12-2012

Dynamic Task Execution on Shared and Distributed Memory Architectures

Asim YarKhan
yarkhan@utk.edu

To the Graduate Council:

I am submitting herewith a dissertation written by Asim YarKhan entitled "Dynamic Task Execution on Shared and Distributed Memory Architectures." I have examined the final electronic copy of this dissertation for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Doctor of Philosophy, with a major in Computer Science.

Jack J. Dongarra, Major Professor

We have read this dissertation and recommend its acceptance:

Michael W. Berry, Kenneth Stephenson, Stanimire Tomov

Accepted for the Council:
Carolyn R. Hodges

Vice Provost and Dean of the Graduate School

(Original signatures are on file with official student records.)

Dynamic Task Execution on Shared and Distributed Memory Architectures

A Dissertation

Presented for the

Doctor of Philosophy

Degree

The University of Tennessee, Knoxville

Asim YarKhan

December 2012

This dissertation is dedicated to my family.

To my wife, Sissie, for her love, support, and encouragement throughout this seemingly never-ending task.

To my parents, Wajid and Edelgard, who got me started on this path.

To my boys, Alexander and Nicholas, who bring joy to my life.

Acknowledgements

I would like to thank my adviser, Dr. Jack Dongarra for his guidance and support during the completion of this dissertation. I could not have completed this work without the support of the Innovative Computing Laboratory and all the people here.

I would like to thank Dr. Mike Berry, Dr. Ken Stephenson and Dr. Stan Tomov for being on my committee and providing me with feedback. Dr. Berry encouraged me to restart my PhD program when I had considered putting it aside. Dr. Tomov provided a regular sounding board, where I could bring my problems and concerns.

I would also like to thank my friends and collaborators at the Innovative Computing Lab for many pleasant times spent together, at lunch, shooting the breeze, during long coffee breaks, or going to state parks. Thanks to Jakub Kurzak, Heike McCraw ("Are you done yet?"), Stan Tomov, Piotr Luszczek, Dulcinea Becker, George Bosilca, Aurelien Bouteiller, Mathieu Faverge, Thomas Herault, Fengguang Song, Azzam Haidar, Hatem Ltaief, Julien Langou, Anthony Denalis, and Paul Peltz.

My wonderful wife Sissie, who put up with my decades long quest to complete this dissertation. Thank you again for all your patience and perseverance. I would never have got it all together without you. My two boys, Alexander and Nicholas, who bring so much fun, excitement, and entertainment to my life. Thank you both!

Abstract

Multicore architectures with high core counts have come to dominate the world of high performance computing, from shared memory machines to the largest distributed memory clusters. The multicore route to increased performance has a simpler design and better power efficiency than the traditional approach of increasing processor frequencies. But, standard programming techniques are not well adapted to this change in computer architecture design.

In this work, we study the use of dynamic runtime environments executing data driven applications as a solution to programming multicore architectures. The goals of our runtime environments are productivity, scalability and performance. We demonstrate productivity by defining a simple programming interface to express code. Our runtime environments are experimentally shown to be scalable and give competitive performance on large multicore and distributed memory machines.

This work is driven by linear algebra algorithms, where state-of-the-art libraries (e.g., LAPACK and ScaLAPACK) using a fork-join or block-synchronous execution style do not use the available resources in the most efficient manner. Research work in linear algebra has reformulated these algorithms as tasks acting on tiles of data, with data dependency relationships between the tasks. This results in a task-based DAG for the reformulated algorithms, which can be executed via asynchronous data-driven execution paths analogous to dataflow execution.

We study an API and runtime environment for shared memory architectures that efficiently executes serially presented tile based algorithms. This runtime is used to enable linear algebra applications and is shown to deliver performance competitive with state-of-the-art commercial and research libraries.

We develop a runtime environment for distributed memory multicore architectures extended from our shared memory implementation. The runtime takes serially presented algorithms designed for the shared memory environment, and schedules and executes them on distributed memory architectures in a scalable and high performance manner. We design a distributed data coherency protocol and a distributed task scheduling mechanism which avoid global coordination. Experimental results with linear algebra applications show the scalability and performance of our runtime environment.

Contents

Chapter 1

Introduction and Motivation

1.1 Motivation

Starting in the last couple of decades, computational modeling and simulation have often been referred to as the "third pillar of science", standing next to theory and experimentation as a way of exploring and understanding the world. However, computational science is a relatively young methodology and it has a long way to go before it is mature. There have been many remarkable successes enabled by computational science, such as sequencing the human genome, but the future of this approach is still wide open.

Enabling that promise of computational science will require much focus on the vital core of computational science, software and the mathematical models and algorithms encoded by the software (Buttari et al., 2007). Complexities introduced by newer hardware design, such as many-core processors, and very large distributed-memory clusters, make the task of programming and using this hardware efficiently a very large challenge for the computational scientist. The increase in the number of available resources increases the relative cost of any stall in the computational execution. For example, consider an algorithm that has a sequential step; if a dual-core node is executing this algorithm one of the cores is idle for a short time; however, if a 48-core machine is executing this algorithm, then the opportunity cost of the stall is much higher since 47 cores remain idle for that time.

It is difficult to require that a computational scientist should adapt their code to each hardware platform, and to adapt the code as the platform changes with increased hardware

resources. As is often the case in computer science, the complexities of adapting to the underlying hardware should be encapsulated in some level of abstraction.

In this work, a solution is proposed for the problems that computational scientists face in the presence of these complex hardware resources. A software framework that simplifies the process of developing applications that achieve high performance and scalability on a variety of platforms. The software framework described in this thesis exposes an application programming interface (API) that a computational scientist could use to develop dynamic, adaptive applications which scale from shared memory multicore machines to large distributed memory clusters.

1.2 Introduction

Recent years have seen changes in computer architecture that result in larger machines with increasing number of cores per CPU. This trend can be expected to continue for the near future, with ever larger numbers of cores available in large shared memory machines, which are combined into even larger distributed memory clusters as in Fig. 1.1. From a users point of view, the availability of all this parallel power is welcome, but the ability to create applications that efficiently and effectively use this architecture is challenging. The complexities of using such architectures start at the level of the highly multicore machines, and any complexities are made much greater with the addition of the distributed memory clusters.

Traditional scientific libraries and development methodologies are finding it difficult to efficiently manage and use the high number of computational cores that have become available in these complex architectures. Even though alternative programming approaches exist for development on these many-core and distributed memory architectures, MPI (Snir et al., 1998) remains a standard for achieving high performance in distributed memory machines. However, MPI is not designed to fully utilize these many-core architectures designs in a shared memory environment, so programming is complicated by the addition of a thread management systems such as Pthreads or OpenMP. As a result, developing efficient and scalable software for a complex, many-core, distributed-memory architecture remains a challenging and arduous task. In order to ease the task of programming such

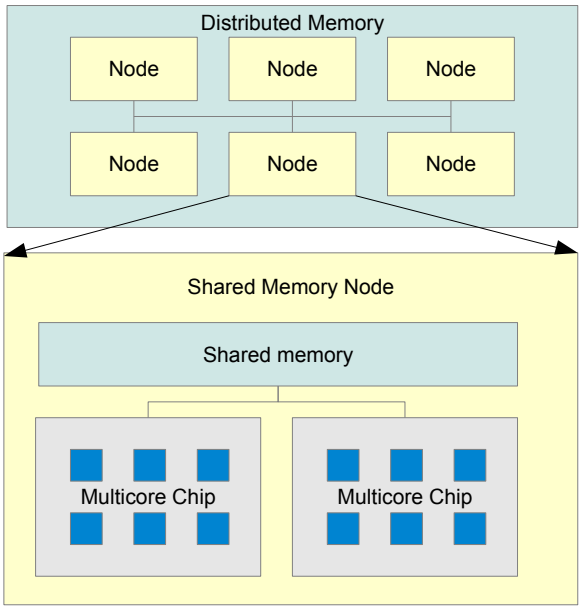


Figure 1.1: Distributed memory architecture containing many-core nodes

software, we investigate and develop a programming model with an easy-to-use API which supports transparent message passing between distributed memory nodes, and an efficient shared memory runtime within a many-core node.

As a very concrete example of a scientific software library, this work will use the development of the PLASMA linear algebra library (Agullo et al., 2010) as its driving application. Linear algebra algorithms are vital to many areas of scientific computing, and any improvement in the performance of these libraries can have a beneficial effect on a variety of fields. The development of the PLASMA library has been spurred by ubiquitous adoption of many-core architectures, and the inability of earlier software libraries to fully take advantage of these architectures.

This work is developed and presented in two parts. The first part develops QUARK (QUeuing And Runtime for Kernels), a simple API and runtime environment for many-core shared memory machines. QUARK enables PLASMA to implement complex algorithms using small computational kernels, and QUARK executes these kernels in a asynchronous, dynamic, superscalar fashion that transparently preserves all the data dependencies between the kernels. QUARK additionally provides support for constructs that enable specializations for linear algebra libraries.

The second part of this work develops QUARKD (QUeuing and Runtime for Kernels in Distributed Memory), an extension to QUARK for distributed memory clusters. QUARKD transparently supports distributed memory nodes without requiring changes at the algorithmic level. QUARKD manages task scheduling, data movement, scalability and hardware adaptation for the computational scientist accessible via a productive, easy-to-use programming interface.

1.3 Thesis Statement

The main objective of this dissertation is to investigate how to dynamically schedule a sequentially expressed algorithm to achieve scalability and resource efficiency in many-core and distributed memory architectures.

This dissertation addresses the problems of productivity, scalability and efficiency in current complex architectures, and develops an asynchronous, dynamic, data-driven approach that addresses these problems. This approach is exposed via a productive, easy-to-use, serial task-insertion API and implemented in a runtime environment. The runtime environment is initially studied in the context of multicore, shared-memory machines. The dynamic data-driven approach is then extended and examined on distributed memory multicore architectures.

1.4 Contributions

The goals and criteria for success in this project are measured in terms of productivity, scalability and efficiency. Keeping these criteria in mind, the contributions of this dissertation can be summarized as follows:

- A task-insertion API for expressing and implementing kernel based computations that is *productive* and *expressive*. This API has been used successfully to implement the linear algebra library PLASMA. This simple function-call style API has inspired a similar API in the StarPU project (Augonnet, 2011, pg. 74).

- An asynchronous, dynamic, data-driven runtime environment for multicore shared memory machines that is *scalable* and *efficient*. The QUARK runtime environment is

designed to be easily integrated into any code that is constructed as a serial sequence of kernel operations. QUARK is deployed as a part of the successful PLASMA linear algebra library and it has also has been distributed for stand alone use. Experimental results show high performance and scalability on linear algebra algorithms.

- A distributed memory runtime environment that extends the asynchronous, data-driven execution paradigm to a distributed-memory cluster and demonstrates *scalability* and *efficiency* at large scale. The QUARKD runtime is shown to be competitive on linear algebra applications at scales of 1200 cores in a distributed memory machine. QUARKD works from the same algorithmic structure as the shared-memory PLASMA implementation, thus increasing the *productivity* of the user.

- A distributed algorithm based on a serial task-insertion API that schedules distributed tasks and manages data coherency without requiring global coordination.

New and original contribution This dissertation shows that the highly productive *serial task-insertion* programming style can give a scalable and high performance distributed memory execution that is competitive on large scale machines.

1.5 Outline of the Dissertation

This dissertation is organized as follows.

- Chapter 2 introduces the historical background to data-driven computing and discusses recent research and projects that are relevant. Since linear algebra applications are used to motivate and evaluate this research, background for tile-based algorithms is presented.

- Chapter 3 describes QUARK (QUeuing and Runtime for Kernels) the multicore shared-memory runtime environment for scheduling and executing sequentially expressed code. Experimental results validate our data-driven, superscalar approach to the execution of task kernels.

- Chapter 4 describes QUARKD (QUeuing and Runtime for Kernels in Distributed Memory), our dynamic, distributed-memory runtime for executing serially expressed

code. We describe the algorithms and protocols for task scheduling and distributed non-coordinated data consistency. Experimental results show scalability and performance results.

- Chapter 5 concludes this dissertation, describing the broader impact of this work, and providing directions for future work.

Chapter 2

Background and Related Work

2.1 Introduction

This chapter reviews prior work done to address dataflow programming. Some of the more recent projects are motivated by the difficulties raised by the emergence of many-core shared-memory and distributed memory architectures. We present some historical context leading to the current research directions and discuss a few relatively recent projects that have implemented solutions to the expression and implementation of data flow programs.

We also present the background for linear algebra applications, which act as a driver for the implementation of our solution. The background for classic block-structured algorithms is presented. Newer tile-based linear algebra algorithms are shown to result in tasks with data dependencies, and thus to Directed Acyclic Graphs (DAGs) of tasks.

2.2 Dataflow Execution

Research in graph models of execution or dataflow execution started at the end of the 1960s (Rodrigues, 1969). Dataflow execution research was motivated by the desire to exploit the massive parallelism of large scale distributed systems by enabling asynchronous execution paths. Since classical von Neumann systems were thought to be unsuitable to dataflow execution, the early 1970s saw implementations of various forms of dataflow architectures that relied on specialized hardware to implement fine-grained instruction-level dataflow operations (Dennis, 1980) (Gurd et al., 1985). During the 1970s and 1980s, there was

an often-expressed view that dataflow architectures would eventually replace von Neumann architecture. However, a 1986 survey article (Veen, 1986) reviewed various implementations of experimental dataflow machines and pointed out some of the difficulties in fine grain dataflow architectures. The primary problem is with storage space, where the effective utilization is about 40 percent because of all the meta-data information maintained for each data item. In terms of computation overhead, fine grained dataflow architectures have a numerical efficiency that is substantially lower (2 to 10 times less) than standard architectures because of the complex flow control that needs to be managed.

During the 1990s, the research focus in dataflow moved from fine-grain architecture design to implementing larger grained dataflow algorithms on traditional von Neumann style architectures. During this period there was a lot of research on languages for describing dataflows (see survey article by Johnston et al. (2004)). In specific there was a lot of work on dataflow based visual programming languages such as SciRun (Parker and Johnson, 1995) and LabView (Johnson and Jennings, 2001). However, the focus of this dataflow research was primarily programming productivity rather than exploiting parallelism.

During the last decade, a combination of various factors has driven hardware designers to building processors that have an increasing number of computational cores. From a practical perspective, this change in hardware means that many applications and algorithms have to be rewritten to take advantage of the fine-grained parallelism provided by these multi-core processors. Thus, the ideas and concepts from dataflow are being revisited within the context of the current architecture designs.

2.3 Current and Related Work

Many current projects approach task parallelism using coarse-grain operations expressed via functional parallelism and executed in a shared memory architecture. Functional parallelism, even though it is a relatively straightforward approach to generating parallel execution paths, has the problem that it tends to result in a fork-join or bulk-synchronous form of parallelism. The resulting loss in parallelism at the synchronization points has become relatively more expensive, given the increasing number of cores in modern multi-core and many-core architectures. However, several projects have tried to extend beyond the

simple fork-join style model of parallelism by implementing dataflow based asynchronous execution. The following discussions are roughly broken down into shared and distributed memory projects and are presented chronologically,

2.3.1 Task Execution in Shared Memory

Charm++ (Kale and Krishnan, 1993) and Adaptive MPI (AMPI) (Huang et al., 2003) use processor virtualization and asynchronous remote method invocation for parallelism in a distributed memory architecture. Programs are decomposed into a number of message-passing, cooperating units called *chares*. Charm++ and AMPI use an implicit dependency graph with a "block and yield" model to handle dependency coordination. However, because of this, Charm++ and AMPI have a problem managing large call graphs, so Charisma (Huang and Kale, 2007) was developed to orchestrate the tasks in the call graph and to specify the producer-consumer communication patterns between objects.

The highly influential Cilk project for shared-memory machines (Blumofe et al., 1995) implements a compiler directed divide-and-conquer style functional parallelism that leads to a fork-join style of execution. The scheduling mechanism in Cilk keeps idle processors busy by using work stealing, where an idle processor *steals* work from another processor in order to remain productive. The use of simple compiler extensions for programming Cilk makes it very easy to incorporate parallelism into a pre-existing program, however the parallelism is limited to the granularity that is exposed when the functions are defined. The fork-join style parallelism that is primarily supported in Cilk does not allow us to utilize the hardware to the desired degree.

OpenMP (Dagum and Menon, 1998) is a well known standard for developing shared-memory application. It uses compiler directive to insert directives into code to specify data layout, parallelization and synchronization. These directives are then used to generate applications that use library and runtime support to execute in parallel. The most common use for OpenMP is in parallelizing regular loops, though other modes of parallelism exist. The standard use of OpenMP for parallelizing regular loops results in fork-join style parallelism. The OpenMP 3.0 standard (Ayguade et al., 2009) has added some support for task level parallelism and dynamic sections. However, it is difficult to express complex

relationships using OpenMP, and it remains the users responsibility to ensure that there are no dependencies between parallel sections.

The SMARTS project (Vajracharya et al., 1999) is a runtime system for shared memory architectures which partitions loop iterations into sub-iterations with data dependencies, and performs a data-parallel graph-based execution respecting the data dependencies. This enables a form of task parallelism in combination with data parallelism. However, the level of the task parallelism contained in these loop iterations is too fine grained for our DAG applications and would result in unnecessary overhead.

The SuperMatrix project (Chan et al., 2007; Chan, 2010) provides a runtime environment in support of the FLAME linear algebra project (Gunnels et al., 2001) on shared-memory machines. This runtime environment takes its inspiration from techniques for dynamic scheduling and out-of-order operations as seen in superscalar processors. It implements in a dataflow style implementation where the dependency analysis and execution is managed by a Tomasulo style scoreboard (Tomasulo, 1967). The described implementation of SuperMatrix exposes the entire dependence graph before executing, and this is likely to hinder scalability as the size of the problem increases.

The SMP Superscalar (SMPSs) project (Pérez et al., 2008) is a compiler based system that uses PRAGMA directives to annotate tasks that can be run in parallel and to declare read/write information about their data parameters. Then, the compiler infers the dependencies between tasks from the data declarations and a runtime system executes a task-DAG in a shared-memory environment. SMPSs assigns tasks to cores in order to increase data locality, uses work stealing as a scheduling mechanism, and puts an emphasis on avoiding data hazards by using data copying if possible. SMPSs is part of family of projects, grouped together under the name *StarSs*. The various projects target different architectures, e.g., GPUSs targets GPUs, CellSs targets the Cell processor. In a shared memory environment, SMPSs has similar goals and structure to our QUARK project. In a distributed memory environment, Marjanović et al. (2010) implement a mode of MPI over SMPSs, where the communication is explicitly encapsulated in user specified tasks. The explicate MPI calls make it harder to express complex distributed communication patterns. Additionally, the standard usage of SMPSs is via a compiler front end, which makes it somewhat cumbersome when used within the context of a linear algebra library and in

combination with other multithreaded libraries. Nevertheless, SMPSs shares many goals and performance characteristics with the work described here.

The StarPU project (Augonnet et al., 2009) is a runtime environment for task scheduling on shared memory architectures with an emphasis on supporting heterogeneous (e.g., GPU) execution and an expressive data management protocol. Data localization in the heterogeneous environment is managed internally using cache coherence style protocols. StarPU uses data dependencies to create a task DAG. It profiles task execution and uses historical runtime data to schedule tasks on the appropriate core of the heterogeneous machine. StarPU has been extended to provide MPI support, but the focus is on small scale clusters with GPUs and not large distributed installations. StarPU shares many the goals of our work, but focuses on supporting GPUs and other heterogeneous hardware.

2.3.2 Distributed Memory Implementations

The ScaLAPACK project (Blackford et al., 1996) provides the reference implementation of a distributed memory linear algebra library. It is designed to be highly scalable and load balanced in a homogeneous environment. However, ScaLAPACK implements a bulk synchronous form of parallelism which suffers from the same fork-join problems seen in LAPACK, where highly parallel BLAS3 operations are interspersed with operations with much lower parallelism. We believe that using a dataflow model is going to expose a higher degree of overall parallelism and enable greater parallelism during execution.

Husbands and Yelick (2007) have demonstrated a distributed memory implementation of graph-based execution for dense LU factorization code, using co-operative thread scheduling techniques along with UPC's partitioned global address space for remote information management. The performance demonstrated this approach is impressive, but a more generalized technique would be required to handle a complete linear algebra library.

The TBLAS project by Song, YarKhan, and Dongarra (2009) describes an earlier effort directed toward creating a task based runtime environment. This work grew out of an attempt to create a task-based implementation of BLAS to use as the underlying computation layer within the ScaLAPACK library. The result of this work was an scalable, well performing implementation of several algorithms, rather than a general purpose library

based on a simple, productive API. The performance achieved by TBLAS was a motivating factor for developing a general purpose task execution library.

The DAGuE project described by Bosilca et al. (2010b) is creating generic framework for architecture aware scheduling and management of micro-tasks on distributed many-core heterogeneous architectures. Applications are represented by a specialized structure similar to a parameterized task graph, which contains all the relationships between tasks in a compact, problem-size independent fashion. This structure can be queried to determine data-dependencies, so the task-DAG is implicitly available at all times. This representation leads to a runtime which has no overhead in determining the data dependencies. DAGuE assigns computation threads to the cores, overlaps communications and computations and uses a dynamic, fully-distributed scheduler based on cache awareness, data-locality and task priority. However, creating the parameterized representation of the tasks is a painstaking process requiring expert knowledge. DAGuE, in its current form, cannot be easily used by the average algorithm designer to create and experiment with numerical algorithms.

2.4 Overview of Linear Algebra*

Linear algebra algorithms form the core of a large number of scientific computing applications. Improvements in the performance of these algorithms can enable substantial advances in science, for example, by making more detailed simulations possible in less time. As a concrete example, weather simulations need to be completed before the weather actually arrives; tsunami simulations would need to be completed before the effects arrive.

2.4.1 Block Algorithms

Linear algebra libraries have evolved over time to match available hardware . The LAPACK (Anderson et al., 1999) library was originally developed in the 1980s for shared-memory systems with a focus on cache memory management. However, the architectures at that time did not provide a large degree of shared memory multicore parallelism, so it was expected that the underlying BLAS (Basic Linear Algebra Subroutines) libraries would

*This section was previously presented in Haidar, A., Ltaief, H., YarKhan, A., and Dongarra, J. (2011). Analysis of dynamically scheduled tile algorithms for dense linear algebra on multicore architectures. *Concurr. Comput. : Pract. Exper.*, 24(3):305–321.

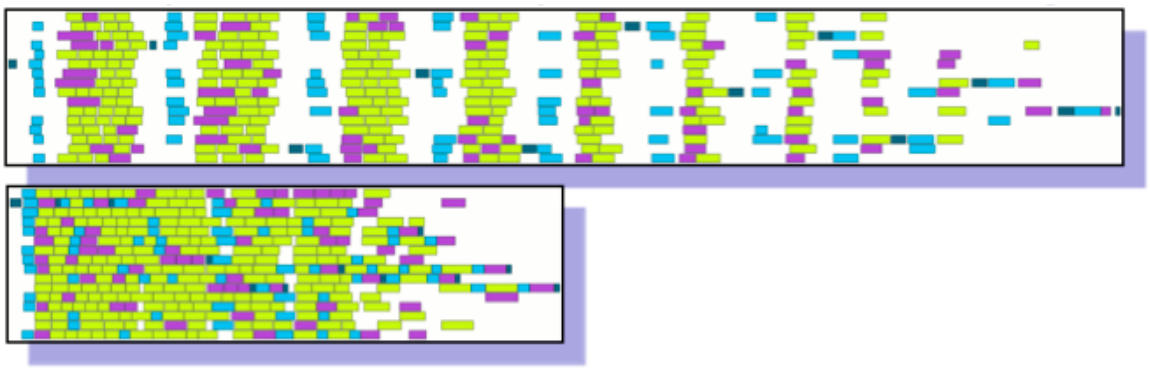


Figure 2.1: Fork-join execution (top) versus asynchronous execution (bottom) of the same task based linear algebra algorithm. The trace shows the execution of different tasks on different threads as tiles of different colors, with white space meaning that a thread is idle.

provide any needed parallelism. The result of this design decision was a fork-join style of parallelism, where single core work may be followed by highly parallel Level-3 BLAS routines, which is then followed by a synchronization point and another serial section of the code. When there were just a few cores available, the loss of performance due to the synchronization was minimal. However, as the number of available cores has increased, this opportunity cost has become substantial. Fig. 2.1 shows both the trace of a fork-join style execution and the data driven asynchronous execution of the same task based algorithm; the difference between the execution time for the two implementations can be substantial.

Here we review the software design behind the LAPACK library for shared-memory. In particular, we focus on three widely used factorizations used in scientific computing, i.e., QR, LU and Cholesky. These factorizations will be used throughout this dissertation to guide and evaluate our research.

LAPACK provides a broad set of linear algebra operations aimed at achieving high performance on systems equipped with memory hierarchies. The algorithms implemented in LAPACK leverage the idea of algorithmic blocking to localize the operations to smaller chunks of data which can be held in the faster, smaller levels of the memory hierarchy. This limits the amount of memory bus traffic in favor of high data reuse from the faster, higher level memories such as L1 and L2 cache memories.

The idea of blocking revolves around an important property of Level-3 BLAS operations (matrix-matrix operations), the surface-to-volume property, which means that for Level-3

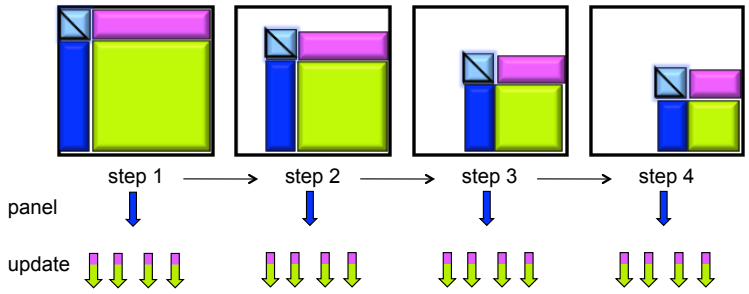


Figure 2.2: Standard steps in a block algorithm, consisting of a panel factorization and a trailing matrix update. These steps lead to a fork-join execution.

operations $\theta(n^3)$ floating point operations are performed on $\theta(n^2)$ data. Because of this property, Level-3 BLAS operations can be implemented in such a way that data movement is minimal and reuse of data in the cache is maximized. Level-3 BLAS operations can be highly efficient, obtaining a large fraction of the theoretical peak performance of the hardware. Block algorithms consist of recasting linear algebra algorithms in a way that only a negligible part of the computations are done as Level-2 BLAS operations (matrix-vector operations), where data reuse is minimal, while most is done in Level-3 BLAS. Most of these blocked algorithms can be described as the repetition of two fundamental steps shown in Fig. 2.2:

- Panel factorization: Depending on the linear algebra operation that has to be performed, a number of transformations are computed for a small portion of the matrix (the so called panel). The effect of these transformations, computed using Level-2 BLAS operations, can be accumulated.

- Trailing submatrix update: In this step, all the transformations that have been accumulated during the panel factorization step can be applied at once to the rest of the matrix (i.e., the trailing submatrix) by means of Level-3 BLAS operations.

The parallelism in block structured algorithms is enabled at the level of each BLAS routine. Although the panel factorization step represents a small fraction of the total number of operations ($\theta(n^2)$ from a total of $\theta(n^3)$), it consists of Level-2 BLAS operations that are memory bound and cannot be efficiently parallelized in shared memory machines. The Level-3 BLAS routines, such as matrix-matrix multiplication, which can be effectively parallelized are responsible for most of the parallelism in the block structured algorithms.

This results in the fork-join execution for block factorizations since such the execution of such a factorization is a sequence of sequential operations (panel factorization) interleaved with parallel ones (updating the trailing submatrix).

The ScaLAPACK library (Blackford et al., 1996) is a distributed memory implementation of linear algebra that is based on the same block-structured algorithms as LAPACK. ScaLAPACK also obtains it's parallelism at the BLAS level, that is, by means of a distributed PBLAS library (Choi et al., 1996). ScaLAPACK has the same limitation as LAPACK, where the Level-2 BLAS operations cannot be efficiently parallelized on distributed multicore architectures.

The fork-join execution of block structured algorithms have the following two problems:

- Scalability is limited because the relative cost of the sequential Level-2 BLAS operations increases as the degree of available parallelism increases.
- Asynchronicity is not available because the block-structured algorithms block all other computational activity while synchronizing around the sequential Level-2 BLAS tasks.

As such, architectures with high number of processors and cores can benefit substantially if algorithms can be rewritten to allow asynchronous execution and greater scalability.

2.4.2 Tile Linear Algebra Algorithms

Here we outline a solution that removes the fork-join overhead seen in block algorithms. Based on tile algorithms, this new model is currently used in shared memory libraries, such as PLASMA from the University of Tennessee (Agullo et al., 2010) and FLAME from the University of Texas at Austin (Gunnels et al., 2001).

The tile-based approach to linear algebra algorithms has been presented and discussed in Buttari et al. (2007), Quintana-Ortí and Van De Geijn (2008), Buttari et al. (2008), Kurzak et al. (2008), and Kurzak and Dongarra (2009b). The tile approach consists of breaking the panel factorization and trailing submatrix update steps into smaller tasks that operate on relatively small $nb \times nb$ tiles (or blocks) of consecutive data which are organized into block-columns as shown in Fig. 2.3. The algorithms can then be restructured as tasks (which are basic linear algebra operations) that act on tiles of the matrix. The data dependencies

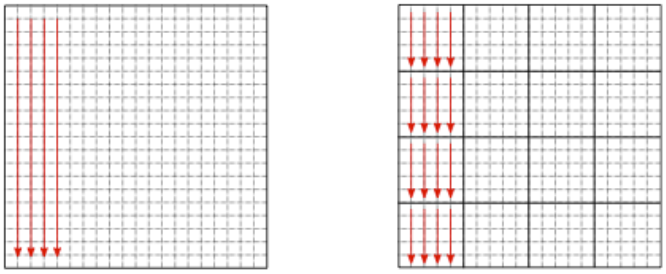


Figure 2.3: Difference between the standard LAPACK layout (left) and the tile data layout (right). The tile data layout uses a contiguous memory region to hold a tile of the matrix.

between these tasks result in a Directed Acyclic Graph (DAG) where nodes of the graph represent tasks and edges represent dependencies 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 this asynchronous scheduling to result in an out-of-order execution where slow, sequential tasks are hidden behind parallel ones. This would be managed by having the sequential tasks start early, as soon as their dependencies are satisfied, while some of the parallel tasks (submatrix updates) from the previous iteration still remain to be performed and can be executed in parallel.

Matrix factorization algorithms form a core operation for scientific computation, since they are used as the first step for finding the solution vector x for a linear system $Ax = b$. The tile versions of the Cholesky, QR and LU factorization algorithms are used to drive the development of asynchronous runtime system developed in this work. For reference, these algorithms are outlined here.

Tile Cholesky factorization

The Cholesky factorization is used during the solution of a linear system $Ax = b$, where A is symmetric and positive definite. Such systems arise often in physics applications, where A is positive definite due to the nature of the modeled physical phenomenon. The Cholesky factorization of an $n \times n$ real symmetric positive definite matrix A has the form $A = LL^T$, where L is an $n \times n$ real lower triangular matrix with positive diagonal elements. The tile Cholesky algorithm processes the matrix by tiles, where the matrix

Algorithm 1 Tile Cholesky factorization algorithm

1: **for** $k = 1, 2$ to NT **do**
2: {Cholesky factorization of the tile $A_{k,k}$}
3: DPOTF2(A_{kk})
4: **for** $i = k + 1$ to NT **do**
5: {Solve $A_{kk}X = A_{ik}$}
6: DTRSM(A_{kk}, A_{ik})
7: {Update $A_{ii} \leftarrow A_{ii} - A_{ik}A_{ik}^T$}
8: DSYRK(A_{ii}, A_{ik})
9: **end for**
10: **for** $i = k + 2$ to NT **do**
11: **for** $j = k + 1$ to i **do**
12: {Update $A_{ij} \leftarrow A_{ij} - A_{ik}A_{jk}$}
13: DGEMM($A_{ij}A_{ik}, A_{jk}$)
14: **end for**
15: **end for**
16: **end for**

consists of $NT \times NT$ tiles. In Algorithm 1, some standard BLAS routines are used during the factorization: DSYRK (symmetric rank-k update), DPOTF2 (unblocked Cholesky factorization), DGEMM (general matrix-matrix multiplication), DTRSM (triangular solver). The dominant operation of the Cholesky factorization comes from the innermost loop of the trailing matrix update and is the very efficient Level-3 BLAS matrix-matrix multiplication (DGEMM).

Tile QR factorization

The QR factorization implemented in LAPACK is a factorization of an $m \times n$ real matrix A is the decomposition of A as $A = QR$, where Q is an $m \times m$ real orthogonal matrix and R is an $m \times n$ real upper triangular matrix. The QR factorization uses a series of elementary Householder matrices of the general form $H = I - \tau vv^T$, where v is a column reflector and τ is a scaling factor.

The tile QR algorithm produces essentially the same factorization as the LAPACK algorithm, but it differs in the Householder reflectors that are produced and the construction of the Q matrix. The algorithm is outlined in Algorithm 2 (for details see Gunter and van de Geijn (2005) or Buttari et al. (2009)).

In order to restructure the QR algorithm as a tile algorithm, the dominant operation from the innermost loop is different from the standard LAPACK implementation. In

Algorithm 2 Tile QR factorization algorithm

```
1: for k = 1, 2 to NT do
2:    DGEQRT(A_{k,k}, T_{k,k})
3:    for n = k + 1 to NT do
4:       DORMQR(A_{kk}, T_{kk}, A_{kn})
5:    end for
6:    for m = k + 1 to NT do
7:       DTSQRT(A_{kk}, A_{mk}, T_{mk})
8:       for n = k + 1 to NT do
9:          DTSMQR(A_{kn}, A_{mn}A_{mk}, T_{mk})
10:      end for
11:   end for
12: end for
```

LAPACK, the dominant operation is the highly optimized DGEMM and in the tile algorithm it is a new kernel operation DTSMQR. The DTSMQR operation, even though it is a matrix-matrix operation, has not been tuned and optimized to the extent of DGEMM, so it reaches a much lower percentage of peak performance on a machine.

Tile LU factorization

In LAPACK, the LU factorization (or LU decomposition) with partial row pivoting of an $m \times n$ real matrix A has the form $A = PLU$, where L is an $m \times n$ real unit lower triangular matrix, U is an $n \times n$ real upper triangular matrix and P is a permutation matrix. In the block formulation of the algorithm, factorization of nb columns (the panel) is followed by the update of the remaining part of the matrix (the trailing submatrix).

The tile LU factorization algorithm in Algorithm 3 follows the same pattern as the QR algorithm, and is described in greater detail in Buttari et al. (2009). The tiled algorithm does not use the partial pivoting strategy, but instead another strategy called incremental pivoting or block-pairwise pivoting. In the tile LU algorithm, only two tiles of the panel are factorized at a time, so the pivoting only takes place within two tiles at a time. This can cause the tile algorithm to be less numerically stable than the partial pivoting LU factorization in LAPACK.

Algorithm 3 Tile LU factorization algorithm

```
1: for k = 1, 2 to NT do
2:    DGETRF(A_kk, IPIV_kk)
3:    for n = k + 1 to NT do
4:       DGESSM(IPIV_kk, A_kk, A_kn)
5:    end for
6:    for m = k + 1 to NT do
7:       DTSTRF(A_kk, A_mk, L_mk, IPIV_mk)
8:       for n = k + 1 to NT do
9:          DSSSSM(A_kn, A_mn, L_mk, A_mk, IPIV_mk)
10:      end for
11:   end for
12: end for
```

2.5 Summary

Dataflow execution has a long history, and it has reemerged in the form of data-driven asynchronous execution for large grained tasks. The most relevant related projects for our research are the StarPU, SMPSs, and DAGuE projects, but none of these has the same goals of productivity, scalability and performance. StarPU does address productivity, but it has a focus on heterogeneity and does not address scalability for large distributed machines. SMPSs addresses productivity by defining language extensions, but its distributed memory management is not as productive since it requires explicit control. DAGuE excels at scalability and performance, but programming DAGuE applications is not productive since it requires the user to write and analyze parameterized task graphs.

Linear algebra algorithms are used as a development driver for our project. We discuss how libraries such as LAPACK and ScaLAPACK obtained their parallelism, and explain why that does not fit with current machine architectures. We discuss how tile based rewrites of linear algebra algorithms result in DAGs of tasks with data dependencies between the tasks. We outline the tile Cholesky, QR and LU factorizations, since these tile based algorithms will be used to guide our implementations of asynchronous data-driven task execution environments.

Chapter 3

Dynamic Task Execution In Shared Memory *

We have introduced the problems involved in developing efficient applications for many-core architectures and we have described some prior work on data-driven execution as a solution to these problems. In this chapter we introduce QUARK (QUeuing And Runtime for Kernels), our solution for executing tasks in a multicore shared-memory architecture. We describe our API for task insertion in QUARK, which is designed to be productive for a programmer. Since the development of QUARK is driven by linear algebra, we describe some linear algebra optimizations included in QUARK and demonstrate some of the advantages of a dynamic runtime, such as DAG composition. Experimental results demonstrate the performance and scalability of QUARK on multicore shared-memory architectures. The shared memory QUARK runtime environment is an independent and vital component of the PLASMA linear algebra library.

*Material in this chapter has been published in the following: (1) Kurzak, J., Luszczek, P., YarKhan, A., Faverge, M., Langou, J., Bouwmeester, H., and Dongarra, J., Multithreading in the PLASMA Library. In *Handbook of Multi and Many-Core Processing: Architecture, Algorithms, Programming, and Applications* (Kurzak et al., 2013); and (2) Haidar, A., Ltaief, H., YarKhan, A., and Dongarra, J., Analysis of dynamically scheduled tile algorithms for dense linear algebra on multicore architectures. *Concurr. Comput. : Pract. Exper.*, (Haidar et al., 2011).

3.1 Introduction

Modern shared memory machines have a large number of cores, ranging from just a few cores to single memory image machines with thousands of cores. It has been argued in Buttari et al. (2009) that to achieve high performance on such architectures, algorithms will have to be expressed at a *fine-granularity* for improved local cache management, and with high *asynchronicity* to take advantage of the available core and reduce synchronization points.

Taking *fine granularity* and *asynchronicity* as our guides, our goal is to enable a programmer to easily obtain high performance from shared memory machines by creating and executing as many concurrent tasks as possible. The programmer should be protected from having to deal with the details of knowing when a task may execute, where it should execute, and what tasks may execute next.

Our solution is to develop an API and runtime environment that allow the programmer to submit tasks to be executed in a serial fashion. The runtime examines the data dependencies between the tasks and executes the tasks in such a way as to preserve the sequential consistency of the original tasks. Additionally, the runtime system strives to take advantage of cache locality and to keep all the cores as busy as possible.

The driving applications that guide the development of our solution and test its efficiency come from linear algebra. The freely available, de-facto standard, linear algebra library LAPACK (Anderson et al., 1999) was written to obtain parallelism via synchronous calls to parallel BLAS routines, interspersed by lower levels of parallelism. This results in a fork-join style of parallelism, which we hope to improve using dataflow approaches, resulting in a new, higher performing freely-available library. Another, a reason to use linear algebra as a driving application is that many commercial linear algebra libraries exist that can be used to judge the quality of our solution, to see if we achieve the competitive performance. More details about tiled algorithms for linear algebra can be found in various studies, including work by Buttari et al. (2007), Kurzak et al. (2008), Buttari et al. (2009), and Quintana-Ortí and Van De Geijn (2008). These motivate the use of tiled algorithms and explain the differences between tiled versions of algorithms and the older block structured algorithms.

The runtime environment that we develop in this work requires that applications are composed of tasks that operate on contiguous data. Not all applications will map easily to this structure, but there is a large class of applications, including those from the linear algebra domain, that are well suited.

3.1.1 Moving to a Dynamic Runtime

The tile based approach to linear algebra application is being explored and implemented within the PLASMA project (Agullo et al., 2010). This project has been re-visiting approaches to linear algebra algorithms in the context of many-core shared-memory machines, and restructuring linear algebra algorithms as a sequence of tasks that operate on contiguous-memory tiles of data. This sequence of tasks can then be scheduled for an out-of-order execution, which can hide the work done by the sequential bottleneck tasks. The tasks along with the data dependencies between the tasks define an implicit DAG where the tasks are the nodes and the data dependencies between the tasks form the edges. We would like to execute the DAG of tasks on shared-memory, many-core architectures in a flexible, efficient and scalable manner.

In early versions of the PLASMA project, a statically defined progress table was used to track task dependencies and to enable the succeeding tasks. The design of this progress table for each algorithm is a formidable undertaking for an expert, and can hinder the creation of complex algorithms. This motivated exploration of a dynamic approach to determining dependencies and scheduling tasks which would not be dependent on a complex progress table. An early proof-of-concept project by Kurzak and Dongarra (2009a) demonstrated the value of dynamic scheduling for linear algebra algorithms in the shared memory environment.

In this chapter of the dissertation, we will present the QUARK project (QUeuing And Runtime for Kernels) that has built a new runtime environment for dynamic task scheduling YarKhan et al. (2011b). QUARK uses enhanced parameter information provided in calls to kernel tasks to determine the data dependencies between these tasks. We will detail how these tasks are managed by the QUARK runtime to enforce data dependencies and schedule execution. QUARK has been tuned to make many optimizations that would be

very difficult to implement using static progress-table scheduling. For example, QUARK uses information about locality to try to reuse data that has been cached in local cache memory. QUARK can take hinting to allow it to reorder certain operations to maximize efficiency.

There are many algorithms currently in PLASMA where designing a static progress table is difficult due to the complex data dependency relationships between the tasks, for example, recursive LU factorization (Kurzak et al., 2013), Cholesky inversion (Agullo et al., 2011), and LU inversion (Dongarra et al., 2011). For these algorithms, using the simplified interfaces of QUARK and allowing QUARK to dynamically manage the data dependencies and use asynchronous out-of-order scheduling to execute the application has greatly improved productivity while simultaneously providing high performance.

Experiments in this chapter will show that QUARK is as efficient as the static progress table scheduling, and in many situations QUARK has performance advantages in addition to the productivity advantage discussed earlier. More specifically, algorithms executing under QUARK can be composed, effectively compressing the DAGs and thus allowing an even greater speedup when multiple algorithmic operations are performed on the same data set; and in a shared-resource environment, the dynamic task scheduling in QUARK will automatically do load balancing, so more processing will occur on lightly loaded resources and less on highly loaded-machines.

3.2 Runtime Architecture

In this section we present an overview of the QUARK runtime environment for dynamic task scheduling in shared memory, guiding the discussion from the perspective of an algorithm writer who is using the runtime to create and to execute an algorithm.

An idealized overview of the architecture of the QUARK runtime environment is show in Fig. 3.1. The user inserts tasks into the system using a serial task-insertion API. After each task is inserted into the runtime, the arguments for the task are used to make a dependency structure, where reads and writes on data are queued. These queues of data requests are checked for data dependencies and the ordering of these dependencies forms an implicit DAG for the tasks. The runtime system schedules the tasks that are not waiting for

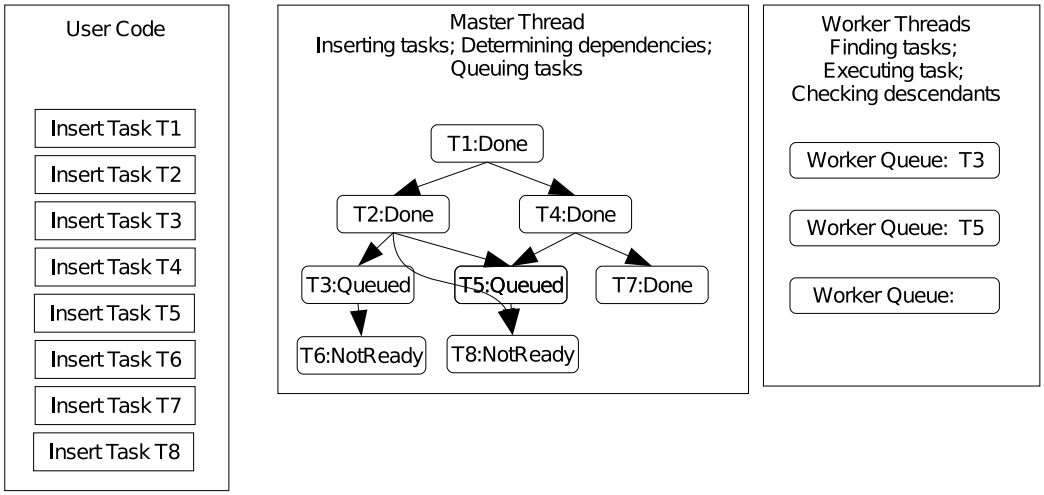


Figure 3.1: Idealized architecture diagram for the QUARK shared-memory runtime environment. The user's thread runs serial code and, when acting as the task master, it inserts tasks into a (implicit) DAG based on their dependencies. Tasks can be in NotReady, Queued or Done states. When dependencies are satisfied, tasks are queued and executed by worker threads. When tasks are completed, further dependencies can be marked as satisfied and further tasks can be scheduled for execution.

dependencies, and execution threads pickup and execute the tasks. After a task completes execution, for each data parameter, the queue of reads and writes are updated to reflect the completed task. Then the runtime schedules any released tasks that do not have to wait for any more dependencies.

3.3 Data Dependencies

In order for the runtime to be able to determine dependencies between the tasks, it needs to know how each task is using its arguments. Constant arguments are marked as `VALUE`, which means that they are not dependencies, but values like constants or loop indices; these values are just stored with the task. Actual dependencies can be `INPUT`, `OUTPUT`, or `INOUT`, which have the expected meanings. Given the sequential order that the tasks are inserted into the runtime system, and the way that the arguments are used, we can infer the relationships between the tasks.

- A task can read a data item that was written by a previous task, a read-after-write (RAW) dependency.

- A task can write a data item that was written by previous task, a write-after-write (WAW) dependency.
- A task can write a data item that was read by a previous task, a write-after-read (WAR) dependency.

These are the dependencies that define the DAG, however this DAG is never explicitly realized in the runtime environment. The DAG is implicit in the way that tasks are queued and block on data items, waiting for the appropriate access to the data.

To summarize, the dependency analysis is done based on two factors:

1. The tasks are added in the desired sequential order.
2. The data items queue up read requests and write requests to the data.

In order to determine if a task can be executed, there are two basic rules that need to be enforced on data access:

- Writes cannot proceed until all earlier access (read or write) to the data is completed.
- Reads cannot proceed until all earlier writes are completed Multiple reads can proceed in parallel.

When the data access rules are enforced an execution ordering of the tasks is obtained. A simple example is shown in Fig 3.2 and demonstrates how a sequence of tasks can be translated into queues of data access requests, which can then be executed via a parallel ordering on the tasks.

The resolution of the task dependencies has an analogue in the instruction level *superscalar* parallelism within a single CPU, where multiple machine instructions can be simultaneously dispatched to different function units provided there are no data hazards between them.

3.3.1 Shared Memory Algorithm

The shared memory algorithm is outlined in Fig. 3.3 in a very simplified form. This ignores the complications in scheduling tasks for data locality, tracking tasks for cancellation, etc.

```
Task A reads v, writes x
Task B reads y, writes z
Task C reads x, writes y
Task D reads x,z, writes v
Task E reads y, writes z

          A     B     C     D     E
v         r                 w
x         w           r     r
y               r     w           r
z               w           r     w

A can proceed at once
B can proceed at once
Note A and B can run in parallel.
C can proceed after A writes x (RAW) and B reads y (WAR)
D can proceed after A writes x (RAW), A reads v (WAR), and B writes z (WAW)
Note C and D can run in parallel.
E can proceed after C writes y (RAW) and D reads z (WAR)

So the task DAG ordering is [A|B] - [C|D] - E
```

Figure 3.2: Simple data dependency example

This algorithm is shown in two stages, since there are two ways a task can reach the point where it is scheduled for execution. Firstly, when a task T_a is inserted, if none of its data accesses (`INPUT`/`INOUT`/`OUTPUT`) have data hazard conflicts with earlier tasks, then it can be scheduled for execution at insertion time. Secondly, when a task T_b has completed its execution, its data item accesses are removed. Each data item may have other accesses that were queued and could not be scheduled until task T_b completed. The other access may now be granted, and any task T_c that now has gained access to all its data parameters can be scheduled for execution.

Shared Memory Task Insertion API The basic task insertion API is shown in Fig. 3.4. The details of the API can be found in Appendix A, but the most important item for determining the data dependencies and the implicit DAG are the argument flags. These flags express the information about how the argument is to be used in the function (`INPUT`, `INOUT` or `OUTPUT`).

```
// When inserting tasks
for each task T_a as it is inserted
    for each dependency A_i in T_a
        enqueue T_a access to A_i using rw-mode
        if no earlier access conflicts, T_a gets access to A_i
    if T_a can get all its A_i dependencies
        schedule T_a for execution

// When tasks complete
for each task T_b as it is completed
    for each dependency A_i in T_b
        remove T_b access to A_i
        release conflicting data hazards that are queued on A_i
        if some other task T_c has no more hazard-blocked dependencies
            schedule T_c for execution
```

Figure 3.3: Outline of shared memory algorithm

```
QUARK_Insert_Task( Quark *quark, void (*function) (Quark *),
    Quark_Task_Flags *tflags,
    int sizeof_arg_1_in_bytes, void *arg_1, int arg_1_flags,
    int sizeof_arg_2_in_bytes, void *arg_2, int arg_2_flags,
    ...,
    0 );
```

Figure 3.4: QUARK shared memory task insertion API

3.3.2 Tile Cholesky Factorization

The outline of the shared memory algorithm, and the API for task insertion have been presented. We will now put these into practice by following a tile Cholesky factorization as implemented in PLASMA and executed by the QUARK runtime environment.

Fig. 3.5 shows the pseudocode for the tile Cholesky factorization as an algorithm designer might view it. Tasks in this Cholesky factorization depend on previous tasks if they use the same tiles of data. If these dependencies are used to relate the tasks, then a directed acyclic graph (DAG) is implicitly formed by the tasks. A small DAG for the Cholesky factorization of a 5×5 tile matrix is shown in Fig. 3.6. The DAG visualizations can be produced automatically by the runtime environment and have proven to be useful for algorithm development, debugging and verification.

The PLASMA library uses function wrappers in order to write code that maps very closely to the pseudocode shown for the Cholesky factorization in Fig. 3.5. This loop-based formulation of code is relatively straightforward for a programmer of linear algebra

algorithms and allows flexibility in easily experimenting with further algorithms. In Fig. 3.7 we see the final code implemented in PLASMA which can be seen to closely resemble the pseudocode from Fig. 3.5. Each of the calls to the core linear algebra routines is represented by a call to a wrapper where the parameters are decorated with information about their sizes and their usage (`INPUT`, `OUTPUT`, `INOUT`, `VALUE`). As an example, in Fig. 3.8 we can see how the `DPOTRF` call is decorated for the use by the runtime environment.

The tasks are inserted into the runtime, which stores them to be executed when all the dependencies are satisfied. The execution of ready tasks is handled by worker threads that simply wait for tasks to become ready and execute. Workers get their tasks using a simple scheduling heuristic discussed in the next section. The thread doing the task insertion is referred to as the master thread. When there are sufficient tasks inserted into the runtime, the master thread will also switch to a mode where it will execute tasks.

3.4 Task Scheduling

When all the dependencies of a task are satisfied, the task is scheduled for execution by placing it in a worker-thread specific ready queue. By default, data locality is used to determine which worker thread executes a task. A task is assigned to the worker thread that has most recently written its output data, thus attempting to reuse data that may still be resident in the cache for that thread. If there is more than one output parameter, the first output parameter is used to determine locality. If the user has knowledge that

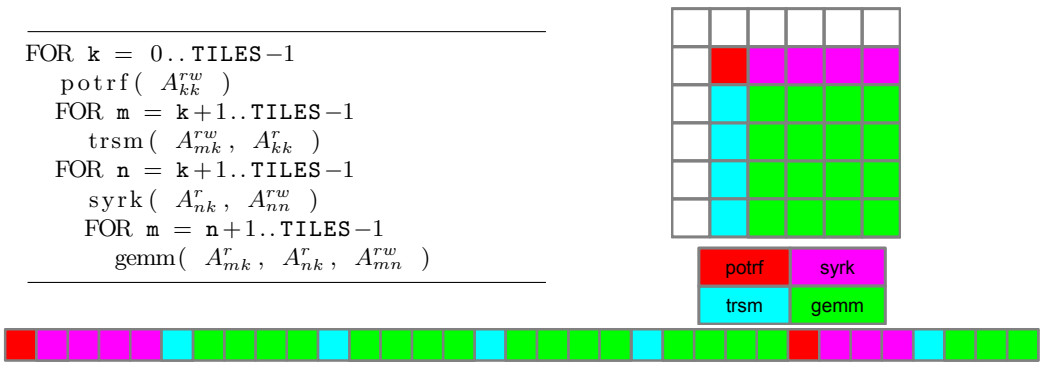


Figure 3.5: Pseudocode for the tile Cholesky factorization, when acting on a matrix. The lower figure visualizes a sequence of tasks unrolled by the loops.

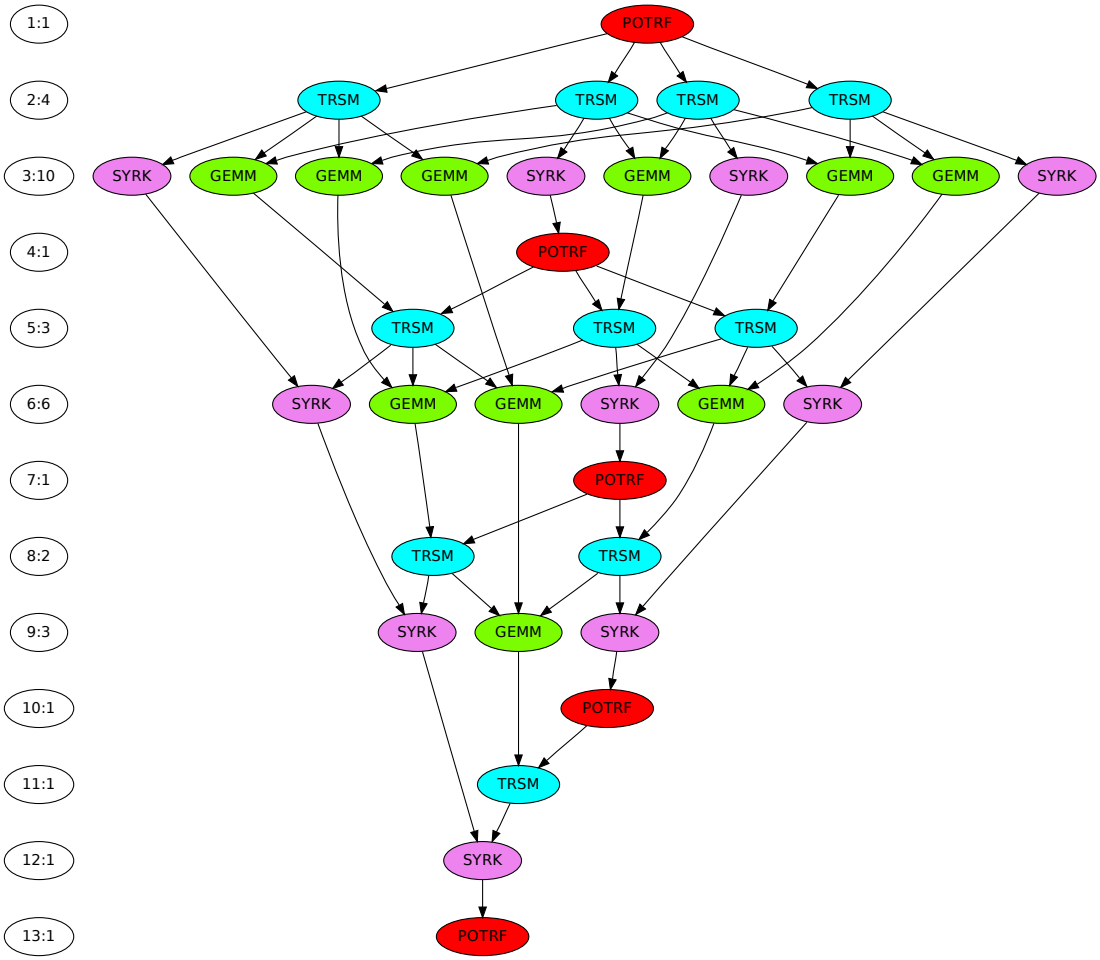


Figure 3.6: DAG for a small Cholesky factorization (right looking version) with five tiles (tile size 200 and matrix size 1000). The column on the left shows the depth:width of the DAG.

```
#define A(m,n) BLKADDR(A, double, m, n)
void plasma_pdpotrf_quark( PLASMA_enum uplo, PLASMA_desc A, ...)
  for (k = 0; k < M; k++)
    QUARK_CORE_dpotrf( quark, A(k, k), ... )
    for (m = k+1; m < M; m++)
      QUARK_CORE_dtrsm( quark, A(k, k), A(m, k), ...);
    for (m = k+1; m < M; m++)
      QUARK_CORE_dsyrk( quark, A(m, k), A(m, m), ...);
      for (n = k+1; n < m; n++)
        QUARK_CORE_dgemm( quark, A(m, k), A(n, k), A(m, n), ...);
```

Figure 3.7: Tile Cholesky factorization as implemented in PLASMA, uses the QUARK runtime to schedule and execute the core linear algebra operations.

```
void QUARK_CORE_dpotrf(Quark *quark, Quark_Task_Flags *task_flags,
       int uplo, int n, int nb, double *A, int lda,
       PLASMA_sequence *sequence, PLASMA_request *request, int iinfo)
  QUARK_Insert_Task(quark, CORE_dpotrf_quark, task_flags,
      sizeof(PLASMA_enum),                  &uplo,      VALUE,
      sizeof(int),                          &n,         VALUE,
      sizeof(double)*nb*nb,                  A,         INOUT,
      sizeof(int),                          &lda,       VALUE,
      sizeof(PLASMA_sequence*),             &sequence,  VALUE,
      sizeof(PLASMA_request*),              &request,   VALUE,
      sizeof(int),                          &iinfo,     VALUE,
      0);
```

```
void CORE_dpotrf_quark(Quark *quark)
{
    int uplo; int n;  double *A; int lda; int info;
    quark_unpack_args_7(quark, uplo, n, A, lda, sequence, ...);
    info=LAPACKE_dpotrf_work(LAPACK_COL_MAJOR, lapack_const(uplo), n, A, lda);
    if (info != 0) plasma_sequence_flush(quark, ...);
}
```

Figure 3.8: Example of inserting and executing the DPOTRF task as part of the Cholesky factorization. The QUARK_CORE_doptrf routine inserts a task into the runtime, passing it the sizes and pointers of arguments and their usage (INPUT, OUTPUT, INOUT, VALUE). Later, when the dependencies are satisfied and the task is ready to execute, the runtime schedules the CORE_dpotrf_quark task to be executed. During execution the arguments are unpacked and the actual dpotrf computation is executed. For simplicity, the error handling part of the code was removed.

can assist in the scheduling of the task, then this default assignment can be overridden by designating some other parameter to be the locality determining parameter.

If a worker has no remaining work assigned to it, a work stealing heuristic comes into play and allows the worker to steal a task from the end of another workers ready queue. The work is stolen from the end of the queue in order to minimize the effect on the data locality. However, any use of work stealing may break the data locality based assignment since the stolen task may cause data to move unexpectedly.

The combination of these two heuristics, locality based assignment and work stealing, results in a simple, effective approach for scheduling in QUARK. The overall effect is that the scheduling mechanism adapts well to modern multicore architectures, which can include mechanisms such as dynamic frequency scaling and may have multiple competing processes on a socket. More complex scheduling methods (e.g., critical path based methods) would find it difficult to adapt to dynamically changing environments.

QUARK's simple, practical approach to scheduling has enabled some interesting heuristics, allowing the programmer to provide hints to the scheduler and allowing the scheduler to reorder tasks as needed. Some of these techniques are discussed in section. 3.6.

3.4.1 A Window of Tasks

QUARK has always been developed with the intention of executing large complex codes, especially since the driving applications come from linear algebra where $\theta(n^3)$ tasks are not unexpected. For example, In Fig. 3.9 we see the DAG for a relatively small LU factorization using 20×20 tiles; even this small problem generates 2870 tasks. For a slightly larger problem size consisting of 1000×1000 tiles, we would have a DAG with $\theta(1000^3)$ (a billion) tasks. In such a case doing a critical path analysis would be a very costly operation. Figure 3.10 depicts the growth of the number of tasks when varying the number of tiles.

If we were to unfold and retain the entire DAG of tasks for a large problem, we may be able to perform some interesting analysis with respect to DAG scheduling and critical paths. However, the size of the data structures would quickly grow overwhelming. Our solution to this is to maintain a configurable window of tasks. The implicit DAG of active tasks is then traversed through this sliding window, which should be large enough to ensure all cores are

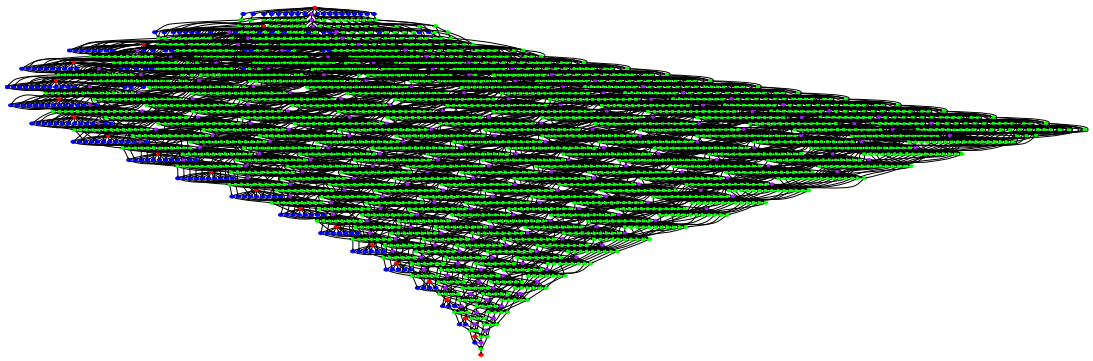


Figure 3.9: DAG for a relatively small LU factorization with 20x20 tiles (tile size 200 and matrix size 4000) generates 2870 tasks. The size of the DAG grows very fast with the number of tiles.

kept busy. When this window size is reached, the master thread which is inserting tasks does not accept any more tasks until some are completed. During this period, the master thread will switch to a computation mode and work on ready tasks.

The use of a task window limits the resources required by the runtime environment, but it also can limit the lookahead available to the scheduler. In theory, this may decrease the amount of parallelism available, however in practice, there is sufficient parallelism available with relatively small window sizes (Haidar et al., 2011).

3.4.2 Effect of Tile Size

The tile size, which corresponds to the amount of work done by a single task, is a very important tuning parameter in QUARK. The amount of work in a task needs to be large enough to hide the overhead of the runtime system. However, if it is too large, then, depending on the algorithm being executed, the amount of available parallelism can be decreased. If the data tile size is too small, then the runtime system can suffer some internal negative feedback effects (e.g. mutex lock contention) which can cause the performance to drop drastically.

Some of the drastic effects of tuning or mistuning the tile size become evident in Fig. 3.11. For QR factorization in PLASMA, there are two different tile block sizes that need to be

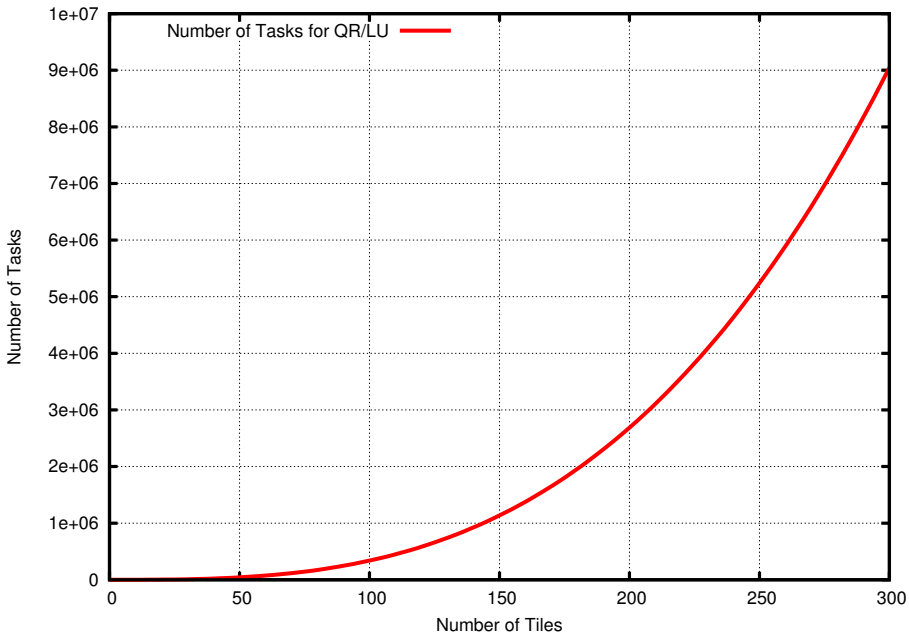


Figure 3.10: The rapid growth of number of tasks for the linear algebra applications. For many linear algebra algorithms the number of tasks grows as $\theta(n^3)$ with the number of tiles. This mean for large problems keeping the entire DAG in memory quickly grows intractable. The QUARK runtime uses a sliding window of active tasks to make the problem manageable.

tuned, an inner block size (IB) and main tile size (NB). In a tuning step not shown here, a large part of the (IB, NB) space was explored and IB=36 was found and is used in the NB exploration in Fig. 3.11. For QR factorization, the IB and NB tile sizes have an effect on each other, which explains jagged shape of the QR curve. For both the Cholesky and QR algorithms shown in the figure, the optimal tile size is found to be NB=180. However, these parameters need to be adjusted from machine to machine since they depend on low level machine characteristics. In future work, it would be interesting to explore autotuning mechanisms that will find the appropriate tuning parameters for a machine.

3.5 Parallel Composition

A vital feature of QUARK is the ability to do *parallel composition*, that is, the ability to compose a number of smaller task graphs into a larger merged task graph. This allows the composed DAG to reveal more parallelism, thus enabling the greater overall performance.

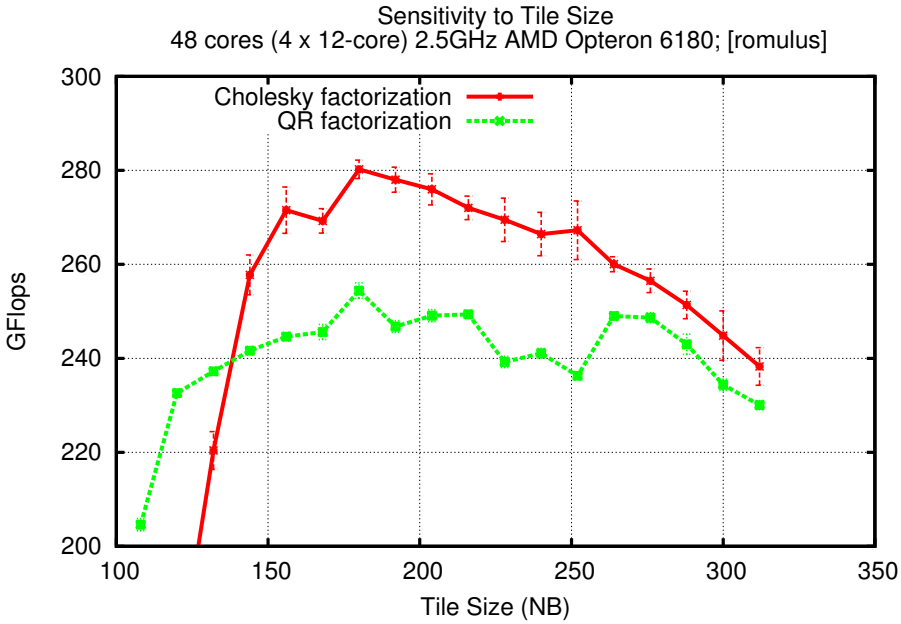


Figure 3.11: The effect of tile size on Cholesky and QR factorization shows that there is an optimal tile size and performance drops sharply when the tile size strays far from that optimum. For QR factorization there is an inner blocking size (IB) that has already been tuned to 36.

The Cholesky inversion routine in PLASMA shows the advantages of parallel composition, and has been presented in detail by Agullo et al. (2011) and Kurzak et al. (2013). The advantages of parallel composition have also been shown in Dongarra et al. (2011) for the LU inversion of a matrix.

3.5.1 Composing the Cholesky Inversion

In the following presentation, the benefits of parallel composition for Cholesky inversion are detailed. The mathematical details of the algorithms can be found in Agullo et al. (2011) and Kurzak et al. (2013), but for the purposes of this discussion, the Cholesky inversion involves three separate components: computing the Cholesky factorization (POTRF), inverting the L factor (TRTRI), and computing the inverse matrix (LAUUM). Each of these components corresponds to a sequence of kernel tasks, where each task is a call to a BLAS operation on a tile of data. These kernel tasks are inserted into the runtime environment, and scheduled and executed by QUARK. Figure 3.12 shows the DAGs of each of the three components

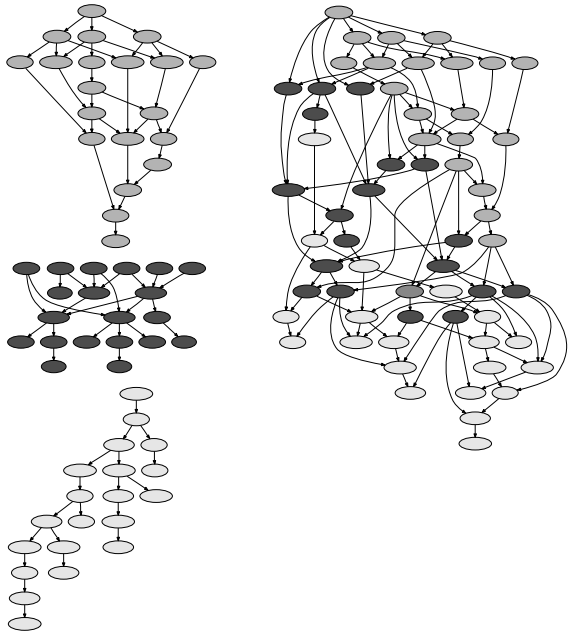


Figure 3.12: DAGs for Cholesky inversion show the three different operations (POTRF, TRTRI, LAUUM) involved in the inversion. The merged DAG on the right shows how composing the DAGs can improve execution.

individually, and then the composed DAG of the entire Cholesky inversion routine. The aggregation of the components into a composed DAG is the automatic default behavior of QUARK when the components and their kernel tasks are inserted into the runtime. If the three components are not composed, then the DAG created is taller and thinner, with less parallelism and a longer critical path. The aggregate DAG provides the runtime with more width, thus exposing more parallelism and resulting in a shorter critical path.

Fig. 3.13 shows an execution trace where each of the phases of the Cholesky inversion are separated by barriers, and a second trace where the three phases have been composed into a single DAG. When the DAGs are composed, the trace results in a shorter overall execution time. Finally, Fig. 3.14 extends the experiment from the small demonstration problems shown so far, and presents the difference between the performance when a larger multicore system is used on larger problems. Using the composed DAG, with no barrier, demonstrates substantial performance benefits.

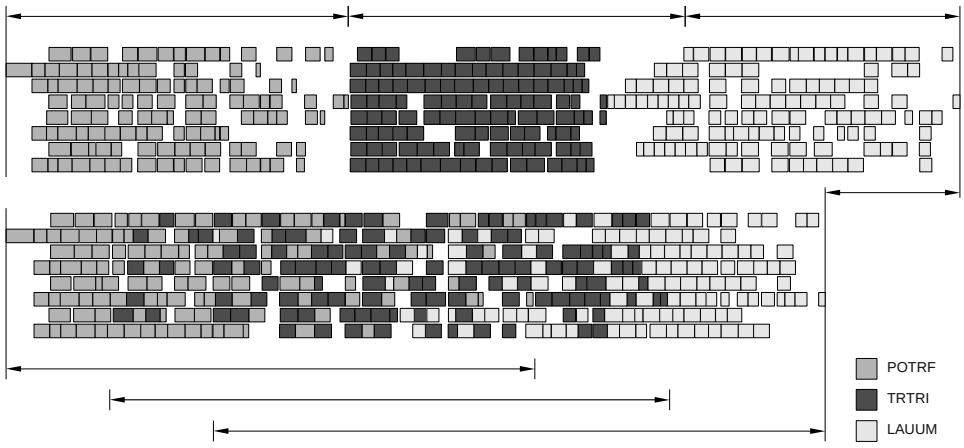


Figure 3.13: Trace for Cholesky inversion of a small matrix run on eight cores. The first trace has a barrier inserted between each operation. The second trace permits QUARK to compose the DAGs, the default behavior. The differences between the traces show that composing the DAGs can result in a decreased execution time.

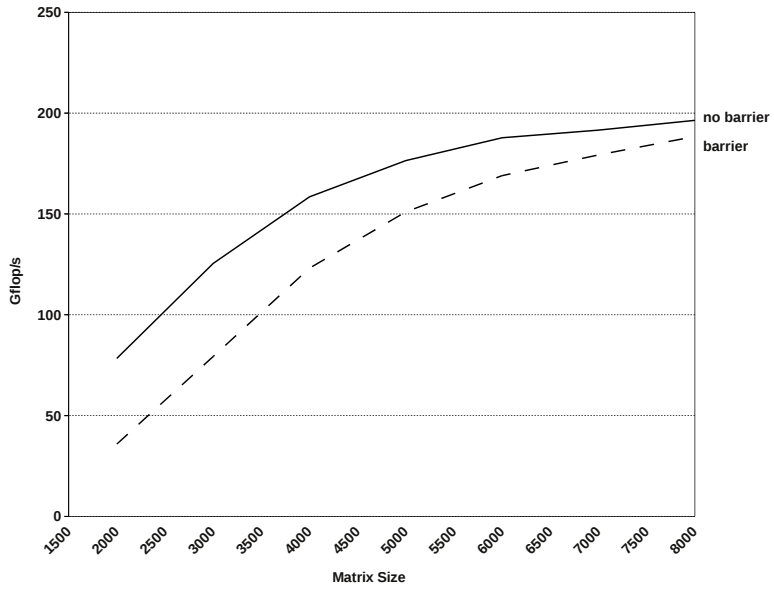


Figure 3.14: Performance for Cholesky inversion on a 48-core shared-memory AMD system. The curve with a barrier between the operations has a lower performance than the curve with the composed operations.

3.6 Extensions to the Runtime

QUARK offers a number of features that allow finer control over the runtime system and the task scheduling and execution. Most of these features are controlled either via the task flags passed into each task, or via the argument flags provided to the various arguments. Some of features are described and summarized here and they are presented in greater detail in Appendix A.

3.6.1 Adjustments to Data Dependencies

By its nature, QUARK depends on information about the data parameters in each task in order to infer data dependencies and task relationships. The user can pass additional flags associated with data parameters in order to change the runtime behavior. A few of the flags are discussed here, but a detailed discussion of these flags can be found in Appendix A.2.

It has been shown in the past that the reuse of memory caches can lead to a substantial performance improvement in execution time. Since we are working with tiles of data that should fit in the local caches on each core, we have provided the algorithm designer with the ability to hint the cache locality behavior. A parameter in a call can be decorated with the *locality flag* in order to tell the scheduler that the data parameter should be used from cache if possible. When a task is inserted with the *locality flag* on its parameters, the scheduler will attempt to assign that task to the same computational thread where the *locality* data item was last modified.

If a parameter is decorated with the *accumulator flag*, then QUARK is informed that it is being used to accumulate output. If a sequence of accesses to that data are all marked with an accumulator flag, then QUARK is free to reorder those accesses. For example, given two operations $C = C + A_1 * B_1$ followed by $C = C + A_2 * B_2$, the order in which the updates to C occur does not matter and the accesses and their corresponding operations may be reordered.

There is a *gatherv flag* that informs QUARK that the user guarantees that the writes to that data block will not conflict. If a sequence of accesses to a data block are marked with a *gatherv* flag, then they can all proceed simultaneously. This feature is used during

the process of data layout conversion, where data in LAPACK layout is translated in place to the block data layout.

In several linear algebra algorithms, the upper triangular, diagonal and lower triangular regions may be accessed independently and using different read-write modes. In QUARK a data tile can be marked as being composed of multiple *data regions*, and different data accesses patterns can be applied to each of the regions. This can enable multiple tasks to access the tile simultaneously and asynchronously.

3.6.2 Task Control

Tasks can be decorated with various flags that provide additional information to the runtime, a detailed discussion of these flags can be found in Appendix A.2 but a few of the more important task control options are reviewed here.

A user can provide a *task priority* value when they insert tasks in order to tell the worker threads how to order the list of ready tasks. This can be used to hint about which tasks are on the critical path. This has been used by several algorithms, for example, the recursively tiled LU factorization algorithm, in order hint the order of task processing and thus increase the priority of the panel factorization tasks.

Tasks can be *locked* to threads or thread masks, ensuring that a specific task will only run on a certain set of threads. This is important if the programmer feels that they must override the default cache locality based task scheduling because they have a greater knowledge of data locality or they wish to enforce a specific data locality. Task locking has also been used needs to use a task to control a GPU, and requires that a specific thread handle all communication with the GPU.

QUARK permits a form of *nested parallelism* that is very useful in DAGs where there are tasks on the critical path that are substantially slower than the other tasks in the DAG. Since these tasks are on the critical path, they can from a bottleneck in the execution of the DAG. QUARK allows the user to designate a task as a multi-threaded task, and can assign a user specified number of threads to this task. That way, a faster multithreaded implementation can be created for that task. The implementation of a LU factorization that uses this feature is discussed in section 3.7.3.

3.6.3 Parameter Aggregation

In certain circumstances, the number of parameters to be assigned to a task is either not known beforehand, or is computed during runtime based on various criteria (e.g., the size of the problem, the number of threads being used for computation). In such a circumstance, QUARK allows the user to add parameters to a task dynamically. This feature has been used by Kurzak et al. (2012) when using QUARK to manage multiple CPUs with a GPU in order to build the GPU parameters.

3.7 Experimental Results

The capabilities of our runtime environment are demonstrated through experimental evaluations. Linear algebra applications have been the main driver for the development of QUARK, and these applications result in very large, complex DAGs of tasks that can thoroughly stress the runtime and test its capabilities.

Execution experiments using the QUARK runtime are compared with the statically scheduled version of the tile algorithm in PLASMA. The statically scheduled implementation has minimal overheads, so when the algorithm designer has scheduled it correctly it can have near optimal asynchronous execution on the hardware. Finally, we compare performance with the high quality commercial implementation of LAPACK provided in the Intel MKL library v.2011.6.233.

Experimental Hardware For our experiments a large shared memory machine was used. The machine *romulus* has a NUMA architecture with 48 cores distributed in 4 sockets with 12 cores per socket, where each core is a 2.5 GHz AMD Opteron 6180 SE.

3.7.1 Cholesky Factorization

The implementation of tile Cholesky factorization was shown in 3.3.2. This factorization has a relatively simple dependency structure, shown in Fig. 3.6, with just one output parameter from each task. Here we present experiments that show that the QUARK runtime can achieve high performance when executing the Cholesky factorization.

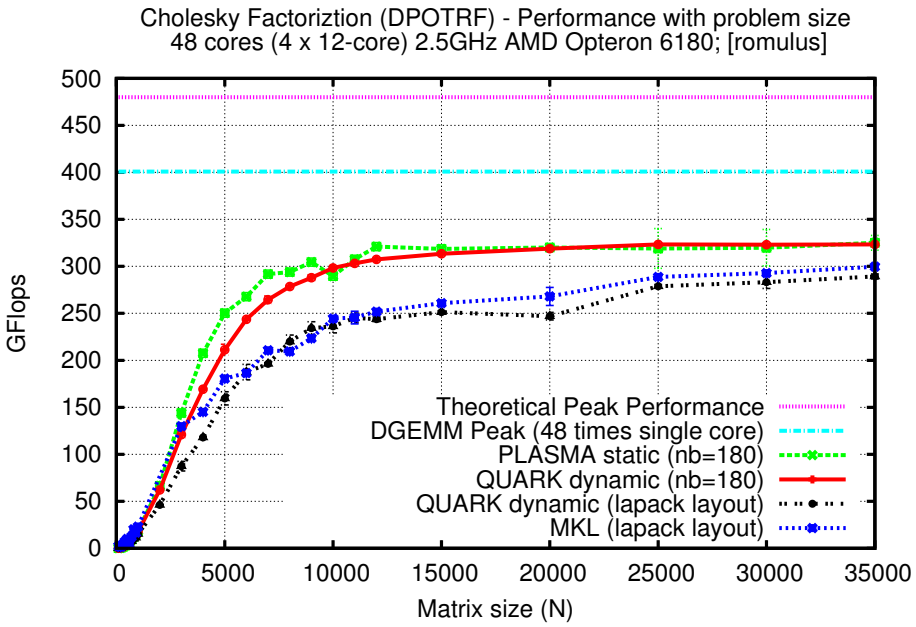


Figure 3.15: Performance for Cholesky factorization on a 48-core shared-memory AMD system.

Fig. 3.15 shows that the dynamically scheduled version of the Cholesky factorization achieves the same performance as the statically scheduled PLASMA implementation. We consider the statically scheduled PLASMA versions of algorithms to be the state-of-the-art publicly available implementations of tiled algorithms. If the dynamic PLASMA implementation performs as well as the PLASMA implementation, then we obtain all the benefits of QUARK (e.g., a productive task-insertion interface, DAG composition, dynamic task scheduling) while masking the overheads behind the computation.

Linear algebra algorithms often have an inner kernel operation that dominates asymptotic performance as the size of the problem increases. So, in an asymptotic sense, if the same kernel is being used by different implementations, then they would reach similar peak performance. In our evaluation, we generally focus on the behavior of the various implementations in the earlier part of the performance curves, before the asymptotic behavior is reached. In that region, the tile based PLASMA algorithms perform substantially better than the LAPACK based MKL implementations, reaching the asymptotic peak at much smaller problem sizes.

For the Cholesky factorization, the tile algorithms reach a slightly higher asymptotic performance than the MKL implementation. This is a little unexpected, since the asymptotic performance of Cholesky should be dominated by the kernel in its innermost loop, the Level-3 BLAS `DGEMM` kernel. This `DGEMM` kernel can also be seen to be the most prominent kernel by viewing the trace in Fig. 3.16. The higher asymptotic performance of the tile kernels may be explained by the fact that the tile size is tuned better for this machine.

One obvious criticism that can be raised when comparing the MKL implementation against the tiled algorithms is that using a *tile based layout* gives the algorithm an advantage. To make a fair comparison, the tiled implementation can include a layout translation step, where the original data starts in the LAPACK layout, then is transformed into a tiled layout, the computation is performed in the tiled layout, and then the result is transformed back into LAPACK layout. This mode of operation allows the tiled algorithms to be used as drop-in replacements for the standard linear algebra libraries which operate on LAPACK layout. As we can see in the line labeled "QUARK dynamic (lapack layout)", the performance with layout translation is competitive with that of the commercial MKL library but it is a little lower.

However, the appropriate usage mode for the PLASMA library would be for application scientists to convert their data to the tile layout and achieve the higher performing curves rather than converting back and forth from LAPACK layout. Alternatively, if a sequence of operations was to be performed on the data, the layout conversion cost would be paid once only and could be amortized over all the operations.

Fig. 3.16 shows an execution trace of the Cholesky factorization algorithm on a smaller set of 12 cores. This form of trace visualization provides a sense of the occupancy of the cores, and can be used to guide tuning and discover problems with the scheduling. There are several small spaces where the cores can be seen to be idle (black). These stalls are often associated with a `DPOTRF` (green) operation. The `DPOTRF` operations are along the critical path of the DAG for Cholesky factorization as shown in Fig. 3.6. So, if the computation threads run out of work at the time that a `DPOTRF` is to be processed, then the computation can stall. The presence of idle time is exaggerated in this example trace because the small

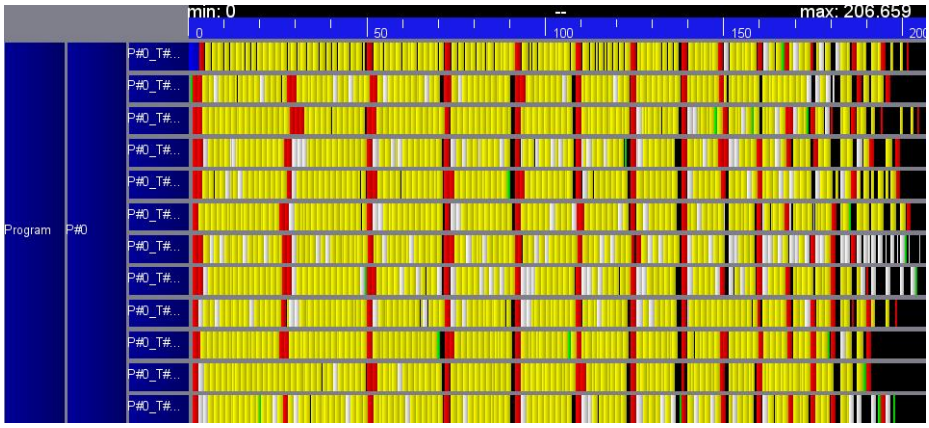


Figure 3.16: Trace for Cholesky factorization on a 12 cores of a shared-memory AMD system. The matrix is 3500x3500 and the tile size is 180. Idle processor time is shown as black. Color key: DPOTRF: green; DSYRK: white; DTRSM: red; DGEMM: yellow.

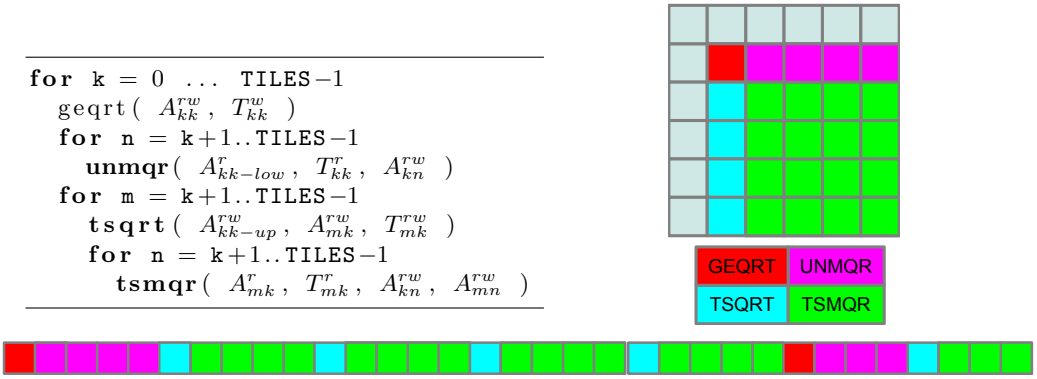


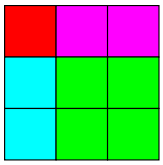
Figure 3.17: Pseudocode for the tile QR factorization.

size of the problem means that there is insufficient `DGEMM` work to keep all the computational threads fully occupied and overlap the `DPOTRF` operations.

3.7.2 QR Factorization

The tile based QR algorithm is described in Section 2.4.2 of the background chapter, and outlined with pseudocode in Fig. 3.17. The mapping from pseudocode to the implementation in PLASMA follows the same pattern as the Cholesky factorization described in Section 3.3.2. The loops in the pseudocode generate a sequence of task insertions as shown on the left of Fig. 3.18. Each of these tasks has a set of data parameters to which it needs access. These data access requests are queued up as shown on the right

F0	geqrt(A_{00}^{rw} , T_{00}^{w})
F1	**unmqr**(A_{00}^{r} , T_{00}^{r} , A_{01}^{rw})
F2	**unmqr**(A_{00}^{r} , T_{00}^{r} , A_{02}^{rw})
F3	**tsqrt**(A_{00}^{rw} , A_{10}^{rw} , T_{10}^{w})
F4	**tsmqr**(A_{01}^{rw} , A_{11}^{rw} , A_{10}^{r} , T_{10}^{r})
F5	**tsmqr**(A_{02}^{rw} , A_{12}^{rw} , A_{10}^{r} , T_{10}^{r})
F6	**tsqrt**(A_{00}^{rw} , A_{20}^{rw} , T_{20}^{w})
F7	**tsmqr**(A_{01}^{rw} , A_{21}^{rw} , A_{20}^{r} , T_{20}^{r})
F8	**tsmqr**(A_{02}^{rw} , A_{22}^{rw} , A_{20}^{r} , T_{20}^{r})
F9	geqrt(A_{11}^{rw} , T_{11}^{w})
F10	**unmqr**(A_{11}^{r} , T_{11}^{w} , A_{12}^{rw})
F11	**tsqrt**(A_{11}^{rw} , A_{21}^{rw} , T_{21}^{w})
F12	**tsmqr**(A_{12}^{rw} , A_{22}^{rw} , A_{21}^{r} , T_{21}^{r})
F13	geqrt(A_{22}^{rw} , T_{22}^{w})



Data dependencies from the first five tasks in the QR factorization	
A_{00} :	$F0^{rw} : F1^{r} : F2^{r} : F3^{rw}$
A_{01} :	$F1^{rw} : F4^{rw}$
A_{02} :	$F2^{rw} : F5^{rw}$
A_{10} :	$F3^{rw} : F4^{r} : F5^{r}$
A_{11} :	$F4^{rw}$
A_{12} :	$F5^{rw}$
A_{20} :	
A_{21} :	
A_{22} :	

Figure 3.18: Data access requests queued by tile QR factorization. We show the data access requests from the first five tasks of a 3×3 tile QR factorization.

of the figure. These data accesses requests are processed according to the rules described in Section 3.2 to enable the tasks to be scheduled for execution.

In Fig. 3.19 both the QUARK and the PLASMA QR implementations demonstrate a different asymptotic peak performance when compared to the MKL implementation. This is because the MKL algorithm uses a *DGEMM* operation at its inner loop, so this highly efficient core operation comes to dominate any other costs as the size of the matrix increases. However, the tiled implementation of QR has a different inner loop operation `DTSMQR`, which has a much lower efficiency. This operation causes the asymptotic peak performance to be lower. It is expected that if the `DTSMQR` operation was to be tuned to the same degree as `DGEMM`, then the final asymptotic performance of the tiled implementations would be much higher. But, we note that The tile implementations are able to ramp up performance faster than the MKL implementation, so if the user is interested in smaller problem sizes, then the tiled algorithms would be preferred.

The QUARK and PLASMA algorithms implement the same tile QR algorithm, so their performance can be compared directly. The performance of statically scheduled PLASMA and dynamically scheduled QUARK can be seen to be approximately the same, with a very small cost for dynamic scheduling over part of the range. This overhead cost is hidden

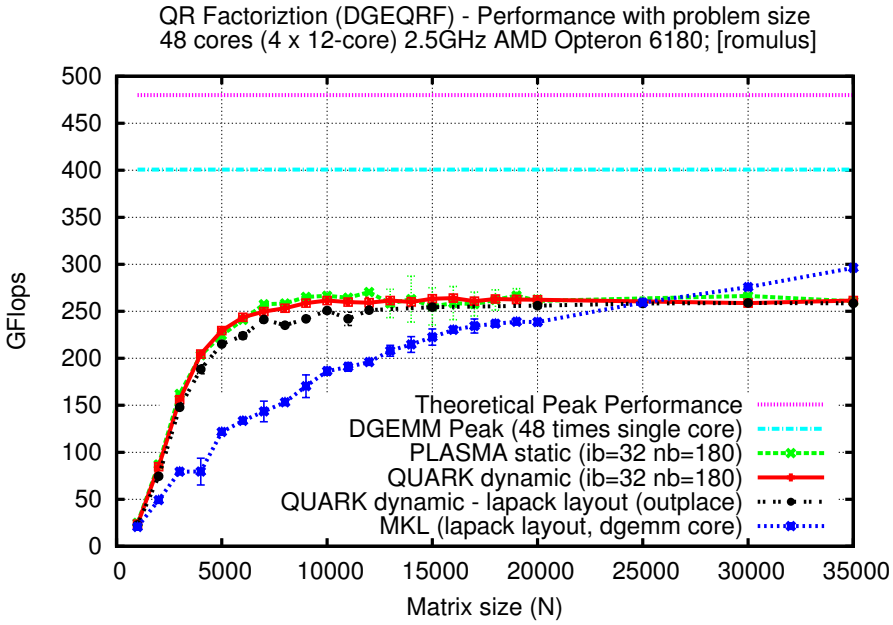


Figure 3.19: Performance for QR factorization on a 48-core shared-memory AMD system. The PLASMA algorithm performance is limited by a inner kernel operation with low efficiency. The MKL QR algorithm has DGEMM as its main operation, so its asymptotic performance is higher.

as the problem size increases towards the asymptotic peak. Once again, we can have the benefits of a dynamic environment with almost no loss in performance.

Fig. 3.16 shows a trace of a smaller QR factorization over a subset of the machine, in order to make the viewing tractable. From the trace it can be seen that the resources are being used effectively without any substantial idle spaces. The effects of using locality as the scheduling strategy can be seen by observing that the DTSQRT (red) operations tend to occur on the same core; a sequence of these operations write the same diagonal tile so if they run on the same core they can reuse the data.

3.7.3 LU Factorization

The tile LU factorization with incremental pivoting described in Section 2.4.2 is not the one implemented in shared memory PLASMA. This algorithm has several numerical problems, so the LU implementation in PLASMA is much closer to that in LAPACK.

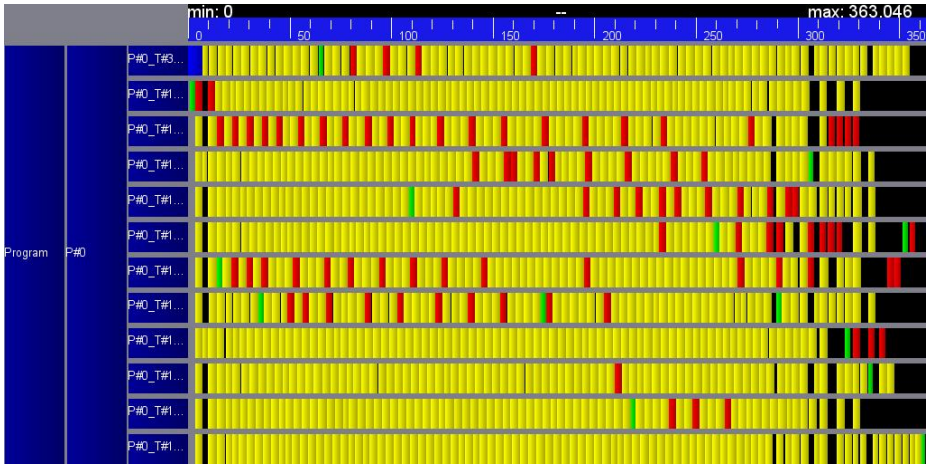


Figure 3.20: Trace for QR factorization on a 12 cores of a shared-memory AMD system. The matrix is 2500x2500 and the tile size is 180. Idle processor time is shown as black. Color key: DGEQRT: green; DTSQRT: red; DTSMQR: yellow; DORMQR: yellow.

This implementation of LU factorization demonstrates a different and unique feature of the QUARK runtime environment. Partial pivoting LU requires an operation to occur on the entire column of the matrix (panel) before the next step of the algorithm can proceed. If this operation was to be executed by a single core, it would effectively form a bottleneck in the execution. To avoid this bottleneck, QUARK provides a unique *nested parallelism* feature, the ability to assign a subset of the available computational threads to a single task. In LU factorization, some of the threads are assigned to the single task panel factorization, while the rest of the threads can be dynamically scheduled to perform more efficient tasks. This can hide the negative effect of the less efficient panel scheduling. This implementation has been discussed in detail by Kurzak et al. (2013) and is outlined in Fig. 3.21. Using this algorithm for PLASMA LU makes the innermost kernel function a DGEMM operation, which gives the algorithm the same asymptotic performance as that of MKL.

Given the complexity of managing a multi-threaded task in conjunction with the standard task scheduling, a statically scheduled PLASMA lookup-table version this LU algorithm could not implemented. Once again, we can see the advantages of having a dynamic runtime environment that automatically determines the data dependencies and data flow for complex algorithmic implementations.

The performance of the QUARK implementation is compared to the MKL LU implementation in Fig. 3.22. There is one point to note in this performance graph. As

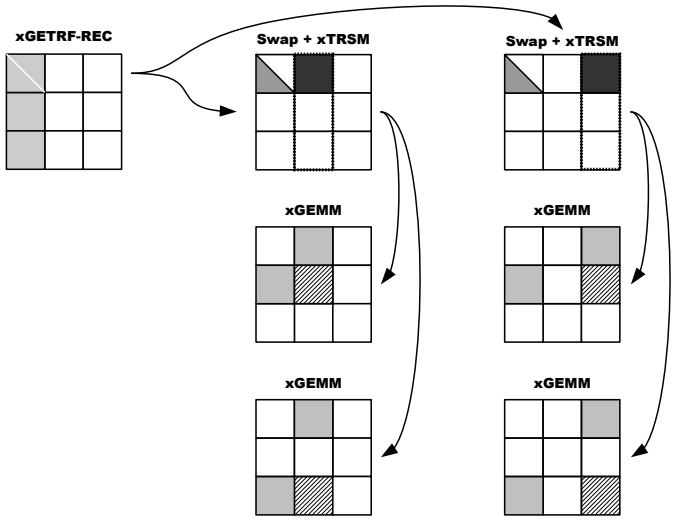


Figure 3.21: Recursive panel tile LU factorization. The factorization of the first panel uses a multi-threaded kernel (GETRF-REC) acting on the whole panel. The updates to the trailing submatrix expose a lot of parallelism and are executed as single threaded tile-based tasks.

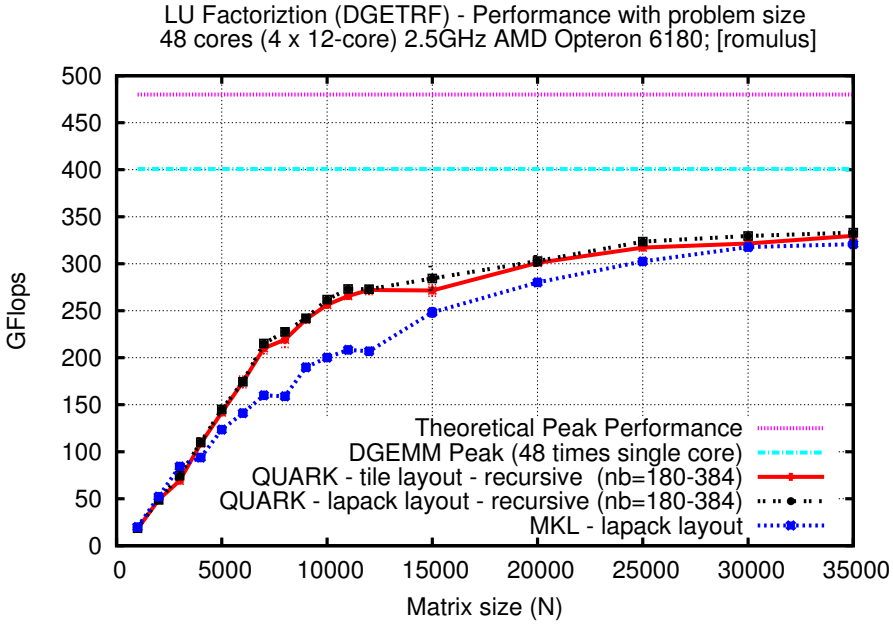


Figure 3.22: Performance for LU factorization on a 48-core shared-memory AMD system. There is no static PLASMA implementation, since the algorithm uses a QUARK feature to create and manage multi-threaded tasks .

Figure 3.23: Trace for LU factorization on a 12 cores of a shared-memory AMD system. The matrix is 3200x3200 and the tile size is 180. Idle processor time is shown as black. Color key: DGETRF: green; DTRSM: red; DGEMM: yellow

described in Section 3.4.2, tuning the tile size has a great effect on the performance of applications using this runtime environment. In this recursive tiled LU algorithm, the tile size needs to be adjusted with the problem size, and varies from 180 to 384 as the problem size increases. Due to the multi-threaded panel task this algorithm is more sensitive to the tile size than the other algorithms discussed. Future research will explore autotuning to find the best tuning parameters for a machine and problem size.

The dynamic QUARK implementation exceeds the performance of MKL throughout the range, and it even appears to reach a slightly higher asymptotic performance. Our explanation for the higher asymptotic performance is that our tiled algorithm adjusts the tile size as the problem grows, allowing slightly better efficiency for the core DGEMM operation. When we look at smaller problems sizes the performance of the QUARK algorithms is much better than MKL.

Once again, in order to claim a fair comparison against the MKL implementation, there is a dynamically scheduled QUARK version for LAPACK layout. This algorithm is somewhat unusual because it operates directly on LAPACK data distribution, rather than performing data layout conversion. This dynamically scheduled LAPACK layout implementation can be seen to have the same performance characteristics as the implementation that operates on the tiled layout.

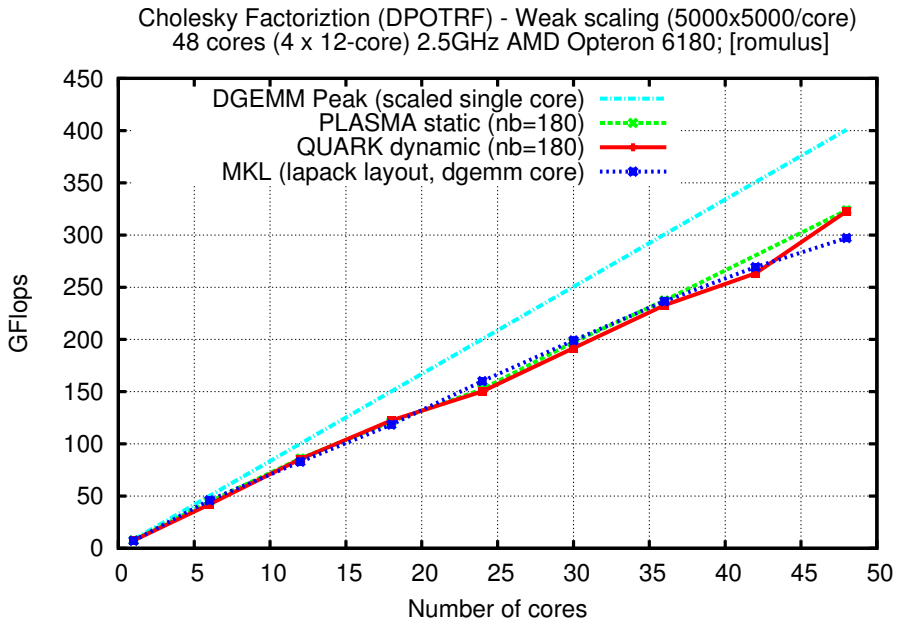


Figure 3.24: Weak scaling performance for Cholesky factorization using a problem size of 5000x5000/core on a 48-core shared-memory AMD system. The QUARK, PLASMA and MKL implementations all show a similar scaling and it is hard to tell the curves apart.

Fig. 3.23 shows a trace of the QUARK LU factorization of a small matrix. In this trace the multi-threaded DGETRF (green) task can clearly be seen, as several threads enter this task and collaborate on the panel factorization, then exit together. It can also be seen that QUARK is coordinating the data-flow execution and allowing other tasks to overlap the panel factorization.

3.7.4 Weak Scaling for Cholesky

Weak scaling gives a measure of how well an algorithm scales with the number of cores if the problem size per core is held constant. We study weak scaling in the context of Cholesky factorization because, of all the algorithms discussed, it is the one that uses the same kernels for the QUARK, PLASMA and MKL implementations. This makes it easier to compare the weak scaling performance across the implementations.

In Fig. 3.24 the weak scaling performance of the three implementations can be seen to be very similar, increasing fairly smoothly with the number of cores. These scaling curves are very close because of the chosen problem size 5000×5000 per core, which results in

a matrix size of approximately 35000 when using 48 cores. All the implementations are executing in the asymptotic region at the tested size, so the asymptotic performance of the DGEMM kernel dominates. As was shown in Fig. 3.15, the differences between all the implementations is minimal in the asymptotic performance region. The figure shows that the dynamically scheduled QUARK runtime scales as well as the other implementations with increasing core counts, meaning that we have obtained the benefits of QUARK without losing performance.

3.8 Summary

The use of a data-driven dynamic runtime environment addresses several problems in software development: productivity, scalability and performance.

A data-driven, superscalar runtime environment eliminates the complexity of writing software for multicore processors by automatically extracting the parallelism in algorithms defined in sequential code, and guaranteeing parallel correctness of sequentially expressed algorithms. For some linear algebra algorithms, manually defining the parallelism is relatively straightforward, for example, statically scheduled Cholesky factorization in PLASMA. But for many other operations, manually defining parallelism is nontrivial. Defining a static schedule that combined all the phases in a Cholesky inversion would be a daunting challenge, so having a runtime that can dynamically compose all the phases provides a powerful tool.

Using a simple API to insert tasks into the runtime enables the application developer to be much more productive. Developing code for multicore architectures can be complex, labor intensive, and error prone. This combination of a simple API and a runtime environment encourages rapid prototyping, since the details of multicore implementation and parallel algorithmic correctness are all hidden by the environment.

The QUARK runtime environment also provides a performance benefit by taking advantage of any scheduling opportunities that may be missed by a application developer. The runtime is resilient to variations in task execution time and problems that arise due to resource sharing. The behavior of the dynamic scheduler is for performance to gracefully degrade in the presence of jitter, rather than causing stalls in the parallel execution pipeline.

Experiments have demonstrated that the performance provided by the dynamic runtime is very competitive to that of statically scheduled implementations and commercial products. We gain all the benefits of a dynamic runtime without losing performance.

Chapter 4

Dynamic Task Execution In Distributed Memory

Asynchronous data driven task execution has been presented as a solution to productively developing scalable high performance applications for multicore architectures. Here we present QUARKD, a runtime environment for executing such task based applications on distributed memory machines. The mechanism for that extension is discussed and performance results will be presented. Trace data will show the mapping of tasks to execution units and larger scale runs will demonstrate the scalability of our runtime environment. QUARKD presents an easy-to-use, productive, serial, task-insertion API, greatly simplifying the effort of programming large scale machines.

4.1 Introduction

Many large scale scientific computing resources consist of a large number of distributed memory nodes, where each node contains a number of computational cores that have a shared memory. There are many programming models available to address such resources, varying from MPI libraries and thread libraries to partitioned global address space (PGAS) languages such as UPC (Unified Parallel C) or Chapel (from Cray). However, even given many years of effort, these machines remain hard to program productively while still achieving scalable, high performance.

In this work, we detail the development of our distributed memory runtime environment QUARKD (QUeuing and Runtime for Kernels in Distributed Memory). QUARKD builds on the QUARK shared-memory execution environment and extends it to distributed memory. The goal for QUARKD is to enable a programmer to develop an algorithm using the serial task insertion API of QUARK and have that algorithm run on distributed memory architectures. QUARKD facilitates productivity by taking a shared memory algorithm and enabling it to execute on a distributed memory architecture with minimal changes.

This distributed memory runtime is intended to be competitive with highly tuned algorithmic implementations (e.g. ScaLAPACK, DAGuE) in terms of performance. QUARKD is focused on productivity, scalability and performance. The QUARKD runtime can bring a substantial amount of the scaling and performance seen with highly tuned solutions with a much lower barrier to coding.

The driving applications for the development of the QUARKD runtime environment will be drawn from linear algebra. Specifically, we enable distributed memory execution for algorithms from the shared memory linear algebra library PLASMA (Agullo et al., 2010). The changes required to the PLASMA library will be detailed and experiments will show the performance achieved.

4.2 Distributed Algorithm

Design Principles In designing QUARKD a major desired feature was high productivity and in writing applications. This productivity is enabled by having a simple serial API for adding tasks into the system. This API is then used in conjunction with a smart runtime environment that determines data dependencies, performs transparent communication, and schedules tasks. The user can to provide additional information to tune the execution, but even in the absence of that information the runtime should make reasonable choices about where tasks execute and when data is moved. QUARKD should make all runtime decisions using local knowledge, without requiring any global coordination. All runtime actions should proceed asynchronously without blocking for completion.

The QUARKD runtime is designed as an extension to the shared memory runtime that we have already developed. QUARKD builds on the shared memory runtime

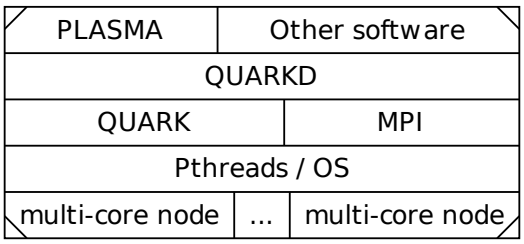


Figure 4.1: Block architecture of QUARKD, showing its place with respect to other hardware and software components.

environment provided by QUARK, and enhances it with a dependency tracking system and a communication protocol. The QUARKD project is designed and built to be independent of the PLASMA linear algebra library, though PLASMA is the main driver for its development. It is intended that QUARKD be usable in any application that inserts tasks that read and write data held in contiguous chunks of memory. The software and hardware stack for QUARKD can be represented as in Fig. 4.1.

4.2.1 Description of Distributed Algorithm

QUARKD proceeds from serial task insertion operations, where each task is inserted into the runtime environment at each process. As each task is inserted, information from the data parameters is used by each process to independently come to the same decision about which process is going to execute that task. By default, this decision about the executing process is based on which parameter is going to be written by the task, however, the decision can be overridden by the programmer. The outline of this distributed memory algorithm is shown in Fig 4.2.

Once the decision is made about which process is going to execute the task, each of its data parameters is examined. If the executing process does not have a valid copy of the data, then it inserts tasks to *receive* a valid copy. If another process is the current owner of the data, and it notes that the executing task does not have a valid copy of the data, it inserts tasks to *send* that data. If the executing process is going to write the data item, then that process becomes the current owner of the data and all other copies are marked as invalid. All processes track the current owner and validity of the copies of the data parameter.

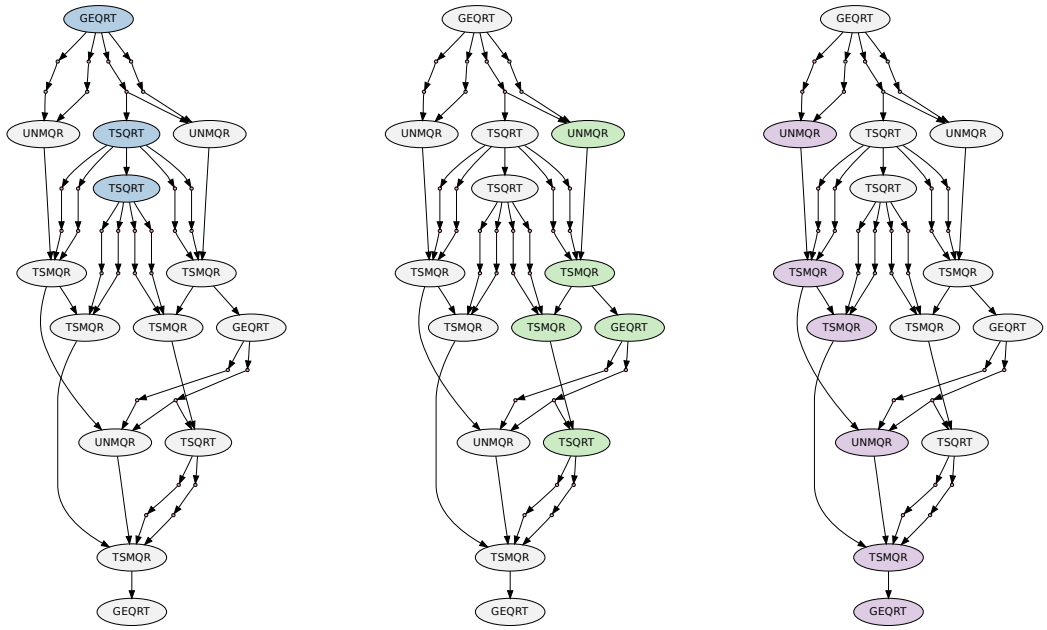
```
// running at each distributed node
for each task T as it is inserted
    // determine P_exe based on dependency to be kept local
    P_exe = process that will run task T
    for each dependency A_i in T
        if (I am P_exe ) && ( A_i is invalid here )
            insert receive tasks (A_i^rw)
        else if ( P_exe has invalid A_i ) && ( I own A_i )
            insert send tasks (A_i^r)
        // track who is current owner, who has valid copies
        update dependency tracking
    if ( I am P_exe )
        insert task T into shared memory runtime
```

Figure 4.2: Distributed memory algorithm

After any *send* or *receive* tasks are inserted, the original task is inserted into the shared memory runtime of the executing process. This sequential ordering of the tasks ensures that when the task finally executes, its data is already valid and available.

The tile QR factorization was shown in pseudocode in Fig. 3.17 and the pseudocode was expanded to a list of tasks and data requests in Fig. 3.18. The tile QR factorization is now viewed in the context of multiple processes. Fig. 4.3 shows the result of executing the QR factorization on multiple processes using the distributed memory algorithm. Below, we present a high level view of the actions taking place. Discussion of the details involved in task scheduling, data coherency, and communication engine follow later.

Each process sees the sequence of tasks as they are inserted, and independently, reaches the same scheduling decisions about where the task will run. The data used by the task is checked to see if it needs to be transferred between processors. The dependency relationships from previous usage of the data create the implicit DAG based on data hazards: read-after-write (RAW), write-after-read (WAR), write-after-write (WAW). The data transfer tasks add new data usage and dependencies. In Fig. 4.3, each processor has seen the DAG, and tasks that will execute locally on each processor are highlighted. To increase clarity in the DAGs, data movement tasks are shown just as dot connectors on the data dependency edges.



(a) P0 **(b)** P1 **(c)** P2

Figure 4.3: Execution of a small QR factorization (DGEQRF). Three processes (P0, P1, P2) are running the factorization on 3x3 tile matrix. Each process independently determines who is running the task, and local tasks are colored on each process. Data sends and receives are asynchronously managed by the communication engine.

```
QUARKD_Insert_Task(Quark *quark, void *func, Quark_Task_Flags *tflags,
  int a_flags, int size_a, void *a, int a_home_process, a_key,
  int b_flags, int size_b, void *b, int b_home_process, b_key,
  ..., 0 );
```

Figure 4.4: Distributed memory task insertion API

4.2.2 Distributed Task Insertion API

In the shared memory runtime, when a task was inserted each argument had to be provided with a size, a pointer to the data, and a set of flags that specified how that data was to be used; either as a static value, or an INPUT/INOUT/OUTPUT dependency. In the distributed memory context, each argument exists on a specific process, and the memory addresses in that process are not relevant to another process. So, the arguments need to specify which process is the home of the data and they need to specify a process-specific key for the data. This key is required since the data item does not reside at every process, we do not have an easy handle to use at every process, thus some other key value needs to be constructed to refer to the data. The key does not need to be consistent across all the processes, it just needs to be consistent within a single process space.

The API that is defined for task insertion in the distributed memory environment is shown in Fig. 4.4. Other than task insertion, the rest of the API is inherited from the shared memory QUARK runtime and can be seen in Appendix A.

4.2.3 Window of Tasks

As was discussed in Section 3.4.1, the number of tasks in a linear algebra application can grow very rapidly, creating $\theta(n^3)$ tasks with a problem size of n. This would lead to problems holding the entire DAG in memory for large problems, so we use the same solution as in the shared memory implementation. We maintain a sliding window on task insertion that keeps a limited number of active tasks. Using serial task insertion, we can be assured that the runtime environments cannot diverge and explore different parts of the DAG on different distributed memory processes, so there will be no deadlocks.

Claim 1. *Using a window of tasks does not cause a deadlock in the distributed memory algorithm.*

Proof. For a deadlock to occur, a circular wait condition must exist, where there is a set of waiting processes $P = (P_1, P_2, ..., P_n)$ such that P_1 is waiting for a resource held by P_2, and so on till P_n is waiting for a resource held by P_1.

Suppose such a circular wait condition does exist when using a window of tasks. Each process P_i has a corresponding blocked task t_i. The blocked task t_i cannot be a communication task, because communication tasks cannot be blocked; they require only one resource (data item to send or receive) and are only put into the ready list if that resource is available. So, the blocked task t_i must be a task in the computation DAG, and the only thing a task in a DAG can be blocked on is one of its parent tasks. However, there cannot be a circle of tasks waiting on each other if the tasks are part of a parent-child chain in a DAG (there are no cycles in a directed acyclic graph!).

The only way that we could have deadlock is if the tasks were from different regions of the DAG. This may be possible if the DAG were being explored independently at each process. However, since there is a sequential task insertion process, the tasks have the same order of insertion.

Consider the task that it that is highest in the DAG and is also a part of this circular wait (there may be more than one). Without loss of generality, assume it is t_1 on P_1. The only thing it can be waiting for is a parent task $parent(t_1) \in t_2, ..., t_n$. But the parent tasks cannot be blocked, since this task t_1 is the highest waiting task and all the tasks were inserted in the same order. Eventually, since the parent task is not blocked as part of this circular wait, it will be inserted in the task window of a process and it must execute and task t_1 will not be blocked.

So, no matter what the window size is, provided it is greater than one, there can be no cycle of waiting processes and no deadlock. □

4.3 Distributed Data Coherency

In a distributed data environment, there needs to be a way for each process to determine the data distribution, i.e., the location of any data item. The runtime also needs to track data movement and the validity of any existing copies of the data. All this should occur

asynchronously, without requiring any global communication or coordination. This section discusses the implementation of data management in our runtime environment.

4.3.1 Data Distribution

In the case of PLASMA applications, the data distribution used is defined by a simple function which is based on a 2D block cyclic distribution, as described by Blackford et al. (1996) for the ScaLAPACK package. This distribution has been shown by Dongarra et al. (1992) to have several advantages for block-synchronous linear algebra algorithms, including a lower communication volume, small load imbalance and good scalability. The main difference from the block-cyclic specified in ScaLAPACK is that the data tiles used by QUARKD are contiguous in memory.

As noted earlier, in a distributed memory environment we need to know which distributed memory process holds a data item and we need a key to reference it. The key is required so that we have a handle to refer to the data on processes that do not have it locally,

The block-cyclic distribution gives an easy map from any data tile to the process that is the owner of that tile. Fig. 4.5 shows how the process grid is mapped to the data tiles. The key used by PLASMA to reference data tiles is constructed out of easily known elements; the base address of a matrix and the row and column index to the tile. But, it is to be noted that any key value that uniquely refers to a tile can be used.

On top of the block cyclic distribution we implement an additional optimization for the runtime environment. There are many linear algebra algorithms that have linear dependencies down a column or across a row of a matrix. To minimize communication in this scenario, PLASMA uses a *supertiles* overlay on the distribution, which groups tiles in either the column or row direction. The effect of supertiles is to substantially reduce the communication volume for several linear algebra algorithms. Fig. 4.5 shows a processor grid and gives a visualization of how that processor grid is mapped onto data tiles to give a block cyclic distribution of the data, and then finally, a visualization how supertiling causes neighboring tiles of data to be kept on the same node.

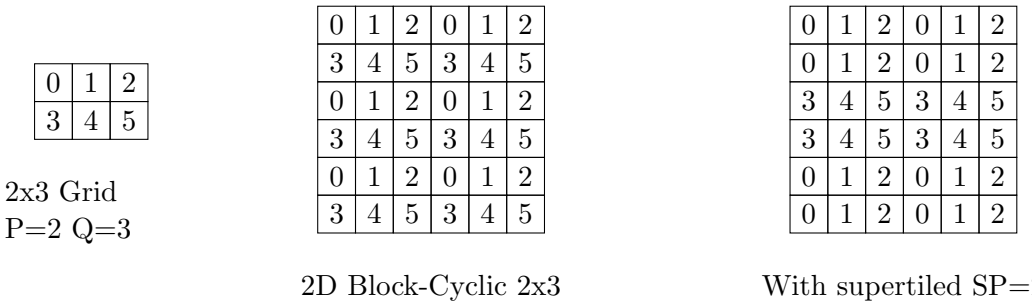


Figure 4.5: Data layout for a distributed matrix where the numbers refer to the process numbers. Six processes (nodes) are arranged as a 2x3 process grid. The 2D block-cyclic distribution overlays the 2x3 process grid on the 6x6 matrix tiles. The supertiled layout (SP=2) repeats each row of the process grid in the P direction. A supertiling can also be made in the Q direction.

Note that there is nothing in the QUARKD implementation that is aware of the data distribution. At the time that a task is inserted into QUARKD, it provides the home process and memory address of all the data parameters in the task. From that point, QUARKD uses the more abstract key to manage the data, maintaining a distributed knowledge about the ownership and state of the data. If during the execution of a task, the data is migrated to another process and overwritten there, QUARKD will manage the state and validity of all the copies of the data using an internal data coherency protocol.

4.3.2 Data Coherency Protocol

When a data parameter is first seen by a process during the serial insertion of tasks, a structure is created to hold the state of that data item. The initial process owning the data and the address on that process are known, since they are provided by the API during the insertion of the task. All non-owner processes can be assumed not to have copies of the data at this time.

If the data is transferred to another process, and the task at the receiver is going to write the data, then the ownership of the data is transferred to the receiver process and any other copies are marked as invalid. If the data is only required for read, then the ownership stays with the sender, and the receiver is marked as having a valid copy. All processes track the movement of data ownership of current data items at all times, based on the information provided when each task is inserted.

This data coherency protocol enables us to minimize the transfer of data. Since information about valid copies of the data is available at all times, no unnecessary transfers are required. In order to reduce the footprint of this data coherency protocol, the information about data that was not recently used can be flushed at well structured, regular intervals. The data coherency protocol runs at task insertion time, not at the task execution time, so we can use the serial task insertion to guide the regular intervals of data flushing. Using structured, regular intervals allows the distributed processes to do the flush and keep the distributed state of the data consistent.

4.4 Distributed Task Scheduling

Scheduling in QUARKD uses a mix of static and dynamic scheduling to assign tasks to the different processes and threads. Static selection is used to determine the process where a task will execute. However, the order in which a task executes in a multi-threaded process is dynamic, depending on many factors including cache layout, network communication speeds, task size, priority, etc.

The *static* scheduling refers to the fact that a task is scheduled for execution at a specific distributed memory process in a manner that is independent of the current state of the execution. The scheduling criteria have to be such that all the processes can independently come to the same scheduling decision. The task scheduling also considered *static* because it is fixed at the time that the task is inserted. In general, this is accomplished by having all the nodes agree that a task will be scheduled at the home location of one of its data parameters. The algorithm designer can specify the data parameter to be used in making this decision effectively saying "task T is to be run on the node which is the home of data parameter D". Fixing the execution process at the time the task is inserted enables QUARKD to avoid the complexities and coordination involved in distributed scheduling, distributed work balancing and data management.

The *dynamic* scheduling in QUARKD occurs at the multi-threaded shared-memory level and is the same as that implemented in shared memory QUARK runtime. Data locality is used to assign a task to a specific thread within the multi-threaded process. All the threads look for and execute tasks that are assigned to them. However, if there are no more tasks

assigned to a thread, it will attempt to use work-stealing to obtain a task from another thread. This dynamic scheduling keeps the execution load balanced between the threads in a process. In Chapter 3 the shared memory runtime system was shown to be capable of achieving high performance.

4.4.1 Communication Engine

In QUARKD the communications are inferred from the data usage by tasks in conjunction with the current distribution of the data. For example, if a process is currently the owner of a piece of data, and that data is to be written by a task scheduled to be executed by another process, than the two processes independently and asynchronously insert tasks that manage the sending and receiving of that data. The main distributed memory algorithm shown in Fig. 4.2 contains the base case for the data communication.

The key features of the QUARKD communication engine are that it is dynamic, non-blocking and asynchronous, meaning that the engine will manage the transfer data as needed, the data transfers will not block any of the computational threads, and the data transfers are done using asynchronous techniques. The goal is to allow communication to overlap any computation that can be performed simultaneously. The communications are all point-to-point, from the task that is the current owner of the data to any tasks that need that data.

In a standard execution of QUARKD there is an independent thread that manages the communication. This thread takes communication requests from queue of requests and handles them. The communication thread will normally share a core with a computation thread, but on some architectures, if the context switch interval is large the communication thread can be assigned a core of its own. On machines which do not permit an independent communication thread to run efficiently, communications can also be multiplexed within a computational thread.

From the point of view of the master thread doing serial task insertion, the data movement is initiated if the data consistency protocol requires it. To send a data item, a `send_init` task is inserted which accesses the data as `INPUT` and a `send_done` task is inserted which also accesses the data as `INPUT`. The data can still be used by any other

task which wishes to read the data, but it cannot be written till these tasks are completed. To receive a data item a `recv_init` task is inserted which accesses the data as `INOUT` and a `recv_done` task is inserted which also accesses the data as `INOUT`. Any following task wishing to use this data will need to wait till the receive is completed before it can be scheduled for execution.

All communications are initiated by the execution of a `send_init` task, which enqueues a send request into the message queue. The independent communication thread dequeues the send request from the message queue and sends a `send_ready` message to the destination process. The destination process then checks to see if it can handle another asynchronous receive (if not, the `send_ready` is queued) and it starts a asynchronous receive operation and sends a `ready_ack` message to the sender. The sender checks if it can handle another asynchronous send, (if not, queue `ready_ack`) and start the asynchronous send. The asynchronous operations are regularly checked for completion by the communication thread and additional operations are started when possible.

At the receiver, when a receive completes, the communication thread manages the `receive_init` and `receive_done` tasks to indicate that the data is received. These tasks reference the data in an `INOUT` (read-write) fashion, so they block the execution of any following tasks. The completion of the `receive_done` task triggers the check and possible scheduling of any task waiting to access that data.

At the sender, once the send operation completes, the communication thread manages the `send_done` task indicating that the data is sent. The `send_init` and `send_done` tasks reference the data in a `INPUT` (read) fashion, so they block any write of the data until the send is completed. The completion of the `send_done` triggers the check and possible scheduling of any task waiting to write that data.

The completion of any of the message steps triggers a check of the queues of deferred communications and any waiting communications are processed. In QUARKD, the number of simultaneous, asynchronous sends and receives is limited to a small number that gives good performance based on experimental probing. For our test platforms, communications are limited to 5 simultaneous asynchronous sends and 5 simultaneous asynchronous receives.

Claim 2. *The distributed algorithm in Fig. 4.2 in conjunction with the communication protocol above ensure that any task will eventually get its data parameters and be scheduled for execution.*

Proof. Consider any arbitrary task T which has k data parameters $A_1, ..., A_k$. The k parameters are currently owned by several processes $P_1, ..., P_k$, where P_i owns A_i, and the P_i may repeat. While examining the task sequence, since there is a serial task insertion without deadlocks, each process sees task T at some point and each process comes to the same decision about P_{exe}, the process that will execute task T. To prove that P_{exe} eventually schedules T, we need to show that (1) all the data parameters A_i eventually arrive at P_{exe} and (2) regardless of the communication order, task T will be eventually be scheduled.

For each data parameter A_i, the following cases need to be considered:
Case 1: If P_{exe} has an invalid copy of A_i, then P_i queues send tasks and P_{exe} queues receive tasks on data A_i. This ensures that T will have a valid A_i when it is queued.
Case 2: If A_i is valid on P_{exe}, do nothing. This means that T will have a valid A_i when queued.

By examination of the distributed memory algorithm in Fig. 4.2 the task T with dependencies on its data parameters $A_1, ..., A_k$ is queued on P_{exe} after all the required data *receive* tasks for the $A_1, ..., A_k$ have been queued. Since the same serial unrolling is occurring at the all the processes, all the required data *send* tasks for the $A_1, ..., A_k$ will also be queued. Eventually all the *send* tasks at the other process will execute and the data will be send to P_{exe}. By construction of the shared memory queuing mechanism and the dependencies involved, any required data *receive* tasks will have to be completed before T is scheduled for execution.

Now we need to confirm that regardless of the order of communication, task T will eventually be scheduled. Each process P_i will independently process all the tasks, so each process may reach T at different times. When they do reach T, the P_i will start any required data transfer, queuing any required *send* of A_i to process P_{exe}.

To confirm that the order of communication is correctly handled, assume that each data parameter $A_1, , .., A_k$ is received at P_{exe} at times $t_1, ..., t_k$ respectively, and that task T is inserted at time t_0 on process P_{exe}.

Case 1: $t_0 < t_1, ..., t_k$. Task t_0 has inserted receive operations on P_{exe} before any of the data arrives, so when the remote data arrives and all local dependencies are ready T will be scheduled for execution.
Case 2: $t_1, ..., t_k < t_0$. When the data arrives, the receives that are not yet posted, so the data will be cached on P_{exe}. When T is inserted, the local data cache is checked, and since all the data is here, T will be scheduled for execution.
Case 2: $t_1, ..., t_g < t_0 < t_h ..., t_k$. Some of the remote data parameters arrive and are cached before T is seen on P_{exe}. When T is seen on P_{exe} the cached data is matched. Then the rest of the data arrives at P_{exe} and is attached to T. When the last parameter arrives, it is attached to T, and since T needs no more parameters it is scheduled for execution.
In all cases, T eventually receives its data and is scheduled for execution. □

4.5 Limitations of QUARKD

In this section, a few of the limitations of QUARKD are described. Some of these limitations are specific to the design choices that were made during the implementation of QUARKD. Other limitations are simply due the process of serial task insertion, and cannot be fixed without removing the productivity benefits that are provided by serial task insertion.

Tiled Layout Constraint QUARKD requires that all the computations be based on operations on tiles of data. If at any point the programmer accesses any of the dependency data outside the declared tile parameters, then QUARKD will not be register it. There are a few algorithms in the shared memory linear algebra library where algorithm designers move outside the purely tiled approach. For example, in shared memory PLASMA, there is an LU factorization that uses nested parallelism to perform partial pivoting on a panel of the matrix. This use of nested parallelism requires that the entire panel be resident on a single process, which causes problems when implementing in a distributed memory environment.

Serial Unrolling Bottleneck The productivity gain in QUARKD is due to serial presentation of code, with the runtime system analyzing the dependencies and maintaining

serial consistency. This creates a limitation in QUARKD as the size of the distributed memory machine grows. Since fewer of the unrolled tasks are being executed at the local node, there is increasing overhead with respect to the computation work. We can compensate for some of this overhead by having larger tiles of data and thus fewer tasks and more work for each unrolled local task. On the other hand, larger tiles would mean less available parallelism. The serial unrolling is expected to be a problem for performance as the machine size keeps growing.

Consistent DAG Requirement For its dependency tracking, QUARKD requires that all the distributed memory nodes see exactly the same set of tasks. This removes flexibility in programming applications where minimally connected components are computed at different memory nodes, and the components exchange data at very localized data boundaries (e.g. ocean and climate models). Such applications could be constructed using QUARKD for the components, and having the data exchange and coordination take place with a higher level glue application. But that would require a synchronization phase, and would loose the benefits of asynchronous execution.

Effect on Optimizations Many of the optimizations from the shared memory runtime environment work until a data transfer occurs. The data transfers in QUARKD take place at the whole tile level and cause `INPUT` and `INOUT` data dependencies to be inserted for those tiles. The insertion of these data dependencies can interrupt the effect of the shared memory optimizations. Since data transfer occurs on whole tiles, any dependency that is using tile regions will be interrupted. For example, a sequence of reads on the lower triangular region of tile will be interrupted by communicating the whole tile. Task priorities will function in the distributed memory system as expected. The use of locality flags to hint the scheduling will have an appropriate effect, i.e,, tasks will be executed on the node that is the home for the data tagged with the locality flag. In summary, the distributed memory runtime does not break the shared memory implementation, but the data transfers introduce additional dependencies which can reduce the performance.

4.6 Experimental Results

Large scale experiments are performed using the tile Cholesky, QR and LU factorizations which were outlined in Section 2.4.2. Each of these three algorithms exposes different features of the runtime. The Cholesky implementation is relatively simple and has few dependencies or constraints so the runtime can demonstrate the highest performance. The QR implementation is more complex, with more constraints in the DAG. The tiled LU operation replaces the standard partial pivoting in the panel factorization with an incremental pivoting strategy (Sorensen, 1985). This pivoting strategy is numerically stable, but has substantially larger bounds than that of standard partial pivoting algorithms.

We perform weak scalability experiments, which means that as the size of the machine increases, the size of the problem is also increased. In these experiments, the quantity of work performed by a single core is kept constant, and the matrix size is adjusted to reflect this. A measure of whether the implementation will scale to large numbers of cores is to look at the efficiency of the weak scalability. Our efficiency measure is the performance achieved divided by the number of cores. For a perfectly scalable implementation, the performance per core as the number of cores increases will be a constant.

For our experiments with QUARKD, we used two distributed memory clusters.

Small Cluster The *dancer* cluster is a 16 node machine, where each node has 2 Intel Xeon E5520 2.27 GHz quad-core processors. The nodes are connected via Infiniband 20G and there is at least 8GB of memory per node. We used OpenMPI 1.5.5 compiled with gcc, and Intel MKL 11.1 math libraries.

Large Cluster The *Kraken* supercomputer at the Oak Ridge National Laboratory. Kraken is a Cray XT5 machine with 9,408 compute nodes. Each node has two Istanbul 2.6 GHz six-core AMD Opteron processors, 16 GB of memory, and the nodes are connected through the SeaStar2+ interconnect. We used the PGI compilers with Cray MPI and the Cray LibSci math libraries. For our experiments, we used a small subset of the resources on Kraken.

On each platform we also performed experiments using a high quality commercial numerical library that was appropriate for the platform. The exact implementation of the

algorithms in these libraries is not known, but they are highly tuned for the platform. On the small cluster, QUARKD is compared with the Intel MKL 11.1 ScaLAPACK implementation. On the large cluster, QUARK is compared with the Cray LibSci ScaLAPACK implementation. The ScaLAPACK implementations are executed using a process-per-core model with single threaded BLAS in each process.

Experimental results are also compared with DAGuE/DPLASMA on each platform. The DAGuE project (Bosilca et al., 2011) was described in Chapter 2, and it is implementing a distributed memory DAG execution environment that uses compact parameterized DAG descriptions. These compact DAG descriptions are difficult to generate, but the runtime does not have to unroll serial code or determine the dependencies between the tasks. The information about dependencies and dataflow is contained directly in the compact descriptions. The DAGuE runtime has been used to implement a subset of linear algebra applications in the DPLASMA package. In Bosilca et al. (2012) it has been shown that DPLASMA is a scalable high performance library for linear algebra algorithms, highly competitive with other specialized approaches and algorithm specific implementations. Given that DPLASMA provides a state-of-the-art distributed implementation of linear algebra algorithms, it is used to measure the success of the QUARKD implementations. Since DAGuE avoids a substantial part of the overheads in QUARKD while still following the structure of asynchronous DAG execution, it is expected to give a higher performance than QUARKD. The advantage that QUARKD holds over DAGuE is in the productivity of writing serial code over the difficulty of generating compact DAG representations. It should be noted that the DAGuE developers are working on a compiler approach to simplify the generation of compact parameterized DAG descriptions.

4.6.1 QR factorization

Fig. 4.6 shows a slightly refined version of the QR algorithm as implemented by the PLASMA library. This code closely matches the pseudocode, and is essentially the same as the shared memory code implemented in PLASMA. The only difference is shown in the macro at the top, where a tile reference is expanded to contain the address of the tile, the process that contains the tile, and a key for referring to that tile. The fact that we were able

to keep the code changes so minimal implies that QUARKD is achieving the productivity that was sought as of the major goals of the project.

The `DGEQRT` task is inserted into the runtime in a small wrapper routine shown in Fig. 4.7. This wrapper provides the additional information required by the runtime system. Specifically, the information provided includes the usage (INPUT, OUTPUT, INOUT, VALUE) of the parameters, the sizes of the parameters, and additional hinting information provided by the programmer. Note that for each dependency parameter, a *home* node and a local *key* are provided. This required information was discussed in section 4.2.2. The task information is then stored in the QUARKD runtime, where the execution of the task is held until all the data dependencies are satisfied. At that point, the task is ready to be scheduled for execution.

Fig. 4.8 shows the function that is called by QUARKD when the task is eventually executed. In this function, the parameters are extracted from the QUARKD runtime, the arguments and dependencies are unpacked, and the serial core routine is called.

The distributed memory DAG for a QR factorization on a small matrix is shown in Fig. 4.9. Tasks that execute on the same process are marked with the same color, and communication is shown via red arrows. The generation of these DAGs can be done automatically by the QUARKD runtime, and these can be a useful tool in understanding and debugging the code.

A trace of the execution of the QR factorization using 4 distributed memory nodes with 4 computational threads each is shown in Fig. 4.10. This trace shows the tasks keeping the cores busy with computation, with occasional gaps in the trace where the DAG does not have enough parallelism to keeps the cores busy. The gaps in this trace are mostly because a small problem of 16×16 tiles is being traced, but such gaps could occur anytime that the DAG does not provide sufficient parallelism and lookahead to hide the bottleneck tasks. The trace for a larger problem is not shown because it would become too congested. A separate communication thread is shown associated with each process. This thread is sharing one of cores with a computational thread. The trace is produced by the same tracing routines and mechanisms implemented in shared memory PLASMA, underlining the productivity gained by extending a shared memory environment to distributed memory.

```
#define A(m,n) BLKADDR_D(A, double,m,n), MPIHome(m,n), (Key){A.mat,m,n}
#define T(m,n) BLKADDR_D(T, double,m,n), MPIHome(m,n), (Key){T.mat,m,n}

void plasma_pdgeqrf_quark( PLASMA_desc A, PLASMA_desc T, ... )
{
  for (k = 0; k < M; k++) {
    QUARKD_CORE_dgeqrt( quark, ..., A.m, A.n, A(k,k), T(k,k));
    for (n = k+1; n < N; n++)
      QUARKD_CORE_dormqr( quark, .., A(k,k), T(k,k), A(k,n));
    for (m = k+1; m < M; m++) {
      QUARKD_CORE_dtsqrt( quark, ..., A(k,k), A(m,k), T(m,k));
      for (n = k+1; n < N; n++)
        QUARKD_CORE_dtsmqr( quark, ..., A(k,n), A(m,n), A(m,k), T(m,k));
    }
  }
}
```

Figure 4.6: The distributed memory tile QR factorization as implemented in PLASMA and executed using QUARKD. This code very closely matches the pseudo-code. Information about the read/write usage of parameters is provided in a wrapper.

```
void QUARKD_CORE_dgeqrt(Quark *quark, ..., int m, int n,
    double *A, int A_home, key *A_key,
    double *T, int T_home, key *T_key )
{
    QUARKD_Insert_Task( quark, CORE_dgeqrt_quark, ...,
        VALUE, sizeof(int), &m,
        VALUE, sizeof(int), &n,
        INOUT | LOCALITY, sizeof(A), A, A_home, A_key,
        OUTPUT, sizeof(T), T, T_home, T_key, ..., 0);
}
```

Figure 4.7: The DGEQRT task wrapper used in the QR factorization algorithm. This wrapper provides information about the read/write usage of the parameters and inserts the task into the QUARKD runtime.

```
void CORE_dgeqrt_quark(Quark *quark)
{
    int m, n, ib, lda, ldt;
    double *A, *T, *TAU, *WORK;
    quark_unpack_args_9(quark, m, n, ib, A, lda, T, ldt, TAU, WORK);
    CORE_dgeqrt(m, n, ib, A, lda, T, ldt, TAU, WORK);
}
```

Figure 4.8: The DGEQRT task implementation is called by the QUARKD runtime when all the dependency requirements have been met. Parameters are unpacked from the QUARKD environment, and the (final) core routine provided by a library is called.

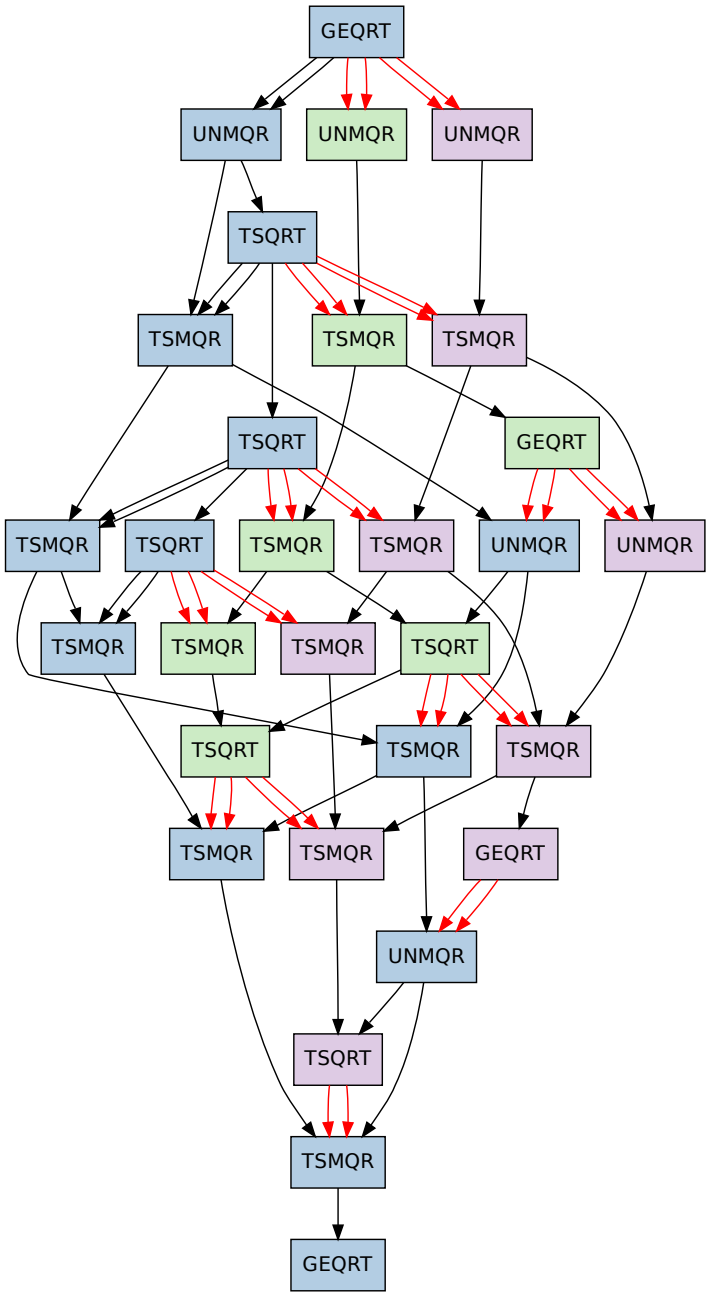


Figure 4.9: DAG generated by QUARKD for a tile QR factorization of a matrix consisting of 4x4 tiles on 3 distributed memory nodes. The colors correspond to different nodes, and the red arrows correspond to communications.

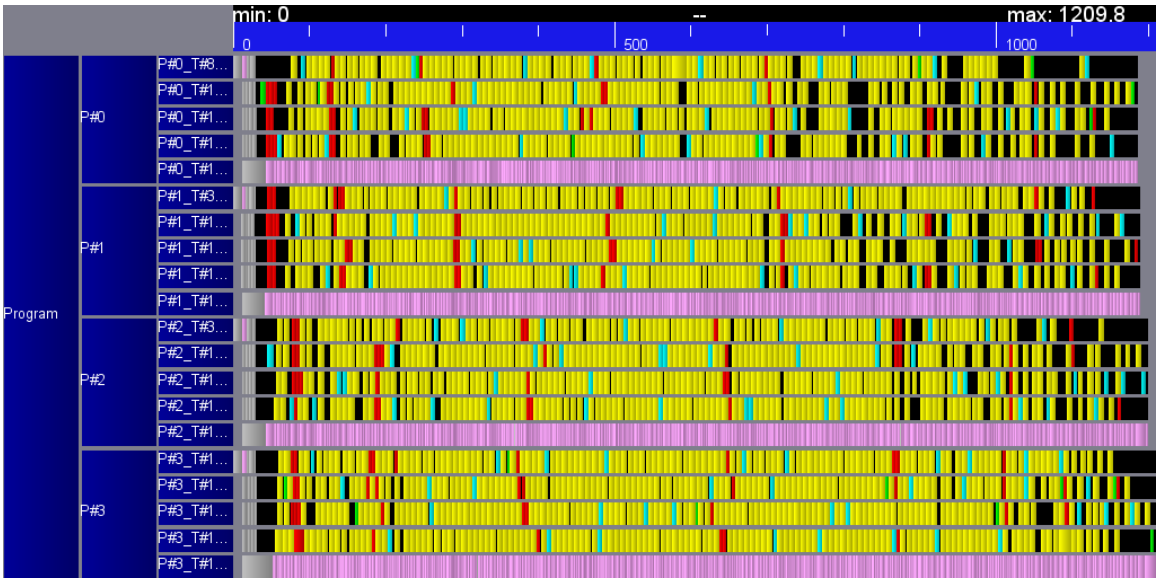


Figure 4.10: Trace of a QR factorization of a matrix consisting of 16x16 tiles on 4 (2x2) distributed memory nodes using 4 computational threads per node. An independent MPI communication thread is also maintained. Color coding: MPI (pink); GEQRT (green); TSMQR (yellow); TSQRT (cyan); UNMQR (red).

Experimental Results for QR Factorization

In order to evaluate the effectiveness of the distributed runtime environment, the performance of the QR factorization is compared with the DAGuE runtime and commercial libraries. Experimental results on the small cluster are given in Fig. 4.11 and show that QUARK is faster than MKL on this platform, and while performance is lower than DAGuE, it is still very competitive. The figure also shows the maximum theoretical upper bound performance achievable on this machine based on the clock frequency.

Fig. 4.12 shows the weak scaling experiment on the large cluster using 1200 cores. In this experiment, we see that QUARKD trails the Cray LibSCI implementation. DAGuE's performance on QR factorization exceeds the LibSCI performance, which validates the use of asynchronous data drive DAG execution. DAGuE and QUARKD implement very similar algorithms, but QUARKD has additional overheads that DAGuE does not have. The theoretical peak performance and the scaled single-core `DGEMM` performance are shown to give a measure of how much of the peak is being achieved.

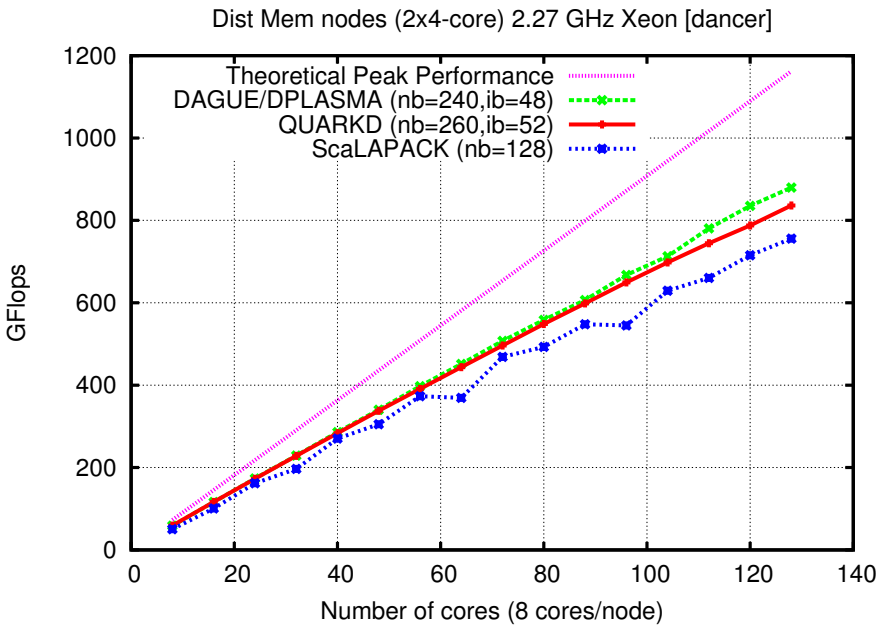


Figure 4.11: Weak scaling performance of QR factorization on a small cluster. Factorizing a matrix (5000x5000/per core) on up to 16 distributed memory nodes with 8 cores per node. Comparing QUARKD, DAGuE and ScaLAPACK (MKL).

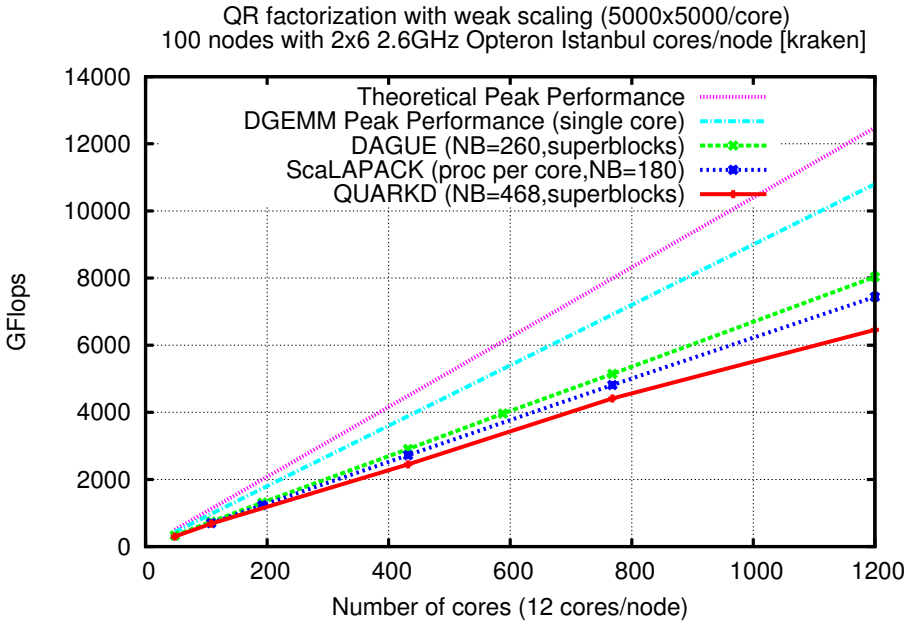


Figure 4.12: Weak scaling performance for QR factorization (DGEQRF) of a matrix (5000x5000/per core) on 1200 cores (100 distributed memory nodes with 12 cores per node). Comparing QUARKD, DAGuE and ScaLAPACK (libSCI).

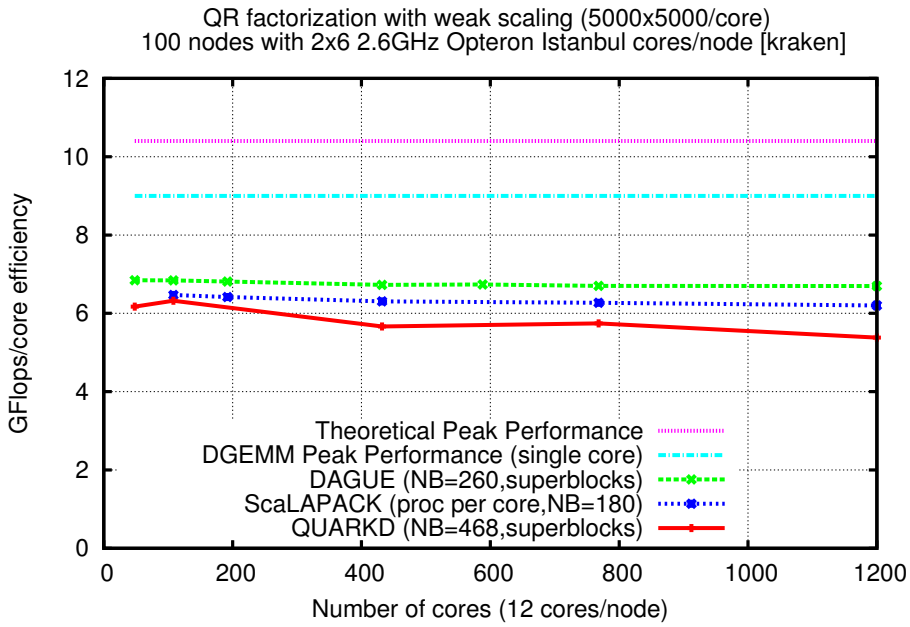


Figure 4.13: Efficiency of weak scaling for QR factorization (DGEQRF) of a matrix (5000x5000/per core) on 1200 cores (100 distributed memory nodes with 12 cores per node). Results are using QUARKD, DAGuE and ScaLAPACK (libSCI).

An important aspect of weak scaling is the efficiency achieved as the number of cores increases. For an implementation to keep scaling to larger number of cores, the efficiency should be as close as possible to a flat line. In Fig. 4.13 the efficiency of various QR implementations is shown. The DAGuE and LibSCI implementations achieve very flat lines, however the QUARKD implementation shows a slight decrease in the scaling efficiency as the number of cores increases. This can be attributed to the serial unrolling process, since each of the nodes unrolls the entire DAG even if the node ignore most of the tasks that it sees. As the problem size grows this large scale serial task unrolling acts as a bottleneck to performance. Here we see some of the cost that QUARKD pays for having a simpler, more productive, serial task insertion API. This problem can be alleviated by increasing the size of work performed relative to the amount of serial unrolling.

It is important to remember that the tile QR implementations implemented by DAGuE and QUARKD have `DTSMQR` as the dominant operation in the innermost loop, whereas the commercial implementations have the much higher optimized `DGEMM` as the dominant operation. In spite of that handicap, the QUARKD and DAGuE implementations performed

very well. It is likely that if the `DTSMQR` kernel is optimized to the same point as the `DGEMM` kernel, then the tile algorithms will show increased performance.

4.6.2 Cholesky Factorization

The Cholesky factorization forms a relatively simple DAG structure with a single output dependency from each task as shown in Fig. 4.14. This means that data ownership does not need to be transferred to other processes, since data writes can always be done at the home process of the data.

A small scale trace of a Cholesky factorization is shown in Fig. 4.15. Gaps in the trace can be seen that are centered around POTRF (green) operation. The POTRF operations are on the critical path for the Cholesky factorization, and for the small factorization shown here, there is not enough work to keep the cores busy in order to overlap the POTRF operations.

Experimental Results for the Cholesky Factorization

In Fig. 4.16 experimental results on the small cluster compare Cholesky factorization performance using QUARKD, Intel MKL, and DAGuE. The QUARKD implementation scales better than the MKL implementation but not as well as the DAGuE.

The weak scaling performance on the large 1200 core cluster is shown in Fig. 4.17. On this large cluster the QUARKD implementation has better performance than the Cray LibSCI library. The DAGuE implementation reaches a higher performance than the other implementation, but as we discussed earlier, DAGuE is competitive with the best algorithm specific implementations. The main problem with DAGuE is productivity since writing compact DAG representations remains a difficult process. QUARKD focuses on the productivity gained by writing serial style, loop based code. In spite of this ease of coding, QUARK is able to produce better performance than the commercial implementations.

Fig. 4.18 shows the efficiency of Cholesky implementations; the performance normalized by the number of cores. In order to achieve long term scalability this efficiency curve should be as flat as possible. The DAGuE implementation shows an excellent efficiency, suggesting that it will scale very well. Both the ScaLAPACK and the QUARKD implementations show

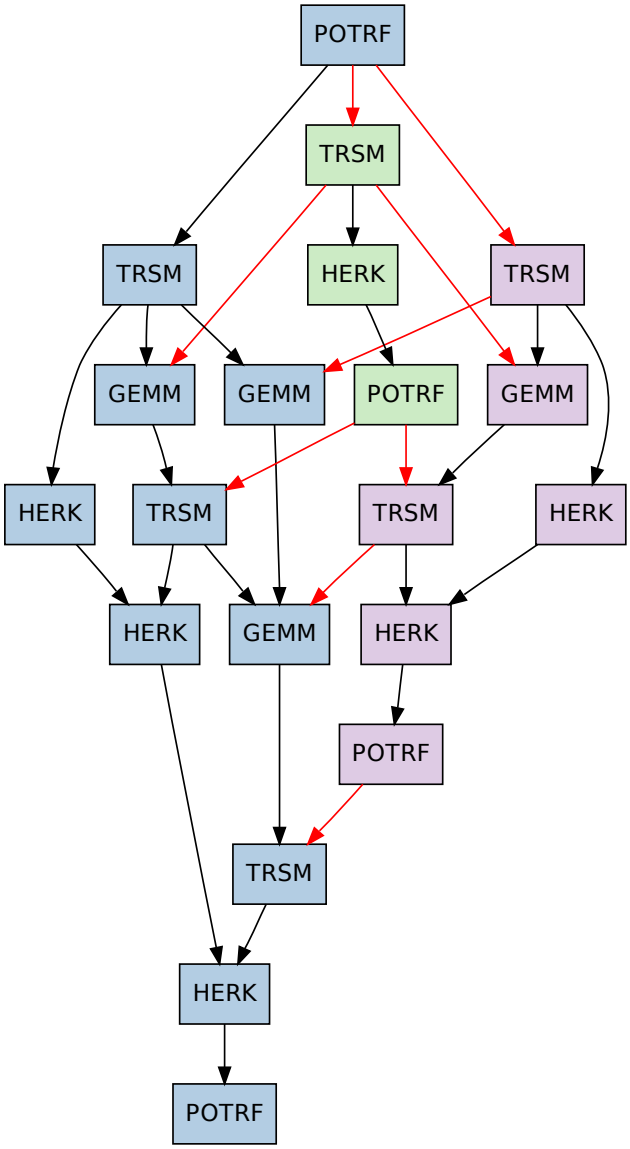


Figure 4.14: DAG generated by QUARKD for a Cholesky factorization operation of a matrix consisting of 4x4 tiles on 3 distributed memory nodes. The colors correspond to different nodes, and the red arrows correspond to communications.

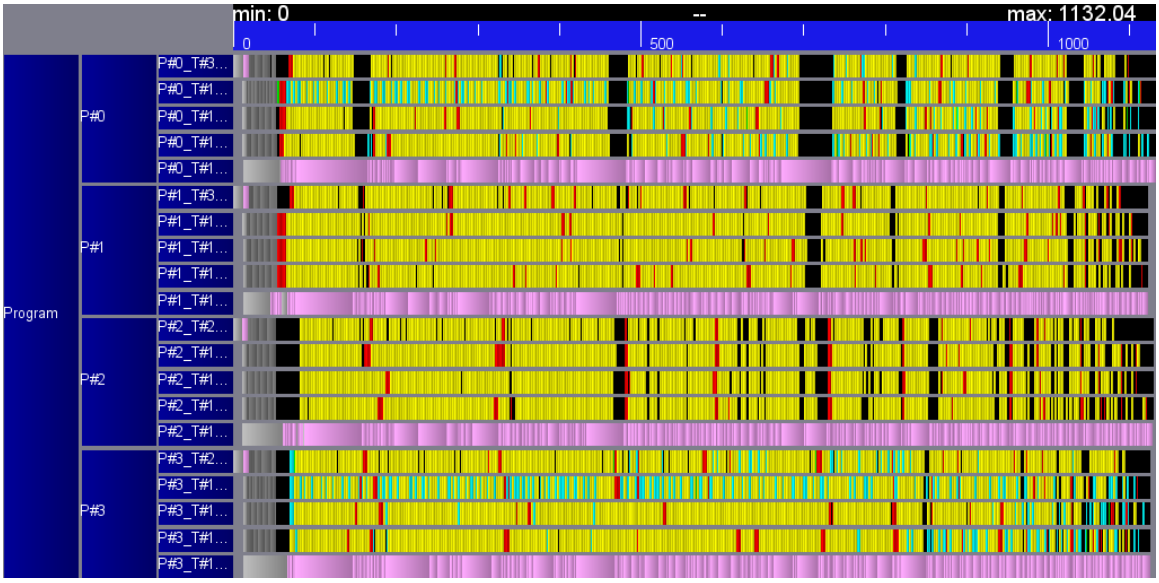


Figure 4.15: Trace of a Cholesky factorization of a matrix consisting of 30x30 tiles on 4 (2x2) distributed memory nodes using 4 computational threads per node. An independent MPI communication thread is also maintained. Color coding: MPI (pink); POTRF (green); GEMM (yellow); SYRK (cyan); TRSM (red).

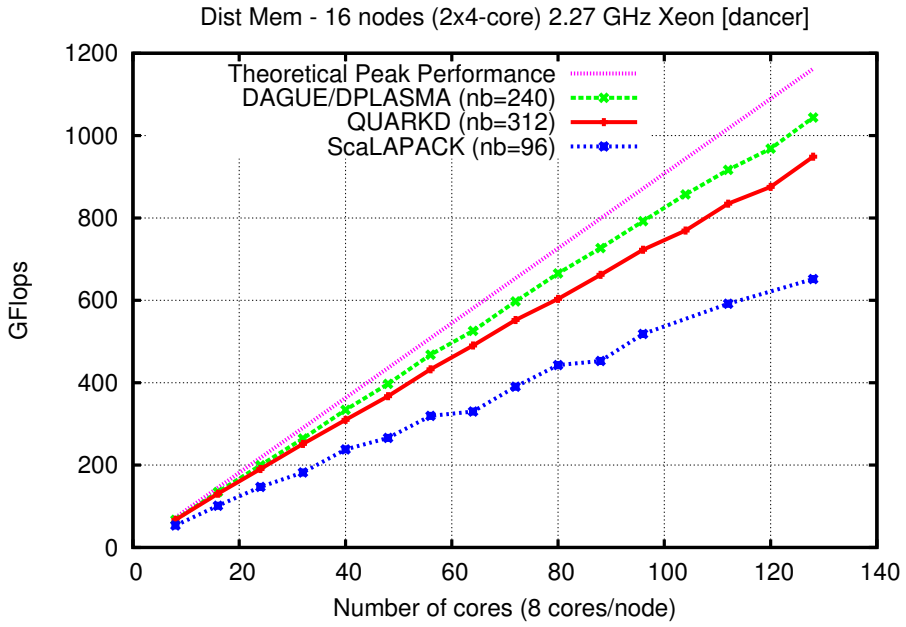


Figure 4.16: Weak scaling performance of Cholesky factorization (DPOTRF) of a matrix (5000x5000/per core) on 16 distributed memory nodes with 8 cores per node. Comparing QUARKD, DAGuE and ScaLAPACK (MKL).

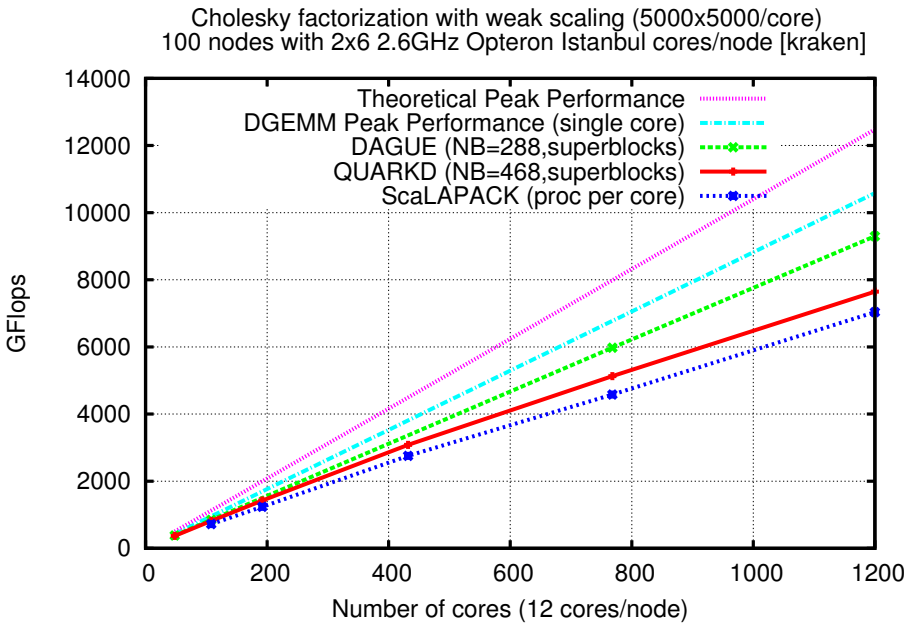


Figure 4.17: Weak scaling performance for Cholesky factorization (DPOTRF) of a matrix (7000x7000/per core) on 1200 cores (100 distributed memory nodes with 12 cores per node). Comparing QUARKD, DAGuE and ScaLAPACK (libSCI).

decreasing efficiency as the number of cores increases. But, in the experimental range, the QUARKD implementation scales approximately as well as the LibSCI implementation.

4.6.3 LU Factorization

The LU factorization is a special case because the commercial implementations use the standard partial pivoting scheme, whereas the tile LU factorization in QUARKD and DAGuE uses a pairwise incremental pivoting strategy as discussed in Section 2.4.2. Even though this is a stable strategy, there can be substantial increase in the error bounds (Sorensen, 1985). However, the partial pivoting scheme does not map simply to a tiled implementation since determining the pivot element and applying it is a single action that occurs over a panel of the matrix, and a panel of a matrix need not be resident on one node. For these experiments, the two different LU algorithm are compared, but it is important to keep in mind that the two algorithms are not equivalent.

In Fig. 4.19, LU factorization is performed on 1200 cores of the large cluster. The QUARKD and LibSCI implementations perform very similarly, with a slight decrease in

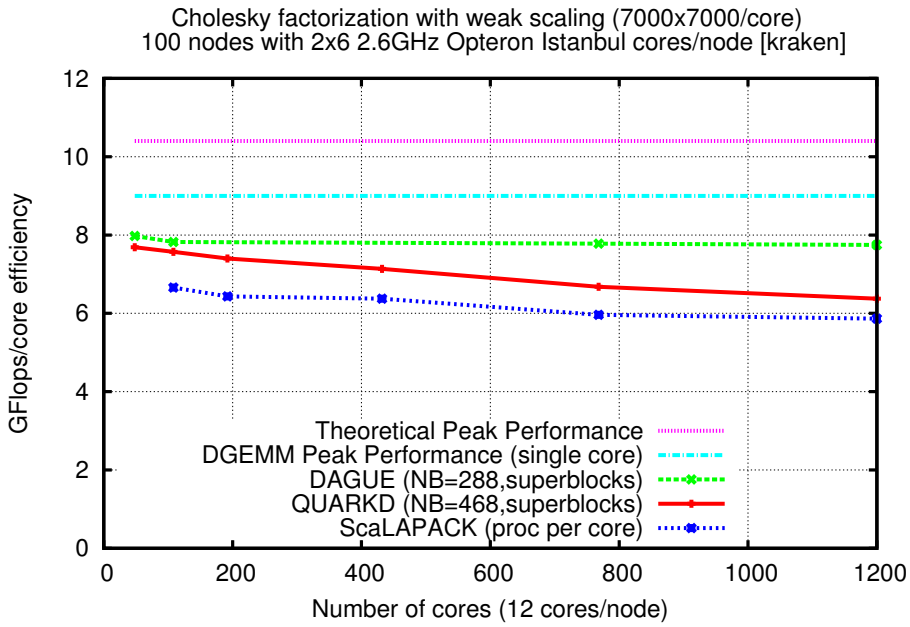


Figure 4.18: Efficiency of weak scaling for Cholesky factorization (DPOTRF) of a matrix (7000x7000/per core) on 1200 cores (100 distributed memory nodes with 12 cores per node). Results are using DAGuE and ScaLAPACK (libSCI).

performance for QUARKD as the size grows. The DAGuE implementation has the highest performance, substantially exceeding the other two implementations. Once again, this can be attributed to the fact that DAGuE implements an asynchronous, data-driven, task execution engine that avoids the overheads unrolling the DAG and determining dependencies, at the cost of increased complexity and reduced productivity in creating compact parameterized DAGs.

Examining the efficiency of the weak scaling in Fig. 4.20, it is evident that the DAGuE implementation is scaling very well as the number of cores grows. LibSCI and QUARKD lose efficiency as the size of number of cores grows. However, QUARKD is still competitive with LibSCI for LU factorization.

4.7 Summary

We have designed a runtime system to schedule and execute task based applications on distributed memory systems. The runtime accepts tasks via a serialized task insertion process, and determines the task and data dependencies dynamically based on the data

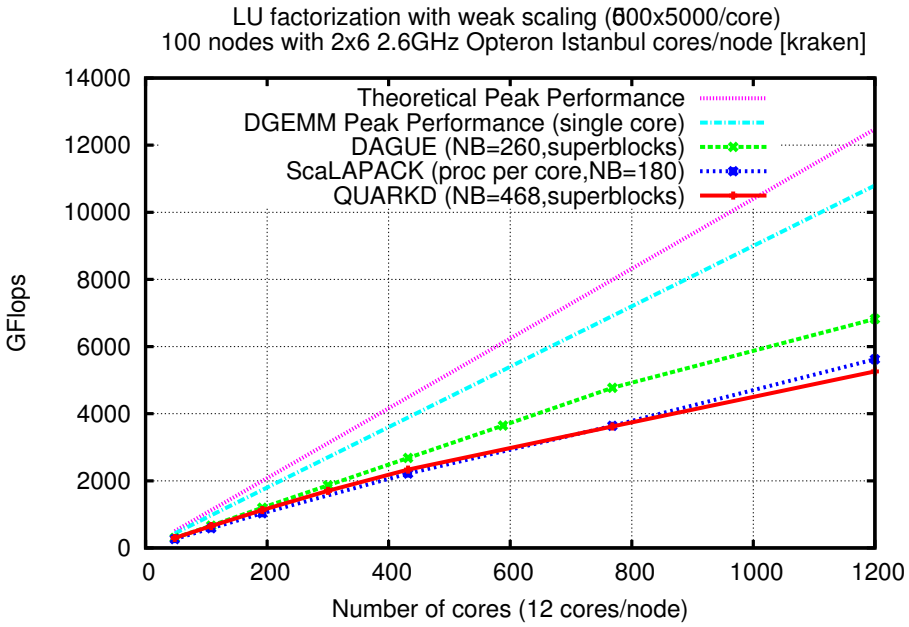


Figure 4.19: Weak scaling performance for LU factorization of a matrix (5000x5000/per core) on up to 1200 cores of the large cluster. DAGuE and QUARKD use a tile LU algorithm with incremental pivoting that is numerically inferior.

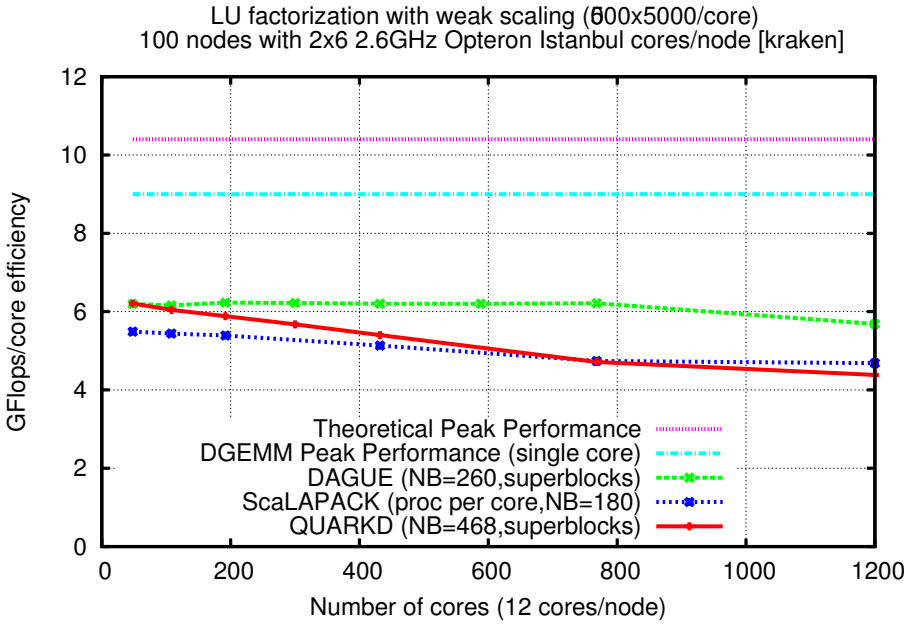


Figure 4.20: Efficiency of weak scaling for LU factorization (DGETRF) of a matrix (5000x5000/per core) on up to 1200 cores. DAGuE and QUARKD use a tile LU algorithm with incremental pivoting that is numerically inferior.

usage patterns. The task scheduling and data movement are managed without any global coordination. A distributed data coherency protocol ensures that the copies of the data are managed in order to decrease data movement. A dynamic, non-blocking communication engine handles asynchronous data movement. In order to manage large problems, a fixed size moving window of active tasks is used from the serialized task insertion.

QUARKD is designed for productivity, scalability and performance. To demonstrate productivity, the PLASMA linear algebra library and algorithms are executed using QUARKD. This required minimal additional information added to the original shared memory API used by PLASMA. Since task-based shared memory algorithms can be transferred easily to a distributed memory environment, QUARKD is proving itself as a productive tool.

The scalability and performance of QUARKD is compared to that of commercial linear algebra libraries and to the DAGuE runtime environment. DAGuE implements a dynamic, data driven, task based runtime which achieves performance and scalability the could not be matched by either the commercial libraries or by QUARKD. However, DAGuE algorithms require a compact parameterized DAG representation and are are complex to write, whereas QUARKD has a productive easy-to-use interface. Experiments performed on a 128 cores of a small cluster and a 1200 cores of a large cluster show QUARKD can be scalable and have performance competitive with the commercial solutions.

Chapter 5

Conclusions and Future Work

The increasing presence of multicore and many-core computer architectures in all computer designs, from shared memory machines to large distributed memory clusters, is presenting new challenges to software designers. Software that used to automatically benefit from increases in clock speeds in the older designs, now has to be completely rewritten to take advantage of multicore processors. On top of that, since the CPU clock speeds may actually decrease when multiple cores are introduced, software performance may even decrease if it is not rewritten. Even if the software is expressed with parallel components, the cost of any serialization or synchronization in the software becomes proportionally greater as the number of available cores grows.

These costs will affect any software, but computational science problems tend to need all the computational power available from the hardware. Computational science is working with some of the most difficult and important problems in the world, including climate prediction, ocean modeling, nuclear simulation, genome analysis, etc. The numerical techniques for addressing these problems require large amounts of computational resources, and make use of high performance libraries that can extract performance from these resources.

In this dissertation we develop productive APIs and runtime environments that can be used to create and execute highly parallel asynchronous software on multicore architectures. This software can be used to enable libraries to extract more performance from the underlying hardware.

Linear algebra libraries are a vital component in the solution of many computational science problems. The LAPACK and ScaLAPACK libraries are standard, well-known, highly regarded libraries that define and implement the standard interfaces for doing dense linear algebra on shared memory and distributed memory machines. However, these libraries were written before multicore architectures became prevalent in high performance computing. So, the standard, open-source, reference implementations for these libraries suffer because they intersperse kernels with high degrees parallelism with kernels with low degrees of parallelism. In the shared memory LAPACK library, this form of execution can be referred to as fork-join execution. In the distributed memory ScaLAPACK library, the algorithms are written in block synchronous parallel style (BSP), which synchronizes at communication steps. Both of these situations can be thought of as having undesired synchronizations.

The commercial implementations of LAPACK and ScaLAPACK may include mechanisms that alleviate some of the synchronization effects. However, we would like to address the synchronization problems in an algorithmic way within the freely available linear algebra libraries. These new linear algebra algorithms and libraries may then be further optimized by commercial institutions in order to increase performance.

To this end, the PLASMA project is designing linear algebra algorithms to be executed as a asynchronous data-driven execution of kernel tasks. These algorithms can be viewed as a DAG of kernel tasks connected by data dependencies. The process of writing software that can do asynchronous data-driven execution of complex DAGs is complex and labor intensive, so a simple programming methodology would greatly improve productivity.

In this dissertation, we develop a productive interface for writing task-based programs. We develop shared memory and distributed memory runtime environments that can execute these task-based programs in asynchronous, highly parallel fashion. We show the performance and scalability of these runtime data dependencies.

5.1 Conclusions

The solutions developed in this dissertation are evaluated in the context of productivity, scalability and performance.

In Chapter 3 of this dissertation we developed QUARK, a runtime environment that accepts serial task insertion, determines dependencies, and will schedule the execution of kernel tasks when their dependencies are satisfied. QUARK is designed for execution in shared memory architectures and is specialized for linear algebra applications. QUARK enables the PLASMA linear algebra library to program algorithms with a simple task-insertion API and still achieve high performance and scalability on multicore machines.

Programming *productivity* is a challenge when targeting complex multicore and distributed memory architectures. There are many details like scheduling, data locality, hazard avoidance, and asynchronous execution that all add up to make it difficult to define simple programming interfaces for designing software. We have described scenarios where efficiently programming a complex algorithm such as Cholesky inversion is prohibitively difficult when using static scheduling techniques. We have implemented an interface that makes task-based programming relatively easy. Using this interface, we can write code that maps very closely to pseudocode produced by algorithm designers.

Experimental results show that dynamic scheduling using QUARK achieves the same *performance* as the optimized statically scheduled PLASMA code. When DAGs are run in sequence, then QUARK can compose them to generate shorter wider DAGs and get greater performance than running the DAGs sequentially. QUARK is shown to match the asymptotic performance of the highly tuned commercial MKL linear algebra implementations, provided the algorithms are using the same computational kernels. Additionally, the tile-based asynchronous algorithms are able to reach their asymptotic performance much faster than libraries based on the fork-join computational model.

In weak scaling experiments, the dynamic runtime environment of QUARK is shown to be as *scalable* as the static scheduling technique or the commercial MKL library. This demonstrates that the overhead of the dynamic scheduling is sufficiently low that it can be hidden by the computational.

In Chapter 4 the ideas and design of our asynchronous execution environment are extended to distributed memory architectures. The QUARKD runtime environment retains the productive API for serial task insertion, and transparently manages the details of distributed task scheduling, data coherency, and communication. QUARKD demonstrated its *productivity* by enabling the algorithms from the PLASMA library, which was designed

for shared memory execution, to easily and transparently execute on distributed memory machines. The sample code shown demonstrates using the simple interface for inserting tasks into the runtime.

In our distributed memory experiments, the DAGuE runtime is used as a performance measure for QUARKD. DAGuE is an asynchronous tile based execution environment that avoids the overheads in the serial task insertion by using compact parameterized DAG representations of algorithms. This means that DAGuE does not have to do many of the activities associated with serial task insertion, such as examining every task at every node, determining dependencies, and managing tasks that are currently inactive. On the other hand, programming algorithms in DAGuE has a much higher complexity because it requires writing compact parameterized DAG descriptions.

Since DAGuE avoids the overheads implied by the tile-insertion interface in QUARKD, it trades productivity for performance. Both the QUARKD and DAGuE runtime environments perform asynchronous, data driven, tile-based execution, but DAGuE has substantially lower overheads, so it achieves better performance and scalability.

But, experimental results show that algorithms implemented using QUARKD can achieve *performance* comparable to the commercial Cray LibSCI and Intel MKL libraries, and are *scalable* to up to 1200 distributed memory cores.

It is expected that the QUARKD runtime will be used to extend additional tile-based shared memory algorithms to distributed memory in a natural and transparent manner. The only requirement is that the original algorithm interacts with its data purely through the tile based interface. If that requirement is met, other algorithms should immediately be able to take advantage of high performance and scalable distributed memory execution.

We have found that serial task insertion in combination with asynchronous, data-driven execution can be unexpectedly effective in distributed memory machines.

5.2 Future Work

This dissertation has focused on construction of productive, efficient runtime environments for the data-driven execution of task based algorithms. Starting from these runtime environments, many areas of further research work are available.

The scheduling methodology used for assigning tasks to processing elements is very simple, there is a lot of work that could be done to improve this scheduling. A starting point could consider the tradeoffs in locality assignment versus work stealing and alter the work stealing strategies based on locality details. More complex scheduling research could use information from the window of upcoming tasks to prioritize the tasks with the most children, thus attempting to mimic a critical path analysis within a task window.

Further scheduling improvements can be made by maintaining performance histories for each kernel on each computational resource. These would provide performance prediction capabilities to the runtime, so scheduling strategies such as list scheduling can be used to minimize the overall schedule length and execution time.

Research into autotuning would greatly benefit this project. There are many occasions where the tile size needs to be adjusted to achieve the best performance depending on the algorithm, machine architecture, problem size, number of cores, etc. An autotuning project could determine a functional or experimental mapping from all these variables to recommend an appropriate tile size.

The dynamic runtime can be used to investigate the energy footprint of various algorithms and this can be used to select between algorithms if power consumption is a concern. For a given algorithm, the runtime could dynamically adjust the power consumption of the cores via dynamic voltage scaling if it detects that the width of the DAG is small and it cannot use all the available cores. Dongarra et al. (2012) have begun investigating the energy footprints of tile linear algebra algorithms using PLASMA and QUARK.

The runtime could be extended to support transparent heterogeneous computing in order to take advantage of the compute power provided by GPUs. Kurzak et al. (2012) have used QUARK to manage multiple CPUs with a GPU, but the GPU data movement and execution were not transparently managed. In order to support GPUS and other heterogeneous resources the runtime could be extended to abstract tasks and data movement.

Bibliography

Bibliography

Agullo, E., Bouwmeester, H., Dongarra, J., Kurzak, J., Langou, J., and Rosenberg, L. (2011). Towards an efficient tile matrix inversion of symmetric positive definite matrices on multicore architectures. In *Proceedings of the 9th international conference on High performance computing for computational science*, VECPAR'10, pages 129–138, Berlin, Heidelberg. Springer-Verlag. 23, 34, 106

Agullo, E., Dongarra, J., Hadri, B., Kurzak, J., Langou, J., Langou, J., Ltaief, H., Luszczek, P., and YarKhan, A. (2010). PLASMA Users Guide. Technical report, University of Tennessee, Innovative Computing Laboratory. 3, 15, 22, 52, 95

Anderson, E., Bai, Z., Bischof, C., Blackford, L. S., Demmel, J., Dongarra, J. J., Du Croz, J., Hammarling, S., Greenbaum, A., McKenney, A., and Sorensen, D. (1999). *LAPACK Users' guide (third ed.)*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA. 12, 21

Augonnet, C. (2011). *Scheduling Tasks over Multicore machines enhanced with Accelerators: a Runtime System's Perspective*. Phd thesis, Universit'e Bordeaux 1. 4

Augonnet, C., Thibault, S., Namyst, R., and Wacrenier, P.-A. (2009). 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. Springer-Verlag. 11

Ayguade, E., Copty, N., Duran, A., Hoeflinger, J., Lin, Y., Massaioli, F., Teruel, X., Unnikrishnan, P., and Zhang, G. (2009). The Design of OpenMP Tasks. *Parallel and Distributed Systems, IEEE Transactions on*, 20(3):404 –418. 9

Blackford, L. S., Choi, J., Cleary, A., Petitet, A., Whaley, R. C., Demmel, J., Dhillon, I., Stanley, K., Dongarra, J., Hammarling, S., Henry, G., and Walker, D. (1996). ScaLAPACK: a portable linear algebra library for distributed memory computers - design issues and performance. In *Supercomputing '96: Proceedings of the 1996 ACM/IEEE conference on Supercomputing (CDROM)*, page 5, Washington, DC, USA. IEEE Computer Society. 11, 15, 58

Blumofe, R. D., Joerg, C. F., Kuszmaul, B. C., Leiserson, C. E., Randall, K. H., and Zhou, Y. (1995). Cilk: An efficient multithreaded runtime system. *SIGPLAN Not.*, 30:207–216. 9

Bosilca, G., Bouteiller, A., Danalis, A., Faverge, M., Haidar, A., Herault, T., Kurzak, J., Langou, J., Lemarinier, P., Ltaief, H., Luszczek, P., YarKhan, A., and Dongarra, J. (2011). 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. IEEE Computer Society. 67

Bosilca, G., Bouteiller, A., Danalis, A., Faverge, M., Haidar, H., Herault, T., Kurzak, J., Langou, J., Lemarinier, P., Ltaief, H., Luszczek, P., YarKhan, A., and Dongarra, J. (2010a). Distributed-Memory Task Execution and Dependence Tracking within DAGuE and the DPLASMA Project. Technical Report 232, LAPACK Working Note.

Bosilca, G., Bouteiller, A., Danalis, A., Herault, T., Lemarinier, P., and Dongarra, J. (2010b). DAGuE: A generic distributed DAG engine for high performance computing. *Innovative Computing Laboratory, University of Tennessee, Tech. Rep.* 12

Bosilca, G., Bouteiller, A., Danalis, A., Herault, T., Lemarinier, P., and Dongarra, J. (2012). DAGuE: A generic distributed DAG engine for High Performance Computing. *Parallel Computing*, 38(1-2):37–51. 67

Buttari, A., Dongarra, J., Kurzak, J., Langou, J., Luszczek, P., and Tomov, S. (2007). The Impact of Multicore on Math Software. In Kågström, B., Elmroth, E., Dongarra, J., and Wasniewski, J., editors, *Applied Parallel Computing. State of the Art in Scientific*

Computing, volume 4699 of *Lecture Notes in Computer Science*, pages 1–10. Springer Berlin / Heidelberg. 1, 15, 21

Buttari, A., Langou, J., Kurzak, J., and Dongarra, J. (2008). Parallel tiled QR factorization for multicore architectures. *Concurrency and Computatation: Practice and Experience*, 20(13):1573–1590. 15

Buttari, A., Langou, J., Kurzak, J., and Dongarra, J. (2009). A class of parallel tiled linear algebra algorithms for multicore architectures. *Parallel Comput.*, 35(1):38–53. 17, 18, 21

Chan, E. (2010). *Application of Dependence Analysis and Runtime Data Flow Graph Scheduling to Matrix Computations*. PhD thesis, University of Texas, Austin. 10

Chan, E., Quintana-Orti, E. S., Quintana-Orti, G., and van de Geijn, R. (2007). Supermatrix out-of-order scheduling of matrix operations for smp and multi-core architectures. In *Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures*, SPAA '07, pages 116–125, New York, NY, USA. ACM. 10

Choi, J., Dongarra, J., Ostrouchov, S., Petitet, A., Walker, D. W., and Whaley, R. C. (1996). A Proposal for a Set of Parallel Basic Linear Algebra Subprograms. In *Proceedings of the Second International Workshop on Applied Parallel Computing, Computations in Physics, Chemistry and Engineering Science*, PARA '95, pages 107–114, London, UK, UK. Springer-Verlag. 15

Dagum, L. and Menon, R. (1998). OpenMP: an industry standard API for shared-memory programming. *Computational Science Engineering, IEEE*, 5(1):46 –55. 9

Dennis, J. B. (1980). Data flow supercomputers. *Computer*, 13(11):48–56. 7

Dongarra, J., Faverge, M., Ltaief, H., and Luszczek, P. (2011). High performance matrix inversion based on lu factorization for multicore architectures. In *Proceedings of the 2011 ACM international workshop on Many task computing on grids and supercomputers*, MTAGS '11, pages 33–42, New York, NY, USA. ACM. 23, 34

Dongarra, J., Ltaief, H., Luszczek, P., and Weaver, V. (2012). Energy footprint of advanced dense numerical linear algebra using tile algorithms on multicore architecture. In *2nd International Conference on Cloud and Green Computing (CGC 2012), Xiangtan, China*. 85

Dongarra, J., van de Geijn, R., and Walker, D. (1992). A look at scalable dense linear algebra libraries. In *Scalable High Performance Computing Conference, 1992. SHPCC-92. Proceedings.*, pages 372 –379. 58

Ellson, J., Gansner, E., Koutsofios, L., North, S. C., and Woodhull, G. (2002). GraphViz—Open Source Graph Drawing Tools. 102, 105

Gunnels, J. A., Gustavson, F. G., Henry, G. M., and van de Geijn, R. A. (2001). FLAME: Formal Linear Algebra Methods Environment. *ACM Trans. Math. Softw.*, 27:422–455. 10, 15

Gunter, B. C. and van de Geijn, R. A. (2005). Parallel out-of-core computation and updating the QR factorization. *ACM Transactions on Mathematical Software*, 31(1):60–78. 17

Gurd, J. R., Kirkham, C. C., and Watson, I. (1985). The Manchester prototype dataflow computer. *Commun. ACM*, 28(1):34–52. 7

Haidar, A., Ltaief, H., YarKhan, A., and Dongarra, J. (2011). Analysis of dynamically scheduled tile algorithms for dense linear algebra on multicore architectures. *Concurr. Comput. : Pract. Exper.*, 24(3):305–321. 20, 32

Huang, C. and Kale, L. (2007). Charisma: Orchestrating migratable parallel objects. In *HPDC '07: Proceedings of the 16th international symposium on High performance distributed computing*, pages 75–84, New York, NY, USA. ACM. 9

Huang, C., Lawlor, O., and Kalé, L. V. (2003). Adaptive MPI. In *Proceedings of the 16th International Workshop on Languages and Compilers for Parallel Computing (LCPC 03)*, pages 306–322. 9

Husbands, P. and Yelick, K. (2007). Multi-threading and one-sided communication in parallel LU factorization. In *Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, SC '07, pages 31:1–31:10, New York, NY, USA. ACM. 11

Johnson, G. W. and Jennings, R. (2001). *LabVIEW Graphical Programming*. McGraw-Hill Professional, 3rd edition. 8

Johnston, W. M., Hanna, J. R. P., and Millar, R. J. (2004). Advances in dataflow programming languages. *ACM Comput. Surv.*, 36:1–34. 8

Kale, L. V. and Krishnan, S. (1993). CHARM++: A portable concurrent object oriented system based on C++. *SIGPLAN Not.*, 28(10):91–108. 9

Kurzak, J., Buttari, A., and Dongarra, J. (2008). Solving Systems of Linear Equations on the CELL Processor Using Cholesky Factorization. *IEEE Trans. Parallel Distrib. Syst.*, 19:1175–1186. 15, 21

Kurzak, J. and Dongarra, J. (2009a). Fully dynamic scheduler for numerical scheduling on multicore processors. Technical Report LAWN (LAPACK Working Note) 220, UT-CS-09-643, Innovative Computing Lab, University of Tennessee. 22

Kurzak, J. and Dongarra, J. (2009b). QR factorization for the Cell Broadband Engine. *Scientific Programming*, 17(1):31–42. 15

Kurzak, J., Luszczek, P., YarKhan, A., Faverge, M., Langou, J., Bouwmeester, H., and Dongarra, J. (2013). Multithreading in the PLASMA Library. In *Handbook of Multi and Many-Core Processing: Architecture, Algorithms, Programming, and Applications*, Computer and Information Science Series. Chapman and Hall/CRC. 20, 23, 34, 45

Kurzak, J., Nath, R., Du, P., and Dongarra, J. (2010). An Implementation of the Tile QR Factorization for a GPU and Multiple CPUs. In *PARA'10: State of the Art in Scientific and Parallel Computing,*, Reykjavík, Iceland. 107

Kurzak, J., Nath, R., Du, P., and Dongarra, J. (2012). An implementation of the tile QR factorization for a GPU and multiple CPUs. In *Proceedings of the 10th international*

conference on Applied Parallel and Scientific Computing - Volume 2, PARA'10, pages 248–257, Berlin, Heidelberg. Springer-Verlag. 39, 85

Marjanović, V., Labarta, J., Ayguadé, E., and Valero, M. (2010). Overlapping communication and computation by using a hybrid MPI/SMPSs approach. In *Proceedings of the 24th ACM International Conference on Supercomputing*, ICS '10, pages 5–16, New York, NY, USA. ACM. 10

Parker, S. and Johnson, C. (1995). SCIRun: A Scientific Programming Environment for Computational Steering. In *Supercomputing, 1995. Proceedings of the IEEE/ACM SC95 Conference*, page 52. 8

Pérez, J. M., Badia, R. M., and Labarta, J. (2008). 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. 10

Quintana-Ortí, E. S. and Van De Geijn, R. A. (2008). Updating an LU Factorization with Pivoting. *ACM Trans. Math. Softw.*, 35(2):11:1–11:16. 15, 21

Rodrigues, J. E. (1969). A graph model for parallel computations. Technical report, Massachusetts Institute of Technology, Cambridge, MA, USA. 7

Snir, M., Otto, S., Huss-Lederman, S., Walker, D., and Dongarra, J. (1998). *MPI-The Complete Reference, Volume 1: The MPI Core*. MIT Press, Cambridge, MA, USA, 2nd. (revised) edition. 2

Song, F., YarKhan, A., and Dongarra, J. (2009). Dynamic task scheduling for linear algebra algorithms on distributed-memory multicore systems. In *SC '09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, pages 1–11, New York, NY, USA. ACM.

Sorensen, D. (1985). Analysis of Pairwise Pivoting in Gaussian Elimination. *Computers, IEEE Transactions on*, C-34(3):274 –278. 66, 77

Tomasulo, R. M. (1967). An Efficient Algorithm for Exploiting Multiple Arithmetic Units. *IBM Journal of Research and Development*, 11(1):25 –33. 10

Vajracharya, S., Karmesin, S., Beckman, P., Crotinger, J., Malony, A., Shende, S., Oldehoeft, R., and Smith, S. (1999). SMARTS: Exploiting temporal locality and parallelism through vertical execution. In *Proceedings of the 13th international conference on Supercomputing*, ICS '99, pages 302–310, New York, NY, USA. ACM. 10

Veen, A. H. (1986). Dataflow machine architecture. *ACM Comput. Surv.*, 18(4):365–396. 8

YarKhan, A., Kurzak, J., and Dongarra, J. (2011a). QUARK: Development of a Task Execution Environment Driven by Linear Algebra Applications. In *SIAM Conference on Computational Science and Engineering (CSE 11), Session on Superscalar Scheduling for Multicore Systems and Accelerators*. Unrefereed conference presentation.

YarKhan, A., Kurzak, J., and Dongarra, J. (2011b). QUARK Users' Guide: QUeueing And Runtime for Kernels. Technical report, Innovative Computing Laboratory, University of Tennessee. 22, 95

Appendix

Appendix A

QUARK API and Users Guide *

QUARK (QUeueing And Runtime for Kernels) is a runtime environment for the dynamic scheduling and execution of applications that consist of precedence-constrained kernels on multicore, multi-socket shared memory systems. The goal of the project is to provide an easy-to-use interface to application programming, with an efficient implementation which will scale to large shared memory architectures. The main design principle behind QUARK is implementation of the dataflow model, where scheduling is based on data dependencies between tasks in a task graph. The data dependencies are inferred through a runtime analysis of data hazards implicit in the data usage by the kernels.

The focus of QUARK development is to support dynamic linear algebra algorithms for the PLASMA linear algebra project Agullo et al. (2010). However, QUARK is capable of supporting other application domains where the application can be decomposed into tasks with data dependencies.

Please note, the terms function, kernel and task may be used interchangeably to refer to a function that is to be executed by the QUARK runtime system. Additionally, depending on the context, the terms argument, parameter or data item may all be used to refer to the parameters of a function.

*The material in this appendix has been published in YarKhan, A., Kurzak, J., and Dongarra, J., QUARK Users' Guide: QUeueing And Runtime for Kernels. Technical report, Innovative Computing Laboratory, University of Tennessee (YarKhan et al., 2011b).

A.1 Basic Usage of the QUARK API

QUARK is often used as a library for a serial application, where QUARK does all the thread and resource management. There is an alternate mode where QUARK does not do internal thread management so it can be used as part of a multi-threaded library; this is described later.

Hello World

Fig. A.1 shows a variation of "Hello World" which demonstrates many of the basic features of QUARK. This is a fully functional C program that demonstrates how a simple program in QUARK may be implemented.

First, a new QUARK instance is created in the main program with two threads, the master and one spawned worker thread. Then, at each iteration of the loop in the main program, a new is inserted into the QUARK runtime system. The task will call the function `hello_world_task` when it is executed by the runtime. The `idx` argument is passed as `VALUE` parameter, since is unchanged by the function. The `str` argument is marked `INOUT` since the string will be altered by the `hello_world_task`. In this simple program, the use of a `INOUT` dependency keeps the serialized semantics of the original loop.

In `hello_world_task`, the data is unpacked from QUARK and assigned to the `idx` and `str` variables using the unpack macro. The string is then printed, and after that it is altered by changing the `idx` character to an underscore. This allows us to see how the string is changed by each task, and to note that the original serial order of the code is being preserved by multiple tasks running on multiple threads. The output of this program is shown in Fig. A.2

More details of the various QUARK calls involved in this code are given here.

Initializing QUARK The `QUARK_New` call initializes the library, and spawns and manages `num_threads` threads (including the master) to handle the computation. If `num_threads < 1`, QUARK will first check the `QUARK_NUM_THREADS` environment variable, then use as many threads as there are cores.

```
#include "quark.h"

/* This task unpacks and prints a string.  At each call, the idx
   character  is replaced with an underscore, changing the string. */
void hello_world_task( Quark *quark ) {
  int idx; char *str;
  quark_unpack_args_2( quark, idx, str );
  printf( "%s\n", str );
  str[idx] = '_';
}

/* A simple variation of "Hello World" */
main() {
  int idx;
  char str[] = "Hello World";
  Quark *quark = QUARK_New( 2 );
  for ( idx=0; idx<strlen(str); idx++ )
      QUARK_Insert_Task( quark, hello_world_task, NULL,
          sizeof(int), &idx, VALUE,
          strlen(str)*sizeof(char), str, INOUT,
          0 );
  QUARK_Delete( quark );
}
```

Figure A.1: Basic QUARK program, showing the setup of QUARK, task insertion with simple dependency information, and shutdown of QUARK.

```
Hello World
_ello World
__llo World
___lo World
____o World
_____ World
______World
_______orld
________rld
_________ld
__________d
```

Figure A.2: Output from the Basic QUARK program in Fig. A.1.

```
Quark *QUARK_New(int num_threads);
```

Adding tasks After initializing, tasks can be added to the QUARK runtime system by the master thread. The `QUARK_Insert_Task` call has many options, however only the basic usage is shown here.

```
QUARK_Insert_Task( Quark *quark, void (*function) (Quark *),
    Quark_Task_Flags *tflags,
    int sizeof_arg_1_in_bytes, void *arg_1, int arg_1_flags,
    int sizeof_arg_2_in_bytes, void *arg_2, int arg_2_flags,
    ...,
    0 );
```

The first two parameters are the QUARK data structure and a pointer to the function that is to be executed. The task flags `tflags` can be used to provide tasks specific information, such as the priority of the task or a sequence tag used to group related tasks. For basic usage, the tasks flags can be set to NULL. More advanced usage of the task flags will be described later.

The argument parameters to given to the `function` are passed as `varargs`. Each argument is presented as a triplet: the size of the argument in bytes, a pointer to the argument, and a flag indicating the way that the argument is to be used. This sequence of triplets is terminated by sending a 0 as the argument size. The argument flag can be one of `VALUE`, `INPUT`, `OUTPUT`, `INOUT`, `NODEP` and `SCRATCH`. These denote different ways that the argument is going to be used by `function`. Here, we describe meaning of `VALUE` and `INOUT`. The other argument flags will be described later in this guide.

`VALUE` The parameter data is copied to the QUARK task and is not used for dependency resolution. Since the parameter is copied over, it can be altered by the master thread without affecting the task.

`INOUT` The parameter is used as input as well as output in the `function`. Preceding reads and writes must be satisfied before this `function` can be executed.

QUARK schedules the `function` for execution when all the dependencies for the data arguments (`arg_1`, `arg_2`, ...) are satisfied.

Finalizing QUARK When the user is done with inserting all their tasks, they will need call `QUARK_Delete` to wait for the QUARK runtime to finish executing any remaining tasks and release the QUARK data structures. All the spawned worker threads will also be finalized by this call.

```
void QUARK_Delete(Quark * quark);
```

A.2 Advanced Usage

QUARK offers a number of features that allow finer control over the runtime system and the task scheduling and execution. Most these features are controlled either via the task flags passed into each task, or via the argument flags provided to the various arguments. A few global features are enabled using environment variables. The features are described and summarized here.

Task Scheduling In order to make sense of some of the features, the default task scheduling method in QUARK needs to be outlined. At a very high level, QUARK uses data locality information to assign tasks whose dependencies are satisfied to threads where there may be data reuse. The threads execute assigned tasks using a FIFO priority queue. A thread that does not have any assigned tasks may steal tasks from the back of the queue of another thread using LIFO work-stealing policy.

Data Locality QUARK will attempt to use data locality information in making decisions about where to schedule tasks. In the current version of QUARK, the arguments will be examined to determine the output data. The task will be assigned by default to the same thread that has previously written that output data. This heuristic should enable cache reuse under many practical circumstances.

Argument Flags

When a task is inserted into the QUARK runtime system, its parameters are specified as triplets. The parameter size in bytes, a pointer to the parameter (even scalers need to be passed by reference), and some flags that are used to specify how the parameter is going to be used by the task and how dependencies on that parameter are to be resolved. The parameter must always specify its usage as one of the following: `VALUE`, `INPUT`, `OUTPUT`, `INOUT`, `NODEP` and `SCRATCH`.

```
QUARK_Insert_Task( Quark *quark, void (*function) (Quark *),
    int sizeof_arg_1_in_bytes, void *arg_1, int arg_1_flags,
    int sizeof_arg_2_in_bytes, void *arg_2, int arg_2_flags,
    ...,
    0 );
```

`VALUE` The parameter data is copied to the QUARK task and is not used for dependency resolution.

`INPUT` The parameter is used as input only in the `function`; preceding write operations on this data must be satisfied before this `function` can be executed.

`INOUT` The parameter is used as input as well as output in the `function`. Preceding reads and writes must be satisfied before this `function` can be executed.

`OUTPUT` This parameter is used as output. Preceding reads and writes must be satisfied before this `function` can be executed.

`NODEP` The parameter is declared by the programmer to not cause any dependency. This allows a programmer flexibility in forcing scheduling; however it should be used with caution. Sometimes this is used if the programmer knows that sufficient dependencies are maintained by other parameters.

`SCRATCH` The parameter is declared as temporary scratch space, and if the parameter pointer is NULL, QUARK will allocate the data when needed and pass it to `function`.

In addition, for parameters that may involve dependencies (i.e., `INPUT`, `INOUT`, `OUTPUT`) some optional information can be passed in via the parameter flags.

`ACCUMULATOR` A parameter that is flagged as ACCUMULATOR and accessed successively by a set of tasks is accumulating data within that parameter. This lets QUARK know that the access to this parameter by those tasks can be safely reordered.

`GATHERV` This flag declares to QUARK that the data is being gathered into the data parameter in a non-conflicting manner. A successive series of `GATHERV` accesses to this data parameter can safely be performed simultaneously.

`LOCALITY` This flag lets QUARK know that data locality and cache reuse should follow this data item. If possible, multiple tasks using this data item should be run on the same thread.

`QUARK_REGION_X` A parameter can be viewed as consisting as the union of eight sub-regions (`QUARK_REGION_0` thru `QUARK_REGION_7`). Using these region flags allows the dependency processing to work on subsets of the entire data region. If regions are used, they should combine to cover eight sub-regions.

The following gives some idea of how the various parameter flags may be combined to provide information to QUARK.

```
QUARK_Task_Insert( quark, function, &tflags,
    size_of_data, ptr_to_data, arg_flags,
    sizeof(int), &N, VALUE,
    N*N*sizeof(double), A, INOUT | ACCUMULATOR,
    N*N*sizeof(double), B, INOUT | LOCALITY,
    N*N*sizeof(double), C, INPUT | QUARK_REGION_1 | QUARK_REGION_2,
    N*N*sizeof(double), D, INOUT ,
    N*N*sizeof(double), E, INOUT | QUARK_REGION_5 | QUARK_REGION_6,
    N*N*sizeof(double), F, INOUT | GATHERV,
    N*N*sizeof(double), G, INPUT,
    ... , 0 )
```

Task Flags

QUARK passes task specific information to the runtime system using task flags. These flags can specify things such as a color and label for visualizing the task DAG, priorities used for scheduling the task, and a way to collect tasks into sequences for error handling. Task flags are created and initialized as shown here; any flag that is not specified will have a default value. More information about the usage of the various flags follows.

```
Quark_Task_Flags tflags = Quark_Task_Flags_Initializer;
QUARK_Task_Flag_Set( &tflags, TASK_COLOR, "red" );
QUARK_Task_Flag_Set( &tflags, TASK_LABEL, "myDGEMM" );
QUARK_Task_Flag_Set( &tflags, TASK_PRIORITY, 100 );
QUARK_Task_Flag_Set( &tflags, TASK_LOCK_TO_THREAD, thread_number );
QUARK_Task_Flag_Set( &tflags, TASK_SEQUENCE, sequence_ptr );
QUARK_Task_Flag_Set( &tflags, TASK_THREAD_COUNT, num );
QUARK_Task_Flag_Set( &tflags, TASK_SET_TO_MANUAL_SCHEDULING, 0_or_1 );
QUARK_Task_Insert( quark, function, &tflags, ..., 0 )
// Task flags for the current task can be obtained using
intptr val = QUARK_Task_Flag_Get( quark, \textit{TASK_FLAG_NAME} );
```

Visualizing Runtime DAGs

QUARK can generate GraphViz Ellson et al. (2002) files at runtime that represent the DAG of executed tasks. To enable this feature, set the environment variable `QUARK_DOT_DAG_ENABLE=1`. By default, the graph will have no labels on nodes and all nodes have the same color. It is possible to change the display of the DAG nodes by setting task flags when the `QUARK_Insert_Task` function is called. The color strings that can be used are described in the GraphViz documentation.

```
Quark_Task_Flags tflags = Quark_Task_Flags_Initializer;
QUARK_Task_Flag_Set( &tflags, TASK_COLOR, "red" );
QUARK_Task_Flag_Set( &tflags, TASK_LABEL, "myDGEMM" );
QUARK_Task_Insert( quark, function, &tflags, ..., 0 )
```

After the program executes. the DAG will be written to file in the execution directory with the name "`dot_dag_file.dot`". Please see the GraphViz documentation on how to manipulate this file. For example, to setup DAG generation, execute, and translate the generated DAG to a PDF format use the following commands.

```
export QUARK_DOT_DAG_ENABLE=1
./execute_my_quark_binary
dot -Tpdf -o mydag.pdf dot_dag_file.dot
```

Standard dependencies between tasks are shown as black arrows, red arrows represent Write-After-Read (WAR) dependencies that could be eliminated by making data copies, green arrows mark parallel task execution allowed by special `GATHERV` data dependencies specified by the developer.

Assigning Priorities to Tasks

If the developer has knowledge of the algorithm that specifies some tasks should have higher priority than other tasks, that information can be provided to the QUARK runtime environment. After the dependencies for various tasks are satisfied, the tasks are assigned to worker priority-queues. At that point, higher priority tasks get executed earlier, which may lead to a execution path that more closely matches the critical path of the DAG. The priority value can be any integer, with the default being priority 0.

```
Quark_Task_Flags tflags = Quark_Task_Flags_Initializer;
QUARK_Task_Flag_Set( &tflags, TASK_PRIORITY, 100 );
QUARK_Task_Insert( quark, function, &tflags, ... , 0 )
```

Task Sequences and Error Handling

Linear algebra programs can be composed of multiple algorithmic sequences which can fail or succeed independently. In order to handle this requirement, the QUARK dynamic runtime system provides a way to manipulate sequences of tasks. If an error situation is detected during the execution of one algorithm, then the associated sequence can be canceled, without affecting other parts of the program. If a task is added to a sequence that has been canceled, it is silently skipped.

```
Quark_Sequence *seq = QUARK_Sequence_Create( quark );
Quark_Task_Flags tflags = Quark_Task_Flags_Initializer;
QUARK_Task_Flag_Set( &tflags, TASK_SEQUENCE, seq );
QUARK_Task_Insert( quark, function, &tflags, ..., 0 )
// If an error occurs, a task can call
Quark_Sequence *seq = QUARK_Task_Flag_Get( quark, TASK_SEQUENCE )
QUARK_Sequence_Cancel( quark, seq );
```

Multi-threaded Tasks

Under certain circumstances it can be beneficial to assign multiple thread to a single task. One example of this is in the panel factorization step of LU factorization. Since this task deals with the entire panel, it can become a bottleneck for the execution unless multiple threads are assigned to it. QUARK will call the task multiple times using multiple threads, however, the management of these multiple threads is left up to the developer. There is support to find out the rank of a thread within the set of multiple threads, so that a developer can take the appropriate action based on the rank. Note, multi-threaded tasks are locked to their pre-assigned threads and cannot be stolen and executed by other threads.

```
Quark_Task_Flags tflags = Quark_Task_Flags_Initializer;
QUARK_Task_Flag_Set( &tflags, TASK_THREAD_COUNT, 4 );
QUARK_Task_Insert( quark, function, &tflags, ..., 0 )
// Within the task, a developer can call
int rank = QUARK_Get_RankInTask( quark );
```

Locking Tasks to Threads

For the developer who knows best, it is possible to lock tasks to threads, This may be done if there is strong cache benefit of forcing a certain set of tasks to run on some specific thread (or core). When a task is locked to a thread, it cannot be stolen or reassigned to another thread.

```
Quark_Task_Flags tflags = Quark_Task_Flags_Initializer;
QUARK_Task_Flag_Set( &tflags, TASK_LOCK_TO_THREAD, 3 );
QUARK_Task_Insert( quark, function, &tflags, ..., 0 )
```

Manual Thread Scheduling

There are occasions when the developer may want to switch the scheduling mode of a thread from automatic to manual. For example, this thread is going to control a GPU and you wish to assign GPU specific tasks to the thread with full control. That thread will no longer take part in automatic task assignment or work stealing. The action of switching a thread to manual scheduling is done by assigning a specific task to the desired thread, and setting the boolean `THREAD_SET_TO_MANUAL_SCHEDULING` to 1. When that task executes, the thread's scheduling mode will be set to manual.

```
Quark_Task_Flags tflags = Quark_Task_Flags_Initializer;
// I want thread 2 to control GPUs and nothing else
QUARK_Task_Flag_Set( &tflags, TASK_LOCK_TO_THREAD, 2 );
QUARK_Task_Flag_Set( &tflags, THREAD_SET_TO_MANUAL_SCHEDULING, 1 );
QUARK_Task_Insert( quark, some_special_function, &tflags, ... , 0 )
```

A.3 Environment Variables

There are some environment variable that affect the behavior of QUARK.

Task Window Sizes and DAGs For many linear algebra algorithms, the number of tasks is on the order of N^3. This means that even for relatively small applications, the DAG can be enormous. Because of this, QUARK uses a window of active tasks to keep the number of tasks manageable in terms of memory usage and scheduling overhead. The task window size is kept at reasonable defaults, but may be altered via several environment variable. The task window size per thread can be set by `QUARK_UNROLL_TASKS_PER_THREAD=500`, or the total task window size over all threads can be set by `QUARK_UNROLL_TASKS=5000`. An interesting usage of this variable for debugging purposes is to use `QUARK_UNROLL_TASKS=1` to cause a serial execution of the program, with one task being inserted into QUARK, executing, and then the next task being inserted.

Generating DAGs for Visualization QUARK can generate GraphViz Ellson et al. (2002) files at runtime that represent the DAG of executed tasks. Setting environment

variable `QUARK_DOT_DAG_ENABLE=1` will cause a file `dot_dag_file.dot` to be created in the execution directory. It is possible to change the visualized display of the DAG nodes by setting task flags when the `QUARK_Insert_Task` function is called.

A.4 Other Topics

Pipelining DAGs

A dynamic scheduler such as QUARK is expected to be slower than a static scheduler if they are both executing the same single algorithm because dynamic scheduling has overheads that are not present in the static scheduler. However, a dynamic scheduler has advantages in certain circumstances. Firstly, if there are several algorithms that need to be executed in succession, the static scheduler will run them one after another, synchronizing between the algorithms. However, a dynamic scheduler can interleave the tasks from the various algorithms, scheduling them when their dependencies are satisfied. This can lead to a faster execution for certain programs and data sizes Agullo et al. (2011). Secondly, a dynamic scheduler can perform runtime optimizations, such as reordering dependencies (e.g. see `ACCUMULATOR` argument flag), effectively altering the DAG and speeding up execution.

Incorporating QUARK in Another Library

QUARK was designed to cooperate with other multi-threaded libraries, so it can use computation threads spawned externally to QUARK. This allows the PLASMA library to easily switch between its internal static scheduling and the dynamic runtime environment offered by QUARK. If a developer wants to manage threads outside QUARK, the master thread needs to call `QUARK_Setup` to setup Quark data structures . Then each worker thread which the developer has created externally needs to call `QUARK_Worker_Loop`, after which the worker will wait for tasks. The master thread then adds tasks in the normal way, When the master thread is done adding tasks, it calls `QUARK_Waitall`. The worker threads will finish the current tasks, and return control to the developers program. When the master is all done, it can call `QUARK_Free` to free all structures allocated by `QUARK_Setup`.

```
// Master thread sets up data structures
Quark *QUARK_Setup( int num_threads );
```

```
// Worker threads hand control to QUARK temporarily
void QUARK_Worker_Loop( Quark *quark, int thread_rank );
// Master tells spawned threads to finish work and return control
void QUARK_Waitall( Quark * quark );
// Master frees data structures
void QUARK_Free( Quark * quark );
```

List API for Task Arguments

QUARK provides a secondary list-style API for adding arguments to a function. This proves useful in the situation that there are a very large number of dependencies for a function. For example, a function to be executed on a GPU may take a large number of data items, and it would be simpler to provide that data by looping over indicesKurzak et al. (2010). A second situation where a list-style API is useful is when the actual number of dependencies of a function are not known until runtime, so it is not possible to use the standard varargs based `QUARK_Insert_Task` method of adding arguments.

```
// Create a task data structure to hold arguments
Quark_Task *task = QUARK_Task_Init( quark, function, task_flags )
// Add (or pack) the arguments into a task data structure
QUARK_Task_Pack_Arg( quark, task, arg_size, arg_ptr, arg_flags )
// Insert the packed task data structure into the scheduler for execution
QUARK_Insert_Task_Packed( quark, task )
```

License Information

QUARK is a software package provided by University of Tennessee. QUARK's license is a BSD-style permissive free software license (properly called modified BSD). It allows proprietary commercial use, and for the software released under the license to be incorporated into proprietary commercial products. Works based on the material may be released under a proprietary license as long as QUARK's license requirements are maintained, as stated in the LICENSE file, located in the main directory of the QUARK distribution. In contrast to copyleft licenses, like the GNU General Public License,

QUARK's license allows for copies and derivatives of the source code to be made available on terms more restrictive than those of the original license.

Appendix B

Publications

Song, F., YarKhan, A., and Dongarra, J. (2009). Dynamic task scheduling for linear algebra algorithms on distributed-memory multicore systems. In *SC '09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, pages 1–11, New York, NY, USA. ACM

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

Bosilca, G., Bouteiller, A., Danalis, A., Faverge, M., Haidar, H., Herault, T., Kurzak, J., Langou, J., Lemarinier, P., Ltaief, H., Luszczek, P., YarKhan, A., and Dongarra, J. (2010a). Distributed-Memory Task Execution and Dependence Tracking within DAGuE and the DPLASMA Project. Technical Report 232, LAPACK Working Note

Bosilca, G., Bouteiller, A., Danalis, A., Faverge, M., Haidar, A., Herault, T., Kurzak, J., Langou, J., Lemarinier, P., Ltaief, H., Luszczek, P., YarKhan, A., and Dongarra, J. (2011). 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. IEEE Computer Society

YarKhan, A., Kurzak, J., and Dongarra, J. (2011b). QUARK Users' Guide: QUeueing And Runtime for Kernels. Technical report, Innovative Computing Laboratory, University of Tennessee

YarKhan, A., Kurzak, J., and Dongarra, J. (2011a). QUARK: Development of a Task Execution Environment Driven by Linear Algebra Applications. In *SIAM Conference on Computational Science and Engineering (CSE 11), Session on Superscalar Scheduling for Multicore Systems and Accelerators*. Unrefereed conference presentation

Haidar, A., Ltaief, H., YarKhan, A., and Dongarra, J. (2011). Analysis of dynamically scheduled tile algorithms for dense linear algebra on multicore architectures. *Concurr. Comput. : Pract. Exper.*, 24(3):305–321

Kurzak, J., Luszczek, P., YarKhan, A., Faverge, M., Langou, J., Bouwmeester, H., and Dongarra, J. (2013). Multithreading in the PLASMA Library. In *Handbook of Multi and Many-Core Processing: Architecture, Algorithms, Programming, and Applications*, Computer and Information Science Series. Chapman and Hall/CRC

Vita

Asim YarKhan was born in Karachi, Pakistan on November 1, 1965. He received his Bachelor of Arts in Computer Science and Mathematics in 1988 from the College of Wooster in Wooster, Ohio. He received a Master of Science in Applied Mathematics in 1993 from the University of Akron, Ohio and a Master of Science in Computer Science in 1994 from the Pennsylvania State University, Pennsylvania. He worked as a Research Scientist in data mining from 1999 to 2001 at 212 Studios Inc. in Knoxville, Tennessee. He has worked as a Senior Research Associate at the Innovative Computing Laboratory at the University of Tennessee since 2001. He received his Doctor of Philosophy degree in Computer Science from the University of Tennessee in 2012.