The Foundations of Thread-level Parallelism in the SuperMatrix Runtime System[*]

Ernie Chan[†]

Abstract

In this paper, we describe the interface and implementation of the SuperMatrix runtime system. SuperMatrix exploits parallelism from matrix computations by mapping a linear algebra algorithm to a directed acyclic graph (DAG). We give detailed descriptions of how to dynamically construct a DAG where tasks consisting of matrix operations represent the nodes and data dependencies between tasks represent the edges of the graph. We show the algorithm that, given a DAG as input, dispatches and schedules tasks to threads. Different scheduling heuristics and optimizations implemented as part of SuperMatrix are discussed, demonstrating the flexibility and portability that results from the separation of concerns. Using this flexible framework, we compare several scheduling algorithms, such as work stealing, that optimize for either load balancing or data locality, and we demonstrate that a relatively simple, single queue implementation provides exceptional performance while also allowing for the widest flexibility for further enhancements. Performance results from a sixteen core machine are provided.

1 Introduction

With the emergence of multicore architectures, exploiting parallelism will be become vital for the performance of computationally intensive applications. Many libraries such as the Linear Algebra PACKage (LAPACK) [2] have to be reimplemented. Simply linking to multithreaded Basic Linear Algebra Subprograms (BLAS) [12] libraries is not an efficient solution since implicit synchronization points exists between calls to each BLAS operation, so many opportunities for parallelism are lost. Here we discuss the implementation details of the SuperMatrix runtime system that allow the runtime system to be highly portable and flexible. SuperMatrix, as being part of the broader FLAME project, not only addresses the programmability issue [19] for these new architectures but also provides exceptional performance when parallelizing matrix computations.

[*]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).*

[†]Department of Computer Sciences, The University of Texas at Austin, Austin, Texas 78712, `echan@cs.utexas.edu`

Within the Formal Linear Algebra Method Environment (FLAME) project, we have developed a notation [15] that allows for the mechanical derivation of algorithms [3]. We implemented linear algebra algorithms using the FLAME API for the C programming language [5]. This API mirrors the notation used to express the algorithms using object-based matrix data structures, thereby abstracting away implementation details such as array indexing. A library of commonly used linear algebra operations has been implemented and released as open source software in `libflame` [21].

By storing matrices hierarchically [13] 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*. In `libflame` we have abstracted away the need to specify the matrix storage within the implementation of each linear algebra algorithm. As a result we can seamlessly execute sequential algorithms with flat matrices stored in the traditional column-major order or leverage the same code to execute algorithms-by-blocks using hierarchical matrices with arbitrary depth.

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 [10, 11]. 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*. During the analyzer phase, the execution of tasks is delayed, and instead the DAG is constructed dynamically. Once the analyzer is done, the dispatcher phase is invoked which dispatches and schedules tasks to threads. In this paper, we discuss the details of the interface for invoking SuperMatrix and its implementation for both phases.

The specific contributions of the present paper include:

- We discuss in depth the implementation details of the SuperMatrix runtime system along with several optimizations.
- As a result, we expose the portability of the SuperMatrix implementation which only depends upon a small set of threading mechanisms.
- SuperMatrix is shown to be highly flexible in that we can use different scheduling algorithms by only changing a few highly modularized routines.
- We make quantitative comparisons between more deterministic scheduling algorithms and work stealing [6] from which we show the benefits of relatively simple algorithms such as a single queue implementation.

Together these contributions show that the separation of concerns between the algorithms and runtime system allows us to provide high performance and gather further insight into the scheduling issues presented by this problem domain.

The rest of the paper is organized as follows. In Section 2 we discuss the matrix storage and interface required for the SuperMatrix runtime sytem. Section 3 describes the process for dynamically constructing a DAG, and Section 4 presents the algorithm for dispatching tasks to threads. We enumerate different scheduling algorithms and how to incorporate those into the dispatcher in Section 5. Section 6 provides performance results and an analysis of work stealing. We discuss related work in Section 7 and conclude the paper in Section 8.

```
FLASH_Queue_begin();
FLASH_Chol(
  FLA_LOWER_TRIANGULAR, AH );
FLASH_Trinv(
  FLA_LOWER_TRIANGULAR,
  FLA_NONUNIT_DIAG,     AH );
FLASH_Ttmm (
  FLA_LOWER_TRIANGULAR, AH );
FLASH_Queue_end();
```

```
#define ENQUEUE_FLASH_Gemm(  \
            transA, transB,    \
            alpha, A, B,       \
            beta, C, cntl )    \
          FLASH_Queue_push(    \
            (void *) FLA_Gemm, \
            (void *) cntl,     \
            ‘‘Gemm’’,          \
            2, 2, 2, 1,        \
            transA, transB,    \
            alpha, beta,       \
            A, B, C )
```

Figure 1: Left: Parallel region interface to invoke the SuperMatrix runtime. Right: Macro definition for storing a task.

2 Interface

In this section, we describe the necessary matrix storage and interface to invoke the SuperMatrix runtime system.[1]

2.1 Hierarchical matrix storage

Even though the methodology to exploit parallelism through DAG scheduling can be applied to flat matrices, SuperMatrix currently requires the matrix operands to be stored hierarchically. As a result each submatrix block is stored contiguously for spatial locality and easily demarcated in order to detect dependencies between tasks. We use the FLASH [17] extension to the FLAME/C API for creating and accessing hierarchical matrices.

Here we present one particular routine that copies a flat matrix to a hierarchical one:

`FLASH_Obj_create_hier_copy_of_flat( A, depth, &blksizes, &AH );`

where `A` is the original flat matrix, `depth` is number of levels for blocking, `blksizes` is an integer array specifying the number of elements in a block at each level, and `AH` is the resulting hierarchical matrix [21]. Typically, only a depth of one is used in order to create matrices with storage-by-blocks, also called block data layout, where each element in the top-level matrix is a pointer to the contiguously stored submatrix blocks.

2.2 Parallel regions

SuperMatrix uses parallel regions to identify the operations from which to exploit parallelism. In Figure 1 (left), we give an example of a parallel region for inversion of a symmetric positive definite matrix, which is computed using the Cholesky factorization (CHOL) $A \rightarrow LL^T$, followed by inversion of a triangular matrix (TRINV) $R := L^{-1}$, and then triangular matrix multiplication by its transpose (TTMM) $A^{-1} := R^T R$. Each of these operations reads and

[1]The interface and implementation of SuperMatrix are implemented in C, but this approach is language independent and can be applied to Java or C++, but that discussion is beyond the scope of this paper.

then overwrites the matrix AH with the result of the computation. We present an algorithm for computing CHOL in Section 3, which is representative of the algorithms for computing TRINV and TTMM. See [4, 11] for more details about SPD inverse.

All the code in between FLASH_Queue_begin and FLASH_Queue_end specifies the parallel region. At the start of the parallel region, the analyzer is invoked which stores all tasks and constructs the DAG. The dispatcher will then be called at the end of the parallel region in order to execute all the tasks. If the begin and end routines are not placed in the code, then each of the three operations will be parallelized individually by SuperMatrix where implicit synchronization points between each operation will exist as opposed to SuperMatrix exploiting parallelism between all three operations in concert. If SuperMatrix is turned off, those begin and end routines essentially become no-ops, and the operations are all executed sequentially. The begin and end routines can also be nested recursively, similar to the OpenMP sections construct, and across multiple functions interprocedurally, so the analyzer phase starts storing tasks at the outermost begin, and the dispatcher phase will not be invoked until reaching the outermost end.

Only SuperMatrix enabled operations can exist within a parallel region, a list of which can be found in [21]; otherwise sequential semantics will not be maintained. For example, the routine FLASH_Random_matrix fills a matrix with random values, yet it is not a SuperMatrix supported routine. If this routine is placed within the parallel region overwriting AH, then the data will be corrupted before the tasks are executed in parallel.

2.3 Tasks

The only information needed to detect dependencies are the input and output operands of each task. The FLAME/C API for general matrix-matrix multiplication (GEMM) $C := \alpha AB + \beta C$ is given by:

```
    FLA_Gemm( FLA_NO_TRANSPOSE, FLA_NO_TRANSPOSE, alpha, A, B, beta, C );
```

Figure 1 (right) shows the macro definition for storing a GEMM task in SuperMatrix. The first operand of FLASH_Queue_push is a pointer to the function to be executed. The second is a pointer to a control structure only used internally for which users can pass in a null pointer, and the string is also only used internally for debugging purposes. The four integers indicate the number of operational arguments, constant multipliers, input matrix operands, and output matrix operands.

All tasks have at least one output matrix operand for which the matrix is first read and then overwritten, but a few have more than one. All matrix operands must be contiguously stored blocks, so the algorithm-by-blocks that creates and stores tasks must recursively stride through the hierarchical matrices. The constant multipliers are also matrix objects for which the values are typically FLA_ZERO, FLA_ONE, or FLA_MINUS_ONE. The possible transposition arguments for GEMM consist of FLA_NO_TRANSPOSE, FLA_TRANSPOSE, and FLA_CONJ_TRANSPOSE.

3 Analyzer

We present the algorithm for dynamically constructing a DAG used by the analyzer phase in Figure 2 (left). For notational purposes, here we assume that the linear algebra algorithm

```
Data: Linear algebra algorithm $S$,
      block size $b$
Result: Directed acyclic graph $(V, E)$
$V := \emptyset$; $E := \emptyset$;
foreach $A \in \mathbb{R}^{m \times n}$ accessed by $S$ do
    for $i$ to $\frac{m}{b}$ do
        for $j$ to $\frac{n}{b}$ do
            $A^w_{i,j} := \emptyset$; $A^r_{i,j} := \emptyset$;
        end
    end
end
while Execute $S$ do
    Store task: $V := V \cup \{t\}$;
    foreach $A_{i,j}$ accessed by $t$ do
        $\hat{i} := \frac{i}{b}$; $\hat{j} := \frac{j}{b}$;
        $A^r_{\hat{i},\hat{j}} := A^r_{\hat{i},\hat{j}} \cup \{t\}$;
        if $A^w_{\hat{i},\hat{j}} \neq \emptyset$ and $A^w_{\hat{i},\hat{j}} \neq t$ then
            Store flow dependency:
            $E := E \cup \{(A^w_{\hat{i},\hat{j}}, t)\}$;
        end
        if $A_{i,j}$ is overwritten by $t$ then
            foreach $t^r \in A^r_{\hat{i},\hat{j}}$ do
                if $t^r \neq t$ then
                    Store anti-dependency:
                    $E := E \cup \{(t^r, t)\}$;
                end
            end
            $A^w_{\hat{i},\hat{j}} := t$; $A^r_{\hat{i},\hat{j}} := \emptyset$;
        end
    end
end
```

```
for $j = 0, \ldots, N-1$ do
    $A_{j,j} := \text{Chol}(A_{j,j})$; (CHOL)
    for $i = j+1, \ldots, N-1$ do
        $A_{i,j} := A_{i,j} A^{-T}_{j,j}$; (TRSM)
    end
    for $k = j+1, \ldots, N-1$ do
        $A_{k,k} := A_{k,k} - A_{k,j} A^T_{k,j}$; (SYRK)
        for $i = j+1, \ldots, N-1$ do
            $A_{i,k} := A_{i,k} - A_{i,j} A^T_{k,j}$;
            (GEMM)
        end
    end
end
```

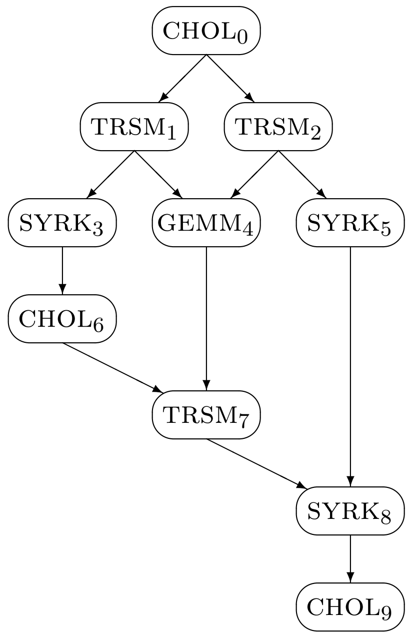


Figure 2: Left: The algorithm that constructs a DAG given a linear algebra algorithm and block size. Top right: A loop-based algorithm that computes the Cholesky factorization by blocks where $A \in \mathbb{R}^{n \times n}$, $N = \frac{n}{b}$, $A_{i,j} \in \mathbb{R}^{b \times b}$ where $0 \leq i, j < N$. Bottom right: The DAG for the Cholesky factorization on a 3×3 matrix of blocks where the subscripts denote the order in which the tasks are stored.

accesses flat matrices and strides through the matrix according to the block size b. The SuperMatrix implementation embeds pointers within the contiguously stored blocks of a hierarchical matrix in order to perform the same data dependency analysis where the block size manifests itself as the size of each block.

We use CHOL as a motivating example for describing the analyzer where we provide a loop-based algorithm for computing this operation in Figure 2 (top right). We provide the resulting DAG for CHOL on a 3×3 matrix of blocks in Figure 2 (bottom right).

As we execute the algorithm within the analyzer, we store tasks using the macro definitions similar to the one found in Figure 1 (right) instead of executing those tasks. For the Cholesky factorization, we store the first task CHOL_0 which reads and overwrites block $A_{0,0}$. Since $A_{0,0}$ has not been accessed before, we know that block is ready to be used. We then record that CHOL_0 was the last task to overwrite $A_{0,0}$. Next we store TRSM_1, which reads $A_{0,0}$, so we now have an occurrence of a flow dependency between CHOL_0 and TRSM_1. Flow dependencies, often referred to as "true" dependencies, occur when a block is read by one task after it is overwritten by a previous task (read-after-write). This process is repeated until all tasks are stored, and the entire DAG is constructed. In order to construct the DAG for SPD inverse, CHOL is executed as stated above followed sequentially by TRINV and TTMM, both of which are implemented using similar loop-based algorithms.

Only flow dependencies occur within CHOL, but anti-dependencies also exist within TRINV and TTMM [11]. Anti-dependencies occur when a task must read a block before another task can write to the same block (write-after-read). Even though only one task can overwrite a block one at a time, multiple tasks can read a block simultaneously. As a result, we maintain a list of tasks that have read a particular block. Care must be made to clear all the lists of tasks stored by each block once all tasks are executed in order to avoid memory leaks.

Output dependencies, the final class of dependencies, occur when the result of one task is overwritten by the result of a subsequent task (write-after-write). The copy routine is the only BLAS operation that strictly overwrites a matrix without first reading it, which rarely occurs in linear algebra algorithms. As such, we use the same mechanism as flow dependencies in order to detect output dependencies.

4 Dispatcher

In Figure 3 (left) we present the algorithm that all threads execute in order to dispatch tasks to threads. In Figure 3 (right) we present two optimizations prefetching and thread preference, which we elaborate further in this section. We discuss the implementation of the *enqueue* and *dequeue* routines in Section 5. In this section, we assume that all tasks are placed onto a single first-in first-out (FIFO) order queue from which all threads access.

When the dispatcher is invoked, it first spawns the appropriate number of threads. Then each thread performs the algorithm in Figure 3 (left) where all of the initial ready tasks are enqueued. We define a ready task as one for which all of the tasks it depends upon have been executed. Then all the threads enter the **while** loop and then spin wait until successfully dequeuing a task. Once a task has been obtained, that particular thread will execute it, update all of the dependent tasks, and then free the task data structure. We define a dependent task as one that has an incoming data dependency from the executed task. We

```
foreach task in DAG do
    if task is ready then
        Acquire lock
        Enqueue task
        Release lock
    end
end
Condition := true
while condition do
    Acquire lock
    Dequeue task
    Release lock
    if task $\neq \emptyset$ then
        Execute task
        foreach dependent task do
            Acquire lock
            Update dependent task
            Release lock
            if dependent task is ready
            then
                Enqueue dependent task
            end
        end
        Free task
    else if work stealing then
        Acquire lock
        Steal task
        Release lock
    end
    Acquire lock
    if tasks are not available then
        Condition := false
    end
    Release lock
end
```

```
foreach task in DAG do
    if task is ready then
        Enqueue task
    end
end
Prefetch blocks
while tasks are available do
    if saved task $\neq \emptyset$ then
        Dequeue task
    else
        Use saved task
        Saved task := $\emptyset$
    end
    if task $\neq \emptyset$ then
        Execute task
        Dequeue saved task
        foreach dependent task do
            Update dependent task
            if dependent task is ready
            then
                if saved task = $\emptyset$ then
                    Save dependent task
                else if dependent > saved
                task then
                    Enqueue saved task
                    Save dependent task
                else
                    Enqueue dependent
                    task
                end
            end
        end
        Free task
    end
end
```

Figure 3: The algorithm that all threads execute in order to dispatch and schedule tasks from a directed acyclic graph with mutual exclusion included (left) and the same algorithm incorporating prefetching and thread preference without mutual exclusion enumerated (right) where a task is greater than another task if it has a higher priority.

store the data dependency data structures as the outgoing edges of a task as opposed to being the incoming edges. In Figure 2 (bottom right), CHOL_0 is the only initial ready task, and its dependent tasks are TRSM_1 and TRSM_2. After all tasks have been executed, the threads will exit the **while** loop, and then the dispatcher will join the threads back together.

4.1 Mutual exclusion

The need for mutual exclusion arises at four locations within the dispatcher, shown in Figure 3 (left). The first two occur at the enqueue and dequeue routines. Since we use a single queue implementation from which all threads access, we must provide mutual exclusion for that queue.

Each task has a counter of how many tasks it is dependent upon, which is the number of a node's incoming edges in the DAG. That counter is decremented as tasks execute and update their dependent tasks. Once that counter reaches zero, then the task is ready. A task may be dependent upon multiple tasks that may be executing in parallel, which then subsequently update it. This situation leads to the third use of mutual exclusion. We can use multiple locks to provide mutual exclusion for updating several tasks at the same time.

The fourth location lies with detecting the terminating condition for the **while** loop. We maintain a global counter of the total number of executed tasks. Once that counter reaches the total number of tasks, then we know to end the loop for all threads and then exit the dispatcher. Work stealing provides a potential extra fifth need of mutual exclusion, which we will discuss further in Section 5.

We only need the ability to acquire and release individual locks to perform mutual exclusion. While more efficient constructs such as an atomic update provided in OpenMP exist, we instead chose to make SuperMatrix more portable.

4.2 Prefetching

When threads start executing tasks, the caches of each processor associated with a thread are virtually devoid of any data pertinent to the execution of those tasks. As a result the first several tasks will always incur a cache miss when accessing its blocks. In order to warm up the caches, we *prefetch* the blocks accessed by those first tasks.

In Figure 3 (right), we first enqueue all the initial ready tasks. As we traverse all the stored tasks to find the ready ones, we log the K blocks accessed by the first tasks where K is the size of the L3 or L2 cache divided by the storage block size of the hierarchical matrix. Every thread then touches those K blocks before entering the **while** loop by reading each block, striding through them with an increment the size of an individual L1 cache line. In doing so we preload the caches with blocks without needing to access each element and incur more cache misses during the execution of a task. In Figure 2 (bottom right), the first three blocks accessed are $A_{0,0}$, $A_{1,0}$, and $A_{2,0}$ by the tasks CHOL_0, TRSM_1, and TRSM_2.

4.3 Thread preference

We also present the second optimization thread preference in Figure 3 (right). When tasks enqueue dependent tasks, it is more likely that another thread will dequeue the newly enqueued task since idle threads are continously polling to dequeue a task. In many situations, we want the same thread that enqueues a task to also execute that a dependent task in order to preserve data locality. We implement thread preference in order to allow the enqueuing thread to have the first chance at dequeuing a task. For instance in Figure 2 (bottom right), we want the same thread to execute $\textsc{syrk}_8$ and $\textsc{chol}_9$ since both tasks update $A_{2,2}$.

After a thread executes a task, it will first dequeue and save a new task. While updating the dependent tasks and if any of those are ready, that thread will see any of the ready dependent tasks have a higher priority than the recently saved task. If so, that thread will replace the saved task and execute it once the updating is complete. We will discuss priorities of each task in Section 5. Since we are assuming a single FIFO queue implementation, the priorities are set by arrival times, so no saved task will be replaced, and thus in this case thread preference will only occur when there are no other tasks waiting in the queue.

5 Scheduling

In this section, we present different implementations of the enqueue and dequeue functions in order to perform different scheduling algorithms.

We have already discussed the single FIFO queue in Section 4. All threads enqueue tasks to the tail of the queue and dequeue from the head. Each thread must first gain mutual exclusion through a single lock in order to access the queue, which can become a bottleneck.

Multiple queues: Instead of a single queue, we can use multiple queues, one dedicated to each thread. We then would have multiple locks, one for each queue. A thread will only dequeue from the head of its dedicated queue. Threads can potentially enqueue tasks to another thread's queue depending on the task to thread assignment. In such a case, a thread will need to acquire the opposing thread's lock for which a task is assigned in order to enqueue the task.

One particular task to thread assignment is data affinity where a thread executes all tasks that overwrite a particular block [10]. We can then assign blocks to threads using a two-dimensional block cyclic distribution (2D). Each contiguously stored block has its index within the larger matrix of blocks for which we use to compute 2D data affinity.

Work stealing: Multiple queues eliminate the bottleneck of the single queue, but depending on the task to thread assignment, wide load inbalances may occur. Work stealing was proposed to alleviate this problem when using multiple queues. If a thread is idle, it will attempt to *steal* a task from a randomly selected thread's queue, which is highlighted in Figure 3 (left).

When using work stealing, a thread will always enqueue dependent tasks to the tail of its own dedicated queue. All the initial ready tasks will be assigned to a particular queue, usually to the first thread. If an idle thread's queue is empty, it will attempt to steal a task by acquiring the lock of the randomly selected thread and dequeue a task from the tail of that queue, as opposed to the head, and then enqueue it onto the tail of its own queue. As usual, all threads will dequeue from the head of queue to obtain tasks to execute.

Priority queues: Each of the three scheduling algorithms use FIFO queues where tasks are essentially sorted by their arrival times. We can use priority queues sorted by different heuristics. Whenever we enqueue a task, we will use insertion sort in this case.

One particular heuristic is the height of each task within the DAG where we define height as the distance from a node to its farthest leaf. A leaf is a node without any outgoing edges. This heuristic optimizes for tasks on the critical path of execution. For instance in Figure 2 (bottom right), we would want to execute $\textsc{gemm}_4$ before $\textsc{syrk}_5$. We can compute the height of each task during the initial traversal of tasks to find the initial ready ones.

6 Performance

In this section, we show the performance of the SuperMatrix runtime system and compare it to a high-performance dense linear algebra library GotoBLAS 1.26 [14].

6.1 Target architecture

All experiments were performed on a single SMP node of `ranger.tacc.utexas.edu`, which contains 3,936 nodes.[2] Each node consists of four sockets with 2.3 GHz AMD Opteron Quad-Core 64-bit processors with a total of 16 cores providing a theoretical peak performance of 147.2 GFLOPS. Each node contains 32 GB of memory, and each socket has a 2 MB L3 cache, which is shared between the four cores. The OpenMP implementation provided by the Intel compiler 10.1 served as the underlying threading mechanism used by SuperMatrix, yet SuperMatrix is also implemented using POSIX Threads. Performance was measured by linking to the GotoBLAS library as a high-performance implementation of BLAS for the execution of individual tasks.

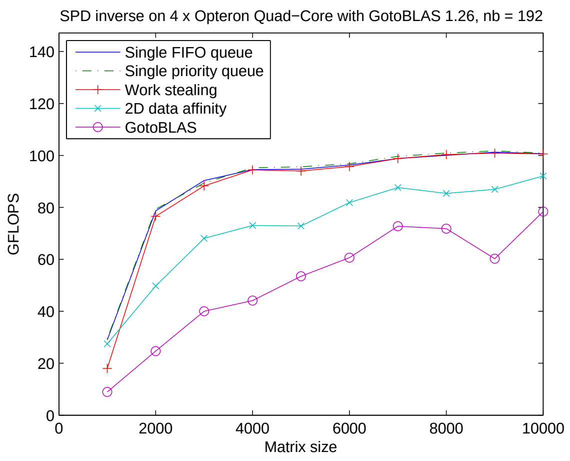
6.2 Implementations

We report the performance (in GFLOPS) for implementations of inversion of an SPD matrix from two libraries using double-precision floating-point arithmetic. 16 threads mapped to each processor and a storage block size of 192 were used for all experiments when possible.

SuperMatrix: We used the SuperMatrix runtime system embedded within `libflame`. We gathered results using the various scheduling algorithms implemented by SuperMatrix: single FIFO queue, single priority queue sorted by the heights of each task in the DAG, work stealing, and 2D data affinity.

GotoBLAS: We compare SuperMatrix with the highly optimized GotoBLAS library. GotoBLAS also provides LAPACK functionality on top of its namesake. We gathered results with multithreaded implementations of `dpotrf`, followed by `dtrtri`, and then `dlauum`, which correspond to LAPACK implementations of CHOL, TRINV, and TTMM, to compute SPD inverse. Since each of these operations are parallelized individually by GotoBLAS, there exist

[2]We thank Kent Milfeld for access to `ranger.tacc.utexas.edu`.



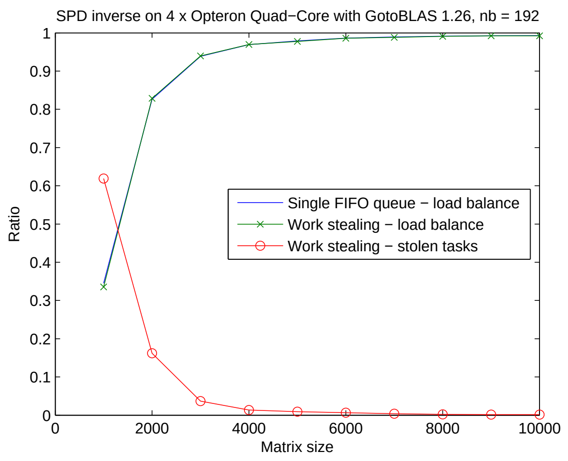


Figure 4: Performance of inversion of an SPD matrix using different scheduling algorithms implemented in SuperMatrix compared against GotoBLAS (left) and ratios of threads performing useful computation to the total parallel execution time and the number of steals with the total number of tasks (right).

synchronization points between each operation, and no parallelism is exploited between the three operations.

6.3 Results

Performance results are reported in Figure 4 (left). Several comments are in order:

- The GotoBLAS implementation of inversion of an SPD matrix does not perform as well as the SuperMatrix implementations because many opportunities to exploit parallelism between the three operations are lost when parallelizing each operation individually.

- The single priority queue attains the same performance signature as the single FIFO queue. These matrix operations perform $O(n^3)$ operations for $O(n^2)$ data where n is the matrix dimension. As a result, SPD creates a rather large DAG of tasks from which there is an abundant amount of parallelism that SuperMatrix can exploit. Sorting the tasks according to its height may reduce the critical path, but in this case the total number of tasks divided by the number of threads is far greater than the critical path length. For example, SPD inverse on a $5,000 \times 5,000$ matrix has $10,962$ tasks yet a critical path length of 135.

- The single FIFO queue and work stealing also exhibit the same performance signature since both scheduling algorithms optimize for load balancing. Since the computational cost of executing a task consisting of a matrix operation on 192×192 submatrix blocks far greatly outweighs the cost of gaining mutual exclusion, the advantages of using multiple queues with work stealing are amortized and mitigated over the entire execution of tasks, which we discuss further in Section 6.4.

- 2D data affinity performs quite poorly compared to the other scheduling algorithms because of the relatively small size of the L3 cache on each socket. Since data affinity attempts to optimize for data locality, there is a very small probability that the blocks being updated currently reside on the cache of the processor executing the task. Since using multiple queues may lead to load inbalances, the lack of data locality on this system adversely affects performance.

We have developed a scheduling algorithm that attempts to optimize for both load balance and data locality simultaneously by augmenting to a single queue implementation within the SuperMatrix framework, which attains a significant performance increase, but that is beyond the scope of this paper. The prefetching and thread preference optimizations give a slight performance increase of about 1 GFLOPS for each when augmented to the single FIFO queue. While these are minimal performance gains considering an unoptimized single FIFO queue attains over 100 GLOPS, these optimizations greatly benefit the new scheduling algorithm presented in [8].

We have also found that sorting priority queues via the height of each node usually provides the best performance compared to other heuristics such as the number of dependent tasks or the total number of tasks in its dependent sub-graph.

The analyzer phase is essentially sequential overhead that can potentially limit speedups via Amdahl's law. We measured the time spent during the dispatcher phase and found that for SPD inverse the percentage of time SuperMatrix incurred sequential execution out of the total execution time was only around 2% for all problem sizes. Also the space required for storing the DAG might become prohibitive, but for instance given SPD inverse on a $5,000 \times 5,000$ matrix, the $10,962$ tasks take up 3.76 MB, yet that is quite small compared to the 190.73 MB required just to store the matrix.

6.4 Work stealing analysis

In Figure 4 (right) we show the ratios for load balance where we computed the average time each thread performed useful computation, which only consists of executing a task, and divided that by the total time spent during the dispatcher phase. As this ratio converges to one, then there is near perfect load balance since the threads spend the entire time during dispatcher only executing tasks. As we can clearly see, both the work stealing and single FIFO queue implementations attain the exact same load balancing properties and hence the same performance signatures in Figure 4 (left).

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 a task to a thread's mailboxes.

We also show the ratio of number of stolen tasks compared to the total number of tasks in Figure 4 (right). The number of steals for SPD inverse on a $1,000 \times 1,000$ matrix is so high because there are only 168 tasks to execute, and at the initial stages of the dispatcher, threads must continuously attempt to steal tasks to execute since all initial ready tasks are assigned to a single thread. Once a steady state is reached where all threads have a sufficient amount of tasks on their queues, the number of steals incurred converges close to zero. As a result, the mailbox optimization for work stealing will not provide any performance gains since the only time the mailbox is accessed is in the event of a steal. Even if every mailbox resulted in better data locality, which is unlikely since those steals occur at the beginning of the dispatcher when the caches of each processor are cold, then less than one percent of the tasks would benefit for problem sizes greater than $5,000 \times 5,000$.

Prefetching may be applied to work stealing to alleviate cache misses incurred by the first tasks executed regardless of the use of the mailbox, but thread preference cannot be applied to work stealing since its default behavior is to enqueue a task to the same thread that executes that task. Despite this fact, work stealing can potentially negate this behavior by allowing threads to steal a task that could benefit from thread preference.

Despite providing the same performance as the single FIFO queue, work stealing adds a level of non-determinism to scheduling to where we cannot effectively predict where tasks will execute. The new scheduling algorithm in [8] limits the threads to which tasks can execute in order to achieve better locality. This type of scheduling would be, at best, difficult to replicate within the framework of work stealing.

7 Related Work

SMP Superscalar (SMPSs) [18] 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 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 [9].

Several other general-purpose parallel schedulers exist such as Cilk [7] and Intel Thread Building Blocks (TBB) [16], both of which also use work stealing, and TBB implements the mailbox optimization [20]. These systems do not create a DAG from data dependencies found in the computation but rather employ directives in the code that allow for routines to specify its dependent tasks. This approach exploits control-level parallelism, which is specified by the order in which operations appear in the code, and handles divide and conquer algorithms quite well. SMPSs extends the scheduling ideas first developed for Cilk to handle data flow parallelism. In this paper, we attempt to analyze and compare the scheduling algorithms themselves as opposed to the implementations of these different systems.

8 Conclusion

In this paper, we have shown the interface and implementation of the SuperMatrix runtime system. We believe that the implementation is quite portable since we only rely upon an underlying threading mechanism to spawn and join threads along with acquiring and releasing multiple locks. We have also shown that SuperMatrix is quite flexible since it can quite easily facilitate the use of different scheduling algorithms and heuristics, such as priority queues and work stealing, strictly within the enqueue and dequeue routines. We compared work stealing with a single FIFO queue implementation by providing results where both attain the same performance since both only optimize for load balance. We also showed that optimizations, such as thread preference, can be more easily augmented to a single queue implementation to provide better performance than work stealing. Finally, the performance results demonstrate that exploiting parallelism at a coarser-level of granularity enables the more efficient use of multicore architectures.

References

[1] Umut A. Acar, Guy E. Blelloch, and Robert 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, 2000.

[2] 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.

[3] Paolo Bientinesi, John A. Gunnels, Margaret E. Myers, Enrique S. Quintana-Ortí, and Robert A. van de Geijn. The science of deriving dense linear algebra algorithms. *ACM Transactions on Mathematical Software*, 31(1):1–26, 2005.

[4] Paolo Bientinesi, Brian Gunter, and Robert A. Van de Geijn. Families of algorithms related to the inversion of a symmetric positive definite matrix. *ACM Transactions on Mathematical Software*, 35(1):3:1–3:22, 2008.

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

[6] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. *Journal of the ACM*, 46(5):720–748, 1999.

[7] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. *Journal of Parallel and Distributed Computing*, 37(1):55–69, 1996.

[8] Ernie Chan. Runtime data flow scheduling of matrix computations. Technical report, The University of Texas at Austin, Department of Computer Sciences, 2009. Submitted to *ASPLOS '10: The Fifteenth International Conference on Architectural Support for Programming Languages and Operating Systems.*

[9] Ernie Chan, Jim Nagle, Field G. Van Zee, and Robert 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, 2009.

[10] Ernie Chan, Enrique S. Quintana-Ortí, Gregorio Quintana-Ortí, and Robert 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, 2007.

[11] Ernie Chan, Field G. Van Zee, Paolo Bientinesi, Enrique S. Quintana-Ortí, Gregorio Quintana-Ortí, and Robert 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, 2008.

[12] Jack J. Dongarra, Jeremy Du Croz, Sven Hammarling, and Iain Duff. A set of level 3 basic linear algebra subprograms. *ACM Transactions on Mathematical Software*, 16(1):1–17, 1990.

[13] Erik Elmroth, Fred Gustavson, Isak Jonsson, and Bo Kagstrom. Recursive blocked algorithms and hybrid data structures for dense matrix library software. *SIAM Review*, 46(1):3–45, 2004.

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

[15] John A. Gunnels, Fred G. Gustavson, Greg M. Henry, and Robert A. van de Geijn. FLAME: Formal linear algebra methods environment. *ACM Transactions on Mathematical Software*, 27(4):422–455, 2001.

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

[17] Tze Meng Low and Robert 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, 2004.

[18] Josep M. Perez, Rosa M. Badia, and Jesus 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, 2008.

[19] Gregorio Quintana-Ortí, Enrique S. Quintana-Ortí, Robert A. van de Geijn, Field G. Van Zee, and Ernie Chan. Programming matrix algorithms-by-blocks for thread-level parallelism. *ACM Transactions on Mathematical Software*, 36(3):14:1–14:26, 2009.

[20] Arch Robison, Michael Voss, and Alexey 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, 2008.

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