26, 27]. For each benchmark, a lock-step pseudo-code was written based on the PRAM algorithm for the benchmark, then based on that pseudo-code two versions were implemented: an XMTC version that is manually optimized for best performance, and the ICE version. The ICE versions was compiled using the

Table 5.1: *A list of the non-nested benchmarks.*

Benchmark	**Problem Size**	**Abrv.**
Integer Sort*	1048576	INT
Merging*	1000000	MRG
Sample Sort*	131072	SMP
Breadth First Search*	32768 nodes 65536 edges	BFS
Graph Connectivity*	32768 nodes 65536 edges	CVTY
Maximum Finding	262144	MAX
Tree Contraction	32768 nodes	CTRC
Tree Rooting*	32768 nodes 65536 edges	RANK
2D Jacobi Stencil Computation	512x512	JAC
LU Factorization	512x512	LU
Cholesky Factorization	512x512	CHO

new ICE compiler, then the compiler's output XMTC code is compiled using the XMTC compiler. The same XMTC compiler is used for compiling both the hand-optimized XMTC program and the automatically generated XMTC code that was translated from ICE. After the XMTC compiler have produced an XMTC executable, it is executed using the 64-TCUs XMT FPGA.

Table 5.2: *A list of the nested benchmarks.*

Benchmark	**Nesting Depth**	**Problem Size**	**Abrv.**
Topological Sort	2	32768 nodes 65536 edges	TOBO
Breadth First Search*	2	32768 nodes 65536 edges	NBFS
2D Jacobi Stencil Computation	2	512x512	NJAC
LU Factorization	2	512x512	NLU
Cholesky Factorization	2	512x512	CHO

5.2 Ease of use and Code size

This section examines the differences in code sizes for both ICE and XMTC implementations of all benchmarks. The code size is used as a measure of ease of programming. This is fair because ICE and XMTC are extensions of the C language, each featuring an extra keyword to express parallelism: `pardo` to lock-step parallelism in ICE, and `spawn` for expressing threads in XMTC. Both languages are identical otherwise. This means that for the same pseudo-code of an algorithm with same inputs and outputs, smaller code indicates simpler programs, and the increase in code size was due to the elaboration needed to ensure correctness

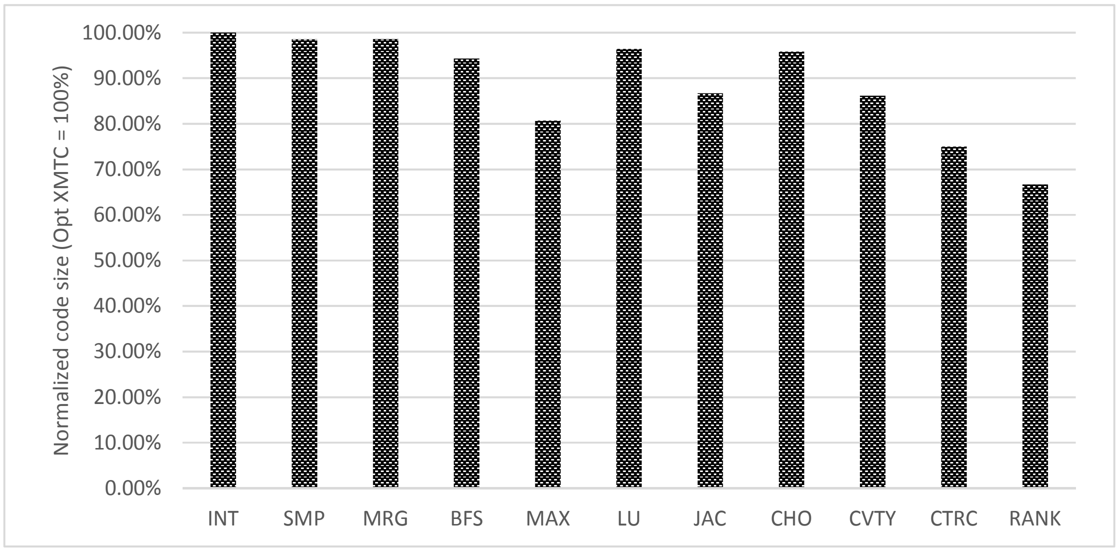


Figure 5.1: *Code size for the entire program normalized to XMTC.*

and/or higher performance, as is the case in the example in figure 3.2. Thus, we believe comparing lines of code to approximate ease of programming is a valid approach to demonstrate the ease of programming of ICE compared to XMTC.

Two different measurement of code size are provided: a measurement for the parallel algorithmic part only to examine the ICE language ability in help programmers write simpler parallel code, and a measurement for the entire program provided for completeness. For both measures, the number of lines of code naturally excludes white spaces and comments. Also, each variable is declared on a separate

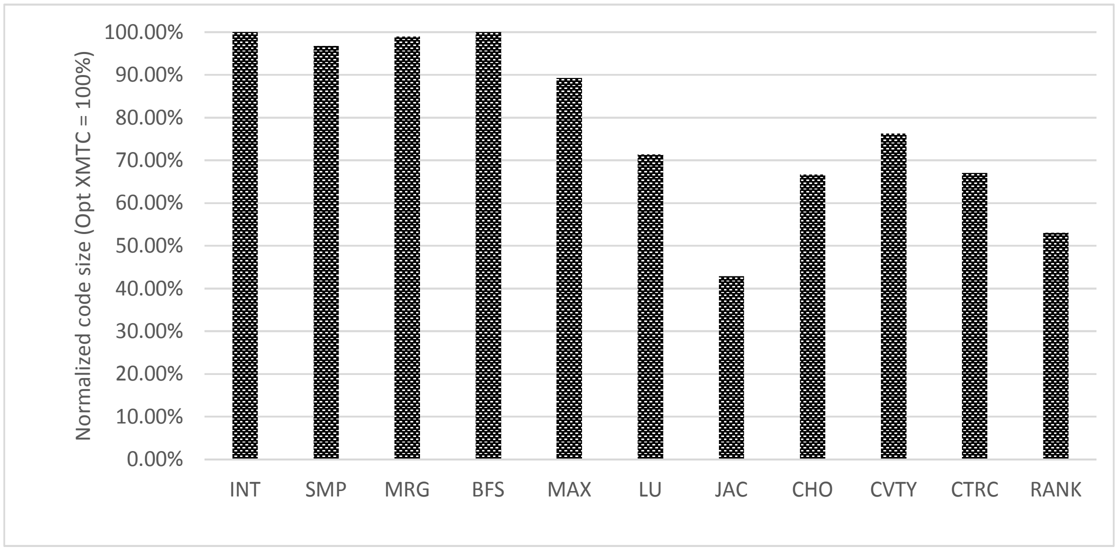


Figure 5.2: *Code size of the algorithm's parallel sections normalized to XMTC.*

line. For the algorithmic parallel portion of the code, we measure only the benchmark's code size for parallel sections only, excluding all shared variable declarations and non-recurring initializations, all serial algorithms used as part of the main parallel algorithm (*i.e.*, serial sorting or summation, etc.), the `main` function, and all preprocessor directives.

Figure 5.1 shows a comparison of the reduction in the entire program code size for non-nested ICE normalized to optimized XMTC. This graph shows that ICE has a smaller code size when compared to XMTC for seven out of our eleven benchmarks. The other four benchmarks saw no reduction in code size, since they contain none of the cases that ICE can help programmers with. These benchmarks were included only as a base-line case. ICE provides a reduction in the size of code by 11.01% on average for the entire benchmark suite, and 16.08% on average for the benchmarks that showed an improvement.

Figure 5.2 shows the percentage of code size reduction for the parallel algorithm part of the benchmark for non-nested ICE when normalized to the XMTC version. Here as well, ICE provides the largest reduction in size of code when compared to XMTC with reduction of up to 57.14% in some cases. ICE provides an average reduction of 21.61% for the entire set, and 33.35% for benchmarks that showed an improvement. This shows the potential of ICE to reduce code size (and therefore programming effort) compared to XMTC, which is a more traditional threaded language.

figures 5.3 and 5.4 show that the ease of programming benefit of ICE extends to nested ICE as well. Figure 5.3 provides a comparison of the reduction in size of the code for the entire program for nested ICE programs normalized to optimized nested XMTC programs, while figure 5.4 shows the percentage of code size reduction for the parallel al-

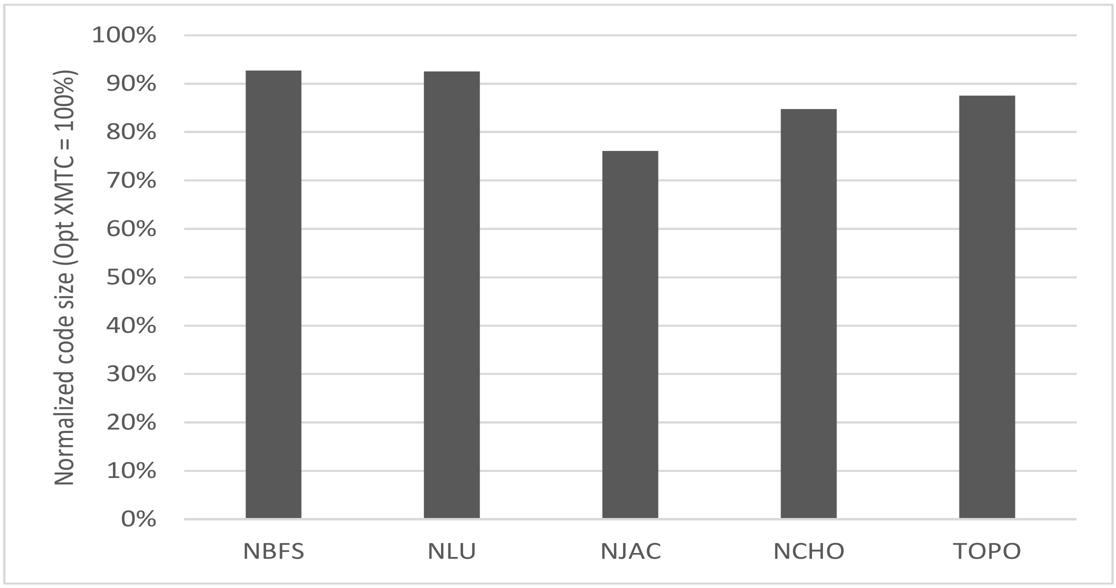


Figure 5.3: *Code size for the entire program normalized to XMTC for nested benchmarks.*

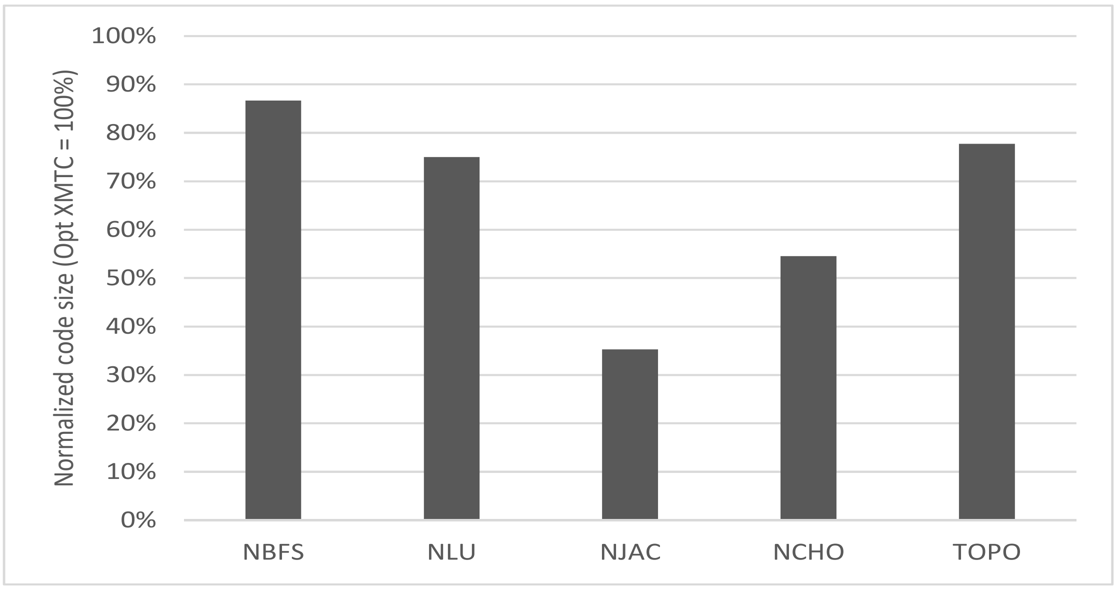


Figure 5.4: *Code size of the algorithm's parallel sections normalized to XMTC for nested benchmarks.*

gorithmic part of the benchmarks for nested ICE when normalized to the nested XMTC version. In both figures, ICE provides an average reduction in code size of 13.28% for the entire program, and 34.14% for the parallel algorithm portion of the code. We also notice in figure 5.4 that the maximum reduction in code size for the algorithmic part of the program was 64.71%.

Finally, When examining the reduction in code size for the entire benchmark suite, ICE provides an average reduction in the size of code by 11.72% for the entire program, and 25.53% for the parallel algorithmic part only of the code.

5.3 Accuracy

In this section we take a look at the ICE compiler's accuracy and effectiveness in translating to XMTC. We look at the number of spawn blocks and temporaries[1] used to implement our benchmarks. We believe that this will help demonstrate the ICE compiler's effectiveness in producing high performance XMTC programs, due to the effect spawns and temporaries has on the runtime performance of the translated XMTC code as discussed in section **??**

[1]Each temporary was used to store only one value that may be read multiple times.

We look at table 5.3 to see the number of spawns and temporaries used by the programmer and the ICE compiler. This table shows that nine out of 15 of the 16 benchmarks had the same number of spawns and temporaries in both XMTC versions. The benchmark that is left had more spawns and temporaries compared to hand-written XMTC. This benchmark had multiple independent indirect memory references that cannot be detected by compilers. However, the programmer for the hand-written version was able to avoid the extra splits and temporaries.

The ability of the ICE compiler to generate high quality code is highly reliant on the performance of the alias analysis used to determine the dependencies between memory accesses. These dependency relationships are used during the clustering step to determine the number of required splits and `spawn` blocks as was discussed in 4.3.1. Whenever uncertain about a dependency, the compiler conservatively assumes a dependence exists anyway. This means that whenever alias analysis is queried about memory references and it provide definitive answers of no-alias, the clustering algorithm makes better clustering decisions, and ultimately produces a more efficient code. Alias analysis is

Table 5.3: *Number of spawn blocks and temporaries in both XMTC programs.*

Benchmark	**Hand-written XMTC**		**Generated XMTC**	
	Spawns	Temp.	Spawns	Temp.
Integer Sort	3	0	3	0
Merging	4	0	4	0
Sample Sort	8	0	8	0
Breadth First Search	3	0	3	0
Graph Connectivity	12	2	13	3
Maximum Finding	4	0	4	0
Tree Contraction	7	4	7	4
Tree Rooting	5	2	5	2
Jacobi	2	1	2	1
LU Factorization	1	0	1	0
Cholesky Factorization	2	0	2	0
Topological Sort	5	0	5	0
Nested Breadth First Search	3	0	3	0
Nested Jacobi	4	1	4	1
Nested LU Factorization	2	0	2	0
Nested Cholesky Factorization	3	0	3	0

a large field of compiler theory research and any advancements within the field will benefit the operation of the ICE compiler. However, it is outside the scope of this thesis and we will not discuss it any further.

5.4 Performance

The XMT platform is excellent at exploiting parallelism in irregular algorithms. A list of examples of published work that shows XMT's speedups against commodity superscalar architectures was discussed in section 2.7.1. This also validates the choice of comparing the performance of the ICE language to the XMTC language for binaries executed over the XMT platform.

In this section, we will focus on the performance comparison between binaries compiled from ICE and XMTC. The XMT FPGA, which has 64 TCUs, was used to obtain these performance measurements listed below for both hand-optimized XMTC and ICE versions of the same algorithm pseudo-code. Figures 5.5 and 5.6 provides the speedup of ICE normalized to hand-optimized XMTC for both nested and non-nested cases. Figures 5.7 and 5.8 shows the net run-time improvement of ICE relative to hand-optimized

XMTC, normalized to hand-optimized XMTC for both nested and non-nested cases. At the end of executing a binary, the XMT FPGA provides the number of cycles required to execute the XMT binary. These numbers were collected for all 16 benchmarks, and then used for performance comparisons. The cycle count of hand-optimized XMTC is used as basis for the comparison, and these figures show the the performance results for the ICE code normalized to hand-optimized XMTC programs.

To ensure that ICE is being compared to the fastest hand-optimized XMTC many steps were taken. Since memory accesses are the biggest source of overhead in XMT, temporaries were not used in XMTC programs unless it was necessary. This is can be seen in table 5.3 where thirteen benchmarks used no temporaries and fifteen use two temporaries or less. The other lesser source of overhead comes from the creation and termination of threads. This overhead is very small in XMT and have negligible effect on the validity of the comparisons presented in this section.

ICE achieves comparable performance to hand-optimized XMTC, which takes considerably more programming effort to write than ICE. Figure 5.7 shows that ICE has a 0.76%

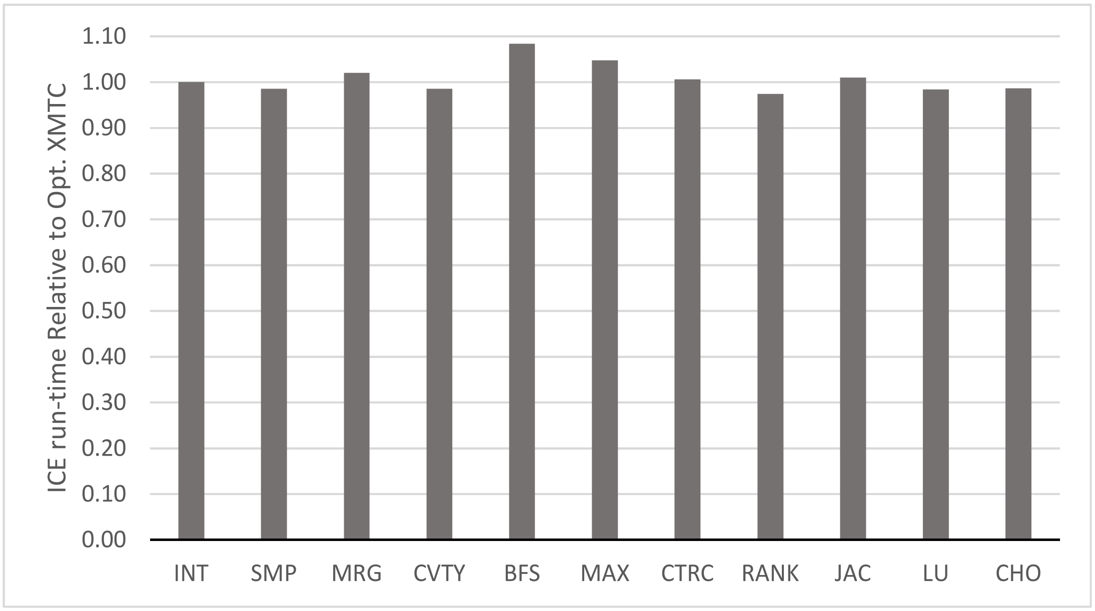


Figure 5.5: *64 TCU XMT processor speedup comparison of non-nested ICE programs normalized to performance of hand-optimized XMTC*

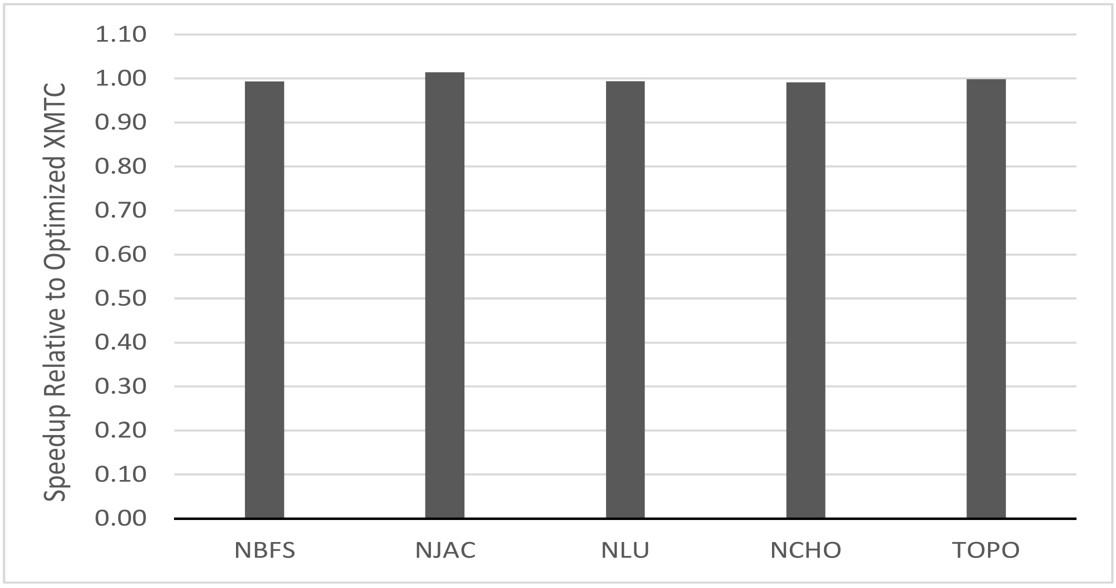


Figure 5.6: *64 TCU XMT processor speedup comparison for nested ICE programs normalized to performance of hand-optimized XMTC*

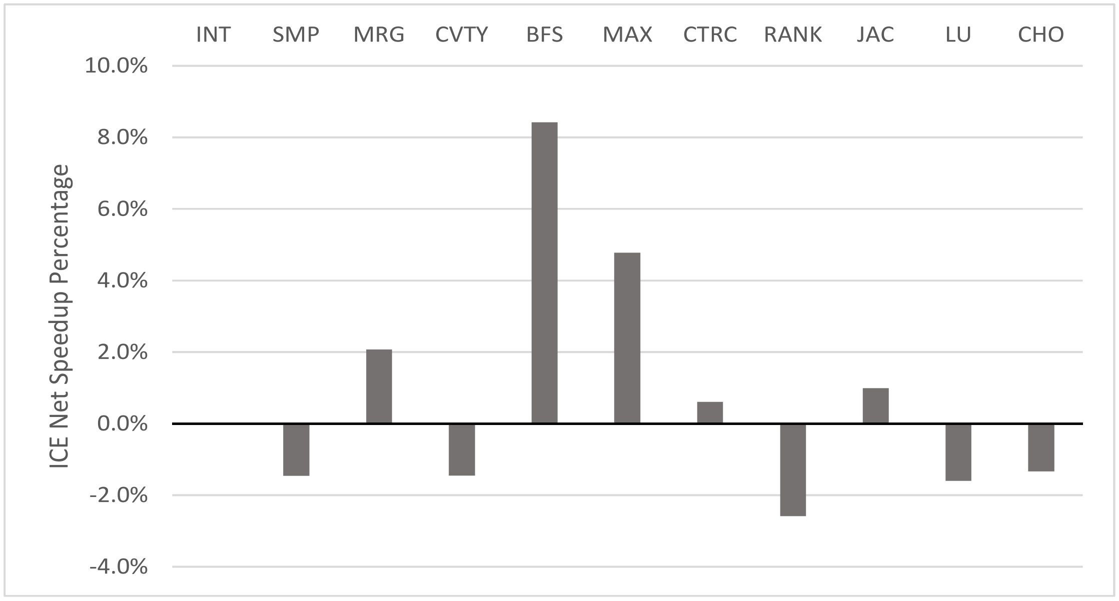


Figure 5.7: *64 TCU XMT net speedup of non-nested ICE programs normalized to hand-optimized XMTC*

speedup on average for non-nested benchmarks, with maximum slowdown of 2.5% when compared to the performance of optimized XMTC. Figure 5.8, shows that ICE nets no slowdown on average when compared to hand-optimized XMTC, with a maximum slowdown of 0.91%. Such minor performance penalties for a much easier programming effort is an obvious good choice for programmers. For non-performance-expert programmers who cannot write highly optimized XMTC code, ICE might even provide a speedup.

The figures also show that for some benchmarks, ICE has achieved a speed up when compared to hand-optimized XMTC. In this work, we do not claim that ICE can provide

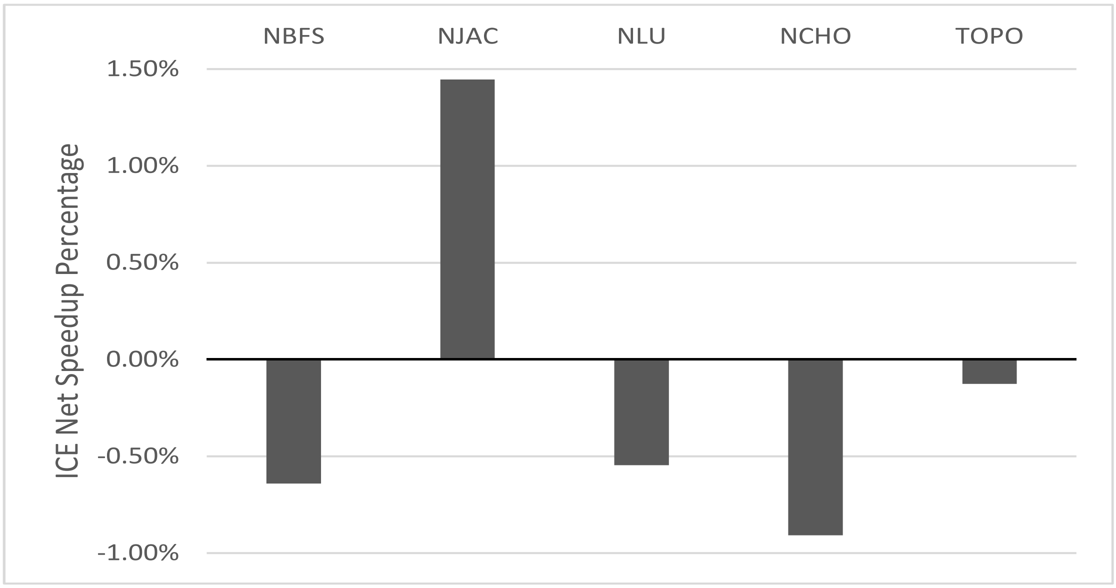


Figure 5.8: *64 TCU XMT net speedup of ICE normalized to optimized XMTC for the nested benchmarks*

speed ups over XMTC for expert programmers, since intuitively hand optimized parallel code should always be faster. Upon investigating, it was found that there are multiple factors contributing to the observed speed ups. For some benchmarks (Merging benchmark, Maximum finding benchmark, non-nested Jacobi benchmark), the ICE code was accurately translated to its equivalent XMTC code (*i.e.*, It has the same number of `spawn` blocks and temporaries). However, the program layout of both version is different. This suggests that the performance gain is a result of factors unrelated to the translation such as data location in the read-only cache, instruction scheduling, the whether the relevant data was pre-fetched, or what optimizations opportunities were recognized and performed by the XMTC compiler. For another benchmark subset (non-nested BFS benchmark, and tree contraction benchmark), the performance gain was a result of the LLVM compiler's native optimizations which is more recent than the GCC compiler used in XMTC implementation. This is combined with the ICE compiler specific optimization that were implemented. When a PRAM algorithm requires multiple synchronization points within a deep nested if-else block, the condition needs to be re-evaluated after each point. However, the ICE compiler used bit vectors to record the evaluation results for multiple branches which will require a single memory read per a spawn block will be sufficient to retrieve the control flow information as was discussed in 4.3.2. Since a programmer is very unlikely to use bit vectors to record results of multiple branches, multiple reads per spawn block are needed for condition evaluation.

To compare the performance of ICE for both the nested and non-nested versions of an algorithms, figure 5.9 is provided. This figure shows a comparison of both the nested and non-nested versions net speedups as compared to hand-

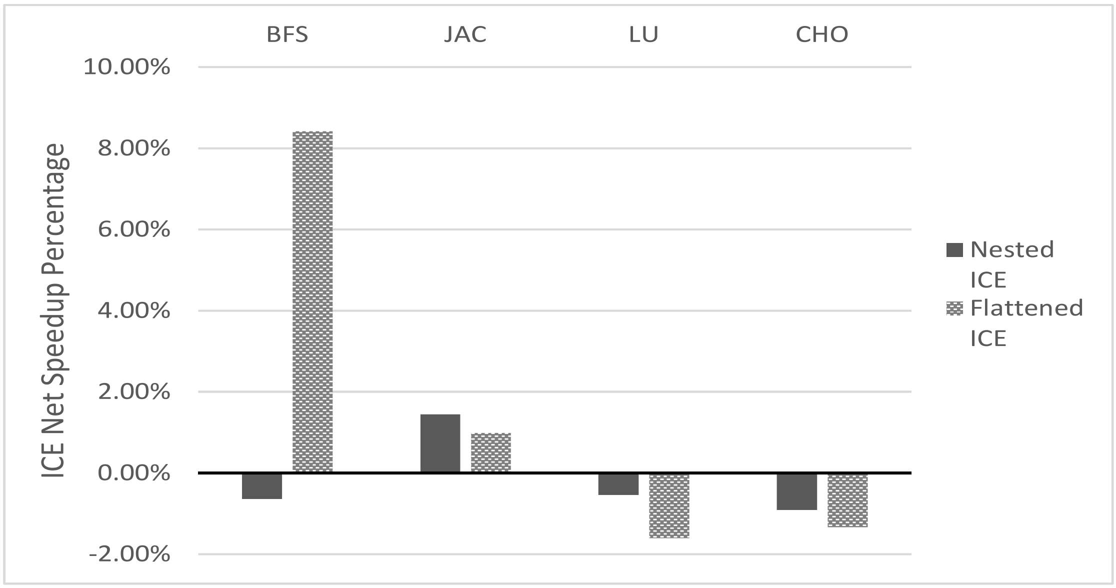


Figure 5.9: *64 TCU XMT net speedup comparison between nested and non-nested ICE normalized to optimized XMTC*

optimized XMTC. it can be seen from the figure that for three of the four benchmark pairs, the nested versions achieved slightly better speedups compared to the non-nested versions of the benchmarks, whereas for the fourth benchmark, the nested version achieved significantly lower performance when compared to its non-nested counterpart. We believe that this was mainly due to the minor changes made to the algorithm to be able to write a non-nested version of it. We do not think that we can make conclusions on which method is better based on such a small subset.

The ease of programming of ICE permitted the writing of programs directly from a (PRAM) algorithm with less effort than that required for writing hand-optimized XMTC, while maintaining comparable performance through using a compiler that automated the process of optimizing the the produced XMTC code.

Chapter 6: Related work and Conclusion

6.1 Related work

As mentioned in the introduction, over 225 parallel languages have been proposed. It is not practical to discuss all of them here. We will focus on languages that are most closely related, either because they have an algorithmic foundation, such as PRAM, or have an ICE-like lock-step execution model; or are meant for hardware like XMT suited to irregular programs. In summary, we have not found any related work that has the full ecosystem that ICE offers of an easy-to-program language, with a sound algorithmic foundation in PRAM theory, a capable compiler mapping to threaded programs, and a hardware that is capable of exploiting fine-grained irregular parallel programs.

The goal of this proposal is to allow programmers to use as freely as possible an extended form of lock-step programming similar to the way parallel algorithms are expressed in the PRAM literature, whenever they prefer to do so. We call this extended form ICE programming. The general work-depth framework (due to [30]) is used in several parallel algorithms texts [25, 26, 27] for describing PRAM algorithms. ICE is the immediate concurrent execution abstraction presented in [2] for popularizing this framework.

To facilitate this goal, the statement of work in the proposal includes mapping the ICE lock-step semantics onto the XMT multi-threaded semantics while achieving the best performance we can. As we start from lock-step specification, the performance objective entails reducing synchrony in an automatic way. So far, XMT programming of PRAM algorithms was done using the modest XMTC extension to C. In particular, [29] suggested a “programmer’s workflow” that guides the programmer on how to advance the ICE abstraction of an algorithm, called there high-level work-depth (HLWD), to an XMTC program and how to tune its performance. The original XMT hardware allowed us to achieve strong speedups over the best serial algorithm for many parallel algorithms implemented using this workflow [29]. Follow-up architecture, compiler and run-time enhancements further improve these speedups. The current proposal seeks to significantly reduce the algorithm-to-computer-program effort by the programmer. Instead of an XMTC program, the programmer will handoff a specification of the algorithm using ICE programming. The XMT implementation (e.g., run time) should be “on par” with hand-optimized XMTC code.

DARPA launched the HIGH Productivity Computing Systems (HPCS) program with the purpose of building systems that can be programmed productively. It resulted into three languages; Cray’s CHAPEL [31], SUN’s Fortress [32], and IBM’s X10 [33]. Although all these languages have ease of programming and high productivity as a goal, none is suited for the lock-step model of PRAM algorithms. Further all these languages require manual specification of synchrony and concurrency, whereas the ICE compiler automates the process. Finally, these languages are intended to be mapped to traditional coarse-grained hardware; hence they perform poorly on irregular programs when compared to XMT.

APL is an early example of high-level programming that allows for lock-step parallelism. A series of papers that appears to have culminated with [34] sought execution of compiler-extracted parallelism from APL programs on the IBM RP3. The IBM RP3 built on the NYU Ultracomputer project, which also inspired XMT. However, APL did not provide sufficient support for the PRAM parallel algorithms literature. The V-RAM [35] appears to be the first lock-step programming model aimed at implementing this literature. However, it was a lock-step model targeting vector hardware. NESL that followed was not lock-step, but, still appears to have targeted machine models for which synchronization was relatively easy; see, e.g., [36]. In any case, we are unaware of speedup results for these approaches (APL, V-RAM, NESL, etc.) that approach XMT results, especially for irregular applications.

The case for (lock-step, nested) ICE programming It is instructive to go back and read the motivation for NESL in [37]. Parallel programming constructs were needed for specifying parallel algorithms. Blelloch examined parallel algorithms that are described in the literature and their pseudo-code. He found that nearly all are described as parallel operations over collections of values. For example "in parallel for each vertex in a graph, find its minimum neighbor", or "in parallel for each row in a matrix, sum the row". The algorithms usually consist of many such parallel calls interleaved with operations that rearrange the order of a collection, and can be called recursively in parallel. He used Quicksort as an example. Hillis and Steele [38] called this ability to operate in parallel over sets of data data-parallelism. The languages based on it are often referred to as data-parallel languages. Quite a few parallel languages comprise data-parallel features as well as other forms of parallelism [39, 40, 41, 42].

Following on his earlier work [35], Blelloch contrasts flat data-parallel languages, where a function can be applied in parallel over a set of values, but the function itself must be sequential with nested data-parallel languages; in the latter, any function can be applied over a set of values, including parallel functions. For example, for multiplying a matrix by a vector, the summation of each row of the matrix could itself execute using parallel summation. We concur with his claim that the ability to nest parallel calls is critical for expressing algorithms in a way that matches our high-level intuition of how they work; e.g., nested parallelism can be used to implement nested loops and divide-and-conquer algorithms in parallel.

As the multi-threaded architectures gained popularity, the need for nesting, encouraged by Blelloch's work, gained momentum. Cilk [43, 44] would be a good example for general multi-threaded programming. Per [35], flattening nested parallelism was important for the implementation of high-level languages since they allow a compiler to translate the high-level description of nested operations onto its low-level implementation on a flat real (vector-like) machine. Multi-threaded architectures, on the other hand, allowed greater implementation flexibility. Cilk contributed important compiler and run-time techniques such as work-stealing for implementation of nested parallelism. Our prior work [45] has already built on this flexibility by further optimizing work stealing to an improvement called Lazy Binary Splitting (LBS). Cilk++ [46] has incorporated a concept of reducers. They showed that their work-stealing scheduler can support reducers without incurring significant overhead. We will also try to extend our LBS enhancement of work-stealing scheduling to reducers.

In contrast to Cilk programming, ICE stays clear of race conditions and other complexities that led to the grave productivity concerns with respect to general multi-threaded programming. ICE also directly connects with the parallel algorithms literature – the original problem that nested data-parallelism addressed. Namely, while being inspired by the compiler and run-time support in [44], the XMTC program handoff point was already closer to the algorithm; thus, reducing the human effort. The proposed work will get it even closer, further reducing the human effort. The gap between Cilk and the parallel algorithms literature is made clear by comparing the multi-threaded algorithms section in [47] with parallel algorithms texts [25, 26, 27]. A key difference is that while [47] also favors work-depth performance analysis, it does not equip the programmer with the same level of freedom for designing for work-depth performance. This point was demonstrated with respect to parallel algorithms for merging in [2]. Cilk, of course, has the important advantage of being supported by today's commodity hardware and is, in fairness, more accommodating to programmers than much of its immediate competition. However, commodity hardware cannot exploit irregular parallelism as effectively as XMT.

Our central question is: How should the programming (handing off human input to machine processing) of parallel machines be? The thinking on this central question has been affected by changes in technology and parallel architectures over time (e.g., the move from vector to multi-threaded architectures). In contrast, the literature on basic parallel algorithms appears to be more resilient to these changes, in spite of vigorous attempts by numerous researchers. This resilience suggests that there is considerable intellectual and practical merit in advancing programming specification that will make it easier to unleash the wealth of this knowledge base: 1. This specification should be as close to the original parallel algorithm and simple to produce as possible. 2. It should be efficiently implemented on some architecture platform, as part of its proof of concept. 3. Once the proof of concept is established, a proper set of algorithm specification could create a benchmark for guiding future parallel architectures; obviously different architectures will need different adjustments to compiler and run time solutions. 4. However, the main lesson for the success of XMT on ease-of-programming is that support of the theory of parallel algorithms, and in particular *its concept of parallel algorithmic thinking is no less important for the design of parallel systems than any set of specific applications or features*. This is also the biggest departure from standard computer architecture practice.

6.2 Conclusion

We present ICE, a new lock-step easy-to-program parallel programming language based on the PRAM algorithmic model. We present the ICE compiler that we developed which translates the lock-step ICE programs into a traditional threaded XMTC programs. We demonstrate that the ICE compiler can provide comparable performance to highly-optimized XMTC programs while requiring much less effort from the programmer. We show how ICE easiness-to-program works in synergy with XMT's efficient parallelization of irregular programs to strike the ever-sought balance between the compiler and the programmer roles in producing parallel programs, where the programmer needs only to specify parallelism and rely on the compiler to do the rest. Finally, given the relatively slow progress in parallel programming language technologies for irregular programs, our works suggests new opportunities for benchmarking parallel machines by their efficient support of high-level parallel algorithmic languages.

We conclude with a broader perspective on the significance of our contribution. It should be clear that ICE (or work-depth) parallelism exists in every serial algorithm. The only effort needed when we wish to use parallelism inherent in a serial algorithm is to express it, which in our experience is just a matter of skill, with no creativity involved. In contrast, practically all commercial approaches to parallel programming are based on partitioning the work to be done among processors or threads. There is no clear path for deriving that from a serial algorithm, and, when doable, requires significant creativity; in fact, in many cases it either cannot be done or cannot be done beyond very limited levels of parallelism. This extra level of creativity raises the bar on the skill and effort of programmers, and has greatly limited the adoption of many cores among programmers and application software vendors. Our paper, along with prior XMT work, establishes that there is a way to avert the above practice, which arguably amounts to throwing the parallel programmer under the bus, through proper hardware and software design choices.

Chapter A: Dependence Tests and Analysis

Dependence testing is a method by which it is determined whether dependence exist between two subscripted memory references to the same array in nested loops [48]. It is difficult to calculate data dependencies for arrays, due to arrays referencing many different memory locations. This Appendix will give an overview description of the methods used for the high-precision testing of data dependencies.

Dependence testing has two main goals; to prove no dependence exists, and provide best possible characterization of the possible dependence, in the form of distance and direction vectors. Dependence tests are conservative in nature, so if a test cannot prove the independence of two memory references, it must assume that a dependence exists. All the dependence tests below assume that all the index variables of loop nests have been identified, and that all auxiliary induction variables have been detected and replaced by functions of the loop indexes.

To build the dependence graph it uses for splitting `pardo` regions, the clustering pass in the ICE compiler relies heavily on information provided by LLVM's native Dependence Analysis (DA) pass. The DA pass implements all the tests discussed within the chapter, and decides which test (or group of tests) to use, based on the subscripted reference pairs being queried. This appendix is intended to give a better understanding of the kinds of dependencies the DA pass, and by extension the ICE compiler, will be able to handle. For that reason, this appendix will only discuss the dependence tests implemented within LLVM's DA pass.

A.1 Overview of Dependence Testing

As the various array subscripts classifications have differing complexity levels involved in testing them. Subscripts should be partitioned according to their complexities, and be tested accordingly. This allows testing procedures of array reference pairs based on the partitioning. These procedures are:

1. The subscripts are partitioned into minimal and separable coupled groups.

2. Classify subscripts according to how they are indexed (*i.e.*, ZIV, SIZ, MIV)

3. For each subscript the appropriate subscript test should be applied (ZIV, MIV, SIV tests). These tests are determined based on a subscript's complexity. The goal of these tests is either prove independence, or attempt to calculate distance and direction vectors between an array reference pair.

4. If two memory accesses subscripts were proven to be independent, by any of the tests used, that means the memory references are independent and testing should be terminated.

5. For each subscript group, apply multiple tests to produce distance and direction vectors for the occurring indexes within that subscript group.

6. otherwise, For every memory reference pair, merge all the distance/direction vectors calculated in previous steps in a single distance/direction vector.

A.2 Zero Index Variable Test

The Zero Index Variable (ZIV) test is a loop-carried dependence test that looks at array subscripts has no index variables (*i.e.*, subscripts that are loop invariant). Examples of subscripts where ZIV test can be used are $A[3]$ and $A[4]$, or $A[t]$ and $A[t+1]$ where t is not an induction variable

In this test, if two subscripts are proved to be unequal, then the corresponding memory references are independent. However, if the test failed, then caluclating distance/directional vectors can be ignored. Since they contain no induction variables, the subscripts have no contribution to any distance/direction vectors. An example of subscripts

A.3 Single-Subscript Dependence Tests

After the partitioning and classification of subscripts have been completed, the tests used to check for the existence of dependences between memory references are applied. When a dependence exists, the tests will attempt to provide as much information as possible about the nature of the dependence in the form of the distance/direction vectors.

This sections looks at tests that are applied over single subscripts.

A.3.1 Single Index Variable Tests

The Single Index Variable tests is a group of loop-carried dependence tests that looks at array subscript pairs that use a single loop induction variable. These subscripts take the form $a_1 i + c_1$ and $a_2 i + c_2$, where a_1, a_2 are constants, c_1, c_2 are loop invariant variables, and i is the loop induction variable. SIV scripts is the most commonly occurring form of array subscripts.

An exact SIV test for linear SIV subscripts is suggested in [49, 50, 51]. These methods rely on finding all possible solutions for a two-variables linear Diophantine equation. Other simpler methods exists where the SIV subscripts are categorized into weak SIV and strong SIV. All these methods are discussed below.

A.3.1.1 Strong SIV Test

A pair of SIV subscripts is said to be strong SIV, if it takes the form $ai + c_1$ and $ai + c_2$ where i is an induction variable, a is a constant, and c_1 and c_2 are loop invariants; *i.e.*, when it is linear, and the index i coefficients are constant and equal. This test is capable of proving independence, and if it fails to do that, it can calculate the distance/direction vectors.

A geometric representation of the subscript pair would translate into two parallel lines, due to which common elements will always be separated by a constant distance between them for the different loop iterations.

To find the distance separating the subscript pair, working from the beginning:

$$a_1 i + c_1 = a_2 i' + c_2 \tag{A.1}$$

Since $a_1 = a_2$, the equation is simplified giving:

$$ai + c_1 = ai' + c_2 \tag{A.2}$$

After working out the above equation, the dependence distance d is:

$$d = i' - i = \frac{c_1 - c_2}{a} \tag{A.3}$$

A dependence exists between a memory reference pair if and only if common access happen to elements within loop bounds. This translates to that subscript pair are independent when d does not have an integer value, or $d > (U - L)$, where U is the upper bound of the loop, and L is the lower bound of the loop. However, if d has an integer value, and

$|d| \leq (U - L)$, then a dependence exists and the direction of the dependence is defined as :

$$direction = \begin{cases} < & d > 0 \\ = & d = 0 \\ > & d < 0 \end{cases}$$

An exception to the dependence rule is when $|d| = 0$, that means that the subscript pair are dependent only when they have the same array index, which in turn means that the memory reference pair are loop independent references[1].

One of the main advantages of this test is its ability to be easily extended to handle loop-invariant symbolic expressions. This is accomplished by first evaluating the distance symbolically, then if the distance evaluates to a constant, the test proceeds as discussed above. However, if that is not the case, loop bounds difference is calculated and is compared with the distance symbolically. An example of this case is the subscript pair $(i + 2N), (i + N)$ referenced within a loop bounded by $[1, N]$. When the distance is calculated, it is found to be $d = N$. This is compared to the the difference between the loop bounds $(N - 1)$. However, $N > (N - 1)$, which proves that this pair are independent.

[1]However, that does not mean that the reference pair are independent from one another within the same iteration of the loop

A.3.1.2 Weak SIV Subscripts

A pair of SIV subscripts is said to be weak SIV if they take the form $a_1 i + c_1$ and $a_2 i + c_2$ where i is an induction variable, a_1, a_2 are constants, and c_1 and c_2 are loop invariants. This can be solved using the exact SIV test. However, there are special cases where it is easier to test using their specific properties. A geometric representation of the subscript is two lines intersecting at a specific point where:

$$a_1 i + c_1 = a_2 i' + c_2 \tag{A.4}$$

A weak SIV test can be formulated to check if the point where the lines intersect is of an integer value within the loop bounds. There are two special cases based on this; the Weak-Zero SIV and the Weak-Crossing SIV. finding either of those two cases allows testing without requiring the use of exact SIV test.

A.3.1.3 Weak-Zero SIV Test

A pair of SIV subscripts is said to be zero-weak SIV that take the form $ai + c_1$ and c_2 where i is an induction variable, a is a constant, and c_1 and c_2 are loop invariants.

This test is capable of determining independence. If it fails to prove that, it can sometimes refine for direction.

This test relies on the fact that one of the coefficients a_1, a_2 is 0. This causes the dependence equation to reduce to:

$$i = \frac{c_2 - c_1}{a} \tag{A.5}$$

This suggests that one of the subscript pair references only one specific array element. So this tests amounts to checking at what integer loop index causes the subscript pair to access the same array element, and whether it is within loop bounds.

The weak-zero SIV test checks for dependences caused by a specific loop iteration. If this is the first or last loop iteration, which can be eliminated by the loop peeling optimization[2].

A.3.1.4 Weak-Crossing SIV Test

A pair of SIV subscripts is said to be weak-crossing SIV if they take the form $ai+c_1$ and $-ai+c_2$ where i is an induc-

[2]Loop peeling is an optimization where the first or last loop iteration is split and executed separately from the rest of the loop. loop bound are adjusted accordingly.

tion variable, a_1, a_2 are constants, and c_1 and c_2 are loop invariants. This test is capable of proving independence, failing that it can someimtes be refined for direction.

This test relies on the fact that $a_2 = -a_1 = a$, since symmetry helps simplify the analysis. This suggests that as the loop index progresses, the subscript pair are moving away from a specific point at the same rate. This point is the crossing point between the subscript pair. To locate the crossing point, i' is set to i, and a_2 is substituted with $-a_1$ which results into:

$$a_1 i + c_1 = -a_1 i + c_2 \tag{A.6}$$

Solving for i results into:

$$i = \frac{c_2 - c_1}{2a_1} \tag{A.7}$$

Determining whether dependence exists amounts to checking whether the crossing point is within the loop bounds, and has a value that is either an integer or non-integer multiple of $\frac{1}{2}$. Since the subscript pair are moving with the same rate away from the crossing point, a point that is not in the middle between two integers, then the pair cannot intersect at an integer, thus proving independence.

Weak-crossing SIV dependences can be eliminated by loop splitting optimization[3].

A.3.1.5 Exact SIV Test

As discussed earlier on in this section, the exact SIV test is used for array subscript pairs that has the form $a_1 i + c_1$ and $a_2 i' + c_2$ where i is an induction variable, a_1, a_2 are constants, and c_1 and c_2 are loop invariants. This test is slower than any of the specialized SIV tests (*i.e.*, strong, weak-zero, weak-crossing), thus it is recommended to use them instead. These tests are also better with symbolics and strong SIV test can even calculate distances as discussed in A.3.1.1. This test can prove dependence, failing that it can refine for direction.

Starting from the intersection point

$$a_1 i + c_1 = a_2 i' + c_2 \tag{A.8}$$

This test requires looking for all possible solutions for the equation:

$$a_1 i - a_2 i' = c_2 - c_1 \tag{A.9}$$

This equation system has a solution if and only if $\frac{gcd(a_1,a_2)}{c_2-c_1}$ is an integer value.

[3]Loop Splitting is a compiler optimization where dependences within a loop are eliminated by breaking the loop into multiple loops with same bodies and different bounds covering the entire index range of the original loop

A.4 Multiple Index Variable Test

ZIV and SIV subscripts are relatively simple linear mappings from Z to Z, when restricted to linear functions of loop induction variables, and where Z is the set of natural numbers. However, being linear mappings from Z^m to Z, (m is the number of subscript's loop induction variables), MIV subscripts are more complicated. As such, in order to accurately determine dependences, MIV subscript pairs require use of more advanced mathematical methods. This section explains the general dependence equations used in the various MIV dependence tests.

MIV tests are useful for loops that take the general form:

```
for $i_1$ = $L_1$ to $U_1$ do
 for $i_2$ = $L_2$ to $U_2$ do
  ...
   for $i_n$ = $L_n$ to $U_n$ do
     A[$f(i_1, i_2, ..., i_n)$] = ...
    ... = A[$g(i_1, i_2, ..., i_n)$]
   EndFor
  ...
 EndFor
EndFor
```

To determine if a dependence exists and has a direction vector $D = (D_1, D_2, ..., D_n)$ is equivalent to determining if an integer solution to the following equation system exists:

$$f(v_1, v_2, ..., v_n) = g(u_1, u_2, ..., u_n) \quad \text{(A.10)}$$

where v, u are defined by

$$L_i \leq v_i, u_i \leq U_i, \forall i, 1 \leq i \leq n \tag{A.11}$$

and direction vector added restriction

$$v_i D_i y_i \forall i, 1 \leq i \leq n, and D_i \in \{<, =, >\} \tag{A.12}$$

The equation has a solution if

$$h(v_1, v_2, ..., v_n, u_1, u_2, ..., u_n) = f(v_1, v_2, ..., v_n) - g(u_1, u_2, ..., u_n) = 0 \tag{A.13}$$

has an integer solution inside the region defined by A.11 and A.12. However, exactly solving this equation in restricted space is very difficult, which means finding an approximate solution with acceptable precision for compilers. A simplification to the requirement for the solution to be integer, which will make the solution space continuous. The absence of a real number solution indicates that the equation cannot have an integer solution, which proves that no dependence exists. This section will look only at affine functions of the form:

$$f(v_1, v_2, ..., v_n) = a_0 + a_1 v_1 + a_2 v_2 + ... + a_n v_n \tag{A.14}$$

$$g(u_1, u_2, ..., u_n) = b_0 + b_1 u_1 + b_2 u_2 + ... + b_n u_n \tag{A.15}$$

And to solve the dependence problem means finding solutions in the region defined by A.11 and A.12 to the linear equation:

$$a_0 - b_0 + a_1 v_1 - b_1 u_1 + a_2 v_2 - b_2 u_2 + \ldots + a_n v_n - b_n u_n = 0 \tag{A.16}$$

There are two important dependence tests used for MIV subscripts, the *GCD* test, and the *Benarjee Inequality* test. The Benarjee Inequality test will not be discussed further.

A.4.1 The Greatest Common Denominator Test

When equation A.16 terms are rearranged, it yields:

$$a_1 v_1 - b_1 u_1 + a_2 v_2 - b_2 u_2 + \ldots + a_n v_n - b_n u_n = b_0 - a_0 \tag{A.17}$$

This equation has the linear Diophantine equation's standard form. A theorem about these equations is:

Theorem A.4.1 *A linear Diophantine equation of the form:*

$$a_1 x_1 - b_1 y_1 + a_2 x_2 - b_2 y_2 + \ldots + a_n x_n - b_n y_n = b_0 - a_0$$

has a solution if and only if the $GCD(a_1, a_2, \ldots, a_n, b_1, b_2, \ldots, b_n)$ *divides* $b_0 - a_0$

When this theorem is applied to equation A.17, independence is proved if the GCD of all coefficients a, b does not divide the constant term $b_0 - a_0$.

Testing for a specific direction vector **D** $= (D_1, D_2, ...D_n)$, with at least one direction being $' =' $, tightens the condition of solution being within the region defined by A.11 and A.12 since $v_i = u_i$. Substituting in equation A.17 yields:

$$a_1 v_1 - b_1 u_1 + a_2 v_2 - b_2 u_2 + ... + (a_i - b_i) v_i + ... + a_n v_n - b_n u_n = b_0 - a_0 \tag{A.18}$$

In this case the GCD, should include the term $(a_i - b_i)$, and exclude the terms a_i, b_i, making the process more precise. This also indicates the that strong SIV test discussed in subsection A.3.1.1 is a special case of this test.

A.4.2 Restricted Double Index Variable

RDIV is a special case of MIV subscripts, in which a pair of subscripts take the form $a_1 i + c_1$ and $a_2 j + c_2$ where a_1, a_2 are constants, c_1 and c_2 are loop invariants, and i, j are induction variables. This test is an easy adaptation of the exact SIV test.

A.5 Testing in Coupled Groups

Coupled subscripts are any two subscripts containing the same index variable. Recognition of coupling is important since in Multidimensional array references, coupled subscripts can cause imprecisions during dependence testing. When tests used for separable subscripts are used on each subscript of a couple group, and the test proves independence then no dependency exists. However, testing subscript-by-subscript can also indicate false dependences.

A better test would be to test each subscript separately, intersecting the resulting direction vectors sets [52], which permits efficient testing, while in some cases, it conservatively approximates the set of directions within a coupled group, yielding non-existent direction vectors. A very effective strategy for addressing this is by intersecting distance rather than direction vectors. There are many other methods for multiple subscript tests suggested in research such as [53, 54, 55].

A.5.1 Delta Test

This is an intuitive, exact and efficient test used in testing common coupled subscripts. The main idea behind it is based on the intuition of distance vector intersection. Given that SIV subscripts are the most common in practice and that the SIV tests are easy to perform, and gives exact results in most cases, the information obtained by using them may be used to make it easier to test other subscripts within same group.

In the delta test, each SIV subscript in the coupled group is examined in order to produce constraints used with other subscripts of the same group, which usually results in simplification producing a precise set of directional vector.

The Delta test gets its name from representing the distance between memory references informally with a ΔI. As such, the index variable in the source memory references is assumed to be a specific value I, and the sink memory reference is assumed to have the same value as the source with an added distance $I + \Delta I$.

The Delta test can detect independence if any of the SIV tests involved detects dependence. If not, then it converts all SIV subscripts into constraints, which in turn are used with MIV subscripts whenever possible. This process is repeated until no more constraints can be found. These constraints are used to simplify RDIV subscripts. All remaining MIV subscripts are then tested, and the results are intersected with other existing constraints.

A.6 Symbolic Tests

This test is intended for cases where the subscript pair take the form $a_1 i + c_1$ and $a_2 j + c_2$ where a_1, a_2 are constants, c_1 and c_2 are loop invariants, and i, j are induction variables, with bounds $L_1 \leq i \leq U_1$ and $L_2 \leq j \leq U_2$.

For a dependence to exist the subscript pair must point to the same memory location, within the bounds of both index variable i, j. As such

$$a_1 i + c_1 = a_2 j + c_2 \tag{A.19}$$

$$a_1 i - a_2 j = c_2 - c_1 \tag{A.20}$$

Testing for dependence require computing $c_2 - c_1$ and making sure it is in the range of the maximum and minimum possible values of $a_1 \times i - a_2 \times j$. Based on the signs of a_1, a_2 there are four possibilities:

- if $a_1 \geq 0$ and $a_2 \geq 0$
$$a_1L_1 - a_2U_2 \leq c_2 - c_1 \leq a_1U_1 - a_2L_2 \tag{A.21}$$
- if $a_1 \geq 0$ and $a_2 \leq 0$
$$a_1L_1 - a_2L_2 \leq c_2 - c_1 \leq a_1U_1 - a_2U_2 \tag{A.22}$$
- if $a_1 \leq 0$ and $a_2 \geq 0$
$$a_1U_1 - a_2U_2 \leq c_2 - c_1 \leq a_1L_1 - a_2L_2 \tag{A.23}$$
- if $a_1 \leq 0$ and $a_2 \leq 0$
$$a_1U_1 - a_2L_1 \leq c_2 - c_1 \leq a_1L_1 - a_2U_2 \tag{A.24}$$

$c_2 - c_1$ does not satisfy the inequality case from above that applies according to the signs of a_1, a_2, then a dependence cannot exist.

This test can handle some RDIV cases and is only capable of disproving dependence, and it cannot calculate any distance or direction information. It can be useful in case of memory reference pairs that are in two different nested loops of the same level. Furthermore, These equations can be used to determine dependence with the existence of symbolic values for $c_1, c_2, L_1, L_2, U_1, U_2$. It better serves as a backup for the RDIV test.

It can also be used for dependencies in the same loop after setting $L_2 = L_1$ and $U_2 = U_1$. This test can handle some SIV cases when i, j are equal (*i.e.*, same variable). substituting in the above equations will result in the following two inequalities:

- if a_1 and a_2 have the same sign

$$a_1L_1 - a_2U_1 \leq |c_2 - c_1| \leq a_1U_1 - a_2L_1 \tag{A.25}$$

- if a_1 and a_2 have different signs

$$L_1 \leq |\frac{c_2 - c_1}{(a_1 - a_2)}| \leq U_1 \tag{A.26}$$

Bibliography

[1] D. Culler et al. “LogP: towards a realistic model of parallel computation”. In: *SIGPLAN Not.* 28.7 (1993), pp. 1–12.

[2] U. Vishkin. “Using simple abstraction to guide the reinvention of computing for parallelism”. In: *CACM* 54,1 (2011), pp. 75–85.

[3] X. Wen and U. Vishkin. “FPGA-based prototype of a PRAM-on-chip processor”. In: *Proc. ACM Computing Frontiers*. 2008.

[4] D. Naishlos et al. “Towards a first vertical prototyping of an extremely fine-grained parallel programming approach”. In: *Proc. 13th annu. ACM symp. on Parallel algorithms and architectures (SPAA)*. 2001.

[5] U. Vishkin et al. “Explicit multi-threading (XMT) bridging models for instruction parallelism (extended abstract)”. In: *Proc. 10th annu. ACM symp. on Parallel algorithms and architectures (SPAA)*. 1998.

[6] U. Vishkin. “Prefix sums & an application thereoff.” In: *U.S. Patent* 6 542 918 (2003).

[7] A. Balkan et al. “Layout-Accurate Design and Implementation of a High-Throughput Interconnection Network for Single-Chip Parallel Processing”. In: Los Alamitos, CA, USA: IEEE Computer Society, 2007.

[8] A. Balkan, G. Qu, and U. Vishkin. “An Area-Efficient High-Throughput Hybrid Interconnection Network for Single-chip Parallel Processing”. In: *45th Design Automation Conference*. Anaheim, CA, June 2008.

[9] U. Vishkin. “Spawn-join instruction set architecture for providing explicit multi-threading (XMT)”. In: *U.S. Patent* 6 463 527 (2002).

[10] G. C. Caragea et al. “Resource-Aware Compiler Prefetching for Many-Cores”. In: *International Symposium on Parallel and Distributed Computing*. 2010.

[11] X. Wen and U. Vishkin. “PRAM-on-chip: first commitment to silicon”. In: *SPAA ’07: Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures*. San Diego, California, USA: ACM Press, 2007, pp. 301–302.

[12] A. O. Balkan and U. Vishkin. *Programmer’s Manual for XMTC Language, XMTC Compiler and XMT Simulator*. Tech. rep. UMIACS-TR-2005-45. University of Maryland Institute for Advanced Computer Studies (UMIACS), 2006. URL: http://www.umiacs.umd.edu/users/vishkin/XMT/manual4xmtc1out-of2.pdf.

[13] A. Gottlieb et al. “The NYU Ultracomputer: designing a MIMD, shared-memory parallel machine (Extended Abstract)”. In: *ISCA ’82: Proceedings of the 9th annual symposium on Computer Architecture*. Austin, Texas, United States: IEEE Computer Society Press, 1982, pp. 27–42.

[14] G. C. C. Bryant C. Lee Uzi Vishkin. *Handbook of Parallel Computing: Models, Algorithms and Applications, Chapter Models for Advancing PRAM and Other Algorithms into Parallel Programs for a PRAM-On-Chip Platform*. 1st ed.

[15] J. A. Edwards and U. Vishkin. “Better Speedups Using Simpler Parallel Programming for Graph Connectivity and Biconnectivity”. In: *Proceedings of the 2012 International Workshop on Programming Models and Applications for Multicores and Manycores (PMAM)*. ACM, 2012, pp. 103–114.

[16] J. A. Edwards and U. Vishkin. “Brief Announcement: Truly Parallel Burrows-wheeler Compression and Decompression”. In: *Proceedings of the Twenty-fifth Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. ACM, 2013, pp. 93–96.

[17] Z. He and B. Hong. “Dynamically tuned push-relabel algorithm for the maximum flow problem on cpu-gpu-hybrid platforms”. In: *Proc. 24th IEEE Int. Parallel and Distributed Processing Symp.* 2010.

[18] G. Caragea and U. Vishkin. “Better speedups for parallel max-flow”. In: *Proc. 23rd ACM Symp. on Parallel Algorithms and Architectures (SPAA)*. 2011.

[19] J. A. Edwards and U. Vishkin. “Parallel algorithms for Burrows-Wheeler compression and decompression”. In: *Theor. Comput. Sci.* 525 (2014), pp. 10–22.

[20] A. Saybasili et al. “Highly parallel multi-dimensional fast Fourier transform on fine- and course-grained many-core approaches”. In: *In Proc. 21st Conference on Parallel and Distributed Computing Systems (PDCS)*. Cambridge, MA, Nov. 2009.

[21] P. Gu and U. Vishkin. “Case study of gate-level logic simulation on an extremely fine-grained chip multiprocessor”. In: *J. Embedded Comp.* 2 (2006), pp. 181–190.

[22] S. Torbert et al. “Is Teaching Parallel Algorithmic Thinking to High School Students Possible?: One Teacher’s Experience”. In: *Proceedings of the 41st ACM Technical Symposium on Computer Science Education*. SIGCSE ’10. Milwaukee, Wisconsin, USA, 2010, pp. 290–294.

[23] L. Hochstein et al. “A Pilot Study to Compare Programming Effort for Two Parallel Programming Models”. In: *J. Syst. Softw.* 81.11 (Nov. 2008), pp. 1920–1930.

[24] D. Padua, U. Vishkin, and J. Carver. “Joint UIUC/UMD parallel algorithms/programming course”. In: *Proc. 1st NSF/TCPP Workshop on Parallel and Distributed Computing Education (EduPar-11)*. in conjunction with 25th IEEE Int. Parallel and Distributed Processing Symp., 2011.

[25] J. JaJa. *An Introduction to Parallel Algorithms*. Addison-Wesley Publishing Company, 1992.

[26] J. Keller, C. Kessler, and J. Traeff. *Practical PRAM Programming*. Wiley-Interscience, 2001.

[27] U. Vishkin. “Thinking in Parallel: Some Basic Data-Parallel Algorithms and Techniques - Course class notes”. URL: http://www.umiacs.umd.edu/users/vishkin/PUBLICATIONS/classnotes.pdf.

[28] R. E. Tarjan and U. Vishkin. “An Efficient Parallel Biconnectivity Algorithm”. In: *SIAM J. Comput.* 14.4 (1985), pp. 862–874.

[29] U. Vishkin, G. Caragea, and B. Lee. “Models for Advancing PRAM and Other Algorithms into Parallel Programs for a PRAM-On-Chip Platform. In Handbook on Parallel Computing (Editors: S. Rajasekaran, J. Reif)”. In: Chapman and Hall/CRC Press, 2008.

[30] Y. Shiloach and U. Vishkin. “An O$(n^2 \log n)$ parallel max-flow algorithm”. In: *J. Algorithms* 3 (1982), pp. 128–146.

[31] *The Chapel Parallel Programming Language*. URL: http://chapel.cray.com/.

[32] *Project Fortress*. URL: http://projectfortress.java.net/.

[33] *X10: Performance and Productivity at Scale*. URL: http://x10-lang.org/.

[34] W. Ching and D. Ju. “Execution of automatically parallelized API programs on RP3”. In: *IBM J. of research and Development* 35 (5/6 1991), pp. 767–778.

[35] G. E. Blelloch. *Vector Models for Data-Parallel Computing*. MIT Press, 1990.

[36] G. Blelloch and J. Greiner. “A Provable Time and Space Efficient Implementation of NESL”. In: *ACM SIGPLAN Int. Conf. on Functional Programming*. 1996.

[37] G. E. Blelloch. “Programming parallel algorithms”. In: *Commun. ACM* 39 (3 Mar. 1996), pp. 85–97. ISSN: 0001-0782.

[38] W. D. Hillis and G. L. Steele, Jr. “Data Parallel Algorithms”. In: *Commun. ACM* 29.12 (1986), pp. 1170–1183.

[39] K. M. Chandy and J. Misra. *Parallel Program Design: A Foundation*. Addison Wesley, 1988.

[40] Arvind, R. S. Nikhil, and K. K. Pingali. “I-Structures: data structures for parallel computing”. In: *ACM Trans. on Programming Languages and Syst.* 11.4 (Oct. 1989), pp. 598–632.

[41] J. T. Feo, D. C. Cann, and R. R. Oldehoeft. “A Report on the Sisal Language Project”. In: *J. of Parallel and Distributed Computing* 10.4 (Dec. 1990), pp. 349–366.

[42] P. Mills et al. “Prototyping parallel and distributed programs in Proteus”. In: *Symp. Parallel and Distributed Processing 1991*. IEEE Comput. Soc.

[43] M. Frigo, C. E. Leiserson, and K. H. Randall. “The implementation of the Cilk-5 multithreaded language”. In: *PLDI ’98: Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*. Montreal, Quebec, Canada: ACM Press, 1998, pp. 212–223. ISBN: 0-89791-987-4. DOI: http://doi.acm.org/10.1145/277650.277725.

[44] *The MIT Cilk home page: http://supertech.csail.mit.edu/cilk/*.

[45] A. Tzannes et al. “Lazy binary-splitting: a run-time adaptive work-stealing scheduler”. In: *Proc. 15th ACM SIGPLAN symp. on Principles and practice of parallel programming (PPOPP)*. 2010.

[46] M. Frigo et al. “Reducers and other Cilk++ hyperobjects”. In: *Proc. 21st Annu. ACM Symp. on Parallelism in Algorithms and Architectures (SPAA)*. 2009.

[47] T. Cormen et al. *Introduction to Algorithms, 3rd Ed.* MIT Press, 2009.

[48] C. S. Michael Joseph Wolfe and L. Ortega. *High Performance Compilers for Parallel Computing*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995. ISBN: 0805327304.

[49] U. K. Banerjee. *Dependence Analysis for Supercomputing*. Norwell, MA, USA: Kluwer Academic Publishers, 1988. ISBN: 0898382890.

[50] W. L. Cohagen. “Vector optimization for the asc”. In: 1973.

[51] M. J. Wolfe. *High Performance Compilers for Parallel Computing*. Ed. by C. Shanklin and L. Ortega. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995. ISBN: 0805327304.

[52] M. J. Wolfe. “Optimizing Supercompilers for Supercomputers”. AAI8303027. PhD thesis. Champaign, IL, USA, 1982.

[53] Z. Li, P.-C. Yew, and C.-Q. Zhu. “Data Dependence Analysis on Multi-dimensional Array References”. In: *Proceedings of the 3rd International Conference on Supercomputing*. ICS ’89. Crete, Greece: ACM, 1989, pp. 215–224.

[54] D. R. Wallace. “Dependence of Multi-dimensional Array References”. In: *Proceedings of the 2Nd International Conference on Supercomputing*. ICS ’88. St. Malo, France: ACM, 1988, pp. 418–428.

[55] M. Wolfe and C. W. Tseng. “The Power Test for Data Dependence”. In: *IEEE Trans. Parallel Distrib. Syst.* 3.5 (Sept. 1992), pp. 591–601.