# Runtime Data Flow Scheduling of Matrix Computations

FLAME Working Note #39

Ernie Chan
Department of Computer Sciences
The University of Texas at Austin
Austin, Texas 78712
echan@cs.utexas.edu

## ABSTRACT

We investigate the scheduling of matrix computations expressed as directed acyclic graphs for shared-memory parallelism. Because of the data granularity in this problem domain, even slight variations in load balance or data locality can greatly affect performance. Well-known scheduling algorithms such as work stealing have proven time and space bounds, but these bounds do not provide a discernable indicator of performance between different scheduling algorithms and heuristics. We provide a flexible framework for scheduling matrix computations, which we use to empirically quantify different scheduling algorithms. By building software solutions based on hardware techniques through leveraging a cache coherence protocol, we develop a scheduling algorithm that addresses both load balance and data locality simultaneously and show its performance benefits.

## Categories and Subject Descriptors

D.1.3 [**Software**]: Concurrent Programming

## General Terms

Algorithms, Performance

## Keywords

algorithm-by-blocks, cache coherence protocol, directed acyclic graph, queueing theory

## 1. INTRODUCTION

With the emergence of multicore architectures, exploiting parallelism has become paramount for the performance of computationally intensive applications. The scheduling literature contains a wide range of papers and books, but many of those scheduling algorithms and heuristics can only be applied statically [26, 37, 41] or mainly deal with load balance without considering data communication [11, 20]. Work stealing is a well-known scheduling algorithm that has the proven time bound of $O(T/p + D)$ where $T$ is the total amount of work, $p$ is the number of threads, and $D$ is the critical path length of the computation [12]. Here we deal strictly with the scheduling of matrix computations on shared-memory architectures, and that bound does not provide any practical insight about the potential performance of work stealing compared to other scheduling algorithms. In this paper, we present an analysis of different scheduling algorithms and leverage the useful aspects of each to develop a new scheduling algorithm for matrix computations.

In our previous papers, we have primarily focused on programmability by leveraging abstractions from the Formal Linear Algebra Method Environment (FLAME) project [7, 8, 30, 40]. These abstractions allow us to create the separation of concerns that completely hides the exploitation of parallelism from the code that implements the linear algebra algorithms. By doing so, we developed a clean solution for parallelizing matrix computations, once thought by many to be a difficult and daunting problem. In those papers, we focused this methodology on parallelizing Linear Algebra PACKage (LAPACK) [4] level operations such as the LU and QR factorizations. We used relatively simple scheduling algorithms and did not address the many complex scheduling issues presented by this problem domain yet were still able to demonstrate impressive performance [17, 19, 47].

In order to attain high performance, it is well understood that matrix algorithms must be cast in terms of blocked computations so that the bulk of the computation is in matrix-matrix multiplication [28]. By storing matrices hierarchically [23] and viewing submatrix blocks as the unit of data and operations with blocks (tasks) as the unit of computation, we introduced the concept of *algorithms-by-blocks*. We link to optimized Basic Linear Algebra Subprograms (BLAS) [21, 22, 38] libraries for the execution of individual tasks. We describe the process to map matrix computations to algorithms-by-blocks and different matrix storage schemes in [17, 47].

The key abstraction for exploiting parallelism lies with mapping an algorithm-by-blocks to a directed acyclic graph (DAG) where tasks represent the nodes of the graph and data dependencies (flow, anti, and output) between tasks represent the edges. See [19] on how to detect all three types of data dependencies within linear algebra algorithms.

We developed the SuperMatrix runtime system through a clear separation of concerns where we divide the process of

exploiting parallelism into two phases: *analyzer* and *dispatcher*. The analyzer phase constructs a DAG by storing tasks while sequentially stepping through the execution of an algorithm-by-blocks alongside detecting data dependencies between all tasks. Only the input and output parameters of each task are needed to detect dependencies. Once the analyzer is done, the dispatcher phase is invoked which dispatches and schedules tasks to threads. We focus on this second phase and the ramifications of scheduling in this paper.

The current paper brings the following contributions to the forefront:

- A description of the algorithm that dispatches tasks to threads.
- Discussions of queueing theory and its application to the problem domain of scheduling matrix computations.
- A review and analysis of dynamic scheduling algorithms, which show that solely focusing on load balancing is insufficient for attaining the best performance.
- A new scheduling algorithm that addresses both data locality and load balancing simultaneously by using hardware techniques in software.

Together these contributions show that scheduling should incorporate information about the underlying architecture as opposed to being oblivious to it, all done without adding exorbitant complexity to the runtime system.

The rest of the paper is organized as follows. In Section 2 we describe the process for dispatching tasks to threads in SuperMatrix. We discuss queueing theory in Section 3. Different scheduling algorithms are described in Section 4. We introduce a new scheduling algorithm in Section 5 for which we provide performance results in Section 6. We provide related work in Section 7 and conclude the paper in Section 8 where we also discuss future work.

# 2. SUPERMATRIX RUNTIME SYSTEM

In this section, we focus on the dispatcher phase of the SuperMatrix runtime system. We use the Cholesky factorization as a motivating example to illustrate the general algorithm for dispatching tasks to threads. Despite using this operation in several of our previous papers, it continues to be highly representative of other linear algebra operations and is an excellent vehicle for describing the scheduling mechanisms in this paper.

## 2.1 Cholesky factorization

The Cholesky factorization of a symmetric positive definite matrix $A \in \mathbb{R}^{n \times n}$ is given by $A \rightarrow LL^T$ where $L$ is lower triangular.

The blocked right-looking algorithm for computing the Cholesky factorization consists of a recursive subproblem (CHOL), followed by a block-panel triangular solve with multiple right-hand sides (TRSM), then a panel-panel symmetric rank-k update (SYRK), and finally the recursive invocation. This recursive algorithm is typically reformulated as a loop-based algorithm where the recursive subproblem is implemented via an unblocked algorithm. We can then take the iterative algorithm and convert it to an algorithm-by-blocks. See [19] for further details about this operation and its different algorithmic variants.

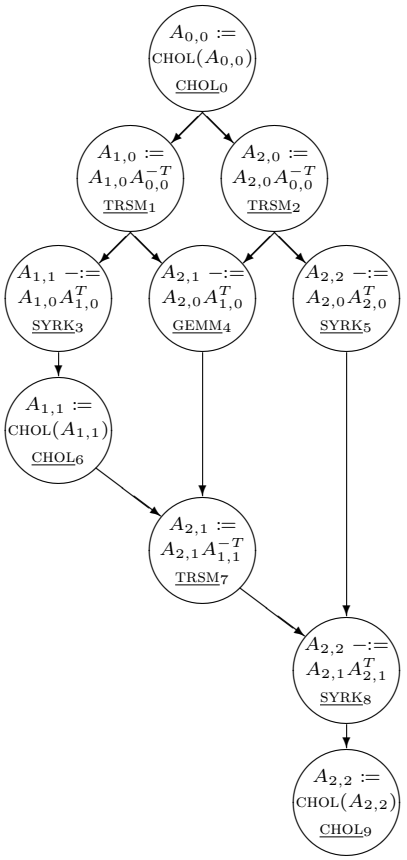



**Figure 1: The directed acyclic graph for the Cholesky factorization on a $3 \times 3$ matrix of blocks where $A -:= B$ represents $A := A - B$.**

In Figure 1, we illustrate the DAG for the Cholesky factorization on a $3 \times 3$ matrix of blocks with each task's input and output parameters. The names of each task are underlined where the subscript denotes an order in which a task can be sequentially executed by the algorithm-by-blocks.

In the DAG, a CHOL task overwrites a block that is later read by a TRSM task, which leads to flow dependencies from the CHOL to the TRSM tasks. For instance, CHOL$_0$ overwrites $A_{0,0}$ while both TRSM$_1$ and TRSM$_2$ read it. Similar flow dependencies occur between TRSM and SYRK. General matrix-matrix multiplication (GEMM) resides in the DAG since it is a subproblem of the panel-panel SYRK. The recursive invocation for the Cholesky factorization accesses the submatrix, which may consist of a conglomeration of many blocks, that is overwritten by the panel-panel SYRK. This fact leads to the remaining dependencies in the DAG where for instance SYRK$_3$ and CHOL$_6$ both overwrite $A_{1,1}$.

```
foreach task in DAG do
    if task is ready then
        Enqueue task
    end
end
while tasks are available do
    Dequeue task
    Execute task
    foreach dependent task do
        Update dependent task
        if dependent task is ready then
            Enqueue dependent task
        end
    end
end
```

**Figure 2: The algorithm that all threads execute in order to dispatch and schedule tasks from a directed acyclic graph.**

## 2.2 Dispatcher

Once the DAG has been fully constructed, the dispatcher phase is invoked where the threads are spawned, tasks are scheduled and dispatched to threads, and finally the threads are joined back together once all tasks have been executed. Figure 2 presents the algorithm that every thread performs in order to dispatch tasks. For more details on the interface and implementation of the SuperMatrix runtime system, see [18].

If there is a directed edge from node $t_i$ to node $t_j$, then $t_i$ is the **parent** of $t_j$, and $t_j$ is the **child** of $t_i$. A **root** is a node with no parents. We define a **ready** task as one that is either a root or all of its parents have been executed.[1] A task is **available** if it is ready and waiting to be executed. A **dependent** task is simply the child of a certain task. We **update** a dependent task by notifying it that one of its parents has been executed. The *enqueue* and *dequeue* routines perform the scheduling of tasks, which we will discuss further in Section 4.

For example in Figure 1, $\text{CHOL}_0$ is the only root in the DAG and thus is enqueued as an initial ready task. A particular thread will dequeue that task, execute it, and then update its dependent tasks $\text{TRSM}_1$ and $\text{TRSM}_2$, both of which will then become ready.

The need for mutual exclusion arises only at four modularized locations within the dispatcher. The first two occur when invoking the enqueue and dequeue routines. The third involves updating each dependent task, and the final one comes when checking the terminating condition of the loop to see if any ready and available tasks remain.

## 2.3 Proof of correctness

When an algorithm-by-blocks is mapped to DAG, directed edges exist because of the nature where one task produces a value and a subsequent task consumes that value. The graph is acyclic since a sequential execution of tasks exists where a task cannot depend upon a task that occurs after it does, and hence there cannot exist any loops. A tree is not formed since a task can depend on more than one task.

The algorithm in Figure 2 is guaranteed to terminate given an arbitrary DAG. A simple proof by contradiction can be used to validate this fact. Assume that there exists at least one task that is not executed. That extraneous task must have at least one un-executed parent; otherwise it would be ready and available. This one un-executed parent can only exist if inductively one of its parents also has not been executed. Since we have a finite set of tasks in the DAG, there must exist an un-executed task that is a root. If no such task exists, then there exists a loop in the graph, and this fact breaks the constraint of no longer having an acyclic graph. If there does exist a root that has not been executed, then this situation leads to a contradiction since that task would be ready and subsequently executed by the dispatcher.

# 3. QUEUEING THEORY

In this section, we give an overview of classic definitions and results from queueing theory. We apply queueing theory to the scheduling of matrix computations mapped to a DAG in order to better help us analyze different scheduling algorithms in Section 5.3 instead of attempting to provide asymptotic time and space bounds.

Queueing theory was first developed to analyze communication networks [24], and it is the study of waiting lines where **customers** from a **source** must wait for a **service** [29]. In this problem domain, customers are tasks, and the source is a DAG, and a service is execution of those tasks on a processing element by a thread.

We use Kendall's notation [33] for describing queueing systems:

$$A/B/c/K/m/Z.$$

$A$ is the arrival distribution, and $B$ is the service time distribution. $c$ is the number of the servers in the system. $K$ is the maximum capacity of a queue, and $m$ is the number of customers in the source. $Z$ is the the queue discipline.

We assume that the arrival and service time distributions are both exponential, which is denoted with $M$. For instance the exponential distribution for arrival times can be described as

$$P[\tau \leq t] = 1 - e^{-\lambda t}$$

where $\tau$ is the interarrival time of the next customer to the system and $\lambda$ is the arrival rate of customers at some time $t$ [3]. There is also a general distribution $G$ where no assumptions are made about the distribution. The number of servers equals the number of threads $p$. Kendall's notation can be abbreviated to the first three categories where typically the maximum capacity and the size of the source are both assumed to be infinite and the queue discipline is first-in first-out (FIFO). For instance an $M/M/p$ system represents $M/M/p/\infty/\infty/FIFO$.

In Table 1 we present some common symbols used in queueing theory. Little's formula [39] for the expected number of customers in the queueing system in a **steady state** is

$$L = \lambda W. \tag{1}$$

[1]We use the words task and node interchangeably.

| Queueing Theory Definitions | |
|---|---|
| $L$ | Expected number of customers in the system |
| $\lambda$ | Average customer arrival rate to the system |
| $W$ | Expected time a customer spends in the system |
| $W_q$ | Expected time a customer spends in a queue |
| $W_s$ | Expected customer service time |
| $\mu$ | Average customer service rate per server |
| $\rho$ | Server utilization |

**Table 1: A list of some basic queueing theory definitions where the expected values result from a steady state of the queueing system.**

A queueing system reaches a steady state after a sufficient number of customers have arrived to fill the queues, and thus the expected number of customers in the system is independent of the time elapsed. The expected time of customers spent in the system is

$$W = W_q + W_s \tag{2}$$

which equals the waiting time in a queue plus the time for servicing. The average service rate per server is

$$\mu = \frac{1}{W_s} \tag{3}$$

which is the rate customers are processed when a server is busy. Server utilization is then

$$\rho = \frac{\lambda}{c\mu} \tag{4}$$

which represents the probability any server in the system is busy. We will assume that $\rho < 1$ in order for the queueing system to reach a steady state.

Whenever a queueing system decreases its expected waiting time or increases server utilization, we equate that to an increase in load balance. We can also model a decrease in data communication by a decrease in the expected service time or inversely as an increase in the average service rate.

## 3.1 Single vs. multiple queues

In Figure 3, we depict two multi-server queueing systems. A single-queue multi-server system is in the left where all servers access one queue, which represents an $M/M/p$ system. A multi-queue multi-server system is in the right where each server has its own dedicated queue, which represents a system with $p$ separate $M/M/1$ queues.

It is quite intuitive that servicing multiple servers using a single queue is more efficient than doing so from separate queues. There is nonzero probability that a customer will be waiting for a service in one of the $p$ independent queues while another server is idle. On the other hand, that situation cannot occur in an $M/M/p$ queue since a customer at the head of the queue is serviced as soon as a server is available. As an example, grocery stores use independent $M/M/1$ queues to service each cashier in order to delay the time shoppers spend waiting to get checked out, which allows the stores to entice shoppers with last minute candy and magazine purchases. Banks use an $M/M/p$ queue where a single line services $p$ tellers to reduce the time customers spend waiting in line.

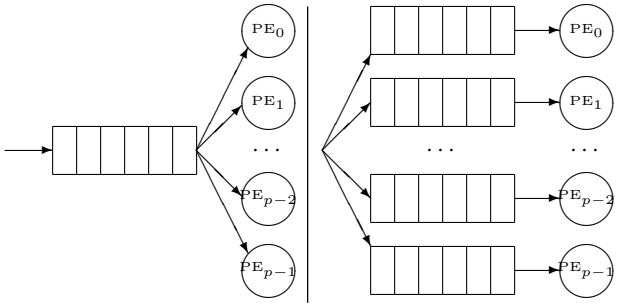



**Figure 3: Depictions of a single-queue multi-server system (left) and a multi-queue multi-server system (right) each with $p$ processing elements.**

This fact was proven in [51]. If we denote the expected waiting time within a queue of an $M/M/p$ queue as $W_q^1$ and $p$-$M/M/1$ queues as $W_q^p$, then the authors showed that

$$W_q^1 \leq W_q^p$$

where $W_s^1 = W_s^p$ and $\lambda^1 = \lambda^p$. One of their methods of proof does not require the assumption of an exponential distribution for either the arrival or service times, so this analysis can apply to $G/G/p$ and $p$-$G/G/1$ queueing systems.

## 3.2 Conservation law of priority queues

Many queue disciplines exist besides FIFO such as last-in first-out (LIFO), random selection for service (RSS), or priority service (PRI). A priority service is one where customers are assigned priorities, and the queue is sorted according to those priorities. Ties can be broken between customers with the same priority according to a FIFO order. We will deal with **nonpreemptive** priority queues where a customer will complete its service even if a customer with a higher priority arrives.

If we denote the expected waiting time of customers in a system with a FIFO queue discipline as $W^{FIFO}$ and one with priority service as $W^{PRI}$, then the conservation law states that

$$W^{FIFO} = W^{PRI}.$$

The conservation law applies to any queue discipline as long as the priority is not dependent upon service times and servers are not left intentionally idle while customers wait. This law was proven in [34, 35] for $M/G/1$ queues and then extended to $G/G/1$ queues in [49] and finally to $G/G/p$ queues in [6]. The conservation law is also quite intuitive since it is an example of a zero-sum game. The time a customer saves waiting in the queue by getting serviced earlier results in another customer having to wait longer. While the expected waiting time will stay the same, its distribution will change to have a wider variance when using PRI instead of FIFO queue discipline. The arrival and service time distributions are not affected by the varying use of queue disciplines.

## 3.3 Application of queueing theory to matrix computations

A single queue implementation requires the need for mutual exclusion when enqueueing and dequeueing from that shared

resource. This bottleneck can potentially limit the speedup of parallel applications through serializing the access to that single queue. This overhead is nearly negligible in this problem domain because the data granularity of each task greatly outweighs the cost of acquiring mutual exclusion. Each task is composed of matrix computations which perform $O(n^3)$ operations on $O(n^2)$ data where $n$ is the matrix dimension.

In order to quantify the scheduling of matrix computations to queueing theory, we need to make several simplifying assumptions. The arrival times of tasks do not quite follow an exponential distribution since the enqueueing of tasks is dependent on the structure of the source DAG and is only done when a task's parent tasks have finished execution. Since threads execute and enqueue tasks simultaneously once reaching a steady state, we make the simplifying assumption of an exponential arrival time distribution.

As we will show in Section 5, the service times of each task do not follow a Markovian property where each is "memoryless" of the ones executed before itself. For the time being, we will assume that we can model the execution times of each task using its floating point operation count, which do not depend upon the state of tasks executed previously. We store hierarchical matrices where every submatrix block has the same block size $b$ only with the exception of the edge cases. As a result each task operates on blocks of equal size, so the computational costs of each task have the same order of magnitude $O(b^3)$. We generally chose a block size so that the matrix operands of a task roughly fill intermediate levels of cache on a processing element.

We know that the maximum queue capacity and the size of the source is $K$ where $K$ is now defined as the number of tasks in a DAG. We then have $M/M/p/K/K$ and $p$-$M/M/1/K/K$ queues where each have a FIFO or PRI queue discipline. In the $p$-$M/M/1$ queue system, each queue is bounded to $K$ because of the case where all tasks are assigned to one particular queue. The analysis of a queueing system can be simplified by assuming infinite queue capacity and source size.

Through these simplifying assumptions, we can apply the analysis of these multi-server systems for scheduling matrix computations. Note that both the single versus multiple queue and priority queue analyses can be applied to systems with general arrival and service time distributions. As such, the use of a single queue implementation results in better load balancing since it reduces the time tasks spend waiting to be executed by a thread and thus increases the server utilization where the threads are kept busy executing more frequently.

# 4. SCHEDULING ALGORITHMS

How threads enqueue and dequeue tasks within the dispatcher determines the scheduling of those tasks. We present three scheduling algorithms and heuristics for priority queues, which can applied each of those three.

## 4.1 Single FIFO queue

Here all tasks are enqueued at the tail of a single queue, and all threads dequeue tasks from the head, which results in a FIFO order. Whenever tasks are ready and available, an idle thread will immediately dequeue a task to execute as soon as it gains mutual exclusion to the global queue. Because of this property, this scheduling algorithm reduces the amount of time each thread does not perform useful computation.

## 4.2 Data affinity

We introduced the idea of data affinity in [17] where tasks are assigned to threads according to a simple owner computes rule. A thread executes all tasks that overwrite a particular block. One task to thread assignment is a two-dimensional block cyclic (2D) distribution where blocks are mapped to a mesh of threads according to its row and column indices within the matrix of blocks. This 2D assignment was inspired by its previous use on distributed-memory architectures in libraries such as ScaLAPACK [9]. Another example is a round-robin assignment of blocks to threads.

Each thread has its own associated FIFO queue from which it dequeues. A thread will need to enqueue ready dependent tasks to the particular queue that task is assigned, which might not be itself.

Data affinity attempts to optimize for data locality. By only allowing one thread to overwrite a particular block, we try to restrict the total number of threads that access a block. If thread affinity is used where a thread is bound to a certain processing element by the operating system, then we reduce data communication through this conceptually simple scheduling algorithm.

## 4.3 Work stealing

Work stealing also has separate FIFO queues for each thread. All initial ready tasks are assigned to a particular thread. As tasks execute, all of the ready dependent tasks are enqueued to the same thread that executes their parent. If a task has multiple parents, then the thread that performs the last update to enable that dependent task to become ready will enqueue the task onto its own queue. If an idle thread's associated queue is empty, it will attempt to *steal* a task from the tail of a randomly selected thread's queue until one is found or all tasks are executed [12].

A work stealing optimization for data locality was addressed in [1] where each thread has an associated mailbox along with its queue. A thread will enqueue a ready dependent task onto its own queue as usual but will also place it in the task's assigned mailbox. Before a thread attempts to steal, it will first check its mailbox to see if there are any ready and available tasks. Again, a 2D distribution can be used to assign tasks to threads' mailboxes.

## 4.4 Priority queue heuristics

The above three scheduling algorithms all use FIFO queues where the scheduling of tasks is dependent upon the order in which the tasks are stored. Instead, each of the three can use priority queues where tasks are sorted according to a certain heuristic.

**Incomparable** nodes in a DAG are a set of nodes for which there is not a directed **path** between any pair of nodes where a path is a sequence of edges connecting two nodes. As such, incomparable nodes can potentially be executed in parallel.

| Scheduling | Queues | Queueing System |
|---|---|---|
| Single FIFO queue | 1 | $M/M/p$ |
| Data affinity | $p$ | $p$-$M/M/1$ |
| Work stealing | $p$ | $p$-$M/M/1$ |
| Cache affinity | 1 | $M/M/p/\infty/\infty/PRI$ |

**Table 2: A list of the different scheduling algorithms and the number of queues and the type of queueing system used by each where $p$ is the number of threads.**

In Figure 1, $\text{SYRK}_5$ and $\text{CHOL}_6$ are incomparable. When given the choice between scheduling those two tasks, $\text{CHOL}_6$ could be selected first because it has three **descendants** versus just two of $\text{SYRK}_5$. A node $t_j$ is a descendant of node $t_i$ if there is a directed path from $t_i$ to $t_j$. This example demonstrates the use of sorting tasks according to the **height** of a node where the height is the longest path from a node to a **leaf**. A leaf is a node with no children, and $\text{CHOL}_9$ is the only leaf in this DAG. This heuristic favors tasks on the critical path of execution, the height of a root in the DAG, which in this example is seven where $\text{CHOL}_0 \rightarrow \text{TRSM}_1 \rightarrow \text{SYRK}_3 \rightarrow \text{CHOL}_6 \rightarrow \text{TRSM}_7 \rightarrow \text{SYRK}_8 \rightarrow \text{CHOL}_9$ must be executed in order.

Another heuristic is the number of children of a task. For instance, $\text{TRSM}_2$ would be selected before $\text{SYRK}_3$ in this case despite both tasks having the same height. The total number of descendants of a task is another heuristic. We can also augment the different heuristics by using a weighted sum according to the floating point operation count of each task. These heuristics are all dependent upon the structure of the DAG.

## 4.5 Summary

We provide an overview of these different scheduling algorithms along with the number of queues required for each and the type of queueing systems each represents in Table 2 from which we will discuss cache affinity in Section 5. Work stealing is akin to a multi-queue multi-server queueing system with **jockeying** where tasks can switch queues to ones with shorter waiting lines.

A single queue implementation reduces the time threads are idle despite introducing a bottleneck through the need to gain mutual exclusion in order access the global queue. Scheduling with data affinity is highly dependent upon the task to thread assignment which may create wide load inbalances, yet the cost of enqueuing and dequeuing from queues dedicated to each thread is much less expensive. Work stealing also optimizes for load balancing without the global bottleneck, yet it introduces an extra level of randomness into the scheduling.

We do not provide results for work stealing and data affinity in Section 6 because we show in [18] that work stealing and a single FIFO queue attain the same performance signature since both attempt to only optimize for load balance and that 2D data affinity usually under-performs compared to the others because of its poor load balancing properties. We also explain that the mailbox optimization for work stealing does not provide a significant performance increase since the number of steals incurred in this problem domain typically is quite low compared to the total number of tasks.

# 5. CACHE AFFINITY

In Section 4, we presented several different scheduling algorithms that attempt to optimize for either load balance or data locality. In this section, we describe and analyze a new scheduling algorithm that strives to balance these two aspects simultaneously.

The performance of a single FIFO queue and 2D data affinity are roughly the same when executing the Cholesky factorization with a $5000 \times 5000$ matrix on the architecture described in Section 6.1. After instrumenting the code and gathering the beginning and ending times of each task, we found that each thread was idle for almost 25% of the parallel execution time when using 2D data affinity while the single FIFO queue incurred idle threads only 6% of the time. A single queue provides better load balance than multiple queues as explained in Section 3, which is highlighted by this example, yet the explanation for the similar performance signatures despite this discrepancy lies with the reduction in data communication.

## 5.1 A brief overview of a cache coherence protocol

We will assume a write-once cache coherence protocol is used to provide the shared-memory parallelism abstraction [5, 27].

A cache line can be one of four states. An **invalid** cache line is one that has been invalidated by updates to the same data on another cache, so the data is incoherent. It is **valid** if it has been read into the cache and is not modified. A cache line is **reserved** if it is the only copy of the data in any cache. It is **dirty** if the cache line has been modified, but the data has not been written back to main memory.

When a thread accesses a cache line, only one of four occurrences can happen. On a **read hit**, the data is in the cache and no state change occurs. A thread reads from main memory for a **read miss**, but if there is a dirty copy in any other cache, it first must be written back. Any dirty or reserved cache lines will then become valid ones. On a **write hit**, dirty or reserved cache lines are modified and set to dirty. Valid cache lines become reserved, and then the modified data is immediately written back to main memory, and all other copies will become invalid. A **write miss** invokes the same state changes as a read miss followed by a write hit.

## 5.2 Scheduling with software caches

Data affinity performs particurlarly well because there is a high likelihood that there is a cache hit for the blocks updated by each task. We want to replicate this behavior without needing the restriction of using multiple queues. In order to do so, we maintain a software cache for each thread that approximates the contents of the physical cache for the associated processing element to which a thread is mapped. The data granularity in this problem domain allows us to amortize the cost of managing these software caches quite efficiently.

| Stage | Thread 1 | Thread 2 |
|---|---|---|
| 1 | $\text{CHOL}_0$ | |
| | $(A_{0,0})$ | |
| 2 | $\text{TRSM}_1$ | $\text{TRSM}_2$ |
| | $(A_{1,0}, A_{0,0})$ | $(A_{2,0}, A_{0,0})$ |
| 3 | $\text{SYRK}_3$ | $\text{GEMM}_4$ |
| | $(A_{1,1}, A_{1,0}, A_{0,0})$ | $(A_{2,1}, A_{2,0}, A_{1,0}, A_{0,0})$ |
| 4 | $\text{SYRK}_5$ | $\text{CHOL}_6$ |
| | $(A_{2,2}, A_{2,0}, A_{1,0})$ | $(A_{1,1}, A_{2,1}, A_{2,0}, A_{1,0})$ |
| 5 | $\text{TRSM}_7$ | |
| | $(A_{2,1}, A_{1,1}, A_{2,2}, A_{2,0})$ | |
| 6 | $\text{SYRK}_8$ | |
| | $(A_{2,2}, A_{2,1}, A_{1,1}, A_{2,0})$ | |
| 7 | $\text{CHOL}_9$ | |
| | $(A_{2,2}, A_{2,1}, A_{1,1}, A_{2,0})$ | |

**Figure 4: Simulated SuperMatrix execution of the DAG from Figure 1 using a single FIFO queue with two threads where the first thread gains mutual exclusion before the second thread. For illustrative purposes, this simulation assumes that each thread performs lock-step synchronization after executing a single task, which we denote as a single stage. We also show the valid contents of the cache for each thread after the execution of every stage updated via the simplified write-once cache coherence protocol. Each cache has a capacity of four blocks and is ordered according to the most recently used from left to right. The blocks colored in blue incur a cache miss while the ones in red have a cache hit.**

We make several simplifying abstractions. The software cache uses a least recently used (LRU) replacement policy. Each cache line is the size of a submatrix block, and the cache is fully associative. The cache is kept coherent using a simplified write-once protocol where each cache line is either valid or invalid, ignoring the reserved and dirty states. We can also simulate a subset of threads sharing a cache.

In Figure 4, we present the blocks residing in the cache of each thread during the simulated execution of the DAG from Figure 1. In the first two stages, there is only one cache hit with $A_{0,0}$ on thread 1 since the caches are cold. On the other hand, the last two stages have three cache hits since the cache on thread 1 has sufficiently warmed up. Since we are using a single FIFO queue scheduling algorithm where the first thread always gains mutual exclusion before the second thread, thread 1 executes $\text{SYRK}_5$ while thread 2 gets $\text{CHOL}_6$ in stage 4. Ideally thread 1 would execute $\text{CHOL}_6$ since that task updates $A_{1,1}$ which resides in thread 1's cache beforehand. Instead accessing $A_{1,1}$ results in a cache miss for thread 2 and incurs a write invalidation on thread 1.

While only using a single queue, the dequeue routine leverages the software cache to provide cache affinity, which is described in Figure 5. By returning a task with a cache hit or the head of the queue, data communication is reduced or the task with the highest priority is executed. Instead of enqueuing tasks in a simple FIFO order where priority is set by the arrival times of each task, we modify the enqueue routine to maintain a priority queue by sorting tasks according to their height within the DAG. After dequeuing a task, the dispatcher now must have to update the software caches according to the simplified write-once cache coherence protocol.

```
foreach task in queue do
    foreach output block of task do
        if block resides in software cache then
            Record first task with cache hit
        end
    end
end
if task has cache hit then
    Return task
else
    Return head task of queue
end
```

**Figure 5: The routine for dequeuing a task when using cache affinity.**

A similar software cache mechanism has been used to parallelize out-of-core matrix computations with graphics processors [44]. Since GPUs require the explicit communication of data into their local stores, blocks residing on each GPU are logged in a software cache to control the movement of data. That approach only uses the software cache to manage memory whereas we use it here for the scheduling of tasks.

## 5.3 Analysis

The proposed cache affinity priority queue scheduling algorithm strives to balance data locality and load balancing. We will analyze this scheduling algorithm using the basic concepts from queueing theory described in Section 3.

### 5.3.1 *Load balancing*

We start with a single queue implementation because of its load balancing properties.

Let's first assume that we just use a priority queue sorted by the heights of each task in order to further improve the load balancing properties and thus not leverage the software caches. If the arrival remains constant, the conservation law states that waiting time is the same in a FIFO queue as a priority queue.

Prioritizing tasks on the critical path of execution results in a faster arrival of tasks so that

$$\lambda^{FIFO} \leq \lambda^{PRI}.$$

We then apply Little's formula in (1) to get

$$L^{FIFO} \leq L^{PRI}.$$

This inequality invalidates the conservation law when applied to DAG scheduling since we cannot assume that the waiting times remain constant.

Since the average arrival rate increases and the service time $W_s$ is unaffected, we know from (4) that

$$\rho^{FIFO} \leq \rho^{PRI}.$$

The single-queue multi-server system has the property that idle threads will immediately attempt to dequeue any ready and available tasks. Since the expected number of tasks on the queue may increase with the use of a priority queue, there is a higher probability that a thread will have a task to dequeue and execute. Even though the waiting times of tasks on the queue may increase, the use of this heuristic decreases the time tasks wait till becoming ready and available. Since the rate at which threads execute tasks remains the same or even better, the overall time to execute all tasks in a DAG will then decrease.

#### 5.3.2 Data locality

When a cache miss occurs for a task, $O(b^2)$ data must be moved from main memory to the cache. On modern microarchitectures, a memory operation (memop) is at least an order magnitude slower than a floating point operation (flop). As a result, this data communication overhead can be prohibitive to performance despite each task performing $O(b^3)$ operations if a significant number of tasks incur a cache miss. We prioritize the output operands in order to reduce the number of write invalidations incurred during a write hit, which do not happen for just a read miss.

The probability of a cache hit $\gamma$ for a task is a function of five parameters: the number of threads, cache size, block size, problem size, and access patterns within the DAG. The last parameter makes this probability dependent upon the tasks that have previously executed by a thread. This fact breaks the Markovian property of the exponential distribution for service times. The only parameter that we can tune is the block size in order to adjust the cache hit ratio. Each thread groups tasks into two priorities, cache hit or miss, according to the blocks residing in its software cache. This prioritization sorts tasks according to service times, so once again the conservation law cannot be applied here.

Let's now assume that we just use a cache hit or miss priority service without sorting tasks according to their heights and that $C > 0$ where

$$C = E[\gamma]$$

which is the expected value of $\gamma$ for all tasks. By reducing the number of cache misses, we decrease the time to execute each task, and thus from (3)

$$\mu^{FIFO} \leq \mu^{PRI}.$$

Since the enqueuing of tasks is directly tied to their execution, we will assume that

$$\lambda^{PRI} - \lambda^{FIFO} \simeq (\mu^{PRI} - \mu^{FIFO})p$$

so that from (4)

$$\rho^{FIFO} \simeq \rho^{PRI}.$$

If server utilization is roughly the same, then the resulting reduction in the service time will decrease the overall time it takes to execute all tasks in the DAG.

#### 5.3.3 Addressing both simultaneously

By sorting tasks according to their heights and prioritizing tasks with a cache hit, we attempt to keep the server utilization the same while simultaneously reducing the service time and increasing the arrival rate of tasks. We improve the load balance because threads are kept busy through the use of a single queue without sacrificing data locality.

The performance of this scheduling algorithm depends upon the the expected value of $\gamma$. If $C = 0$, then no cache hits will occur, and threads will always dequeue tasks at the head of the queue which have the greatest height. Load balance will be improved as we have shown beforehand. If $C = 1$, then each task will incur a cache hit, and once again threads will dequeue the head of the queue where the service times of each task will decrease. This situation will lead to the near optimal scheduling of tasks for both load balance and data locality. Since $0 < C < 1$, both aspects will contribute to decrease the overall time to execute all tasks.

## 6. PERFORMANCE

In this section, we show that the use of cache affinity yields a performance improvement. We also compare SuperMatrix with other high-performance dense linear algebra libraries.

### 6.1 Target architecture

All experiments were performed on an SGI Altix 350 machine using double-precision floating-point arithmetic. This cache coherent non-uniform memory access (ccNUMA) architecture consists of eight nodes, each with two 1.5 GHz Intel Itanium2 processors, providing a total of 16 CPUs and a theoretical peak performance of 96 GFLOPS ($96 \times 10^9$ floating point operations per second). The nodes are connected via an SGI NUMAlink connection ring and collectively provide 32 GB ($32 \times 2^{30}$ bytes) of general-purpose physical RAM, and each Itanium2 processor contains a 6 MB ($6 \times 2^{20}$ bytes) L3 cache and a 256 KB ($256 \times 2^{10}$ bytes) L2 cache. The OpenMP implementation provided by the Intel compiler 9.0 served as the underlying threading mechanism used by SuperMatrix. Performance was measured by linking to the Intel Math Kernel Library (MKL) 8.1 library as a high-performance implementation of the BLAS. We linked to MKL instead of GotoBLAS 1.26, another high-performance BLAS library, because MKL provides superior performance for the execution of individual tasks given their small problem sizes on this particular computer architecture.

### 6.2 Implementations

We report the performance (in GFLOPS) of three different implementations for the Cholesky, LU, and QR factorizations. 16 threads mapped to each processor and storage block sizes of 192 and 256 were used for all experiments when possible.

- **SuperMatrix**
  We adapted the implementations provided within the open source library `libflame` [52] which require matrices to be stored hierarchically. For the execution of individual tasks on each thread, we linked to serial MKL. We compare two implementations of SuperMatrix: a single FIFO queue and cache affinity.

  Since the traditional algorithms for computing the LU and QR factorizations do not map well to algorithms-by-blocks, SuperMatrix uses LU with incremental pivoting [43, 45] and incremental QR based on Householder transformations [31, 46].

- **Multithreaded MKL**
  The default implementations provided by MKL exploit parallelism internally and assume matrices are stored in the traditional column-major order.

  MKL implements traditional LU with partial pivoting and blocked QR factorizations.

- **PLASMA**
  These implementations also assume flat storage but internally convert matrices into hierarchical ones with one level of blocking. We adjusted the storage block size to match those used by SuperMatrix, but otherwise we did not further tune the code.

  PLASMA 2.0.0 uses *static pipelining* to perform its parallel scheduling [2]. Each thread executes an algorithm-by-blocks sequentially and stalls if a task is not ready or tries to execute the first ready and available task otherwise. A DAG is never explicitly constructed. Static pipelining is akin to in-order scheduling compared to out-of-order scheduling with SuperMatrix.

  PLASMA uses POSIX Threads as its underlying threading mechanism, and we linked it to serial MKL. PLASMA also implements LU with incremental pivoting and incremental QR based on Householder transformations [14, 15].

## 6.3 Results

Performance results are reported in Figures 6 (left) and 7. Several comments are in order:

- Cache affinity provides the same load balancing behavior as a single FIFO queue, so the difference in performance between the two stems strictly from the decrease in data communication.

  We illustrate this behavior in Figure 6 (right) for the Cholesky factorization. The idle thread ratio, which is the complement to server utilization, shows the total time all threads perform no useful computation divided by the parallel execution time of the dispatcher for which both the single FIFO queue and cache affinity are nearly identical. As this ratio converges to zero, load balance is maximized since all threads will be executing tasks throughout the entire duration of their lifetimes.

  Despite having the same load balancing properties, cache affinity still significantly outperforms a single FIFO queue in Figure 6 (left). Since we simulate the blocks residing in the software caches, we can determine which tasks incur a cache hit or miss. We also show the simulated cache hit ratio for each task's output blocks compared to the total number of tasks. As this ratio converges to one, data locality is optimized since every task incurs a cache hit when overwriting its output blocks. Here we can clearly see a divergence in behavior between the two scheduling algorithms and hence the difference in performance.

- The performance benefits of cache affinity on this machine exist because of the large cache on an Itanium2 processor. Approximately twenty-one $192 \times 192$ blocks can fit into the 6 MB L3 cache on each processor. The benefits of optimizing for data locality can be mitigated when the lower levels of cache are quite small. For instance the AMD Opteron quad-core processor has a 2 MB L3 cache that is shared between all four cores. As a result, only seven $192 \times 192$ blocks can reside in the cache, and thrashing will occur frequently where threads continuously evict blocks from the cache since tasks in this problem domain access from one up to three or more matrix operands.

- MKL 8.1 is not the latest version, but it is the only one available on this machine. The performance of multithreaded `dpotrf` in Figure 6 (left) is dramatically improved in later versions of MKL, but we expect the newer implementations of the Cholesky, LU, and QR factorizations to roughly match the performance curves of `dgetrf` and `dgeqrf` from Figure 7.

- SuperMatrix outperforms PLASMA because of its ability to better exploit parallelism from a DAG and the matrix copy overhead in PLASMA.

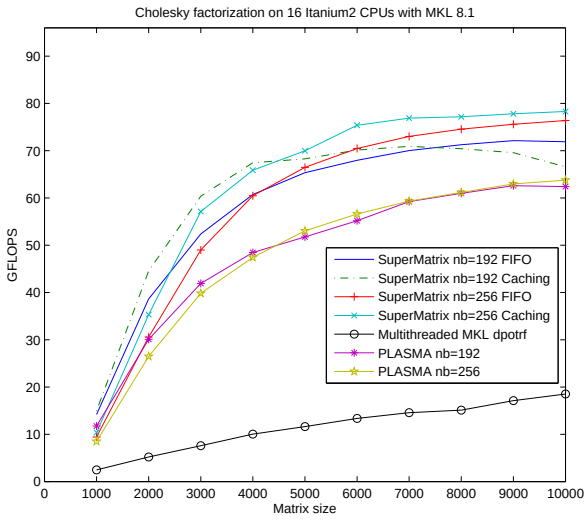
- There is a precipitous decline in performance displayed by SuperMatrix implementations for asymptotically large problem sizes. We conjecture that due to the out-of-order scheduling, blocks are accessed in an erratic order across the entire matrix, which causes a greater number of translation lookaside buffer (TLB) misses for large problem sizes. Cache affinity optimizes for lower levels of cache but does not yet consider the TLB. Tuning the block size to be evenly divisible by the page size slightly alleviates this problem.

# 7. RELATED WORK

The implications of optimizing for the cache of shared-memory parallel architectures were explored in [53]. Even though the disparity between memory latency and processor speeds in architectures at the time did not provide any tangible performance gains, the authors had the foresight to anticipate the need for cache affinity. Their approach saves the last task executed by a thread in order to find subsequent tasks whereas we use a more data-centric approach that records the blocks accessed by each thread.

SMP Superscalar (SMPSs) [42] is a general purpose runtime system that also constructs a DAG using the input and output operands of tasks, which are denoted using compiler directives in the code. SMPSs uses work stealing for its parallel scheduling of tasks, yet it is not domain-specific to matrix computations like PLASMA and SuperMatrix. SMPSs begins execution of tasks as soon as one is generated whereas SuperMatrix delays execution until the entire DAG has been constructed, yet we have evidence that the DAG for matrix computations can be statically generated [16].

Several other general-purpose parallel schedulers exist such as Cilk [13] and Intel Thread Building Blocks (TBB) [36], both of which also use work stealing, and TBB implements the mailbox optimization [48]. These systems are different than the ones previously introduced because Cilk and TBB construct a DAG from user defined dependencies between tasks. This approach exploits control-level parallelism, which is specified by the order in which operations appear in the



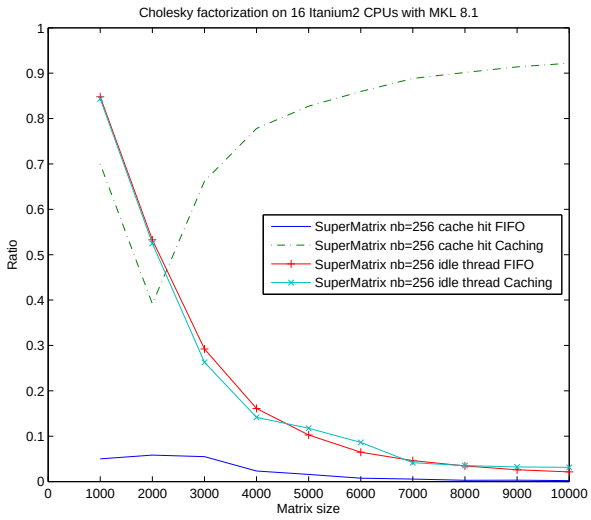
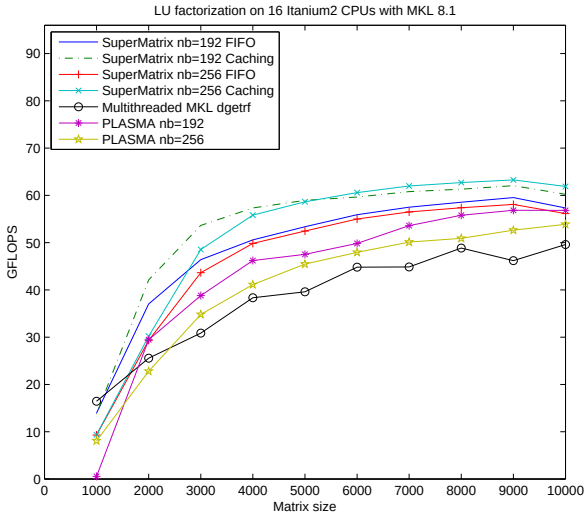




**Figure 6: Left: Performance of Cholesky factorization on a 16 Itanium2 processor system linked with the Intel MKL 8.1 library. Right: Comparison of load balance versus data locality for the Cholesky factorization. The cache hit ratio measures data locality, which consists of the number of tasks that incur a simulated cache hit for one of its output matrix operands divided by the total number of tasks. The idle thread ratio measures load balance, which consists of the average time each thread does not perform any useful computation divided by the total time incurred by the dispatcher.**



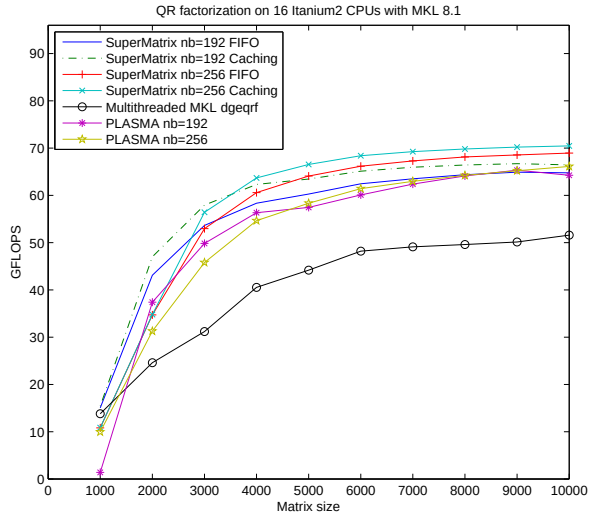




**Figure 7: Performance of LU (left) and QR (right) factorizations.**

code, as opposed to the data flow parallelism employed by SuperMatrix.

Cache-oblivious algorithms are designed using recursive divide and conquer algorithms, which have proven asymptotic time and memory traffic bounds, without specific knowledge of the underlying memory hierarchy [10, 25, 32, 55]. Cache-oblivious algorithms recursively subdivide a problem until a subproblem fits into the cache regardless of the cache's size, so this approach eliminates the need to tune algorithms for specific computer architectures. Our cache affinity scheduling algorithm is fundamentally different from the cache-oblivious approach since we inherently depend upon the size of the cache to exploit data locality.

# 8. CONCLUSION

In this paper, we have shown that data locality is paramount for scheduling. Along with load balancing, both aspects must be considered simultaneously rather than separately. By leveraging the cache coherence protocol on shared-memory

parallel architectures, we have developed a cache affinity scheduling algorithm that attains a significant increase in performance.

## Future work

The use of a single queue to which all threads access will not be scalable to many-core architectures. We are investigating heuristics for assigning tasks to multiple queues, each of which are accessed by a subset of threads. We intend to exploit the recursive nature of dense linear algebra algorithms to aide these heuristics. In doing so, we anticipate that this partitioning of threads can help reduce the number of TLB misses for asymptotically large problem sizes.

We only presented results with two block sizes that roughly match either the L2 cache or page size. This issue begs the question of how to tune the block size. Thus far, auto-tuning of matrix computations has only dealt with sequential kernels [50, 54], but adjusting the block size now brings into account the extra dimension of parallelism versus the granularity of tasks.

## Additional information

For additional information on FLAME visit

`http://www.cs.utexas.edu/users/flame/`.

# 9. ACKNOWLEDGMENTS

We thank the members of the FLAME team for their support. This research was partially sponsored by NSF grants CCF–0540926 and CCF–0702714.

*Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation (NSF).*

# 10. REFERENCES

[1] U. A. Acar, G. E. Blelloch, and R. D. Blumofe. The data locality of work stealing. In *SPAA '00: Proceedings of the Twelfth ACM Symposium on Parallel Algorithms and Architectures*, pages 1–12, Bar Harbor, ME, USA, July 2000.

[2] E. Agullo, B. Hadri, H. Ltaief, and J. Dongarra. Comparative study of one-sided factorizations with multiple software packages on multi-core hardware. LAPACK Working Note #217 UT-CS-09-642, The University of Tennessee at Knoxville, Department of Computer Science, April 2009.

[3] A. O. Allen. *Probability, Statistics, and Queueing Theory with Computer Science Applications*. Academic Press, San Diego, 1990.

[4] E. Anderson, Z. Bai, C. Bischof, L. S. Blackford, J. Demmel, J. J. Dongarra, J. D. Croz, S. Hammarling, A. Greenbaum, A. McKenney, and D. Sorensen. *LAPACK Users' Guide (Third ed.)*. SIAM, Philadelphia, 1999.

[5] J. Archibald and J.-L. Baer. Cache coherence protocols: Evaluation using a multiprocessor simulation model. *ACM Transactions on Computer Systems*, 4(4):273–298, November 1986.

[6] B. Bartsch and G. Bolch. A conservation law for $G/G/m$ queueing systems. *Acta Informatica*, 10(1):105–109, March 1978.

[7] P. Bientinesi, J. A. Gunnels, M. E. Myers, E. S. Quintana-Ortí, and R. A. van de Geijn. The science of deriving dense linear algebra algorithms. *ACM Transactions on Mathematical Software*, 31(1):1–26, March 2005.

[8] P. Bientinesi, E. S. Quintana-Ortí, and R. A. van de Geijn. Representing linear algebra algorithms in code: The FLAME application programming interfaces. *ACM Transactions on Mathematical Software*, 31(1):27–59, March 2005.

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

[10] G. E. Blelloch, R. A. Chowdhury, P. B. Gibbons, V. Ramachandran, S. Chen, and M. Kozuch. Provably good multicore cache performance for divide-and-conquer algorithms. In *SODA '08: Proceedings of the Nineteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 501–510, San Francisco, CA, USA, January 2008.

[11] G. E. Blelloch, P. B. Gibbons, and Y. Matias. Provably efficient scheduling for languages with fine-grained parallelism. *Journal of the ACM*, 46(2):281–321, March 1999.

[12] R. Blumofe and C. Leiserson. Scheduling multithreaded computations by work stealing. *Journal of the ACM*, 46(5):720–748, September 1999.

[13] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. *Journal of Parallel and Distributed Computing*, 37(1):55–69, August 1996.

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

[15] A. Buttari, J. Langou, J. Kurzak, and J. Dongarra. A class of parallel tiled linear algebra algorithms for multicore architectures. *Parallel Computing*, 35(1):38–53, January 2009.

[16] E. Chan, J. Nagle, F. G. Van Zee, and R. van de Geijn. Transforming linear algebra libraries: From abstraction to parallelism. FLAME Working Note #38 TR-09-17, The University of Texas at Austin, Department of Computer Sciences, May 2009.

[17] E. Chan, E. S. Quintana-Ortí, G. Quintana-Ortí, and R. van de Geijn. SuperMatrix out-of-order scheduling of matrix operations for SMP and multi-core architectures. In *SPAA '07: Proceedings of the Nineteenth ACM Symposium on Parallelism in Algorithms and Architectures*, pages 116–125, San Diego, CA, USA, June 2007.

[18] E. Chan. Principles and practice of thread-level parallelism in the SuperMatrix runtime system. Technical report, The University of Texas at Austin, Department of Computer Sciences, August 2009. Submitted to *IPDPS '10: The 24th IEEE*

*International Parallel and Distributed Processing Symposium*.

[19] E. Chan, F. G. Van Zee, P. Bientinesi, E. S. Quintana-Ortí, G. Quintana-Ortí, and R. van de Geijn. SuperMatrix: A multithreaded runtime scheduling system for algorithms-by-blocks. In *PPoPP '08: Proceedings of the Thirteenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 123–132, Salt Lake City, UT, USA, February 2008.

[20] M. Cosnard and E. Jeannot. Compact DAG representation and its dynamic scheduling. *Journal of Parallel and Distributed Computing*, 58(3):487–514, September 1999.

[21] J. J. Dongarra, J. Du Croz, S. Hammarling, and I. Duff. A set of level 3 basic linear algebra subprograms. *ACM Transactions on Mathematical Software*, 16(1):1–17, March 1990.

[22] J. J. Dongarra, J. Du Croz, S. Hammarling, and R. J. Hanson. An extended set of FORTRAN basic linear algebra subprograms. *ACM Transactions on Mathematical Software*, 14(1):1–17, March 1988.

[23] E. Elmroth, F. Gustavson, I. Jonsson, and B. Kagstrom. Recursive blocked algorithms and hybrid data structures for dense matrix library software. *SIAM Review*, 46(1):3–45, 2004.

[24] A. K. Erlang. The theory of probabilities and telephone conversations. *Nyt Tidsskrift for Matematik*, 20(B):33–39, 1909.

[25] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *FOCS '99: Proceedings of the 40th Annual Symposium on Foundations of Computer Science*, pages 285–297, New York, NY, USA, October 1999.

[26] A. Gerasoulis and Y. Yang. A comparison of clustering heuristics for scheduling directed acycle graphs on multiprocessors. *Journal of Parallel and Distributed Computing*, 16(4):276–291, December 1992.

[27] J. R. Goodman. Using cache memory to reduce processor-memory traffic. In *ISCA '83: Proceedings of the Tenth Annual International Symposium on Computer Architecture*, pages 124–131, Stockholm, Sweden, 1983.

[28] K. Goto and R. A. van de Geijn. Anatomy of a high-performance matrix multiplication. *ACM Transactions on Mathematical Software*, 34(3):12:1–12:25, May 2008.

[29] D. Gross and C. M. Harris. *Fundamentals of Queueing Theory (Third ed.)*. John Wiley & Sons, New York, 1998.

[30] J. A. Gunnels, F. G. Gustavson, G. M. Henry, and R. A. van de Geijn. FLAME: Formal linear algebra methods environment. *ACM Transactions on Mathematical Software*, 27(4):422–455, December 2001.

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

[32] F. G. Gustavson. Recursion leads to automatic variable blocking for dense linear-algebra algorithms. *IBM Journal of Research and Development*, 41(6):737–756, November 1997.

[33] D. G. Kendall. Stochastic processes occurring in the theory of queues and their analysis by the method of the imbedded Markov chain. *Annals of Mathematical Statistics*, 24(3):338–3854, 1953.

[34] L. Kleinrock. A conservation law for a wide class of queueing disciplines. *Naval Research Logistics Quarterly*, 12(2):181–192, 1965.

[35] L. Kleinrock. *Communication Nets; Stochastic Message Flow and Delay*. Dover Publications, 1972.

[36] A. Kukanov and M. J. Voss. The foundations for scalable multi-core software in Intel® Threading Building Blocks. *Intel Technology Journal*, 11(4):309–322, November 2007.

[37] Y.-K. Kwok and I. Ahmad. Dynamic critical-path scheduling: An effective technique for allocating task graphs to multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 7(5):506–521, May 1996.

[38] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. Basic linear algebra subprograms for Fortran usage. *ACM Transactions on Mathematical Software*, 5(3):308–323, September 1979.

[39] J. D. C. Little. A proof for the queuing formula: $L = \lambda W$. *Operations Research*, 9(3):383–387, May–June 1961.

[40] T. M. Low and R. van de Geijn. An API for manipulating matrices stored by blocks. FLAME Working Note #12 TR-04-15, The University of Texas at Austin, Department of Computer Sciences, May 2004.

[41] C. McCreary, A. Khan, J. Thompson, and M. McArdle. A comparison of heuristics for scheduling DAGs on multiprocessors. In *IPPS '94: Proceedings of the Eighth International Parallel Processing Symposium*, pages 446–451, Cancun, Mexico, April 1994.

[42] J. M. Perez, R. M. Badia, and J. Labarta. A dependency-aware task-based programming environment for multi-core architectures. In *Cluster '08: Proceedings of the 2008 IEEE International Conference on Cluster Computing*, pages 142–151, Tsukuba, Japan, September 2008.

[43] E. Quintana-Ortí and R. A. van de Geijn. Updating an LU factorization with pivoting. *ACM Transactions on Mathematical Software*, 35(2):11:1–11:16, July 2008.

[44] G. Quintana-Ortí, F. D. Igual, E. S. Quintana-Ortí, and R. A. van de Geijn. Solving dense linear systems on platforms with multiple hardware accelerators. In *PPoPP '09: Proceedings of the Fourteenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 121–130, Raleigh, NC, USA, February 2009.

[45] G. Quintana-Ortí, E. S. Quintana-Ortí, E. Chan, R. van de Geijn, and F. G. Van Zee. Design of scalable dense linear algebra libraries for multithreaded architectures: The LU factorization. In *MTAAP '08: Proceedings of the 2008 Workshop on Multithreaded Architectures and Applications*, pages 1–8, Miami, FL, USA, April 2008.

[46] G. Quintana-Ortí, E. S. Quintana-Ortí, E. Chan, R. A. van de Geijn, and F. G. Van Zee. Scheduling of

QR factorization algorithms on SMP and multi-core architectures. In *PDP '08: Proceedings of the Sixteenth Euromicro International Conference on Parallel, Distributed and Network-Based Processing*, pages 301–310, Toulouse, France, February 2008.

[47] G. Quintana-Orti, E. S. Quintana-Orti, R. A. van de Geijn, F. G. Van Zee, and E. Chan. Programming matrix algorithms-by-blocks for thread-level parallelism. *ACM Transactions on Mathematical Software*, 36(3):14:1–14:26, July 2009.

[48] A. Robison, M. Voss, and A. Kukanov. Optimization via reflection on work stealing in TBB. In *HIPS '08: Proceedings of the Thirteenth International Workshop on High-Level Parallel Programming Models and Supportive Environments*, pages 1–8, Miami, FL, USA, April 2008.

[49] L. Schrage. An alternative proof of a conservation law for the queue $G/G/1$. *Operations Research*, 18(1):185–187, January–February 1970.

[50] J. G. Siek, I. Karlin, and E. R. Jessup. Build to order linear algebra kernels. In *POHLL '08: Proceedings of the 2008 Workshop on Performance Optimization for High-Level Languages and Libraries*, pages 1–8, Miami, FL, USA, April 2008.

[51] D. R. Smith and W. Whitt. Resource sharing for efficiency in traffic systems. *Bell System Technical Journal*, 60(1):39–55, January 1981.

[52] F. G. Van Zee. `libflame`*: The Complete Reference*. `http://www.lulu.com/content/5915632/`, 2009.

[53] R. Vaswani and J. Zahorjan. The implications of cache affinity on processor scheduling for multiprogrammed, shared memory multiprocessors. *ACM SIGOPS Operating Systems Review*, 25(5):26–40, October 1991.

[54] R. C. Whaley and J. J. Dongarra. Automatically tuned linear algebra software. In *SC '98: Proceedings of the 1998 ACM/IEEE Conference on Supercomputing*, pages 1–27, San Jose, CA, USA, November 1998.

[55] A. N. Yzelman and R. H. Bisseling. Cache-oblivious sparse matrix-vector multiplication by using sparse matrix partitioning methods. *SIAM Journal on Scientific Computing*, 31(4):3128–3154, July 2009.