Portable and Productive High-Performance Computing

by

Ekanathan Palamadai Natarajan

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2017

Author . . . . . .
Department of Electrical Engineering and Computer Science
September 16, 2016

Certified by . . . . . .
Alan Edelman
Professor
Thesis Supervisor

Accepted by . . . . . .
Leslie A. Kolodziejski
Chairman, Department Committee on Graduate Students

Portable and Productive High-Performance Computing

by

Ekanathan Palamadai Natarajan

Submitted to the Department of Electrical Engineering and Computer Science
on September 16, 2016, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Abstract

Performance portability of computer programs, and programmer productivity in writing them are key expectations in software engineering. These expectations lead to the following questions:

Can programmers write code once, and execute it at optimal speed on any machine configuration? Can programmers write parallel code to simple models that hide the complex details of parallel programming?

This thesis addresses these questions for certain "classes" of computer programs. It describes "autotuning" techniques that achieve performance portability for serial divide-and-conquer programs, and an abstraction that improves programmer productivity in writing parallel code for a class of programs called "Star".

We present a "pruned-exhaustive" autotuner called ***Ztune*** that optimizes the performance of serial divide-and-conquer programs for a given machine configuration. Whereas the traditional way of autotuning divide-and-conquer programs involves simply coarsening the base case of recursion optimally, Ztune searches for optimal divide-and-conquer trees. Although Ztune, in principle, exhaustively enumerates the search domain, it uses pruning properties that greatly reduce the size of the search domain without significantly sacrificing the quality of the autotuned code. We illustrate how to autotune divide-and-conquer stencil computations using Ztune, and present performance comparisons with state-of-the-art "heuristic" autotuning. Not only does Ztune autotune significantly faster than a heuristic autotuner, the Ztuned programs also run faster on average than their heuristic autotuner tuned counterparts. Surprisingly, for some stencil benchmarks, Ztune actually autotuned faster than the time it takes to execute the stencil computation once.

We introduce the Star class that includes many seemingly different programs like solving symmetric, diagonally-dominant tridiagonal systems, executing "watershed" cuts on graphs, sample sort, fast multipole computations, and all-prefix-sums and its various applications. We present a programming model, which is also called ***Star***, to generate and execute parallel code for the Star class of programs. The Star model abstracts the pattern of computation and interprocessor communication in the Star class of programs, hides low-level parallel programming details, and offers ease of expression, thereby improving programmer productivity in writing parallel code. Besides, we also present parallel algorithms, which offer asymptotic improvements over prior art, for two

programs in the Star class — a ***Trip*** algorithm for solving symmetric, diagonally-dominant tridiagonal systems, and a ***Wasp*** algorithm for executing watershed cuts on graphs. The Star model is implemented in the Julia programming language, and leverages Julia's capabilities in expressing parallelism in code concisely, and in supporting both shared-memory and distributed-memory parallel programming alike.

Thesis Supervisor: Alan Edelman
Title: Professor

To Lakshmi, Shreyas, and Tejas.

Acknowledgments

This section is perhaps the most important part of the thesis. I am deeply grateful to Alan Edelman, my advisor, for being a great teacher and mentor. It is rare to find scientists who are both brilliant and pleasant to work with, and Alan is one of them. I can't thank him enough for introducing me to the Julia language, and for his guidance, support, and inspiration all along the way.

Two other people at MIT played an important role in my graduate student career. I am grateful to Charles Leiserson for teaching me many things about research, and about technical writing. I am grateful to Luca Daniel for inspiring me to pursue a PhD, and for encouraging me to apply to graduate school.

It has been a privilege to have Luca Daniel and Matei Zaharia on my thesis committee. Many thanks to Luca and Matei.

The work in this thesis is the result of collaboration with many other people. Chapter 2 on autotuning stencil computations [55] is joint work with Maryam Mehri Dehnavi, and Charles Leiserson. Chapter 3 on the Star [54] abstraction is joint work with Andreas Noack, and Alan Edelman. Thanks to Maryam for providing a lot of encouragement and support on the autotuning project. Thanks to Andreas for the collaboration on the Star project. Thanks to Payut Pantawongdecha for working with me on autotuning matrix-vector product and matrix multiplication.

Several other people indirectly contributed to this thesis. I am thankful to Jiahao Chen, Shashi Gowda, Tim Kaler, Xiangdong Liang, Tanmay Mohapatra, David Sanders, Viral Shah, Aleksandar Zlateski, and many others who i might have missed mentioning here, for useful conversations on the problems at different times. Many thanks to Janet Fischer and Alicia Duarte at the EECS graduate office, Shirley Entzminger, and Leslie Kolodziejski for help at several occasions. I am grateful to Tim Davis, my former advisor at the University of Florida, for launching me into a scientific career.

I am thankful to Aramco, Intel ISTC, DARPA xdata, and NSF for their generous financial support. The NSF support was provided under the grant DMS-1312831.

Finally, pursuing a PhD would have been impossible if not for my wife Lakshmi and my children — Shreyas and Tejas. I dedicate this thesis to them. Lakshmi has been a source of inspiration and support to me throughout this journey, and Shreyas and Tejas have been tolerant of me spending a lot of time on the computer.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Performance portability of computer programs, and programmer productivity in writing them are key expectations in software engineering. The diversity in the hardware architecture of computers ranging from hand-held devices to supercomputers makes it difficult to guarantee that a program will run at the fastest possible speed on a given computer. Besides the hardware architecture, many other settings like the choice of compiler, the version of the operating system software, etc. affect the performance of a program. Often a programmer spends significant time in "handtuning" her program's performance on a given computer. Handtuning can be error prone, however, and might not necessarily yield optimal performance. More importantly, the optimal performance settings to execute a program on one computer might produce suboptimal performance when the same program is executed on another computer with a slightly different configuration. Hence the idea of "Write code once, and have it run at optimal speed on any computer" seems far from reality.

In a similar vein, the abundance of parallel programming models ranging from simple shared-memory models to complex models for large clusters and graphics processing units makes it difficult to write generic parallel code that will automatically run under any model. Hence the idea of "Write parallel code once, and have it run under any programming model" also seems far from reality.

Writing parallel programs, just for a single parallel programming model, can be a difficult exercise. Though a programmer's primary focus is on devising simple and efficient parallel al-

gorithms to problems, she is often burdened with the task of mastering the parallel programming model and its low level details like distributing work among processors, interprocessor communication, synchronization, and many more. Such details not only act as hurdles to writing parallel code, but also make it cumbersome to express parallelism in code concisely. Besides, many seemingly different parallel programs have the same pattern of computation and communication, and hence it is redundant to write code for such patterns repetitively. Parallel programming constructs [1, 5, 20, 46, 47] that abstract the pattern of computation and interprocessor communication, hide low level programming details, and offer ease of expression, greatly improve programmer productivity in writing parallel programs.

This thesis addresses the performance portability and programmer productivity expectations for certain "classes" of computer programs. It describes "autotuning" techniques that achieve performance portability for serial divide-and-conquer programs, and an abstraction that improves programmer productivity in writing parallel code for a class of programs called "Star".

In the remainder of this chapter, we motivate the problem of autotuning serial divide-and-conquer programs, and the need for efficient and easy-to-use abstractions for the Star class of programs. We conclude the chapter with a summary of our contributions.

1.1 Performance portability

High-performance computer programs like stencil computation [8, 19, 22, 23, 30, 31, 37–39, 41, 51, 52, 57, 61, 62, 68], matrix multiplication, matrix-vector product, and FFT [27, 28] constitute a dominant part of the compute time of many important scientific applications. Hence it is essential that such programs run at optimal speeds on any given computer, be it a hand-held device or a supercomputer. To achieve high performance, these programs are implemented efficiently using algorithms that not only have the best asymptotic runtimes, but also optimally use the computer's memory hierarchy and other resources. In addition, these programs define a set of key constants, whose values can be chosen to produce faster runtimes on a target machine. For example, Figure 1-1 shows a stable merge sort routine defined in stl_algo.h in the C++ standard template library. If

```
1  template<typename _RandomAccessIterator, typename _Compare>
2  void __inplace_stable_sort(_RandomAccessIterator __first,
3                             _RandomAccessIterator __last, _Compare __comp) {
4    if (__last - __first < 15) {
5      std::__insertion_sort(__first, __last, __comp);
6      return;
7    }
8    _RandomAccessIterator __middle = __first + (__last - __first) / 2;
9    std::__inplace_stable_sort(__first, __middle, __comp);
10   std::__inplace_stable_sort(__middle, __last, __comp);
11   std::__merge_without_buffer(__first, __middle, __last,
12                               __middle - __first,
13                               __last - __middle,
14                               __comp);
15 }
```

Figure 1-1: Hand optimized stable merge sort in the C++ standard template library.

the size of the array is less than 15, then the sorting routine switches to insertion sort to avoid "recursive function-call" overhead. The value 15 was probably chosen by the programmer since it gave a faster runtime on the machine where she executed the routine. However, a different value might produce faster runtimes on modern machines with larger cache sizes. Similarly, efficient matrix multiplication involves dividing the given matrices into smaller submatrices such that the multiplication optimally uses the memory hierarchy, vectorization, and other system resources, and results in faster runtimes. The program would define the submatrix sizes as constants so that the programmer can choose values that produce optimal runtimes on a target machine. The programmer can choose values for these constants manually by ***handtuning***, that is, trying out different combinations of values using a trial-and-error strategy, and selecting the values that result in faster runtimes. However, handtuning can be error prone, time consuming, and handtuning the values for multiple constants simultaneously can be a difficult exercise. Besides, the handtuning exer-

cise has to be repeated on each new machine configuration on which the code needs be executed. To complicate the situation, the original programmer may no longer be around to handtune the performance on a new machine configuration. To address these difficulties, the problem of ***autotuning*** [2, 15, 17, 19, 27, 28, 37, 40, 50, 66, 67] — automatically choosing values for key constants in a program to optimize its performance — has emerged as an important technology to achieve performance portability.

Historically, the earliest application-specific autotuners [27, 28, 67] were ***exhaustive***: they simply enumerated the search domain.[1] Since exhaustive enumeration can be time consuming for some applications, later autotuners [2, 15, 17, 19, 37, 40, 50, 66] were ***heuristic***, using heuristics and sophisticated machine-learning methods to find parameter settings that produced good runtimes. More recently, heuristic autotuning frameworks, such as OpenTuner [3], have enabled programmers to build application-specific autotuners quickly. Despite this recent trend towards heuristic autotuning, this thesis explores if exhaustive autotuning with pruning is viable for the domain of serial divide-and-conquer programs.

At a higher level, a divide-and-conquer program executes a fixed recursion. It either divides a given problem (for example, a matrix) into subproblems (submatrices) and recurs, or terminates the recursion and executes a base case function when the size of the problem falls beneath a threshold constant. Choosing suitable values for these threshold constants minimizes recursive function-call overhead, makes optimal use of the computer's memory hierarchy and vector instructions, and executes "application-specific" computations efficiently.

The traditional way of autotuning divide-and-conquer programs is to simply parameterize the threshold constants, and to search for optimal parameter values. In contrast, we present a different autotuning strategy that searches for optimal recursion trees. We define a search domain that generalizes a divide-and-conquer program, where at each step of recursion, a parameter is created that chooses whether to divide a given problem in one of several ways and recur, or execute the base case. Different choices for the parameters at each recursive step results in different recursion

[1] We use the term ***search domain*** rather than the more conventional ***search space*** to avoid confusion with the geometric use of "space" endemic to applications like stencil computations.

trees. We present a pruned-exhaustive autotuner called ***Ztune*** that searches the domain of recursion trees to find the fastest recursion tree that a program can use to execute a given input problem on a given machine. Although Ztune, in principle, exhaustively enumerates the search domain, it uses three ***pruning properties*** — "equivalence", "divide subsumption", and "favored dimension" — which prune the search domain effectively. These properties reduce the autotuning time by orders of magnitude without significantly sacrificing the performance of the tuned program.

We explored autotuning over the two search domains — the domain of values for the threshold constants, and the domain of recursion trees. It turns out that heuristic autotuning over the domain of recursion trees is practically infeasible due to the sheer size of that domain and the inability of heuristic autotuning to leverage the pruning properties. Consequently, we used heuristic autotuning over the domain of values for the threshold constants. We then compared the performance of the pruned-exhaustive Ztune autotuner searching over the domain of recursion trees with that of the heuristic OpenTuner [3] autotuner in tuning the divide-and-conquer stencil algorithm in Pochoir [62] stencil compiler. On tuning ten stencil benchmarks across two machines with different hardware configurations, the Ztuned code ran 5%–12% faster on average, and the OpenTuner tuned code ran from 9% slower to 2% faster on average, than Pochoir's default code. In the best case, the Ztuned code ran 40% faster, and the OpenTuner tuned code ran 33% faster than Pochoir's default code. Whereas the autotuning time of Ztune for each benchmark could be measured in minutes, to achieve comparable results, the autotuning time of OpenTuner was typically measured in hours or days. Surprisingly, for some benchmarks, Ztune actually autotuned faster than the time it takes to perform the stencil computation once.

Thesis Statement 1: *Pruned-exhaustive autotuning is more effective than heuristic autotuning for serial divide-and-conquer stencil computations.*

1.2 Programmer productivity

Historically, the art of developing abstractions [32] focussed simply on sequential or single processor programs. The emergence of multicore and manycore architectures necessitated research on

developing abstractions to make parallel programming easier. Since developing a universal abstraction that works efficiently for all parallel programs is cumbersome, the trend [5, 20, 46, 47, 69] has been to identify patterns of computation and communication in parallel programs, and to develop abstractions for those patterns. The advent of "BigData" applications processing large amounts of data across hundreds or thousands of commodity PCs has spurred the need for parallel abstractions, and has resulted in simple, yet powerful constructs like MapReduce [20]. More recently, programming models like Spark [70] that support a much wider class of applications than MapReduce have been created.

This thesis introduces the "Star" class of computer programs that have a few key properties. Each program in the ***Star*** class

- executes an associative computation over ordered sets of data; and
- can be parallelized by splitting its computation into a sequence of three subcomputations.

Many seemingly different programs fall in the Star class. These include solving symmetric, diagonally-dominant tridiagonal systems [14, 21, 33, 35, 60], executing watershed cuts [16] on graphs, sample sort [10, 26], fast multipole computations [24], and all-prefix-sums and its various applications [9] like radix sort, solving recurrences, and dynamic processor allocation. As an example program in the Star class, consider the all-prefix-sums program, also referred to as "scan" for short. The ***scan*** operation takes as input a binary associative operator $\oplus$, and an ordered set of n elements $[a_1, a_2, \ldots, a_n]$, and returns the ordered set $[a_1, a_1 \oplus a_2, \ldots, a_1 \oplus a_2 \oplus \ldots \oplus a_n]$. For example, if $\oplus$ is addition, then scan computes the "cumulative sum" of the elements in the ordered set. The cumulative sum operation can be parallelized using a sequence of three subcomputations namely — SUM, EXCLUSIVE-CUMSUM, and CUMSUM— which we will define in Section 3.2. Besides, the parallel Star programs have the same pattern of computation and interprocessor communication. Given the reasonably large size of the Star class, a model that abstracts the pattern of computation and communication in the parallel Star programs, and automatically generates and executes parallel code will greatly improve programmer productivity in writing such programs.

At a higher level, we are interested in a model that abstracts the different steps in a parallel Star program called ***three-phase computation***, whose pseudocode is shown in Figure 1-2. Pro-

```
THREE-PHASE-COMPUTATION(A)
1  Partition array A into a sequence of p subarrays, and let A_1, ..., A_p be
   the "references" of the subarrays.
2  let R be an array of size p.
   // Execute the first subcomputation on subarray A_i.
3  parallel for i = 1 to p
4      R[i] = FIRST-PHASE(A_i)
   // Execute the second subcomputation on the elements of R.
5  SECOND-PHASE(R)
   // Execute the third subcomputation on subarray A_i using R[i] as the
   // initial value.
6  parallel for i = 1 to p
7      THIRD-PHASE(R[i], A_i)
```

Figure 1-2: Pseudocode for a three-phase computation. Procedure THREE-PHASE-COMPUTATION takes as input an array A. The definitions of the FIRST-PHASE, SECOND-PHASE, and THIRD-PHASE procedures, which are highlighted in blue, are application-specific and are not shown.

cedure THREE-PHASE-COMPUTATION operates on an array A, which represents an ordered set, partitioned into a sequence of p subarrays. It consists of three phases, namely "first", "second", and "third". In the first phase shown in lines 3–4, procedure FIRST-PHASE, which represents the first subcomputation, is executed in parallel on the processors where the subarrays A_i are stored. The results of the subcomputation are collected in an array R, which is also an ordered set, such that $R[i]$ is the result from subarray A_i. In the second phase shown in line 5, procedure SECOND-PHASE, which represents the second subcomputation, is executed locally on array R. In the third phase shown in lines 6–7, procedure THIRD-PHASE, which represents the third subcomputation, is executed in parallel on the processors where the subarrays A_i are stored. The execution of a parallel Star program using three-phase computation is illustrated in Figure 3-2.

The computations and communication in THREE-PHASE-COMPUTATION follow a pattern, which can be abstracted in a model. The three subcomputations have the following properties:

- The subcomputations follow a sequence, that is a parallel first phase followed by a serial second phase, followed by a parallel third phase;
- The second subcomputation is noncommutative;

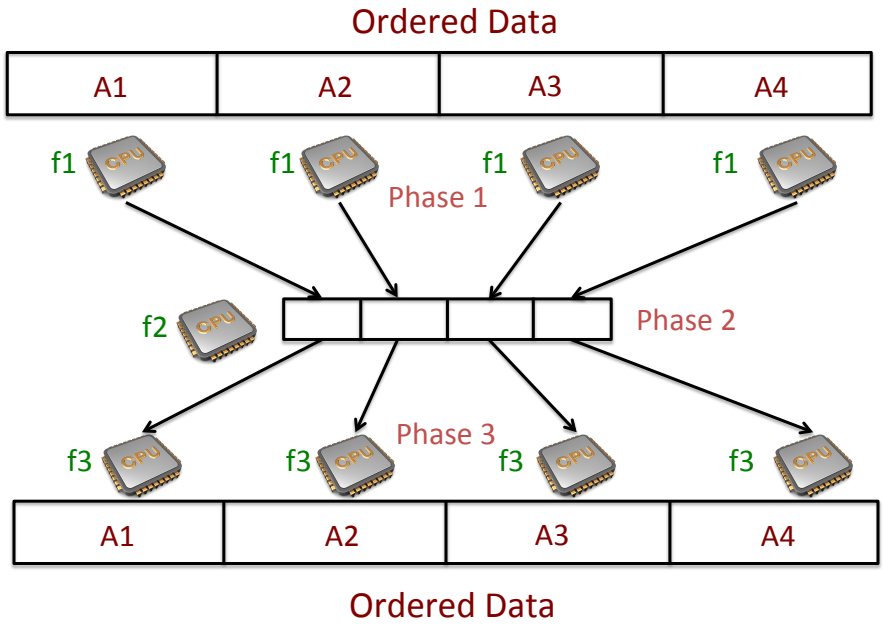


Figure 1-3: Parallel execution of a Star program. The three subcomputations are indicated as f1, f2, and f3. The data partitions are marked as A1, A2, A3, and A4. The partitions at the bottom are the same as those at the top, and have been duplicated to illutrate the flow of data.

- Communication happens only in the first and third phases; and
- Communication is asymptotically smaller than computation, that is, the size of data communicated is asymptotically smaller than the total time spent in the three subcomputations.

Communication in the first phase at line 4 involves sending a reference of subarray A_i to a processor, and receiving the result $R[i]$ back. Communication in the third phase at line 7 involves sending the value $R[i]$ and the reference to a processor. The inputs to THREE-PHASE-COMPUTATION, the definitions of procedures FIRST-PHASE, SECOND-PHASE and THIRD-PHASE, and their inputs and outputs change with the application. For example, computing the cumulative sum of an array and solving a tridiagonal system will have different inputs to THREE-PHASE-COMPUTATION, and different definitions for procedures FIRST-PHASE, SECOND-PHASE and THIRD-PHASE. We are

interested in a model that takes as inputs user-defined procedures and their corresponding arguments for the three phases, and generates and executes THREE-PHASE-COMPUTATION code for a Star program. Such a model can hide the parallel programming details in the first and third phases, and offer a simple interface to the user to program.

Can the MapReduce model be used to simulate THREE-PHASE-COMPUTATION? Given its simplicity, it is tempting to push the limits of MapReduce to parallelize a Star program. It turns out that MapReduce can indeed be used to simulate THREE-PHASE-COMPUTATION, but at the cost of asymptotically higher than optimal computation and communication. For example, consider the scan program from the Star class. It is circuitous to express scan in MapReduce, since MapReduce is oblivious to the order of data, whereas scan being noncommutative, operates over an ordered set of data. We defer a step-by-step MapReduce solution for THREE-PHASE-COMPUTATION to Section 3.5, and briefly highlight the complexity in writing scan using MapReduce here. A naive MapReduce program to perform scan on an ordered set $[a_1, a_2, \ldots, a_n]$ using an operator $\oplus$, can be defined using the following signatures for the map and reduce functions. For $1 \leq i \leq n$,

$$\begin{array}{rcll} map & : & (i, a_i) & \rightarrow [(i, a_i), (i+1, a_i), \ldots, (n, a_i)] \\ reduce & : & (i, [a_1, a_2, \ldots, a_i]) & \rightarrow (i, a_1 \oplus a_2 \oplus \ldots \oplus a_i) \end{array}$$

The naive program is, of course, inefficient since it performs $\Theta(n^2)$ "$\oplus$" operations, incurs $\Theta(n^2)$ communication, and takes $\Theta(n)$ time given n processors. In contrast, a serial scan program takes only $\Theta(n)$ "$\oplus$" operations, and $\Theta(n)$ time using just one processor. A more efficient but convoluted MapReduce program can be devised, where the map function takes as input a pair (i, a_i), $1 \leq i \leq n$, and emits a tuple $(1, (i, a_i))$. The special key "1" is to make sure that the same MapReduce "worker" receives all the (i, a_i) pairs. Since the list of (i, a_i) pairs received by the MapReduce worker might not be sorted in an increasing order of the indices i, the reduce function first sorts the list on i. Then, it performs a scan on the values a_i of the sorted list. Using a standard sorting algorithm, this solution takes $\Theta(n \lg n)$ time, and $\Theta(n)$ communication, which is still inefficient.

MapReduce also incurs higher than necessary communication overhead for THREE-PHASE-COMPUTATION. Since procedures FIRST-PHASE and THIRD-PHASE are executed locally on the

processors where each subarray is stored, the elements of the subarrays are not communicated over the network. Then the only communication overhead that THREE-PHASE-COMPUTATION incurs is in receiving the results $R[i]$ from each processor in line 4, and in sending the values $R[i]$ to each processor in line 7. If $R[i]$ is w words long, then the total communication overhead of THREE-PHASE-COMPUTATION is $\Theta(pw)$. In contrast, a MapReduce version of THREE-PHASE-COMPUTATION incurs $\Theta(p^2w)$ communication overhead, since MapReduce is oblivious to the order of the subarrays, and consequently needs to broadcast the entire R array to each processor during the third phase in line 7. This quadratic growth in communication with the number of processors affects scalability for a Star program. For example, THREE-PHASE-COMPUTATION can perform a cumulative sum operation on an array of n elements using p processors in $\Theta(n/p+p)$ time, and achieve the smallest runtime of $\Theta(\sqrt{n})$ when $p=\sqrt{n}$. But when $p=\sqrt{n}$, MapReduce's total communication grows to $\Theta(nw) = \Theta(n)$, where the word length $w=\Theta(1)$ for cumulative sum. Equivalently, MapReduce would communicate the entire input array when $p=\sqrt{n}$.

Taking inspiration from the need for an efficient and easy-to-express model for THREE-PHASE-COMPUTATION, we designed a simple abstraction that is also called ***Star***. Unlike MapReduce, which is oblivious to the order of data, Star is a restricted model where the global order of data is known apriori. Hence Star avoids unnecessary overheads that MapReduce incurs in simulating THREE-PHASE-COMPUTATION. Users specify application-specific functions and their corresponding arguments for the three phases as inputs to Star, which composes them to generate and execute THREE-PHASE-COMPUTATION code. We implemented Star in the Julia [6] programming language, leveraging Julia's capabilities in expressing parallelism in code concisely, and in supporting both shared-memory and distributed-memory parallel programming alike.

Thesis Statement 2: *Specialized abstractions like Star are more effective for the Star class of programs than generic abstractions like MapReduce.*

1.3 Contributions

This dissertation shows how to achieve performance portability for serial divide-and-conquer programs, and how to improve programmer productivity in writing parallel code for the Star class of programs. In specific, we make the following contributions:

- We describe Ztune, a pruned-exhaustive autotuner for serial divide-and-conquer programs, and demonstrate its efficacy in tuning divide-and-conquer stencil computations.
- We identify three pruning properties — equivalence, divide subsumption, and favored dimension — which improve the runtime of Ztune significantly, and we document the advantage that each property accrues to Ztune.
- We demonstrate empirically that Ztune can produce faster divide-and-conquer stencil codes with less tuning time than state-of-the-art heuristic autotuning.
- We present a Star programming model that abstracts the pattern of computation and communication in the Star class of programs, and automatically generates and executes parallel code, thereby improving programmer productivity in writing such programs.
- We present parallel algorithms, which achieve asymptotically better time and/or communication complexities than prior art, for two programs in the Star class:
  - A Trip algorithm to solve symmetric, diagonally-dominant tridiagonal systems. Given p processors to solve a system of n unknowns, Trip takes $\Theta(n/p+p)$ time and incurs $\Theta(p)$ communication.
  - A Wasp algorithm to execute watershed cuts on images and graphs. Given p processors to perform watershed cuts on a 2D square image with n pixels, Wasp incurs $O(n/p+\sqrt{np}\ \alpha(\sqrt{np}))$ time and $\Theta(\sqrt{np})$ communication, where α is the slowly growing inverse Ackermann's function.
- We show that MapReduce incurs asymptotically higher overheads in computation and communication in parallelizing programs in the Star class.

1.4 Outline

The remainder of the thesis is organized in two parts. Part one is on performance portability of divide-and-conquer programs, and incudes Chapter 2. Part two is on programmer productivity in writing parallel code for the Star class of programs, and includes Chapter 3. Chapter 2 explores autotuning strategies for serial, divide-and-conquer stencil computations. It describes the pruned-exhaustive Ztune algorithm and its pruning properties, and compares the efficicacy of Ztune with that of the heuristic OpenTuner autotuner in tuning different stencil benchmarks. Chapter 3 describes the Star programming model, and its applications including the Trip and Wasp algorithms. We conclude and discuss possible areas for future work in Chapter 4.

Chapter 2

Autotuning stencil computations

This chapter explores autotuning strategies for serial divide-and-conquer stencil computations, comparing the efficacy of traditional heuristic autotuning with that of pruned-exhaustive autotuning. We present the pruned-exhaustive Ztune autotuner that searches for optimal divide-and-conquer trees for stencil computations. Ztune uses three pruning properties for tuning stencil codes — "space-time equivalence", "divide subsumption", and "favored dimension" — that greatly reduce the size of the search domain without significantly sacrificing the quality of the autotuned stencil code.

2.1 Introduction

Many compute-intensive scientific applications perform stencil computations. A ***stencil*** defines the value of a grid point in a d-dimensional spatial grid at time t as a function of the values of neighboring grid points at recent times before t. A ***stencil computation*** [8, 19, 22, 23, 30, 31, 37–39, 41, 44, 45, 51, 52, 57, 61, 62, 68] involves computing the stencil at each grid point for several time steps. Stencil computations are conceptually simple to implement using nested loops, but naive looping implementations suffer from poor cache performance. ***Tiling*** [19, 58, 59] can enhance performance, although tiling can make the code overly specific to a particular cache size, decreasing portability across machines. Cache-oblivious [29] divide-and-conquer stencil codes based on Frigo

and Strumpen's trapezoidal decomposition [30, 31] are robust to changes in cache size and provide an asymptotic improvement in cache efficiency over looping implementations.

Because stencil computations constitute a dominant part of the compute time of many important scientific applications, autotuning [2, 15, 17, 19, 27, 28, 37, 40, 50, 66, 67] has emerged as an important technology for stencil computations (see, for example, [15, 19, 37, 40]). Autotuning is especially valuable whenever a stencil code must be ported to a machine with a different hardware configuration from where the code was originally developed. Without proper tuning, significant performance can be lost. Because the original programmer may no longer be around to tune the adjustable constants in the code, it makes sense to automate the tuning process.

The introduction of this thesis described a brief history of exhaustive and heuristic autotuning, and the recent trend towards building heuristic autotuners for different application domains. Despite this trend, we examine if exhaustive autotuning with pruning is viable for the domain of divide-and-conquer stencil computations. We describe a pruned-exhaustive autotuning strategy, and compares its efficacy with state-of-the-art heuristic autotuning in optimizing the performance of a serial divide-and-conquer stencil code based on the TRAP algorithm used in Pochoir stencil compiler [62].

Given the attention that parallelism receives in high performance computing, one may wonder whether optimizing the performance of serial codes is important. However, there are several reasons that favor high performing serial codes. Efficient serial codes tend to be competitive with their parallel counterparts. As an analogy, [42] shows that GraphChi, a disk-based system using just a single PC, can perform large scale graph computations reasonably well when compared with sophisticated distributed systems using several computers. Hence, parallel codes are typically benchmarked against the best serial codes. Besides, efficient serial codes often lead to efficient parallel implementations. Serial codes are simple to autotune, since they do not face a host of issues like memory bandwidth saturation and communication overhead, which occur in parallel execution. Further, many autotuning techniques for serial codes can be applied for tuning parallel codes.

Since our focus is on autotuning serial codes, we modified TRAP to disable parallelism. We

				Nehalem		*Ivy Bridge*	
Benchmark	*Dims*	*Grid size*	*Time steps*	*OpenTuner*	*Ztune*	*OpenTuner*	*Ztune*
APOP	1	$2,000,000$	524,288	0.98	0.98	1.32	0.96
Heat1	2	1000×2000	512	0.93	0.94	0.92	0.92
Heat2	2p	$100 \times 20,000$	8192	0.84	0.88	0.84	0.89
Life	2p	2000×3000	1024	0.97	0.97	0.99	0.99
Heat3	2p	4000×4000	1024	0.94	0.96	0.89	0.91
Heat4	2p	4096×4096	512	1.00	1.00	0.67	0.60
Heat5	2p	$10,000 \times 10,000$	4096	1.16	0.92	1.00	0.87
LBM	3	$100 \times 100 \times 130$	64	0.84	0.98	0.93	0.97
Wave	3	$1000 \times 1000 \times 1000$	64	3.57	0.96	1.45	0.88
Heat6	4	$70 \times 70 \times 70 \times 70$	32	0.98	0.90	0.99	0.87
Geometric mean				1.09	0.95	0.98	0.88

Table 2.1: Performance of autotuned TRAP codes relative to Pochoir's default hand-tuned TRAP code on ten stencil benchmarks. The reported numbers are the better of two runs. OpenTuner was run 16 hours to autotune each benchmark, whereas Ztune took its natural time: less than an hour for Wave and at most a few minutes for the others. The header *Dims* indicates the number of spatial dimensions of the grid. The stencils are periodic if the *Dims* column contains a "p". The header *Grid size* indicates the sizes of the spatial dimensions of the grid. The header *Time steps* indicates the size of the time dimension. The benchmarks are sorted first by the number of spatial dimensions and second by spatial volume. The header *OpenTuner* indicates the ratio of the runtime of OpenTuner tuned TRAP code to the runtime of Pochoir's TRAP code. The header *Ztune* indicates the ratio of the runtime of Ztuned TRAP code to the runtime of Pochoir's TRAP code. A lower ratio indicates that the autotuned code runs faster than Pochoir's code. The last row displays the geometric means of the ratios. The header *Nehalem* gives ratios on the machine, on which Pochoir was developed. The header *Ivy Bridge* gives ratios on a relatively modern machine that supports AVX instructions. The detailed specs of the machines are shown in Figure 2.2.

also disabled the "hyperspace cuts," which enhance the parallelism of TRAP, and replaced them with sequential cuts, as in the original algorithms due to Frigo and Strumpen [30,31]. The modified code performs equivalently to Pochoir's original TRAP code when run serially. For the rest of the thesis, we will use the term TRAP to refer to the serial version of the TRAP code.

At a higher level, the TRAP code executes a fixed recursion, dividing a given problem into subproblems, or terminating the recursion and executing a base case kernel function when the size of the problem falls beneath a threshold constant. The actual code is more complicated in that it uses two different kernel functions, a faster kernel at the interior of the grid, and a slower kernel at the boundaries of the grid that checks if memory accesses fall off the grid. Hence, the code uses two

	Nehalem	*Ivy Bridge*	*Haswell*	*Opteron*
Manufacturer	Intel	Intel	Intel	AMD
CPU	Xeon X5650	Xeon E5-2695 v2	Xeon E5-2666 v3	Opteron 6376
Clock	2.66 GHz	2.4 GHz	2.90 GHz	2.3 GHz
Hyperthreading	Disabled	Enabled	Enabled	Enabled
Turbo Boost	Disabled	Disabled	Disabled	Enabled
Processor cores	12	24	18	32
Sockets	2	2	2	4
L1 data cache/core	32 KB	32 KB	32 KB	16 KB
L2 cache/core	256 KB	256 KB	256 KB	2 MB
L3 cache/socket	12 MB	30 MB	25 MB	6 MB
DRAM	48 GB DDR3	128 GB DDR3	58 GB DDR3	256 GB DDR3
Compiler	ICC 13.1.1	ICC 13.1.1	ICC 13.1.1	ICC 13.1.1
Operating system kernel	Linux 3.13.0	Linux 3.13.0	Linux 4.1.10	Linux 2.6.32
Advanced Vector Extensions	No	Yes	Yes	Yes

Table 2.2: Specifications of the machines used for benchmarking. We disabled Turbo Boost, where possible, to enhance the reliability of time measurements.

different threshold constants, one for problems that lie completely in the interior, and the other for problems that impinge on the boundary. These threshold constants along different "dimensions" of the grid were determined by trial-and-error hand tuning by the authors of Pochoir. Handtuning the values for the threshold constants helps avoid recursive function-call overhead, optimally use the computer's memory hierarchy, and handle stencil computations at the boundaries efficiently.

The traditional way of autotuning TRAP is to simply parameterize the threshold constants, and to search for optimal parameter values over the domain of possible base-case sizes called the ***base domain***. In contrast, we present a different autotuning strategy that gains insight from how TRAP operates. We define a search domain called ***choice domain*** that generalizes TRAP, where at each problem in the recursion tree, a parameter is created that chooses whether to divide the problem in one of several ways and recur, or execute the base case. We present an exhaustive autotuner called ***Ztune*** that searches the choice domain to find the fastest recursion tree for stencil computation. Although Ztune, in principle, exhaustively enumerates the choice domain, it uses three pruning properties that prune the choice domain effectively. These properties reduce the autotuning time by orders of magnitude without significantly sacrificing the performance of the tuned stencil code.

The sheer number of choice parameters in the choice domain, however, renders naive heuristic search over that domain infeasible. Consequently we had to resort to heuristic search over the base domain. To that effect, we used the state-of-the-art OpenTuner framework [3] to perform heuristic search over the the base domain, and find optimal values for the threshold constants. OpenTuner takes as input, the parameterized threshold constants and the the base domain specification for each parameter. It then finds optimal values for these parameters by running the TRAP code with different parameter settings from the base domain, while simultaneously pruning the base domain using heuristics and machine-learning based algorithms.

Figure 2.1 compares the performance of autotuned TRAP codes with that of Pochoir's default TRAP code using the hand-tuned threshold constants, on ten stencil benchmarks on two machines, whose specifications are shown in Figure 2.2. The benchmark suite includes American put stock-option pricing (APOP) [36]; a heat equation [25] on a 2D grid (Heat1), a 2D torus (Heat2, Heat3, Heat4, and Heat5), and a 4D grid (Heat6); Conway's game of Life (Life) [34]; the lattice Boltzmann method (LBM) [48]; and a 3D finite-difference wave equation (Wave) [49]. For this experiment, we ran Ztune until it completed, which took less than an hour for any benchmark, and typically less than 7 minutes. Since OpenTuner took several days to completion, we ran OpenTuner for 16 hours (a nightly run) and recorded the results.

Compared with Pochoir's hand-optimized code across the two machines, the Ztuned code is on average[1] 5%–12% faster, and the OpenTuner tuned code is on average 9% slower to 2% faster. On the Nehalem machine, where Pochoir was developed, the Ztuned code is on average 5% faster. Supporting the contention that tuning attains even more importance when porting code, on the more modern Ivy Bridge machine, which has different hardware configuration from Nehalem, the Ztuned code is on average 12% faster. Moreover, Ztune generally produces a faster tuned code than OpenTuner. In particular, the Ztuned code is on average 11%–15% faster than the OpenTuner tuned code across the two machines.

Figure 2.3 evaluates the performance of Ztuned code relative to Pochoir's code on two more

[1] All averages of ratios in this thesis are geometric means.

Benchmark	*Haswell*	*Opteron*	*Benchmark*	*Haswell*	*Opteron*
APOP	0.95	0.94	Heat4	0.68	0.71
Heat1	0.94	0.93	Heat5	0.86	0.90
Heat2	0.88	0.88	LBM	0.98	0.99
Life	1.00	1.01	Wave	0.85	0.98
Heat3	0.87	0.96	Heat6	0.90	0.93

Table 2.3: Performance of Ztuned TRAP codes relative to Pochoir's default hand-tuned TRAP code on the Haswell and Opteron machines. The headers *Haswell* and *Opteron* indicate the ratio of the runtime of Ztuned TRAP code to the runtime of Pochoir's TRAP code on those machines respectively. A lower ratio indicates that Ztuned code runs faster than Pochoir's code. The geometric mean of the ratios on Haswell is 0.89, and that on Opteron is 0.92. The reported ratios are the better of two runs.

machines with different architectures, namely Haswell and Opteron, whose specifications are shown in Figure 2.2. The Ztuned code is on average 8% faster on Opteron, and 11% faster on Haswell, than Pochoir's code.

The scenario where a scientist repeats the stencil computation on the same grid several times is common. Since autotuning time can be amortized over several runs of stencil computation, it makes sense to autotune the stencil code for a given grid and for a given number of time steps. For cases where the number of time steps is significantly larger than the spatial dimensions of the grid, the pruning properties reduce the tuning problem in practice to one where the number of time steps is proportional to the spatial dimensions, thereby saving considerable autotuning time. Even for our benchmarks, where the spatial and time dimensions are proportional to each other, the autotuning time of Ztune is at most a few multiples of the runtime of the tuned code, and, surprisingly, sometimes the autotuning time is less than the runtime.

One might wonder if the performance of Ztuned divide-and-conquer TRAP code is comparable to that of autotuned tiling-based stencil code. Though making such a comparison is not the focus of this thesis, we examined if the Ztuned TRAP code is indeed competitive. Figure 2.4 compares the runtime of Ztuned TRAP code with the runtimes of autotuned tiling-based codes in Pluto [12, 13] and Patus [15]. We ran the comparison on a few Heat benchmarks, which could be easily specified in Pluto and Patus. Since Pluto recommends using a more recent version of Intel compiler, we used ICC 15.0.6 to compile all the 3 autotuners, and used the same compiler flags. Pluto's built-in

	Tuned runtimes			*Tuning quality*	
Benchmark	*Ztune*	*Pluto*	*Patus*	*Ztune*	*Pluto*
Heat1	1.24	0.93	6.49	0.93	0.88
Heat2	35.91	39.18	—	0.86	0.88
Heat3	17.23	44.10	—	0.87	0.87
Heat4	9.99	23.51	—	0.65	0.77

Table 2.4: Comparison of the tuned runtimes (in seconds) of stencil codes under different autotuners, and "tuning quality" of the autotuners on 4 Heat benchmarks, on the Haswell machine. The headers *Ztune*, *Pluto*, and *Patus* under *Tuned runtimes* indicate the runtimes of the stencil codes autotuned with Ztune, Pluto's built-in autotuner, and Patus respectively. A dash sign (—) indicates that the benchmark could not be specified in Patus since it is periodic, and hence the runtime is unknown. The reported numbers are the better of two runs. The header *Ztune* under *Tuning quality* indicates the ratio of the runtime of Ztuned TRAP code to the runtime of Pochoir's TRAP code. Similarly, the header *Pluto* under *Tuning quality* indicates the ratio of the runtime of Pluto's autotuned tiled code to the runtime of Pluto's default tiled code. A lower ratio indicates that the autotuned code runs faster.

autotuner searches a fixed set of tile sizes to find an optimal tile size. Figure 2.4 reports Pluto's best runtime from among different settings like "tiled" and "lbpar". Since Patus doesn't support periodic boundary conditions, it couldn't be used to autotune the periodic benchmarks Heat2, Heat3, and Heat4. The slow runtime of the Patus tuned Heat1 benchmark is probably due to the fact that it doesn't autotune for temporal blocking. Figure 2.4 also compares the "tuning quality", which is the ratio of the runtime of the autotuned code to the runtime of the default code, of Ztune and Pluto. Since Patus doesn't have a default code to run, we couldn't report on its tuning quality. Figure 2.4 indicates that divide-and-conquer based TRAP codes and Ztune are competitive with their tiling based counterparts for the benchmarks considered. It would be premature to conclude that one strategy is faster than the other, however, since this comparison is not exhaustive and not the focus of this thesis. Previous work [11, 18] describes experiments where tiling is faster than divide-and-conquer strategies. We thank Uday Bondhugula for graciously clarifying all our questions about autotuning the benchmarks in Pluto, and Matthias Christen for helping with Patus related queries.

The rest of this chapter is organized as follows. Section 2.2 reviews the serial TRAP algorithm and the "planned" TRAPPLE algorithm, which is optimized by Ztune. Section 2.3 describes

the Ztune algorithm. Section 2.4 provides an overview of the three pruning properties, reports on the performance of Ztune on varying grid sizes, and discusses the effects of noise in autotuning. Sections 2.5, 2.6, and 2.7 describe each of the pruning properties in detail. Section 2.5 shows theoretical upper bounds for memory consumption and tuning time of Ztune under space-time equivalence. Section 2.8 describes OpenTuner, and reports on relative comparisons between Ztuned and OpenTuner tuned codes, when OpenTuner is run a few multiples of time longer than Ztune. Section 2.9 describes related work, and Section 2.10 offers some concluding remarks.

2.2 The Trap and Trapple algorithms

This section describes the TRAP algorithm used by the Pochoir stencil compiler [62], and how we can adapt it into a "planned" algorithm called TRAPPLE suitable for autotuning by Ztune. This section also defines a "plan-finding" problem for autotuning divide-and-conquer stencil codes.

Although the details of TRAP are generally unimportant for the discussion of autotuning, it is helpful to understand a little about it. TRAP operates recursively on regions of a space-time grid called ***hypertrapezoids***, or simply ***zoids*** [62] for short. A 3-dimensional zoid $Z \subseteq \mathbf{N} \times \mathbf{Z}^2$, can be specified by two time coordinates $ta, tb \in \mathbf{N}$, two x_1-coordinates $xa_1, xb_1 \in \mathbf{Z}$, two x_1-slopes[2] $dxa_1, dxb_1 \in \mathbf{Z}$, two x_2-coordinates $xa_2, xb_2 \in \mathbf{Z}$, and two x_2-slopes $dxa_2, dxb_2 \in \mathbf{Z}$. The zoid defined by these parameters is given by

$$\begin{aligned} Z &= \text{zoid}(ta, tb, xa_1, xb_1, dxa_1, dxb_1, xa_2, xb_2, dxa_2, dxb_2) \\ &= \{(t, x_1, x_2) \in \mathbf{N} \times \mathbf{Z}^2 : ta \le t < tb \ ; \\ &\qquad xa_1 + dxa_1(t - ta) \le x_1 < xb_1 + dxb_1(t - ta) \ ; \text{ and} \\ &\qquad xa_2 + dxa_2(t - ta) \le x_2 < xb_2 + dxb_2(t - ta)\} \ . \end{aligned}$$

[2] Technically, inverse slopes, but we follow the terminology of [30].

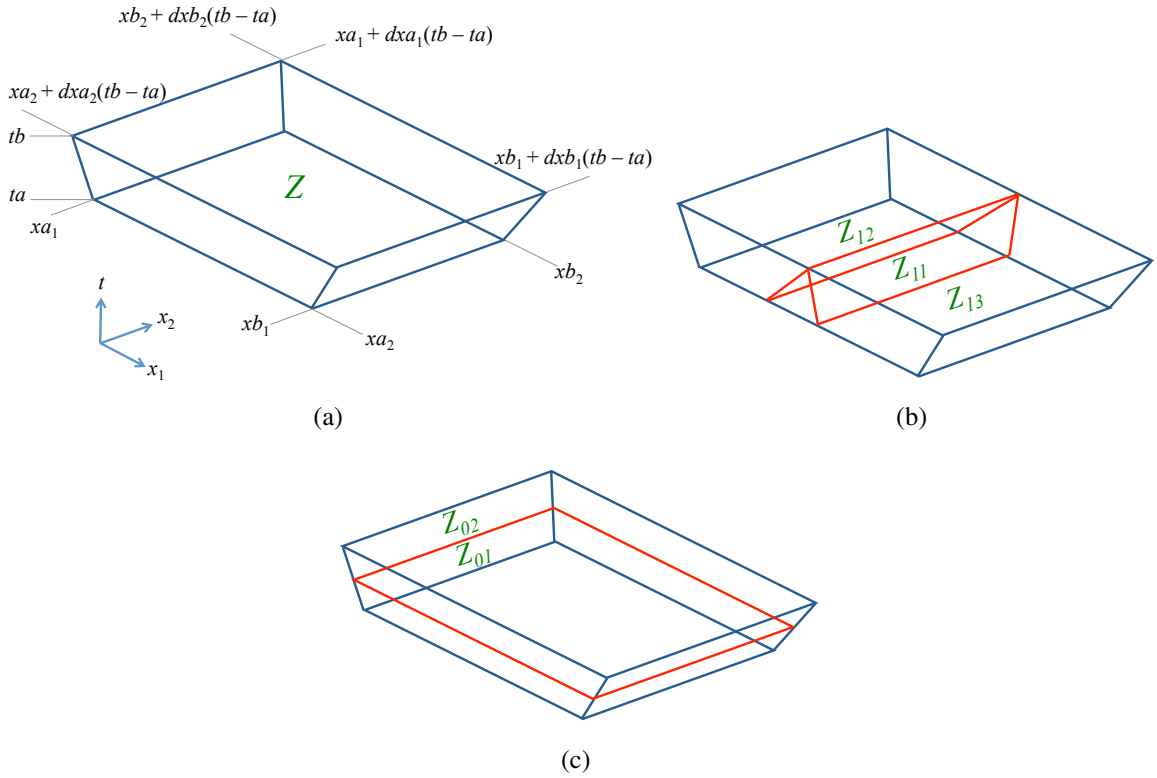


Figure 2-1: Dividing a 3-dimensional zoid with space and time cuts. (a) A zoid $Z = \text{zoid}(ta, tb, xa_1, xb_1, dxa_1, dxb_1, xa_2, xb_2, dxa_2, dxb_2)$. (b) An x_1-cut of Z trisects the zoid into three side-by-side subzoids Z_{11}, Z_{12} and Z_{13}. The stencil dependencies require that Z_{11} be processed before Z_{12} and Z_{13}. (c) A time cut of Z bisects the zoid into bottom and top subzoids Z_{01} and Z_{02}, respectively. The stencil dependencies require that Z_{01} be processed before Z_{02}.

This definition can be straightforwardly extended to a $(d+1)$-dimensional zoid spanning d spatial dimensions and the time dimension. As an example, a 3-dimensional zoid Z is shown in Figure 2-1(a). The ***height*** of Z is given by $\text{height}(Z) = tb - ta$. The ***width*** of Z along the x_1-dimension is the maximum of its two lengths along the x_1-dimension, that is, $\text{width}(Z,1) = max(xb_1 - xa_1, (xb_1 + dxb_1(tb - ta)) - (xa_1 + dxa_1(tb - ta)))$. The width along the x_2-dimension is defined similarly. The ***spatial volume*** of Z is the product of its widths along all spatial dimensions.

2.2.1 The basic algorithm

Figure 2-2 shows the pseudocode for TRAP operating on a 3-dimensional zoid. For didactic purposes, we have abstracted away many details, which are well described in [30, 31, 62]. TRAP

```
TRAP(Z)
 1  h = height(Z)
 2  if h < thresh_0 // base case
 3      for each (t, x_1, x_2) ∈ Z
 4          KERNEL(t, x_1, x_2)
 5  else   // recursive case
 6      w_1 = width(Z, 1)
 7      w_2 = width(Z, 2)
 8      if w_1 ≥ max(thresh_1, 4σ_1 h) // x_1-cut
 9          trisect Z with an x_1-cut into subzoids Z_11, Z_12, and Z_13
10          TRAP(Z_11) ; TRAP(Z_12) ; TRAP(Z_13)
11      elseif w_2 ≥ max(thresh_2, 4σ_2 h) // x_2-cut
12          trisect Z with an x_2-cut into subzoids Z_21, Z_22, and Z_23
13          TRAP(Z_21) ; TRAP(Z_22) ; TRAP(Z_23)
14      else // time cut
15          bisect Z with a time cut into subzoids Z_01 and Z_02
16          TRAP(Z_01) ; TRAP(Z_02)
```

Figure 2-2: Pseudocode for the TRAP divide-and-conquer stencil algorithm operating on a 3-dimensional zoid. TRAP takes as input a 3-dimensional zoid Z. The values $thresh_0$, $thresh_1$, and $thresh_2$ are threshold constants, which can be tuned. The values σ_1 and σ_2 are the slopes [62] of the stencil in the x_1- and x_2-dimensions, respectively. The application-specific KERNEL procedure is not shown.

works as follows. In the base case, which occurs when the height of the zoid falls below a threshold $thresh_0$ tested for in line 2, lines 3–4 perform the stencil computation on each grid point in the input zoid Z using an application-specific KERNEL procedure. In the recursive case, lines 8–16 perform either a ***space cut*** of Z along one of the spatial dimensions or a ***time cut*** of Z along the time dimension, and make recursive calls on the subzoids. An x_1-cut and a time cut on a 3-dimensional zoid are illustrated in Figures 2-1(b) and 2-1(c), respectively. Note that Frigo and Strumpen's original stencil algorithm [30] bisects both in space and time. Since we are autotuning the TRAP code in Pochoir, we follow Pochoir's conventions of trisecting in space and bisecting in time.

TRAP makes specific ***divide choices*** at each recursive step. It may choose to execute the base case (lines 3–4) or divide the given zoid along the x_1 dimension (lines 9–10), or the x_2 dimension (lines 12–13), or the time dimension (lines 15–16) into subzoids. As shown in Figure 2-2, the divide choices are influenced by different threshold constants, which were determined by trial-and-error handtuning by the authors of Pochoir. Different values can yield different runtimes, and

handtuning to find optimal values that minimize runtime can be a laborious task.

For example, suppose that TRAP is passed a zoid Z that fits in the computer's L1 cache, and that height$(Z) \geq 2$. If the threshold constant $thresh_0$ is set to 2, then TRAP would divide the zoid and recur. But terminating the recursion and executing the base case might be faster than dividing and recurring, because it avoids function-call overhead. On the other hand, if $thresh_0$ is set to too large a value, and if Z does not fit in L1, then executing the base case might result in many cache misses, yielding slower code. Thus, $thresh_0$ must be tuned, as must the other tunable constants. The actual TRAP code in Pochoir is more complicated than that shown in Figure 2-2, and uses two kernel functions: a faster kernel at the interior of the grid, and a slower kernel at the boundaries that checks if memory accesses fall off the grid. Hence, two different threshold constants are used in each dimension, one for zoids that lie completely in the interior, and the other for zoids that impinge on the boundary. It might be better to divide zoids that impinge on the boundary, even if they fit in cache, if the subzoids use more of the faster kernel. The tradeoff between tuning for cache locality and for faster kernel usage makes handtuning the threshold constants even harder.

2.2.2 The planned algorithm

The tuning strategy employed by Ztune is not to choose values for the tunable constants like $thresh_0$, however. Instead, Ztune focuses on the divide choices themselves, associating each recursive instantiation of TRAP function with its own divide choice. Given a $(d+1)$-dimensional zoid Z, each instantiation of TRAP has at most $d+2$ divide choices: it can make a time cut; it can make an x_i-cut for some $i \in \{1,2,\ldots,d\}$; or it can terminate the recursion and execute the base case. We shall represent the divide choice for Z as an integer in the set $\{-1,0,1,\ldots,d\}$, where -1 corresponds to the base case, 0 corresponds to the time dimension, and choices $1,2,\ldots,d$ correspond to the spatial dimensions $x_1,x_2,\ldots,x_d$, respectively.

Of course, we must modify TRAP so that it is parametrized by the divide choices rather than the threshold constants. Figure 2-3 shows how TRAP can be transformed into a ***planned algorithm*** called TRAPPLE, which uses divide choices for making decisions about the divide-and-conquer

```
TRAPPLE(Z, z)
 1  if z.choice = −1 // base case
 2      for each (t, x1, x2) ∈ Z
 3          KERNEL(t, x1, x2)
 4  else // recursive case
 5      if z.choice = 1 // x1-cut
 6          trisect Z with an x1-cut into subzoids Z11, Z12, and Z13
 7          let z.children = ⟨z11, z12, z13⟩
 8          TRAPPLE(Z11, z11) ; TRAPPLE(Z12, z12) ; TRAPPLE(Z13, z13)
 9      elseif z.choice = 2 // x2-cut
10          trisect Z with an x2-cut into subzoids Z21, Z22, and Z23
11          let z.children = ⟨z21, z22, z23⟩
12          TRAPPLE(Z21, z21) ; TRAPPLE(Z22, z22) ; TRAPPLE(Z23, z23)
13      elseif z.choice = 0 // time cut
14          bisect Z with a time cut into subzoids Z01 and Z02
15          let z.children = ⟨z01, z02⟩
16          TRAPPLE(Z01, z01) ; TRAPPLE(Z02, z02)
17      else error "invalid choice"
```

Figure 2-3: Pseudocode for the planned algorithm TRAPPLE operating on a 3-dimensional zoid Z. In addition to Z, TRAPPLE takes as input the root node z of a "plan" for Z. TRAPPLE assumes that the children of z and the subzoids of Z are maintained in the same order.

execution of the code. TRAPPLE operates on a ***plan***, which is an ordered tree of divide choices made by the algorithm at each recursive step of computation. Each node z in the plan correponds to a zoid Z that arises during the execution of TRAPPLE, and has the following fields:

- $z.choice$: the divide choice for Z, drawn from $\{-1, 0, 1, \ldots, d\}$.

- $z.children$: an ordered list of the children of z, or NIL if the divide choice is -1, corresponding to a base case.

If the divide choice for a node z, which corresponds to a zoid Z, is to divide Z into k subzoids $Z_1, Z_2, \ldots, Z_k$, then z has k ordered children $z_1, z_2, \ldots, z_k$, where node z_i corresponds to zoid Z_i. As TRAPPLE executes, it consults the root node z of the plan for the input zoid Z. Using the divide choice stored in $z.choice$, it either executes the base case, or divides Z into subzoids and recursively calls itself on each, passing the subzoid Z_i and the corresponding child node z_i as arguments.

2.2.3 Plan-finding

The autotuning problem for divide-and-conquer stencil codes can now be stated as follows. Define the ***(execution) cost*** of a piece of code to be the time taken to execute the code on a given computer. The ***cost*** of a plan is the time taken to execute the plan. The ***plan-finding*** problem is to find an ***optimal plan***, that is, a plan with the minimum cost, to perform stencil computation on a given grid on a given computer. At its heart, Ztune is simply a plan-finding algorithm for TRAPPLE. Two computers may have different costs for the same plan, due to differences in their hardware architecture, operating system, and compiler, among other things. For example, the cost of a plan may vary with the cache size. Hence, a solution to the plan-finding problem may not be portable across computers.

2.3 Ztune

This section presents the Ztune algorithm, which finds an optimal plan for the TRAPPLE stencil algorithm. Ztune can be slow, however, but later sections will show how to make it run fast. This section also formalizes the search domain explored by Ztune.

Figure 2-4 shows the pseudocode for procedure ZTUNE, which instruments the TRAPPLE algorithm with timer calls, and finds an optimal plan for stencil computation on a $(d+1)$-dimensional zoid Z. Although performance is not exactly repeatable, our experience with Ztune shows that instrumentation can be sufficiently accurate to produce highly efficient plans. ZTUNE uses the following auxiliary procedures:

- TIC() starts a timer.
- TOC() stops the timer and returns the elapsed time since the last call to TIC.
- CHOICE(Z) returns the set of possible divide choices for Z, where the set excludes the choice -1 for executing Z as a base case.
- BASE-CASE(Z, z), shown in Figure 2-5, chooses base case as the divide choice for Z, if executing the base case is no more expensive than the current plan for Z.

```
ZTUNE(Z)
 1  call-cost = TOC()
 2  z = LOOKUP(Z)
 3  if z == NIL
 4      Allocate plan node z
 5      z.cost = ∞
 6      z.children = NIL
 7      C = CHOICE(Z)
 8      for each choice c ∈ C
 9          TIC()
10          Divide Z using choice c into k_c subzoids Z_c1, Z_c2, ..., Z_ck_c
11          rec-cost = TOC()
12          for i = 1 to k_c
13              TIC()
14              (z_ci, call-cost_ci) = ZTUNE(Z_ci)
15              ret-cost_ci = TOC()
16              rec-cost += call-cost_ci + z_ci.cost + ret-cost_ci
17          if rec-cost < z.cost
18              z.cost = rec-cost
19              z.choice = c
20              z.children = ⟨z_c1, z_c2, ..., z_ck_c⟩
21      BASE-CASE(Z, z)
22      INSERT(Z, z)
23  TIC()
24  return (z, call-cost)
```

Figure 2-4: Procedure ZTUNE, which finds an optimal plan for the TRAPPLE stencil algorithm operating on a $(d+1)$-dimensional zoid Z. ZTUNE returns an ordered pair consisting of the root node z of the optimal plan and the partial function-call overhead *call-cost*. The sections of code that are timed are highlighted in blue. Although line 14 is shown as being timed, only the function-call overhead in calling ZTUNE is measured. The definitions of the TIC, TOC, CHOICE, INSERT, and LOOKUP procedures are not shown.

Besides ZTUNE uses two procedures INSERT and LOOKUP, which store and look up optimal plans for zoids, avoiding repeated plan-finding for the same zoid. We shall discuss the implementation of INSERT and LOOKUP in Section 2.5, where we introduce the notion of equivalence among zoids. ZTUNE measures the costs of all possible divide choices for zoid Z, and stores the best choice in the corresponding node z. Each node z in the plan has an additional field $z.cost$, which stores the ***optimal cost***, that is, the cost of an optimal plan for the corresponding zoid. Determining where to place TIC and TOC within code in order to properly measure all relevant costs without

```
BASE-CASE(Z, z)
1  TIC()
2  for each (t, x_1, x_2, ..., x_d) ∈ Z
3      KERNEL(t, x_1, x_2, ..., x_d)
4  base-cost = TOC()
5  if base-cost ≤ z.cost
6      z.cost = base-cost
7      z.choice = −1 // base case
8      z.children = NIL
```

Figure 2-5: Pseudocode to execute the base case. BASE-CASE takes as input a zoid Z and the root node z of a plan for Z. The sections of code that are timed are highlighted in blue.

also capturing bookkeeping overhead is a bit tricky. To measure function-call overhead in calling ZTUNE, a call to ZTUNE must be immediately preceded by a TIC and immediately succeeded by a TOC.

ZTUNE works as follows. Line 1 measures the ***call cost***, that is, the function-call overhead in invoking ZTUNE. Line 23 starts the timer, which the caller of ZTUNE can stop and measure the ***return cost***, that is, the function-call overhead in returning from ZTUNE. The sum of the call and return costs is the ***link cost***, which is the total function-call overhead in calling ZTUNE on Z. If LOOKUP finds the root node z of an optimal plan for zoid Z in line 2, then ZTUNE simply returns z in line 24. Otherwise, ZTUNE proceeds as follows.

Recursive case For each possible divide choice c of zoid Z, lines 9–11 measure the ***divide cost*** of Z, that is, the time taken to divide Z using choice c into subzoids. The ***recursion cost*** of Z for choice c is the sum of the divide cost of Z, and the link and optimal costs of subzoids created using choice c. Lines 11–16 compute the recursion cost for choice c, which is maintained in variable *rec-cost*, as follows. Line 11 initializes *rec-cost* with the divide cost of Z. Line 14 recursively finds an optimal plan for subzoid Z_{ci} and the call cost. Line 15 measures the return cost. Line 16 increments *rec-cost* with the sum of the link and optimal costs of Z_{ci}. Lines 18–20 update the attributes of node z, if *rec-cost* is smaller than the cost of the current plan for Z.

Base case Line 21 invokes procedure BASE-CASE, which is shown in Figure 2-5. BASE-CASE works as follows. Lines 1–4 measure the ***base-case cost*** of Z, that is, the time to perform stencil

computation on Z using an application-specific procedure KERNEL. If the measured base-case cost is at most the cost of the current plan for Z, then lines 6–8 update the attributes of node z accordingly.

Line 22 of ZTUNE invokes INSERT, which inserts node z, as the root of an optimal plan for zoid Z, into a lookup table. Procedure ZTUNE is guaranteed to terminate, if TRAPPLE terminates for each divide choice of Z.

2.3.1 Search domain

In principle, the basic Ztune algorithm explores the search domain of all possible ways of executing the planned divide-and-conquer stencil algorithm TRAPPLE in order to select the best plan. To close this section, we formalize this notion.

The ***choice domain*** for TRAPPLE run on a given $(d+1)$-dimensional zoid Z can be viewed as the set of all possible plans for Z. Many of the plans have a common structure, however, and in particular, the roots of all plans share one of the at most $d+1$ divide choices that actually divide Z, as well as the -1 base-case divide choice. We can represent the entire choice domain as a single tree as follows. For any node z in the tree corresponding to a zoid Z, let k_c be the number of subzoids produced by choice $c \in \{0,1,\ldots,d\}$. Then node z has $k = \sum_{c \in \text{CHOICE}(Z)} k_c$ children, where the child z_{ci} corresponds to the ith zoid produced by dividing Z according to choice c. Since each node in the choice domain corresponds to a particular zoid, for convenience, we shall sometimes view the choice domain as this tree of zoids. ZTUNE can now be viewed as walking this search domain tree, finding the optimal cost at each node in the tree, selecting the best divide choice, and recording the results in a plan, which is a subtree of the choice domain.

2.3.2 Analysis

The number of plans in the choice domain is huge, however, and is exponential in the height of the zoid at the root, as shown in the following theorem.

Theorem 1. *Consider a zoid Z with height h. The number of plans in the choice domain rooted at*

Benchmark	*STE + DS + FD*	*STE + DS*	*STE*
APOP	1,246.98	1,248.84	—
Heat1	1.70	1.61	1.62
Heat2	50.77	50.01	50.24
Life	17.04	16.57	—
Heat3	24.75	25.01	—
Heat4	19.61	19.65	—
Heat5	559.01	575.07	—
LBM	13.24	13.40	13.45
Wave	815.12	826.19	—
Heat6	18.99	18.81	—

Table 2.5: Runtimes (in seconds) of the tuned TRAPPLE code on *Ivy Bridge* under different combinations of the three pruning properties. A dash sign (—) indicates that plan-finding timed out after two days. The reported runtimes are the better of two runs.

Z is $\Omega(2^{h/2})$.

Proof. Without loss of generality, let $h \in \mathbf{Z}^+$ be a power of two. At each time cut, the TRAPPLE algorithm bisects a zoid with height $h > 1$ into two subzoids of height $h/2$. It follows that the set S of heights of zoids in the choice domain is given by $\{h, h/2, \ldots, 1\}$. We use induction over the heights in S to prove the theorem. Let $P(h)$ be the number of plans in the choice domain rooted at a zoid with height h. Then the inductive hypotheis is given by $P(h) \geq 2^{h/2}$. The base case holds for $h = 2$, since a height 2 zoid can be divided in time, or executed as a base case. For the inductive step, assume that $P(h/2)$ is true. Zoid Z with height h can be divided in time into two subzoids of heights $h/2$, or executed as a base case. Then the number of plans in the choice domain rooted at Z is $P(h) = P(h/2).P(h/2) + 1 \geq 2^{h/2}$. □

The subsequent sections of this chapter improve the exploration of the choice domain using three pruning properties.

Benchmark	*STE + DS + FD*	*STE + DS*	*STE*
APOP	35	35	∞
Heat1	20	52	11,927
Heat2	395	549	83,638
Life	264	506	∞
Heat3	66	221	∞
Heat4	47	133	∞
Heat5	393	1,187	∞
LBM	34	715	103,441
Wave	3,114	54,317	∞
Heat6	345	33,925	∞

Table 2.6: Plan-finding times (in seconds) of Ztune on *Ivy Bridge* under different combinations of the three pruning properties. An infinity sign (∞) indicates that plan-finding timed out after two days. The reported plan-finding times are the better of two runs.

Benchmark	*Ratio*	*Benchmark*	*Ratio*
APOP	0.03	Heat4	2.41
Heat1	11.50	Heat5	0.70
Heat2	7.77	LBM	2.54
Life	15.48	Wave	3.82
Heat3	2.65	Heat6	18.17

Table 2.7: Ratio of plan-finding time to runtime under all three pruning properties.

2.4 Ztune's pruning properties

This section provides an overview of Ztune's pruning properties — space-time equivalence (STE), divide subsumption (DS), and favored dimension (FD) — which speed up plan-finding significantly while preserving the runtime performance of the tuned TRAPPLE code. A detailed description of the three pruning properties is deferred to Sections 2.5, 2.6, and 2.7. This section presents empirical results comparing autotuning times of Ztune and runtimes of the tuned TRAPPLE code under the three pruning properties. This section also describes the performance of Ztune on various grid sizes, and concludes with a discussion on the effects of noise in autotuning.

The naive ZTUNE procedure is slow, because the choice domain is huge. By reducing the size of the choice domain, the three pruning properties greatly accelerate tuning. The space-time

equivalence property makes the assumption that two zoids with the same shape and size have the same optimal plan, irrespective of where they are located in the space-time grid. FFTW [27, 28] uses a similar notion of equivalence in its tuning strategy. Coarsening the base-case sizes in divide-and-conquer stencil codes and tuning for tile sizes in tiling-based stencil codes implicitly assume equivalence. Because of STE, it makes sense to save plans in a lookup table so that they can be reused elsewhere in the search domain. The divide-subsumption property assumes that if a zoid is divided since it does not fit in cache, then all its ancestors in the search domain can be divided, without computing their base-case costs. The intuition is that none of the ancestors will fit in cache either, and hence measuring their base-case costs higher up in the tree is fruitless. The favored-dimension property exploits the layout of spatial grids as a linear array in memory. Computations along the unit-stride dimension execute faster than those along the other dimensions because of prefetching and vectorization. Consequently, FD restricts the search to zoids that have a longer unit-stride dimension. [18] describes experiments in tiling-based stencil codes, where choosing tiles that are longer in the unit-stride dimension results in faster runtimes.

Figure 2.5 shows that the various pruning properties do not significantly affect the runtimes of the tuned TRAPPLE codes. All three codes perform nearly the same for the three benchmarks all could compute. The tuned code generated under STE + DS + FD runs on average at the same speed as that generated under STE + DS. The worst-case behavior occurs in the Heat1 benchmark, where the tuned code generated under STE + DS + FD was $1.70/1.61 - 1 = 6$ percent slower.

Figure 2.6 shows that the pruning properties greatly accelerate plan-finding. Whereas Ztune with no pruning properties fails to find a plan for any benchmark in two days (and hence the table contains no column for no properties), STE alone succeeds in finding plans for 3 of the 10 benchmarks. STE + DS successfully finds plans for all benchmarks, speeding up plan-finding under just STE by a factor of 172 on average. Plan-finding under STE + DS + FD ran on average 4.9 times faster than plan-finding under STE + DS. The best speedup in this case occurs in the Heat6 benchmark, where plan-finding under all three properties was $33,925/345 = 98$ times faster.

Figure 2.7 shows the ratio of plan-finding time of Ztune to the runtime of the tuned TRAPPLE code under all three properties. This ratio is important, since it indicates how long it takes to

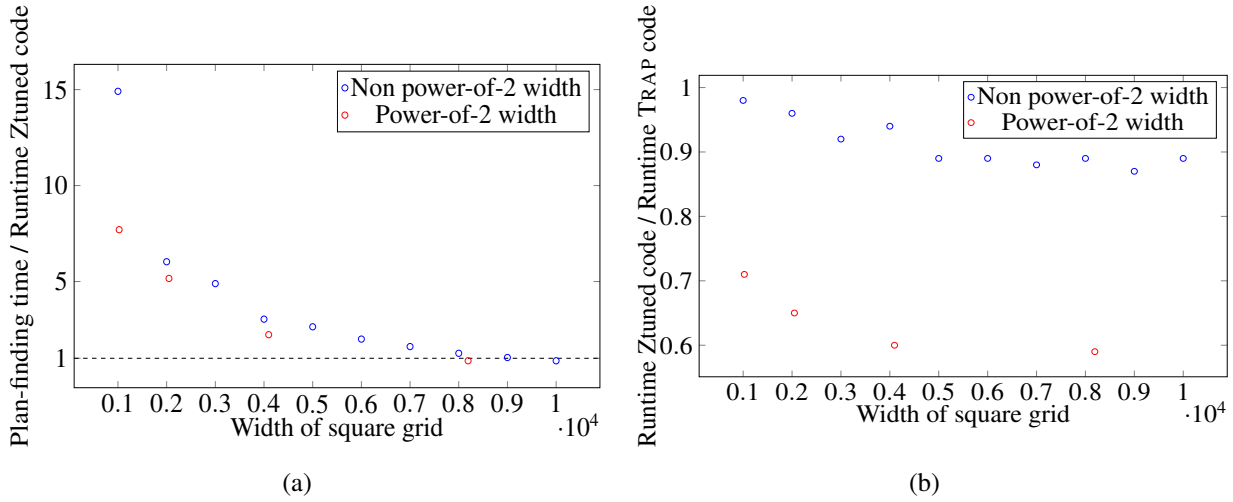


Figure 2-6: Performance of the Ztuned code in computing the 2D periodic heat stencil for 512 time steps on square grids with different sizes, on Ivy Bridge. The grid sizes include power-of-2 and non power-of-2 widths. (a) Ratios of plan-finding time to runtime of the Ztuned code. (b) Ratios of the runtime of Ztuned TRAPPLE code to the runtime of Pochoir's TRAP code. A lower ratio indicates that the Ztuned code runs faster than Pochoir's code. The reported numbers are the better of two runs.

amortize the autotuning or plan-finding time over the runtime of the tuned TRAPPLE code for a given benchmark. Interestingly, plan-finding is actually faster than the runtime for the APOP and Heat5 benchmarks. Plan-finding under all three properties, in practice, seems to take time at most a few multiples of the runtime.

Figure 2-6 shows the performance of Ztuned code on increasing grid sizes. Figure 2-6(a) shows that the ratio of plan-finding time to runtime of the tuned code decreases as grid sizes increase. Due to the pruning properties, most of the plan-finding time is spent in measuring the base-case costs of different zoids in the choice domain that fit in cache. Since the fraction of zoids that fit in cache is high for smaller grids, their plan-finding times are higher. As this fraction is low for bigger grids, their plan-finding times are also relatively lower. Figure 2-6(b) shows that larger grids benefit more from tuning. The Ztuned code is at most 12% faster for grids with non power-of-2 widths, and at most 40% faster for grids with power-of-2 widths.

The results from plan-finding are highly reproducible. In 10 plan-finding runs on each benchmark under all three pruning properties on the Ivy Bridge machine, the tuned runtimes differed by

Benchmark	*Predicted*	*Actual*	*Ratio*
APOP	1,215.50	1,246.98	1.03
Heat1	1.68	1.70	1.01
Heat2	48.59	50.77	1.05
Life	16.21	17.04	1.05
Heat3	24.05	24.75	1.03
Heat4	18.52	19.61	1.06
Heat5	534.93	559.01	1.05
LBM	12.14	13.24	1.09
Wave	795.66	815.12	1.02
Heat6	19.69	18.99	0.96

Table 2.8: Comparison of "predicted" and "actual" runtimes (in seconds) on *Ivy Bridge* under all three pruning properties. The header *Predicted* indicates the estimated runtime of the Ztuned code. The header *Actual* indicates the actual runtime of the Ztuned code. The header *Ratio* indicates the ratio of actual to predicted runtimes. Ratios close to 1.0 indicate that the predicted and actual runtimes agree, and that autotuning is less prone to noise. The reported comparison is the better of two runs.

at most 1% on average.

2.4.1 Effects of noise on measurements

The accuracy of time measurements can be affected by different sources of noise. Noise during autotuning can be broadly classified into two types. Noise from sources external to the autotuner and the tuned program, such as hardware and system effects, are ***exogenous***. Noise internal to the auotuner and the tuned program, due to their implementation and the assumptions made by the pruning properties, are ***endogenous***. To keep exogenous noise minimal, we ran the experiments in a quiesced machine environment. We further disabled noise sources like Intel's Turbo Boost technology that can change the processing core's operating speed during autotuning. We took care to avoid endogenous noise sources like printf and cout statements, which interfere with measurements during tuning. The effects of ***cold misses***, which are incurred the first time data is brought into cache, on measurements are negligible since the number of cold misses is significantly smaller than the number of memory accesses for stencil codes. For example, given a zoid with spatial volume n and height h, the number of cold misses is n, whereas the number of memory accesses is

$\Theta(nh)$. To keep the cold miss effects minimal, we warmed up the cache by performing stencil computation on the entire spatial grid for a few time steps before tuning.

Figure 2.8 gives a qualitiative measure of noise in autotuning. We define ***Predicted time*** to be the cost of an optimal plan found by plan-finding, that is, the estimated runtime of the Ztuned code. ***Actual time*** is the time taken to execute the plan, that is, the actual runtime of the Ztuned code. Figure 2.8 indicates that the predicted and actual runtimes of the plans agree well for most benchmarks, and differ on average by 3%. This empirical evidence indicates that Ztune's tuning strategy is less prone to noise.

We shall examine the pruning properties in detail in the next 3 sections.

2.5 Space-time equivalence property

This section describes Ztune's "space-time equivalence" (STE) property, which speeds up plan-finding by assuming that zoids with the same shape and size have the same optimal plan irrespective of their location in the space-time. We show that plan-finding under STE incurs $O(2^d n \lg h)$ memory overhead, where n and h are the spatial volume and height, respectively, of a $(d+1)$-dimensional zoid Z. We also show that plan-finding for Z under STE takes $O(2^d n^2 h)$ time. This section concludes with a discussion on the practical validity of STE. As mentioned before, STE succeeds in finding plans on 3 of the 10 benchmarks given two days of plan-finding time for each benchmark.

Two 3-dimensional zoids $Z \subseteq \mathbf{N} \times \mathbf{Z}^2$ and $Z' \subseteq \mathbf{N} \times \mathbf{Z}^2$ are ***space-time equivalent*** if there exists a bijection $f : Z \to Z'$ and constants $T, X_1, X_2 \in \mathbf{Z}$ such that for all $(t, x_1, x_2) \in Z$, we have $f(t, x_1, x_2) = (t + T, x_1 + X_1, x_2 + X_2)$, that is, Z' is a translation of Z in space-time. Space-time equivalent zoids in higher dimensions are defined similarly. The ***space-time equivalence*** property states that two space-time equivalent zoids have the same optimal plan. Two zoids that are not space-time equivalent are ***space-time distinct***, or ***distinct*** for short. The functionality of INSERT and LOOKUP auxiliary procedures under space-time equivalence can now be defined.

- INSERT(Z, z) inserts the root node z of an optimal plan for zoid Z into a lookup table, which

maintains the root nodes of optimal plans for distinct zoids. The lookup table can be a dictionary, for example a hash table.

- LOOKUP(Z) searches the lookup table for the root node of an optimal plan for Z.

2.5.1 Analysis

Before we analyze space-time equivalence, it is important to note that the TRAPPLE algorithm takes $\Theta(n)$ memory and $\Theta(nh)$ time to perform stencil computation on a $(d+1)$-dimensional zoid with spatial volume n and height h. Hence, it would be useful if plan-finding does not incur significantly more memory and time overheads than TRAPPLE.

We introduce some definitions, many of which are borrowed or adapted from [30, 31, 62]. A 3-dimensional zoid $Z = \text{zoid}(ta, tb, xa_1, xb_1, dxa_1, dxb_1, xa_2, xb_2, dxa_2, dxb_2)$ is ***well-defined*** if its height is positive, its widths along the x_1- and x_2- dimensions are positive, and the lengths of its bases along the x_1- and x_2- dimensions are nonnegative. We assume that the zoids we consider are well-defined. Define the ***projection trapezoid*** Z_1 of zoid Z along spatial dimension x_1 to be the 2D trapezoid that results from projecting Z onto the dimensions x_1 and t. The projection trapezoid Z_1 has two bases (sides parallel to the x_1 axis). We say that Z_1 is ***upright***, if the longer base of Z_1 corresponds to time ta, and ***inverted*** otherwise. These definitions can be extended to higher dimensional zoids. The ***space-time volume*** of a zoid is the product of its spatial volume and height.

Now suppose we assume ***no equivalence***, that is, no two zoids in the search domain have the same optimal plan. Then plan-finding under no equivalence does not need to store and look up plans of zoids. However, it can still incur a large memory overhead in maintaining the optimal plan, as stated in the following theorem.

Theorem 2. *Consider a $(d+1)$-dimensional zoid Z with spatial volume n and height h. Let L be the average space-time volume of zoids that correspond to the leaves of an optimal plan for Z. Then the optimal plan has at least nh/L nodes.*

Proof. The space-time volume of Z is given by nh. Since a leaf has space-time volume L on average, the optimal plan has nh/L leaves. □

More importantly, no-equivalence doesn't find plans for any benchmark within two days.

We analyze the memory and time overheads of space-time equivalence in the following. We state the following lemma, which we use in our analysis, and skip its proof.

Lemma 3. *Consider the recursion tree R obtained by dividing an integer $m > 0$ into $\lfloor m/2 \rfloor$ and $\lceil m/2 \rceil$, and recursively dividing those two integers similarly, until we have 1. There exist at most 2 distinct integers $\lfloor m/2^k \rfloor$ and $\lceil m/2^k \rceil$ at level $k \geq 0$ of tree R.* □

Since the recursion tree R has $\lg m + 1$ levels, it follows from Lemma 3 that R has at most $2 \lg m + 2$ distinct integers.

Lemma 4. *Consider a zoid Z with height h. The zoids in the search domain tree rooted at Z have $\Theta(\lg h)$ different heights.*

Proof. The height of a zoid decreases only when it is cut in time. At each time cut, the TRAPPLE algorithm bisects a given zoid with height $h' > 1$ into two subzoids with heights $\lfloor h'/2 \rfloor$ and $\lceil h'/2 \rceil$. The result then follows from Lemma 3. □

Theorem 5. *Consider a $(d+1)$-dimensional zoid Z with spatial volume n and height h. The search domain tree $\mathscr{T}$ rooted at Z has $O(2^d n \lg h)$ distinct zoids.*

Proof. We count the number of distinct zoids for each possible height $h' \in \{h, \lfloor h/2 \rfloor, \lceil h/2 \rceil, \ldots, 1\}$. Without loss of generality, let the width of Z in each spatial dimension be $n^{1/d}$. The number of upright (or inverted) distinct 2D projection trapezoids of a given height h' along a spatial dimension is at most $n^{1/d}$, the width along that dimension. Hence the number of distinct $(d+1)$-dimensional zoids of height h' is at most $(2n^{1/d})^d = 2^d n$. Since the zoids in $\mathscr{T}$ have $\Theta(\lg h)$ different heights from Lemma 4, the result follows. □

Theorem 6. *Consider a $(d+1)$-dimensional zoid Z with spatial volume n and height h. Procedure* ZTUNE *takes $O(2^d n^2 h)$ time to find an optimal plan for Z, under space-time equivalence.*

Proof. To find the plan-finding time for Z, we sum the base-case costs of all distinct zoids in the search domain tree $\mathscr{T}$ rooted at Z. The base-case cost of a zoid is proportional to its space-time volume. We assume that the divide cost, link cost, and the costs of INSERT and LOOKUP

procedures are negligible compared to the base-case cost of a given zoid. The base-case cost of a zoid of a given height h' is at most nh', the maximum space-time volume for height h'. From Theorem 5, there are $2^d n$ distinct zoids of height h'. Hence the sum of base-case costs of distinct zoids of height h' is at most $2^d n^2 h'$. And the sum of base-case costs of distinct zoids of all heights $h' \in \{h, \lfloor h/2 \rfloor, \lceil h/2 \rceil, \ldots, 1\}$ is at most $\sum_{h' \in \{h, \lfloor h/2 \rfloor, \lceil h/2 \rceil, \ldots, 1\}} 2^d n^2 h' = O(2^d n^2 h)$. □

However, the memory and time bounds in Theorems 5 and 6 might not be tight. In specific, we conjecture that the memory overhead under space-time equivalence is $\Theta(n)$, which is also the memory overhead of TRAPPLE. We also considered ***time equivalent*** zoids, which are translations in just the time dimension, and a ***time equivalence*** property, which assumes that two time equivalent zoids have the same optimal plan. Time equivalence incurred significantly more memory and plan-finding time overheads, however, and failed to find a plan for any benchmark.

2.5.2 Handling Boundaries

STE, as defined, does not hold at the boundaries of the spatial grid. Recall that TRAP uses a faster base-case kernel function for zoids that lie in the interior of the grid, and a slower kernel function for zoids that impinge on the boundary. Consequently, two space-time equivalent zoids can have different base-case costs if one lies in the interior and the other impinges on the boundary. To address this anomaly, we use STE only for zoids that lie in the interior, and use a variant of STE called "boundary equivalence" for zoids that impinge on the boundary. Suppose we have two 3-dimensional zoids $Z \subseteq \mathbf{N} \times \mathbf{Z}^2$ and $Z' \subseteq \mathbf{N} \times \mathbf{Z}^2$, such that, Z and Z' impinge on the boundary in the x_1 dimension, but lie in the interior in the x_2 dimension. Z and Z' are ***boundary equivalent*** in x_1 if there exists a bijection $f : Z \to Z'$ and constants $T, X_2 \in \mathbf{Z}$ such that for all $(t, x_1, x_2) \in Z$, we have $f(t, x_1, x_2) = (t + T, x_1, x_2 + X_2)$, that is, Z' can be a translation of Z in all dimensions except the spatial dimension x_1. The ***boundary equivalence*** property states that two zoids that are boundary equivalent in x_1 have the same optimal plan. These definitions can be adapted when the zoids impinge on the boundary in the x_2 dimension, or in both the x_1 and x_2 dimensions.

```
DIVIDE-SUBSUMPTION(Z, z)
1  measurebase = TRUE
2  for z' ∈ z.children
3     if z'.choice ≠ −1 // test if a subzoid of Z was divided
4        measurebase = FALSE
5  if measurebase == TRUE
6     BASE-CASE (Z, z)
```

Figure 2-7: Pseudocode for divide-subsumption. DIVIDE-SUBSUMPTION takes as input a zoid Z and the root node z of a plan for Z.

2.5.3 Discussion

Why does STE despite being a theoretical property hold in practice? STE assumes that the "context of execution" of two space-time equivalent zoids is the same, that is, the number of floating-point operations and memory accesses, the memory layout of the spatial grid, cache alignment of the grid points, and the states of the processor, memory hierarchy and other system components remain the same. The number of floating-point operations and memory accesses, which dominate the runtime of stencil codes, and the memory layout of the spatial grid are the same for two space-time equivalent zoids. The cache alignment of the grid points, and the states of the processor, memory hierarchy and other system components might differ, however, during execution of two space-time equivalent zoids. These effects do not dominate the runtime of stencil codes, however. Empirical evidence shown in Figure 2.8, which aggregates noise from all the pruning properties, indicates that STE holds reasonably well in practice.

2.6 Divide-Subsumption property

This section describes Ztune's "divide-subsumption" (DS) property, which speeds up plan-finding significantly, using the notion that it may not be necessary to find the base-case cost of every zoid in the search domain. For example, DS speeds up plan-finding for the APOP benchmark by a factor of at least 4900.

The ***divide-subsumption*** property states that executing the base case cannot be the optimal divide choice for a zoid Z, if it is not the optimal divide choice for a subzoid Z' of Z. The intuition is that procedure ZTUNE chose to divide Z', since Z' did not fit in cache and hence it was expensive to execute the base case on Z'. Since Z is larger than Z', Z will also not fit in cache and hence it will also be expensive to execute the base case on Z. Once ZTUNE chooses to divide a zoid Z' during the bottom-up traversal of a search domain, it can divide the ancestors of Z' in the search domain, and avoid finding their base-case costs under DS. The property cuts the search domain such that zoids that lie above the cut are divided, and zoids that lie below the cut are undivided and have the base case executed on them.

Figure 2-7 shows a DIVIDE-SUBSUMPTION procedure, which works as follows. Lines 2–4 perform the DS test. If a subzoid of Z was divided, then the procedure skips measuring the base-case cost of Z, by not calling BASE-CASE. Otherwise, line 6 calls BASE-CASE, which measures the base-case cost of Z. DIVIDE-SUBSUMPTION assumes that plans from all possible divide choices of Z except the base-case were already found, and that the current plan rooted at node z has the smallest cost of all such plans. To perform plan-finding using DS, line 21 of the ZTUNE procedure in Figure 2-4 can be replaced with a call to DIVIDE-SUBSUMPTION.

Figures 2.5 and 2.6 show the performance of plan-finding under DS and STE on the benchmarks. DS successfully finds plans for all benchmarks, and takes at most 15 hours to find a plan for a benchmark. It speeds up plan-finding under STE by a factor of 144–229. Significantly, faster plan-finding with DS does not affect the quality of the runtime.

2.6.1 Discussion

A few important design choices that we made in our implementation are described in the following. We do not assume DS for small zoids, since measuring their base-case costs can be error-prone. Under DS, plan-finding might erroneously choose to divide a zoid, whereas the correct choice would have been to execute the zoid as a base case. For example, a page fault during measurement can artificially increase the base-case cost of a zoid Z. DS would then divide Z and all the ancestors of Z in the choice domain, resulting in a sub-optimal plan. We say that two zoids Z_1, Z_2 are

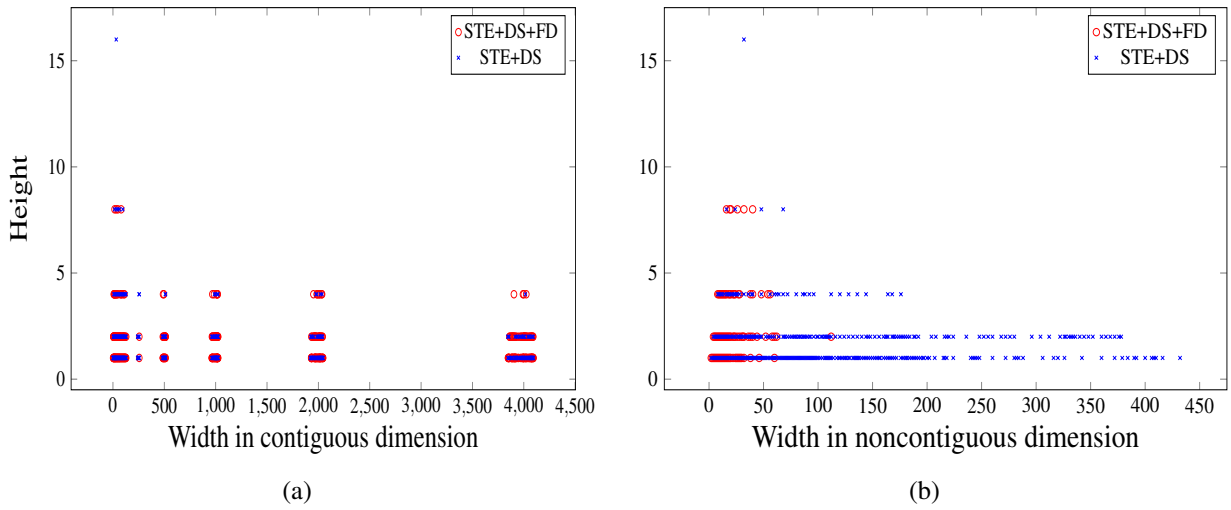


Figure 2-8: Comparison of base case sizes in the plans created with STE+DS and STE+DS+FD for the Heat4 benchmark with grid size 4096×4096, and 512 time steps, on Ivy Bridge. (a) Widths of base case zoids in the "contiguous" dimension. (b) Widths of base case zoids in the noncontiguous dimension.

consecutive descendants of a zoid Z_3, if Z_1 is a subzoid of Z_2 and Z_2 is a subzoid of Z_3. To make DS robust, we measure the base-case cost of each zoid in the choice domain, unless it has two consecutive descendants that were divided.

2.7 Favored-dimension property

This section describes an empirical property used by Ztune called "favored-dimension" (FD), which can greatly reduce plan-finding time, especially for stencils in higher dimensions. As an example, FD speeds up plan-finding for the Heat6 benchmark by a factor of 98. FD exploits architectural features of modern computers, such as vector units, cache blocking, and hardware prefetching, which accelerate performance when a processing core operates on consecutive memory locations. This section also reviews the empirical results that substantiate the effectiveness of the property.

Favored-dimension exploits the fact that when a processing core peforms computations on consecutive locations in the linear array of memory, the hardware operates considerably faster than

```
PRUNE-CHOICE(Z)
1  C = CHOICE(Z)
   // S is the set of noncontiguous dimensions in which Z can be divided
2  S = C ∩ {1, 2, ..., d − 1}
3  if S ≠ ∅
4      Choose a dimension i ∈ S arbitrarily
5      return C ∩ {0, i}
6  else
7      return C
```

Figure 2-9: Pseudocode for favored-dimension. PRUNE-CHOICE takes as input a $(d+1)$-dimensional zoid Z and returns the set of divide choices for Z. Choice 0 corresponds to the time dimension, choices $1, 2, \ldots, d-1$ correspond to the noncontiguous spatial dimensions, and choice d corresponds to the contiguous spatial dimension.

when the computations are performed on nonconsecutive locations. Consequently, the way that the spatial grid is laid out in memory strongly influences performance. [18,58] discuss this observation in tiling-based stencil codes. For a d-dimensional array, the ***stride*** of a dimension i is the distance in the linear array of memory between a grid point and an adjacent grid point, whose coordinates differ only in dimension i. We assume that a d-dimensional spatial grid is stored in memory as a linear array, where the dimensions are sorted by stride, with dimension 1 having the largest stride and dimension d — the ***contiguous*** dimension — having the smallest. For a 2-dimensional spatial grid, this layout corresponds to column-major order. These assumptions are without loss of generality, as the same optimizations could be implemented no matter how the dimensions are permuted. Figure 2-8 shows that an optimal plan found under STE and DS has base cases that are significantly longer in the contiguous dimension than in the noncontiguous dimension. Favored-dimension exploits this observation to prune the choice domain further.

The ***favored-dimension*** property for a $(d+1)$-dimensional zoid Z is illustrated in Figure 2-9 using procedure PRUNE-CHOICE, which works as follows. Line 2 defines S as the set of noncontiguous spatial dimensions along which Z can be divided. If the set S is not empty, then line 4 chooses a noncontiguous spatial dimension $i \in S$ arbitrarily. Line 5 returns a pruned set of divide choices, which includes the spatial dimension i and may include the time dimension 0 depending

on whether time is a possible divide choice for Z. If the set S is empty, then possible divide choices for Z are either the contiguous dimension, or the time dimension, or both, or none. In this case, Line 7 returns the original set C of divide choices, without pruning. To perform plan-finding with favored-dimension, the call to CHOICE(Z) in line 7 of the ZTUNE procedure in Figure 2-4 can be replaced with a call to PRUNE-CHOICE(Z).

Favored-dimension does the following. It restricts that zoid Z be divided in at most 1 spatial dimension. To leverage the compiler and hardware optimizations in the contiguous dimension, the property avoids dividing Z along that dimension, where possible. Let P and P' be the minimum cost plans found with and without favored-dimension respectively, for zoid Z. Then favored-dimension assumes that the cost of plan P is no higher than the cost of plan P'.

Figures 2.5, 2.6, and 2.7 illustrate the performance of favored-dimension on the benchmarks. Favored-dimension speeds up plan-finding for 2 or higher dimensional problems by a factor of 1–98. Although procedure PRUNE-CHOICE allows us to pick a noncontiguous dimension arbitrarily in line 4 of Figure 2-9, for the experiments, we let the procedure choose a noncontiguous dimension $i \in S$, such that, i has the biggest stride among the dimensions in S. Choosing the noncontiguous dimension arbitrarily produces similar runtimes for the tuned TRAPPLE code.

2.8 Comparison with heuristic autotuning

This section compares the pruned-exhaustive tuning strategy in Ztune with the heuristic tuning strategy in the OpenTuner framework [3] used for stencil codes. We briefly describe OpenTuner's search domain, which is different from the choice domain of Ztune. Whereas the introduction of the chapter made an absolute comparison of the performance of Ztuned and OpenTuner tuned codes by running OpenTuner for 16 hours, this section makes a relative comparison of the tuned codes, by running OpenTuner for a few multiples of time longer than Ztune. We also report on the autotuning times taken by OpenTuner so that the OpenTuner tuned code achieves similar runtime as the Ztuned code on the benchmarks. Empirical evidence indicates that Ztune can autotune faster than OpenTuner, explore a more complex search domain, and generally produce tuned code that is

Benchmark	*1*	*2*	*4*	*8*	*16*	*32*
APOP	1.65	1.65	1.65	1.65	1.65	1.65
Heat1	2.37	2.37	1.61	1.61	1.06	1.00
Heat2	12.15	12.15	12.15	4.45	1.53	1.09
Life	3.97	3.15	1.29	1.23	1.00	1.00
Heat3	5.10	1.53	1.53	1.53	1.14	1.14
Heat4	1.67	1.67	1.67	1.67	1.67	1.35
Heat5	1.20	1.20	1.20	1.20	1.20	1.20
LBM	†4.35	†4.35	‡3.03	‡3.03	‡1.77	1.56
Wave	5.79	5.79	2.33	2.33	1.64	1.64
Heat6	1.94	1.93	1.73	1.25	1.14	1.14

Table 2.9: Ratios of the runtimes of OpenTuner tuned and Ztuned codes on *Ivy Bridge*. The column headers indicate how many times longer OpenTuner tuned the benchmark than Ztune. The values indicate the ratios of runtimes of OpenTuner tuned TRAP code to the runtimes of Ztuned TRAPPLE code. A bigger ratio favors Ztune, and a smaller ratio OpenTuner. OpenTuner autotuned each benchmark using the IntegerValues, PowersOfTwo, and ListOfNumbers search domains, and the best OpenTuner tuned runtime among the three search domains was used to compute the ratio. Unannotated runtimes indicate PowersOfTwo search domain, and those annotated with the daggers † and ‡ indicate IntegerValues and ListOfNumbers search domains, respectively. Values are the best of two runs.

faster.

Regrettably for scientific purposes, it is not feasible to configure OpenTuner to autotune TRAPPLE. To do so, one needs to parametrize the divide choice at each node in the choice domain, traversed by Ztune. Parametrizing the divide choice at each node creates a huge memory overhead, however, since the number of nodes in the choice domain is asymptotically larger than the space-time volume of the zoid Z at the root of the choice domain. Recall that TRAPPLE incurs memory overhead just proportional to the spatial volume of Z. Moreover, many of these parameters could not be effectively manipulated by OpenTuner, since their very existence might depend on decisions made higher in the recursion. For example, if a node executes a base case, the node has no descendants, and so the parameters that would be associated with the descendants do not affect the runtime. Nevertheless, the structure of OpenTuner provides no way for OpenTuner *not* to keep twiddling these parameters and wasting time, even though they are irrelevant to the optimal plan.

Consequently, we had to restrict OpenTuner's heuristic search to the domain of base-case sizes,

and optimally coarsen the base case of recursion in the TRAP algorithm. Two OpenTuner parameters were created for each dimension of the space-time grid, one for the base-case size of zoids that impinge on the boundary, and the other for the base-case size of zoids that do not impinge on the boundary. This yielded a sizable search domain for OpenTuner, albeit smaller than Ztune's. OpenTuner uses an ensemble of search techniques that include greedy mutation, differential evolution, and hill-climbing methods.

To provide a fair comparison with Ztune, we configured OpenTuner to tune the TRAP algorithm using three search domains. Two of the search domains are relatively large in size, while the third one is small. We contend that this configuration is fair to OpenTuner, since smaller search domains allow OpenTuner to tune faster, while the larger search domains allow it to find a solution that is closer to optimal at the expense of higher tuning times. The largest search domain defined in OpenTuner is called ***IntegerValues***. The parameter lower and upper bounds were set to cover the grid size in each dimension. OpenTuner can choose any integer value within these bounds. Given a $(d+1)$-dimensional zoid Z with height h, and whose spatial widths are proportional to h, the search domain of IntegerValues has $O(h^d h^d h) = O(h^{2d+1})$ different base-case sizes, where the factor 2 is due to two parameters being tuned in each spatial dimension. In contrast, Ztune's choice domain is larger with $\Omega(2^{h/2})$ different plans. Many values within the lower and upper bounds can be redundant, however, since TRAP divides the spatial and temporal dimensions in a fixed fashion. The next larger search domain called ***ListOfNumbers*** uses the actual widths and heights of zoids traversed by Ztune as the possible values for the parameters along the corresponding dimensions. This reduces the size of the search domain for Z to $O(h^{2d} \lg h)$, since there are only $\Theta(\lg h)$ different heights in the choice domain as shown in Lemma 4. The smallest search domain is called ***PowersOfTwo***. Similar to the IntegerValues search domain, the parameter lower and upper bounds in PowersOfTwo were set to cover the grid size in each dimension, but only power-of-2 integer values can be chosen between the bounds. This reduces the size of the search domain for Z further to $O(\lg^{2d+1} h)$.

Whereas Ztune runs its natural time to autotune, OpenTuner can be run for any length of time. Figure 2.9 shows the ratios of the runtimes of OpenTuner tuned TRAP code to the runtimes of

Benchmark	*Ztune (minutes)*	*OpenTuner (hours)*
APOP	0.58	37.93
Heat1	0.33	0.15
Heat2	6.58	4.17
Life	4.40	1.32
Heat3	1.09	2.66
Heat4	0.79	>100
Heat5	6.55	48.75
LBM	0.56	0.65
Wave	51.90	>100
Heat6	5.75	>100

Table 2.10: Tuning time comparison of OpenTuner and Ztune on *Ivy Bridge*. The second column shows the Ztune tuning times in minutes. The third column shows the number of hours OpenTuner needs to autotune, so that the OpenTuner tuned code achieves the same runtime as the Ztuned code. OpenTuner data was obtained from tuning each benchmark once. Due to the time-consuming nature of the experiment, we couldn't run OpenTuner more.

Ztuned TRAPPLE code, where OpenTuner was run for $1, 2, \ldots, 32$ times longer than Ztune. In general, the longer OpenTuner tunes, faster are its tuned codes. As can be seen from the figure, despite an enormous advantage in tuning time, an OpenTuner tuned code rarely beats the Ztuned code. We also infer that the PowersOfTwo configuration for OpenTuner performs the best for all benchmarks.

Figure 2.10 shows how long OpenTuner must run to produce code that runs as fast as the Ztuned code. For example, to obtain the same runtime as the Ztuned code, OpenTuner must tune APOP for almost 38 hours, whereas Ztune ran in under a minute. To achieve comparable runtimes to the Ztuned code, OpenTuner has to tune most of the benchmarks for over an hour. Despite tuning for 100 hours, the OpenTuner tuned Heat4, Wave, and Heat6 benchmarks couldn't achieve the performance of the Ztuned code. The section concludes that the pruned-exhaustive tuning strategy in Ztune can tune divide-and-conquer stencil problems considerably faster than the heuristic tuning strategies in OpenTuner.

2.9 Related Work

Ztune fits into a rich autotuning literature, which includes a broad range of domain-specific autotuners. FFTW [27, 28] exhaustively enumerates the search domain using dynamic programming. ATLAS [67], which is an empirical autotuner for matrix multiplication, exhaustively searches the search domain. OSKI [66], which is an autotuner for sparse matrix kernels, uses exhaustive and heuristic search methods. PATUS [15] and Sepya [40] autotune tiled stencil computations using sophisticated machine-learning based search methods. SPIRAL [50] is an autotuner for DSP algorithms. Other autotuning packages include PetaBricks [2], which autotunes algorithmic choices in programs, and Active Harmony [17], which is an automated runtime tuning system. OpenTuner [3] is a framework for building domain-specific autotuners.

2.10 Summary

We have presented Ztune, a pruned-exhaustive autotuner for serial divide-and-conquer stencil computations, and described three properties namely space-time equivalence, divide subsumption, and favored dimension, which improve the performance of Ztune significantly. Pruned-exhaustive autotuning in Ztune exploits its knowledge of divide-and-conquer stencil codes, to autotune faster and produce tuned codes with similar or better runtimes than heuristic autotuning. Heuristic autotuners are nevertheless useful tools to tune a broad range of applications, where domain-specific autotuning tools might not necessarily exist.

More generally, we have presented a theoretical autotuning framework that can be used for tuning many divide-and-conquer codes in scientific computing like matrix-vector product, matrix multiplication, convolution, and dynamic-programming problems. We have extended Ztune to autotune divide-and-conquer matrix-vector product and matrix multiplication, the preliminary results of which can be found in [56].

Augmenting Ztune to autotune parallel divide-and-conquer codes is an interesting research area. Parallel codes introduce a host of issues like memory bandwidth saturation, communication overhead, and work-span optimization.

The tuning strategy in Ztune has some obvious drawbacks. Ztune doesn't extrapolate its tuning results, but autotunes every problem from scratch. Though it autotunes faster, the autotuning time can be reduced significantly by extrapolation.

Chapter 3

The Star abstraction

This chapter describes the Star class, which consists of many seemingly different computer programs. We present the Star abstraction, which is a programming model and an associated implementation to generate and execute parallel code for the Star class of programs. We also present parallel algorithms, which offer improvements over prior art, for two programs in the Star class namely — a "Trip" algorithm to solve symmetric, diagonally-dominant tridiagonal systems, and a "Wasp" algorithm to execute watershed cuts on graphs and images.

3.1 Introduction

The introduction of this thesis motivated the need for making parallel programming easier by providing simple constructs that abstract the pattern of computation and interprocessor communication, hide the complex low-level parallel programming details, and offer ease of expression. We also reviewed the history of abstractions in sequential and parallel programming. The trend [5, 20, 46, 47, 69] has been to identify patterns of computation and communication in parallel programs, and to develop abstractions for those patterns. The Star abstraction rides on this trend, and provides a simple programming model for the Star class of computer programs.

The Star class, which we introduce in this thesis, consists of computer programs that have a few key properties. Each program in the ***Star*** class

- executes an associative computation over ordered sets of data; and
- can be parallelized by splitting its computation into a sequence of three subcomputations.

Many seemingly different programs fall in the Star class. These include solving symmetric, diagonally-dominant tridiagonal systems [14, 21, 33, 35, 60], executing watershed cuts [16] on graphs, sample sort [10, 26], fast multipole computations [24], and all-prefix-sums and its various applications [9] like radix sort, solving recurrences, and dynamic processor allocation. As an example program in the Star class, consider the all-prefix-sums program, also referred to as "scan" for short. The ***scan*** operation takes as input a binary associative operator $\oplus$, and an ordered set of n elements $[a_1, a_2, \ldots, a_n]$, and returns the ordered set $[a_1, a_1 \oplus a_2, \ldots, a_1 \oplus a_2 \oplus \ldots \oplus a_n]$. For example, if $\oplus$ is addition, then scan computes the "cumulative sum" of the elements in the ordered set. The cumulative sum operation can be parallelized using a sequence of three subcomputations namely — SUM, EXCLUSIVE-CUMSUM, and CUMSUM— which we will define in Section 3.2. Besides, the parallel Star programs have the same pattern of computation and interprocessor communication. Given the reasonably large size of the Star class, a model that abstracts the pattern of computation and communication in the parallel Star programs, and automatically generates and executes parallel code will greatly improve programmer productivity in writing such programs.

At a higher level, we are interested in a model that abstracts the different steps in a parallel Star program called ***three-phase computation***, whose pseudocode is shown in Figure 3-1. Procedure THREE-PHASE-COMPUTATION operates on an array A, which represents an ordered set, partitioned into a sequence of p subarrays. It consists of three phases, namely "first", "second", and "third". In the first phase shown in lines 3–4, procedure FIRST-PHASE, which represents the first subcomputation, is executed in parallel on the processors where the subarrays A_i are stored. The results of the subcomputation are collected in an array R, which is also an ordered set, such that $R[i]$ is the result from subarray A_i. In the second phase shown in line 5, procedure SECOND-PHASE, which represents the second subcomputation, is executed locally on array R. In the third phase shown in lines 6–7, procedure THIRD-PHASE, which represents the third subcomputation, is executed in parallel on the processors where the subarrays A_i are stored. The execution of a parallel Star program using three-phase computation is illustrated in Figure 3-2.

```
THREE-PHASE-COMPUTATION(A)
1  Partition array A into a sequence of p subarrays, and let A_1, ..., A_p be
   the "references" of the subarrays.
2  let R be an array of size p.
   // Execute the first subcomputation on subarray A_i.
3  parallel for i = 1 to p
4      R[i] = FIRST-PHASE(A_i)
   // Execute the second subcomputation on the elements of R.
5  SECOND-PHASE(R)
   // Execute the third subcomputation on subarray A_i using R[i] as the
   // initial value.
6  parallel for i = 1 to p
7      THIRD-PHASE(R[i], A_i)
```

Figure 3-1: Pseudocode for a three-phase computation. Procedure THREE-PHASE-COMPUTATION takes as input an array A. The definitions of the FIRST-PHASE, SECOND-PHASE, and THIRD-PHASE procedures, which are highlighted in blue, are application-specific and are not shown.

The computations and communication in THREE-PHASE-COMPUTATION follow a pattern, which can be abstracted in a model. The three subcomputations have the following properties:

- The subcomputations follow a sequence, that is a parallel first phase followed by a serial second phase, followed by a parallel third phase;
- The second subcomputation is noncommutative;
- Communication happens only in the first and third phases; and
- Communication is asymptotically smaller than computation, that is, the size of data communicated is asymptotically smaller than the total time spent in the three subcomputations.

Communication in the first phase at line 4 involves sending a reference of subarray A_i to a processor, and receiving the result $R[i]$ back. Communication in the third phase at line 7 involves sending the value $R[i]$ and the reference to a processor. The inputs to THREE-PHASE-COMPUTATION, the definitions of procedures FIRST-PHASE, SECOND-PHASE and THIRD-PHASE, and their inputs and outputs change with the application. For example, computing the cumulative sum of an array and solving a tridiagonal system will have different inputs to THREE-PHASE-COMPUTATION, and different definitions for procedures FIRST-PHASE, SECOND-PHASE and THIRD-PHASE. We are interested in a model that takes as inputs user-defined procedures and their corresponding argu-

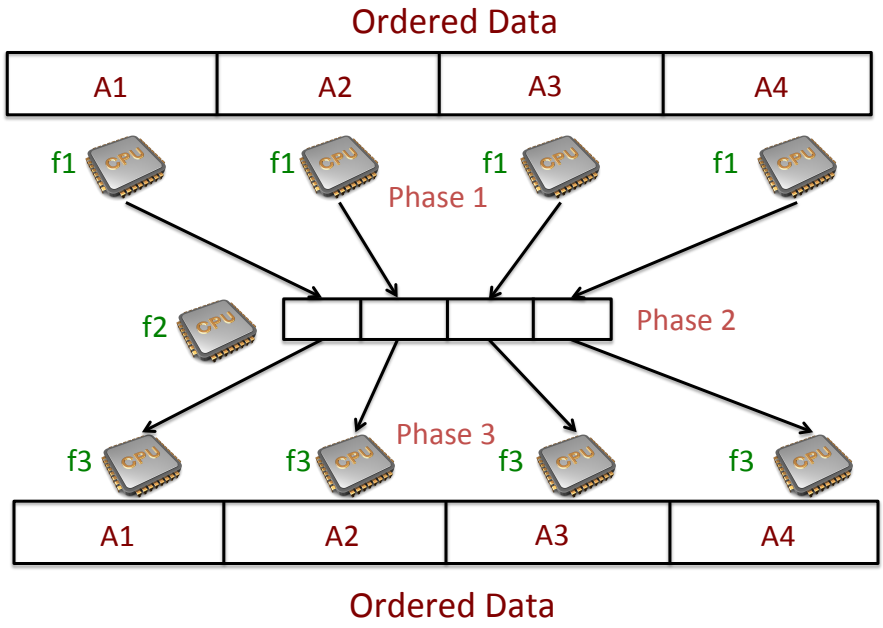


Figure 3-2: Parallel execution of a Star program. The three subcomputations are indicated as f1, f2, and f3. The data partitions are marked as A1, A2, A3, and A4. The partitions at the bottom are the same as those at the top, and have been duplicated to illutrate the flow of data.

ments for the three phases, and generates and executes THREE-PHASE-COMPUTATION code for a Star program. Such a model can hide the parallel programming details in the first and third phases, and offer a simple interface to the user to program.

We illustrated the difficulty in expressing Star programs in the MapReduce model in the introduction of this thesis, using the scan program as an example. We also highlighted the higher than optimal overheads in computation and communication that MapReduce incurs in parallelizing the scan program. Taking inspiration from the need for an efficient and easy-to-express model for THREE-PHASE-COMPUTATION, we designed a simple abstraction that is also called ***Star***. Unlike MapReduce, which is oblivious to the order of data, Star is a restricted model where the global order of data is known apriori. Hence Star avoids unnecessary overheads that MapReduce incurs in

```
function three_phase_cumsum(A)
  ref = Array(Any, nworkers())
  for i = 1 : nworkers()
    ref[i] = @spawnat workers()[i] sum(localpart(A))
  end
  R = [fetch(r) for r in ref]
  exclusive_cumsum(R)
  @sync begin
    for i = 1 : nworkers()
      @spawnat workers()[i] cumsum(R[i], localpart(A))
    end
  end
end
```

Figure 3-3: A parallel cumulative sum program based on three-phase computation in Julia language.

```
Star.third_phase(cumsum, exclusive_cumsum(Star.first_phase(sum, A)), A)
```

Figure 3-4: The parallel cumulative sum program expressed in the Star model in Julia language.

simulating THREE-PHASE-COMPUTATION. Users specify application-specific functions and their corresponding arguments for the three phases as inputs to Star, which composes them to generate and execute THREE-PHASE-COMPUTATION code. We implemented Star in the Julia [6] programming language, leveraging Julia's capabilities in expressing parallelism in code concisely, and in supporting both shared-memory and distributed-memory parallel programming alike.

How useful is writing a program in the Star class using the Star model? To illustrate the usefulness of Star, consider a parallel Julia program called three_phase_cumsum shown in Figure 3-3 that uses three-phase computation to compute the cumulative sum of the elements in the input array A. Suppose that procedures sum, exclusive_cumsum, and cumsum, which are highlighted in blue, correspond to procedures FIRST-PHASE, SECOND-PHASE, and THIRD-PHASE respectively. Other than these three procedures, the remaining parts of the program involve the messy details in writing a parallel, three-phase computation in Julia. Now consider the same cumulative sum program written in the Star model as shown in Figure 3-4. The Star version of the code is concise, enables composition of the three functions, and hides the parallel programming model of Julia completely. We will explain this code in Section 3.2. Star, in principle, can be implemented to generate code for any parallel programming model, and in any language. The more complex

the parallel programming details are, the more useful Star can be. Though Star doesn't support fault-tolerance yet, the model enables adding such features transparently, and without users having to rewrite their Star programs.

The parallel algorithms that we present for solving tridiagonal systems and for performing watershed cuts offer improvements over prior art. We present a parallel algorithm called ***Trip*** to solve a linear system of equations $Ax = b$, where A is a symmetric, diagonally-dominant tridiagonal matrix, and x and b are column vectors. Two popular algorithms for solving such systems are "recursive doubling" [60] and "cyclic reduction" [33]. Both recursive doubling and cyclic reduction are recursive algorithms, and take $\Theta(n/p + \lg p)$ time to solve a system of n unknowns using $p \leq n$ processors. In contrast, Trip is nonrecursive, and consequently takes a little longer $\Theta(n/p + p)$ time to solve. In practice where $p \ll n$, the term n/p dominates the runtime, and hence in such cases Trip's performance is comparable to that of recursive algorithms. The best serial algorithm for solving tridiagonal systems performs $8n$ floating point operations, or flops for short, to solve a system of n unknowns. Whereas recursive algorithms like cyclic reduction have a "redundancy" of 2.7 [33], that is, they perform roughly 2.7 times more flops than the serial algorithm, Trip has a redundancy of 2.1. More importantly, Trip is simple to implement on distributed-memory parallel machines, and, as we shall see, can be easily expressed in the Star model.

Among nonrecursive algorithms for tridiagonal systems, Trip's matrix partitioning strategy is roughly similar to that of the "BABE" [35] algorithm due to Gustafson and Gupta. Though BABE has the same asymptotic runtime as Trip, it incurs significantly more communication than Trip. While Trip incurs an optimal $\Theta(p)$ communication to solve a system of n unknowns on p processors, BABE incurs $\Theta(p^2)$ communication just similar to MapReduce. When $p = \sqrt{n}$, which gives the fastest parallel runtime of $\Theta(\sqrt{n})$ for both BABE and Trip, BABE's communication grows to $\Theta(n)$, which is just the serial time to solve a tridiagonal system.

Our next example for the Star model, which might seem quite unrelated to Star at the outset, is the problem of "watershed cuts" [16] on edge-weighted graphs. Watershed cuts stem from the field of "topography", and are intuitively based on the "drop of water" principle: A drop of water falling on a topographic surface follows a descending path and eventually reaches a minimum. We will

formally define the problem of watershed cuts in Section 3.4. Informally, given an edge-weighted graph $G = (V,E)$ with a set of "minima", the problem of watershed cuts involves labeling each vertex $v \in V$ with a minimum M, such that there is a "steepest descending path" from v to M. The intuition is that a drop of water placed on v will flow along such a path and reach M. Cousty et al. [16] describe a serial watershed algorithm that takes time linear in the number of edges in the input graph.

We present an algorithm called ***Wasp*** that uses the serial linear-time algorithm of Cousty et al. to parallelize watershed cuts. Similar to the Trip tridiagonal solver, Wasp is nonrecursive, has a structure amenable to the Star model, and achieves performance that scales almost linearly with the number p of processors if p is much smaller than the number of edges in the input graph. Suppose an edge-weighted graph $G = (V,E)$ is partitioned into p subgraphs such that each subgraph has roughly $\frac{|E|}{p}$ edges in its "interior", and $\sqrt{\frac{|E|}{p}}$ edges at its "boundary". To perform watershed cuts on G using p processors, Wasp takes $O(\frac{|E|}{p} + \sqrt{p|E|}\alpha(\sqrt{p|E|}))$ time, and $\Theta(\sqrt{p|E|})$ communication, where α is the slowly growing inverse Ackermann's function. For the case of a 2D square image with n pixels, these bounds correspond to $O(n/p + \sqrt{np}\ \alpha(\sqrt{np}))$ time and $\Theta(\sqrt{np})$ communication.

Parallel algorithms for watershed cuts with a structure similar to Wasp have been studied before. Zlateski [71] describes a parallel, out-of-core algorithm for creating a "segmentation hierarchy" on edge-weighted graphs, where the algorithm executes watershed cuts as an initial step followed by other post-processing steps. He doesn't provide time and communication bounds for the step involving watershed cuts, however. Unlike Wasp, which operates in three phases, Zlateski's algorithm works in two phases. When we reduced Zlateski's algorithm to perform just watershed cuts without any post processing, its time bound seems asymptotically larger than that of Wasp. Bienik et al. [7] propose a parallel algorithm that performs watershed cuts by a process of "flooding a topograhic surface from its minima". They do not provide a time complexity of the algorithm, however. We are unaware of any nonrecursive watershed algorithm that achieves similar time and communication bounds as Wasp.

The remainder of the chapter is organized as follows. Section 3.2 describes the Star model and

Program	FIRST-PHASE	SECOND-PHASE	THIRD-PHASE
Cumulative sum	Sum	"Exclusive" cumulative sum	Cumulative sum with initial value
Tridiagonal systems	Factorize and forward solve	Solve reduced system	Back solve
Watershed cuts on images	Label pixels in subimages	Merge pixel labels at the boundaries of subimages	Update pixel labels in subimages
Sample sort	Sort and send "samples"	Sort the samples and send "splitters"	Exchange elements using the splitters, and sort
Compute $u'vw$	Dot product	Sum	Scalar-Vector multiplication

Figure 3-5: Example programs in the Star class. Each row indicates the three subcomputations for a program.

gives a few examples. Sections 3.3 and 3.4 describe the Trip and Wasp algorithms respectively, and present empirical results on the performance of the algorithms across several processors. Section 3.5 provides performance comparisons with MapReduce. Section 3.6 describes related work, and Section 3.7 offers some concluding remarks.

3.2 The Star Programming model

This section presents example programs from the Star class that can be expressed in the Star model. It describes the input and output specifications for the FIRST-PHASE, SECOND-PHASE, and THIRD-PHASE functions that execute the three subcomputations. It concludes with an illustration of the execution of the Star model in the Julia language.

3.2.1 Examples

Figure 3-5 shows a few example programs in the Star class, which we explain in the following:

Cumulative sum: The ***cumulative sum*** operation takes as input an array[1] $A = [a_1, a_2, \ldots, a_n]$ of n elements, and returns the array $[a_1, a_1 + a_2, \ldots, a_1 + a_2 + \ldots + a_n]$. Procedures FIRST-PHASE, SECOND-PHASE, and THIRD-PHASE correspond to functions SUM, EXCLUSIVE-CUMSUM, and CUMSUM respectively. SUM takes as input the array A, and returns the sum of elements in A. EXCLUSIVE-CUMSUM takes as input the array A, and computes the output array $[0, a_1, a_1 + a_2, \ldots, a_1 + a_2 + \ldots + a_{n-1}]$. CUMSUM takes as input an initial value r and the array A, and computes the output array $[r + a_1, r + a_1 + a_2, \ldots, r + a_1 + a_2 + \ldots + a_n]$. Let us quickly examine how these functions operate together in THREE-PHASE-COMPUTATION to compute the cumulative sum. At the end of the first phase in line 4 of Figure 3-1, each element $R[i]$ has the sum of the elements in subarray A_i. Without loss of generality, suppose that the EXCLUSIVE-CUMSUM and CUMSUM functions store their results in place in their input arrays. At the end of the second phase in line 5, $R[1]$ has value 0, and each $R[i], 2 \leq i \leq p$ has the sum of the elements in subarrays $A_1, \ldots, A_{i-1}$. It follows from the definition of the CUMSUM function that line 7 computes the cumulative sum of the elements in subarray A_i using the correct initial value stored in $R[i]$. Similarly, the THREE-PHASE-COMPUTATION framework can be used to parallelize the scan operation for any binary associative operator $\oplus$.

Tridiagonal systems: Consider solving a linear system of equations $Ax = b$, where A is a symmetric, diagonally-dominant tridiagonal matrix, and x and b are column vectors. Suppose the matrix A is partitioned into p submatrices, and the vectors x and b into p subvectors. FIRST-PHASE factorizes a submatrix A_i into its lower and upper triangular factors L_i and U_i respectively, and simultaneously solves the lower triangular system $L_i x_i = b_i$. SECOND-PHASE solves a reduced system of equations $A'x' = b'$, where the tridiagonal matrix A' of size $p \times p$, and vectors x' and b' of size p are formed using the results from the first phase. THIRD-PHASE solves the upper triangular system $U_i x_i = x_i$, using the solution x' received from the second phase. Section 3.3 describes these steps in detail as part of the Trip algorithm.

Watershed cuts: Consider executing watershed cuts [16] on a 2D image A that is partitioned into subimages. FIRST-PHASE labels the pixels of a subimage A_i, and returns the labels of pixels

[1] We use the term "array" to represent an ordered set throughout this paper.

at the boundaries of A_i and the properties of edges that connect A_i with its neighboring subimages. SECOND-PHASE reconciles the labels of pixels at the boundaries of subimages using the results from the first phase. Finally, THIRD-PHASE updates the labels of pixels in subimage A_i using the reconciled label information received from the second phase. Section 3.4 describes these steps in detail as part of the Wasp algorithm.

Stable sample sort: Consider the stable sample sort [10, 26] operation on an array A partitioned into subarrays A_i. FIRST-PHASE sorts the elements in subarray A_i, and returns the "sample" elements. SECOND-PHASE sorts the sample elements received from the first phase, and computes the "splitter" elements. THIRD-PHASE involves exchanging elements among subarrays using the splitters from the second phase, and finally sorting the elements in each subarray.

Computing $u'vw$: Consider the operation $u'vw$ on three column vectors u, v, and w, each of which are partitioned into subvectors. FIRST-PHASE computes the "dot product" of subvectors u_i and v_i. SECOND-PHASE computes the sum of the dot products received from the first phase. THIRD-PHASE multiplies the subvector w_i with the sum received from the second phase. Note that this example doesn't strictly satisfy the properties of the subcomputations of a Star program, since the second subcomputation, which is the sum operation, is commutative.

3.2.2 Input/output specifications

Three-phase computation operates on ordered data, arranged in a sequence of partitions. To express the three-phase computation in Star, users write application-specific FIRST-PHASE, SECOND-PHASE, and THIRD-PHASE functions. The inputs and outputs of these functions, and their types, are also application-specific. We broadly describe the specifications of these functions in the following.

- FIRST-PHASE operates on its input arguments and produces outputs, some of which serve as inputs to the second phase, and hence are returned by FIRST-PHASE. The remaining outputs that are not returned by FIRST-PHASE, are stored locally in the partition where FIRST-PHASE operates. The outputs returned by FIRST-PHASE are called ***intermediate values***, and are asymptotically smaller in size than the time spent in computation in FIRST-PHASE.

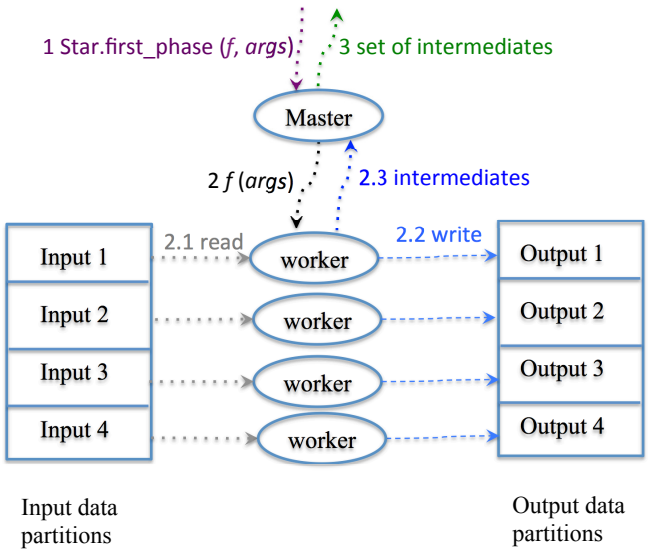


Figure 3-6: Execution of the first phase

Star executes the FIRST-PHASE function in parallel on the partitions, and collects the intermediate values in the order of partitions into an ordered set, which serves as input to SECOND-PHASE.

- SECOND-PHASE takes as input the set of intermediate values and other input parameters if any, and produces an output set R, which in turn serves as input for the third phase. The elements of R are in the same order as the partitions.

- Star distributes the elements of set R output from SECOND-PHASE to the partitions, such that, element $R[i]$ is sent as an input to the THIRD-PHASE function executed on partition i. THIRD-PHASE, which is executed in parallel on the partitions, takes as input the element $R[i]$ and possibly other input arguments, and produces outputs, which are stored locally in the partition where THIRD-PHASE operates.

3.2.3 Execution

Star can be implemented in different parallel programming models ranging from simple models for shared-memory machines to complex models for distributed-memory machines and graphics

processing units. Our implementation in the Julia language is targeted at shared-memory and distributed-memory programming models.

Star has the following steps in its execution. To begin with, the user splits the input data into partitions, and preallocates partitions for output data using Julia's internal functions. Then she loads the application-specific FIRST-PHASE and THIRD-PHASE functions on processors local to the partitions. For simplicity, we assume that each partition has a processor local to it. Star provides two functions namely ***Star.first_phase*** and ***Star.third_phase*** that abstract the pattern of computation and communication in first and third phases respectively. Figure 3-4 illustrates the usage of these functions, which are highlighted in red, in computing the cumulative sum operation.

1. One of the processors acts as a "master", and assigns work to the rest of the processors called "workers". Figure 3-6 shows the steps in the first phase of computation. Given an application-specific FIRST-PHASE function f and its input arguments *args*, the user invokes Star.first_phase on the master as Star.first_phase(f, *args*). Star.first_phase in turn executes function f on each processor passing *args* as arguments. As part of the execution, each processor reads data from its input partition, performs application-specific computation, and possibly writes data to its output partition. Star.first_phase collects the intermediate values back from each processor in the order of the partitions into an ordered set S, and returns it to the user.

2. The user invokes the SECOND-PHASE function on the master passing the set S of intermediate values and potentially other arguments as inputs, and collects the output set say R.

3. Execution of the third phase is similar to that of the first phase, but for two differences. Besides any application-specific inputs, the third phase receives the set R output from SECOND-PHASE as an input. Since the overall computation ends in the third phase, no data needs to be communicated back to the master. Given an application-specific THIRD-PHASE function g and its input arguments *args*, the user invokes Star.third_phase on the master as Star.third_phase(g, R, *args*). Star.third_phase in turn executes function g on each processor operating on a partition i, passing $R[i]$ and *args* as arguments.

$$\begin{pmatrix} a_1 & c_2 & & & \\ c_2 & a_2 & c_3 & & \\ & \ddots & \ddots & \ddots & \\ & & c_{n-1} & a_{n-1} & c_n \\ & & & c_n & a_n \end{pmatrix}$$

Figure 3-7: A symmetric tridiagonal matrix of size $n \times n$.

3.3 The Trip tridiagonal solver

This section describes the parallel Trip algorithm, which can be expressed in the star model, to solve a linear system of equations

$$Ax = b \tag{3.1}$$

where A is a symmetric, diagonally-dominant tridiagonal matrix, and x and b are column vectors. To solve a system of n unknowns using p processors, Trip takes $\Theta(n/p + p)$ time, $\Theta(p)$ communication, and performs $\Theta(n + p)$ total "work" among the p processors. Though the BABE [35] algorithm due to Gustafson and Gupta takes a similar $\Theta(n/p + p)$ parallel time as Trip, it incurs an asymptotically larger $\Theta(p^2)$ communication and performs $\Theta(n + p^2)$ total work. When $p \ll n$, the performance of Trip scales almost linearly with the number of processors in both theory and practice.

3.3.1 A serial algorithm

Before we describe Trip, let us review an efficient serial algorithm called the Thomas algorithm [65] to solve (3.1). A symmetric, tridiagonal matrix A of size $n \times n$ can be represented by 2 size n arrays a and c that constitute the diagonal and off-diagonal [2] of A respectively. Figure 3-7 shows an example symmetric tridiagonal matrix. Figure 3-8 shows the pseudocode for the Thomas algorithm in procedure TRID, which operates as follows. Lines 3–7 factorize the input matrix A into lower

[2] Though the off-diagonal has only $n-1$ elements, for ease of writing parallel code, we use a size n off-diagonal.

```
TRID(a,b,c,x)
     // Arrays a and c represent respectively the diagonal and off-diagonal
     // of a symmetric tridiagonal matrix A.
 1   n = size(a)
 2   allocate array d of length n
     // factorize A into lower and upper triangular matrices L and U
     // respectively, and solve Lx = b
 3   d[1] = a[1] ; x[1] = b[1]
 4   for i = 2 to n
 5       l = c[i]/d[i-1]
 6       d[i] = a[i] - c[i]*l
 7       x[i] = b[i] - x[i-1]*l
     // solve Ux = x
 8   x[n] = x[n]/d[n]
 9   for i = n-1 to 1
10       x[i] = (x[i] - c[i+1]*x[i+1])/d[i]
```

Figure 3-8: Pseudocode for the serial Thomas algorithm to solve (3.1). The algorithm takes as input 4 arrays a, b, c, and x of size n.

and upper triangular, bidiagonal matrices L and U respectively, and solve the lower triangular system $Lx = b$. The diagonal of L consists of all ones. Since the lower triangular system $Lx = b$ is solved immediately, TRID does not store the subdiagonal of L. The diagonal of U, which will be used later during the upper triangular solve, is stored in the array d. The superdiagonal of U is the same as the superdiagonal of A, and is given by the array c. TRID assumes that A is ***diagonally dominant***, that is, the absolute value of the diagonal element in each row i is at least the sum of the absolute values of the off-diagonal elements in row i, that is, $|a_1| \geq |c_2|$, $|a_i| \geq |c_i| + |c_{i+1}|$ for $1 < i < n$, and $|a_n| \geq |c_n|$. Due to this assumption, TRID does not consider pivoting off the diagonal of A. Lines 8–10 solve the upper triangular system $Ux = x$ to compute the solution x. Procedure TRID performs at most $8n$ floating point operations, and uses n extra space to store the diagonal of U. Besides, TRID reads at most $6n$ elements from memory and writes $3n$ elements to memory.

```
TRIP(a, b, c, x)
 1  n = size(a)
 2  Let p be the number of processors.
 3  Allocate arrays d and u each of size n.
 4  Partition array a into a sequence of p subarrays a_1, ... a_p each of
    size m = n/p.
 5  Partition arrays b, c, d, u, and x similarly.
 6  Allocate arrays a_s, b_s, c_s, p_{a_s} and p_{b_s} each of size p.
 7  parallel for k = 1 to p
 8      (a_s[k], b_s[k], c_s[k], p_{a_s}[k], p_{b_s}[k]) = LSOLVE (a_k, b_k, c_k, d_k, u_k, x_k)
 9  Allocate an array X of size p, where each element in X is an ordered
    pair.
10  X = RSOLVE (a_s, b_s, c_s, p_{a_s}, p_{b_s})
11  parallel for k = 1 to p
12      USOLVE (X[k].first, X[k].second, c_k, d_k, u_k, x_k)
```

Figure 3-9: Pseudocode for the Trip algorithm. The algorithm takes as input arrays a, b, c, and x.

3.3.2 The parallel algorithm

Next, we describe the parallel Trip algorithm to solve the system (3.1) of n unknowns using p processors. At a higher level, Trip breaks (3.1) into p independent systems of size $(n-p)/p$, and a reduced system of size p. It executes the computation in 3 steps.

1. **parallel** : {Factorize and execute a lower triangular solve on the p independent systems.}

2. **serial** : {Solve the reduced system using the results from step 1.}

3. **parallel** : {Execute an upper triangular solve on the p independent systems using the results from step 2.}

Figure 3-9 shows procedure TRIP which operates as follows. The arrays a and c represent the diagonal and off-diagonal of a symmetric tridiagonal matrix A of size $n \times n$. Arrays b and x represent the right-hand side and solution of system (3.1) respectively. Line 3 allocates an array d that represents the diagonal of the upper triangular factor U, and an array u that consists of "fill-ins" in U. An element $U(i,j)$ is a ***fill-in*** if $A(i,j) = 0$ and $U(i,j) \neq 0$. Lines 4–5 partition the arrays a, b, c, d, u, and x among the p processors, where the subarrays a_k, b_k, c_k, d_k, u_k, and x_k are local to

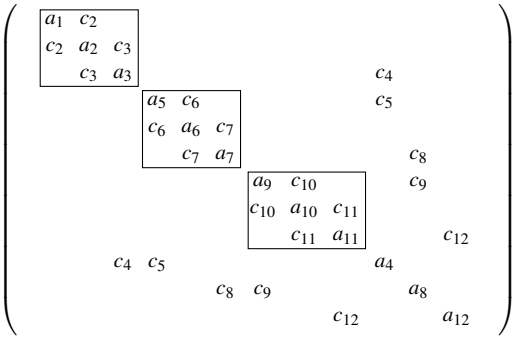


Figure 3-10: A permuted, symmetric tridiagonal matrix of size 12 consisting of 3 symmetric tridiagonal submatrices, which are shown enclosed in boxes.

processor k and have size $m = n/p, m \in \mathbf{Z}^+$. It is not necessary that the partitions be of the same size, but for ease of illustration we assume that they have the same size m. Lines 6–12 execute the 3 steps of computation listed above. We briefly summarize the steps here, and defer a detailed explanation until later in this section. We denote the elements of a_k as $[a_{k1}, \ldots, a_{km}]$. The elements a_{km}, b_{km}, and x_{km}, which are the last elements of the corresponding arrays in partition k, are called ***separators*** since they are at the boundary of partition k and separate partitions k and $k+1$. We refer to a_{km} as the separator diagonal, b_{km} as the separator right-hand side, and x_{km} as the separator unknown. Line 6 allocates arrays for assembling the reduced system of separators. The arrays a_s and b_s represent the separator diagonals $a_{km}, 1 \leq k \leq p$ and the separator right-hand sides b_{km} respectively. The array c_s represents fill-ins that connect separators a_{km} and $a_{(k+1)m}$ in the reduced system. The arrays p_{a_s} and p_{b_s} represent "updates" from partition k to partition $k-1$. Line 8 invokes procedure LSOLVE that factorizes and solves the lower triangular system in partition k, and returns a tuple that consists of the kth elements of arrays $a_s, \ldots, p_{b_s}$. Line 10 solves the reduced system of separators and returns an array X, where $X[k] = (x_{(k-1)m}, x_{km})$ is a tuple of separator unknowns at partitions $k-1$ and k respectively. Finally line 12 invokes procedure USOLVE that solves the upper triangular system in each partition k.

Procedure TRIP operates symbolically on a permuted system $(PAP^T)Px = Pb$, where P is a ***permutation matrix*** defined as $P[1, \ldots, n]^T = [1, \ldots, m-1, m+1, \ldots, 2m-1, 2m+1, \ldots, n-1, m, 2m, \ldots, n]^T$. The permutation creates p independent symmetric, tridiagonal systems of size

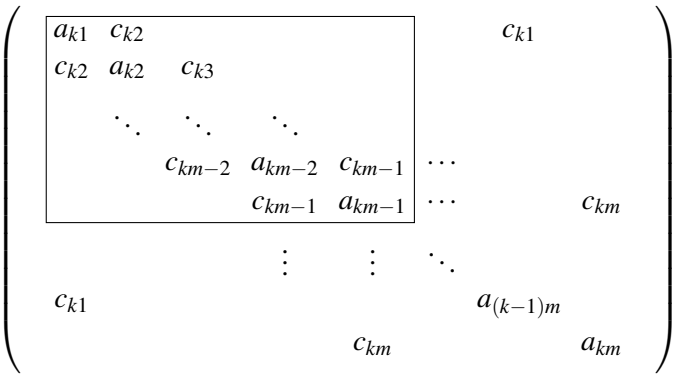


Figure 3-11: A symmetric tridiagonal submatrix A_k shown enclosed in a box, and connected to 2 separators $a_{(k-1)m}$ and a_{km}.

$m-1$ that can be solved in parallel, and a reduced symmetric tridiagonal system of size p that can be solved in serial. The matrix PAP^T consists of p symmetric, tridiagonal submatrices $A_1,\ldots,A_p$ each of size $(m-1)\times(m-1)$, which are factorized first, and p separators $a_m,\ldots,a_{pm}$, which are factorized at the end. Figure 3-10 shows an example permuted matrix (PAP^T) of size $n=12$ with $p=3$ submatrices $A_1,\ldots,A_3$ each of size $m-1=3$, which are shown enclosed in boxes, and 3 separators a_4, a_8, and a_{12}. Similarly, the permuted column vector Pb consists of p subvectors of size $m-1$, and p separator right-hand sides $b_m,\ldots,b_{pm}$. The subvectors and separator unknowns of Px are defined similarly. Factorizing the permuted matrix PAP^T changes the natural order of Gaussian elimination in the original system (3.1). This change in the order of factorization induces fill-ins in the lower and upper triangular factors L and U of PAP^T. As we shall see, the fill-ins appear in the rows of L and columns of U corresponding to the separators $a_m,\ldots,a_{(p-1)m}$. It is necessary to find the structure of fill-ins in the factors so that we can allocate sufficient space for them apriori. Moreover, fill-ins increase the number of floating point operations during factorization and solve, and hence it is important to bound the number of fill-ins.

We identify the nonzero pattern and the values of the fill-ins in the following. The submatrix A_k in partition k consists of the array $[a_{k1},\ldots,a_{k(m-1)}]$ on the diagonal, and the array $[c_{k2},\ldots,c_{k(m-1)}]$ on the off-diagonal. The submatrix A_1 is connected to separator a_m through element c_m. The submatrix $A_k, 2\le k\le p$ is connected to 2 separators $a_{(k-1)m}$ and a_{km} through elements c_{k1} and

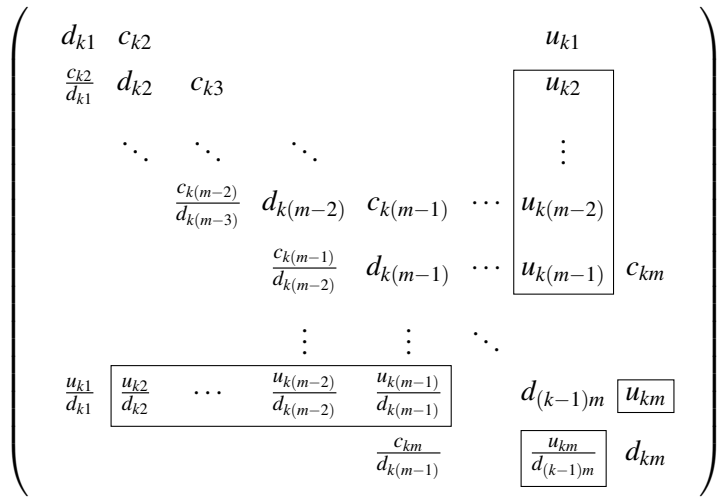


Figure 3-12: The lower and upper triangular factors, which are superimposed on the same matrix, of the augmented matrix shown in Figure 3-11. The elements shown enclosed in a box correspond to fill-ins in the factors.

c_{km} respectively. Figure 3-11 shows an example submatrix A_k and its connectivity to the separators. The algebraic equation relating the augmented matrix in Figure 3-11 and its lower and upper triangular factors is given by

$$\begin{bmatrix} A_k & m_k & \hat{m}_k \\ m_k' & a_{(k-1)m} & 0 \\ \hat{m}_k' & 0 & a_{km} \end{bmatrix} = \begin{bmatrix} L_k & & \\ l_k & 1 & \\ \hat{l}_k & l_{km} & 1 \end{bmatrix} \begin{bmatrix} U_k & u_k & \hat{u}_k \\ & d_{(k-1)m} & u_{km} \\ & & d_{km} \end{bmatrix} \tag{3.2}$$

where L_k and U_k are the lower and upper triangular factors of A_k, $m_k = [c_{k1}, 0, \ldots, 0]'$ and $\hat{m}_k = [0, \ldots, 0, c_{km}]'$ are column vectors of size $m-1$. Similarly $u_k, \hat{u}_k$ are column vectors and $l_k, \hat{l}_k$ are row vectors, all of size $m-1$, and u_{km}, l_{km} are scalars. The equations corresponding to $u_k, \hat{u}_k$ and u_{km} are given by

$$m_k = L_k u_k \tag{3.3}$$

$$\hat{m}_k = L_k \hat{u}_k \tag{3.4}$$

$$0 = l_k \hat{u}_k + u_{km} \tag{3.5}$$

Solving (3.3) yields a dense column vector $u_k = [u_{k1}, u_{k2}, \ldots, u_{k(m-1)}]'$, whose values are given by

$$u_{ki} = \begin{cases} c_{k1} & i = 1 \\ \dfrac{-u_{k(i-1)} c_{ki}}{d_{k(i-1)}} & 2 \leq i \leq m-1 \end{cases} \tag{3.6}$$

where $d_{k1}, \ldots, d_{k(m-1)}$ constitute the diagonal of U_k. Solving (3.4) gives $\hat{u}_k = \hat{m}_k$. Symmetrically, the equations for l_k and $\hat{l}_k$ can be written down and solved. It follows that l_k and $\hat{l}_k$ have identical nonzero structure as u'_k and $\hat{u}'_k$ respectively, and their values are given by $l_k = [\frac{u_{k1}}{d_{k1}}, \frac{u_{k2}}{d_{k2}}, \ldots, \frac{u_{k(m-1)}}{d_{k(m-1)}}]$ and $\hat{l}_k = [0, \ldots, 0, \frac{c_{km}}{d_{k(m-1)}}]$. Finally solving (3.5) gives

$$u_{km} = -\frac{u_{k(m-1)} c_{km}}{d_{k(m-1)}} \tag{3.7}$$

Similarly, solving the value for l_{km} gives $l_{km} = \frac{u_{km}}{d_{(k-1)m}}$. The lower and upper triangular factors of the augmented matrix in Figure 3-11 is shown in Figure 3-12, where the elements enclosed in a box correspond to fill-ins. Since each matrix $A_k, 2 \leq k \leq p$ is connected to 2 separators, A_k induces at most $2m$ total fill-ins in vectors l_k, u_k and scalars l_{km}, u_{km}. Hence, the total number of fill-ins induced by matrices A_k is at most $2pm = 2n$. Since the lower triangular system is solved immediately in each partition k, we do not need to store the contents of vector l_k. Consequently, the total space overhead is at most n.

Figure 3-13 shows procedure LSOLVE that executes the factorization and lower triangular solve steps of computation in partition k. LSOLVE operates as follows. Lines 2–5 execute the first iteration of the for loop that happens in lines 6–13. Lines 7–9 factorize $A_k = L_k U_k$ and solve $L_k \hat{x}_k = \hat{b}_k$, where $\hat{x}_k = [x_{k1}, \ldots, x_{k(m-1)}]'$ and $\hat{b}_k = [b_{k1}, \ldots, b_{k(m-1)}]'$. The diagonal of U_k is stored in the array d_k. Line 10 computes element u_{ki} using equation (3.6). Given a matrix

$$\begin{pmatrix} W & X \\ Y & Z \end{pmatrix}$$

the ***schur complement*** update of W for Z is defined as $YW^{-1}X$. The schur complement update

LSOLVE$(a_k, b_k, c_k, d_k, u_k, x_k)$

1 $m = size(a_k)$ // Assume $m \geq 2$
// Arrays $a_k[1 \ldots m-1]$ and $c_k[2 \ldots m-1]$ represent the diagonal
// and off-diagonal of a symmetric tridiagonal matrix A_k.
2 $d_k[1] = a_k[1]$; $x_k[1] = b_k[1]$; $u_k[1] = c_k[1]$
3 $r = u_k[1]/d_k[1]$
4 $p_{a_{(k-1)m}} = u_k[1] * r$ // update to $a_{(k-1)m}$
5 $p_{b_{(k-1)m}} = x_k[1] * r$ // update to $b_{(k-1)m}$
6 **for** $i = 2$ **to** $m-1$
7 $\quad l = c_k[i]/d_k[i-1]$
8 $\quad d_k[i] = a_k[i] - c_k[i] * l$
9 $\quad x_k[i] = b_k[i] - x_k[i-1] * l$
10 $\quad u_k[i] = -u_k[i-1] * l$
11 $\quad r = u_k[i]/d_k[i]$
12 $\quad p_{a_{(k-1)m}} \mathrel{+}= u_k[i] * r$
13 $\quad p_{b_{(k-1)m}} \mathrel{+}= x_k[i] * r$
14 $l = c_k[m]/d_k[m-1]$
15 $p_{a_{km}} = a_k[m] - c_k[m] * l$
16 $p_{b_{km}} = b_k[m] - x_k[m-1] * l$
17 $u_{km} = -(u_k[m-1]/d_k[m-1]) * c_k[m]$
18 **return** $(p_{a_{km}}, p_{b_{km}}, u_{km}, p_{a_{(k-1)m}}, p_{b_{(k-1)m}})$

Figure 3-13: Pseudocode for Trip's lower triangular solve. The algorithm takes as input arrays $a_k, b_k, c_k, d_k, u_k, x_k$ and returns the tuple of scalars $(p_{a_{km}}, p_{b_{km}}, u_{km}, p_{a_{(k-1)m}}, p_{b_{(k-1)m}})$.

of A_k for the separator $a_{(k-1)m}$ in equation (3.2) is given by $m_k' A_k^{-1} m_k = l_k u_k$. Line 12 computes the dot product $l_k u_k$ incrementally in each iteration. Similarly, the schur complement update of A_k for a_{km} is given by $\hat{m}_k' A_k^{-1} \hat{m}_k = \hat{l}_k \hat{u}_k = c_{km} * c_{km}/d_{k(m-1)}$. Since separator a_{km} is stored in partition k, line 15 subtracts the update from a_{km}, and stores the result in $p_{a_{km}}$. Consider the system of equations

$$\begin{bmatrix} L_k & & \\ l_k & 1 & \\ \hat{l}_k & l_{km} & 1 \end{bmatrix} \begin{bmatrix} \hat{x}_k \\ x_{(k-1)m} \\ x_{km} \end{bmatrix} = \begin{bmatrix} \hat{b}_k \\ b_{(k-1)m} \\ b_{km} \end{bmatrix} \tag{3.8}$$

corresponding to the lower triangular matrix in (3.2), where $x_{(k-1)m}, b_{(k-1)m}$ and x_{km}, b_{km} are the separators at partitions $k-1$ and k respectively. The equation corresponding to $b_{(k-1)m}$ is given by

```
RSOLVE(a_s, b_s, c_s, p_{a_s}, p_{b_s})
 1  p = size(a_s)      // size of the reduced system
 2  for k = 2 to p
 3      a_s[k-1] -= p_{a_s}[k]
 4      b_s[k-1] -= p_{b_s}[k]
 5  Allocate array x_s of size p
 6  TRID(a_s, b_s, c_s, x_s)
 7  Allocate an array X of size p, where each element in X is an ordered
    pair
 8  X[1] = (0, x_s[1])
 9  for k = 2 to p
10      X[k] = (x_s[k-1], x_s[k])
11  return X
```

Figure 3-14: Pseudocode to solve the reduced system of separators. The algorithm takes as input arrays $a_s, b_s, c_s, p_{a_s}, p_{b_s}$, and returns an array X of ordered pairs.

$l_k\hat{x}_k + x_{(k-1)m} = b_{(k-1)m}$, where the term $l_k\hat{x}_k$ is the update from partition k to the separator $b_{(k-1)m}$ that is stored in partition $k-1$. Line 13 computes the dot product $l_k\hat{x}_k$ incrementally in each iteration. Similarly, the equation corresponding to b_{km} is given by $\hat{l}_k\hat{x}_k + l_{km}x_{(k-1)m} + x_{km} = b_{km}$, where the term $\hat{l}_k\hat{x}_k$ is the update to the separator b_{km}. Since b_{km} is stored in partition k, line 16 directly subtracts the update from b_{km} and stores the result in $p_{b_{km}}$. Line 17 uses equation (3.7) to compute fill-in u_{km} that connects separators $a_{(k-1)m}$ and a_{km} in the reduced system. The "partially" updated separators $p_{a_{km}}, p_{b_{km}}$, the fill-in u_{km}, and the updates $p_{a_{(k-1)m}}, p_{b_{(k-1)m}}$ for partition $k-1$ are returned as a tuple in line 18. The tuples returned from each partition are used to solve the reduced system of separators in the next step.

Figure 3-14 shows procedure RSOLVE that solves the reduced system of separators

$$Sx_s = b_s \tag{3.9}$$

where S is a $p \times p$ symmetric tridiagonal matrix with the array a_s on the diagonal and array c_s on the off-diagonal, and x_s is the column vector of separator unknowns. The arrays a_s and b_s consist of the "partially updated" separator diagonals and separator right-hand sides respectively. The phrase

USOLVE$(x_{(k-1)m}, x_{km}, c_k, d_k, u_k, x_k)$
1 $m = size(x_k)$ // size of the partition
2 $x_k[m] = x_{km}$
3 **for** $i = m-1$ **to** 1
4 $\quad x_k[i] = (x_k[i] - c_k[i+1] * x_k[i+1] - u_k[i] * x_{(k-1)m})/d_k[i]$

Figure 3-15: Pseudocode for Trip's upper triangular solve. The algorithm takes as input arrays c_k, d_k, u_k, x_k and scalars $x_{(k-1)m}, x_{km}$.

"partially updated" means that each separator diagonal $a_{km}, 1 \leq k \leq p$ has been updated with the schur complement update from matrix A_k in partition k. Similarly, each separator right-hand side b_{km} has been updated during the lower triangular solve in partition k. The array p_{a_s} consists of the schur complement updates of each matrix $A_k, 2 \leq k \leq p$ in partition k to the separator diagonal $a_{(k-1)m}$ in partition $k-1$. Similarly, the array p_{b_s} consists of the updates from the lower triangular solve of each matrix A_k to the separator right-hand side $b_{(k-1)m}$. Lines 3–4 of RSOLVE apply the updates in arrays p_{a_s}, p_{b_s} from partition k to the corresponding separators in partition $k-1$. Note that each separator diagonal a_{km} receives updates from at most 2 matrices A_k and A_{k+1} that are connected to it. A similar argument holds for the separator right-hand side b_{km}. Line 6 invokes procedure TRID to solve (3.9). Each partition k needs the values of the separator unknowns $x_{(k-1)m}$ and x_{km} for the upper triangular solve that happens in the next step. Consequently, the separator unknowns that are relevant for each partition are packed into ordered pairs in lines 8–10, and returned in line 11.

Procedure USOLVE shown in Figure 3-15 solves an upper triangular system given by

$$\begin{bmatrix} U_k & u_k & \hat{u}_k \end{bmatrix} \begin{bmatrix} \hat{x}_k \\ x_{(k-1)m} \\ x_{km} \end{bmatrix} = \begin{bmatrix} \hat{x}_k \end{bmatrix} \tag{3.10}$$

where we are interested in solving for the column vector $\hat{x}_k = [x_{k1}, \ldots, x_{k(m-1)}]'$ using the values of the separator unknowns $x_{(k-1)m}$ and x_{km} that were computed in the previous step. Note that the

```
Star.third_phase(usolve, rsolve(Star.first_phase(lsolve, a, b, c, d, u, x)...), c, d, u, x)
```

Figure 3-16: The Trip algorithm to solve `Ax = b` expressed in the Star model in Julia. The arrays `a` and `c` indicate the diagonal and offdiagonal of `A` respectively. The arrays `d` and `u` indicate the diagonal and fill-ins, respectively of the upper triangular factor of `A`.

matrix

$$\begin{bmatrix} U_k & u_k & \hat{u}_k \end{bmatrix}$$

corresponds to the first row of the upper triangular matrix in (3.2), and we use the same definitions of U_k, u_k, and $\hat{u}_k$ as in (3.2). Line 2 of USOLVE stores the value of the separator unknown x_{km} at index m of array x_k. Lines 3–4 solve the algebraic equation corresponding to (3.10).

3.3.3 Analysis

Procedures LSOLVE, RSOLVE, and USOLVE perform at most a constant number of operations for each row in the systems they solve. Hence given p processors, Trip takes $\Theta(m+p) = \Theta(n/p+p)$ time. In specific LSOLVE, RSOLVE, and USOLVE each perform at most $12(m-1)$, $10p$, and $5(m-1)$ floating point operations (flops for short) respectively. Hence Trip performs at most $17(m-1)p+10p \leq 17n$ flops in total, and has a redundancy of $17n/8n \approx 2.1$ compared to the serial algorithm. Besides LSOLVE, RSOLVE, and USOLVE read at most $3m$, $8p$, and $4m$ elements respectively, and write at most $3m$, $5p$, and m elements respectively. Hence the total number of reads is at most $7mp+8p = 7n+8p$, and the total number of writes is at most $4mp+5p = 4n+5p$. When compared with the serial algorithm, Trip performs $n+8p$ more reads and $n+5p$ more writes.

3.3.4 Empirical results

Figure 3-16 shows the Trip code expressed concisely using the Star model in Julia. Figure 3-17 shows the performance of Trip in solving a symmetric, tridiagonal system of size 3 billion on an increasing number of cores. The single core Trip code runs roughly 6% slower than the serial Thomas algorithm since Trip performs more floating point operations. The data in Figure 3-17

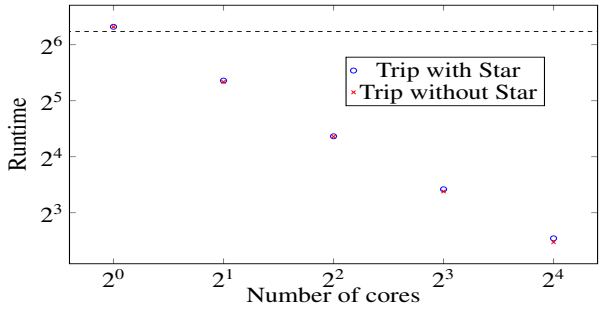


Figure 3-17: Runtimes (in seconds) of the Trip algorithm in solving a symmetric, tridiagonal system of size 3 billion on an increasing number of processor cores. The horizontal and vertical axes are set to logarithmic scale in base 2. The runtimes are the best of 5 runs. The dotted line indicates the runtime of the serial Thomas algorithm running on 1 core. The measurements were made on the machine whose specs are shown in Figure 3.1.

shows that the Trip code expressed in the Star model has a similar runtime as Trip expressed without using Star. This evidence indicates that the ease of expressing code in Star doesn't cost longer runtimes.

3.4 The Wasp watershed algorithm

This section describes the parallel Wasp algorithm, which can be expressed in the star model, to perform watershed cuts on edge-weighted graphs. We first describe the serial watershed algorithm due to Cousty et al. [16] that takes time linear in the number of edges in the input graph. We then present the parallel Wasp algorithm, analyze its runtime, and show empirical results on 2D matrices and images, which can be modeled as edge-weighted graphs. Suppose an edge-weighted graph $G = (V,E)$ is partitioned into p subgraphs such that each subgraph has roughly $\frac{|E|}{p}$ edges in its "interior", and $\sqrt{\frac{|E|}{p}}$ edges at its "boundary". To perform watershed cuts on G using p processors, Wasp takes $O(\frac{|E|}{p} + \sqrt{p|E|}\alpha(\sqrt{p|E|}))$ time, and $\Theta(\sqrt{p|E|})$ communication, where α is the slowly growing inverse Ackermann's function. In practice where $p \ll |E|$ the performance of Wasp scales almost linearly with the number of processors.

We provide a quick recap of the intuition behind watershed cuts, and make a few definitions, many of which are adapted from [16]. Watershed cuts stem from the field of topography, and

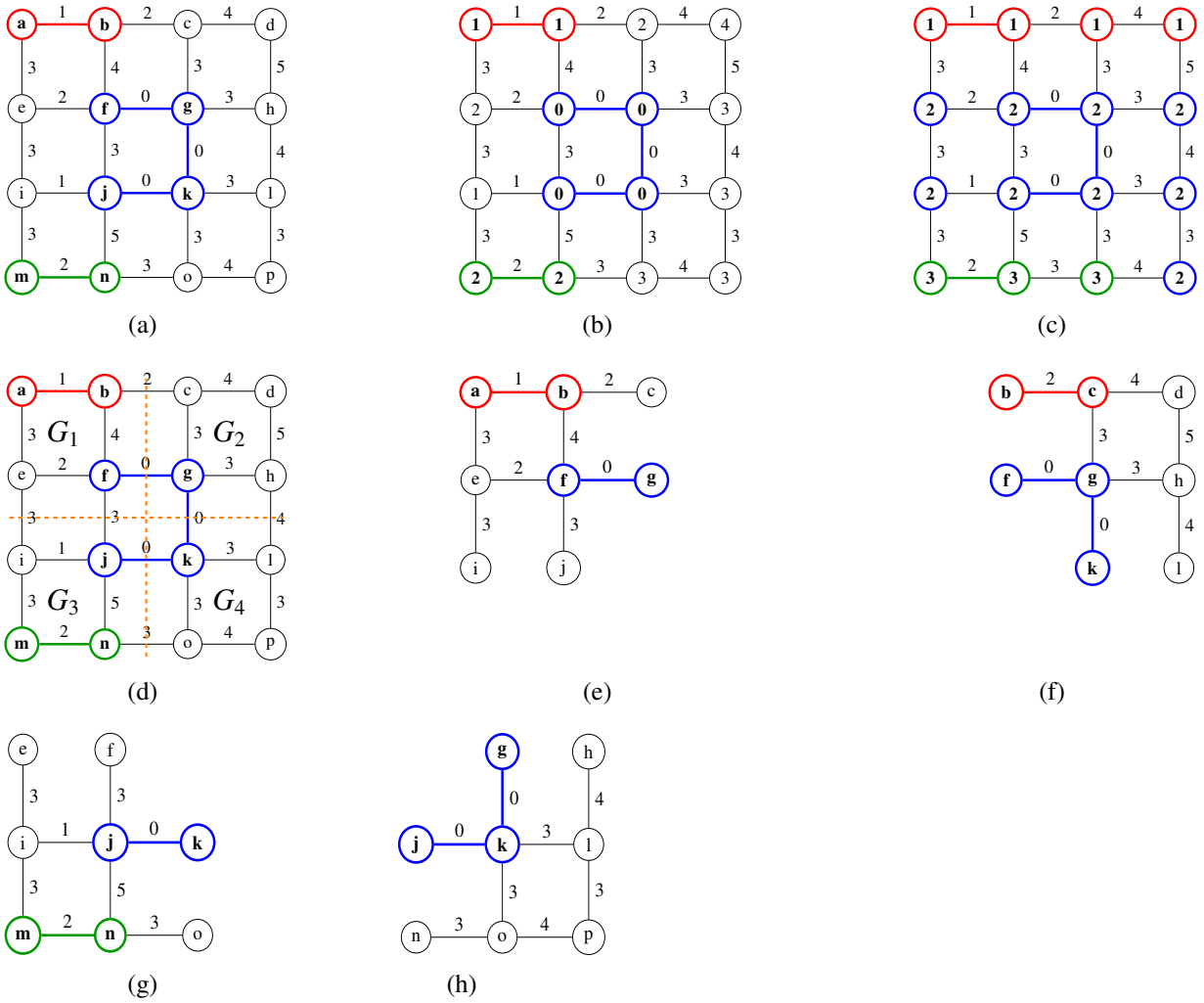


Figure 3-18: An edge-weighted graph G. (a) The minima of G. The vertices and edges in the minima are highlighted in bold and color. The vertices of G are labeled a,..., p. The minimum M_1 has vertices {a,b} and edge (a,b). The minimum M_2 has vertices {f,g,j,k} and edges {(f,g), (g,k), (j,k)}. The minimum M_3 has vertices {m,n} and edge (m,n). (b) Each vertex is labeled with its altitude. (c) Each vertex is labeled and colored with the index and color corresponding to its minimum. (d) A cut of the graph in (a) into four subgraphs $G_1, \ldots, G_4$ that are separated by the dashed lines. The edges cut by the dashed lines do not belong to any of the subgraphs. The subgraph G_1 contains vertices {a,b,e,f}, and edges {(a,b), (e,f), (a,e), (b,f)}. Subgraphs G_2, G_3, and G_4 are defined similarly. Figures (e), (f), (g), and (h) show the "augmented" subgraphs $\hat{G}_1$, $\hat{G}_2$, $\hat{G}_3$, and $\hat{G}_4$ respectively. The minima in $\hat{G}_1$, $\hat{G}_2$, $\hat{G}_3$, and $\hat{G}_4$ are highlighted in bold and color.

are intuitively based on the "drop of water" principle: A drop of water falling on a topographic surface follows a descending path and eventually reaches a minimum. Consider an edge-weighted

undirected graph $G = (V, E)$ whose edges have an associated weight given by a nonnegative weight function $w : E \rightarrow r \in \mathbf{R},\ r \geq 0$. A subgraph $M = (V' \subseteq V, E' \subseteq E)$ of G is a ***minimum*** if all the following conditions hold.

- M is connected;
- M has at least one edge, that is, $|E'| > 1$;
- All edges in M have the same weight k, that is, $w(u,v) = k,\ \forall\ (u,v) \in E'$; and
- Any edge that is not in M but incident on a vertex of M has weight strictly greater than k, that is, $w(u,v) > k,\ \forall\ (u,v) \in E - E' : u \in V' or\ v \in V'$.

For example, Figure 3-18(a) shows a graph with 3 minima. The ***altitude*** of a vertex $u \in V$ denoted by $u.altitude$ is the minimum of the weights of edges that are incident on u, that is, $u.altitude = \min\{w(u,v) : (u,v) \in E\}$. For example, Figure 3-18(b) shows the altitudes of the vertices in the example graph. Let $\pi =< u_1, \ldots, u_k >$ be a path in G, that is, $(u_i, u_{i+1}) \in E,$ for $i = 1, \ldots, k-1$. The path π is a ***steepest descending path*** if $u_i.altitude = w(u_i, u_{i+1}),$ for $i = 1, \ldots, k-1$. For example, the path <d,c,b> in Figure 3-18(a) is a steepest descending path. Let $M_1, \ldots, M_m$ be the set of minima in G. The problem of ***watershed cuts*** is to find a labeling function $\psi : V \rightarrow \{1, \ldots, m\}$ that labels each vertex $u_i \in V$ with an index j corresponding to a minimum $M_j = (V_j, E_j)$, such that

- there is a steepest descending path $\pi =< u_i, \ldots, u_k >$ from u_i to a vertex $u_k \in V_j$; and
- all the vertices $u_i \in \pi,$ for $i = 1, \ldots, k$ have the same label j.

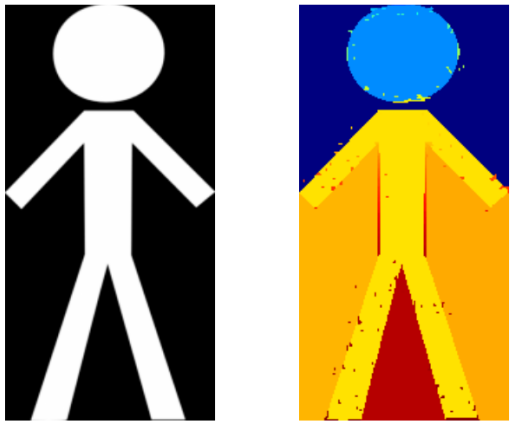
The intuition is that a drop of water placed on u will flow along such a path and reach M_i. Figure 3-18(c) shows a labeling of the vertices in the example graph with the indices of the corresponding minima. A ***stream*** $L \subseteq V$ is the set of vertices in a steepest descending path. For example, the set of vertices {d,c,b} in Figure 3-18(a) is a stream since <d,c,b> is a steepest descending path. A vertex with the highest altitude in a stream is called a ***top***, and that with the lowest altitude a ***bottom***. Suppose that $L_1, L_2 \subseteq V$ are 2 disjoint streams in G. Stream L_1 ***is under*** L_2, denoted by $L_1 \prec L_2$, if there is a steepest descending path from a bottom of L_2 to a bottom of L_1. If there is no stream in G under L, that is, if stream L contains the vertex set of a minimum in G, then L is called an ***inf-stream***.

```
WATERSHED(G, w)
 1  INITIALIZE(G, w)
 2  num_minima = 0
 3  for each u ∈ G.V
 4      if u.label == NIL
 5          (L, label, min_altitude) = STREAM(G, w, u)
 6          if (label == −1) // if L is an inf-stream
 7              num_minima++
 8              label = num_minima // create a new label
 9          for each v ∈ L
10              v.label = label

INITIALIZE(G, w)
1  for each u ∈ G.V
2      u.label = NIL // initialize labels
3      u.altitude = ∞
4      for each v ∈ G.Adj[u]
5          if u.altitude > w(u, v)
6              u.altitude = w(u, v)

STREAM(G, w, u)
 1  L = {u}
 2  L' = ∅
 3  ENQUEUE(L', u)
 4  min_altitude = u.altitude
 5  while L' ≠ ∅
 6      y = DEQUEUE(L')
 7      for each z ∈ G.Adj[y]
 8          if w(y, z) == y.altitude and z ∉ L
 9              if z.label ≠ NIL
10                  return (L, z.label, min_altitude)
11              else
12                  L = L ∪ {z}
13                  if z.altitude < y.altitude
14                      L' = ∅ ; ENQUEUE(L', z) ; min_altitude = z.altitude
15                      break
16                  ENQUEUE(L', z)
17  return (L, −1, min_altitude)
```

Figure 3-19: Pseudocode for the serial watershed algorithm adapted fom [16]. Procedures WATERSHED and INITIALIZE take as input a graph G and a nonnegative weight function w. Procedure STREAM takes as input a vertex u besides G and w, and returns a tuple containing set L, a label, and the minimum altitude among the vertices in L.



(a) (b)

Figure 3-20: Image segmentation using watershed. (a) shows the original input image. (b) shows the output of the serial watershed algorithm. The labels in the output image created by serial watershed are colored using a logarithmic scale.

3.4.1 A serial algorithm

Before we describe Wasp, let us review the serial algorithm due to Cousty et al. to perform watershed cuts on graphs. We present a brief overview of the algorithm here, and more details can be found in [16]. Figure 3-19 shows procedure WATERSHED that operates as follows. Each vertex $u \in V$ of the input graph $G = (V, E)$ has 2 attributes — $u.altitude$ and $u.label$ — that respectively store the altitude of u and the index i of a minimum M_i associated with u. Line 1 invokes procedure INITIALIZE that initializes the labels and altitudes of the vertices in G. The **for** loop in lines 3–10 assigns labels to vertices as follows. For each vertex u without a label, line 5 invokes procedure STREAM that returns a stream L such that u is a top in L, a parameter $label$ that contains the label for L, and the minimum altitude among the vertices in L. The minimum altitude, though unused in procedure WATERSHED, will be used in the parallel watershed code discussed later in this section. If L is an inf-stream then STREAM returns a label -1, and in this case lines 7–8 create a new label for L. Otherwise, there was a stream $L' \prec L$ that was already labeled, and in this case STREAM returns the label of L'. Finally lines 9–10 assign the label to vertices in L.

Procedure STREAM shown in Figure 3-19 operates as follows. Given a vertex u in graph G

and the weight function w, STREAM uses a combination of breadth-first and depth-first searches to find a steepest descending path from u to a minimum in G. The procedure maintains the following invariants:

- The set L is a stream such that the input vertex u is a top in L.
- The queue $L' \subseteq L$ maintains vertices that are the unexplored bottoms in stream L.

The invariants hold during the initialization in lines 1–3. Each iteration of the **while** loop in lines 5–16 maintains the invariants. Suppose the invariants hold at the beginning of a loop iteration. For each bottom y in L', line 8 picks an unexplored vertex z to search such that the edge (y,z) has the smallest weight among all edges incident on y. Hence edge (y,z) is on a steepest descending path, and L remains a stream after line 12 adds z to L. Since $y.altitude \geq z.altitude$ for any edge (y,z) on a steepest descending path, u remains a top in L. Hence the first invariant holds at the end of each iteration. If $z.altitude = y.altitude$, then line 16 adds z to L'. If $z.altitude < y.altitude$, then line 14 clears the contents of L', and adds the new bottom z to L'. In both the cases, the second invariant holds. The loop terminates if a vertex z on the steepest descending path has been labeled already, or if L' becomes empty. In the former case, z must be part of an inf-stream R that was labeled before. Since $R \prec L$, the vertices of L can use the same label as z. Hence line 10 returns the stream L with the label of z. In the latter case, L is an inf-stream since all the bottoms in L' have been explored. Hence line 17 returns the stream L with the label -1. It follows that the invariants hold upon loop termination. Assuming an efficient implementation, procedure WATERSHED takes time linear in the number of edges in the input graph. Figure 3-20 shows an example application of the serial watershed algorithm in segmenting a 2D image, which can be modeled as an edge-weighted graph.

3.4.2 The parallel algorithm

Next, we describe the parallel Wasp algorithm to perform watershed cuts on graphs using p processors. At a higher level, Wasp operates on a graph $G = (V,E)$ cut into p subgraphs $G_1, \ldots, G_p$. It executes the computation in 3 steps.

```
WASP(G, w)
 1  Let p be the number of processors.
 2  let graph G be cut into a sequence of p subgraphs G_1, ... G_p
 3  let the weight function w be cut similarly, such that w_k has the
    weights of edges in G_k.
 4  let G'_k denote the adjacency list of G_k, that is, the set of subgraphs
    G_j, j ≠ k adjacent to G_k, and the set of "boundary edges" incident
    on G_k and its adjacent subgraphs
 5  let m_k denote the number of "local" minima in G_k.
 6  Allocate arrays M and G' of size p.
 7  parallel for k = 1 to p
 8      m_k = PARALLEL-WATERSHED(G_k, w_k, G'_k)
 9      G'[k] = G'_k ; M[k] = m_k
10  (N, P) = MERGE (M, G')
11  parallel for k = 1 to p
12      UPDATE (N[k], P[k], G_k)
```

Figure 3-21: Pseudocode for the Wasp algorithm. The algorithm takes as input a graph G, and a nonnegative weight function w.

1. **parallel** : {Execute the serial watershed algorithm on the subgraphs.}

2. **serial** : {Merge the labels at the "boundaries" of the subgraphs using the results from step 1.}

3. **parallel** : {Update the labels of vertices in the subgraphs using the merged label data from step 2.}

Figure 3-21 shows the pseudocode for Wasp. Wasp assumes that the input graph G is cut into p subgraphs $G_1, \ldots, G_p$, and stored among p processors such that subgraph G_k is local to processor k. We will define the structure of the subgraphs shortly. We say that the minima in the subgraphs $G_1, \ldots, G_p$ are the ***local*** minima, and the minima in graph G are the ***global*** minima. The weight function w is similarly cut into p functions $w_1, \ldots, w_p$, such that w_k defines the weights of edges in G_k. We briefly summarize the 3 steps that procedure WASP executes, and defer a detailed explanation of each step until later in this section. In the first step, line 8 invokes procedure PARALLEL-WATERSHED that labels the vertices of subgraph G_k. PARALLEL-WATERSHED exe-

cutes the labeling based on the local minima in G_k, and is unaware if a local minimum M_k is also a global minimum in the graph G, or if M_k is just part of a steepest descending path that leads to a different global minimum in G. Consequently, though the labels produced by PARALLEL-WATERSHED are correct locally, they might be incorrect globally. Besides, to label the vertices of G_k, PARALLEL-WATERSHED uses labels in the set $\{1,\ldots,m_k\}$, where m_k is the number of local minima in G_k. Consequently, two vertices u and v that are in different subgraphs and in different inf-streams of G might have the same label, resulting in a ***label collision***. To resolve these anomalies, in the next step, line 10 invokes procedure MERGE that reconciles the labels at the boundaries of the subgraphs to produce a globally correct labeling for the graph G. Finally, in the third step, line 12 invokes procedure UPDATE that updates the labels in each subgraph G_k using the reconciled label information received from the previous step.

We formalize the structure of the subgraphs G_k in the following. Suppose that $\hat{G} = (V, E - C)$ is a graph obtained by removing a set of edges $C \subseteq E$ from the graph $G = (V, E)$. Given a constant $p \in \mathbf{Z}^+$, the set C is a ***p-cut*** of graph G if

- The graph $\hat{G}$ has p nonempty subgraphs $G_k = (V_k, E_k),\ 1 \leq k \leq p$ that partition $\hat{G}$, that is, $\cup_{k=1,\ldots,p} V_k = V$, $\cap_{k=1,\ldots,p} V_k = \varnothing$, $\cup_{k=1,\ldots,p} E_k = E - C$, and $\cap_{k=1,\ldots,p} E_k = \varnothing$; and
- Each edge $(u,v) \in C$ is incident on two different subgraphs G_j and G_k, $j \neq k$.

For example, the set C in Figure 3-18(d) is a 4-cut consisting of the edges cut by the dashed lines, and produces four subgraphs $G_1,\ldots,G_4$. Wasp assumes that the graph G has been cut with a p-cut producing subgraphs $G_1,\ldots,G_p$. Let $C_k \subseteq C$ be the set of edges such that each edge in C_k is incident on a vertex in subgraph G_k, that is, $C_k = \{(u,v) : (u,v) \in C$; and $u \in V_k$ or $v \in V_k\}$. Then the edges in the set C_k are the ***boundary edges*** of G_k. The vertices of G_k, on which the edges in C_k are incident, are the ***boundary vertices*** of G_k. For example, in Figure 3-18(d), the edges (b,c), (f,g), (e,i), and (f,j) are the boundary edges of subgraph G_1, and the vertices {b,f,e} are its boundary vertices. Note that a boundary edge $(u,v) \in C_k$ is not in the edge set of subgraph $G_k = (V_k, E_k)$, that is, $(u,v) \notin E_k$. Let $\hat{G}_k$ be the ***augmented graph*** obtained by adding the boundary edges in C_k to the subgraph G_k. For example, Figure 3-18(e) shows the augmented graph $\hat{G}_1$. We say that a boundary edge of G_k is a ***min boundary edge*** if it is in a minimum of the augmented graph $\hat{G}_k$. For

example, the edge (f,g) in Figure 3-18(e) is a min boundary edge. Note that the notion of a min boundary edge is relative to a specific subgraph. A boundary edge (u,v) incident on two subgraphs $G_j, G_k, j \neq k$ might be a min boundary edge for one of the subgraphs but not for the other. For example, the edge (b,c) in Figure 3-18(e) is not in the minimum of the augmented graph $\hat{G}_1$, and hence is not a min boundary edge of G_1. However, the same edge (b,c) in Figure 3-18(f) is in the minimum of the augmented graph $\hat{G}_2$, and hence is a min boundary edge of G_2. Subgraphs $G_j = (V_j, E_j)$ and $G_k = (V_k, E_k)$, $j \neq k$ are ***adjacent*** if there is an edge $(u,v) \in C$ such that $u \in V_j$ and $v \in V_k$, or vice versa. For example, G_1 and G_2 in Figure 3-18(d) are adjacent since the edges (b,c) and (f,g) are incident on G_1 and G_2. The set of boundary vertices in G_k with the same label form a ***pool*** of boundary vertices. Each pool has a ***label*** given by the label of the boundary vertices in it.

Figure 3-22 shows procedure PARALLEL-WATERSHED that executes almost the same steps as procedure WATERSHED but for a few differences. Besides labeling the vertices of subgraph G_k like WATERSHED, PARALLEL-WATERSHED also annotates the boundary edges of G_k as to whether each boundary edge is a min boundary edge or not, and associates a pool with each boundary edge. Line 1 invokes procedure INITIALIZE that initializes the labels and altitudes of vertices in G_k. Lines 2–7 traverse the boundary edges of G_k, and adjust the altitudes of the boundary vertices in G_k using the "weights" of the boundary edges. The adjacency list G'_k of subgraph G_k has the following three fields:

- $G'_k.neighbors$: the set $\{j :$ subgraph G_j is adjacent to $G_k\}$
- $G'_k.boundary\_edges$: an array of size p, where each element $G'_k.boundary\_edges[j]$ is the ordered set of boundary edges incident on G_k and G_j.
- $G'_k.pools$: the set of pools in G_k.

Suppose $e = (u,v)$ is a boundary edge incident on a boundary vertex u of G_k and a boundary vertex v of $G_j, j \neq k$. Then e is stored in the adjacency lists of both G_k and G_j. Each boundary edge has four fields, whose values differ based on which subgraph it is associated with. We assume that the boundary edge e is associated with subgraph G_k in the following.

- $e.boundary\_vertex$: a pointer to the vertex u in the vertex set of G_k.

```
PARALLEL-WATERSHED(G_k, w_k, G'_k)
 1  INITIALIZE(G_k, w_k)
 2  for each j ∈ G'_k.neighbors
 3      S_kj = G'_k.boundary_edges[j]
 4      for each edge e ∈ S_kj
 5          u = e.boundary_vertex
 6          if u.altitude > e.weight
 7              u.altitude = e.weight
 8  num_minima = 0 ; M = ∅ ; P = ∅
 9  for each u ∈ G_k.V
10      if u.label == NIL
11          (L, label, min_altitude) = STREAM(G_k, w_k, u)
12          if (label == −1) // if L is an inf-stream
13              num_minima++
14              label = num_minima // create a new label
15              M[label] = min_altitude
16          for each v ∈ L
17              v.label = label
18  for each j ∈ G'_k.neighbors
19      S_kj = G'_k.boundary_edges[j]
20      for each edge e ∈ S_kj
21          u = e.boundary_vertex
22          p = FIND(P, u.label)
23          if p == NIL
24              Allocate a pool p
25              p.label = u.label ; p.mergeable = FALSE
26              p.representative = u ; P = P ∪ {p}
27          e.pool = p ; e.min_edge = FALSE
28          if M[u.label] == e.weight // e is a min edge of G_k
29              e.min_edge = TRUE ; p.mergeable = TRUE
30  G'_k.pools = P
31  return num_minima
```

Figure 3-22: Pseudocode for the parallel watershed algorithm. Procedure PARALLEL-WATERSHED takes as input a graph G_k, a nonnegative weight function w_k, and adjacency list G'_k, and returns the number of minima in G_k. The definition of procedure FIND is not shown.

- $e.weight$: the weight of the edge given by the weight function $w(u,v)$, where v is the boundary vertex of the other subgraph $G_j, j \neq k$ on which e is incident.
- $e.pool$: a pointer to the pool associated with the vertex u.
- $e.min\_edge$: a boolean value that is TRUE if e is a min boundary edge of G_k, and FALSE

otherwise.

We assume that the fields $e.boundary\_vertex$ and $e.weight$ are precomputed before the execution of PARALLEL-WATERSHED. Line 8 initializes the number of local minima in G_k, a dictionary M, and a set P of pools. The dictionary M maps the label of each inf-stream in G_k to the smallest altitude of the vertices in that stream. Lines 9–17 execute similar code as in WATERSHED except that line 15 inserts the key *label* and the value *min_altitude* into the dictionary M. Lines 18–29 traverse the boundary edges of G_k and annotate them as follows. Line 22 invokes procedure FIND that searches the set P for a pool with the label of vertex u. If such a pool p exists, line 27 assigns the same to $e.pool$. Otherwise, lines 24–26 create and initialize a new pool p, which has the following three fields :

- $p.label$: the label of the pool.
- $p.mergeable$: a boolean value that is TRUE if a min boundary edge of G_k is incident on a boundary vertex in p, and FALSE otherwise.
- $p.representative$: a representative vertex u that can be any of the boundary vertices in p.

Line 28 queries the dictionary M with the key $u.label$. If the minimum altitude for $u.label$ equals the weight of edge e, then line 29 marks e as a min boundary edge and that the pool p is mergeable. It is fairly straightforward to show that a boundary edge e of G_k incident on the boundary vertex u is a min boundary edge of G_k, iff $M[u.label]$ equals $e.weight$. Finally, line 30 assigns the set P of pools to $G'_k.pools$, and line 31 returns the number of local minima in G_k.

Figure 3-23 shows procedure MERGE that reconciles the labels in the subgraphs to produce a globally correct labeling for the graph G. To resolve label collisions in the subgraphs, MERGE computes the cumulative sum of the number of local minima in subgraphs $G_1, \ldots, G_p$. The cumulative sum for subgraph G_k is given by the formula $N[k] = 0$, $k = 1$ and $N[k] = \sum_{j=1}^{k-1} m_j$, $2 \leq k \leq p$, where m_j is the number of local minima in subgraph G_j. Lines 3 and 5 compute the cumulative sum. The reconciliation of labels in the subgraphs involves merging the pools in adjacent subgraphs G_j and $G_k, j \neq k$ by examining if each boundary edge incident on G_j and G_k is a min boundary edge of the subgraphs. To avoid label collisions during merge, line 7 increments the label of each pool in subgraph G_k with the cumulative sum $N[k]$. For efficient merging of pools,

```
MERGE(M, G')
 1  p = size(M) // p is the number of processors
 2  Allocate array N of size p.
 3  N[1] = 0
 4  for k = 2 to p
 5      N[k] = N[k-1] + M[k-1]
 6      for each pool l ∈ G'_k.pools
 7          l.label += N[k]
 8          MAKE-SET(l)
 9  for k = 1 to p
10      for each j ∈ G'_k.neighbors
11          if MERGED(k, j) == FALSE
12              S_kj = G'_k.boundary_edges[j]
13              S_jk = G'_j.boundary_edges[k]
14              MERGE-POOLS (S_kj, S_jk)
15              MERGED(k, j) = TRUE ; MERGED(j, k) = TRUE
16  Allocate array P of size p
17  for k = 1 to p
18      P' = ∅
19      for each pool l ∈ G'_k.pools
20          r = FIND-SET(l)
21          if l ≠ r // label for pool l has changed
22              l.label = r.label
23              P' = P' ∪ {l}
24      P[k] = P'
25  return (N, P)
```

Figure 3-23: Pseudocode for the merge algorithm. Procedure MERGE takes as input arrays M and G' that have the number of local minima and the annotated adjacency list of each subgraph respectively. It returns a tuple of two arrays N and P, where $N[k]$ is the cumulative sum of the number of local minima in subgraphs $G_1, \ldots, G_{k-1}$, and $P[k]$ is the set of pools with new labels in subgraph G_k. The definitions of procedures MAKE-SET, MERGED, and FIND-SET are not shown.

MERGE uses the disjoint-set data structure due to Tarjan [63, 64]. The data structure maintains a collection of disjoint sets, where each set consists of pools that have been merged. Each element in a set can be thought of as a pointer to the corresponding pool in $G'_k.pools$. The disjoint-set data structure provides the following three functions:

- MAKE-SET(l) creates a set with the pool l.
- FIND-SET(l) returns the "representative" pool of the set containing pool l.

- UNION(l_1, l_2) unites or merges the sets that contain l_1 and l_2, say S_1 and S_2, into a new set S that is the union of S_1 and S_2. The pools of set S now have a single label, given by the label of the representative pool of S.

Line 8 creates a set with the pool l as its member. Lines 9–15 traverse the adjacency list of each subgraph $G_k, 1 \leq k \leq p$. For each pair (G_k, G_j) of adjacent subgraphs, line 14 invokes procedure MERGE-POOLS that merges the pools in G_k and G_j using the boundary edges S_{kj} and S_{jk} passed as arguments. Once all the pools in the subgraphs have been merged, lines 17–24 find the set of pools in each subgraph $G_k, 1 \leq k \leq p$ whose labels have changed because of the merge. For each pool l that belongs to a set, say S, in the disjoint-set data structure, line 22 assigns a new label given by the label of the representative pool of S. Finally, line 25 returns a tuple containing the array N of cumulative sums, and the array P of pools with new labels.

Figure 3-24 shows the pseudocode for procedure MERGE-POOLS that merges the pools of two adjacent subgraphs G_k and G_j using the boundary edges that are incident on them. MERGE-POOLS assumes that the edges in the input sets S_{kj} and S_{jk} have the same order, that is, $e_k = S_{kj}[i]$ and $e_j = S_{jk}[i]$ represent the same boundary edge $e = (u_k, u_j)$ incident on vertices $u_k = e_k.boundary\_vertex$ and $u_j = e_j.boundary\_vertex$. The procedure merges the pools $l_k = e_k.pool$ and $l_j = e_j.pool$ associated with the vertices u_k and u_j respectively, depending on whether the pools are mergeable and whether the edge e is a min boundary edge of the subgraphs. There are 9 different cases to consider as shown in Figure 3-25. Edge e is either a min boundary edge of G_k, that is, e is in a minimum of the augmented graph $\hat{G}_k$, or it is not. Similarly, e is either a min boundary edge of G_j, or it is not. In case 1, e is neither a min boundary edge of G_k nor of G_j. For example, consider the boundary edge (b,g) in Figure 3-26, which is neither a min boundary edge of G_1 nor of G_3. The pools l_k and l_j should not be merged since vertices u_k and u_j might be associated with different minima in G_k and G_j.

Procedure MERGE-POOLS maintains the following invariants for each set S in the disjoint-set data structure, and the invariants are useful in understanding the other merge cases.

1. The vertices of pools in S are associated with the same minimum, say M_s. That is, each

```
MERGE-POOLS(S_kj, S_jk)
 1  for i = 1 to size(S_kj)
 2      e_k = S_kj[i] ; e_j = S_jk[i]
 3      l_k = e_k.pool ; l_j = e_j.pool
 4      if e_k.min_edge == TRUE and e_j.min_edge == FALSE
            // Edge is a min edge of G_k but not a min edge of G_j
 5          if l_k.mergeable == TRUE
 6              r_k = FIND-SET(l_k) ; r_j = FIND-SET(l_j)
 7              CLEAR-MERGEABLE-POOLS(r_k)
 8              if r_k ≠ r_j
 9                  UNION(r_k, r_j)
10      elseif e_k.min_edge == FALSE and e_j.min_edge == TRUE
            // Edge is not a min edge of G_k but a min edge of G_j
11          if l_j.mergeable == TRUE
12              r_k = FIND-SET(l_k) ; r_j = FIND-SET(l_j)
13              CLEAR-MERGEABLE-POOLS(r_j)
14              if r_k ≠ r_j
15                  UNION(r_k, r_j)
16      elseif e_k.min_edge == TRUE and e_j.min_edge == TRUE
            // Edge is a min edge of both G_k and G_j
17          if l_k.mergeable == TRUE or l_j.mergeable == TRUE
18              r_k = FIND-SET(l_k) ; r_j = FIND-SET(l_j)
19              if l_k.mergeable == TRUE and l_j.mergeable == FALSE
20                  CLEAR-MERGEABLE-POOLS(r_k)
21              elseif l_k.mergeable == FALSE and l_j.mergeable == TRUE
22                  CLEAR-MERGEABLE-POOLS(r_j)
23              if r_k ≠ r_j
24                  UNION(r_k, r_j)
```

Figure 3-24: Pseudocode for merging the pools of adjacent subgraphs. Procedure MERGE-POOLS takes as input two sets S_{kj} and S_{jk}. The set S_{kj} consists of boundary edges of G_k that are incident on G_j, and S_{jk} consists of boundary edges of G_j that are incident on G_k.

vertex u of a pool in S has a steepest descending path to a vertex v in M_s.

2. Let $S' \subseteq S$ be the set of pools in S that are mergeable, that is, $l.mergeable = \text{TRUE}$ for each $l \in S'$. Then each min boundary edge incident on a boundary vertex of a pool $l \in S'$ is in the minimum M_s.

Cases 2 and 3 occur when (u_k, u_j) is a min boundary edge of G_k but not of G_j. Lines 4–9 of MERGE-POOLS handle these cases. For example, consider the edge (j,k) in Figure 3-26, which is

Case	u_k.*min_edge*	u_j.*min_edge*	l_k.*mergeable*	l_j.*mergeable*	*Merge*
1	FALSE	FALSE	—	—	NO
2	TRUE	FALSE	TRUE	—	YES
3	TRUE	FALSE	FALSE	—	NO
4	FALSE	TRUE	—	TRUE	YES
5	FALSE	TRUE	—	FALSE	NO
6	TRUE	TRUE	FALSE	FALSE	NO
7	TRUE	TRUE	TRUE	FALSE	YES
8	TRUE	TRUE	FALSE	TRUE	YES
9	TRUE	TRUE	TRUE	TRUE	YES

Figure 3-25: Mergeability of pools. A "—" sign indicates that the value can be TRUE or FALSE. The header *Merge* indicates whether the pools l_k and l_j can be merged.

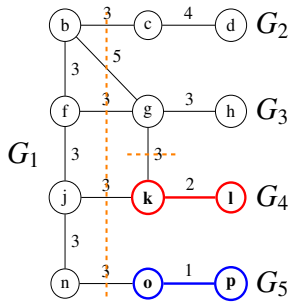


Figure 3-26: An edge-weighted graph G partitioned into five subgraphs $G_1, \ldots, G_5$. The subgraph G_1 contains vertices {b,f,j,n}, and edges {(b,f), (f,j), (j,n)}. G_2 contains vertices {c,d}, and the edge (c,d). G_3 contains vertices {g,h}, and the edge (g,h). G_4 contains vertices {k,l}, and the edge (k,l). G_5 contains vertices {o,p}, and the edge (o,p). The edges cut by the dashed lines are the boundary edges, and do not belong to any of the subgraphs. Graph G has two minima that are highlighted in bold and color. The vertices {k,l} and edge (k,l) constitute the first minimum, say M_4, and the vertices {o,p} and edge (o,p) constitute the second minimum, say M_5.

a min boundary edge of G_1 but not of G_4. Since (u_k, u_j) is not a min boundary edge of G_j, there must be a steepest descending path from u_j to a minimum, say M_j, such that the weight of an edge (u,v) in M_j is strictly smaller than the weight of (u_k, u_j). Note that if $w(u,v) \geq w(u_k, u_j)$, then (u_k, u_j) will be a min boundary edge of G_j. Since there is a steepest descending path from u_k to the minimum M_j through the edge (u_k, u_j), the pool l_k should be merged with l_j as long as l_k is still mergeable, that is, $l_k.mergeable =$ TRUE. Suppose l_k is mergeable. Let S_k and S_j be the sets containing pools l_k and l_j respectively in the disjoint-set data structure. Let $S'_k \subseteq S_k$ be the set of

mergeable pools in S_k. Let E'_k be the set of min boundary edges, which includes (u_k, u_j), incident on the vertices of pools in S'_k. By invariant 1 and 2 of MERGE-POOLS, all the edges in E'_k are in the same minimum, and they will cease to be min boundary edges immediately after merging S_k and S_j. Each edge $(u'_k, u'_j) \in E'_k$ that has not been merged so far, needs to be marked that it is no longer a min boundary edge. Otherwise, (u'_k, u'_j) might be on a steepest descending path leading to a different minimum, say $M'_j \neq M_j$, causing a merger between pools, say l'_k and l'_j, associated with vertices u'_k and u'_j respectively. Such a merger will violate invariant 1, since 2 distinct minima M_j and M'_j will not have a steepest descending path between them. Since it might be expensive to mark each unmerged edge in E'_k, we just mark the pools in S'_k as unmergeable. For example, the number of boundary edges in a subgraph can be much larger than the number of pools in it. To that effect, we augment each set in the disjoint-set data structure to store the list of mergeable pools in it. Each pool in the list can be thought of as a pointer to the corresponding pool in the adjacency list of the subgraph. Line 7 of MERGE-POOLS invokes procedure CLEAR-MERGEABLE-POOLS that marks each pool in the list as unmergeable, and finally clears the list of mergeable pools in the set. During the UNION operation, the lists of mergeable pools in the two sets being united are combined into one list.

Lines 10–15 of MERGE-POOLS handle cases 4 and 5, which are symmetric to cases 2 and 3. Lines 16–24 handle cases 6-9. In case 6, since pools l_k and l_j are unmergeable, they might be associated with different minima, and hence should not be merged. Case 7 is similar to case 2, and case 8 is similar to case 4. Finally, in case 9, the pools l_k and l_j can be merged, since they are both mergeable and the edge (u_k, u_j) is a minimum of both the subgraphs.

Figure 3-28 shows procedure UPDATE that updates the labels in subgraph G_k using the reconciled label information from the merge. The input parameter $offset$ is the cumulative sum of the number of local minima in subgraphs $G_1, \ldots, G_{k-1}$. Lines 1–2 increment the labels of vertices with the cumulative sum so that the labels are free from collision. Lines 3–15 update those vertices in G_k, whose labels have changed because of the merge. For each pool l with a new label, lines 8–15 execute a breadth-first traversal of G_k from the representative boundary vertex u of pool l. The breadth-first traversal visits all vertices of G_k with the old label, and updates them with the

```
CLEAR-MERGEABLE-POOLS(r)
1  if r ≠ NIL
2     for each pool l ∈ r.mergeable_pools
          // Mark each pool that it is no longer mergeable
3        l.mergeable = FALSE
4     r.mergeable_pools = ∅ // Clear the list of mergeable pools
```

Figure 3-27: Pseudocode for clearing the mergeable pools in a set. Procedure CLEAR-MERGEABLE-POOLS takes as input the root node r of a set whose mergeable pools are to be cleared.

```
UPDATE(offset, P_k, G_k)
 1  for each u ∈ G_k.V
 2     u.label += offset
 3  for each pool l ∈ P_k
 4     u = l.representative
 5     new_label = l.label
 6     old_label = u.label
 7     if old_label ≠ new_label
 8        Q = ∅
 9        ENQUEUE(Q, u)
10        while Q ≠ ∅ // do a breadth-first traversal
11           v = DEQUEUE(Q)
12           v.label = new_label
13           for each w ∈ G_k.Adj[v]
14              if w.label == old_label
15                 ENQUEUE(Q, w)
```

Figure 3-28: Pseudocode for updating the labels in a subgraph. Procedure UPDATE takes as input a subgraph G_k, a nonnegative integer $offset$, and a set P_k of pools in G_k whose labels have changed due to the merge.

new label.

3.4.3 Analysis

We analyze the time, space, and communication complexity of the three steps of Wasp in the following. We assume that the input graph $G = (V,E)$ is connected, and that the subgraphs $G_k = (V_k, E_k), 1 \leq k \leq p$ have roughly the same number of edges, that is, $|E_k| = |E|/p$. We further

assume that each subgraph G_k has roughly $\sqrt{|E_k|}$ boundary edges. These assumptions hold for 2D and 3D images, which can be modeled as edge-weighted graphs. Cutting arbitrary graphs to meet these assumptions might be tricky, and is beyond the scope of this thesis. Lines 9–17 of PARALLEL-WATERSHED take $\Theta(|E_k|)$ time to execute the watershed algorithm on the vertices of G_k. The dictionary M maintained by PARALLEL-WATERSHED has size at most the number of vertices in G_k. Hence insertions and queries of the dictionary can be executed in constant time, by using $|V_k|$ space to store the labels. Similarly, finding a pool for a given label in line 22 can also be executed in costant time, by using $|V_k|$ space to store the labels. Hence PARALLEL-WATERSHED takes $\Theta(|E_k|)$ time in total. The number of pools created by PARALLEL-WATERSHED is at most the number of boundary edges in G_k. Since each pool and boundary edge take a constant amount of space, the total space consumption of PARALLEL-WATERSHED is $O(|V_k| + \sqrt{|E_k|})$, excluding the space required to store the input graph G_k and the weight function w_k. Procedure MERGE executes a MAKE-SET and FIND-SET operation for each pool. Procedure MERGE-POOLS executes at most two FIND-SET operations and one UNION operation for each pair of boundary edges. Hence the total number of MAKE-SET, FIND-SET, and UNION operations executed by the two procedures is $O(\sum_{k=1}^{p} \sqrt{|E_k|}) = O(\sqrt{p|E|})$. Using the union-by-rank and path-compression heuristics [63, 64], the total time to execute these operations is given by $O(\sqrt{p|E|}\alpha(\sqrt{p|E|}))$, where α is the slowly growing inverse Ackermann's function. In addition, MERGE-POOLS invokes CLEAR-MERGEABLE-POOLS to clear the mergeable pools in a set. We augment each set in the disjoint-set data structure to store a linked list of mergeable pools in it. Each pool enters the list once, leaves the list if it becomes unmergeable, and never enters the list again. Hence the total time to clear the list of mergeable pools in all sets is $O(\sqrt{p|E|})$. During the UNION operation, the lists of mergeable pools in the two sets being united can be combined into one list in constant time. Hence the total time in combining the lists of mergeable pools is also $O(\sqrt{p|E|})$. Summing up, the total time taken by MERGE and MERGE-POOLS is $O(\sqrt{p|E|}\alpha(\sqrt{p|E|}))$. The procedure MERGED, which tests if pools of two subgraphs G_k and G_j have been merged, can be executed in constant time using p^2 bits. The set of pools with new labels returned by MERGE has at most $O(\sqrt{p|E|})$ pools in it. Hence the total space consumption of MERGE is $O(\sqrt{p|E|} + p^2)$. Finally,

```
Star.third_phase(update, merge(Star.first_phase(parallel_watershed, G, w, G')...), G)
```

Figure 3-29: The Wasp algorithm to execute watershed on graph `G` expressed using the Star model in Julia language. The parameter `w` indicates the nonnegative weight function associated with the edges of `G`. The parameter `G'` indicates the adjacency list of the subgraphs of `G`.

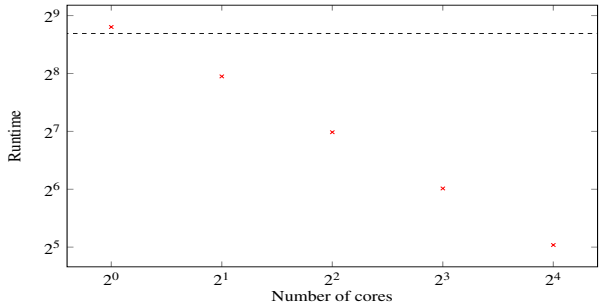


Figure 3-30: Runtimes (in seconds) of the Wasp algorithm in executing watershed on a 2D random matrix of size 16384×16384 on an increasing number of processor cores. The horizontal and vertical axes are set to logarithmic scale in base 2. The runtimes are the best of 5 runs. The dotted line indicates the runtime of the serial watershed algorithm running on 1 core. The measurements were made on the machine whose specs are shown in Figure 3.1.

procedure UPDATE takes $O(|V_k| + |E_k|)$ time and $O(|V_k|)$ space to increment and update the labels of vertices. Assuming $|E_k| \geq |V_k|$, the total time taken by the three steps given p processors is $O(|E|/p + \sqrt{p|E|}\alpha(\sqrt{p|E|}))$. The total space consumption on all p processors, excluding the space required to store the input graph and the weight function, is $O(|V| + \sqrt{p|E|} + p^2)$. The communication between the first and second steps involves the pools and annotated boundary edges from each processor. Similarly, communication between the second and third steps involves the pools with new labels due to merge. Since the number of pools is at most the number of boundary edges in a subgraph, the total communication overhead of Wasp is $O(\sqrt{p|E|})$.

3.4.4 Empirical results

Figure 3-29 shows the Wasp code expressed concisely using the Star model in Julia. Figure 3-30 shows the performance of Wasp in executing watershed on a random 2D matrix of size 16384×16384, which can be modeled as an edge-weighted graph, on an increasing number of cores. Each

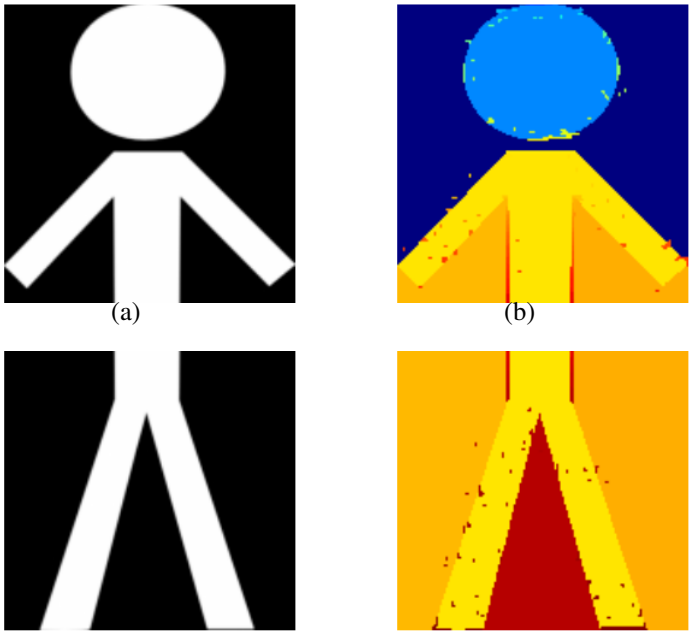


Figure 3-31: Image segmentation using watershed. Figures (b) and (d) show the output of Wasp on the original input image in Figure 3-20 partitioned into two subimages (a) and (c). Wasp was executed using 2 processors, each processor operating on a subimage. The labels in the output images created by watershed are colored using a logarithmic scale.

element in the matrix, say A, represents a node in the graph. Each node at index (i, j) is adjacent to its 4 neighbors whose indices are $(i-1, j), (i+1, j), (i, j-1)$, and $(i, j+1)$. The weight of an edge incident on nodes at indices (i, j) and $(i+1, j)$ is the absolute value of the difference between the two elements given by $|A(i, j) - A(i+1, j)|$. The single core Wasp code runs roughly 8% slower than the serial watershed code since Wasp incurs additional overhead in executing the second and thrid steps, which are redundant for a single core execution. Figure 3-31 shows an example application of watershed in segmenting images. The input image is modeled as a 2D matrix, which in turn can be modeled as an edge-weighted graph as described earlier. Each pixel represents an element in the matrix, and the pixel value represents the value of the corresponding element in the matrix. The quality of the output image from watershed typically improves, if the degree of each

node is increased by adding edges to the northeast, northwest, southeast, and southwest neighbors.

3.5 Comparison with MapReduce

This section compares the performance of Star with that of MapReduce in computing the cumulative sum operation. We describe a MapReduce solution for three-phase computation that takes $O(\frac{n}{p}\lg\frac{n}{p} + p\lg p)$ time and $O(p^2)$ communication to compute the cumulative sum of an array of n elements using p processors. A MapReduce solution implemented in Julia, where we discount the sorting overheads of the MapReduce framework, still takes 3 times longer than Star to compute the cumulative sum of an array of 10 billion elements using 16 processors.

The MapReduce solution consists of a sequence of two steps. The first step executes the first and second phases of three-phase computation, and the second step executes the third phase. The solution uses MapReduce's combine function to aggregate data output from MapReduce's map function, and minimize communication over the network. Without the combine function, the entire output from the map function will be communicated to MapReduce's reduce function, and this communication overhead will outweigh the benefits of parallel execution. Suppose the input array A has n elements, arranged in p partitions, each with $k = n/p$ elements. Let a_{ij} be the ith element in the jth partition. We assume that the p input partitions are distributed among a set of machines, such that, each machine has $m \ll p$ processors, and each processor operates on a partition stored locally.

In the first step, the map and combine functions have the following signatures. For $1 \leq i \leq k$, $1 \leq j \leq p$,

$$\begin{aligned} map &: ((i,j),a_{ij}) \rightarrow (j,(*,a_{ij})) \\ combine &: (j,[(*,a_{1j}),(*,a_{2j}),\ldots,(*,a_{kj})]) \rightarrow (1,(j,b_j = \textstyle\sum_{i=1}^{k} a_{ij})) \end{aligned}$$

where "*" is any integer, b_j is the sum of the elements in partition j, and the special key "1" is to make sure that the same MapReduce worker receives all the (j,b_j) pairs. Note that the "*" integer placeholder is to satisfy the MapReduce requirement that the map and combine functions

have the same output types. The MapReduce reduce function takes as input the key "1" and the list of (j, b_j) pairs. Since the list of (j, b_j) pairs might not be sorted in the increasing order of the partitions j, the reduce function first sorts the list on j. Then it computes the exclusive cumulative sum on the values b_j of the sorted list. The output of the reduce function is an array R of size p, where $R[1] = 0$ and $R[j], 2 \leq j \leq p$ has the sum of elements in partitions $1, \ldots, j-1$. Array R serves as input for the next MapReduce step, where $R[j]$ is used as an initial value to compute the cumulative sum of elements in partition j.

The second step begins with each processor reading the array R and loading its contents into cache. Since $p \ll n$, we assume that R fits in memory. The map function has the following signature. For $1 \leq i \leq k$, $1 \leq j \leq p$,

$$map \;:\; ((i,j), a_{ij}) \;\rightarrow\; (j, (i, a_{ij}))$$

The combine function receives the list of (i, a_{ij}) pairs for a given partition j. Since the list of (i, a_{ij}) pairs might not be sorted in the increasing order of indices i, the combine function first sorts the list on i. Then, it computes the cumulative sum of the elements a_{ij} of the sorted list, using $R[j]$ as the initial value. A MapReduce reduce function is not needed in the second step.

3.5.1 Analysis

We analyze the time and communication complexity of the MapReduce solution in the following. In the first MapReduce step, the map and combine functions each take $O(n/p)$ time, and the reduce function takes $O(p \lg p)$ time using a standard sorting algorithm. The MapReduce framework incurs asymptotically more time, however. A MapReduce framework like Hadoop [4] collects the intermediate key-value pairs emitted by the map function for a given partition into a list, and sorts the list based on the keys. It then invokes the combine function for each unique key in the list, passing the key and the set of values associated with that key as inputs. Consequently, the total time spent in the first MapReduce step increases to $O(\frac{n}{p} \lg \frac{n}{p} + p \lg p)$ due to the sorting overhead. Similarly, the total time spent in the second MapReduce step is $O(\frac{n}{p} \lg \frac{n}{p})$ due to the

Manufacturer	Intel
CPU	Xeon E5-2676 v3
Clock	2.4 GHz
Hyperthreading	Enabled
Turbo Boost	Disabled
Processor cores	20
Sockets	2
L1 data cache/core	32 KB
L2 cache/core	256 KB
L3 cache/socket	30 MB
DRAM	158 GB DDR3
Operating system kernel	Linux 3.13.0
Advanced Vector Extensions	Yes

Table 3.1: Specifications of the machine used in benchmarking. We disabled Turbo Boost, where possible, to enhance the reliability of time measurements.

sorting overheads in the combine function and the MapReduce framework. Hence, the sequence of two MapReduce steps takes $O(\frac{n}{p}\lg\frac{n}{p} + p\lg p)$ time. The total communication overhead is $O(p^2)$, since array R, in its entirety, is read by each processor in the second MapReduce step.

Besides the theoretical analysis, we also compared the practical performance of the MapReduce solution with that of Star in computing the cumulative sum operation. Comparing the performance of MapReduce code written in Hadoop with that of Star code written in Julia can be ambiguous, however. Consequently, we implemented the MapReduce solution in Julia, and simulated the sequence of MapReduce's map, combine, and reduce function calls. The simulated MapReduce solution took 3 times longer than Star to compute the cumulative sum of an input array of 10 billion elements using 16 processor cores on a machine, whose specs are shown in Figure 3.1. Since the simulation was for a specific operation, namely cumulative sum, we avoided MapReduce's overheads in sorting the intermediate key-value pairs output by the map function. Besides, we avoided sorting the list of (i, a_{ij}) pairs in the combine function of the second MapReduce step. If the sorting overheads were included, the MapReduce solution would have taken even longer.

3.6 Related work

Parallel programming models like Cilk [43], and MPI [53] offer abstractions that hide the low-level details of parallel programming. Asanovic et al. [5] identify 13 different parallel programming patterns, which are referred to as "dwarfs". [1] describes a "Our pattern language" that defines several categories of patterns. A couple of books [46, 47] offer a comprehensive study on parallel programming patterns. MapReduce [20] offers a simple and powerful programming model to process massive amounts of data using two simple functions namely, map and reduce. More recently, programming models like Spark [70] that support a much wider class of applications than MapReduce have been created.

3.7 Summary

This chapter described the Star class of computer programs, and a programming model also called Star that generates and executes parallel code for programs in the Star class. We illustrated the inadequacy of existing models like MapReduce in parallelizing programs in the Star class. We also presented parallel algorithms, which offer improvements over prior art, for two programs in the Star class namely, a Trip algorithm for solving symmetric, diagonally dominant tridiagonal systems, and a Wasp algorithm for watershed cuts on graphs.

Several directions for future work arise from the Star model. Given a program in the Star class, is it possible to automatically identify or derive the three subcomputations, which can then be used to parallelize the program? If it is indeed possible, such a derivation will be of huge value to parallel programmers. In the Star model, communication can be congested since all the communication happens between the workers and the master. A program in the Star class can also be expressed in a divide-and-conquer fashion, by dividing the second subcomputation recursively into three subcomputations. For example, consider the parallel-prefix program. A recursive model would ease the congestion in communication by distributing communication among the partitions. Hence, it will be useful to augment the Star model to generate and execute code for recursive Star programs.

Chapter 4

Conclusion

> Now this is not the end. It is not even the beginning of the end. But it is, perhaps, the end of the beginning.
>
> Winston Churchill

Performance, Portability, and Productivity represent the "ideal" expectations in computer programming. Programs should run fast, retain their functionality and speed across computers, operating systems, and compilers, and should be easy to write, debug, and maintain. In practice, however, the focus is largely on performance, and the portability and productivity expectations do not get enough attention, which they deserve. For example, the phrase "high-performance computing" indicates the importance of performance in computing. This thesis addressed the portability and productivity expectations in high-performance computing for certain classes of programs. Ztune achieves performance portability for serial divide-and-conquer programs, and Star improves programmer productivity in writing parallel code for the Star class of programs.

We sketch areas of future work in the following.

Ztuning serial divide-and-conquer codes: There are many serial divide-and-conquer applications like convolution, dynamic programming, and high-performance linear algebra that can benefit from Ztune's autotuning strategy. More generally, Ztune can be developed into an automated framework to autotune any divide-and-conquer program. Ztune can provide an application-

programming interface, where users can specify the possible division choices for a given problem, the subproblems that can be created from a given division choice, and the base case function to execute. See Figure 2-4 for reference. Ztune will then invoke these application-specific functions to enumerate possible plans, apply the pruning properties, and generate an optimal plan for a given divide-and-conquer program operating on a given input.

Extrapolate autotuning: In the present framework, Ztune autotunes each problem from scratch. Though it produces faster autotuning times using the pruning properties, the autotuning times can be reduced further by extrapolation. For example, can autotuning results from tuning a problem A be used to tune a related problem B? Extrapolation can be difficult if A and B produce subproblems that are highly dissimilar in shape and size. For example, consider the zoids created in tuning stencil computations. Nevertheless, we think extrapolation can benefit applications like matrix multiplication and matrix-vector product, where the subproblems have a "regular" shape and size.

Autotuning parallel divide-and-conquer codes: Given the importance of parallelism in high-performance computing, autotuning parallel divide-and-conquer programs will be highly useful in acheiving performance portability for such programs. This thesis lays the necessary foundation in understanding the complexity of autotuning just single-core programs. The framework of Ztune and its pruning properties can now be extended to autotune parallel divide-and-conquer programs. Parallel codes introduce a host of issues like memory bandwidth saturation, communication overhead, nondeterminism in the scheduler, and work-span optimization.

Merge portability and productivity: We haven't connected the portability and productivity expectations into a single framework in this thesis. Whereas Ztune addresses performance portability, Star addresses programmer productivity. Merging these expectations into a single framework, which not only makes parallel programming easier for the classes of divide-and-conquer and Star programs, but also autotunes their performance for a given machine configuration, will be a valuable tool.

Bibliography

[1] Berkeley ParLab. A Pattern Language for Parallel Programming, Version 2.0. EECS Department, University of California, Berkeley.

[2] ANSEL, J., AND CHAN, C. PetaBricks: Building adaptable and more efficient programs for the multi-core era. *XRDS 17*, 1 (2010).

[3] ANSEL, J., KAMIL, S., VEERAMACHANENI, K., O'REILLY, U.-M., AND AMARASINGHE, S. OpenTuner: An extensible framework for program autotuning. Tech. Rep. TR-2013-026, MIT CSAIL, 2013.

[4] APACHE SOFTWARE FOUNDATION. Hadoop.

[5] ASANOVIC, K., BODIK, R., CATANZARO, B. C., GEBIS, J. J., HUSBANDS, P., KEUTZER, K., PATTERSON, D. A., PLISHKER, W. L., SHALF, J., WILLIAMS, S. W., AND YELICK, K. A. The landscape of parallel computing research: A view from berkeley. Technical report UCB/EECS-2006-183, University of California at Berkeley, Dec. 2006.

[6] BEZANSON, J., EDELMAN, A., KARPINSKI, S., AND SHAH, V. B. Julia: A fresh approach to numerical computing. *CoRR abs/1411.1607* (2014).

[7] BIENIEK, A., BURKHARDT, H., MARSCHNER, H., NÖLLE, M., AND SCHREIBER, G. A parallel watershed algorithm. In *Proceedings of the Proc. 10th Scandinavian Conference on Image Analysis, SCIA '97* (June 1997).

[8] BLECK, R., ROOTH, C., HU, D., AND SMITH, L. T. Salinity-driven thermocline transients in a wind- and thermohaline-forced isopycnic coordinate model of the North Atlantic. *J. of Phys. Oceanography 22*, 12 (1992), 1486–1505.

[9] BLELLOCH, G. E. Prefix sums and their applications. Tech. Rep. CMU-CS-90-190, School of Computer Science, Carnegie Mellon University, Nov. 1990.

[10] BLELLOCH, G. E., LEISERSON, C. E., MAGGS, B. M., PLAXTON, C. G., SMITH, S. J., AND ZAGHA, M. A comparison of sorting algorithms for the connection machine cm-2. In *Proceedings of the Third Annual ACM Symposium on Parallel Algorithms and Architectures* (New York, NY, USA, 1991), SPAA '91, ACM, pp. 3–16.

[11] BONDHUGULA, U., BANDISHTI, V., COHEN, A., POTRON, G., AND VASILACHE, N. Tiling and optimizing time-iterated computations on periodic domains. In *PACT* (2014), ACM, pp. 39–50.

[12] BONDHUGULA, U., BASKARAN, M., KRISHNAMOORTHY, S., RAMANUJAM, J., ROUNTEV, A., AND P.SADAYAPPAN. Automatic transformations for communication-minimized parallelization and locality optimization in the polyhedral model. In *International Conference on Compiler Construction (ETAPS CC)* (2008), ACM, pp. 132–146.

[13] BONDHUGULA, U., HARTONO, A., RAMANUJAM, J., AND SADAYAPPAN, P. A practical automatic polyhedral parallelizer and locality optimizer. In *PLDI* (2008), ACM, pp. 101–113.

[14] CHANG, L.-W., STRATTON, J. A., KIM, H.-S., AND HWU, W.-M. W. A scalable, numerically stable, high-performance tridiagonal solver using gpus. In *SC '12* (Salt Lake City, Utah, USA, 2012).

[15] CHRISTEN, M., SCHENK, O., AND BURKHART, H. Patus: A code generation and autotuning framework for parallel iterative stencil computations on modern microarchitectures. In *IPDPS* (2011), IEEE, pp. 676–687.

[16] COUSTY, J., BERTRAND, G., NAJMAN, L., AND COUPRIE, M. Watershed cuts: Minimum spanning forests and the drop of water principle. *IEEE Trans. on Pattern Analysis and Machine Intelligence 31*, 8 (2009), 1362–1374.

[17] ŢĂPUŞ, C., CHUNG, I.-H., AND HOLLINGSWORTH, J. K. Active Harmony: Towards automated performance tuning. In *SC* (2002), ACM/IEEE, pp. 1–11.

[18] DATTA, K., KAMIL, S., WILLIAMS, S., OLIKER, L., SHALF, J., AND YELICK, K. Optimization and performance modeling of stencil computations on modern microprocessors. *SIAM Rev. 51*, 1 (2009), 129–159.

[19] DATTA, K., MURPHY, M., VOLKOV, V., WILLIAMS, S., CARTER, J., OLIKER, L., PATTERSON, D., SHALF, J., AND YELICK, K. Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. In *SC* (2008), ACM/IEEE, pp. 4:1–4:12.

[20] DEAN, J., AND GHEMAWAT, S. Mapreduce: Simplified data processing on large clusters. In *OSDI* (2004), ACM, pp. 137–149.

[21] DEMMEL, J. Applications of parallel computers: Lecture notes, 1996.

[22] DURSUN, H., NOMURA, K.-I., PENG, L., SEYMOUR, R., WANG, W., KALIA, R. K., NAKANO, A., AND VASHISHTA, P. A multilevel parallelization framework for high-order stencil computations. In *Euro-Par* (2009), pp. 642–653.

[23] DURSUN, H., NOMURA, K.-I., WANG, W., KUNASETH, M., PENG, L., SEYMOUR, R., KALIA, R. K., NAKANO, A., AND VASHISHTA, P. In-core optimization of high-order stencil computations. In *PDPTA* (2009), pp. 533–538.

[24] EDELMAN, A., AND PERSSON, P.-O. Fast multipole: It's all about adding functions in finite precision. 2004.

[25] EPPERSON, J. F. *An Introduction to Numerical Methods and Analysis*. Wiley-Interscience, 2007.

[26] FRAZER, W. D., AND MCKELLAR, A. C. Samplesort: A sampling approach to minimal storage tree sorting. *J. ACM 17*, 3 (July 1970), 496–507.

[27] FRIGO, M. A fast Fourier transform compiler. *ACM SIGPLAN Notices 34*, 5 (May 1999), 169–180.

[28] FRIGO, M., AND JOHNSON, S. The design and implementation of FFTW3. *Proceedings of the IEEE 93*, 2 (2005), 216–231.

[29] FRIGO, M., LEISERSON, C. E., PROKOP, H., AND RAMACHANDRAN, S. Cache-oblivious algorithms. In *FOCS* (1999), IEEE, pp. 285–297.

[30] FRIGO, M., AND STRUMPEN, V. Cache oblivious stencil computations. In *ICS* (2005), ACM, pp. 361–366.

[31] FRIGO, M., AND STRUMPEN, V. The cache complexity of multithreaded cache oblivious algorithms. *Theory of Computing Systems 45*, 2 (2009), 203–233.

[32] GAMMA, E., HELM, R., JOHNSON, R., AND VLISSIDES, J. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[33] GANDER, W., AND GOLUB, G. H. Cyclic reduction - history and applications. In *Proceedings of the Workshop on Scientific Computing* (Hong Kong, 1997).

[34] GARDNER, M. Mathematical Games. *Scientific American 223*, 4 (1970), 120–123.

[35] GUSTAVSON, F. G., AND GUPTA, A. A new parallel algorithm for tridiagonal symmetric positive definite systems of equations. In *PARA '96* (Lyngby, Denmark, 1996).

[36] JOHN, C. *Options, Futures, and Other Derivatives*. Prentice Hall, 2006.

[37] KAMIL, S., CHAN, C., OLIKER, L., SHALF, J., AND WILLIAMS, S. An auto-tuning framework for parallel multicore stencil computations. In *IPDPS* (2010), IEEE, pp. 1–12.

[38] KAMIL, S., DATTA, K., WILLIAMS, S., OLIKER, L., SHALF, J., AND YELICK, K. Implicit and explicit optimizations for stencil computations. In *MSPC* (2006), ACM, pp. 51–60.

[39] KAMIL, S., HUSBANDS, P., OLIKER, L., SHALF, J., AND YELICK, K. Impact of modern memory subsystems on cache optimizations for stencil computations. In *MSP* (2005), ACM, pp. 36–43.

[40] KAMIL, S. A. *Productive High Performance Parallel Programming with Auto-tuned Domain-Specific Embedded Languages*. PhD thesis, University of California, Berkeley, 2012.

[41] KRISHNAMOORTHY, S., BASKARAN, M., BONDHUGULA, U., RAMANUJAM, J., ROUNTEV, A., AND SADAYAPPAN, P. Effective automatic parallelization of stencil computations. In *PLDI* (2007), ACM.

[42] KYROLA, A., BLELLOCH, G., AND GUESTRIN, C. Graphchi: large-scale graph computation on just a pc. In *OSDI* (2012), ACM, pp. 31–46.

[43] LEISERSON, C. E. The Cilk++ concurrency platform. *Journal of Supercomputing 51*, 3 (March 2010), 244–257.

[44] MALAS, T., HAGER, G., LTAIEF, H., STENGEL, H., WELLEIN, G., AND KEYES, D. Multicore-optimized wavefront diamond blocking for optimizing stencil updates. *SIAM Journal on Scientific Computing 37*, 4 (2015), 439–464.

[45] MALAS, T. M., HORNICH, J., HAGER, G., LTAIEF, H., PFLAUM, C., AND KEYES, D. E. Optimization of an electromagnetics code with multicore wavefront diamond blocking and multi-dimensional intra-tile parallelization. *arXiv:1510.05218* (2015).

[46] MATTSON, T. G., SANDERS, B. A., AND MASSINGILL, B. L. *Patterns for Parallel Programming*, first ed. Addison-Wesley, 2004.

[47] MCCOOL, M., REINDERS, J., AND ROBISON, A. *Structured Parallel Programming: Patterns for Efficient Computation*, first ed. Morgan Kaufmann, 2012.

[48] MEI, R., SHYY, W., YU, D., AND LUO, L. Lattice Boltzmann method for 3-D flows with curved boundary. *J. of Comput. Phys 161*, 2 (2000), 680–699.

[49] MICIKEVICIUS, P. 3D finite difference computation on GPUs using CUDA. In *GPGPU* (2009), ACM, pp. 79–84.

[50] MOURA, J. M. F., SINGER, B., XIONG, J., JOHNSON, J., PADUA, D., VELOSO, M., AND JOHNSON, R. W. SPIRAL: A generator for platform-adapted libraries of signal processing algorithms. *Int. J. of High Perf. Comp. Appl. 18*, 1 (2004), 21–45.

[51] NAKANO, A., KALIA, R. K., AND VASHISHTA, P. Multiresolution molecular dynamics algorithm for realistic materials modeling on parallel computers. *Comp. Phys. Comm. 83*, 2-3 (1994), 197–214.

[52] NITSURE, A. Implementation and optimization of a cache oblivious lattice Boltzmann algorithm. Master's thesis, Institut für Informatic, Friedrich-Alexander-Universität Erlangen-Nürnberg, 2006.

[53] PACHECO, P. S. *Parallel Programming with MPI*. Morgan Kaufmann, 1997.

[54] PALAMADAI NATARAJAN, E., AND EDELMAN, A. Star: Simplified processing of ordered data. 2016.

[55] PALAMADAI NATARAJAN, E., MEHRI DEHNAVI, M., AND LEISERSON, C. E. Autotuning divide-and-conquer stencil computations. 2015.

[56] PANTAWONGDECHA, P. Autotuning divide-and-conquer matrix-vector multiplication. M.eng., Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, June 2016.

[57] PENG, L., SEYMOUR, R., NOMURA, K.-I., KALIA, R. K., NAKANO, A., VASHISHTA, P., LODDOCH, A., NETZBAND, M., VOLZ, W. R., AND WONG, C. C. High-order stencil computations on multicore clusters. In *IPDPS* (2009), IEEE, pp. 1–11.

[58] RIVERA, G., AND TSENG, C. Tiling optimizations for 3D scientific computations. In *SC* (2000), ACM/IEEE, pp. 32:1–32:23.

[59] SONG, Y., AND LI, Z. New tiling techniques to improve cache temporal locality. In *PLDI* (1999), ACM, pp. 215–228.

[60] STONE, H. S. An efficient parallel algorithm for the solution of a tridiagonal linear system of equations. *J. ACM 20*, 1 (Jan. 1973), 27–38.

[61] TAFLOVE, A., AND HAGNESS, S. C. *Computational Electrodynamics: The Finite-Difference Time-Domain Method*. Artech, 2000.

[62] TANG, Y., CHOWDHURY, R. A., KUSZMAUL, B. C., LUK, C.-K., AND LEISERSON, C. E. The Pochoir stencil compiler. In *SPAA* (2011), ACM, pp. 117–128.

[63] TARJAN, R. E. Efficiency of a good but not linear set union algorithm. *Journal of the ACM 22*, 2 (April 1975), 215–225.

[64] TARJAN, R. E. *Data Structures and Network Algorithms*. Society for Industrial and Applied Mathematics, 1983.

[65] THOMAS, L. Elliptic problems in linear difference equations over a network. Tech. rep., Watson Sci. Comput. Lab. Rept., Columbia University, New York, 1949.

[66] VUDUC, R., DEMMEL, J. W., AND YELICK, K. A. OSKI: A library of automatically tuned sparse matrix kernels. In *J. of Phys.* (2005), vol. 16, p. 521.

[67] WHALEY, R. C., AND DONGARRA, J. Automatically tuned linear algebra software. In *SC* (1998), ACM, pp. 1–27.

[68] WILLIAMS, S., CARTER, J., OLIKER, L., SHALF, J., AND YELLICK, K. Optimization of a lattice Boltzmann computation on state-of-the-art multicore platforms. *JPDC 69*, 9 (2009), 762–777.

[69] YANG, H.-C., DASDAN, A., HSIAO, R.-L., AND PARKER, D. S. Map-reduce-merge: Simplified relational data processing on large clusters. In *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 2007), SIGMOD '07, ACM, pp. 1029–1040.

[70] ZAHARIA, M., CHOWDHURY, M., DAS, T., DAVE, A., MA, J., MCCAULEY, M., FRANKLIN, M. J., SHENKER, S., AND STOICA, I. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation* (Berkeley, CA, USA, 2012), NSDI'12, USENIX Association, pp. 2–2.

[71] ZLATESKI, A. A design and implementation of an efficient, parallel watershed algorithm for affinity graphs. Master's thesis, Massachusetts Institute of Technology Electrical Engineering and Computer Science, 2011.