COMPILATION TECHNIQUES AND LANGUAGE SUPPORT TO FACILITATE DEPENDENCE-DRIVEN COMPUTATION

BY

ALBERT SIDELNIK

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2013

Urbana, Illinois

Doctoral Committee:

Professor David A. Padua, Chair, Co-Director of Research
Research Assistant Professor María J. Garzarán, Co-Director of Research
Doctor Bradford L. Chamberlain, Cray Inc.
Professor Wen-mei W. Hwu
Professor Sanjay J. Patel
Professor Keshav Pingali, University of Texas, Austin

Abstract

As the demand increases for high performance and power efficiency in modern computer runtime systems and architectures, programmers are left with the daunting challenge of fully exploiting these systems for efficiency, high-level expressibility, and portability across different computing architectures. Emerging programming models such as the task-based runtime StarPU and many-core architectures such as GPUs force programmers into choosing either low-level programming languages or putting complete faith in the compiler. As has been previously studied in extensive detail, both development approaches have their own respective trade-offs.

The goal of this thesis is to help make parallel programming easier. It addresses these challenges by providing new compilation techniques for high-level programming languages that conform to commonly-accepted paradigms in order to leverage these emerging runtime systems and architectures. In particular, this dissertation makes several contributions to these challenges by leveraging the high-level programming language Chapel in order to efficiently map computation and data onto both the task-based runtime system StarPU and onto GPU-based accelerators. Different loop-based parallel programs and experiments are evaluated in order to measure the effectiveness of the proposed compiler algorithms and their optimizations, while also providing programmability metrics when leveraging high-level languages. In order to exploit additional performance when mapping onto shared memory systems, this thesis proposes a set of compiler and runtime-based heuristics that determine the profitable processor tile shapes and sizes when mapping multiply-nested parallel loops.

Finally, a new benchmark-suite named P-Ray is presented. This is used to provide machine characteristics in a portable manner that can be used by either a compiler, an auto-tuning framework, or the programmer when optimizing their applications.

For the two best companions, Yuki and Bandita.

Acknowledgements

For the most part, graduate school is a roller coaster of emotions. There are moments where nothing works as you go down the rabbit hole of pursuing a research topic, and then there are other moments where everything just "clicks". The work in this thesis is a culmination of lots of sweat, plenty of expletives thrown at compilers and operating systems, unwanted gray hairs, and many late nights spent in the lab and at various coffee shops. At the end of the day, I am glad to have gone through this journey.

I would like to first start by thanking my advisors David Padua and María Garzarán. If it were not for their support, guidance, and patience, this work would not exist.

I would also like to acknowledge and thank my thesis committee members Brad Chamberlain, Wen-mei Hwu, Sanjay Patel, and Keshav Pingali for their valuable comments. In particular, I would like to give a special thanks to Brad Chamberlain. His feedback during the different stages of my graduate school experience has been extremely helpful.

I am fortunate to have the experience of working and collaborating with some of the sharpest researchers in the field, including Gheorghe Almasi, Bryan Catanzaro, Michael Garland, and Jose Moreira. I have learned an immense amount from them.

It is also important that I give a special thanks to Mary Beth Kelly, Megan Osfar, and Sherry Unkraut. If it were not for them, I would be even more confused than I typically am.

Lastly, I want to thank my friends and family, especially my parents for supporting me in all of my various adventures.

Table of Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

1.1 Overview

The ability to fully leverage parallelism, both explicitly with low-level programming languages, and implicitly through compilers or libraries, has always been a complex and multifaceted challenge. Even with the valiant efforts of researchers over the past five decades, there is still much to learn. Until the beginning of the 2000s, computer architects were able to dodge this problem by focusing their efforts on single-core performance. This meant increased clock speeds, larger memory hierarchies, and deeper pipeline architectures that are able to extract more instruction level parallelism. This has allowed users to get their *free performance lunch* [3]. However, this has led to its own set of challenges due to the physical demands of increased CPU power, more heat dissipation, and the bandwidth limiting disparity between processor and memory speeds known as the *memory wall* [4]. While these problems are still hurdles today, architects have been able to temporarily alleviate their impact by increasing the number of processor cores on the same die, developing even deeper memory hierarchies, and providing larger vector units and other specialized functional units. As a result of this, users are now forced to deal with an ever-increasing amount of parallelism.

Additionally, homogeneous multicomputers are no longer the only form of machine architecture used. Heterogeneous parallelism is pervasive in our culture, from consumer devices containing multiple cores and GPUs such as modern smartphones and automotive electronics to traditional large-scale supercomputers. It is now common to see large-scale heterogeneous architectures consisting of GPU accelerators and multiprocessors with large core counts and vector units [5]. For example, as of November 2012, the number one computer on the Top500 list is Oak Ridge National Laboratory's Titan [6]. This machine is configured with 299,008 AMD Opteron cores and 18,688 NVIDIA Tesla GPUs.

As mentioned earlier, the problems of increased demand in power, heat dissipation, and the memory wall still exist in current architectures. Even beyond the idea of performance portability, there is now a loss of functional portability when compiling and executing the same program across different classes of architectures. For example, having an application that has been hand-tuned for a NVIDIA-based GPU and retargeting it to execute on a multi-core system is not trivial [7, 8, 9]. With these new dimensions of complexity, one major software challenge is having the ability to efficiently exploit all of the hardware parallelism without sacrificing programmer productivity and, at the same time, striving to be portable. One approach that some have taken to address these challenges is to introduce new runtime systems and execution models that are dependence-based [10, 11, 12, 13] (sometimes called *codelets* [14, 15]), new declarative programming languages [16, 17, 18], and libraries [19]. While these systems have shown great promise in terms of application performance, relying on new programming models would force applications written in commonly used imperative programming languages to be rewritten into a declarative programming paradigm. This might not be economical for applications with a large existing code base or when the application is no longer actively maintained.

In addition to increasing processing power and memory sizes, the length of a program's source code and the amount of data it occupies is also increasing. In many cases, this is

making applications more complex, typically requiring domain experts to develop them. In order for these domain experts to have a remote chance of leveraging such complex systems, high-level programming languages (with the help of more intelligent compilers) need to bridge this gap. Moreover, if the HPC community plans to tackle the challenge of reaching exascale levels by the 2020 time frame [20], these complexities need to be addressed with urgency.

This thesis presents methods and ideas to help address the challenges of making parallel programming easier. This will be addressed at four levels: at the programming language level, in the compiler, dynamically in the runtime, and offline during compilation using micro-benchmarks. As a foundation, this work will start with Chapel [21], a high-level language designed from the ground up to deal with parallelism. Using Chapel and leveraging its support for first-class user-defined distributions [22], new compiler techniques are developed to support the mapping of multiply-nested parallel loops onto dependence-driven execution models, such as StarPU or ETI Swarm. In addition to these dependence-based execution models, it will be shown how the same set of data-parallel loops can now also be retargeted onto GPU architectures without any modifications to the loops. The reverse of this will also be shown, where code that has been hand-tuned for a specific platform (such as a GPU) can be mapped back onto a traditional multi-core platform.

To deal with the problem of exploiting the available parallelism, machine cost models will be used to drive new heuristics that will determine the correct amount of parallelism necessary to maximize performance. This is important because there are situations where using all of the parallelism can be detrimental to the performance of an algorithm due to an excess of overhead from the runtime or other libraries. Part of the heuristics will be based on a cost model of the execution model and the underlying memory architecture.

To aid programmers in extracting additional performance for parallel programs, a new micro-benchmark suite will be presented. This benchmark suite is based on simple and portable micro-benchmarks that determine machine characteristics specific to multi-cores

and the memory hierarchy that can be used as parameters to optimize parallel programs by a compiler, an expert programmer, or an auto-tuner.

1.2 Contributions

In order to tackle the challenges described in Section 1.1, this thesis makes the following contributions:

- **Generation of Task Graphs from Explicitly Parallel Loops**

  With the introduction of *agglomerated flow graphs*, and by using interval analysis [1] as a basis, new compilation techniques are developed to hierarchically partition programs around explicitly single and multiply-nested data-parallel loops and other control-dependent statements. This partitioning forms a new intermediate representation based on task-dependence graphs which are composed of executable nodes connected by dependence constraints. The task graphs can then be mapped and scheduled onto a dependence-driven runtime system. This work also presents new algorithms to deal with data placement and communication between the tasks in a task-dependence graph. Once the task-graph form is fully instantiated, code generation is performed to generate the necessary instructions for correct execution. Experimental results will show that only a minimal amount of overhead has been added, with little to no impact on performance scalability.

- **Language Extensions to Express Arbitrary Task Execution Order**

  New language extensions are developed that express the explicit order of execution among different program components (e.g. statements, loops, blocks, procedures). By leveraging the loop partitioning transformations being produced, these language extensions can now provide enough information that a task-dependence graph can be

formed and, thus, mapped onto any given parallel architecture. Additionally, the compiler can leverage these language extensions as a complement (or replacement) for traditional data-dependence analysis to help it determine the legality of loop-based transformations.

- **Methods to Determine Tile Dimensions for Coarse-grain Parallelism**

  A set of compiler and runtime-based heuristics are presented that will be used to determine the appropriate number of processors to use and what the processor multi-dimensional shapes need to be. These heuristics are based on a cost model that is derived from offline micro-benchmark measurements to find certain machine characteristics and overheads in driving the correct processor configurations. Rather than just selecting the maximum amount of parallelism supported on the system, the algorithm chooses the processor configuration that minimizes both the impact of a static load-imbalance and the memory footprint of data used in the loops. The effectiveness of this approach is then compared against other commonly-applied compiler optimizations. Part of this work will also introduce a new loop-transformation that combines tiling with traditional loop-coalescing in order to decrease the synchronization overhead from parallel loops.

- **A Portable Mapping of Data and Computations Between GPU and Traditional Multi-core Architectures**

  New compiler transformations that increase programmer productivity are presented. By leveraging the programming language Chapel, a single source code implementation can be used to target not only conventional multiprocessors, but also GPU architectures. Rather than resorting to different parallel libraries or annotations for a given parallel platform, this contribution will only need to leverage Chapel's support for user-defined distributions and its support for data-parallel loops. Experimental results

from the Parboil benchmark suite are presented and demonstrate that codes written in Chapel achieve performance that are comparable to the original versions implemented in CUDA on both GPUs and multi-core platforms.

- **Portable Micro-benchmark Suite to Determine Machine Characteristics for Parallelism and Locality**

  A suite of micro-benchmarks named P-Ray is presented. These micro-benchmarks provide a way of introspecting hardware characteristics specific to multi-core architectures. Such characteristics include the number of cores that share the L2 cache, the different processors interconnection topologies, and the bandwidth to memory for multi-cores. The presented experiments show that, for several different architectures tested, both desktop and server, P-Ray generates accurate results. By utilizing some of the machine characteristics that were discovered with these micro-benchmarks, a more accurate cost model can be constructed.

1.3 Thesis Organization

For each core contribution of this thesis, the problem will be introduced, along with any necessary background that has not been described yet. In some cases, a set of examples will be provided. Next, the main contribution and all of its necessary algorithms are presented. From there, different experiments will be evaluated, followed by a discussion of the current limitations and relevant future work to that section.

The remainder of the dissertation is structured into chapters as follows:

- **Chapter 2** : Presents a high-level background and overview of concepts necessary to understand the remainder of the dissertation. Specifically, it will give a short overview of components from compiler theory and parallel programming, the Chapel programming language, and dependence-driven execution models.

- **Chapter 3** : Describes the compilation techniques used to translate explicitly-parallel loops into a suitable form so that they can be mapped onto dependence-driven execution models.

- **Chapter 4** : Presents a new heuristic that will be used in determining the ideal processor configuration when targeting different runtime systems. Additionally, a new loop transformation that combines loop tiling and loop coalescing will be provided.

- **Chapter 5** : Discusses the techniques and optimizations used to map data-parallel codes onto heterogeneous architectures. This will also cover the algorithms used to "go backwards" and map GPU-centric code back onto a traditional multi-core platform.

- **Chapter 6** : Present the portable micro-benchmark suite named P-Ray and associated examples.

- **Chapter 7** : Propose future work.

- **Chapter 8** : Final thoughts and conclusions.

Chapter 2

Background

This chapter discusses topics that are important in understanding the remainder of the dissertation. First, this chapter presents an overview and defines some of the concepts in this thesis from the compiler and parallel computing literature. Second, a brief overview of the Chapel programming language will be presented. The material presented will be enough for the reader to understand the main concepts of the language related to data-parallelism. Finally, an overview of dependence-driven execution models will be discussed, which includes an overview of the StarPU runtime system.

2.1 General Compiler Preliminaries

This section describes the necessary high-level ideas relevant to compilers used in this thesis. Most of the topics discussed in this section assume the use of an imperative programming language, primarily because the input language assumed in this is imperative.

2.1.1 Control Flow Analysis

Basic Block (BB)

A *basic block* is set of instructions in a linear sequence, where the only entry point is the first statement in the sequence, and the only exit point being the last statement in the sequence. Aho et al. present a simple algorithm to partition a program into a set of basic blocks [23].

Control Flow Graph(CFG)

A *control flow graph* is a program representation where each node in the graph is a basic block, and the edges connecting the nodes represent a possible execution order (e.g. conditional-branch). More specifically, a CFG is the triple $G = (V, E, s)$, where (V, E) make up a directed graph, V is the set of all basic blocks, E is the set of edges connecting all nodes in V, and $s \in V$ is the entry point of the program. In Figure 2.1, the leftmost graph represents a control flow graph, where each numbered node represents a basic block.

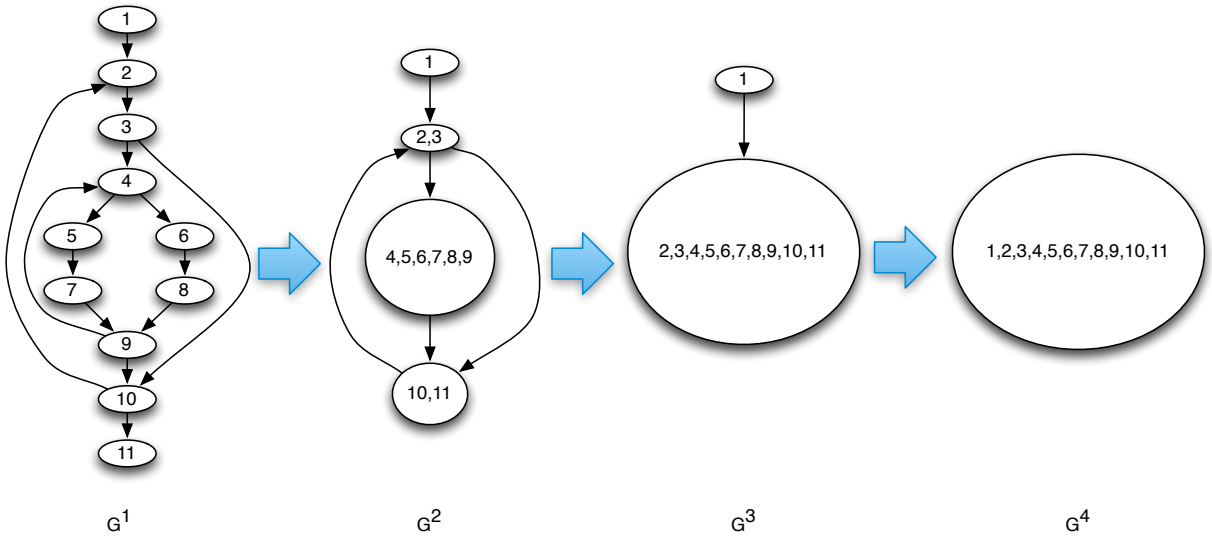


Figure 2.1: Interval Analysis Applied to a CFG

Interval Analysis

One way of finding structure in a control flow graph is to identify "hierarchies" of loops by applying *interval analysis* [1]. Intervals analysis is the technique used to partition a reducible [24] flow graph into disjoint regions named *intervals*. Informally, an interval is a *natural loop* [23] with a set of nodes forming an acyclic subgraph between the header node and a back edge that completes the loop. Let a flow graph $G = (V, E, s)$, and $I_G(h)$ be an interval of G with entry node h. Given an entry node h, an interval has the following properties [2]:

- Entry node h is in $I_G(h)$.
- Every edge that connects to $I_G(h)$ connects *only* through the entry node h.
- Entry node h dominates [25] all other nodes inside of $I_G(h)$.
- Every cycle inside of $I_G(h)$ includes the entry node h.
- Every node whose predecessors are in $I_G(h)$ is added to $I_G(h)$ until there are no such nodes.

Well-known algorithms [1, 2] for constructing an interval node and partitioning a control flow graph into an interval graph are presented in Algorithms 1 and 2. Algorithm 1 constructs a single interval node, while Algorithm 2 partitions the CFG into an interval graph.

To find the nested loop structure (and determine the reducibility) of the flow graph G, interval analysis can be applied successively to create a sequence of *derived graphs*. Starting with G^1 (the original flow graph), each successive pass of Algorithms 1 and 2 creates $\{G^2, G^3, \ldots, G^d\}$, where G^d is the *limit flow graph*. The limit flow graph is the last possible application of interval partitioning where only a single node remains. Starting with a node in G^{i+1} represents an interval in G^i. Once a limit flow graph is encountered, if it contains

Algorithm 1: Interval Node Construction

Input: h : node of graph G
Output: I(h) : interval node with header h
$I(h) \leftarrow h$;
while $\exists$ *a node* m *such that* $m \notin I(h) \wedge m \neq s \wedge$ *all predecessors of* m *are in* $I(h)$ **do**
  $I(h) \leftarrow I(h) \cup m$;

Algorithm 2: Interval Graph Partitioning

Input: Control flow graph G = (V, E, s)
Output: H : set of potential header nodes
Output: L : set of intervals
$H \leftarrow \{s\}$;
$L \leftarrow \emptyset$;
while $H \neq \emptyset$ **do**
  select and delete a node h from H;
  $I(h) \leftarrow$ computed using Algorithm 1;
  $L \leftarrow L \cup I(h)$;
  $H \leftarrow H \cup \{n \mid n$ has a predecessor in $I(h) \wedge n \notin L\}$;

Figure 2.2: Interval Analysis [1, 2]

a single node, it can be said that the graph G is reducible. Conceptually, by consecutively applying interval analysis, a loop nesting structure forms. This nesting structure starts to form from the innermost loop to the outermost per each successive step of the algorithm. This is a feature that will soon be exploited when partitioning nested parallel loops into separate tasks for the dependence-driven runtime.

Consider the graphs shown in Figure 2.1. The leftmost graph represents a control flow graph. Each numbered node in this graph is a basic block. After applying interval analysis to the CFG, an interval graph is formed as demonstrated by the second graph from the left. Applying interval analysis again forms the third graph from the left. Applying interval analysis one final time results in the limit flow graph as shown on the rightmost graph.

It has been shown that the likelihood of a graph being irreducible is rare [26, 27]. In cases where the graph is irreducible, *node splitting* [2] can be used to transform the graph into one that is reducible. For the remainder of this dissertation, it can be assumed that all

programs will be reducible, as Chapel is a structured programming language that does not support unstructured control idioms (e.g. `GOTO`, `setjmp()`, `longjmp()`, etc.). In the case of Chapel's intermediate representation: it does generate `GOTO`s internally, but none of these internal `GOTO`s cause a branch into a loop.

Interval analysis traditionally has had numerous uses, such as being an alternative for iterative dataflow analysis [28]. The work in Chapter 3 leverages interval analysis by partitioning an explicitly-parallel program into a hierarchical set of interval graphs (up to the limit flow graph) that consist of nested-parallel loops with control statements joining them together.

2.1.2 Dependences: Data and Control

Data Dependence

The ability to reason about what dependences exist between data has many uses in compilers. This includes the ability to determine the correctness of compiler transformations when the order of execution between statements has changed, to detect parallelism automatically, and to perform loop vectorization.

A data dependence is simply the relation between two statements in a program that have a partial execution order between them. Given two statements S_1 and S_2, there is a data dependence between S_1 and S_2 if all of the following conditions are true:

1. Both S_1 and S_2 access memory location M
2. Either S_1 or S_2 performs a write to memory location M
3. There is a path of execution from S_1 to S_2 or from S_2 to S_1

A data-dependence can be further classified into one of the following types:

1. Flow Dependence (or True Dependence)

   A statement S_2 has a flow dependence on statement S_1 (denoted as $S_1 \delta S_2$) when there is a path from S_1 to S_2, and S_1 stores a value at location M and S_2 reads from location M.

2. Anti-dependence

   A statement S_2 has an anti-dependence on statement S_1 (denoted as $S_1 \delta^{-1} S_2$) when there is a path from S_1 to S_2, and S_1 reads a value at location M and S_2 writes a value at M.

3. Output Dependence

   A statement S_2 has an output dependence on statement S_1 (denoted as ${}_1\delta^O S_2$) when S_1 writes a value at location M and then S_2 will write a value at M.

An important observation from these definitions is that if there are no dependences between the statements S_1 and S_2, both can be executed in parallel with respect to each other.

Dependences in Loops

The notion of data dependences can also applied in the presence of loops. Similar to how a data dependence was defined earlier, a statement S_1 can have a dependence on a statement S_2 even across different loop iterations. As a result of this, there are two classifications of dependences when related to loops: *loop-carried* and *loop-independent*.

1. Loop-Carried Dependence

   A loop-carried dependence exists when there is a dependence between two statements S_1 and S_2 across different iterations of a loop. This form of dependence limits the amount of parallelism that exists in the loop unless further transformations are applied [29].

2. Loop-Independent Dependence

   A loop-independent dependence exists when there is a dependence between two statements S_1 and S_2 only on the same iteration of a loop. Assuming that all dependences between the statements in a loop are independent, then the iterations of the loop are independent, thus each iteration of the loop can now be fully parallelized.

Control Dependence

Similar to data dependences specifying an ordering between statements, control dependences specify whether a statement has executed. A statement S is said to be control dependent on a conditional branch B_1, if by taking branch B_1 will always cause S to execute. If another branch B_2 of that conditional statement executes, then S does not need to execute.

Computing the control dependences for a graph has led to many advances such as new compiler optimizations [30], the automatic detection of parallelism [31], and computing SSA form [32].

2.1.3 Data-Parallelism

A major component of the work from this thesis depends on data-parallelism. Simply, data-parallelism is a form of parallelism where the same instructions or operation is applied to different sections of data in parallel, either on different processors or different devices within a processor. There are numerous advantages to choosing data-parallelism, including determinacy, and high-level expressiveness. Data-parallelism is fundamental to numerous programming models and languages, including OpenMP [33], NESL [34], ZPL [35], and CUDA [36]. While there are different high-level, data-parallel primitives, including parallel prefix [37], the main focus of this work will be on data-parallel loops.

Data-Parallel Loops

For the remainder of this dissertation, the reader can assume that a data-parallel loop is defined as a loop where the only dependences that occur inside of the loop are loop-independent (i.e. no loop-carried dependences). Each iteration of the loop can now execute independently of the other iterations, as if the loop were executed sequentially. Additionally, the iteration space of the loop is fixed upon entering the loop, and there is no way to prematurely exit the loop. The `parallel for` from OpenMP and the `forall` from Chapel are examples of this. However, in the case of Fortran 95's `forall` loop, it is different from the one defined here, because Fortran 95's `forall` loop can only contain assignments to arrays, as opposed to containing other forms of statements, even including other parallel loops nested inside of it.

In order to ensure correctness after a data-parallel loop has completed, an implicit *barrier synchronization* in the form of an implicit *join* is commonly placed between the end of the loop and the start of the next statement in the program. A join is a synchronization construct that forces the worker threads in a group to stop at the barrier point until all of the worker threads of this group reach the synchronization point. Once all of the threads reach the join, execution for the master thread can resume. When dealing with more dynamic forms of parallelism, there have been improvements to the concept of barriers, including support for *clocks* [38] or *phasers* [39].

The `doacross` [40, 41] is a special form of a parallel loop where the parallelism is constrained due to forward loop-carried dependences across iterations of the loop. In this case, a `doacross` loop's iterations are scheduled onto threads and executed in a pipelined order, which allow portions of the loop body to be overlapped in execution. In order to ensure program correctness, synchronization primitives (e.g. `signal()` and `wait()`) are commonly used to enforce the correct execution order.

A situation can also exist where data-parallel loops are nested inside of data-parallel

loops. Nested parallelism occurs in numerous situations. For example, when operating on a multi-dimensional array where the operations are parallel along all dimensions. There can also be a noticeable amount of overhead as a result of inner parallelism, including overheads from task and barrier creation and task destruction. If the nested-parallel loop bounds are statically known, and if the parallel loops are perfectly-nested, techniques such as loop coalescing [42] can be performed. Given a multi-dimensional loop L that is perfectly-nested and data-parallel, and $L = (N_m, N_{m-1}, \ldots, N_1)$, where $N_m, N_{m-1}, \ldots, N_1$ denote the normalized loop limits, loop coalescing is a compiler optimization that combines the multi-dimensional iteration space of L into a single-dimensional loop L' with $\prod_{i=1}^{m} N_i$ total iterations. Loop coalescing has the benefit of decreasing a substantial amount of overhead in task management in addition to improving loop schedules. This benefit remains as long as the amount of time spent computing the new single-dimensional indices is not greater than the overhead of dynamically spawning the threads for the inner parallel loops. Otherwise, there is no benefit to this transformation. The downside to loop-coalescing is that this transformation is oblivious to locality, and in some cases could decrease it. Part of the work in Chapter 4 introduces a method that combines loop coalescing with tiling for perfectly-nested parallel loops. In the situation where the loops are not perfectly-nested, techniques such as loop distribution [43] may be applied in order to force the loops to become perfectly-nested.

2.2 Dependence-Driven Execution Models

The goal of this section is to provide the reader with an intuition for what dependence-driven execution models are, including the description of an example dependence-driven runtime system named StarPU. While there are differences in terminology, implementation, and overall design between dependence-driven execution models, they all share the core idea that

a computation can begin as soon as all of its dependence constraints (task, data, or control) have been resolved. The root of this is derived from the classic dataflow-architecture based approach [44] where the instructions of a program are executed in a partial ordering that has been defined by its dependences. Unlike the von Neumann architecture, the dataflow model has no program counter and the execution of code can only be performed once all of its input arguments are available to it. The dataflow model has no global state since data (sometimes referred to as a *token*) is communicated along the edges connecting dependent nodes. Due to its stateless nature, the dataflow model is well suited to declarative programming paradigms including functional languages [45, 46, 47].

In a dependence-driven execution model, a program is represented by a directed graph $DFG = (V, E, s)$ where V represents computation nodes (also called *actors*), E are the directed arcs that represent dependences between the nodes, and s is the starting point of the program. A node $n \in V$ can execute (*fire*) as soon as all of its incoming dependences are resolved. The type of dependence and the granularity of a node differs based on the specific programming model. In the early days of dataflow, nodes were fine-grain (at the single instruction level), and as soon as the operands for the instruction were available (i.e. tokens on all of its input edges), the instruction could then execute atomically. While this exposes a high-degree of parallelism, the overheads from scheduling and communication were extremely high. Additionally, exploiting locality was difficult [48]. To alleviate scheduling overheads, the nodes had to be coarsened, effectively making this into *macro-dataflow*. In the macro-dataflow approach, the granularity of the nodes are typically at the task-level (i.e. more than one instruction). This reduces scheduling overhead, and locality can now be exploited within the node. The downside is that this still requires an explicit communication of tokens between the nodes in order to enforce the stateless design of the programming model. For example, in StarPU [11], a node is always a task (sometimes referred to as a codelet [14]).

As the number of processing units increases, so does the degree of parallelism available.

In an attempt to leverage this new-found parallelism, there has been a resurgence of execution models that are dependence-driven. Examples include StarPU [11], Intel Concurrent Collections (CnC) [16], SWARM by ET International [14], and PLASMA [49], among others. This new breed of dependence-driven execution models is fundamentally designed for shared memory multi-core and many-core systems along with some support for distributed memory machines.

2.2.1 StarPU Runtime System

This section will give a brief overview of the StarPU dependence-driven runtime system. For the implementation work that is part of this thesis, StarPU was chosen as the targeted runtime due to its maturity, well-studied performance, and support for data coherence between tasks.

StarPU is a dependence-driven execution runtime system that allows for the parallel execution of code on heterogeneous architectures including multi-core processors and many-core architectures such as GPU accelerators. Using a dynamic task-dependence graph as its input, the StarPU runtime system is able to efficiently schedule tasks onto the heterogeneous components of the machine. The runtime system also supports a software-based distributed shared memory (DSM) with relaxed consistency and data replication onto the components. In addition to automatically selecting a task scheduling policy based on different criteria, there is support for user-provided scheduling policies and performance analysis tools. The remainder of this section will describe the essential components necessary to understand the execution model used.

StarPU Task-Dependence Graph Construction

The simplest method of creating a program for StarPU is to declaratively describe the program in terms of *task* nodes. These tasks are C-style functions that can be executed

```
1  declare_deps(tagB, 1, tagA);
2  declare_deps(tagC, 1, tagA);
3  declare_deps(tagD, 1, tagA);
4  declare_deps(tagE, 1, tagB);
5  declare_deps(tagF, 1, tagB);
6  declare_deps(tagG, 2, tagE, tagF);
7  declare_deps(tagH, 3, tagG, tagC, tagD);

9  taskA->tag_id = tagA;
10 taskB->tag_id = tagB;
11 ...
12 taskH->tag_id = tagH;

14 task_submit(taskA);
15 task_submit(taskB);
16 ...
17 task_submit(taskH);

19 tag_wait(tagH);
```

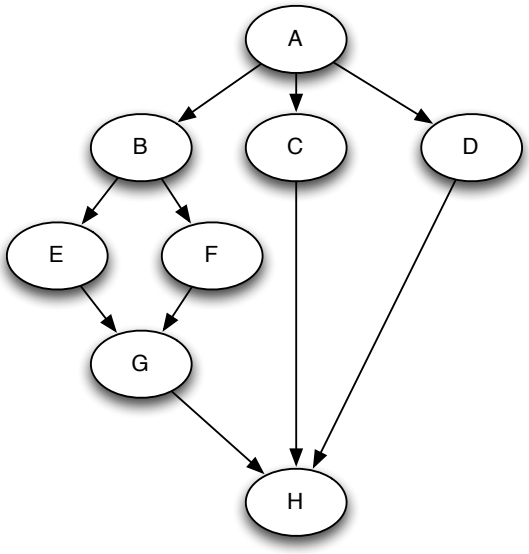


Figure 2.3: StarPU Task-Dependence Graph

independently of the other tasks. These task node are then asynchronously inserted to a task-dependence graph. Since StarPU has support for heterogeneous architectures, the task can be specified to execute either on a traditional CPU or take the form of a data-parallel kernel as used in CUDA [36] or OpenCL [50]. In order to start expressing dependences between the tasks, a *tag* (a task identifier) needs to be associated with the task. Once a task with an associated tag has completed its computation, any tasks that were dependent on that tag (assuming that was their only dependence) can start execution on the device. The creation of tasks and tags are decoupled. This was done in order for dependences between tags to be specified without having had created the task data structures.

Consider the example in Figure 2.3. The directed graph on the right represents a task-dependence graph where the nodes are the tasks, and the edges are their dependences. As shown on the left side of the figure, lines 1-7 construct the tag dependences and lines 9-12 associate a tag with a particular task. Once the graph is constructed, and the the individual tasks are submitted to the runtime, execution can begin. Since node `A` does not have any incoming edges, it can start immediately. Finally, line 19 blocks any subsequent execution

until the task with `tagH` completes.

StarPU Data Management

In order for the programmer to not deal with the explicit data management across different heterogeneous components, StarPU provides facilities to handle data management automatically through its distributed shared memory based on the *modified-shared-invalid* (MSI) coherence protocol. In order for the MSI protocol to keep the system coherent, data is able to be replicated across the different memory nodes (e.g. the CPU and GPU memory spaces).

In order to use the DSM, data is first registered with the runtime; then proceeds to perform the memory allocation on the respective devices and return a data handle back to the main application. This handle is then associated with a task. When a data lookup using the handle occurs inside of the tasks, the runtime performs the correct memory lookup.

2.3 Chapel Language Overview

This section presents a short overview of the Chapel programming language [21] and its support for data parallelism. The remainder of this thesis is based on the Chapel 1.4.0 release [51].

Chapel is an object-oriented parallel programming language designed from first principles, rather than as an extension to an existing language. The base language supports iterator functions, generic programming, and it also takes advantage of type inference to determine data types automatically. Chapel was designed to facilitate programming of next-generation parallel machines well, and make using current-generation machines more productive. Along with X10 [38] and Fortress [52], Chapel grew out of the DARPA *High Productivity Computing Systems (HPCS)* program. Chapel support for data parallelism, index sets, and distributed arrays are derived from ZPL [53] and High Performance Fortran (HPF) [54].

Chapel's concepts of task parallelism and lightweight synchronization are derived from the Cray MTA/XMT's extensions to C and Fortran [55]. Lastly, Chapel supports interoperability with C and CUDA through C-style extern mechanisms. For interoperability with other languages such as C++, Fortran, or Python, the Babel interoperability tool [56] can be used.

The reference implementation of the Chapel compiler is a source-to-source based compiler that generates C source code. There is also an option to use LLVM [57] as the compiler backend, but the work in this thesis does not leverage that.

2.3.1 Domains and Distributed Arrays

The core component for data parallelism in Chapel is the concept of a *domain*, which is an extension to *regions* first described in ZPL [58]. A domain is a first-class language construct that describes an index space [51]. In addition to supporting iterations by loops, domains are used to describe the size and shapes of arrays. Consider the following example in Figure 2.4.

```
var D: domain(2) = [1..n, 1..n];
var A: [D] real;
for xy in D {
   A(xy) = (1, 2);
}
```

Figure 2.4: Example Demonstrating Chapel's Support for Domains

Here, `D` is a 2D domain and is initialized to contain the set of indices *(i,j)* with $i \in \{1, 2, .., n\}$ and $j \in \{1, 2, .., n\}$. The array `A` has its index set defined by the domain, resulting in an $n \times n$ array. The loop performs an iteration over the 2D iteration space defined by domain D where each each iteration index xy is a two–element tuple.

2.3.2 Data Parallelism in Chapel

Chapel has rich support for data parallel computation, making it ideal for SIMD-like architectures such as the GPU [7]. The main construct for data parallelism in Chapel is the `forall` loop which iterates over the indices in a domain's index set or over a subset of the elements in an array. There is also built-in support for the *reduction* and *scan* operators. Chapel also allows users to define their own *reduction* and *scan* operations [59].

2.3.3 Task Parallelism in Chapel

Chapel's support for task parallelism includes `begin`, `cobegin`, and `coforall` statements. The unstructured `begin` statement will asynchronously spawn a task containing the contents of the `begin` block and possibly return at a later synchronization point specified by the programmer. The structured `cobegin` will spawn a separate task for each program statement inside of the `cobegin` block. The parent task that invoked `cobegin` will resume execution as soon as all `cobegin` inner tasks have completed. The structured loop statement `coforall` creates a separate task for each iteration of the loop. Similar to the `cobegin`, the parent task will wait for all `coforall` iteration tasks to complete before proceeding.

2.3.4 Distributions (Built-in and User-defined)

Data distributions in Chapel are essentially a recipe that the compiler uses to map a computation and its associated data to the nodes where the program executes. Languages such as HPF and ZPL have support for distributed arrays, but the semantics of the distributions are hard-wired into the compiler and runtime, leaving the programmer without enough flexibility to manipulate many forms of distributed data (such as sparse arrays). Similar to domains, distributions are first-class objects: they can be named, manipulated, and passed through functions.

Chapel provides a set of commonly-used distributions such as *Block* and *Cyclic*. Additionally, user-defined distributions [60, 22] enable the creation of a wide range of distributions that are application- or target-specific. User-defined distributions are developed directly in Chapel, typically using built-in features (e.g. classes, task parallelism, locales). This has the benefit that distribution developers can directly leverage the high-level facilities offered by the language rather than having to program in a lower-level language.

In order for users to write their own distributions, they must implement routines to fulfill the required interface. Interface components include the ability to create domains and arrays, wholesale assignment of index sets, iterators supporting sequential and parallel iteration over a domain, random access to elements of an array, and support for slicing and reindexing. If the user does not explicitly declare a distribution, Chapel will implicitly use a distribution that targets shared memory parallelism.

Chapel uses the `dmapped` keyword in order to map the domain's indices to the target architecture using the specified distribution. This approach is useful because Chapel distributions are designed so that they can be swapped in order to modify the implementation of a domain and its arrays without changing the code. The advantage to this is that the code is cleaner and more portable: users do not need to maintain a separate code base for each target architecture. The concept of user-defined Chapel distributions will be applied later in Chapter 5 for targeting GPU architectures.

Chapter 3

Compilation for Dependence-Driven Models

The chapter presents compiler algorithms that are used to map entire loop-based parallel imperative programs for efficient execution onto a dependence-driven execution model.

3.1 Introduction

There has been an active push in both industrial and academic labs to build *exascale* machine architectures with the goal of sharply decreasing energy consumption in order to achieve exascale levels of computing performance. For this to work, it would require new circuit topologies and machine architectures. There is also a fresh look needed at different parallel programming models as alternatives to the traditional approaches such as OpenMP [33] or OpenCL [50]. One of the major problems with the traditional loop-based parallel programming models is that parallelism is constrained by the use of barrier synchronization in fork-join styles of parallelism [61, 11, 13]. Any limitation on a program's parallelism could be detrimental since this is a vastly needed commodity for driving exascale levels of performance.

Instead, in the dependence-driven (e.g. dataflow) approaches, the only things limiting parallelism are the natural dependences that exist. For example, as part of the DARPA-sponsored Intel UHPC (Ubiquitous High-Performance Computing) project, Intel and its partners have been developing a novel architecture that is power efficient and scalable to exascale levels. An execution model chosen for this type of architecture is the dependence-driven system named SWARM (SWift Adaptive Runtime Machine) [14]. Since the vast majority of existing HPC applications are imperative and loop-based, the application programmers would need to rewrite their code to suit these dependence-driven models. A downside to this is the difficulty in porting the application to another language, which is not always practical or economical.

For this work, Chapel was chosen as the input language for reasons including:

- Chapel provides an extensible and easy to learn compiler infrastructure with an AST internal representation (IR). This makes adding new compiler passes straightforward and simple.

- The language has first-class support for different parallel constructs such as data-parallelism and task-parallelism.

- Chapel's *user-defined domain maps* [22] help in writing portable applications. The same loops can be used across a different class of parallel machines just by changing the domain map.

In determining a dependence-driven execution model, StarPU was chosen over other existing implementations for the following reasons:

- Support for data-coherence across tasks. This alleviates some of the complexity of coherent data management from the compiler, and pushes it into the runtime system.

- Support for execution on heterogeneous architectures. While not actively pursued as part of this thesis, this allows for the work presented here to be extended in order to leverage StarPU's support for heterogeneous systems.

- StarPU is a stable and production-worthy system that other research projects are currently leveraging [62, 63].

This chapter tries to address the question of whether it is possible (and worthwhile) to compile traditional (i.e. imperative and explicitly parallel) applications to execute on dependence-driven execution models. More concretely, this work will look at the compilation techniques that are necessary to map Chapel applications using multiply-nested parallel loops and execute them using the StarPU dependence-driven runtime system, while still requiring barriers for program correctness. Even though StarPU supports heterogeneous architectures including GPUs, the focus in this chapter will only be multi-core architectures. While the implementation of the concepts discussed in this chapter use both Chapel and StarPU, they would be similarly applicable to other similar languages and runtime systems.

A limitation of compiling data-parallel loops is that in order to ensure correctness, an implicit barrier synchronization (i.e. join) needs to occur after the loop, thus limiting potential parallelism [64]. As will be shown later in this chapter, the performance limitation of an implicit barrier synchronization will be further exacerbated when nested-parallelism is involved. To address this, a new language extension is defined to allow the user to explicitly express the dependences between different statements of a program, thus allowing other statements to execute concurrently if their dependences have been met. Additionally, Chapter 4 will describe an optimization that can reduce barrier synchronizations and find the ideal number of processor resources to utilize for a computation.

Contributions

The contributions of this chapter are as follows:

- The development of new compiler algorithms based on interval analysis that map multiply-nested parallel loops onto dependence-driven runtime systems.

- The introduction of *agglomerated flow graphs* and *agglomerated basic blocks* which represent a coarsened flow graph with disjoint parallel and sequential blocks of code.

- A method of propagating and mapping data declarations, their uses, and their definitions from a scoped imperative program across a set of distributed tasks in a task-dependence graph.

- New language extensions that allow the user to explicitly express ordering constraints between different statements of a program. These extensions effectively convey to the compiler what the required partial ordering is, so that the compiler can now enable loop optimizations that it might not have before.

The remainder of this chapter is organized as follows: Section 3.2 will provide a simple motivating example based on a 2-dimensional Jacobi Iteration. Next, Section 3.3 will present the compiler algorithms used to perform multiply-nested parallel loop partitioning onto the StarPU runtime system. This will also discuss a low-level intermediate language generated by the compiler. Afterwards, a performance evaluation is performed in Section 3.4. Next, Section 3.5 will briefly describe how to map non-loop based parallel constructs (e.g. `cobegin`, `begin`) onto the dependence-driven runtime systems. Section 3.6 will describe language extensions that can be used to express dependences explicitly in the code instead of having the compiler try to infer them. Afterwards, Sections 3.7 and 3.8 will discuss related work and any limitations from the proposed techniques. Conclusions will be presented in Section 3.9.

3.2 Motivation

As a simple motivation, consider the Chapel program in Figure 3.1. This is a 4-point stencil computation based on the Jacobi method used to solve the Laplace equation in two dimensions. Line 1 specifies to the compiler that a user-defined *domain map* is to be used. The domain map `CodeletDist()` informs the compiler to generate code for any declared arrays, domains, and their respective traversals via `forall` loops targeting the StarPU runtime infrastructure. The parallel loop in lines 19–22 performs the 4-point stencil computation. Line 24 is a parallel reduction that returns the maximum difference between the two arrays `XNew` and `X`. Finally, line 26 assigns the contents of `XNew` to `X`. Since lines 24 and 26 are data parallel statements in the form of array operations, the compiler will expand these statements into a loop-based form, and in the case of the reduction, multiple loops will be generated. This will be evident shortly in the generated flow graph.

In order to generate the task-dependence graph that will be used by StarPU, the following set of compiler analyses and transformations need to be performed:

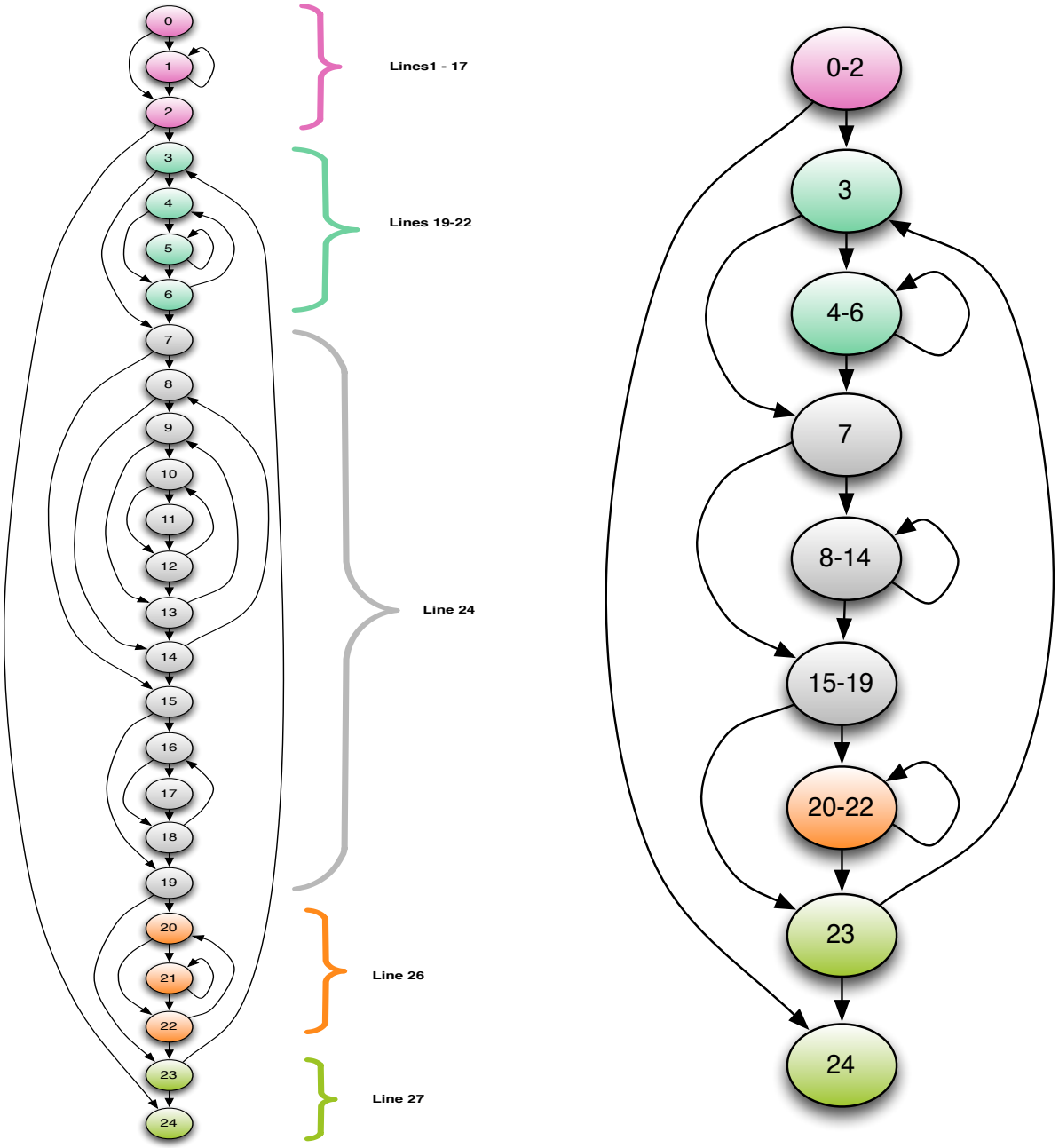
1. **Agglomerated Flow Graph**

   Any block of code that does not have a parallel construct inside of it will have its basic blocks combined together in order to form an *agglomerated flow graph (AFG)*. This step will partition the program into disjoint sequential and parallel components with their respective control statements connecting them. Additionally, this step will simplify the control flow analysis that will be used next. Figure 3.2a is the traditional generated flow graph from the given source code in Figure 3.1. In the flow graph, each color coded section represents a line (or set of lines) from the program source code.

   From the program in Figure 3.1, lines 1–11 represents different variable declarations and definitions. This section of source code corresponds to basic block number 0. The `for` loop (along with its header) from lines 13–15 corresponds to basic block numbers

```
1  const cdist = new CodeletDist();
2  const cPSpace = [1..n, 1..n] dmapped cdist;
3  const cDomain = [0..n+1, 0..n+1] dmapped cdist;
4  var X, XNew : [cDomain] real = 0.0;

6  const epsilon = ...;
7  const delta = ...;
8  const north = (-1,0),
9        south = (1,0),
10       east  = (0,1),
11       west  = (0,-1);

13 for i in 1..n {
14   X(n+1, i) = 1.0;
15 }

17 while (delta > epsilon) {

19   forall ij in cPSpace {
20       XNew(ij) = (X(ij+north) + X(ij+south) +
21                   X(ij+east)  + X(ij+west)) / 4.0;
22   }

24   delta = max reduce abs(XNew(cPSpace) - X(cPSpace));

26   X(cPSpace) = XNew(cPSpace);
27 }
```

Figure 3.1: 2-D Jacobi Method Implemented in Chapel



(a) CFG

(b) Agglomerated CFG

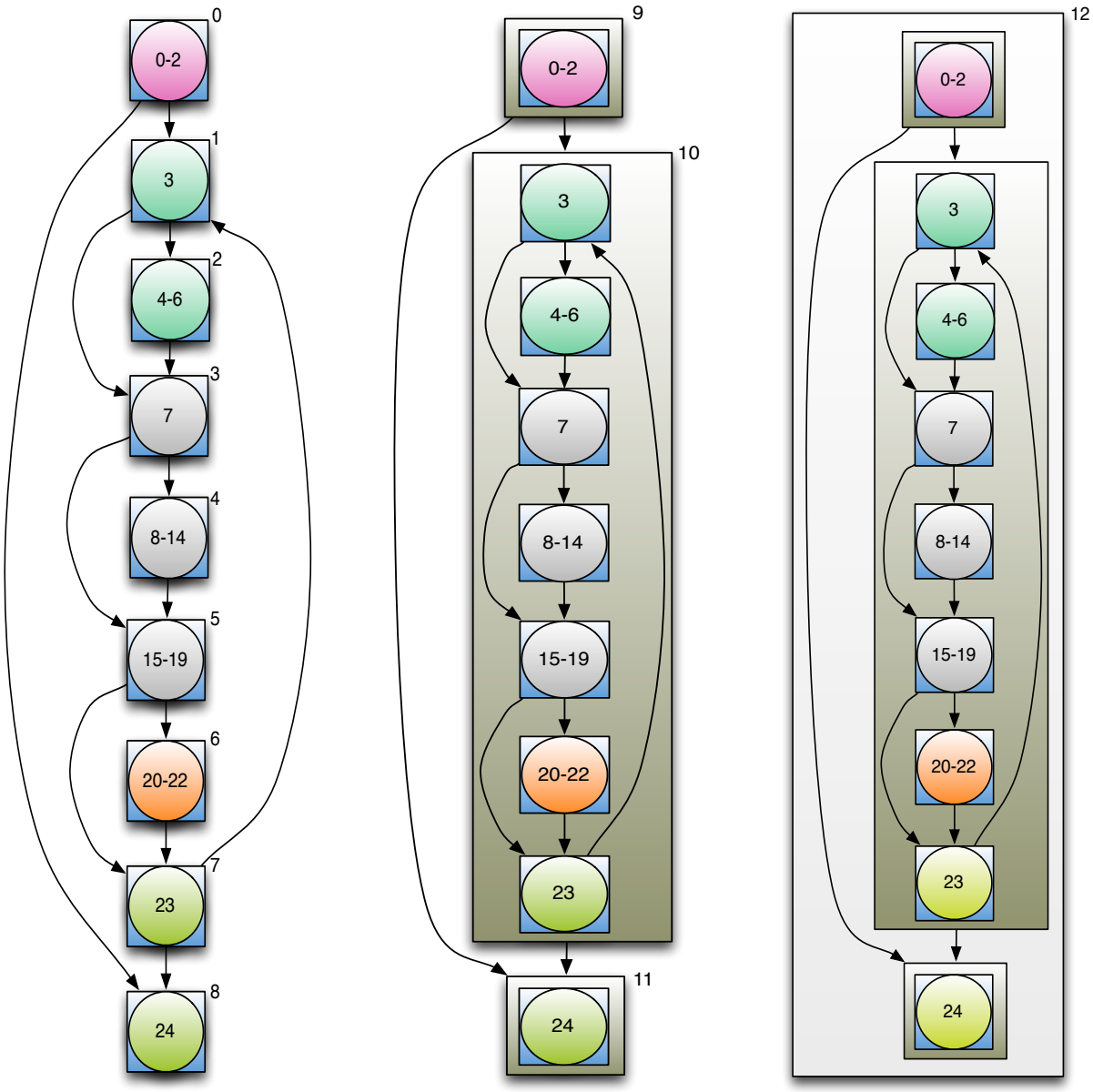
Figure 3.2: Traditional Flow Graph (a) and Agglomerated Flow Graph (b)

0–2. The loop header from the `while` loop is represented by basic block number 2. The `forall` loop and its stencil computation are represented by basic blocks 3–6. In this case, there are two loops that are formed. The outer loop is the parallel loop that spawns a set number of tasks (typically the same as the number of processors). The inner (sequential) loop will iterate through a subset of the total iteration space that was assigned to that task. The reduction operation from line 24 corresponds to basic blocks 7–19. During compilation, a reduction operation will expand into two disjoint loops. The first loop will perform a localized reduction per each task, and store its result into a per task element. The second loop will then perform a sequential accumulation across each localized result. The array assignment from line 26 corresponds to basic blocks 20-22, and this will also expand into a parallel loop. The backedge from the `while` loop and the remainder of the program is represented by basic blocks 23–24.

The AFG form of this same program is provided in Figure 3.2b. A circle node in the agglomerated flow graph represents an *agglomerated basic block*, and the edges represent the change in control or a change from a sequential to parallel block of execution. It should be evident that in the case of the AFG, there are fewer nodes and edges compared to the traditional CFG. For example, from the source code in Figure 3.1, lines 1–15 will map to the first (top-most) block in the AFG. They are not separated out into separate basic blocks since there is no explicit parallelism between them. This agglomeration will also occur for the sections of source code in lines 24 and 26. At this point during the construction of the AFG, `forall` loops are initially treated as sequential `for` loops (i.e. backedges exist for those loops). Section 3.3.3 provides a more formal definition of an AFG and algorithm of how it is constructed.

2. **Interval Analysis**

Instead of the customary case of using a control flow graph as input into the interval



(a) Interval Analysis Pass 1 (b) Interval Analysis Pass 2 (c) Interval Analysis Pass 3

Figure 3.3: Sequence of Derived Interval Graphs

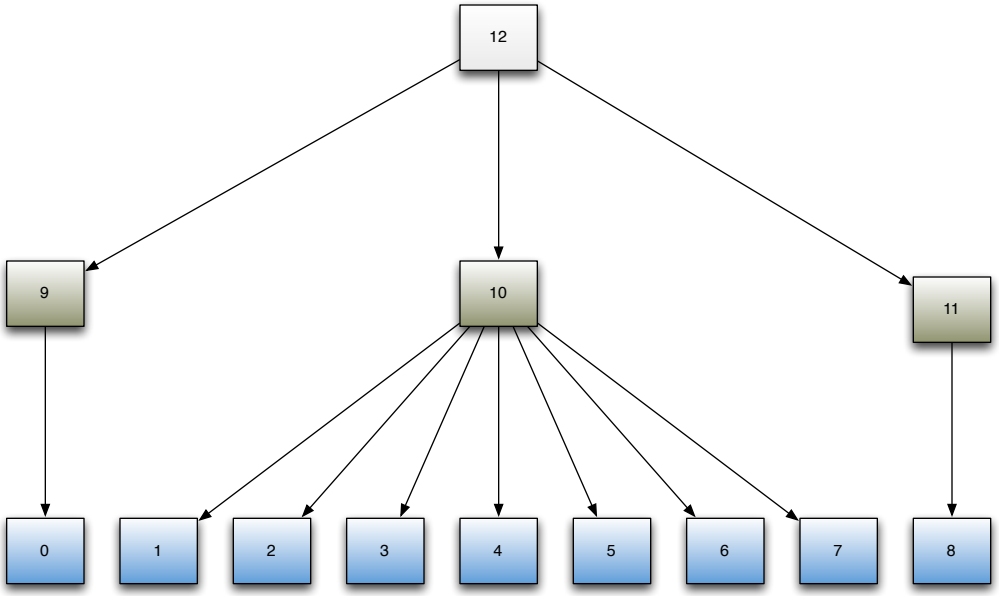


Figure 3.4: Interval Containment Tree

analysis compilation stage, an agglomerated flow graph will be used instead. By using the same example from Figures 3.1-3.2, applying interval analysis successively until the limit flow graph has been reached is shown in Figure 3.3. In this interval graph, square nodes represent intervals, where the interval can contain either a set of agglomerated basic blocks, or intervals from previous passes. Each unique number outside of every interval node represents its interval id. The edges between the intervals are the same as before from the flow graph. For this example, it took three passes of the algorithm to form the limit flow graph.

3. **Interval Containment Tree**

   An alternative representation of the interval graph from Figure 3.3 is the *interval containment tree (ICT)* [65] as shown in Figure 3.4. An ICT is a tree data structure used to represent the parallel loop hierarchy based on the sequence of derived interval graphs. Every node in the ICT corresponds to a node from the derived interval graph. Additionally, each level in the tree represents a pass of interval analysis. In

this figure, all non-leaf nodes represent *meta*-tasks. A meta-task has the responsibility of dynamically submitting tasks (one for each of its child nodes) and assigning the control dependences that were computed during interval analysis. In the situation where a meta-task represents the outer parallel loop of a multiply-nested parallel loop, the meta-task will spawn a number of parallel iterations corresponding to the loop's domain.

The leaves of the ICT represent the actual program code. To decrease task creation/destruction overhead, a meta-task node is replaced by its child node in cases where there are no sibling nodes for that child. As shown in Figure 3.4, this occurs with the leaf nodes 0 and 8 replacing their parents 9 and 11, respectively. As will be shown in Section 3.3, the ICT is used in conjunction with the computed edges of the interval graph in order to perform code generation of the static and dynamic portions of the task-dependence graph. This will be useful in cases where there are multiply-nested parallel loops that are dynamically spawned.

4. **Data Placement and Communication**

   With the requirement that the input program be imperative, and the output be effectively a distributed task-dependence graph that has the potential of executing tasks on different address spaces (e.g. GPUs, cluster, etc.), the compiler needs to perform an analysis to determine on what task nodes to declare data, and to what other task nodes it needs to explicitly communicate the assignments and uses of that data. This analysis attempts to map a program (with scoping rules) to one that that has been decomposed into disjoint tasks. An algorithm that uses the generated interval graphs and the ICT will be provided in Section 3.3.6.

5. **Code Generation**

   Continuing with the previous example from Figures 3.1-3.4, a task-dependence graph

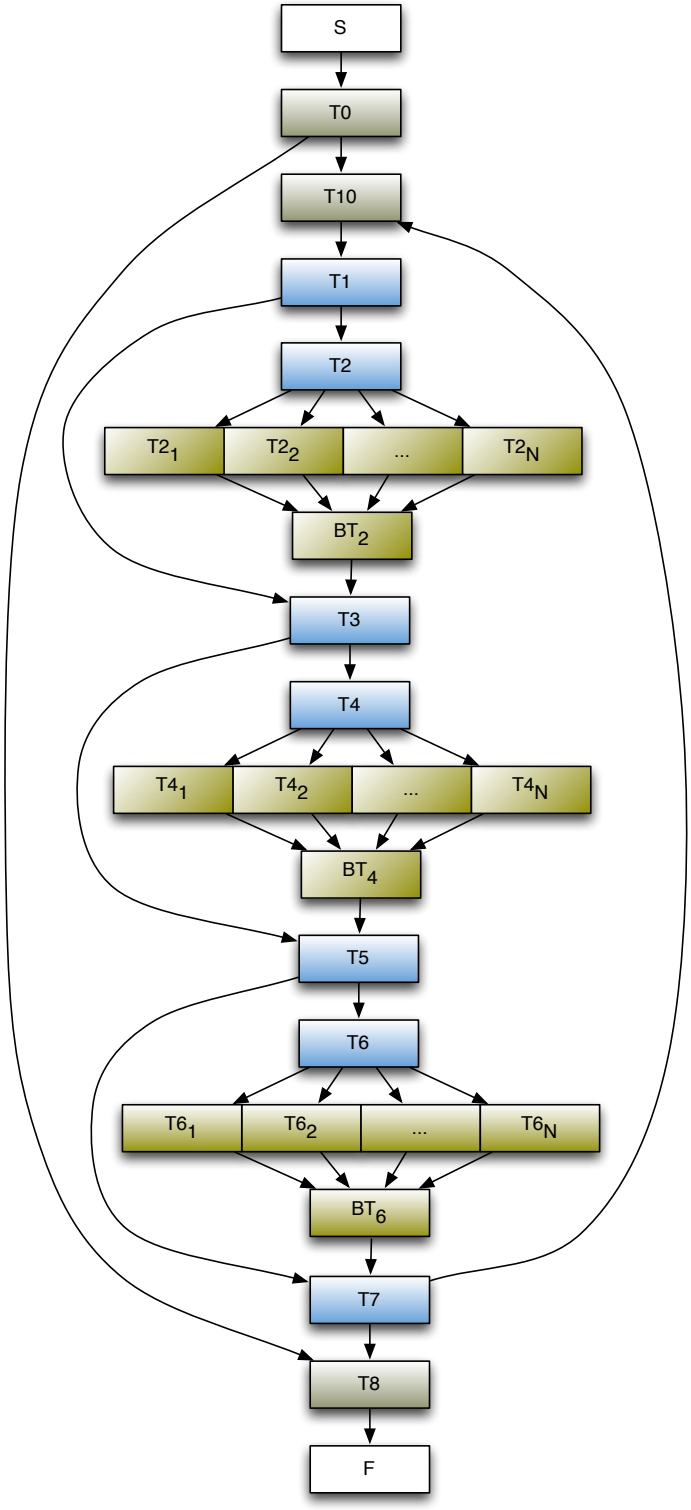


Figure 3.5: Generated Task-Dependence Graph for 2-D Jacobi

```
int main()
{
  /* initialize data structures */
  ....
  seq_task(fn0, 0, 10, 8, handles);
  seq_task(fn10, 10, 11, handles);
  seq_task(fn8, 8, FIN, handles);
  signal_task(0);
  wait_forall_tasks();
}

void fn0(void *buffers[], void *params)
{
  /* Generated code from BB 0-2 */
  ...
  if (delta > epsilon) {
    signal_task(10);
  else
    signal_task(8);
}

void fn10(void *buffers[], void *params)
{
  ...
  seq_task(fn1, 1, 2, 3, handles);
  par_task(fn2, 2, 3, 1, n, handles);
  seq_task(fn3, 3, 4, 5, handles);
  par_task(fn4, 4, 5, 1, n, handles);
  seq_task(fn5, 5, 6, 7, handles);
  par_task(fn6, 6, 7, 1, n, handles);
  seq_task(fn7, 7, 10, 11, handles);
  signal_task(1);
}

void fn8(void *buffers[], void *params)
{
  /* Generated code from BB 24 */
  ....
  /* No need to signal
   * since no successor nodes */
}

void fn1(void *buffers[], void *params)
{
  /* Generated code from BB 3 */
  ...
  if (takeForall)
    signal_task(2);
  else
    signal_task(3);
}

void fn2(void *buffers[], void *params)
{
  /* Generate body of forall loop:
   * BB 4-6 */
  ...
  signal_task_barrier();
}

void fn3(void *buffers[], void *params)
{
  /* Generated code from BB 7 */
  ...
  if (takeForall)
    signal_task(4);
  else
    signal_task(5);
}

void fn4(void *buffers[], void *params)
{
  /* Generate body of forall loop:
   * BB 8-14 */
  ...
  signal_task_barrier();
}

void fn5(void *buffers[], void *params)
{
  /* Generated code from BB 15-19 */
  ...
  if (takeForall)
    signal_task(6);
  else
    signal_task(7);
}

void fn6(void *buffers[], void *params)
{
  /* Generate body of forall loop:
   * BB 20-22 */
  ...
  signal_task_barrier();
}

void fn7(void *buffers[], void *params)
{
  /* Generated code from BB 23 */
  ...
  if (delta > epsilon) {
    signal_task(10);
  else
    signal_task(8);
}
```

Figure 3.6: Low-Level Generated Code for 2-D Jacobi

can now be represented as shown in Figure 3.5. For the task-dependence graph, each rectangular block and an ID represent a unique task. This task and its matching ID correspond to the nodes from the interval graph and the interval containment tree from Figures 3.3 and 3.4.

In addition to the start (S) and end (F) nodes, the task-dependence graph now includes additional nodes related to the number of parallel instances of a loop (e.g. $T2_1, T2_2, \ldots, T2_N$) and their implicit barriers (e.g. BT_1). By performing a top-down breadth-first traversal of the ICT from Figure 3.4, code can be generated. The corresponding generated code for this graph is presented in Figure 3.6. As will be described in Section 3.3.7, the `seq_task` and `par_task` calls are part of the low-level interface between the Chapel runtime system and StarPU. These routines create the necessary StarPU data structures, link dependences between the tasks, and submit the tasks to the StarPU scheduler. Additionally, `par_task` will asynchronously spawn the specified number of parallel iteration tasks with their respective barrier task (i.e. the implicit barrier). The `signal_task(`*id*`)` routine is used to notify a successor task that it can start executing. In the case of a parallel loop, each parallel iteration will notify the barrier task it has completed execution through a call to `signal_task_barrier()`. Once all of the worker tasks have completed and notified their barrier task, the successor of the parallel loop is notified that it can start execution.

Lastly, since nodes 9 and 11 in Figure 3.4 have only a single child, they can be replaced with leaf nodes 0 and 8, respectively.

3.3 Generating Task-Dependence Graphs From Data-Parallel Loops

The main goal in this section is to describe an intermediate representation, based on interval analysis [2], in order to represent a partitioning of explicitly data-parallel loops into a task-dependence graph. By constructing this intermediate representation, an imperative-based parallel program can be efficiently mapped onto the StarPU runtime system.

Consider the synthetic loop example in Figure 3.7. This is a program that has different control statements in the form of conditional statements and nested loops that are both sequential and parallel. It has a more complex set of control statements than the Jacobi iteration example from Figure 3.1. For the remainder of this section, this will be the ongoing example that the compiler transformations will be applied to.

3.3.1 Prerequisites

The requirements regarding the types of supported loops are the following: All sequential loops in a program are supported. Parallel loops, single and multiply-nested, must be explicitly marked (i.e. `forall`) and they can only contain loop-independent data dependences within their bodies. Since all parallel loops are explicitly marked, there is no need for the compiler to perform data dependence analysis.

3.3.2 Loop Partitioning

Algorithm 3 describes the overall approach to partition a program P into the task-dependence graph $G = (T, D, s)$, where T is a set of all decomposed tasks, D are the directed edges (i.e. dependences) between the tasks, and s is the initial starting task. The internal representation of the program P is an abstract syntax tree (AST). The inputs into the algorithm are the AST

```
1  /* initialize data */
2  x = ...
3  M = ...;
4  N = ...;
5  ...
6  while work_to_do {
7    if do_sequential then {
8      for i in 1..M {
9        for j in 1..N {
10         x(i,j) = ...;
11       }
12     }
13   }
14   else if do_parallel then {
15     if do_single_level then {
16       forall i in 1..M {
17         for j in 1..N {
18           x(i,j) = ...;
19         }
20       }
21     }
22     else if do_nested then {
23       forall i in 1..M {
24         forall j in 1..N {
25           x(i,j) = ...;
26         }
27       }
28     }
29   }
30   work_to_do = ...;
31 }
```

Figure 3.7: Synthetic Parallel Loop Example

function nodes, and the output is a code generated representation of the task-dependence graph. Each step of of this algorithm will be described in more detail in the subsequent sections, however the high-level approach is to do the following: the first step is to construct an agglomerated flow graph (AFG) for every function of the program. This partitions the function into the sections that are sequential and those that are parallel. The second step is to construct a sequence of interval graphs for the resulting AFG. These interval graphs provide the necessary loop hierarchy in the presence of control flow. The next step is to create an interval containment tree (ICT), where each interval node in the ICT represents a task in the generated task graph. Additionally, the ICT is generated since it will allow for data to be properly declared and communicated among the different tasks. The final step of

this algorithm is to perform the actual code generation for each task.

Algorithm 3: General Algorithm For Task Partitioning

Input: List *pList* containing all of program P's AST nodes where `ast.type=function`
Output: Generated task-dependence graph
$INTERVALS \leftarrow \emptyset$;
$TASKS \leftarrow \emptyset$;
foreach $F \in pList$ **do**
    $AFG_F \leftarrow$Construct agglomerated flow graph(*F.body*);
    $INTERVALS_F \leftarrow$ Sequence of derived interval graphs(AFG_F);
    $ICT_F \leftarrow$ Construct ICT($INTERVALS_F$);
    foreach $I \in ICT_F$ *in BFS order* **do**
        $TASKS_I \leftarrow$ Create task(I);
        Construct data($TASKS_I$);
    foreach $T \in TASKS$ **do**
        Code generation(T);

3.3.3 Agglomerated Flow Graph

Starting with an AST representation of each function, an *agglomerated flow graph (AFG)* can be created. An AFG is defined as a graph $G = (V, E)$, where the nodes V are made up of *agglomerated basic blocks*, and E are the edges represented by a change of parallel control between the agglomerated basic blocks. An agglomerated basic block is defined as the maximal set of connected basic blocks that do not contain any parallel construct. Anytime a new parallel construct is encountered, a new agglomerated basic block is constructed. Examples of an agglomerated block can be a basic block, branch of an `if-then-else` statement, the body of a loop, or an entire function. Instead of constructing a traditional CFG, the point of an AFG is to partition the program into disjoint sequential and parallel components, while still retaining the control constructs between these components. By agglomerating the basic blocks together, the number of nodes in the flow graph decreases. This has the advantage of making future analysis that is dependent on a flow graph simpler. Additionally, overhead of

Algorithm 4: AgglomerateFG(STMT, ABB)

Input: $STMT$: AST node
Input: ABB_i : Agglomerated Basic Block
Output: AFG : Agglomerated Flow Graph
if $ABB_i = \emptyset$ **then**
  $ABB_i \leftarrow$ create a new empty agglomerated basic block;
switch $STMT.type$ **do** /* What is STMT's ast node type? */
  case $BLOCK\_STATEMENT$ /* Compound set of statements? */
    if $STMT.body = LOOP$ **then** /* Is entire body of STMT a loop? */
      /* A loop nest with no parallelism inside */
      if $STMT \neq$ `forall` $\wedge \nexists$ *loop* $L \in STMT$ *where* $L =$ `forall` **then**
        $ABB_i \leftarrow ABB_i \cup STMT$;
      /* An outer parallel loop with no inner parallel loops */
      else if $STMT =$ `forall` $\wedge \nexists$ *loop* $L \in STMT$ *where* $L =$ `forall` **then**
        $ABB_{i+1} \leftarrow$ create a new empty agglomerated basic block;
        $ABB_{i+1} \leftarrow ABB_{i+1} \cup STMT$;
        $ABB_{i+2} \leftarrow$ create a new empty agglomerated basic block;
        Create edge from $ABB_i \rightarrow ABB_{i+1}$, $ABB_{i+1} \rightarrow ABB_{i+2}$, $ABB_i \rightarrow ABB_{i+2}$;
      else /* A nested parallel loop */
        $ABB_{i+1} \leftarrow$ create a new empty agglomerated basic block;
        foreach $s \in STMT$ **do**
          AgglomerateFG(s, ABB_{i+1});
        $ABB_{i+2} \leftarrow$ create a new empty agglomerated basic block;
        Create edge from $ABB_i \rightarrow ABB_{i+1}$, $ABB_{i+1} \rightarrow ABB_{i+2}$, $ABB_i \rightarrow ABB_{i+2}$;
    else if $STMT.body \neq LOOP$ **then**
      foreach $s \in STMT$ **do**
        AgglomerateFG(s, ABB_i);
  case $CONDITIONAL\_STATEMENT$
    /* If no parallelism in both branches of the conditional */
    if $\nexists$ *loop* $L \in (STMT.then\_branch \wedge STMT.else\_branch)$ *where* $L =$ `forall` **then**
      $ABB_i \leftarrow ABB_i \cup STMT$;
    else
      $ABB_i \leftarrow ABB_i \cup STMT.header$;
      $ABB_{i+1} \leftarrow$ create a new empty agglomerated basic block;
      foreach $s \in STMT.then\_branch$ **do**
        AgglomerateFG(s, ABB_{i+1});
      $ABB_{i+2} \leftarrow$ create a new empty agglomerated basic block;
      foreach $s \in STMT.else\_branch$ **do**
        AgglomerateFG(s, ABB_{i+2});
      Create edge from $ABB_i \rightarrow ABB_{i+1}, ABB_i \rightarrow ABB_{i+2}$;
      $ABB_{i+3} \leftarrow$ create a new empty agglomerated basic block;
      Create edge from $ABB_{i+1} \rightarrow ABB_{i+3}$, $ABB_{i+2} \rightarrow ABB_{i+3}$;
  otherwise /* Any other statement type */
  $ABB_i \leftarrow ABB_i \cup STMT$;

task management will be minimized since there would be fewer tasks.

Algorithm 4 provides a high-level algorithm describing the construction of an AFG. The initial input into the algorithm is an AST node $STMT$. A $STMT$ node can represent one of the following types:

BLOCK_STATEMENT

A statement that contains one or more statements inside the scope of a code block. For example, this can include the body of a basic block, body of a loop, or a branch of a conditional statement.

CONDITIONAL_STATEMENT

Represents an `if-then-else` conditional statement. This includes the respective `BLOCK_STATEMENT` for each `then` and `else` branch.

STATEMENT

Any other valid program statement.

If $STMT$ is the body of a sequential loop, and if it does not contain any embedded parallel loops inside of it, then the contents of $STMT$ are appended to an existing agglomerated block. On the other hand, if $STMT$ is the body of single-level parallel loop (i.e. no nested parallelism), then a new AFG block is created, and the contents of $STMT$ are added to it. Otherwise, if $STMT$ is a nested parallel loop, then the algorithm is recursively called with the body of the parallel loop as its input. Finally, with all of the new AFG blocks, their respective edges need to be created.

If the $STMT$ happens to represent a conditional statement (and its associated `then-else` branches), and there are no parallel constructs inside either of the branches, then the conditional statement is added to the existing AFG block. Otherwise, the algorithm will need to be called recursively on both `then-else` branches. Just as before, the edges between the

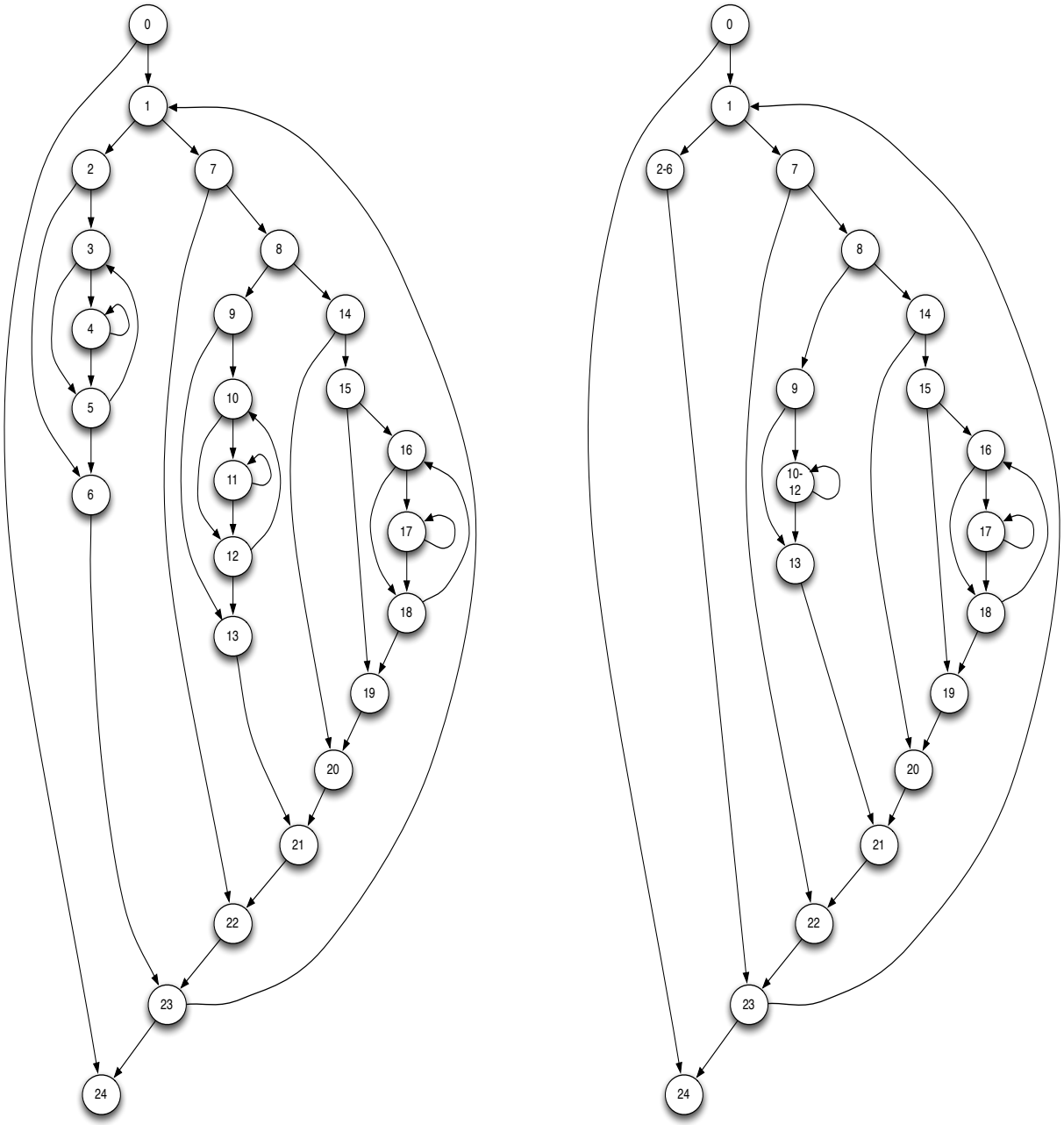
new AFG blocks need to be created. If $STMT$ is anything else, it is added to the existing AFG block.

Using the earlier example from Figure 3.7, a traditional flow graph construction algorithm would generate the CFG shown in Figure 3.8a. In this graph, the number for each basic block is displayed in the node. After applying Algorithm 4 to this program, the resulting AFG is shown in Figure 3.8b. It should be evident that there are now fewer (i.e. more dense) nodes in the graph. In the original flow graph, nodes 2–6 and 10–12 represent sequential `for` loops, but in the AFG, they have been combined into a single node. It is also important to stress that when tasks are generated, there will be fewer of them since the sequential basic blocks have been combined together. This will have the impact of a decreased amount of task creation overhead without any loss in parallelism.

3.3.4 Interval Analysis

Recall from Chapter 2.1.1 that interval analysis is a method of identifying the *hierarchical structure* of multiply-nested loops. Along with other elimination methods [28], partitioning flow graphs into interval graphs has historically been used as an alternative to iterative dataflow methods.

For the purposes of this thesis, interval analysis is used to find the structure of nested-parallel loops, and partition them into a derived sequence of interval graphs. Conceptually this is done in order to separate the sequential and parallel portions of the program so that the agglomerated basic blocks can be combined even further to minimize the task management resources. Mapping the nodes of each derived graph into a task-dependence graph that is suitable for StarPU now becomes trivial due to the direct correlation between the derived interval and task-dependence graphs. Interval nodes correspond directly to the tasks of task-dependence graph, and the edges correspond directly to the order of execution in the presence of control. One caveat is that interval analysis does not work in cases where a flow



(a) Traditional Flow Graph

(b) Agglomerated Flow Graph

Figure 3.8: Traditional Flow Graph and Agglomerated Flow Graph

graph is irreducible. However, since Chapel does not support unstructured statements (e.g. `goto, setjmp, longjmp`), Chapel programs will always reduce to a single node. In cases where the input language does support unstructured statements, node splitting [2] can be used to make the graph reducible.

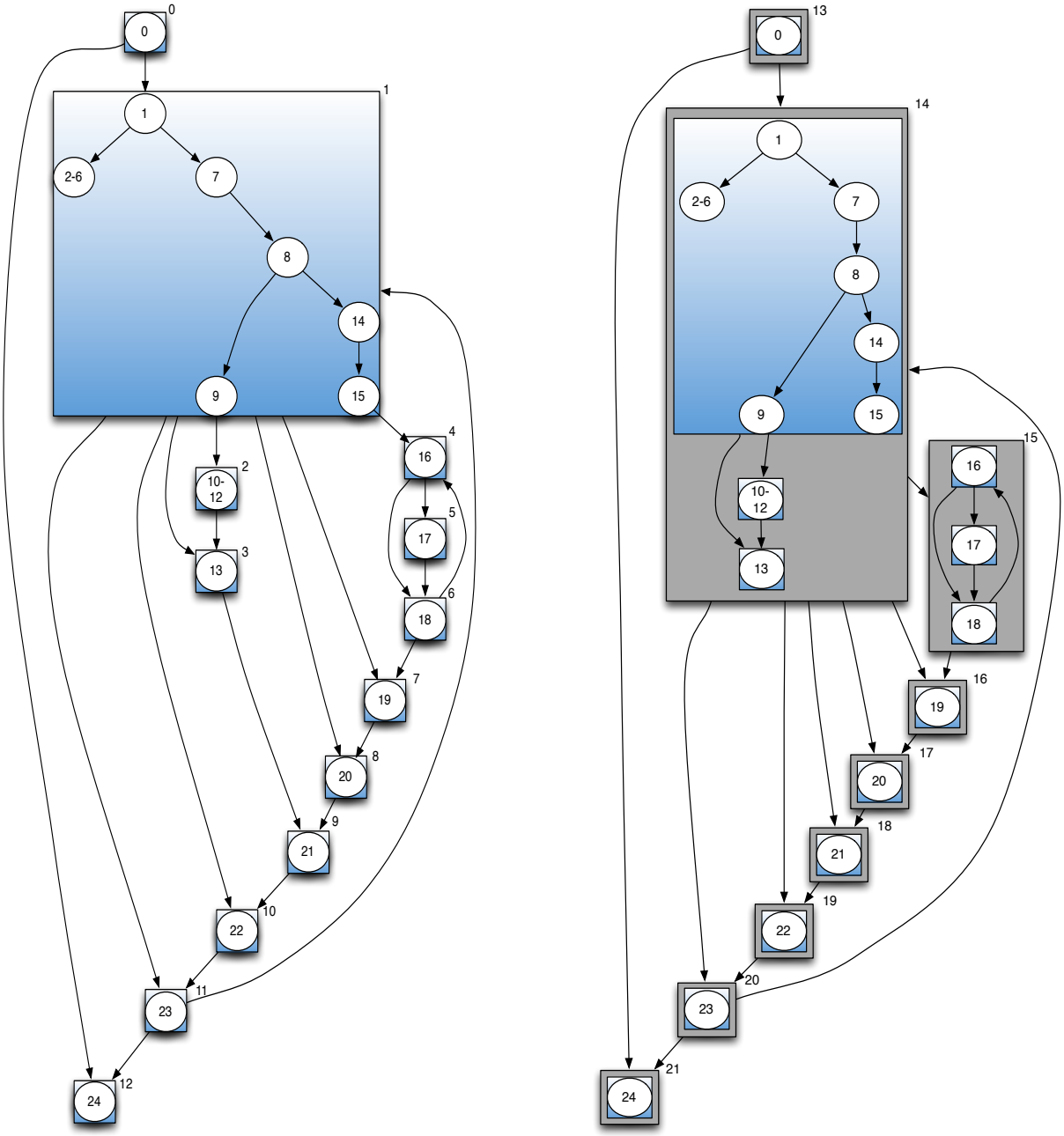
Using an agglomerated flow graph as input, finding intervals and performing the interval partitioning to construct $G = \{G^1, G^2, \ldots, G^d\}$ is described in Algorithms 1 and 2. Given an edge $e \to f$ between the two basic blocks e and f, if after applying interval analysis, e and f are in different intervals E and F, $e \to f$ will now be replaced with the interval edge $E \to F$.

Other techniques such as strongly connected components, detecting cycles, or finding natural loops [23] can be used to identify loops; unfortunately they have their associated downsides to them. Strongly connected components provide a too-coarse representation of the loops and do not capture any form of nesting information inside of the loop. Cycles are too fine-grain in that loops are not disjoint or properly nested. Finding natural loops can be used to find nested loops, but they do not actually partition the graph as intervals do.

Using the example program from Figure 3.7, the four derived interval graphs are shown in Figures 3.9 and 3.10. Each successive pass of interval analysis is applied, and the respective graph is derived until the limit flow graph is reached. In the generated interval graphs, the circles represent the agglomerated basic blocks with their numbers representing the basic block number. The colored rectangles represent the intervals, with the numbers outside of them representing their interval number. By observing these interval graphs, one can start to see the loop hierarchy that forms between each interval graph.

3.3.5 Interval Containment Tree

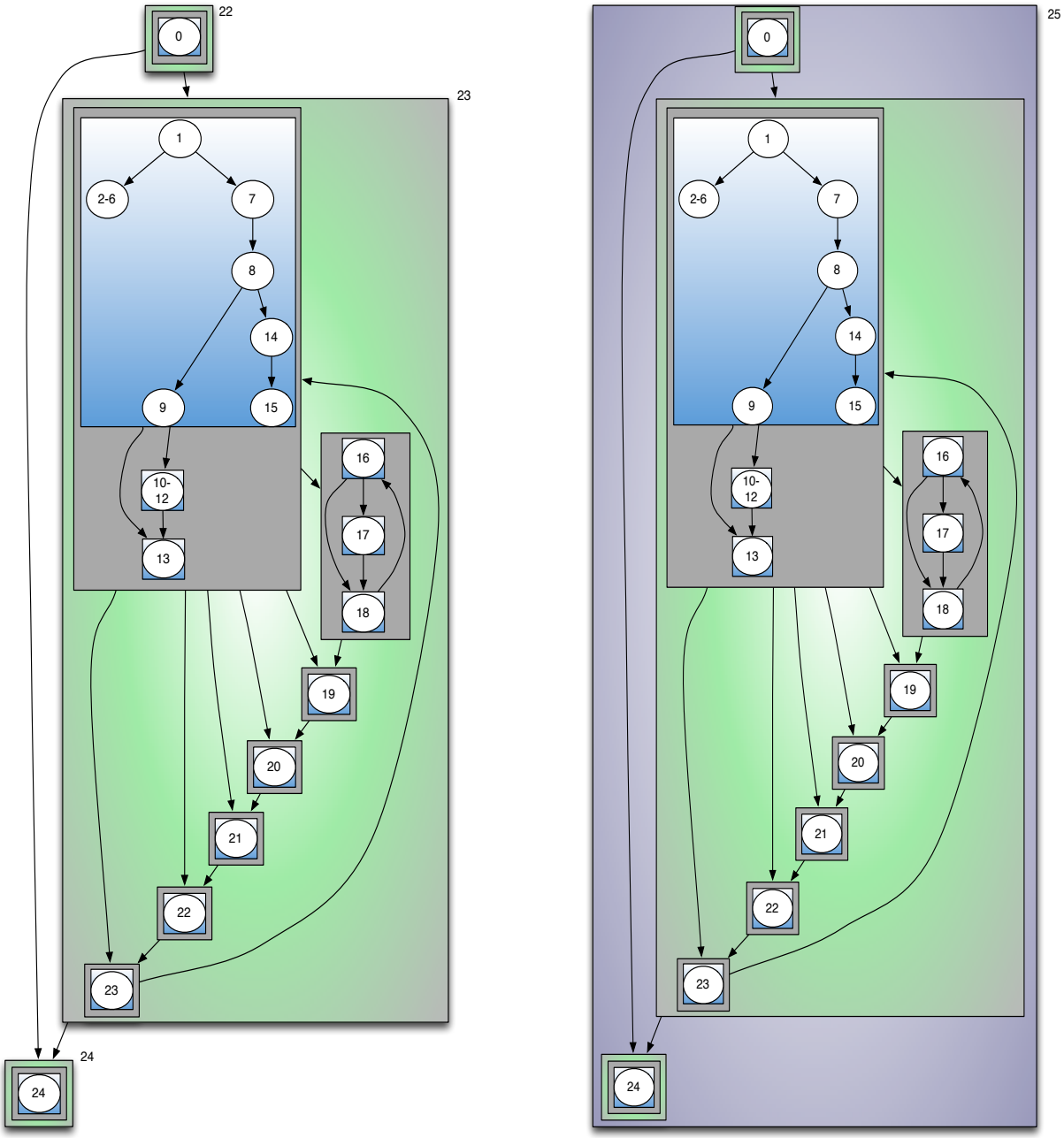
In order to represent the hierarchy from the derived sequence of interval graphs, an *interval containment tree (ICT)* [65], sometimes referred to as a control tree, will be used. An ICT



(a) Interval Analysis Pass 1

(b) Interval Analysis Pass 2

Figure 3.9: Generated Interval Analysis Graphs: Passes 1 and 2



(a) Interval Analysis Pass 3

(b) Interval Analysis Pass 4

Figure 3.10: Generated Interval Analysis Graphs: Passes 3 and 4

is defined as $T = (N, E, G^d)$, where N is the set of all interval nodes that appear in every derived interval graph. Given an edge $(a \rightarrow b) \in E$, interval node $a \in G^{i+1}$ is the node that has embedded inside of it the interval node $b \in G^i$. G^d is the root of T, where T has the following properties:

- The leaves of the ICT T, nodes in G^1, represent the agglomerated basic blocks.
- Non-leaf nodes $G^2, G^3, \ldots, G^d$ are containers of interval nodes $G^1, G^2, \ldots, G^{d-1}$.
- The height h of the ICT T is d, the number of steps in order to reduce G^1 into the limit flow graph G^d.
- The root of T given by the single interval node in G^d, represents the entire flow graph.
- If the flow graph has no parallel constructs, then T contains just a single node G^1.

Using the derived interval graphs from Figures 3.9–3.10, the computed ICT is given in Figure 3.11. Each number in the ICT corresponds to the computed interval number. Since G^4 is the limit flow graph, the ICT height h is 4.

In the situation where a leaf node does not have any siblings, there is an inefficiency of unnecessarily spawning as a task the parent of that leaf node. Redundant non-leaf nodes can be optimized away if the non-leaf node contains only a single child. In this example, nodes 13, 16–22, and 24 contain only just a single child node. If the non-leaf node contains only a single child, the child can then replace its parent. An optimized ICT for this example is provided in Figure 3.12.

3.3.6 Data Placement and Communication

After the ICT has been constructed, the next step is to perform the data placement and communication. By effectively partitioning a loop-based imperative program into a task-dependence graph, data will now need to be communicated from one task node to the

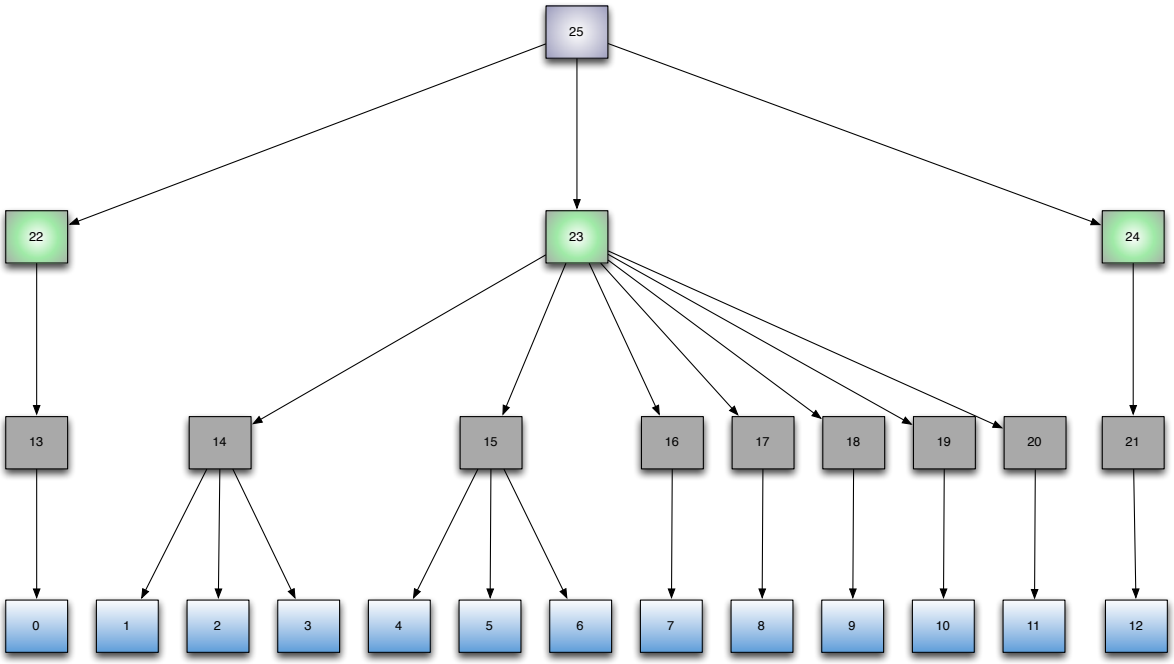


Figure 3.11: Interval Containment Tree

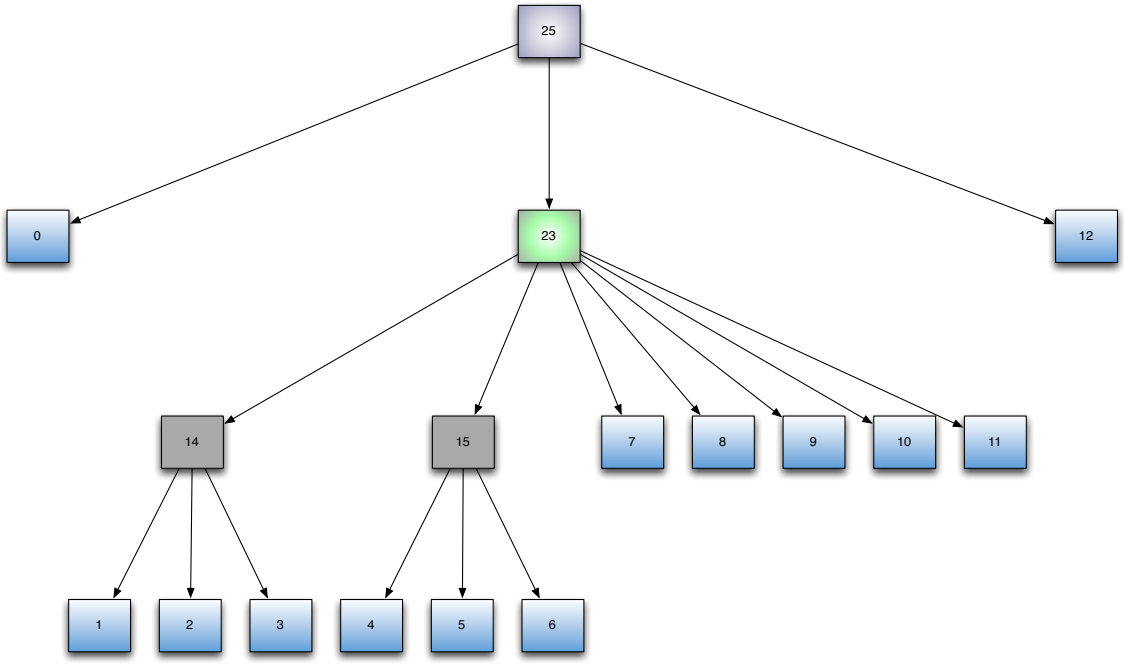


Figure 3.12: Optimized Interval Containment Tree

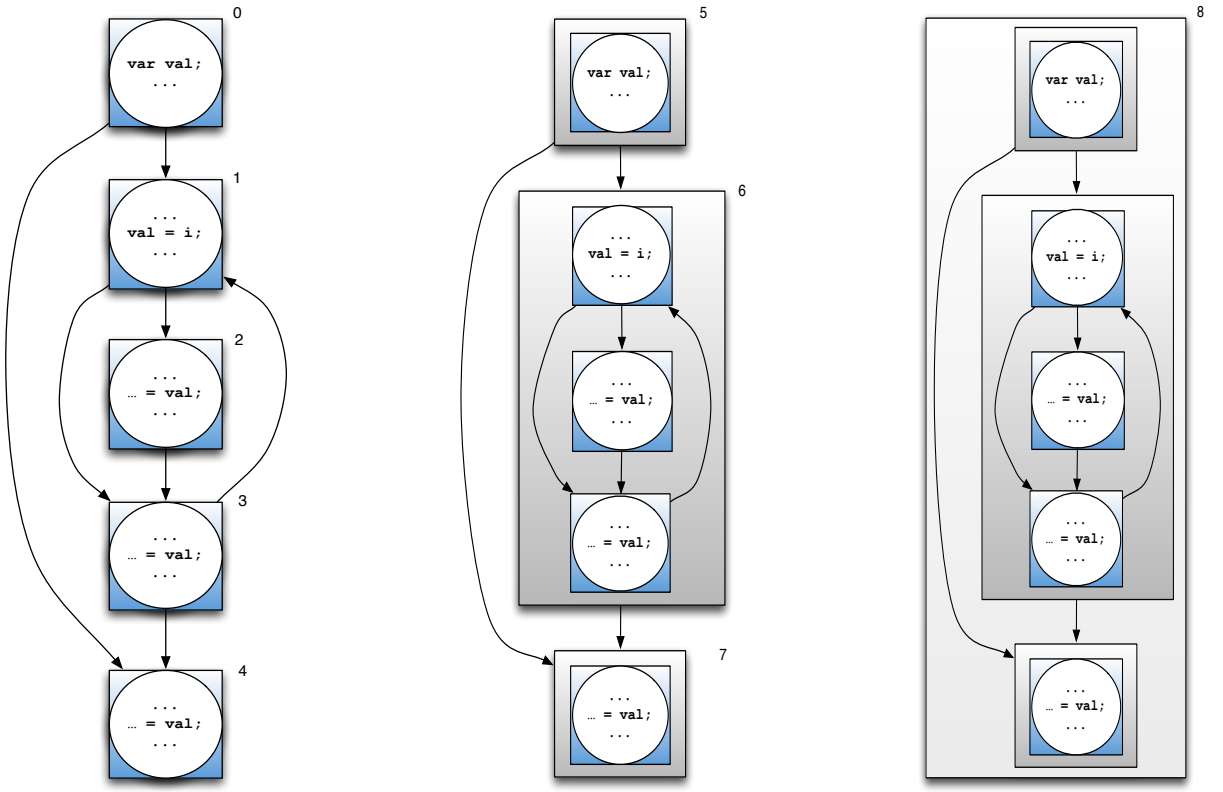
other. If a piece of data (e.g. an integer variable) is declared within scope in the imperative programming style, but due to interval partitioning, the data is defined and used by other tasks, there needs to be a way to communicate updates to the data.

Consider the loop in Figure 3.13. After performing interval analysis and generating an ICT, the declaration, definition, and use of `val` all occur in separate interval nodes as shown in Figures 3.14 and 3.15. The declaration of the variable `val` occurs in interval 0. Interval 1 performs a store operation into `val`, with intervals 2–4 performing reads on `val`. Since the interval nodes will all be mapped into separate tasks, the usage of `val` needs to be explicitly communicated with the tasks where `val` will be read and written.

```
var val : int;
for i in 1..M do {
  val = i;
  forall j in 1..N do {
    writeln(val);
  }
  ... = val;
}
... = val;
```

Figure 3.13: Data Placement Example

The general intuition of solving this problem of data placement and communication between the tasks is to first find the lowest common ancestor node P in the ICT where the data in question spans that parent's leaf nodes. In the case of this example, the root of the ICT is chosen since `val` spans all of its leaves. If for example, `val` was declared inside of the outer `for` loop, then the chosen parent node P would be Interval 6. Once the common ancestor P is located, a *handle* (i.e. remote reference) to the data in question needs to be declared at P, and then propagated down to all of its descendants. In the case of a leaf-node, the handle will be used to do a *lookup* on the data in order to retrieve its pointer and then perform any memory operation against that pointer. Non-leaf nodes that are descendants of P will only propagate the identification of the declared handle down to their children



(a) Interval Analysis Pass 1 (b) Interval Analysis Pass 2 (c) Interval Analysis Pass 2

Figure 3.14: Traditional Flow Graph and Agglomerated Flow Graph

nodes. Since StarPU guarantees data coherence across all of the tasks, any memory writes that occur in one task will be visible in other tasks (using StarPU's data handles).

Algorithm 5 formally describes the data placement algorithm. First, all of the variable declarations in every leaf node need to be located. For every declaration, *var_decl*, its lowest common ancestor P in the ICT needs to be found. Once P has been located, the original declaration will be moved to location P. Next, the variable declaration needs to be registered with the StarPU DSM, and a data handle is to be returned. Afterwards, a search is performed on all occurrences of the variable that is associated with *var_decl*. This includes both assignments and uses. The last step is to then proceed through all occurrences, and replace the original *var_decl* with the handle that was returned by StarPU.

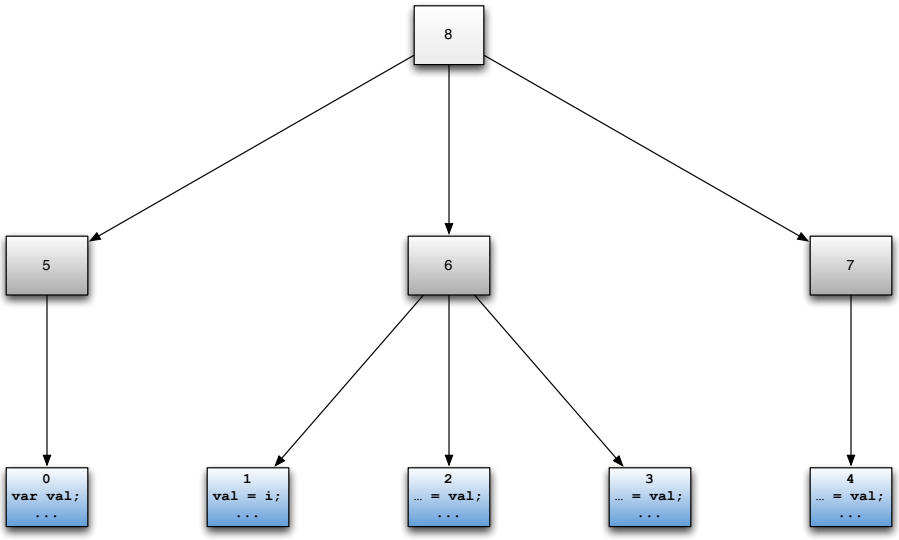


Figure 3.15: Interval Containment Tree

3.3.7 Code Generation

The final step is to perform the actual code generation of the StarPU task-dependence graph. The generated code will consist of an intermediate-level language interface that declares both the sequential and parallel tasks, and their respective control dependences.

Algorithm 5: Data Placement Algorithm

Input: IT : Interval Containment Tree
Input: FN : AST Node representing function
foreach $LEAF \in IT$ **do**
  foreach $var\_decl \in LEAF$ **do**
    $P \leftarrow$ find lowest common ancestor of all nodes containing var_decl;
    Place var_decl at P;
    $handle_{var\_decl} \leftarrow$ register StarPU data handle with var_decl;
    $S \leftarrow$ find all occurrences of $var\_decl.var \in FN$;
    foreach $s \in S$ **do**
      replace s with $handle_{var\_decl}$;

Intermediate-Language Interface

Before describing the code generation process, it is important to first define the intermediate-level language that will be generated by the compiler and used to interface with the StarPU runtime. The following interfaces were developed to act as an insulation between between the compiler-generated code and the StarPU runtime, so that in the future, it could be possible for an alternate runtime to be used. As shown previously in Figure 2.3, a StarPU program consists of dependence declarations, task definitions, and the links connecting the dependences to the tasks.

1. **Task Definitions**

   A task in StarPU is the unit of execution that is scheduled by the runtime. It is declared as the following C function prototype:

   `void (*task_fn)(void* buffers[], void *params)`

   - `void* buffers[]`

     Pointer to an array of handles that are maintained by the StarPU's coherency mechanism.

   - `void *params`

     Pointer to a buffer that is not managed by StarPU's coherency mechanism. In this thesis, the `params` field is not used. Instead, `buffers[]` will be used for all data communication between tasks.

2. **Task-Dependence Declarations**

   Using the following set of routines, either a *sequential* or *parallel* task can be submitted to the StarPU scheduler. The intuition for separating the tasks into sequential and parallel types, is so that the runtime does not need to dedicate cycles (i.e. overhead) related to SPMD parallelism that occurs in the parallel tasks.

(a) `seq_task(task_fn,self_id,dst_id`$_1$`,`*`[dst_id`*$_2$*`,...,dst_id`*$_n$*`]`*`,handles[])`

- `void (*task_fn)()`

  A StarPU task function.

- `int64_t self_id`

  A `int64_t` used to self identify the task.

- `int64_t dst_id`$_1$`,`*`[dst_id`*$_2$*`,...,dst_id`*$_n$*`]`*

  Used to designate the successor task. A destination value of -1 represents the final node. If there are more than one destination tasks, they are listed from `dst_id`$_2$`,...,dst_id`$_n$

- `task_data_handle_t handles[]`

  An array of type `task_data_handle` that contains handles to variables that have been registered with StarPU's coherency mechanism.

A `seq_task` routine registers with StarPU all of the data in `handles[]` and a single instance of `task_fn` to execute as soon as its input dependence (a predecessor in the interval graph) has been resolved. Once `task_fn` completes, one of the following outgoing edges {`dst_id`$_1$`,dst_id`$_2$`,...,dst_id`$_n$} will be signaled to execute.

(b) `par_task(task_fn,self_id,dst_id,start_idx,end_idx,stride,handles[])`

- `void (*task_fn)()`

  The task function that was defined just previously.

- `int64_t self_id`

  A `int64_t` used to self identify the task.

- `int64_t dst_id`

  A `int64_t` used to designate the destination task. Unlike the `seq_task()` case, there is only one outgoing task since parallel tasks (i.e. `forall` loops)

do not have a back edge.

- `task_data_handle_t handles[]`

  An array of type `task_data_handle` that contains handles to variables that have been registered with StarPU's coherency mechanism. Each *handle* of `handles[]` is associated with the task.

- `int64_t start_idx`

  The starting index for the given parallel 1–D iteration space.

- `int64_t end_idx`

  The ending index for the given parallel 1–D iteration space.

- `int64_t stride`

  A stride value specifying the amount the index value of the parallel loop is incremented.

The `par_task` represents parallel nodes from the interval graph, which in this case, is a Chapel `forall` loop. The routine differs from `seq_task` in that `par_task` will dynamically spawn $i = \left\lceil \frac{|end\_idx - start\_idx| + 1}{stride} \right\rceil$ parallel instances of `task_fn`, and a barrier task named `barrier_task`. When each $\texttt{task\_fn}_{[0..i]}$ completes, it will send a signal notification to `barrier_task`. After all of the inputs to `barrier_task` are available, the successor of the parallel task will be signaled by `barrier_task` to execute.

3. **Task Notification**

There are two approaches in specifying the completion of a task to any of its outgoing tasks: *pull* and *push*. The pull scenario specifies that any task that is dependent on another task can start executing as soon as its predecessor has completed. This occurs with the runtime routinely performing a test to see if all of the required dependences for a task have been resolved. As soon as all of these dependences have been met,

the runtime will start executing the outgoing task. The push scenario specifies that upon completion of a task, the outgoing task is notified explicitly (through a signal mechanism) that its predecessor has completed execution. As soon as the outgoing task has been notified by all of its required predecessors, the outgoing task can start execution. For example, after every worker thread in a parallel loop has completed its set of iterations, the worker thread will send an explicit notification to the barrier task. As soon as all of required worker threads have notified the barrier task, the barrier task can then start execution and signal its successor to start execution.

The approach taken in this thesis is to use the push mechanism in order to notify the specific successor that its predecessor has completed. This was done in order to handle control flow from loops. For example, in a `do-while` loop, based on the loop condition, either a backedge to the header, or the successor edge will be taken. In the push scenario, it is easy to determine at runtime which task to notify based on the value of the boolean condition. On the other hand, in the pull scenario it is not clear whether a header or successor is to start execution since the flow of control can go to either depending on the outcome of the condition.

The following two instructions are used to notify an outgoing edge:

(a) `signal_task(int_64_t dest_id))`

Send an explicit signal to the task with the id `dest_id`.

(b) `signal_task_barrier(void)`

Invoked at the end of each `par_task` in order to notify `barrier_task` that the instance of $\texttt{par\_task}_i$ has completed execution.

4. **Data Registration**

In order for data to be coherent across different tasks that could exist across different address spaces, a handle to the data in question needs to be registered with the runtime

system's distributed shared memory (DSM) mechanism. The following two routines describe how to register data with the DSM and how to extract a pointer to the coherent data in order to use it within the task.

(a) `task_register_data(handle, var, size)`

- `task_data_handle_t *handle`

  A pointer to an already allocated `task_data_handle`.

- `void *var`

  A pointer to an already allocated variable that needs to be registered with the runtime system's DSM.

- `size_t size`

  The size of `var` in bytes.

This routine registers the data pointed to by `var` with a size of `size`. The registration can now be referenced by `handle`.

Example:

```
int a = 0, b = 1, c = 1;
task_data_handle_t *handle = malloc(sizeof(task_data_handle_t *) * 3);
...
task_register_data(handle[0], &a, sizeof(int));
task_register_data(handle[1], &b, sizeof(int));
task_register_data(handle[2], &c, sizeof(int));
```

(b) `void * task_get_data(buffer)`

- `void *buffer`

  An index into the `buffers[]` parameter from the task function.

Returns a pointer to the data that is pointed to by `buffer`. The returned pointer could point to either the original copy of the data, or a coherent copy.

Example:

```
void task_func(void *buffers[], void *params)
{
  int *a, *b, *c;
  a = task_get_data(buffers[0]);
  b = task_get_data(buffers[1]);
  c = task_get_data(buffers[2]);
  *a = *b + *c;
}
```

Code Generation Implementation

The general approach taken for code generation is to perform a level-order (breadth-first) traversal of the ICT and generate a unique `task_fn()` procedure for each interval node. The non-leaf nodes ($G^2, \ldots, G^d$) of the ICT, generated using the low-level language constructs, are simply *meta-tasks* that dynamically spawn their children nodes as tasks. The leaf nodes (G^1 of the interval graph) represent that actual agglomerated basic blocks. If the interval node contains more than one successor, the compiler will generate the branch statement at the end of the interval. The body of the `then-else` will only contain the notifier (i.e. `task_signal()`), in order to signal the successor task that needs to be taken. Figure 3.16 presents the overall task graph for the earlier example from Figure 3.7.

3.4 Evaluation

By mapping a program consisting of data-parallel loops onto a dependence-driven runtime system, several overheads have been introduced that are not typically present in other imperative parallel programming models. This includes separate tasks to deal with program control and parallel loop barriers. The goal of this section is to measure the effectiveness of the proposed compiler transformations and understand the overall impact on program scalability that these additional overheads introduce. The scalability figures presented here

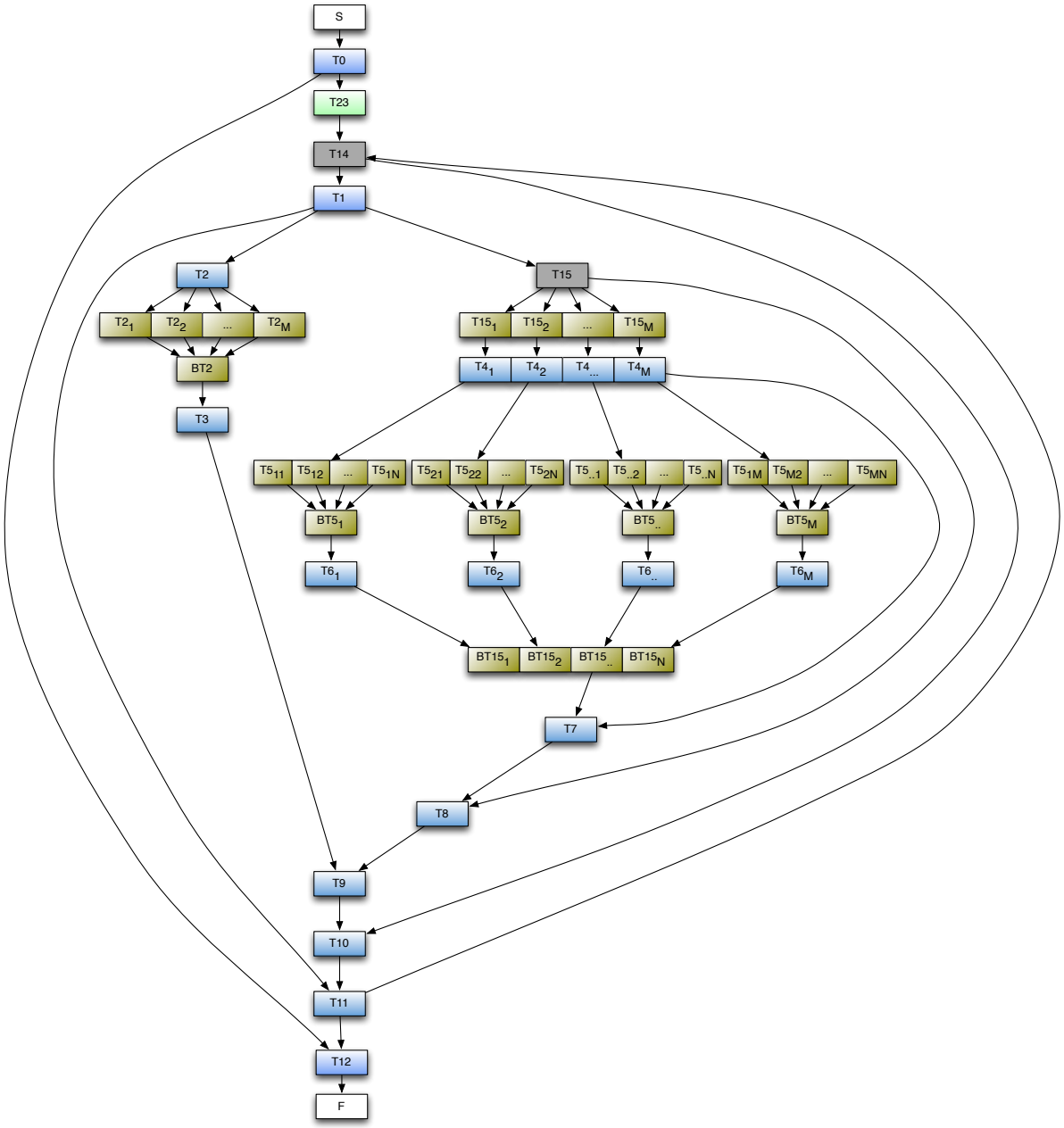


Figure 3.16: Task Graph for a Program Containing Multiply-Nested Parallel Loops (from Figure 3.7)

will also be compared to that of the native Chapel compiler where its default runtime system leverages *POSIX Threads* for shared memory support.

Section 3.4.1 describes the environment used for the experiments. Section 3.4.2 presents the methodology used to perform the evaluation. Section 3.4.3 describes each of the benchmarks. Finally, Section 3.4.4 presents the overall results.

3.4.1 Environmental Setup

Table 3.1 describes the environment that was used to perform the experiments.

CPU Architecture	X86-64 Intel Xeon E7-4860
Num cores (threads per core)	40 (2)
Total threads	80
Clock Rate (GHz)	2.26
Total Memory (GB)	128
L3 Cache size (MB)	24
OS (Kernel)	Scientific Linux 6.3 (2.6.32)
Compiler	GCC 4.4.6
Chapel Compiler	1.4.0
StarPU Runtime	1.0.3

Table 3.1: Architecture Tested

3.4.2 Experimental Methodology

The experiments in this section will be used to measure processor scalability ($1 \rightarrow 32$ processors logarithmically). This evaluation will provide a way of understanding what overhead has been incurred when comparing the dependence-driven work presented in this chapter, to a native Chapel implementation, where the only difference between the two sets of benchmark implementations is the target distribution. The baseline for the scalability measurements is the native Chapel implementation using one processor.

The methodology used to evaluate each of the benchmarks will be to do the following experiment comparisons:

1. Outer Loop Parallelization

   Only the outer loop is to be parallelized with the remainder of the loop nest being sequential.

2. Nested Parallelization

   A similar approach to the outer loop parallel test, except that in addition to the outer loop being parallelized, each of the inner loops will be marked as parallel. This is done in order to understand what the incurred impact on scalability is from the additional overhead of using nested parallel loops when compared to just a single level of parallelism.

An experiment that was also considered was to compare the performance difference between the compilation approach proposed in this thesis and directly in StarPU. However, this evaluation was not done because the Chapel compiler does not yet support many of the scalar optimizations that are currently applied by other compilers. Since comparing with StarPU requires a C-based backend compiler, it will be difficult to assess whether the difference in performance between the approach proposed in this thesis and StarPU is due to the Chapel compiler or due to overheads from the proposed approach.

As an example, Figure 3.17 shows the difference in performance for a 3–D Jacobi Method written in both Chapel and OpenMP. The OpenMP code was compiled using GCC 4.4.6 with the `-O3 -fopenmp` compiler flags, and run on the same environment described in Table 3.1. This program was executed using different input sizes and run with a total of 32 threads. The Chapel implementation is based on the native compiler with none of the modifications presented earlier. Based on the size of the input, the OpenMP implementation of the same algorithm leads to a speedup from 38% to 210%. In addition to the comparison shown here,

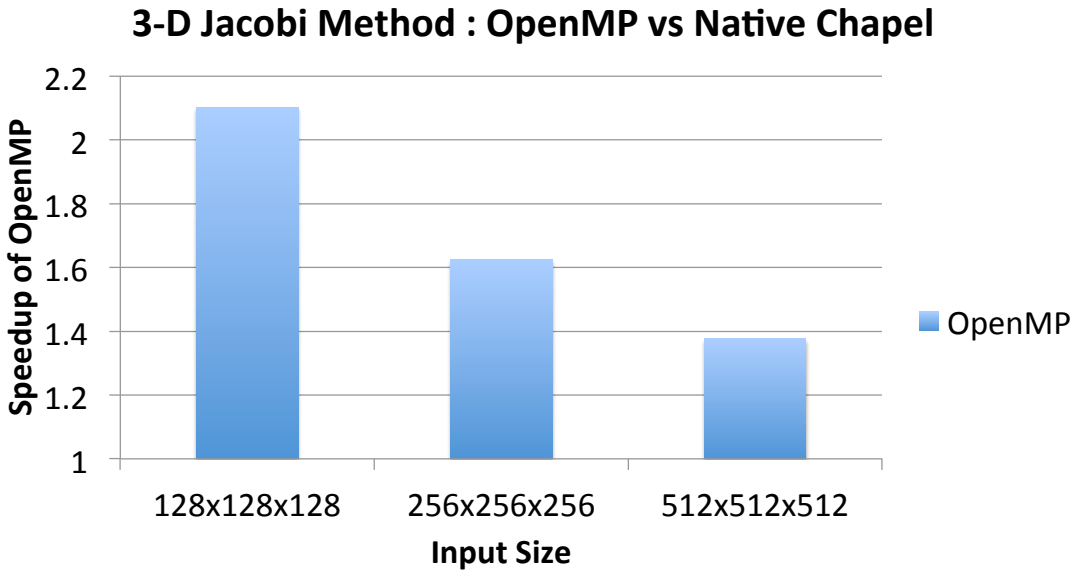


Figure 3.17: Speedup of OpenMP (over Native Chapel) Using 32 Threads

there have been other evaluations showing similar performance trends [66, 67, 68]. As a result of this, this makes any comparison of the techniques presented in this thesis with that of a tuned implementation, difficult. However, as the scalar performance of the Chapel compiler improves over time, these differences in overall performance will become closer, thus making the comparison more fair.

3.4.3 Experiments

In all of the selected applications in this section, the input sizes will vary in order to show what effect on overhead that particular input size has.

The following applications have been used as part of the evaluation process:

1. **3–D Jacobi Method**

   27–point stencil computation based on the Jacobi method that is used to solve Laplace's equation in 3–D. Experiments will be performed on the 3–D grid sizes $\{128\times128\times128\}$, $\{256\times256\times256\}$, and $\{512\times512\times512\}$. In these experiments, either the outer loop will be parallelized (in the outer loop parallelization scenario), or the outer two loops

will be parallelized (in the nested parallel scenario). However, in both experiments, the inner-most loop will be left sequential. In addition to the main stencil computation, the benchmark has two other parallel loops: a reduction to find the minimum, and a parallel swap operation between the two arrays. Using an inner dimension size of $\{128, 256, 512\}$, it is expected that there will be an impact on scalability as the problem size increases. This implementation is similar to the previous example from Figure 3.1.

2. **Sparse Matrix-Vector Multiplication (SpMV)**

   A standard sparse matrix-vector multiplication kernel based on a compressed sparse row (CSR) data storage format. The parallelization will occur at two levels: horizontal slices down the matrix, and across a set number of vectors (16,384) that are multiplied against the sparse matrix. The iterations across a row will not be parallelized in this evaluation since that would require synchronization. In the outer level parallelization test, the outer level loop containing the multiple vectors will be parallelized, whereas in the nested parallelization test, both the loop that iterates across vectors, and the loop that iterates across the rows of the matrix will be parallelized. Multiple vectors were used in this evaluation in order to understand the effects on load-balancing and locality. Table 3.2 lists the sparse matrices and their associated properties. The graphical representation of each sparse matrix is provided in Appendix A. They all have been imported from the University of Florida Sparse Matrix Collection [69].

Matrix	**Dimensions**	**Non-Zeros**	**Graph Figures (Appx.)**
Meszaros/ex3sta1	17,443 x 17,516	68,789	Figure A.1
Meszaros/stat96v5	2,307 x 75,779	233,921	Figure A.2
LPnetlib/lp_osa_14	2,337 x 54,797	317,097	Figure A.3
Andrianov/ex3sta1	16,782 x 16,782	347,890	Figure A.4
Rommes/bips07_2476	16,861 x 16,864	66,498	Figure A.5

Table 3.2: Sparse Matrices Evaluated

3. **Coulombic Potential (CP)**

   Coulombic potential grid calculation over a plane on a 3–D grid. This benchmark is based on the time consuming portions of the "*cionize*" ion placement tool in VMD [70]. CP was ported over from the Parboil Benchmark Suite [71]. Four different grid sizes were evaluated: $\{64 \times 64 \times 1\}$, $\{128 \times 128 \times 1\}$, $\{256 \times 256 \times 1\}$, and $\{512 \times 512 \times 1\}$.

4. **Magnetic Resonance Imaging FHd (MRI-FHd)**

   Computation of an image-specific matrix FHd, used in a 3D magnetic resonance image reconstruction algorithm in non-Cartesian space [72]. MRI-FHd has been ported to Chapel from the Parboil Benchmark Suite. Unlike the other benchmarks evaluated in this section, MRI-FHd has only a single level of parallelism in its main computation kernel. Two sizes will be evaluated: $\{32 \times 32 \times 32\}$ and $\{64 \times 64 \times 64\}$.

5. **Magnetic Resonance Imaging Q (MRI-Q)**

   Computation of a matrix Q, representing the scanner configuration, used in a 3D magnetic resonance image reconstruction algorithm in non-Cartesian space [72]. MRI-Q has been ported to Chapel from the Parboil Benchmark Suite. Similar to MRI-FHd, MRI-Q has only a single level of parallelism in its main kernel. Two sizes will be evaluated: $\{32 \times 32 \times 32\}$ and $\{64 \times 64 \times 64\}$.

6. **2–D Image Histogram Computation**

   A histogram is a statistical representation of different image pixel frequencies inside an image. For this benchmark, a 2–D histogram will be used to represent the number of pixels that have different light levels. A 1–D integer array will be used to store histogram values. To implement this without any explicit synchronization, a private (per task) histogram computation is performed, and then a final parallel reduction combines results together. Three different 2–D image sizes will be evaluated: $\{4,096 \times 4,096\}$, $\{8,192 \times 8,192\}$, and $\{16,384 \times 16,384\}$. The number of bins in the histogram will

be set to 256.

7. **Synthetic.Trig**

   The kernel for this program contains a multiply-nested parallel loop that performs a trigonometric function repeatedly for a set number of iterations ($1,024 \rightarrow 32,768$ logarithmically). This kernel is purely computation bound, as there is no access to data in the main loop. The main goal here is to see what effect there is on processor scalability when the kernel is compute-bound. This kernel is similar to the one that will be shown later in Figure 4.1.

3.4.4 Results

This section will first present a summary of the results, followed by a more in depth analysis of each of the benchmarks measured.

Results Summary

The common trend in all of the following results (Figures 3.18-3.24) show that when outer loop parallelization is applied to the benchmarks, the dependence-driven and native Chapel implementations scale similarly, except near the end where the dependence-driven approach typically scales higher. The main reason for this is that since both implementations generate the same kernel (i.e. compute) code, their only major difference is the underlying runtime system used. One possibly scenario to this is that the dependence-driven approach scales better due it having a more efficient tasking system over the tasking support in the native Chapel's runtime system.

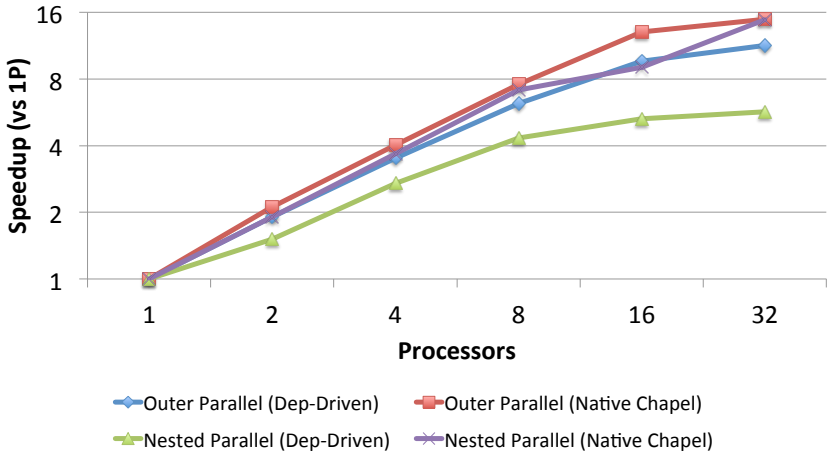
In the case of nested parallelism, it is evident from the majority of results that nested-parallelism in the dependence-driven approach introduces too much overhead compared to the native Chapel implementation. In a few cases such as the 2–D Image Histogram bench-

mark, the overhead introduced by nested parallelism will actually cause a slowdown, as shown in Figure 3.23. Part of the work in Chapter 4 will be to look at an alternative loop transformation that is similar to nested parallelism, but without all of associated overheads.

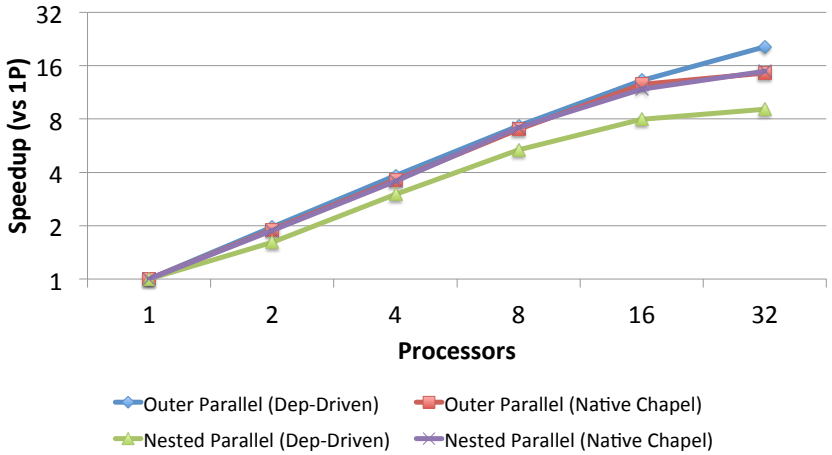
3–D Jacobi Method

Figure 3.18 shows the effect on processor scalability as the problem size increases for the 3–D Jacobi Method. For this result, it is obvious that scalability is limited, especially when the problem size is small (e.g. $128 \times 128 \times 128$). There are three important limiting factors that cause this performance degradation. First, since the outer loop of this kernel is a time step, naively, this loop will not be parallelized due to loop-carried dependences. A second reason for the performance degradation is due to the implicit barriers at the end of the parallel loops. Section 3.6 and Chapter 4 will look at new ways of decreasing the number of barriers, or removing them completely. In addition to the stencil, there are two other parallel operations inside of this kernel: a parallel reduction to find the minimum value (line 24 in Figure 3.1), and an array swap performed in parallel (line 26 in Figure 3.1). Just as before, these loops also include implicit barriers at the end of them. The third limiting factor is due to the overhead of spawning additional tasks and the associated data management between them. This occurs because a new task is spawned on every outer loop iteration, and an inner barrier task in the case of nested parallelism. Also, any data that is used inside of the newly spawned tasks needs to be copied through the coherence mechanism. This last limiting factor does not occur in the native Chapel implementation.

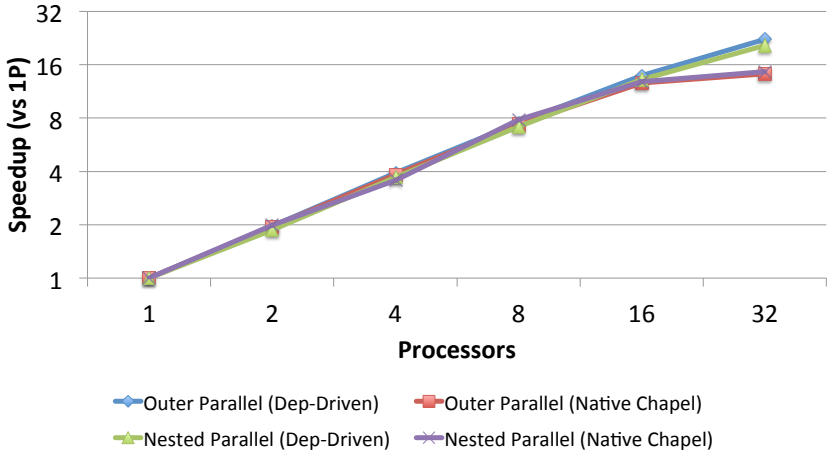
When comparing the outer loop parallel scalability between the dependence-driven implementation and that of the native Chapel implementation, both scale similarly. On the other hand, in the case of nested parallelism, the dependence-driven approach does worse than the native Chapel implementation on smaller input sizes. This difference in performance is due to the overheads of spawning additional tasks and performing data management, which is



(a) $128 \times 128 \times 128$



(b) $256 \times 256 \times 256$



(c) $512 \times 512 \times 512$

Figure 3.18: 3–D Jacobi Method Scalability for Nested-Parallelism : $1 \rightarrow 32$ Processors

not prevalent in the native Chapel implementation. However, as the input size increases, as shown in Figure 3.18c, the difference between both implementations is minimal.

As the maximum number of processors are used, there is a divergence between the two implementations. The reason that this occurs is that it is primarily a result of the underlying runtime system. Since the vast majority of the compiler generated code is the same between both implementations, the only major difference is the runtime systems used.

Sparse Matrix-Vector Multiplication (SpMV)

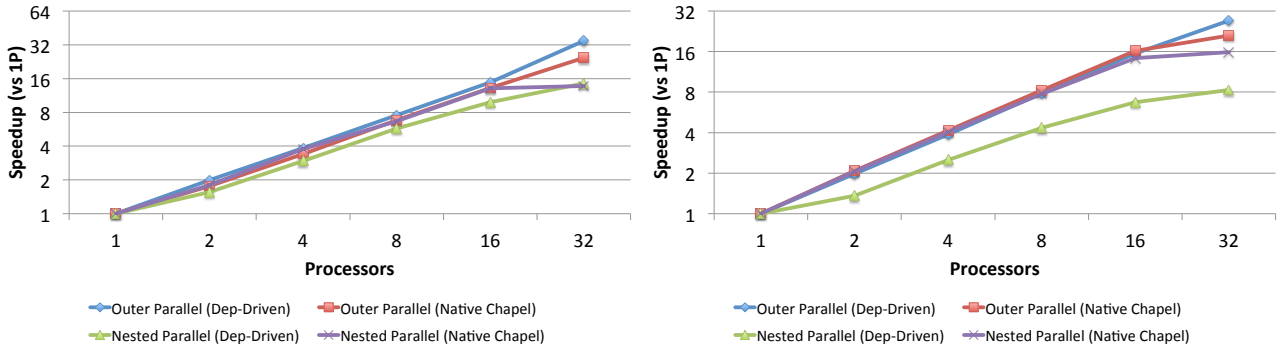
The graph in Figure 3.19 presents the scalability for a SpMV kernel applied onto a set number of vectors, which in this case is 16,384. Depending on which sparse matrix is used, the impact on scalability varies, with `Andrianov/ex3sta1` having the highest scalability, `Meszaros/ex3sta1` having the lowest, and the remainder of matrices in between.

When comparing the dependence-driven approach to that of the native Chapel implementation, both follow a similar scaling trend, with the exception of the dependence-driven nested-parallelism approach. Also, similar to before, the outer loop dependence-driven approach tends to scale at the higher spectrum compared to native Chapel.

Coulombic Potential (CP)

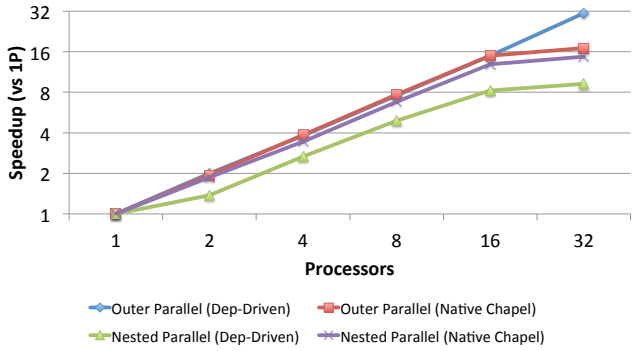
The scalability results for Coulombic Potential (CP) are presented in Figure 3.20. The highest amount of scalability ($\sim$30x speedup at 32 processors) occurs when the problem size is at its largest, which in this case is $\{512 \times 512 \times 512\}$. When the problem size is smaller, such as $\{64 \times 64 \times 64\}$, the amount of scaling diminishes ($\sim$14x speedup at 32 processors). This is due to amount of overhead compared to the amount of useful work in the kernel for the given data size.

Comparing the dependence-driven approach to that of the native Chapel implementation, there is a drop off in scalability going from $16 \rightarrow 32$ processors in the case of the native Chapel

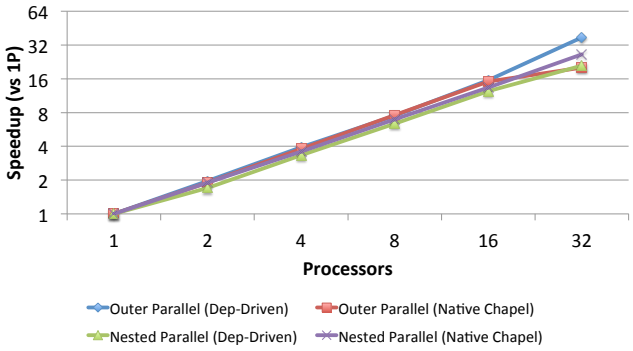


(a) Meszaros/stat96v5

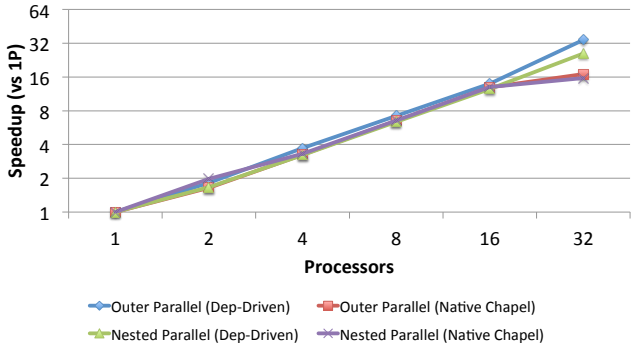
(b) Meszaros/ex3sta1



(c) Rommes/bips07_2476

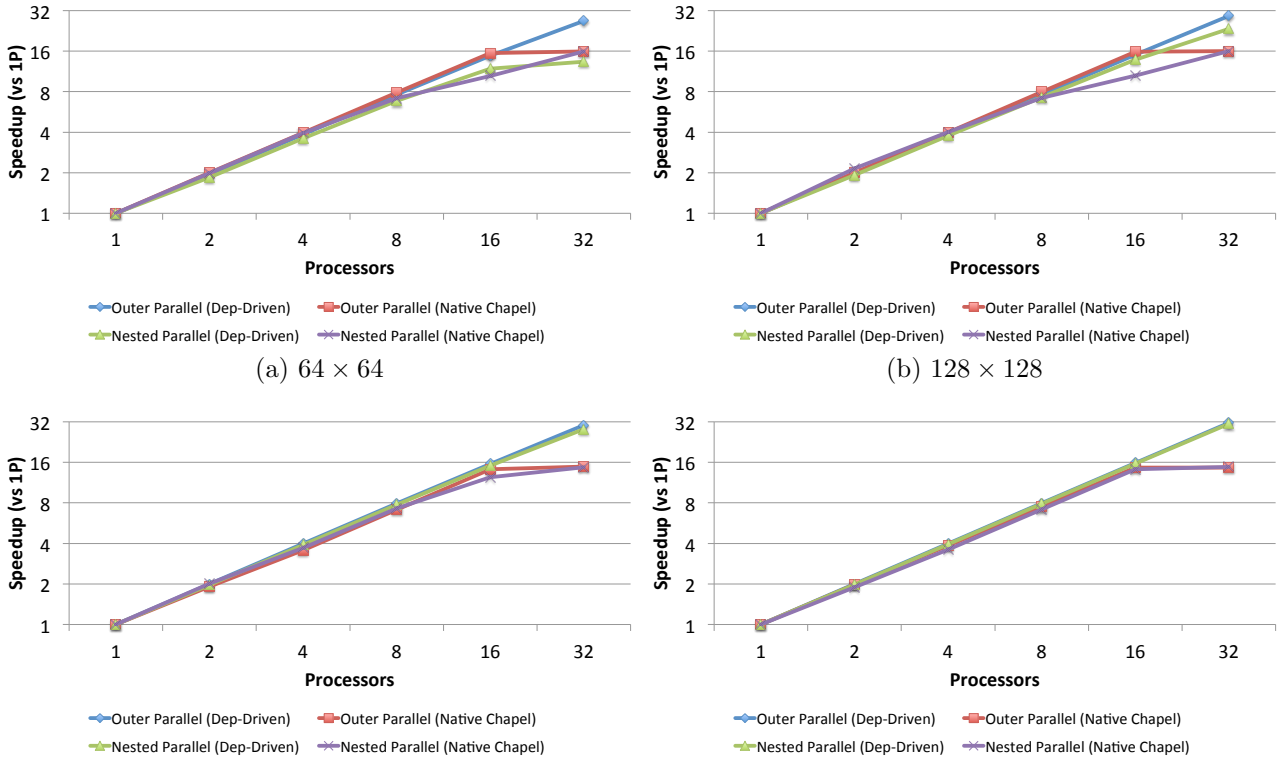


(d) Andrianov/ex3sta1



(e) LPnetlib/lp_osa_14

Figure 3.19: Sparse-Matrix Vector Multiplication Scalability for Nested-Parallelism with Multiple Vectors ($v = 16,384$)



(a) 64×64

(b) 128×128

(c) 256×256

(d) 512×512

Figure 3.20: Coulombic Potential (CP) Scalability for Nested- and Outer-Parallelism : $1 \rightarrow 32$ Processors

implementation. This does not occur when targeting the StarPU runtime system.

Magnetic Resonance Imaging FHd (MRI-FHd)

Another benchmark from the Parboil is the MRI-FHd application. Figure 3.21 presents the overall scalability when using this benchmark. In this application, two input sizes were tested, where the larger input size $\{64 \times 64 \times 64\}$ scales slightly better ($\sim$24x speedup at 32 processors) compared to the smaller input size ($\sim$22x speedup at 32 processors).

Both the dependence-driven and the native Chapel approaches perform similar with a slight advantage to the dependence-driven technique as throughout the processor range for the larger input size.

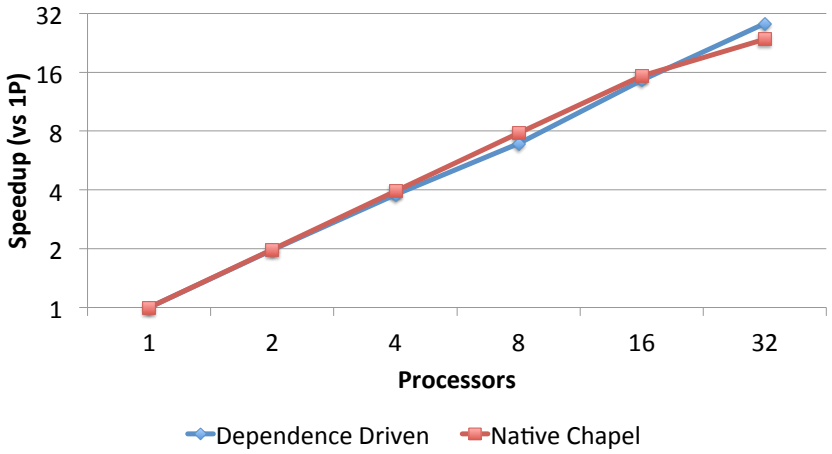
Magnetic Resonance Imaging Q (MRI-Q)

The results for MRI-Q are provided in Figure 3.22. The results are very similar to those previously seen from MRI-FHd. The application scales, and with with the exception at the high end, both the dependence-driven and native Chapel approaches are similar in performance.

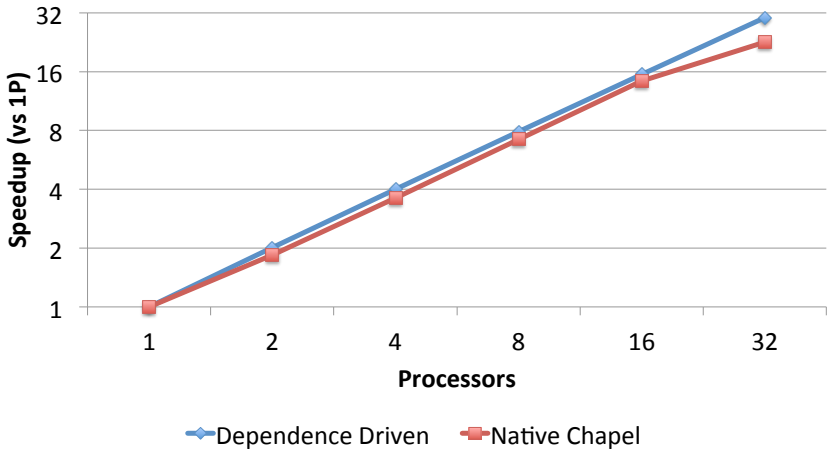
2–D Image Histogram Computation

Histogram scalability results are provided in Figure 3.23. In this benchmark, scalability was limited ($\sim$14x speedup at 32 processors) when the size of the image is small $\{4,096 \times 4,096\}$, and the program scales slightly better ($\sim$23x speedup at 32 processors) as the image size increases $\{16,384 \times 16,384\}$. Similar to the ongoing trend from the previous results, the outer loop parallel scenario performs better than the nested case.

When comparing the dependence-driven approach to that of native Chapel, their performance is comparable to each other, except in the case of nested parallelism. When dealing with nested parallelism for the dependence-driven runtime, the major performance degradation occurs due to the amount of additional overhead introduced compared to the amount of

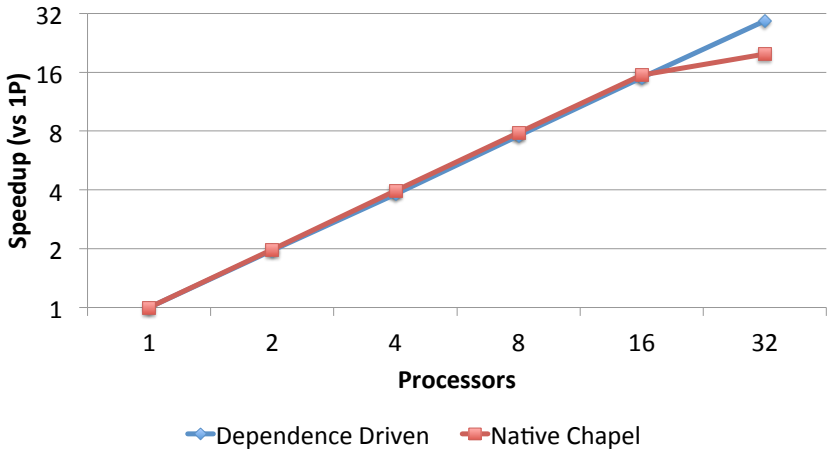


(a) $32 \times 32 \times 32$

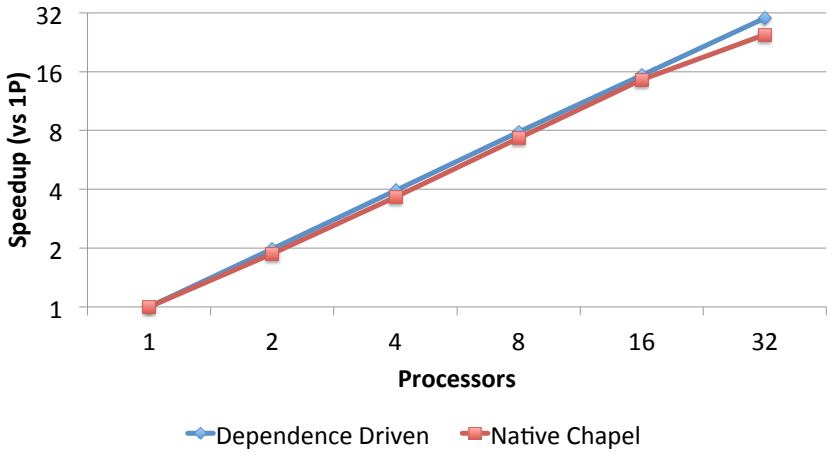


(b) $64 \times 64 \times 64$

Figure 3.21: MRI-FHd Scalability With Single-Level Parallelism : $1 \rightarrow 32$ Processors

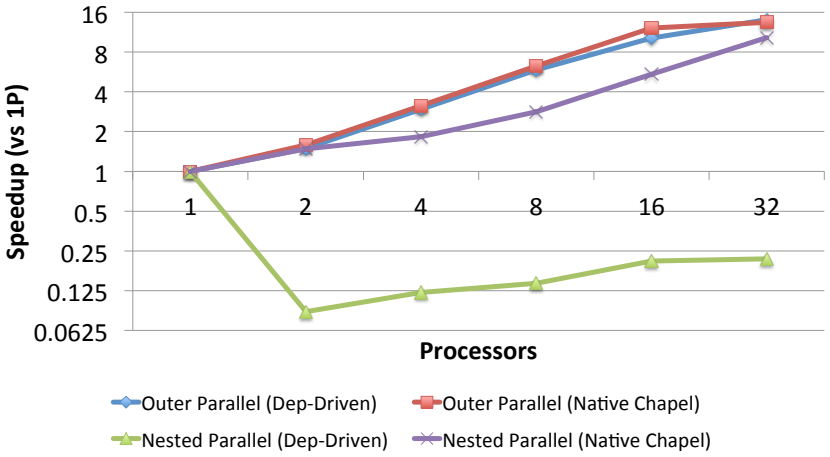


(a) $32 \times 32 \times 32$

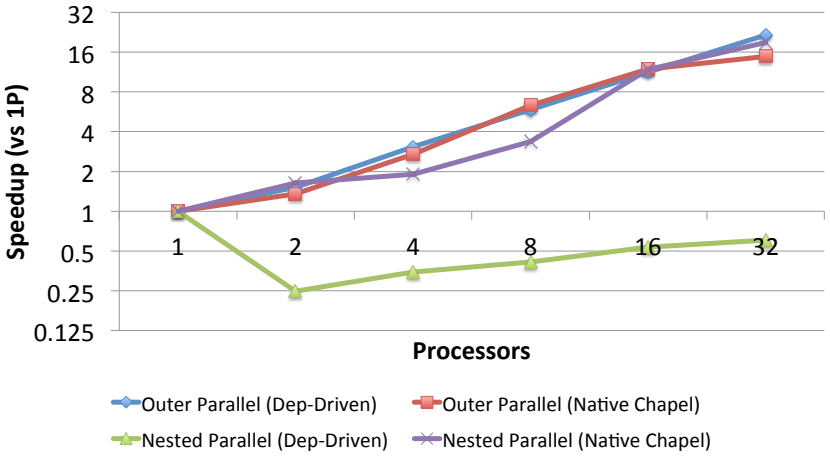


(b) $64 \times 64 \times 64$

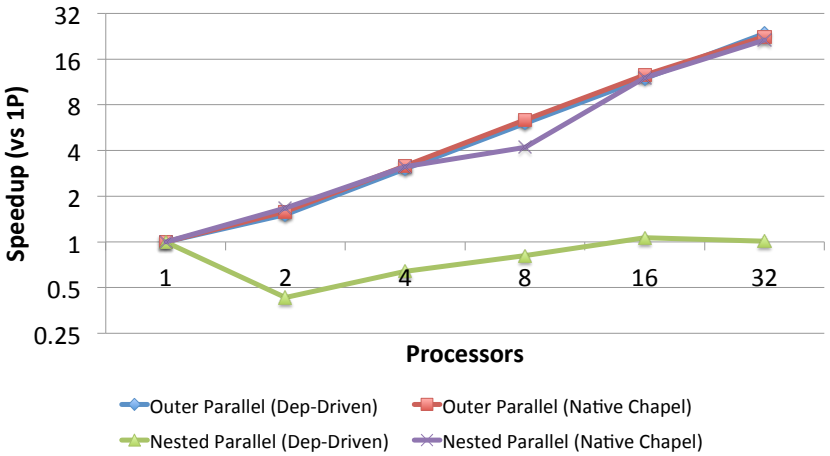
Figure 3.22: MRI-Q Scalability With Single-Level Parallelism : $1 \rightarrow 32$ Processors



(a) $4,096 \times 4,096$



(b) $8,192 \times 8,192$



(c) $16,384 \times 16,384$

Figure 3.23: 2–D Image Histogram Computation Scalability : $1 \rightarrow 32$ Processors

actual computation that occurs inside of the main parallel loop. The computation happens to be simply an integer increment. Because the overhead of nested parallelism in Chapel is not as high, the overall impact on performance in the native Chapel version does not suffer as much as the dependence-driven approach. Chapter 4 will discuss optimizations that can help alleviate some of the overhead.

Synthetic.Trig

The last benchmark that is evaluated is the Synthetic.Trig application. This is a purely compute-bound multiply-nested parallel kernel. As the results show in Figure 3.24, the program scales very well. For example, at a larger input size of 32K inner-iterations, there is a speedup of ∼31x at 32 processors, and a speedup of ∼21x at 32 processors when the input is set to a smaller size of 1K inner-iterations.

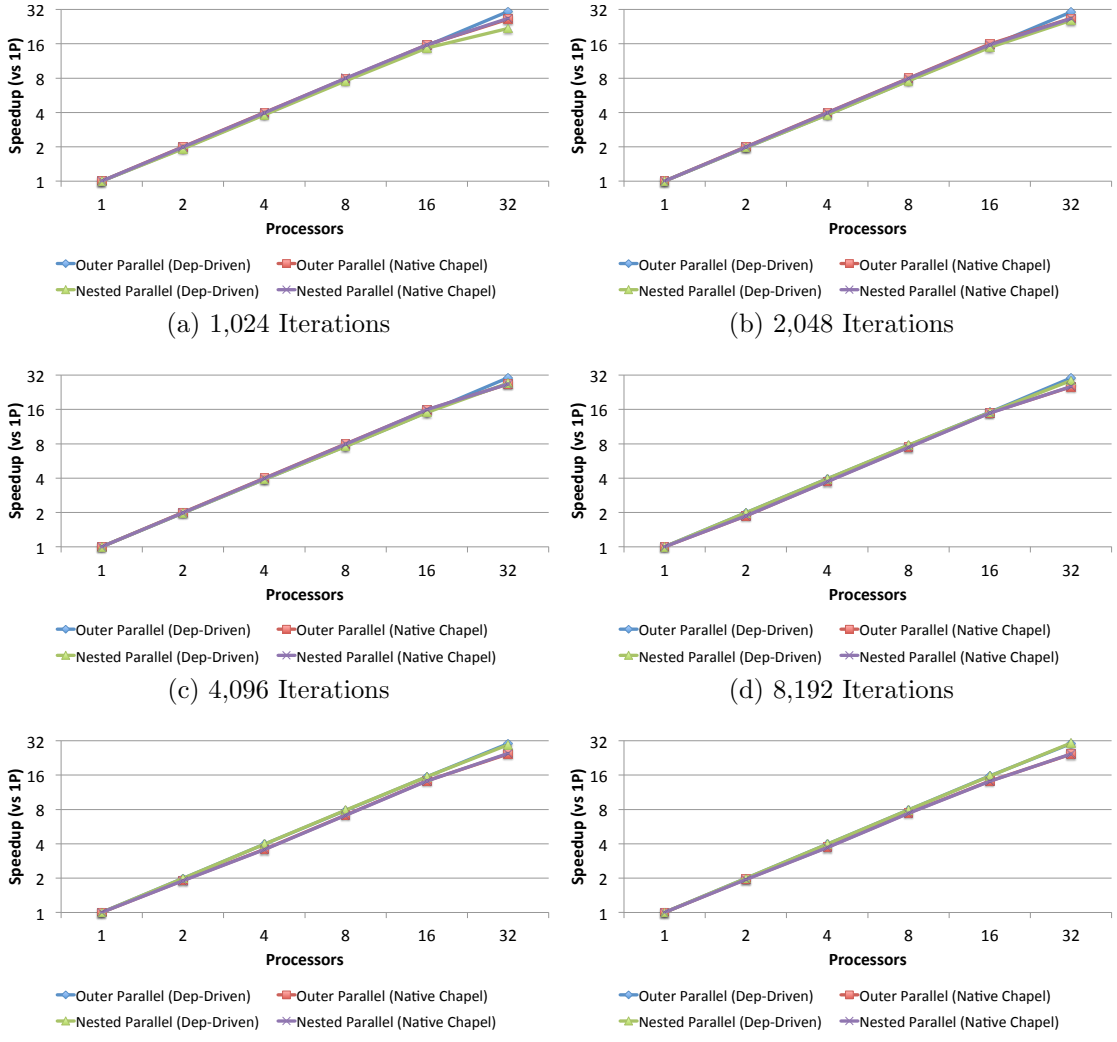
Similar to before, the dependence-driven approach scales slightly better than the native Chapel implementation mostly due to the StarPU's task scheduler being more efficient.

3.5 Supporting Non-Loop Based Parallel Constructs

This section deals with supporting more than just the compilation of data-parallel loops into the task-dependence graph model. In particular, this proposes a way of supporting the task-parallel constructs of Chapel to also use the dependence-driven compilation techniques described earlier, including partitioning the flow graph into an interval graph and its ICT so that generated tasks can be scheduled asynchronously by the runtime.

Consider the parallel construct `cobegin`. In a `cobegin` block, every statement is spawned as a separate task. The end of the `cobegin` represents an implicit join that waits for all the concurrent statements to complete execution before the master thread proceeds.

The general approach to translating a `cobegin` into the dependence-driven model is



(a) 1,024 Iterations

(b) 2,048 Iterations

(c) 4,096 Iterations

(d) 8,192 Iterations

(e) 16,384 Iterations

(f) 32,768 Iterations

Figure 3.24: Synthetic.Trig Scalability for Nested-Parallelism : $1 \rightarrow 32$ Processors

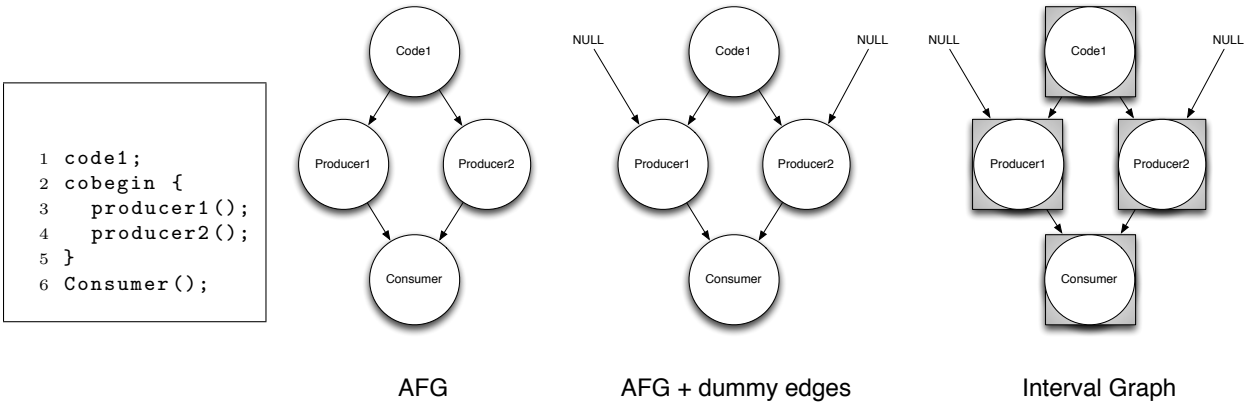


Figure 3.25: Translating a Task-Parallel `cobegin` into an Interval Graph

similar to the techniques of computing the agglomerated flow graph and then performing interval analysis on the AFG. Since every statement represents a task, the statement will be its own agglomerated basic block with an incoming edge from the header of the `cobegin` block, and an outgoing edge to the implicit join at the end of the `cobegin`. The subtle difference is that since there is only a single predecessor edge to each of the statements in the `cobegin`, this would complicate interval analysis due to the algorithm incorporating the newly created nodes into the existing interval. Thus, the entire `cobegin` block would collapse into a single interval which would prevent any expected parallelism. In order to keep the `cobegin` statements in separate interval nodes, a temporary artificial edge or *dummy edge* to each `cobegin` statement is inserted. Now that each node will have more than one predecessor, the nodes will not be collapsed together as a result of interval analysis. After interval analysis has been performed and the interval nodes have been generated, the *dummy edges* are removed. As an example, consider the producer-consumer program in Figure 3.25. This shows the graphical intuition for the approach of preventing `cobegin` statements from being collapsed together. After the interval graph has been generated, and the dummy edges have been removed, the remainder of the compilation process is the same as before in Section 3.3.

3.6 Language Support for Arbitrary Execution Order

The work presented earlier in this chapter assumes that parallelism is invoked explicitly through means such as `forall` loops. The downside to this approach is that it introduces barriers which are too coarse-grain. These barriers fail to express point-to-point dependences (both data and control) between program statements, which will then limit program performance. As an alternative to this, there are programming languages and runtime systems provided where one can declaratively express tasks and their point-to-point dependences, thus in effect, explicitly constructing a task-dependence graph. Examples of this include Intel Concurrent Collections (CnC) [16] or the StarPU runtime system described earlier. However, the problem with these programming models are that the vast majority of programs are written in an imperative style which differs from the declarative programming models used by CnC or StarPU. Judging by the existing programming languages available, programmers have historically preferred imperative languages. This will then limit overall programmer acceptance of these declarative-based programming models.

In order for program correctness to be maintained, a compiler has to be conservative in the optimizations that it applies. As a result of this, a program's data dependences can then have a limiting effect on which of these optimizations are applied. This includes potentially limiting loop transformations that change the order of execution in order to increase a program's locality or parallelism. The problem is that, in many cases, the compiler might not have enough information to correctly determine the program's dependences, possibly because dependences could be a function of the input, there is a lack of alias information due to pointers, or no compile-time information about external third-party libraries. Rather than the compiler trying to automatically compute these dependences and possibly failing to find a legal transformation, if it were provided dependence information explicitly in the form of loop ordering constraints, it would have the freedom and flexibility to do any optimization

that was legal within these imposed constraints.

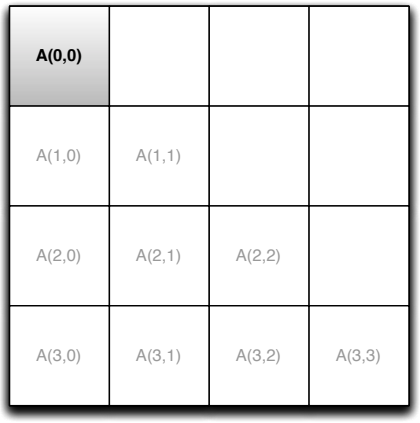
```
for k in 0..TILES-1 {
  A(k,k) = DPOTF(A(k,k))

  for i in k+1..TILES-1 {
    A(i,k) = DTRSM(A(k,k),A(i,k))
  }
  for j in k+1..TILES-1 {
    A(j,j) = DSYRK(A(j,k),A(j,j))
    for i in j+1..TILES-1 {
      A(i,j) = DGEMM(A(i,k),A(j,k),A(i,j))
    }
  }
}
```

Figure 3.26: Tiled Cholesky Factorization

For example, consider the Tiled Cholesky Factorization ($A = LL^T$) program in Figure 3.26. Cholesky factorization is a method of decomposing a symmetric positive-definite $n \times n$ matrix A into a lower triangular $n \times n$ matrix L and its transpose L^T, such that $A = LL^T$. Cholesky factorization can be used to solve systems of linear equations, and is known to be more efficient than LU factorization. The tiled implementation of Cholesky consists of the following four kernel operations: `DPOTF`, `DTSRM`, `DSYRK`, and `DGEMM`. `DPOTF` performs a Cholesky factorization of the diagonal tile $A(k, k)$. `DTSRM` performs a triangular system solve down a column of tiles using the diagonal component computed in `DPOTF`. `DSYRK` performs a symmetric rank-k update onto the diagonal tile. `DGEMM` performs a matrix-matrix multiplication on the off-diagonal components.

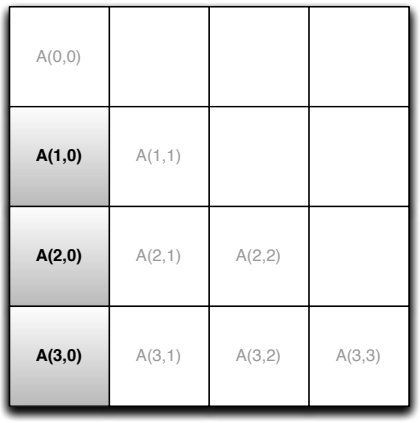
The three main steps of the algorithm are given in Figures 3.27a–3.27c. The first step of the algorithm is to do a block-level Cholesky factorization on the A(0,0) tile. Next, using the newly computed A(0,0) tile, a triangular system solve down the column is performed on tiles A(1,0)–A(3,0). Finally, a symmetric rank-k update is performed across the rows 1–4. On the next iteration of k, the process starts all over again using the adjacent column of tiles to the right. Since more than likely the compiler does not have a priori knowledge of what data

```
1 for k in 0..TILES-1 do {
2   A(k,k) = DPOTF(A(k,k));
3
4   for i in k+1..TILES-1 do
5     A(i,k) = DTRSM(A(k,k), A(i,k));
6
7   for j in k+1..TILES-1 do {
8     A(j,j) = DSYRK(A(j,k), A(j,j))
9
10    for i in j+1..TILES-1 do
11      A(i,j) = DGEMM(A(i,k),A(j,k),A(i,j));
12  }
13 }
```



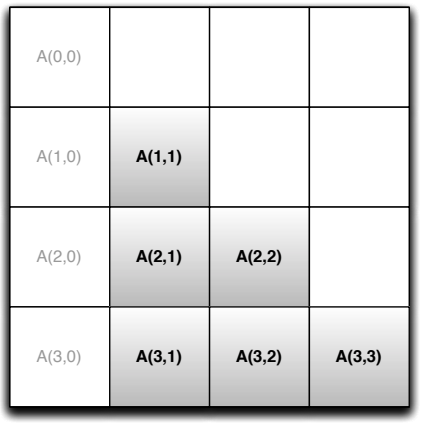
(a) Block Factorization of A(0,0)

```
1 for k in 0..TILES-1 do {
2   A(k,k) = DPOTF(A(k,k));
3
4   for i in k+1..TILES-1 do
5     A(i,k) = DTRSM(A(k,k), A(i,k));
6
7   for j in k+1..TILES-1 do {
8     A(j,j) = DSYRK(A(j,k), A(j,j))
9
10    for i in j+1..TILES-1 do
11      A(i,j) = DGEMM(A(i,k),A(j,k),A(i,j));
12  }
13 }
```



(b) Triangular Solve of A(1:3,0) using A(0,0)

```
1 for k in 0..TILES-1 do {
2   A(k,k) = DPOTF(A(k,k));
3
4   for i in k+1..TILES-1 do
5     A(i,k) = DTRSM(A(k,k), A(i,k));
6
7   for j in k+1..TILES-1 do {
8     A(j,j) = DSYRK(A(j,k), A(j,j))
9
10    for i in j+1..TILES-1 do
11      A(i,j) = DGEMM(A(i,k),A(j,k),A(i,j));
12  }
13 }
```



(c) Symmetric Rank-K update using A(1:3, 0)

Figure 3.27: Sequential Tiled Cholesky Factorization

is modified within the different LAPACK and BLAS routines, it has to be conservative on the loop transformations that it can apply. If the compiler had this knowledge, it could, for example, start pipelining computation across different iterations of k. If $k = 1$, then `DPOTF` can start execution concurrently as soon as `DSYRK` has completed execution in iteration $k = 0, j = 1$.

Main Goal The work in this section defines a language extension to the Chapel programming language in order to explicitly specify the legal ordering constraints that can occur between program statements. The specified ordering of execution is user-provided, rather than relying on the compiler to try and determine this automatically by computing control- and data-dependences. This is done so that programmers can still write applications in a loop-based imperative style, rather than relying on a declarative form as required by existing dependence-driven systems. There are two main venues of research that the work in this section will lead to. First, this would permit the compiler to perform loop transformations that would have been prevented earlier due to the possibility of a data dependence. This includes the large assortment of loop transformations [43] that require dependence analysis to have been performed. Second, loop constructs (both sequential and parallel) can now be generalized where a single notation can express different forms of parallelism, including data parallelism, pipeline parallelism, and task parallelism.

Besides the optimization potential from the compiler, these extensions allow programmers to incrementally parallelize their applications, similar to the popularity of directives as used in OpenMP. It should be noted that this technique is fully complimentary to the existing compilers used today that rely on dependence analysis. If the compiler is not able to precisely compute dependences for various reasons, it can *fall back* onto the proposed method in order to still perform its transformations.

3.6.1 Language Extensions

This section describes the proposed language extensions which can be used to specify the ordering constraints between statements. One point of this exercise is to maintain high productivity by finding a minimal set of necessary extensions to support the ability of providing the compiler enough dependence information.

Prerequisite

The semantics of the statements in the language remain the same. That is, the execution of loops and statements stay the same by executing sequentially. Parallelism is induced explicitly through the existing mechanisms of `[co]begin` and `[co]forall`.

```
1 if (...) then {
2   statement1;
3   statement2;
4   statement3;
5 } else {
6   statement4;
7   statement5;
8   statement6;
9 }
```

```
1 if (...) then {
2   begin statement1;
3   begin statement2;
4   begin statement3;
5 } else {
6   begin statement4;
7   begin statement5;
8   begin statement6;
9 }
```

Figure 3.28: Implicit versus Explicit Parallel Model

An alternative to this approach would be to have every statement implicitly parallel. That is, the only method to enforce a particular ordering would be through the proposed extension or by marking a block of code as `serial`. Consider the simple example in the leftmost block of Chapel code in Figure 3.28. If the assumption is that every statement in a compound statement sequence can execute in parallel with the others, then the equivalent in the non-implicitly parallel scenario is the rightmost block of Chapel code where every statement is spawned as a separate task with the asynchronous Chapel `begin` construct. Here, every statement would be fully parallel with no particular constraint on their ordering. While this alternative approach could result in less cluttered code, it is not pursued

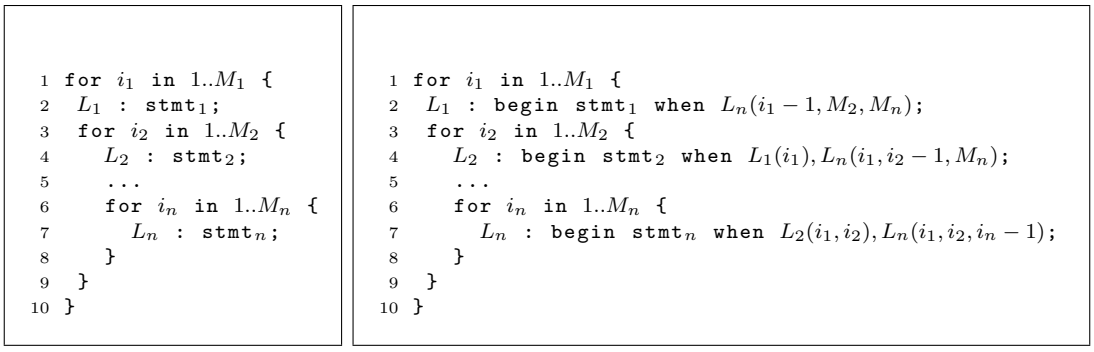


Figure 3.29: Loop Nest: Before and After Language Extension

since it would effectively alter the semantics of the language, rather than compliment the existing programming model. One could also argue that programmers are worse at implicit parallelism relative to explicit.

The `when` Language Statement

Consider the loop nest on the left in Figure 3.29. The loop has an iteration vector $\vec{i}$ that represents a particular loop instance $(i_1, i_2, \ldots, i_n)$. The iteration vector $\vec{i}$ makes up the total iteration space $\vec{I}$ as follows:

$$\vec{i} \in \vec{I} = \{(i_1, \ldots, i_n) | 1 \leq i_1 \leq M_1, \ldots, 1 \leq i_n \leq M_n; i_1, \ldots, i_n \in \mathbb{Z}\} \tag{3.1}$$

The main language extension comes from keyword `when` :

$$L : S \texttt{ when } L_1(i_1), [L_2(i_1, i_2), \ldots, L_n(i_1, i_2, \ldots, i_n)] \tag{3.2}$$

The `when` extension specifies that in order to execute a given statement S at label L, it must wait for *all* of the statements specified by labels $L_1[, L_2, \ldots, L_N]$ at a particular instance in the loop iteration vector $\vec{i}$ to have completed. Any statement using the `when` annotation that is prepended with a `begin`, will be asynchronously pushed onto a queue of tasks. It is

then the job of the task-scheduler to start execution of tasks once their ordering constraints have been resolved. In the case where the `begin` instruction is not used, statement S will block until its dependences have been resolved. If a statement is performing a `when` on a particular ordering constraint that has not been encountered yet, that ordering constraint will be ignored for this instance. This approach was taken so that initial statements can start and make progress without cluttering the program with numerous constraint conditionals.

In the example on the right of Figure 3.29, there are three `when` statements. The statement at L_1 in instance i_1 can start execution as soon as all L_n have completed from the previous iteration (i.e. L_n has completed iteration $(i_n - 1, M_2, M_n)$). The statement at L_2 can start execution as soon as L_1 finishes in the current iteration of i_1, and both L_2 and L_n from the previous iteration of i_1 have completed. Lastly, statement L_n can start execution as soon as L_2 in the current (i_1, i_2) iteration completes, and L_n from the previous iteration of $(i_1, i_2, i_n - 1)$ has fully finished.

```
1  for i1 in M1..M2 {
2    L1 : begin stmt1 when L1(i1 − 1), L2(i1 − 1);
3    L2 : for i2 in 1..M2 when L1(i1) {
4              begin stmt2;
5    }
6    ...
7    for in in 1..Mn {
8      Ln : begin stmtn when L1(i1), L2(i1 − 2);
9    }
10 }
```

Figure 3.30: Using a `when` Statement Across a Block

The granularity of the statement(s) can be of variable size. In addition to S being a single statement, it can also be a loop or an arbitrary block of code. For example, as shown in Figure 3.30 the loop at L_2 can start execution as soon as the previous statement at L_1 has completed execution. Additionally, if the statement is dependent on the completion of an entire loop (as in L_n), the provided extensions are flexible enough to allow this. In this example, the statement L_n can start execution as soon as statement L_1 in the same iteration

i_1 and statement L_2 two iterations ago (i.e. $i_1 - 2$) have completed.

```
1 for i1 in 100..200 {
2   L1 : begin stmt1 when L1(i1 − 1);
3   for in in 1..Mn {
4     L2 : begin stmtn when L1(100..150);
5   }
6 }
```

Figure 3.31: Using a `when` Statement Across a `domain`

A named subset of indexes, known as a `domain` in Chapel, can also be used in place of the scalar loop *index* in order to specify the ordering constraint on a set of instances. Rather than wait for a single instance of a statement to complete, a set of loop instances can be used. In Figure 3.31, the statement at L_2 will not start execution until L_1 has completed execution for iterations 100–150.

3.6.2 Examples

This section provides two short examples written in Chapel that utilize the `when` language extension. In addition to the example source code, the respective *task-dependence graph* that is generated will be presented.

Tiled Cholesky Factorization: $A = LL^T$

Figure 3.32a shows an asynchronous implementation of a tiled Cholesky factorization kernel and Figure 3.32b is the associated unrolled task-graph using a 4×4 tiled grid. This algorithm is asynchronous due to each statement being asynchronously spawned as a task using the `begin` instruction. The red portions of the example represent the new language extensions necessary to express the ordering constraints between statements. As the task-dependence graph shows, once the first instance of `DPOTF` completes, three instances of `DTRSM` can execute in parallel. Dependent on which `DTRSM` completes, the next set of calls can be made to

either DSYRK or DGEMM. The next instance of DPOTF can be invoked as soon as the first instance (based on *i,j*) of DSRK completes. This pattern proceeds until the program eventually completes.

It is important to note that similar to the progressive flexibility of introducing/removing parallelism offered by OpenMP, if these language extensions (and the respective `begin`) were to be removed, the program would still be functionally correct.

Tiled QR Factorization: $A = QR$

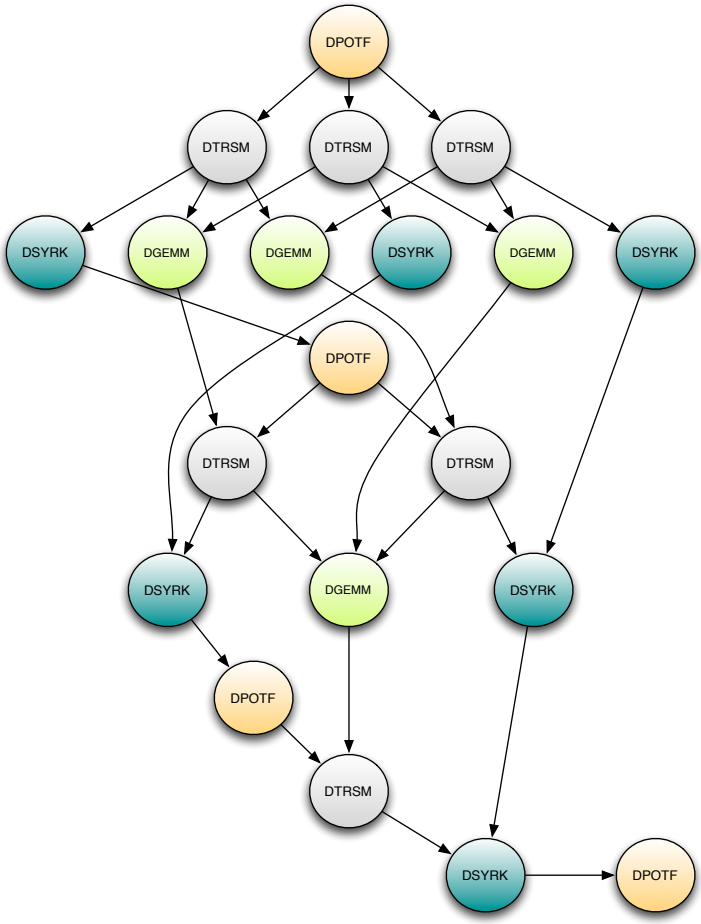
QR factorization is a stable method of decomposing an $m \times n$ matrix A into the product QR, where Q is a $m \times m$ orthogonal matrix, and R is a $m \times n$ upper triangular matrix. QR factorization is commonly used to solve linear least squares and also can be used in computing eigenvalues.

Figure 3.33 shows a tiled QR factorization implementation using the following kernel operations: DGEQRT, DTSQRT, DORMQR, and DSSMQR. DGEQRT performs a QR factorization on a diagonal tile. The output consists of A and T. A is made up of the upper triangular matrix R, and the lower triangular matrix V which contains the Householder reflectors. T is made up of the upper triangular block reflectors which are stored in compact form [73]. DTSQRT performs a QR factorization using the R matrix computed in DGEQRT or from a previous call to DTSQRT, and a tile below the matrix diagonal. This subroutine produces an updated R matrix, a square matrix V containing the Householder reflectors, and T. DORMQR uses the transformations computed in DGEQRT to apply to the tile on the right of the diagonal. DSSMQR applies the reflectors V and matrix T that were both computed in DGEQRT to the two tiles that were factorized by DGEQRT.

The respective unrolled task-dependence graph for 4×4 tiled QR factorization is provided in Figure 3.34b. If the user (or compiler) were to compute the dependences for this purely based on the loop indices, a task-graph can be formed as shown in Figure 3.34a.

```
for k in 0..TILES-1 {
  S1: begin A(k,k) = DPOTF(A(k,k)) when S3(k-1,k)

  for i in k+1..TILES-1 {
    S2: begin A(i,k) = DTRSM(A(k,k),A(i,k))
                              when S1(k),S4(k-1,k,i)
  }
  for j in k+1..TILES-1 {
    S3: begin A(j,j) = DSYRK(A(j,k),A(j,j))
                              when S2(k,j),S3(k-1,j)
    for i in j+1..TILES-1 {
      S4: begin A(i,j) = DGEMM(A(i,k),A(j,k),A(i,j))
                    when S2(k,i),S2(k,j),S4(k-1,j,i)
    }
  }
}
// barrier
```

(a) Cholesky Factorization With `when` Ordering Constraints



(b) Task-Dependence Graph for Cholesky Factorization

Figure 3.32: Asynchronous Cholesky Factorization

```
for k in 0..TILES-1 {
  (A(k,k),T(k,k)) = DGEQRT(A(k,k))
  for i in k+1..TILES-1 {
    (A(k,k),A(i,k),T(i,k)) = DTSQRT(A(k,k),A(i,k),T(i,k))
  }
  for j in k+1..TILES-1 {
    A(k,j) = DORMQR(A(k,k),T(k,k),A(k,j))
    for i in k+1..TILES-1 {
      (A(k,j),A(i,j)) = DSSMQR(A(i,k),T(i,k),A(k,j),A(i,j))
    }
  }
}
```

Figure 3.33: Tiled QR Factorization

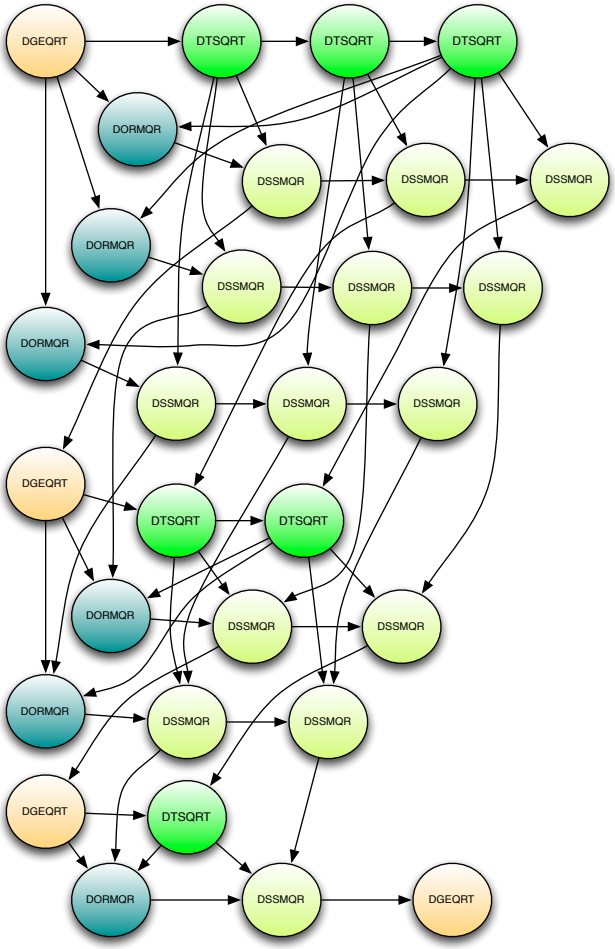
Optimized Tiled QR Factorization

As mentioned earlier, in many cases a compiler can have trouble extracting all the possible dependences statically, either due to input dependences, non-affine indices, aliasing, or linked-in libraries. In the QR factorization example shown earlier, calls are made to various BLAS subroutines. More often than not, the compiler would not be aware of what data structures are modified internally as part of the linked-in third party BLAS routines. Functionally, `DORMQR` only uses the lower triangular portion of $A(k,k)$ (i.e. $(V(k,k))$, whereas `DTSQRT` will only output to the upper triangular portion of $A(k,k)$ (i.e. $R(k,k)$). Thus, there is no overlap between the two that could cause a dependence violation. Naively, the compiler would determine there was a dependence between the two even though there is no overlap in the portions of the array being read and written to.

Since the programmer now has a way of expressing the ordering constraints explicitly to the compiler, they can safely remove any constraint between these two statements. Figure 3.35 shows an implementation of QR factorization with the dependence between `DORMQR` and `DTSQRT` explicitly ignored. By removing (or relaxing) this edge between the two, the calls to `DORMQR` can start immediately as soon as `DGEQRT` (in the current iteration) has completed, thus increasing parallelism.

```
for k in 0..TILES-1 {
  S1: begin (A(k,k),T(k,k)) = DGEQRT(A(k,k)) when S5(k-1,k,k)

  S2: for i in k+1..TILES-1 {
    S3: begin (A(k,k),A(i,k),T(i,k)) = DTSQRT(A(k,k),A(i,k),T(i,k))
                                       when S1(k),S3(k,i-1),S5(k-1,k,i)
  }
  for j in k+1..TILES-1 {
    S4: begin A(k,j) = DORMQR(A(k,k),T(k,k),A(k,j))
                       when S1(k),S2(k),S5(k-1,j,k)
    for i in k+1..TILES-1 {
      S5: begin (A(k,j),A(i,j)) = DSSMQR(A(i,k),T(i,k),A(k,j),A(i,j))
                         when S3(k,i),S4(k,j),S5(k-1,j,i),S5(k,j,i-1)
    }
  }
}
// barrier
```

(a) QR Factorization With `when` Ordering Constraints

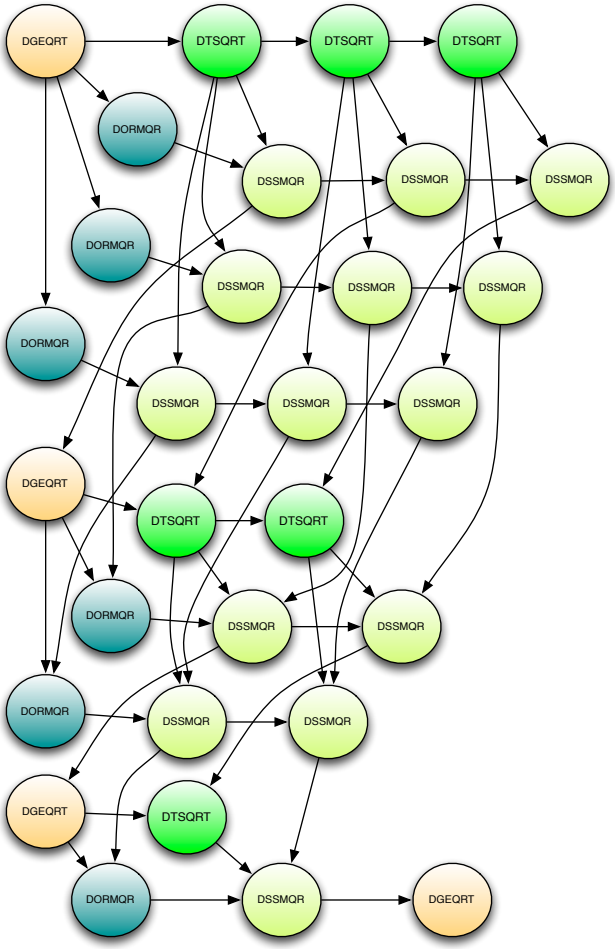


(b) Task-Dependence Graph for QR Factorization

Figure 3.34: Asynchronous QR Factorization

```
for k in 0..TILES-1 {
  S1: begin (A(k,k),T(k,k)) = DGEQRT(A(k,k)) when S4(k-1,k,k)

  for i in k+1..TILES-1 {
    S2: begin (A(k,k),A(i,k),T(i,k)) = DTSQRT(A(k,k),A(i,k),T(i,k))
                                       when S1(k),S2(k,i-1),S4(k-1,k,i)
  }
  for j in k+1..TILES-1 {
    S3: begin A(k,j) = DORMQR(A(k,k),T(k,k),A(k,j))
                       when S1(k),S4(k-1,j,k)
    for i in k+1..TILES-1 {
      S4: begin (A(k,j),A(i,j)) = DSSMQR(A(i,k),T(i,k),A(k,j),A(i,j))
                         when S2(k,i),S3(k,j),S4(k-1,j,i),S4(k,j,i-1)
    }
  }
}
// barrier
```

(a) QR Factorization With `when` Ordering Constraints



(b) Task-Dependence Graph for QR Factorization

Figure 3.35: Optimized QR Factorization With Fewer Dependences

3.7 Current Limitations

The main limitation from the proposed techniques on compiling data-parallel loops for demand-driven runtime systems is that it does not fully exploit the maximum asynchrony these runtime systems offer. This is due to the imposed implicit join at the end of all `forall` loops. While one could prefix such `forall` loops with a `begin` to avoid the implicit join, it is still not general enough to handle all possible situations (e.g. pipeline-parallelism). The proposed language extension `when` is one method to address this, but this requires explicit input from the programmer. One could try to automatically construct the asynchronous task-dependence graph, but this still leads to problems related to the conservative nature of the compiler in preserving correctness.

While Chapter 5 addresses compilation of loops to execute on data-parallel many-core architectures, the work presented here is primarily concerned with targeting dependence-driven runtime systems that could map the tasks onto numerous architecture backgrounds. While the techniques proposed in this chapter are agnostic to the back-end architecture, the implementation is focused primarily on multi-core architectures. One area of future work would be to map these tasks onto other back end architectures. This would effectively combine the techniques presented here and those in Chapter 5.

One last limitation in the implementation is that it currently does not leverage Chapel's recent support for the LLVM compiler infrastructure. Utilizing LLVM would allow the generated code to have richer compiler optimization support than what it currently provided by Chapel right now.

3.8 Related Work

3.8.1 Macro-Dataflow Compilation

There has been previous research that has looked into compiling imperative programs into macro-dataflow form [74, 75, 31, 76]. For the vast majority of related work in this section, the overall goal has focused on extracting parallelism automatically from sequential applications and then generating task-graphs, whereas the goal of this thesis is to efficiently partition an existing parallel program into a task-dependence graph. This simplifies the job of the compiler by not requiring it to perform dependence-analysis, since the programmer has explicitly stated what is parallel versus what is not. In cases where asynchrony needs to be exploited, the programmer will describe the dependence constraints.

The work of Kasahara et al. [75, 77] has looked at the compilation of Fortran programs to the macro-dataflow based compiler and architecture named Oscar. They are able to decompose the program into a set of *macrotasks* into a *macroflow* graph, which is then converted into a *macrotask graph*. The techniques discussed in this thesis differ since the focus is only on partitioning explicitly parallel loops, whereas the work in Oscar looks to extract parallelism automatically from serial code by analyzing both control and data dependences. The work in this thesis does not need to perform any data dependence analysis to determine where parallelism exists. Additionally, in the task-dependence graph that is generated, the edges of the graph only specify the order of execution. In Oscar's macroflow graph, there are two forms of edges to represent data and control dependences between the macrotasks. Lastly, the nodes in the macroflow graph can be classified as either basic blocks, loops, or subroutines. Since the goal of this section is to explicitly partition parallel loops from the rest of the program, the nodes are partitioned into intervals that are more coarse than what is provided in Oscar's macroflow graph. In particular, the interval nodes are not restricted to the node type classification as used in Oscar. This coarsening of interval nodes could lead

to less overhead of the fine-grain tasks generated by the Oscar compiler.

Girkar and Polychronopoulos have presented work [74, 78, 79] on the automatic extraction of task-level parallelism from serial programs into a *hierarchical task graph* (HTG) intermediate representation. While there is a strong similarity between their hierarchical task graph IR [78] and the interval graph and associated ICT from this work, the technique to construct these graphs differ. Strongly connected regions [80] are used to construct the HTG, while interval analysis is used to construct the task graph. Their claim for not using interval analysis is that it does not support reducible loops. In real world applications, unstructured programs that lead to an irreducible program are extremely rare [26, 27]. Additionally the granularity of tasks from the HTG are much finer since they are trying to automatically parallelize serial code, while the work presented here does this on already explicitly parallel loops.

There has been work done by Sarkar [81] on a compile-time partitioning algorithm of the functional language Sisal for macro-dataflow architectures. Unlike the work presented in this thesis, the major component used in the partitioning is strictly based on an analytical performance model that takes into consideration both computation and communication costs.

Beck and Pingali [76] present a technique of compiling sequential Fortran programs into dataflow form. When the CFG has cycles in it, they use an interval decomposition of the CFG. An important difference between their work and the one from this thesis, is that they use a traditional fine-grain dataflow architecture that works on memory operands, as opposed to the coarser task-dependence model which works on larger and more coarse executable chunks. Additionally, as with the other approaches already discussed, this scheme does not deal with explicitly parallel programs.

Leveraging both control and data dependence, Cytron [31, 82] presents a method that automatically generates DAG parallelism from sequential programs. Similar to the approach used in this thesis, they construct an interval hierarchy. The two major differences between

the work presented here and theirs, is that first, they do not target a dependence-driven execution model, which is done in this thesis. However, their proposed techniques could potentially be applicable to dependence-driven systems. The second difference is that since they are trying to automatically parallelize serial programs, they honor data dependences in order to determine an ordering between the statements and iterations of loops in a program. In the approach presented here, this is not necessary as it is only focused on translating explicitly parallel loops where computing data dependences is not necessary.

3.8.2 Language Extensions to Express Dependences

There has been some work in providing language annotations to describe dependences. In Duran et al. [17], the OpenMP 3.0 `task` construct is extended with additional clauses that specify data dependence relationships between different tasks. In particular, these clauses specify whether a task has an input and/or output dependence on a particular variable. On the other hand, the clauses do not specify the order of execution. In the approach described in this thesis, specific data dependences are not required in order for task-dependence edges to be resolved. Instead, the user specifies the exact ordering constraints on a statement, basic block, or loop. Forward progress is made as soon as an ordering constraint has been resolved.

Cytron [31] presents a method to construct a DAG of constraints among processes at the same nesting level within a `cobegin...coend` pair. The work described in this thesis is more general purpose, since in addition to being able to place ordering constraints on `cobegin..coend` pairs, they can be placed on individual loop iterations, entire loops, or statements.

The DAGuE [12] project is a system focused on dense linear algebra. DAGuE consists of a runtime engine and a set of tools to compile a compact representation of a DAG. They have both a low-level and high-level language that one can program in. In the low-level

language, the programmer needs to provide the implementation of all of the low-level BLAS routines, along with the dependences associated with those routines. If the user programs in their high-level language, the dependences are not specified. The difference to this thesis is that in their high-level language, they leave it to the compiler to compute data dependences, which severely limits the types of applications that one can run on their system.

Padua [40] and Cytron [41] have introduced the concepts of the `doacross` loop. The doacross is a special form of a parallel loop where the parallelism is constrained due to forward loop-carried dependences. In order to enforce correct execution order, one would use `post(`*id*`)` and `wait(`*id*`)`. The language extension proposed in this thesis is similar, except that it is much more unstructured since the programmer can place ordering constraints onto any statement in the program, as opposed to the loop as used in `doacross`.

Work was done in ZPL on exploring methods of providing array language support for pipelining wavefront computations [83]. This is different than the language extensions proposed here, since the extensions work on more than just whole-array statements. Another difference is that in this case ZPL primarily focused on wavefront computations, whereas the work presented here tries to generalize ordering constraints for all computations.

The concept of a language *future* [84] or *promise* [85] provides a way of asynchronously binding the result of a computation with its caller. An important distinction between the techniques proposed here, is that using the language extension techniques, the ordering between a statement and its result is asynchronously bound to a statement at some specific point in the iteration space, whereas in the future, a statement will just wait for the result to occur before it can proceed.

Lastly, Ke et al. propose a method of doing safe parallel programming using dynamic dependence hints that specify possible dependences between tasks [86]. A major difference between the proposed solution here is that they rely on software speculation to synchronize between tasks [86], whereas the proposed work does not.

3.9 Discussions

This chapter presented new methods of partitioning an already existing imperative parallel program into a task-dependence graph using compiler techniques such as agglomerated flow graphs, interval analysis, and interval containment trees. These techniques work for both single and multi-nested data-parallel loops. Since this work is effectively translating a program with global state to one that is stateless, a new algorithm was developed to efficiently place data and communicate it an interval containment tree. In addition to parallel loops, a method was developed to support non-loop parallel constructs. Lastly, in order to deal with the limitations that implicit barriers have on performance and to fully extract the asynchrony offered by demand-driven runtime systems, a new language extension was developed that allows a programmer to explicitly provide to a compiler the ordering constraints between different statements. This allows the compiler to either use this in conjunction with or replace its own dependence analysis, to perform different transformations.

Chapter 4

Loop Optimizations for Dependence-Driven Models

This chapter extends and optimizes the loop compilation techniques for dependence-driven systems that was presented in Chapter 3. Specifically, this chapter presents a new algorithm that combines multi-dimensional tiling and loop coalescing for multiply-nested parallel loops, and determines heuristically through a dependence-driven specific cost model (built on micro-benchmarks), a processor configuration to best match the set of tiled loops.

4.1 Introduction

A major goal of Chapter 3 was to partition imperative parallel programs (with and without complex control flow graphs) in order to map the computation onto dependence-driven execution models. The challenge is that in compiling data-parallel loops, implicit barriers at the end of the loops can have a serious impact on the performance of the program. In the case when nested or inner loop parallelism is leveraged, overhead from the inner barriers become an even bigger detriment. In a majority of loops that leverage parallelism, a single level of

parallelism (commonly the outer-loop) is typically sufficient if the number of loop iterations are large enough and the amount of work per iteration justifies the need for parallelism. However, there are situations when nested parallelism can also provide benefit over using a single level [87, 88].

```
1 forall i in 1..U_i {
2   forall j in 1..U_j {
3     forall k in 1..U_k {
4       for w in 1..U_w {
5         W(i,j,k,w) = ...;
6       }
7     }
8   }
9 }
```

Figure 4.1: Nested Parallel Loops With Static Loop Bounds

Consider the triply nested parallel loop in Figure 4.1. For the three parallel loops in this figure, assume static loop scheduling and execution on a multi-core architecture with a maximum number of processors P_{max}, with the constraint that $P_{max} = U_i U_j U_k$, and $U_i > 1$, $U_j > 1$, $U_k > 1$. The loop body $\mathcal{W}$ represents the workload and is a function of the loop indices (i, j, k, w). Also, assume that U_w is a large value, so that there is a substantial (and uniform) workload per parallel iteration. If the programmer were to only parallelize a single loop in this program, not all of P_{max} would effectively be utilized. The only way to utilize all of P_{max} would be for all of the loops to be parallelized, as shown in the example. However, since parallelization occurs with `forall` loops, the following limitations to execution performance will occur:

- A decrease in parallelism as a result of the unnecessary implicit join synchronization from `forall` loops, even though the loop iterations are fully independent from one another. The decrease in parallelism could result in multiple scenarios. In the Cholesky factorization example shown previously in Figure 3.27, if any of the inner loops were to be parallelized, the outer-most k loop would be prevented from overlapping with the

rest of the computation. Due to this, the language extensions introduced in Chapter 3.6 were one way of increasing asynchronous parallelism from an imperative language.

- An increase in overhead from the dynamic creation and destruction of inner-parallel tasks as opposed to creating and killing the tasks once. Depending on the runtime implementation, this overhead could be either large or small. If the runtime needs to spawn heavyweight threads on each encounter, this could result in more overhead. The more likely scenario is that the tasks already exist in a thread-pool. However, even in this scenario, cycles will need to be spent on activating and releasing those tasks.

- Increased communication costs between tasks in order to provide data coherency between them. Since each parallel loop will result in a separate task, any data that is shared between the loops will need to be propagated by the DSM.

Loop coalescing is one method that can be used in order to reduce the overhead associated with nested-parallel loops. Assuming normalized loops with invariant loop-bounds and no loop-carried data dependences, loop coalescing is a transformation that translates a multiply-nested loop into a single loop and updates the loop bounds of the single-dimensional loop to the product of the dimensions of the inner loops. Start with an original loop nest of the form $L = (N_m, N_{m-1}, \ldots, N_1)$ where $(N_m, N_{m-1}, \ldots, N_1)$ represents the trip counts of each loop in the loop nest. Also let L have the loop indices $\mathcal{I} = (i_m, i_{m-1}, \ldots, i_1)$ and array indices of the form $A(i_m, i_{m-1}, \ldots, i_1)$. Applying loop coalescing will form the single loop nest $L' = (N_m N_{m-1} \ldots N_1)$, which now has a single loop index $\mathcal{J}$ and array indices of the form $A(f_m(\mathcal{J}), f_{m-1}(\mathcal{J}), \ldots, f_1(\mathcal{J}))$. The following formula [42] is used to compute the new array indices:

$$f_k(\mathcal{J}) = \left\lceil \frac{\mathcal{J}}{\prod_{i=1}^{k-1} N_i} \right\rceil - N_k \left\lfloor \frac{\mathcal{J}-1}{\prod_{i=1}^{k} N_i} \right\rfloor, \qquad k = m, m-1, \ldots, 1 \tag{4.1}$$

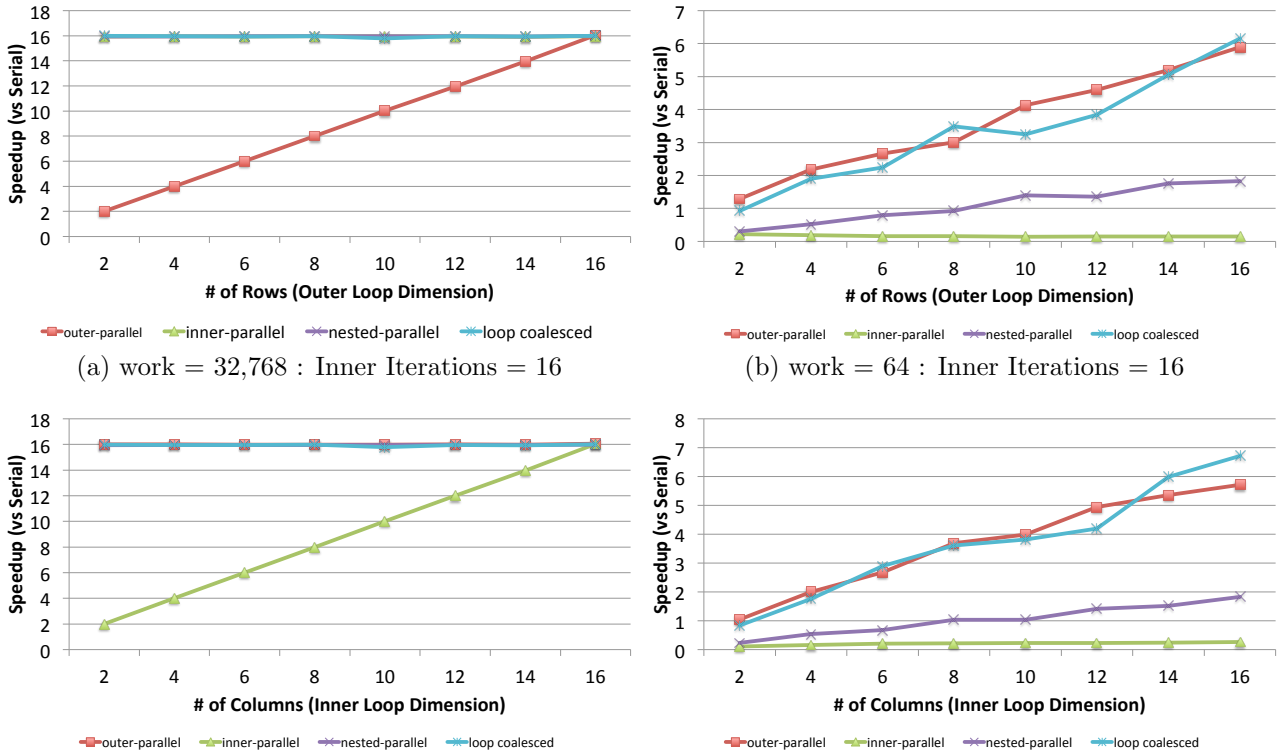
When loop coalescing is applied to the previous example, the loop is transformed into the one shown in Figure 4.2. As this example clearly shows, all tasks are now created initially, and the amount of inner loop barrier synchronization is removed at the expense of an additional computation from computing the new indices. Additionally, scheduling has now been simplified.

```
1 forall ijk in 1..U_i U_j U_k {
2   var i = ⌈ijk / (U_j U_k)⌉
3   var j = ⌈ijk / U_k⌉ − U_j ⌊(ijk−1) / (U_j U_k)⌋
4   var k = ijk − U_k ⌊(ijk−1) / U_k⌋
5   for w in 1..U_w {
6     W(i, j, k, w) = ...;
7   }
8 }
```

Figure 4.2: Loop Coalescing Example

In cases where the multiply-nested parallel loop is non-perfectly nested, loop coalescing can still be performed either by inserting conditional code that ensures the correct section of code is to be executed [89], or by performing other loop transformations such as loop distribution [43]. In this thesis, the assumption will be that all multiply-nested parallel loops are to be perfectly nested, and if they are not, loop distribution is performed by the compiler earlier in the compilation process. If loop distribution cannot be performed, a last measure taken by the compiler is to still perform loop coalescing, but now the compiler must insert a conditional statement into the non-perfectly nested loop so that the correct portion of code is executed in the different loop nest levels [89].

The different figures in Figure 4.3 present an evaluation where a single level of loop parallelism (both inner and outer) is insufficient compared to other loop parallelization techniques such as loop coalescing and nested parallelism, thus, demonstrating a need for multi-dimensional parallelism. The point of this experiment is to show the scenario where some form of nested parallelism is advantageous compared to utilizing only single level loop par-



(a) work = 32,768 : Inner Iterations = 16

(b) work = 64 : Inner Iterations = 16

(c) work = 32,768 : Outer Iterations = 16

(d) work = 64 : Outer Iterations = 16

Figure 4.3: Single Level Parallelism Compared with Multi-Dimensional Parallelism

allelism when the amount of work per iteration is both small and large.

The benchmark used in this experiment is similar in structure to that of the code snippet in Figure 4.1. Also, the machine platform used in this evaluation is the same that was used earlier from Table 3.1 in Chapter 3. For this experiment, up to 16 hardware threads will be used. In this program, two sets of 2–D grid sizes are evaluated: $\{16 \times [1 : 16]\}$ and $\{[1 : 16] \times 16\}$. This implies that the number of outer loop iterations are fixed to 16 iterations, and the inner loop iterations vary from 1 to 16, or vice versa, with the outer loop iterations varying from 1 to 16 and the inner loop iterations fixed to 16. Additionally, these 2–D grid sizes represent the maximum number of processors per loop dimension. This was done in order to show that by parallelizing only a single dimension of the loop nest, not all of the resources will be leveraged. However, by using a technique such as loop coalescing or

nested parallelism, those resources will be utilized. This evaluation is performed using large and small problem sizes in order to understand what task-specific overheads are introduced. The x-axis in this evaluation represents a varying number of iterations for the outer (or inner) loop dimension. The y-axis represents performance scalability over a serial implementation of the program.

Starting with Figure 4.3a, for outer-level parallelism to perform as well as the other loop techniques, the number of rows (the outer dimension) needs to increase. Once the number of rows matches the number of hardware threads (16 in this case), outer-level parallelism performs similarly to the other approaches. When the problem size is small, as shown in Figure 4.3b, the different loop parallelization approaches perform differently. In this case, the overheads associated with nested-parallelism have a negative impact on the overall performance. The major slowdown associated with inner-level parallelism is due to the overhead of dynamically spawning and joining inner parallel loops. The remainder of the parallel loops all perform comparably.

Instead of increasing the number of rows, Figures 4.3c and 4.3d show what happens when the number of columns (the inner dimension) increases. For the majority of the loops, the performance is the same as before, except now, the performance of the inner-parallel loops has been reversed with that of the outer-parallel loops from Figures 4.3a and 4.3b.

Applying the loop compilation techniques from Chapter 3 introduces additional overhead from different sources, such as time spent by the runtime in performing data communication between tasks or from time spent on task creation and destruction for inner parallel loops and their barriers. Additionally, the loop itself might not have a sufficient amount of work to justify full parallelization when the overheads are taken into consideration. Instead, there can be a situation where some amount of parallelization can still increase program performance without leveraging all of the available processors in the system. Part of the work in this chapter will be to develop a heuristic that finds a suitable amount of parallelization for the

given multiply-nested parallel loop, runtime, and associated overheads.

Contributions The contributions of this chapter include:

- The introduction of a new loop transformation that combines loop-coalescing and blocking in order to map multiply-nested parallel loops onto dependence-driven runtime systems without having to introduce an implicit join synchronization for the inner parallel loop.
- The automatic selection of tile size and shapes based on a heuristic that evaluates overheads related to the hardware and the runtime system. In many cases, the selected tile configuration will end up using fewer resources than what the machine can offer.
- Experimental results that compare the approach of using a tiled-coalesced loop transformation to other forms of parallel loops including loops that have had loop coalescing applied to then, and multiply-nested parallel loops.

This chapter is organized as follows: Section 4.2 describes a new transformation that combines tiling with loop coalescing. Section 4.3 will present a heuristic that can determine tile sizes and shapes to use when performing tiled-coalescing. Afterwards, Section 4.4 will present experimental results for these optimizations. Finally, Sections 4.5-4.7 will discuss current limitations, related work, and then final discussions.

4.2 Multi-Dimensional Blocked-Coalesced Form

This section describes a technique that will combine loop coalescing with tiling for multiply-nested parallel loops in order to decrease the overhead of barrier synchronization, potentially increase locality, and provide evenly load-balanced tasks to the runtime scheduler.

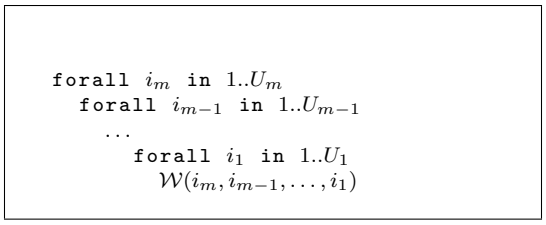


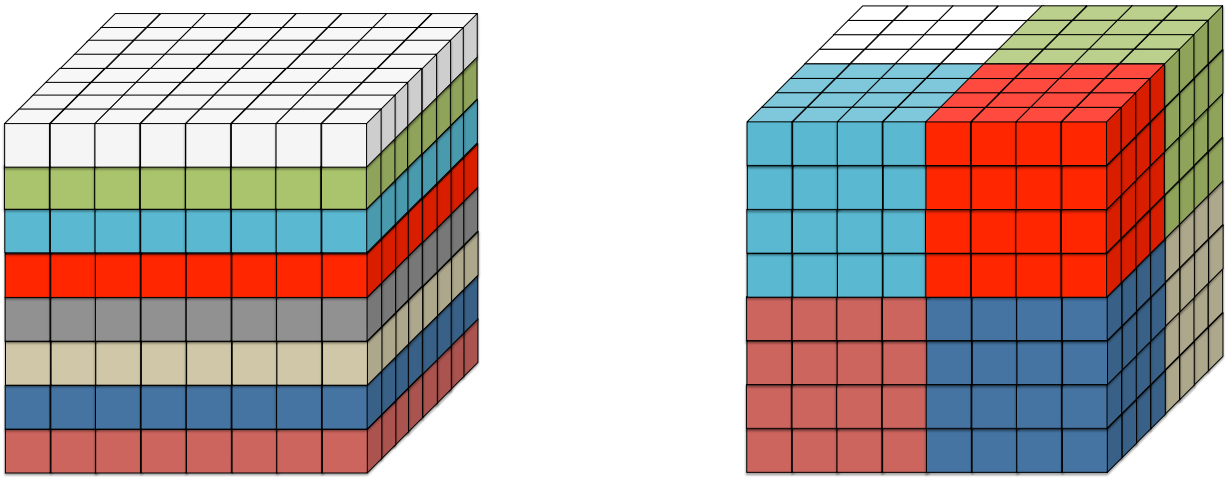
Figure 4.4: Multiply-Nested Parallel Loop

Loop Coalesced Form

The 1–D loop coalesced form for loop nest L is $L'_{coalesced} = (U_m U_{m-1} \ldots U_1)$, with the single loop index $\mathcal{J}$, and array indices of the form $A(f_m(\mathcal{J}), f_{m-1}(\mathcal{J}), \ldots, f_1(\mathcal{J}))$. The 1-D iteration space of $L'_{coalesced}$ is J and it is defined as follows:

$$J = \left\{ j | 1 \leq j \leq \prod_{i=1}^{m} U_i; j \in \mathbb{Z} \right\} \tag{4.2}$$

There are trade-offs to be made when comparing traditional nested parallelism with loop coalescing. First, in the case of loop coalescing, by combining the total loop nest L into a single level L', the overhead of dynamically spawning and joining the inner tasks has been removed. However, there are two potential downsides to using loop coalescing. The first is that it introduces additional overhead in the computation of loop indices. Fortunately, in most situations this source of overhead is negligible because the computing of indices does not occur in the innermost loop. The second downside is that loop coalescing does not take multi-dimensional locality into consideration; by itself, loop coalescing does not perform n-D blocking (where $n > 1$). For example, in many numerical algorithms, tiling for locality and parallelism is a classic and well-known optimization. In this case, it is advantageous to decompose the matrices into square or rectangular shapes rather than along just a single dimension. On the other hand, when using nested parallel loops (e.g. `forall`), they are commonly strip-mined [43] among a number of processors. The strip-mining of each parallel loop forms a natural blocked access pattern for each processor. This scenario will be made



(a) Loop Coalesced Iteration Space

(b) Tiled Iteration Space

Figure 4.5: Loop Coalescing vs Tiled Comparison of Iteration Spaces Partitioned Among 8 Threads

evident in the following example.

Continuing with the earlier example from Figure 4.2, assume that $P_{max} = 8$, and that the original nested parallel loop L has loop limits $(8, 8, 8)$, where each parallel loop has been blocked by a factor of 2. This blocking factor implies that each loop in loop-nest L, will have 2 threads dedicated to it. Also in this example, assume that the loop has been statically scheduled across P_{max} with at most $\left\lceil \frac{U_i U_j U_k}{P_{max}} \right\rceil = \left\lceil \frac{512}{8} \right\rceil = 64$ iterations per processor. Figure 4.5a shows the effect on the access pattern that loop coalescing has when compared to a 3-D blocking of the iteration space as shown in Figure 4.5b. Assume a row-major order for accessing data. If the length of the row was very large for Figure 4.5a, an increase in the number of cache misses will likely result (ignoring any possible hardware prefetching). On the other hand, in Figure 4.5b, each thread's access pattern represents a cube, where no dimension is as large compared to the non-tiled scenario. This will likely increase the possibility of reuse, thus leading to higher performance.

Due to the trade-offs between loop coalescing and nested parallelism, the major contribution of this chapter is to develop a transformation that leverages the main benefit of loop

coalescing for its removal of dynamically spawning and joining of inner-level parallel threads, and leverage the main benefit of nested parallelism's support for blocking the iteration space to potentially increase locality.

Tiled-Coalesced Form

In the tiled-coalesced form, the original loop-nest L has been transformed into the loop-nest $L'_{tile-coalesced}$, where the loop $n \in L$ has been blocked by a factor of B_n. The blocking of each loop $n \in L$ produces the *chunk* $C_n = \left\lceil \frac{U_n}{B_n} \right\rceil$. Now that L has been transformed into $L'_{tile-coalesced}$, the associated array indices need to be updated from $A(i_m, i_{m-1}, \ldots, i_1)$ to $A(f_m(\mathcal{J}), f_{m-1}(\mathcal{J}), \ldots, f_1(\mathcal{J}))$. Assuming that $\mathcal{J}$ starts at 1, Equation 4.5 which is made up of Equations 4.3 and 4.4, is used to convert the array indices into a tiled-coalesced form. The portion of the equation that iterates across the tiles is provided in Equation 4.3, and the portion that iterates through the elements inside of the tile is given in Equation 4.4.

$$f_{k\_block\_idx}(\mathcal{J}) = \left(\left\lfloor \frac{\mathcal{J}-1}{\left(\prod_{i=1}^{k-1} U_i\right)\left(\prod_{i=k}^{m} C_i\right)} \right\rfloor \bmod \frac{U_k}{C_k} \right) C_k, \qquad k = m, m-1, \ldots, 1 \tag{4.3}$$

$$f_{k\_insideblock\_idx}(\mathcal{J}) = \left\lfloor \frac{\mathcal{J}-1}{\prod_{i=1}^{k-1} C_i} \right\rfloor \bmod C_k, \qquad k = m, m-1, \ldots, 1 \tag{4.4}$$

$$f_k(\mathcal{J}) = f_{k\_block\_idx}(\mathcal{J}) + f_{k\_insideblock\_idx}(\mathcal{J}) + 1, \qquad k = m, m-1, \ldots, 1 \tag{4.5}$$

By applying Equation 4.5 to a loop that has already been transformed into a coalesced form, Figure 4.5b would be an example of what the tiled iteration space would look like. Note that the work in this section does not automatically generate tiled loops. It only maps the indices from a multi-dimensional loop nest to the tiled-coalesced form. It would be up to the compiler to actually generate the loops.

4.3 Heuristics for Tile Sizes

This section describes the steps necessary to construct a heuristic based on a machine and software cost model that determines multi-dimensional block shape dimensions.

4.3.1 Off-Line Timing Benchmarks

In order to construct a heuristic based on a cost model of the machine and software characteristics, a set of micro-benchmarks are to be developed. When used together, they will help in providing an estimated execution time for the parallel loop. In this case, the parallel loop used here is the one that has been formed by applying the tiled-coalesced transformation from the previous section. The particular characteristics that are measured are given in Table 4.1 followed by a description of each micro-benchmark for the remainder of this section.

Measurement	**Description**
T_{serial}	Execution time of loop body for one iteration
T_{create}	Execution time for creation of a single task
$T_{barrier}$	Execution time for overhead associated with barriers
T_{data}	Time spent performing data communication

Table 4.1: Timing Benchmarks

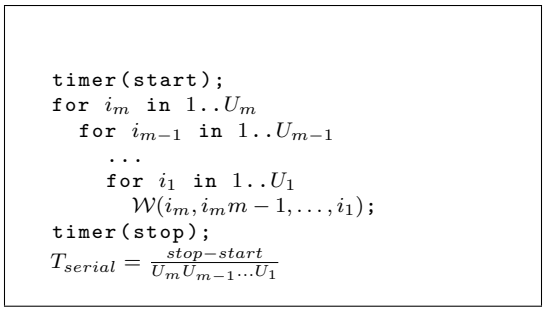


Figure 4.6: Loop Body Timing Micro-benchmark

T_{serial} : Loop body execution time for one iteration

This micro-benchmark measures the execution time for only a single instance of the loop body in the sequential nested inner loops. T_{serial} is computed using the formula:

$$T_{serial} = \frac{\texttt{timer}(stop) - \texttt{timer}(start)}{\prod_{i=1}^{m} U_i} \tag{4.6}$$

As the micro-benchmark shows in Figure 4.6, the timer routine `timer`(*start*) is placed directly before the start of the nested loop, and `timer`(*stop*) is placed directly at the end of the loop. In the case that loop workload is not uniform per parallel iteration, T_{serial} ends up giving the average time for the body of the loop. For sparse or irregular computations such as sparse matrix-vector multiplication, this could lead to execution times with a wide variance, and for dense and regular computations this measurement would be a reasonable approximation. One improvement to this would be to use a runtime sampling measurement in order to dynamically adjust in situations dealing with irregular computations. However, this approach is not taken here, and is left for future work. Another alternative approach would be construct a purely analytical model to estimate the execution time of the program [90, 91] without having to rely on micro-benchmarks as was done here.

```
timer(start);
forall i in 1..1 {
  timer(stop);
}
T_create = stop − start;
```

Figure 4.7: Task Creation Micro-benchmark

T_{create} : Execution time for creation of a single task

The benchmark measures the amount of time needed to spawn a *single* task. T_{create} is trivially computed using the following:

$$T_{create} = \texttt{timer}(stop) - \texttt{timer}(start) \tag{4.7}$$

As Figure 4.7 shows, the routine `timer`($start$) is placed directly outside of a one-dimensional parallel loop that has only a single iteration and no work inside of it. The routine `timer`($stop$) is the only instruction in the body of the parallel loop. Since there is only a single iteration of the parallel loop, there is no concern for any race conditions when writing to T_{create}.

$T_{barrier}$: Barrier overheads

```
forall i in 1..1 {
  timer(start);
}
timer(stop);
T_create = stop − start;
```

Figure 4.8: Barrier Overhead Micro-benchmark

The benchmark in Figure 4.8 measures the total overhead associated with the implicit (compiler-generated) barrier task at the end of each parallel loop for a single task. This includes the time to spawn the barrier task, and the time to wait at the barrier for a single

task. Later, this metric will be used to extrapolate the total implicit barrier synchronization overhead for P_{max} tasks. Just like computing T_{create}, the trivial formula used to compute $T_{barrier}$ is the following:

$$T_{barrier} = \texttt{timer}(stop) - \texttt{timer}(start) \tag{4.8}$$

Recall from Section 3.3.7, that the compiler generates a separate barrier task that is used to notify its successor when all iterations of the parallel loop have been completed. In this case, `timer`(start) occurs as the only statement in the body of the parallel loop. The routine `timer`(*stop*) is placed directly outside of the parallel loop.

T_{data} : Data communication

```
var x1, x2, ..., xn;
timer(start);
forall idx in 1..1 {
  x1 = ...;
  x2 = ...;
  ...
  xn = ...;
  W(x1, x2, ..., xn, idx);
}
timer(stop);
T_data = stop - start - T_serial - T_barrier - T_create;
```

Figure 4.9: Data Copying Overhead Micro-Benchmark

The last micro-benchmark is used to measure T_{data}, the total time spent by the dependence-driven runtime in copying data between the tasks in order to guarantee coherency. In order to compute T_{data}, it is dependent on the earlier micro-benchmark measurements. The formula for T_{data} is as follows:

$$T_{data} = \texttt{timer}(stop) - \texttt{timer}(start) - T_{serial} - T_{barrier} - T_{create} \tag{4.9}$$

As Figure 4.9 shows, T_{data} is computed by measuring the execution time for the parallel loop with only a single task, and then removing the time spent from the previous micro-benchmark results. By subtracting away the previous micro-benchmark results, the only thing that remains is the time spent by the runtime in copying data into the tasks. While this technique provides a good estimation of the data communication overhead, a more precise alternative would be to perform a dataflow analysis and determine what data is coherent or not, and then extrapolate the communication time for each piece of data that is to be kept coherent. Even though this alternative technique would lead to a more precise estimate of data communication costs, this approach was not taken in order to simplify the cost model. Additionally, as the results will show, the time spent performing data communication is a small fraction of the overall execution time. Finally, one can also make T_{data} a function of P due to a likely increase in coherence traffic. However, this approach was not taken for similar reasons just mentioned.

4.3.2 Cost Model

Using the micro-benchmark measurements described in Section 4.3.1, the overall goal here is to determine the ideal number of processors that will be used to parallelize the set of tile-coalesced parallel loops. If the number of iterations are large enough, the amount of processors to use should be P_{max}. However, in reality, the ideal number of processors is not always P_{max} since the amount work per task (i.e. T_{serial}) and the other overheads might not justify using all available processors. In this case, if P_{max} number of processors were to be used, a slow down in performance could result.

Consider the function in Equation 4.10 that estimates the total execution time of the coalesced loop.

$$f(P) = T_{parallel} = T_{serial}\frac{\prod_{i=1}^{m} U_i}{P} + T_{create}P + T_{barrier} + T_{data} \tag{4.10}$$

This function combines the following: the serial amount of time it takes to compute a block (for all of the blocks), the time it takes to create and spawn P tasks, and the overheads of creating a barrier and performing the data copy. This equation is simplified in two areas: first, the cost model assumes that the total time spent waiting inside of a barrier by each task is fully overlapped with the others and does not account for imbalance. As a result of this, the $T_{barrier}$ is constant and not a function of the number of processors P. This also applies to T_{data}, where the total time spent copying data between tasks is overlapped making T_{data} a constant value. Second, as an approximation, $T_{create}P$ is a function of P since it assumes that there will be no other coalesced loops inside of the already transformed coalesced loop.

One method of validating this cost model is to understand how well it performs when used with the heuristic that is presented in Section 4.3.4. If the heuristic and its associated cost model provide a similar output to that of an exhaustive search, this would be one measure of validation.

The goal here is to find a $P_{optimal}$ that minimizes execution time such that:

$$1 \leq P_{optimal} \leq P_{max} \tag{4.11}$$

The global minimum $P_{optimal}$ (i.e. P) is computed as follows:

$$\frac{d}{dp}f(P) = T_{create} - T_{serial}\frac{\prod_{i=1}^{m} U_i}{P^2} \tag{4.12}$$

$$T_{create} - T_{serial}\frac{\prod_{i=1}^{m} U_i}{P^2} = 0$$

$$T_{create} = T_{serial}\frac{\prod_{i=1}^{m} U_i}{P^2}$$

$$P_{optimal} = P = \sqrt{T_{serial}\frac{\prod_{i=1}^{m} U_i}{T_{create}}} \tag{4.13}$$

Equation 4.13 returns the number of processors that minimizes execution time. The only

aspect of this equation that is unique to the program (and its input dependences) is the serial execution time, and knowledge of the loop bounds from the given loop nest. The T_{create} value is constant for the machine, and can be computed once off-line. The value T_{serial} is unique to the program, and could either be computed online through sampling, or off-line as was done in the implementation. The downside to the off-line approach for computing T_{serial} is that it needs to be computed once per program, or every time if the program's inputs affect its execution (e.g. unique sparse matrices from SpMV).

While Equation 4.13 returns the whole number of processors $P_{optimal}$, part of the job of the heuristic in Section 4.3.4 will be to determine the integer factors of $P_{optimal}$. Each integer factor is what will be used as a blocking factor to the original parallel loop nest.

4.3.3 Memory Footprint

In the situation where multiple candidate blocking factors of $P_{optimal}$ are found that result in the same estimated execution time, one potential optimization would be to select the blocking factors that lead to an overall minimal memory footprint [92] of the array accesses in the parallel loop nest. The memory footprint is the total amount of cache or memory that is used by an array reference. This will be the tactic used by the heuristic in Section 4.3.4. In order to simplify the approximation, it is assumed that cache is fully associative.

If the data being requested is not already in cache, the memory footprint of a variable or array reference d, as shown in Equation 4.14 is simply the cache line size CLS that is brought in on a cache miss:

$$C(d) = CLS \tag{4.14}$$

In order to determine the cache line size CLS, the size can either be looked up in vendor specific documentation, or determined programmatically through micro-benchmarks. Chap-

ter 6 presents the P-Ray multi-core benchmark suite [93], whose main goal is to determine machine characteristics including the cache line size CLS for multi-core architectures.

Starting with the simple case of no memory reuse, Equation 4.15 provides the memory footprint for an array reference d inside of a loop at depth k. This is simply the product of the cache line size and the number of loop iterations (the trip count for N_k).

$$C(d)_k = CLS \cdot N_k, \qquad k = m, m-1, \ldots, 1 \tag{4.15}$$

Equation 4.16 provides the total memory footprint for the array reference d in the entire loop nest. This now becomes the product of the trip counts for all loops in the loop nest and the memory footprint for d at loop depth k (computed in Equation 4.15).

$$C(d)_{total} = \prod_{i=k}^{m} (CLS \cdot N_k), \qquad k = m, m-1, \ldots, 1 \tag{4.16}$$

Equation 4.16 implies that every array reference d that occurs per iteration of the loop nest will result in a cache miss, thus, bringing in a new cache line per memory access. This cost is then propagated across the entire loop nest in order to form $C(d)_{total}$.

Memory reuse (locality) can help decrease the overall memory footprint. A reuse factor (denoted as R_k) represents the number of times the same array element (or cache line) is referenced inside of a loop at depth k [94, 92, 95]. Reuse occurs when an array's footprint size is smaller than the iteration space of a particular loop at depth k. With this measurement available, the heuristic can then choose block dimensions that not only minimize execution time based on the number of processors to use, but also by the block dimensions which minimize the memory footprint. Presently, the Chapel compiler implementation does not provide any means of computing R_k automatically, so for this thesis, each unique variable's R_k is computed by hand.

An approximation for the memory footprint of an array reference d at a given loop depth

k is the following equation:

$$C(d)_{reuse} = \prod_{i=k}^{m} \left(CLS \cdot \frac{N_k}{R_k} \right), \qquad k = m, m-1, \ldots, 1 \tag{4.17}$$

The previous Equation 4.17 computed the memory footprint for only a single array reference. However, it is important to compute the footprint for all of the array references in the loop nest. Equation 4.18 computes the total memory footprint for the set of all array references D:

$$Total\_Cost = \sum_{j=1}^{D} C(j)_{reuse}, \qquad k = m, m-1, \ldots, 1 \tag{4.18}$$

While the approach taken in this thesis is to use a simplified estimation in computing the total memory footprint $Total\_Cost$, other techniques exist that could be used in place of Equation 4.18 in order to provide a more accurate estimation of what locality exists in the loop nest [96, 97].

4.3.4 Heuristic

The heuristic, shown as Algorithm 6 in Figure 4.10 combines the previously discussed timing measurements computed by different micro-benchmarks used to measure the dependence-driven task characteristics, and the memory footprint for each array used in the nested loop that was discussed in Section 4.3.3. The overall goal of this heuristic is to determine an optimal number of processors to use, $P_{optimal}$, which will then be decomposed into a set of factors. These are the factors that will be used to perform the actual blocking on the original parallel loop nest. The heuristic will then search through all the possible integer factors of $P_{optimal}$ in order to find the blocking factors that lead to a minimal execution time. In cases where multiple factors are found that lead to the same execution time, a refinement process

occurs needs to occur. Of the found blocking factors that lead to the same execution time, the blocking factor that results in a minimal memory footprint will be the chosen set of factors to be used.

The first step of the heuristic in line 1 is to create and initialize the candidate lists that will eventually contain different tile sizes and the number of tiles per dimension. For the second step in lines 2–3, the algorithm initializes the tuples $\{BestNTiles_m, \ldots, BestNTiles_1\}$ and $\{BestTileDim_m, \ldots, BestTileDim_1\}$ with the number of tiles and tile size per loop. In this case, the initial number of tiles per loop is set to 1, and the tile size is set to the entire loop width. The third step of the heuristic in line 4, is to compute the estimated serial execution time for the loop nest, and use that as an interim best case execution time. In line 5, the heuristic determines an approximate number of processors, $P_{optimal}$, using Equation 4.13. In line 6, the computed value $P_{optimal}$ will then be decomposed into the best blocking factors for each loop in the given loop-nest, where each of these blocking factors will then be used by the tiled-coalescing loop transformation. Any integer decomposition algorithm can be used to decompose $P_{optimal}$ into a set of factors $\{q_m q_{m-1} \ldots q_1\}$. For the implementation used as part of this thesis, trial division [98] was the integer factorization algorithm used due to its simplicity of implementation. An alternative to this would be to do a brute force search through all possible processor configurations ($1 \leq P \leq P_{max}$), but this comes at the expense of performing a brute force search. In order to take into account for small variations in running time from the previous micro-benchmarks and other errors, not only are the factors of $P_{optimal}$ considered, but so are the additional factors for $P_{optimal} \pm \sigma$. This is done in order to increase the precision in finding the correct blocking factors to use. As the algorithm proceeds in lines 7–16, the heuristic tries to search for the block size combination that results in the lowest estimated parallel execution time (given by $T_{parallel}$). For each pass of this loop, every time a lower estimated parallel execution time has been found, the running time and associated block numbers and dimensions are added to a list of candidates.

Afterwards in line 17, the top 10% of lowest execution times are saved, with the remainder being discarded. A 10% cutoff point was chosen as an estimation, due to any possible errors or inaccuracies of the previous micro-benchmarks. Next, an upper limit on the memory footprint size (for an unblocked loop nest) is computed in line 18. Afterwards, using the top 10% of the blocks and their dimensions that were saved in line 17, the total memory footprint for the blocked loop nest is computed in line 20. Finally, just as before when finding the minimum parallel execution time, lines 20–24 determine the block sizes that lead to a minimal memory footprint. With these newly found dimensions and the number of tiles, the tiled-coalesced loop transformation will be applied.

4.4 Evaluation

There are two main goals for this evaluation:

1. Measure and evaluate the effectiveness of the tiled-coalesced loop transformation on codes that are to be executed on a dependence-driven runtime system.

2. Determine how well the heuristic from Section 4.3 can match an exhaustive scheme in finding the correct number of processor resources to utilize for the given parallel loops.

The outline for the remainder of this chapter is: Section 4.4.1 will describe the environment used to perform the experiments. Section 4.4.2 will describe each of the benchmarks that will be measured. Section 4.4.3 will discuss the experimental methodology that is applied. Finally, Section 4.4.4 will present the evaluation of these experiments.

4.4.1 Environmental Setup

Table 4.2 describes the environment that was used to perform the experiments. The experiments in this section will use between 1 and 16 processor cores. Even though the given

Algorithm 6: Heuristic to Determine Blocking Dimensions

Input: $\{U_m, \ldots, U_1\}$: Loop limits for each loop level m:1
Input: P_{max} : Maximum number of processors
Input: T_{serial} : Execution time for loop body
Input: T_{create} : Time to create and spawn a single task
Input: T_{data} : Time to copy/communicate all data for the given loop
Input: $T_{barrier}$: Overhead associated with task barrier
Output: $\{NTiles_m, NTiles_{m-1}, \ldots, NTiles_1\}$: Optimal number of tiles for loop levels m:1
Output: $\{TileDim_m, TileDim_{m-1}, \ldots, TileDim_1\}$: Blocking factors for loop levels m:1

$\{TopNTiles, TopTileDim\} \leftarrow \{\emptyset, \emptyset\}$;
$\{BestNTiles_m, \ldots, BestNTiles_1\} \leftarrow \{1, \ldots, 1\}$;
$\{BestTileDim_m, \ldots, BestTileDim_1\} \leftarrow \{U_m, \ldots, U_1\}$;
$BestTime \leftarrow T_{serial} \prod_{i=1}^{m} U_i$;
$P_{optimal} \leftarrow \sqrt{\frac{T_{serial} \prod_{i=1}^{m} U_i}{T_{create}}}$;
foreach $(q_m q_{m-1} \ldots q_1 == P_{optimal} \pm \sigma) \leq P_{max}$ **do**
    $\{TileDim_m, \ldots, TileDim_1\} \leftarrow \left\{ \left\lceil \frac{m}{q_m} \right\rceil, \ldots, \left\lceil \frac{m}{q_1} \right\rceil \right\}$;
    $\{NTiles_m, \ldots, NTiles_1\} \leftarrow \left\{ \left\lceil \frac{m}{TileDim_m} \right\rceil, \ldots, \left\lceil \frac{m}{TileDim_1} \right\rceil \right\}$;
    $T_{parallel} \leftarrow T_{serial} \prod_{i=1}^{m} TileDim_i + T_{create} \prod_{i=1}^{m} NTiles_i + T_{barrier} + T_{data}$;
    if $BestTime > T_{parallel}$ **then**
        $BestTime \leftarrow T_{parallel}$;
        $\{BestNTiles_m, \ldots, BestNTiles_1\} \leftarrow \{NTiles_m, \ldots, NTiles_1\}$;
        $\{BestTileDim_m, \ldots, BestTileDim_1\} \leftarrow \{TileDim_m, \ldots, TileDim_1\}$;
        $TopBestTimes \leftarrow TopBestTimes \cup BestTime$;
        $TopNTiles \leftarrow TopNTiles \cup \{BestNTiles_m, \ldots, BestNTiles_1\}$;
        $TopTileDim \leftarrow TopTileDim \cup \{BestTileDim_m, \ldots, BestTileDim_1\}$;
Keep 10% of $\{TopBestTimes, TopTileDim, TopNTiles\}$ with lowest times and discard the rest;
$FootPrint_{minimum} \leftarrow compute\_footprint(\{U_m, \ldots, U_1\}, \{1, \ldots, 1\})$;
foreach $\{\{TileDim_m, \ldots, TileDim_1\}, \{NTiles_m, \ldots, NTiles_1\}\} \in \{TopTileDim, TopNTiles\}$ **do**
    $FootPrint \leftarrow compute\_footprint(\{TileDim_m, \ldots, TileDim_1\}, \{NTiles_m, \ldots, NTiles_1\})$;
    if $FootPrint_{minimum} > FootPrint$ **then**
        $FootPrint_{minimum} \leftarrow FootPrint$;
        $\{BestNTiles_m, \ldots, BestNTiles_1\} \leftarrow \{NTiles_m, \ldots, NTiles_1\}$;
        $\{BestTileDim_m, \ldots, BestTileDim_1\} \leftarrow \{TileDim_m, \ldots, TileDim_1\}$;
return $\{\{BestNTiles_m, \ldots, BestNTiles_1\}, \{BestTileDim_m, \ldots, BestTileDim_1\}\}$;

Figure 4.10: Heuristic to Determine Blocking Dimensions

CPU Architecture	X86-64 Intel Xeon L7555
Num cores (threads per core)	32 (2)
Total threads	64
Clock Rate (GHz)	1.866
Total Memory (GB)	128
L3 Cache size (MB)	24
OS (Kernel)	Scientific Linux 6.3 (2.6.32)
Compiler	GCC 4.4.6
Chapel Compiler	1.4.0
StarPU Runtime	1.0.3

Table 4.2: Architecture Tested

machine has more processor resources, part of the goal is to find an ideal processor configuration using both the heuristic from Figure 4.10 and through an exhaustive search. By limiting the processor size to 16, the time to perform the exhaustive search is minimized, yet still provides the computation with enough parallelism.

4.4.2 Experiments

Benchmark	Nested Parallel Loop(s) Depth
3–D Jacobi Method	2, 2, 2
Sparse Matrix-Vector Multiplication (SpMV)	2
Coulombic Potential	2
Synthetic.Trig	2
2–D Image Histogram	2,2

Table 4.3: Evaluated Benchmarks and Their Parallel Loop Depths

The following applications (provided in Table 4.3) have been used as part of the evaluation process:

1. **3–D Jacobi Method**

   This is the same 27–point stencil computation as used in Chapter 3.4. The experiments here will be performed on 3–D grid sizes (a triply-nested loop), where the outer loop

represents a varying number of rows that from 1 to 16 parallel iterations, the middle loop represents a varying number of columns from 1 to 16 parallel iterations, and the innermost loop representing the depth and fixed to one of $\{128, 256, 512\}$ serial iterations. This was done in order to keep the two outer dimensions small, and the innermost dimension of variable size so that the heuristic could potentially choose processor P configurations $\{1 \leq P \leq P_{max}\}$ instead of always choosing P_{max} as the default configuration. Using inner dimension sizes of $\{128, 256, 512\}$, we expect to see a transition where initially it is better to use fewer processors, and as the problem size increases, there is a transition to requiring more resources.

2. **Sparse Matrix-Vector Multiplication (SpMV)**

   The SpMV kernel used here is the same one as used in Chapter 3.4. The main difference is that multiple vector sizes ($1 \rightarrow 256$ logarithmically) will be used instead of just a single vector size as before.

3. **Coulombic Potential (CP)**

   The CP benchmark used here is based on the same benchmark from Chapter 3.4. The difference is that the input sizes used now are smaller because the goal of this chapter is to find a transition point between using one and P_{max} processors. In this case, the following five different input sizes were evaluated: $\{4 \times 4 \times 1\}$, $\{8 \times 8 \times 1\}$, $\{16 \times 16 \times 1\}$, $\{32 \times 32 \times 1\}$, $\{64 \times 64 \times 1\}$.

4. **Synthetic.Trig**

   This is the same benchmark from Chapter 3.4 with the main difference being the number of inner iterations being tested now will be smaller than before. The number of inner loop iterations to be tested are $\{4, 16, 64, 256, 1024\}$.

5. **2–D Image Histogram Computation**

   The same benchmark from Chapter 3.4, except with six different smaller image in-

put sizes being $\{128 \times 128\}$, $\{256 \times 256\}$, $\{512 \times 512\}$, $\{1024 \times 1024\}$, $\{2048 \times 2048\}$, $\{4096 \times 4096\}$.

4.4.3 Experimental Methodology

The methodology used to evaluate each of the benchmarks described in the previous section is to do the following experimental comparisons. In all of these experiments, unless specified, the benchmark was run numerous times using different processor or tile configurations in order to empirically determine the configuration that leads to a minimal execution time.

1. Outer Loop Parallelization

   Only the outer loop is to be parallelized with the remainder of the loop nest being sequential. The amount of parallelism to be used is the configuration that returns the fastest execution time by doing an exhaustive search of 1–D processor configurations.

2. Nested Parallelization

   Nested parallelization will be applied to all of the data-parallel loops that are used in the benchmarks. The amount of parallelism applied to each loop is the configuration that returns the fastest execution time by doing an exhaustive search of 2–D processor configurations.

3. Loop Coalesced

   Apply loop coalescing to the loop nest and then statically schedule the parallel loop onto the selected number of processors. In this experiment, other than the 1–D partitioning from parallelism, no explicit tiling is performed. The amount of parallelism to be used is the configuration that returns the fastest execution time by doing an exhaustive search of 1–D processor configurations.

4. Heuristic

Use the heuristic from Section 4.10 to determine ideal processor configuration, and then apply the tiled-coalesced loop transformation using the computed blocking factors.

5. Maximum Parallelism

   Use the maximum number of processors applied to the tiled-coalesced loop transformation.

6. Fastest Exhaustive (*baseline*)

   Performs an exhaustive search for tile dimensions that result in a minimal (*fastest*) execution time which is then applied to the tiled-coalesced loop transformation. This benchmark will be set to the baseline measurement in the experiment evaluation.

7. Tile-Coalesced (*exhaustive - slowest*)

   Performs an exhaustive search for tile dimensions that result in a maximum (*slowest*) execution time, and then apply the tiled-coalesced loop transformation.

For all of the subsequent evaluations, the base metric that is used is the overall *slowdown* compared to the fastest execution time when using a tiled-coalesced exhaustive search:

$$slowdown = \frac{experiment}{tiled\_coalesced_{exhaustive\_fastest}}$$

Thus, when $slowdown < 1$, the particular experiment executed faster than the baseline "Fastest Exhaustive" experiment.

An additional metric that is provided for each benchmark is a table comparing the tile sizes computed by the heuristic versus an exhaustive search using the tiled-coalesced technique. This will show how close the dimension sizes and areas are between the two techniques.

4.4.4 Results

This section will first present a summary of the results, followed by a more in depth analysis of each of the benchmarks measured.

Results Summary

Figures 4.11-4.17 present the performance results for the different benchmarks described earlier in this section. The general trend that is evident in these results show that in the vast majority of cases the heuristic is able to find tile sizes (i.e. multi-dimensional processor configuration) that are similar to tile sizes found through an exhaustive search of all possible sizes. This then implies that the heuristic works.

In many cases, the exact same tile size is found between the heuristic and an exhaustive search, and in other cases, tile sizes are found to have an area that is similar between both techniques. Only in certain scenarios, such as in SpMV, does the heuristic sometimes have trouble finding an ideal tile size compared to the exhaustive search. For example, in Figure 4.14d, selecting an incorrect tile size by the heuristic leads a to a slowdown of 38% when the vector size is set at 2. The reason for this primarily due to the irregular workload associated with SpMV that does occur in the other benchmarks.

Another conclusion is that outer-level parallelism does not perform as well compared to the loop techniques when there is little to parallelize at the outer-level. This is evident in the results from SpMV in Figures 4.14. As the number of outer-level vectors increase, does the performance of outer-level parallelism start to converge with the other approaches.

In many cases, loop coalescing performs on par with the heuristic and the exhaustive search. This implies that in many of these benchmarks, the tiling of the iteration space is not effectively being leveraged to its full capability. However, in some scenarios such as Figures 4.14 and 4.16 does loop coalescing perform significantly worse than the heuristic, in some cases close to a 4x slowdown.

The nested parallelism experiment performs noticeably worse compared to many of the other loop techniques, and is never close to the heuristic. The major reason for this is due to the vast amount of associated overheads such as the dynamic spawning of inner loop tasks and their barriers, compared to the actual amount of useful computation for the loop.

3–D Jacobi Iteration

Figures 4.11-4.13 present the evaluation of different loop strategies that are compiled onto a dependence-driven runtime system for a 3–D Jacobi Method computation. This evaluation occurs on a set of 3-D grids, where for each chart, the x and z component of the 3-D grid space remain fixed, and the y component increases from $1 \rightarrow 16$.

In all of these results, it is clear that the heuristic matches the exhaustive search, thus demonstrating that not using all available parallelism can be beneficial depending on the workload. This result is backed up by Table 4.4, which provides the computed tile dimensions of both the heuristic (*top value*) and that of an exhaustive search using the tiled-coalesced loop transformation (*bottom value*). As shown in this table, in the vast majority of cases, the heuristic either matches that of the exhaustive search, or is very close to it in terms of the tile area.

Initially when the x and z dimensions are small, the overhead is high enough that there is no benefit to any parallelism. As the problem dimension sizes (all x, y, z) increase, the performance of the maximum parallelism experiment will start to converge with that from the heuristic and exhaustive scenarios, while the slowest exhaustive test will diverge from the rest, thus showing the parallelism in this case starts to make a positive impact on the overall performance of the loops.

In the case of nested parallelism, the overhead associated with the barriers and the dynamic spawning of tasks in the inner loops causes a performance degradation. In many cases, as the number of columns continues to grow, the performance starts to decrease even

more, even surpassing the slowest exhaustive search.

For this benchmark, loop coalescing performs similarly to the tiled-coalesced form, thus in this case, there is no clear benefit to tiling (for parallelism).

It is interesting to note that when comparing the outer loop parallel scenario, it performs similar to the other experiments. This occurs due to the relatively small input sizes that are being used. However, as Figure 4.13a demonstrates, when the problem size starts to become larger ($k = 512$), and there is no parallelism on the outer loop, then logically performance will start to suffer.

Sparse Matrix-Vector Multiplication (SpMV)

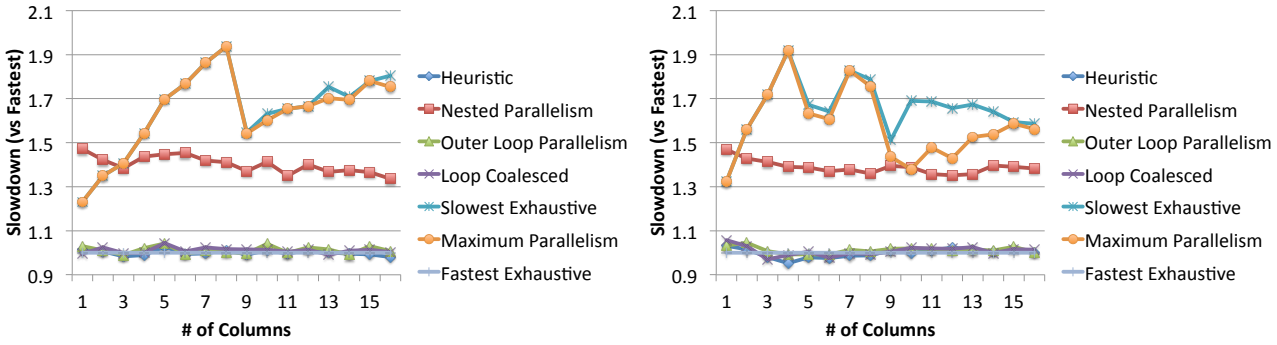
Figure 4.14 presents performance results for the SpMV computation across five different matrices.

Compared to the other loop techniques, the heuristic closely follows the exhaustive search in many cases, even matching the exact tile shape and sizes. In the worst case, as shown in Figure 4.14d, there is a slowdown of 38% when the vector size is 2. Looking at Table 4.5, the heuristic finds a tile dimension of 1×11, and the exhaustive search found the fastest dimension to be 1×13.

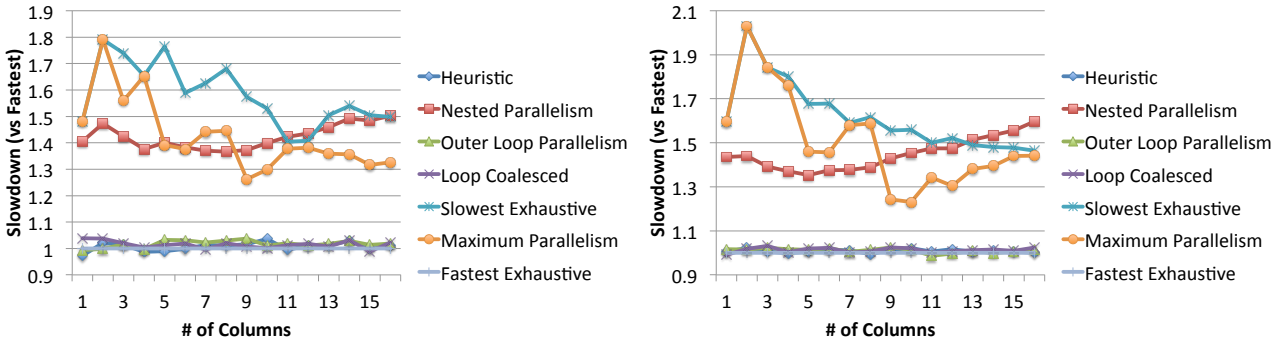
The nested parallelism approach typically has the worst performance, sometimes up to 350% slowdown as shown in Figure 4.14a.

In the case of using outer level parallelism when the number of vectors is small, the performance is limited. This is primarily due to the impact from a load imbalance, and not enough resources to parallelize at the outer loop level. As the number of vectors increases, the performance of outer loop parallelism will start to converge with the other techniques.

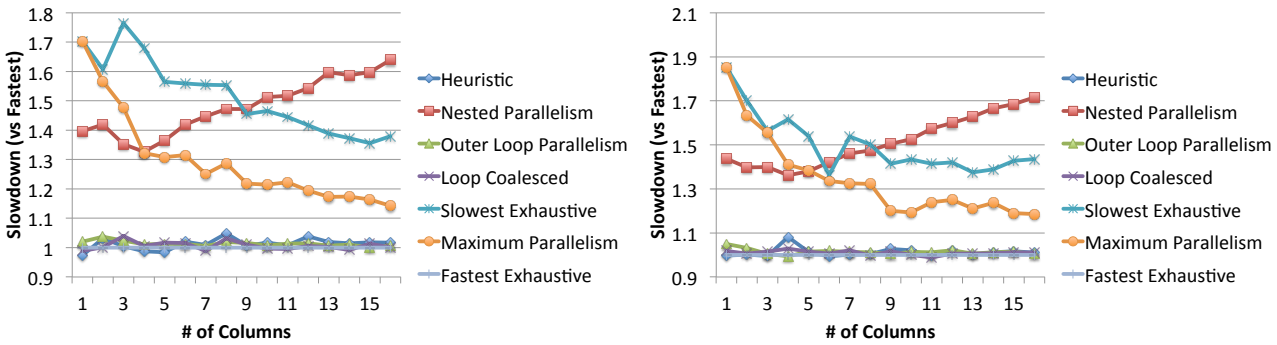
Traditional loop coalescing is also a detriment on performance in this scenario. For example, in Figure 4.14a, there is an 88% slowdown when a vector size of 1 is used. In some cases, loop coalescing performs just as well, or even slightly surpasses that of the base metric.



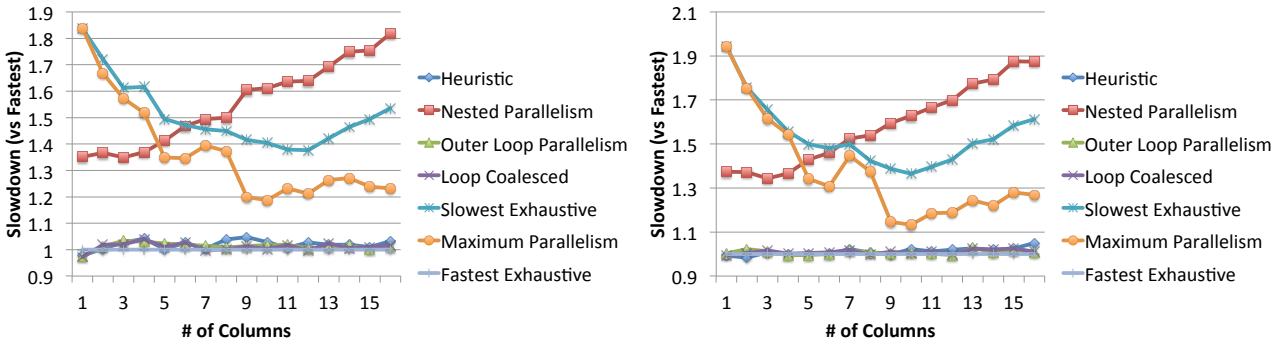
(a) $2 \times [1:16] \times 128$ (b) $4 \times [1:16] \times 128$



(c) $6 \times [1:16] \times 128$ (d) $8 \times [1:16] \times 128$

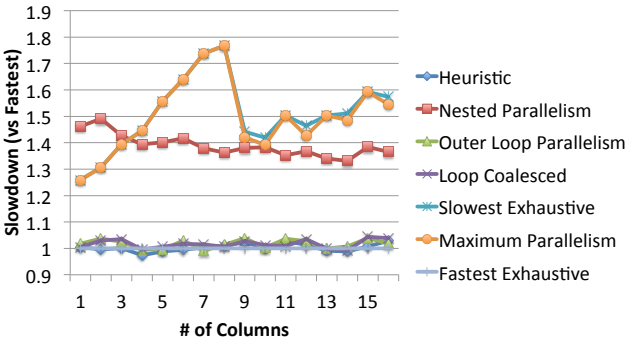


(e) $10 \times [1:16] \times 128$ (f) $12 \times [1:16] \times 128$

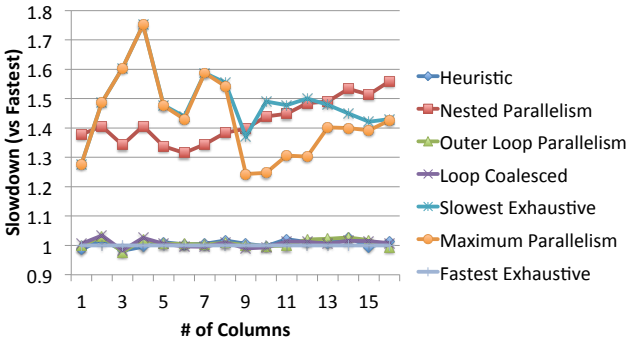


(g) $14 \times [1:16] \times 128$ (h) $16 \times [1:16] \times 128$

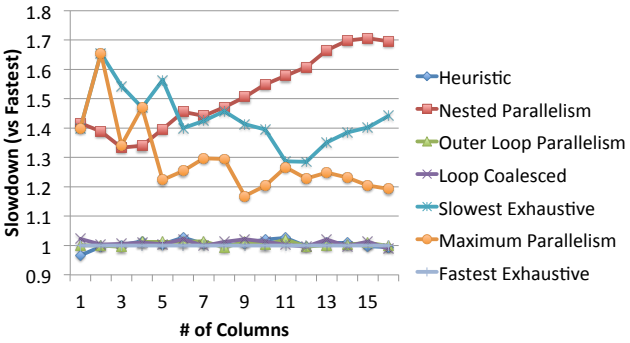
Figure 4.11: 3–D Jacobi Iteration : $[1:16] \times [1:16] \times 128$



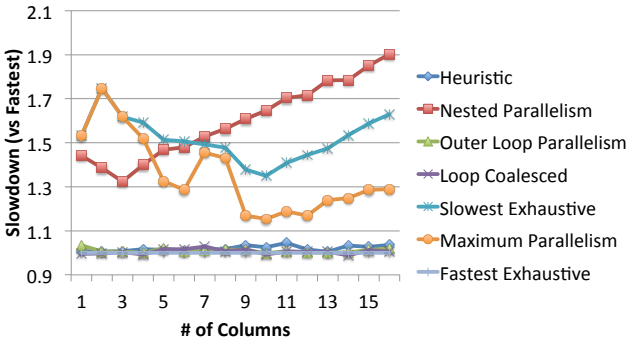
(a) $2 \times [1:16] \times 256$



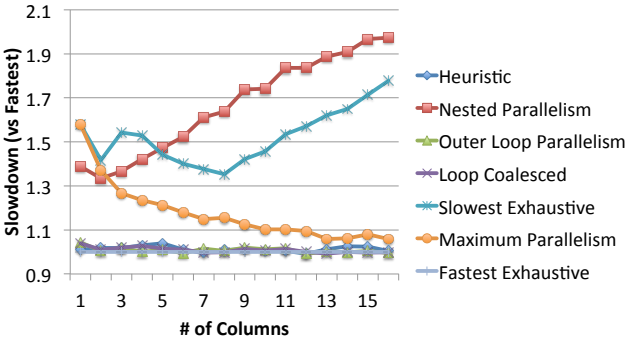
(b) $4 \times [1:16] \times 256$



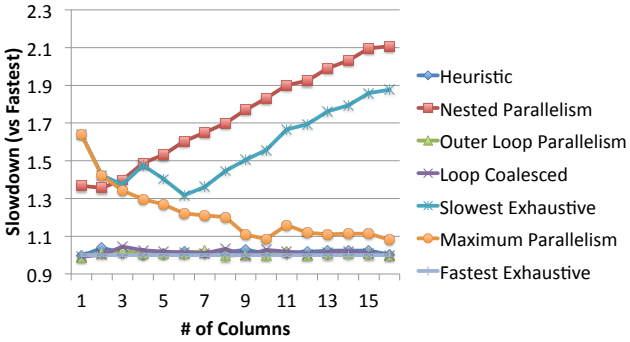
(c) $6 \times [1:16] \times 256$



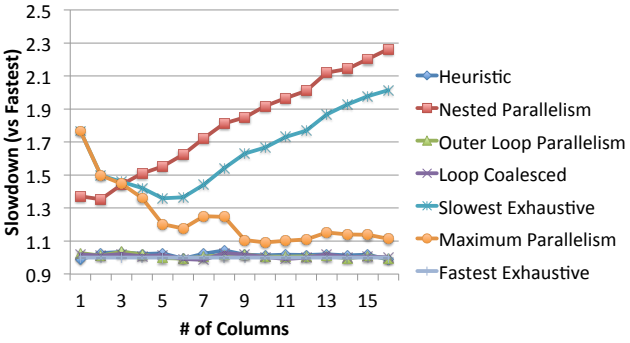
(d) $8 \times [1:16] \times 256$



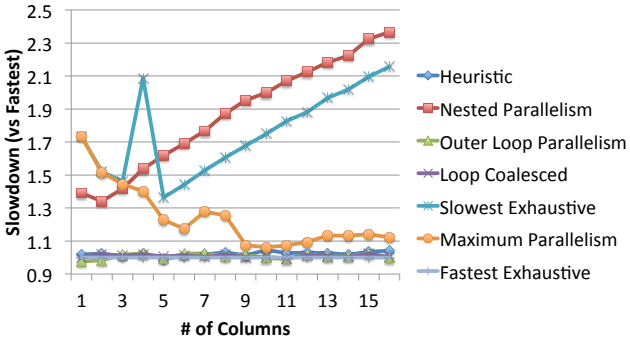
(e) $10 \times [1:16] \times 256$



(f) $12 \times [1:16] \times 256$

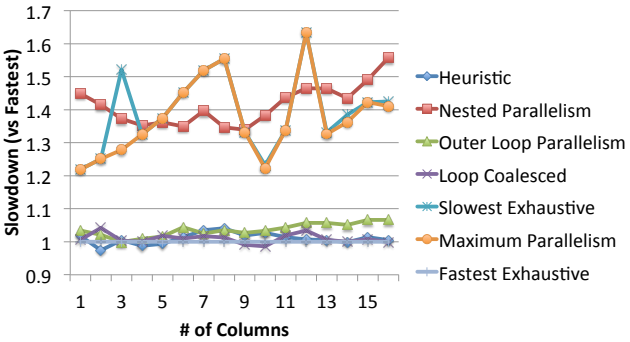


(g) $14 \times [1:16] \times 256$

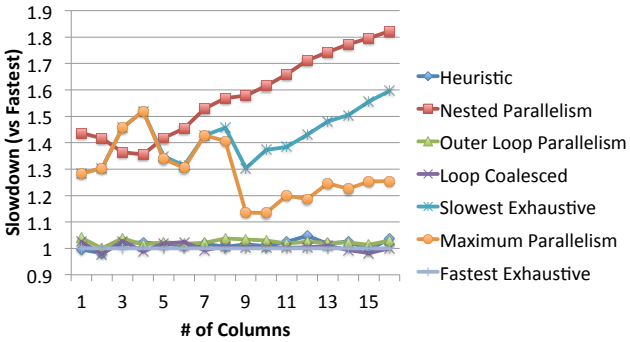


(h) $16 \times [1:16] \times 256$

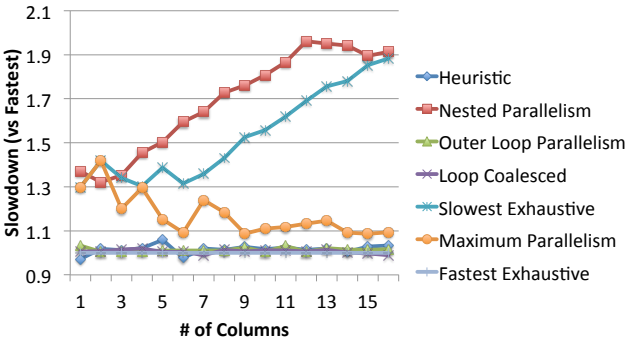
Figure 4.12: 3–D Jacobi Iteration : $[1:16] \times [1:16] \times 256$



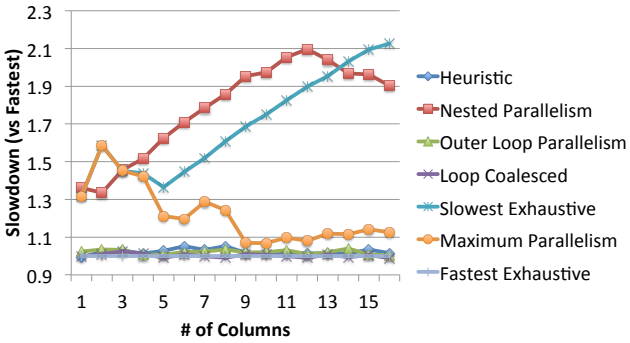
(a) $2 \times [1:16] \times 512$



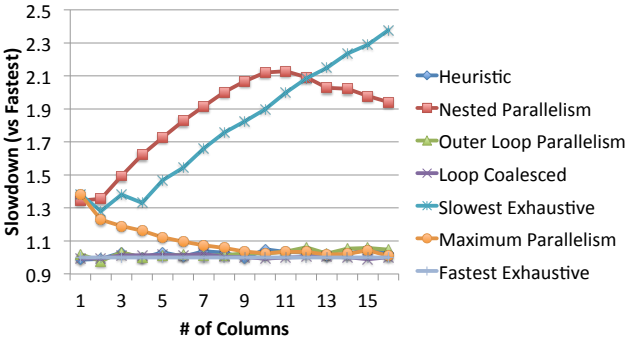
(b) $4 \times [1:16] \times 512$



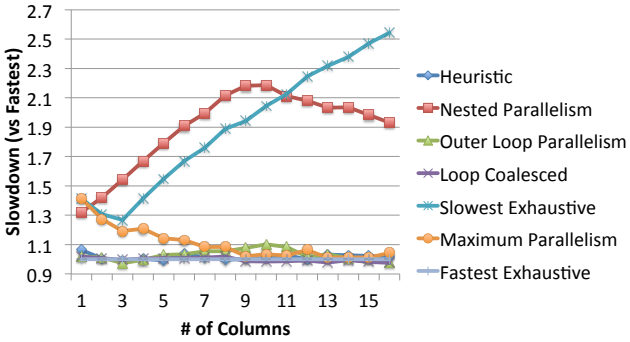
(c) $6 \times [1:16] \times 512$



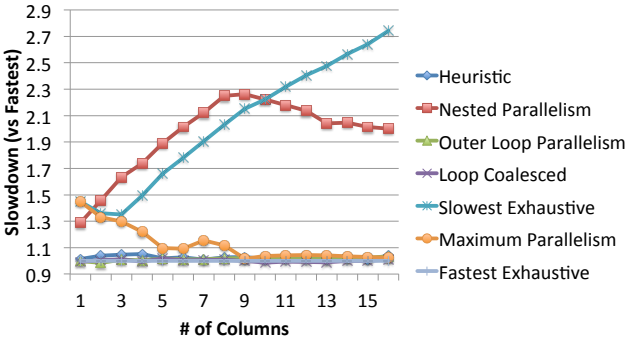
(d) $8 \times [1:16] \times 512$



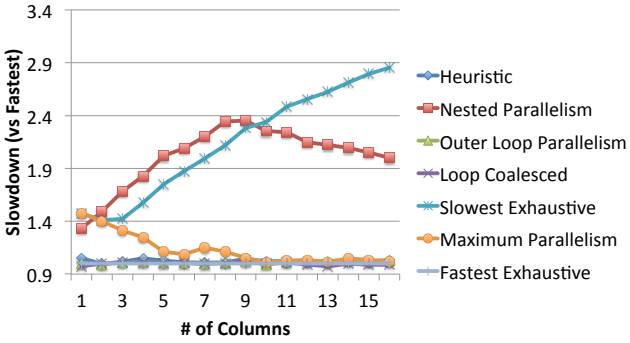
(e) $10 \times [1:16] \times 512$



(f) $12 \times [1:16] \times 512$



(g) $14 \times [1:16] \times 512$



(h) $16 \times [1:16] \times 512$

Figure 4.13: 3–D Jacobi Iteration : $[1:16] \times [1:16] \times 512$

Rows \ Columns	2	4	6	8	10	12	14	16
2	1 × 1 1 × 1	1 × 1 1 × 1	1 × 1 1 × 1	1 × 1 1 × 1	1 × 1 1 × 1	1 × 1 1 × 1	1 × 1 1 × 1	1 × 1 1 × 1
4	1 × 1 1 × 1	1 × 1 1 × 1	1 × 1 1 × 1	1 × 1 1 × 1	1 × 1 1 × 1	1 × 2 1 × 2	1 × 2 1 × 3	1 × 2 1 × 3
6	1 × 1 1 × 1	1 × 1 1 × 1	1 × 1 1 × 1	1 × 2 1 × 3	3 × 1 1 × 2	1 × 3 5 × 1	3 × 1 4 × 1	3 × 1 4 × 1
8	1 × 1 1 × 1	1 × 1 1 × 1	1 × 2 3 × 1	1 × 2 2 × 1	4 × 1 5 × 1	1 × 3 3 × 1	4 × 1 6 × 1	1 × 4 5 × 1
10	1 × 1 1 × 1	1 × 1 1 × 1	1 × 3 2 × 1	1 × 4 1 × 3	2 × 2 1 × 4	1 × 4 2 × 2	2 × 2 2 × 2	1 × 4 9 × 1
12	1 × 1 1 × 1	1 × 2 1 × 2	1 × 3 3 × 1	3 × 1 2 × 2	4 × 1 5 × 1	1 × 4 5 × 1	4 × 1 4 × 1	1 × 4 7 × 1
14	1 × 1 1 × 1	1 × 2 3 × 1	1 × 3 3 × 1	1 × 4 4 × 1	2 × 2 5 × 1	1 × 4 6 × 1	2 × 2 5 × 1	1 × 4 3 × 2
16	1 × 1 1 × 1	1 × 2 1 × 3	1 × 3 5 × 1	1 × 4 1 × 4	4 × 1 2 × 2	1 × 4 3 × 2	4 × 1 1 × 5	1 × 4 1 × 6

(a) $[1:16] \times [1:16] \times 128$

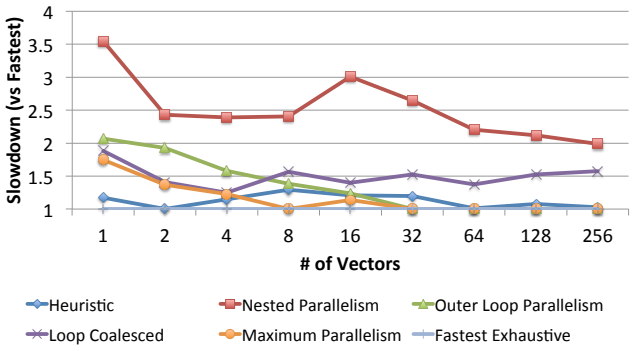
Rows \ Columns	2	4	6	8	10	12	14	16
2	1 × 1 1 × 1	1 × 1 1 × 1	1 × 1 1 × 1	1 × 1 1 × 1	1 × 1 1 × 1	1 × 2 1 × 1	1 × 2 1 × 3	1 × 2 1 × 3
4	1 × 1 1 × 1	1 × 1 1 × 1	1 × 2 3 × 1	1 × 2 1 × 3	4 × 1 3 × 2	1 × 3 1 × 4	4 × 1 1 × 3	1 × 4 4 × 1
6	1 × 1 1 × 1	1 × 2 2 × 1	1 × 3 5 × 1	3 × 1 1 × 5	2 × 2 2 × 2	1 × 4 1 × 3	2 × 2 1 × 6	1 × 4 6 × 1
8	1 × 1 1 × 1	1 × 2 3 × 1	1 × 3 4 × 1	1 × 4 1 × 5	4 × 1 5 × 1	1 × 4 4 × 1	4 × 1 4 × 1	1 × 4 3 × 2
10	1 × 1 1 × 1	1 × 4 2 × 3	2 × 2 1 × 5	1 × 4 9 × 1	1 × 5 3 × 2	5 × 1 3 × 2	5 × 1 8 × 1	5 × 1 1 × 7
12	1 × 2 1 × 2	3 × 1 3 × 1	4 × 1 5 × 1	1 × 4 5 × 1	1 × 5 7 × 1	1 × 6 8 × 1	6 × 1 2 × 3	6 × 1 10 × 1
14	1 × 2 3 × 1	1 × 4 3 × 1	2 × 2 6 × 1	1 × 4 6 × 1	1 × 5 1 × 6	1 × 6 1 × 6	1 × 7 1 × 11	7 × 1 1 × 8
16	1 × 2 2 × 1	1 × 4 2 × 3	4 × 1 3 × 2	1 × 4 7 × 1	1 × 5 3 × 2	1 × 6 13 × 1	1 × 7 3 × 3	1 × 8 9 × 1

(b) $[1:16] \times [1:16] \times 256$

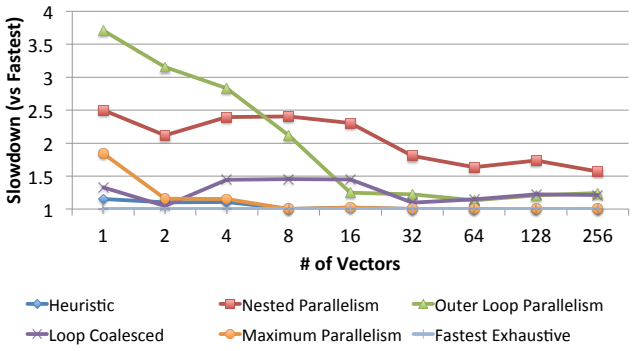
Rows \ Columns	2	4	6	8	10	12	14	16
2	1 × 1 1 × 1	1 × 1 1 × 1	1 × 3 2 × 1	1 × 4 1 × 2	2 × 2 2 × 2	1 × 4 1 × 3	2 × 2 1 × 5	1 × 4 1 × 5
4	1 × 1 1 × 1	1 × 4 3 × 1	4 × 1 1 × 3	1 × 4 3 × 2	1 × 5 1 × 7	1 × 6 1 × 10	1 × 7 4 × 1	1 × 8 3 × 3
6	3 × 1 1 × 1	1 × 4 4 × 1	2 × 2 3 × 1	6 × 1 1 × 6	6 × 1 1 × 7	1 × 6 6 × 1	1 × 7 5 × 2	1 × 8 1 × 13
8	4 × 1 3 × 1	1 × 4 1 × 4	1 × 6 3 × 2	1 × 8 3 × 2	8 × 1 3 × 2	8 × 1 3 × 3	8 × 1 3 × 3	1 × 8 1 × 15
10	2 × 2 2 × 2	5 × 1 2 × 3	1 × 6 3 × 2	1 × 8 3 × 3	1 × 10 3 × 2	2 × 4 3 × 3	10 × 1 3 × 3	10 × 1 7 × 2
12	4 × 1 3 × 1	6 × 1 3 × 2	1 × 6 9 × 1	1 × 8 6 × 1	4 × 2 4 × 2	3 × 3 3 × 3	4 × 2 7 × 2	12 × 1 2 × 7
14	2 × 2 4 × 1	7 × 1 6 × 1	7 × 1 4 × 2	1 × 8 4 × 2	1 × 10 6 × 2	2 × 4 4 × 3	2 × 5 3 × 3	14 × 1 6 × 2
16	4 × 1 2 × 2	8 × 1 3 × 3	8 × 1 2 × 4	1 × 8 14 × 1	1 × 10 2 × 7	1 × 12 3 × 4	1 × 14 3 × 5	1 × 16 3 × 5

(c) $[1:16] \times [1:16] \times 512$

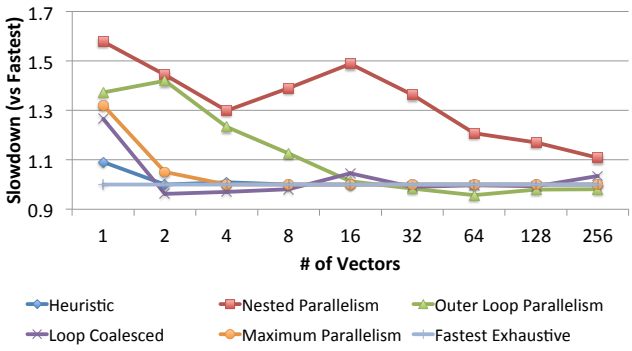
Table 4.4: Tile-Dimensions (ROWS x COLS) for Jacobi 3–D : Heuristic (top) vs Exhaustive Search (bottom)



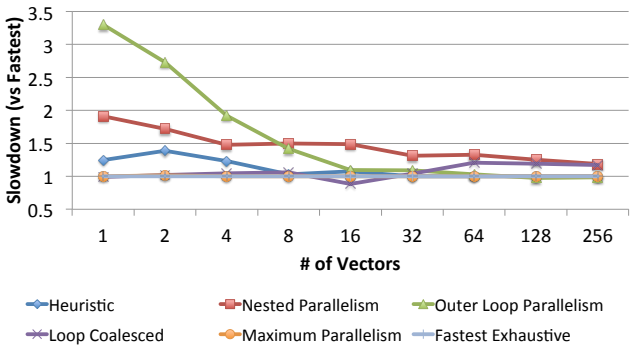
(a) SpMV: Meszaros/ex3sta1



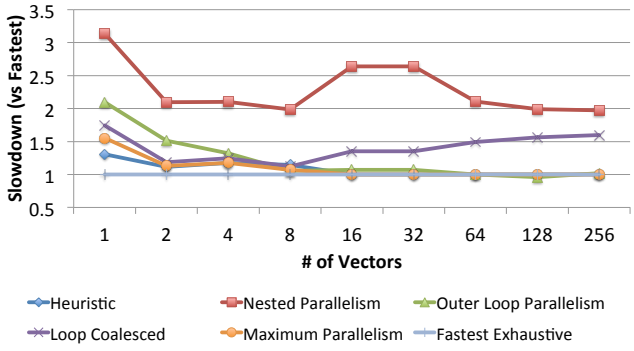
(b) SpMV: stat96v5



(c) SpMV: lp_osa_14



(d) SpMV: Andrianov/ex3sta1



(e) SpMV: Rommes/bips07_2476

Figure 4.14: Sparse Matrix-Vector Multiplication With Multiple Vectors

Sparse Matrices	Vectors								
	1	2	4	8	16	32	64	128	256
Meszaros/ex3sta1	1 × 5 1 × 7	1 × 7 1 × 7	1 × 9 1 × 14	1 × 14 1 × 16	1 × 16 3 × 5	1 × 16 16 × 1	1 × 16 16 × 1	1 × 16 16 × 1	1 × 16 16 × 1
Meszaros/stat96v5	1 × 6 1 × 13	1 × 8 1 × 9	1 × 12 1 × 13	1 × 16 1 × 16	1 × 16 1 × 14	1 × 16 1 × 16	1 × 16 1 × 16	1 × 16 1 × 16	1 × 16 1 × 16
LPnetlib/lp_osa_14	1 × 7 1 × 9	1 × 10 2 × 4	1 × 15 4 × 4	1 × 16 8 × 2	1 × 16 16 × 1	1 × 16 16 × 1	1 × 16 16 × 1	1 × 16 16 × 1	1 × 16 16 × 1
Andrianov/ex3sta1	1 × 8 1 × 16	1 × 11 1 × 13	1 × 16 1 × 16	1 × 16 4 × 4	1 × 16 8 × 2	1 × 16 16 × 1	1 × 16 16 × 1	1 × 16 16 × 1	1 × 16 16 × 1
Rommes/bips07_2476	1 × 5 1 × 7	1 × 7 2 × 6	1 × 10 1 × 9	1 × 14 4 × 3	1 × 16 16 × 1	1 × 16 16 × 1	1 × 16 16 × 1	1 × 16 16 × 1	1 × 16 16 × 1

Table 4.5: Tile-Dimensions (ROWS x COLS) for SpMV : Heuristic (top) vs Exhaustive Search (bottom)

Coulombic Potential (CP)

The performance of Coulombic Potential is given in Figure 4.15. With the exception of one grid dimension, the heuristic either matches, or is extremely close to the exhaustive search technique. In the worst case at dimension 8×8, as Table 4.6 shows, the heuristic selects a tile size of 8×1, and the fastest exhaustive search finds a tile dimension of 3×5. This leads to a slowdown of 16%.

With the exception of using nested parallelism, all of the other loop techniques perform similarly. Only when the grid dimension becomes big enough (64×64) does the overhead of using nested parallelism start to recede.

Tile Dimensions (ROWS X COLUMNS)				
4 × 4	8 × 8	16 × 16	32 × 32	64 × 64
4 × 1 4 × 1	8 × 1 3 × 5	16 × 1 2 × 8	16 × 1 8 × 2	16 × 1 16 × 1

Table 4.6: Tile-Dimensions (ROWS x COLS) for Coulombic Potential : Heuristic (top) vs Exhaustive Search (bottom)

2–D Image Histogram Computation

Figure 4.16 presents the performance results for the 2–D Histogram benchmark.

In most cases, the heuristic finds tile dimensions that closely match or are the same as

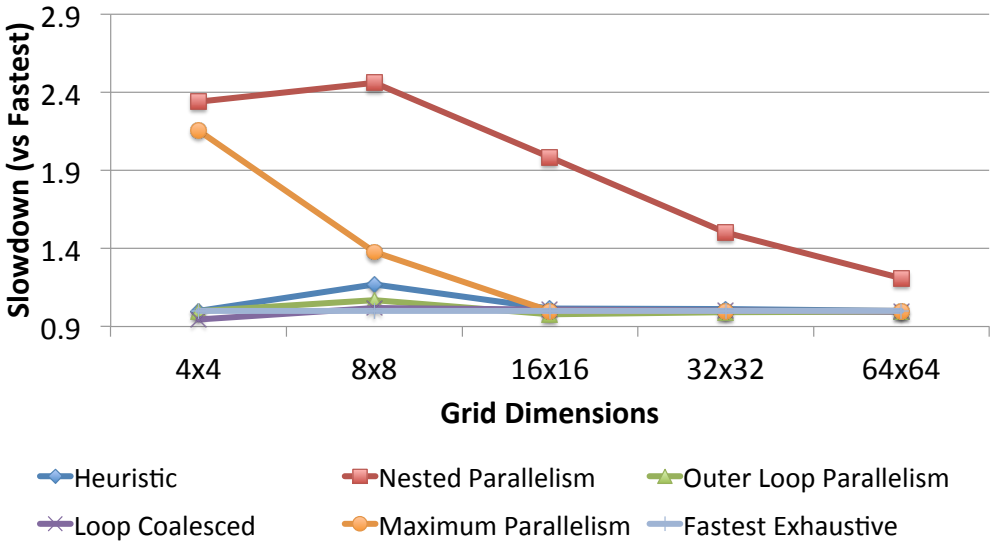


Figure 4.15: Coulombic Potential

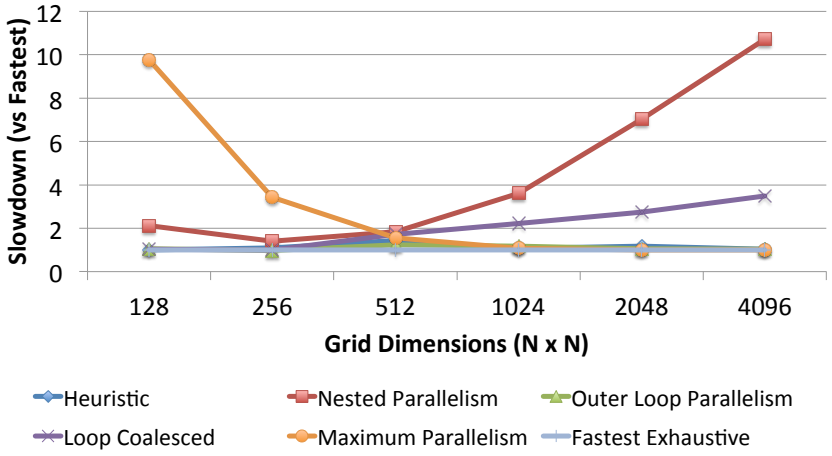


Figure 4.16: 2–D Image Histogram

Tile Dimensions (ROWS X COLUMNS)					
128×128	256×256	512×512	1024×1024	2048×2048	4096×4096
1×1	4×1	8×1	16×1	16×1	16×1
1×1	1×1	1×4	1×15	5×3	8×2

Table 4.7: Tile-Dimensions (ROWS x COLS) for 2–D Image Histogram Computation : Heuristic (top) vs Exhaustive Search (bottom)

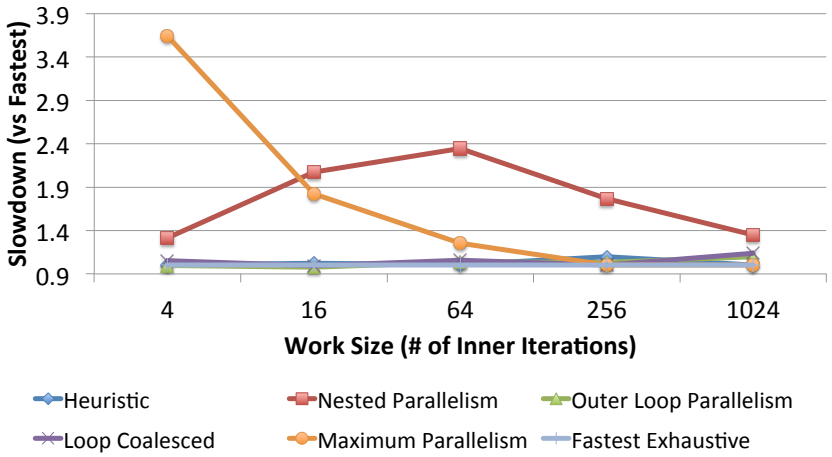


Figure 4.17: Synth.Trig

the exhaustive technique. The only two exceptions to this occurs when the image size is 256×256 and 512×512. In this scenarios, as Table 4.7 shows, the heuristic finds a tile dimension of 4×1, while the exhaustive technique finds a tile dimension of 1. This leads to a performance slowdown of 10%. When the image size is 512×512, the heuristic finds a tile of 8×1, and the exhaustive techniques finds a tile dimension of 1×4. This leads to a performance slowdown of 43%.

The nested parallel approach leads to a significant amount of slowdown (up to 1070% at image size of 4096×4096). This happens primarily because of small amount of useful computation relative to the overheads introduced.

Loop coalescing performs reasonably well when the image sizes are small ($< 512 \times 512$). As the size of image increases, the impact on locality takes its toll, and coalescing has a slowdown of 350% at an image of 4096×4096.

Synthetic.Trig

Figure 4.17 present the performance results for the Synthetic.Trig benchmark.

Similar as before, the heuristic matches or is closely the same as the exhaustive search technique. The worst performing tile found is when the number of inner iterations is 256.

Tile Dimensions (ROWS X COLUMNS)				
4	16	64	256	1024
1×1	1×4	1×8	1×16	1×16
1×1	2×2	3×3	2×8	1×16

Table 4.8: Tile-Dimensions (ROWS x COLS) for Synth.Trig : Heuristic (top) vs Exhaustive Search (bottom)

In this case, as Table 4.8 shows, the heuristic computes a tile dimension of 1×16, and the exhaustive search discovers a tile dimension of 2×8. This leads to a performance slowdown of 9.8%.

Another similar trend has been the performance slowdown of using nested parallelism compared to the other loop techniques. For example, when the number of inner iterations are at 64, nested parallelism has a slowdown of 234%.

4.5 Limitations

The heuristic introduced in this chapter is best suited in the situation where parallelism for loops is worthwhile, but not to the point of using all available processors. In the scenario where the workload is large enough, the user (or compiler) can dedicate all the parallel resources to the parallel loop nest without the need of an algorithm to help guide them.

As the results show, in many cases, the heuristic works well for loops with regular data access patterns, but in some cases struggles with codes that have irregular access patterns. An approach that was not taken here, but could result in better performance for irregular computations, is to do on-line sampling of the execution time for loops with a given processor configuration. During execution, the runtime system can then dynamically adjust these resources to potentially increase performance that would be difficult to predict statically in the compiler.

While the heuristic determines block sizes suitable for coarse-grain parallelism, the tile

itself is left alone. Instead, the tile could be further decomposed into sub-tiles suitable for memory locality. This would effectively be performing multi-level tiling.

4.6 Related Work

4.6.1 Loop Coalescing

Loop coalescing is a well-known and studied loop transformation technique that most of the work from this chapter is based on. The main distinction is that the approach introduced in the thesis is based on tiling the iteration spaces first, then coalescing them together, and then finally statically scheduling the chunks onto the different processors. In the traditional loop coalescing transformation, loop coalescing does not deal with locality at all; it simply combines the loop nests' iteration spaces together, and then schedules the blocked iteration space onto the processors.

Sarkar [99, 100] and Wolf [101] present similar techniques to perform loop coalescing as part of a framework to perform loop transformations. Sarkar presents a framework to perform iteration-reordering transformations such as blocking, parallelization, and coalescing. In the case of Wolf et al., they present a framework based on unimodular loop transformations, where the last step in the compilation process is to perform loop coalescing. In both cases, the maximum amount of parallelism is used, whereas in this thesis, part of the goal of the heuristic is to determine an ideal number of processors to use for the given workload and the associated runtime overheads. Additionally, in the case of Sarkar, the input program's loops always start sequential, and then through a set of unimodular loop transformations, are made parallel. Only at the end of this process is loop coalescing then applied. In both cases, dependence analysis also needs to be performed.

4.6.2 Tile Size and Shape Selection

There has been a substantial amount of research in analytically determine data locality [95, 97, 92]. For example, Coleman and McKinley [95] present a method of automatically selecting tile sizes based on the cache organization of the machine. In their research, they focus on techniques to get rid of capacity and self-interference misses, while reducing cross-interference misses as well. An important distinction is that in this thesis, tile sizes were determined for parallelism, and in their case, they are tiling for locality. One area of future work would be to incorporate their work into this thesis so that parallel tiles could be further tiled for locality.

4.7 Discussion

This chapter presented an optimization to parallel loop nests that combines tiling and loop coalescing in order to decrease synchronization overhead for dependence-driven execution runtime systems, and increase locality for coarse grain tasks. Part of this optimization is a heuristic that leverages a cost model built from micro-benchmarks that determine runtime overheads in order to determine the ideal processor configuration comprised of tile shapes and sizes.

Chapter 5

Compilation for Heterogeneous Architectures

5.1 Introduction

Programmability and the ability to optimize for performance and power are considered major difficulties introduced by heterogeneous systems. These difficulties arise for two main reasons. First, with todays tools, it is necessary to use a different programming model for each system component: CUDA [36], OpenCL [50], or OpenACC [102] is often used to program GPU architectures, C or C++ extended with OpenMP [33] or Intel TBB [19] are used for conventional multi-cores, and MPI is used for distributed memory clusters. This results in an increased complexity in programming and porting across different architectures, as one must now fully develop and maintain separate copies of the code. The second difficulty is the need to schedule across device classes: the user must decide how to partition and correctly schedule the execution between the devices. This difficulty is typically compounded by each device having separate address spaces, forcing the user to take care of the allocation, deallocation, and communication of data across devices.

This chapter builds on the native data parallel support of Chapel to improve the programmability of heterogeneous systems containing GPU accelerator components, while retaining performance and portability across other architectures. Rather than rely completely on the compiler for performance optimizations, this work leverages Chapel's multiresolution philosophy of allowing a programmer to start with a high-level specification, then drop to a lower level if the compiler is not providing sufficient performance. This gives expert programmers the ability to tune their algorithm's performance with capabilities similar to those of lower level notations such as CUDA.

In Chapter 3, the focus was on using the Chapel compiler to target a software-based dependence-driven runtime system. This chapter will instead focus on retargeting the Chapel program so that it can be mapped and tuned for a GPU, while still providing backward support to map the code back onto a multi-core platform. This retargeting of Chapel code onto both GPU and multi-core platforms is provided through Chapel's support for user-defined distributions [22]. The eventual goal will be to combine both together so that tasks can execute concurrently on both traditional multi-core and many-core GPU architectures.

Contributions

- The presentation of a high-level and portable approach for developing applications with a single unified language, instead of libraries or annotations, that can target both GPU and multi-core parallel architectures. This would enable a single code that can be used to efficiently target GPUs, traditional multi-cores, or a combination of both.

- Compiler transformations that map a high-level language onto GPU accelerator architectures. This also includes an algorithm for moving data automatically between a host and the device. These techniques would be applicable to other high-level languages, such as Python or Java, with the goal of targeting GPUs.

- A compiler algorithm is presented to generate code for multi-cores from Chapel applications that have been hand-tuned for GPU architectures. In order to run efficiently, this technique extracts coarse-grain parallelism from code that contains both fine and coarse granularities.

- Experimental results to show that the performance of the hand-coded CUDA Parboil benchmarks [71] are comparable to the Chapel implementation for both GPUs and multi-cores, with the Chapel code being simpler, shorter, and easier to read and maintain.

To measure the validity of the proposed approach, the Parboil benchmark suite was ported to Chapel. The Chapel compiler was then modified to generate the appropriate code based on the target architecture. This chapter evaluates code generation techniques for both GPUs and multi-cores. The performance results show that Chapel generates codes that are competitive with the hand-tuned CUDA implementations for GPUs and their multi-core counterparts.

This chapter is organized as follows: Section 5.2 gives motivation for this work by providing a simple example. Sections 5.3 and 5.4 provide the implementation details for running on GPU and CPU architectures. Section 5.5 discusses optimizations applied to the GPU. Section 5.6 presents short Parboil benchmark examples written in Chapel used to target both CPUs and GPUs. Section 5.7 describes the initial results using the Parboil benchmark suite. Sections 5.8 and 5.9 present what the limitations of this implementation are, and then some related work. Finally, conclusions are provided in Section 5.10.

5.2 Motivation

As a motivating example, consider the STREAM Triad benchmark (part of the HPCC Benchmark Suite [103]), which computes a scaled vector addition. Figure 5.1 provides a comparison

```
#define N 2000000
int main() {
 float *host_a, *host_b, *host_c;
 float *gpu_a, *gpu_b, *gpu_c;
 cudaMalloc((void**)&gpu_a, sizeof(float)*N);
 cudaMalloc((void**)&gpu_b, sizeof(float)*N);
 cudaMalloc((void**)&gpu_c, sizeof(float)*N);
 dim3 dimBlock(256);
 dim3 dimGrid(N/dimBlock.x );
 if( N % dimBlock.x != 0 ) dimGrid.x+=1;
 set_array<<<dimGrid,dimBlock>>>(gpu_b,0.5f,N);
 set_array<<<dimGrid,dimBlock>>>(gpu_c,0.5f,N);
 float scalar = 3.0f;
 STREAM_Triad<<<dimGrid,dimBlock>>>(gpu_b,
     gpu_c, gpu_a, scalar, N);
 cudaThreadSynchronize();
 cudaMemCpy(host_a, gpu_a, sizeof(float)*N,
     cudaMemcpyDeviceToHost);
 cudaFree(gpu_a);
 cudaFree(gpu_b);
 cudaFree(gpu_c);
}
__global__ void set_array(float *a, float value,
    int len) {
 int idx = threadIdx.x+blockIdx.x*blockDim.x;
 if(idx < len) a[idx] = value;
}
__global__ void STREAM_Triad(float *a, float *b,
    float *c, float scalar, int len) {
  int idx = threadIdx.x+blockIdx.x*blockDim.x;
  if(idx < len) c[idx] = a[idx]+scalar*b[idx];
}
```

(a) STREAM Triad written in CUDA

```
const alpha = 3.0;
config const N = 2000000;
const space = [1..N] dmapped GPUDist(rank=1);
var A, B, C : [space] real;
B = 0.5;
C = 0.5;
forall (a,b,c) in (A,B,C) do
      a = b + alpha * c;
```

(b) STREAM Triad written in Chapel for a GPU

```
const alpha = 3.0;
config const N = 2000000;
const space = [1..N] dmapped Block(boundingBox=[1..N]);
var A, B, C : [space] real;
B = 0.5;
C = 0.5;
forall (a,b,c) in (A,B,C) do
      a = b + alpha * c;
```

(c) STREAM Triad written in Chapel for a cluster

Figure 5.1: Comparison of STREAM Triad Implementations

of different implementations of STREAM Triad. A CUDA implementation is provided in Figure 5.1(a), while Figure 5.1(b) is a Chapel implementation used to target a GPU. The comparison between them shows that the Chapel implementation has noticeably fewer lines of code and is more readable. This is achieved using Chapel distributions, domains, data parallel computations through the `forall` loop, and variable type inference [51, 21]. Furthermore, the Chapel implementation is easier to port. In fact, for the code in Figure 5.1(b), if users wanted to target a multi-core platform, they could either declare a different target distribution (shown in line 3) or simply set an environment variable specifying the target platform. This declaration specifies how the index set as well as arrays and loops declared over it, is to be mapped to the target architecture. To demonstrate portability, Figure 5.1(c) shows an implementation of STREAM Triad written for a cluster using a standard *Block* data distribution. The only difference between this implementation and the Chapel-GPU code in Figure 5.1(b) is the target distribution in line 3.

Figure 5.2 shows performance results for the STREAM Triad benchmark written in Chapel running on both a 32-node instance of the Cray XT4 supercomputer and a GPU with a problem size of n = 85,983,914. As the last two bars show, performance for the Chapel and CUDA implementations are equivalent. It is important to emphasize that for the cluster and Chapel-GPU results, only the declared distribution was changed. In contrast, the CUDA code does not support the same degree of portability. In addition to single-node multi-cores and GPUs, this code has run on large-scale Cray configurations, achieving over 1.1TB/s of performance using 2048 nodes [104].

5.3 Generating Code for GPU Accelerators

This section describes the Chapel language and compiler extensions that were added to target GPU platforms. Sections 5.3.1 and 5.3.2 discuss the Chapel GPU distribution and

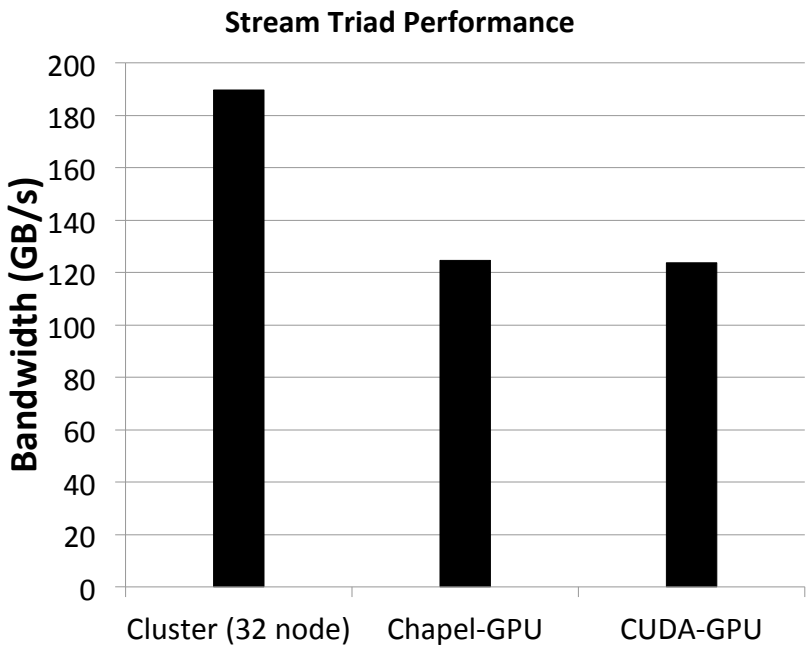


Figure 5.2: Results for the STREAM Triad Benchmark Comparing a 32-node Cray XT4 2.1 GHz Quad-Core AMD Opteron and NVIDIA GTX280 GPU

its associated support for domains and arrays. Sections 5.3.3 and 5.3.4 describe support for data movement between the device and host, and how code is executed on the GPU. Finally, Sections 5.3.5–5.3.8 will present code generation and other low-level facilities.

5.3.1 GPU User-Defined Distribution

In this work, several distributions have been defined. `GPUDist` indicates that data must reside in the memory of the GPU and that computations will also be performed on the device. As shown later in this chapter, there are other user-defined distributions that have been provided. In particular, they are used to support specialized GPU memory, as well as providing a different method of handling data movement.

In order to target the GPU, the user instantiates a `GPUDist` distribution class whose constructor takes the following arguments:

- `rank`: The dimensionality of the problem.

- `blockSizeX`, `blockSizeY`, `blockSizeZ [optional]`: The thread block size in the X, Y, and Z dimensions. These correlate to the equivalent thread block size dimensions used in CUDA (or the similar `work-group` concept in OpenCL). If the user does not explicitly initialize one of these values, they can be set using a heuristic approach based on the occupancy of the kernel and device. This is similar to the technique used in the Thrust library [105]. In the case of compiling for a multi-core, this value is also useful, as will be shown in Section 5.4.

5.3.2 GPU Domains and Distributed Arrays

A GPU domain and its arrays are declared identically to those with standard distributions. When an array is declared with a GPU domain, `cudaMalloc()` [106] is invoked to allocate its space, rather than the standard library `malloc()`.

In the following array declaration:

```
1 var gpuD = [1..n,1..n] dmapped GPUDist(rank=2);
2 var A: [gpuD] real;
```

Line 1 defines a GPU distribution and domain with a rank of 2, while line 2 declares the 2D array `A` that is allocated on the GPU. Section 5.3.6 describes how this technique applies to the other types of GPU memory.

5.3.3 Data Movement

Since the GPU and the host have different address spaces, most GPU programming models require users to manage the movement of data. This complicates programming and reduces portability. To address this problem, this work provides two methods of data movement in the Chapel code: *implicit* and *explicit*.

```
1 const space = [1..m] dmapped GPUDist(rank=1);
2 var input, output : [space] real;
3 input = ... // load input data
4 for 1..n {
5   forall j in space {
6     ...
7     output[j] = input[j];
8   }
9   ... = output;
10 }
```

Figure 5.3: Implicit Data Movement Example

Implicit Data Movement

In this approach, the programmer declares a single logical variable that can be accessed by the host and the device. The system automatically creates temporary storage and transfers the value(s) between the host and the GPU. The implicit data movement scheme is dependent on compiler analysis to determine when to move data. Section 5.5.1 discusses the compiler algorithm that generates the implicit data movement code.

An example of Chapel code which utilizes implicit data movement is shown in Figure 5.3. Since the array `input` is declared when the task is running on the CPU, but read from inside of the `forall` loop (when executing on the GPU), it is implicitly copied to the device before the `forall` is executed. Similarly, after the loop completes execution (on the GPU), the array `output` is copied out to the host implicitly because it is modified inside of the `forall` loop. To the programmer, the arrays declared on line 2 are treated the same regardless of whether they are inside or outside of a `forall` loop. In other words, the arrays and their elements can be accessed or manipulated as a non-GPU array throughout the program.

Explicit Data Movement

For complete control of data movement between the host and device, the user can explicitly transfer the data. This is done to deal with cases where the compiler must be conservative to determine when to copy data between the host and device. In this current implementation,

```
1 const space = [1..m] dmapped GPUExplicitDist(rank=1);
2 var h_input, h_output : [1..m] real;
3 var g_input, g_output : [space] real;
4 for 1..n {
5   h_input = ... // load input data
6   g_input = h_input;
7   forall j in space {
8     ...
9     g_output[j] = g_input[j];
10  }
11  h_output = g_output;
12  ... = h_output;
13 }
```

Figure 5.4: Explicit Data Movement Example

only synchronous data transfers are supported.

Consider the example in Figure 5.4. Line 1 declares a domain and a new type of GPU distribution named `GPUExplicitDist()`. This distribution takes the same parameters as `GPUDist()`. On line 2, the user declares the corresponding host variables. On line 3, the GPU-specific arrays are declared using the distribution and domain from line 1. The assignment operation on line 6 performs the explicit data copy from host space into GPU space. After the parallel computation is complete, the assignment operation on line 11 performs the explicit copy of the results back to the host.

5.3.4 Parallel Execution on the GPU

Chapel's `forall` loops that have been declared over a GPU domain enable parallel execution on GPUs. Each iteration of the loop corresponds to a light-weight GPU thread. Using the compiler-generated (or user-specified) block size, the compiler strip-mines the `forall` loop into block-sized units that correspond to thread blocks. As will be shown in Section 5.4, the compiler performs a similar transformation when targeting a multi-core. Figure 5.5 is based on the previous STREAM Triad example of Figure 5.1(b). Here, `space` represents a distributed domain from 1 through `M`. Because `blockSizeX` = 256 and `M` = 1024, there are $\lceil \frac{1024}{256} \rceil = 4$ thread blocks for execution on the GPU. This block size provides the necessary

```
const M=1024;
const Space=[1..M] dmapped GPUDist (rank=1,BlockSizeX=256);
forall i in Space { ... }
```

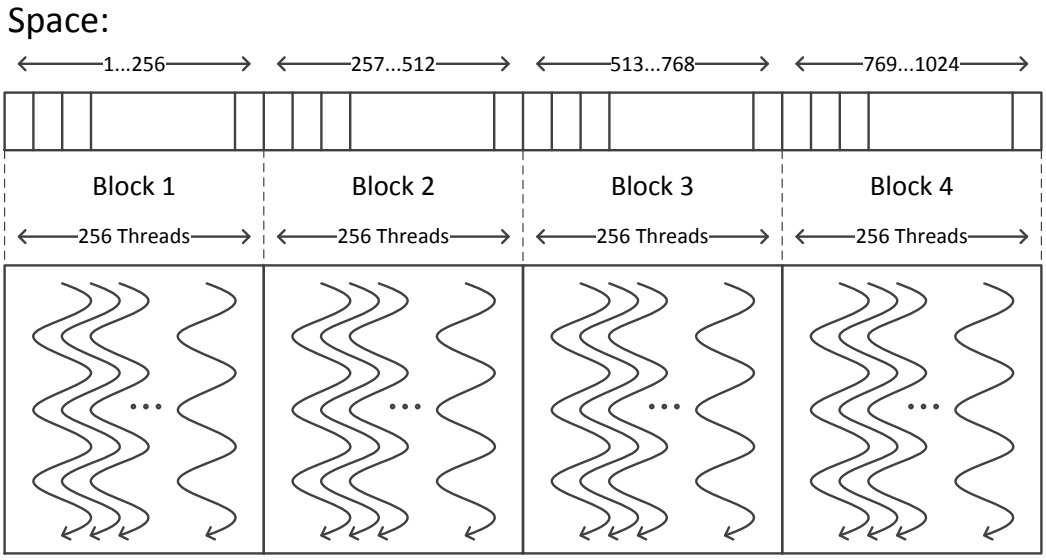


Figure 5.5: Mapping a Chapel 1D Domain Onto CUDA's Thread Blocks

information to map each iteration `i` of the `forall` loop onto a particular block and its associated thread.

5.3.5 Code Generation for the GPU

A high-level view of the compilation process is shown in Figure 5.6. The Chapel source-to-source compiler takes as input a Chapel source file. When the compiler lowers a `forall` loop that iterates over a GPU domain, it will generate both C and CUDA source for the host and GPU. Otherwise, it will just generate C. The body of the `forall` is code generated as a CUDA kernel. By having the compiler analyze the body of the loop, it is able to determine whether any variables that are used in the loop are declared before the loop begins. In that case, the compiler will automatically pass them in as arguments to the kernel function. The host portion performs the thread block creation, along with passing the correct parameters

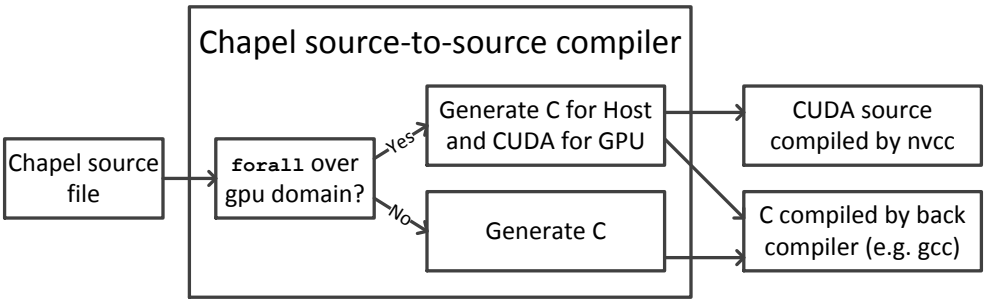


Figure 5.6: Overview of Chapel Compilation Process

into the CUDA kernel. As a final step, the generated GPU code (for both host and device) is compiled by NVIDIA's *nvcc* compiler, and the remaining code is compiled by the back-end compiler (e.g. *gcc*).

In addition to `forall` loops, Chapel supports the data-parallel primitives *reduce* and *scan*. When the compiler determines that a `reduce` or `scan` operator uses data elements that have been declared on the GPU, the compiler makes a call to a highly-tuned library implementation of that operation. This is a similar approach as others have taken [107]. An example using the `reduce` primitive will be shown later in Section 5.6.1.

5.3.6 Targeting Specialized GPU Memory Spaces

A common strategy to maximize performance for GPUs is to exploit the different physical memories in cases where locality exists [108]. On NVIDIA-based GPUs, a programmer has access to on-chip shared memory, read-only constant cache memory, and read-only texture memory. The trade-off that occurs in using any of these specialized memories in Chapel is that the user compromises portability for performance. However, this GPU-specialized code can be transformed into code that runs on traditional processors by applying simple compiler transformations, as discussed in Section 5.4 and in other publications [9, 8, 109]. To use any of the specialized memories within the GPU, users need to declare their arrays using the following distributions:

Shared Memory

NVIDIA's GPUs offer an on-chip scratchpad memory that is user-programmed to optimize for locality. Shared memory is faster than global memory and, unlike texture and constant cache memory, is writable from the kernel. The data stored into shared memory is only visible by threads within the same thread block where the writing takes place. A compiler error occurs if the user attempts to write or read into a shared memory array when not executing a `forall`. In order to leverage this memory from Chapel, the user declares the distribution `GPUSharedDist()`.

Constant Cache Memory

Constant memory is used to hold constant values. This data is hardware-cached to optimize for temporal locality. There is only one cycle of latency when a cache hit occurs if all the threads in a warp access the same location. Otherwise, accesses to constant memory are serialized if threads read from different locations. With GPUs such as the NVIDIA GTX280, up to 64KB of data may be placed into constant memory. In Chapel, the data to be stored in constant memory must be declared with the `GPUCCDist()` distribution.

Figure 5.7 provides an example where the user would leverage the GPU constant memory within Chapel. On lines 1–4, the program creates a constant memory distribution, domain, and the respective constant memory array. Line 5 shows the array being loaded with data from the host. Finally, in line 9, the constant memory array is accessed, as with any typical array.

Texture Cache Memory

Similar to constant memory, texture memory is read-only and uses hardware caching for locality. Performance increases are seen when applications have spatial locality, as in stencil computations or even irregular computations [110]. The Chapel compiler

```
1 const myspace = [1..m] dmapped GPUDist(rank=1);
2 const ccspace = [1..m] dmapped GPUCCDist(rank=1);
3 var input: [ccspace] real;
4 var output: [myspace] real;
5 input = ... // load data into constant memory
6 for 1..n {
7   forall j in myspace {
8     ...
9     output[j] = input[j];
10  }
11  ... = output;
12 }
```

Figure 5.7: Constant Cache Example

does not yet support storage in this memory.

5.3.7 Synchronization

CUDA uses `__syncthreads()` as a barrier between threads in a thread block. Chapel takes the approach that the programmer needs to also provide a similar synchronization primitive `thread_barrier()` to ensure correctness in GPU code. While using a `thread_barrier()` is not needed to program traditional CPUs, Algorithm 7, as described in Section 5.4, provides a compiler technique used to remove calls to `thread_barrier()` when targeting CPUs.

5.3.8 GPU Low-Level Extensions

There are cases in which a user must leverage certain facilities offered only by the CUDA programming model. For example, CUDA provides fast math intrinsics that are implemented in hardware, such as `__fsinf()`, instead of the more accurate (but slower) `sin()`. In order to interoperate with these routines through Chapel, the compiler translates invocations to these routines into their software equivalent when targeting CPUs.

5.4 Generating Code for Multi-core

To address the portability argument, this section presents transformations necessary to take a single program text written with the GPU abstractions from the previous section and compile and execute it efficiently on a traditional multi-core platform. This is achieved by taking advantage of coarse- and fine-grain parallelism that is exposed when targeting GPUs (i.e. CUDA thread blocks and threads within a thread block). Due to false-sharing issues and programming for locality, multi-cores are ideally suited for coarse-grain parallelism. To address this, adjacent thread blocks are assigned as work units to the processors. Therefore, this transforms all `forall` loops that iterate over GPU domains to doubly-nested loops: the outer loop is a standard Chapel `forall` that iterates over blocks of threads, and the inner loop is a sequential `for` that iterates over threads within a block.

In order to preserve correctness, shared memory arrays are simply declared per-block as a non-distributed array. In other words, a non-distributed array is declared between the outer standard `forall` and the inner `for` loop.

This approach has two main advantages. First, Chapel handles the outer `forall` loop by distributing workloads evenly in a block-wise manner such that adjacent blocks are assigned to a core. Second, by serializing the inner loop, all calls to `thread_barrier()` can be removed.

The disadvantage of the above approach is that serializing the inner loop that iterates over threads inside of a block is not always trivial. Algorithm 7 describes how to create an outer parallel loop L_1 and an inner sequential loop L_2. Algorithm 8 describes the technique used to serialize L_2. In the scenario where there are no calls to `thread_barrier()`, nothing is modified. If there is a `thread_barrier()` *barr*, which is not contained in any inner loop within L_2, it is important to distribute the loop over the code before and after *barr* in order to remove the barrier. If there is a variable v declared in the original L_2 and it is accessed

before and after *barr*, the declaration of v needs to move before the distributed loops. If there is an assignment to v that is dependent on `threadId`, that is accessed before and after *barr*, array expansion is applied to v by the thread-block size. This is because each thread needs its own private copy of v. Array expansion is applied at most once for each v. On the other hand, if *barr* is within an inner loop L', L_2 needs to be distributed around three sections of code: the code before L', L' itself, and the code after L'. Next, L' and the distributed L_2 can be interchanged since all the threads in a block must reach *barr* an equal number of times. This interchange is valid since the outer loop can run in parallel. Finally, Algorithm 8 is called recursively to handle a deeper loop nest.

As an example demonstrating Algorithm 7, consider the simplified kernel of RPES from the Parboil Benchmark Suite shown in Figure 5.8. The code in Figure 5.8(a) shows the GPU-centric Chapel code, while Figure 5.8(b) shows the transformed Chapel code after performing the algorithm. First, the original `forall` loop is converted into another `forall` loop that iterates over thread blocks and a sequential `for` loop that iterates over threads within a block. Both the number of thread blocks and the number of threads within a thread block are computed based on the `blockSize` value specified in Section 5.3.1. Second, the shared memory array `Data` has to be declared just inside the new `forall` loop. Next, loop distribution of the sequential `for` loop is performed since there are calls to `thread_barrier()`. Finally, the two sequential loops are interchanged, and the inner call to `thread_barrier()` is removed.

5.5 Compiler Transformations and Optimizations

This section first presents a simple algorithm used to perform implicit data transfers between the host and device. Afterwards, this section will present algorithms to optimize the generated GPU code.

```
1 var Data = [0..BLOCK_SIZE-1] dmapped GPUSharedDist();
2 forall gpuSpace_reduc {
3   const blid = getBlockID_x();
4   const thid = getThreadID_x();
5   Data[thid] = getData(blid, thid, ...);
6   thread_barrier();
7   for s in 0..5 do {
8     const i = 1 << (s * -1 + LOG_BLOCK_SIZE - 1);
9     if thid < i then
10      Data[thid] += Data[thid + i];
11    thread_barrier();
12  }
13  if thid == 0 then
14    ReductionSum[Offset + blid] = Data[0];
15 }
```

(a) Original code in Chapel with GPU `forall`

```
1 forall blid in 0..num_blocks-1 {
2   var Data = [0..BLOCK_SIZE-1];
3   for thid in 0..BLOCK_SIZE-1 {
4     Data[thid] = getData(blid, thid, ...);
5   }
6   for s in 0..5 {
7     for thid in 0..BLOCK_SIZE-1 {
8       const i = 1 << (s * -1 + LOG_BLOCK_SIZE - 1);
9       if thid < i then
10        Data[thid] += Data[thid + i];
11    }
12  }
13  for thid in 0..BLOCK_SIZE-1 {
14    if thid == 0 then
15      ReductionSum[Offset + blid] = Data[0];
16  }
17 }
```

(b) Translated code into Chapel multi-core `forall`

Figure 5.8: Translation of a GPU `forall` into Multi-core `forall`

Algorithm 7: Loop Transformation for Multi-core

Input: List $forallList$ containing every `forall` loop with a GPU Distribution
foreach $L \in forallList$ **do**
    $L_1 \leftarrow$ Create standard Chapel `forall` loop that iterates over thread-blocks with loop index `blockId`;
    $L_2 \leftarrow$ Create sequential `for` loop that iterates over threads of a block with loop index `threadId`;
    foreach *shared array* $s \in L$ **do**
        Declare s as a standard array in body of L_1;
    The body of L becomes the body of L_2;
    $OutList \leftarrow$ Call `LoopDist(`$L2$`)`;
    Add $OutList$ to the body of L_1;

5.5.1 Implicit Data Transfers Between Host and Device

As mentioned in Section 5.3.3, implicit transfers between the host and GPU require compiler support. Algorithm 9 gives a conservative approach for determining and generating the code necessary to transfer the data. If an array that has been declared with a GPU distribution, is accessed within a `forall` loop, the compiler will compute the read and write sets of the array. If the read set is not empty, the compiler will generate code to copy the data into the device. Similarly, if the write set is not empty, the data is copied to the host after the loop completes.

In the previous example from Figure 5.3, the user never explicitly copies data between the device and host before calling the `forall` loop. To the user, the variable appears normal without the knowledge that it can only be used on the GPU. Based on the algorithm, the compiler will always copy data from the host to the device since `input` is read inside the `forall` loop. Also, the array `output` is written to, causing the compiler to copy that data out to the host. As the example shows, since the `forall` loop is nested inside of a `for` loop, the array `input` is copied into the kernel redundantly. An improvement over the conservative approach taken here would be to analyze the complete program outside of the kernel to detect redundant copying.

Algorithm 8: `LoopDist()` Function

Input: Sequential loop L
Output: List of Variable Declarations and Loops
if $\exists$ *barr* $\in L$ *where barr is a* `thread_barrier()` **then**
  List $VarDecl \leftarrow \emptyset$;
  foreach *variable declaration* $v \in L$ **do**
    if $\exists$ *an assignment to* v *that is dependent on* `threadId` **and** v *is accessed before and after barr* **and** v *has not been expanded before* **then**
      $v_{exp} \leftarrow$expansion of v by `blockSize`;
      $VarDecl \leftarrow VarDecl \cup \{v_{exp}\}$;
      Replace all occurrences of v with v_{exp}[`threadId`];
    else
      $VarDecl \leftarrow VarDecl \cup \{v\}$;
  if $\exists$ *a loop* L' *in the body of* L **and** *barr* $\in L'$ **then**
    $DL_1 \leftarrow$ Loop with code before L' surrounded by header of L;
    $DL_2 \leftarrow$ Loop with same header as L with inner loop L' as the body;
    $DL_2 \leftarrow$ Loop interchange L with L' in DL_2;
    $DL_3 \leftarrow$ Loop with code after L' surrounded by header of L;
    return $< VarDecl,$ `LoopDist`$(DL_1),$ `LoopDist`$(DL_2),$ `LoopDist`$(DL_3) >$;
  else
    $DL_1 \leftarrow$ Distribute L over code before *barr*;
    $DL_2 \leftarrow$ Distribute L over code after *barr*;
    return $< VarDecl,$ `LoopDist`$(DL_1),$ `LoopDist`$(DL_2) >$;
else
  return $< L >$;

5.5.2 Scalar Replacement of Aggregates and Dead Argument Elimination

Compiling from a higher-level language like Chapel down to CUDA opens doors to possible optimizations. Chapel has support for multidimensional arrays with arbitrary index ranges. For this purpose, the Chapel compiler creates structures containing meta-data about the array, including start and end points, array strides, and a pointer to the raw data. The program has additional levels of indirection that it uses to look up the member variables

Algorithm 9: Implicit Data Transfer

Input: Array G declared with the GPU Distribution
Input: List $forallList$ containing every `forall` loop with a GPU Distribution
foreach $L \in forallList$ **do**
    if $USE(G) \neq \emptyset$ **then**
        Generate statement to copy G from host to device before L begins;
    if $DEF(G) \neq \emptyset$ **then**
        Generate statement to copy G from device to host after L completes;

of the structure, and therefore requires more memory operations than the typical array access in C. To avoid this increase, scalar replacement of aggregates [111] is performed. This technique flattens fields from a structure with single scalar elements. In particular, this is applied on all structures that are used within a `forall` loop that executes over an array or domain declared with a GPU distribution. The scalarized fields are then placed onto the formal argument list of the calling kernel routine. After this transformation is complete, dead argument elimination is performed on the original structures that were passed in as formals, as they are no longer necessary.

5.5.3 Kernel Argument Spilling to Constant Memory

As a result of the previous optimization performed in Section 5.5.2, the number of formal parameters to the GPU kernel will likely have increased depending on the number of fields in the original structures. Because shared memory resources are reserved for arguments up to a maximum size of 256 bytes [106], there will be a greater performance impact as more arguments are passed. Additionally, if this size limit is exceeded, a back-end compiler error will be thrown. To get around this, Algorithm 10 describes a mechanism based on dataflow analysis that will spill scalar arguments into constant memory after a certain argument list size has been reached. In this algorithm, constant memory variables are generated with the `__constant__` modifier and are assigned from the host using the CUDA routine

Algorithm 10: Spill Scalar Args Into Constant Mem

Input: List *argList* containing each formal argument of the kernel function
Input: Spill threshold *threshold*
$totalSize \leftarrow 0$;
foreach $arg_i \in argList$ **do**
    if `sizeof`(arg_i)+$totalSize > threshold$ **and** $DEF(arg_i) = \emptyset$ **then**
        Declare constant memory variable new_i outside of kernel;
        $new_i \leftarrow arg_i$;
        Remove arg_i from $argList$;
        foreach $u_i \in USE(i)$ **do**
            $u_i \leftarrow new_i$;
    else
        $totalSize$`+=sizeof`(arg_i);

`cudaMemcpyToSymbol()`. In the unlikely event the algorithm is not able to spill enough of *argList* into constant memory, the remainder of the arguments will be spilled into GPU global memory. The default threshold value for when to spill is set through a compiler flag (*--max-gpu-args=#*). It should be noted that, while this algorithm does not increase performance, it is necessary for correctness because of the limit on the number of arguments supported by the CUDA compiler.

5.6 Example Codes

The goal of this section is to discuss two examples and illustrate the portability of these codes across different parallel architectures. The examples are a 2D Jacobi method and a code to compute Coulombic Potential [112]. Afterwards, performance results for execution on a multi-core, followed by results on a GPU are presented. In the GPU case, there was an evaluation of the two techniques of transferring data between the host and device discussed in Section 5.3. The hardware used for these experiments is the same as described in Section 5.7.2.

```
1  const gdist = new GPUDist(rank=2,
2                            blockSizeX=16,
3                            blockSizeY=16);
4  const gPSpace = [1..n, 1..n] dmapped gdist;
5  const gDomain = [0..n+1, 0..n+1] dmapped gdist;
6  var X, XNew : [gDomain] real;
7  var tempDiff : [gPSpace] real;

9  /* initialize data */
10 ....
11 do {
12   forall ij in gPSpace {
13     XNew[ij] = (X[ij+north] + X[ij+south] +
14                 X[ij+east] + X[ij+west]) / 4.0;
15     tempDiff[ij] = fabs(XNew[ij] - X[ij]);
16   }
17   delta = max reduce tempDiff;
18   X[gPSpace] = XNew[gPSpace];
19 } while (delta > epsilon);
```

Figure 5.9: Chapel Implementation of Jacobi 2D

5.6.1 2D Jacobi

Figure 5.9 illustrates the 2D Jacobi method that targets a GPU. This algorithm computes the solution of a Laplace equation over a 2D grid. The point of this code is to show an elegant high-level implementation of the algorithm rather than present the reader with a low-level and highly-tuned implementation. Line 1 of the algorithm declares a GPU distribution with a rank of 2. Lines 4–5 declare two distributed domains, and lines 6–7 declare the associated arrays. Lines 12–16 are a parallel stencil computation for the GPU. Line 17 represents a maximum reduction. Finally, line 18 performs a sliced array copy of the inner domain `gPSpace`.

Figure 5.10 shows the performance of this code. First, performance on a multi-core using 4 tasks is shown. Then, the performance on a GPU is shown using both the implicit and explicit data transfer algorithm. It's important to note that, in this example, no lines of code were changed to port the code between the GPUs and multi-cores. Also, the Chapel multi-core implementation is based on the implicit GPU version. However, either version (implicit and explicit) can be run. The Chapel GPU version of the code that uses the implicit data

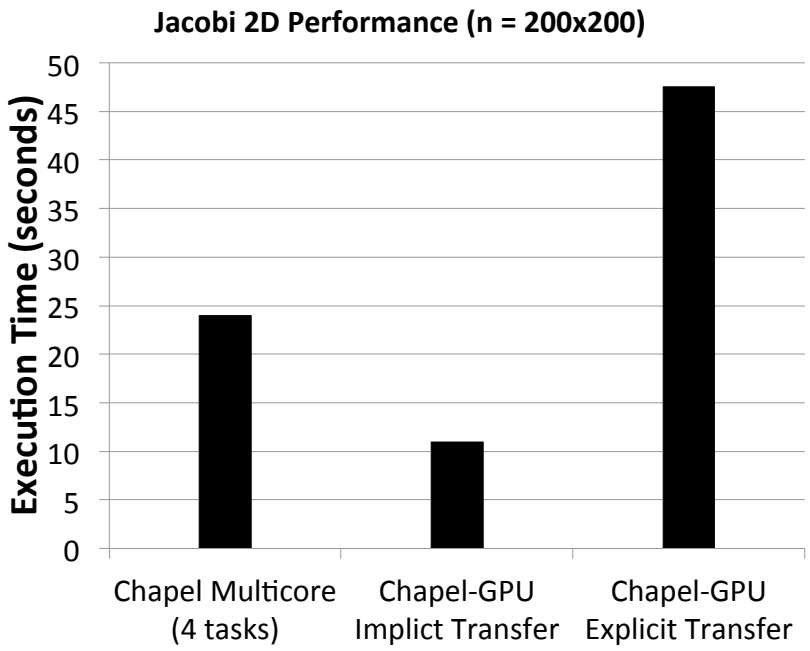


Figure 5.10: Performance of Jacobi 2D

transfer algorithm shows degradation in performance due to the redundant data transfers that occur. This is the result of the conservative approach taken in the compiler algorithm.

5.6.2 Coulombic Potential

The code for the Coulombic Potential (CP) application is shown in Figure 5.11. On lines 3–5, two different GPU distributions are declared. On lines 6–8, distributed domains are declared with the previously declared distributions. Lines 9 and 11 declare the input and output arrays. Lines 13–23 perform the parallel `forall` computation on the GPU. It should be mentioned that, on line 13, each iteration of the `forall` returns a two-tuple containing x and y coordinates.

Figures 5.12a and 5.13a present the results for CP running on both a GPU and a multicore. As in the previous example, no changes to the source code were made. These results will be discussed in more detail in Section 5.7.3.

```
const volmemsz = VOLSIZEX * VOLSIZEY;
const volmemsz_dom = [1..VOLSIZEY,1..VOLSIZEX];
const dst = new GPUDist(rank=1);
const totdst = new GPUDist(rank=2,
        blockSizeX=BLOCKSIZEX, blockSizeY=BLOCKSIZEY);
const totspace = volmemsz_dom dmapped totdst;
const energyspace = [1..volmemsz] dmapped dst;
const atomspace = [1..MAXATOMS] dmapped dst;
var energygrid : [energyspace] real = 0.0;
/* initialize atominfo from input file */
var atominfo : [atomspace] float4 = ...;

forall (xindex,yindex) in totspace {
  var energyval = 0.0;
  var (coorx,coory) = (gspacing*xindex,
                       gspacing*yindex);
  for atom in atominfo {
    var (dx,dy) = (coorx-atom.x, coory-atom.y);
    var r_1 = 1.0 / sqrt(dx*dx + dy*dy + atom.z);
    energyval += atom.w * r_1;
  }
  energygrid[yindex,xindex] += energyval;
}
```

Figure 5.11: Coulombic Potential in Chapel

5.7 Evaluation

This section presents experimental results. Section 5.7.1 will describe the benchmarks used for the experiments. Section 5.7.2 describes the experimental setup used. Section 5.7.3 evaluates the effectiveness of the compiler. For that, three types of evaluations are performed. First, an evaluation of the GPU performance of the Chapel code is compared with codes from the hand-coded Parboil benchmark suite. Second, an evaluation is made of the performance from the same benchmarks on a traditional multi-core platform. To compile and execute the Parboil CUDA code on a multi-core, both the PGI CUDA-X86 [109] and Ocelot compilers [8] are used. The third evaluation estimates the *productivity* benefits of using Chapel. This is based on the difference in code size between the Chapel and CUDA source as the metric of *productivity*. This is a simple and not always compelling metric, but in this case, it is believed that it gives a reasonable indication of the *productivity* advantage of using Chapel.

5.7.1 Parboil Benchmarks

The Parboil benchmark codes used are Coulombic Potential (CP), MRI-FHD and MRI-Q [72], Rys Polynomial Equation Solver (RPES) [113], and the Two Point Angular Correlation Function (TPACF) [114]. The benchmark Sum of Absolute Differences (SAD) is not studied because it relies on texture memory, which the implementation does not currently support.

In the case of the GPU evaluation, the Chapel codes that are compared use both implicit and explicit data transfers to see what additional overhead results from the conservative implicit data transfer algorithm introduced in Section 5.5.1.

5.7.2 Environmental Setup

For the GPU evaluations, each benchmark was run using an NVIDIA GTX280. The host code was executed on an Intel Quad-core 2.83GHz Q9550. For timing measurements, the evaluation uses CUDA's kernel profiling mechanism (i.e. *CUDA_PROFILE=1*) that measures the execution time spent in the kernel along with execution time spent on data transfers between the host and the device.

The multi-core evaluation uses an Intel Quad-core 2.67GHz Nehalem i7 920, where each core supports two hyperthreads. The generated C code from Chapel was then compiled by the back-end PGI 11.8 C/C++ compiler. The CUDA codes were also compiled using the PGI CUDA-X86 compiler that generates OpenMP with the flags `-Mcudax86 -fast`. Similarly, the CUDA codes were compiled by the Ocelot compiler with optimization flag `optimizationLevel:full` being set.

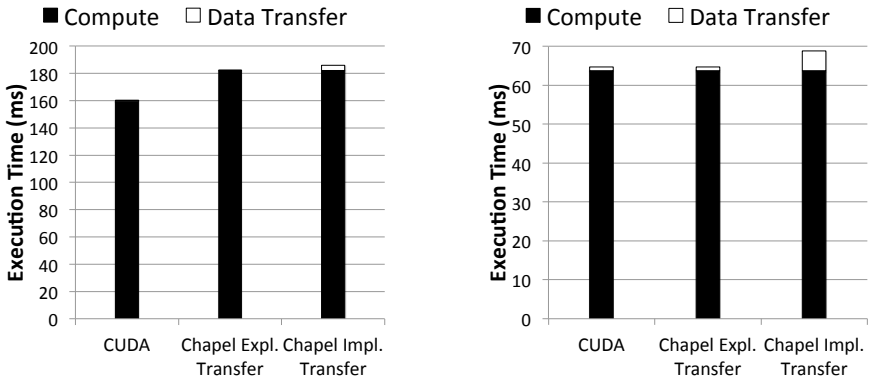
5.7.3 Experimental Results

Figures 5.12a–5.12e and 5.13a–5.13e demonstrate the performance, in execution time, of the Parboil benchmarks running on a GPU and multi-core platform, respectively. Due to the difference in magnitude of times (milliseconds vs seconds) between GPU and multi-core executions, they were plotted separately and to different scales.

GPU Evaluations

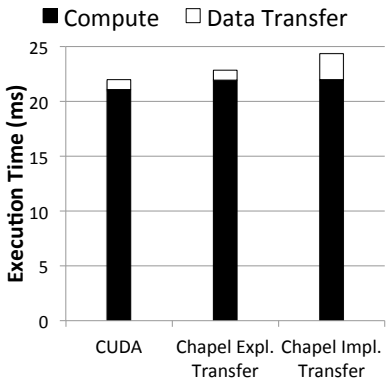
In Figures 5.12a–5.12e, each bar is broken down into two portions: the total time spent performing data transfers vs. time performing the computation. In Figures 5.12b–5.12e, the difference in compute performance was minimal between the CUDA and the Chapel implementations. When attention is solely on the compute performance in Figures 5.12a and 5.12d, the CUDA reference implementations have slightly better performance when compared with the Chapel implementations. In these cases, the performance difference was due to additional overhead, such as `for` loops being generated inefficiently. As the Chapel compiler matures, it is expected that these minor differences in compute performance will decrease.

The additional overhead due to the conservative implicit data transfer algorithm is apparent in Figures 5.12b, 5.12c, and 5.12e. These three benchmarks demonstrate the deficiencies in the conservative approach taken by the compiler for selecting which data to transfer into and out of the kernel. In the RPES algorithm, similar to the previous example in Figure 5.3, there is a parallel `forall` loop nested inside of a `for` loop. In the CUDA and Chapel explicit data transfer implementations, data is not transferred within the top-level `for` loop iterations; but, in the case of the implicit data transfer algorithm, the data is copied redundantly. The CP and TPACF algorithms in Figures 5.12a and 5.12d have an insignificant amount of overhead associated with the implicit data transfer scheme.

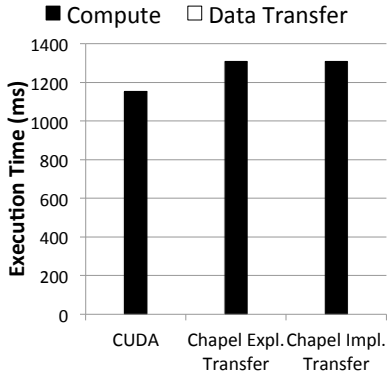


(a) Coulombic Potential

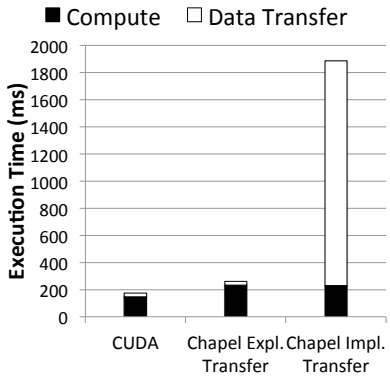
(b) MRI-FHD



(c) MRI-Q



(d) TPACF



(e) RPES

Figure 5.12: GPU Performance of the Parboil Benchmarks Comparing Chapel to CUDA

Benchmark	# Lines (CUDA)	# Lines (Chapel)	% Shorter	# of Kernels
CP	186	154	17.2%	1
MRI-FHD	285	145	49.1%	2
MRI-Q	250	125	50.0%	2
RPES	633	504	20.4%	2
TPACF	329	209	36.5%	1

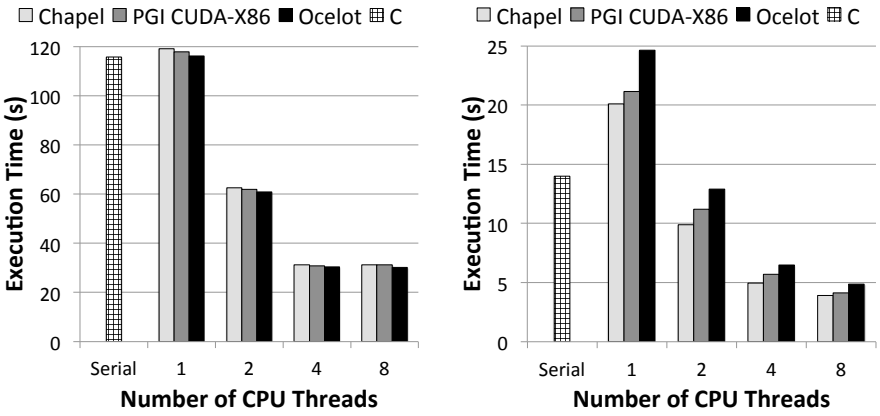
Table 5.1: Parboil Source Code Comparison (Chapel vs CUDA)

Multi-core Evaluation

Figures 5.13a–5.13e present the original Parboil benchmarks running on a multi-core platform. This work is comparing a serial C implementation of the benchmark with that of the Chapel, PGI CUDA-X86, and Ocelot compilers. In the case of the Coulombic Potential (CP) benchmark in Figure 5.13a, it can be observed that the performance is similar for all three compilers. In the remaining Figures 5.13b–5.13e, there is a noticeable performance difference between the three compilers. In particular, for Figure 5.13e, it was observed that the PGI compiler was deadlocking on the RPES benchmark when run with 8 threads. RPES and TPACF are the only benchmarks tested that use GPU shared memory and thus rely on thread barrier synchronization. One likely possibility for the difference in performance (and possibly correctness) between the compilers is that PGI and Ocelot might not be fully removing all possible barriers when targeting this code on a multi-core. This would also explain why deadlocks occur on RPES when running with 8 threads.

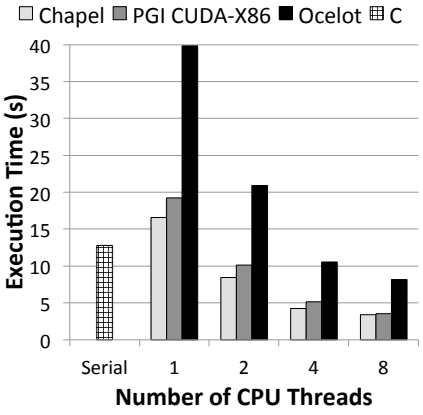
Productivity Evaluation

Table 5.1 shows a comparison between the Chapel and CUDA implementations with the primary metric being the difference in lines of source code. In order to compute the total number of lines of code, all comments, timing mechanisms, and blank lines were removed. For all the benchmarks, the Chapel code was shorter by 17–50%. Not only is the code shorter, but as the examples in this dissertation show, it is cleaner and more elegant.

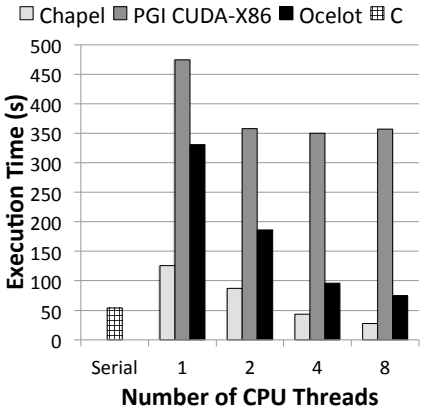


(a) Coulombic Potential

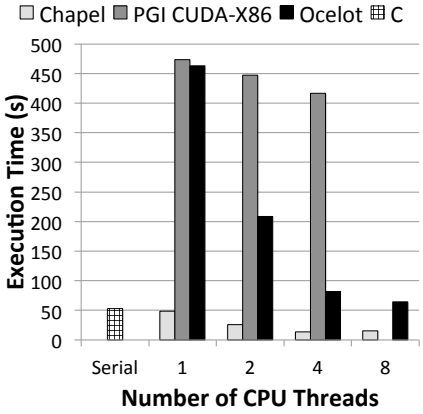
(b) MRI-FHD



(c) MRI-Q



(d) TPACF



(e) RPES

Figure 5.13: Multi-core Performance of the Parboil Benchmarks

In summary, these results demonstrate that when using a language such as Chapel, it is possible to achieve performance that is comparable to that of the GPU-specific CUDA, while making the code portable to execute efficiently on a traditional CPU platform. In addition, there are productivity and elegance gains in using Chapel over CUDA due to the lower amount of necessary code.

5.8 Limitations

There are certain limitations that are present in the current implementation. Currently, there is no automatic (though the compiler) means of exploiting non-DRAM GPU memory. In the implementation presented, it is up to the programmer to specify which form of memory they want to leverage in their algorithm. Related to this, the programmer also needs to explicitly place the barriers into their loops.

As shown in Section 5.7.3, the algorithm used for implicitly transferring data between the devices is too conservative, leaving room for improvement. One method for exploration would be to perform an interprocedural compiler analysis that looks across multiple `forall` loops and, based on the usage of the data, generates the necessary transfer code when required. The second alternative would be to incorporate the work done in the GMAC (Global Memory for Accelerators) project [115]. GMAC is a library-based system that provides coherency between data on the device and on the CPU.

5.9 Related Work

Improving the programmability of accelerator architectures is currently an active area of research. The works of CUDA-X86 [116], MCUDA [9], and Ocelot [8] take the approach of having the programmer implement their algorithms in CUDA before having the compiler

target a multi-core platform. The approach that Chapel takes is different since it starts with a higher-level language that can be used to target GPUs, multi-cores, and clusters. There has also been some work providing language bindings to target the GPU [117, 118], but in these methods, the actual kernel is still expressed in CUDA or OpenCL.

Another approach that some have taken in translating their application code into the GPU accelerator space uses annotations or compiler directives on existing languages [102, 119, 109, 120, 107, 121]. Chapel differs here in that it does not depend on annotations to induce the parallelism over a GPU, resulting in code believed to be more readable to the user. Additionally, the annotation-based approaches do not have as strong of support for high-level loop abstractions. This means a user of an annotation-based language needs to go in and decorate their loops when they want the code to be parallel. In contrast, Chapel's domains and iterators permit such things to be factored away in a more structured manner.

The work of OmpSs [122] addresses the issue of programming portability for heterogeneous multi-core architectures. The approach presented here differs in that Chapel is a single unified and high-level language that is used to express computations that will execute on both GPUs and multi-cores. In the OmpSs approach, they extend OpenMP and have the user provide implementation-specific kernels that will be mapped onto the respective devices. These approaches differ since the compute kernels are typically written as embedded CUDA or OpenCL.

X10 [123] and Habanero [124] both have shown support for GPUs, but they use different techniques to handle single-source portability across other architectures. They do not provide any mechanism for reverting to a multi-core platform from a tuned GPU implementation. Also, there does not seem to be support for implicit data copies; programmers themselves have to explicitly perform the copies. Lastly, not enough data could be found to show how these approaches apply to larger scaled GPU applications.

5.10 Conclusion

This chapter presented new methods to increase programmer productivity by leveraging an emerging programming language built for parallelism and locality control. By utilizing Chapel's support for user-defined distributions, programmers are offered a concise and elegant approach to targeting GPU-based architectures. Additionally, this work shows that it is possible to be portable across multi-core architectures and yet retain performance without resorting to different parallel libraries or language annotations such as pragmas or directives.

Chapter 6

Multi-core Micro-benchmark Suite

6.1 Introduction

With multi-core processors as the current dominant trend, and architectures becoming more complex, finding hardware specifications is becoming increasingly difficult. Knowledge of hardware features can be useful in driving program optimization, such as in library generators. ATLAS [125], SPIRAL [91], and FFTW [126] are examples of such library generators. ATLAS generates linear algebra routines (BLAS) with a focus on matrix-matrix multiplication. SPIRAL and FFTW are similar to ATLAS, but generate signal processing libraries. Analytical models have been [90], and are being [127] developed for library generators that use hardware characteristics to reduce the search time. For example, ATLAS will use knowledge of the L2 cache size in order to determine optimal tile sizes for matrix-matrix multiplication.

Programs such as X-Ray [128], hwloc [129], and other benchmark suites [130, 131] try to address the problem of automatically finding machine characteristics, but focus on features of uniprocessor super-scalars. In this chapter, a new benchmark suite named P-Ray is presented. The goal of P-Ray is to take the existing set of hardware characterizing benchmarks

from X-Ray, and extend them to multi-cores in order to find machine characteristics such as the number of caches shared by the cores, the processors' interconnection topologies, the effective bandwidth, and the block size used by the cache coherence mechanism. Experimental results will show that for three different platforms, P-Ray generates accurate results.

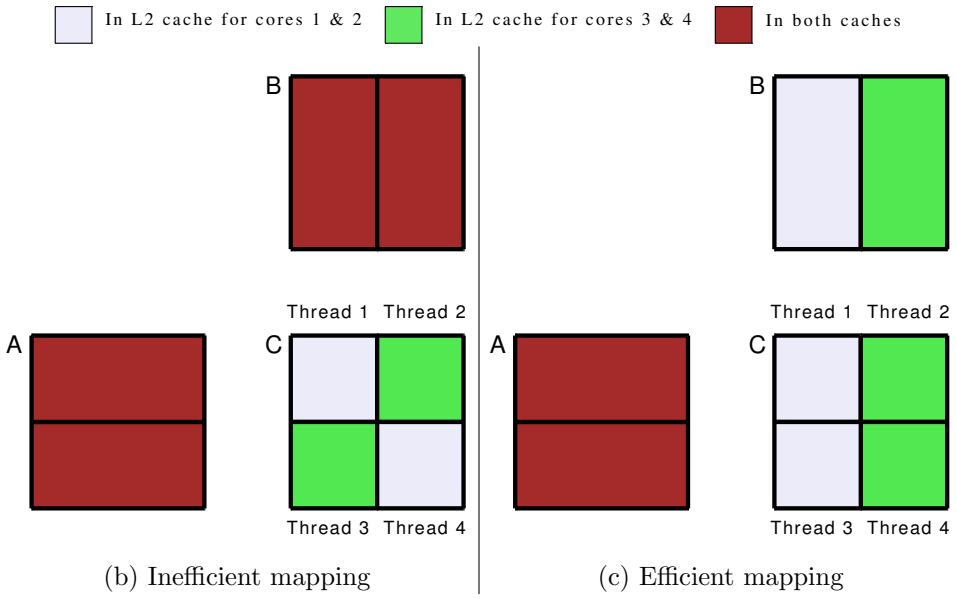
The use of P-Ray in discovering machine characteristics can aid in the computation of the machine cost model as used in Chapter 4.3. For example, machine characteristics such as cache block size can be utilized to determine the memory footprint.

The remainder of this chapter is organized as follows: Section 6.2 provides motivating examples for this work. Section 6.3 presents the different hardware characteristics studied. Section 6.4 describes the implementation requirements and details. Section 6.5 summarizes the experimental environment and discusses results. Section 6.6 describes related work. Finally Section 6.7 summarizes the work and offers concluding remarks.

6.2 Motivation

Multi-threaded matrix-matrix multiplication

Library generators need detailed knowledge of the architectural features of the machine to generate high-performance code. To show that this is the case, an implementation of matrix-matrix multiplication ($C = A * B$) is executed. This implementation uses POSIX threads on an Intel Core 2 Quad desktop that has four cores and two L2 caches, each cache shared by two cores. Figure 6.1 shows two different possible mappings for the matrices depending on thread affinity. For this experiment, matrix C is split into four sub-matrices, and each thread is assigned to one quadrant. Matrices are of size 800×800 each, so that they fit in memory but not in the L2 cache. With the mapping in Figure 6.1b, both matrices A and B need to be loaded in both L2 caches. Using the mapping of Figure 6.1c, matrix B can be split so that one half goes to one L2 cache and the other half goes to the second. The



(b) Inefficient mapping (c) Efficient mapping

Figure 6.1: Data Locality Depending on Thread to Core Affinity

experimental results show that an inefficient mapping can run up to 32% slower than an efficient one. To correctly map the threads to cores as in Figure 6.1c, it was necessary to use P-Ray to obtain the ID of the cores that share the L2 cache. Thread affinity has been used in the past to pin a thread to a core in order to prevent the OS from migrating it to a different core (and subsequent cache trashing) after a context switch [132]. However, in current architectures where several cores could share a cache, thread affinity can be used to place the data shared by two threads into L2. In most cases, this use of thread affinity can only be done if the programmer has the information provided by a tool such as P-Ray, by exhaustive search of all the possibilities, or if additional operating support is provided.

6.3 Targeted Characteristics

This section presents each part of the hardware that will characterized, and then afterwards, the algorithms used to discover these characterizations will be provided. Implementation specifics and detailed interpretation of the produced results will be discussed in Sections 6.4

and 6.5.

6.3.1 Cache Coherence Protocol Block Size

Knowing the block size used by the coherence protocol can aid the programmer in reducing false sharing misses. Other solutions already exist to measure a cache line size, but are slower than the one proposed in this thesis. By exploiting false sharing, the solution presented here infers the block size in a fast and simple way.

Algorithm 11: Calculate Block Size

measure-size($core_1$,$core_2$) {
char $data[MAXLSIZES]$
$i \leftarrow 1$
while $i \leq MAXLSIZE$ **do**
  Start timing
  Spawn thread-work($core_1$,0)
  Spawn thread-work($core_2$,i)
  Wait for threads to complete
  Stop timing
  Print i, *timing*
  $i \leftarrow 2 * i$
end while
}

thread-work(*core*,*index*) {
Set-thread-affinity(*core*)
$i \leftarrow 0$
while $i \leq SAMPLES$ **do**
  $data[index] \leftarrow data[index] + 1$
  $i \leftarrow i + 1$
end while
}

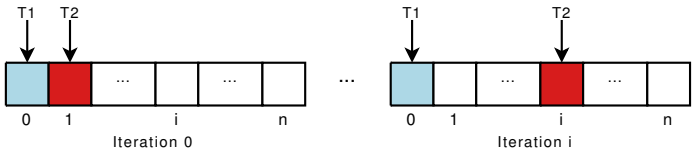


Figure 6.2: Coherence Block Size Benchmark

Figure 6.2 illustrates Algorithm 11, that is used to compute the block size. Two threads are spawned to work on a shared array consisting of C `char` variables. Both threads modify the shared data in order to induce coherence traffic. The data is also read to ensure it resides in L1: some architectures implement write-through write-no-allocate caches[1].

[1]Sun Niagara T1: `http://opensparc-t1.sunsource.net/specs/OpenSPARCT1_Micro_Arch.pdf`

Thread one will always access the first element of the array. Thread two starts accessing the second element of the array; however, with each iteration, it accesses an element which is further from the first one.

At first, both threads will access the same cache line and have poor performance due to false sharing; as the spacing between accesses increases, the performance stays poor until both accessed values are on two separate cache lines. At this point, execution time decreases drastically and the coherence block size can then be automatically inferred.

This algorithm can also provide information on the block size of different levels of cache. When the threads are mapped to cores that share an L2, this algorithm measures the block size of L1. However, when threads are mapped to cores that do not share a L2, this algorithm measures the block size of L2. When there is not enough information about the mapping of a core to the caches, the second thread can be mapped to different cores in the system. By comparing the execution times of the different mappings, P-Ray can determine whether the block size corresponds to the L1 or L2 cache. When there is no coherency between the caches, this mechanism cannot determine the block size.

6.3.2 Cache Mapping

With this program, discovery of the number of caches at a given level of the system and the cores that share them, is now possible.

For this to work, knowledge of the cache size for the given level is required. For completeness, P-Ray includes a program to approximate it; however cache size can also be measured with other programs [128, 131]. Algorithm 12 calculates the number of caches. Each thread accesses a private array approximately sized to L2 in order to cause misses between cores that share the same cache. This array is initialized as described in Algorithm 15 below. The first step of the algorithm is to measure the time it took for one thread to read the elements of this array when running in isolation. This time will be used as a reference to interpret

Algorithm 12: Calculate Cache Mapping

```
cache-mapping(core_1, core_2) {
i ← 1
while i ≤ SAMPLES do
   Spawn thread-work(core_1,1)
   Spawn thread-work(core_2,2)
   Wait on thread barrier
   Wait for threads to complete
   i ← i + 1
end while
}
```

```
thread-work(core, id) {
Set-thread-affinity (core)
Pointer p ← Initialize local data
Wait on thread barrier
Start timing
for i ← 0 to SIZE do
   p ← *p
end for
Stop timing
if id = 1 then
   Print core pair, timing
end if
}
```

the results.

Then, a similar test is executed with two threads. Each thread sets its affinity to a different core, initializes its workset, and waits on a barrier for the other thread. Once both threads leave the barrier, the time it takes for the threads to read their arrays while running simultaneously is then measured. If the measured execution time is higher than the reference time, the conclusion is that both threads ran on cores that share a cache, and that performance degraded due to the worksets of the two threads competing for cache space. If it is the same, it will be inferred that both threads ran on separate caches.

This test is run for all pairs of cores on the system. After gathering all the results, P-Ray determines the number of caches on the system and the ID of the cores that map to them.

6.3.3 Processor Mapping

Here is a solution to determine the processors' interconnection topology.

Algorithm 13 uses two threads sharing a workset the size of the L1 cache, but running on separate cores. This algorithm functions by having one thread that reads and modifies its workset (i.e. brings it into L1) and measuring the time it takes for the second thread

Algorithm 13: Calculate Processor Mapping

```
                                    Require: Pointer p is global
                                      thread-work2(core_id1) {
                                      Set-thread-affinity (core_id1)
thread-work1(core_id2) {              Wait on thread barrier
Set-thread-affinity (core_id2)        Start timing
p ← InitData(data,size,stride)        for i ← 0 to L1SIZE do
Wait on thread barrier                  p ← *p
}                                     end for
                                      Stop timing
                                      Print (core_id1,core_id2), timing
                                      }
```

to read the data. By comparing the different access times of all possible pairs of cores, this program will determine the different relative distances between all cores.

6.3.4 Effective Bandwidth

This is the solution used to measure the available bandwidth for one core to memory by saturating it from one or several threads. In the following description, the term "memory" will be used both for memory and caches unless otherwise specified.

Algorithm 14: Calculate Bandwidth

```
Iteration1() {                        Iteration2() {
Start timing                          Start timing
for i ← 0 to N_ITER do                for i ← 0 to N_ITER/2 do
  p ← *p                                p1 ← *p1
end for                                 p2 ← *p2
Stop timing                           end for
Print timing                          Stop timing
}                                     Print timing
                                      }
```

An array is used that does pointer chaining with multiple entry points (this data structure and its initialization are described in Algorithm 15 and Figure 6.3 below). The offset between two entry points is set to the size of a memory page. The stride between accesses is set to

the smallest multiple of the page size that avoids overlap between chains.

To target a specific level in the memory hierarchy, the number of elements in the pointer chain is controlled before the loop back. This ensures that any reuse would happen after the data was displaced from levels closer to the core. Moreover, when measuring memory bandwidth, L2 is flushed after initialization.

Single-threaded bandwidth

The first step is to measure the bandwidth to memory for an isolated thread.

In the first iteration, the program traverses the array through a single entry pointer, as shown by `Iteration1()` in Algorithm 14. The code in `Iteration1()` serializes array accesses, as the access to the next element of the array cannot be issued until the pointer load returns. The second iteration of this program traverses the array through two entry pointers, as shown by `Iteration2()` in Algorithm 14. This loop has two independent accesses that can be sent simultaneously to the memory. However, accesses in an iteration depend on the accesses of the previous iteration for the same pointer chain due to the loop-carried dependences for all pointers. The program proceeds by increasing the number of independent requests. By measuring the execution time of these loops, P-Ray can determine the number of requests that a core can have in-flight as well as its saturation point.

To calculate effective bandwidth, the following equation is used:

$$BW_{effective} = \frac{ClockFreq \times ReadSize}{CyclesPerRead} \tag{6.1}$$

$CyclesPerRead$ is obtained by dividing the execution cycles at any of the saturation points by the number of iterations in the access loop. $ClockFreq$ is the clock rate for the given core. $ReadSize$ is the size of the cache line being read.

Multi-threaded bandwidth

The next metric is to look at the memory bandwidth when multiple cores are sending requests simultaneously. This is provided by Algorithm 14, where it is run in parallel over multiple threads. When considering a cache, it is important to execute the program with threads on the cores that share that cache. When considering memory, the program is executed with any number of cores in the system.

To better understand the impact of concurrent access on the bandwidth for the targeted memory, different numbers of threads are tested: anywhere between two and the number of cores sharing the targeted memory.

6.4 Implementation

6.4.1 Requirements

The P-Ray software has two major requirements:

1. A high resolution wall timer. On Intel machines, the RDTSC instruction is used to get timing in clock cycles [133].

2. Library and operating system support to set thread-to-core affinity.

6.4.2 Implementation Details

Pointer chaining

The main data structure used in most of the solutions is an array of pointers where each element of the array contains the address to the next element to access when traversing the structure. A similar data structure has been used by X-ray [128] and LMbench [131], but here it is initialized using different techniques.

A picture of the data structure is shown in Figure 6.3, and the algorithm for its initialization is shown in Algorithm 15. The initialization algorithm takes five arguments:

- `data` : a pointer to the allocated memory.
- `size` : the size in memory of the data structure.
- `stride` : the distance between two consecutive accesses.
- `offset` : the distance between two entry points.
- `entries` : the number of entry pointers.

The initialization routine uses a stride larger than page size to circumvent the hardware prefetcher and offset larger than the cache line size to prevent consecutive entry pointers from sharing a cache line. Since some experiments need to run for a large number of iterations, a limit on the size of the array is placed by having the last element of the chain point back to the first element (bottom line of Algorithm 15).

Algorithm 15: Pointer Chaining

Init-data($data$,$size$,$stride$,$offset$,$entries$)

$i \leftarrow 0$
while $i < entries$ **do**
  $uoffset \leftarrow i * offset$
  Init-entry($data$,$size$,$stride$,$uoffset$)
  $i \leftarrow i + 1$
end while
Init-entry($data$,$size$,$stride$,$offset$)
$i \leftarrow offset$
while $i \leq size - stride$ **do**
  $data[i] \leftarrow \&data[i + stride]$
  $i \leftarrow i + stride$
end while
$data[i] \leftarrow \&data[offset]$

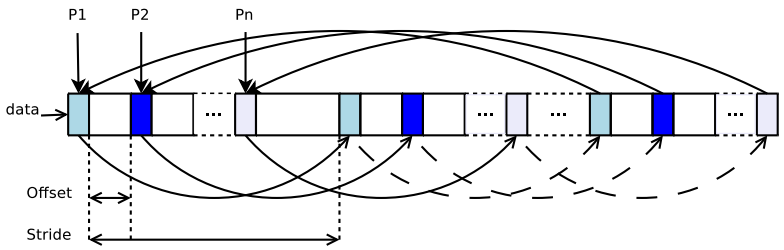


Figure 6.3: Pointer Chaining: General Case

This structure has many advantages: i) it minimizes overhead, as no address has to be computed, ii) it allows for easy ways to experiment with different access patterns by tuning the initialization parameters, and iii) it prevents compiler optimizations that could interfere with performance measurements.

Loop Overhead

The loop overhead should not be considered in the timing. Thus, in order to minimize the control overhead, the main data access loops are unrolled by a factor of 512. This value was chosen because it reduces loop overhead without adding substantial instruction cache pressure. Additionally, an empty loop is timed in order to remove the control overhead from the timing. This way, the final time reflects the actual access times.

Code Reordering

In order to prevent the compiler from performing any reordering of instructions within the timed kernel, `volatile` data modifiers are used. It is important to be careful not to excessively mark every variable volatile when used outside of timed kernels, as this can hurt performance substantially.

System Noise

To deal with the problem of system noise from the operating system and other user applications, numerous timing samples are taken, and the sample with the minimum value is the result. This compensates for other programs and daemons running on the system.

6.5 Evaluation

6.5.1 Experimental Environment

Three different architectures that were evaluated are described in Table 6.1.

	X86-64 Intel Xeon Harpertown	X86-64 Intel Core 2 Quad Kentsfield	Sun UltraSPARC T1 Niagara
Num cores	8[a]	4	8 (32 threads)
Clock Rate (GHz)	2.0	2.4	1.2
L2 Cache size (MB)	6	4	3
OS (Kernel)	Fedora 8 (2.6.24)	Fedora 8 (2.6.24)	Solaris 10
Compiler	GCC 4.1.2	GCC 4.1.2	GCC 3.4.3

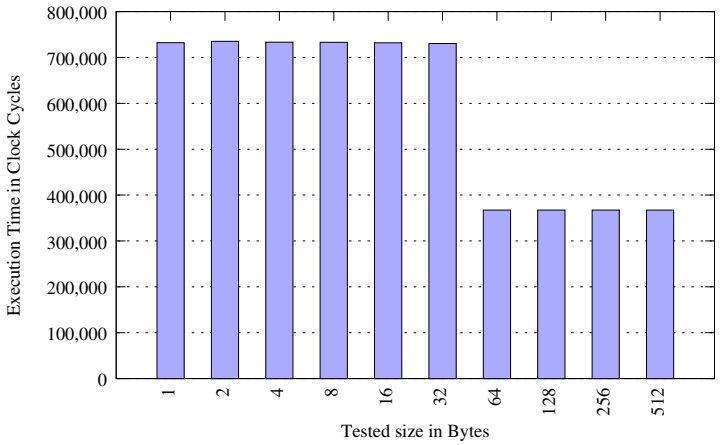
[a]Composed of two four-core chips

Table 6.1: Architectures Tested

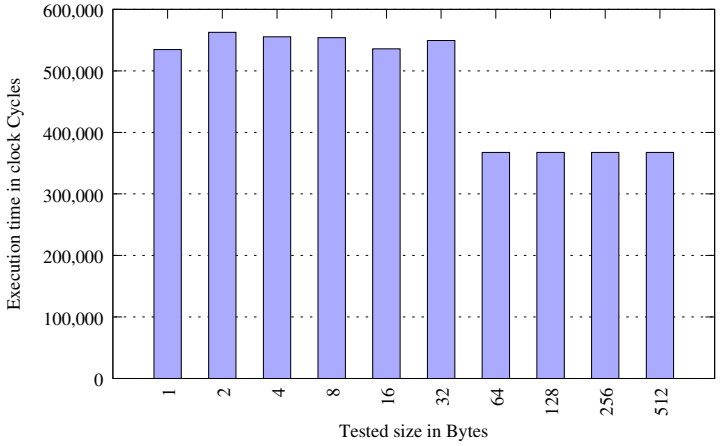
6.5.2 Experimental Results

Coherence Block Size

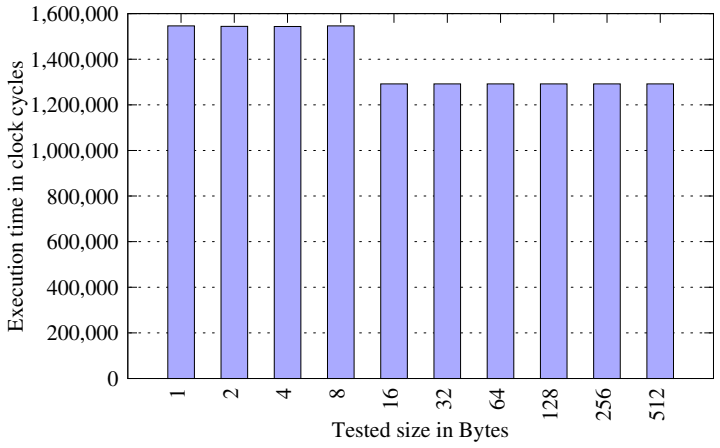
Figure 6.4 illustrates the results for Algorithm 11. On the Intel machines (Figure 6.4a and 6.4b), the threads were mapped to cores not sharing the L2 cache. The results show that, on the Intel machines, there is a notable time decrease as soon as the two accesses are 64 bytes apart. From this, the conclusion is that the coherence protocol on these machines uses 64-byte blocks.



(a) Harpertown



(b) Kentsfield



(c) Niagara

Figure 6.4: Coherence Block Size Results

On the Sun UltraSPARC T1 (Figure 6.4c), the observation is that the largest performance difference occurs at the 16-byte block size, which corresponds to the size of the L1 data cache block.

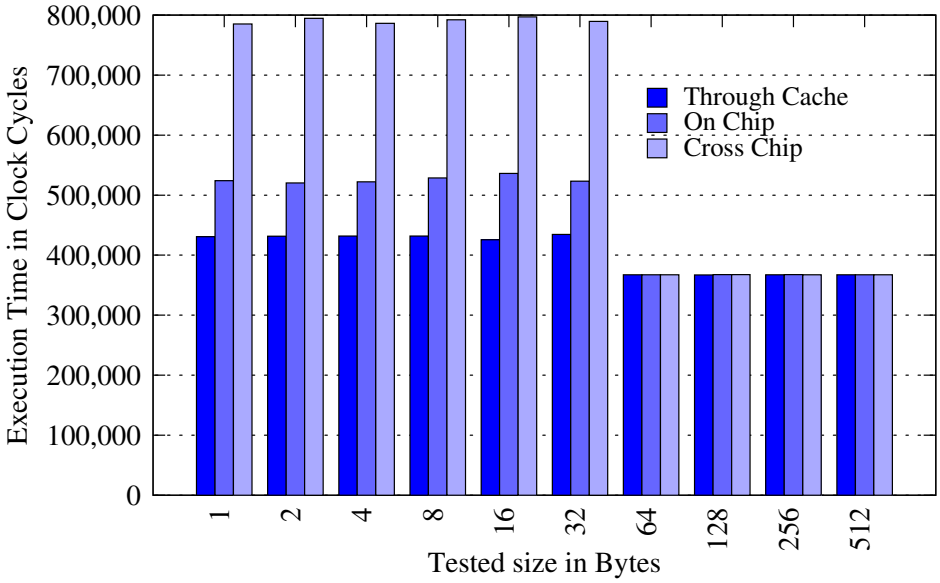
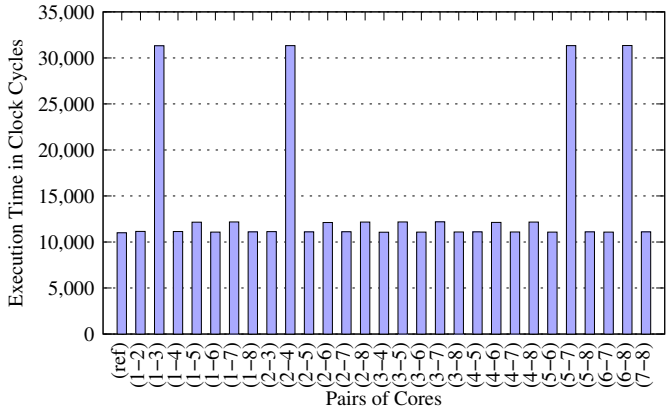
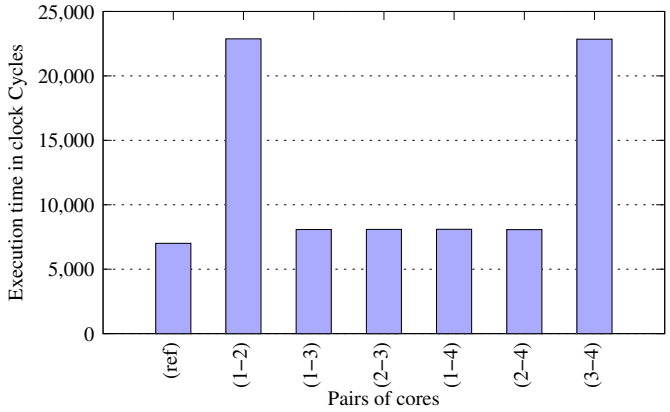


Figure 6.5: Coherence Block Size and Communication Latency

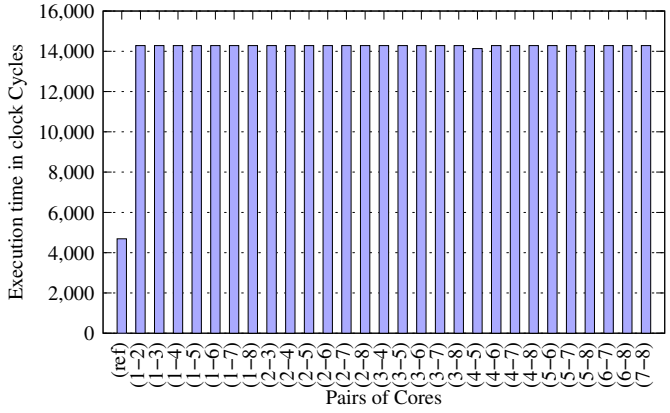
To show block sizes at the different cache levels and communication latencies, an evaluation of different mappings of threads to cores is performed. Figure 6.5 shows the results for the Intel Harpertown architecture, which has eight cores. The first step is to map the threads to the two cores on the same dual-core. The second step is to map the threads to cores on the same chip but not the same cache. The final mapping is the threads to cores on different chips. For this machine, it is observable that while the block size is always 64 bytes, the different values of execution time show the different communication latencies among the different cores. There is a higher communication latency when the communicating cores are on different chips, and the communication cost is lower when the cores are closer together.



(a) Harpertown



(b) Kentsfield



(c) Niagara

Figure 6.6: Cache Mapping

Cache Mapping

Figure 6.6 illustrates the results for Algorithm 12. The results clearly show which of the cores share a cache.

By looking at the pairs of cores with the highest access times, it is observable that for the Harpertown (Figure 6.6a), core pairs of ID $(1-3)$,$(2-4)$,$(5-7)$, and $(6-8)$ share a cache, and for the Kentsfield (Figure 6.6b), the core pairs ID $(1-2)$ and $(3-4)$ share a cache.

For the Sun UltraSPARC (Figure 6.6c), all core pairs show the same performance. When comparing this performance with the single thread reference time, it is noticeable that all core pairs perform poorly. From this, it can be inferred that all cores share the same cache.

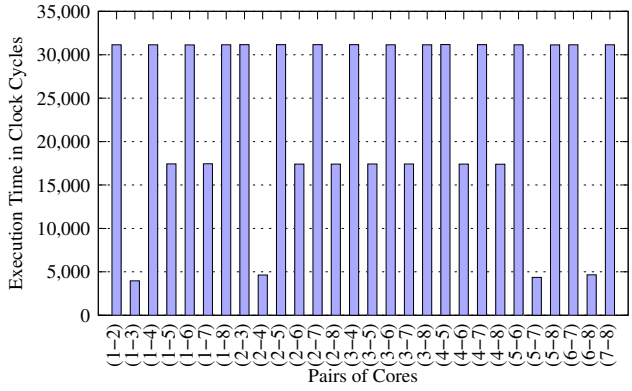
Processor Mapping

Figure 6.7 illustrates the results for Algorithm 13. For the Harpertown (Figure 6.7a), three different distances are observed. First, there are the pairs of cores that are the closest. These pairs correspond to those that share a cache in Figure 6.6a: $(1-3)$,$(2-4)$,$(5-7)$, and $(6-8)$. Then there are two groups of four cores, where communicating between pairs in a group is faster than communicating between cores in different groups: $((1-3)(5-7))$ and $((2-4)(6-8))$. Those results confirm what is found on the design of the two architectures: the machine is composed of two four-core chips, with each four-core chip composed of two combined dual-cores. For the Kentsfield (Figure 6.7b), it is noticeable that pairs of cores that share a cache communicate faster.

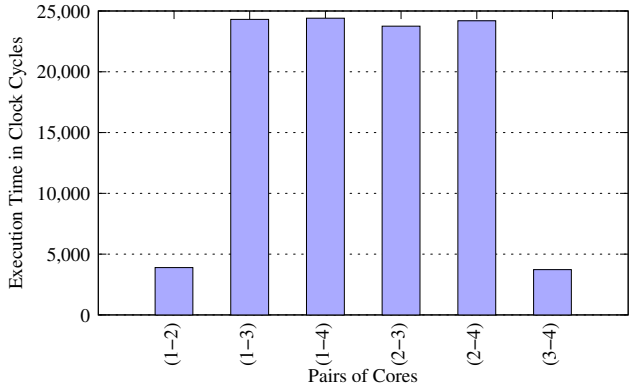
For the Niagara (figure 6.7c), results show that all cores are equidistant, which confirms the results obtained for the cache mapping.

Effective L2 Bandwidth

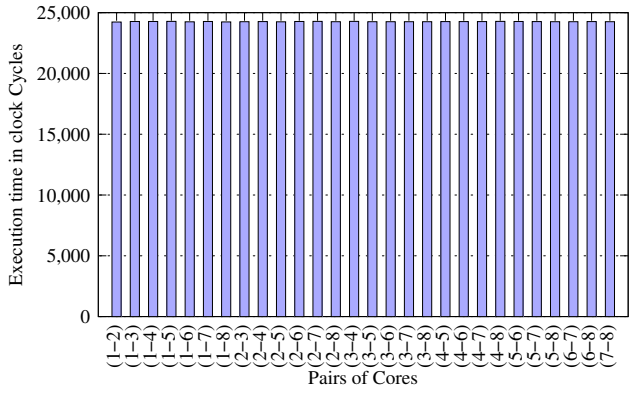
Figure 6.8 illustrates the results for Algorithm 14. Data is shown for each platform when a thread is run in isolation, and when two or more threads run concurrently. By looking at results for one thread, the observation is that, for all the tested Intel machines, the execution time decreases as the number of independent accesses that can be issued simultaneously in-



(a) Harpertown



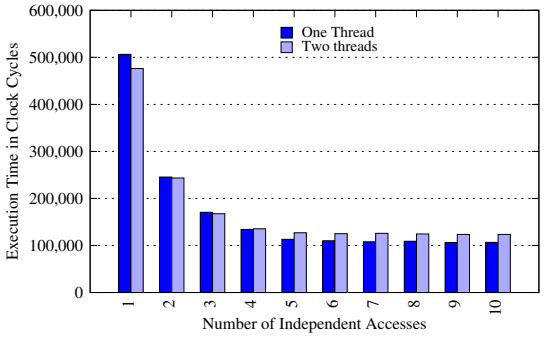
(b) Kentsfield



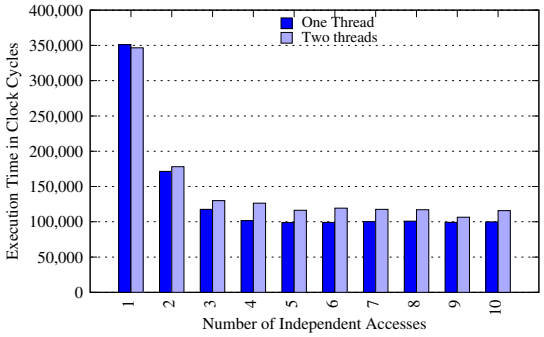
(c) Niagara

Figure 6.7: Processor Mapping

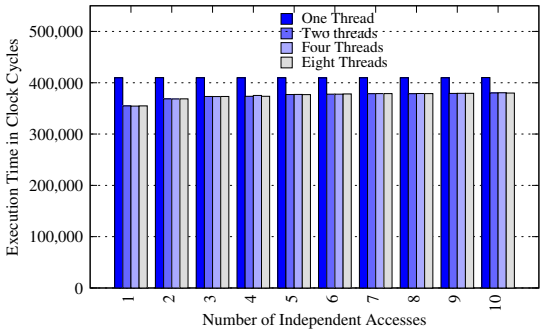
creases. There is a saturation point where the execution time remains constant. The smallest number of independent accesses where the saturation point is reached tell us the number of requests that can be served in parallel. When two or more threads run concurrently, the



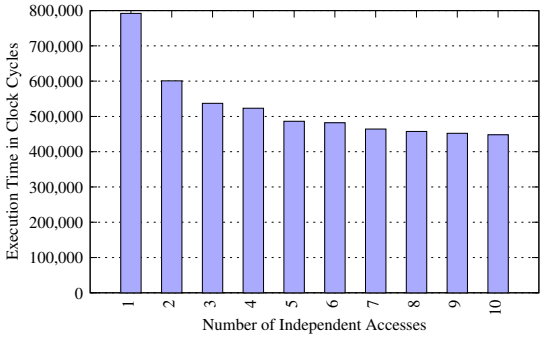
(a) Harpertown



(b) Kentsfield



(c) Niagara



(d) Niagara with branch penalty

Figure 6.8: Effective Bandwidth to L2 Results

bars have a similar trend, but have a slightly higher execution time.

Figure 6.8d shows the execution time for the program running without removing the loop overhead. The improvement in execution time looks like it comes from more parallelism between memory requests. In fact, the Niagara does not have branch prediction; the reduction in execution time is only due to the decrease in number of iterations (i.e. number of branches per access).

Finally Table 6.2 shows the bandwidth computed using the formula shown in Section 6.3.4 for the different number of threads.

Machine	L1 Block	Cycles per Access (concurrent accesses)				Effective Bandwidth GB/s			
# threads		1	2	4	8	1	2	4	8
Harpertown	64B	6.2 (9)	7.5 (9)	-	-	19.3	15.8	-	-
Kentsfield	64B	6.1 (6)	6.5 (9)	-	-	23.7	22.0	-	-
Niagara	16B	25.0 (6)	21.7 (1)	21.6 (1)	21.7 (1)	0.7	0.8	0.8	0.8

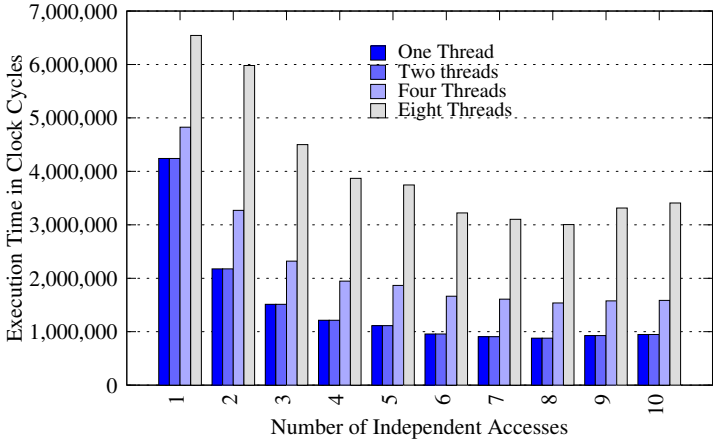
Table 6.2: Effective Bandwidth to L2

Effective Memory Bandwidth

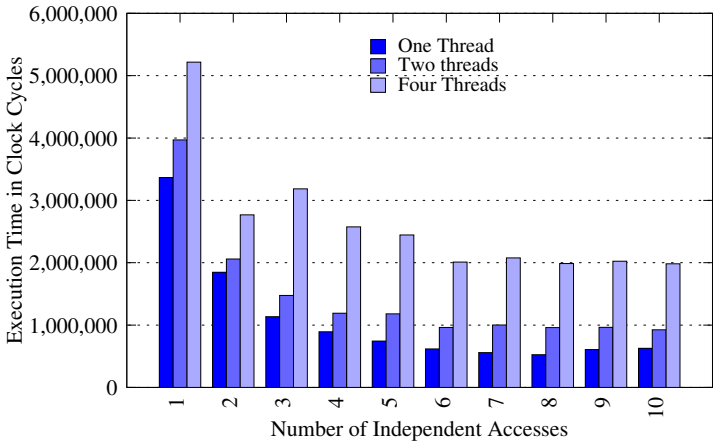
Figure 6.9 illustrates the results for Algorithm 14. For the Intel processors (Figure 6.9a and 6.9b), as the number of cores accessing memory increases, the observation is a substantial decrease in bandwidth. On the Niagara (Figure 6.9c), the results are similar as for the L2 cache; the bandwidth available to the cores stays the same regardless of the number of concurrent requests. Finally, Table 6.3 shows the values for the effective bandwidth.

Machine	L2 Block	Cycles per Access (concurrent accesses)				Effective Bandwidth GB/s			
# threads		1	2	4	8	1	2	4	8
Harpertown	64B	52.1 (8)	53.6 (8)	93.9 (9)	183.3 (9)	2.3	2.2	1.3	0.7
Kentsfield	64B	32.0 (8)	56.5 (10)	121.0 (10)	-	4.5	2.5	1.2	-
Niagara	64B	111.6 (10)	107.6 (8)	108.5 (8)	110.1 (8)	0.6	0.7	0.7	0.7

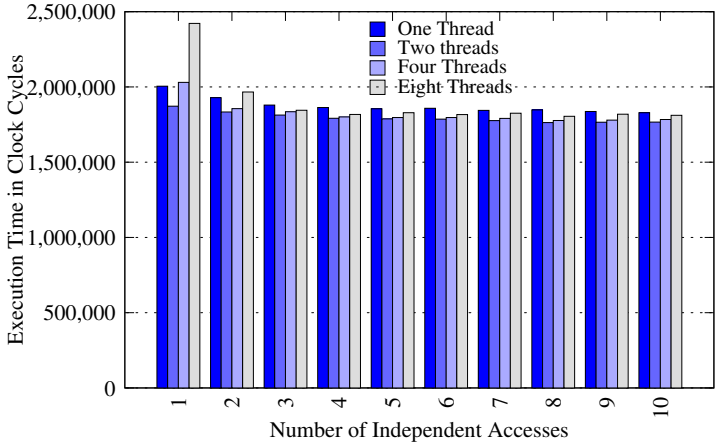
Table 6.3: Effective Bandwidth to Memory



(a) Harpertown



(b) Kentsfield



(c) Niagara

Figure 6.9: Effective Bandwidth to Memory Results

6.6 Related Work

As discussed in the introduction, there are other software suites such as LMBench[131], Saavedra[130], and X-Ray[128] that measure architectural characteristics and, while they focus on single core features, P-Ray focuses on multi-core specific features. Benchmark suites like SPEC OMP[2] are designed for shared memory multiprocessors. Such benchmarks only give a relative performance scale, but do not give any information about the characteristics of the targeted system.

Since P-Ray was originally developed [93], there has been newer benchmark suites used to measure multi-core characteristics. The hwloc [129] project is a portable software package that is similar to P-Ray in computing the topology of the machine cores and aches. An important distinction is that hwloc computes topology information through the operating system interface, whereas in P-Ray, the majority of information is computed with minimal support from the operating system. Servet [134] is a similar approach to P-Ray in that it computes the same machine characteristics, with the major difference being that Servet provides an additional benchmark used to determine communication latencies between cores.

6.7 Conclusion

The work that has been presented here is a suite of conceptually simple solutions that focus on multi-core characteristics. This benchmark suite returns results that are in accordance with vendor specifications when available and coherent when they are not.

The main difference between P-Ray and existing software is that P-Ray offers a unique view of the system design, showing the position of the different cache levels and relative distances between (virtual) cores in the system. With this information at hand, a programmer has the ability to use more efficient hardware-aware optimizations in their applications.

[2]Standard Performance Evaluation Corporation. `http://www.spec.org/omp`

In addition, P-Ray provides a faster means to calculate a cache block size by exploiting false sharing. Finally, the execution and analysis framework is extensible, allowing for the addition of further hardware characterization.

Chapter 7

Future Work

Compilation for Dependence-Driven Execution Models

In the approach taken of compiling data-parallel loops onto dependence-driven execution models, the default scheduling strategy of *static* scheduling was used. In many dense workloads, static scheduling is sufficient, but as soon as there is an irregularity in the workload, performance due to load-imbalance can suffer. An area of future work would be to also support other scheduling strategies such as dynamic scheduling.

In Chapter 6, a micro-benchmark suite was proposed in order to determine multi-core machine specific characteristics. One of these benchmarks were used to find the cache line size, which was then used as an input in computing the memory footprint of multiply-nested loop from Section 4.3.3. One area of future work would be for the other machine characteristics to also aid in the optimization of the generated code. For example, determining which processors share a cache could help the compiler and runtime in pinning specific tasks to processors in order to decrease communication traffic for processors that do not directly share a cache.

Language Support to Express Dependences

Chapter 3.6 provides an overall design of language extensions to support explicit ordering constraints. One area of future work would be to interface these language extensions with an existing compiler infrastructure in order to provide new optimization capabilities. This would then be able to sidestep the compiler since, traditionally, the compiler needs to be conservative to guarantee program correctness. For example, one ability would be for the programmer to turn off dependence analysis for a set of loops, and to provide their own set of constraints. An evaluation of the usefulness of this would then be to see what compiler transformations can be enabled with and without these constraints.

This work could also be complimentary to an existing dependence analysis phase of the compiler. Instead of turning off the dependence analysis phase for a set of loops, the programmer could specify what portion of the loops are independent so that the compiler has more freedom in enabling certain loop transformations.

Support for Hierarchical Parallel Architectures

While Chapel already supports execution on traditional clusters containing multi-cores, the work presented in this thesis is primarily focused on single-node architectures containing a GPU. An area of future work would be to combine the work of compiling multiply-nested data-parallel loops onto dependence-driven runtime systems with the work of targeting heterogeneous architectures such as GPUs. In this case, any compiled loop (either for a GPU, or multi-core) would be represented as a task to the runtime. Since it is the job of the runtime to schedule tasks onto the given hardware, any of the tasks from the compiled loops could be mapped onto any available hardware resource. A scenario for this would be to have a distributed system consisting of multiple nodes, where each node contains multiple CPUs and GPU accelerator cards. By expressing both synchronous and asynchronous data-parallel loops, the scheduler can now map each of those onto any of the hardware resources. This

would be beneficial not only for exploiting all the available parallelism, but also increase programmer productivity since the programmer will express parallel loops, and not deal with the communication and machine complexity.

In addition to the computation hierarchy just described, it will be important to also deal with the memory hierarchy for these complex systems. This includes the user needing to develop a technique of sharing data across the global address space of the hierarchy, in addition to the separate address spaces of each node.

Improving GPU Compilation Support

There are many areas of future direction for improving GPU compilation support. For instance, one could strive to avoid exposing too much of the GPU low-level centric code to the programmer. While this is beneficial to compilation of the GPU-centric loops onto a multi-core, it still forces the programmer to think in terms of CUDA in some aspects. For example, this includes relying on CUDA's synchronization primitives. One method to address this is to use the method in which programmers express their algorithms purely in terms of nested `forall`, `coforall`, and `for` loops, and leave it to the compiler to generate the necessary kernel from that. The second method, and possibly complementary approach, is to provide whole-array operation support. The Chapel language currently supports bulk-array operations that convert to parallel `forall` loops. This work can then be expanded upon by having the compiler optimize these operations through techniques such as loop fusion, where the compiler could then safely combine multiple `forall` loops into a single kernel.

Chapter 8

Conclusions

This thesis has attempted to address compilation techniques and language extensions that will be used to target dependence-driven execution models such as the StarPU runtime system. This involved designing new algorithms based on interval analysis to perform loop partitioning into a task-dependence graph. Additionally, a new language extension was introduced to help the programmer explicitly express the ordering constraints between different parts of their source code. This has the benefit of aiding (or replacing) the compiler's traditional dependence analysis framework to help determine which loop transformations are legal to the user.

A new optimization was introduced to help decrease barrier synchronization overhead from multiply-nested parallel loops. This optimization is based on a heuristic that determines the essential tile dimensions for a loop nest, and then coalesces the loop nest together. Part of this work also helps to explore the area of choosing the correct number of processors for execution, as it is not always ideal to use the maximum available resources due to runtime overheads.

In addition to compiling data-parallel loops for different runtime systems, this work also explored compilation techniques to map data-parallel loops onto different architecture back-

ends, primarily GPU-based accelerators. This work is then extended. By taking source code that has been already hand-tuned for GPUs, the compiler can now map those codes back onto a traditional multi-core platform.

To help aid in optimization of applications, it is important to understand the underlying machine characteristics that the application will execute on. A set of benchmarks named P-Ray were developed to help address this. In particular, these benchmarks help characterize multi-core specific characteristics for a set of different platforms. These characteristics can then be applied to the work presented earlier in the thesis. For example, by having P-Ray compute the cache block size, a more precise approximation of the memory footprint for different parallel loop kernels can then be found.

Bibliography

[1] F. E. Allen and J. Cocke. A program data flow analysis procedure. *Commun. ACM*, 19(3):137–, March 1976.

[2] Matthew S. Hecht. *Flow Analysis of Computer Programs*. Elsevier Science Inc., New York, NY, USA, 1977.

[3] Herb Sutter. The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software. *Dr. Dobbs Journal*, 30(3):202–210, 2005.

[4] Wm. A. Wulf and Sally A. McKee. Hitting the Memory Wall: Implications of the Obvious. *SIGARCH Comput. Archit. News*, 23(1):20–24, March 1995.

[5] Volodymyr Kindratenko. Scientific Computing with GPUs. *Computing in Science and Engineering*, 14(3):8–9, 2012.

[6] C. Baker, G. Davidson, T. M. Evans, S. Hamilton, J. Jarrell, and W. Joubert. High Performance Radiation Transport Simulations: Preparing for Titan. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 47:1–47:10, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

[7] Albert Sidelnik, Saeed Maleki, Bradford L. Chamberlain, María Jesús Garzarán, and David A. Padua. Performance Portability with the Chapel Language. In *Parallel and Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*, 2012.

[8] Gregory Diamos, Andrew Kerr, Sudhakar Yalamanchili, and Nathan Clark. Ocelot: A Dynamic Compiler for Bulk-synchronous Applications in Heterogeneous Systems. In *PACT '10: The Nineteenth International Conference on Parallel Architectures and Compilation Techniques*, 2010.

[9] John A. Stratton, Vinod Grover, Jaydeep Marathe, Bastiaan Aarts, Mike Murphy, Ziang Hu, and Wen-mei W. Hwu. Efficient Compilation of Fine-grained SPMD-threaded Programs for Multicore CPUs. In *CGO '10: Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*, pages 111–119, New York, NY, USA, 2010. ACM.

[10] Daniel D. Gajski, David J. Kuck, and David A. Padua. Dependence Driven Computation. In *COMPCON*, pages 156–161. IEEE Computer Society, 1981.

[11] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. In *Euro-Par*, pages 863–874, 2009.

[12] George Bosilca, Aurelien Bouteiller, Anthony Danalis, Thomas Hérault, Pierre Lemarinier, and Jack Dongarra. DAGuE: A Generic Distributed DAG Engine for High Performance Computing. *Parallel Computing*, 38(1-2):37–51, 2012.

[13] Eduard Ayguadé, Rosa M. Badia, Francisco D. Igual, Jesús Labarta, Rafael Mayo, and Enrique S. Quintana-Ortí. An Extension of the StarSs Programming Model for Platforms with Multiple GPUs. In *Proceedings of the 15th International Euro-Par Conference on Parallel Processing*, Euro-Par '09, pages 851–862, Berlin, Heidelberg, 2009. Springer-Verlag.

[14] Inc. ET International. SWARM (SWift Adaptive Runtime Machine). White paper, 2011.

[15] Stéphane Zuckerman, Joshua Suetterlein, Rob Knauerhase, and Guang R. Gao. Using a "Codelet" Program Execution Model for Exascale Machines: Position Paper. In *Proceedings of the 1st International Workshop on Adaptive Self-Tuning Computing Systems for the Exaflop Era*, EXADAPT '11, pages 64–69, New York, NY, USA, 2011. ACM.

[16] Kathleen Knobe. Ease of Use with Concurrent Collections (CnC). In *Proceedings of the First USENIX conference on Hot topics in parallelism*, HotPar'09, pages 17–17, Berkeley, CA, USA, 2009. USENIX Association.

[17] Alejandro Duran, Roger Ferrer, Eduard Ayguadé, Rosa M. Badia, and Jesús Labarta. A Proposal to Extend the OpenMP Tasking Model with Dependent Tasks. *International Journal of Parallel Programming*, 37(3):292–305, 2009.

[18] Daniel A. Orozco. *TIDeFlow: A Dataflow-Inspired Execution Model for High Performance Computing Programs*. PhD thesis, University of Delaware, 2012.

[19] James Reinders. *Intel Threading Building Blocks*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, first edition, 2007.

[20] Jack Dongarra, Peter H. Beckman, and Terry Moore et al. The International Exascale Software Project Roadmap. *IJHPCA*, 25(1):3–60, 2011.

[21] Bradford L. Chamberlain, David Callahan, and Hans P. Zima. Parallel Programmability and the Chapel Language. *International Journal of High Performance Computing Applications*, 21:291–312, 2007.

[22] Bradford L. Chamberlain, Steven J. Deitz, David Iten, and Sung-Eun Choi. User-defined Data Distributions in Chapel: Philosophy and Framework. In *HotPar '10: Proc. 2nd Workshop on Hot Topics in Parallelism*, 2010.

[23] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 1988.

[24] Matthew S. Hecht and Jeffrey D. Ullman. Flow Graph Reducibility. In *Proceedings of the fourth annual ACM symposium on Theory of computing*, STOC '72, pages 238–250, New York, NY, USA, 1972. ACM.

[25] Edward S. Lowry and C. W. Medlock. Object Code Optimization. *Commun. ACM*, 12(1):13–22, January 1969.

[26] Donald E. Knuth. An Empirical Study of FORTRAN Programs. *Softw., Pract. Exper.*, 1(2):105–133, 1971.

[27] James Stanier and Des Watson. A Study of Irreducibility in C Programs. *Softw. Pract. Exper.*, 42(1):117–130, January 2012.

[28] Barbara G. Ryder and Marvin C. Paull. Elimination Algorithms for Data Flow Analysis. *ACM Comput. Surv.*, 18(3):277–316, September 1986.

[29] David A. Padua and Michael J. Wolfe. Advanced Compiler Optimizations for Supercomputers. *Commun. ACM*, 29(12):1184–1201, December 1986.

[30] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The Program Dependence Graph and Its Use in Optimization. *ACM Trans. Program. Lang. Syst.*, 9(3):319–349, July 1987.

[31] Ron Cytron, Michael Hind, and Wilson C. Hsieh. Automatic Generation of DAG Parallelism. In *PLDI*, pages 54–68, 1989.

[32] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Trans. Program. Lang. Syst.*, 13(4):451–490, October 1991.

[33] Robit Chandra, Leonardo Dagum, Dave Kohr, Dror Maydan, Jeff McDonald, and Ramesh Menon. *Parallel Programming in OpenMP*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001.

[34] Guy E. Blelloch. NESL: A Nested Data-Parallel Language (Version 2.6). Technical report, Pittsburgh, PA, USA, 1993.

[35] Lawrence Snyder. *A Programmer's Guide to ZPL*. MIT Press, 1999.

[36] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable Parallel Programming with CUDA. *Queue*, 6(2):40–53, 2008.

[37] Richard E. Ladner, Michael, and J. Fischer. Parallel Prefix Computation. *Journal of the ACM*, 27:831–838, 1980.

[38] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: An Object-oriented Approach To Non-uniform Cluster Computing. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 519–538, New York, NY, USA, 2005. ACM.

[39] Jun Shirako, David M. Peixotto, Vivek Sarkar, and William N. Scherer. Phasers: A Unified Deadlock-free Construct for Collective and Point-To-Point Synchronization. In *Proceedings of the 22nd annual international conference on Supercomputing*, ICS '08, pages 277–288, New York, NY, USA, 2008. ACM.

[40] David Alejandro Padua. *Multiprocessors: Discussion of Some Theoretical and Practical Problems*. PhD thesis, Champaign, IL, USA, 1980. AAI8018194.

[41] Ronald Gary Cytron. *Compile-time Scheduling and Optimization for Asynchronous Machines (Multiprocessor, Compiler, Parallel Processing)*. PhD thesis, Champaign, IL, USA, 1984. AAI8502121.

[42] University of Illinois at Urbana-Champaign. Center for Supercomputing Research, Development, and C.D. Polychronopoulos. *Loop Coalescing: a Compiler Transformation for Parallel Machines*. CSRD. University of Illinois, Center for Supercomputing Research and Development, 1987.

[43] Ken Kennedy and John R. Allen. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002.

[44] Jack B. Dennis. Data Flow Computer Architecture. In *Encyclopedia of Parallel Computing*, pages 508–512. 2011.

[45] K.P. Gostelow and W. Plouffe. *The (preliminary) Id Report: An Asynchronous Programming Language and Computing Machine*. Technical report (University of California, Irvine. Dept. of Information and Computer Science). Department of Information and Computer Science, University of California, Irvine, 1978.

[46] John Feo, David C. Cann, and R. R. Oldehoeft. A Report on the Sisal Language Project. *J. Parallel Distrib. Comput.*, 10(4):349–366, 1990.

[47] James R. McGraw. The VAL Language: Description and Analysis. *ACM Trans. Program. Lang. Syst.*, 4(1):44–82, January 1982.

[48] D. D. Gajski, D. A. Padua, D. J. Kuck, and R. H. Kuhn. A Second Opinion on Data Flow Machines and Languages. *Computer*, 15(2):58–69, February 1982.

[49] Jack Dongarra and Piotr Luszczek. PLASMA. In *Encyclopedia of Parallel Computing*, pages 1568–1570. 2011.

[50] Khronos OpenCL Working Group. *The OpenCL Specification, version 1.0.29*, 8 December 2008.

[51] Chapel Specification 0.82. `http://chapel.cray.com`.

[52] Eric Allen, David Chase, Joe Hallett, Victor Luchangco, Jan-Willem Maessen, Sukyoung Ryu, Guy L. Steele Jr., and Sam Tobin-Hochstadt. The Fortress Language Specification. Technical report, Sun Microsystems, Inc., 2007.

[53] Lawrence Snyder. The Design and Development of ZPL. In *Proceedings of the third ACM SIGPLAN conference on History of programming languages*, HOPL III, pages 8–1–8–37, New York, NY, USA, 2007. ACM.

[54] Ken Kennedy, Charles Koelbel, and Hans Zima. The Rise and Fall of High Performance Fortran: An Historical Object Lesson. In *Proceedings of the third ACM SIGPLAN conference on History of programming languages*, HOPL III, pages 7–1–7–22, New York, NY, USA, 2007. ACM.

[55] Mike Ringenburg and Sung-Eun Choi. Optimizing Loop-level Parallelism in Cray XMT (TM) Applications. In *Cray Users Group (CUG) 2009)*, Atlanta, GA, 2009.

[56] Adrian Prantl, Thomas Epperly, Shams Imam, and Vivek Sarkar. Interfacing Chapel with Traditional HPC Programming Languages. In *Fifth Conference on Partitioned Global Address Space Programming Model*, Galveston Island, Texas, October 2011.

[57] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[58] Bradford L. Chamberlain. *The Design and Implementation of a Region-Based Parallel Language*. PhD thesis, University of Washington, November 2001.

[59] Steven J. Deitz, David Callahan, Bradford L. Chamberlain, and Lawrence Snyder. Global-view Abstractions for User-defined Reductions and Scans. In *PPoPP '06: Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 40–47, New York, NY, USA, 2006. ACM.

[60] Bradford L. Chamberlain, Sung-Eun Choi, Steven J. Deitz, David Iten, and Vassily Litvinov. Authoring User-Defined Domain Maps in Chapel. *Cray Users Group Conference (CUG)*, 2011.

[61] Elkin Garcia and Guang Gao. Strategies For Improving Performance and Energy Efficiency on a Many-core. In *Proceedings of the ACM International Conference on Computing Frontiers*, CF '13, pages 9:1–9:4, New York, NY, USA, 2013. ACM.

[62] Stanimire Tomov, Rajib Nath, Hatem Ltaief, and Jack Dongarra. Dense Linear Algebra Solvers for Multicore with GPU Accelerators. In *IPDPS Workshops*, pages 1–8, 2010.

[63] Johan Enmyren and Christoph W. Kessler. SkePU: A Multi-backend Skeleton Programming Library for Multi-GPU Systems. In *Proceedings of the fourth international workshop on High-level parallel programming and applications*, HLPP '10, pages 5–14, New York, NY, USA, 2010. ACM.

[64] Azzam Haidar, Hatem Ltaief, Asim YarKhan, and Jack Dongarra. Analysis of Dynamically Scheduled Tile Algorithms for Dense Linear Algebra on Multicore Architectures. *Concurr. Comput. : Pract. Exper.*, 24(3):305–321, March 2011.

[65] Yong-Fong Lee, Barbara G. Ryder, and Marc E. Fiuczynski. Region Analysis: A Parallel Elimination Method for Data Flow Analysis. *IEEE Trans. Softw. Eng.*, 21(11):913–926, November 1995.

[66] Karlin et al. Exploring Traditional and Emerging Parallel Programming Models using a Proxy Application. In *Parallel and Distributed Processing Symposium (IPDPS), 2013 IEEE 27th International*, 2013.

[67] Sebastian Nanz, Scott West, and Kaue Soares da Silveira. Benchmarking Usability and Performance of Multicore Languages. *CoRR*, abs/1302.2837, 2013.

[68] Nan Dun and K. Taura. An Empirical Performance Study of Chapel Programming Language. In *Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW), 2012 IEEE 26th International*, pages 497–506, 2012.

[69] Timothy A. Davis and Yifan Hu. The University of Florida Sparse Matrix Collection. *ACM Trans. Math. Softw.*, 38(1):1:1–1:25, December 2011.

[70] William Humphrey, Andrew Dalke, and Klaus Schulten. VMD – Visual Molecular Dynamics. *Journal of Molecular Graphics*, 14:33–38, 1996.

[71] Parboil Benchmark Suite. http://impact.crhc.illinois.edu/parboil.php.

[72] Samuel S. Stone, Justin P. Haldar, Stephanie C. Tsao, Wen-mei W. Hwu, Zhi-Pei Liang, and Bradley P. Sutton. Accelerating Advanced MRI Reconstructions on GPUs. In *CF '08: Proc. 5th conference on Computing frontiers*, pages 261–272, 2008.

[73] Christian H. Bischof and Charles Van Loan. The WY Representation for Products of Householder Matrices. Technical report, Ithaca, NY, USA, 1985.

[74] M. Girkar and C. D. Polychronopoulos. Automatic Extraction of Functional Parallelism from Ordinary Programs. *IEEE Trans. Parallel Distrib. Syst.*, 3:166–178, March 1992.

[75] Hironori Kasahara, Hiroki Honda, M. Iwata, and M. Hirota. A Compilation Scheme for Macro-Dataflow Computation on Hierarchical Multiprocessor Systems. In *ICPP (2)*, pages 294–295, 1990.

[76] Micah Beck and Keshav Pingali. From Control Flow to Dataflow. In *ICPP (2)*, pages 43–52, 1990.

[77] Hironori Kasahara, Hiroki Honda, A. Mogi, A. Ogura, K. Fujiwara, and Seinosuke Narita. A Multi-Grain Parallelizing Compilation Scheme for OSCAR (Optimally Scheduled Advanced Multiprocessor). In *LCPC*, pages 283–297, 1991.

[78] Milind Girkar and Constantine D. Polychronopoulos. The Hierarchical Task Graph as a Universal Intermediate Representation. *Int. J. Parallel Program.*, 22:519–551, October 1994.

[79] Milind Girkar and Constantine D. Polychronopoulos. Extracting Task-Level Parallelism. *ACM Trans. Program. Lang. Syst.*, 17(4):600–634, 1995.

[80] J.T. Schwartz. *A Design for Optimizations of the Bitvectoring Class: By J.T. Schwartz and M. Sharir*. Courant Institute of Mathematical Sciences, New York University, 1979.

[81] Vivek Sarkar and John L. Hennessy. Partitioning Parallel Programs for Macro-Dataflow. In *LISP and Functional Programming*, pages 202–211, 1986.

[82] Ron Cytron, J. Ferrante, and V. Sarkar. Experiences Using Control Dependence in PTRAN. In *Selected papers of the second workshop on Languages and compilers for parallel computing*, pages 186–212, London, UK, UK, 1990. Pitman Publishing.

[83] Bradford L. Chamberlain, E. Christopher Lewis, and Lawrence Snyder. Language Support for Pipelining Wavefront Computations. In *LCPC*, pages 318–332, 1999.

[84] Henry C. Baker, Jr. and Carl Hewitt. The Incremental Garbage Collection of Processes. In *Proceedings of the 1977 symposium on Artificial intelligence and programming languages*, pages 55–59, New York, NY, USA, 1977. ACM.

[85] D.P. Friedman and D.S. Wise. *The Impact of Applicative Programming on Multiprocessing*. Technical report (Indiana University, Bloomington. Computer Science Dept.). Indiana University, Computer Science Department, 1976.

[86] Chuanle Ke, Lei Liu, Chao Zhang, Tongxin Bai, Bryan Jacobs, and Chen Ding. Safe Parallel Programming Using Dynamic Dependence Hints. In *OOPSLA*, pages 243–258, 2011.

[87] Eduard Ayguade, Marc Gonzalez, Xavier Martorell, and Gabriele Jost. Employing Nested OpenMP for the Parallelization of Multi-zone Computational Fluid Dynamics Applications. *J. Parallel Distrib. Comput.*, 66(5):686–697, May 2006.

[88] Eduard Ayguade, Xavier Martorell, Jesus Labarta, Marc Gonzalez, and Nacho Navarro. Exploiting Multiple Levels of Parallelism in OpenMP: A Case Study. In *Proceedings of the 1999 International Conference on Parallel Processing*, ICPP '99, pages 172–, Washington, DC, USA, 1999. IEEE Computer Society.

[89] C.D. Polychronopoulos. *Parallel Programming and Compilers*. Kluwer international series in engineering and computer science. Kluwer Academic, 1988.

[90] K. Yotov, X. Li, G. Ren, M.J.S. Garzaran, D. Padua, K. Pingali, and P. Stodghill. Is Search Really Necessary to Generate High-Performance BLAS? *Proceedings of the IEEE*, 93(2):358–386, Feb. 2005.

[91] Markus Püschel, José M. F. Moura, Jeremy Johnson, David Padua, Manuela Veloso, Bryan Singer, Jianxin Xiong, Franz Franchetti, Aca Gacic, Yevgen Voronenko, Kang Chen, Robert W. Johnson, and Nicholas Rizzolo. SPIRAL: Code Generation for DSP Transforms. *Proceedings of the IEEE, special issue on "Program Generation, Optimization, and Adaptation"*, 93(2):232– 275, 2005.

[92] M. Wolfe. More Iteration Space Tiling. In *Proceedings of the 1989 ACM/IEEE conference on Supercomputing*, Supercomputing '89, pages 655–664, New York, NY, USA, 1989. ACM.

[93] Alexandre X. Duchateau, Albert Sidelnik, María Jesús Garzarán, and David Padua. Languages and compilers for parallel computing. chapter P-Ray: A Software Suite for Multi-core Architecture Characterization, pages 187–201. Springer-Verlag, Berlin, Heidelberg, 2008.

[94] Monica D. Lam, Edward E. Rothberg, and Michael E. Wolf. The Cache Performance and Optimizations of Blocked Algorithms. *SIGPLAN Not.*, 26(4):63–74, April 1991.

[95] Stephanie Coleman and Kathryn S. McKinley. Tile Size Selection Using Cache Organization and Data Layout. In *Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation*, PLDI '95, pages 279–290, New York, NY, USA, 1995. ACM.

[96] Chen Ding and Yutao Zhong. Predicting Whole-Program Locality Through Reuse Distance Analysis. *SIGPLAN Not.*, 38(5):245–257, May 2003.

[97] Calin Cascaval and David A. Padua. Estimating Cache Misses and Locality Using Stack Distances. In *Proceedings of the 17th annual international conference on Supercomputing*, ICS '03, pages 150–159, New York, NY, USA, 2003. ACM.

[98] Peter L. Montgomery. A Survey of Modern Integer Factorization Algorithms. *CWI Quarterly*, 7:337–366, 1994.

[99] Vivek Sarkar. Loop Transformations for Hierarchical Parallelism and Locality. In *Selected Papers from the 4th International Workshop on Languages, Compilers, and Run-Time Systems for Scalable Computers*, LCR '98, pages 57–74, London, UK, UK, 1998. Springer-Verlag.

[100] Vivek Sarkar and Radhika Thekkath. A General Framework for Iteration-Reordering Loop Transformations. In *Proceedings of the ACM SIGPLAN 1992 conference on Programming language design and implementation*, PLDI '92, pages 175–187, New York, NY, USA, 1992. ACM.

[101] Michael E. Wolf. *Improving Locality and Parallelism in Nested Loops*. Stanford University, 1992.

[102] OpenACC Working Group. The OpenACC Application Programming Interface, Version 1.0. November 2011.

[103] Piotr R Luszczek, David H Bailey, Jack J Dongarra, Jeremy Kepner, Robert F Lucas, Rolf Rabenseifner, and Daisuke Takahashi. The HPC Challenge (HPCC) Benchmark Suite. In *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 213, 2006.

[104] Bradford L. Chamberlain, Steven J. Deitz, David Iten, and Sung-Eun Choi. HPC Challenge Benchmarks in Chapel. Technical report, Cray, Inc., 2009.

[105] CUDA Thrust Library. http://code.google.com/p/thrust.

[106] *NVIDIA CUDA Compute Unified Device Architecture - Programming Guide*, 2011.

[107] Seyong Lee, Seung-Jai Min, and Rudolf Eigenmann. OpenMP to GPGPU: A Compiler Framework for Automatic Translation and Optimization. In *PPoPP '09: Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 101–110, 2009.

[108] Shane Ryoo, Christopher I. Rodrigues, Sara S. Baghsorkhi, Sam S. Stone, David B. Kirk, and Wen-mei W. Hwu. Optimization Principles and Application Performance Evaluation of a Multithreaded GPU using CUDA. In *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 73–82, 2008.

[109] Michael Wolfe. Implementing the PGI Accelerator Model. In *GPGPU '10: Proc. 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, pages 43–50, 2010.

[110] Albert Sidelnik, I-Jui Sung, Wanmin Wu, María Jesús Garzarán, Wen-mei Hwu, Klara Nahrstedt, David Padua, and Sanjay J. Patel. Optimization of Tele-Immersion Codes. In *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, GPGPU-2, pages 85–93, New York, NY, USA, 2009. ACM.

[111] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[112] John E. Stone, James C. Phillips, Peter L. Freddolino, David J. Hardy, Leonardo G. Trabuco, and Klaus Schulten. Accelerating Molecular Modeling Applications with Graphics Processors. *Journal of Computational Chemistry*, 28(16):2618–2640, 2007.

[113] J. Rys, M. Dupuis, and H. F. King. Computation of Electron Repulsion Integrals Using the Rys Quadrature Method. *Journal of Computational Chemistry*, 4(2):154–157, 1983.

[114] Stephen D Landy and Alexander S. Szalay. Bias and Variance of Angular Correlation Functions. *Astrophysical Journal*, 412(1):64–71, 1993.

[115] Isaac Gelado, John E. Stone, Javier Cabezas, Sanjay Patel, Nacho Navarro, and Wen-mei W. Hwu. An Asymmetric Distributed Shared Memory Model for Heterogeneous Parallel Systems. In *Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems*, ASPLOS '10, pages 347–358, New York, NY, USA, 2010. ACM.

[116] Portland Group CUDA-X86. `http://www.pgroup.com/resources/cuda-x86.htm`.

[117] Yonghong Yan, Max Grossman, and Vivek Sarkar. JCUDA: A Programmer-Friendly Interface for Accelerating Java Programs with CUDA. In *Euro-Par '09: Proc. 15th International Euro-Par Conference on Parallel Processing*, pages 887–899, 2009.

[118] Andreas Klöckner, Nicolas Pinto, Yunsup Lee, Bryan C. Catanzaro, Paul Ivanov, and Ahmed Fasih. Pycuda: GPU Run-Time Code Generation for High-Performance Computing. *CoRR*, abs/0911.3456, 2009.

[119] Tianyi David Han and Tarek S. Abdelrahman. hiCUDA: A High-level Directive-based Language for GPU Programming. In *GPGPU-2: Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, pages 52–61, New York, NY, USA, 2009. ACM.

[120] Francois Bodin and Stephane Bihan. Heterogeneous Multicore Parallel Programming For Graphics Processing Units. *Sci. Program.*, 17(4):325–336, 2009.

[121] Michael D. McCool, Kevin Wadleigh, Brent Henderson, and Hsin-Ying Lin. Performance Evaluation of GPUs Using the RapidMind Development Platform. In *SC '06: Proc. 2006 ACM/IEEE conference on Supercomputing*, page 181, 2006.

[122] Alejandro Duran, Eduard Ayguad, Rosa M. Badia, Jess Labarta, Luis Martinell, Xavier Martorell, and Judit Planas. OmpSs: A Proposal for Programming Heterogeneous Multi-Core Architectures. *Parallel Processing Letters*, 21(2):173–193, 2011.

[123] Dave Cunningham, Rajesh Bordawekar, and Vijay Saraswat. GPU Programming in a High Level Language Compiling X10 to CUDA. In *X10'11 Workshop*, 2011.

[124] Yonghong Yan, Jisheng Zhao, Yi Guo, and Vivek Sarkar. Hierarchical Place Trees: A Portable Abstraction for Task Parallelism and Data Movement. In *LCPC*, volume 5898 of *Lecture Notes in Computer Science*, pages 172–187. Springer, 2009.

[125] R.C. Whaley, A. Petitet, and J. Dongarra. Automated Empirical Optimizations of Sofware and the ATLAS Project. *Parallel Computing*, 27(1-2):3–35, 2001.

[126] M. Frigo. A Fast Fourier Transform Compiler. In *PLDI'99 — Conference on Programming Language Design and Implementation*, 1999.

[127] In personal communication with Basilio B. Fraguela, Universidade da Coruña.

[128] Kamen Yotov, Keshav Pingali, and Paul Stodghill. X-Ray: A Tool for Automatic Measurement of Hardware Parameters. In *QEST '05: Proceedings of the Second International Conference on the Quantitative Evaluation of Systems on The Quantitative Evaluation of Systems*, page 168. IEEE Computer Society, 2005.

[129] Francois Broquedis, Jérôme Clet-Ortega, Stéphanie Moreaud, Nathalie Furmento, Brice Goglin, Guillaume Mercier, Samuel Thibault, and Raymond Namyst. hwloc: A Generic Framework for Managing Hardware Affinities in HPC Applications. In *Proceedings of the 2010 18th Euromicro Conference on Parallel, Distributed and Network-based Processing*, PDP '10, pages 180–186, Washington, DC, USA, 2010. IEEE Computer Society.

[130] Rafael H. Saavedra and Alan J. Smith. Analysis of Benchmark Characteristics and Benchmark Performance Prediction. *ACM Trans. Comput. Syst.*, 14(4):344–384, 1996.

[131] Larry McVoy and Carl Staelin. lmbench: Portable Tools for Performance Analysis. In *ATEC '96: Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 23–23. USENIX Association, 1996.

[132] Josep Torrellas, Andrew Tucker, and Anoop Gupta. Evaluating the Performance of Cache-Affinity Scheduling in Shared-Memory Multiprocessors. *J. Parallel Distrib. Comput.*, 24(2):139–151, 1995.

[133] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual*. Number 253669-033US. December 2009.

[134] J. González-Domínguez, G.L. Taboada, B.B. Fraguela, M.J. Martín, and J. Touri no. Servet: A Benchmark Suite for Autotuning on Multicore Clusters. In *24th IEEE International Parallel and Distributed Processing Symposium, IPDPS'10*, page 9 pages, Atlanta, GA, USA, 2010.

Appendix A

Sparse Matrices Representation

The sparse matrices presented here provide a visual representation of the inputs used for the different performance evaluations from Chapters 3 and 4. These matrices and their graphical representations were taken from the University of Florida Sparse Matrix Collection [69].

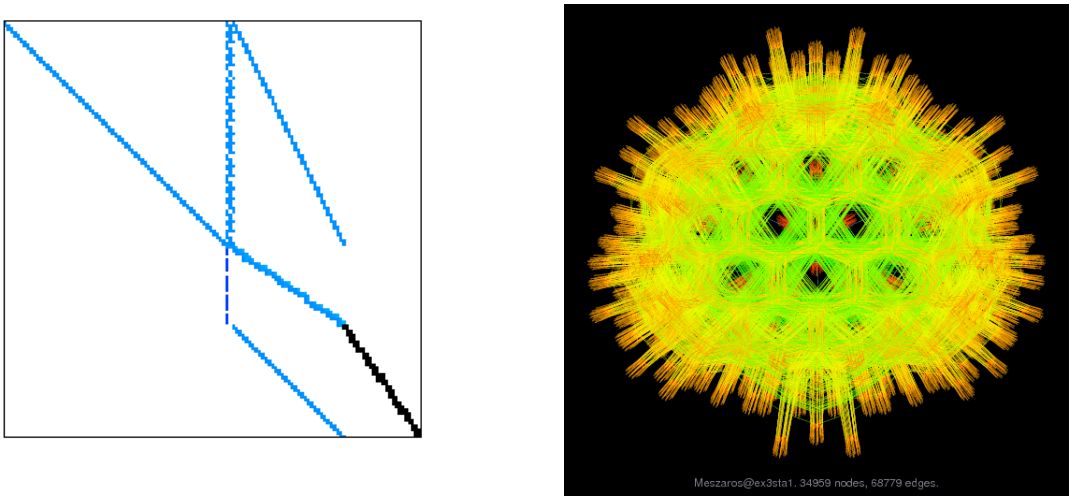


Figure A.1: Matrix: Meszaros/ex3sta1

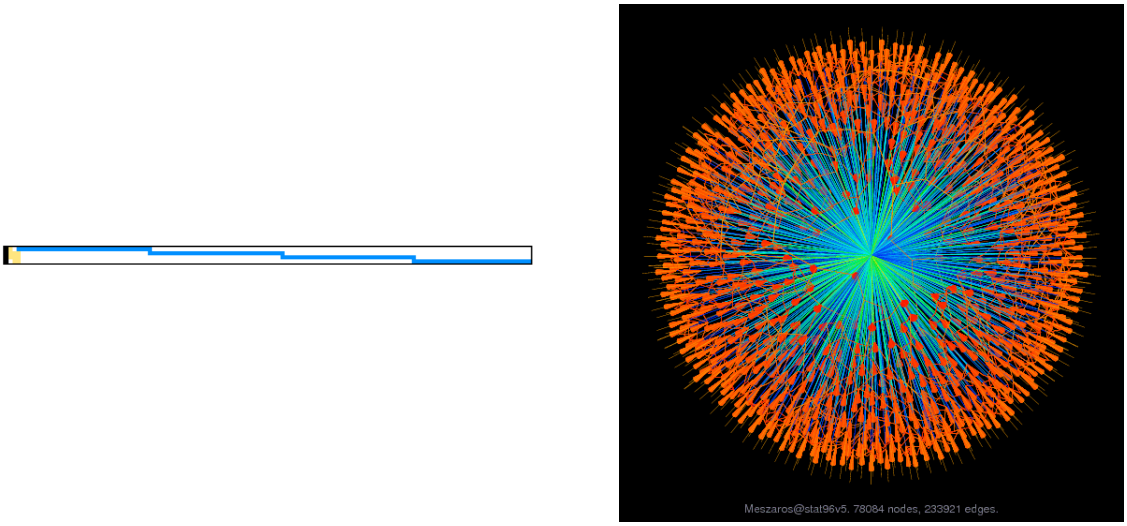


Figure A.2: Matrix: Meszaros/stat96v5

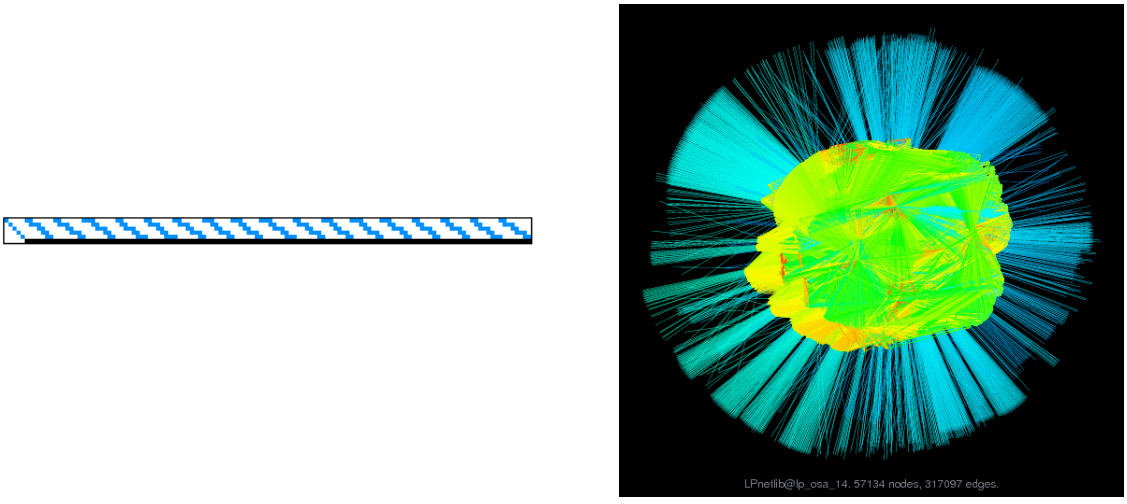


Figure A.3: Matrix: LPnetlib/lp_osa_14

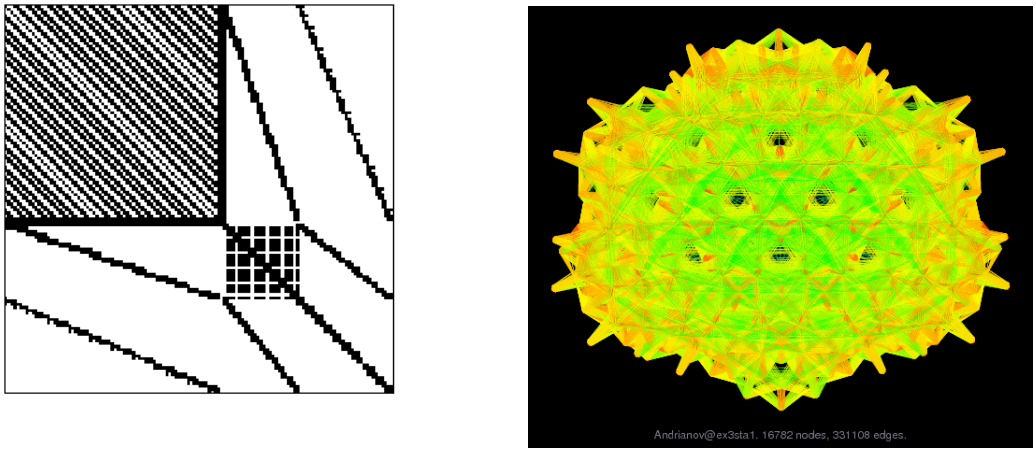


Figure A.4: Matrix: Andrianov/ex3sta1

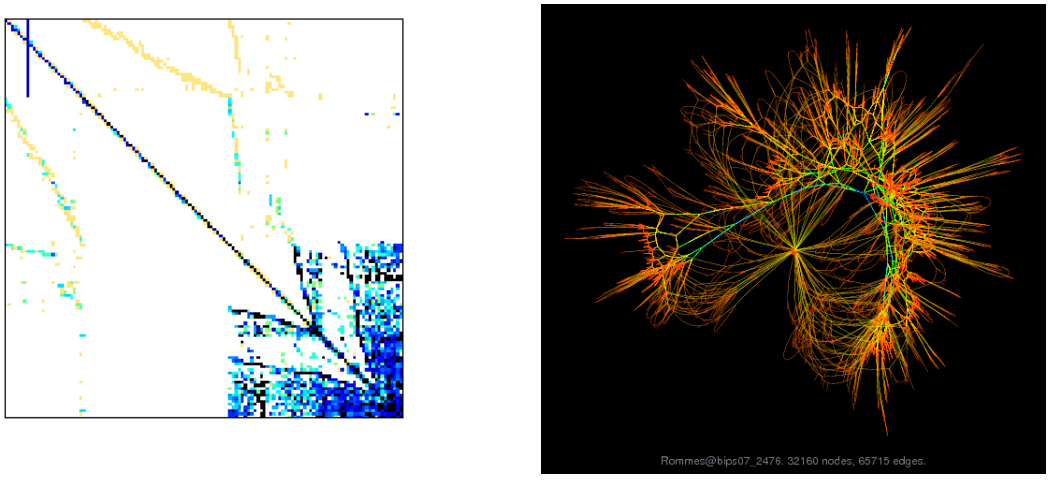


Figure A.5: Matrix: Rommes/bips07_2476