2014 43rd International Conference on Parallel Processing

Parallel Simulation of Superscalar Scheduling

Blake Haugen
Innovative Computing Laboratory
University of Tennessee Knoxville
bhaugen@utk.edu

Jakub Kurzak
Innovative Computing Laboratory
University of Tennessee Knoxville
kurzak@icl.utk.edu

Asim YarKhan
Innovative Computing Laboratory
University of Tennessee Knoxville
yarkhan@icl.utk.edu

Piotr Luszczek
Innovative Computing Laboratory
University of Tennessee Knoxville
luszczek@eecs.utk.edu

Jack Dongarra
Innovative Computing Laboratory
University of Tennessee Knoxville
Oak Ridge National Laboratory
University of Manchester
dongarra@eecs.utk.edu

***Abstract*—Computers have been moving toward a multicore paradigm for the last several years. As a result of the recent multicore paradigm shift, software developers must design applications that exploit the inherent parallelism of modern computing architectures. One of the areas of research to simplify this shift is the development of dynamic scheduling utilities that allow the developer to specify serial code that can be parallelized using a library or compiler technology. While these tools certainly increase the developer's productivity, they can obfuscate performance bottlenecks. For this reason, it is important to evaluate algorithm performance in order to ensure that the performance of a given algorithm is being realized using dynamic scheduling utilities.**

This paper presents the methodology and results of a new performance analysis tool that aims to accurately simulate the performance of various superscalar schedulers, including OmpSs, StarPU, and QUARK. The process begins with careful timing of each of the computational routines that make up the algorithm. The simulation tool then uses the timing of the computational kernels in conjunction with the dependency management provided by the superscalar scheduler in order to simulate the execution time of the algorithm. This tool demonstrates that simulation results of various algorithms can accurately predict the performance of a complex dynamic scheduling system.

I. Introduction

Multicore processors are prevalent in mobile, server, and HPC computing, and their ubiquity continues to push for a tremendous change in software development methodology. In the single core era, software developers could write a single routine and expect the performance to increase with each newer, faster generation of processors. This trend of ever-increasing CPU frequencies came to an end when new CPUs presented multiple cores but clock frequency stagnated.

The multicore hardware required an enormous shift in the way software was developed in order to achieve maximum performance. As one of many paradigms, multi-threaded programming allows software to use all of the available cores to perform computations simultaneously in order to obtain a much larger percentage of the theoretical peak performance on cache-resident calculations, such as the ones originating in dense linear algebra. While POSIX threads, or equivalent interface on non-POSIX platforms, allow the developer to express parallel algorithms, the process of software development can be quite challenging due to the tedious process of tracking the data and control flow while maintaining correct concurrent access and facilitate independent execution of code on the CPU cores.

To address this, many tools have been developed to increase the programmer's productivity while allowing developers to exploit the power of multicore architectures. Arguably the best known example of this is the OpenMP standard [1]. OpenMP allows the user to develop sequential code and then augment with `#pragma` directives to annotate loops that the compiler should parallelize across the cores. In July 2013, the latest version of the OpenMP standard was released: version 4.0. This included support for hardware accelerators such as GPUs but also several extensions to allow the programmer to use dynamic task scheduling. Currently, the standard only supports control dependences. For many applications, the developer is left with tracking all data dependences by, for example, translating them into task dependences that are handled by the OpenMP runtime.

Cilk is another software system that allows the developer to specify the code that may be executed in parallel, and allows the compiler to parallelize portions of the code [2]. The two potential drawbacks of this approach are: 1) the developer must explicitly recognize and annotate the parallelism for the compiler, and 2) the BSP parallelism model [3], [4] incurs unnecessary synchronization, which often does not allow for the most efficient use of the computational resources as it leads to memory contention and load imbalance.

Another class of parallel technologies employs superscalar execution of tasks. In this paradigm, the developer creates tasks with input and output (or both) dependences. These tasks are then submitted to the scheduler in a serial fashion. The tasks are scheduled dynamically at runtime by ensuring that dependences are satisfied and the user data flows between the tasks to trigger execution of new tasks that have all inputs available.

Superscalar runtime schedulers are effective when exploiting parallelism by guaranteeing fulfillment of dependences found in serial code. These systems generally make scheduling decisions at runtime and respond dynamically to any slowdowns to execution. This makes modeling such systems a very difficult task because any changes to the execution environment, such OS jitter, could change the order of execution and

0190-3918/14 $31.00 © 2014 IEEE
DOI 10.1109/ICPP.2014.21

alter memory traffic patterns, thus affecting the performance of tasks, which might cascade throughout the execution. The goal of this work is to answer this challenge by providing a model of the performance of superscalar scheduling using a discrete event simulation and make it portable across a variety of schedulers. The simulation relies on the scheduler to resolve the task dependencies and make scheduling decisions while the simulator keeps track of kernel execution times and builds a virtual execution trace. The accuracy of the simulation is evaluated using two dense linear algebra factorizations and three schedulers.

II. Related Work

Since the Minimum Multiprocessor Scheduling Problem is NP-complete [5], nearly all optimal scheduling problems in complex environments are NP-complete. This means most scheduling decisions are reached using heuristic algorithms, many of which can be found in the survey article [6]. The combination of complicated hardware configurations and scheduling heuristics make the search space too large and complex for analytical models. As an alternative to most analytical models, developers often resort to empirical and simulation-based models [7], [8].

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

Simulation is not a new concept to computer scientists, and simulation tools seem to fall into two broad categories. The first is architecture simulation where the goal is to simulate the operation of a processor or system in order to analyze the accuracy of the output or performance characteristics. These simulations often do not focus on parallelism, but rather focus on fine-grain, instruction level simulation. The gem5 [9] and SESC [10] simulators are two examples of this type of simulator. An important aspect of both of these tools is the ability to simulate out of order executions, which are common in modern computer architectures.

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

The diverse array of computing resources used in grid computing makes scheduling a very challenging problem, and the lack of reproducibility in the performance of each run made simulation a logical choice. Tools like SimGrid [11] and GridSim [12] were designed for these types of simulations. ChicSim [13] was another simulator built on top of a simulation language called Parsec.

The Optorsim project [14] is another example of a grid computing simulator, and was developed to evaluate the performance of various data duplication algorithms. Data is often duplicated in a grid computing environment in order to deal with the geographic distribution of computing resources. The duplication of data decreases data access times and accelerates job performance. The Optorsim project aimed to simulate the performance of grid computations based on the data replication strategies employed.

III. Contributions

Novel Simulation Environment for Modern Multi-threaded Workloads. The goal of the work presented in this paper is to create a *hybrid* of the two simulation methodologies described in Section II. The tasks being simulated will be larger than a single instruction modeled in the architectural simulator, but may have similar or smaller task sizes to those scheduled in a grid computing environment. The concept of out-of-order task completion, however, will need to be addressed similarly to the architectural simulations. The authors are not aware of any other work performing simulations of superscalar scheduling at the time of this writing.

Parallel Simulation. Our simulation runs execute in parallel and the only limiting factor is the speed of the scheduler. The users tend to make the tasks' running times become longer to reduce the overhead of the scheduler, and at the same time, such long tasks benefit our simulation approach.

Accelerated Simulation Time. With the use of our simulation approach to reduce the time to generate the execution traces, a two-fold speedup is not uncommon. This is in sharp contrast to some hardware simulations that tend to incur orders of magnitude slow-down. Our aim is not cycle-accurate simulation as no superscalar scheduler can ever be cicle-accurate due to direct influence of external stimuli. Nevertheless, we achieve the appreciative reduction of simulation time while still being within a few percentage points of the true running time and preserving the essential phenomena that may be observed in the execution trace.

Portability Across Diverse Implementations of Superscalar Scheduling. Our approach is agnostic with respect to the underlying superscalar scheduler, and we tested three such schedulers that constitute large code bases that would have been hard to instrument and analyze outside of our simulation strategy. Our approach makes it possible to analyze both the application and the underlying scheduler without the need to interact with the large code base of either.

Accurate Simulation Traces. Despite introducing parallelism, accelerated simulation time, and portability into our approach, the accuracy of the generated data remains very high in two essential categories. Firstly, the execution time remains very close (few percentage points) to the time predicted by the simulation. Secondly, the execution trace from the simulation retains the essential features of the trace from the actual run – an absolute must for accurate trace simulation.

IV. Background

A. Schedulers

The simulations in this paper are presented with results from three superscalar schedulers: QUARK, StarPU, and OmpSs. But the simulation framework is not limited to those and could also be extended to other task-based schedulers with minimal changes to the code base.

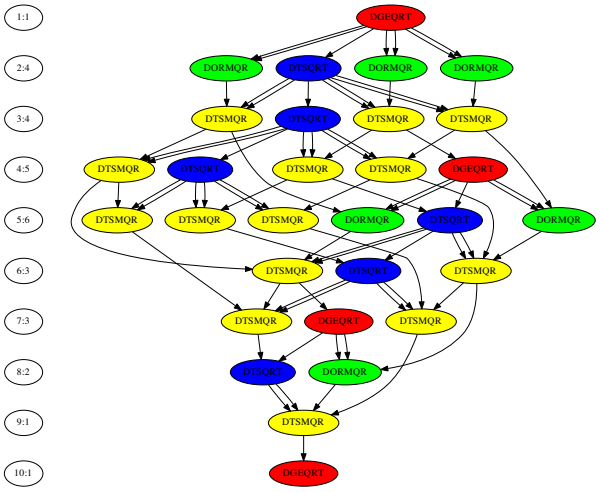


Fig. 1: An example of the DAG generated by a QR factorization of a matrix that is 4 tiles by 4 tiles. Each vertex represents one task and each edge represents a dependence. Notice that some vertices have multiple edges from a parent node indicating that there is more than one data dependence that must be satisfied.

When developing a parallel algorithm for a superscalar scheduler, the work must be broken into tasks and each input and output for the task must be explicitly designated by the developer. The tasks are then submitted to the scheduler in a serial fashion. The scheduler analyzes the Read-after-Write (RaW), Write-after-Read (WaR), and Write-after-Write (WaW) hazards and schedules each of the tasks, while maintaining all data dependencies. Each scheduler has its own method for annotating the dependences of a specific task.

The dependences for task execution form a Directed Acyclic Graph (DAG). Each vertex in the graph represents a task and each edge represents a task dependence: it connects output of one task with an input to another task. Developers visualize these DAGs in order to gain a greater understanding of how well their algorithms could perform. An example of a DAG for a QR factorization is shown in Figure 1. QR is a common linear algebra operation often used for finding the least-squares solution of linear systems. For more information see section IV-B. DAG generation as well as support for heterogeneous computing and distributed task execution are provided by some of the schedulers discussed here.

1) OmpSs: The OmpSs system, developed at the Barcelona Supercomputing Center, dates back to 1994. It was originally targeting grid environments, and was called GridSs [15]. It was later adapted to the IBM Cell B. E. processor under the name CellSs [16], and then to classic multicore processors (x86 and alike) under the name SMPSs [17], [17], [18]. The extension to GPUs (GPUSs) was introduced in 2009 [19]. The project is currently named OmpSs to underline the effort to extend the OpenMP standard with support for superscalar scheduling [20]. Due to the multiplicity of names, the project has also been intermittently referred to as StarSs [21]. The best known variant is the SMPSs multicore implementation, which is a compiler-based system that uses `#pragma` directives to annotate tasks that can be run in parallel and to decorate the data parameters with read/write usage information.

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

2) StarPU: The StarPU system developed at INRIA Bordeaux was first published in 2008 [22], [23], [24]. It is a runtime environment for task scheduling on shared memory architectures, with the original motivation of exploring task scheduling in a hybrid CPU/GPU environment.

StarPU provides multiple interfaces for task execution which gives the developer great flexibility in expressing an algorithm. One of the key abstractions of the StarPU library is the codelet. The codelet is a small structure that allows the developer to describe various versions of a particular kernel using a single interface. For example, the developer might want to define a matrix multiplication task for use in their algorithm. The user can define a codelet that provides a CPU interface as well as a GPU interface allowing StarPU to execute the code on either of the target resources. StarPU uses implicit data dependences to create a task DAG. It also profiles each task execution and uses historical runtime data to schedule tasks on the appropriate resources in heterogeneous systems, assigning tasks to CPU cores as well as GPU resources. StarPU provides a large set of interfaces and extensive functionality including execution trace, DAG generation, and several scheduling policies.

3) QUARK: QUARK (QUeuing And Runtime for Kernels) was developed at the Innovative Computing Laboratory at the University of Tennessee Knoxville. It was originally developed as the main scheduler for the Parallel Linear Algebra for Scalable Multicore Architectures (PLASMA) library [25]. It has since been released as a standalone project [26] and has been used outside its original design to schedule for a wider variety of scientific codes. In general, QUARK provides a relatively small API but it still allows the user greater flexibility in code development. The library includes a number of features critical to the operation of a numerical software suite, such as error handling extensions and task cancellation capabilities. It also provides the user with the ability to save the execution DAG to visualize the dependences present in a particular algorithm.

QUARK was originally aimed at scheduling for homogeneous multicore systems with shared memory. It has since been used to develop software for systems that contain GPUs as well as traditional CPUs [27]. It should be noted that QUARK does not provide any specific interface for accelerator support. It is the responsibility of the developer to ensure that data is transferred properly during the execution of the algorithm. It has also been extended to applications in distributed memory environments [28].

B. Tile Linear Algebra Algorithms

As a case study, this paper will use the Cholesky and QR matrix factorizations to demonstrate the accuracy of the simulation environments. One of the ways of expressing these numerical algorithms is in a tile-based fashion. The tile-based approach to linear algebra algorithms has been extensively

Algorithm 1 Tile Cholesky factorization algorithm

```
for k = 1, 2 to NT do
    {Cholesky factorization of the tile A_{k,k}}
    DPOTF2(A_{k,k})
    for i = k + 1 to NT do
        {Solve A_{k,k} X = A_{i,k}}
        DTRSM(A_{k,k}, A_{i,k})
        {Update A_{i,i} ← A_{i,i} − A_{i,k} A_{i,k}^T}
        DSYRK(A_{i,i}, A_{i,k})
    end for
    for i = k + 2 to NT do
        for j = k + 1 to i do
            {Update A_{i,j} ← A_{i,j} − A_{i,k} A_{j,k}}
            DGEMM(A_{i,j}, A_{i,k}, A_{j,k})
        end for
    end for
end for
```

presented and discussed before [29], [30], [31], [32], [33]. The tile approach consists of breaking the matrix panel factorization and trailing submatrix update steps into smaller tasks that operate on relatively small $nb \times nb$ tiles (or submatrices) of consecutive data which are organized into blocks-of-columns. The algorithms can then be restructured as tasks (which are basic linear algebra operations) that act on tiles of the matrix. The data dependences between these tasks result in a DAG where nodes of the graph represent tasks and edges represent dependences among the tasks.

The execution of the tiled algorithm is performed by asynchronously scheduling the tasks in a way that dependences are not violated. Optimally, we would like this asynchronous scheduling to result in an out-of-order superscalar execution where slower tasks are overlapped in time with fast ones, which use cache more effectively. This would be managed by having the slower tasks start early, as soon as their dependences are satisfied, while some of the parallel tasks (submatrix updates) from the previous iterations still remain to be performed and can be executed in parallel when a core becomes available.

Matrix factorization algorithms form core operations for scientific computations, 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 and QR factorization algorithms are outlined below.

1) 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 consists of $NT \times NT$ tiles. In Algorithm 1, some standard BLAS (Basic Linear Algebra Subprogram) routines are used during the factorization: DSYRK (symmetric rank-k update), DPOTF2 (unblocked Cholesky factorization), DGEMM (general matrix-matrix multiplication), and DTRSM (triangular solve). 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).

2) Tile QR factorization: The QR factorization, as implemented in LAPACK, is the decomposition of an $m \times n$ real

Algorithm 2 Tile QR factorization algorithm

```
for k = 1, 2 to NT do
    DGEQRT(A_{k,k}, T_{k,k})
    for n = k + 1 to NT do
        DORMQR(A_{k,k}, T_{k,k}, A_{k,n})
    end for
    for m = k + 1 to NT do
        DTSQRT(A_{k,k}, A_{m,k}, T_{m,k})
        for n = k + 1 to NT do
            DTSMQR(A_{k,n}, A_{m,n} A_{m,k}, T_{m,k})
        end for
    end for
end for
```

matrix 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 and details are provided elsewhere [34], [35].

In the tile QR algorithm, the dominant operation from the innermost loop is different from the standard LAPACK implementation. In LAPACK, the dominant operation is the highly optimized DGEMM, and in the tile algorithm it is a new kernel operation called DTSMQR. The DTSMQR operation, even though it is a matrix-matrix operation, has not been tuned and optimized to the extent that DGEMM has been optimized by vendors, so it reaches a lower percentage of peak performance on a machine.

C. Going from QR Factorization to a DAG

The tile based QR algorithm described in Algorithm 2 is outlined with pseudocode in Fig. 2.

The loops in the pseudocode generate a sequence of tasks as shown at the bottom of Fig. 2. Each of these tasks has a set of data parameters to which it needs access. The tasks are executed using superscalar execution, with the data access constraints preserved, and the tasks executing as soon as possible, provided there are no data hazards and there is a core available to handle the execution. Additionally, multiple tasks may have read access to a specific data parameter at the same time.

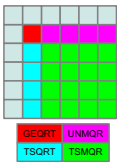
V. Simulation Methodology

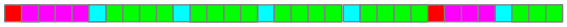
Our ultimate goal is to simulate a trace of the algorithm's execution with high fidelity. From the trace, we can gain information about scheduling decisions, execution time, and ultimately performance.

As a foundational principle, in our simulation environment we aim to have the scheduler performing the dependence tracking work, while at the same time, the work inside the tasks is not done. In other words, the scheduler keeps track of all data dependences and makes all scheduling decisions as usual, but the tasks no longer contribute useful work towards the completion of the algorithm.

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

```
for k = 0 ... TILES-1
  geqrt( A[k,k]^rw , T[k,k]^w )
  for n = k+1..TILES-1
    unmqr( A[k,k-low]^r , T[k,k]^r , A[k,n]^rw )
  for m = k+1..TILES-1
    tsqrt( A[k,k-up]^rw , A[m,k]^rw , T[m,k]^rw )
    for n = k+1..TILES-1
      tsmqr( A[m,k]^r , T[m,k]^r , A[k,n]^rw , A[m,n]^rw )
```





```
F0  geqrt( A[0,0]^rw , T[0,0]^w )
F1  unmqr( A[0,0]^r , T[0,0]^r , A[0,1]^rw )
F2  unmqr( A[0,0]^r , T[0,0]^r , A[0,2]^rw )
F3  tsqrt( A[0,0]^rw , A[1,0]^rw , T[1,0]^w )
F4  tsmqr( A[0,1]^rw , A[1,1]^rw , A[1,0]^r , T[1,0]^r )
F5  tsmqr( A[0,2]^rw , A[1,2]^rw , A[1,0]^r , T[1,0]^r )
F6  tsqrt( A[0,0]^rw , A[2,0]^rw , T[2,0]^w )
F7  tsmqr( A[0,1]^rw , A[2,1]^rw , A[2,0]^r , T[2,0]^r )
F8  tsmqr( A[0,2]^rw , A[2,2]^rw , A[2,0]^r , T[2,0]^r )
F9  geqrt( A[1,1]^rw , T[1,1]^w )
F10 unmqr( A[1,1]^r , T[1,1]^w , A[1,2]^rw )
F11 tsqrt( A[1,1]^rw , A[2,1]^rw , T[2,1]^w )
F12 tsmqr( A[1,2]^rw , A[2,2]^rw , A[2,1]^r , T[2,1]^r )
F13 geqrt( A[2,2]^rw , T[2,2]^w )
```

Fig. 2: Pseudocode for the tile QR factorization, showing all the tasks as they are sequentially generated. The data references tasks are decorated with their read and/or write status, implying data-hazards while executing the tasks.

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

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

The novelty of our simulation approach is the complete reliance on the scheduler to provide the facilities to maintain the task dependences and make all scheduling decisions. In order to create a simulation, the programmer simply replaces each task function with a call to the simulation library. Only a few lines of initialization and cleanup code before and after the execution of the algorithm simulation are needed to perform a simulation. This makes our approach portable since we neither make any assumptions about the underlying algorithm being scheduled or about data-dependence tracking, nor do we require any invasive changes to the existing implementation of tasks.

A. Tracing

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

This lead us to the following decision. Rather than attempting to modify an existing trace generation tool, we created a rudimentary trace generation environment that allows the user to log tasks during execution with the simulation (user-specified) time. After the completion of the algorithm, the trace can be converted to an SVG (Scalable Vector Graphics) file that visualizes the trace and may be rasterized at the appropriate resolution for the right amount of detail. The trace data can also be stored in a plain text file for further processing.

B. Model of Kernel Executed inside a Task

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

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

1) Timing Methodology: One of the challenges a developer faces in modeling a kernel is timing each kernel. It initially seems obvious that one could very quickly call each kernel in isolation in order to obtain an estimate of the time required for the completion of that kernel. This will likely give the developer an idea of the execution time of the kernel, but this is unlikely to give results with high accuracy. The developer must be careful to consider where the sub-matrix will be in the cache hierarchy. In the context of a true execution, the kernel may or may not have its data available at the top of the cache hierarchy.

It is possible for the developer to time the kernels in a cold cache or warm cache scenario [36]. The kernel can be accurately timed in each of these scenarios, but it is likely that the cache residency may be somewhere between warm and cold cache. Worse yet, some of the invocations of a particular kernel may occur with warm cache and others with cold cache. We circumvent this inherent limitation of a single kernel measurement by using the actual execution of the algorithm to provide the actual empirical data for future estimation. The run is done for a relatively small problem or even a portion of the problem using the desired dynamic scheduler. In practice, this solution was the most accurate in representing what the kernel performance is, not only conceptually but also in our experiments.

The results of runs shown in this paper were performed by linking with the Intel MKL library in order to obtain the best performance. As is common for large libraries, which require resource allocation, the MKL library initializes its internal state upon the first execution of a kernel and for each thread of execution. This may be easily observed as the first kernel on each thread will take significantly longer to execute than

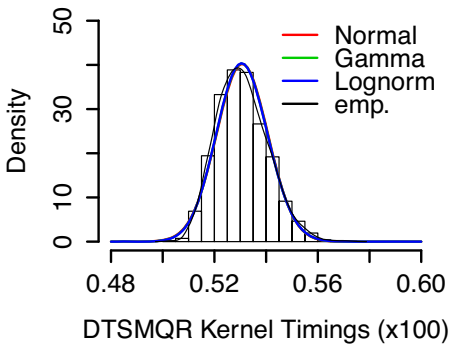


Fig. 3: A plot of the kernel execution times of the DTSMQR (part of the QR factorization) kernel along with the fitted distribution curves. The kernel execution times are multiplied by 100 for readability. The normal, gamma, and lognormal distributions appear to fit equally well.

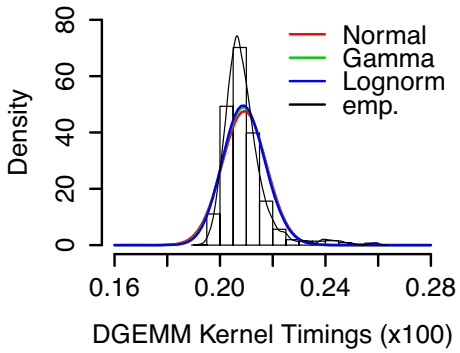


Fig. 4: A plot of the kernel execution times of the DGEMM (part of the Cholesky factorization) kernel along with the fitted distribution curves. The kernel execution times are multiplied by 100 for readability. The simple distributions do not fit quite as well as the DTSMQR kernels, but they seem to model the kernel execution times better than a constant or uniform distribution.

the following kernels. These extreme outliers can drastically affect the model fitting. For this reason, each of the threads is initialized with another call to the MKL library in order to ensure that this initialization is performed before the trace is collected.

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

In dense linear algebra, the kernels are most commonly described using the normal distribution of execution times, but similar distributions may also be used to model execution time. This assumes that the kernels are approximated with simple distributions, which is indeed the case in our experience. For this example, the authors model the kernel execution times using normal, gamma, and log-normal distributions. To test how appropriate these distributions are, we fitted the empirical distributions of completion times and found that they were, for all practical purposes, nearly identical for each model and the log-normal distribution has slightly outperformed the others in some cases. Figures 3 and 4 show the distributions for the DTSMQR and DGEMM kernels during an execution of the QR and Cholesky factorizations, respectively. These are the most computationally intensive kernels in each of their respective factorizations.

C. Task Execution Queue

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

In the case of simulated execution, the order in which these dependences are satisfied must be maintained in order for the simulations to be accurate. The key element of the simulation environment is the Task Execution Queue. This is the data structure that ensures that the tasks that are currently in the execution state (Note: a task in the execution state is not actually computing the function it simulates) within the simulation maintain the proper completion order. The queue is implemented as a priority queue which is prioritized by the simulated completion time of a task. In other words, a task may know the time of its own completion if it combines the information from the execution time distribution and the Task Execution Queue.

D. Simulation Task Function

In order to use the simulation library, the developer simply replaces the calls to each computational kernel with a call to the simulated kernel. This simulated kernel requires an identifier and an approximate execution time such as the distribution-based estimator as well as any handles or pointers that will create a dependence in the real simulation. Although the memory is never accessed, the actual memory location in the process's address space is required in order to ensure that all of the dependences will be maintained. Furthermore, some schedulers perform copies of the data to deal with anti-dependences and real memory locations are required for such copies to succeed. The simulated tasks are inserted into the task graph using the scheduler's API in an identical fashion to a real kernel.

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

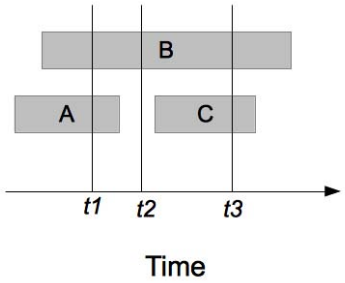


Fig. 5: A simple trace that demonstrates the race condition that can occur in the simulation.

E. Scheduling Race Condition

One of the challenging aspects of ensuring the correctness of the simulation stems from a race condition that can occur in some situations. The race condition can occur when a task is at the front of the Task Execution Queue while the scheduler is inserting new tasks. It is possible that the new tasks being inserted will not be in the front of the queue. While the newly inserted task might be the next to complete, the task previously in the queue may have already returned, which results in an inconsistency of the trace.

This situation may seriously affect the accuracy of the simulated trace and we would like to take the time to explain this in detail. The problem may be illustrated with a simple example shown in Figure 5. In this situation, we are assuming that there are only two cores and three tasks: A, B, and C. At time *t1*, task A is at the front of the priority queue, followed by task B. Once the simulated task A has completed all required bookkeeping, it will remove itself from the queue, update the clock, and return. After the function representing task A returns, the scheduler will recognize that task C is ready to be executed and schedule it on the open core. The race condition occurs when task B attempts to complete and task C attempts to start at time *t2*. If task C acquires the Task Execution Queue lock first, the task will be inserted in the queue correctly and the simulation will accurately describe what is occurring during a real execution. On the other hand, if task B acquires the lock first and returns before task C, the simulation will be inaccurate. When task B updates the simulation clock it will be updated to the completion time of task B. Task C will then estimate its completion time based on this updated simulation clock and be placed in the simulated trace much later than it would have been in reality.

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

The other solution to this problem that is portable for all schedulers is a judicious use of the `sleep()` function. This is used so that the simulated kernel will sleep for a fraction of a second and thus allow the scheduler to complete any bookkeeping. A further enhancement of this is a call to the kernel `sched_yield()` that will allow the scheduler thread to make the appropriate bookkeeping progress. This technique is portable even to non-POSIX systems because similar calls are uniformly present in a variety of modern OS's.

VI. Experimental Results

The simulation library has been tested with implementations of the Cholesky and QR factorizations – linear algebra algorithms described earlier. The algorithms were expressed in a tile-based fashion and were implemented using OmpSs, StarPU, and QUARK.

A. Traces

One way of determining the accuracy of a simulated trace is by direct comparison to a real trace. Figures 6 and 7 do just that. They represent a real trace and a simulated trace, respectively. The algorithm compared in the figures is the QR factorization performed using the QUARK scheduler. The code used a tile size of 180 and used all 48 cores of a quad-socket machine with four 12-core AMD Magny-Cours CPUs. The two traces are presented with identical time scales along the x-axis in order to show the nearly perfect correspondence of the two execution times. These two traces demonstrate how closely the simulation can model the execution of a real computation.

The traces are almost identical with two differences worth mentioning. The first difference is the length of time that the initial kernel operates on each core. In the real execution, the first kernel on each core is significantly longer than most of the remaining kernels of the same variety. The other difference is the number of tasks scheduled to run on the core 0. This is the core used to insert tasks and to maintain the dependence graph. It should also be noted that the scheduling decisions are non-deterministic which means the traces will not look exactly the same upon each execution.

B. Predicted Performance

One of our ultimate goals of creating a simulation library is for use in an autotuning framework. If it is possible to predict performance of an algorithm running on a particular scheduler configuration in a reduced time period, it will be possible to try a larger number of possible scheduling and algorithmic parameters that will allow for a more thorough tuning in a reduced amount of time with a potential for discovering parameter sets much closer to optimal, that would have otherwise been missed due to the much less thorough search through actual (and longer) runs on the hardware. With this use case in mind, a performance comparison to "real" algorithm executions is the desired benchmark to us, but is also a common requirement of most simulation frameworks.

Figures 8, 9, and 10 clearly show that the performance levels predicted by the simulations are accurate to within a few percentage points. Each scheduler was used to implement the QR (shown in blue) and Cholesky (shown in red) matrix factorization algorithms as described above. The solid lines represent the performance of the algorithm while performing the real algorithm. The dashed lines represent the performance predicted by the simulation of each algorithm and the dotted lines represent the percentage error of the simulation. The worst case error for any simulation with any simulator is approximately 16%, but the vast majority of test cases show less than 5% error. This serves as a clear testimony to the simulator's accuracy.

VII. Future Work

The current simulations only support the single threaded tasks and are thus missing the nested parallelism feature that

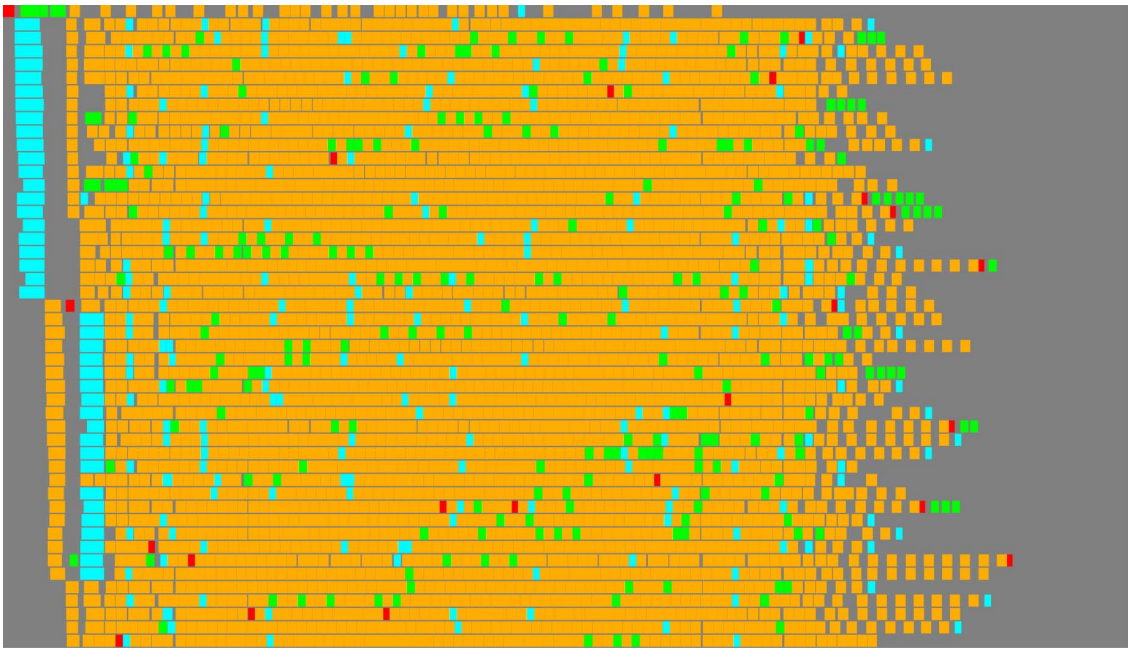


Fig. 6: A real trace of a QR factorization of a matrix. Matrix Size: 3960 Tile Size: 180 System: AMD Opteron 6180SE (4×12 Cores)

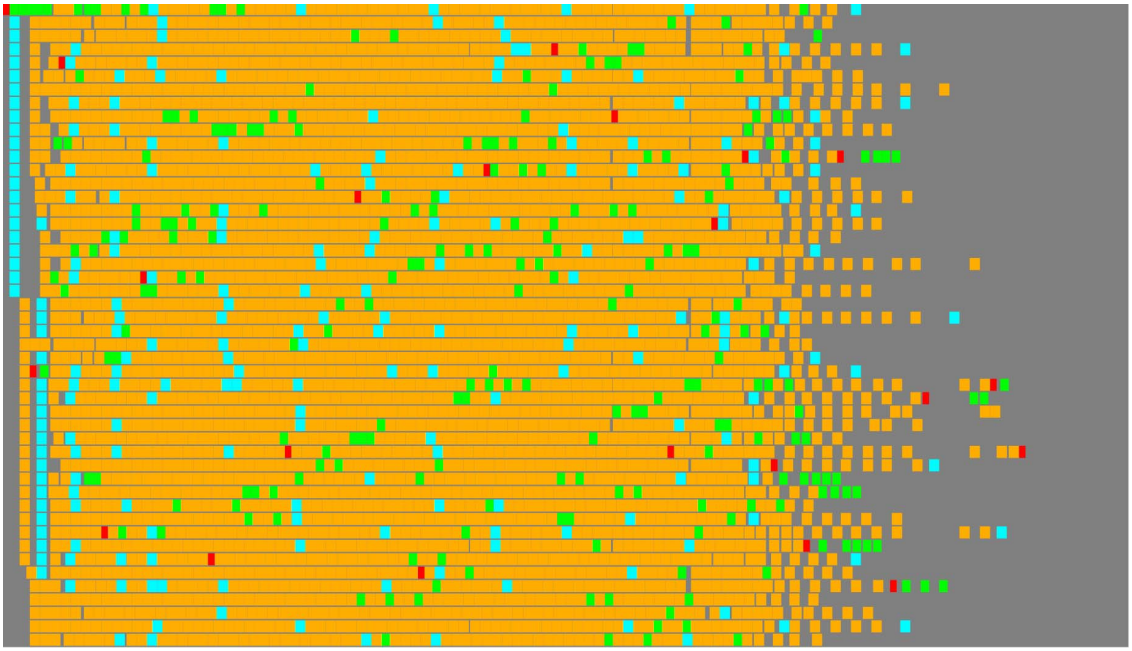


Fig. 7: A simulated trace of a QR factorization of a matrix. Matrix Size: 3960 Tile Size: 180 System: AMD Opteron 6180SE (4×12 Cores)

These two traces demonstrate how closely the simulation can model the execution of a real computation. The two traces are shown with the same timescale in order to show the total execution time of the algorithm is nearly identical.

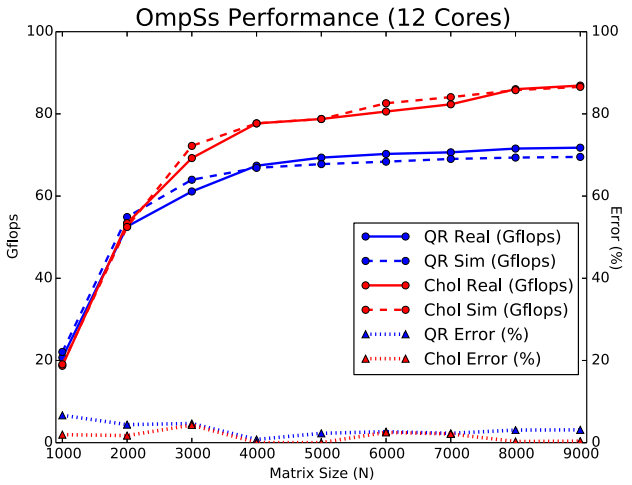


Fig. 8: A performance plot for QR (blue) and Cholesky (red). Each algorithm is implemented using OmpSs and compares the simulated (dashed) and true (solid) performance. The dotted lines represent the percentage error of the simulation for each factorization. Tile Size: 200

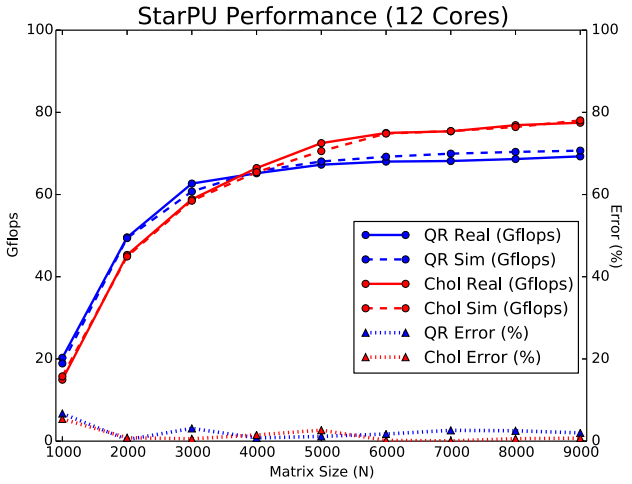


Fig. 9: A performance plot for QR (blue) and Cholesky (red) using StarPU. Each algorithm is implemented using StarPU and compares the simulated (dashed) and true (solid) performance. The dotted lines represent the percentage error of the simulation for each factorization. Tile Size: 200

is available through multi-threaded tasks in QUARK. Both QUARK and StarPU support GPU tasks and the simulations do not support those in the current implementation. Both of these extensions are worth pursuing. Each of the scheduling libraries provides a different way of specifying GPU tasks, which makes this a challenge to portability, and needs to be solved in a simulation environment which aims to operate with OmpSs, StarPU, and QUARK.

One of the keys to accurately simulating a workload is the ability to describe and model the time for a single kernel execution. The current simulation assumes that each kernel type can be described using simple distributions. This is a simplification of what actually occurs in most workloads. It may be possible to improve the accuracy of the simulations by improving that kernel model.

The data points that show the greatest error in Figures 8, 9, and 10 all occur for relatively small problem sizes. This is likely because any start-up performance penalties are a much

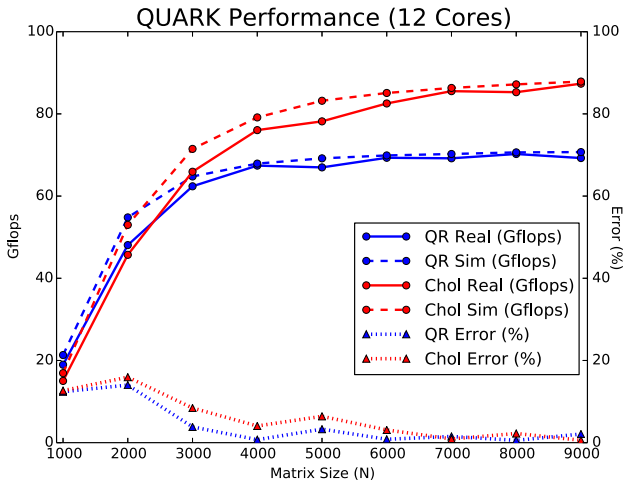


Fig. 10: A performance plot for QR (blue) and Cholesky (red) using QUARK. Each algorithm is implemented using QUARK and compares the simulated (dashed) and true (solid) performance. The dotted lines represent the percentage error of the simulation for each factorization. Tile Size: 200

larger portion of the total execution time. The simulator may be improved in the future in order to accurately model this start-up penalty and improve the simulation accuracy for small problem sizes.

VIII. Conclusion

We have presented a dynamic scheduling simulation library that can accurately simulate algorithms that use dynamic, superscalar schedulers. Accurate simulation results using OmpSs, StarPU, and QUARK have shown that the library operates with multiple schedulers and requires very little or no modification to the existing code base. The experimental results indicate that the library can provide useful insights into the performance and operation of the superscalar schedulers.

Acknowledgment

This research was supported by the National Science Foundation through Award #1339822.

References

[1] L. Dagum and R. Menon, "OpenMP: An Industry Standard API for Shared-Memory Programming," *Computational Science Engineering, IEEE*, vol. 5, no. 1, pp. 46 –55, 1998.

[2] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, "Cilk: An efficient multithreaded runtime system," *SIGPLAN Not.*, vol. 30, pp. 207–216, August 1995. [Online]. Available: http://doi.acm.org/10.1145/209937.209958

[3] L. G. Valiant, "A bridging model for parallel computation," *Communications of the ACM*, vol. 33, no. 8, Aug. 1990, dOI 10.1145/79173.79181.

[4] ——, "Bulk-synchronous parallel computers," in *Parallel Processing and Artificial Intelligence*, M. Reeve, Ed. John Wiley & Sons, 1989, pp. 15–22.

[5] R. G. Michael and D. S. Johnson, "Computers and Intractability : A guide to the theory of NP-completeness," *WH Freeman & Co., San Francisco*, 1979.

[6] Y.-K. Kwok and I. Ahmad, "Static Scheduling Algorithms for Allocating Directed Task Graphs to Multiprocessors," *ACM Comput. Surv.*, vol. 31, no. 4, pp. 406–471, Dec. 1999. [Online]. Available: http://doi.acm.org/10.1145/344588.344618

[7] E. Agullo, J. Dongarra, R. Nath, and S. Tomov, "A Fully Empirical Autotuned Dense QR Factorization for Multicore Architectures," in *Proceedings of the 17th International Conference on Parallel Processing - Volume Part II*, ser. Euro-Par'11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 194–205. [Online]. Available: http://dl.acm.org/citation.cfm?id=2033408.2033430

[8] R. Vuduc, J. W. Demmel, and J. A. Bilmes, "Statistical Models for Empirical Search-Based Performance Tuning," *Int. J. High Perform. Comput. Appl.*, vol. 18, no. 1, pp. 65–94, Feb. 2004. [Online]. Available: http://dx.doi.org/10.1177/1094342004041293

[9] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, pp. 1–7, Aug. 2011. [Online]. Available: http://doi.acm.org/10.1145/2024716.2024718

[10] J. Renau, B. Fraguela, J. Tuck, W. Liu, M. Prvulovic, L. Ceze, S. Sarangi, P. Sack, K. Strauss, and P. Montesinos, "SESC simulator," January 2005, http://sesc.sourceforge.net.

[11] H. Casanova, A. Legrand, and M. Quinson, "Simgrid: a generic framework for large-scale distributed experiments," in *Proceedings of the Tenth International Conference on Computer Modeling and Simulation*, ser. UKSIM '08. Washington, DC, USA: IEEE Computer Society, 2008, pp. 126–131. [Online]. Available: http://dx.doi.org/10.1109/UKSIM.2008.28

[12] R. Buyya and M. Murshed, "Gridsim: A toolkit for the modeling and simulation of distributed resource management and scheduling for grid computing," *CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE (CCPE*, vol. 14, no. 13, pp. 1175–1220, 2002.

[13] K. Ranganathan and I. Foster, "Decoupling computation and data scheduling in distributed data-intensive applications," in *High Performance Distributed Computing, 2002. HPDC-11 2002. Proceedings. 11th IEEE International Symposium on*, 2002, pp. 352–358.

[14] W. H. Bell, D. G. Cameron, A. P. Millar, L. Capozza, K. Stockinger, and F. Zini, "Optorsim: A grid simulator for studying dynamic data replication strategies," *International Journal of High Performance Computing Applications*, vol. 17, no. 4, pp. 403–416, 2003. [Online]. Available: http://hpc.sagepub.com/content/17/4/403.abstract

[15] R. M. Badia, J. Labarta, R. Sirvent, J. M. Perez, J. M. Cela, and R. Grima, "Programming grid applications with GRID Superscalar," *J. Grid Comput.*, vol. 1, no. 2, pp. 151–170, 2003, .

[16] J. M. Perez, P. Bellens, R. M. Badia, and J. Labarta, "CellSs: Making it easier to program the Cell Broadband Engine processor," *IBM J. Res. & Dev.*, vol. 51, no. 5, pp. 593–604, 2007, .

[17] J. M. Pérez, R. M. Badia, and J. Labarta, "A dependency-aware task-based programming environment for multi-core architectures," in *Proceedings of the 2008 IEEE International Conference on Cluster Computing, 29 September - 1 October 2008, Tsukuba, Japan*. IEEE, 2008, pp. 142–151.

[18] R. M. Badia, J. R. Herrero, J. Labarta, J. M. Perez, E. S. Quintana-Orti, and G. Quintana-Orti, "Parallelizing dense and banded linear algebra libraries using SMPSs," *Concurrency Computat. Pract. Exper.*, vol. 21, no. 18, pp. 2438–2456, 2009, .

[19] E. Ayguadé, R. M. Badia, F. D. Igual, J. Labarta, R. Mayo, and E. S. Quintana-Ortí, "An Extension of the StarSs Programming Model for Platforms with Multiple GPUs," in *Proceedings of the 15th International Euro-Par Conference on Parallel Processing*. Springer-Verlag, 2009, pp. 851–862.

[20] A. Duran, E. Ayguade, R. Badia, J. Labarta, L. Martinell, X. Martorell, and J. Planas, "OmpSs: A proposal for programming heterogeneous multi-core architectures," *Parallel Process. Lett.*, vol. 21, no. 2, pp. 173–193, 2011, .

[21] J. Planas, R. M. Badia, E. Ayguadé, and J. Labarta, "Hierarchical task-based programming with StarSs," *Int. J. High Perf. Comput. Applic.*, vol. 23, no. 3, pp. 284–299, 2009, .

[22] C. Augonnet and R. Namyst, "A unified runtime system for heterogeneous multicore architectures," in *Proceedings of the Euro-Par 2008 Workshops - Parallel Processing*, ser. Lecture Notes in Computer Science. Las Palmas de Gran Canaria, Spain: Springer, August 2008, pp. 174–183, .

[23] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier, "StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures," in *Proceedings of the 15th International Euro-Par Conference on Parallel Processing*, ser. Euro-Par '09. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 863–874. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-03869-3_80

[24] C. Augonnet, S. Thibault, R. Namyst, and P. Wacrenier, "StarPU: a unified platform for task scheduling on heterogeneous multicore architectures," *Concurrency Computat. Pract. Exper.*, vol. 23, no. 2, pp. 187–198, 2011, .

[25] E. Agullo, J. Dongarra, B. Hadri, J. Kurzak, J. Langou, J. Langou, H. Ltaief, P. Luszczek, and A. YarKhan, "PLASMA Users Guide," University of Tennessee, Innovative Computing Laboratory, Tech. Rep., 2010.

[26] A. YarKhan, J. Kurzak, and J. Dongarra, "QUARK Users' Guide: QUeueing And Runtime for Kernels," Innovative Computing Laboratory, University of Tennessee, Tech. Rep., 2011.

[27] J. Kurzak, P. Luszczek, M. Faverge, and J. Dongarra, "LU factorization with partial pivoting for a multicore system with accelerators," *IEEE Trans. Parallel Distrib. Syst.*, 2012, .

[28] A. YarKhan, "Dynamic task execution on shared and distributed memory architectures," Ph.D. dissertation, University of Tennessee, December 2012.

[29] A. Buttari, J. Dongarra, J. Kurzak, J. Langou, P. Luszczek, and S. Tomov, "The Impact of Multicore on Math Software," in *Applied Parallel Computing. State of the Art in Scientific Computing*, ser. Lecture Notes in Computer Science, B. Kågström, E. Elmroth, J. Dongarra, and J. Wasniewski, Eds. Springer Berlin / Heidelberg, 2007, vol. 4699, pp. 1–10. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-75755-9_1

[30] E. S. Quintana-Ortí and R. A. Van De Geijn, "Updating an LU Factorization with Pivoting," *ACM Trans. Math. Softw.*, vol. 35, no. 2, pp. 11:1–11:16, Jul. 2008. [Online]. Available: http://doi.acm.org/10.1145/1377612.1377615

[31] A. Buttari, J. Langou, J. Kurzak, and J. Dongarra, "Parallel tiled QR factorization for multicore architectures," *Concurrency and Computatation: Practice and Experience*, vol. 20, no. 13, pp. 1573–1590, 2008.

[32] J. Kurzak, A. Buttari, and J. Dongarra, "Solving Systems of Linear Equations on the CELL Processor Using Cholesky Factorization," *IEEE Trans. Parallel Distrib. Syst.*, vol. 19, pp. 1175–1186, September 2008. [Online]. Available: http://portal.acm.org/citation.cfm?id=1444382.1444414

[33] J. Kurzak and J. Dongarra, "QR factorization for the Cell Broadband Engine," *Scientific Programming*, vol. 17, no. 1, pp. 31–42, 2009.

[34] B. C. Gunter and R. A. van de Geijn, "Parallel out-of-core computation and updating the QR factorization," *ACM Transactions on Mathematical Software*, vol. 31, no. 1, pp. 60–78, 2005.

[35] A. Buttari, J. Langou, J. Kurzak, and J. Dongarra, "A class of parallel tiled linear algebra algorithms for multicore architectures," *Parallel Comput.*, vol. 35, no. 1, pp. 38–53, Jan. 2009. [Online]. Available: http://dx.doi.org/10.1016/j.parco.2008.10.002

[36] R. C. Whaley and A. M. Castaldo, "Achieving accurate and context-sensitive timing for code optimization," *Softw. Pract. Exper.*, vol. 38, no. 15, pp. 1621–1642, Dec. 2008. [Online]. Available: http://dx.doi.org/10.1002/spe.v38:15