# Performance Analysis and Modeling of Task-Based Runtimes

A Dissertation Presented for the

Doctor of Philosophy

Degree

The University of Tennessee, Knoxville

Blake Andrew Haugen

May 2016

© by Blake Andrew Haugen, 2016

All Rights Reserved.

*This dissertation is dedicated to my family:*

*My wife, Rebekah, for her love and support,*

*My parents, John and Marsha, who instilled a lifelong love of learning, and*

*My brother, Mark, for bringing humor to every situation.*

# Acknowledgements

I would like to thank my advisor, Dr. Jack Dongarra, for his patience and support throughout my time at the University of Tennessee. He and the ICL team made my graduate studies a period of incredible learning including many memories that will last a lifetime.

Thank you to Dr. Chad Steed, Dr. Vasilios Alexiades, and Dr. Gregory Peterson for serving on my doctoral committee and providing feedback on my dissertation and research.

I will be forever grateful to my colleagues and mentors at the ICL for their patience and guidance throughout my time here. I specifically would like to thank Dr. Jakub Kurzak, Dr. Piotr Luszczek, Dr. Asim YarKhan, Dr. Mark Gates, Dr. Anthony Danalis, Dr. Hartwig Anzt, Dr. Ichitaro Yamazaki, and Dr. George Bosilca for the many conversations that have profoundly challenged me and shaped my research.

# Abstract

The shift toward multicore processors has transformed the software and hardware landscape in the last decade. As a result, software developers must adopt parallelism in order to efficiently make use of multicore CPUs. Task-based scheduling has emerged as one method to reduce the complexity of parallel computing. Although task-based scheduling has been around for many years, the inclusion of task dependencies in OpenMP 4.0 suggests the paradigm will be around for the foreseeable future.

While task-based schedulers simplify the process of parallel software development, they can obfuscate the performance characteristics of the execution of an algorithm. Additionally, they can create a challenge for users to analyze the performance of their software and tune algorithmic parameters accordingly.

We will present the basic principles of task-based runtimes as well as two new tools developed to assist engineers developing these runtimes and users employing them to parallelize their workloads. The first is a tool allowing users to simulate the execution of their algorithm. The second is an extension to the common execution trace which includes information about task dependencies.

# Table of Contents

**1 Introduction** **1**

1.1 Motivation . . . 1

1.2 Introduction . . . 2

1.3 Thesis Statement and Original Contributions . . . 3

1.4 Outline of the Dissertation . . . 3

**2 Background** **5**

2.1 Hardware Landscape . . . 5

2.1.1 Multicore CPUs . . . 6

2.1.2 SMP and NUMA . . . 8

2.1.3 Dynamic Frequency Scaling and Power Capping . . . 9

2.1.4 Accelerators and Hybrid Computing . . . 10

2.2 Software Landscape . . . 12

2.2.1 Task-Based Runtimes . . . 12

2.3 Tile-Based Linear Algebra . . . 17

**3 Workload Simulation** **23**

3.1 Related Work . . . 23

3.2 Discrete Event Simulation . . . 26

3.2.1 Simulation Methodology . . . 26

3.2.2 Tracing . . . 28

3.2.3 Model of Kernel Executed inside a Task . . . 29

3.2.4 Task Execution Queue . . . . . . . . . . . . . . . . . . . . . . 39
3.2.5 Simulation Task Function . . . . . . . . . . . . . . . . . . . . 39
3.2.6 Scheduling Race Condition . . . . . . . . . . . . . . . . . . . 40
3.3 Spin Simulations . . . . . . . . . . . . . . . . . . . . . . . . . . . 43

**4 Simulation Results and Applications 45**
4.1 Comparison of Schedulers . . . . . . . . . . . . . . . . . . . . . . . 47
4.2 Simulation Scalability . . . . . . . . . . . . . . . . . . . . . . . . . 50
4.2.1 Varying Core Counts . . . . . . . . . . . . . . . . . . . . . . . 50
4.2.2 Varying Task Granularity . . . . . . . . . . . . . . . . . . . . 58
4.3 LU Simulation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 62
4.4 Intel Xeon Phi Cholesky Simulation . . . . . . . . . . . . . . . . . . 69
4.5 Reverse Trace Performance Modeling . . . . . . . . . . . . . . . . . 78
4.6 Kastors SparseLU Simulation . . . . . . . . . . . . . . . . . . . . . 81
4.7 Conclusions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 83

**5 Trace Visualization 85**
5.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 85
5.1.1 DAG . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 86
5.1.2 Trace . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 87
5.2 Visualization Design . . . . . . . . . . . . . . . . . . . . . . . . . . 89
5.2.1 Implementation . . . . . . . . . . . . . . . . . . . . . . . . . 93
5.3 Applications . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 94
5.4 Trace Visual Analytics System . . . . . . . . . . . . . . . . . . . . 100

**6 Conclusions 104**
6.1 Future Work . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 106

**Bibliography 107**

**Vita 117**

# List of Tables


3.1 DGEMM Cache Scenarios . . . . . . . . . . . . . . . . . . . . . . . . 33

4.1 Floating Point Operations . . . . . . . . . . . . . . . . . . . . . . . . 46

4.2 DGEQRT task descriptive statistics (25 Data Points) . . . . . . . . . 53

4.3 DGEQRT t-test p-values . . . . . . . . . . . . . . . . . . . . . . . . . 53

4.4 DORMQR task descriptive statistics (300 Data Points) . . . . . . . . 54

4.5 DORMQR KS test p-values . . . . . . . . . . . . . . . . . . . . . . . 54

4.6 DTSQRT task descriptive statistics (300 Data Points) . . . . . . . . . 55

4.7 DTSQRT KS test p-values . . . . . . . . . . . . . . . . . . . . . . . . 55

4.8 DTSMQR task descriptive statistics (4900 Data Points) . . . . . . . . 56

4.9 DTSMQR KS test p-values . . . . . . . . . . . . . . . . . . . . . . . . 56

4.10 Cholesky Task Flop Counts . . . . . . . . . . . . . . . . . . . . . . . 70

4.11 Cholesky Task Performance (Intel Xeon Phi) . . . . . . . . . . . . . . 70

4.12 Cholesky Task Performance (Intel Haswell) . . . . . . . . . . . . . . . 72

4.13 Improved DSYRK & DPOTRF Kernel Models . . . . . . . . . . . . . 73

# List of Figures


2.1 CPU DB Processor Clock Rate . . . . . . 6

2.2 CPU DB Number of Cores . . . . . . 7

2.3 Top 500 Total Cores . . . . . . 8

2.4 Tile Layout . . . . . . 19

2.5 Pseudocode for Tile QR Factorization . . . . . . 21

2.6 OpenMP Cholesky Implementation . . . . . . 22

3.1 DGEMM Serial Benchmark . . . . . . 35

3.2 DGEMM Serial Benchmark . . . . . . 35

3.3 DGEMM NUMA Benchmark . . . . . . 36

3.4 DGEMM NUMA Benchmark . . . . . . 36

3.5 DGEMM Threaded Benchmark . . . . . . 37

3.6 DGEMM Threaded Benchmark . . . . . . 37

3.7 Simulation Race Condition . . . . . . 42

3.8 Simulation Race Condition Small Error . . . . . . 42

3.9 Simulation Race Condition Large Error . . . . . . 42

4.1 QUARK Simulation Results . . . . . . 48

4.2 StarPU Simulation Results . . . . . . 48

4.3 OmpSs Simulation Results . . . . . . 49

4.4 OpenMP Simulation Results . . . . . . 49

4.5 DGEQRT KDE . . . . . . 53

4.6 DORMQR KDE . . . . . . . . . . . . . . . . . . . . . . . . . . . . 54
4.7 DTSQRT KDE . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 55
4.8 DTSMQR KDE . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 56
4.9 QR Simulation Scalability . . . . . . . . . . . . . . . . . . . . . . . 57
4.10 Cholesky 48 Core Discrete Event Simulation . . . . . . . . . . . . . . 60
4.11 Cholesky 48 Core Spin Simulation . . . . . . . . . . . . . . . . . . . 60
4.12 Cholesky 12 Core Simulation . . . . . . . . . . . . . . . . . . . . . . 61
4.13 Cholesky 12 Core Spin Simulation . . . . . . . . . . . . . . . . . . . 61
4.14 LU Panel Timing . . . . . . . . . . . . . . . . . . . . . . . . . . . . 65
4.15 LU Simulation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 65
4.16 Real LU Factorization N=5000 NB=200 . . . . . . . . . . . . . . . . . 66
4.17 Simulated LU Factorization N=5000 NB=200 . . . . . . . . . . . . . . 66
4.18 Real LU Factorization N=6400 NB=128 . . . . . . . . . . . . . . . . . 67
4.19 Simulated LU Factorization N=6400 NB=200 . . . . . . . . . . . . . . 67
4.20 Real LU Factorization N=10000 NB=200 . . . . . . . . . . . . . . . . 68
4.21 Simulated LU Factorization N=10000 NB=200 . . . . . . . . . . . . . 68
4.22 Simulated Cholesky Factorization (Accelerated Tasks) . . . . . . . . . 74
4.23 Real Cholesky Factorization . . . . . . . . . . . . . . . . . . . . . . 75
4.24 Simulated Cholesky Factorization . . . . . . . . . . . . . . . . . . . 75
4.25 Simulated Cholesky Factorization (Accelerated DSRYK) . . . . . . . 76
4.26 Simulated Cholesky Factorization (Accelerated DPOTRF) . . . . . . 76
4.27 Simulated Cholesky Factorization (Accelerated DPOTRF & DSYRK) 76
4.28 Simulated Intel Hybrid Architecture . . . . . . . . . . . . . . . . . . 77
4.29 Cholesky Reverse Trace Performance Modeling . . . . . . . . . . . . . 80
4.30 QR Reverse Trace Performance Modeling . . . . . . . . . . . . . . . . 80
4.31 SparseLU Simulation . . . . . . . . . . . . . . . . . . . . . . . . . . 82
4.32 SparseLU Simulation . . . . . . . . . . . . . . . . . . . . . . . . . . 82

5.1 Example QR DAG . . . . . . . . . . . . . . . . . . . . . . . . . . . 87

5.2 Example QR Trace . . . . . . . . . . . . . . . . . . . . . . . . . . . 88
5.3 Complete QR Trace with DAG . . . . . . . . . . . . . . . . . . . . . 90
5.4 Interactive QR Trace with DAG . . . . . . . . . . . . . . . . . . . . . 91
5.5 LU Trace without Priorities . . . . . . . . . . . . . . . . . . . . . . . 95
5.6 Large LU DAG . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 97
5.7 Small LU DAG . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 98
5.8 LU Trace with Priorities . . . . . . . . . . . . . . . . . . . . . . . . 98
5.9 NUMA Performance Issue . . . . . . . . . . . . . . . . . . . . . . . . 99
5.10 Memory Analysis Trace . . . . . . . . . . . . . . . . . . . . . . . . . 101
5.11 Trace Filtering . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 101

# Chapter 1

# Introduction

## 1.1 Motivation

In the last decade the microchip industry has shifted to a multicore paradigm and consequently altered the path of software development. Until this time period, developers could expect their software to see performance improvements with each new generation of computing architecture because the clock frequency of the new chip would boost the performance. During this era, modifications to software were not necessary to increase performance. The frequency of new microprocessors stabilized while the number of cores began to increase. Developers now had to modify their software to make performance gains on new hardware [50]. Unfortunately, adding parallelism to software is often a non-trivial task.

A developer can develop parallel applications using primitive, low-level APIs such as POSIX threads (Pthreads) in a shared memory context or the Message Passing Interface (MPI) standard for distributed memory systems. While effective, these tools generally require expert level knowledge of parallel programming and their application.

A number of higher level parallel programming APIs have emerged in an effort to simplify the process of developing high performance parallel software. One of

the programming models that has emerged is a task-based paradigm in which the developer defines his/her computation as a series of tasks executed in parallel by a scheduler at runtime. While this model provides another layer of abstraction to simplify the development process, it also obscures many of the fine-grained details necessary to obtain optimal or near optimal performance. This work presents two new tools designed to give developers a greater understanding of these task-based schedulers.

## 1.2 Introduction

Task-based runtimes simplify the development process by inferring and respecting data dependencies based on developer defined data hazards. In order to use a task-based scheduler, developers must break their workloads into tasks and define the input and output parameters of each task. The scheduler then uses the order of these tasks and their input and output parameters to generate a Directed Acyclic Graph (DAG) of tasks which can be used to schedule the tasks in parallel while respecting all data dependencies. Chapter 2 will give a more extensive introduction to this programming model.

Many of the traditional linear algebra algorithms can be defined as a series of tasks perfectly suited for these task-based schedulers. The tile-based formulation of three common matrix factorizations (Cholesky, LU, and QR) will provide example applications for the remainder of this dissertation. These factorizations and their tile-based implementations will be presented in greater detail in Chapter 2.

The first tool is a simulation utility which can be used to provide insights that guide developers in the process of tuning their task-based applications and the second is an extension to the common execution trace visualization.

## 1.3 Thesis Statement and Original Contributions

The primary goal of this dissertation is to investigate the simulation of task-based runtimes in the context of multicore shared memory architectures. The issues of portability across schedulers and hardware, the accuracy of the performance predictions, and the usefulness to developers are addressed in this document.

The three primary contributions of this dissertation are as follows:

- A novel simulation framework for task-based runtimes. The framework is portable to many task-based schedulers and architectures while providing accurate performance predictions.
- An extension of task benchmarking and timing to multicore machines including extensions for NUMA architectures.
- A novel visualization extension that provides an interactive tool to explore trace and DAG visualizations simultaneously.

## 1.4 Outline of the Dissertation

This dissertation is organized as follows:

- **Chapter 2** introduces task-based scheduling and several of the utilities that employ this programming paradigm. Tile-based linear algebra will also be presented in order to provide details about many of the applications which are analyzed throughout this document.
- **Chapter 3** describes the simulation framework.
- **Chapter 4** presents several applications and a wide variety of performance results for the simulator.

- **Chapter 5** presents a novel trace visualization utility designed with extensions for task-based applications.
- **Chapter 6** concludes the dissertation and discusses possible future extensions to the work presented here.

# Chapter 2

# Background

Portions of this chapter are drawn from the following publications:

- Haugen, Blake, Jakub Kurzak, Asim YarKhan, Piotr Luszczek, and Jack Dongarra. "Parallel Simulation of Superscalar Scheduling." In the *43rd International Conference on Parallel Processing (ICPP), 2014*, pp. 121-130. IEEE, 2014.
- Haugen, Blake, Stephen Richmond, Jakub Kurzak, Chad A. Steed, and Jack Dongarra. "Visualizing Execution Traces with Task Dependencies." In Proceedings of the *2nd Workshop on Visual Performance Analysis*, p. 2. ACM, 2015.

I was responsible for the design and implementation of the software corresponding to each of these publications. In addition, I served as the primary author.

## 2.1 Hardware Landscape

Early generations of computing hardware were relatively simple and homogeneous compared with todays systems. In order to deal with power limitations and the desire for ever-increasing application performance, the high performance computing industry

has adopted an increasingly diverse set of complex architectures and technologies to provide the best performance possible.

### 2.1.1 Multicore CPUs

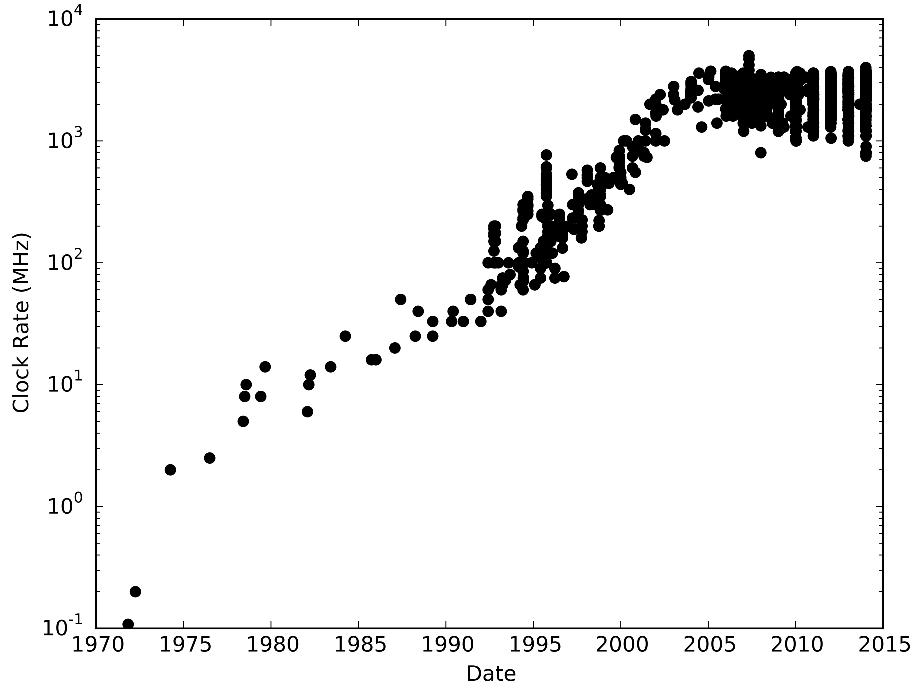



**Figure 2.1:** The CPU clock rate stagnates around 2005.

The last decade has ushered in a dramatic shift in computer architecture with the introduction and market saturation of multicore processors. Multicore processors have even spread from traditional high end computing platforms to mobile devices such as tablets and smart phones. Prior to the multicore shift, software developers could expect their applications to see significant performance increases with each new architecture. One of the primary reasons for this increase was the ever-increasing clock rate on each processor entering the market. This trend can be seen in Figure 2.1 based on data from the CPU DB data set [23] provided by researchers at Stanford University. The plot shows this ever-increasing clock rate stops fairly suddenly around 2005. Some of the newer processors even have a slower clock rate than older models.

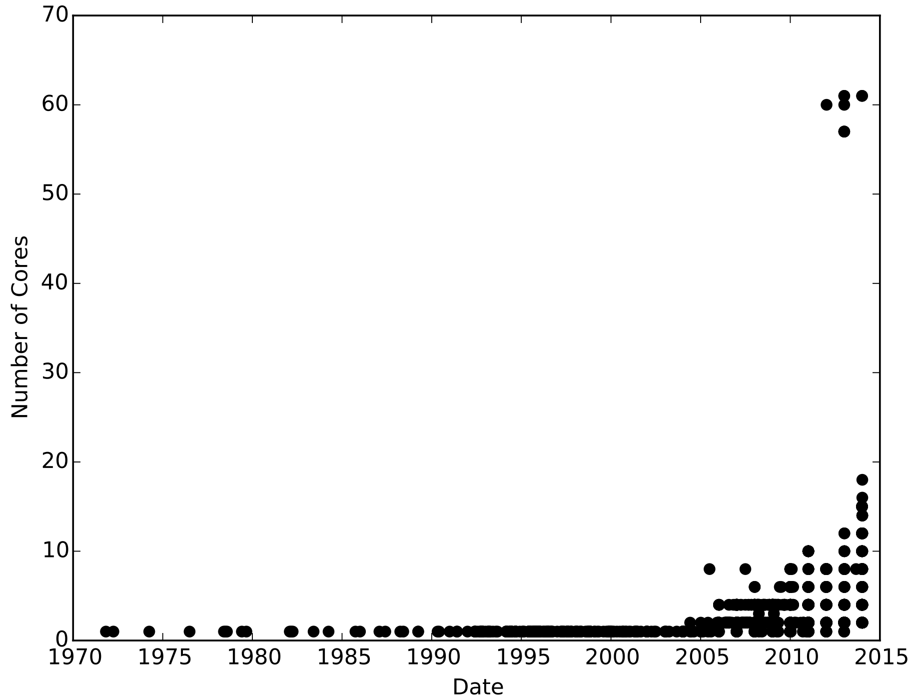



**Figure 2.2:** The number of cores in a processor begins to increase around 2005.

The CPU DB data set also contains information about the number of cores in each of the processors. This data is plotted in Figure 2.2. Around the same time the clock rate for new processors stagnates, the earliest multicore CPUs begin to emerge. The earliest multicore processors had two cores on the same die but they quickly released chips with many more cores. There are a few data points in the top right corner of Figure 2.2 that are between 57 and 61 cores. These data points correspond to the Intel Xeon Phi manycore architecture (to be discussed in greater depth later).

This trend toward an increasing number of cores can also be seen by examining the list of Top 500 supercomputers compiled every 6 months. Figure 2.3 shows a dramatic spike in the number of cores on the top system around 2005.

Perhaps the greatest impact of this multicore shift has been to the software development community. In the past they could run their old software on a new architecture and generally expect drastic performance improvement. In the move to a multicore architecture, however, developers must rewrite their software to make use

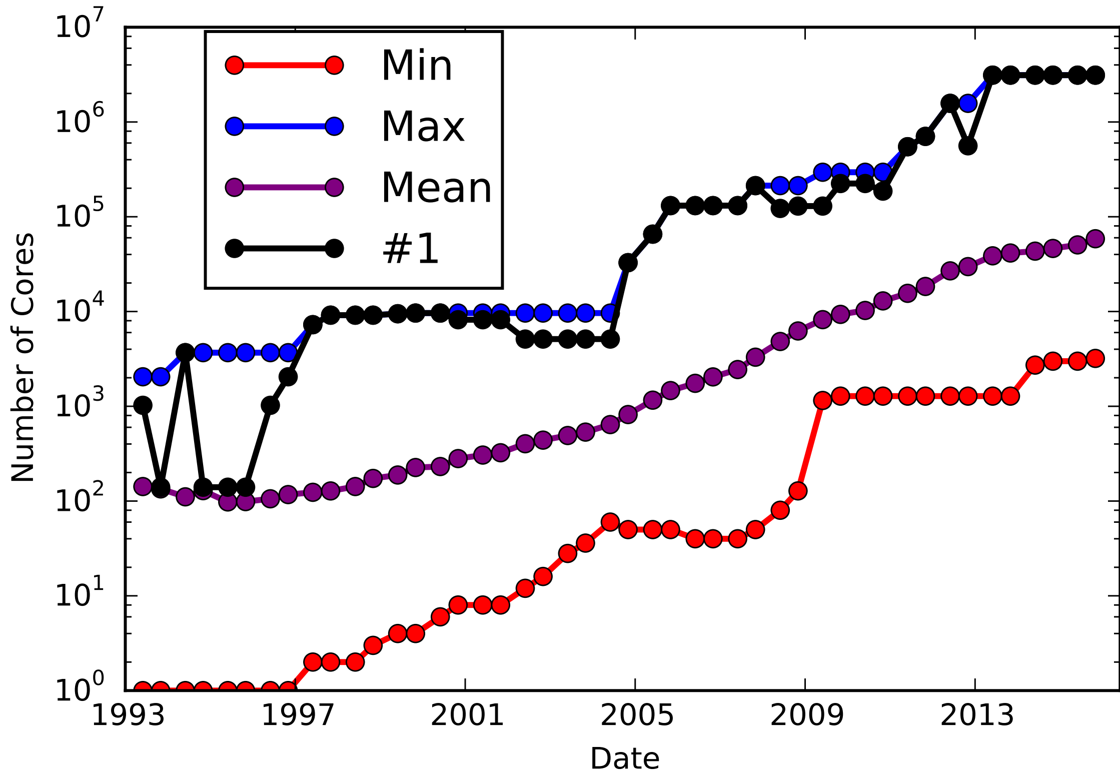



**Figure 2.3:** The number of cores on the Top 500 supercomputer list begins to dramatically accelerate around 2005.

of the parallelism that is available to them. Unfortunately, this is often a non-trivial task requiring a great deal of effort.

### 2.1.2 SMP and NUMA

Multicore shared memory systems are often broken into two classes which provide different performance characteristics for parallel applications.

The first type of multicore shared memory system is a Symmetric MultiProcessing (SMP) system. An SMP system is made up of multiple processors (cores or chips) connected to a single, shared main memory. This means the main memory is equally accessible by each processor in the system. As a result, it does not matter (assuming the data is not in the cache of another processor) where an application or function is executed because the processors all have the same access to the memory. Most of the

desktops and laptops produced today could be considered SMP systems where each core is a processor and all of the cores share a single main memory.

The other type of multicore shared memory system is a Non-Uniform Memory Access (NUMA) system. A NUMA system also has multiple processors but each processor has access to local memory and non-local memory. The process can access data on any of the NUMA "nodes" but it is faster to access local memory as opposed to non-local memory. This can increase memory throughput for some applications because the processors can each access its own local memory simultaneously. Unfortunately, this architecture can be harder to optimize performance because ideally a developer should execute an application or function on a processor that is "closest" to the memory where the data resides. The developer also has the ability to control where memory is allocated. The most common memory location policy is the "first touch" policy that says the data will be located on the memory closest to the processor that first touches it. The policy for memory placement can also be altered using the numactl utility.

### 2.1.3 Dynamic Frequency Scaling and Power Capping

One of the primary reasons the clock frequency began to stagnate around 2005 was a problem often called the "power wall." The power a processor consumes is directly related to the frequency of the clock rate. As a result, the ever-increasing clock rate translated to an ever-increasing need for power and the ability to dissipate the heat created with an increase in power.

One of the methods devised to deal with the issue of power is dynamic frequency scaling. These processors have the ability to operate at a variety of clock frequencies determined by the load on the system. The clock rate can be temporarily boosted (consuming more energy) for a short time when the system is under heavy load. Conversely, there is no need to have the system run at the maximum frequency when

the system is idle or nearly idle. In this situation the system clock frequency can be reduced and save energy.

Another area of research and development is the concept of power capping where the frequency of the processor can be scaled in such a way that a set power limit is not exceeded. This type of technology can be useful in large scale data centers where power is of great concern. One of the most well-known utilities for power monitoring and control is Intel Running Average Power Limit (RAPL). Developers can use RAPL to get an idea of how much power their CPUs are consuming at any time during the execution of an application. Users can also set a power limit that must be met as long as it is within a safe operating range for the system. The processor will then adjust the performance of the processor in order to stay within that power limit.

It is imperative developers consider technologies like dynamic frequency scaling and power capping when designing and executing their applications. For example, the performance of a section of code may be dramatically affected by the current load on the system or the power capping settings on the machine.

### 2.1.4 Accelerators and Hybrid Computing

High performance computing has also seen the introduction of accelerators or co-processors that can be used to accelerate portions of a computational application. NVIDIA originally produced Graphic Processing Units (GPUs) as an extra chip dedicated to rendering graphics. However, these highly parallel architectures were an excellent fit for many applications. The architectures, however, were extremely difficult to use for general purpose computing. As a result, NVIDIA introduced the CUDA architecture and API to make it easier to develop other applications for these architectures.

OpenCL was developed as an open standard for programming accelerators and it is the primary API for programming AMD GPUs. As an open standard, OpenCL

can be used to program a wide array of hardware including GPUs, CPUs, and even Field Programmable Gate Arrays (FPGAs).

Intel has blurred the lines between traditional CPUs and accelerators with the introduction of the Intel Xeon Phi (formerly called the Many Integrated Core architecture or MIC). The Xeon Phi is a Manycore architecture that has a large number (61 cores in the most recent model) of x86 cores. Each of the cores can run 4 threads per core and has a 512 bit AVX vector unit. The Xeon Phi was originally designed as a co-processor that can be used to offload heavy computational workloads. However, the next generation of Xeon Phi products (called Knights Landing) will be available as a co-processor or a self-hosted processor. One of the key selling points for the Intel Xeon Phi is the compatibility with the x86 instruction set that makes it relatively simple to port almost any preexisting code base to the new architecture.

Much of the hybrid computing landscape to this point has employed two separate chips in order to create a hybrid machine. This paradigm works in many applications but it also has its drawbacks. The most obvious is the necessity to move data from one device to another because the accelerators typically have their own memory that is separate from the system main memory. Some of this memory transfer can be hidden from the developer but the data transfer can still be a bottleneck in some applications.

Two projects seem to point to the possibility of a more unified hybrid architecture in the future. The first is the AMD Accelerated Processing Unit or APU that combines the traditional CPU cores and GPU cores on the same die. In the embedded field, NVIDIA has released the TK1 and TX1 as part of their Tegra line of processors designed for mobile and embedded applications. The new TX1 includes 4 ARM Cortex-A57 cores, 4 ARM Cortex-A53 cores and a 256 core Maxwell GPU. These two projects suggest that even if the accelerator and CPU unify on the same chip, developers may still be challenged to produce software that efficiently uses the variety of resources available.

## 2.2 Software Landscape

### 2.2.1 Task-Based Runtimes

Task-based scheduling has emerged as a key strategy to deal with the increasing parallelism in modern high performance computing. In order to apply the task-based computation model, the developer must first break a computational workload into tasks. For some applications, each of the tasks may be independent and can be performed without regard for order. These types of workloads are often described as being embarrassingly parallel. Generally, this class of problems has been relatively easy to solve using a master-worker paradigm.

There are other workloads, however, that may require that tasks be completed in a specific order to ensure the correctness of the computed solution. In the past this problem has often been solved using fork-join parallelism or bulk-synchronous programming. While this programming model does exploit the parallelism of modern computing architectures, it often is not the most efficient method to schedule these tasks. This is particularly true as modern computing architectures have increasing levels of parallelism making global synchronizations more costly. As a result, a new programming model emerged that reduces or eliminates global synchronizations in favor of asynchronous execution.

This model is often referred to as task-based scheduling, a task-based runtime, or task-superscalar execution. The systems that fall under this category tend to have a few characteristics in common. The first and most obvious commonality of these utilities is that the computation must be broken into a number of tasks that must be executed. The second characteristics is a set of dependencies between the task that must be respected in order to ensure the accurate completion of the algorithm. These dependencies are often depicted and represented by a Directed Acyclic Graph (DAG.) In this DAG, each of the nodes in the graph represents one of the tasks in the computational workload and each edge represents a data dependency. In most

cases (with the exception PARSEC), this DAG does not need to be explicitly defined by the developer. Rather than define the DAG manually, the developer must label the parameters for each task as one of the following:

- **INPUT** - A parameter designated as *INPUT* will be read during the task but will not be modified.
- **INPUT** - A parameter designated as *OUTPUT* will be modified or written during the execution of the task.
- **INPUT** and **OUTPUT** - These parameters are often referred to as *INOUT* and are used to designate parameters that will be read **and** written to during the execution of the task.

The tasks are then presented to the scheduler in a sequential fashion. Based on the parameter labeling and the order they are presented, the scheduler is able to generate the DAG. The dependencies are classified as one of the following:

- **Read after Write (RAW)** - A RAW dependency implies a task must wait until the previous task has written a piece of data before it can be read by another task.
- **Write after Write (WAW)** - A WAW dependency implies a task must wait until an earlier task has written a piece of data before it can be written again.
- **Write after Read (WAR)** - A WAR dependency implies a task must wait until another task has written a piece of data before it can be read and used.

These dependencies are generally queued on the data objects or pointers and each task must wait until any prior dependency for one of its parameters has been satisfied.

This type of execution is sometimes referred to as task-superscalar because of the similarities to superscalar instruction scheduling in computer architecture. Tomasulo's algorithm [52] allows for an efficient out-of-order execution of instructions

in modern computing architectures. In this fashion, tasks may be scheduled in any order as long as there are no data hazards that must be satisfied.

Another hallmark of these task-superscalar schedulers is an execution that is non-deterministic. This means tasks may not be executed in the same order or on the same resources from one execution to the next. This allows the scheduler to make decisions at runtime and makes them less susceptible to unexpected performance issues. For example, if one of the worker threads has completed its allotted work, it may "steal" work from another worker in order to balance the workload across the system.

### OmpSs

The OmpSs system, developed at the Barcelona Supercomputing Center, dates back to 1994. It was originally targeting grid environments, and was called GridSs [11]. It was later adapted to the IBM Cell B. E. processor under the name CellSs [39], and then to classic multicore processors (x86 and alike) under the name SMPSs [40, 40, 10]. The extension to GPUs (GPUSs) was introduced in 2009 [9]. The project is currently named OmpSs to underline the effort to extend the OpenMP standard with support for superscalar scheduling [25]. Due to the multiplicity of names, the project has also been intermittently referred to as StarSs [42]. The best known variant is the SMPSs multicore implementation, which is a compiler-based system that uses `#pragma` directives to annotate tasks that can be run in parallel and to decorate the data parameters with read/write usage information.

The main thrust in OmpSs is to become part of the OpenMP standard. Therefore, for the most part, OmpSs follows the OpenMP philosophy of offering a set of simple language extensions for quickly parallelizing algorithms. However, OmpSs does lack some of the flexibility of other libraries such as StarPU and QUARK. The project relies on the Mercurium compiler and the runtime environment is maintained by a library called Nanos++.

### QUARK

QUARK (QUeuing And Runtime for Kernels) was developed at the Innovative Computing Laboratory at the University of Tennessee Knoxville. It was originally developed as the main scheduler for the Parallel Linear Algebra for Scalable Multicore Architectures (PLASMA) library [1]. It has since been released as a standalone project [59] and has been used outside its original design to schedule for a wider variety of scientific codes. In general, QUARK provides a relatively small API but it still allows the user greater flexibility in code development. The library includes a number of features critical to the operation of a numerical software suite, such as error handling extensions and task cancellation capabilities. It also provides the user with the ability to save the execution DAG to visualize the dependences present in a particular algorithm.

QUARK was originally aimed at scheduling for homogeneous multicore systems with shared memory. It has since been used to develop software for systems containing GPUs as well as traditional CPUs [32]. It should be noted that QUARK does not provide any specific interface for accelerator support. It is the responsibility of the developer to ensure data is transferred properly during the execution of the algorithm. It has also been extended to applications in distributed memory environments [58].

### StarPU

The StarPU system developed at INRIA Bordeaux was first published in 2008 [5, 7, 6]. It is a runtime environment for task scheduling on shared memory architectures with the original motivation of exploring task scheduling in a hybrid CPU/GPU environment.

StarPU provides multiple interfaces for task execution which gives the developer great flexibility in expressing an algorithm. One of the key abstractions of the StarPU library is the codelet. The codelet is a small structure that allows the developer to describe various versions of a particular kernel using a single interface. For example,

the developer might want to define a matrix multiplication task for use in his/her algorithm. The user can define a codelet providing a CPU interface as well as a GPU interface allowing StarPU to execute the code on either of the target resources. StarPU uses implicit data dependencies to create a task DAG. It also profiles each task execution and uses historical runtime data to schedule tasks on the appropriate resources in heterogeneous systems, assigning tasks to CPU cores as well as GPU resources. StarPU provides a large set of interfaces and extensive functionality including execution trace, DAG generation, and several scheduling policies.

### OpenMP

OpenMP [22] has a long history in parallel computing and continues to evolve to suit a growing number of applications. Early OpenMP standards provided compiler directives for loop level parallelism. In these applications the developer could write a simple `for` loop and OpenMP would execute the iterations of the loop in parallel.

OpenMP 3.0 [36] was released in May 2008 and added the first task constructs to the standard. These tasks, however, would not be considered task-superscalar because they largely followed the fork-join model of parallelism. In this model the tasks are generated and executed in parallel while a *taskwait* construct is used to synchronize the tasks.

OpenMP 4.0 [37] was released in July 2013 and extended the task constructs to include task-superscalar concepts. The developer can now specify the input and output dependencies for each task block and the OpenMP scheduler will execute the tasks based on the inferred dependencies. Many in the task-based scheduling community view the inclusion of task-superscalar concepts as validation of the field and suggest it will be an active area of research for years to come. OpenMP 4.0 also included constructs for accelerators and SIMD instructions.

OpenMP 4.5 [38] was released in November 2015 and included one key feature not available in version 4.0. The priority clause has been added to the task construct and can be useful in many applications. The developer can now give each task a priority

level. If more than one task is available for execution, the task with higher priority should be executed first. This is a concept often used to give priority to tasks known to be bottlenecks or an important part of the critical path of the application.

OpenMP supports Fortran, C, and C++ and is implemented in many open source and proprietary compilers. Perhaps the two most known implementations of OpenMP are available in GCC and the Intel compiler suite. OpenMP has also received interest from the accelerator community because it is one of the primary programming models which can be used to program the many-core Intel Xeon Phi chips.

#### PARSEC

PARSEC [15] is a dataflow scheduler requiring explicit dependencies from the developer but it provides much greater scalability. The computation is represented in a job description format (JDF) file defining the tasks and dependencies in a compact format. This format allows the runtime to determine dependencies without unrolling the entire DAG. The ability to determine dependencies independently makes the runtime far more scalable. PARSEC provides the underlying scheduling and runtime for a scalable dense linear algebra library called DPLASMA. In contrast with many of the other task-based schedulers, PARSEC focuses on scheduling scalability in a distributed computing environment.

## 2.3 Tile-Based Linear Algebra

Dense linear algebra algorithms provide the basis for many scientific computing problems and remain an area of active research. These algorithms have evolved over time in order maximize performance with each new generation of hardware. Block algorithms were introduced with LAPACK [4] in order to make use of the caches on the newest architectures of the day. Achieving optimal performance for each new architecture would have been a challenging task. Block algorithms, however, simplified this problem by defining a set of Basic Linear Algebra Subroutines (BLAS)

that would be the building blocks for the higher level algorithms. Rather than adapting every algorithm to achieve optimal performance, developers could optimize the smaller collection of BLAS routines in order to achieve high performance for each architecture and the linear algebra operations built on top of them would also achieve high performance. There are three classes of operations in the BLAS often referred to as Level 1, Level 2, and Level 3. They refer to vector-vector operations, matrix-vector operations, and matrix-matrix operations respectively. Level 3 BLAS (matrix-matrix) operations are generally preferred because of the data reuse inherent in their algorithms. This data reuse exploits the caches in modern architectures and reduces data movement which generally resulting in higher performance. The Level 1 and Level 2 operations are memory bound and as a result, they generally do not reach optimal performance.

The basic formulation for many of the block algorithms is a two step approach. The first step is often referred to as the panel factorization where a small portion of the matrix is factorized and transformations are accumulated. This factorization is generally memory bound and do not achieve optimal performance. The second portion of the operation is often called the trailing submatrix update. This step applies the transformations from the panel factorization to the remaining portion of the matrix. The update step is generally applied using a Level 3 BLAS operation.

Parallelization of block linear algebra algorithms has often been performed using a fork-join model where the parallelism is generally expressed in the trailing submatrix update. This paradigm, however, struggles to provide the best performance on modern highly parallel architectures because of the costs of synchronization. Tile-based linear algebra algorithms have been developed in order to make better use of multicore resources. Two of the primary reasons these tile-based approaches achieve higher performance are the reduction in global synchronization and the more efficient use of caches.

One of the key differences between the two approaches is the way the data is laid out in memory. Previous algorithms required the matrix to be allocated in a single

array using column-major or row-major layout as seen on the left in Figure 2.4. For a tile-based approach, however, the matrix is stored by blocks as shown on the right in Figure 2.4.

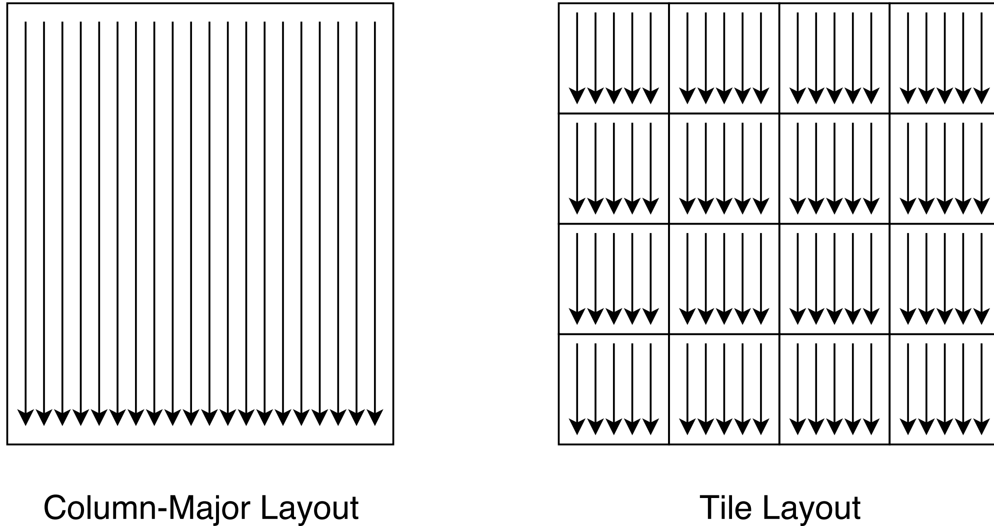



**Figure 2.4:** Colmn-Major Layout vs Tile Layout

These tile-based algorithms can be found in the PLASMA library developed at the University of Tennessee as well as the FLAME library from the University of Texas at Austin [28]. Tile-based linear algebra has been extensively studied [24, 29, 33, 2, 17]. The Cholesky, LU, and QR factorizations implemented in PLASMA will provide the basis for many of the experiments presented in this paper.

The tile approach consists of breaking the matrix panel factorization and trailing submatrix update steps into smaller tasks that operate on relatively small $NB \times NB$ tiles (or submatrices) of consecutive data which are organized into blocks-of-columns. The algorithms can then be restructured as tasks (which are basic linear algebra operations) that act on tiles of the matrix. The data dependencies between these tasks result in a DAG where nodes of the graph represent tasks and edges represent dependences among the tasks.

The execution of the tiled algorithm is performed by asynchronously scheduling the tasks in a way that dependencies are not violated. Optimally, we would like

**Algorithm 1** Tile Cholesky Factorization Algorithm

```
1: for k = 1,2 to NT do
2:    {Cholesky factorization of the tile A_{k,k}}
3:    DPOTRF(A_{k,k})
4:    for i = k+1 to NT do
5:       {Solve A_{k,k}X = A_{i,k}}
6:       DTRSM(A_{k,k}, A_{i,k})
7:       {Update A_{i,i} ← A_{i,i} − A_{i,k}A^T_{i,k}}
8:       DSYRK(A_{i,i}, A_{i,k})
9:    end for
10:   for i = k+2 to NT do
11:      for j = k+1 to i do
12:         {Update A_{i,j} ← A_{i,j} − A_{i,k}A_{j,k}}
13:         DGEMM(A_{i,j}, A_{i,k}, A_{j,k})
14:      end for
15:   end for
16: end for
```

this asynchronous scheduling to result in an out-of-order superscalar execution where slower tasks are overlapped in time with fast ones, which use cache more effectively. This would be managed by having the slower tasks start early, as soon as their dependencies are satisfied, while some of the parallel tasks (submatrix updates) from the previous iterations still remain to be performed and can be executed in parallel when a core becomes available. The scheduling of tasks is performed by the task-based runtime or scheduler.

Figure 2.5 presents the loops for the QR factorization in pseudocode and includes decorators for each tile to specify whether the tile will be read, written, or both. These dependencies are then used to infer the DAG and schedule the tasks accordingly.

Each of the tasks and corresponding dependencies must be presented to the scheduler. Each scheduling library provides their own API that is used to annotate the tasks and their dependencies. Some of them, like QUARK and StarPU, have an explicit interface that is used to “insert” tasks. OmpSs and OpenMP, however, provide compiler directives that are used to specify the tasks and their dependencies. The Cholesky algorithm is described in pseudocode Algorithm 1. Figure 2.6 presents the Cholesky algorithm implemented using the OpenMP compiler directives.

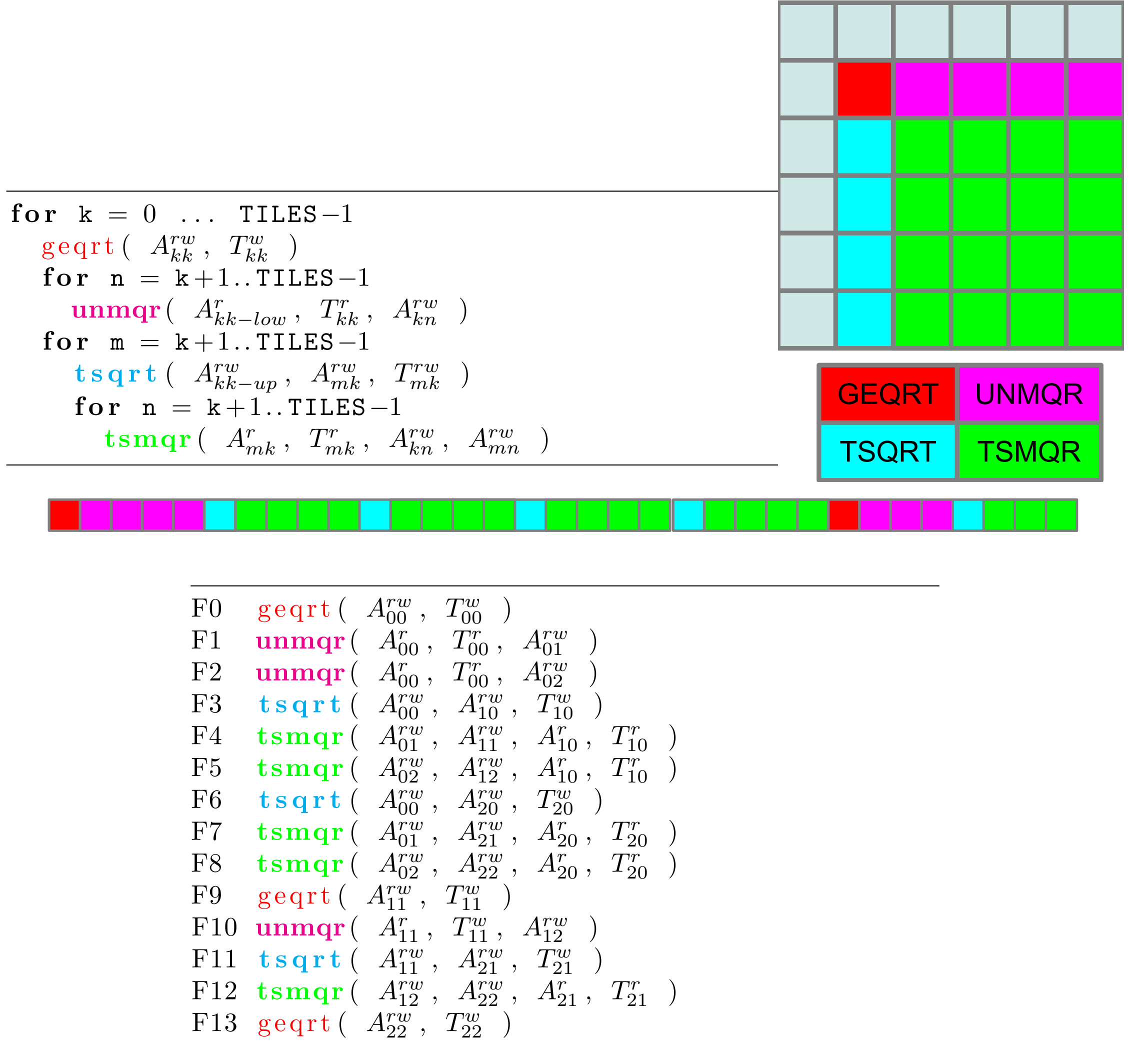



**Figure 2.5:** Pseudocode for the tile QR factorization showing all the tasks as they are sequentially generated. The data references tasks are decorated with their read and/or write status, implying data-hazards while executing the tasks.

```
#pragma omp parallel
#pragma omp master
{
    for (k = 0; k < nt; k++) {
        #pragma omp task depend(inout:A(k, k)[0:nb*nb])
        LAPACKE_dpotrf_work(
            LAPACK_COL_MAJOR,
            'L', nb, A(k, k), nb);

        for (m = k+1; m < nt; m++) {
            #pragma omp task depend(in:A(k, k)[0:nb*nb]) \
                             depend(inout:A(m, k)[0:nb*nb])
            cblas_dtrsm(
                CblasColMajor,
                CblasRight, CblasLower,
                CblasTrans, CblasNonUnit,
                nb, nb,
                1.0, A(k, k), nb,
                     A(m, k), nb);
        }
        for (m = k+1; m < nt; m++) {
            #pragma omp task depend(in:A(m, k)[0:nb*nb]) \
                             depend(inout:A(m, m)[0:nb*nb])
            cblas_dsyrk(
                CblasColMajor,
                CblasLower, CblasNoTrans,
                nb, nb,
                -1.0, A(m, k), nb,
                 1.0, A(m, m), nb);

            for (n = k+1; n < m; n++) {
                #pragma omp task depend(in:A(m, k)[0:nb*nb]) \
                                 depend(in:A(n, k)[0:nb*nb]) \
                                 depend(inout:A(m, n)[0:nb*nb])
                cblas_dgemm(
                    CblasColMajor,
                    CblasNoTrans, CblasTrans,
                    nb, nb, nb,
                    -1.0, A(m, k), nb,
                          A(n, k), nb,
                     1.0, A(m, n), nb);
            }
        }
    }
}
```

**Figure 2.6:** The tile-based Cholesky factorization implemented in OpenMP. Corner cases are ignored to improve clarity.

# Chapter 3

# Workload Simulation

This chapter and Section 4.1 are based on the following publication by Blake Haugen et al.:

- Haugen, Blake, Jakub Kurzak, Asim YarKhan, Piotr Luszczek, and Jack Dongarra. "Parallel Simulation of Superscalar Scheduling." In the *43rd International Conference on Parallel Processing (ICPP), 2014*, pp. 121-130. IEEE, 2014.

My contributions to this paper include (i) design of the simulation framework, (ii) implementation of simulation software, (iii) collection of experimental data, (iv) analysis of the experimental results, and (v) authorship of the majority of the text.

## 3.1 Related Work

Since the Minimum Multiprocessor Scheduling Problem is NP-complete [35], nearly all optimal scheduling problems in complex environments are NP-complete. This means most scheduling decisions are reached using heuristic algorithms, many of which can be found in the survey article [34]. The combination of complicated hardware configurations and scheduling heuristics make the search space too large

and complex for analytical models. As an alternative to most analytical models, developers often resort to empirical and simulation-based models [3, 55].

Simulation is not a new concept to computer scientists, and simulation tools seem to fall into two broad categories. The first is architecture simulation where the goal is to simulate the operation of a processor or system in order to analyze the accuracy of the output or performance characteristics. These simulations often do not focus on parallelism, but rather focus on fine-grain, instruction level simulation. The gem5 [14] and SESC [44] simulators are two examples of this type of simulator. An important aspect of both of these tools is the ability to simulate out of order executions, which are common in modern computer architectures.

At the other end of the spectrum are large scale simulations of parallel computing systems. The grid computing community has been particularly interested in simulation. Grid computing resources may be heterogeneous in nature and dispersed geographically and, for this reason, reproducibility of performance results may vary widely. Each allocation of grid resources may be very different and drastically change the performance of a grid computing job. Simulations have been commonly used to evaluate algorithms in this type of environment where it may not be possible to obtain reproducible results.

The diverse array of computing resources used in grid computing makes scheduling a very challenging problem, and the lack of reproducibility in the performance of each run made simulation a logical choice. Tools like SimGrid [19] and GridSim [18] were designed for these types of simulations. ChicSim [43] was another simulator built on top of a simulation language called Parsec (not the same as the task-based scheduler).

The Optorsim project [13] is another example of a grid computing simulator, and was developed to evaluate the performance of various data duplication algorithms. Data is often duplicated in a grid computing environment in order to deal with the geographic distribution of computing resources. The duplication of data decreases data access times and accelerates job performance. The Optorsim project aimed to

simulate the performance of grid computations based on the data replication strategies employed.

The Prometheus project [30] provides utilities to simulate task-based applications on many-core systems. Prometheus currently works with Cilk++ but they hope to extend the framework to other programming models in the future. The first step of the simulations is to extract the DAG for the application. This is currently done by intercepting the the Cilk++ keywords and generating a DAG. The second portion of the simulation is a hardware contention model. The hardware contention model is used to model the length of each task in the DAG based on some sort of performance model. These models can be created from workload measurements, cycle accurate simulation or an analytical performance model. Once the DAG and performance models are in place, the simulation proceeds with any number of scheduling algorithms.

The StarPU project has employed SimGrid to provide simulation capabilities within the scheduler. [48, 49, 47] This simulation, like the StarPU scheduler, has put a great deal of focus on hybrid computing systems where data must be transferred between the host and the device. The authors have studied dense linear algebra and a sparse linear algebra solver. When StarPU scheduler is used, it collects performance information about each of the tasks. This information is stored in an XML file that describes the performance characteristics of the system and the performance of the various tasks when they are performed on the system. The XML configuration serves as an input for the simulation and provides performance models for the various tasks in the application. The simulation employs the StarPU scheduler to keep track of the task dependencies and schedule them accordingly. The tasks, however, are simulated and do not perform actual computational work. The SimGrid library provides a thread API that allows the simulation to take control of all thread scheduling decisions.

## 3.2 Discrete Event Simulation

Discrete Event Simulations (DES) have been used to model problems in a variety of fields from healthcare to manufacturing. A DES is an excellent tool for understanding the performance obtained when scheduling various tasks. In general, each task is considered a single unit that does not change the system while it is occurring. The only changes to the system occur when a new task starts or ends. This simplification allows the simulation to ignore each time slice in a traditional continuous simulation.

In a serial context, a DES is trivial because there is only one event occurring at any given time. Therefore, each event is completely independent and it is not necessary to coordinate across multiple events. In a parallel context, a DES becomes more complicated because the events must properly synchronize before the simulation can continue. While non-trivial, this is still relatively easy to accomplish in the context of Fork-Join parallel applications because each event in the simulation must wait for the other events in order to continue with the simulation. Task-based schedulers, however, depart from the fork-join parallel model and make synchronization in any simulation a challenge.

### 3.2.1 Simulation Methodology

The ultimate goal of the simulator is to simulate a trace of the algorithm's execution with high accuracy. From the simulated trace information can be gained about scheduling decisions, execution time, and ultimately performance.

As a foundational principle, the simulation environment aims to have the scheduler performing the dependence tracking work while the computational work inside the tasks is not performed. In other words, the scheduler keeps track of all data dependences and makes all scheduling decisions as usual, but the tasks no longer contribute useful work toward the completion of the algorithm.

Arguably the most challenging aspect of creating correct simulated traces is the necessity to maintain the correct order of task completion. If each simulated

task simply records its information in the trace and exits, it is very likely the task dependences will be satisfied in a different order than the original, which can ultimately cause drastic alterations to the simulated trace. The main reason for this is that the original tasks perform useful computations and take time to do so while also interacting with other resources such as shared caches, the memory system, and the OS. A task that records a small piece of trace information and exits will have very little interaction with the hardware resources.

The simulation generally relies on three crucial elements. The first element is the simulation clock which keeps track of the simulation time. The clock is stored as a double precision floating point number which is of sufficient resolution for the simulated tasks that operate at the microsecond resolution. The simulation library must also keep track of the simulated trace (the second element and the output of relevance to the developer) as well as a queue of tasks that are currently executing (the third element).

There are two primary assumptions that must be true in order to ensure accurate simulations.

- **The scheduler overhead is small relative to the tasks being scheduled.** After each task in the workload is completed, the scheduler must perform the necessary bookkeeping and schedule the next task. This time between tasks is often called scheduling overhead. One of the most common problems for this class of schedulers is that it struggles when the length of each task becomes too small. In this case, the scheduler becomes a bottleneck and cannot feed enough work to the processors. Another scheduling issue occurs when the number of processors begins to grow. This is logical because there is generally only one thread doing the scheduling. If there are a large number of cores, it can be difficult for the single threaded scheduler to keep up with the demand for tasks from the workers. These are known problems with many of the task-based schedulers and they often cause increasing error in the simulations as well.

- **The scheduler does not behave differently because of the simulation.** The simulations often do not run at the speed of the real execution. (The speed of the simulations depends on a complex interaction of the number of simulated cores, the number structure of the DAG corresponding to the workload, and the length of each task. In some cases the simulations are faster and in some cases slower.) If the scheduler makes decisions based on the length of each task it may make different scheduling decisions when it is simulating an application.

Examples of the errors are caused when these assumptions are violated will be presented later.

The novelty of our simulation approach is the complete reliance on the scheduler to provide the facilities to maintain the task dependences and make all scheduling decisions while still being portable across multiple schedulers. In order to create a simulation, the programmer simply replaces each task function with a call to the simulation library. Only a few lines of initialization and cleanup code before and after the execution of the algorithm simulation are needed to perform a simulation. This makes our approach portable since there is no assumption about the underlying algorithm being scheduled or about data-dependence tracking, nor do we require any invasive changes to the existing implementation of tasks. The simulation also allows the user to simulate the behavior of the scheduler independent of the computational platform. Once a problem configuration has been defined, the user can simulate their workloads on another machine regardless of the number or type of processors available.

### 3.2.2 Tracing

In order to simulate a given trace, it is necessary to have complete control over the generation of the execution trace. Most general purpose tracing utilities and frameworks are designed to create traces based on true (or wall-clock) execution time, but the simulation requires a trace based on the simulated (or virtual) execution time.

This lead to the following decision. Rather than attempting to modify an existing trace generation tool, a rudimentary trace generation environment was created allowing the user to log tasks during execution with the simulation (user-specified) time. After the completion of the algorithm, the trace is stored in a CSV file. The CSV format was chosen to simplify the process of analyzing the trace data. Many other trace file formats require libraries to read the data, while many languages already provide utilities for manipulation the CSV. This can be useful for performing statistical or structural analysis of the trace. The trace file can also be used to generate a visual representation of the data such as an SVG (Scalable Vector Graphics) or an interactive visualization like the one presented in Chapter 5.

This trace environment is also available to assist developers in collecting information about the tasks in their workload which can be used to build a statistical model for each of the tasks.

### 3.2.3 Model of Kernel Executed inside a Task

One of the key factors for performing accurate simulations is the ability to accurately measure and describe the execution time of a kernel. Each of the kernels provides the building block of the simulated trace. If the model of a single kernel is inaccurate, the effects will be compounded as the trace is simulated and the kernel invocation repeats. This can be a source of a sizable error in the simulation.

In order to more realistically simulate the execution of an algorithm, each task's running time is not fixed, but rather is determined by a probabilistic distribution. For example, it is unlikely each DGEMM kernel requires exactly the same time to execute in any given trace. The distribution of these kernel times will vary from application to application, or even between the runs of the same application. The generation of running time of the simulated kernels based on a prescribed distribution adds an element of randomness to the trace, which is essential for the accuracy.

### Timing Methodology

One of the challenges a developer faces in modeling a kernel is timing each kernel. It initially seems obvious that one could very quickly call each kernel in isolation in order to obtain an estimate of the time required for the completion of that kernel. This will likely give the developer an idea of the execution time of the kernel, but this is unlikely to give results with high accuracy. The developer must be careful to consider where the sub-matrix will be in the cache hierarchy. In the context of a true execution, the kernel may or may not have its data available at the top of the cache hierarchy. To make the task even more challenging, on a NUMA machine it is possible that the data required for the task is stored on non-local memory.

In order to quantify the differences in timing methodology, a series of benchmarks were designed to evaluate the performance of a DGEMM task in a number of different scenarios. The benchmark methodology is largely based on the work of Whaley and Castaldo [57] including new extensions for NUMA architectures.

In order to demonstrate the challenges associated with developing a benchmark that accurate reflects the scenarios presented in a real workload, the double precision general matrix-matrix multiplication (DGEMM) task will be studied in great detail. DGEMM is defined as:

$$C \Leftarrow \alpha A \times B + \beta C$$

where $A$, $B$, and $C$ are double precision matrices and $\alpha$ and $\beta$ are scalar multipliers. This task occurs in a number of dense linear algebra applications. In this case, we will examine the DGEMM task in the context of a Cholesky factorization and a tile-based implementation of DGEMM. In order to ensure all of the results were comparable, the DGEMM tasks being executed in the real workloads and the artificial benchmarks were all done with the same configurations. (TransA = Trans, TransB = NoTrans, $m = 200$, $n = 200$, $k = 200$, $\alpha = -1.0$, and $\beta = 1.0$) This ensures all of the workloads are identical and the only differences are the context in which they are executed. In the case of the Cholesky factorization, a matrix of size 5000 with a tile size of 200 was

used. In the case of the DGEMM, $A$, $B$, and $C$ are square matrices of size 2600 and a tile size of 200. The distribution of the DGEMM tasks in these two workloads are used as a baseline to evaluate the results of each of the synthetic benchmarks. These distributions are shown in red and blue in Figures 3.1, 3.2, 3.3, 3.4, 3.5, and 3.6.

The experiments were performed on two machines. One machine was composed of two 8 core Intel Xeon E5-2690 processors and 2 NUMA nodes. The second machine was equipped with four 12 core AMD Opteron 6180 SE processors and 8 NUMA nodes.

In order to evaluate the feasibility of deriving task timing models from synthetic benchmarks, a number of factors must be considered. Each of the benchmarks executes and collects the execution time of several iterations of the task. Presented here are six synthetic benchmarks fall into three primary categories as follows:

- **Serial** - The Serial class of benchmarks is the most common and could be considered the naive implementation. In this case the memory is allocated and the tasks are computed. Assuming the "first touch" memory place policy, it is assumed the $A$, $B$, and $C$ matrices will be initialized on the memory closest to the processor that will be performing the computations. This benchmark is generally unable to model the effects of NUMA architectures on the task timing. The results of the Serial benchmarks are shown in Figures 3.1 and 3.2.

- **NUMA** - The NUMA class of benchmarks improves on the Serial benchmarks by accounting for the effects of non-local memory access during the computations. In this class of benchmarks, the process is initially bound to a single core from which the operands are allocated and initialized. This ensures the data will be placed on a single NUMA node based on the "first touch" rule. When the tasks are executed, they are performed on each of the cores in a sequential fashion. (i.e. the first n tasks are executed on the first core followed by the next n tasks on the second core etc.) The distribution derived from these benchmarks is often multimodal based on the distance between each of the processors and

the NUMA node containing the data. The results of the NUMA benchmarks are shown in Figures 3.3, 3.4.

- **Threaded** - The Threaded class of benchmarks aims to improve on the NUMA benchmarks by accounting for the memory contention that occurs during the real execution of a task-based workload. The Threaded benchmarks are multithreaded with each task explicitly bound to one core of the system. Each of the threads starts by allocating and initializing the operands in parallel. This ensures the operands are distributed across all of the NUMA nodes on the machine. If each thread only executed on the operands it was responsible for allocating, the benchmark would likely be artificially faster than expected because the tasks would never retrieve data from non-local memory. As a result, the operands are "shuffled" between the threads causing some of the operands to be in local memory while others are in non-local memory. Unlike the NUMA benchmarks, the Threaded benchmarks are executed in parallel to accurately stress the memory bandwidth as it would in a real task-based workload. The results of the Threaded benchmarks are shown in Figures 3.5, 3.6.

The synthetic benchmarks must also consider whether the operands are present in the cache at the time of execution. Operands not located in the cache must be retrieved from the main memory which increases the time to execute the task. In order to simulate the situation where the operands are in cache, the task is called repeatedly with the same operands. In order to simulate the situation where the data is not present in the cache, several operands are allocated and initialized prior to the execution of the tasks. Each task is called with a different operand ensuring the data must be retrieved from the main memory for each task.

The ability to account for warm cache and cold cache scenarios only complicates the design of an artificial benchmark. In the case of DGEMM, there are three primary operands ($A$, $B$, and $C$), raising the question of which operands should be in cache

**Table 3.1:** The 8 cache scenarios for $A$, $B$, and $C$ in the synthetic DGEMM benchmarks. "IN" indicates the operand is in cache while "OUT" indicates the operand is out of cache.

| **A** | **B** | **C** |
|---|---|---|
| IN | IN | IN |
| IN | IN | OUT |
| IN | OUT | IN |
| IN | OUT | OUT |
| OUT | IN | IN |
| OUT | IN | OUT |
| OUT | OUT | IN |
| OUT | OUT | OUT |

and which should not be in cache. With three operands that can be in cache or out of cache the number of possible benchmarks is 8 as shown in Table 3.1.

To complicate matters further, it is unlikely any of the operands will always be in cache. Perhaps a percentage of the time the operand is in cache. This can be simulated by slightly modifying the out of cache algorithm to use the same operand for some of the iterations while selecting a new operand for other iterations. It is also not possible at this time to consider a case where part of an operand is in cache. In the case of the Cholesky workload, there are different types of tasks being executed that can require varying amounts of memory bandwidth.

When the three classes are considered in conjunction with the numerous cache configurations it becomes a challenge to select the scenario that accurately reflects the distribution of task times in a real workload. Here we will examine just six of the many possible benchmarks.

The first two benchmarks are the cold cache and warm cache, Serial benchmarks. In these two benchmarks all of the operands are either in or out of cache and the data allocation, initialization, and task execution all occur on the same core. The results of these two benchmarks are shown in Figures 3.1 and 3.2. The cold cache benchmark is slower than the warm cache benchmark on both machines, but neither appears to

accurately model the distribution of task times from the Cholesky factorization or the DGEMM workloads. On the Intel machine with two NUMA nodes, the synthetic benchmarks appear to "bound" the actual distributions while both appear to be faster than the real distributions on the AMD machine. This is likely due to the fact that the AMD machine has a greater number of cores and NUMA nodes that are not accounted for in the benchmarks.

Figures 3.3, 3.4 present the results of two NUMA benchmarks. The two benchmarks represent the case where all or none of the operands are in cache. When all of the operands are in cache the benchmark suggests artificially fast task time due to the decreased memory bandwidth requirements. The cold cache scenarios provide a multimodal distribution with modes corresponding to the distances between the processors and the memory containing the operands. Again, the benchmarks appear to "bound" the real distributions on the Intel machine while both benchmarks underestimate the task time on the AMD machine. This is likely due to the fact that the tasks are executed in serial and do not account for the memory contention present in a real workload.

Figures 3.5, 3.6 present the results for the warm and cold cache Threaded benchmarks. Again, the benchmarks on the Intel machine "bound" the actual distributions. On the AMD machine, however, the warm cache scenario under-predicts the execution time of the tasks, but the cold cache scenario appears to be much closer to the actual distribution of the task times.

The wide variety of benchmarks with varying accuracy makes synthetic benchmarks unappealing for building statistical models of the execution time for each task. As a result, data collected from the execution of a real workload provides the most accurate data for building task models.

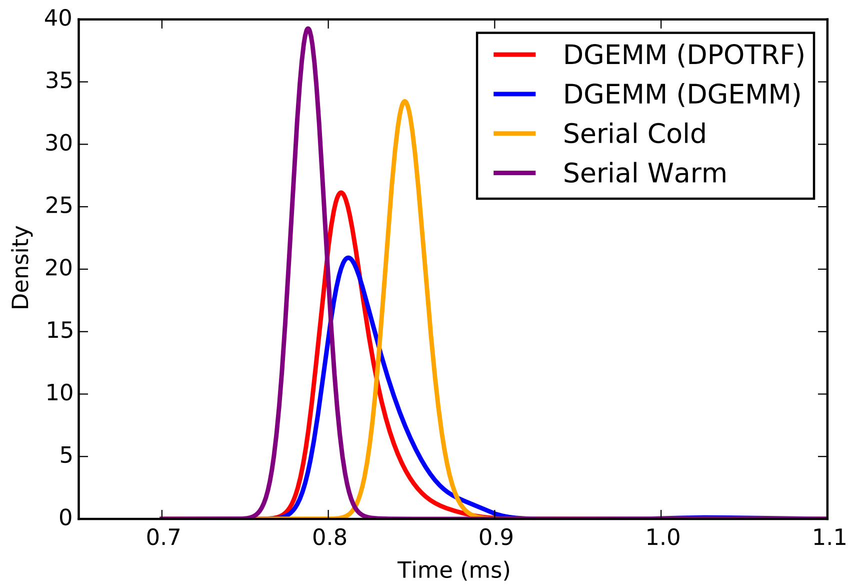



**Figure 3.1:** Kernel Density Estimation curves for DGEMM including serial benchmark data sets. 2 x 8 Core Intel Xeon E5-2690

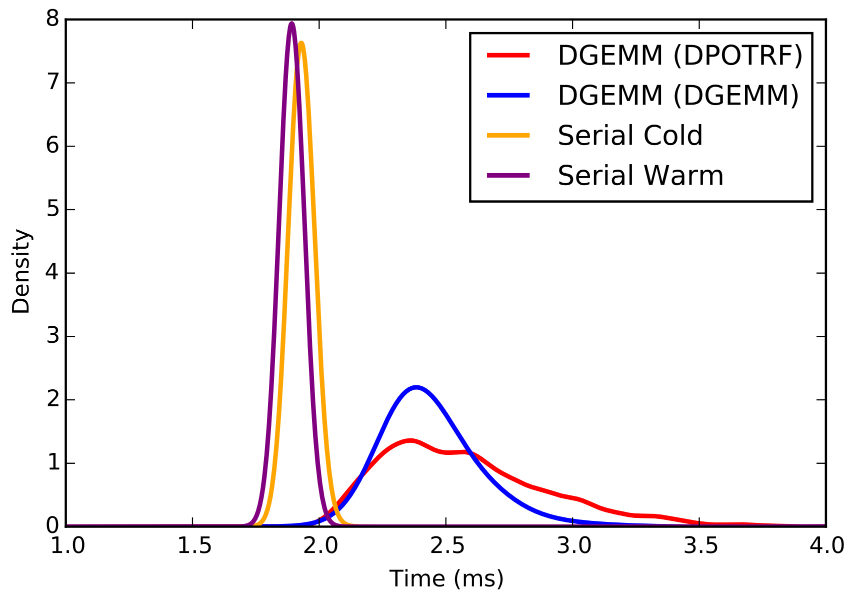



**Figure 3.2:** Kernel Density Estimation curves for DGEMM including serial benchmark data sets. 4 x 12 Core AMD Opteron 6180 SE

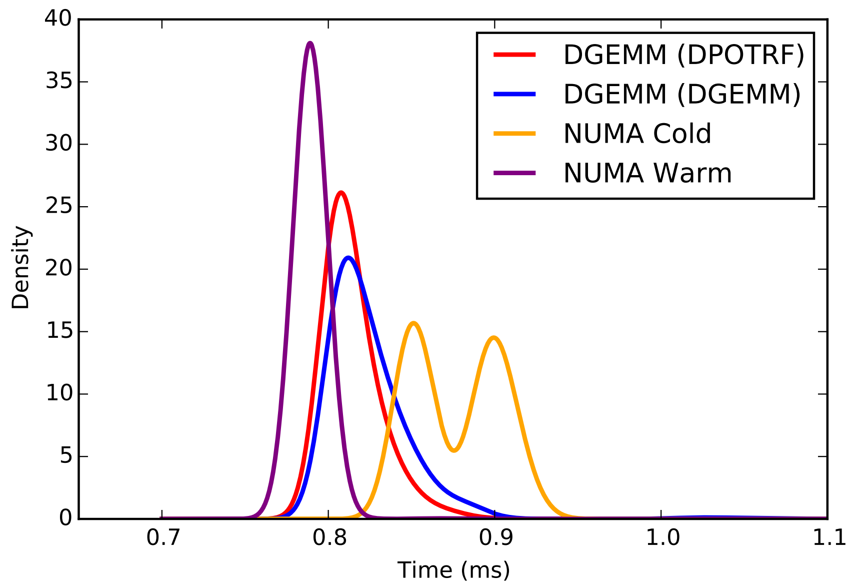



**Figure 3.3:** Kernel Density Estimation curves for DGEMM including NUMA benchmark data sets. 2 x 8 Core Intel Xeon E5-2690

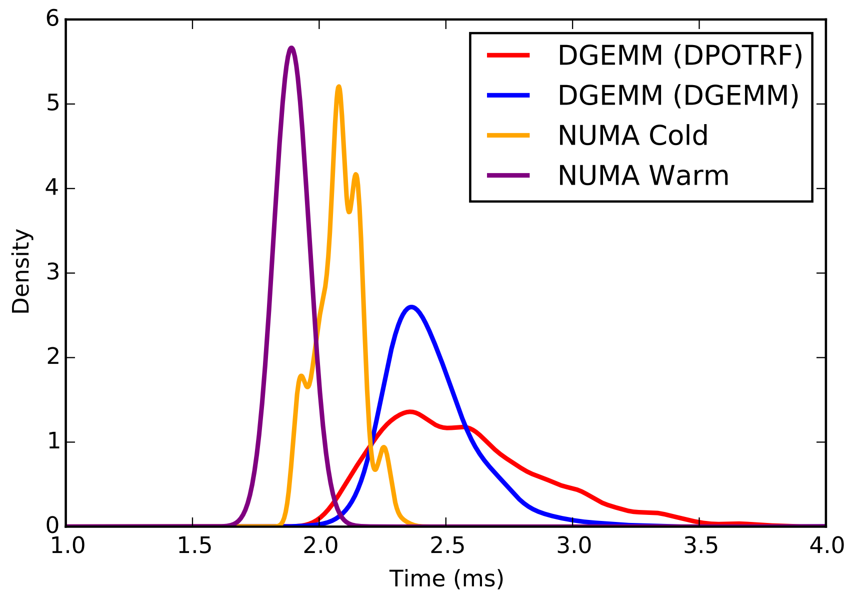



**Figure 3.4:** Kernel Density Estimation curves for DGEMM including NUMA benchmark data sets. 4 x 12 Core AMD Opteron 6180 SE

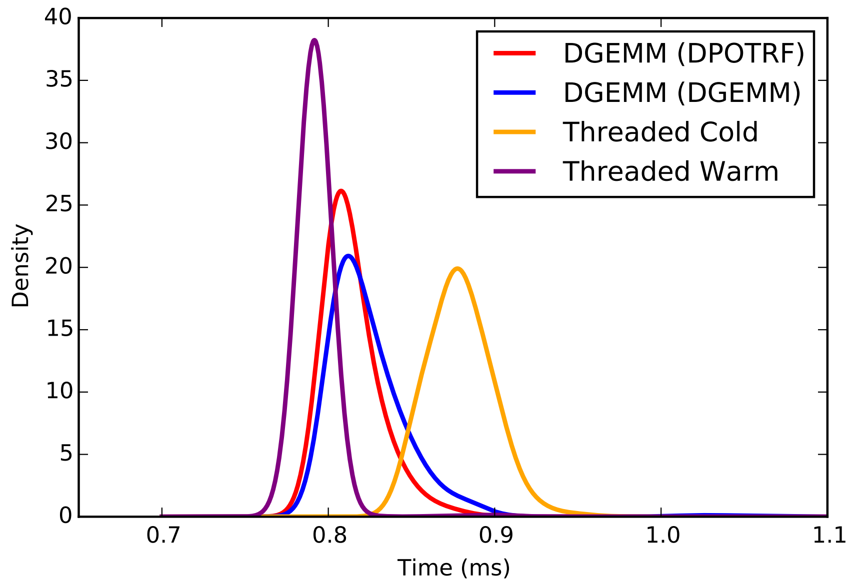



**Figure 3.5:** Kernel Density Estimation curves for DGEMM including Threaded benchmark data sets. 2 x 8 Core Intel Xeon E5-2690

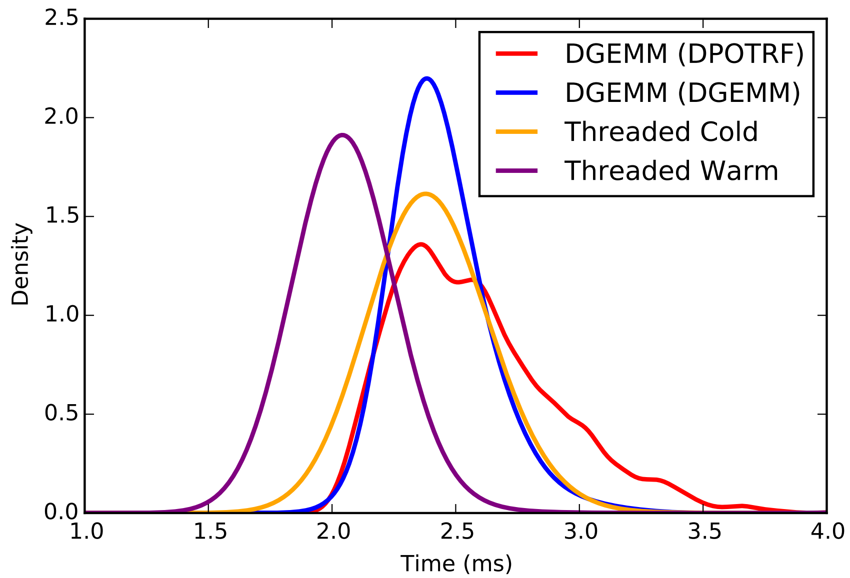



**Figure 3.6:** Kernel Density Estimation curves for DGEMM including Threaded benchmark data sets. 4 x 12 Core AMD Opteron 6180 SE

The results of runs shown in this paper were performed by linking with the Intel MKL library in order to obtain the best performance. As is common for large libraries, which require resource allocation, the MKL library initializes its internal state upon the first execution of a kernel and for each thread of execution. This may be easily observed as the first kernel on each thread will take significantly longer to execute than the following kernels. These extreme outliers can drastically affect the model fitting. For this reason, each of the threads is initialized with another call to the MKL library in order to ensure that this initialization is performed before the trace is collected. The same is done for the synthetic benchmarks as well.

**Dense Linear Algebra Kernel Modeling**

The sample problems examined here are Dense Linear Algebra applications. Their implementations are based on the PLASMA library where each high-level linear algebra routine is composed of several smaller tasks which can be scheduled based on their dependencies. Each of these tasks is a kernel belonging to any one of various classes of kernels, depending on the operation being performed. As mentioned above, each kernel of a given type does not have identical performance due, primarily, to the fact that each execution of the kernel will have different cache residencies. For example, one execution may have most of the data in cache while another execution has very little of the data in cache, which relates to, for example, task placement policies and to what extent the scheduler tracks data affinity.

In dense linear algebra, the kernels are most commonly described using the normal distribution of execution times, but similar distributions may also be used to model execution time. The simulation library currently supports normal, lognormal, and uniform time distributions as well as constant time models. In order to simplify the process of defining model configurations, a configuration file (based on a CSV format) is used to define the models for the simulator. These models can also be overwritten with a custom task time for a task. This will be described in greater detail in Section 4.3. Experience has shown that lognormal and normal distributions

have produced nearly identical simulation accuracy and can provide more accurate simulation than the uniform and constant task timing models.

### 3.2.4 Task Execution Queue

In general, the dynamic scheduler maintains a dependence graph which is used to determine whether the dependences for a specific task have been satisfied. Whenever a task finishes its execution, the tasks waiting for the output of that task have a "waiting" dependence removed. Once all dependences have been removed for a task, the scheduler marks it to be available for execution.

In the case of simulated execution, the order in which these dependences are satisfied must be maintained in order for the simulations to be accurate. The key element of the simulation environment is the Task Execution Queue. This is the data structure ensuring the tasks that are currently in the execution state (Note: a task in the execution state is not actually computing the function it simulates) within the simulation maintain the proper completion order. When each task is executed in the simulation it already knows when it will end in the simulation based on the simulation clock time at the start time of the simulation and the expected execution time of the task based on a statistical model. In order to ensure the tasks (and implied dependencies) are completed (and released) in the same order, the tasks are released one at a time based on the order in the queue. The queue is ordered based on the simulated ending time of the tasks that are currently executing.

### 3.2.5 Simulation Task Function

In order to use the simulation library, the developer simply replaces the calls to each computational kernel with a call to the simulated kernel. This simulated kernel requires an identifier as well as any handles or pointers that will create a dependence in the real execution of the algorithm. Although the memory is never accessed, the actual memory location in the process' address space is required in order to ensure all

of the dependencies will be maintained. Furthermore, some schedulers perform copies of the data to deal with anti-dependences and real memory locations are required for such copies to succeed. The simulated tasks derive their execution time based on a random task time generated within a distribution provided in the simulation configuration file. The user also has the ability to specify a custom task time which will be described later. The simulated tasks are inserted into the task graph using the scheduler's API in an identical fashion to a real kernel.

The scheduler continuously maintains dependences and schedules each task accordingly. When a simulated kernel is executed, the simulation begins by checking the simulation clock to determine when the kernel is starting. Based on the kernel starting time and the estimated time of kernel execution (based on the kernel's model of completion time), the ending time can be obtained. The simulated kernel then acquires the lock on the Task Execution Queue and is added to the queue. The kernel information can now be added to the simulated trace and is ready to exit. However, the task must wait until it is at the front of the queue in order to allow the function to return. From the scheduler's perspective, the task is still executing until the function (which represents the task) returns. Before finishing, however, the simulated kernel must also update the global simulation clock to the completion time from the model distribution before the function returns.

### 3.2.6 Scheduling Race Condition

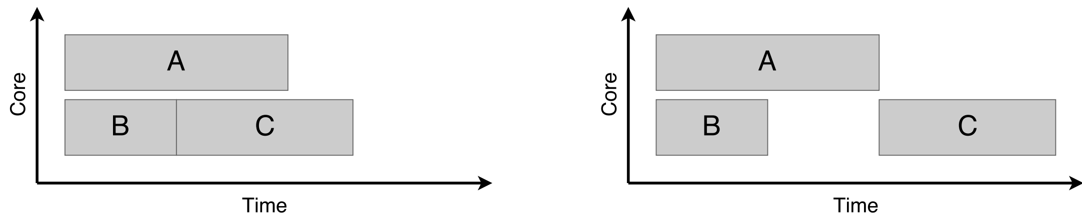
One of the challenging aspects of ensuring the correctness of the simulation stems from a possible race condition. The race condition can occur when a task is at the front of the Task Execution Queue while the scheduler is inserting new tasks. Each task starts by determining a start time by querying the simulation clock. Each task ends by updating the simulation clock. The race condition arises when a task attempts to complete while another task is determining what time it started in the simulation. If the new task gets the simulation clock before the update by another task, the

results will be accurate. If, however, the other task completes and updates the clock before the new task can query the simulation time, the start time of the task will be incorrect. The magnitude of the error depends on the structure of the algorithm and the tasks involved.

Figure 3.7 presents a more concrete example of the effects of this race condition. This simple workload includes three tasks and is performed on a two core system. Tasks A and B are independent and can be scheduled at the same time. Task C depends on data from B but is independent of Task A. Figure 3.7a represents what the actual execution of the workload would look like. Tasks A and B are scheduled on the two available cores. Once Task B completes, the scheduler recognizes that the data Task C is waiting for has been satisfied with the completion of Task B. As a result, the scheduler executes Task C on the second core of the system.

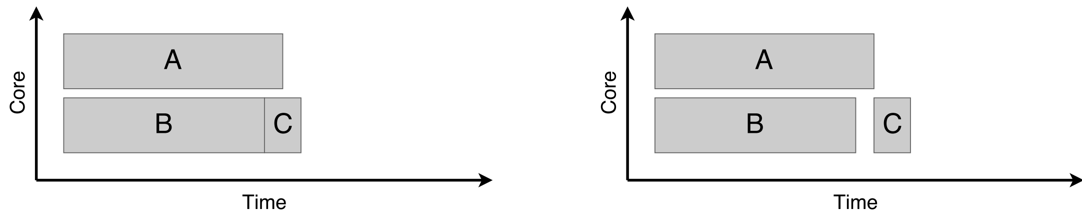
When the race condition does not cause an error, the simulation should look identical. The simulation task queue already contains Tasks A and B. Task B is at the front of the queue because the simulated ending time is earlier than the simulated end of Task A. Now that Task B is at the front of the queue, it will add its information to the trace, update the simulation clock, and return. Once Task B returns, the scheduler releases the corresponding data dependence(s) and executes Task C on the second core of the system. After Task B completes, Task A is now moved to the front of the queue of running tasks. This is where the race condition can occur. If Task C queries the simulation clock before Task A updates the simulation clock, the resulting trace will be correct and look like Figure 3.7a. However, if Task A updates the simulation clock before Task C can query the simulation clock, the race condition will cause an error because the simulation clock will not accurately reflect the time the task would have started. The effects of this error can be seen in Figure 3.7b. Here the simulation would suggest the task starts later than it would in a native execution of the algorithm.

The effects of this race condition can vary widely depending on the structure of the application and the characteristics of the tasks. Figure 3.8 shows a workload



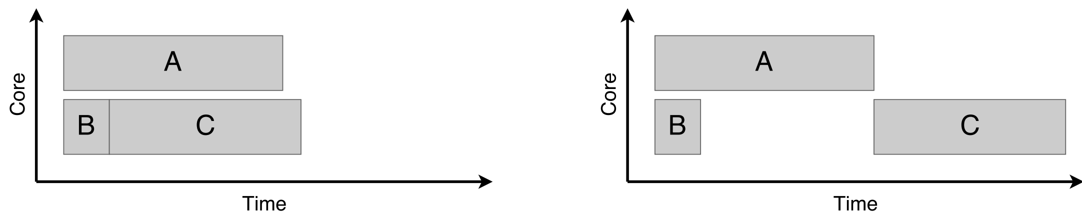

**(a)** Race Condition Success **(b)** Race Condition Error

**Figure 3.7:** A demonstration of race conditions in the simulation.




**(a)** Race Condition Success **(b)** Race Condition Small Error

**Figure 3.8:** A workload where the race condition causes a relatively small error.




**(a)** Race Condition Success **(b)** Race Condition Large Error

**Figure 3.9:** A workload where the race condition causes a relatively large error.

where the error caused by the race condition is relatively small due to the length of the tasks. Task C wouldn't start much later than expected because of the relatively small clock update from Task A. However, Figure 3.9 illustrates a situation where the error caused by the race condition could drastically change the accuracy of the simulation. In this case, Task A updates the simulation clock and moves the simulated start time of Task C much earlier than expected.

There are currently two solutions to eliminate this race condition. The first is a function recently added to QUARK. The function allows the developer to determine if the scheduler has completed all bookkeeping related to scheduling. This means the task can also query the scheduler to ensure this race condition will not occur. The obvious downside of this technique is that it is not portable across schedulers.

The other solution to this problem that is portable for all schedulers is a judicious use of the `usleep()` function. This is used so that the simulated kernel will sleep for a fraction of a second and thus allow the scheduler to complete any bookkeeping.

## 3.3 Spin Simulations

One of the shortcomings of the Discrete Event Simulations described in Section 3.2, is the inability to account for scheduling overhead. This is the basis for the assumption that scheduling overhead be small relative to the length of the tasks. When the overhead is small, the DES assumes they are negligible and can safely be ignored. This assumption can be problematic when there are a large number of cores or the tasks are short and overwhelm the single core responsible for scheduling tasks. As a result, the simulation framework has included a second simulation mode called Spin.

Spin simulations are not as versatile and portable as the DES but they can be helpful in situations where the scheduler may be overwhelmed by the workload being processed. The basic principle behind the Spin simulations is to replace each task with a call to a "sleep" function based on the statistical model of the task time. This simulation mode should provide the same rate of work for the scheduler and thus

provide accurate results even when the scheduler is overwhelmed and the scheduling overhead is a non-negligible portion of the run time. In this mode, the trace is collected based on a real "wall clock" time as opposed to the virtual simulation time in the DES.

When implementing this simulation mode, the choice of "sleep" function can have a significant effect on the accuracy of the simulations. Many of the sleep functions available on the operating system are defined to sleep for **at least** the specified amount of time but this could be more based on implementation and context specific details. In order to evaluate a number of different implementations, a benchmark was devised to compare three different implementations. The benchmark consisted of 1000 repeated calls to the "sleep" function where each call should sleep for 1 millisecond. The time for this loop should result in a total time of exactly one second.

The first two implementations simply called the `usleep` and `nanosleep` functions provided by the operating system. These sleep functions have microsecond and nanosecond resolutions respectively. However, when these functions are called in a loop that should complete in exactly one second, the resulting loop for each is 1.057669 s for `usleep` and 1.057613 s for `nanosleep`. (Tests are performed on an 8 Core Intel Xeon E5-2690.) This error may not appear to be much, but in the context of the simulations where thousands of tasks will be modeled, these errors quickly add up.

As a comparison, a custom "spin" function was created. This function doesn't sleep in a traditional sense but rather spins in a loop until a specific period of time has passed. In this case, one millisecond. This spin loop, when executed 1000 times results in a time of 1.000000 s and provides far more accurate results than the operating system defined sleep functions. This spin function is far less efficient in terms of processor usage, but the primary goal of this function is accurate timing, rather than efficiency. For this reason, a custom "spin" function is used to model the time for each of the tasks in the Spin simulation.

# Chapter 4

# Simulation Results and Applications

In order to analyze the accuracy of the simulations, the simulated traces should be compared to the results of the real execution of the workload. It is important to remember the scheduling decisions performed by task-based runtimes are generally nondeterministic. This means the simulated trace is unlikely to look identical to the trace collected from the real execution of the algorithm. They should, however, share many of the same characteristics.

The most common method to determine the accuracy of the trace is to compare the length of time a workload takes to complete. The time between the start of the first task and the end of the last task in an application should be similar between the simulated and native execution traces. In the field of dense linear algebra, we are often interested in the rate at which floating point operations (flops) are performed. On modern computing architectures this rate is often expressed in Gigaflops, Teraflops, or even Petaflops for the largest parallel computing systems.

$$Gflops = \frac{flops}{time}$$

**Table 4.1:** Floating Point Operations

| **Algorithm** | **Floating Point Operations** |
|---|---|
| Cholesky | $\frac{1}{3}N^3$ |
| LU | $\frac{2}{3}N^3$ |
| QR | $\frac{4}{3}N^3$ |

In many of the dense linear algebra applications the number of floating point operations is fixed based on the size of the problem. Table 4.1 presents the commonly used formulas to calculate the number of operations for the Cholesky, LU, and QR factorizations.

Some of the algorithms actually perform more floating point operations than the formulas expressed in Table 4.1 because of the tile-based formulation of the problem. Even in this case, the formulas here are generally used for any formulation of the problem in order to have an effective flop rate as opposed to a literal flop rate. This allows for comparison of multiple implementations of the same algorithm. The accuracy of the simulations can be compared quantitatively based on the simulated and real runtime or the simulated and real flop rate.

Some of the properties of a trace can be harder to quantify. These qualities are often best evaluated by visualizing the trace and examining it. Many of the properties of the trace can be observed but hard to make quantitative comparisons. Some of the questions may be as follows: Where do the tasks get scheduled? Are there gaps in the trace? Where is the trace sparse? Where is the trace dense? Are there unique visual artifacts of the trace? These are all questions we can often answer qualitatively by examining the trace.

In some cases the simulation does not provide accurate estimations of performance and time. These usually arise when the initial assumptions of the simulation are violated. The trace visualizations can often be used to determine what caused the error.

It should also be noted that these schedulers generally are not deterministic. As a result, it is unlikely any two traces, even from the same algorithm, will be identical. The tasks in a workload are not generally bound to a certain thread. It is important to recognize that whether the traces are real or simulated, they are unlikely to be identical but they should be similar.

## 4.1 Comparison of Schedulers

The first version of the simulation was aimed at the QUARK scheduler. (Figure 4.1 presents the performance results for these simulations.) This implementation makes use of the QUARK extension allowing the simulation to query the scheduler to determine whether the scheduler has completed scheduling other tasks in order for the simulation to proceed. The simulation error for small matrices can reach nearly 20% but the errors quickly drop and are near zero for larger matrices. The larger error for small matrices is common among all of the schedulers. This error is likely an artifact of the simulators inability to accurately model any startup costs associated with the application. The smaller runtimes for smaller matrices also means even relatively small errors can be a relatively large percentage of the total runtime.

Figure 4.2 and Figure 4.3 present the simulation results from the StarPU and OmpSs schedulers. The real performance curves for each of the factorizations was collected from an implementation in each scheduler. Again, each of these schedulers has a larger percentage error for the smaller matrices but the error is close to zero for larger matrices.

Figure 4.4 demonstrates the simulation accuracy of the applications as implemented using OpenMP. OpenMP is an open standard and each implementation of the standard will have the standard API but other details and performance are implementation dependent. Figure 4.4 uses the GCC implementation of OpenMP but some of the results later in the paper will make use of the Intel implementation

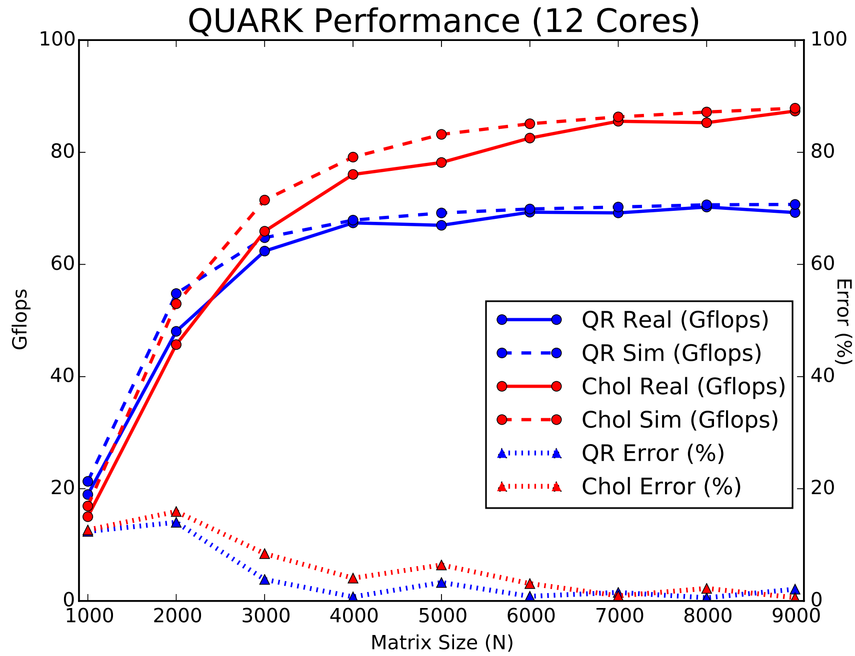



**Figure 4.1:** Cholesky and QR performance results using the QUARK scheduler. $NB = 200$ 12 Core AMD Opteron 6180 SE

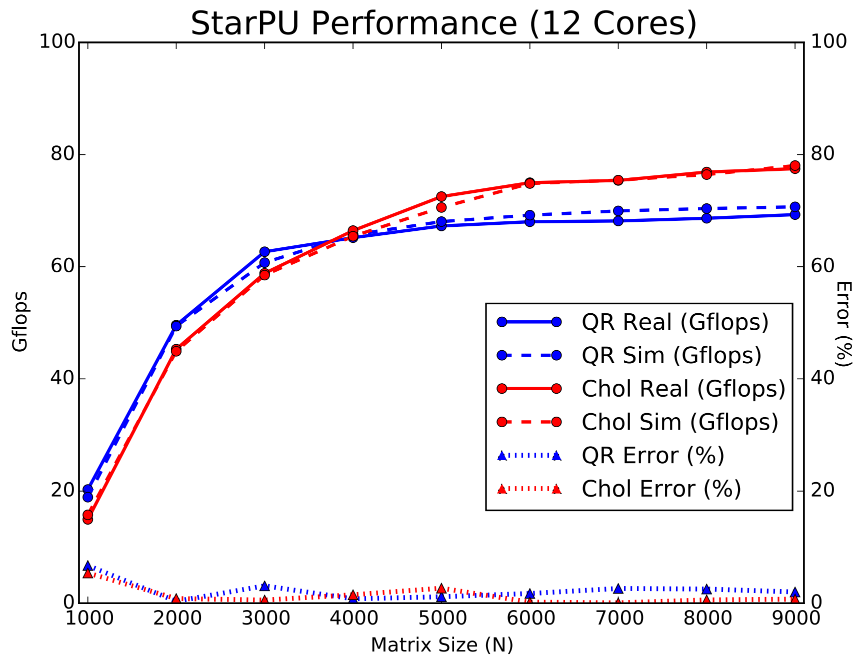



**Figure 4.2:** Cholesky and QR performance results using the StarPU scheduler. $NB = 200$ 12 Core AMD Opteron 6180 SE

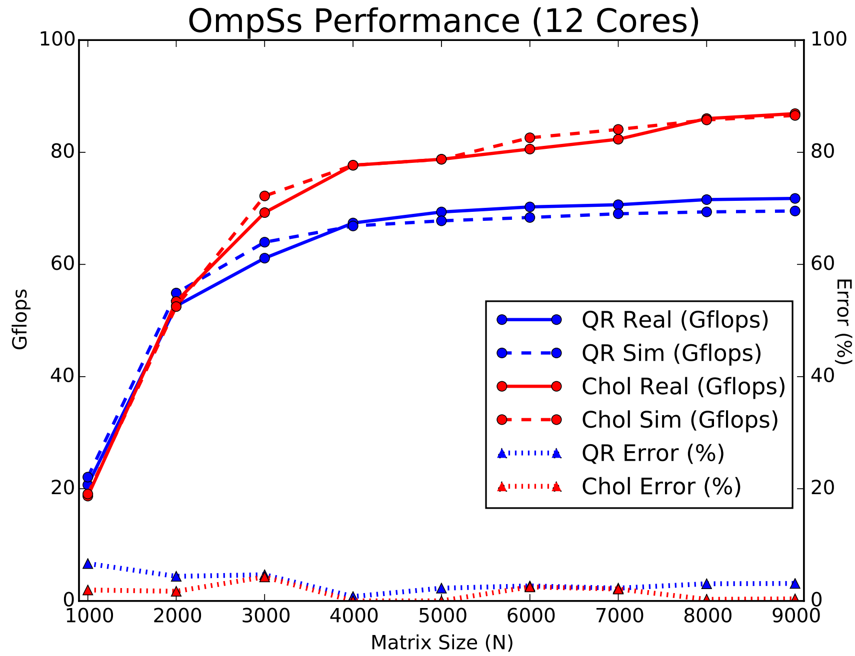



**Figure 4.3:** Cholesky and QR performance results using the OmpSs scheduler. $NB = 200$ 12 Core AMD Opteron 6180 SE

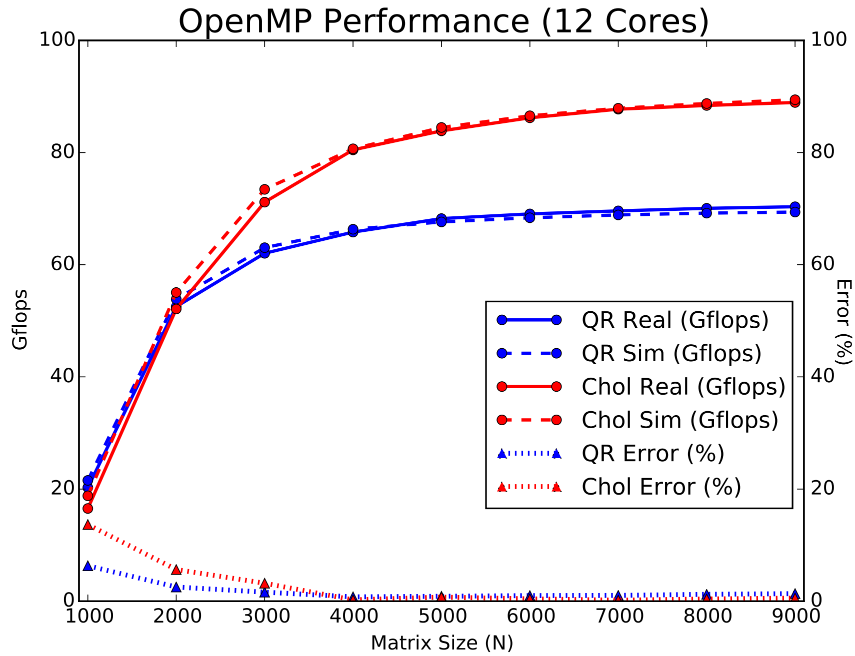



**Figure 4.4:** Cholesky and QR performance results using the OpenMP scheduler implemented in GCC. $NB = 200$ 12 Core AMD Opteron 6180 SE

of OpenMP. Most of the results presented in the remainder of this chapter will use the Intel or GCC OpenMP implementations.

One of the novel contributions of this work is the ability to use the same library across multiple task-based schedulers with little or no modification. In the case of StarPU and SimGrid simulations, the simulation is directly tied to the scheduler. This certainly gives StarPU simulations an advantage in some situations but it does not allow a developer to use the simulations portably with other schedulers. In the case of Prometheus, the DAG for the workload must be extracted in a format recognizable to the task simulator. Prometheus currently only supports Cilk++ and in order to use the simulator with another scheduler the DAG collection tool must be modified to intercept or understand the task structures of another task-based framework.

## 4.2 Simulation Scalability

### 4.2.1 Varying Core Counts

One of the simplifying assumptions of the discrete event simulation presented here is the inclusion of task runtime models and resource contention into a single model. This is, in part, due to the fact that it is difficult, if not impossible, to model the effects contention will have on the runtime of each task. Even if it is possible, the simulations would have to keep track of each piece of data in order to apply the contention model to the length of each task. As a result, the runtime model for each task includes the effects of any resource contention.

Making a single model that includes the basic computational runtime model with contention makes it relatively easy to collect the runtime information from the trace of an execution of the algorithm and use this information to build a model for each class of tasks. The downside to this paradigm is that we must have access to the resources in order to calibrate the task models. It may seem like a single processor could be used to build these performance models. This would be true and provide

accurate results if there were no resource contention. Unfortunately, each task must access data that may or may not be on the local memory on a NUMA system. If a task must access data on non-local memory it will take longer for the task to complete.

The following example demonstrates the impact task modeling can have on the simulation results on a 48 core NUMA machine. The hardware used in this experiment has 4 sockets that each contain a 12 core AMD Opteron 6180SE processor operating at 2.5 GHz. The system has 256 GB of RAM across 8 NUMA nodes. The application being modeled is a QR factorization of a matrix that is $5000 \times 5000$. The matrix is blocked into tiles of $200 \times 200$ resulting in a matrix that is 25 tiles on a side. The inner blocking factor is 40. The workload is identical in every case except the number of cores used in the computation. A trace is collected using 12, 24, 36, and 48 cores corresponding to 1, 2, 3, and 4 sockets respectively.

Generally, memory placement on a NUMA system is based on the "first touch" policy. This policy means portions of the process address space are mapped to the NUMA node closest to the processing element (core) closest to it (assuming there is available space). When PLASMA initializes the matrix, it is done in parallel in order to ensure the tiles in the matrix are distributed across the system memory. This means that if PLASMA is executed using 12 of the 48 cores, the data will only be distributed across the memory closest to the 12 cores performing the data initialization. When the QR factorization is computed on the same 12 cores, the tasks will only access the memory nearby and, as a result, they tend to be slightly faster than if the data was distributed across all 8 NUMA nodes.

In the case where the application uses all 48 cores, the data is initialized and distributed across all 8 of the NUMA nodes. As a result, many of the tasks must access data in non-local memory that is further away or may be in the cache of another processor. Whenever this data is accessed, the tasks take longer to complete because the task must wait for the data.

The four QR factorization traces were used to analyze the performance of each task and build models as an input to the simulation. Tables 4.2, 4.4, 4.6, and 4.8

provide descriptive statistics about each of the four tasks (DGEQRT, DORMQR, DTSQRT, and DTSMQR). As expected, in almost every case the average time for each class of tasks increases with the number of cores. This is explained by the wider distribution of the matrix across the machine. The tasks must more frequently access non-local memory in order to complete the computations. The skewness in most cases (except DGEQRT) tends to be positive meaning the distribution has a wider tail on the right side than the left. This is a common occurrence for task timing distribution. This is likely because occasionally bad cache effects or scheduling and OS jitter can cause a task to be slower. These slower tasks tend to positively skew the timing distributions. Each of the data sets was also tested for normality using the D'Agostino-Pearson normality test. The DORMQR, DTSQRT, and DTSMQR data all had p-values below 0.05 (with the exception of the 24 core DTSQRT data set) suggesting the data is not normally distributed while the data from DGEQRT had p-values above 0.05 so the null hypothesis could not be rejected.

The t-test and Kolmogorov-Smirnov tests were used in order to determine if the difference in the task times caused by the different CPU configurations was statistically significant. In the case of the DGEQRT data set, the t-test was used because the data was assumed to be normal based on the normality test. The t-test suggests the 12 core and 24 core data sets are similar as are the 36 and 48 core data sets. The DORMQR, DTSQRT, and DTSMQR data sets were compared using the Kolmogorov-Smirnov test because the data sets did not pass the test for normality. In all cases the various system configurations (12, 24, 36, and 48 cores) all produced data sets that were different. The p-values for these tests are shown in Tables 4.3, 4.5, 4.7, and 4.9.

Figures 4.5, 4.6, 4.7 ,and 4.8 show the kernel density estimation curves for each of the four classes of task and the four hardware configurations.

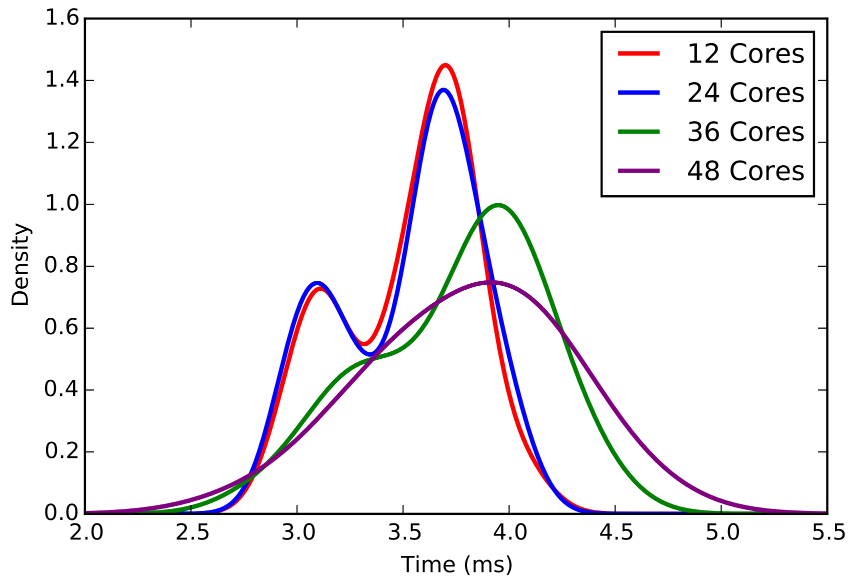



**Figure 4.5:** Kernel Density Estimation curves for DGEQRT tasks

**Table 4.2:** DGEQRT task descriptive statistics (25 Data Points)

| Cores | Min | Max | Mean | Variance | Skewness | Kurtosis | Normality p-value |
|---|---|---|---|---|---|---|---|
| 12 | 2.942 | 4.072 | 3.521 | 0.0964 | -0.42 | -0.94 | 0.30 |
| 24 | 2.950 | 4.033 | 3.520 | 0.1070 | -0.39 | -1.11 | 0.15 |
| 36 | 2.850 | 4.389 | 3.752 | 0.1581 | -0.56 | -0.58 | 0.39 |
| 48 | 2.799 | 4.605 | 3.800 | 0.1828 | -0.35 | -0.34 | 0.70 |

**Table 4.3:** DGEQRT t-test p-values

| | **12 Cores** | **24 Cores** | **36 Cores** | **48 Cores** |
|---|---|---|---|---|
| **12 Cores** | 1.00 | 0.99 | 0.03 | 0.01 |
| **24 Cores** | 0.99 | 1.00 | 0.03 | 0.01 |
| **36 Cores** | 0.03 | 0.03 | 1.00 | 0.68 |
| **48 Cores** | 0.01 | 0.03 | 0.68 | 1.00 |

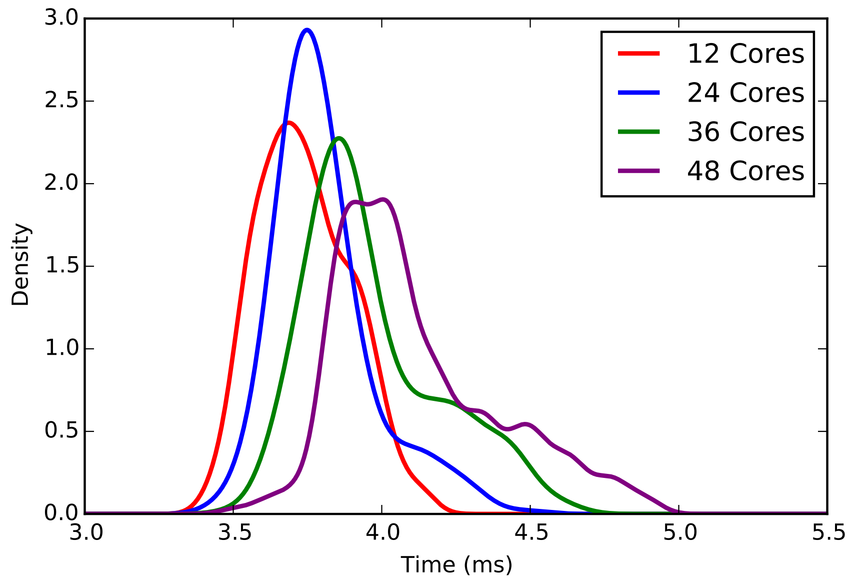



**Figure 4.6:** Kernel Density Estimation curves for DORMQR tasks

**Table 4.4:** DORMQR task descriptive statistics (300 Data Points)

| Cores | Min | Max | Mean | Variance | Skewness | Kurtosis | Normality p-value |
|---|---|---|---|---|---|---|---|
| 12 | 3.422 | 4.149 | 3.740 | 0.0235 | 0.32 | -0.53 | 0.00 |
| 24 | 3.455 | 4.488 | 3.804 | 0.0274 | 1.26 | 1.87 | 0.00 |
| 36 | 3.504 | 4.633 | 3.965 | 0.0513 | 0.83 | -0.07 | 0.00 |
| 48 | 3.499 | 4.909 | 4.115 | 0.0759 | 0.84 | 0.11 | 0.00 |

**Table 4.5:** DORMQR KS test p-values

| | **12 Cores** | **24 Cores** | **36 Cores** | **48 Cores** |
|---|---|---|---|---|
| **12 Cores** | 1.00 | 0.00 | 0.00 | 0.00 |
| **24 Cores** | 0.00 | 1.00 | 0.00 | 0.00 |
| **36 Cores** | 0.00 | 0.00 | 1.00 | 0.00 |
| **48 Cores** | 0.00 | 0.00 | 0.00 | 1.00 |

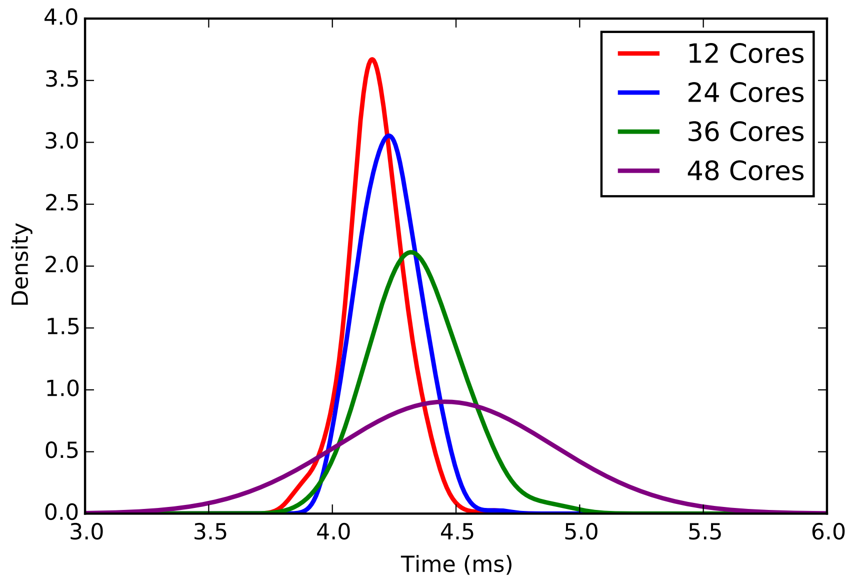



**Figure 4.7:** Kernel Density Estimation curves for DTSQRT tasks

**Table 4.6:** DTSQRT task descriptive statistics (300 Data Points)

| Cores | Min | Max | Mean | Variance | Skewness | Kurtosis | Normality p-value |
|---|---|---|---|---|---|---|---|
| 12 | 3.847 | 4.654 | 4.176 | 0.0129 | 0.08 | 1.14 | 0.01 |
| 24 | 3.990 | 4.668 | 4.230 | 0.0138 | 0.26 | -0.11 | 0.19 |
| 36 | 3.948 | 4.937 | 4.341 | 0.0303 | 0.44 | 0.46 | 0.00 |
| 48 | 3.990 | 10.490 | 4.478 | 0.1605 | 11.29 | 167.24 | 0.00 |

**Table 4.7:** DTSQRT KS test p-values

| | **12 Cores** | **24 Cores** | **36 Cores** | **48 Cores** |
|---|---|---|---|---|
| **12 Cores** | 1.00 | 0.00 | 0.00 | 0.00 |
| **24 Cores** | 0.00 | 1.00 | 0.00 | 0.00 |
| **36 Cores** | 0.00 | 0.00 | 1.00 | 0.00 |
| **48 Cores** | 0.00 | 0.00 | 0.00 | 1.00 |

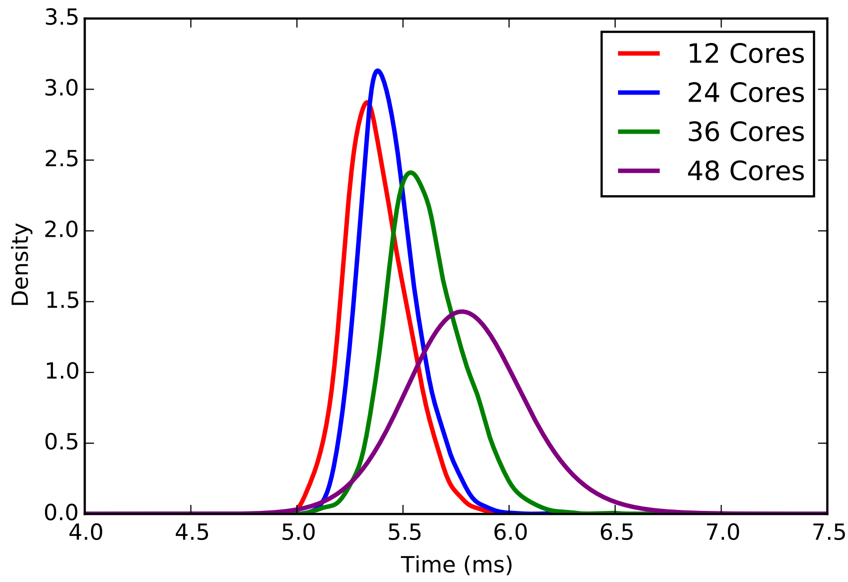



**Figure 4.8:** Kernel Density Estimation curves for DTSMQR tasks

**Table 4.8:** DTSMQR task descriptive statistics (4900 Data Points)

| **Cores** | **Min** | **Max** | **Mean** | **Variance** | **Skewness** | **Kurtosis** | **Normality p-value** |
|---|---|---|---|---|---|---|---|
| 12 | 5.023 | 5.975 | 5.376 | 0.0204 | 0.43 | 0.13 | 0.00 |
| 24 | 5.064 | 6.203 | 5.437 | 0.0185 | 0.64 | 0.73 | 0.00 |
| 36 | 5.067 | 6.506 | 5.605 | 0.0313 | 0.50 | 0.63 | 0.00 |
| 48 | 4.945 | 11.210 | 5.799 | 0.0609 | 3.55 | 60.37 | 0.00 |

**Table 4.9:** DTSMQR KS test p-values

| | **12 Cores** | **24 Cores** | **36 Cores** | **48 Cores** |
|---|---|---|---|---|
| **12 Cores** | 1.00 | 0.00 | 0.00 | 0.00 |
| **24 Cores** | 0.00 | 1.00 | 0.00 | 0.00 |
| **36 Cores** | 0.00 | 0.00 | 1.00 | 0.00 |
| **48 Cores** | 0.00 | 0.00 | 0.00 | 1.00 |

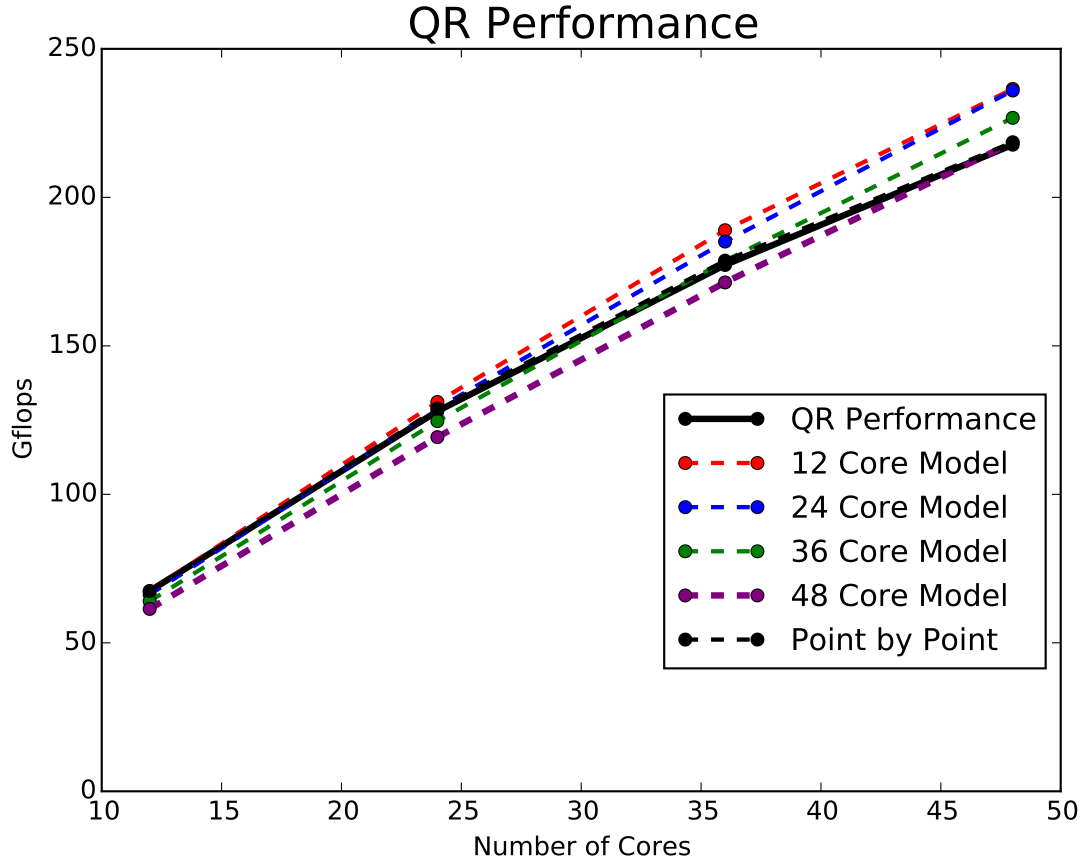



**Figure 4.9:** QR simulation performance results using the OpenMP scheduler implemented in GCC. Each of the dashed lines represent a different set of task models as input. N=5000, NB=200, 4 x 12 Core AMD Opteron 6180 SE

Each of data sets is used as the basis for a set of task models. The task models are fed into the simulation to determine what effect they would have on the accuracy of the simulation results. Although many of the task timing data sets did not pass the normality test, a normal distribution seems to model the tasks closely enough to provide accurate simulation results. Figure 4.9 shows the simulation performance prediction for the applications with each of the input configurations. The solid black line indicates the actual performance of the workload with 12, 24, 36, and 48 cores. The red, blue, green, and purple lines indicate the simulation results based on each of the configuration inputs. For example, the red line is the simulation using the configuration created based on the 12 core data set. The tasks in this data set tend

to be a bit faster because of the memory access patterns. If this configuration is used to simulate the entire 48 core system, it will overpredict the performance of the workload. The purple line, on the other hand, represents the simulation results from a configuration created with a 48 core data set. In this case the simulation tends to underpredict the performance for a smaller number or cores. The figure also has a black, dashed line labeled as “Point by Point”. This line represents the simulation results when the correct configuration file is used for each data point. It is difficult to see this line because it is generally on top of the solid black line indicating the simulation error is near zero.

### 4.2.2 Varying Task Granularity

In the case of tile-based linear algebra, a decrease in task granularity generally increases parallelism but can reduce the efficiency of each task. When the tasks get too small, they can often overwhelm the scheduler. In order to quantify the accuracy of the simulations, this section will examine the accuracy of the simulations for a Cholesky factorization as the task size decreases. The Cholesky factorization was chosen because the most common task is a DGEMM which is a highly optimized BLAS operation that can be computed very quickly for even moderate size problems. When the tile size decreases, the time for each task quickly drops and increasingly stresses the scheduler. The tests presented here were performed on a 4 socket machine with 12 core AMD Operon 6180 SE processors. The tests were performed once with all 48 cores (Figures 4.10 and 4.11) and a second time using only a single socket containing 12 cores (Figures 4.12 and 4.13). The differing results for these two scenarios will demonstrate the interaction of the task granularity and number of cores and the effects they can have on the accuracy of the simulations. The tests will also be performed using the discrete event simulation mode (Figures 4.10 and 4.12) as well as the spin simulation mode (Figures 4.11 and 4.13).

When evaluating the simulation accuracy for all 48 cores shown in Figures 4.10 and 4.11, it appears that both simulation modes provide accurate simulations for the larger tile sizes of 160 and 192. Once the tile size decrease to 128, the two simulation modes begin to diverge in accuracy. The Spin simulation accurately predicts the performance while the discrete event simulation has begun to overpredict the performance of the factorization. This is likely due to the fact that the DES does not account for any scheduling overhead that occurs with these smaller tasks on a large 48 core machine. Once the tile size decreases to 96, both simulation modes significantly overpredict the performance of the algorithm. It should be noted that at this tile size, the scheduler struggles to keep up even when executing the real workload. For example, when factorizing a matrix where $N = 4800$ and $NB = 96$ 100 times, the performance achieved using OpenMP varies from 38.69 Gflops to 141.98 Gflops. The average DGEMM task in this case is 0.399 ms and the other tasks take even less time. At this granularity and hardware configuration, the scheduler has broken down as well as the simulations.

The same workloads were also performed using only one socket containing 12 cores. These simulations provide much more accurate results than the previous examples even though they have the same task granularity. In fact, the decreased distribution of data in the 12 core scenario actually decreases the average task time for the same granularity. For example, the same DGEMM task that took 0.399 ms on average for the 48 core configurations only requires 0.249 ms for the 12 core configuration. However, with only 12 cores to schedule the runtime is not stressed as much which results in smaller scheduling overhead and increased simulation accuracy. The simulations still overpredict for small tile sizes but the error isn't nearly as large as the error in from the 48 core configuration.

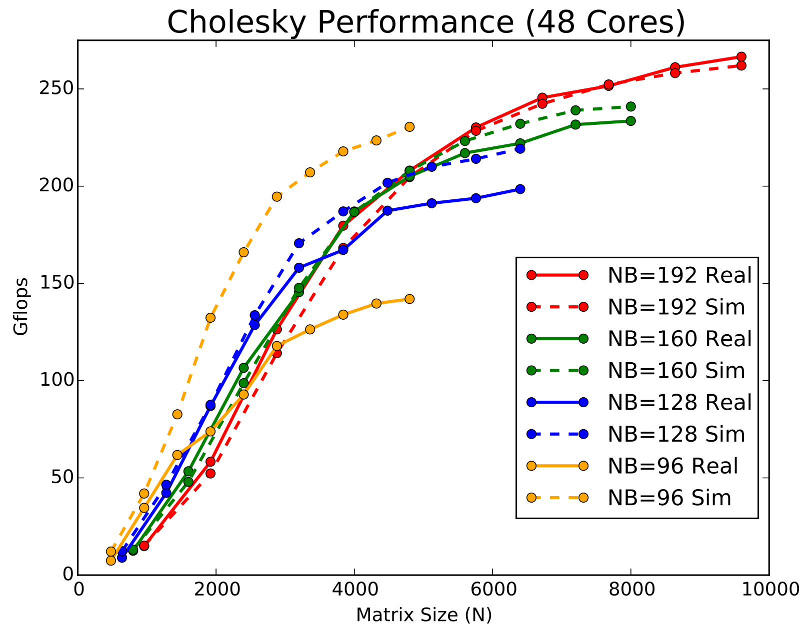



**Figure 4.10:** Cholesky discrete event simulation performance results using the OpenMP scheduler implemented in GCC. The simulation accuracy decreases as the size of each tile decreases. 4 x 12 Core AMD Opteron 6180 SE

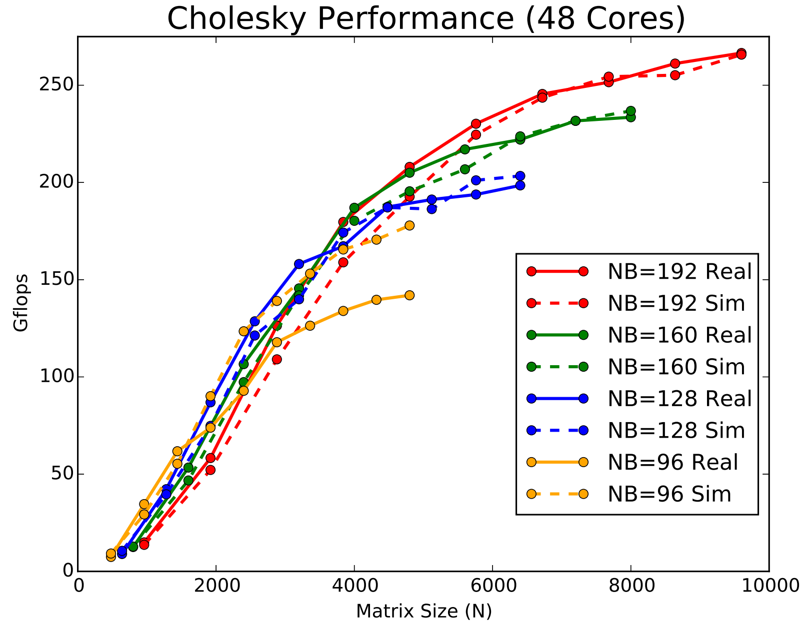



**Figure 4.11:** Cholesky Spin simulation performance results using the OpenMP scheduler implemented in GCC. The simulation accuracy decreases as the size of each tile decreases. 4 x 12 Core AMD Opteron 6180 SE

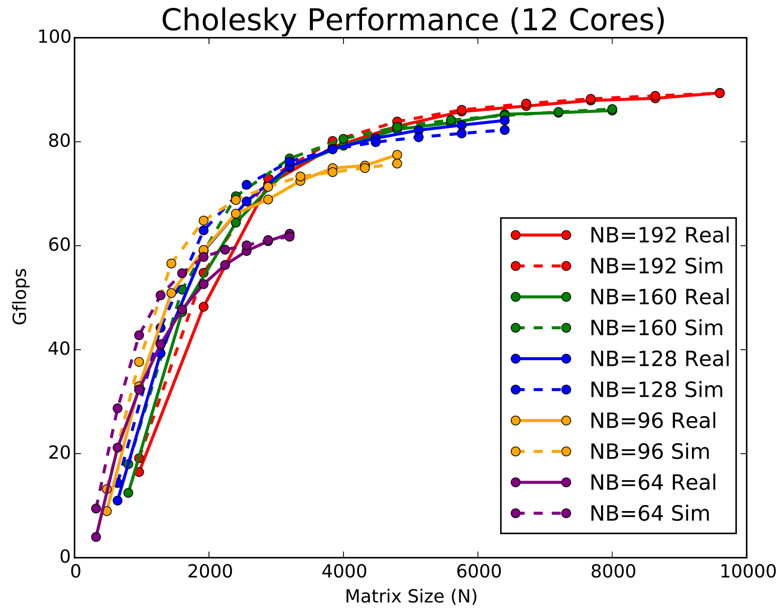



**Figure 4.12:** Cholesky discrete event simulation performance results using the OpenMP scheduler implemented in GCC. The simulation accuracy still decreases with the smaller tile sizes but it isn't as drastic as for a larger number of cores. 12 Core AMD Opteron 6180 SE

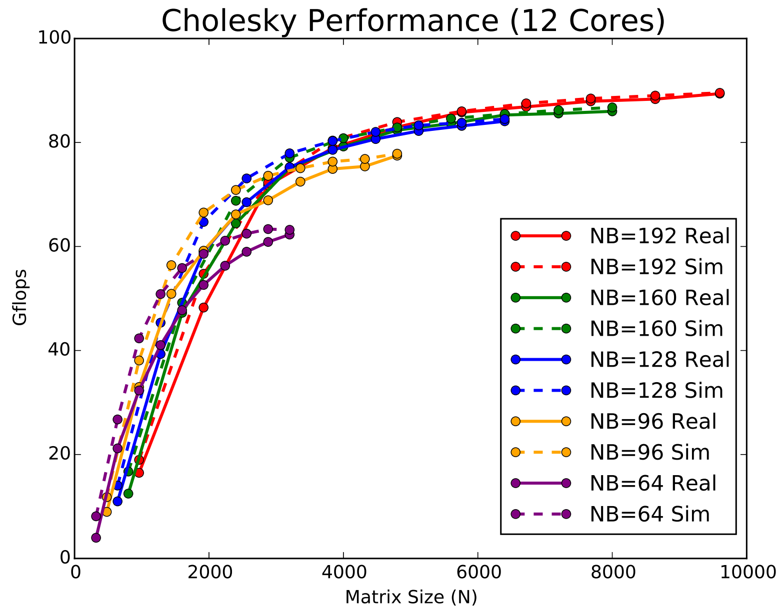



**Figure 4.13:** Cholesky spin simulation performance results using the OpenMP scheduler implemented in GCC. The simulation accuracy still decreases with the smaller tile sizes but it isn't as drastic as for a larger number of cores. 12 Core AMD Opteron 6180 SE

## 4.3 LU Simulation

One of the original assumptions for the simulation was that each class of tasks was fairly uniform in terms of computation time. This assumption was that each task performed the same number of operations and the difference in task times was relatively small. It was also assumed that the temporal location of the tasks was independent of the task time. Unfortunately, this assumption does not hold true for all workloads.

One such example is the LU factorization implemented in PLASMA (using a single threaded task to compute the panel factorization). The LU factorization, in order to perform proper pivoting, must operate on an entire column of tiles during the panel factorization. The first panel includes all of the rows in the first column of tiles. Each subsequent panel factorization requires one less tile for the panel factorization. As a result, the panel factorizations decrease in the time required throughout the algorithm. This phenomenon can be seen in Figure 4.16. The first panel factorization (shown in orange) is relatively long but each of the following panel factorizations decreases in time.

In order to account for this type of problem, the developer can specify a custom task time for the simulator. In the case of the LU factorization, it is relatively easy to represent the time for the panel factorization as a linear relationship.

$$time = m \times rows + b$$

Figure 4.14 includes the data points for the LU factorization using two different tile sizes. In each case, a linear relationship seems to accurately model the task times (in ms). In the case of $NB = 200$ the following equation describes the line of best fit:

$$time = 0.00364 \times rows - 0.235$$

In the case of $NB = 128$ the following equation describes the line of best fit:

$$time = 0.00187 \times rows + 0.084$$

Using these formulas to calculate a custom time for each panel factorization ensures this task is accurately modeled throughout the simulation. This can be seen by comparing Figure 4.16 and Figure 4.17 showing the trace of a real workload and a simulated workload respectively. While these two traces are not identical, they show many of the same characteristics including the decreasing time for the panel factorization throughout the workload. The two traces also use the same length along the x-axis to demonstrate the simulated workload almost perfectly matches the runtime of the real algorithm.

Figure 4.15 shows the performance curves for the LU factorization using these two tile sizes across a range of matrix sizes. For the case where the tile size is 200, the simulated performance and the real performance are very similar. The same is true for the case where $NB = 128$ until the matrix gets a bit larger. At this point, the simulated results start to diverge from the real workload. A close examination of the traces where the simulation accuracy decrease reveals the act of simulating the workload has changed the scheduling behavior and has violated one of our simulation assumptions.

Figures 4.18 and 4.19 correspond to a real and simulated workload of an LU factorization of a matrix where $N = 6400$ and $NB = 128$. In the real workload shown in Figure 4.18 very few tasks are scheduled on the first core for the first third of the trace. This is likely because this core is in charge of "inserting" tasks and performing the necessary scheduling overhead. As a result, the panel factorizations often performed on the first core, are moved to another core and cause the scheduler to make different scheduling choices. These panel factorizations seem to become a bottleneck in this case and do not overlap with the other tasks nearly as well as they do later in the trace. In the simulated case shown in Figure 4.19, the change in the

speed of the execution of the simulation relative to the real workload causes the tasks to generally stay on the first core and overlap with the other tasks. As a result, the simulation would suggest results that are faster than they actually are. Figures 4.20 and 4.21 demonstrate the LU factorization with the same number of tiles, but a large tile size. In this case the artifact described earlier does not appear due to the different tile sizes.

It is likely this scheduling artifact that can be reduced in the future when task priority constructs are available in OpenMP. In this case, the panel factorization will be given priority and should be overlapped with many of the other tasks and the resulting increased workload performance and simulation accuracy.

The ability to specify a custom task time for the simulations allows for accurate simulations of the LU factorization but may also be useful for simulating other workloads where the tasks are not uniform in size.

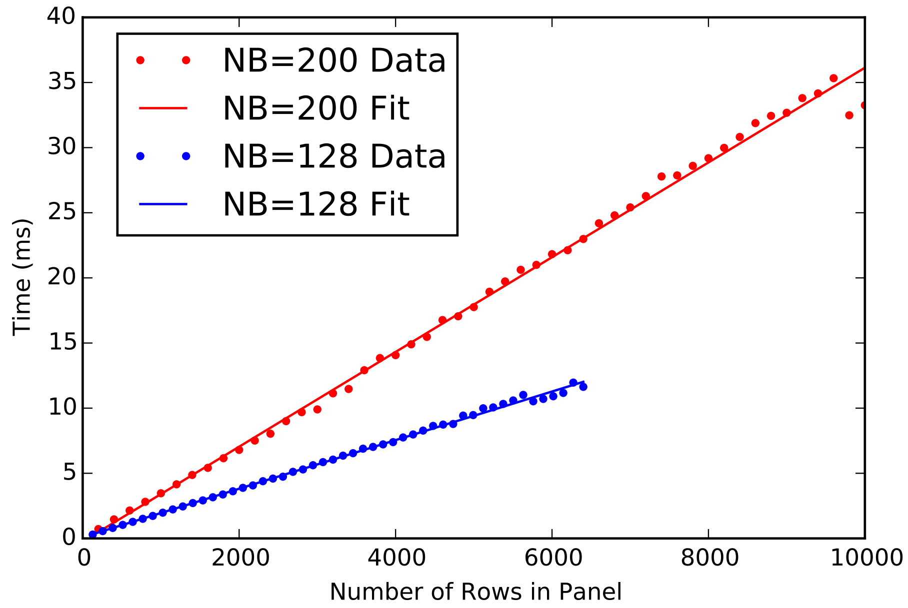



**Figure 4.14:** The time of each panel is plotted against the number of rows in the tile column being factorized. The data points are plotted with their line of best fit. 2 x 8 Core Intel Xeon E5-2690

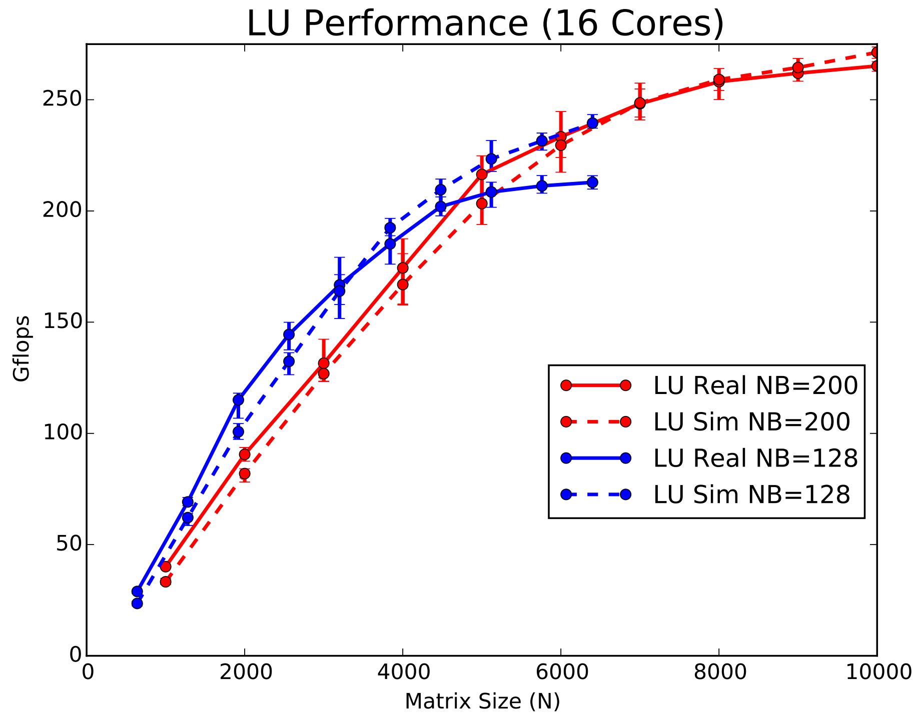



**Figure 4.15:** LU Factorization simulated performance vs Real LU Factorization performance. Implemented in OpenMP using GCC. 2 x 8 Core Intel Xeon E5-2690

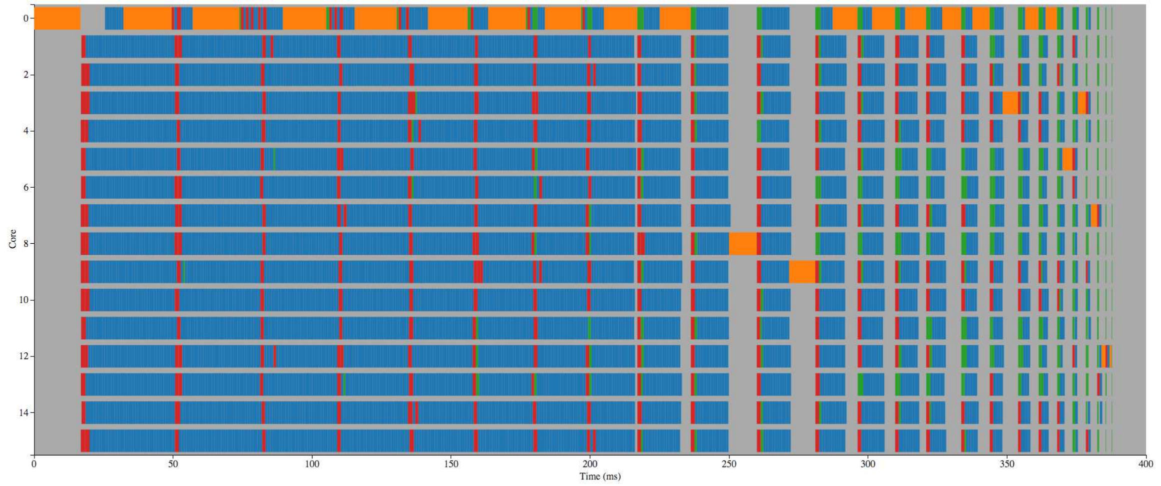



**Figure 4.16:** A real trace of an LU Factorization where N=5000 and NB=200. 2 x 8 Core Intel Xeon E5-2690

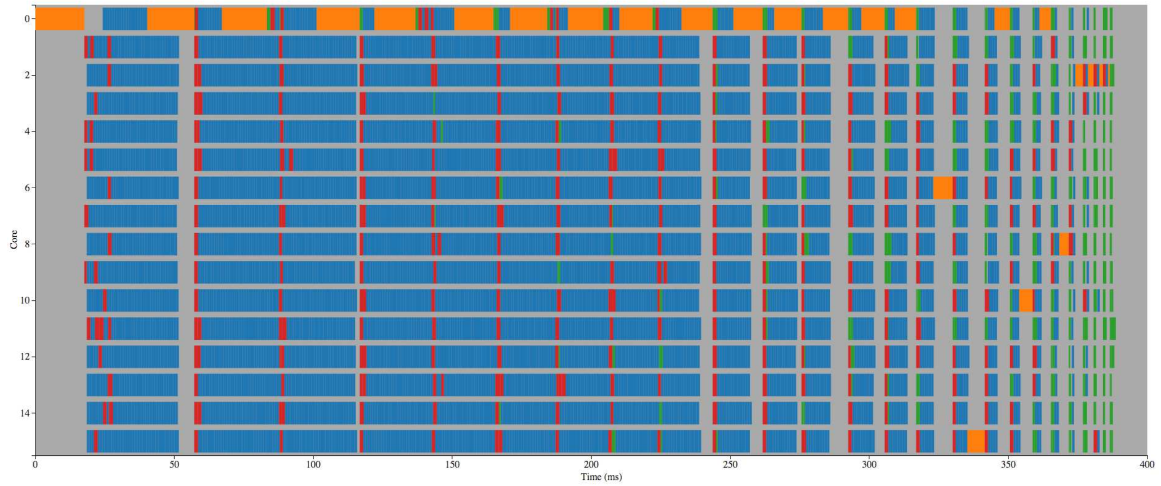



**Figure 4.17:** A simulated trace of an LU Factorization where N=5000 and NB=200. 2 x 8 Core Intel Xeon E5-2690

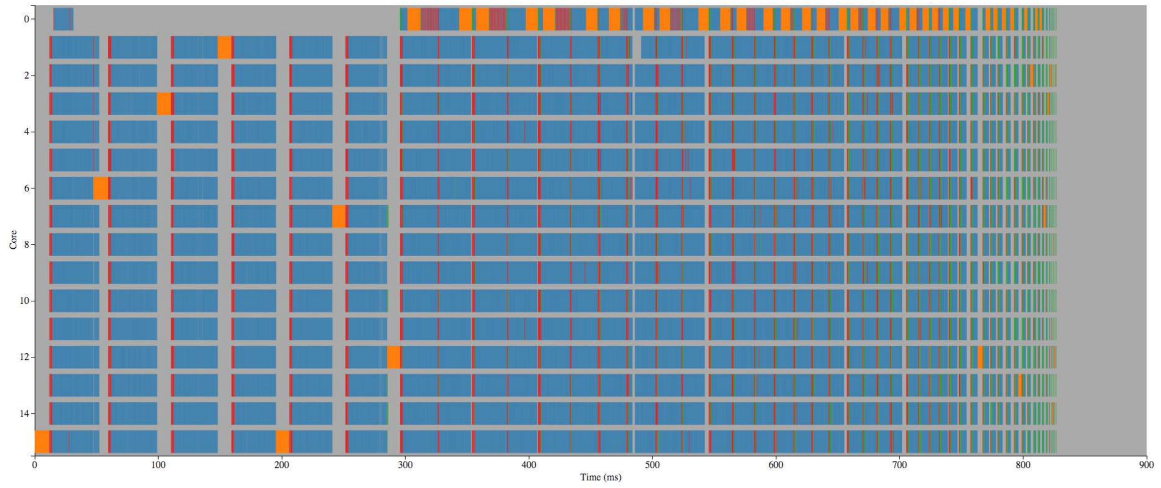



**Figure 4.18:** A real trace of an LU Factorization where N=6400 and NB=128. 2 x 8 Core Intel Xeon E5-2690

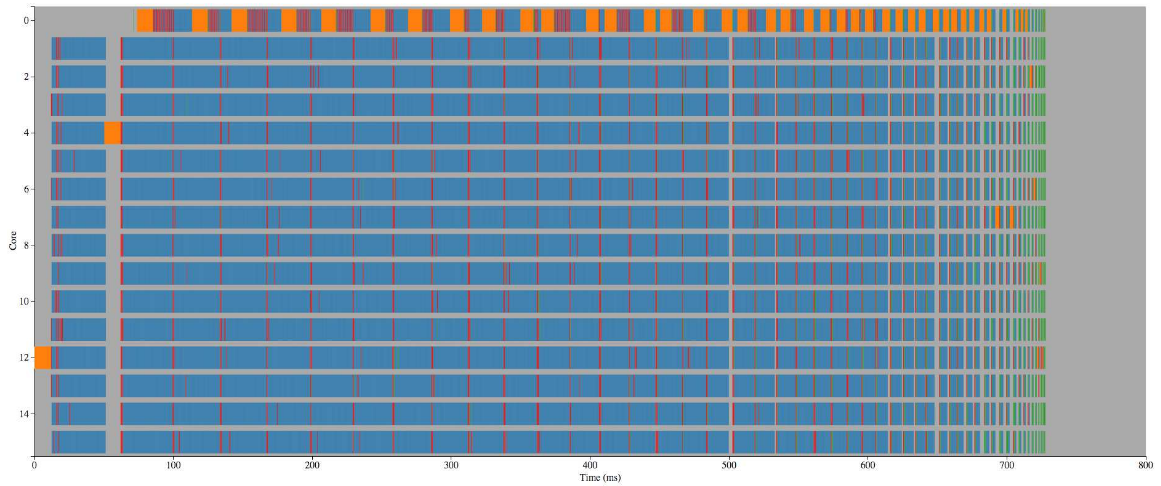



**Figure 4.19:** A simulated trace of an LU Factorization where N=6400 and NB=128. 2 x 8 Core Intel Xeon E5-2690

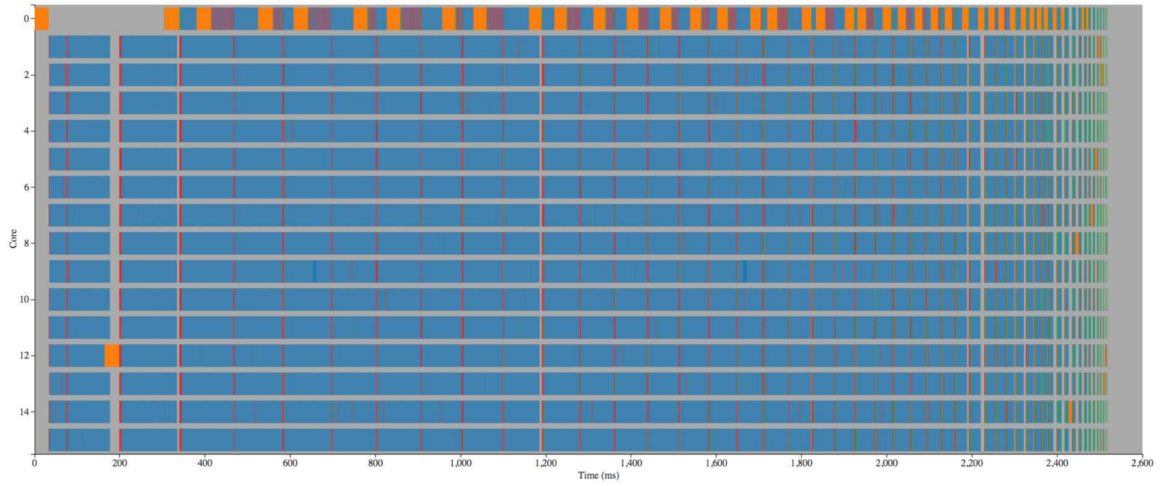



**Figure 4.20:** A real trace of an LU Factorization where N=10000 and NB=200. 2 x 8 Core Intel Xeon E5-2690

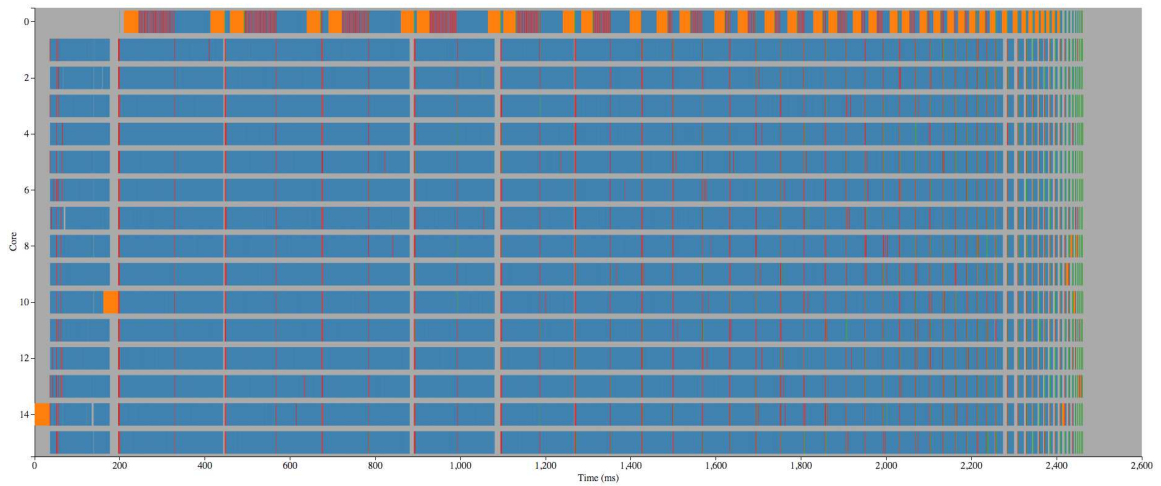



**Figure 4.21:** A simulated trace of an LU Factorization where N=10000 and NB=200. 2 x 8 Core Intel Xeon E5-2690

## 4.4 Intel Xeon Phi Cholesky Simulation

In serial applications and many fork-join parallel workloads, it is relatively easy to determine when and where to optimize portions of the code. It should also be assumed that if one phase of the computation is faster, likewise the overall runtime will be faster. In the case of a task-based runtime, this analysis and performance modeling is not particularly easy. For example, one task may be fairly slow, but it is usually overlapped with other computations occurring on the other cores. In this case, even if the task is improved dramatically, the total runtime for the algorithm may not change significantly. In fact, optimization may not change the runtime of the computation at all. However, you may have a task that is not called many times but often creates a bottleneck for the computation. In this case, optimization may dramatically improve the performance of the algorithm.

In many cases, each of the tasks can be improved but it can be a time-consuming process. Before investing the time and resources in code optimization, it can be helpful to determine whether an optimized task implementation will even accelerate the workload. This is an excellent application for the task simulator.

One example where this can be used is in the work of porting the tile-based Choleskey factorization in PLASMA to run on the Intel Xeon Phi. In this case, a 61 core Intel Xeon Phi 7120 was used. Once the tile-based Cholesky factorization was ported to OpenMP it was trivial to run it on the new hardware. The performance of the factorization, however, was far from the theoretical peak rate for floating point operations for the Xeon Phi. The theoretical peak for a machine can be calculated as follows:

$$Gflops = ClockRate \times \frac{Flops}{ClockCycle} \times NumberOfCores$$

**Table 4.10:** The number of floating point operations for each task in a Cholesky factorization. $m = 256$, $n = 256$, $k = 256$.

| **Kernel** | **Flops Formula** | **Total Flops** |
|---|---|---|
| DGEMM | $2mnk$ | 33554432 |
| DTRSM | $nm^2$ | 16777216 |
| DSYRK | $kn(n+1)$ | 16842752 |
| DPOTRF | $\frac{1}{3}n^3 + \frac{1}{2}n^2 + \frac{1}{6}n$ | 5625216 |

**Table 4.11:** Cholesky Task Performance on 61 core Intel Xeon Phi 7120

| **Kernel** | **Flops** | **Mean Time (ms)** | **Gflops** | **% Peak** | **% GEMM** |
|---|---|---|---|---|---|
| DGEMM | 33554432 | 3.854 | 8.707 | 43.96% | 100.00% |
| DTRSM | 16777216 | 2.253 | 7.447 | 37.60% | 85.52% |
| DSYRK | 16842752 | 5.282 | 3.189 | 16.10% | 36.62% |
| DPOTRF | 5625216 | 8.818 | 0.638 | 3.22% | 7.33% |

For double precision floating point operations on the Intel Xeon Phi 7120 the following represents the theoretical peak:

$$1208\ Gflops = 1.238(GHz) \times 16(\frac{Flops}{ClockCycle}) \times 61(Cores)$$

or 19.808 Gflops per core. However, the Cholesky factorization using a tile size of 256 only reaches a few hundred Gflops. One of the primary challenges in optimizing any workload is determining where optimization can be applied most effectively.

The starting point for an analysis of task performance is to compute the number of floating point operations for each task. The formulas and calculations for each of these tasks are shown in Table 4.10. These values are calculated using $m = 256$, $n = 256$, and $k = 256$. These operation counts, along with the average task time, are used in Table 4.11 to calculate the flop rate for each of the task types. The table also includes the percentage of the theoretical peak for a single core. It is nearly impossible to reach the theoretical peak performance for any workload. As a result,

matrix multiplication (GEMM) is often used as an artificial benchmark for how much code can be optimized on a specific architecture. For this reason, the percentage of the GEMM performance is also calculated.

Obviously, none of the tasks reaches the theoretical peak performance but it is unclear whether we should expect better performance. As a baseline, a similar analysis was done by analyzing the performance of each task on a 10 core Intel Xeon E5-2650 v3 (Haswell) with the following theoretical peak performance:

$$368\ Gflops = 2.3(GHz) \times 16(\frac{Flops}{ClockCycle}) \times 10(Cores)$$

or 36.8 Gflops per core. The performance for each of the tasks on the Haswell architecture is shown in Table 4.12.

When comparing the performance of each of the tasks between the two architectures it becomes obvious the tasks do not perform nearly as well on the Xeon Phi as they do on the Haswell. For example, the DGEMM tasks on the Haswell perform at more that 83% of the theoretical peak while they only perform at 44% of the theoretical peak on the Xeon Phi. However, it is likely unfair to make comparisons based on the percentage of theoretical peak because the architectures are very different and provide different challenges for optimizing code. In order to make a more reasonable comparison, the performance of the DGEMM task is used as the baseline performance. With this baseline, it appears the DTRSM tasks are optimized equally well on the two architectures (82.19% of GEMM performance vs. 85.52% of GEMM performance). However, the DSYRK and DPOTRF tasks do not appear to be very well optimized when they are compared with their counterparts on the Haswell. This would suggest these are the two kernels we might want to examine first in the process of optimizing the Cholesky factorization.

The first step was to validate our simulation is accurate in simulating this workload and architecture. This can be seen in the black line and dashed black line in

**Table 4.12:** Cholesky Task Performance on 10 core Intel Xeon E5-2650 v3

| Kernel | Flops | Mean Time (ms) | Gflops | % Peak | % GEMM |
|---|---|---|---|---|---|
| DGEMM | 33554432 | 1.098 | 30.560 | 83.04% | 100.00% |
| DTRSM | 16777216 | 0.668 | 25.116 | 68.25% | 82.19% |
| DSYRK | 16842752 | 0.680 | 24.769 | 67.31% | 81.05% |
| DPOTRF | 5625216 | 0.427 | 13.180 | 35.82% | 43.13% |

Figure 4.22. The simulation provides accurate results when the task models are built based on the real factorization task times.

In order to determine what effects a faster task would have on the overall runtime, the DPOTRF and DSYRK task models were replaced with task models that would represent a task equally well-optimized for the Haswell and Xeon Phi. These modeling calculations are shown in Table 4.13. Notice the DGEMM and DTRSM models are unchanged in the "improved" portion of the table.

Figure 4.22 shows the expected performance based on the simulations using these new models. The plot shows the performance when only the DPOTRF model is improved, only the DSYRK model is improved, and when both are improved. It should be noted that optimizing the DPOTRF kernel seems to improve the performance more than the DSYRK kernel. It is useful to examine the simulated traces in order to understand why this might be.

Figure 4.23 is the trace of the real execution of a Cholesky factorization on a matrix where $N = 5120$ and $NB = 256$. Using the models derived from the real workload, a simulation of the application is shown in Figure 4.24. The DPOTRF and DSYRK tasks are shown in orange and green respectively across all of these traces. All of the traces use the same scale on the x-axis to show the differences in runtime for the application in each case. Figure 4.25 is a simulated trace where the original model for the DSYRK kernel is replaced with a faster task model. The resulting trace demonstrates that the workload takes less time to complete but not significantly so. This is likely due to the fact that these tasks are often overlapped with other tasks and

**Table 4.13:** The improved DSYRK and DPOTRF kernel performance used in the Cholesky Simulations.

| | **Kernel** | **Mean Time (ms)** | **Gflops** | **% Peak** | **% GEMM** |
|---|---|---|---|---|---|
| **Original** | DGEMM | 3.854 | 8.707 | 43.96% | 100.00% |
| | DTRSM | 2.253 | 7.447 | 37.60% | 85.52% |
| | DSYRK | 5.282 | 3.189 | 16.10% | 36.62% |
| | DPOTRF | 8.818 | 0.638 | 3.22% | 7.33% |
| **Improved** | DGEMM | 3.854 | 8.707 | 43.96% | 100.00% |
| | DTRSM | 2.253 | 7.447 | 37.60% | 85.52% |
| | DSYRK | 2.387 | 7.057 | 35.63% | 81.05% |
| | DPOTRF | 1.498 | 3.755 | 18.96% | 43.13% |

do not create a large bottleneck in the application. On the other hand, Figure 4.26 demonstrates the workload is significantly faster when the DPOTRF kernel is replaced with a faster task model. This is likely due to the fact that the DPOTRF task is often a bottleneck that must be completed before the computation can continue. It is also common for the DPOTRF task to not be overlapped with many tasks in the trace. Finally, Figure 4.27 demonstrates the expected performance if both of the tasks were replaced with an optimized versions.

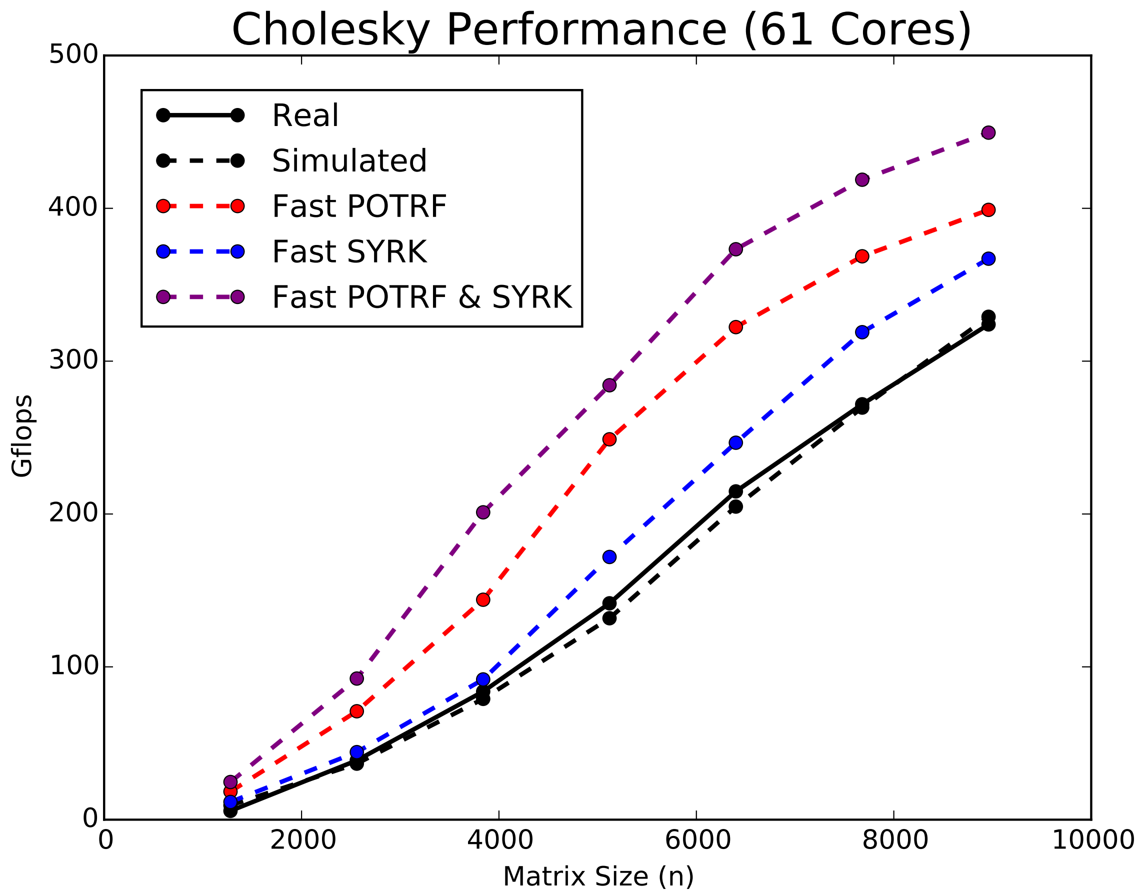



**Figure 4.22:** Performance of Cholesky Factorization where NB=256 implemented with Intel OpenMP. The black line indicates the real performance and the black, dashed line indicates the simulated performance. The red, blue and purple lines indicate the performance expected if the DPOTRF and DSYRK kernels were optimized to get better performance. 61 Core Intel Xeon Phi 7120

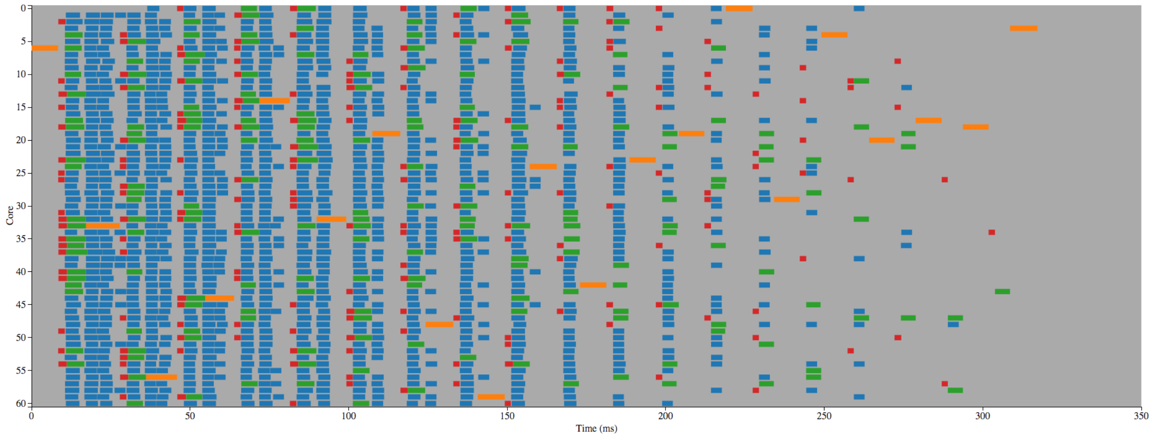



**Figure 4.23:** A real trace of a Cholesky Factorization where N=5120 and NB=256. 61 Core Intel Xeon Phi 7120

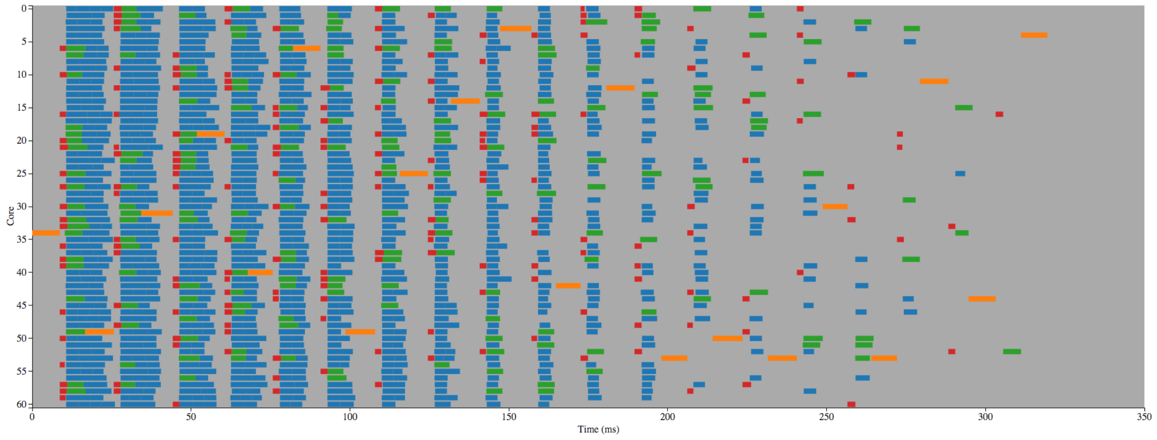



**Figure 4.24:** A simulated trace of a Cholesky Factorization where N=5120 and NB=256. 61 Core Intel Xeon Phi 7120

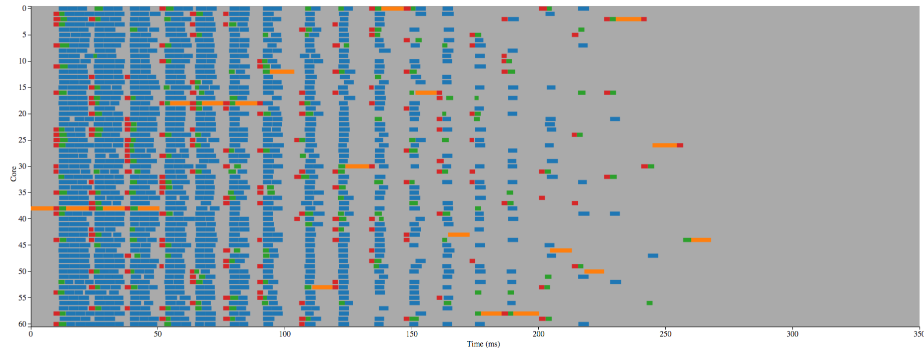



**Figure 4.25:** A simulated trace of a Cholesky Factorization where N=5120 and NB=256. The DSYRK (shown in green) has been accelerated in the simulation. 61 Core Intel Xeon Phi 7120

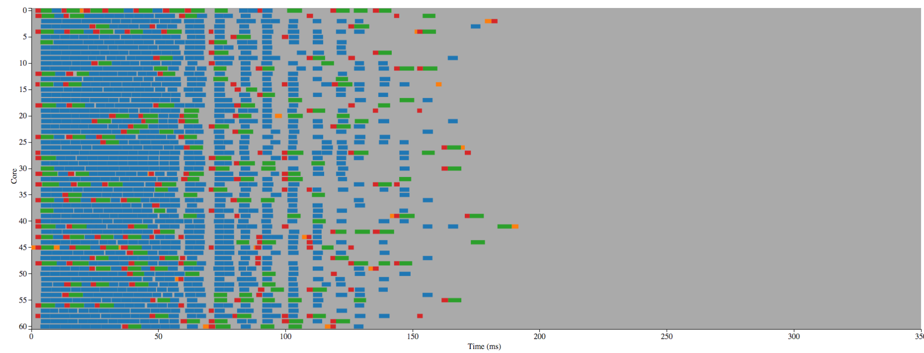



**Figure 4.26:** A simulated trace of a Cholesky Factorization where N=5120 and NB=256. The DPOTRF (shown in orange) has been accelerated in the simulation. 61 Core Intel Xeon Phi 7120

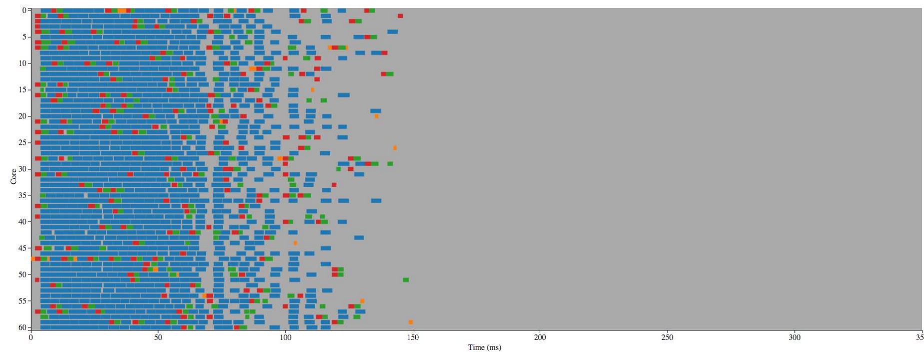



**Figure 4.27:** A simulated trace of a Cholesky Factorization where N=5120 and NB=256. The DPOTRF and DSYRK (shown in orange and green respectively) have been accelerated in the simulation. 61 Core Intel Xeon Phi 7120

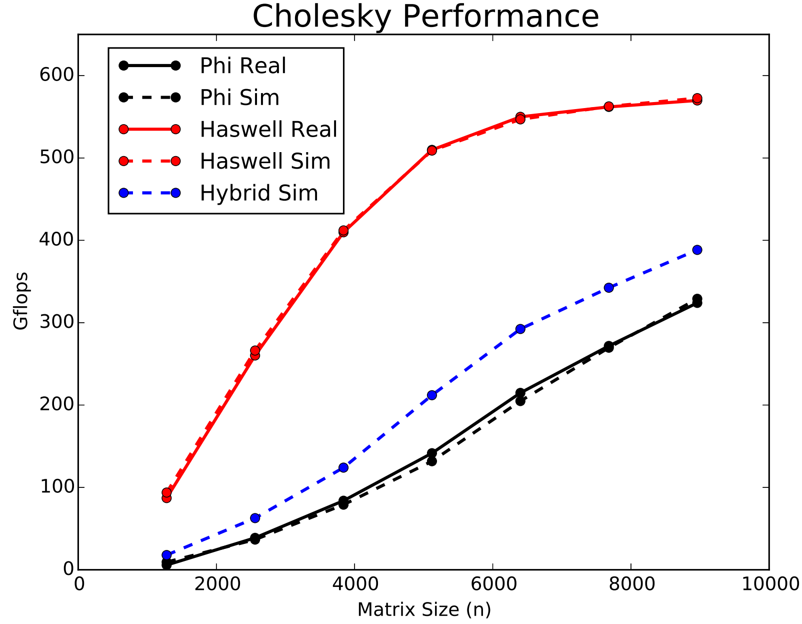



**Figure 4.28:** Real and simulated Cholesky performance on a two intel architectures (Red = 2 x 10 core Intel Xeon E5-2650 v3 & Black = 61 Core Intel Xeon Phi 7120) and a hypothetical hybrid architecture (9 Haswell Cores + 31 Xeon Phi Cores). NB = 256

The simulator could also serve as a tool for studying hardware configurations that are non-existent or difficult to access. For example, in this experiment we will create a new hypothetical architecture and attempt to determine what kind of performance the Cholesky factorization will achieve. Our hypothetical architecture will consist of a combination of Intel Haswell cores and Intel Xeon Phi cores. The current Haswell processors are available in configurations consisting of 2 to 18 cores. For the purposes of our hypothetical architecture, we will use 9 Haswell cores or half of the largest Haswell chip. The Intel Xeon Phi 7120 contains 61 cores. The hypothetical processor will take half of that number (rounded up) or 31 cores. The hypothetical hybrid processors consists of half of a Xeon Phi chip and half of a Haswell chip with a total of 40 cores. It is possible to use the simulator to estimate the performance of our factorization even thought this chip does not and will likely never exist.

In order to simulate this architecture, two distinct models for tasks must be defined for the simulator. The first 9 cores of the architecture will execute tasks

that correspond to the model obtained from a Haswell processor. The remaining 31 cores will use the model obtained from the Xeon Phi. These results are show in Figure 4.28. The solid red line is obtained on a machine with 2 x 10 core Intel Xeon E5-2650 v3 and the red dashed line verifies that the simulations accurately model this architecture. The black solid line was obtained on an Intel Xeon Phi 7120 and the black dashed line verifies the simulation accuracy for this architecture. The blue dashed line is the simulation of our hypothetical hybrid architecture based on the task timing models from the other two machines.

The ability to predict the performance on new hardware can be useful when making design or purchasing decisions for new architectures. In this example, it is possible to get performance models for each of the tasks from preexisting hardware, however these models could be obtained by using a cycle accurate simulation. It may also be possible to run a number of possible models in order to get an rough idea of how the software would perform on a new machine or hardware. For example, it might be interesting to determine how much slower a computation would be if one of the chips was experiencing reduced performance due to power capping. A few quick calculations could adjust the task timing models to be 5%, 10%, 15%, or 20% slower. Once the new models are used for the slower chip, the simulator can quickly give an estimate of the performance without any adjustments to the hardware. This low cost, low risk, reduced hassle performance estimation enabled by simulation can allow developers to ask questions that would not have been possible before.

## 4.5 Reverse Trace Performance Modeling

One of the most common tuning problems for task-based scheduling is the task granularity selection. If the problem is broken down into large tasks, each task generally performs well because it uses all of the cache available to it. However, the large task size results in a decrease in parallelism that can cause a decrease in performance for smaller problems. With small task sizes, each task generally does not

perform as well and scheduling overhead can become a performance issue. However, these small task sizes often create more parallelism that can actually boost the overall performance for small problems.

When selecting the task granularity for tile-based dense linear algebra, there is not one single tile size that provides the greatest performance for all possible matrices. Rather, the optimal tile size depends on the architecture and the size of the matrix. For example, for a smaller problem it is often better to select a smaller tile size in order to make use of all of the cores available on a machine. However, larger matrices often achieve higher performance with a larger tile size. This phenomenon can be seen in Figures 4.29 and 4.30.

It is challenging to build a model that accurately describes the performance of an algorithm based on the task granularity. As a result, empirical tuning approaches are often the best option. Agullo et al. [3] described one such approach to tune the dense QR factorization for multicore architectures. The difficulty with an empirical approach is the length of time it can take to tune an algorithm. It generally involves executing an algorithm several times with varying tile sizes in order to obtain performance curves for each tile size. The optimal tile size changes with the crossover points of the performance curves.

The process of obtaining these performance curves, while simple, can take a large amount of time. One of the methods to deal with this in some problems is a technique called Reverse Trace Performance Modeling (RTPM). Given the structure of the computation, it is possible to obtain an approximation of the performance curve from one large matrix factorization. Given a matrix size and tile size, it is possible to calculate the number of tasks for a workload. These tasks are selected from the end of a large trace and approximate the runtime for the algorithm. This method has been shown to work when using a real trace to perform the analysis. The results shown here suggest that the method also works for simulated traces.

The experiment here is performed on a machine with two 8 core Intel Xeon E5-2690 processors. The workloads are the tile-based Cholesky and QR factorizations

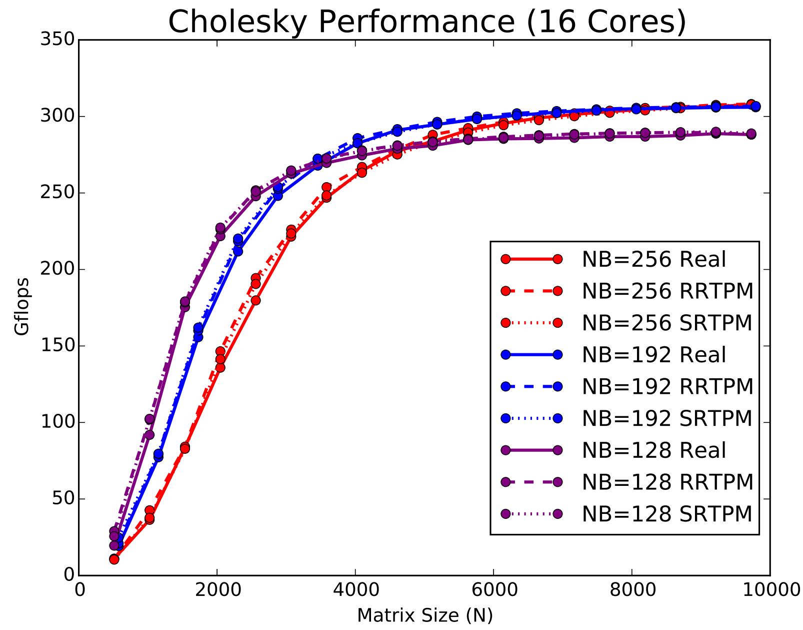



**Figure 4.29:** Cholesky performance predicted by RTPM of a real trace (RRTPM) and a simulated trace (SRTPM) 2 x 8 Core Intel Xeon E5-2690

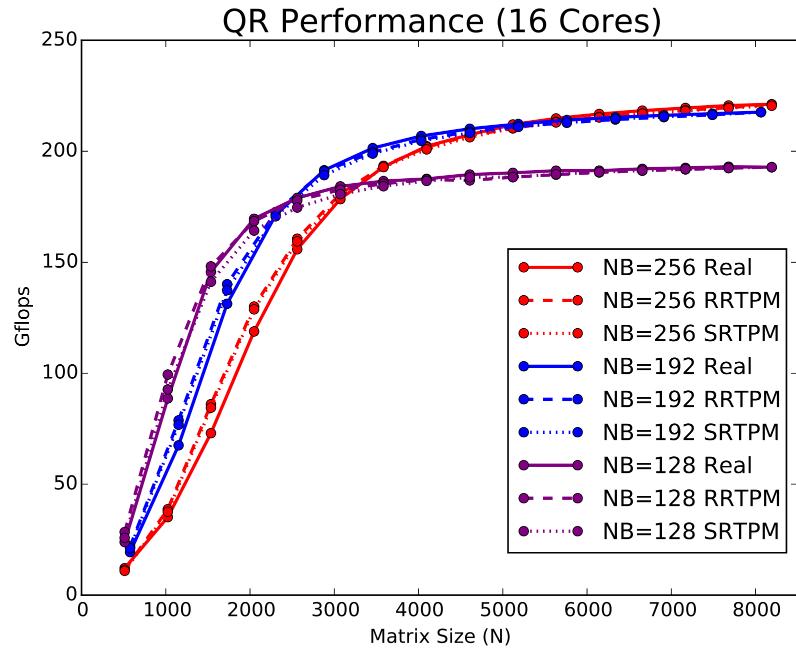



**Figure 4.30:** QR performance predicted by RTPM of a real trace (RRTPM) and a simulated trace (SRTPM) 2 x 8 Core Intel Xeon E5-2690

fromm PLASMA implemented in OpenMP using the GCC compiler. Figures 4.29 and 4.30 demonstrate the accuracy of RTPM in comparison to the real execution of the algorithm. In each case the solid line represents a real data point obtained by running the algorithm. The dashed line represents the performance obtained using the RTPM method to derive a full performance curve based on a single real trace. The dotted line represents the performance obtained using the RTPM method to derive a full performance curve based on a single simulated trace. RTPM provides very accurate performance results for both real and simulated traces.

## 4.6 Kastors SparseLU Simulation

A team at INRIA has implemented a suite of benchmarks to evaluate the latest task constructs in OpenMP. This suite of benchmarks is called Kastors [54] and includes portions of the PLASMA library as well as a number of other workloads. In order to demonstrate the accuracy and utility of the simulations outside of dense linear algebra, a Sparse LU factorization was selected from the benchmark suite. The code was instrumented to enable simulations. The workload has two parameters. The first is the matrix size that is expressed by the number of blocks. The results shown here keep the default size of 64 blocks. The second parameter is the SubMatrix size which was varied from 32 to 192 by steps of 32 as well as a submatrix size of 16. The timing results for each of these configurations are shown in Figure 4.31. Figure 4.32 zooms in on the smaller tile sizes in order to demonstrate the differing simulation results that are achieved based on the two simulation modes. The small tile size of 16 causes stress on the scheduler which actually **increases** the length of time to compute this workload. It is difficult to know exactly what causes this increase in runtime but it is likely due to contention for a lock in the scheduler.

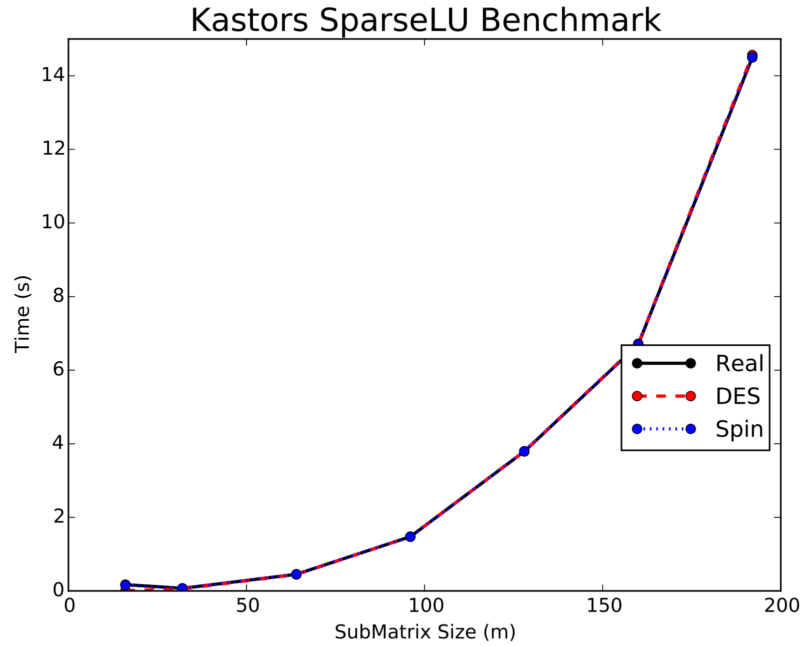



**Figure 4.31:** Runtime for SparseLU factorization including discrete event and spin simulations. 2 x 8 Core Intel Xeon E5-2690

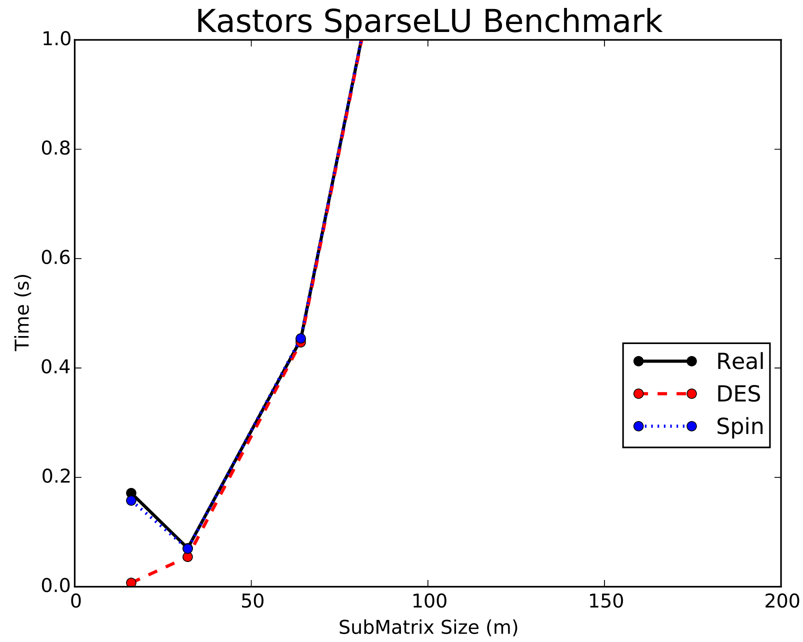



**Figure 4.32:** Runtime for SparseLU factorization including discrete event and spin simulations. This plot zooms in on the same data presented in Figure 4.31. 2 x 8 Core Intel Xeon E5-2690

## 4.7 Conclusions

Chapter 3 presented the design and implementation of a novel simulation framework for task-based runtimes and this chapter has presented a number of experiments validating the accuracy and usefulness of such a simulation library. The experiments have been performed on a variety of multicore shared memory hardware including AMD CPUs, Intel CPUs and the manycore Intel Xeon Phi. The example workloads have included the Cholesky, LU, and QR factorizations as well as the SparseLU factorization from the Kastors benchmark suite.

The design of this simulation framework has many similarities to the Prometheus and StarPU simulation implementations, but also improves on them in many ways. The most obvious improvement is the scheduler portability in our new simulation framework. In the case of Prometheus, the user is confined to the Cilk++ framework. The author suggests this could be extended to other scheduling libraries in the future, but this would require a modification of the code base in order to be able to collect the DAG representation for each workload. In the case of StarPU, the simulation is heavily tied into the scheduler.

Like the StarPU simulations, our new simulation relies on the same code to make scheduling decisions as in a real world execution of a workload. In Prometheus, a predefined scheduling algorithm is used to make scheduling decisions which may or may not accurately reflect the decisions made by the real scheduler.

In terms of workload, the StarPU simulations have focused on heterogeneous applications in which the tasks tend to be rather large in order to make efficient use of the highly parallel accelerators. As a result, they have not focused on smaller task sizes and NUMA architectures. Accurate simulation for NUMA architectures requires careful task modeling in order to obtain the best results.

It should also be noted that our simulation seems to break down in two distinct scenarios. The first scenario is when the act of simulating an algorithm alters the decisions made by the scheduler. This does not seem to be a common problem, but

can occur in some cases. This error is something to watch for as the complexity of task-based schedulers increases and developers attempt to optimize their scheduling algorithms. The other case where the simulators struggle to accurately predict the performance of a workload is when the task granularity is small and/or the number of cores is fairly large. This, however, is also where the efficiency of the schedulers begins to breakdown as well. In conclusion, the simulator presented here represents forward progress that increases functionality and increases usability for the accurate modeling of task-based runtimes.

# Chapter 5

# Trace Visualization

This chapter is based on the following publication by Blake Haugen et al.:

- Haugen, Blake, Stephen Richmond, Jakub Kurzak, Chad A. Steed, and Jack Dongarra. "Visualizing Execution Traces with Task Dependencies." In Proceedings of the *2nd Workshop on Visual Performance Analysis*, p. 2. ACM, 2015.

My contributions to this paper include (i) design of the visualization method, (ii) implementation of visualization software, (iii) collection of experimental data, and (iv) primary authorship of the text.

## 5.1 Introduction

Task-based schedulers often provide rich data sets that can be used to analyze and evaluate the characteristics of an algorithm as well as how the problem was mapped to hardware resources. The first data set is the Dircted Acyclic Graph (DAG) that defines the tasks and their dependencies. The other data set is often called a trace and collects information about the execution of each task in the workload.

Execution traces have been employed to provide users and developers a greater understanding of their software. However, these tools are relatively static and can be

improved for the workloads of task-based schedulers. The visual information-seeking mantra of "overview first, zoom and filter, then details on demand" [46] certainly applies to trace visualizations. The work presented here extends the current methods to provide users with more "details on demand" about their computational workloads. This chapter will present the data sets and how they can be combined to create a new interactive data visualization tool.

### 5.1.1 DAG

Task-based schedulers ultimately rely on the dependencies between tasks. Whether the developer explicitly states the task dependencies or the scheduling library infers them, the data dependencies must be observed in order to ensure accurate computation. These dependencies are often represented by a DAG.

Figure 5.1 shows the DAG for a small linear algebra problem that only has 55 tasks. Each vertex in the graph represents a task and is depicted in the figure by an oval. (Each oval is labeled with the type of task it represents.) Each of the edges in the graph represents the dependencies that must be observed when scheduling the tasks. The data set produced by a small QR factorization from the PLASMA library was used to generate Figures 5.1, 5.2, 5.3, and 5.4. The algorithm was executed on a single 8-core CPU. This small problem size was selected to illustrate the underlying structure of the problem rather than a real world application. Larger, more realistic problem sizes will be used later.

The SMPSs, StarPU, PARSEC, and QUARK libraries generate the DAG in a DOT file which can be used by many applications and libraries to visualize and interact with the DAG. Figure 5.1 was produced from the execution of the workload using QUARK. The resulting DOT file was visualized using the GraphViz toolkit.

The TEMANAJO project [16] also aims to visualize the task dependency graphs for task-based parallel computing. The project gives the developer a visualization of the dependencies but it is primarily used for debugging.

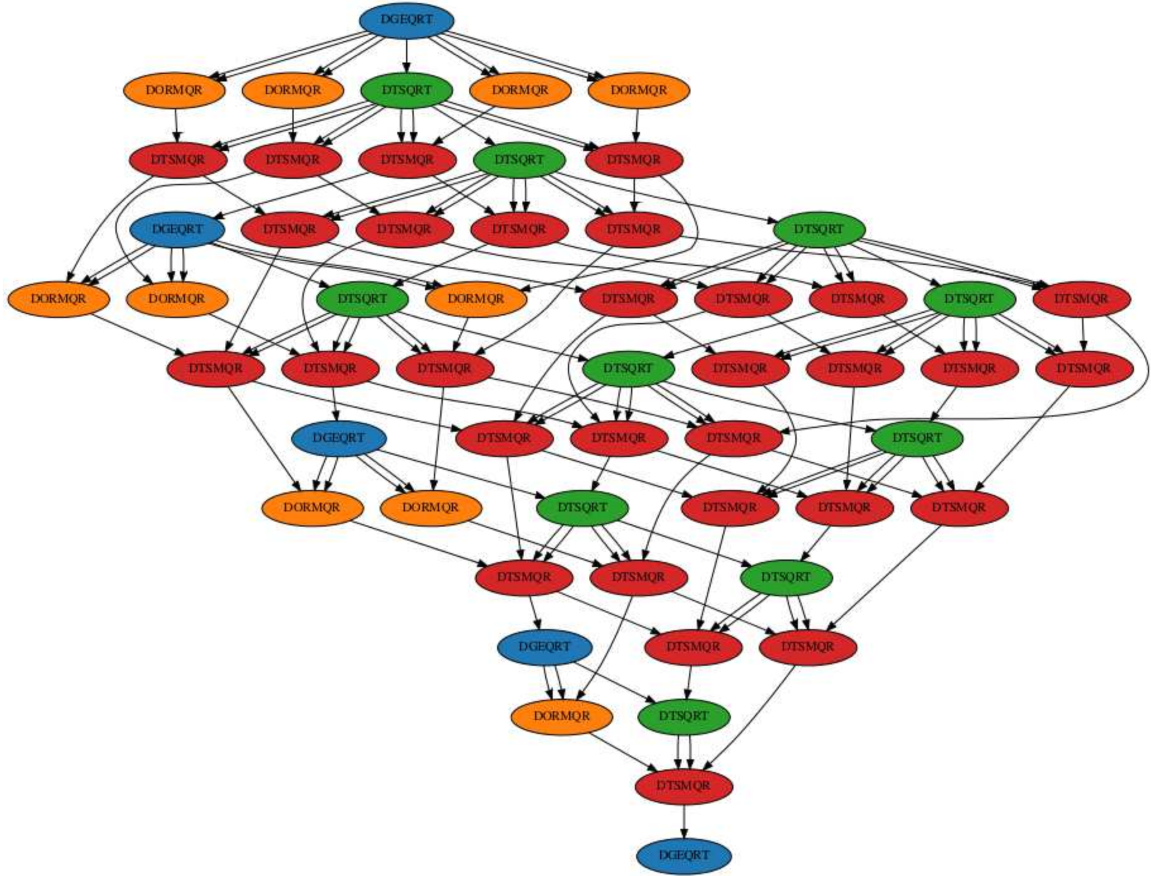



**Figure 5.1:** This graph is an example of a small DAG from the QR factorization implemented in the PLASMA library. The tasks are labeled and colored by class and each arrow represents a data dependency.

### 5.1.2 Trace

Execution traces collect basic information about each task in an execution. These data sets generally include a label for the task as well as timestamps indicating when the task started and stopped. The trace also includes information about the computational resource used to execute the task such as the core, node, or accelerator that completed the task. The execution trace may also collect other information about the tasks such as hardware counters queried using the PAPI library.

Execution traces are frequently visualized using a Gantt chart like the one shown in Figure 5.2. The trace visualization was generated from a small QR factorization from the PLASMA library. (This is the same problem used to create the DAG in

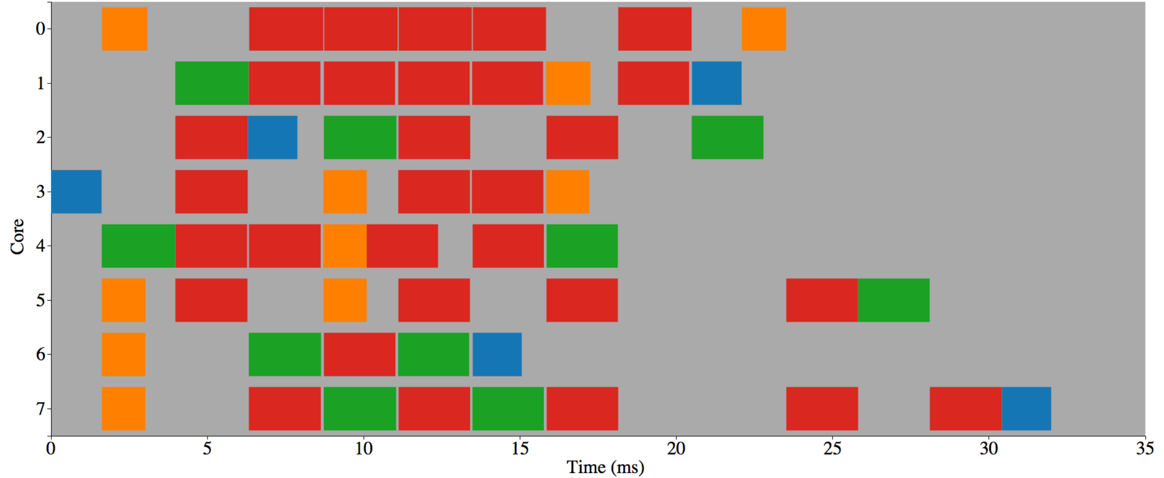



**Figure 5.2:** This is an example of a trace from the QR factorization in the PLASMA library. The tasks are colored to match the task classes in Figure 5.1.

Figure 5.1. The same color scheme is used for the tasks in the DAG as well as the trace.) The workload was executed on 8 cores of a shared memory system. The x-axis is used to depict the time (in milliseconds) while each row is used to represent a single core on the system. Each of the rectangles represents one of the tasks comprising the parallel workload. The rectangles in this figure are colored to convey the type of task represented. However, the color and texture of the boxes can be used to depict any number of task properties.

Unfortunately, the wide variety and complex interoperability of trace collection, analysis, and visualization tools make it difficult to accurately describe the landscape of the field briefly. There are several trace collection tools producing intermediate data formats which can often be converted to use a variety of analysis and visualization tools to analyze the execution trace.

SLOG-2 and Jumpshot [20] were developed at Argonne National Laboratory for trace collection and analysis. The focus of the work was to provide a file format and viewer that could scale to very large trace sizes. The trace information is stored in the file hierarchically which provides efficient access to any portion of the trace.

The TAU performance system [45] focuses on providing an instrumentation toolkit (Program Data Toolkit or PDT) that collects the event data. TAU also provides

ParaProf and PerfExplorer for detailed analysis and visualization of many of the performance characteristics of an algorithm. The tracing information can also be converted to a variety of common formats for viewing with several event trace viewers.

Researchers at the Barcelona Supercomputing Center have also developed an ecosystem of tools for collecting and analyzing event trace data. Extrae [26] is used to instrument a parallel program and collect the event trace. Paraver [41] is used to visualize the trace while Dimemas [12] is used to manipulate it and simulate execution under a variety of conditions.

EZTrace [53, 8] was built on top of the Generic Trace Generator [21] library which is capable of producing various trace file formats including Open Trace Format (OTF) and Pajé. These traces can be viewed with the ViTE trace viewer or Vampir.

Arguably the most common trace viewer and analysis toolkit in the field is Vampir. This viewer has the ability to view trace files in Open Trace Format (OTF) or OTF2 which can be collected using a variety of instrumentation toolkits. Vampir also provides a number of features and tools allowing the developer to interact and analyze the event trace [31].

Finally, the PARSEC project [15] has implemented an embedded execution data collection framework creating a binary file with a variety of performance information including an execution trace. The data in the PARSEC Trace Table (PTT) can be read and analyzed using a Python library or converted to a Pajé trace file which can be viewed using ViTE.

## 5.2 Visualization Design

The concept of visualizing communication in an execution trace is not new and has been implemented in many trace environments. However, the current methods can be improved. The current tracing methods often instrument the code automatically for the user. Each invocation of a function is recorded with a starting and stopping time as well as information about the computational resource performing the computation.

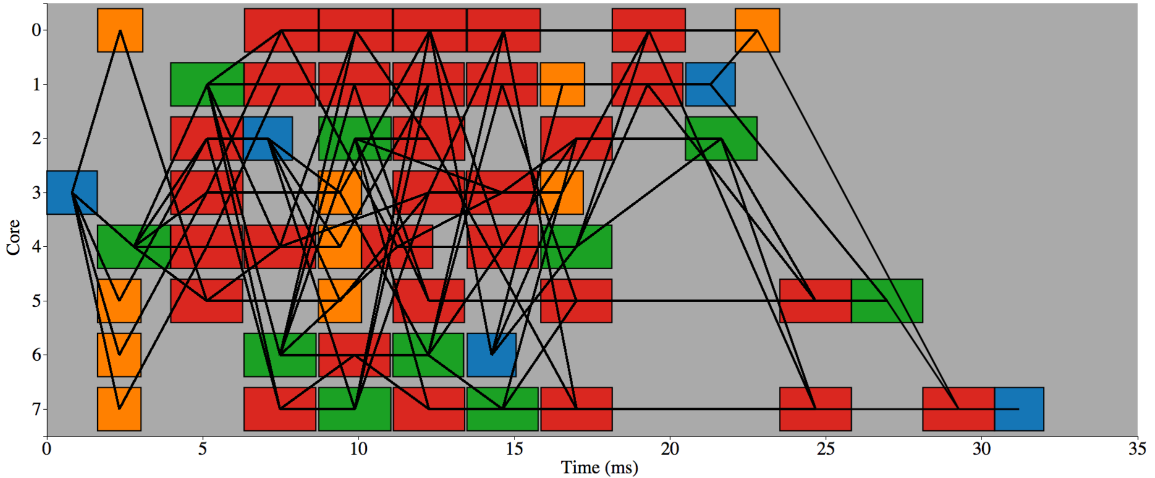



**Figure 5.3:** This trace is identical to Figure 5.2 except all of the dependencies from Figure 5.1 are drawn using black lines. This demonstrates how quickly the dependencies can overwhelm the user.

In the context of an MPI program, it is also possible to instrument all of the communication functions. The communication functions are traditionally represented with a line between the two nodes on the execution trace. This depiction clearly communicates data movement has occurred, but it is often overwhelming to the user when all of the communications are shown simultaneously.

This communication visualization method is perfectly suited for software which uses MPI because each time the program moves data it must call an MPI communication function that can easily be tracked. In a shared memory setting, however, this methods breaks down. There is no communication function which can easily be instrumented to log data movement. The user must have knowledge of the algorithmic structure and what data movement must occur. It is hard to know exactly how the data transfer takes place but it must occur in order for the computation to continue. The task-based schedulers can provide information about where the computations happen and where the data was before it was performed.

It should be noted that a dependency between tasks implies the later task must wait until the earlier task has completed. This means the second task is waiting for some piece of data from the earlier one. If these tasks are executed consecutively on

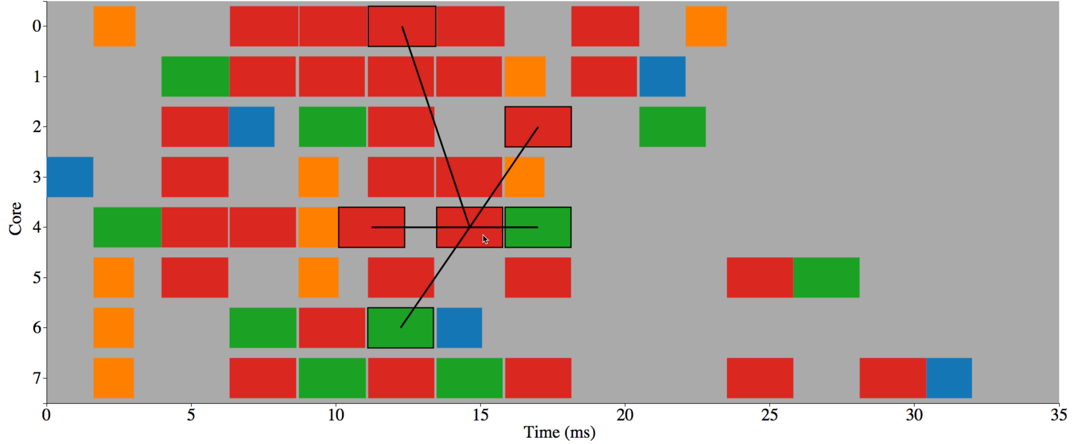



**Figure 5.4:** This trace demonstrates the basic interactivity of the software. The user has selected the red task in the middle by placing the cursor over it. The trace highlights the dependencies on either side of that task with a black border and a line to depict the dependency.

the same core or device, the data should already be in cache and the communication cost should be relatively low. On the other hand, if the tasks are performed on a different core, device, or node the scheduler must move data across the memory hierarchy or communicate with another node. As a result, when considering task-based scheduling, a dependency implies the requirement of communication unless the tasks are computed on the same core. Even if the tasks are computed on the same node it is possible data will have to move through the memory hierarchy if the data has been evicted from the processor cache.

Perhaps the most obvious way to depict the execution trace and the task dependencies is to visualize the trace and the DAG simultaneously. Adding interactivity with mouseovers or mouseclicks would allow the user to select a task in the trace which would also highlight the corresponding task in the DAG. The opposite could also be true. However, the size of the DAG and trace quickly grow to extremely large datasets which make it difficult for the user to comprehend the information on a problem of any reasonable size. As a result, the two visualizations need to be combined into a single visual representation.

The first version of the tool employed the depictions used by many common MPI tracing tools. The tasks were represented using the same methodology as Figure 5.2. A simple line between tasks, a common visual representation for MPI communication, was used to represent the dependencies and data communications required by the algorithm. This representation, applied to the same data used in Figures 5.2 and 5.1, can be seen in Figure 5.3.

Figure 5.3 now shows all of the tasks and their dependencies in one visual space. However, even for small problems the number of dependencies quickly overwhelms the user. The visual “hairball” shown in Figure 5.3 can be greatly improved by making the dependency lines an interactive feature.

Figure 5.4 demonstrates a visualization of the small problem presented earlier, but with interactive features. Without having the mouse hover over any of the tasks in the diagram, the users see a trace that looks identical to the trace in Figure 5.2. When the user moves the mouse over one of the tasks, however, the trace highlights the task as well as the tasks for which it is waiting. It also highlights any tasks that are waiting for it to complete. Additionally, the tasks are connected to the task in focus to represent the dependencies. In terms of the DAG, each of the lines represents the edges connected to the highlighted task. Lines connected to tasks earlier in the trace are edges directed into the highlighted node. Conversely, lines connected to tasks in the future represent edges leaving the highlighted node in the DAG.

Solid black lines were chosen to represent the dependencies for a number of reasons. First, the dependency represents data that must be moved in order for the computation to proceed. Most trace visualization tools use lines to represent communication or data movement in an MPI application. Next, the lines are used in the DAG to represent the dependencies making them a logical choice in the new visualization. Finally, it has been shown that these “leader lines” are a good visual cue that frees up other visual techniques, such as color, to represent other information [27].

Many of these features can be configured to allow the user to adjust the behavior of the visualization. For example, the user may want to only highlight (add a black

border to the task) dependencies without drawing the lines. The user may choose to only show the tasks in the past or only the tasks waiting in the future. The user may also want to see tasks more than one step away from the task in focus. These are all features the user can configure in order to make the visualization useful.

### 5.2.1 Implementation

In order to create the visualization, two separate data sets must be combined. The first data set is from the execution trace. This data generally contains information about each task including when it started, when it ended, on what core it executed, and likely the type of task. This data set may also contain other information about the task. For example, the code may be instrumented with PAPI counters which collect information about cache misses or instruction counts.

The second data set is the DAG of tasks and dependencies. QUARK and PARSEC currently provide the DAG for the workload in a DOT file. This information can be used to visualize the DAG using any number of software libraries. The file can also be used to identify the dependencies (edges) of the graph.

The challenging part of combining these data sets is finding the tasks in the execution trace corresponding to each of the nodes in the DAG. The earlier discussion about tracing collection and storage utilities highlights the challenge of dealing with data produced by different schedulers and instrumentation libraries. The code currently supports data from QUARK and PARSEC, although it could be extended to support various other data formats and schedulers in the future.

QUARK provides a task id unique to each task in the execution. The task id is included in each node of the DOT file containing the DAG representing the computation. The trace for the experiments shown later was collected by instrumenting each task with a start and stop time stamp as well as the corresponding task id. Once the two files are produced, the tasks in the DOT file are matched to the corresponding tasks in the trace.

PARSEC, however, automatically collects and records all of the necessary data to match tasks from the DAG with tasks from the trace. The DAG is collected in pieces on each node and post-processed to generate a single DOT file. The trace is collected in the PTT file discussed earlier. The distributed nature of the PARSEC execution makes it difficult to have a single task id unique to each task. Therefore, each task has three id properties (hid, did, and tid) that combine to create a unique identifier for each of the tasks. The three ids are present in the DOT and PTT files created by PARSEC and are used to match the tasks from the two data sets.

There is no standardized method for uniquely identifying tasks in a trace or DAG at this time. In order to port this method to the data provided by other schedulers, the user must be able to uniquely identify tasks in the trace as well as the DAG.

The visualization is implemented as a client-server architecture. The server is implemented in Python while the client is implemented using Javascript. The Python server is better suited to complete the heavy computational workloads and perform various analytical tasks. Javascript (and associated libraries) are well suited for making interactive visualizations.

This architecture also gives developers a flexible way to improve, adjust, and expand the capabilities of this system. The next section will demonstrate how this visualization can be used and an extension using a kernel density estimation (KDE) plot in conjunction with the trace visualization.

## 5.3 Applications

Perhaps the simplest use case for the combined interactive visualization is to determine why there is idle time in the trace. For example, the trace shown in Figure 5.4 has many sections where it would appear the system is underutilized. The task the user has selected has idle time preceding it in the trace. This may suggest to a novice user that the runtime system is not efficiently mapping the tasks onto the hardware. With the addition of the interactive dependency visualization, the user

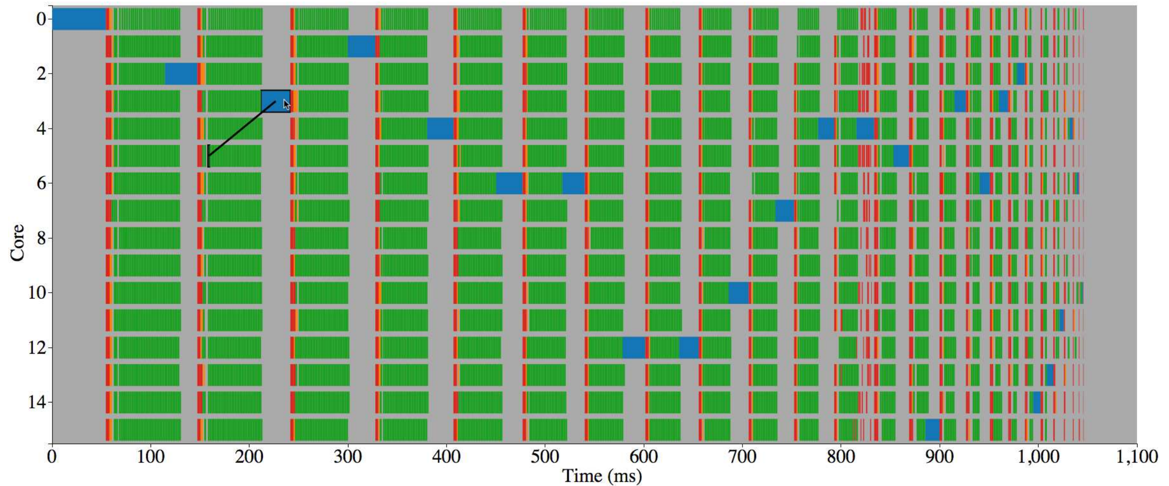



**Figure 5.5:** The trace for an LU factorization of a matrix of size 5000 with a tile size of 200. This means the matrix is $25 \times 25$ tiles. The factorization is performed using QUARK and no priority hints.

can now see that the task could not have been executed earlier because it was waiting for data from another task.

This visualization can also be used by those developing task-based runtimes to determine whether their runtime is working as expected. For example, if the trace shows idle space when all of the dependencies have already been satisfied, a developer may want to examine his/her scheduling algorithm or look at work-stealing policies that could improve performance.

Task-based schedulers are often very efficient when mapping a workload to the hardware but they can sometimes be improved if the developer provides scheduling hints about each task. One common type of hint is priority. An experienced developer with excellent knowledge of his/her workload may know which tasks are on the critical path and should be executed as soon as possible in order to reduce the effect of any bottlenecks in the workload.

The LU factorization in PLASMA is one example of an algorithm with a bottleneck that can drastically reduce the performance of the factorization. The trace for an LU factorization on a matrix of 5000 elements and a tile size of 200 is shown in Figure 5.5. The DGETRF tasks (shown in blue) clearly create a bottleneck because the scheduler

must wait for them to complete in order to continue with the computation. If a developer does not have an extensive knowledge of his/her application and a strong intuition for the dependencies present in the algorithm, it is likely unclear if this can be improved.

Even if the user examines the DAG and the trace it still may not be entirely clear whether this performance can be improved. The first problem is that the size of the DAG can often be so large it is hard to render. Even if it can be "rendered", the DAG may be completely unreadable. For example, a small portion of the DAG corresponding to the workload presented in Figure 5.5 is shown in Figure 5.6. Even when zoomed in on the DAG it is impossible to understand the structure of the DAG. Figure 5.7 shows the DAG for the same problem but on a much smaller matrix. This may begin to give the developer a sense of the problem structure but the difference in scale and the difficulty of matching the task in the trace and the DAG still make it challenging to understand whether the scheduling in Figure 5.5 can be improved.

It is clear the DGETRF tasks (shown in blue) are the bottleneck so the user has selected one of them. When the task is selected it also highlights any dependencies it is waiting for and it becomes clear the task has been waiting even though there are no outstanding dependencies for the task. This suggests to the user that elevating the priority of this task can assist the scheduler in overlapping this task with other work and accelerating the workload.

Figure 5.8 shows the trace of the same problem but the DGETRF (shown in blue) tasks are given a higher priority than the other tasks. The scheduler uses this information to move the DGETRF tasks to the front of the scheduling queue. When these tasks are given priority, they reduce the effects of the bottleneck and drastically improve the performance of the workload. The factorization without any priority hints (Figure 5.5) performs at 166 GFLOP/s while the addition of priority hints (Figure 5.8) improves the performance to 226 GFLOP/s.

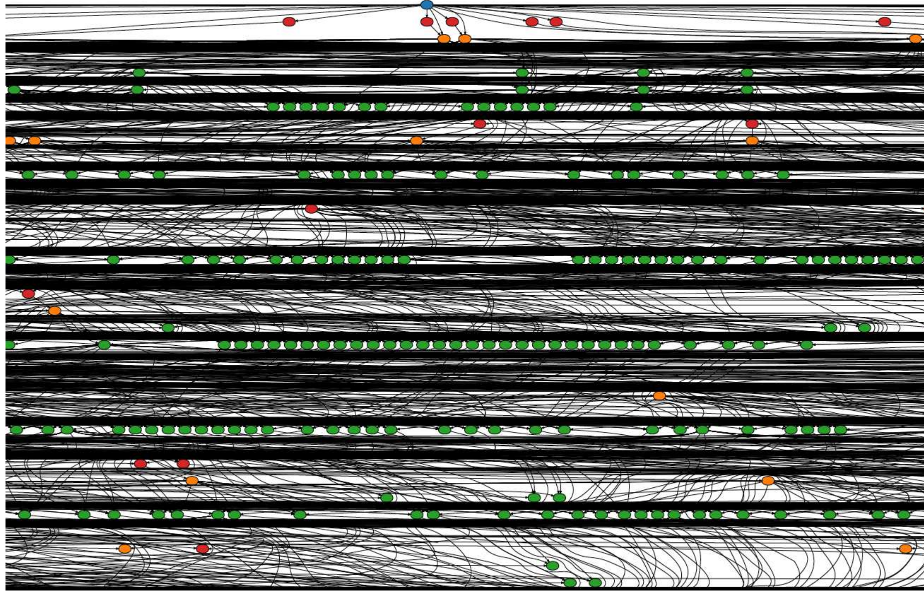


**Figure 5.6:** A portion of the DAG for a moderately sized LU factorization on a matrix that is $25 \times 25$ tiles. This is an excellent example of how large these task DAGs can be. GraphViz requires several minutes to “render” the DAG and the results are completely unreadable.

StarPU and QUARK already allow the developer to specify the priority for a given task. The standard for OpenMP 4.5 also includes support for priority hints and should be supported in future implementations of OpenMP.

Another feature added in the visualization is the ability to modify the color of the tasks based on their relative execution times. Another challenge of trace visualization can be comparing the length of tasks. In order to show thousands of tasks on the trace, they must be relatively small and it could be challenging for the user to perceive the relative time for various tasks.

In order to determine the relative execution time, a z-score is computed for each task using the following formula:

$$z = \frac{x - \mu}{\sigma}$$

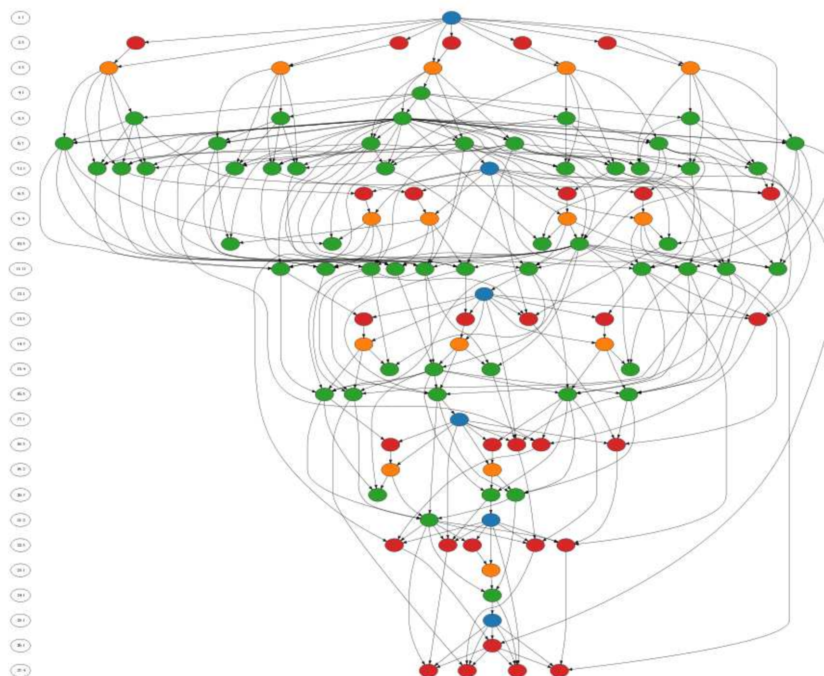


**Figure 5.7:** A DAG for a smaller LU factorization. This DAG allows the developer to see the problem structure without being overwhelmed by the size of the graph like Figure 5.6

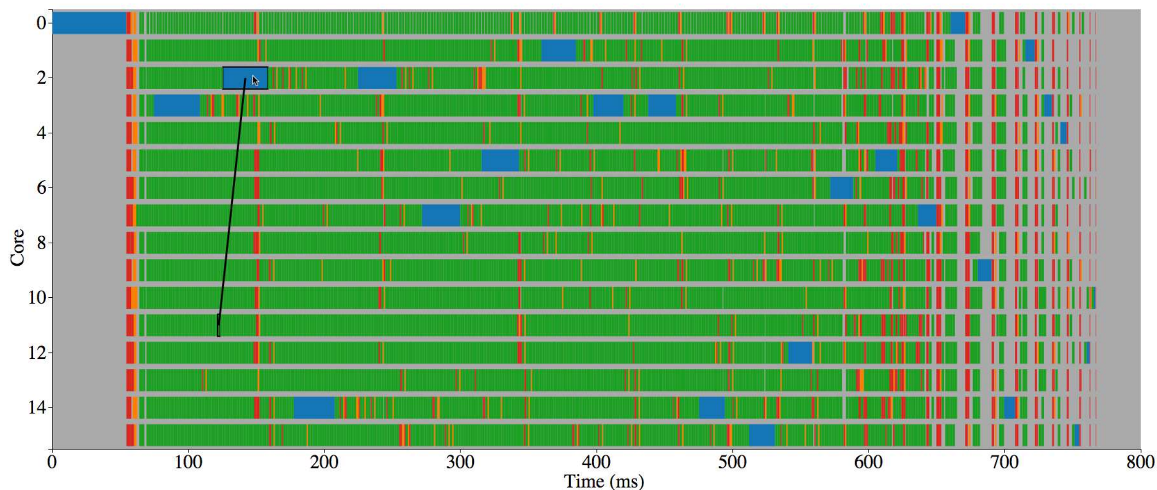



**Figure 5.8:** The trace for an LU factorization of a matrix of size 5000 with a tile size of 200. This means the matrix is $25 \times 25$ tiles. The factorization is performed using QUARK and elevates the priority of the DGETRF tasks (shown in blue).

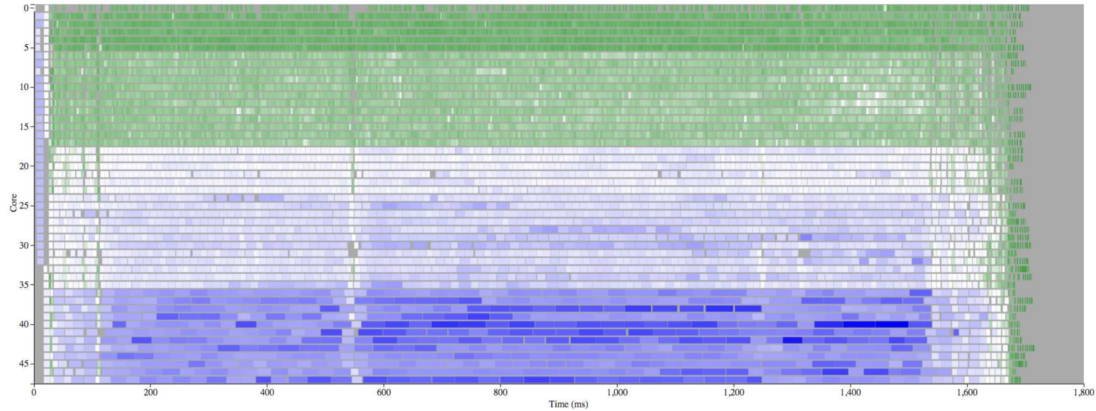



**Figure 5.9:** A trace where all of the data is initialized on one NUMA node.

Where $x$ is the raw data point, $\mu$ is the mean and $\sigma$ is the standard of deviation. If this is calculated across the entire data, it may be misleading because each of the tasks likely has distinct performance characteristics. As a result, the z-score is calculated based on the mean and standard deviation for each type of task in the trace. In other words, each task is compared with the performance of tasks of the same type as opposed to the entire data set.

Once the z-score is calculated for each task, it is visualized on a continuous scale centered at zero. The center of the color scale is white and represents tasks which have z-scores near zero and are near the average time for that type of task. As a task's execution increases relative to the mean, the z-score grows in the positive direction and the color of the task is an increasingly bright blue. If the task is faster than average, the z-score drops below zero and the color of the task becomes a brighter green.

This relative task time encoding can be used to diagnose a common performance issue on a NUMA machine. Figure 5.9 shows the relative task time encoding for a workload performed on a NUMA machine with 8 AMD Opteron 8358 SE processors. This encoding of the trace makes it clear there must be some sort of performance issue that may be improved. All of the tasks on the first 6 cores (first socket) are green indicating they are faster than average. The last 6 cores (the last socket), however,

are filled with tasks that are blue indicating they are much slower than average. This pattern often indicates the data is allocated and initialized on one NUMA node. The non-uniformity of the memory access speed is clearly visible with this representation.

The first possible solution to this performance problem is to initialize the data in parallel. Memory placement generally follows a "first touch" rule so the memory is placed in the section of memory closest to the core. By initializing the memory in parallel, the data is spread across the system. The PLASMA library has matrix initialization routines to perform this matrix generation in parallel and as a result the data is distributed across the machine.

Alternatively, numactl can be used to control where the memory is placed. Numactl is a linux utility commonly used to control the NUMA policy for a process. One of the options is an "interleave" policy that determines upon which nodes the memory will be placed. The "interleave=all" option sets the policy to distribute the data across all of the memory nodes. Having the memory distributed will likely mean some of the tasks that were fast before will take longer to compute but the slower tasks will generally be computed much more quickly. Having the data dispersed across the machine will also reduce the contention present when all of the cores attempt to access memory on the same NUMA node.

## 5.4 Trace Visual Analytics System

Visual analytics is defined as "the science of analytical reasoning facilitated by interactive visual interfaces." [51] The visualization method presented earlier can be applied independently like the example in Figure 5.4 and can be considered a visual analytic tool. However, it is also possible to employ this technique in conjunction with other data visualization techniques and create a coordinated multiple view visualization. Multiples view systems are defined as "systems that use two or more distinct views to support the investigation of a single conceptual entity." [56] The coordination of these views with techniques like brushing can provide powerful tools

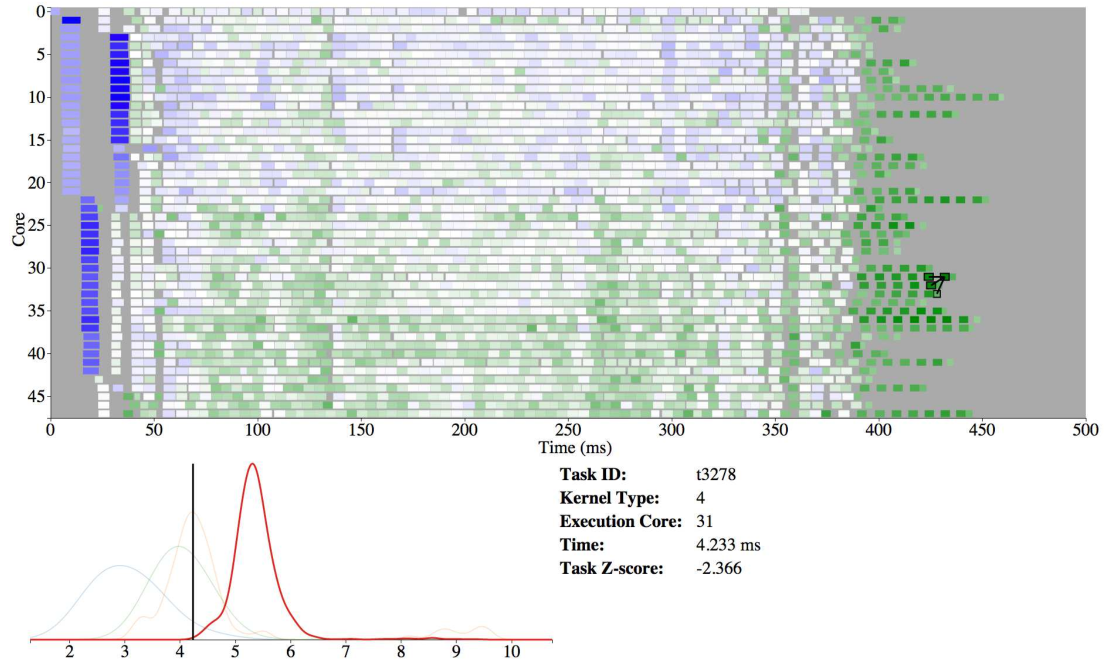



**Figure 5.10:** An example of the trace utility applied to a linear algebra workload. The tasks are colored based on their relative speeds.

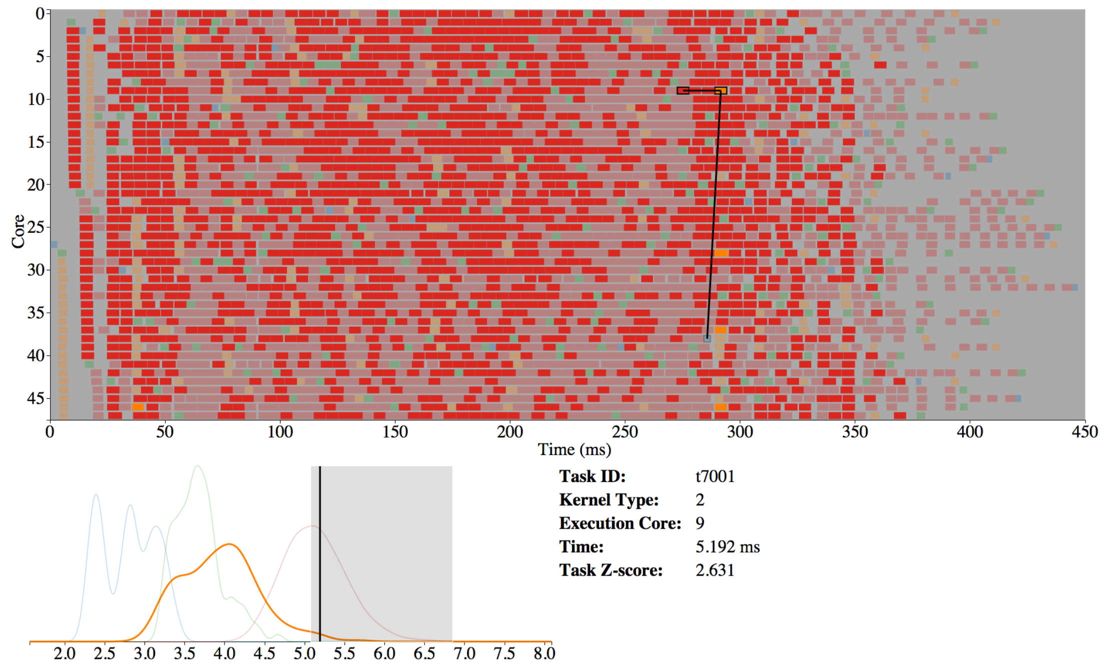



**Figure 5.11:** An example of the trace utility applied to a linear algebra workload. The KDE plot was used to highlight tasks in the trace based on execution time.

that allow users to interact with their data and gain new insight. Figures 5.10 and 5.11 present two examples of the visualization library applied to a real world data set in a coordinated multiple view visual analytics tool.

Figure 5.10 demonstrates the new trace visualization on the same linear algebra application presented earlier. However, this example uses a larger problem size which more closely resembles a real world problem. The tasks in the trace have been colored based on their relative task times as described earlier. This representation allows the user to quickly determine which tasks are slower or faster than average. There are several blue tasks at the beginning of the trace. This is likely due to library and data initialization costs at the start of the algorithm.

Several of the tasks near the end are green indicating they are faster than average. It is likely this is caused in part by the smaller number of tasks being executed and the resulting reduction in memory contention. By selecting one of the brightest green tasks, the visualization also shows the user the data dependencies all come from the same CPU. Therefore, the data is likely to be in cache instead of the main memory or cache on another chip. As a result, the data movement is likely to be much faster than other tasks.

The plot at the bottom left shows four KDE curves for the four types of tasks. The red KDE curve is highlighted which indicates the selected task is part of this density estimator. The black vertical line indicates where the selected task falls in relation to the distribution of task times. In this case, the selected task is likely one of the fastest of its kind.

One of the elements of the visual information-seeking mantra is the ability to filter the data and make it easier to focus on information deemed most interesting by the user. Figure 5.11 demonstrates how the KDE plot can be used to highlight tasks in a specific range with a filter based on execution times. In this case, the user is interested in the relatively slow tasks. The tasks in the trace which have execution times falling within the range of the gray box on the KDE plot are highlighted, while the others tasks have been obscured by a reduction in opacity.

The user has selected an orange task in order to determine why it was relatively slow. The visualization shows two dependencies for the selected task. One is on the same core while the other is on another CPU. However, closer inspection reveals two other tasks were executed on the same core between the two tasks linked in the trace. Thus, the data from the dependency has likely been evicted from the cache. As a result, the task likely had to load two dependencies from memory or another CPU which caused an increase in task execution time.

The new dependency visualization technique is intended for people who develop task-based schedulers as well as the developers who use them. The developer of a scheduler can use this method to evaluate the performance of it and determine if it is performing as intended. Developers using a task-based scheduling library to parallelize their application can also use the visualization to guide their use of extended task information such as locality hints and task priority that are available in some of the schedulers.

# Chapter 6

# Conclusions

As computer architectures are becoming increasingly parallel, the need to adapt and tune software for new platforms will become an important part of computational science. Task-based runtimes provide one of the many models for expressing parallel computations. The inclusion of task constructs in the latest version of the OpenMP standard suggests that task-based scheduling will play an important role in parallel computing for the foreseeable future. While these scheduling utilities provide a layer of abstraction and increase developer productivity, they can also make performance analysis and prediction a challenging task. In this dissertation, a novel simulation framework and a trace visualization extension have been presented. They provide new methods for performance analysis to the developers creating such runtimes as well as the users who employ them to implement their workloads.

In Chapter 3, a novel task-based simulation was presented. The framework was portable across a number of schedulers (QUARK, StarPU, OmpSs, and OpenMP) with no modification. The simulations also allow users to simulate the performance of their software independent of the hardware. The simulations employ the chosen scheduler to make all decisions about task scheduling to ensure any artifacts due to the scheduling choices will be present in the simulations as well. Chapter 4 presented a number of experiments demonstrating the accuracy of the simulations. It also

discussed a few of the errors that can occur when simulating workloads in a context that violates the initial assumptions of the simulation library. Perhaps the greatest application of this simulation framework is the ability to predict the performance of an algorithm under a wide variety of circumstances.

One of the keys to accurate simulation is the ability to accurately model the runtime of each of the types of tasks in a computational workload. Poor models can result in diminished accuracy of the simulations. Previous work for benchmarking individual computation tasks was focused on the effects of caching for the timing of a task. The work presented in Chapter 3 extended this methodology to explore the effects of multicore, NUMA systems. While it is possible to design a number of benchmarks that consider many possible cache and memory access scenarios, it is exceedingly difficult to design a benchmark which perfectly matches the context of a task in a real world application. Even if it is possible to create such a benchmark, there are so many possible benchmarks to choose from that a selection process would likely be time consuming and may not even provide the best results.

Chapter 5 presented a novel extension to the common trace visualization techniques. The extension allows users to interactively explore the trace while including information from the DAG corresponding to the workload. Previously, the DAG and trace data sets could only be viewed as two separate entities which made it difficult to correlate the data from one visualization to the other. The interactive nature of the visualization also allows for the exploration of data sets previously thought to be of intractable size. This framework also presents the basis for a visual analytic system for task-based runtimes.

This dissertation presents a novel simulation framework for task-based runtimes. It demonstrates these simulations can be performed across a number of schedulers and a variety of hardware including the manycore Intel Xeon Phi. It shows these simulations are accurate and can be useful for a variety of workloads.

## 6.1 Future Work

While the simulation framework presented here works for a number of task-based schedulers in a shared memory context, it is not immediately clear how to apply it to scheduling problems in a distributed setting. The inherent synchronization required for the current simulation approach would likely be too costly for efficient simulations in a distributed context. Distributed memory systems must also perform more costly data transfer than a shared memory system which must be considered in the simulations. In the case of heterogeneous computing with accelerators, this data transfer must also be considered. Heterogenous and distributed systems could be targets for future development in the task-based simulation framework.

The trace visualization currently provides support for the QUARK and PARSEC runtimes, but this could be extended to a number of other schedulers and trace data formats. The visual analytics approach to analyzing the performance of a given workload could also be extended to include a number of other visualizations beyond the KDE curves shown at the conclusion of Chapter 5.

# Bibliography

[1] E. Agullo, J. Dongarra, B. Hadri, J. Kurzak, J. Langou, J. Langou, H. Ltaief, P. Luszczek, and A. YarKhan. PLASMA Users Guide. Technical report, University of Tennessee, Innovative Computing Laboratory, 2010. 15

[2] Emmanuel Agullo, Jim Demmel, Jack Dongarra, Bilel Hadri, Jakub Kurzak, Julien Langou, Hatem Ltaief, Piotr Luszczek, and Stanimire Tomov. Numerical Linear Algebra on Emerging Architectures: The PLASMA and MAGMA Projects. In *Journal of Physics: Conference Series*, volume 180, page 012037. IOP Publishing, 2009. 19

[3] Emmanuel Agullo, Jack Dongarra, Rajib Nath, and Stanimire Tomov. A Fully Empirical Autotuned Dense QR Factorization for Multicore Architectures. In *Proceedings of the 17th International Conference on Parallel Processing - Volume Part II*, Euro-Par'11, pages 194–205, Berlin, Heidelberg, 2011. Springer-Verlag. 24, 79

[4] E. Anderson, Z. Bai, C. Bischof, L. S. Blackford, J. Demmel, Jack J. Dongarra, J. Du Croz, S. Hammarling, A. Greenbaum, A. McKenney, and D. Sorensen. *LAPACK Users' Guide (Third ed.)*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1999. 17

[5] C. Augonnet and R. Namyst. A Unified Runtime System for Heterogeneous Multicore Architectures. In *Proceedings of the Euro-Par 2008 Workshops - Parallel Processing*, Lecture Notes in Computer Science, pages 174–183, Las Palmas de Gran Canaria, Spain, August 2008. Springer. DOI: 10.1007/978-3-642-00955-6_22. 15

[6] C. Augonnet, S. Thibault, R. Namyst, and P. Wacrenier. StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. *Concurrency Computat. Pract. Exper.*, 23(2):187–198, 2011. DOI: 10.1002/cpe.1631. 15

[7] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. In *Proceedings of the 15th International Euro-Par Conference on Parallel Processing*, Euro-Par '09, pages 863–874, Berlin, Heidelberg, 2009. Springer-Verlag. 15

[8] Charles Aulagnon, Damien Martin-Guillerez, Franois Ru, and Franois Trahay. Runtime Function Instrumentation with EZTrace. In *Euro-Par 2012: Parallel Processing Workshops*, volume 7640 of *Lecture Notes in Computer Science*, pages 395–403. Springer Berlin Heidelberg, 2013. 89

[9] Eduard Ayguadé, Rosa M Badia, Francisco D Igual, Jesús Labarta, Rafael Mayo, and Enrique S Quintana-Ortí. An Extension of the StarSs Programming Model for Platforms with Multiple GPUs. In *Proceedings of the 15th International Euro-Par Conference on Parallel Processing*, pages 851–862. Springer-Verlag, 2009. 14

[10] R. M. Badia, J. R. Herrero, J. Labarta, J. M. Perez, E. S. Quintana-Orti, and G. Quintana-Orti. Parallelizing Dense and Banded Linear Algebra Libraries Using SMPSs. *Concurrency Computat. Pract. Exper.*, 21(18):2438–2456, 2009. DOI: 10.1002/cpe.1463. 14

[11] R. M. Badia, J. Labarta, R. Sirvent, J. M. Perez, J. M. Cela, and R. Grima. Programming Grid Applications with GRID Superscalar. *J. Grid Comput.*, 1(2):151–170, 2003. DOI: 10.1023/B:GRID.0000024072.93701.f3. 14

[12] Rosa M Badia, Jess Labarta, Judit Gimenez, and Francesc Escale. Dimemas: Predicting MPI Applications Behavior in Grid Environments. In *Workshop on Grid Applications and Programming Tools (GGF8)*, volume 86, pages 52–62, 2003. 89

[13] William H. Bell, David G. Cameron, A. Paul Millar, Luigi Capozza, Kurt Stockinger, and Floriano Zini. Optorsim: A Grid Simulator for

Studying Dynamic Data Replication Strategies. *International Journal of High Performance Computing Applications*, 17(4):403–416, 2003. 24

[14] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. The gem5 Simulator. *SIGARCH Comput. Archit. News*, 39(2):1–7, August 2011. 24

[15] George Bosilca, Aurelien Bouteiller, Anthony Danalis, Mathieu Faverge, Azzam Haidar, Thomas Herault, Jakub Kurzak, Julien Langou, Pierre Lemarinier, Hatem Ltaief, Piotr Luszczek, Asim YarKhan, and Jack Dongarra. Flexible Development of Dense Linear Algebra Algorithms on Massively Parallel Architectures with DPLASMA. In *Proceedings of the 2011 IEEE International Symposium on Parallel and Distributed Processing Workshops*, IPDPSW '11, pages 1432–1441, Washington, DC, USA, 2011. IEEE Computer Society. 17, 89

[16] Steffen Brinkmann, José Gracia, Christoph Niethammer, and Rainer Keller. TEMANEJO - A Debugger for Task Based Parallel Programming Models. *CoRR*, abs/1112.4604, 2011. 86

[17] Alfredo Buttari, Julien Langou, Jakub Kurzak, and Jack Dongarra. A Class of Parallel Tiled Linear Algebra Algorithms for Multicore Architectures. *Parallel Computing*, 35(1):38–53, 2009. 19

[18] Rajkumar Buyya and Manzur Murshed. GridSim: A Toolkit for the Modeling and Simulation of Distributed Resource Management and Scheduling for Grid Computing. *Concurrency and Computation: Practice AND Experience (CCPE*, 14(13):1175–1220, 2002. 24

[19] Henri Casanova, Arnaud Legrand, and Martin Quinson. SimGrid: A Generic Framework for Large-Scale Distributed Experiments. In *Proceedings of the Tenth*

*International Conference on Computer Modeling and Simulation*, UKSIM '08, pages 126–131, Washington, DC, USA, 2008. IEEE Computer Society. 24

[20] Anthony Chan, William Gropp, and Ewing Lusk. An Efficient Format for Nearly Constant-Time Access to Arbitrary Time Intervals in Large Trace Files. *Scientific Programming*, 16(2-3):155–165, 2008. 88

[21] Kevin Coulomb, Augustin Degomme, Mathieu Faverge, and Franois Trahay. An Open-Source Tool-Chain for Performance Analysis. In *Tools for High Performance Computing 2011*, pages 37–48. Springer Berlin Heidelberg, 2012. 89

[22] L. Dagum and R. Menon. OpenMP: An Industry Standard API for Shared-Memory Programming. *Computational Science Engineering, IEEE*, 5(1):46 –55, 1998. 16

[23] Andrew Danowitz, Kyle Kelley, James Mao, John P Stevenson, and Mark Horowitz. CPU DB: Recording Microprocessor History. *Communications of the ACM*, 55(4):55–63, 2012. 6

[24] Simplice Donfack, Jack Dongarra, Mathieu Faverge, Mark Gates, Jakub Kurzak, Piotr Luszczek, and Ichitaro Yamazaki. A Survey of Recent Developments in Parallel Implementations of Gaussian Elimination. *Concurrency and Computation: Practice and Experience*, 27(5):1292–1309, 2015. 19

[25] A. Duran, E. Ayguade, R. Badia, J. Labarta, L. Martinell, X. Martorell, and J. Planas. OmpSs: A Proposal for Programming Heterogeneous Multi-Core Architectures. *Parallel Process. Lett.*, 21(2):173–193, 2011. DOI: 10.1142/S0129626411000151. 14

[26] HS Gelabert and GL Sánchez. Extrae User Guide Manual for Version 2.2.0. *Barcelona Supercomputing Center (B. Sc.)*, 2011. 89

[27] Amy L Griffin and Anthony C Robinson. Comparing Color and Leader Line Highlighting Strategies in Coordinated View Geovisualizations. *Visualization and Computer Graphics, IEEE Transactions on*, 21(3):339–349, 2015. 92

[28] John A. Gunnels, Fred G. Gustavson, Greg M. Henry, and Robert A. van de Geijn. FLAME: Formal Linear Algebra Methods Environment. *ACM Trans. Math. Softw.*, 27:422–455, December 2001. 19

[29] Azzam Haidar, Jakub Kurzak, and Piotr Luszczek. An Improved Parallel Singular Value Algorithm and its Implementation for Multicore Hardware. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, page 90. ACM, 2013. 19

[30] Gokcen Kestor, Roberto Gioiosa, and Daniel Chavarrıa-Miranda. Prometheus: Scalable and Accurate Emulation of Task-Based Applications on Many-Core Systems. In *Performance Analysis of Systems and Software (ISPASS), 2015 IEEE International Symposium on*, pages 308–317. IEEE, 2015. 25

[31] Andreas Knüpfer, Holger Brunst, Jens Doleschal, Matthias Jurenz, Matthias Lieber, Holger Mickler, Matthias S Müller, and Wolfgang E Nagel. The Vampir Performance Analysis Tool-Set. In *Tools for High Performance Computing*, pages 139–155. Springer, 2008. 89

[32] J. Kurzak, P. Luszczek, M. Faverge, and J. Dongarra. LU Factorization with Partial Pivoting for a Multicore System with Accelerators. *IEEE Trans. Parallel Distrib. Syst.*, 2012. DOI: 10.1109/TPDS.2012.242. 15

[33] Jakub Kurzak, Hatem Ltaief, Jack Dongarra, and Rosa M Badia. Scheduling Dense Linear Algebra Operations on Multicore Processors. *Concurrency and Computation: Practice and Experience*, 22(1):15–44, 2010. 19

[34] Yu-Kwong Kwok and Ishfaq Ahmad. Static Scheduling Algorithms for Allocating Directed Task Graphs to Multiprocessors. *ACM Comput. Surv.*, 31(4):406–471, December 1999. 23

[35] R Garey Michael and David S Johnson. Computers and Intractability : A Guide to the Theory of NP-Completeness. *WH Freeman & Co., San Francisco*, 1979. 23

[36] OpenMP Architecture Review Board. OpenMP Application Programming Interface Version 3.0, May 2008. 16

[37] OpenMP Architecture Review Board. OpenMP Application Programming Interface Version 4.0, July 2013. 16

[38] OpenMP Architecture Review Board. OpenMP Application Programming Interface Version 4.5, November 2015. 16

[39] J. M. Perez, P. Bellens, R. M. Badia, and J. Labarta. CellSs: Making it Easier to Program the Cell Broadband Engine Processor. *IBM J. Res. & Dev.*, 51(5):593–604, 2007. DOI: 10.1147/rd.515.0593. 14

[40] Josep M. Pérez, Rosa M. Badia, and Jesús Labarta. A Dependency-Aware Task-Based Programming Environment for Multi-Core Architectures. In *Proceedings of the 2008 IEEE International Conference on Cluster Computing, 29 September - 1 October 2008, Tsukuba, Japan*, pages 142–151. IEEE, 2008. 14

[41] Vincent Pillet, Jesús Labarta, Toni Cortes, and Sergi Girona. Paraver: A Tool to Visualize and Analyze Parallel Code. In *Proceedings of WoTUG-18: Transputer and occam Developments*, volume 44, pages 17–31. mar, 1995. 89

[42] J. Planas, R. M. Badia, E. Ayguadé, and J. Labarta. Hierarchical Task-Based Programming with StarSs. *Int. J. High Perf. Comput. Applic.*, 23(3):284–299, 2009. DOI: 10.1177/1094342009106195 . 14

[43] K. Ranganathan and I. Foster. Decoupling Computation and Data Scheduling in Distributed Data-Intensive Applications. In *High Performance Distributed Computing, 2002. HPDC-11 2002. Proceedings. 11th IEEE International Symposium on*, pages 352–358, 2002. 24

[44] Jose Renau, Basilio Fraguela, James Tuck, Wei Liu, Milos Prvulovic, Luis Ceze, Smruti Sarangi, Paul Sack, Karin Strauss, and Pablo Montesinos. SESC Simulator, January 2005. http://sesc.sourceforge.net. 24

[45] Sameer S Shende and Allen D Malony. The TAU Parallel Performance System. *International Journal of High Performance Computing Applications*, 20(2):287–311, 2006. 88

[46] Ben Shneiderman. The Eyes Have It: A Task by Data Type Taxonomy for Information Visualizations. In *Visual Languages, 1996. Proceedings., IEEE Symposium on*, pages 336–343. IEEE, 1996. 86

[47] L. Stanisic, E. Agullo, A. Buttari, A. Guermouche, A. Legrand, F. Lopez, and B. Videau. Fast and Accurate Simulation of Multithreaded Sparse Linear Algebra Solvers. In *Parallel and Distributed Systems (ICPADS), 2015 IEEE 21st International Conference on*, pages 481–490, Dec 2015. 25

[48] Luka Stanisic, Samuel Thibault, Arnaud Legrand, Brice Videau, and Jean-François Méhaut. Modeling and Simulation of a Dynamic Task-Based Runtime System for Heterogeneous Multi-Core Architectures. In *Euro-Par 2014 Parallel Processing*, pages 50–62. Springer, 2014. 25

[49] Luka Stanisic, Samuel Thibault, Arnaud Legrand, Brice Videau, and Jean-François Méhaut. Faithful Performance Prediction of a Dynamic Task-Based Runtime System for Heterogeneous Multi-Core Architectures. *Concurrency and Computation: Practice and Experience*, 27(16):4075–4090, 2015. 25

[50] H. Sutter. The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software. *Dr. Dobb's Journal*, 30(3), 2005. 1

[51] James J Thomas. *Illuminating the Path:[The Research and Development Agenda for Visual Analytics]*. IEEE Computer Society, 2005. 100

[52] R. M. Tomasulo. An Efficient Algorithm for Exploiting Multiple Arithmetic Units. *IBM J. Res. Devel.*, 11(1):25–33, 1967. DOI: 10.1147/rd.111.0025. 13

[53] François Trahay, Yutaka Ishikawa, François Rue, Raymond Namyst, Mathieu Faverge, and Jack Dongarra. EZTrace: A Generic Framework for Performance Analysis. In *Cluster, Cloud and Grid Computing (CCGrid), 2011 11th IEEE/ACM International Symposium on*, pages 618–619. IEEE, 2011. 89

[54] Philippe Virouleau, Pierrick Brunet, François Broquedis, Nathalie Furmento, Samuel Thibault, Olivier Aumage, and Thierry Gautier. Evaluation of openmp dependent tasks with the kastors benchmark suite. In *Using and Improving OpenMP for Devices, Tasks, and More*, pages 16–29. Springer, 2014. 81

[55] Richard Vuduc, James W. Demmel, and Jeff A. Bilmes. Statistical Models for Empirical Search-Based Performance Tuning. *Int. J. High Perform. Comput. Appl.*, 18(1):65–94, February 2004. 24

[56] Michelle Q Wang Baldonado, Allison Woodruff, and Allan Kuchinsky. Guidelines for Using Multiple Views in Information Visualization. In *Proceedings of the working conference on Advanced Visual Interfaces*, pages 110–119. ACM, 2000. 100

[57] R. Clint Whaley and Anthony M. Castaldo. Achieving Accurate and Context-Sensitive Timing for Code Optimization. *Softw. Pract. Exper.*, 38(15):1621–1642, December 2008. 30

[58] Asim YarKhan. *Dynamic Task Execution on Shared and Distributed Memory Architectures*. PhD thesis, University of Tennessee, December 2012. 15

[59] Asim YarKhan, Jakub Kurzak, and Jack Dongarra. QUARK Users' Guide: QUeueing And Runtime for Kernels. Technical report, Innovative Computing Laboratory, University of Tennessee, 2011. 15

# Vita

Blake Haugen was born and raised in Northeast Iowa. He graduated from Waverly-Shell Rock High School in May 2006. Upon graduation, he attended Wartburg College. He graduated in May 2010 with a Bachelor of the Arts degree with a double major in Computer Science and Engineering Science. In August of 2010, Blake started work as a Graduate Research Assistant at the Innovative Computing Laboratory at the University of Tennessee. He received his M.S. in Computer Science in May 2012 and plans to graduate with a Ph.D. in Computer Science in May 2016.