High-performance Computing with PetaBricks and Julia

by

Yee Lok Wong

B.S., Duke University (2006)

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
SEP 0 2 2011
LIBRARIES

Submitted to the Department of Mathematics
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

ARCHIVES

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2011

© Yee Lok Wong, 2011. All rights reserved.

The author hereby grants to MIT permission to reproduce and to distribute publicly paper and electronic copies of this thesis document in whole or in part in any medium now known or hereafter created.

Author .....
Department of Mathematics
April 28, 2011

Certified by .....
Alan Edelman
Professor of Applied Mathematics
Thesis Supervisor

Accepted by .....
Michel Goemans
Chairman, Applied Mathematics Committee

High-performance Computing with PetaBricks and Julia

by

Yee Lok Wong

Submitted to the Department of Mathematics
on April 28, 2011, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Abstract

We present two recent parallel programming languages, PetaBricks and Julia, and demonstrate how we can use these two languages to re-examine classic numerical algorithms in new approaches for high-performance computing.

PetaBricks is an implicitly parallel language that allows programmers to naturally express algorithmic choice explicitly at the language level. The PetaBricks compiler and autotuner is not only able to compose a complex program using fine-grained algorithmic choices but also find the right choice for many other parameters including data distribution, parallelization and blocking. We re-examine classic numerical algorithms with PetaBricks, and show that the PetaBricks autotuner produces nontrivial optimal algorithms that are difficult to reproduce otherwise. We also introduce the notion of variable accuracy algorithms, in which accuracy measures and requirements are supplied by the programmer and incorporated by the PetaBricks compiler and autotuner in the search of optimal algorithms. We demonstrate the accuracy/performance trade-offs by benchmark problems, and show how nontrivial algorithmic choice can change with different user accuracy requirements.

Julia is a new high-level programming language that aims at achieving performance comparable to traditional compiled languages, while remaining easy to program and offering flexible parallelism without extensive effort. We describe a problem in large-scale terrain data analysis which motivates the use of Julia. We perform classical filtering techniques to study the terrain profiles and propose a measure based on Singular Value Decomposition (SVD) to quantify terrain surface roughness. We then give a brief tutorial of Julia and present results of our serial blocked SVD algorithm implementation in Julia. We also describe the parallel implementation of our SVD algorithm and discuss how flexible parallelism can be further explored using Julia.

Thesis Supervisor: Alan Edelman
Title: Professor of Applied Mathematics

Acknowledgements

Throughout my graduate studies, I have been very fortunate to have met many wonderful people. Without their kind support and help, this thesis would not have been possible.

First and foremost, I would like to thank my advisor Alan Edelman for introducing me to the world of high-performance computing. His enthusiasm, energy and creativity in approaching problems was a constant source of motivation. I am very grateful also for his academic and moral support throughout my time in MIT, and particularly during some of my most difficult time here.

I would also like to express my gratitude to Saman Amarasinghe, who has provided me with great advice for the PetaBricks project. Thanks goes to the other thesis committee member Gilbert Strang, who offered helpful suggestions for this thesis work.

Special thanks goes to the PetaBricks team, especially Jason Ansel and Cy Chan for the collaboration and inspiring conversations about the PetaBricks project. The Julia team has also been extremely helpful for my work. I owe a special thanks to Jeff Bezanson for implementing a lot of our ideas in Julia in addition to his already long list of tasks. I would also like to thank Martin Bazant, Chris Rycroft and Ken Kamrin for their help and guidance for my early work in MIT.

On the non-academic side, I am extremely grateful to my friends in and outside of MIT. Siu-Chung Yau has provided strong support by his humiliating remarks over our countless phone conversations and by his constant reminder of what my "greatest achievement" is. He also provided motivation by showing by example how bad things could have gone if I had pushed my work and thesis writing to the very last minute. Chloe Kung has been a great friend, with whom I could always have a good laugh and not worry about the stress of work back at school. She also made sure I got real food by going around eating with me every time I went home for Christmas. Annalisa Pawlosky is the most supportive person in MIT, and she believed in my ability probably more than I do myself. She was always there when I needed a friend

to talk to, and I will never forget the many stories that she has shared with me.

Throughout my five years in MIT, the Warehouse graduate dorm has been my home. I would like to extend my heartfelt gratitude to my previous and current housemasters, Lori and Steve Lerman, Anne Carney and John Ochsendorf, for making the Warehouse such a comfortable and warm home for my graduate school life. I must also thank the members of the Warehouse dorm government, especially Allen Hsu, for being great friends throughout these years. I would also like to thank my friends in my second home a.k.a. the Math Department, Linan Chen, Lu Wang, Carol Xia Hua, Ramis Movassagh and Jiawei Chiu, for the inspiring conversations and great times we had in our office.

I must also thank friends of mine outside of MIT who have continued to be a part of my life: Wai Yip Kong, Leo Lam, Samuel Ng, Siyin Tan, Tina Chang, Edward Chan, Gina Gu, Henry Lam, Jacky Chang, LeVan Nguyen, Nozomi Yamaki and Hiroaki Fukada.

Finally and most importantly, I thank my brother and parents for their continuous support and encouragement since I came to the States for my undergraduate studies nine years ago. The phone conversations with my parents have been a refreshing source of comfort, because talking to my parents was one of the very limited times when I felt smart since I came to MIT. Since the end of my second year here in MIT, my mom has been asking me what I actually do in school without classes and exams, and my answers never seemed to really convince her that I was indeed doing something. Fortunately, I can now give her a better answer: I wrote this thing called a PhD thesis.

Contents

List of Figures

THIS PAGE INTENTIONALLY LEFT BLANK

List of Tables

THIS PAGE INTENTIONALLY LEFT BLANK

Chapter 1

Introduction

1.1 Background and Motivation

High-performance computing has become ubiquitous in scientific research and become more and more common even in everyday lives. Many laptops now have multicore processors and smartphones with dual-core processors have entered the market. Performance growth for single processors has approached its limit, but Moore's law still predicts that the number of transistors on a chip will double about every two years, and the physical limits would not be reached until the earliest in the year 2020 according to Intel [1, 71]. With the doubling of chip density every 2 years and clock speed not catching up with this growth rate, exposing and managing parallelism between multiple processors by software is the key to getting further performance gain.

One major aspect of parallel computing is to identify enough paralleism. The well-known Amdahl's law [3] states that if f is the fraction of work done sequentially, and $(1-f)$ is the fraction parallelizable with n processors with no scheduling overhead, the speedup is given by

$$\begin{aligned} \text{Speedup}(f,n) &= \frac{1}{f + \frac{1-f}{n}} \\ &\leq \frac{1}{f} \end{aligned} \tag{1.1}$$

Even if the parallel part speeds up perfectly, parallel performance is still limited by the sequential component. In practice, parallelism involves overhead such as communicating data, synchronization and managing different threads. A good parallel implementation also needs to take care of locality and dependencies of data, load balancing, granularity, race conditions and synchronization issues.

Historically, the first standardized library for high-performance computing systems was the Message Passing Interface (MPI), which allows the programmer to have control over every detail. Each message passing call has to be individually written, addressed, sent and received. Some of the other early parallel programming standards include High Performance Fortran (HPF) and OpenMP. Following the evolution of hardware, there has been an emergence of different programming languages and libraries.

However, designing and implementing parallel programs with good performance is still nontrivial despite decades of research and advances in both software and hardware. One major challenge is that the choice of algorithms depends not only on the problem size, but becomes more complicated with factors such as different data distribution, underlying architecture, and number of processors available. In terms of design, it is often not obvious how to choose the optimal algorithms for a specific problem. Complexity analysis gives an idea of how well algorithms perform asymptotically, but it gets more complicated when parallelism is involved. Take the symmetric eigenproblem as an example, the common serial algorithms all have $O(n^3)$ complexity to find all eigenvalues and eigenvectors, so it is difficult to find the right composition and cutoffs for an optimal hybrid algorithm in parallel. Asymptotically optimal algorithms, called cache-oblivious algorithms [42], are designed to achieve optimality without depending on hardware parameters such as cache size and cache-line length. Cache-oblivious algorithms use the ideal-cache model, but in practice memory system behavior is much more complicated. In addition, even when an optimal algorithm is implemented for a specific configuration, completely different algorithms may provide the best performance for various architectures, available computing resources and input data. With the advent of multiple processors, architectures are changing at a fast

rate. Implementing algorithms that can perform well for generations of architecture has become increasingly important.

Another challenge is the flexibility of parallelism available with the current programming languages. Parallel programming introduces complications not present in serial programming, such as keeping track of which processor data is stored on, sending and receiving data while minimizing communication and distributing computation among processors. Currently existing languages and API with low-level parallel constructs such as MPI, OpenMP and OpenCL give users full control over the parallelziation process. However, it can be difficult to program since communication is performed by explicitly managing the send and receive operations in the code. Messages and operations on each thread must also be explicitly crafted. On the other end of the spectrum, there are high-level programming languages such as Matlab's parallel computing toolbox, pMatlab [58] and Star-P [21], which allow higher level parallel data structures and functions. Serial programs can be transformed to have parallel functionality with minor modifications, so parallelism is almost "automatic". The trade-off for drastically reducing the difficulty of programming parallel machines is the lack of control of parallelsim. A parallel computing environment, which offers the right level of abstraction and maintains a good balance between parallelization details and productivity of program development and debugging, is still lacking.

In this thesis, we present two recent parallel programming languages, **PetaBricks** [6] and **Julia** [12], and demonstrate how these two languages address some of the difficulties of high-performance computing. Using PetaBricks and Julia, we re-examine some classic numerical algorithms in new approaches for high-performance computing.

PetaBricks is a new implicitly parallel language and compiler where having multiple implementations of multiple algorithms to solve a problem is the natural way of programming. Algorithmic choice is made to be a first class construct of the language, such that it can be expressed explicitly at the language level. The PetaBricks compiler autotunes programs by making both fine-grained as well as algorithmic choices. Choices also include different automatic parallelization techniques, data distributions,

algorithmic parameters, transformations, blocking and accuracy requirement. A number of empirical autotuning frameworks have been developed for building efficient, portable libraries in specific domains. PHiPAC [13] is an autotuning system for dense matrix multiply, generating portable C code and search scripts to tune for specific systems. ATLAS [88, 89] utilizes empirical autotuning to produce a cache-contained matrix multiply, which is then used in larger matrix computations in BLAS and LAPACK. FFTW [40, 41] uses empirical autotuning to combine solvers for FFTs. Other autotuning systems include SPIRAL [76] for digital signal processing, SPARSITY [55] for sparse matrix computations, UHFFT [2] for FFT on multicore systems, OSKI [87] for sparse matrix kernels, and autotuning frameworks for optimizing sequential [62, 63] and parallel [73] sorting algorithms. To the best of our knowledge, PetaBricks is the first language that enables programmers to express algorithmic choice at the language level and provides autotuned optimized programs that are scalable and portable in general purpose applications.

Julia is a new high-level programming language that aims at filling the gap between traditional compiled langauges and dynamic languages by providing a programming tool that is easy to use while not sacrificing performance. The Julia project is an ongoing project based at MIT. One of the main goals of the project is to provide flexible parallelism without extensive effort. For example, can the number of processors used not be fixed by the user, but scale up or down depending on the work waiting? This will become increasingly important when a cloud API is ready for Julia and an external cloud provider is used. How can we minimize unnecessary idle time of threads? What is the right level of abstraction, such that the user will not lose too much control of the parallelism while still being able to program without the need to specify every piece of detail? The Julia project is working to create a cloud friendly environment such that the Julia language is fast and general, easy to use, open source and readily available.

1.2 Scope and Outline

The rest of this thesis proceeds as follows:

In chapter 2, we introduce the PetaBricks language and describe the implementation of the compiler and autotuning system. The PetaBricks compiler and autotuner is not only able to compose a complex program using fine-grained algorithmic choices but also find the right choice for many other parameters including data distribution, parallelization and blocking. We re-examine classic numerical algorithms with PetaBricks, and present experimental results to show that the PetaBricks autotuner produces nontrivial optimal algorithms. Specifically, we give a review of classic algorithms used to solve the symmetric eigenvalue problem and LU Factorization. We then show that the optimal PetaBricks algorithm composition for the symmetric eigenvalue problem is different from the one used in LAPACK. We also demonstrate the speedup of LU Factorization implemented using our autotuned nontrivial varaible blocking algorithm over conventional fixed recursive blocking strategies.

We continue with PetaBricks in Chapter 3 by introducing the notion of variable accuracy. We present the programming model with PetaBricks where trade-offs between time and accuracy are exposed at the language level to the compiler. We describe the language extensions in PetaBricks to support variable accuracy, and outline how the PetaBricks compiler automatically searches the space of algorithms and parameters (optimal frontier) to construct an optimized algorithm for each accuracy level required. We demonstrate the accuracy/performance trade-offs by two examples, k-means clustering and preconditioned conjugate gradient. With our experimental results, we show how nontrivial algorithmic choice can change with different accuracy measure and requirements. In particular, we show how k-means clustering can be solved without specifying the number of clusters k, and show that the optimal k can be determined accurately with PetaBricks using relevant training data. We also show that the optimal choice of preconditioners can change with problem sizes, in addition to the system matrix.

In Chapter 4, we discuss a problem of analyzing a large set of raw terrain data,

which motivates the use of the Julia language. We focus on the downsampled dataset in chapter 4 and perform serial computations because the dataset is too large to fit on the memory of a regular machine. We perform various analysis to study the terrain profiles and show how classical filtering techniques and Singular Value Decomposition (SVD) can be applied to study road bumps and noise in various scales. We propose a systematic way to classify surface roughness and also suggest a useful measure μ based on the SVD to quantify terrain surface roughness. The methodology described does not require extensive knowledge and modeling of the terrian and vehicle movement. The algorithms suggested in Chapter 4 is generic and not domain-specfic, so they can be applied to give reproducible results on different sets of terrain data.

We introduce Julia in Chapter 5. We first give a brief tutorial of Julia and present some elementary results of Julia. An example Julia code, with syntax similar to Matlab, is presented. We also describe language features supported by Julia that are convenient and may not be available in common high-level programming packages. We then discuss the implementation of a serial blocked SVD algorithm. We run the SVD-based algorithm for terrain analysis presented in Chapter 4 but implemented with Julia, and show that the values of the roughness measure μ obtained agree with our prediction. We also describe the parallel implementation of our SVD algorithm and discuss how potentially further and more flexible paralleism can be explored in Julia.

We end with concluding remarks and future work in Chapter 6.

1.3 Contributions

The specific contributions of this dissertation are as follows:

- **Automatic optimal hybrid algorithm composition and cutoff points in symmetric eigenproblem:** We show that algorithmic choices can be expressed at the language level using PetaBricks and combined into an optimal hybrid algorithm by the PetaBricks compiler for significant speedup. In particular, we show that the optimal algorithm for the symmetric eigenvalue problem is

different from the one used in standard scientific package LAPACK. Compared with hard-coded composition of algorithms in standard numerical linear algebra packages, our approach produces autotuned hybrid algorithmic composition and automatic selection of cutoff points for algorithms even when the program is re-run on a different machine. Our implementation gives more portable performance than standard algorithms with static cutoff parameters. (Chapter 2)

- **Nontrivial variable blocking strategies for LU Factorization:** Our program written with PetaBricks explores performance of LU Factorization using non-fixed block sizes which are set by autotuning, as opposed to common fixed blocking with equal sizes. Recursive calls on different sizes of subproblems are made in our PetaBricks implementation to produce a variable blocking scheme. Our optimal algorithm uses uneven block sizes in the factorization of the same matrix, while a composition of varying block sizes is difficult to test with standard algorithms. (Chapter 2)

- **K-clustering with dynamic clusters:** We demonstrate how k-means clustering can be solved without specifying the number of clusters k, and show that the optimal k can be determined accurately with PetaBricks. To the best of our knowledge, our approach is the first to solve any general-purpose k-means clustering problem without domain-specific modeling and any solution of optimization problems by the user to specify the number of clusters k. Our approach with PetaBricks also makes cluster identification and assignments flexible for any applications, since it is easy for the user to modify the accuracy metric and input training data used. (Chapter 3)

- **Optimal Preconditioning without prior information of system matrix:** We incorporate accuracy requirement into autotuning the problem of preconditioning, and show that the optimal choice of preconditioners can change with problem sizes and input data. Autotuning with PetaBricks gives a systematic way to pick between different kinds of preconditioners, even when the user is

not certain about which preconditioner gives the best convergence. Our implementation allows automatic choices of optimal preconditioners without prior knowledge of the specific system matrix, as compared with standard practice of analyzing the properties and behavior of the system of equations to devise a specific preconditioner. (Chapter 3)

- **Singular Value Decomposition as a non-domain specific tool in terrain analysis:** We show how classic numerical kernels, namely Gaussian filtering and SVD, can be applied as non-domain specific tools in terrain analysis to capture noise and bumps in data. Our results are reproducible since we do not make any assumptions on the underlying terrain properties. (Chapter 4)

- **Creation of an SVD-based roughness measure on terrain data:** We propose a systematic method and measure using the SVD to quantify roughness level in large-scaled data without domain-specific modeling. Using terrain data as an example, we show how our measure successfully distinguishes between two road tracks with different levels of surface roughness. (Chapter 4)

- **Exploring new parallelism with asynchronous work scheduling in blocked SVD with Julia:** We introduce a new high-level programming language Julia and discuss how implementation of parallel SVD algorithms in Julia can give flexible parallelism in large-scale data processing. Our proposed implementation of parallel SVD suggests the following improvement in parallelism: starting the bidiagonalization of an diagonal block concurrently with the matrix multiplication updates of the trailing blocks, and minimizing idle times by starting trailing submatrix updates earlier when a portion of the intermediate matrices are ready. We outline how flexible parallelism can be explored by asynchronous work scheduling in the blocked SVD algorithm. The idea can be extended to many classical blocked numerical linear algebra algorithms, which have not been explored in standard scientific packages. (Chapter 5)

Chapter 2

Algorithmic Choice by PetaBricks

2.1 Introduction

2.1.1 Background and Motivation

Obtaining the optimal algorithm for a specific problem has become more challenging than ever with the advances in high-performance computing. Traditional complexity analysis provides some rough idea for how fast each individual algorithm runs, but it gets more complicated when choices for data distributions, parallelism, transformations and blocking comes into consideration. If a composition of multiple algorithms is needed for the optimal hybrid algorithm, the best composition is often difficult to be found by human analysis. For example, it is often important to make algorithmic changes to the problems for high performance when moving between different types of architectures, but the best solution to these choices is often tightly coupled to the underlying architectures, problem sizes, data, and available system resources. In some cases, completely different algorithms may provide the best performance.

One solution to this problem is to leave some of these choices to the compiler. Current compiler and programming language techniques are able to change some of these parameters, but today there is no simple way for the programmer to express or the compiler to choose different algorithms to handle different parts of the data. Existing solutions normally can handle only coarse-grained, library level selections or

hand coded cutoffs between base cases and recursive cases.

While traditional compiler optimizations can be successful at optimizing a single algorithm, when an algorithmic change is required to boost performance, the burden is put on the programmer to incorporate the new algorithm. If a composition of multiple algorithms is needed for the best performance, the programmer must write both algorithms, the glue code to connect them together, and figure out the best switch over points. Today's compilers are unable to change the nature of this composition because it is constructed with traditional control logic such as loops and switches. The needs of modern computing require a language construct like an *either* statement, which would allow the programmer to give a menu of algorithmic choices to the compiler.

Hand-coded algorithmic compositions are commonplace. A typical example of such a composition can be found in the C++ Standard Template Library (STL)[1] routine `std::sort`, which uses merge sort until the list is smaller than 15 elements and then switches to insertion sort. Tests in [6] have shown that higher cutoffs (around 60-150) perform much better on current architectures. However, because the optimal cutoff is dependent on architecture, cost of the comparison routine, element size, and parallelism, no single hard-coded value will suffice.

This problem has been addressed for certain specific algorithms by autotuning software, such as ATLAS [88] and FFTW [40, 41], which have training phases where optimal algorithms and cutoffs are automatically selected. Unfortunately, systems like this only work on the few algorithms provided by the library designer. In these systems, algorithmic choice is made by the application without the help of the compiler.

In this chapter, we describe a recent language PetaBricks with new language constructs that allow the programmer to specify a menu of algorithmic choices and new compiler techniques to exploit these choices to generate high performance yet portable code.

[1] From the version of the libstdc++ included with GCC 4.3.

2.1.2 PetaBricks for Auotuning Algorithmic Choice

PetaBricks [6], a new implicitly parallel language and compiler, was designed such that multiple implementations of multiple algorithms to solve a problem can be provided by the programmer at the language level. Algorithmic choice is made to be a first class construct of the language. Choices are provided in a way such that information is used by the PetaBricks compiler and runtime to create and autotune an optimized hybrid algorithm. The PetaBricks compiler autotunes programs by making both fine-grained as well as algorithmic choices. Other non-algorithmic choices include different automatic parallelization techniques, data distributions, transformations, and blocking.

We present the PetaBricks language and compiler in this chapter, with a focus on its application in picking optimal algorithms for some classical numerical computation kernels. Some of the materials in this chapter appear in [6]. This chapter discusses the algorithms and autotuned results in more detail, and adds an additioanl set of benchmark results for dense LU factorization. We show that algorithmic choices can be incorporated into an optimal hybrid algorithm by the PetaBricks compiler for significant speedup. In particular, we show that the optimal algorithm for the symmetric eigenvalue problem is different from the one used in standard scientic package LAPACK [5]. Compared with hard-coded composition of algorithms in standard numerical linear packages, our autotuning approach allows automatic selection of cutoff points for algorithms when the underlying architecture changes. We also demonstrate the effects of different blocking strategies in LU Factorization. Our approach with PetaBricks explores performance of LU using non-fixed block sizes which are set by autotuning, as opposed to common fixed blocking with equal sizes.

2.2 The PetaBricks Language and Compiler

For more information about the PetaBricks language and compiler see [6]; the following summary is included for background.

2.2.1 Language Design

The main goal of the PetaBricks language was to expose algorithmic choice to the compiler in order to allow choices to specify different granularities and corner cases. PetaBricks is an implicitly parallel language, where the compiler automatically parallelizes PetaBricks programs.

The language is built around two major constructs, *transforms* and *rules*. The *transform*, analogous to a function, defines an algorithm that can be called from other transforms, code written in other languages, or invoked from the command line. The header for a transform defines *to*, *from*, and *through* arguments, which represent inputs, outputs, and intermediate data used within the transform. The size in each dimension of these arguments is expressed symbolically in terms of free variables, the values of which must be determined by the PetaBricks runtime.

The user encodes choice by defining multiple *rules* in each transform. Each rule defines how to compute a region of data in order to make progress towards a final goal state. Rules have explicit dependencies parametrized by free variables set by the compiler. Rules can have different granularities and intermediate states. The compiler is required to find a sequence of rule applications that will compute all outputs of the program. The explicit rule dependencies allow automatic parallelization and automatic detection and handling of corner cases by the compiler. The rule header references *to* and *from* regions which are the inputs and outputs for the rule. The compiler may apply rules repeatedly, with different bindings to free variables, in order to compute larger data regions. Additionally, the header of a rule can specify a *where* cla]use to limit where a rule can be applied. The body of a rule consists of C++-like code to perform the actual work.

Figure 2-1 shows an example PetaBricks transform for matrix multiplication. The transform header is on lines 1 to 3. The inputs (line 2) are $m \times p$ matrix A and $p \times n$ matrix B. The output (line 3) is C, which is a $m \times n$ matrix. Note that in PetaBricks notation, the first index refers to column and the second index is the row index. The first rule (Rule 0 on line 6 to 9) is the straightforward way of computing

a single matrix element $C_{ij} = \sum_{k=1}^{p} A_{ik}B_{kj}$. With the first rule alone the transform would be correct, the remaining rules add choices. Rules 1, 2, and 3 (line 13 to 40) represent three ways of recursively decomposing matrix multiply into smaller matrix multiplies. Rule 1 (line 13 to 19) decomposes both A and B into half pieces: $C = (A_1|A_2) \cdot \left(\frac{B_1}{B_2}\right) = A_1B_1 + A_2B_2$. Rule 2 (line 23 to 30) decomposes B in two column blocks: $C = (C_1|C_2) = A \cdot (B_1|B_2) = (AB_1|AB_2)$. Rule 3 (line 33 to 40) decomposes A into two row blocks: $C = \left(\frac{C_1}{C_2}\right) = \left(\frac{A_1}{A_2}\right) \cdot B = \left(\frac{A_1B}{A_2B}\right)$. The compiler must pick when to apply these recursive decompositions and incorporate all the choices to form an optimal composition of algorithm.

In addition to choices between different algorithms, many algorithms have configurable parameters that change their behavior. A common example of this is the branching factor in recursively algorithms such as merge sort or radix sort. To support this PetaBricks has a `tunable` keyword that allows the user to export custom parameters to the autotuner. PetaBricks analyzes where these tunable values are used, and autotunes them at an appropriate time in the learning process.

PetaBricks contains additional language features such as rule priorities, `where` clauses for specifying corner cases in data dependencies, and `generator` keyword for specifing input training data. These features will not be discussed here in detail.

2.2.2 Compiler

The PetaBricks implementation consists of three components: a source-to-source compiler from the PetaBricks language to C++, an autotuning system and choice framework to find optimal choices and set parameters, and a runtime library used by the generated code.

The relationship between these components is depicted in Figure 2-2. First, the source-to-source compiler executes and performs static analysis. The compiler encodes choices and tunable parameters in the output code so that autotuning can be performed. When autotuning is performed (either at compile time or at installation time), it outputs an application configuration file that controls when different choices are made. This configuration file can be edited by hand to force specific choices. Op-

tionally, this configuration file can be fed back into the compiler and applied statically to eliminate unused choices and allow additional optimizations.

To help illustrate the compilation process we will use the example transform `CumulativeSum`, shown in Figure 2-3. `CumulativeSum` computes the cumulative (sometimes called rolling) sum of the input vector A, such that the output vector B satisfies $B(i) = A(0) + A(1) + ... + A(i)$. There are two rules in this transform, each specifying a different algorithmic choice. Rule 0 (line 6-8) simply sums up all elements of A up to index i and stores the result to $B[i]$. Rule 1 (line 11-14) uses previously computed values of B to get $B(i) = A(i) + B(i-1)$. An algorithm using only Rule 0 carries out more computations ($\Theta(n^2)$ operations), but can be executed in a data parallel way. An algorithm using only Rule 1 requires less arithmetic ($\Theta(n)$ operations), but has no parallelism and must be run sequentially.

The PetaBricks compiler works using the following main phases. In the first phase, the input language is **parsed** into an abstract syntax tree. Rule dependencies are normalized by converting all dependencies into region syntax, assigning each rule a symbolic *center*, and rewriting all dependencies to be relative to this center. (This is done using the Maxima symbolic algebra library [78].) In our `CumulativeSum` example, the center of both rules is equal to i, and the dependency normalization does not do anything other than replace variable names.

Next, **applicable regions** (regions where each rule can legally be applied, called an *applicable*) are calculated for each possible choice using an inference system. In rule 0 of our `CumulativeSum` example, both `b` and `in` (and thus the entire rule) have an applicable region of $[0, n)$. In rule 1, `a` and `b` have applicable regions of $[0, n)$ and `leftSum` has an applicable region of $[1, n)$ because it would read off the array for $i = 0$. These applicable regions are intersected to get an applicable region for rule 1 of $[1, n)$.

The applicable regions are then aggregated together into **choice grids**. The choice grid divides each matrix into rectilinear regions where uniform sets of rules can be

applied. In our `CumulativeSum` example, the choice grid for B is:

$$\begin{aligned} [0,1) &= \{\text{rule 0}\} \\ [1,n) &= \{\text{rule 0, rule 1}\} \end{aligned}$$

and A is not assigned a choice grid because it is an input. For analysis and scheduling these two regions are treated independently. Rule priorities are also applied in this pharse if users have specified priorities using keywords such as `primary` and `secondary`. Non-rectilinear regions can also be created using `where` clauses on rules.

Finally, a **choice dependency graph** is constructed using the simplified regions from the choice grid. The choice dependency graph consists of edges between symbolic regions in the choice grids. Each edge is annotated with the set of choices that require that edge, a direction of the data dependency, and an offset between rule centers for that dependency. Figure 2-4 shows the choice dependency graph for our example `CumulativeSum`. The three nodes correspond to the input matrix and the two regions in the choice grid. Each edge is annotated with the rules that require it along with the associated directions and offsets. These annotations allow matrices to be computed in parallel if parallelism is possible with the rules selected. The choice dependency graph is encoded in the output program for use by the autotuner and parallel runtime. It contains all information needed to explore choices and execute the program in parallel. These processes are explained in further detail in [6].

PetaBricks **code generation** has two modes. In the default mode choices and information for autotuning are embedded in the output code. This binary can be dynamically tuned, which generates a configuration file, and later run using this configuration file. In the second mode for code generation, a previously tuned configuration file is applied statically during code generation. The second mode is included since the C++ compiler can make the final code incrementally more efficient when the choices are eliminated.

2.2.3 Parallelism in Output Code

The PetaBricks runtime includes a parallel work stealing dynamic scheduler, which works on *tasks* with a known interface. The generated output code will recursively create these tasks and feed them to the dynamic scheduler to be executed. Dependency edges between tasks are detected at compile time and encoded in the tasks as they are created. A task may not be executed until all the tasks that it depends on have completed. These dependency edges expose all available parallelism to the dynamic scheduler and allow it to change its behavior based on autotuned parameters.

The generated code is constructed such that functions suspended due to a call to a spawned task can be migrated and executed on a different processor. This exposes parallelism and helps the dynamic scheduler schedule tasks in a depth-first search manner. To support the fucnction's stack frame and register migration, *continuation points*, at which a partially executed function may be converted back into a task so that it can be rescheduled to a different processor, are generated. The continuation points are inserted after any code that spawns a task. This is implemented by storing all needed state to the heap.

The code generated for dynamic scheduling incurs some overhead, despite being heavily optimized. In order to amortize this overhead, the output code that makes use of dynamic scheduling is not used at the leaves of the execution tree where most work is done. The PetaBricks compiler generates two versions of every output function. The first version is the dynamically scheduled task-based code described above, while the second version is entirely sequential and does not use the dynamic scheduler. Each output transform includes a tunable parameter (set during autotuning) to decide when to switch from the dynamically scheduled to the sequential version of the code.

2.2.4 Autotuning System and Choice Framework

Autotuning is performed on the target system so that optimal choices and cutoffs can be found for that architecture. The autotuning library is embedded in the output program whenever choices are not statically compiled in. Autotuning outputs an

application configuration file containing choices. This file can either be used to run the application, or it can be used by the compiler to build a binary with hard-coded choices.

The autotuner uses the *choice dependency graph* encoded in the compiled application. This choice dependency graph contains the choices for computing each region and also encodes the implications of different choices on dependencies. This choice dependency graph is also used by the parallel scheduler.

The intuition of the autotuning algorithm is that we take a bottom-up approach to tuning. To simplify autotuning, we assume that the optimal solution to smaller sub-problems is independent of the larger problem. In this way we build algorithms incrementally, starting on small inputs and working up to larger inputs.

The autotuner builds a multi-level algorithm. Each level consists of a range of input sizes and a corresponding algorithm and set of parameters. Rules that recursively invoke themselves result in algorithmic compositions. In the spirit of a genetic tuner, a population of candidate algorithms is maintained. This population is seeded with all single-algorithm implementations. The autotuner starts with a small training input and on each iteration doubles the size of the input. At each step, each algorithm in the population is tested. New algorithm candidates are generated by adding levels to the fastest members of the population. Finally, slower candidates in the population are dropped until the population is below a maximum size threshold. Since the best algorithms from the previous input size are used to generate candidates for the next input size, optimal algorithms are iteratively built from the bottom up.

In addition to tuning algorithm selection, PetaBricks uses an n-ary search tuning algorithm to optimize additional parameters such as parallel-sequential cutoff points for individual algorithms, iteration orders, block sizes (for data parallel rules), data layout, as well as user-specified tunable parameters.

All choices are represented in a flat configuration space. Dependencies between these configurable parameters are exported to the autotuner so that the autotuner can choose a sensible order to tune different parameters. The autotuner starts by tuning the leaves of the graph and works its way up. In the case of cycles, it tunes

all parameters in the cycle in parallel, with progressively larger input sizes. Finally, it repeats the entire training process, using the previous iteration as a starting point, a small number of times to better optimize the result.

2.2.5 Runtime Library

The runtime library is primarily responsible for managing parallelism, data, and configuration. It includes a runtime scheduler as well as code responsible for reading, writing, and managing inputs, outputs, and configurations. The runtime scheduler dynamically schedules tasks (that have their input dependencies satisfied) across processors to distribute work. The scheduler attempts to maximize locality using a greedy algorithm that schedules tasks in a depth-first search order. Work is distributed with thread-private double-ended queues (deques) and a task stealing protocol following the approach taken by Cilk [43]. A thread operates on the top of its deque as if it were a stack, pushing tasks as their inputs become ready and popping them when a thread needs more work. When a thread runs out of work, it randomly selects a victim and steals a task from the *bottom* of the victim's deque. This strategy allows a thread to steal another thread's most nested continuation, which preserves locality in the recursive algorithms we observed. Cilk's `THE` protocol is used to allow the victim to pop items of work from its deque without needing to acquire a lock in the common case.

2.3 Symmetric Eigenproblem

2.3.1 Background

The symmetric eigenproblem is the problem of computing the eigenvalues and/or eigenvectors of a symmetric $n \times n$ matrix. It often appears in mathematical and scientific applications such as mechanics, quantum physics, structural engineering and perturbation theory. For example, the Hessian matrix is a square matrix of the second-order partial derivatives of a function, which is always symmetric if the mixed

partials are continuous. Thus solving the symmetric eigenproblem is important in applications that involve multivariate calculus and differential equations.

Deciding on which algorithms to use depends on the number of eigenvalues required and whether eigenvectors are needed. To narrow the scope, here we study the problem in which all of the n eigenvalues and eigenvectors are computed. Specifically, our autotuning approach allows automatic selection of cutoff points for composition of algorithms when the underlying architecture changes, while in standard numerical linear packages, the composition of algorithms is hard-coded.

2.3.2 Basic Building Blocks

To find all the eigenvalues λ and eigenvectors x of a real $n \times n$ matrix, we make use of three primary algorithms, (i) QR iteration, (ii) Bisection and inverse iteration, and (iii) Divide-and-conquer.

The computation proceeds as follows:
(1) The input matrix A is first reduced to a tridiagonal form: $A = QTQ^T$, where Q is orthogonal and T is symmetric tridiagonal.
(2) All the eigenvalues and eigenvectors of the tridiagonal matrix T are then computed by one of the three primary algorithms, or any hybrid algorithm composed by the three primary ones.
(3) The eigenvalues of the original input A and those of the tridiagonal matrix T are equal. The eigenvectors of A are obtained by multiplying Q by the eigenvectors of T.

The total work needed is $O(n^3)$ for reduction of the input matrix and transforming the eigenvectors, plus the cost associated with each algorithm in step (2) which is also $O(n^3)$ [29]. Since steps (1) and (3) do not depend on the algorithm chosen to compute the eigenvalues and eigenvectors of T, we analyze only step (2) in this section.

We now give a review of the three primary algorithms as follows:

- **QR iteration** applies the QR decomposition iteratively until T converges to a diagonal matrix. The idea behind this algorithm is as follows: With a quantity

s called the shift, the QR factorization of the shifted matrix $A - sI = QR$ gives the orthogonal matrix Q and upper traingular R. Then multiplication in reverse order RQ gives

$$A_{new} = RQ + sI = Q^T(A - sI)Q + sI = Q^T AQ. \tag{2.1}$$

As each of this iteration is applied, the matrix A becomes more upper triangular. Since the input $A = T$ is tridiagonal, T will eventually converge to a diagonal matrix, and the entries are the eigenvalues. QR Iteration computes all eigenvalues in $O(n^2)$ flops but requires $O(n^3)$ operations to find all eigenvectors.

- **Bisection**, followed by **inverse iteration**, finds k eigenvalues and the corresponding eigenvectors in $O(nk^2)$ operations, resulting in a complexity of $O(n^3)$ for finding all eigenvalues and eigenvectors. Given a real symmetric $n \times n$ matrix A, the eigenvalues can be computed by finding the roots of the polynomial $p(x) = \det(A - xI)$. Let $A^{(1)}, \ldots, A^{(n)}$ denote the upper-left square submatrices. The eigenvalues of these matrices interlace, and the number of negative eigenvalues of A equals the number of sign changes in the Strum sequence [85]

$$1, \det(A^{(1)}), \ldots, \det(A^{(n)}) \tag{2.2}$$

Denote the k-th diagonal and superdiagonal elements of A by d_k and e_k. Expanding $\det(A^{(k)})$ by minors with respect to the k-th row (with entries d_k and e_{k-1}) gives

$$\det(A^{(k)}) = d_k \det(A^{(k-1)}) - e_{k-1}^2 \det(A^{(k-2)}) \tag{2.3}$$

With shift xI and expressing $p^{(k)}(x) = \det(A^{(k)} - xI)$, we get the recurrence

$$p^{(k)}(x) = (d_k - x)p^{(k-1)}(x) - e_{k-1}^2 p^{(k-2)}(x) \tag{2.4}$$

Applying this recurrence for a succession of values of x and counting sign changes, the bisection algorithm identifies eigenvalues in arbitrary intervals

$[a, b)$. Each eigenvalue and eigenvector thus can be computed independently, making the algorithm "embarrassingly parallel".

- The eigenproblem of tridiagonal T can also be solved by a **divide-and-conquer** approach. Observing that T is almost block diagonal, we can express T as the

sum of a block diagonal matrix plus a rank-1 correction:

$$
\begin{aligned}
T = \left[\begin{array}{c|c} \hat{T}_1 & \hat{T}_{12} \\ \hline \hat{T}_{21} & \hat{T}_2 \end{array}\right] &= \left[\begin{array}{cccc|cccc} a_1 & b_1 & & & & & & \\ b_1 & \ddots & \ddots & & & & & \\ & \ddots & a_{m-1} & b_{m-1} & & & & \\ & & b_{m-1} & a_m & b_m & & & \\ \hline & & & b_m & a_{m+1} & b_{m+1} & & \\ & & & & b_{m+1} & \ddots & \ddots & \\ & & & & & \ddots & \ddots & b_{n-1} \\ & & & & & & b_{n-1} & a_n \end{array}\right] \\
&= \left[\begin{array}{cccc|cccc} a_1 & b_1 & & & & & & \\ b_1 & \ddots & \ddots & & & & & \\ & \ddots & a_{m-1} & b_{m-1} & & & & \\ & & b_{m-1} & a_m - b_m & & & & \\ \hline & & & & a_{m+1} - b_m & b_{m+1} & & \\ & & & & b_{m+1} & \ddots & \ddots & \\ & & & & & \ddots & \ddots & b_{n-1} \\ & & & & & & b_{n-1} & a_n \end{array}\right] \\
&\quad + \left[\begin{array}{cccc|cccc} & & & & & & & \\ & & & & & & & \\ & & & b_m & b_m & & & \\ \hline & & & b_m & b_m & & & \\ & & & & & & & \\ & & & & & & & \\ & & & & & & & \end{array}\right] \\
&= \left[\begin{array}{c|c} T_1 & 0 \\ \hline 0 & T_2 \end{array}\right] + b_m u u^T, \qquad (2.5)
\end{aligned}
$$

where $u^T = [0, \ldots, 0, 1, 1, 0, \ldots, 0]$.

The only difference between T_1 and $\hat{T}_1$ is that the lower right entry in T_1 has been replaced with $a_m - b_m$ and similarly, in T_2 the top left entry has been replaced with $a_{m+1} - b_m$. The eigenvalues and eigenvectors of T_1 and T_2 can be computed recursively to get $T_1 = Q_1\Lambda_1 Q_1^T$ and $T_2 = Q_2\Lambda_2 Q_2^T$. Finally, the eigenvalues of T can be obtained from those of T_1 and T_2 as follows:

$$\begin{aligned} T &= \begin{bmatrix} T_1 & 0 \\ 0 & T_2 \end{bmatrix} + b_m uu^T \\ &= \begin{bmatrix} Q_1\Lambda_1 Q_1^T & 0 \\ 0 & Q_2\Lambda_2 Q_2^T \end{bmatrix} + b_m uu^T \\ &= \begin{bmatrix} Q_1 & 0 \\ 0 & Q_2 \end{bmatrix} \left(\begin{bmatrix} \Lambda_1 & 0 \\ 0 & \Lambda_2 \end{bmatrix} + b_m vv^T \right) \begin{bmatrix} Q_1^T & 0 \\ 0 & Q_2^T \end{bmatrix}, \end{aligned} \tag{2.6}$$

where

$$v = \begin{bmatrix} Q_1^T & 0 \\ 0 & Q_2^T \end{bmatrix} \cdot u = \begin{bmatrix} \text{last column of } Q_1^T \\ \text{first column of } Q_2^T \end{bmatrix} \tag{2.7}$$

Thus the eigenvalues of T is the same as the eigenvalues of the matrix $D + b_m vv^T$, where $D = \begin{bmatrix} \Lambda_1 & 0 \\ 0 & \Lambda_2 \end{bmatrix}$ is a diagonal matrix, and b_m and v are obtained as indicated above. The eigenvalues of $D + b_m vv^T$ can be computed by solving an equation called the secular equation. For details on solving the secular equation, refer to [61]. Divide-and-conquer algorithm requires $O(n^3)$ flops in the worst case.

2.3.3 Experimental Setup

The PetaBricks transforms for these three primary algorithms are implemented using LAPACK routines `dlaed1`, `dstebz`, `dstein` and `dsteqr`. Note that MATLAB's polyalgorithm `eig` also calls LAPACK routines. Our optimized hybrid PetaBricks algorithm computes the eigenvalues Λ and eigenvectors X by automating choices of these three basic algorithms. The pseudo code for this is shown in Figure 2-5.

There are three algorithmic choices, two non-recursive and one recursive. The two non-recursive choices are QR iterations, or bisection followed by inverse iteration. Alternatively, recursive calls can be made. At the recursive call, the PetaBricks compiler will decide the next choices, i.e. whether to continue making recursive calls or switch to one of the non-recursive algorithms. Thus the PetaBricks compiler chooses the optimal cutoff for the base case if the recursive choice is made.

The results were gathered on a 8-way (dual socket, quad core) Intel Xeon E7340 system running at 2.4 GHz. The system was running 64 bit CSAIL Debian 4.0 with Linux kernel 2.6.18 and GCC 4.1.2.

2.3.4 Results and Discussion

After autotuning, the best algorithm choice was found to be divide-and-conquer for $n \times n$ matrices with n larger than 48, and switching to QR iterations when the size of matrix $n \leq 48$.

We implemented and compared the performance of five algorithms in PetaBricks: QR iterations, bisection and inverse iteration, divide-and-conquer with base case $n = 1$, divide-and-conquer algorithm with hard-coded cutoff at $n = 25$, and our autotuned hybrid algorithm. In figure 2-6, these are labelled QR, Bisection, DC, Cutoff 25 and Autotuned respectively. The input matrices tested were symmetric tridiagonal with randomly generated values. Our autotuned algorithm runs faster than any of the three primary algorithms alone (QR, Bisection and DC). It is also faster than the divide-and-conquer strategy which switches to QR iteration for $n \leq 25$, which is the underlying algorithm of the LAPACK routine `dstevd` [5].

We see that the optimal algorithmic choice can be nontrivial even in a problem as common as the eigenproblem. For instance, bisection may seem very attractive to apply in parallel [32], but it is not included in our Petabricks autotuned results, which takes care of parallelism automatically. Although our optimal Petabricks hybrid algorithm differs from the widely used LAPACK eigenvalue routine only by the recursion cutoff size, the LAPACK routine has a hardcoded algorithmic choice and cutoff value of 25. In contrary, Petabricks allows autotuning to be rerun easily whenever the

underlying architecture and the available computing resources change.

Our autotuned algorithm is automatically parallel. Figure 2-7 shows the parallel scalability for eigenproblem. Speedup is calculated by $S_p = \frac{T_1}{T_p}$, where p is the number of threads and T_p is the execution time of the algorithm with p threads. The plot was generated for three input sizes $n = 256, 512, 1024$ using up to 8 worker threads. The parallel speedup is sublinear, but we obtained greater speedup as the problem size n increases.

2.4 Dense LU Factorization

LU Factorization is a matrix decomposition which writes a matrix A as a product of a lower traingular matrix L and an upper traingular matrix U. It is the simplest way to obtain the direct solutions of linear systems of equations. The most common formulation of LU factorization is known as Gaussian elimination, and is perhaps one of the most widely known numerical algorithms. LU Facotrization and its variants, QR and Cholesky decomposition, have been well studied in the literature. For simplicity, we focus on square $n \times n$ matrix A in this section, though analysis for rectangular matrices follow naturally.

2.4.1 Traditional Algorithm and Pivoting

Let $A \in \mathbb{R}^{n \times n}$. LU Factorization transforms A into an $n \times n$ upper triangular matrix U by zeroing elements below the diagonal, starting from the first column to the last. The elements below the diagonal are eliminated by subtracting multiples of each row from subsequent rows, which is equilavent to multiplying A by a sequence of lower triangular matrices L_k:

$$L_{n-1} \cdots L_2 L_1 A = U \tag{2.8}$$

Let $L^{-1} = L_{n-1} \cdots L_2 L_1$, or $L = L_1^{-1} L_2^{-1} \cdots L_{n-1}^{-1}$, we get $A = LU$, where L is lower triangular with all of its diagonal elements equal to 1, and U is an upper traingular matrix.

One simple implementation of LU Factorization without pivoting ($O(n^3)$ flops) is shown in Figure 2-8.

Unfortunately, this implementation is *not backward stable*[2]. Consider the following matrix as an example

$$A = \begin{bmatrix} 10^{-18} & 1 \\ 1 & 1 \end{bmatrix}$$

Computing the LU Factorization without pivoting (row exchanges) in double-precision arithmetic gives the follwing:

$$\tilde{L} = \begin{bmatrix} 1 & 0 \\ \text{fl}(1/10^{-18}) & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 10^{18} & 1 \end{bmatrix}$$

$$\tilde{U} = \begin{bmatrix} 10^{-18} & 1 \\ 0 & \text{fl}(1 - \text{fl}(10^{18} \cdot 1)) \end{bmatrix} = \begin{bmatrix} 10^{-18} & 1 \\ 0 & -10^{18} \end{bmatrix}$$

Note that

$$\tilde{L}\tilde{U} = \begin{bmatrix} 10^{-18} & 1 \\ 1 & 0 \end{bmatrix} \neq A$$

To improve numerical stability, **pivoting** is applied to Gaussian elimination such that $PA = LU$, where P is a permutation matrix. The most common practice is *partial pivoting*, which swaps rows to ensure that the entry in the pivot position (the upper left entry A_{jj} to be divided by each element $A_{kj}, k > j$ below it) has the greatest absolute value in that column on or below that row. Partial pivoting is believed to be stable in practice. Another strategy is called *complete pivoting*, which always swaps rows and columns to ensure that the entry in the pivot position has the greatest absolute value among all entries in the remaining submatrix. Complete pivoting is rarely used, because the improvement in numerical stability in practice is not signficiant compared to the extra cost searching for the largest element in the whole submatrix. Error bounds and conjectures on growth factors of complete pivoting have been studied in the liteature [23, 38, 46, 90]. A study of various common pivoting

[2] *backward stability* is one property of numerical stability. For more details, see [29] or [85].

strategies can be found in [19]. A recently proposed algorithm for LU Factorization uses a different pivoting strategy, which the authors call *incremental pivoting* [77].

2.4.2 Recursive and Block Algorithms

To achieve better performance, recursive algorithms of LU Factorization (and its variant QR factorization) have been formulated [4, 39, 51, 84]. The recursive algorithm with partial pivoting treats $A \in \mathbb{R}^{n \times n}$ as a block matrix

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix},$$

where each block A_{jk} is of order $n/2$-by-$n/2$. The algorithm works as follows:

1. Recursively factor the left part of A, such that

$$P \begin{bmatrix} A_{11} \\ A_{21} \end{bmatrix} = \begin{bmatrix} L_{11} \\ L_{21} \end{bmatrix} U_{11}$$

2. Permute the right part of A

$$\begin{bmatrix} \hat{A}_{12} \\ \hat{A}_{22} \end{bmatrix} := P_1 \begin{bmatrix} A_{12} \\ A_{22} \end{bmatrix}$$

3. Solve the triangular system to get $U_{12} = L_{11}^{-1} \hat{A}_{12}$.
4. Update lower-right block $\hat{A}_{22} := \hat{A}_{22} - L_{21} U_{12}$.
5. Recursively factor the remaining submatrix to get $P_2 \hat{A}_{22} = L_{22} U_{22}$, and permute lower-left block $\hat{L}_{21} := P_2 L_{21}$.
6. Return $P = P_2 P_1$, $L = \begin{bmatrix} L_{11} & 0 \\ \hat{L}_{21} & L_{22} \end{bmatrix}$ and $U = \begin{bmatrix} U_{11} & U_{12} \\ 0 & U_{22} \end{bmatrix}$.

Similar blocked version of LU Factorizatoin, such as the one used by the LAPACK routine `dgetrf` and the ScaLAPACK routine `pdgetrf`, works similarly. By dividing the matrix into blocks of size $b \times b$, at each step i of the iteration, the left block

$A(i:n, i+b-1)$ is factorized by calling some nonblocked LU routines. The row blocks on the right, $A(i:i+b-1, i+b:n)$ are updated by triangular solve, and the trailing submatrix is updated as a matrix multiply $A(i+b:n, i+b:n) = A(i+b:n, i+b:n) - A(i+b:n, i:i+b-1) * A(i:i+b-1, i+b:n)$. Most of the recursive and block algorithms depend on delaying the updates of submatrix in blocks, and performing optimized matrix multiplication and triangular solve, which is commonly achieved by some optimized version of BLAS [16, 35, 36, 60]. For more information on blocked algorithms, see for example [5, 15].

2.4.3 PetaBricks Algorithm and Setup

The underlying algorithm for our PetaBricks LU implementation offers three choices: unblocked version of LU, recursive algorithm (as described in section 2.4.2) by dividing the problem into half $n/2$, and recursive algorithm by using size $n/4$ as the upper left block. The pseudo code for it is shown in Figure 2-9. At each recursive call, the PetaBricks compiler will decide the next choices, i.e. whether to continue making recursive calls on $n/2$ or $n/4$, or switch to the unblocked algorithm. As a consequence of the composition of possible choices, the PetaBricks compile explores the possibility of varying sizes of blocking and gives the optimal choices.

In our implementation, the output L and U is stored in the same output matrix B, such that the upper triangular part of $B = U$ and the lower triangular part of $B =$ the lower traingular part of L (since the diagonal elements of L are all 1, we do not need to store them). Our unblocked LU code is also autotuned with a choice of left-looking or right-looking unblocked LU. *Right-looking* or *eager* codes perform updates of the current column at each step and the updates of all columns to the right of the current column immediately. *Left-looking* or *lazy* codes perform updates of the current column from the previous columns and then the computations for the current column at each step, i.e. all updates to a column from previous columns are performed as late as possible. The simple Matlab implementation of traditional unblocked LU in Figure 2-8 is an example of right-looking codes. For more details on left-looking and right-looking formulations, see [69].

The results were gathered on a 8-way (dual socket, quad core) Intel Xeon E7340 system running at 2.4 GHz. The system was running 64 bit CSAIL Debian 4.0 with Linux kernel 2.6.18 and GCC 4.1.2.

2.4.4 Results and Discussion

After autotuning the unbocked algorithm, the best composition of algorithms was found to be right-looking for $3 < n \leq 48$ and $n > 192$, and left-looking for $n < 3$ and $48 < n \leq 192$. Table 2.1 summarizes the algorithmic choices, and Figure 2-10 plots the timing results.

Size of input $n \times n$ matrix	Algorithm
$1 \leq n \leq 3$	Left-looking
$3 < n \leq 48$	Right-looking
$48 < n \leq 192$	Left-looking
$n > 192$	Right-looking

Table 2.1: Algorithm selection for autotuned unblocked LU

Using the autotuned unblocked algorithm as the base case for our composite recursive algorithm (Figure 2-9), we implemented and compared the performance of four algorithms in PetaBricks: Unblocked, Recursive $n/2$ down to $n = 1$, Recurisve $n/4$ down to $n = 1$, and our autotuned composite algorithm. In figure 2-11, these are labelled Unblocked, Divide-by-2, Divide-by-4 and Autotuned respectively. The input matrices tested were real square matrices with randomly generated entries. Our autotuned algorithm runs faster than any of the three primary algorithms alone as shown.

Our autotuned PetaBricks algorithm makes a recursive call on subblock size $n/4$ when $n > 384$. When $96 < n \leq 384$, recursive call on subblocks of size $n/2$ is made. The algorithm switches back to recursive $n/4$ when $24 < n \leq 96$ and once again changes to recusrive $n/2$ when n decreases further to the range $12 < n \leq 24$. When $3 < n \leq 12$, the unblocked code is called. On very small input sizes $1 < n \leq 3$, recursive algorithm on subblock sizes $n/2$ is used. Table 2.2 summarizes the algorithmic choices for our composite LU PetaBricks transform.

Size of input $n \times n$ matrix	Algorithm
$1 \leq n \leq 3$	Divide-by-2
$3 < n \leq 12$	Unblocked
$12 < n \leq 24$	Divide-by-2
$24 < n \leq 96$	Divide-by-4
$96 < n \leq 384$	Divide-by-2
$n > 384$	Divide-by-4

Table 2.2: Algorithm selection for autotuned LU

We see that the optimal algorithmic choice is nontrivial. As shown in Table 2.2, the subproblem sizes on which recursive calls are made change with n. This gives a varying blocking LU algorithm, as opposed to fixed blocking size usually implemented in common scientific pacakge such as LAPACK. Our autotuned PetaBricks transform serves as a first experiment of variable blocking strategies. Adding PetaBricks language and compiler support for variable block sizes can be helpful for further algorithmic study and performance improvement.

Similar to the eigenproblem, our autotuned LU algorithm is automatically parallel. Figure 2-12 shows the parallel scalability for autotuned LU factorization. Speedup is calculated by $S_p = \frac{T_1}{T_p}$, where p is the number of threads and T_p is the execution time of the algorithm with p threads. The plot was generated for three input sizes $n = 256, 512, 1024$ using up to 8 worker threads. The parallel speedup is sublinear, but we again obtained greater speedup as the problem size n increases.

2.4.5 Related Work

There has been a large number of studies on parallel LU factorization for both the dense and sparse cases. A large portion of both early and recent studies have mainly focused on load distribution, pivoting cost, communications, data layout, pipelining and multithreading [17, 22, 25, 44, 54].

Parallel sparse LU fatorization is based on the elimination tree [24, 66] and subtree-to-cube mapping [45, 75]. Using the idea of elimination trees and data dependency, different scheduling and ordering algorithms have been proposed and studied [48, 49, 53] .

SuperLU [64] is a general purpose library for the direct solution of large, sparse, nonsymmetric systems of linear equations on high performance machines. There are three variations of the SuperLU package, for sequential machines [30], shared memory parallel machines [31], and distributed memory parallel machines [65].

A recent study [86] showed that dense LU can be optimized in parallel using NVIDIA GPUs. The paper achieved their LU performance by techniques such as look-ahead, overlapping CPU and GPU computation, autotuning, optimizing blocked matrix multiply, and picking the right memory layout.

Algorithms that minimize communication in parallel in the expense of more arithmetics have also been proposed and studied [10, 27]. CALU, a communication avoiding LU factorization algorithm based on a new pivoting strategy referred to as ca-pivoting by the authors, is presented in a recent paper [28].

2.5 Chapter Summary

In this chapter we introduced PetaBricks, a recent implicitly parallel language that allows programmers to naturally express algorithmic choice explicitly at the language level. The PetaBricks compiler and autotuner is not only able to compose a complex program using fine-grained algorithmic choices but also find the right choice for many other parameters including data distribution, parallelization and blocking. We re-examined classic numerical algorithms with PetaBricks, and showed that the PetaBricks autotuner produces nontrivial optimal algorithms. Our results showed that the autotuned hybrid algorithms always perform better than any of the individual algorithms. In particular, we showed that that the optimal algorithm for the symmetric eigenvalue problem is different from the one used in LAPACK. Compared with hard-coded composition of algorithms in standard numerical linear packages, our autotuning approach allows automatic selection of cutoff points for algorithms when the underlying architecture changes. We also demonstrated the speedup of LU Factorization implemented using our autotuned nontrivial varaible blocking algorithm over conventional fixed recursive blocking strategies with fixed blocking sizes. Our im-

plementations using PetaBricks give portable performance that can adapt to changes in architecture and produce the optimal algorithmic choices and cutoff accordingly.

```
1  transform MatrixMultiply
2  from A[p,m], B[n,p]
3  to C[n,m]
4  {
5    // Rule 0: Base case, compute a single element
6    to(C.cell(j,i) out)
7    from(A.row(i) a, B.column(j) b) {
8      out = DotProduct(a,b);
9    }
10
11   // Rule 1: Recursively decompose A in half
12   //         col-blocks and B in half row-blocks
13   to(C c)
14   from(A.region(0,   0, p/2, m  ) a1,
15        A.region(p/2, 0, p,   m  ) a2,
16        B.region(0,   0, n,   p/2) b1,
17        B.region(0, p/2, n,   p  ) b2) {
18     c = MatrixAdd(MatrixMultiply(a1, b1),
19                   MatrixMultiply(a2, b2));
20   }
21
22   // Rule 2: Recursively decompose B in half col-blocks
23   to(C.region(0,   0, n/2, m  ) c1,
24      C.region(n/2, 0, n,   m  ) c2)
25   from( A a,
26         B.region(0,   0, n/2, p  ) b1,
27         B.region(n/2, 0, n,   p  ) b2) {
28     c1 = MatrixMultiply(a, b1);
29     c2 = MatrixMultiply(a, b2);
30   }
31
32   // Rule 3: Recursively decompose A in half row-blocks
33   to(C.region(0, 0,   n, m/2) c1,
34      C.region(0, m/2, n, m  ) c2)
35   from(A.region(0,   0, p,   m/2) a1,
36        A.region(0, m/2, p,   m  ) a2,
37        B b) {
38     c1 = MatrixMultiply(a1, b);
39     c2 = MatrixMultiply(a2, b);
40   }
41 }
```

Figure 2-1: PetaBricks source code for MatrixMultiply

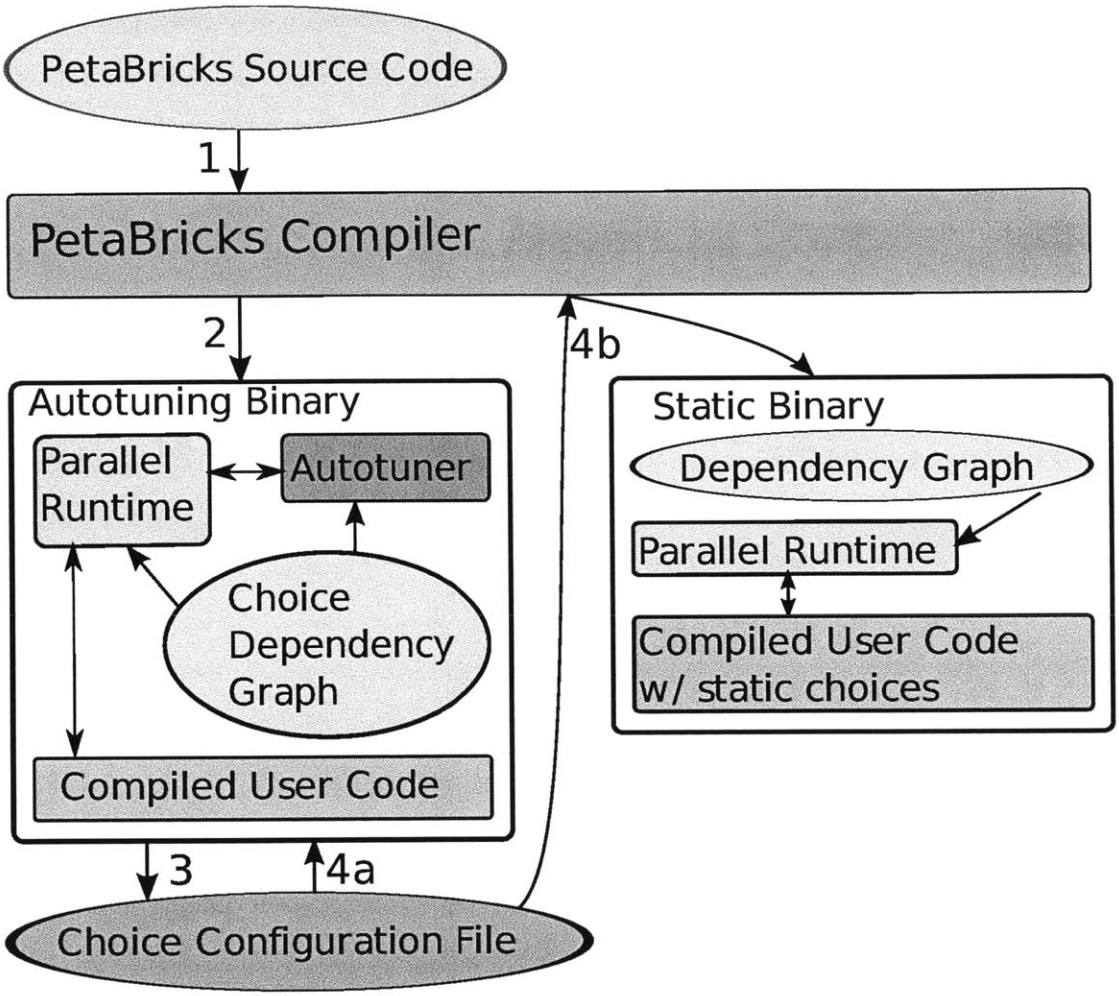

Figure 2-2: Interactions between the compiler and output binaries. First, the compiler reads the source code and generates an autotuning binary (Steps 1 and 2). Next (Step 3), autotuning is run to generate a choice configuration file. Finally, either the autotuning binary is used with the configuration file (Step 4a), or the configuration file is fed back into a new run of the compiler to generate a statically chosen binary (Step 4b).

```
1  transform CumulativeSum
2  from A[n]
3  to B[n]
4  {
5    //Rule 0: sum all elements to the left
6    to(B.cell(i) b) from(A.region(0, i) in) {
7      b=sum(in);
8    }
9
10   //Rule 1: use the previously computed sum
11   to(B.cell(i) b) from(A.cell(i) a,
12                        B.cell(i-1) leftSum) {
13     b=a+leftSum;
14   }
15 }
```

Figure 2-3: PetaBricks source code for CumulativeSum. A simple example used to demonstrate the compilation process. The output element B_k is the sum of the input elements $A_0, \ldots, A_k$.

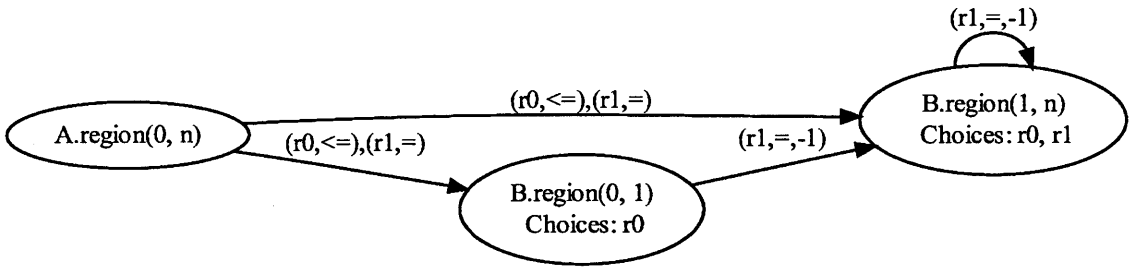

Figure 2-4: *Choice dependency graph* for CumulativeSum (in Figure 2-3). Arrows point the opposite direction of dependency (the direction data flows). Edges are annotated with rules and directions, offsets of 0 are not shown.

EIG(T)

1: **either**
2: Use QR to find Λ and X
3: Use BISECTION to find Λ and X
4: Recursively call EIG on submatrices T_1 and T_2 to get Λ_1, X_1, Λ_2 and X_2. Use results to compute Λ and X.
5: **end either**

Figure 2-5: Pseudo code for symmetric eigenproblem. Input T is tridigonal

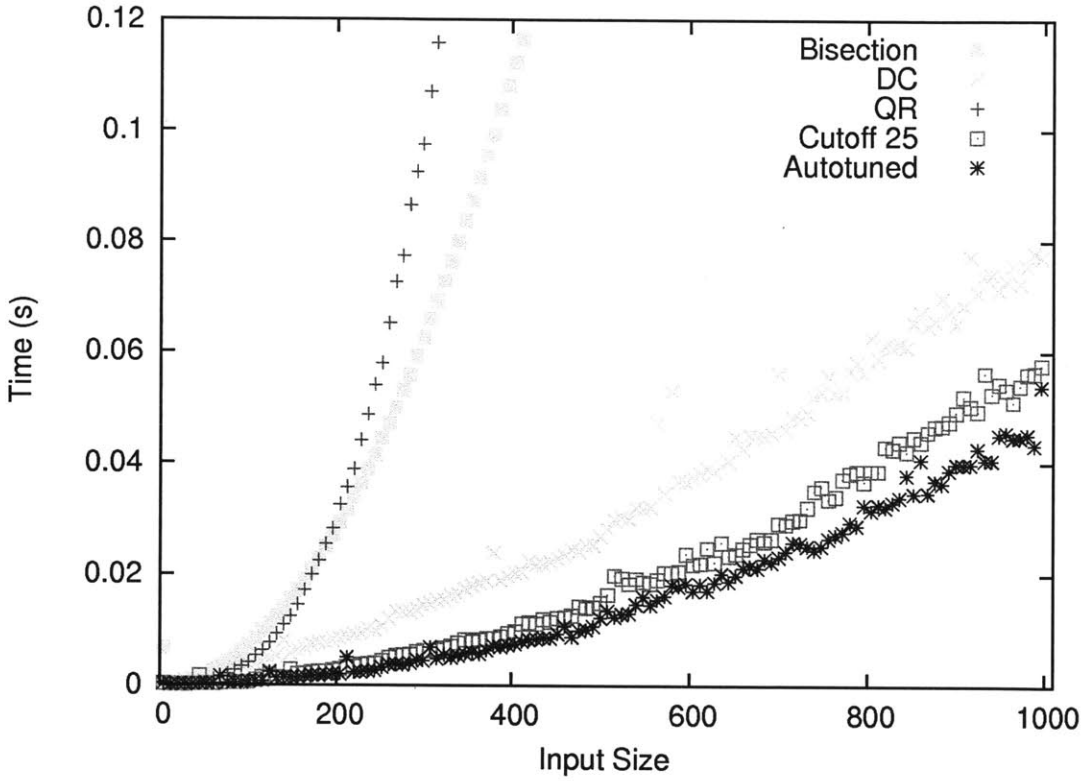

Figure 2-6: Performance for Eigenproblem on 8 cores. "Cutoff 25" corresponds to the hard-coded hybrid algorithm found in LAPACK.

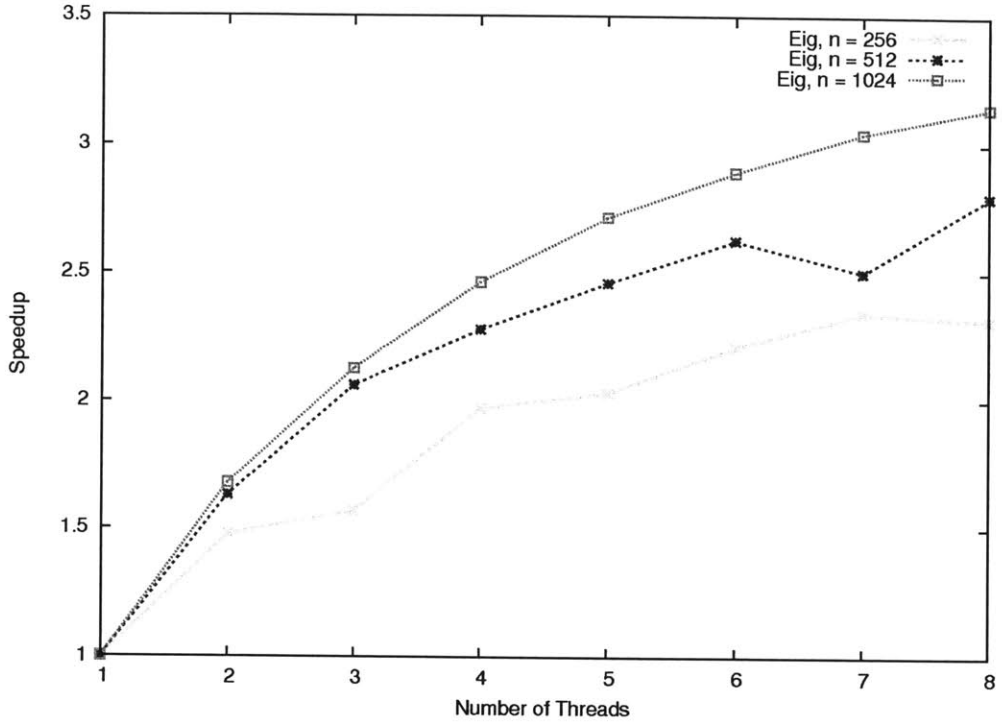

Figure 2-7: Parallel scalability for eigenproblem: Speedup as more worker threads are added. Run on an 8-way (2 processor 4 core) x86 64 Intel Xeon System.

```
1 function [L U]=lunopivot(A);
2 n=size(A,1);
3 U = A;  L = eyes(n);  % Initialize U = A, and L = I
4 for k = 1 to n-1
5     for j = k+1 to m
6         L(j,k) = U(j,k) / U(k,k);
7         U(j,k:m) = U(j,k:m) - L(j,k)*U(k,k:m);
8     end
9 end
```

Figure 2-8: Simple Matlab implementation of right-looking LU

LU(A)

1: **either**
2: Use `LUunblocked` to find L and U
3: Recursively call `LU` on subproblems of equal size $n/2$.
4: Recursively call `LU` on subproblems of sizes $n/4$ and $3n/4$.
5: **end either**

Figure 2-9: Pseudo code for LU Factorization. Input A is $n \times n$

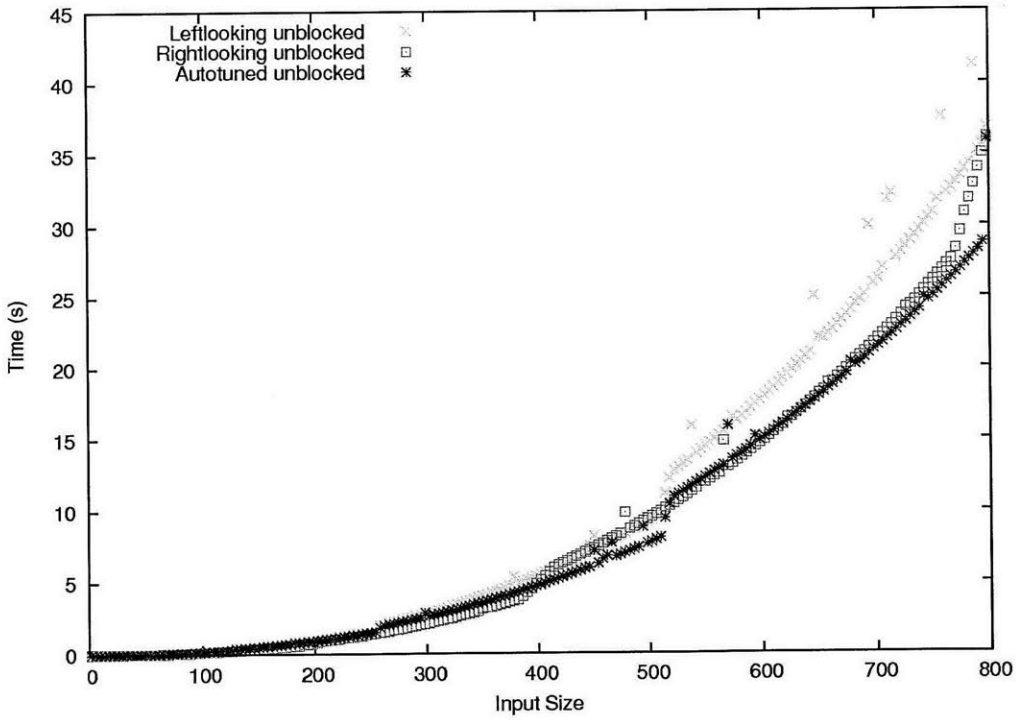

Figure 2-10: Performance for Non-blocked LU Factorization on 8 cores.

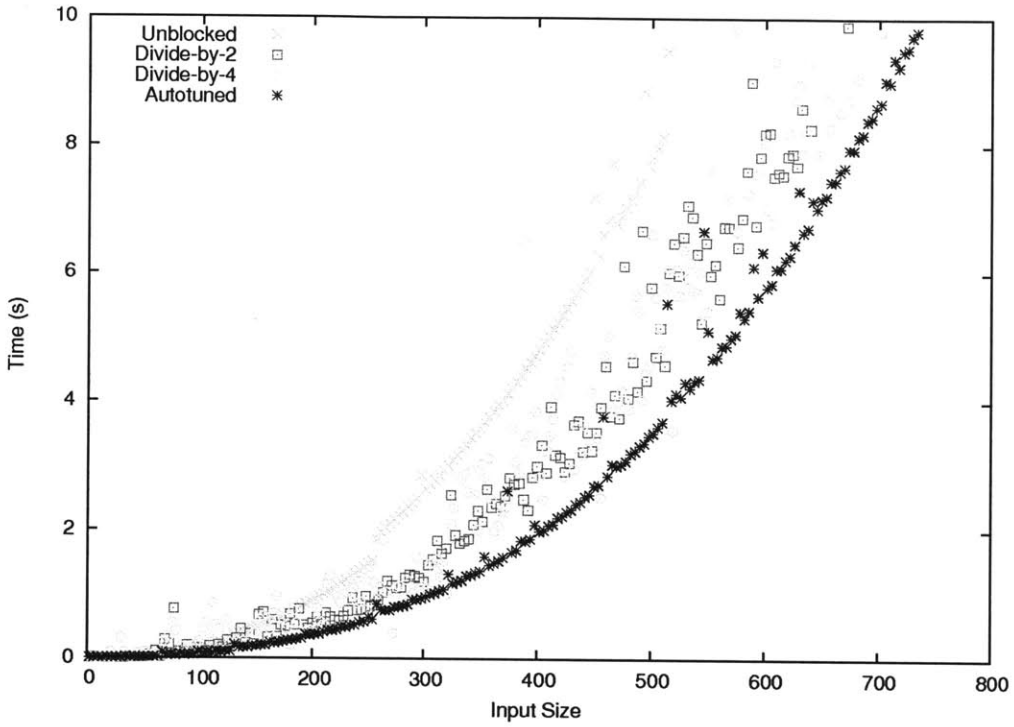

Figure 2-11: Performance for LU Factorization on 8 cores. "Unblocked" uses the autotuned unblocked transform from Figure 2-10.

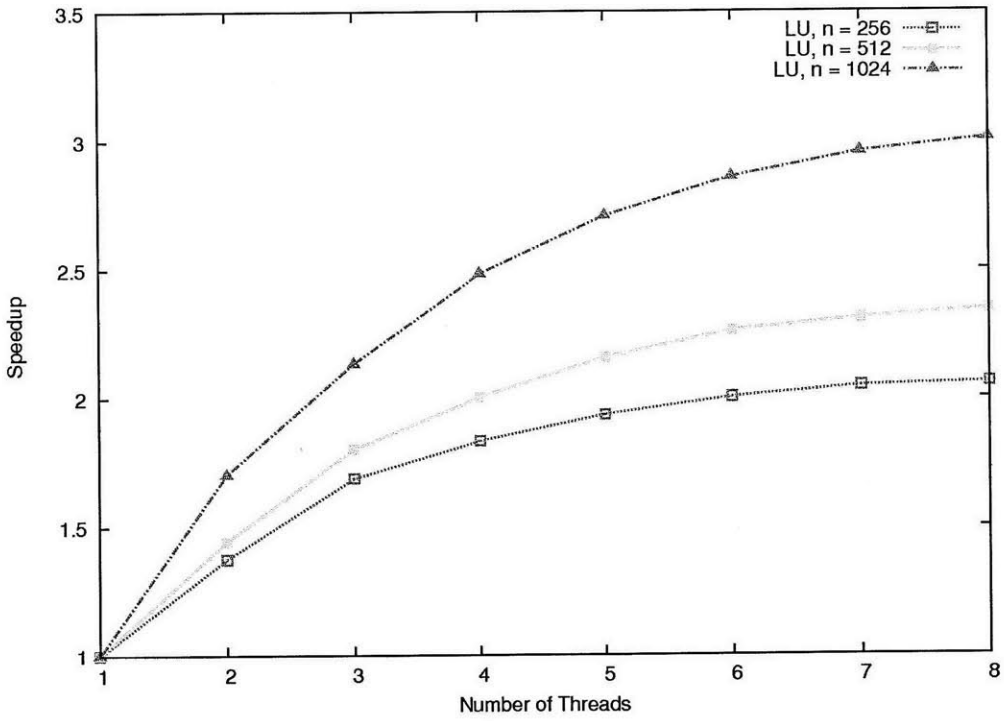

Figure 2-12: Parallel scalability for LU Factorization: Speedup as more worker threads are added. Run on an 8-way (2 processor 4 core) x86 64 Intel Xeon System.

Chapter 3

Handling Variable-Accuracy with PetaBricks

3.1 Introduction

In chapter 2, we demonstrated how PetaBricks can be used to revisit some classical numerical kernels with multiple algorithmic choice being autotuned. The examples shown were only limited to problems with direct solutions. However, for certain classes of applications, such as NP-hard problems or problems with tight computation or timing constraints, we are often willing to sacrifice some level of accuracy for faster performance. In this chapter, we broadly define these types of problems as *variable accuracy* algorithms, and discuss how PetaBricks handle the notion of variable accuracy.

One class of variable accuracy algorithms are *approximation algorithms* in the area of soft computing [94]. Approximation algorithms are used to find approximate solutions to computationally difficult tasks with results that have provable quality. For many computationally hard problems, it is possible to find such approximate solutions asymptotically faster than it is to find an optimal solution. A good example of this is BinPacking. Solving the BinPacking problem is NP-hard, yet arbitrarily accurate solutions may be found in polynomial time [26]. Like many soft computing problems, BinPacking has many different approximation algorithms, and the best

choice often depends on the level of accuracy desired.

Another class of variable accuracy algorithms are *iterative algorithms* used extensively in the field of applied mathematics. These algorithms iteratively compute approximate values that converge toward an optimal solution. Often, the rate of convergence slows dramatically as one approaches the solution, and in some cases a perfect solution cannot be obtained without an infinite number of iterations [93]. In many cases, *convergence criteria* are created to decide when to terminate the iteration. However, deciding on a convergence criteria has become increasingly difficult with more complex memory system, architecture and advances in multicore computing.

A third class of variable accuracy algorithms are algorithms in the signal and image processing domain. In this domain, the accuracy of an algorithm can be directly determined from the problem specification. For example, when designing digital signal processing (DSP) filters, the type and order of the filter can be determined directly from the desired sizes of the stop, transition and pass-bands as well as the required filtering tolerance bounds in the stop and pass-bands. When these specifications change, the optimal filter type may also change. Since many options exist, determining the best approach is often difficult, especially if the exact requirements of the system are not known ahead of time.

A key challenge when writing and using codes for variable accuracy algorithms arises from the optimal composition of algorithms with accuracy requirements. For example, a user may know all of the potential algorithms to solve a variable accuracy problem but may not know how to link together the algorithms and determine what parameter values should be associated with each. In some cases, a user may know very little about the underlying algorithms, but he/she may just need to solve a problem to some target accuracy level.

An example is the `fmincon()` function in Matlab, which attempts to find the minimum of a user-specified nonlinear multivariate function subject to a set of specified constraints. `fmincon()` takes accuracy and optimization options specified by an `options` structure. This structure contains 42 fields that the user can set to specify

various options such as which of three algorithms to use, how many iterations to run, and what tolerances to use. Additionally, there are a number of options specific to each of the three algorithms, some of which further affect additional algorithmic choices. For example, the value specified in the `PrecondBandWidth` option used by the `trust-region-reflective` algorithm will indirectly affect both the number of preconditioning iterations performed, as well as the type of factorization algorithm used during the preconditioning phase. Relying on the user to specific all the options is simply not the most effective way to obtain the optimal algorithm.

In this chapter, we demonstrate how a novel set of language extensions to PetaBricks and an accuracy-aware compiler can address the challenges in writing variable accuracy codes. With our extensions, accuracy time trade-offs are made visible to the compiler, enabling it to perform empirical autotuning over both the algorithmic search space and the parameter space to find the best composition of nested calls to variable accuracy code. The resulting code will perform well across architectures as none of the accuracy-based decisions need to be hard-coded.

Some of the materials in this chapter appear in our recent paper [7]. This chapter discusses in more detail the results of our autotuned benchmark algorithms. In particular, we demonstrate how k-means clustering can be solved without specifying the number of clusters k, and show that the optimal k can be determined accurately with PetaBricks. To the best of our knowledge, our approach is the first to solve any general-purpose k-means clustering problem without domain-specific modeling and any solution of optimization problems by the users to specific the number of clusters k. We also show how accuracy requirement is incorporated into the problem of preconditioning, and show that the optimal choice of preconditioners can change with problem sizes. Compared with common preconditioning techniques, our implementation allows automatic choices of optimal preconditioners without prior knowledge of the specific system matrix.

3.2 PetaBricks for Variable Accuracy

At a high level, the language extensions to PetaBricks extend the idea of algorithmic choice to include choices between different accuracies. The extensions also allow the user to specify how accuracy should be measured. Our new accuracy-aware autotuner then searches to optimize for both time and accuracy. The result is code that probabilistically meets users' accuracy needs. Optionally, users can request hard guarantees that utilize runtime checking of accuracy.

For more information about the PetaBricks language and compiler support for variable accuracy, see [7]; the following summary is included for background.

3.2.1 Variable Accuracy Extensions

In order to support variable accuracy, the following extensions were made to PetaBricks:

- The `accuracy_metric` keyword in the `transform header` allows the programmer to specify the name of another user-defined transform to compute accuracy from an input/output pair. This allows the compiler to test the accuracy of different candidate algorithms during training. It also allows the user to specify a domain specific accuracy metric of interest to them.

- The `accuracy_variable` keyword in the `transform header` allows the user to define one or more algorithm-specific parameters that influence the accuracy of the program. These variables are set automatically during training and are assigned different values for different input sizes. The compiler explores different values of these variables to create candidate algorithms that meet accuracy requirements while minimizing execution time.

- The `accuracy_bins` keyword in the `transform header` allows the user to define the range of accuracies that should be trained for and special accuracy values of interest that should receive additional training. This field is optional and the compiler can add such values of interest automatically based on how a transform is used. If not specified, the default range of accuracies is 0 to 1.0.

- The `for_enough` statement defines a loop with a compiler-set number of iterations. This is useful for defining iterative algorithms. This is syntactic sugar for adding an `accuracy_variable` to specify the number of iterations of a traditional loop.

- The keyword `verify_accuracy` in the rule body directs the compiler to insert a runtime check for the level of accuracy attained. If this check fails the algorithm can be retried with the next higher level of accuracy or the user can provide custom code to handle this case. This keyword can be used when strict accuracy guarantees, rather than probabilistic guarantees, are desired for all program inputs.

3.2.2 Example Psueudocode

Figure 3-1 presents our kmeans example with our new variable accuracy extensions. This kmeans program groups the input `Points` into a number of clusters and writes each points cluster to the output `Assignments`. Internally the program uses the intermediate data `Centroids` to keep track of the current center of each cluster. The transform header declares each of these data structures as its inputs (`Points`), outputs (`Assignments`), and intermediate or "through" data structures (`Centroids`) (line 4-7).

First, the keyword `accuracy_metric`, on line 2, defines an additional transform, `kmeansaccuracy`, which computes the accuracy of a given input/output pair to kmeans. PetaBricks uses this transform during autotuning and sometimes at runtime to test the accuracy of a given configuration of the kmeans transform. The accuracy metric transform computes the value $\sqrt{\frac{2n}{\sum D_i^2}}$, where D_i is the Euclidean distance between the i-th data point and its cluster center.

The `accuracy_variable` k, on line 3 controls the number of clusters the algorithm generates by changing the size of the array `Centroids`. The variable k can take different values for different input sizes and different accuracy levels. The compiler will automatically find an assignment of this variable during training that meets each

required accuracy level.

The rules contained in the body of the transform define the various pathways to construct the `Assignments` data from the initial `Points` data. The first two rules (line 9-21) specify two different ways to initialize the `Centroids` data needed by the iterative kmeans solver in Rule 3. The third rule (line 23-32) specifies how to produce the output Assignments using both the input `Points` and intermediate `Centroids`. Note that since the third rule depends on the output of either the first or second rule, the third rule will not be executed until the intermediate data structure `Centroids` has been computed by one of the first two rules. The `for_enough` loop on line 26 is a loop where the compiler can pick the number of iterations needed for each accuracy level and input size. During training, the compiler will explore different assignments of k, algorithmic choices of how to initialize the `Centroids`, and iteration counts for the `for_enough` loop to try to find optimal algorithms for each required accuracy.

To summarize, when our transform is executed, the cluster centroids are initialized either by the Rule 1, which performs random initialization on a per-column basis with synthesized outer control flow, or by Rule 2, which calls the `CenterPlus` algorithm. Once `Centroids` is generated, the iterative step in Rule 3 is called. Our actual implemented version of k-means clustering code varies slightly and incoporates more algorithmic choice. A more detailed discussion on our clustering benchmark can be found in Section 3.3.

3.2.3 Accuracy Guarantees

PetaBricks supports the following three types of accuracy guarantees:

- *Statistical guarantees* are the most common technique used, and the default behavior of our system. They work by performing off-line testing of accuracy using a set of program inputs to determine statistical bounds on an accuracy metric to within a desired level of confidence.

- *Runtime checking* can provide a hard guarantee of accuracy by testing accuracy at runtime and performing additional work if accuracy requirements are not

met. Runtime checking can be inserted using the `verify_accuracy` keyword. This technique is most useful when the accuracy of an algorithm can be tested with low cost and may be more desirable in case where statistical guarantees are not sufficient.

- *Domain specific guarantees* are available for many types of algorithms. In these cases, a programmer may have additional knowledge, such as a lower bound accuracy proof or a proof that the accuracy of an algorithm is independent of data, that can reduce or eliminate the cost of runtime checking without sacrificing strong guarantees on accuracy.

  As with variable accuracy code written without language support, deciding which of these techniques to use with what accuracy metrics is a decision left to the programmer.

3.2.4 Compiler Support for Autotuning Variable Accuracy

The main difficulty of representing variable accuracy algorithms is that variable accuracy adds a new dimension to how one can evaluate candidate algorithms. With fixed accuracy algorithms, the metric of performance can be used to order algorithms. With variable accuracy, we plot candidates on an accuracy/time grid. This naturally leads to an *optimal frontier* of algorithms for which no other algorithm can provide a greater accuracy in less time. It is not possible to evaluate the entire optimal frontier, however, since it can potentially be of infinite size. Instead, to make this problem tractable, we discretize the space of accuracies by placing each allowable accuracy into a bin. The discretization can be specified by the user or can be automatically inferred by the compiler based on how a variable accuracy algorithm is used. For example, if an algorithm is called with a specific accuracy, that specific accuracy can be added as extra bin boundary by the compiler. An example of the optimal frontier and bins is shown in Figure 3-2.

In the compiler, we represent these bins by extending and using the representation for templates. A variable accuracy algorithm is called with the syntax "`Foo<accuracy>`"

and, similar to templates, each requested accuracy is considered by the compiler as a separate type. When variable accuracy code calls other variable accuracy code, the sub-accuracy is automatically determined by the compiler. This is done by representing the sub-accuracy as an algorithmic choice to call one of any of the accuracy bins. If a user wishes to call a transform with an unknown accuracy level, we support dynamically looking up the correct bin that will obtain a requested accuracy.

The PetaBricks autotuner then searchs the algorithm spaces using this optimal frontier. Using a dynamic programming approach, it produces the optimal algorithmic choice that meets the accuracy requirement. For details of the actual tuning algorithm and phases, see [7].

3.3 Clustering

3.3.1 Background and Challenges

Clustering is the problem of grouping similar objects. Given a set of input, clustering divides the data into clusters based on a similarity measure, which is often specific to the domain of application. Clustering is a common technique for statistical data analysis in areas including machine learning, pattern recognition, image segmentation, medicine, computational biology. Many clustering algorithms, data structures and cluster modeling have been studied [52, 59].

For two objects i, j, a common way to measure similarity is to define a distance measure $D(i, j)$. Objects are considered to be more similar with a smaller distance $D(i, j)$ between each other. Common distance functions include the Euclidean distance (2-norm), the Manhattan distance (1-norm), the supremum norm and the Hamming distance. Since the choice of distance measure will affect the shape of clusters, it depends on the application and requires some prior knowledge of the dataset.

In this section, we study a popular algorithm of partitional clustering, the *k-means clustering* using PetaBricks. K-means clustering gives a partition of n objects into k clusters, measured by a mean error. The large number of possible partitions makes it

difficult and impratical to search for an absolute minimum configuration. Thus local optimization algorithms are usually applied.

Many clustering algorithms, including k-means, require the specification of the number of clusters to be used, prior to exceution of the algorithms. The problem of k-clustering is NP-hard for general k in a plane [67] (for $k < n$). If k is fixed, k-clustering can be solved in polynomial time [56]. However, determining the optimal number k is a difficult problem by itself. The best choice of k is often not obvious since it depends on the underlying distribution of data and desired accuracy. There have been many studies on choosing k, such as setting k simply to $k = \sqrt{n/2}$ [68], the Elbow Method [47], and by an information theoretic approach [82]. We present our Petabricks solutions to the k-means clustering problem, which includes algorithmic selection and automatic computation of the number of clusters k by the Petabricks autotuner based on a preset accuracy metric. We demonstrate how k-means clustering can be solved without specifying the number of clusters k, and show that the optimal k can be determined accurately with PetaBricks. To the best of our knowledge, our approach is the first to solve any general-purpose k-means clustering problem without domain-specific modeling and any solution of optimization problems by the users to specific the number of clusters k.

3.3.2 Algorithms for k-means clustering

The first step of solving the k-means problem is to find the number of clusters k in the data set. In our PetaBricks transform, the number of clusters, k, is the accuracy variable to be determined on training by the autotuner.

Taking k as given (which will be determined by the autotuner), we implemented a variant of Lloyd's algorithm [72] for k-means clustering. Lloyd's algorithm starts with k initial centers chosen randomly or by some heuristics. Each data point is assigned to its closest center, measured by some distance metric. We pick $D(i, j)$ to be the Euclidean distance in our implementation. The cluster centers are then updated to be the mean of all the points assigned to the corresponding clusters. The steps of partitioning points and recalculating cluster centers are repeated until convergence

to a local optimal configuration. Several algorithmic choices are implemented in our version of k-means clustering: The initial set of k cluster centers are either chosen randomly with a uniform distribution among the n data points, or according to the k-means++ algorithm [8], which selects subsequent centers from the remaining data points with probability proportional to the distance squared to the closest center. Once the initial cluster centers are computed, the final cluster assignments and center positions are determined by iterating, either until a fixed point is reached or in some cases when the compiler decides to stop early.

3.3.3 Experimental Setup - Acuracy Metric and Training Data

The pseudo code for k-means clustering is shown in Figure 3-3. There are two places for algorithmic choices. First, the initial set of k cluster centers are either chosen randomly, or by the k-means++ algorithm. During the iterative steps, the compiler can choose to continue the iterative phase until fixed cluster centers and assignments are reached, or stop the iteration when no more than 50% of the cluster assignments change, or no more than 25% of the points are assigned a different cluster, or stop after only one round of iteration. The Petabricks compiler incorporates the accuracy metric and accuracy level requirements in making these algorithmic choices.

The training data is a randomly generated clustered set of n points in two dimensions. First, $\sqrt{n}$ "center" points are uniformly generated from the region [-250, 250]×[-250, 250]. The remaining $n - \sqrt{n}$ data points are distributed evenly to each of the $\sqrt{n}$ centers by adding a random number generated from a standard normal distribution to the corresponding center point. Note that the optimal number of clusters $k_{\text{optimal}} = k_{\text{source}} = \sqrt{n}$ is not known to the autotuner.

Rather than assigning a fixed k through a heuristic (such as the commonly used $k = \sqrt{n/2}$), we define k as an accuracy variable and allow the autotuner to set it. This allows the number of clusters to change based on how compact clusters the user of the algorithm requests through the accuracy requirement. The accuracy metric

used is

$$\text{Accuracy\_metric} = \sqrt{\frac{2n}{\sum D_i^2}}, \tag{3.1}$$

where D_i is the Euclidean distance between the i-th data point and its cluster center. The reciprocal of the distance is chosen as the accuracy metric such that a smaller sum of distance squared gives a higher accuracy.

The accuracy levels used are 0.05, 0.10, 0.20, 0.50, 0.75, and 0.95. The accuracy metric is chosen such that with the input training data, the resulting accuracy produced will lie between [0,1] (which is an arbitrary choice).

We performed all tests on a 3.16 GHz 8-core (dual-Xeon X5460) system. All codes are automatically parallelized by the PetaBricks compiler and were run and trained using 8 threads.

3.3.4 Results and Analysis

Figure 3-4 shows the speedups that are attainable when a user is in a position to use an accuracy lower than the maximum accuracies of our benchmarks. On the largest tested input size, our Clustering benchmark speedups range from 1.1 to 9.6x. Such dramatic speedups are a result of algorithmic changes made by our autotuner that can change the asymptotic performance of the algorithm (For example, $O(n)$ vs $O(n^2)$) when allowed by a change in desired accuracy level. Because of this, speedup can become a function of input size and will grow arbitrarily high for larger and larger inputs. These speedups demonstrate some of the performance improvement potentials available to programmers using our system.

Table 3.1 illustrates the algorithmic results of autotuning our k-means benchmark on our sample input of size $n = 2048$. The results show interesting algorithmic choices and number of clusters k chosen by the autotuner. For example, at accuracies greater than 0.2, the autotuned algorithm correctly uses the accuracy metric (based on Euclidean distances between data points and cluster centers) to construct an algorithm that picks a k value that is close to 45, which is the number of clusters generated by

Accuracy	k	Initial Center	Iteration Algorithm
0.10	4	random	once
0.20	38	k-means++	25% stabilize
0.50	43	k-means++	once
0.75	45	k-means++	once
0.95	46	k-means++	100% stabilize

Table 3.1: Algorithm selection and initial k value results for autotuned k-means benchmark for various accuracy levels with $n = 2048$ and $k_{\text{source}} = 45$

our training data (which is not known to the autotuner).

At accuracy 0.1, the autotuner determines 4 to be the best choice of k and chooses to start with a random cluster assignment with only one level of iteration. While this is a very rough estimate of k and a very rough cluster assignment policy, it is sufficient to achieve the desired low level of accuracy. To achieve accuracy 0.2, the autotuner uses 38 clusters, which is slightly less than the predetermined value. Our autotuned algorithm determines the initial cluster centers by k-means++, and iterates until no more than 25% of the cluster assignments change. For accuracy 0.5 and 0.75, the values of k picked by the autotuner algorithm are 43 and 45 respectively, which are only slightly smaller or equal to the predetermined k. The initial centers are decided by k-means++ and only one iteration is used. By successfully finding a number of clusters that is close to the predetermined k and picking good initial centers, only one iteration is needed on average during training to achieve a high level of accuracy. Finally, to achieve the highest accuracy of 0.95, the algorithm uses k value of 46. Initial centers are determined by k-means++ and iterations are performed until a fixed point is reached. It is interesting to note that on average, the autotuner finds that a value of k that is one higher than the k used to generate the data, is best to minimize the user specified accuracy metric.

Our autotuner is able to produce the optimal number of clusters accurately with most of the accuracy levels and only overpredicts the number by 1 with a very strict accuracy requirement specificed by user. As a comparison to highlight the fact that input training data is important, we also ran our k-means clustering benchmark on an input training data randomly generated from uniform distribution without any

clustering built in. In that case, we found that the number of clusters k produced by our autotuner simply increased with increasing accuracy requirements.

3.4 Preconditioning

3.4.1 Background and Challenges

Solving a linear system of equations $Ax = b$ is a common problem in both scientific research and real-world applications such as cost optimization and asset pricing. Non-iterative (or "direct") algorithms for general $n \times n$ matrices require $O(n^3)$ flops, which usually makes solving the equation $Ax = b$ the bottleneck of any application, especially when n gets large. Thus, iterative methods are often used to provide approximate solutions. Preconditioning is a technique that speeds up the convergence of an iterative solver.

The convergence of a matrix iteration depends on the properties of the matrix A, one of which is the condition number. A preconditioner M of a matrix A is a matrix that if well chosen, the condition number of $M^{-1}A$ is smaller than that of A. Although the preconditioned system $M^{-1}Ax = M^{-1}b$ has the same solution as the original system $Ax = b$, the rate of convergence depends on the condition number of $M^{-1}A$. The preconditioner $M = A$ has the optimal condition number, but evaluating $M^{-1}b = A^{-1}b$ is equivalent to solving the original system. If $M = I$, then the preconditioned system is the same as the original system, so it accomplished nothing even though the operation $M^{-1}b$ in this case is trivial. Achieving a faster convergence rate of the iterative solver (finding a preconditioner M that is close to A) while keeping the operation of M^{-1} simple to compute is the key to finding a good preconditioner.

For M to be considered close to A, we want the eigenvalues of $M^{-1}A$ to be close to 1 and the 2-norm $||M^{-1}A - I||_2$ to be small. In that case, the iterative solvers can be expected to converge quickly. A general rule of thumb suggested by [85] is that *a preconditioner M is good if $M^{-1}A$ is not too far from normal and its eigenvalues are*

clustered (A matrix A is *normal* if $A^*A = AA^*$).

In this section, we present our PetaBricks implementation of Preconditioned Conjugate Gradient (PCG). In our approach, we incorporate accuracy requirement into the problem of preconditioning, and show that the optimal choice of preconditioners can change with problem sizes. Our implementation allows automatic choices of optimal preconditioners without prior knowledge of the specific system matrix.

3.4.2 Overview of Preconditioners

We first give a survey of common preconditioners based on [85, 11]. Some examples of preconditioners are defined independent of the properties of the underlying system $Ax = b$, while other preconditioners are designed to take advantage of the specific structures of the original matrix A.

- *Jacobi preconditioner* This is perhaps the simplest preconditioner, defined by $M = \text{diag}(A)$. This transformation can speed up the iteration considerably for certain problems.

- *Polynomial preconditioner* This type of preconditioner aims at approximate directly the inverse of A. A polynomial preconditioner is a matrix polynomial $M^{-1} = p(A)$ such that $p(A)A$ has better convergence properties than A itself. One way to obtain the polynomial $p(A)$ is from the first few terms of the Neumann series $A^{-1} = I + (I - A) + (I - A)^2 + ...$

- *A few steps of classical iterative method* Another common preconditioning technique is to apply one or more steps of "classical iterative methods" such as Jacobi, Gauss-Seidel, SOR or SSOR. Popular choices are Jacobi and SSOR. For details of these iterative methods, see [11].

- *Multigrid* This is a common technique particularly in solving partial differential equation or integral equations. The idea is to restrict the discretized equation on a coarser grid, solve it on the coarse grid, and interpolate back to the finer grid.

3.4.3 Experimental Setup - Acuracy Metric and Training Data

Our preconditioner PetaBricks transform implements three choices of preconditioners and solves the system by Conjugate Gradient Method (CG)[1]. The pseudo code is shown in Figure 3-5. The first choice is the Jacobi preconditioner $M = \text{diag}(A)$ coupled with Preconditoned Conjugate Gradient (PCG). Another choice is to apply the polynomial preconditioner $M^{-1} = p(A)$, where $p(A)$ is an approximation of the inverse of A by using the first three terms of the series expansion of A^{-1}, and solve the preconditioned system with PCG. We also implemented the Conjugate Gradient method (CG) which solves the system without any preconditioning.

The number of iterations `NumIterations` is defined as an accuracy variable and we allow the autotuner to set it based on the algorithm chosen and required accuracy level. The accuracy metric used is defined as the ratio between the RMS error of the initial guess Ax_{in} to the RMS error of the output Ax_{out} compared to the right hand side vector b, converted to log-scale, which is equal to

$$\text{Accuracy\_metric} = \log\left(\sqrt{\frac{\sum_i((Ax_{in})_i - b_i)^2}{\sum_i((Ax_{out})_i - b_i)^2}}\right), \tag{3.2}$$

For training data, we used randomly generated entries for the RHS vector b. For the coefficient matrix A, we tried two different sets of training data, (1) set A to be the discretized operator of the 2D Poisson Equation, and (2) randomly generated symmetric positive-definite matrix A for comparison in terms of algorithmic choices with the Poisson operator.

The accuracy levels used are 0, 0.5, 1, 1.5, 2 and 3. With the accuracy metric defined as above, the accuracy levels require increasing orders of magnitude of improvements of the error norm compared to the initial guess.

We performed all tests on a 3.16 GHz 8-core (dual-Xeon X5460) system. All codes are automatically parallelized by the PetaBricks compiler and were run and trained

[1]Conjugate Gradient is a Krylov subspace iterative algorithm to solve the equation $Ax = b$ for symmetric positive-definite matrices A. For details, see [29, 70]

using 8 threads.

3.4.4 Results and Analysis

Figure 3-6 shows the speedups that are attainable when a user is in a position to use an accuracy lower than the maximum accuracies of our benchmarks. On the largest tested input size, our Preconditioner benchmark speedups range from 1.1 to 2.2x. The speedups are not as significant as the Clustering benchmark (Figure 3-4). One reason is that algorithmic choices made by our autotuner are limited to the kind of preconditioners used and number of iterations performed. Many of the classical preconditioners studied here have a large sequential component, so there are less room for parallel speedups than other benchmarks examined in this chapter and in the paper [7].

n (Poisson A)	NumIterations	n (Random A)	NumIterations
2^1	2	2^1	2
2^2	3	2^2	4
2^3	7	2^3	6
2^4	12	2^4	14
2^5	21	2^5	34
2^6	38	2^6	95
2^7	91	2^7	240
2^8	174	2^8	> 500

Table 3.2: Values of `NumIterations` for autotuned preconditioning benchmark for accuracy level 2 and various size of the $n \times n$ input A

To study the effects of algorithmic choice, we focus on the accuracy level 2, and examine the autotuned results. Table 3.2 lists the value of the accuracy variable `NumIterations` after autotuning our preconditioning benchmark on A, where A is either the Poission operator or a randomly generated coefficient matrix. As expected, the values of `NumIterations` increases as n increases. The number of iterations needed to attain the same level of accuracy is larger for a random system matrix A than the Poisson operator.

The values of `NumIterations` are coupled with the algorithmic choices made by the PetaBricks autotuner, as shown in Table 3.3. For the Poisson operator A, at

Size of input $n \times n$ matrix	Iteration Algorithm
$1 \leq n \leq 3$	CG
$3 < n \leq 24$	PCG with Jacobi Preconditioner
$24 < n \leq 48$	CG
$n > 48$	PCG with Jacobi Preconditioner

Table 3.3: Algorithm selection for autotuned preconditoining benchmark, accuracy level = 2 and input A is the Poisson operator

accuracy level 2, the autotuned optimal algorithmic choices is PCG with Jacobi preconditioner when $n > 48$. The iteration algorithm is switched to CG without preconditioning when $24 < n \leq 48$. It switches back to PCG with Jacobi when n falls below 24 and the optimal algorithm for very small input $n \leq 3$ is ordinary CG. It is expected that Jacobi preconditioning is effectve for the Poisson opeartor, but the switch between PCG with Jacobi and CG without preconditioner is rather interesting. One reason could be that our test measures execution times of the entire kernel, instead of using the number of iterations as the only measure as in some classical studies in the math literature. Since modern machines are highly optimized to perform matrix-matrix and matrix-vector multiplications (which are the main components of Conjugate Gradients), the number of iterations itself do not give a good indication of actual convergence rate. Furthermore, with the increasing complexity of computer architecture and parallelism involved, it is difficult to find the optimal algorithmic configurations manually.

In comparison, when A is a randomly generated matrix, the Petabricks compiler always chooses CG as the optimal choice. This makes sense since without any specific structure, it is unlikely our available choices of preconditioner will speed up the convergence.

3.4.5 Related Work

A detailed study of another common preconditioning technique, Multigrid method, using PetaBricks is presented in [18]. In the paper, the dynamic programming method of searching the exponential space of tuned algorithms (iterative algorithms and num-

ber of iterations at each recursion level) is discussed in detail. Nontrivial multigrid cycle shapes, as opposed to common V-shaped and W-shaped cycles, are found to be the optimal algrithmic paths for convergence. These cycle shapes determine the orders in which grid coarsening and grid refinement are performed with both direct methods and iterative methods, including Jacobi and SOR. The autotuned multigrid cycle shapes are targeted to the user's specific combination of underlying problem, hardware, and accuracy requirements.

3.5 Chapter Summary

In this chapter, we presented a new programming model where trade-offs between time and accuracy are exposed at the language level to the PetaBricks compiler. To the best of our knowledge, PetaBricks is the first programming language that incorporates a comprehensive solution for choices relating to algorithmic accuracy. We have outlined how the PetaBricks compiler automatically search the space of algorithms and parameters (optimal frontier) to construct an optimized algorithm for each accuracy level required. Using PetaBricks, writing programs for variable accuracy problems can be more effective, since the users can change their required accuracy metrics and accuracy levels, and PetaBricks can adapt to these changes easily. We demonstrated the accuracy/performance trade-offs by two examples, k-means clustering and preconditioning, and show how nontrivial algorithmic choice can change with different user requirements. In particular, we showed how k-means clustering can be solved without specifying the number of clusters k, and showed that the optimal k can be determined accurately with PetaBricks using relevant training data. To the best of our knowledge, our approach is the first to solve any general-purpose k-means clustering problem without domain-specific modeling and requiring the user to solve some optimization problems to obtain the number of clusters k. In the problem of preconditioning, we showed that the optimal choice of preconditioners can change with problem sizes, in addition to the system matrix. Autotuning with PetaBricks provides a systematic way to incorporate different kinds of preconditioners

even without prior information about the system matrix. This approach is especially useful when the user is not certain about which one improves convergence the most.

```
1  transform kmeans
2  accuracy_metric kmeansaccuracy
3  accuracy_variable k
4  from Points[n,2] // Array of n points (each column
5                   // stores x and y coordinates)
6  through Centroids[k,2]
7  to Assignments[n]
8  {
9    // Rule 1:
10   // One possible initial condition: Random
11   // set of points
12   to(Centroids.column(i) c) from(Points p) {
13     c=p.column(rand(0,n))
14   }
15
16   // Rule 2:
17   // Another initial condition: Centerplus initial
18   // centers (kmeans++)
19   to(Centroids c) from(Points p) {
20     CenterPlus(c, p);
21   }
22
23   // Rule 3:
24   // The kmeans iterative algorithm
25   to(Assignments a) from(Points p, Centroids c) {
26     for_enough {
27       int change;
28       AssignClusters(a, change, p, c, a);
29       if (change==0) return; // Reached fixed point
30       NewClusterLocations(c, p, a);
31     }
32   }
33 }
34
35 transform kmeansaccuracy
36 from Assignments[n], Points[n,2]
37 to Accuracy
38 {
39   Accuracy from(Assignments a, Points p){
40     return sqrt(2*n/SumClusterDistanceSquared(a,p));
41   }
42 }
```

Figure 3-1: Pseudocode for variable accuracy kmeans illustrating the new variable accuracy language extension.

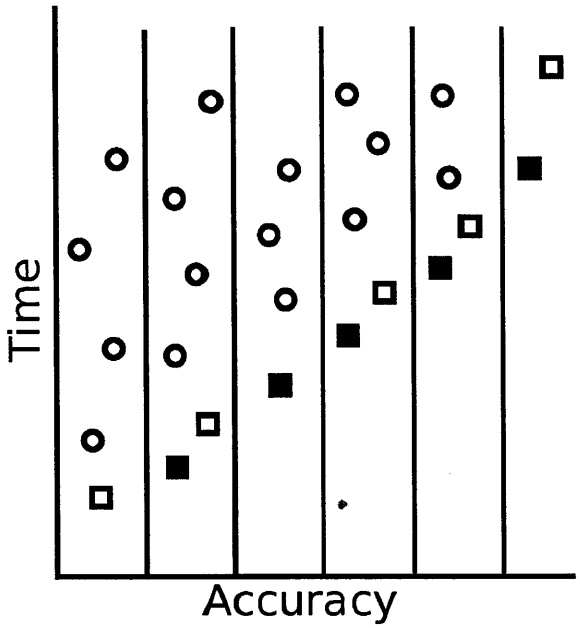

Figure 3-2: Possible algorithmic choices with optimal set designated by squares (both hollow and solid). The choices designated by solid squares are the ones remembered by the PetaBricks compiler, being the fastest algorithms better than each accuracy cutoff line.

KMEANS($X, Centers$)

1: **either**
2: Assign random initial cluster centers
3: Use `kmeans++` to set initial configuration
4: **end either**
5: Apply `kmeans` iterative algorithm, and stop when:
6: **either**
7: local optimum is reached, i.e. no further changes in cluster assignments
8: number of cluster changes $\leq 0.5n$
9: number of cluster changes $\leq 0.25n$
10: one iterative step is performed
11: **end either**

Figure 3-3: Pseudo code for k-means clustering

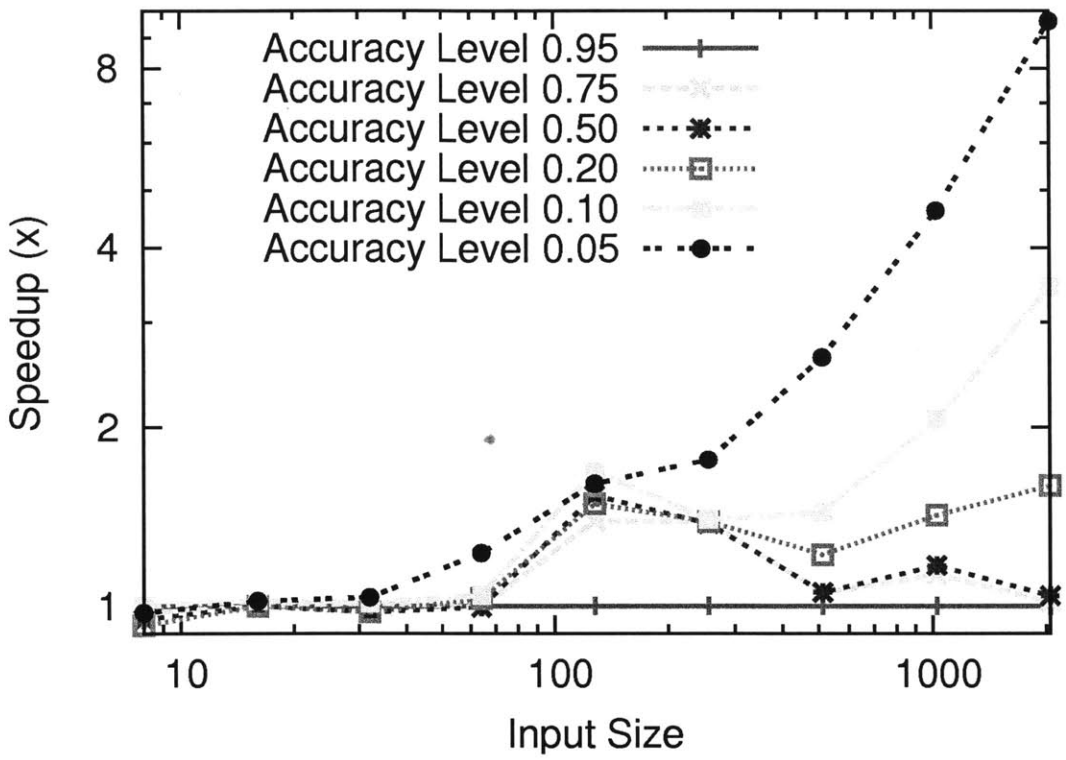

Figure 3-4: k-means clustering: Speedups for each accuracy level and input size, compared to the highest accuracy level for each benchmark. Run on an 8-way (2×4-core Xeon X5460) system.

`PRECONDITIONER`(A, x, b)

1: Iterate with `NumIterations` using one of the algorithms:
2: **either**
3: Preconditioned Conjugate Gradient (PCG) with `Jacobi Preconditioner`
4: PCG with `Polynomial Preconditioner`
5: Conjugate Gradient (CG) i.e. no preconditioning
6: **end either**

Figure 3-5: Pseudo code for Preconditioner

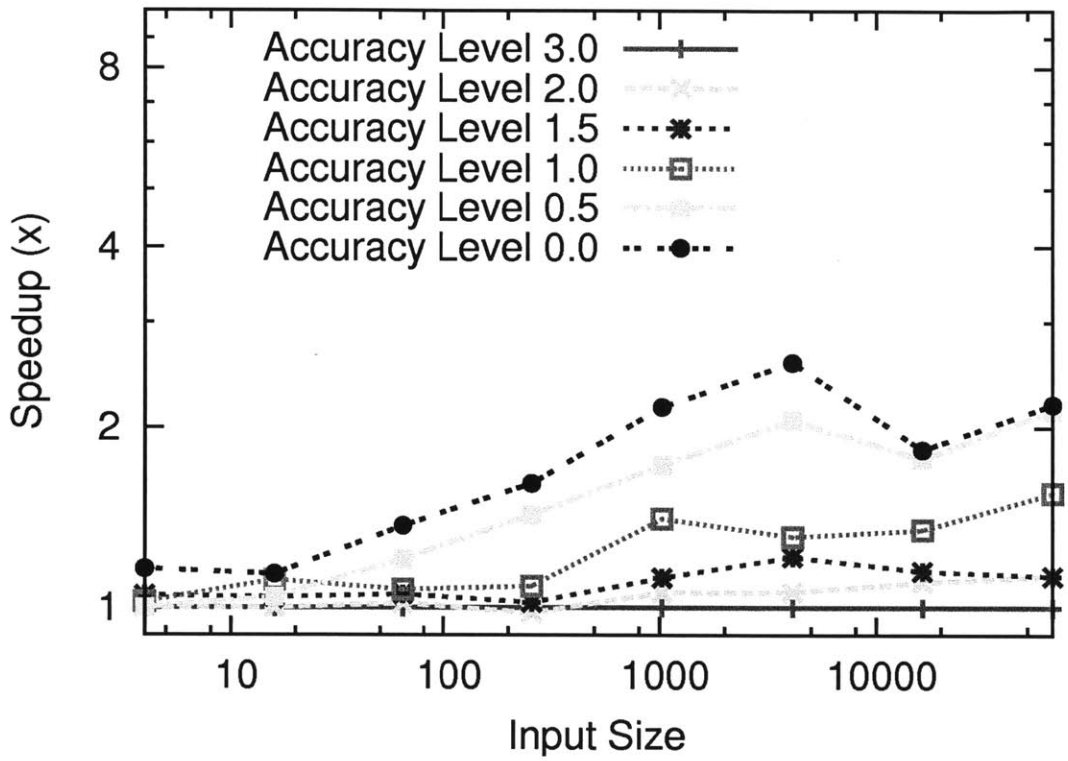

Figure 3-6: Preconditioning: Speedups for each accuracy level and input size, compared to the highest accuracy level for the Poisson operator A. Run on an 8-way (2×4-core Xeon X5460) system.

THIS PAGE INTENTIONALLY LEFT BLANK

Chapter 4

Analysis of Terrain Data

4.1 Introduction

In this chapter, we study a problem in large-scale terrain data analysis which motivates the use of a new high-level programming language Julia. Terrain analysis plays a key role in environmental modeling, land-use management and military operations. The process of analyzing and interpreting features within the area of terrain provides an understanding of the impact and limitations of the area of operations, which can be used as a base for planning and operational decisions.

The focus of terrain analysis by researchers are mainly on capturing local variation of terrain properties by climate changes and distribution of human activities. Evaluating and quantifying terrian properties which could change constantly is another major challenge in this field. Typical terrain analysis methods include spatial modeling, Monte-Carlo simulations and Digital Elevation Models [92]. Other advanced techniques include probabilistic modeling, learning algorithms for parameter tuning [83], roughness analysis using Geographic Information System(GIS) [80], multi-scale modeling and segmentation-based terrain classification [95, 81]. These techniques often require a thorough understanding and modeling of the underlying terrain and a comprehensive dataset for analysis.

Our purpose and approach are different: We are given a large set of Cartesian coordinates, taken from a vehicle moving across a couple of terrains. We would

like to capture and study interesting features without detailed information of the underlying terrains. In this chapter, we apply general filtering methods and Singular Value Decomposition (SVD) on downsampled datasets since the original datasets would not fit on the memory of a typical machine. We perform analysis in serial on terrain data and devise a systematic method to classify the terrain. Our methodology does not depend on what kind of terrain or area the data comes from, and does not require extensive modeling. In chapter 5, we apply the same techniques from this chapter using Julia, a newly developed programming language, and discuss the parallel implementation.

The remainder of this chapter proceeds as follows: First, we introduce our dataset and the pre-processing procedures in Section 4.2. In section 4.3, we demonstrate the use of Laplacian of Gaussian (LoG) filter to detect large-scale road bumps. In section 4.4, we analyze the surface roughness of tracks by studying the high-frequency noise captured by a Gaussian filter and Fast Fourier Transform (FFT). We demonstrate in section 4.5 how Singular Value Decomposition (SVD) can be applied in terrain analysis and how SVD can be used to distinguish different terrain profiles. We end this chapter by summarizing in section 4.6.

4.2 Pre-processing of Data

We are given a large set of raw terrain data taken from a moving vehicle across some terrains. The data set contains datapoints from two courses, each of which is divided into several segments. We describe here how we reformat and downsample the data such that it loads into a nice grid in Matlab. We provide a way to organize the large sets of raw data and visualize the tracks.

4.2.1 Downsampling and reformatting

The data provided to us came in a series of chunk files which contains a header and a list of points corresponding to the Cartesian coordinates. After some experimenting, we established that the points represent intertwining sets of scanlines as the vehicles

go along the tracks. The raw data is too large to fit into the memory of any regular machine, so we downsampled and reformatted the data as described below.

The header information from all the data files are removed such that the remaining data in the files load as a large matrix in Matlab using a cluster with sufficient memory. The distance between successive datapoints in the xy-plane is then computed as $\Delta s = \sqrt{(\Delta x)^2 + (\Delta y)^2}$. By setting the right threshold value, we can find the points which satisfy $\Delta s >$ threshold. These points correspond to the start of a new scanline. Using this algorithm and with some experimentation, we observed that in the raw datafiles, 940 points correspond to one full scan. We further downsampled this in a "94-wide" format (sampled every 10 points from the original datafiles).

4.2.2 Overview of the tracks

The resulting downsampled files contain Cartesian coordinates from two different tracks. After loading into a Matlab grid, each of the xyz coordinates is reshaped into a rectangular matrix of dimension $94 \times n$, where n varies with each datafile. Each column of the matrix contains the 94 downsampled data points measured by the same scanline. Each row of the matrix contains n data points, with each point taken from a different scanline as the vehicle moves along the track. Figure 4-1 shows a portion of the track plotted with the downsampled data. Each scanline in the figure is plotted with a different color, and corresponds to each column of the rectangular matrix.

Using the downsampled data, the two tracks were plotted on the xy-plane as in Figure 4-2. Track 1 consists of mostly straight roads with two big turns. The data for track 1 is divided into 5 segments. Track 2 consists of 11 segments with more turns and curves. One segment of the original raw datafiles for track 2 was corrupted: Many newline characters are missing and only partial coordinates (missing x) are present for some datapoints. As a result, only 10 segments are plotted in Figure 4-2. However, it is obvious that the missing segment connects the consecutive parts of the track and makes Track 2 a closed loop. Throughout the rest of this chapter, we will refer to these two tracks as Track 1 and 2.

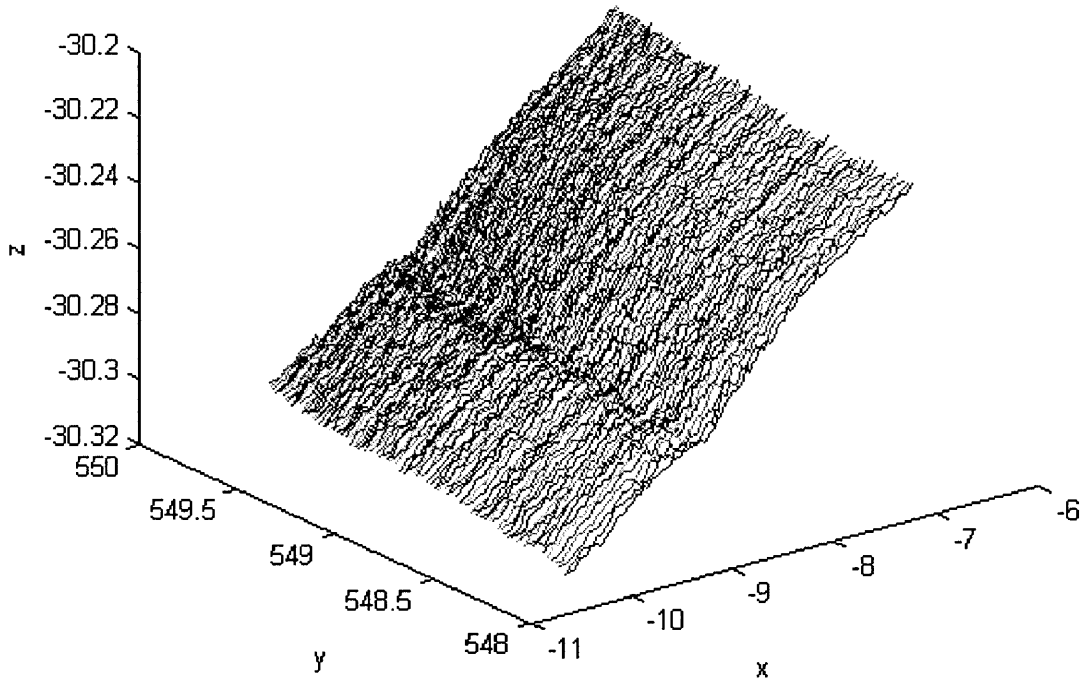

Figure 4-1: Terrain visualization with downsampled data

4.3 Road Bumps Detection

A sample terrain profile from a segment in Track 2 is plotted in Figure 4-3. The plot is generated using one row of the downsampled matrix as described in the previous section. In Figure 4-3, the horizonal axis is s, the horizontal distance traveled, calculated by $\Delta s = \sqrt{(\Delta x)^2 + (\Delta y)^2}$, where the Δx and Δy are the difference of adjacent x, y coordinates recorded as the vehicle travels. The height of the terrain (z-coordinate data) is plotted on the vertical axis to provide a visualization of the terrain profile. The data of this track appears to be fairly smooth and does not contain small-scale noises. However, a number of bumps are observed (for example, at approximately $s = 75, 175, 200$). We describe how to use filtering techniques with an appropriate window width to detect and remove the bumps.

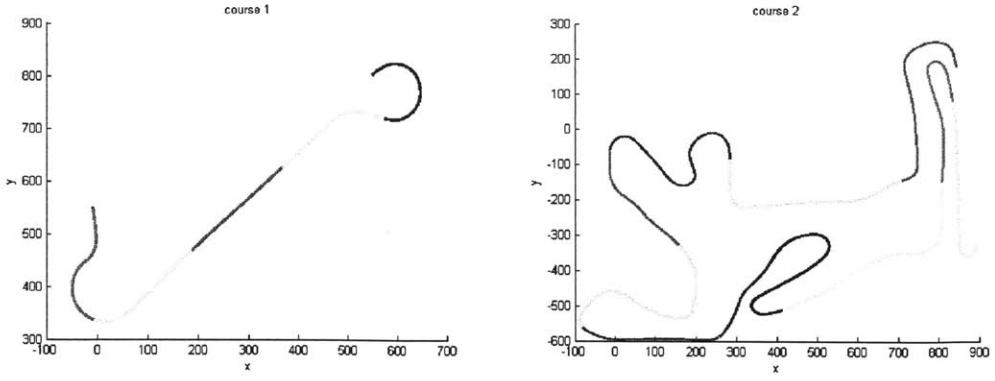

Figure 4-2: Track 1 (left) and Track 2 (right) plooted on the xy-plane

4.3.1 Laplacian of Gaussian filter

The Laplacian of Gaussian (LoG) filter is a common technique used for edge detection in image processing [79]. We apply the LoG filter to detect the road bumps observed in the data of Track 2. These road bumps will not be picked up by a regular filter which detects small-scale high-frequency noises. To identify these bumps, a 1D normalized LoG filter with the right half-window width k is passed to the data:

$$LoG(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-x^2/2\sigma^2} \left(\frac{x^2 - \sigma^2}{\sigma^4} \right), \tag{4.1}$$

where $\sigma = k/4$.

The LoG filter approximates the second derivative (Laplacian) of the data and smooths out Gaussian high-frequency noises. The road bumps can be located at the local minimum of the filtered data. For the terrain profile in Figure 4-3, a half-window width of $k = 288$ is used (We discuss in the next subsection 4.3.2 how this value is chosen). The bumps are identified as shown in Figure 4-4. After detecting the locations of bumps, they are removed from the data and the resulting terrain data can be interpolated using Matlab's command `spline` to produce a non-bumpy profile (Figure 4-5). The isolated bumps are calculated by subtracting the processed z-coordinates from the original data.

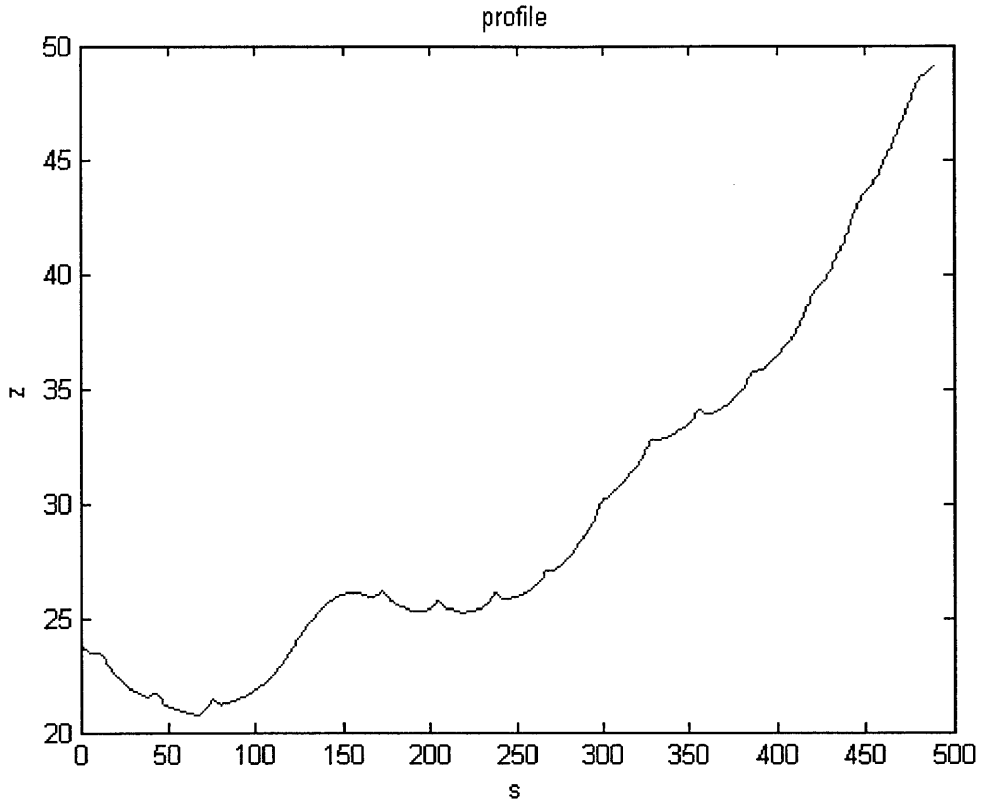

Figure 4-3: Terrain Profile with downsampled data (6200 data points are plotted in this figure)

4.3.2 Half-window width k

A main problem when applying the LoG filter is that the output depends on both the size of the road bumps and the size of the Gaussian kernel used for pre-smoothing. To capture all the bumps with sizes which are not known in advance, a LoG filter with the right half-window width k is required. If k is too small, the filter will pick up more "bumps" than the actual ones. If k is too large, some of the bumps will not be detected and the scale of the bumps removed may be larger than necessary. To capture the right scale, we apply the following algorithm:

1. Repeat the bump detection algorithm for values of k, starting from some pre-set $k_{\min}$ to $k_{\max}$. Assuming that the scale of bumps will not be excessively large, we set $k_{\max}$ to be 10% of the number of data points N.

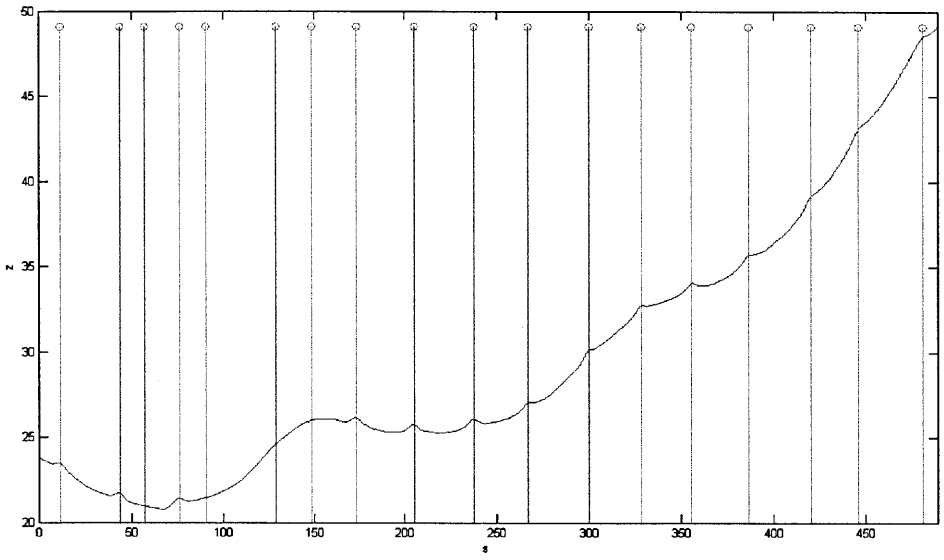

Figure 4-4: Vertical lines in red indicate the positions of bumps using the LoG filter with $k = 288$

2. For each run of k, record the number of bumps detected.

3. Record the variance of the peaks of the bumps, Var_p for each run.

4. Examine the number of bumps detected as k increases. Take the set of k with number of bumps that is close to a converged value. Among this set, take the k_{optimal} to be the k with the minimum Var_p.

The idea behind this algorithm is that for $k < k_{\text{optimal}}$, the false detection of bumps will make the number of bumps captured to be much larger. As k approaches k_{optimal}, the number of bumps will start to converge to the real value. This is incorporated in step (2) of the algorithim to avoid over-detection. When $k > k_{\text{optimal}}$, some bumps will not be detected. And as the bumps detected are removed, data with a larger window width will be removed since the window width k used is too large. After interpolation of the data with bump removal, the isolated bumps will have a larger variation in value than the optimal set obtained with k_{optimal}. Recording and examining the variance Var_p in Step (3) of our algorithm captures this property.

When this algorithm was applied to the segment in Figure 4-3, $k_{\text{optimal}} = 288$ was

chosen. We observe that the bumps are detected accurately in Figure 4-4 and 4-5. Two extra "bumps" were accidentally detected, but it does not appear to affect much the data after bump removal. This algorithm was also applied to other segments of the same track. Different values of k_{optimal} were obtained and all of the output show accurate detection of road bumps. The plots look similar to the ones shown in Figure 4-3 to 4-5, so we omit them here.

4.4 Noise Analysis

Another task we identified is to study the surface roughness by analyzing the noise. Noisy terrain data is passed to a Gaussian filter with a predetermined filter width. The noise is then calculated by the difference of the original and the smoothed data. The same filter is passed to two sets of data. The first data set (track 1) appears to have small-scale high frequency noise, and the second set (track 2) is the same as the dataset from the previous section which contains fairly smooth data but large-scale bumps are present.

Track 1 with noisy data and the smoothed profile is plotted as in Figure 4-6. The Q-Q plot of the noise versus Standard Normal distribution, and the historgram of the noise is plotted in Figure 4-7. The same analysis was performed on Track 2. Raw data and the filtered profile is plotted in Figure 4-8. There is little high-frequency noise in dataset 2 compared with dataset 1. The large-scaled bumps discussed in Section 3 are smoothed a little bit but still apparent in the filtered data. The Q-Q plot of the noise versus Standard Normal distribution, and the historgram of the noise for Track 2 is plotted in Figure 4-9.

The noise was obtained by subtracting the filtered data from the original data. A number of statistics of the noise is computed as in Table 4.1.

The noise of this segment of track 1 appears to be symmetric, close to but slightly different from Gaussian. Noise of track 2 has an unsymmetric, skewed distribution. The distribution observed is significantly different from a normal distribution.

	Mean	Variance	Skewness	Kurtosis
Track 1	0.0019	2.5226×10^{-4}	-0.0284	2.6495
Track 2	0.0021	7.0484×10^{-4}	-1.7673	8.3357

Table 4.1: Statistics of noise for both tracks

Instead of using a Guassian filter, another approach is to take the Fast Fourier Transform (FFT) of the data, and remove the high-frequency components. One disadvantage of using the FFT is that the cutoff of high-frequency contributions is sharp. The resulting output in time domain (after an inverse FFT) may be complex-valued, and we can only look at the magnitude of the filtered output. We applied FFT to both sets of data and obtained similar results as using Gaussian filters.

Since Gaussian filtering is a linear filtering operation, which filters out higher order noise from the frequency spectrum and does not fundamentally change the distribution, the difference of noise for both tracks is due to the data source. We experimented with different widths of the filter and observed similar results on the two sets of data. As another check, we examined the Fourier Transform of the data before and after filtering. We observed that the frequency spectrums only differ significantly on the higher frequency components, so the filtering operations function as we intended. Therefore we conjecture that the difference is due to the different surface roughness of the tracks.

4.5 Singular Value Decomposition (SVD)

In this section, we show how to apply Singular Value Decomposition (SVD) on noise analysis of our terrain data, and propose a measure of surface roughness with the use of SVD. We first give a brief overview of SVD.

For any $m \times n$ matrix A, the Singular Value Decomposition (SVD) of A is

$$A = U\Sigma V^*,$$

where U is an $m \times m$ unitary matrix, V^* is the Hermitian conjugate of V, which

is an $n \times n$ unitary matrix, and Σ is an $m \times n$ diagonal matrix with non-negative decreasing real entries σ_i.

The columns u_i of U are called the left singular vectors. The columns v_i of V are called the right singular vectors. The diagonal values σ_i of Σ are called the singular values. One way to interpret SVD is as follows. Given an $m \times n$ real matrix A, think of it as a linear operator from $\mathbb{R}^n$ into $\mathbb{R}^m$. Consider the SVD of A, as U and V are orthogonal, the columns u_i and v_i form an orthonormal basis of $\mathbb{R}^m$ and $\mathbb{R}^n$ respectively. We have $Av_i = \sigma_i u_i$, i.e. the image in $\mathbb{R}^m$ of a right singular vector v_i (in $\mathbb{R}^n$) is equal to σ_i times the left singular vector u_i. It follows that any vector $x = \sum_{i=1}^{n} \alpha_i v_i$ is mapped to $y = Ax = \sum_{i=1}^{n} \sigma_i \alpha_i u_i$, where α_i are arbitrary constants. Each singular vector is a component of the transformation by A, and the associated singular value tells us how dominant that particular component is.

4.5.1 Noise Filtering by Low-rank Matrix Approximation

The singular values σ_i are in decreasing order such that σ_1 is the largest. Consider the SVD of a matrix A, in practice it is common to observe several dominant singular values $\sigma_1, \ldots, \sigma_k$ for some k. These singular values correspond to k principal components and this idea can be used to approximate the original matrix A.

The best rank-k approximation of A is given by

$$A_k = \sum_{i=1}^{k} \sigma_i u_i v_i^*$$

One common application of low-rank matrix approximation is in image compression. Using SVD, only a small number k of singular values and singular vectors need to be stored. Using this idea, we can apply SVD to filter high-frequency noises in our terrain data and produce results similar to the ones obtained from usual filtering techqniues in Section 4.4.

The SVD of the downsampled $94 \times N$ matrices of z-coordinates of each segment of both tracks was computed. We observed that the largest singular value σ_1 is significantly dominant. The ratio $\frac{\sigma_1}{\sum_i \sigma_i}$ was calculated and found to be > 0.99 for

all of our data. When high-frequency noise is present in the data, they make up a low-rank noise matrix and contribute to the smaller singular values. Thus, one simple way to filter out a signficant portion of the random noise is to use a rank-1 approximation of A by SVD: $A_1 = \sigma_1 u_1 v_1^*$.

One segment of noisy terrain data and the filtered data are plotted in Figure 4-10. We see that the SVD apprixmation can filter out some high-frequency noise and the output data is cleaner. The noise filtered out is also close to Gaussian and gives similar statistics as in Section 4.4. Filtering by SVD appears to filter less noise than the usual Gaussian filter. However, the advantage of this approach is that it is easy to determine the number of singular values k to use by looking at the ratio $\frac{\sum_{i=1}^{k}\sigma_i}{\sum_{i=1}^{n}\sigma_i}$. On the other hand, effort is necessary in finding the right window width and cut-off for a Gaussian filter.

4.5.2 Surface Roughness Classification using SVD

We have seen in previous sections that high-frequency noise present in terrain data can account for how rough the surface is. To calculate a useful measure of roughness, we take an approximation of $\frac{dz}{ds}$, and apply SVD on it.

We define $df = \frac{\Delta z}{\Delta s}$, where $\Delta s = \sqrt{(\Delta x)^2 + (\Delta y)^2}$. The quantities Δz and Δs are calculated by the difference between two consecutive data points along the direction at which the vehicle travels. The quantity df of a noisy segment (Track 1) and a smooth segment (Track 2) are plotted in Figure 4-11. df_1 appears to be random noise for the noisy data, while df_2 of a smooth track is a more continuous curve with some noise in it.

Since df is an approximation of how z changes with distance traveled, a rougher surface should give a more rapidly changing df. If we take SVD on this df matrix, we expect the distribution of singular values to be more even and there will be no significantly dominant singular values. The first 5 singular values of df_1 and df_2 are listed in Table 4.2. It is clear that there is one principal component associated for df_2 (which corresponds to the smooth track), but not as much for df_1 (noisy track).

	σ_1	σ_2	σ_3	σ_4	σ_5
df_1	46.3414	35.2553	32.8501	23.7086	20.3377
df_2	67.2090	6.6638	5.5797	5.0638	4.6563

Table 4.2: First 5 singular values of df_1 (noisy) and df_2 (smooth)

	Data 1 (smoothed)	Data 2 (some noise added)	Data 3 (noisy data)
μ	0.8995	0.4276	0.2334

Table 4.3: μ of the same segment with different level of noise. μ gets closer to 1 as the data gets smoother.

Based on this observation, we propose the following measure of roughness, μ:

$$\mu = \frac{\sigma_1^2}{\sum_i \sigma_i^2},$$

where σ_i are the singular values of the matrix df.

μ is a measure of how dominant the largest singular value is, and can take any value in the range $0 < \mu < 1$. If the track is rough, the high-frequency noise contributes to a df matrix without significantly dominant principal components, so σ_1 contributes to a smaller portion of the sum of the singular values. Thus, a smaller value of μ implies a higher surface roughness of the track. To show this, we took the df of three sets of data: (1) a smoothed segment of Track 1 after noise filtering, (2) the same segment after filtering and adding only 1/5 of the noise, and (3) the original noisy segment. The values of μ obtained are in Table 4.3 and we see that μ decreases with more noise in the data.

4.6 Chapter Summary

Analyzing a large set of raw terrain data poses different challenges in both the methodology and quantification of terrain properties. In this chapter, we performed various analysis to study the terrain profiles and proposed a useful measure μ based on the Singular Value Decomposition (SVD) to quantify terrain surface roughness. We have

shown how classical filtering techniques and SVD can be applied to study road bumps and noise in various scales. We also proposed a systematic way to classify surface roughness. Our methdology does not require extensive knowledge and modeling of the terrian and vehicle movement. The algorithms suggested in this chapter are generic and not domain-specfic, so they can be applied to give reproducible results on different sets of terrain data. Future directions can include studying the elevation or roll angles, the forces on the vehicle as it travels and the rate of change of these quantities. Another potential direction for further terrain data analysis is to combine the idea behind our SVD-based algorithm with more information of the terrain and vehicle, such as the speed of the vehicle moving along the track and how fast the angles measured change.

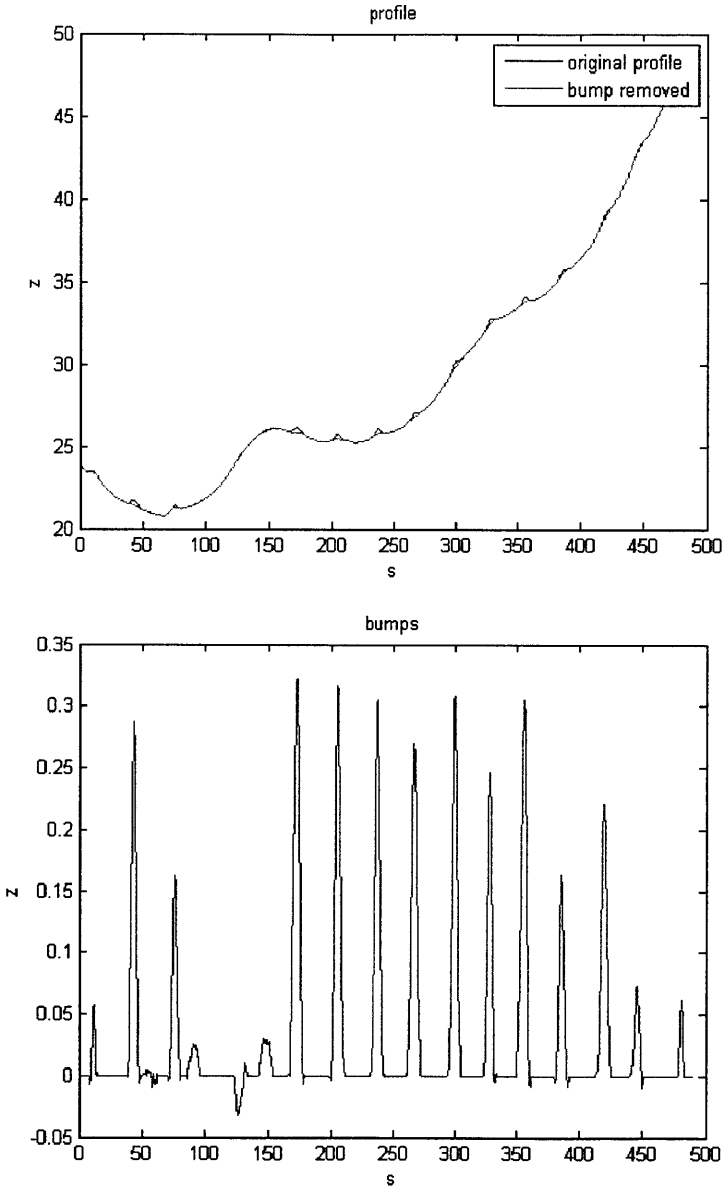

Figure 4-5: Top: Terrain Profile with bumps removed. Bottom: Bumps isolated

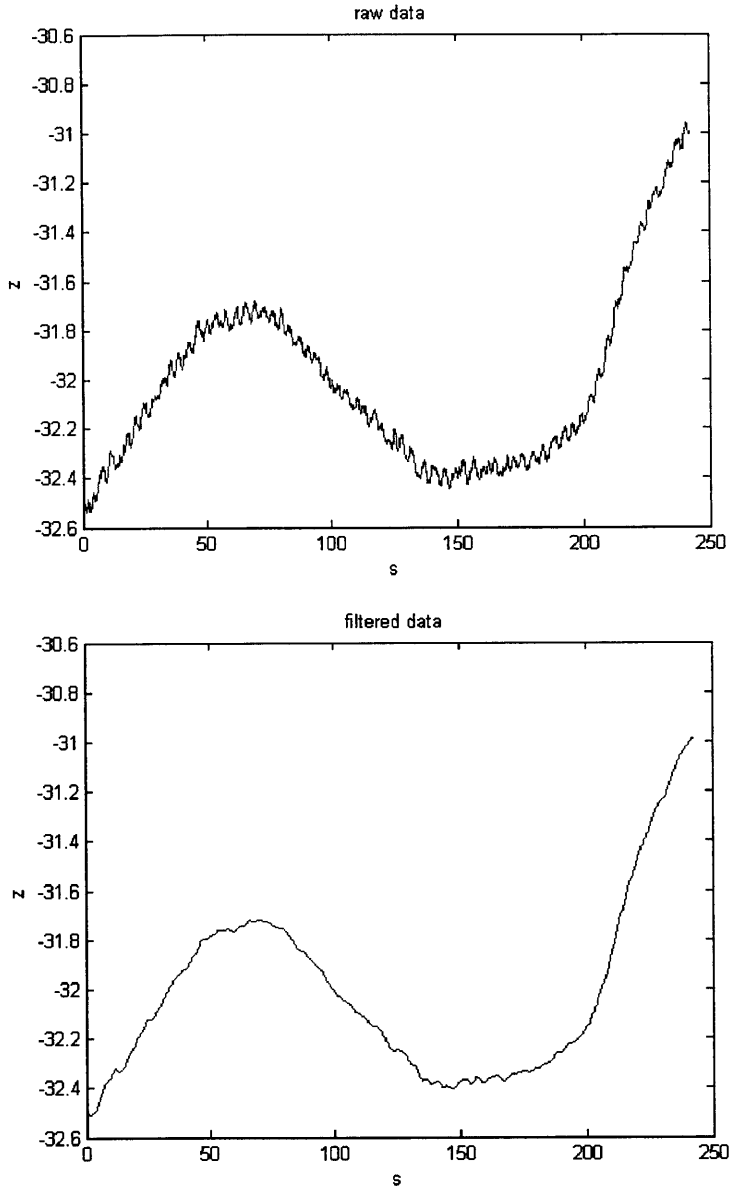

Figure 4-6: Top: Noisy terrain profile. Bottom: Smoothed data

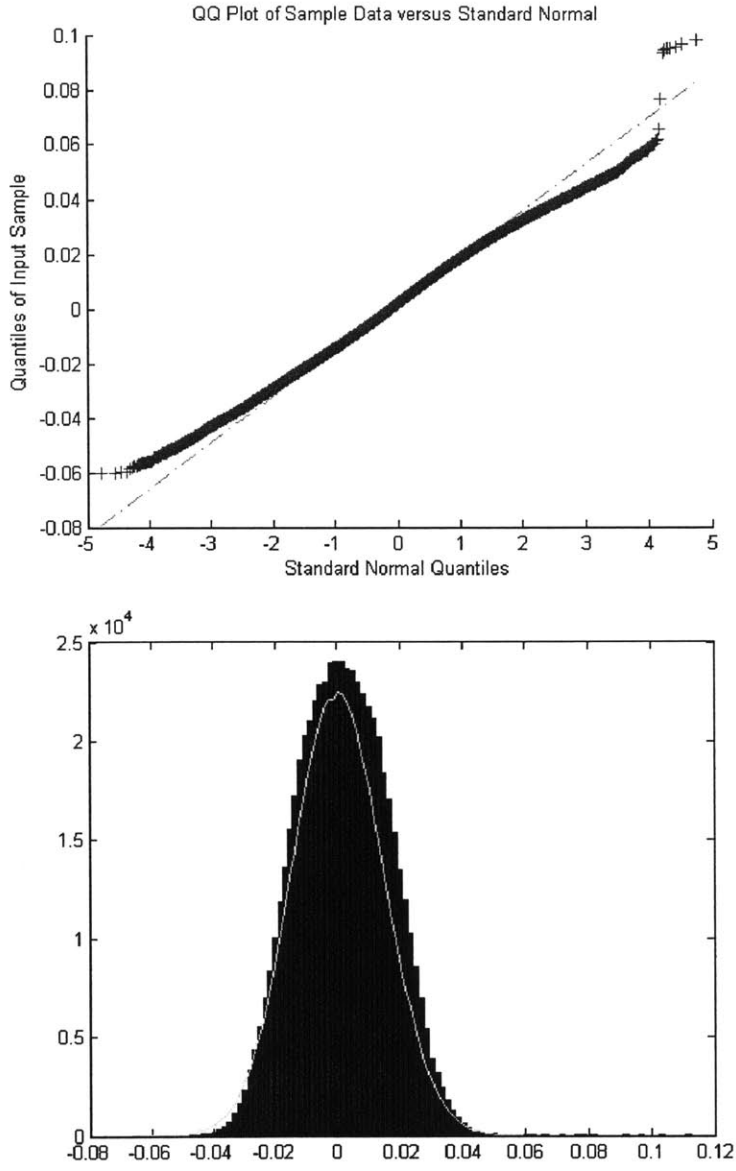

Figure 4-7: Top: Q-Q plot of noise, compared with Standard Normal. Bottom: Histogram of noise, and plot of Normal Distribution with the same mean and variance

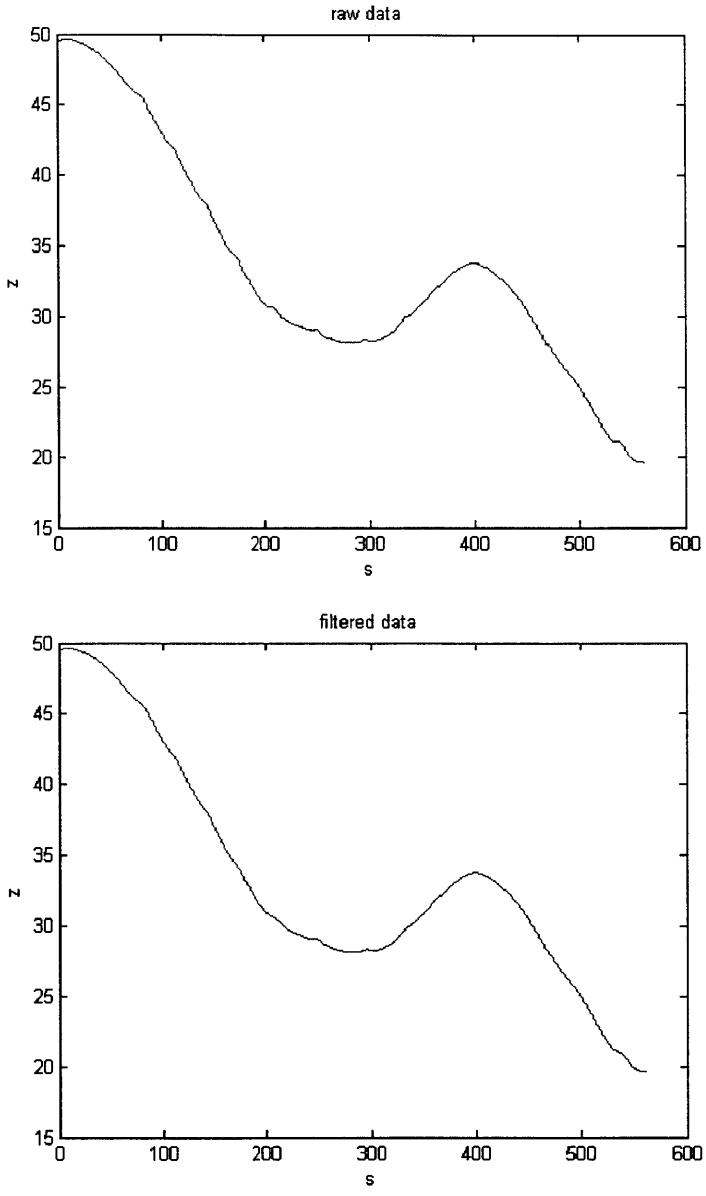

Figure 4-8: Top: Raw terrain profile (Track 2). Bottom: Filtered data

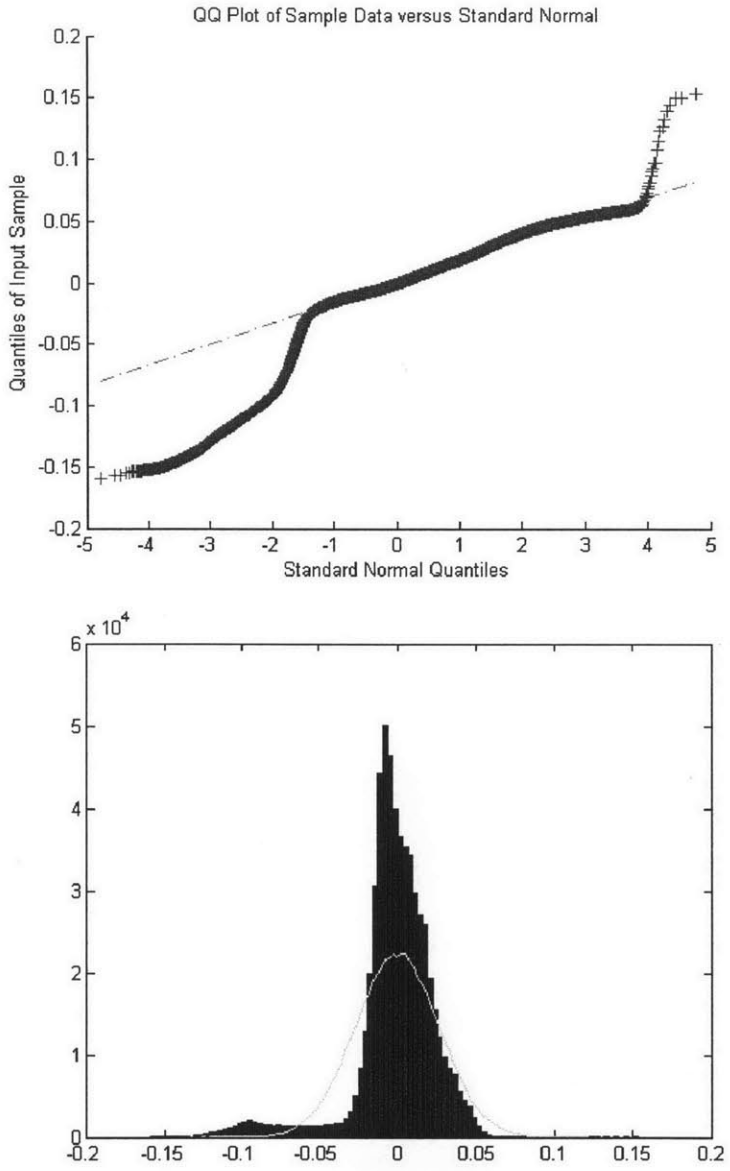

Figure 4-9: Top: Q-Q plot of noise, compared with Standard Normal. Bottom: Histogram of noise, and plot of Normal Distribution with the same mean and variance

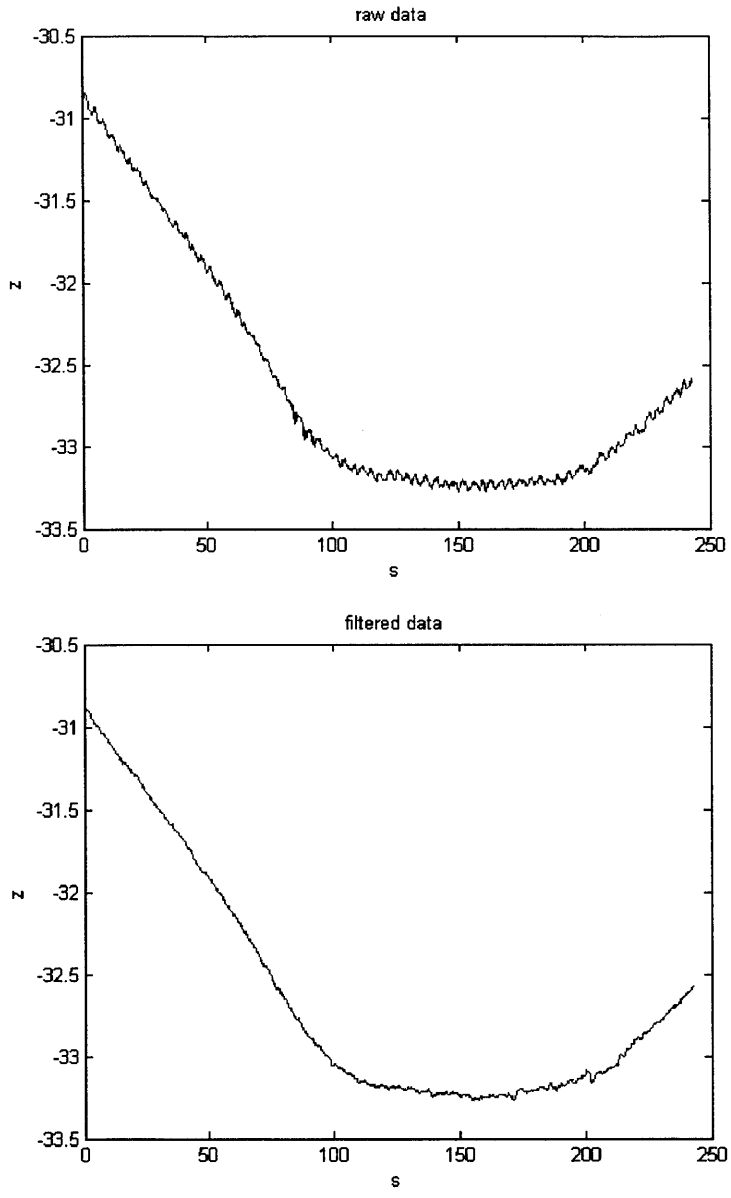

Figure 4-10: Top: Noisy terrain profile. Bottom: Smoothed data by SVD

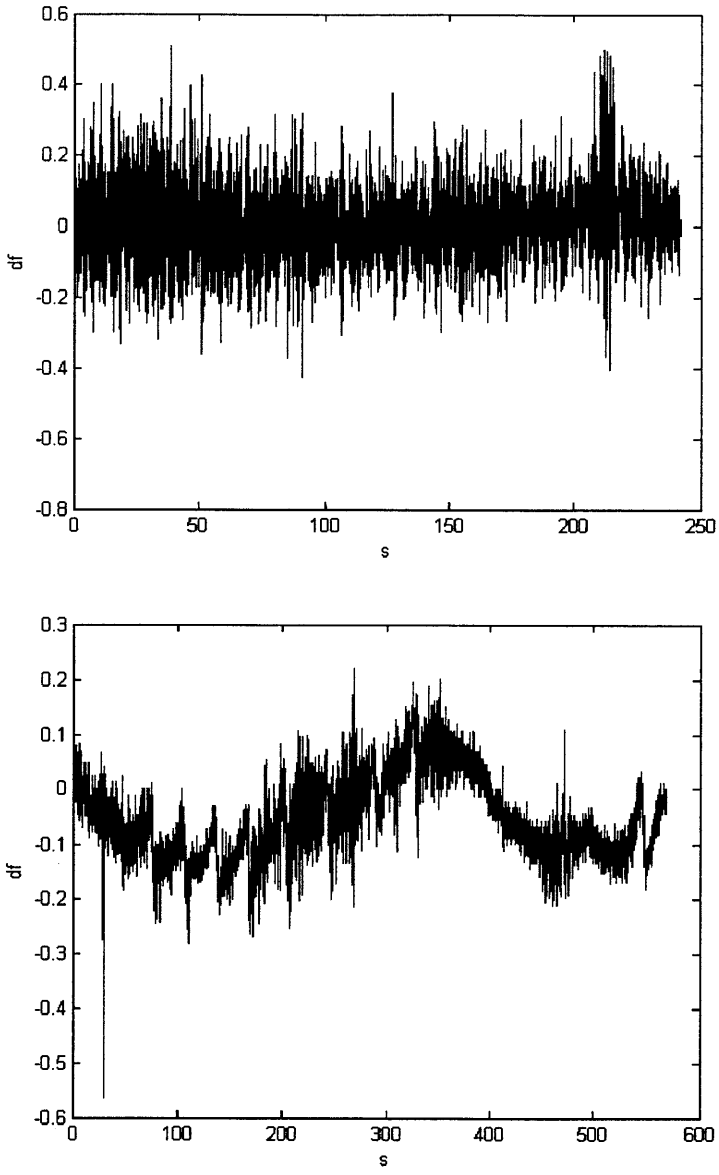

Figure 4-11: Top: df_1 of a noisy segment of terrain. Bottom: df_2 of a smooth segment

Chapter 5

Large-scale Data Processing with Julia

5.1 Introduction

In the previous chapter, we showed how we could apply general filtering methods and Singular Value Decomposition (SVD) on downsampled datasets of Cartesian coordinates to capture and study features of the terrains. Since the original datasets would not fit on the memory of a typical machine, parallel data distribution and a parallel programming language are necessary to process the original large-scale datasets and perform mathematical analysis. In this chapter, we introduce Julia, a newly developed programming language, and discuss how we can perform high-performance computing with our terrain data set with Julia.

The Julia project is based in MIT, involving computer scientists, applied mathematicians and researchers from a number of different places. As scientific computing advances in different domains, a large group of experts such as the numerical linear algebra community have moved to dynamic programming languages for their applications. However, dynamic programming languages can be slower and users of them have to pay for the performance trade-off for other reasons such as ease of use and efficiency of adding new codes for their applications. One common example is Matlab, which is very common in the numerical analysis community but does not give

the highest performance as compared with other low-level languages such as C. One of the goal of the Julia project is to fill the gap: Juia is designed to be a dynamic language which is easy to use for scientific computing, achieves performance comparable to traditional compiled languages, and offers flexible parallelism without extensive effort. Another goal of Julia is to enable cloud computing from a provider of choice, and users can upload data from anywhere and perform big computations easily.

The syntax of Julia is similar to Matlab, so users of Matlab should feel immediately comfortable with Julia. In addition, Julia keeps Matlab's ease and expressiveness for high-level scientific computing, but offers more programming possibilities outside of the scope of numerical computation. To achieve this, Julia borrows much from the C programming language, and is strongly influenced by the lineage of dynamic languages: Lisp, Perl, Python, Lua and Ruby [12].

The remainder of this chapter proceeds as follows: First, we introduce the syntax of some basic commands in Julia and show an example code in Section 5.2. In section 5.3, we pick one algorithm from Chapter 4 to implement in Julia, and show that the serial implementation gives results that agree with Matlab . In section 5.4, we describe in detail the implementation of the function for Singular Value Decomposition (SVD) in the Julia library, and discuss potential algorithms to expose parallelism. Our proposed implementation of parallel SVD suggests asynchronous work in bidiagonalization and trailing matrix update, which has not been explored in standard numerical linear algebra packages. We end this chapter by summarizing and discussing future work in section 5.5.

5.2 Julia

The syntax of Julia is similar to Matlab. We give a brief tutorial in this section and present an example code. More information can be found in [12].

5.2.1 Brief Tutorial

The basic commands for arithmetic and functions of numbers, vectors and matrices in Julia are very similar to Matlab. Below is a demo of basic commands. Note that the comments after the % signs are added as explanations in this writeup and were not entered to the command line to generate the outputs.

```
               _
   _       _ _(_)_     |
  (_)     | (_) (_)    |  pre-release version
   _ _   _| |_  __ _   |
  | | | | | | |/ _` |  |
  | | |_| | | | (_| |  |  (C)2009-2011 contributors
 _/ |\__'_|_|_|\__'_|  |
|__/                   |

julia> 3*4
12

julia> x=ans;

julia> x
12

julia> (1+2im)*(1-4im)  % in Julia, im is the complex number i
9 - 2im

julia> x=[2;3;4]        % Define a vector
[2,3,4]

julia> x[2]             % 2nd element of x
3

julia> x[2:end]         % 2nd to last elements of x
[3,4]

julia> A=rand(2,2)      % Uniform distributed 2x2 matrix
0.3378690370870434 0.1727834085356039
0.9429980507798121 0.4499985391348107

julia> A[2,2]
0.4499985391348107
```

```
julia> A[2,2]=0.5        % Change the (2,2) entry of A to 0.5
0.3378690370870434 0.1727834085356039
0.9429980507798121 0.5

julia> A[1,:]            % First row of A
0.3378690370870434 0.1727834085356039

julia> b=randn(2,1)      % Gaussian distributed 2x1 matrix
-0.2371815329099335
1.0608188891212531

julia> A\b               % Solution x to the system Ax=b
-50.312930659199246
97.0116288595319816

julia> exp(A)            % Exponential of each element of A
1.4019568870243506 1.1886086347173097
2.5676678890742402 1.6487212707001282

julia> inv(A)            % Inverse of A
83.3319293659423437 -28.7967495913914107
-157.1636939196092442 56.3105574669529076

julia> ans*A             % Verify the answer is indeed the inverse
1.0000000000000071 0.0
-7.1054273576010019e-15 1.0
```

Despite the similarity in syntax with Matlab, there are additional language supports in Julia that make expressions much cleaner and more convenient:

- Julia allows variables to be immediately preceded by a numeric literal, implying multiplication.

  ```
  julia> x=4
  4

  julia> 4(x-1)^2 - 3(x-1) + 1
  28
  ```

- There are two Julia constructs for a single compound expression to evaluate several subexpressions.

```
julia> z = begin
       x = 3
       y = 5
       x * y
       end
15

julia> z = (x = 2; y = 4; x/y)
0.5
```

- A `map` function can be used to apply a function to each element of an array or matrix. Here is an example for computing $\sin(x)$ for each entry of an matrix A:

```
julia> A = [0 pi()/2; pi() 3pi()/2; 2pi() pi()/6]
0.0 1.5707963267948966
3.1415926535897931 4.7123889803846897
6.2831853071795862 0.5235987755982988

julia> map(sin,A)
0.0 1.0
1.2246467991473532e-16 -1.0
-2.4492935982947064e-16 0.4999999999999999
```

- Julia introduces a new programming construct called a *multidimensional array comprehension*. Each dummy variable used in the array comprehension corresponds to a dimension of the output array; if the comprehension expression value itself has non-zero dimension then the total dimension of the output is the number of dummy variables plus the dimension of the value. Many common vector, matrix and tensor operations can be expressed concisely and clearly using array comprehensions.

```
julia> A = [1 2 3; 4 5 6; 7 8 9]
1 2 3
4 5 6
7 8 9

julia> row_sums = [sum(A[i,:]) | i=1:size(A,1)]
{6,15,24}
```

```
julia> column_sums = [sum(A[:,j]) | j=1:size(A,2)]
{12,15,18}

julia> diagonal = [ A[i,i] | i=1:min(size(A)) ]
{1,5,9}
```

5.2.2 Example Code

```
1  function randmatrixtest()
2      t=200000
3      n=20
4
5      v =
6      @pfor (+) i=1:t begin
7          a = 2*float64(rand(n,n)<.5) - 1
8          (q,r) = qr(randn(n,n))
9          a = q*a
10         (a[1,1]*a[2,2])^2 - 1
11     end
12     v/t
13 end
```

Figure 5-1: Julia example code

Figure 5-1 shows a working example Julia code. The code computes t independent trials of random matrix experiments, where t is a large number (set to 200000 here). In each trial, a $n \times n$ random matrix A with entries ± 1 with equal probability is generated (line 7). This random matrix A is then rotated by a random orthogonal matrix Q (line 8-9). Finally, the value $[A(1,1) \cdot A(2,2)]^2 - 1$ is stored for this trial, and the mean of this number is calculated for a total number of t trials. There are several things to note in the syntax in this example. First, the value of each trial returned is added to the variable `v` using the Julia compound expression (lines 5 to 11). The parallel for-loop from lines 6 to 11 is evaluated as a series of subexpressions and the value return by the for-loop is stored to `v` (line 5). On line 6, the macro `@pfor` is defined to run the for-loop in parallel, and the `(+)` following `@pfor` indicates the addition of the value returned to `v`. On line 10, even though there is no `return` keyword used, since

it is the last expression within the for-loop, the value `(a[1,1]*a[2,2])^2 - 1` is returned and added to `v`. Similarly, since the expression `v/t` is the last expression in the function `randmatrixtest`, by default this value is returned by our Julia function.

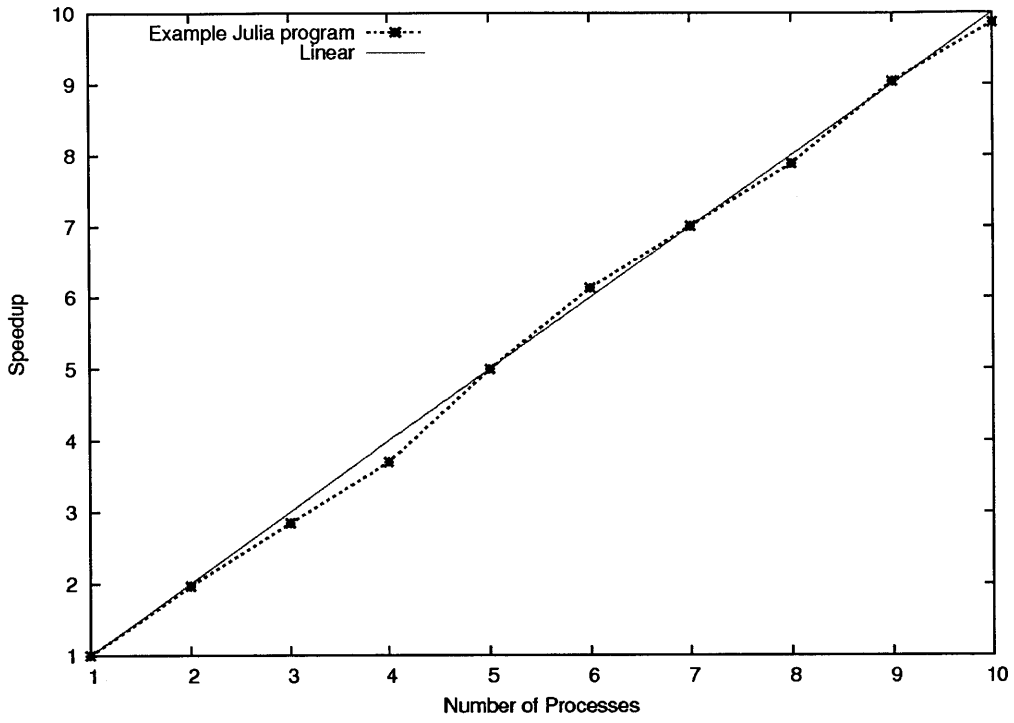

Figure 5-2: Parallel scalability for Julia example program: Speedup as more processes are added.

Figure 5-2 shows the timing results when the code is run using 1 to 10 processes. Speedup is calculated by $S_p = \frac{T_1}{T_p}$, where p is the number of processes and T_p is the execution time of the algorithm with p processes. The code is run on a cluster with 20 compute nodes with dual hex-core Xeons. Each node has 24 GB of memory shared between the 12 cores. We obtained linear speedup as expected because the example code is embarassingly parallel.

5.3 Terrain Analysis with Julia

Before applying the terrain analysis tools we developed in Chapter 4 on our full datasets, we experimented with the downsampled datasets using Julia. We implement in serial with Julia the algorithm in Section 4.5.2, which computes the measure μ to quantify surface roughness using SVD. Recall the algorithm as follows:

(1) Calculate Δz and $\Delta s = \sqrt{(\Delta x)^2 + (\Delta y)^2}$, where the quantities Δx, Δy and Δz are calculated as the difference between two consecutive data points in each row.
(2) Form the matrix df, where $(df)_{ij} = \frac{(\Delta z)_{ij}}{(\Delta s)_{ij}}$.
(3) Compute the singular values σ_i of the matrix df.
(4) Calculate the measure of roughness μ.

$$\mu = \frac{\sigma_1^2}{\sum_i \sigma_i^2},$$

The Julia code for this algorithm in serial is included in Appendix C. Running the codes in Julia for several segments in Track 1 and Track 2 of our data gives the results shown in Table 5.1.

	Track 1	Track 2
μ	0.1282	0.9009
	0.2076	0.9572
	0.2334	0.9170
	0.1857	0.9406
	0.1177	0.8521

Table 5.1: μ of several segments of the two tracks, computed using Julia.

We see that μ computed from Track 1 has an average value of 0.1745, while the mean of μ obtained from Track 2 is 0.9136. This agrees with our implementation with Matlab, and also agrees with our proposition that data from a rougher road track (Track 1) gives a value of μ closer to 0, and μ is closer to 1 when the data is smoother (Track 2).

5.4 Parallel Implementation

In section 5.2.2, we showed an example of how independent iterations can be implemented in parallel using parallel for-loops easily in Julia. To implement the algorithm in section 5.3 for measuring terrain surface roughness on our full dataset, the bottleneck is the step when SVD is applied to the $m \times n$ rectangular matrix A, where m and n can be large. To run SVD on a large input matrix which would not fit on the memory of a single machine, it is necessary to distribute the matrix across several processors, and apply a parallel SVD on the distributed data. In this section, we give first the blocked SVD Algorithm implemented in Julia and discuss potential places for further parallelism.

5.4.1 SVD Algorithm and Blocked Bidiagonalization

The singular value decomposition algorithm implemented in Julia consists of two major components: Reducing the matrix A to bidiagonal form by orthogonal transformation (for example by Householder reflections), and applying an iterative QR algorithm such that the bidiagonal matrix converges to diagonal form, with the diagonal entries being the singular values. Here we focus on the discussion of the first component, bidiagonalization, because it is usually the more time-consuming step in the algorithm. The idea of the second component QR iteration was introduced in section 2.3.2, and was implemented in Julia with LAPACK routine calls to ensure good convergence and precision, especially for small singular values.

The blocked bidiagonalization algorithm we implement is based on the ones used in [14, 20, 37]. The main idea is to aggregate Householder transformations and to apply them to trailing submatrices in a blocked fashion that is rich in matrix-matrix multiply, thus allowing updates to be performed in parallel and achieving speedups from optimized BLAS operations.

For an $m \times n$ matrix A, a series of Householder reflections is applied to zero out the entries under the diagonal and the entries to the right of the first superdiagonal (or subdiagonal). This reduces A to the bidiagonal form B such that $QAP = B$,

where Q and P are the matrices storing the Householder reflectors, and B is upper bidiagonal if $m \geq n$ or lower bidiagonal otherwise. In the following, assume $m \geq n$. $A^{(k)}$ denotes A after the $(k-1)$-th update, and $A_{i:j,k:l}$ denotes the submatrix of A consisting of rows i through j and columns k through l.

For a sequential unblocked algorithm, after the k-th update, the first k columns of A has zeros below the diagonal and the first k rows of A has all zeros to the right of the superdiagonal. Rewriting the Householder transformation in terms of rank-one updates, we have

$$\begin{aligned} A^{(k+1)} &= Q^{(k)} A^{(k)} P^{(k)} = (I - u_k u_k^T) A^{(k)} (I - v_k v_k^T) \\ &= A^{(k)} - u_k u_k^T A^{(k)} - A^{(k)} v_k v_k^T + u_k u_k^T A^{(k)} v_k v_k^T \\ &= A^{(k)} - u_k y_k^T - (A^{(k)} v_k - u_k y_k^T v_k) v_k^T , \end{aligned}$$

where u_k and v_k are the left and right Householder reflectors for the k-th step, $y_k = A^{(k)^T} u_k$. Let $x_k = A^{(k)} v_k - u_k y_k^T v_k$, we have

$$A^{(k+1)} = A^{(k)} - u_k y_k^T - x_k v_k^T \tag{5.1}$$

Using induction, we can show that

$$A^{(k+1)} = A^{(1)} - U_k Y_k^T - X_k V_k^T , \tag{5.2}$$

where $U_k = (u_1, \cdots, u_k)$, $V_k = (v_1, \cdots, v_k)$, $X_k = (x_1, \cdots, x_k)$ and $Y_k = (y_1, \cdots, y_k)$. Using Equation (5.2), we get

$$\begin{aligned} y_k &= A^{(k)^T} u_k = (A^{(1)} - U_{k-1} Y_{k-1}^T - X_{k-1} V_{k-1}^T)^T u_k \\ &= (A^{(1)^T} - Y_{k-1} U_{k-1}^T - V_{k-1} X_{k-1}^T) u_k \qquad (5.3) \\ x_k &= A^{(k)} v_k - u_k y_k^T v_k = (A^{(1)} - U_{k-1} Y_{k-1}^T - X_{k-1} V_{k-1}^T) v_k - u_k y_k^T v_k \\ &= (A^{(1)} - X_{k-1} V_{k-1}^T - (U_{k-1}, u_k) \cdot (Y_{k-1}, y_k)^T) v_k \\ &= (A^{(1)} - X_{k-1} V_{k-1}^T - U_k Y_k^T) v_k \qquad (5.4) \end{aligned}$$

The blocked bidiagonalization algorithm follows naturally. For simplicty we assume A to be a square $n \times n$ matrix and that A is divided into square blocks of size $n_b \times n_b$. In step k of the algorithm, the k-th block $A_{k_1:k_2,k_1:k_2}$, where $k_1 = (k-1)n_b + 1$ and $k_2 = kn_b$, is reduced to bidiagonal form. The block reflectors U and V, and the corresponding matrices X and Y are formed and used to update the trailing submatrix $A_{k_2+1:n,k_2+1:n}$. Figure 5-3 shows step k of the blocked algorithm.

`BlockBidiag-k`(A, k, n_b)

1: Let $k_i = (k-1)n_b + i$. Repeat steps 2 to 7 for $i = 1, \cdots, n_b$.
2: Update column k_i of A.
3: Compute the column Householder reflector u_i.
4: Compute y_i using equation (5.3).
5: Update row k_i of A.
6: Compute the row Householder reflector v_i.
7: Compute x_i using equation (5.4).
8: Update the $A_{kn_b+1:n,kn_b+1:n}$ trailing submatrix by $A_{\text{sub}} = A_{\text{sub}} - UY^T - XV^T$.

Figure 5-3: k-th step of Blocked Bidiagonalization. Input A is $n \times n$

Using the blocked bidiagonalization algorithm, converting the sequential implementation to parallel is straightforward. The input data A can be distributed according to the blocking scheme, and the main operations boil down to parallel matrix-vector multiply and matrix-matrix multiply. Blocked bidiagonalization returns the diagonal and superdiagonal (or subdiagonal) elements as the output, which are in turn passed as input to an iterative QR algorithm to compute the singular values and vectors.

5.4.2 Further Parallelism

The blocked bidiagonalization outlined above leads to a straightforward parallel implementation. One goal of the Julia project is to allow flexible parallelism beyond what is already used in common numerical algorithms in practice. We discuss potential source for further parallelism here. In the discussion that follows, we assum A is a square matrix divided into $p \times p$ blocks of equal size $n_b \times n_b$ and assume the data

are distributed in parallel with some blocked layout:

$$A = \begin{bmatrix} A_{11} & A_{12} & \cdots & A_{1p} \\ A_{21} & A_{22} & \cdots & A_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ A_{p1} & A_{p2} & \cdots & A_{pp} \end{bmatrix} \tag{5.5}$$

- **Concurrent bidiagonalization of upper-left block and update of trailing submatrix** Consider the first step of the bidiagonalization algorithm, after the upper-left block A_{11} is bidiagonalized, the matrices U, V, X, Y are returned and used to update the trailing lower-right matrix blocks in terms of parallel matrix-matrix multiplications $A_{ij} = A_{ij} - U_{ij}Y_{ij}^T - X_{ij}V_{ij}^T$, where $(i, j) \in \{2, \cdots, n\} \times \{2, \cdots, n\}$. The second step of the bidiagonalization algorithm starts with bidiagonalization of the next upper-left block A_{22}, which depends on the second column panel $A_{22}, A_{32}, \cdots, A_{p2}$ and row panel $A_{22}, A_{23}, \cdots, A_{2p}$. When the matmul updates from the first step on the second column and row panels are finished, then the bidiagonalization of A_{22} can start, and it does not need to wait for the matmul update of the remaining blocks in the lower-right submatrix. This concurrent work scheduling is possible in every k-th step of the algorithm. Whether this kind of dependency should be handled by the work scheduler or the information should be provided by the user is one of the tasks currently worked on in the Julia project.

- **Minimizing idle time before updating trailing submatrix** As described above, after the upper-left block A_{11} is bidiagonalized, the trailing submatrix is updated by $A = A - UY^T - XV^T$. Recall from the algorithm in Figure 5-3, when each column and row within the block A_{11} is updated with Householder transformation, the corresponding vectors u_i, v_i, x_i and y_i are formed, and each of these vector forms the i-th column of the matrices U, V, X and Y respectively. From now on, we refer to the matrices U, V, X, and Y collectively as *updator matrices*. Following the algorithm, the update of the trailing submatrix (the

lower-right blocks starting at A_{22}) can only be updated once all the updator matrices are ready. However, recall that the trailing submatrix updates using matrix-matrix multiply are derived from grouping rank-one updates:

$$A = A - UY^T - XV^T = A - \sum_{i=1}^{n_b} u_i y_i^T - \sum_{i=1}^{n_b} x_i v_i^T \tag{5.6}$$

Since the bidigaonlization within each block A_{kk} is serial, the columns of the updator matrices wil be filled in serially from left to right. During this series of serial operations, all the other processors storing the lower-right submatrix will just be idle waiting for the updator matrices. One question we ask is can we break down the update into smaller parts and start the trailing matrix update earlier? For example, when half of the columns of the updator matrices are ready, we can start forming those updates and reduce idle times of many processors:

$$\begin{aligned} A = A - UY^T - XV^T &= A - \sum_{i=1}^{n_b/2} u_i y_i^T - \sum_{i=1}^{n_b/2} x_i v_i^T - \sum_{i=n_b/2+1}^{n_b} u_i y_i^T - \sum_{i=n_b/2+1}^{n_b} x_i v_i^T \\ &= A - U_a Y_a^T - X_a V_a^T - U_b Y_b^T - X_b V_b^T, \end{aligned}$$

wher $U_a = (u_1, \cdots, u_{n_b/2})$, $U_b = (u_{n_b/2+1}, \cdots, u_{n_b})$ and similarly for other updator matrices.

Will starting the updates earlier save enough idle time to compensate for the larger communication costs? Since 4 matrix multiplications with half of the matrix sizes are performed instead, will starting the updates earlier be worth the decrease in speedup from BLAS-3? How can Julia expose this kind of possible parallelism control to the user? Finding the right level of abstraction is key to finding the right answers. A level that is too high does not give enough control in this kind of parallelism, while very low-level makes it difficult to program.

The kinds of update outlined in this section are common in many blocked algorithms in numerical algebra, such as Cholesky factorization and tridiagonalization. They all reduce to performing matrix-matrix multiplications $C = AB$, when not all the columns/rows of A and B are ready at the same time. Finding a good way to experiment with this kind of scheduling choices in Julia, which is lacking in standard numerical linear algebra packages, can lead to improvement in parallel performance of many common kernels.

5.4.3 Other SVD Algorithms

[50, 57] present divide-and-conquer algorithms for the SVD of a bidiagonal matrix. The LAPACK routine `xbdsdc` uses variation of these algorithms. Divide-and-conquer can possibly be implemented to replace the QR iteration step in the future parallel implementation of Julia. Another bidiagonal SVD algorithm based on the Multiple Relatively Robust Representation (MRRR) algorithm [33, 34, 74] for computing numerically orthogonal eigenvectors is presented in [91]. A recent divide-and-conquer SVD algorithm, which uses randomized rank-revealing decompositions and aims at minimizing communication in parallel, is studied in [9].

5.5 Chapter Summary

This chapter introduced Julia, a new high-level programming language. The ongoing Julia project aims at filling the gap between dynamic language and traditional compiled languages by giving good performance and offering ease of use and flexible parallelism without extensive effort. We gave a brief tutorial of Julia and presented some elementary results of Julia. By implementing a serial blocked SVD algorithm, we repeated the SVD-based algorithm for terrain analysis presented in Chapter 4, and found that the values of the roughness measures μ obtained agree with our prediction. We also discussed parallel implementation of our SVD algorithm and discussed how potentially further paralleism can be explored in Julia. Our proposed implementation of parallel SVD suggests asynchronous work in bidiagonalization and trailing matrix

update, which has not been explored in standard numerical linear algebra packages.

THIS PAGE INTENTIONALLY LEFT BLANK

Chapter 6

Conclusion

In this thesis, we presented two recent parallel programming languages, PetaBricks and Julia.

We have re-examined a number of numerical algorithms including the symmetric eigenproblem, LU Factorization, k-means clustering, preconditioned conjugate gradient, with PetaBricks. PetaBricks allows users to express algorthimc choice and accuracy requirements explicitly so the compiler can perform deeper optimization. We have shown that even with familiar classic numerical algorithms, the optimal selection of algorithms is nontrivial when fine-grained algorithmic choice is added to the many other choices involved in parallel computation. PetaBricks can increase the lifetimes of programs since the compiler and autotuner can easily start the autotuning process again and compose a complex program with different choices to produce an optimal composition with changing architecture. This gives programs the portable performance needed for longer effective lifetimes.

Future work for PetaBricks includes extending the benchmark suite and support for different data structures and storage format, such as sparse matrices and graphs. Dynamic choices of algorithms where run-time analysis for algorithm and parameter tuning can also be explored. Another future direction is adding a distributed memory backend to the PetaBricks compiler, which can further highlight the importance of choices for algorithms. Exploring compiler for interesting architectures such as CPU-GPU heterogeneous systems and embedded systems can also be an interesting

direction.

We also introduced a new high-level programming language Julia, and applied it to perform terrain data analysis. We presented some results of Julia and demonstrated that programming in Julia can be as easy as one of the most popular high-level language Matlab for the numerical computing community. While still an ongoing project, Julia aims at providing easy but flexible parallelism. In this thesis, we used the parallel SVD algorithm as an example and motivation, but once the ideas are explored further and implemented with our SVD algorithm, many other numerical kernels can be written in similar fashion in parallel using Julia easily without losing too much control over the parallelization. We believe we have introduced what is the first step of truly flexible and elegant parallelism.

Throughout the thesis, we have presented new ways to approach high-performance computing, including autotuning of parallel algorithmic choices, and using a new dynamic programming language to fill in the gap between full-control low-level programming interface, and high-level tool that hides every detail from the user. With the advent of multicore processors, parallel computing has been brought to the mainstream, but the complications involved with parallel computing compared to serial codes still present major challenges for researchers and programmers. By no means we are suggesting that PetaBricks and Julia are the only solutions to the difficulties, but we believe these two languages provide promising potential to address some of the major issues in high-performance computing.

Appendix A

PetaBricks Code

A.1 Symmetric Eigenproblem

A.1.1 BisectionTD.pbcc

```
#ifndef BISECTIONTD_PBCC
#define BISECTIONTD_PBCC

%{
extern "C" void dstebz_(char *range, char *order, int *n, double *vl,
double *vu, int *il, int *iu,  double *abstol,  double *d, double *e,
int *m, int *nsplit, double *w, int *iblock, int *isplit,
double *work, int *iwork, int *info);

extern "C" void dstein_(int *n, double *d, double *e, int *m, double *w,
int *iblock, int *isplit, double *z, int *ldz, double *work, int *iwork,
int *ifail, int *info);
%}

// Find eigenvalues and eigenvectors of input matrix IN (symmetric)
// by LAPACK Bisection routine
transform BisectionTD
from Diag[n], Subdiag[n-1]
to Eigenvalue[n], Eigenvector[n,n]
```

```
{

  to (Eigenvalue Eig, Eigenvector Vec) from (Diag Din, Subdiag Ein)
  {
    char range = 'A';
    char order = 'B';
    int size=n;
    int info = 0;
    int il;
    int iu;
    int m;
    int nsplit;
    int nsquared=n*n;
    double vl;
    double vu;
    double abstol=0;

    // allocate and initialize matrices
    double *Z = Vec.base();
    double *D = Din.storage()->data();
    double *E = Ein.storage()->data();
    double *work = new double[4*size];
    int *iwork = new int[3*size];
    double *W = Eig.base();
    int *iblock = new int[size];
    int *isplit = new int[size];
    int *ifail = new int[size];

    // call LAPACK bisection routine for tridiagonal matrix
    dstebz_(&range, &order, &size, &vl, &vu, &il, &iu, &abstol, D,
            E, &m, &nsplit, W, iblock, isplit, work, iwork, &info);

    // call LAPACK dstein routine for eigenvectors
    delete [] work;
    work = new double[5*n];
    delete [] iwork;
```

```
    iwork = new int[n];
    dstein_(&size, D, E, &size, W, iblock,
    isplit, Z, &size, work, iwork, ifail, &info);


    delete [] work;
    delete [] iwork;
    delete [] iblock;
    delete [] isplit;
    delete [] ifail;
  }

}

#endif // BISECTIONTD_PBCC
```

A.1.2 QRTD.pbcc

```
#ifndef QRTD_PBCC
#define QRTD_PBCC

%{
extern "C" void dsteqr_(char *compz, int *n, double *D, double *E,
double *Z, int *ldz, double *work, int *info);
%}

#include "../simple/copy.pbcc"

// Find eigenvalues and eigenvectors of tridiagonal matrix
// by LAPACK QR iteration routines
transform QRTDsub
from Subdiag[n-1]
to Eigenvalue[n], Eigenvector[n,n]
{

  to (Eigenvalue Eig, Eigenvector Vec) from (Subdiag Ein)
```

```
  {


    char compz='I';
    int size=n, lwork=2*n-2, info=0, nsquared=n*n;

    // allocate and initialize matrices
    double *Z = Vec.base();
    double *D = Eig.base();
    double *E = Ein.storage()->data();
    double *work = new double[1];

    if (lwork > 1) { work = new double[lwork];}
    dsteqr_(&compz, &size, D, E, Z, &size, work, &info);

    delete [] work;
  }

}

transform QRTD
from Diag[n], Subdiag[n-1]
to Eigenvalue[n], Eigenvector[n,n]
{

  to (Eigenvalue Eig, Eigenvector Vec) from (Diag Din, Subdiag Ein)
  {
        Copy1D(Eig, Din);
        QRTDsub(Eig, Vec, Ein);
  }

}

#endif // QRTD_PBCC
```

A.1.3 EigTD.pbcc

```
#ifndef EIGTD_PBCC
#define EIGTD_PBCC

%{
extern "C" void dlaed1_(int *n, double *D, double *Q, int *ldq,
int *indxq, double *rho,  int *cutpnt, double *work, int *iwork,
int *info);
%}

#include "../simple/copy.pbcc"
#include "QRTD.pbcc"
#include "BisectionTD.pbcc"

transform mergeQ
from Q1[n1,n1], Q2[n2,n2]
to Q[n1+n2,n1+n2]
{
   to (Q.region(0,0,n1,n1) qin) from(Q1 q1){
        Copy2D(qin,q1);
   }

   to (Q.region(n1,n1,n1+n2,n1+n2) qin) from(Q2 q2){
        Copy2D(qin,q2);
   }

   secondary Q.cell(i,j) from () { return 0;}
}



transform ComputeEig
from Vec1[n1,n1],Vec2[n2,n2], RHO
to Eigenvalue[n1+n2], Eigenvector[n1+n2,n1+n2],
   WORK[(n1+n2)*(n1+n2)+4*(n1+n2)]
{
  to (Eigenvalue eig, Eigenvector vec, WORK work)
```

```
  from (Vec1 q1, Vec2 q2, RHO b)
  {

   if(n1==0) PetabricksRuntime::abort();

    int i;
    int info;
    int size=n1+n2;
    int cutpnt=n1;
    int nsquared=size*size;
    double rho=b;
    double *D = eig.base();
    double *Q = vec.base();
    int *indxq = new int[size];
    int *iwork = new int[4*size];

    mergeQ(vec,q1,q2);

    for(i=0; i<cutpnt; i++) {
      indxq[i]=i+1;
    }
    for(i=cutpnt; i<size; i++){
      indxq[i]=i-cutpnt+1;
    }

    dlaed1_(&size, D, Q, &size, indxq, &rho, &cutpnt, work.base(),
            iwork, &info);
    delete [] iwork;
    delete [] indxq;
  }
}

// Find eigenvalues and eigenvectors of input matrix T (tridiagonal symmetric)
transform EigTD
from Diag[n], Subdiag[n-1]
to Eigenvalue[n], Eigenvector[n,n], TMP[n,n], WORK[n*n+4*n]
```

```
{

    //Bisection
    recursive
    to (Eigenvalue eig, Eigenvector vec, TMP tmp, WORK work)
    from  (Diag Din, Subdiag Ein) {
      if (n == 1) {
        eig.cell(0) = Din.cell(0);
        vec.cell(0,0) = 1;
      }
      else {
        SPAWN(BisectionTD, eig, vec, Din, Ein);
      }
    }


  //QR Iterations
  recursive
  to (Eigenvalue eig, Eigenvector vec, TMP tmp, WORK work)
  from  (Diag Din, Subdiag Ein){
    if (n == 1) {
        eig.cell(0) = Din.cell(0);
        vec.cell(0,0) = 1;
    }
    else {
        SPAWN(QRTD, eig, vec, Din, Ein);
    }
  }

  //Recursive: Divide and Conquer
  recursive
  to (Eigenvalue eig, Eigenvector vec, TMP tmp, WORK work)
  from  (Diag Din, Subdiag Ein){
    if (n == 1) {
        eig.cell(0) = Din.cell(0);
```

```
        vec.cell(0,0) = 1;
    }
    else if (n<=3)  {
        QRTD(eig,vec,Din,Ein);
    }
    else {
    int cutpnt=n/2;
    double rho=Ein.cell(cutpnt-1);
    MatrixRegion2D q1 = tmp.region(0,   0,   n/2, n/2);
    MatrixRegion2D q2 = tmp.region(n/2, n/2, n,   n);
    MatrixRegion2D t1 = tmp.region(n/2, 0,   n,   n/2);
    MatrixRegion2D t2 = tmp.region(0,   n/2, n/2, n);
    int wSize =  (n/2)*(n/2) + 4*(n/2);
    JASSERT(wSize < work.count()/2);
    MatrixRegion1D w1 = work.region(0, wSize);
    MatrixRegion1D w2 = work.region(work.count()/2, work.count()/2 + wSize);

    Copy1D(eig, Din);

    eig.cell(cutpnt-1) -= rho;
    eig.cell(cutpnt)   -= rho;

    SPAWN(EigTD, eig.region(0,cutpnt),q1, t1, w1,
          eig.region(0,cutpnt),Ein.region(0,cutpnt-1));
    SPAWN(EigTD, eig.region(cutpnt,n),q2, t2, w2,
          eig.region(cutpnt,n),Ein.region(cutpnt,n-1));
    SYNC();
    SPAWN(ComputeEig, eig, vec, work, q1, q2, Ein.cell(cutpnt-1));
    }
  }
}

transform EigTDMain
from Diag[n], Subdiag[n-1]
through TMP[n,n], WORK[n*n+4*n], Eigvectemp[n,n]
to Eigenvalue[n], Eigenvector[n,n]
```

```
{
  to (Eigenvalue eig, Eigenvector vec, Eigvectemp vectemp,
      TMP tmp, WORK work)
  from  (Diag Din, Subdiag Ein){
    EigTD(eig, vectemp, tmp, work, Din, Ein);
    Transpose(vec, vectemp);
  }
}


#endif // EIGTD_PBCC
```

A.2 LU Factorization

A.2.1 PLU.pbcc

```
#ifndef PLU_PBCC
#define PLU_PBCC

#include "../simple/copy.pbcc"

transform PLUleftwork
from IN[n,m]
to OUT[n,m]
{

  primary to (OUT.column(n-1) out)  from (IN in)
  {
        ElementT sum;
        int minjn;

        out.cell(0)=in.cell(n-1,0);

        for (int j=1; j<m; j++)
        {
                sum = 0;
```

```
                if (j<n)
                {
                        minjn=j;
                }
                else
                {
                        minjn=n-1;
                }

                for (int k=0; k<minjn; k++)
                {
                        sum+=in.cell(k,j)*out.cell(k);
                }
                out.cell(j)=in.cell(n-1,j)-sum;
        }
  }

  OUT.cell(j,i)
  from (IN.cell(j,i) in)
  {
        return in;
  }
}

transform PLUscalecolumn
from IN[n,m]
to OUT[n,m]
{
  primary OUT.cell(0,0) from (IN.cell(0,0) in)
  {
        return in;
  }

  OUT.cell(0,j) from (IN.cell(0,0) diag, IN.cell(0,j) in)
  {
        return in/diag;
```

```
  }


  secondary OUT.cell(i,j) from (IN.cell(i,j) in)
  {
        return in;
  }
}


transform PLUleft
from IN[n,m], Pin[m]
through TEMP[n,m], Ptemp[m]
to OUT[n,m], Pout[m]
{

   to (OUT out, Pout pout, TEMP temp, Ptemp ptemp)
   from (IN in, Pin pin)
   {
        ElementT MaxElement;
        int MaxIndex;
        int minmn;
        int j;
        int k;

        Copy2D(temp, in);
        Copy1D(ptemp, pin);

        if ( m >= n ) {
                minmn=n;
        }
        else {
                minmn=m;
        }

        for (j=0; j<minmn; j++)
```

```
{
    if (j>0)
    {
       PLUleftwork(temp.region(0,0,j+1,m),out.region(0,0,j+1,m));
    }

    Copy1D(out.column(j),temp.column(j));

    //find pivot
    MaxElement=temp.cell(j,j);
    MaxIndex=j;
    for (k=j+1; k<m; k++)
    {
            if (abs(temp.cell(j,k)) > abs(MaxElement)){
            MaxElement=temp.cell(j,k);
            MaxIndex=k;
            }
    }

    //swap rows
    pout.cell(j)=ptemp.cell(MaxIndex);
    Copy1D(out.row(j), temp.row(MaxIndex));
    for (k=j+1; k<m; k++)
    {
            if (k==MaxIndex)
            {
                    pout.cell(k)=ptemp.cell(j);
                    Copy1D(out.row(k),temp.row(j));
            }
            else
            {
                    pout.cell(k)=ptemp.cell(k);
                    Copy1D(out.row(k), temp.row(k));
            }
    }
    Copy1D(ptemp.region(j,m),pout.region(j,m));
```

```
            Copy2D(temp.region(0,j,n,m),out.region(0,j,n,m));
            PLUscalecolumn(out.region(j,j,j+1,m),temp.region(j,j,j+1,m));
        }

        for (j=minmn; j<n; j++)
        {
            PLUleftwork(temp.region(0,0,j+1,m),out.region(0,0,j+1,m));
            Copy1D(out.column(j),temp.column(j));
        }
    }
}

transform PLUrightwork
from IN[n,m]
to OUT[n,m]
{

  primary OUT.cell(j,0) from (IN.cell(j,0) in)
  {
        return in;
  }

  OUT.cell(0,j) from (IN.cell(0,j) in, IN.cell(0,0) diag)
  {
        return in/diag;
  }


  secondary OUT.cell(j,i)
  from (IN.cell(j,i) aPrev,
        OUT.cell(0,i) left,
        IN.cell(j,0) up)
  {
        return aPrev - left * up;
  }
```

```
}

transform PLUright
from IN[n,m], Pin[m]
through TEMP[n,m], Ptemp[m]
to OUT[n,m], Pout[m]
{

   to (OUT out, Pout pout, TEMP temp, Ptemp ptemp) from (IN in, Pin pin)
   {
        ElementT MaxElement;
        int MaxIndex;
        int minmn;
        int j;
        int k;

        Copy2D(temp, in);
        Copy1D(ptemp, pin);

        if ( m >= n ) {
                minmn=n;
        }
        else {
                minmn=m;
        }

        for (j=0; j<minmn; j++)
        {
                MaxElement=temp.cell(j,j);
                MaxIndex=j;
                for (k=j+1; k<m; k++)
                {
                   if ( (temp.cell(j,k) != 0 ) && (abs(temp.cell(j,k))
                         > abs(MaxElement)))
                   {
                        MaxElement=temp.cell(j,k);
```

```
                MaxIndex=k;
            }
        }

        pout.cell(j)=ptemp.cell(MaxIndex);
        Copy1D(out.row(j), temp.row(MaxIndex));
        for (k=j+1; k<m; k++)
        {
                if (k==MaxIndex)
                {
                        pout.cell(k)=ptemp.cell(j);
                        Copy1D(out.row(k),temp.row(j));
                }
                else
                {
                        pout.cell(k)=ptemp.cell(k);
                        Copy1D(out.row(k), temp.row(k));
                }
        }


        Copy1D(ptemp.region(j,m),pout.region(j,m));
        Copy2D(temp.region(0,j,n,m),out.region(0,j,n,m));

        PLUrightwork(temp.region(j,j,n,m),out.region(j,j,n,m));

        if (j == minmn-1)
        {
          Copy2D(out.region(j,j,n,m),temp.region(j,j,n,m));
        }
    }
  }
}

transform PLUnoblock
from IN[n,m], Pin[m]
```

```
to OUT[n,m], Pout[m]
{
   to (OUT out, Pout pout) from (IN in, Pin pin)
   {
        PLUright(out, pout, in, pin);
   }

   to (OUT out, Pout pout) from (IN in, Pin pin)
   {
        PLUleft(out, pout, in, pin);
   }

}

transform PLU
from IN[n,m]
through Pinitial[m]
to OUT[n,m], P[m]
{
   Pinitial.cell(j) from() { return j;}

   to (OUT out, P pout) from (IN in, Pinitial pin)
   {
       PLUnoblock(out,pout,in,pin);
   }
}

#endif // PLU_PBCC
```

A.2.2 PLUblockdecomp.pbcc

```
#ifndef PLUBLOCKDECOMP_PBCC
#define PLUBLOCKDECOMP_PBCC


#include "LUtrisolve.pbcc"
```

```
#include "PLU.pbcc"
#include "LAPACKmatmul.pbcc"

transform SwapElements
from IN[n], P[n]
to OUT[n]
{
  to (OUT out) from (IN in, P p)
  {
        for (int j=0; j<n; j++)
        {
                if (p.cell(j) >= 0 && p.cell(j) < n) {
                        out.cell(j)=in.cell(p.cell(j));
                }
                else {
                        out.cell(j)=in.cell(j);
                }
        }
  }
}

transform SwapRows
from IN[n,m], P[m]
to OUT[n,m]
{
  to (OUT out) from (IN in, P p)
  {
        for (int j=0; j<m; j++)
        {
                if (p.cell(j) >= 0 && p.cell(j) < m) {
                        Copy1D(out.row(j), in.row(p.cell(j)));
                }
                else {
                        Copy1D(out.row(j),in.row(j));
                }
        }
```

```
  }
}

transform PLUblock2
from IN[n, n], Pin[n]
through TEMP[n, n], Ptemp[n], Phalf[n-n/2]
to OUT[n, n], Pout[n]
{
  //get LU of left block by recursion
  to (TEMP.region(0,0,n/2,n) temp, Ptemp ptemp)
  from (IN.region(0,0,n/2,n) in, Pin pin)
  {
        PLUnoblock(temp,ptemp,in, pin);
  }

  //upper left block is final
  to (OUT.region(0,0,n/2,n/2) out) from (TEMP.region(0,0,n/2,n/2) temp)
  {
        Copy2D(out, temp);
  }

  //first half of Permutation vector is final
  to (Pout.region(0,n/2) pout) from (Ptemp.region(0,n/2) ptemp)
  {
        Copy1D(pout, ptemp);
  }

  //swap right block
  to (TEMP.region(n/2,0,n,n) temp)
  from (IN.region(n/2,0,n,n) in, Ptemp ptemp)
  {
        SwapRows(temp, in, ptemp);
  }

  //upper right block
  to (OUT.region(n/2,0,n,n/2) out)
```

```
from (TEMP.region(n/2,0,n,n/2) swappedin,
      TEMP.region(0,0,n/2,n/2) leftdiag)
{
      LUtrisolve(out,leftdiag,swappedin);
}

//lower right block
to (OUT.region(n/2,n/2,n,n) out, Phalf phalf)
from (TEMP.region(n/2,n/2,n,n) in,
      OUT.region(n/2,0,n,n/2) up,
      TEMP.region(0,n/2,n/2,n) left)
{
      //Atemp = in - left*up (this updates the lower right block)
      MatrixRegion2D Atemp = MatrixRegion2D::allocate(n-n/2, n-n/2);
      //MatrixMultiply(out,left,up);
      //MatrixSub(Atemp,in,out);

      LAPACKmatmul(Atemp,left,up,in);

      //recursion to get LU of the remaining updated block
      PLUrecur(out,phalf,Atemp);
}

//lower left block
to (OUT.region(0,n/2,n/2,n) out)
from (TEMP.region(0,n/2,n/2,n) temp,  Phalf phalf)

{
      SwapRows(out,temp,phalf);
}

//last half of Permutation vector
to (Pout.region(n/2,n) pout) from (Ptemp.region(n/2,n) ptemp, Phalf phalf)
{
      SwapElements(pout, ptemp, phalf);
}
```

```
}

transform PLUblock4
from IN[n, n], Pin[n]
through TEMP[n, n], Ptemp[n], Phalf[n-n/4]
to OUT[n, n], Pout[n]
{
  //get LU of left block by recursion
  to (TEMP.region(0,0,n/4,n) temp, Ptemp ptemp)
  from (IN.region(0,0,n/4,n) in, Pin pin)
  {
        PLUnoblock(temp,ptemp,in, pin);
  }

  //upper left block is final
  to (OUT.region(0,0,n/4,n/4) out) from (TEMP.region(0,0,n/4,n/4) temp)
  {
        Copy2D(out, temp);
  }

  //first half of Permutation vector is final
  to (Pout.region(0,n/4) pout) from (Ptemp.region(0,n/4) ptemp)
  {
        Copy1D(pout, ptemp);
  }

  //swap right block
  to (TEMP.region(n/4,0,n,n) temp)
  from (IN.region(n/4,0,n,n) in, Ptemp ptemp)
  {
        SwapRows(temp, in, ptemp);
  }

  //upper right block
  to (OUT.region(n/4,0,n,n/4) out)
  from (TEMP.region(n/4,0,n,n/4) swappedin,
```

```
      TEMP.region(0,0,n/4,n/4) leftdiag)
{
      LUtrisolve(out,leftdiag, swappedin);
}

//lower right block
to (OUT.region(n/4,n/4,n,n) out, Phalf phalf)
from (TEMP.region(n/4,n/4,n,n) in,
      OUT.region(n/4,0,n,n/4) up,
      TEMP.region(0,n/4,n/4,n) left)
{
      //Atemp = in - left*up (this updates the lower right block)
      MatrixRegion2D Atemp = MatrixRegion2D::allocate(n-n/4, n-n/4);
      //MatrixMultiply(out,left,up);
      //MatrixSub(Atemp,in,out);

      LAPACKmatmul(Atemp,left,up,in);

      //recursion to get LU of the remaining updated block
      PLUrecur(out,phalf,Atemp);
}

//lower left block
to (OUT.region(0,n/4,n/4,n) out)
from (TEMP.region(0,n/4,n/4,n) temp,  Phalf phalf)

{
      SwapRows(out,temp,phalf);
}

//last half of Permutation vector
to (Pout.region(n/4,n) pout) from (Ptemp.region(n/4,n) ptemp, Phalf phalf)
{
      SwapElements(pout, ptemp, phalf);
}
```

```
}

#endif // PLUBLOCKDECOMP_PBCC
```

A.2.3 PLUrecur.pbcc

```
#ifndef PLURECUR_PBCC
#define PLURECUR_PBCC

#include "PLUblockdecomp.pbcc"
#include "PLU.pbcc"

transform PLUrecurinner
from IN[n,n], Pin[n]
to OUT[n,n], Pout[n]
{

  to (OUT out, Pout pout) from (IN in, Pin pin)
  {
        if (n < 2) {
                PLUnoblock(out, pout, in, pin);
        }
        else {
                PLUblock2(out, pout, in, pin);
        }
  }

  to (OUT out, Pout pout) from (IN in, Pin pin)
  {
        if (n < 4) {
                PLUnoblock(out, pout, in, pin);
        }
        else {
                PLUblock4(out, pout, in, pin);
        }
  }
```

```
  to (OUT out, Pout pout) from (IN in, Pin pin)
  {
        PLUnoblock(out,pout,in,pin);
  }

}

main transform PLUrecur
from IN[n,n]
through Pinitial[n]
to OUT[n,n], Pout[n]
{
  Pinitial.cell(j) from () { return j;}

  to (OUT out, Pout pout) from (IN in, Pinitial pin)
  {
        PLUrecurinner(out,pout,in,pin);
  }
}

#endif // PLURECUR_PBCC
```

A.3 k-means clustering

A.3.1 newclusterlocation.pbcc

```
#ifndef NEWCLUSTERLOCATIONS_PBCC
#define NEWCLUSTERLOCATIONS_PBCC

transform NewClusterLocationsGen
to X[n,2], A[n]
{
  X.cell(i,j) from() { return PetabricksRuntime::randInt(-50,50); }
  A.cell(i) from() { return PetabricksRuntime::randInt(0,n-1); }
}
```

```
transform NewClusterLocations
param k //input parameter: number of clusters
from X[n,2], A[n]
through Sum[k,2], Count[k]
to C[k,2]
generator NewClusterLocationsGen
{
  to (Sum s, Count count) from (X x, A a)
  {
    int i, j;
    //zero s and count
    for(i=0;i<k;i++){
      s.cell(i,0)=0;
      s.cell(i,1)=0;
      count.cell(i)=0;
    }
    for (i=0; i<n; i++) {
      j=a.cell(i);
      JASSERT(j>=0 && j<=k)(j);
      s.cell(j,0)+=x.cell(i,0);
      s.cell(j,1)+=x.cell(i,1);
      count.cell(j)+=1;
    }
  }

  to (C.column(i) c) from (Sum.column(i) s, Count.cell(i) count)
  {
    if (count == 0) {
      c.cell(0)=0;
      c.cell(1)=0;
    }
    else {
      c.cell(0)=s.cell(0)/count;
      c.cell(1)=s.cell(1)/count;
    }
```

```
  }
}

#endif // NEWCLUSTERLOCATIONS_PBCC
```

A.3.2 assignclusters.pbcc

```
#ifndef ASSIGNCLUSTERS_PBCC
#define ASSIGNCLUSTERS_PBCC

transform AssignClustersGen
to X[n,2], C[n/10,2], A[n]
{
  X.cell(i,j) from() { return PetabricksRuntime::randDouble(-50,50); }
  C.cell(i,j) from() { return PetabricksRuntime::randDouble(-50,50); }
  A.cell(i) from() { return PetabricksRuntime::randInt(0,n/10-1); }
}

transform AssignClusters
from X[n,2], C[k,2], A[n]
through D[n,k], ctemp[n]
to Anew[n], cflag
generator AssignClustersGen
{

        to (D.cell(i,j) d) from (X.column(i) x, C.column(j) c)
        {
                Distance(d,x,c);
        }

        to(Anew.cell(i) anew, ctemp.cell(i) changed)
        from(D.column(i) d, A.cell(i) aold)
        {
                IndexT oldindex = aold;
                IndexT minindex = 0;
                ElementT mindist=d.cell(0);
```

```
                for (int j=1; j<k; j++) {
                        if (d.cell(j) < mindist) {
                                minindex=j;
                                mindist = d.cell(j);
                        }
                }
    anew = minindex;
    changed = (oldindex!=minindex);
        }

        cflag from (ctemp c) {
                int i;
                ElementT sum=0;
                for (i=0; i<n; i++) {
                        sum+=c.cell(i);
                }
                return sum;
        }
}

#endif // ASSIGNCLUSTERS_PBCC
```

A.3.3 kmeans.pbcc

```
#ifndef KMEANS_PBCC
#define KMEANS_PBCC

#include "../simple/copy.pbcc"
#include "newclusterlocations.pbcc"
#include "assignclusters.pbcc"
#include "../simple/rollingsum.pbcc"

transform Distance
from A[2], B[2]
to Dis
{
```

```
  Dis from (A a, B b)
  {
        ElementT xdiff, ydiff;
        xdiff = a.cell(0)-b.cell(0);
        ydiff = a.cell(1)-b.cell(1);
        return sqrt(xdiff*xdiff + ydiff*ydiff);
  }
}

transform DistanceSQ
from A[2], B[2]
to DisSQ
{
  DisSQ from (A a, B b)
  {
        ElementT xdiff, ydiff;
        xdiff = a.cell(0)-b.cell(0);
        ydiff = a.cell(1)-b.cell(1);
        return xdiff*xdiff + ydiff*ydiff;
  }
}

transform GetD
from X[m,2], C[n,2]
to D[m,n]
{
  to (D.cell(i,j) d) from (X.column(i) x, C.column(j) c)
  {
    Distance(d,x,c);
  }
}

transform GetMin
from X[m,n]
to MinX[m]
{
```

```
  MinX.cell(i) from(X.column(i) x){
    int j;
    ElementT minvalue=x.cell(0);
    for (j=1; j<n; j++) {
      if (x.cell(j) < minvalue) {
        minvalue=x.cell(j);
      }
    }
    return minvalue*minvalue;
  }
}

transform randomcenter
from X[n,2]
through Xtemp[n,2]
to C[k,2]
generator kmeansinputgen
{
  to (C c, Xtemp x) from (X xin){
    int l;
    int m;

    Copy2D(x,xin);
    for (m=0; m<k; m++)
    {
      l = PetabricksRuntime::randInt(m,n);
      Copy1D(c.column(m),x.column(l)); //new center picked
      if (l!=m) {
        Copy1D(x.column(l), x.column(m));
      }//swap columns for next iteration
    }
  }
}

transform centerplus
from X[n,2]
```

```
through D[n,k], DMIN[n], DSQ[n], Xtemp[n,2]
to C[k,2]
generator kmeansinputgen
{
  to (C c, D d, DMIN dmin, DSQ dsq, Xtemp x) from (X xin){
    int l;
    int m;
    ElementT rvalue;


    Copy2D(x,xin);
    Copy1D(c.column(0), x.column(0));

    for (m=1; m<k; m++) {

      //get distance of all remanining x with current cluster centers
      GetD(d.region(m,0,n,m), x.region(m,0,n,2), c.region(0,0,m,2));

      //find minimum of each column, squared and compute cumulative sum
      GetMin(dmin.region(m,n), d.region(m,0,n,m));
      RollingSum(dsq.region(m,n),dmin.region(m,n));

      //pick center with probability proportional to dmin (D(x)^2)
      rvalue=PetabricksRuntime::randDouble(0,1)*dsq.cell(n-1);
      for (l=m; l<n; l++){
        if (rvalue<=dsq.cell(l)){
          Copy1D(c.column(m),x.column(l)); //new center picked
          if (l!=m) { Copy1D(x.column(l), x.column(m)); }
          //swap columns for next iteration
          break;
        }
      }
    }
  }
}
```

```
main transform kmeans
from X[n,2]    // X - x,y coordinates of n points
through Ctemp[k,2]
to C[k,2], A[n]
// C - centroids, A - cluster assignment,
// WCSS - within-cluster sum of squares  (error measures)
accuracy_metric WCSS
accuracy_bins 0.025, 0.05, 0.1, 0.2, 0.5, 0.75, 0.95
accuracy_variable k(1,1)//min=1, initial=1
generator kmeansinputgen
{

  //Assign initial cluster centers randomly
  to (Ctemp ctemp) from (X x)
  {
    //Copy2D(ctemp,x.region(0,0,k,2));
    if(k>n)
      PetabricksRuntime::abort();
    randomcenter(ctemp, x);
  }

  //or Assign initial cluster centers using k-means++ algorithm
  to (Ctemp ctemp) from (X x)
  {
    if(k>n)
      PetabricksRuntime::abort();
    centerplus(ctemp, x);
  }

  //iteratively find local optimum
  to (C c, A a) from (X x, Ctemp ctemp)
  {
    ElementT change=1;
    Copy2D(c,ctemp);
    AssignClusters(a,change,x,c,a);
    for_enough {
```

```
      if (change > 0) {
        NewClusterLocations(c, x, a);
      }else{
        break;
      }
      AssignClusters(a,change,x,c,a);
    }
  }
}

transform WCSS
from C[k,2], A[n], X[n,2]
to Accuracy
{

  //accuracy measure
  Accuracy from(X x, C c, A a)
  {
    ElementT dis;
    ElementT sum;
    int i;
    sum=0;
    for (i=0; i< n; i++) {
      DistanceSQ(dis, x.column(i), c.column(a.cell(i)));
      sum+=dis;
    }

    if(sum<=0)
      return 0;
    return sqrt(2*n/sum);
  }

}

transform kmeansinputgen
from IN[n]
```

```
to X[n,2]
{
  //PetabricksRuntime::randNormal(double mean, double sigma);
  to (X x) from() {
        int i,j,k;
        int numclus=sqrt(n);
        int binlength=(n-numclus)/numclus;

        for (i=0; i < numclus; i++) {
                x.cell(i,0) = PetabricksRuntime::randDouble(-250,250);
                x.cell(i,1) = PetabricksRuntime::randDouble(-250,250);
                for (j=0; j < binlength; j++) {
                  k = numclus + i*binlength + j;
                  x.cell(k,0)=x.cell(i,0) + PetabricksRuntime::randNormal(0,1);
                  x.cell(k,1)=x.cell(i,1) + PetabricksRuntime::randNormal(0,1);
                }
        }

        for (i=k+1; i<n; i++){
           x.cell(i,0)=x.cell(n-i-1,0) + PetabricksRuntime::randNormal(0,1);
           x.cell(i,1)=x.cell(n-i-1,1) + PetabricksRuntime::randNormal(0,1);
        }
  }
}
#endif // KMEANS_PBCC
```

A.4 Preconditioning

A.4.1 poissionprecond.pbcc

```
#ifndef POISSONPRECOND_PBCC
#define POISSONPRECOND_PBCC

#include "CG.pbcc"
#include "PCG.pbcc"
#include "jacobipre.pbcc"
```

```
#include "polypre.pbcc"
#include "demv.pbcc"
#include "ComputeError.pbcc"
#include "../simple/transpose.pbcc"
#include "../multiply/multiply.pbcc"

main transform poissonprecond
from X[n], A[n,n], B[n]
// X - initial guess, A - input matrice, B - RHS vector
to OUT[n]
generator PoissonGenerator
accuracy_metric ResidualNorm
accuracy_bins 0, 0.5, 1, 1.5, 2, 3
accuracy_variable NumIterations
{

        //Jacobi preconditioner
        to (OUT out) from (X x, A a, B b)
        {
                JacobiPre(out, x, a, b, NumIterations);
        }

        //Polynomoial preconditioner
        to (OUT out) from (X x, A a, B b)
        {
                MatrixRegion2D p = MatrixRegion2D::allocate(n,n);
                PolyPre(p,a);
                PCG(out, x, a, b, p, NumIterations);
        }

        //no preconditioner
        to (OUT out) from (X x, A a, B b)
        {
                CG(out, x, a, b, NumIterations);
        }
}
```

```
transform PoissonGenerator
to  X[n], A[n,n], B[n]
{

  B.cell(i) from() { return PetabricksRuntime::randDouble(0,100); }

  X.cell(i) from() { return 0; }

  to (A a) from()
  {
        int i,j;
        for (i=0; i<n; i++) {
                for (j=0; j<n; j++) {
                        if (i== j) {  a.cell(i,j) = 2; }
                        else if (i==j+1) { a.cell(i,j) = -1;}
                        else if (j==i+1) { a.cell(i,j) = -1;}
                        else { a.cell(i,j) = 0; }
                }
        }
  }

}

transform ResidualNorm
from OUT[n], X[n], A[n,n], B[n]
through AY[n], AX[n],E1,E2
to Accuracy
{
  to (Accuracy acc, AX ax, AY ay, E1 e1, E2 e2) from(B b, A a, X in, OUT out)
  {
        int i;
        ElementT error;
        demv(ax,a,in);
        demv(ay,a,out);
        ComputeError(e1,ax,b);
```

```
        ComputeError(e2,ay,b);
        if (e2 == 0) {
            acc = 10;
        }
        else {
            acc=log10(e1/e2);
        }
  }
}

#endif // POISSONPRECOND_PBCC
```

THIS PAGE INTENTIONALLY LEFT BLANK

Appendix B

Matlab Code

B.1 rmbump.m

```
xyz=load('94w_C2_e.xyz');
x=xyz(:,1);y=xyz(:,2);z=xyz(:,3);
x=reshape(x,94,[]);y=reshape(y,94,[]);z=reshape(z,94,[]);
N=size(xyz,1)/94;
kmax=N/10;
kmax=kmax-mod(kmax,4)+4;

zs=z(1,:);
dx=diff(x(1,:)); dy=diff(y(1,:));
ds=sqrt(dx.^2+dy.^2);
t=[0 cumsum(ds)];
global t
global zs

figure,
plot(t,zs)
xlabel('s')
ylabel('z')
title('profile')

k=16;
```

```
kindex=1;
while (k<kmax)
    sigma=k/4;
    vx=[-k:k];
    v=1./sqrt(2*pi)/sigma*((vx.^2-sigma^2)/sigma^4).*exp(-vx.^2/2/sigma^2) ;
    v=v/abs(sum(v));
    w=conv(zs,v);
    k=(length(w)-length(zs))/2;

    lm=(length(v)-1)/2;
    zs2=[zs(1)*ones(1,lm) zs zs(end)*ones(1,lm)];
    w=conv(zs2,v);
    ws=w(2*lm+1:end-2*lm);

    ws2d=diff(ws);
    zsd=diff(zs);
    ws3=ws.*([0 ws2d]<0).*([ws2d 0]>0);  %local min of LoG
    index=find(abs(ws3)>0);

    zs2=zs;
    t2=t;
    len=sigma*ones(1,length(index));
    len(1)=sigma/2; len(end)=sigma/2;

    for i=1:length(index)
        i1=max(index(i)-len(i),2);
        i2=min(index(i)+len(i),N-1);
        t2=t;
        zs2(i1:i2)=[];
        t2(i1:i2)=[];
        zs2=spline(t2,zs2,t);
    end

     figure
     plot(t,zs)
     hold on
```

```
     plot(t,zs2,'r')
     title('profile')
     xlabel('s')
     ylabel('z')
     legend('original profile', 'bump removed')

    bumps=z-zs2;  % get the bumps
     figure, plot(t,bumps), title('bumps')
     xlabel('s')
     ylabel('z')
    numofneg(kindex)=sum(bumps(index)<0);
    numofpos(kindex)=sum(bumps(index)>0);
    kvar(kindex)=var(bumps(index));
    krange(kindex)=max(bumps(index))-min(bumps(index));
    kvalue(kindex)=k;
    k=k+8;
    kindex=kindex+1;
end

numofbumps=numofneg+numofpos;
kop=find((diff(numofneg)>=0).*(diff(numofpos)>=0))+1;
koptimal=kvalue(find(kvar==min(kvar(kop))))
```

B.2 filternoise.m

```
xyz=load('94w_C1_b.xyz');
x=xyz(:,1);y=xyz(:,2);z=xyz(:,3);
x=reshape(x,94,[]);y=reshape(y,94,[]);z=reshape(z,94,[]);
zs=z;
for i=1:94
    zs1=zs(i,:);
    k=length(zs(i,:))/100*2;
    zs(i,:)=gaussfilter(zs1,k);
end

n2=zs-z;
```

```
noise=n2(:);
dx=diff(x(i,:)); dy=diff(y(i,:));
ds=sqrt(dx.^2+dy.^2);
t=[0 cumsum(ds)];

figure,
plot(t,zs1)
xlabel('s')
ylabel('z')
title('noisy data')

hold on
plot(t,zs(i,:),'r')
xlabel('s')
ylabel('z')
title('smoothed data')

figure, qqplot(noise)
figure, hist(noise,100)

%Compare with Normal
A=randn(size(noise,1),1);
A=A-mean(A)+mean(noise); A=A*std(noise);
[NormalX,NormalY]=hist(A,100);
hold on
plot(NormalY, NormalX, 'g')

mean(noise), var(noise), skewness(noise), kurtosis(noise)
```

B.3 gaussfilter.m

```
function fx=gaussfilter(x,k)
    sigma=k/4;
    vx=[-k:k];
    v=1./sqrt(2*pi)/sigma*exp(-vx.^2/2/sigma^2);
    lm=(length(v)-1)/2;
```

```
    x2=[x(1)*ones(1,lm) x x(end)*ones(1,lm)];
    w=conv(x2,v);
    fx=w(2*lm+1:end-2*lm);
end
```

B.4 roughness.m

```
function mu=roughness(data)

xyz=load(data);
x=xyz(:,1);y=xyz(:,2);z=xyz(:,3);
x=reshape(x,94,[]);y=reshape(y,94,[]);z=reshape(z,94,[]);


mu=svdratio(x,y,z);
end

function sr=svdratio(x,y,z)

dx=diff(x,1,2); dy=diff(y,1,2); dz=diff(z,1,2);
ds=sqrt(dx.^2+dy.^2);
df1=dz./ds;

[Ua Sa Va]=svd(df1,'econ');
Sda=diag(Sa);
sr=Sda(1).^2/sum(Sda.^2);

end
```

THIS PAGE INTENTIONALLY LEFT BLANK

Appendix C

Julia Code

C.1 randmatrixtest.j

```
function randmatrixtest()
    t=200000
    n=20

    v =
    @pfor (+) i=1:t begin
        a = 2*float64(rand(n,n)<.5) - 1
        (q,r) = qr(randn(n,n))
        a = q*a
        (a[1,1]*a[2,2])^2 - 1
    end
    v/t
end
```

C.2 roughness.j

```
function roughness()
    f=open("94w_C1_b.bin")
    data=read(f,Array(Float64,582800,3))
    x=data[:,1];
    y=data[:,2];
```

```
    z=data[:,3];
    x=reshape(x,94,6200);
    y=reshape(y,94,6200);
    z=reshape(z,94,6200);
    dx=diff(x,2);
    dy=diff(y,2);
    dz=diff(z,2);
    ds=sqrt(dx.^2+dy.^2);
    df=dz./ds;
    S=svd(df);
    sigma=diag(S[2]);
    sigma[1]*sigma[1]/sum(sigma.*sigma)
end
```

Bibliography

[1] *Moore's law: Made real by intel innovations.* http://www.intel.com/technology/mooreslaw/index.htm.

[2] A. Ali, L. Johnsson, and J. Subhlok, *Scheduling FFT computation on SMP and multicore systems*, in Proceedings of the ACM/IEEE Conference on Supercomputing, New York, NY, USA, 2007, ACM, pp. 293–301.

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

[4] B. S. Andersen, J. Waśniewski, and F. G. Gustavson, *A recursive formulation of cholesky factorization of a matrix in packed storage*, ACM Trans. Math. Softw., 27 (2001), pp. 214–244.

[5] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen, *LAPACK Users' Guide*, Society for Industrial and Applied Mathematics, Philadelphia, PA, third ed., 1999.

[6] J. Ansel, C. Chan, Y. L. Wong, M. Olszewski, Q. Zhao, A. Edelman, and S. Amarasinghe, *Petabricks: A language and compiler for algorithmic choice*, in ACM SIGPLAN Conference on Programming Language Design and Implementation, Dublin, Ireland, Jun 2009.

[7] J. Ansel, Y. L. Wong, C. Chan, M. Olszewski, A. Edelman, and S. Amarasinghe, *Language and compiler support for auto-tuning variable-accuracy algorithms*, in CGO, Chamonix, France, Apr 2011.

[8] D. Arthur and S. Vassilvitskii, *k-means++: the advantages of careful seeding*, in ACM-SIAM Symposium on Discrete Algorithms, January 2007.

[9] G. Ballard, J. Demmel, and I. Dumitriu, *Minimizing communication for eigenproblems and the singular value decomposition*, tech. rep., EECS Department, University of California, Berkeley, Feb 2011.

[10] G. Ballard, J. Demmel, and A. Gearhart, *Communication bounds for heterogeneous architectures*, Tech. Rep. UCB/EECS-2011-13, EECS Department, University of California, Berkeley, Feb 2011.

[11] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. V. der Vorst, *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, 2nd Edition*, SIAM, Philadelphia, PA, 1994.

[12] J. Bezanson, S. Karpinski, and V. Shah, *The Julia Programming Language.* User Manual for Julia, Jan. 2011.

[13] J. Bilmes, K. Asanovic, C.-W. Chin, and J. Demmel, *Optimizing matrix multiply using PHiPAC: a portable, high-performance, ANSI C coding methodology*, in Proceedings of the ACM/IEEE Conference on Supercomputing, New York, NY, USA, 1997, ACM, pp. 340–347.

[14] C. H. Bischof and C. V. Loan, *The wy representation for products of householder matrices.*, in PPSC'85, 1985, pp. 2–13.

[15] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley, *ScaLAPACK Users' Guide*, Society for Industrial and Applied Mathematics, Philadelphia, PA, 1997.

[16] L. S. Blackford, J. Demmel, J. Dongarra, I. Duff, S. Hammarling, G. Henry, M. Heroux, L. Kaufman, A. Lumsdaine, A. Petitet, R. Pozo, K. Remington, and R. C. Whaley, *An updated set of basic linear algebra subprograms (blas)*, ACM Trans. Math. Softw., 28 (2002), pp. 135–151.

[17] J. J. Buoni, P. A. Farrell, and A. Ruttan, *Algorithms for lu decomposition on a shared memory multiprocessor*, Parallel Computing, 19 (1993), pp. 925 – 937.

[18] C. Chan, J. Ansel, Y. L. Wong, S. Amarasinghe, and A. Edelman, *Autotuning multigrid with petabricks*, in ACM/IEEE Conference on Supercomputing, Portland, OR, Nov 2009.

[19] O. Y. K. Chen and A. B, *A comparison of pivoting strategies for the direct lu factorization.*

[20] J. Choi, J. J. Dongarra, and D. W. Walker, *The design of a parallel dense linear algebra software library: Reduction to hessenberg, tridiagonal, and bidiagonal form*, Tech. Rep. 92, LAPACK Working Note, Feb. 1995.

[21] R. Choy, A. Edelman, J. R. Gilbert, V. Shah, and D. Cheng, *Star-p: High productivity parallel computing*, in In 8th Annual Workshop on High-Performance Embedded Computing (HPEC 04), 2004.

[22] E. CHU AND A. GEORGE, *Gaussian elimination with partial pivoting and load balancing on a multiprocessor*, Parallel Computing, 5 (1987), pp. 65 – 74.

[23] C. W. CRYER, *Pivot size in Gaussian elimination*, 12 (1968), pp. 335–345.

[24] T. A. DAVIS, *Direct Methods for Sparse Linear Systems*, Society for Industrial and Applied Mathematics, Philadephia, PA, 2006.

[25] C. DE BLAS CARTON, A. GONZALEZ-ESCRIBANO, AND D. R. LLANOS, *Effortless and efficient distributed data-partitioning in linear algebra*, High Performance Computing and Communications, 10th IEEE International Conference on, 0 (2010), pp. 89–97.

[26] W. F. DE LA VEGA AND G. S. LUEKER, *Bin packing can be solved within 1+epsilon in linear time*, Combinatorica, 1 (1981), pp. 349–355.

[27] J. DEMMEL, L. GRIGORI, M. F. HOEMMEN, AND J. LANGOU, *Communication-optimal parallel and sequential qr and lu factorizations*, Tech. Rep. UCB/EECS-2008-89, EECS Department, University of California, Berkeley, Aug 2008. Current version available in the ArXiv at http://arxiv.org/pdf/0809.0101.

[28] J. DEMMEL, L. GRIGORI, AND H. XIANG, *Calu: A communication optimal lu factorization algorithm*, Tech. Rep. UCB/EECS-2010-29, EECS Department, University of California, Berkeley, Mar 2010.

[29] J. W. DEMMEL, *Applied Numerical Linear Algebra*, SIAM, August 1997.

[30] J. W. DEMMEL, S. C. EISENSTAT, J. R. GILBERT, X. S. LI, AND J. W. H. LIU, *A supernodal approach to sparse partial pivoting*, SIAM J. Matrix Analysis and Applications, 20 (1999), pp. 720–755.

[31] J. W. DEMMEL, J. R. GILBERT, AND X. S. LI, *An asynchronous parallel supernodal algorithm for sparse gaussian elimination*, SIAM J. Matrix Analysis and Applications, 20 (1999), pp. 915–952.

[32] J. W. DEMMEL, M. T. HEATH, AND H. A. VAN DER VORST, *Parallel numerical linear algebra*, in Society for Industrial and Applied Mathematics, SIAM, 1997.

[33] I. S. DHILLON, *A New $O(n^2)$ Algorithm for the Symmetric Tridiagonal Eigenvalue/Eigenvector Problem*, PhD thesis, EECS Department, University of California, Berkeley, Oct 1997.

[34] I. S. DHILLON AND B. N. PARLETT, *Multiple representations to compute orthogonal eigenvectors of symmetric tridiagonal matrices*, Linear Algebra and Appl, 387 (2004), pp. 1–28.

[35] J. J. DONGARRA, J. DU CROZ, S. HAMMARLING, AND I. S. DUFF, *A set of level 3 basic linear algebra subprograms*, ACM Trans. Math. Softw., 16 (1990), pp. 1–17.

[36] J. J. DONGARRA, J. DU CROZ, S. HAMMARLING, AND R. J. HANSON, *An extended set of fortran basic linear algebra subprograms*, ACM Trans. Math. Softw., 14 (1988), pp. 1–17.

[37] J. J. DONGARRA, D. C. SORENSEN, AND S. J. HAMMARLING, *Block reduction of matrices to condensed forms for eigenvalue computations*, Journal of Computational and Applied Mathematics, 27 (1989), pp. 215 – 227. Special Issue on Parallel Algorithms for Numerical Linear Algebra.

[38] A. EDELMAN, *The complete pivoting conjecture for gaussian elimination is false*, The Mathematica Journal, 2 (1992), pp. 58–61.

[39] E. ELMROTH AND F. G. GUSTAVSON, *Applying recursion to serial and parallel qr factorization leads to better performance*, IBM Journal of Research and Development, 44 (2000), pp. 605 –624.

[40] M. FRIGO AND S. G. JOHNSON, *FFTW: An adaptive software architecture for the FFT*, in Proceedings of the IEEE International Conference on Acoustics Speech and Signal Processing, vol. 3, IEEE, 1998, pp. 1381–1384.

[41] M. FRIGO AND S. G. JOHNSON, *The design and implementation of FFTW3*, Proceedings of the IEEE, 93 (2005), pp. 216–231. Special issue on "Program Generation, Optimization, and Platform Adaptation".

[42] M. FRIGO, C. E. LEISERSON, H. PROKOP, AND S. RAMACHANDRAN, *Cache-oblivious algorithms*, in Proceedings of the 40th Annual Symposium on Foundations of Computer Science, FOCS '99, Washington, DC, USA, 1999, IEEE Computer Society, pp. 285–.

[43] M. FRIGO, C. E. LEISERSON, AND K. H. RANDALL, *The implementation of the Cilk-5 multithreaded language*, in Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, Montreal, Quebec, Canada, Jun 1998, pp. 212–223. Proceedings published ACM SIGPLAN Notices, Vol. 33, No. 5, May, 1998.

[44] G. A. GEIST AND C. H. ROMINE, *lu factorization algorithms on distributed-memory multiprocessor architectures*, SIAM Journal on Scientific and Statistical Computing, 9 (1988), pp. 639–649.

[45] A. GEORGE, J. W. H. LIU, AND E. NG, *Communication results for parallel sparse cholesky factorization on a hypercube*, Parallel Computing, 10 (1989), pp. 287 – 298.

[46] N. GOULD, *On growth in gaussian elimination with complete pivoting*, SIAM Journal on Matrix Analysis and Applications, 12 (1991), pp. 354–361.

[47] C. GOUTTE, P. TOFT, E. ROSTRUP, F. . NIELSEN, AND L. K. HANSEN, *On clustering fmri time series*, NeuroImage, 9 (1999), pp. 298 – 310.

[48] L. GRIGORI, E. G. BOMAN, S. DONFACK, AND T. A. DAVIS, *Hypergraph-based unsymmetric nested dissection ordering for sparse lu factorization*, SIAM Journal on Scientific Computing, 32 (2010), pp. 3426–3446.

[49] L. GRIGORI AND X. S. LI, *A new scheduling algorithm for parallel sparse lu factorization with static pivoting*, in Supercomputing, ACM/IEEE 2002 Conference, 2002, p. 25.

[50] M. GU AND S. C. EISENSTAT, *A divide-and-conquer algorithm for the bidiagonal svd*, SIAM Journal on Matrix Analysis and Applications, 16 (1995), pp. 79–92.

[51] F. G. GUSTAVSON, *Recursion leads to automatic variable blocking for dense linear-algebra algorithms*, IBM J. Res. Dev., 41 (1997), pp. 737–756.

[52] J. A. HARTIGAN, *Clustering algorithms*, Wiley New York, 1975.

[53] P. HÉNON, P. RAMET, AND J. ROMAN, *PaStiX: A Parallel Direct Solver for Sparse SPD Matrices based on Efficient Static Scheduling and Memory Managment*, in Tenth SIAM Conference on Parallel Processing for Scientific Computing, Portsmouth United States, 2001.

[54] P. HUSBANDS AND K. YELICK, *Multi-threading and one-sided communication in parallel lu factorization*, SC Conference, 0 (2007), pp. 1–10.

[55] E. IM AND K. YELICK, *Optimizing sparse matrix computations for register reuse in SPARSITY*, in Proceedings of the International Conference on Computational Science, Springer, 2001, pp. 127–136.

[56] M. INABA, N. KATOH, AND H. IMAI, *Applications of weighted voronoi diagrams and randomization to variance-based k-clustering: (extended abstract)*, in Proceedings of the tenth annual symposium on Computational geometry, SCG '94, New York, NY, USA, 1994, ACM, pp. 332–339.

[57] E. R. JESSUP AND D. C. SORENSEN, *A parallel algorithm for computing the singular value decomposition of a matrix*, SIAM J. Matrix Anal. Appl., 15 (1994), pp. 530–548.

[58] J. KEPNER, *Parallel MATLAB for Multicore and Multinode Systems*, SIAM Press, 2009.

[59] A. B. LAWSON AND D. G. T. DENISON, *Spatial Cluster Modelling*, Chapman & Hall CRC, London, 2002.

[60] C. L. LAWSON, R. J. HANSON, D. R. KINCAID, AND F. T. KROGH, *Basic linear algebra subprograms for fortran usage*, ACM Trans. Math. Softw., 5 (1979), pp. 308–323.

[61] R. Li, *Solving secular equations stably and efficiently*, tech. rep., University of California at Berkeley, Berkeley, CA, USA, 1993.

[62] X. Li, M. J. Garzaran, and D. Padua, *A dynamically tuned sorting library*, in Proceedings of the International Symposium on Code Generation and Optimization, March 2004, pp. 111–122.

[63] X. Li, M. J. Garzarn, and D. Padua, *Optimizing sorting with genetic algorithms*, in Proceedings of the International Symposium on Code Generation and Optimization, IEEE Computer Society, 2005, pp. 99–110.

[64] X. S. Li, *An overview of superlu: Algorithms, implementation, and user interface*, ACM Trans. Math. Softw., 31 (2005), pp. 302–325.

[65] X. S. Li and J. W. Demmel, *SuperLU_DIST: A scalable distributed-memory sparse direct solver for unsymmetric linear systems*, ACM Trans. Mathematical Software, 29 (2003), pp. 110–140.

[66] J. W. H. Liu, *The role of elimination trees in sparse factorization*, SIAM J. Matrix Anal. Appl., 11 (1990), pp. 134–172.

[67] M. Mahajan, P. Nimbhorkar, and K. Varadarajan, *The planar k-means problem is np-hard*, in WALCOM: Algorithms and Computation, S. Das and R. Uehara, eds., vol. 5431 of Lecture Notes in Computer Science, Springer Berlin / Heidelberg, 2009, pp. 274–285. 10.1007/978-3-642-00202-1_24.

[68] K. V. Mardia, J. Kent, and B. J.M., *Multivariate Analysis*, Academic Press, 1980.

[69] V. Menon and K. Pingali, *Look left, look right, look left again: An application of fractal symbolic analysis to linear algebra code restructuring*, Int. J. Parallel Comput, 32 (2003), p. 2004.

[70] G. Meurant, *The Lanczos and Conjugate Gradient Algorithms: From Theory to Finite Precision Computations (Software, Environments, and Tools)*, SIAM, 2006.

[71] G. E. Moore, *Cramming more components onto integrated circuits*, Electronics, 38 (1965), pp. 114–117.

[72] A. Okabe, B. Boots, and K. Sugihara, *Spatial tessellations: concepts and applications of Voronoi diagrams*, John Wiley & Sons, Inc., New York, NY, USA, 1992.

[73] M. Olszewski and M. Voss, *Install-time system for automatic generation of optimized parallel sorting algorithms*, in Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications, 2004, pp. 17–23.

[74] B. N. PARLETT AND I. S. DHILLON, *Relatively robust representations of symmetric tridiagonals*, Linear Algebra and Appl, 309 (1999), pp. 121–151.

[75] A. POTHEN AND C. SUN, *A mapping algorithm for parallel sparse cholesky factorization*, SIAM Journal on Scientific Computing, 14 (1993), pp. 1253–1257.

[76] M. PÜSCHEL, J. M. F. MOURA, B. SINGER, J. XIONG, J. R. JOHNSON, D. A. PADUA, M. M. VELOSO, AND R. W. JOHNSON, *Spiral: A generator for platform-adapted libraries of signal processing alogorithms*, IJHPCA, 18 (2004), pp. 21–45.

[77] G. QUINTANA-ORTI, E. S. QUINTANA-ORTI, E. CHAN, AND ET AL., *Design and scheduling of an algorithm-by-blocks for lu factorization on multithreaded architectures*, tech. rep., 2007.

[78] R. H. RAND, *Computer algebra in applied mathematics: an introduction to MACSYMA*, no. 94 in Research notes in mathematics, 1984.

[79] M. ROUSHDY, *Comparative study of edge detection algorithms applying on the grayscale noisy image using morphological filter*, ICGST International Journal on Graphics, Vision and Image Processing, 06 (2007), pp. 17–23.

[80] P. SISKA AND I. HUNG, *Advanced digital terrain analysis using rougness-dissectivity parameters in gis*, in Proceedings of the 2004 ESRI International User Conference, San Diego, California, USA, 2004.

[81] J. STROBL, *Segmentation-based terrain classification*, in Advances in Digital Terrain Analysis, Q. Zhou, B. Lees, and G.-a. Tang, eds., Lecture Notes in Geoinformation and Cartography, Springer Berlin Heidelberg, 2008, pp. 125–139. 10.1007/978-3-540-77800-4_7.

[82] C. A. SUGAR, GARETH, AND M. JAMES, *Finding the number of clusters in a data set: An information theoretic approach*, Journal of the American Statistical Association, 98 (2003), pp. 750–763.

[83] S. THRUN, M. MONTEMERLO, AND A. ARON, *Probabilistic terrain analysis for high-speed desert driving*, in Proceedings of Robotics: Science and Systems, Philadelphia, USA, August 2006.

[84] S. TOLEDO, *Locality of reference in lu decomposition with partial pivoting*, SIAM JOURNAL ON MATRIX ANALYSIS AND APPLICATIONS, 18 (1997), pp. 1065–1081.

[85] L. N. TREFETHEN AND D. BAU, *Numerical Linear Algebra*, SIAM: Society for Industrial and Applied Mathematics, June 1997.

[86] V. VOLKOV AND J. W. DEMMEL, *Benchmarking gpus to tune dense linear algebra*, in Proceedings of the 2008 ACM/IEEE conference on Supercomputing, SC '08, Piscataway, NJ, USA, 2008, IEEE Press, pp. 31:1–31:11.

[87] R. VUDUC, J. W. DEMMEL, AND K. A. YELICK, *OSKI: A library of automatically tuned sparse matrix kernels*, in Proceedings of the Scientific Discovery through Advanced Computing Conference, Journal of Physics: Conference Series, San Francisco, CA, USA, June 2005, Institute of Physics Publishing.

[88] R. C. WHALEY AND J. J. DONGARRA, *Automatically tuned linear algebra software*, in ACM/IEEE Conference on Supercomputing, Washington, DC, USA, 1998, IEEE Computer Society, pp. 1–27.

[89] R. C. WHALEY AND A. PETITET, *Minimizing development and maintenance costs in supporting persistently optimized BLAS*, Software: Practice and Experience, 35 (2005), pp. 101–121.

[90] J. H. WILKINSON, *Error analysis of direct methods of matrix inversion*, J. ACM, 8 (1961), pp. 281–330.

[91] P. R. WILLEMS, B. LANG, AND C. VOEMEL, *Computing the bidiagonal svd using multiple relatively robust representations*, SIAM J. Matrix Anal. Appl., 28 (2006), pp. 907–926.

[92] J. P. WILSON AND J. C. GALLANT, *Terrain Analysis: Principles and Applications*, Wiley, New York, 2000.

[93] D. YOUNG, *Iterative solution of large linear systems*, Dover Publications, 2003.

[94] L. A. ZADEH, *Fuzzy logic, neural networks, and soft computing*, Commun. ACM, 37 (1994), pp. 77–84.

[95] L. ZHILIN, *Multi-scale digital terrain modelling and analysis*, in Advances in Digital Terrain Analysis, Q. Zhou, B. Lees, and G.-a. Tang, eds., Lecture Notes in Geoinformation and Cartography, Springer Berlin Heidelberg, 2008, pp. 59–83. 10.1007/978-3-540-77800-4_4.