Managing the Complexity of Lookahead for LU Factorization with Pivoting

Ernie Chan and Robert van de Geijn
Department of Computer Sciences
The University of Texas at Austin
Austin, Texas 78712
{echan,rvdg}@cs.utexas.edu

Andrew Chapman
Microsoft Corporation
One Microsoft Way
Redmond, Washington 98052
andrew.chapman@microsoft.com

ABSTRACT

We describe parallel implementations of LU factorization with pivoting for multicore architectures. Implementations that differ in two different dimensions are discussed: (1) using classical partial pivoting versus recently proposed incremental pivoting and (2) extracting parallelism only within the Basic Linear Algebra Subprograms versus building and scheduling a directed acyclic graph of tasks. Performance comparisons are given on two different systems.

Categories and Subject Descriptors

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

General Terms

Algorithms, Performance

Keywords

LU factorization with partial pivoting, algorithm-by-blocks, directed acyclic graph, lookahead

1. INTRODUCTION

LU factorization with partial pivoting is simultaneously perhaps the most important operation for solving linear systems and often the most difficult one to parallelize due to the pivoting step. In this paper, we compare different strategies for exploiting shared-memory parallelism when implementing this operation. A simple approach is to link to multithreaded Basic Linear Algebra Subprograms (BLAS) [11] libraries. A strategy that requires nontrivial changes to libraries like Linear Algebra PACKage (LAPACK) [2] is to add *lookahead* to classical LU factorization with partial pivoting. A recently proposed *algorithm-by-blocks* with *incremental pivoting* [5, 26] changes the pivoting strategy to increase opportunities for parallelism, at some expense to the numerical stability of the algorithm. To manage the resulting complexity, we introduced the SuperMatrix runtime system [8] as a general solution for parallelizing LU factorization with pivoting, which maps an algorithm-by-blocks to a directed acyclic graph (DAG) and schedules the tasks from the DAG in parallel. This approach solves the programmability issue that faces us with the introduction of multicore architectures by separating the generation of a DAG to be executed from the scheduling of tasks.

The contributions of the present paper include:

- An implementation of classical LU factorization with partial pivoting within a framework that separates programmability issues from the runtime scheduling of a DAG of tasks.
- A comparison of different pivoting strategies for LU factorization.

Together these contributions provide further evidence that the SuperMatrix runtime system solves the problem of programmability while providing impressive performance.

In our previous SPAA paper [7], we first introduced this concept of using out-of-order scheduling to parallelize matrix computation using the Cholesky factorization as a motivating example, an operation which directly maps to an algorithm-by-blocks. On the other hand, LU factorization with partial pivoting does not easily map well to an algorithm-by-blocks. Our solution addresses programmability since we can use the same methodology to parallelize this more complex operation without adding any extra complexity to the code that implements LU factorization with partial pivoting.

The rest of the paper is organized as follows. In Section 2, we present LU factorization with partial pivoting and several traditional methods for parallelizing the operation. We describe the SuperMatrix runtime system in Section 3. In Section 4, we describe LU factorization with incremental pivoting and its counterpart for QR factorization. Section 5 provides performance results, and we conclude the paper in Section 6.

2. LU FACTORIZATION WITH PARTIAL PIVOTING

We present the right-looking unblocked and blocked algorithms for computing the LU factorization with partial pivoting using standard Formal Linear Algebra Method Environment (FLAME) notation [16] in Figure 1. The thick and thin lines have semantic meaning and capture how the algorithms move through the matrix where the symbolic partitions reference different submatrices on which computation occurs within each iteration of the loop.

We first describe the updates performed within the loop of the unblocked algorithm. The SWAP routine takes the vector $\begin{pmatrix} \alpha_{11} \\ a_{21} \end{pmatrix}$, finds the index of the element with the largest magnitude in that vector, which is stored in π_1, and exchanges that element with α_{11}. Next, the pivot is applied (PIV) where the rest of the π_1-th row is

SPAA'10, June 13–15, 2010, Thira, Santorini, Greece.

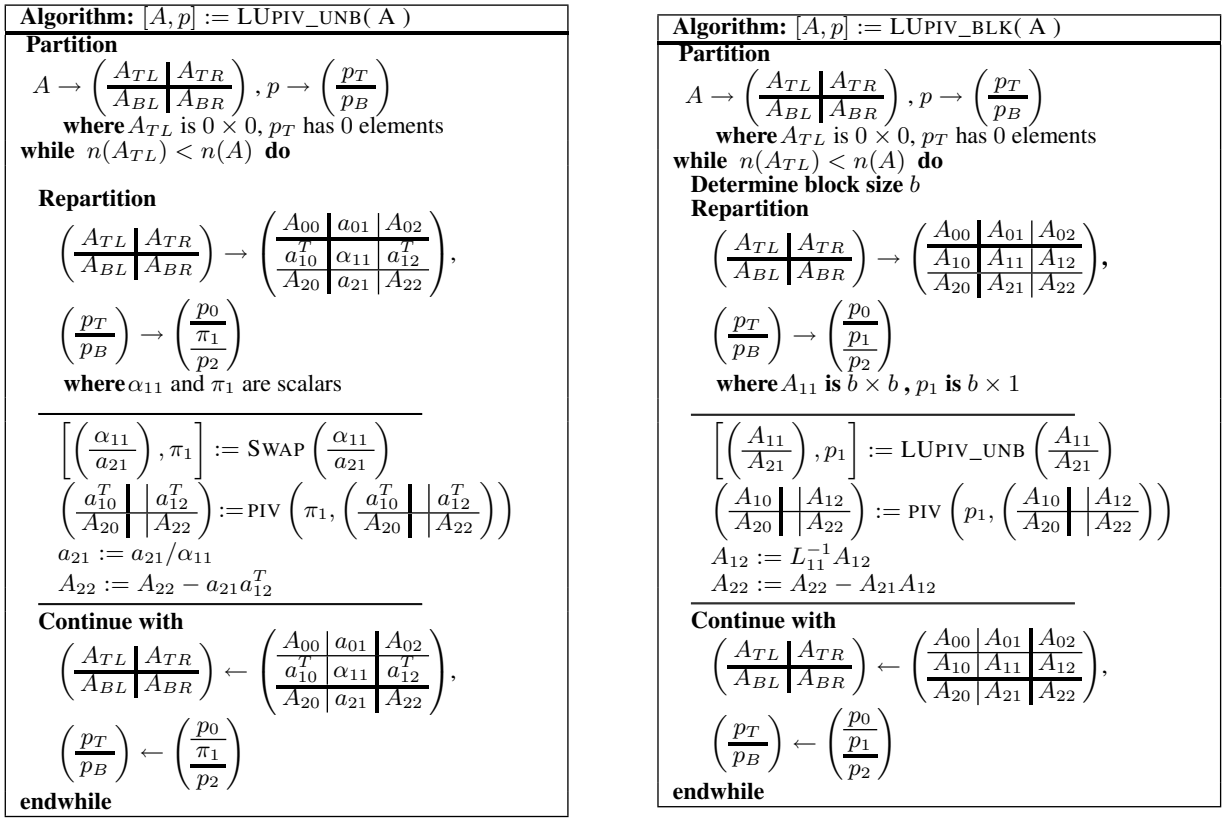

Figure 1: The right-looking unblocked and blocked algorithms (left and right, respectively) for computing the LU factorization with partial pivoting. Here the matrix is pivoted like LAPACK does so that $\text{PIV}(p, A) = LU$ upon completion. In this figure, L_{ii} denotes the unit lower triangular matrix stored over A_{ii}, and $n(A)$ stands for the number of columns of A.

interchanged with $\left(\begin{array}{c|c} a_{10}^T & a_{12}^T \end{array}\right)$. Finally, a_{21} is scaled by $\frac{1}{\alpha_{11}}$, and a rank-one update is performed over A_{22}.

In the blocked algorithm, the LU factorization (LUPIV) subproblem calls the unblocked algorithm, which updates the column panel $\left(\begin{array}{c} A_{11} \\ \hline A_{21} \end{array}\right)$ and stores all the pivot indices in p_1. We then apply all of those pivots to the left and right of the current column panel. Next, a triangular solve with multiple right-hand sides (TRSM) is performed over A_{12} with L_{11}, which is the unit lower triangular matrix of A_{11}. Finally, A_{22} is updated with general matrix-matrix multiplication (GEMM). Both TRSM and GEMM are examples of level-3 BLAS operations [11].

The problem instances of TRSM and GEMM incurred within this right-looking blocked algorithm are quite easily parallelized. The bulk of the computation in each iteration lies in the GEMM call so that straight forward implementations (e.g., LAPACK's blocked implementation `dgetrf`) can exploit parallelism by only linking to multithreaded BLAS libraries and thus attain high performance. As such, many opportunities for parallelism are lost since implicit synchronization points exist between each call to a parallelized BLAS routine.

2.1 Algorithm-by-blocks

By storing matrices hierarchically [12] and viewing submatrix blocks as the unit of data and operations with blocks (tasks) as the unit of computation, we reintroduced the concept of algorithms-by-blocks [19]. We reformulate the blocked algorithm presented in Figure 1 (right) as an algorithm-by-blocks in Figure 2 (left) using the FLASH [21] extension to the FLAME application programming interface (API) for the C programming language [3] for creating and accessing hierarchical matrices. Notice that the FLAME/C and FLASH application programming interfaces were typeset to closely resemble the FLAME notation and thus easily facilitate the translation from algorithm to implementation. We assume that the matrix A and the vector p are both stored hierarchically with one level of blocking. This storage scheme has an additional benefit in that spatial locality is maintained when accessing the contiguously stored submatrices.

The matrix object `A` Figure 2 (left) is itself encoded as a matrix of matrices in FLASH where the top-level object consists of references to the submatrix blocks. We stride through the matrix using a unit block size while decomposing the subproblems into operations on individual blocks. The algorithmic block size b in Figure 1 (right) now manifests itself as the storage block size of each contiguously stored submatrix block.

In Figure 2 (right), we illustrate the tasks that overwrite each block in a 3×3 matrix of blocks within each iteration of the loop in Figure 2 (left). We will use the notation $A_{i,j}$ to denote the i, j-th block within the matrix of blocks.

In the first iteration, we perform the task LUPIV_0 on the entire left column panel of the matrix where the symbolic partition A_{11} references $A_{0,0}$, and A_{21} references $A_{1,0}$ and $A_{2,0}$. For convenience, we choose to make each LUPIV task updating a column panel of blocks an atomic operation because it cannot be easily partitioned into finer-grained tasks operating on individual blocks. For example, if we call the unblocked algorithm to perform LUPIV_0, a

```
1  FLA_Error FLASH_LU_piv_blk( FLA_Obj A, FLA_Obj p )
2  {
3     FLA_Obj ATL, ATR,   A00, A01, A02,     pT,   p0,
4             ABL, ABR,   A10, A11, A12,     pB,   p1,
5                         A20, A21, A22,           p2,
6                         AB0, AB1, AB2;
7
8     FLA_Part_2x2( A,     &ATL, &ATR,
9                        &ABL, &ABR,     0, 0, FLA_TL );
10    FLA_Part_2x1( p,     &pT,
11                       &pB,     0, FLA_TOP );
12
13    while ( FLA_Obj_width( ATL ) < FLA_Obj_width( A ) )
14    {
15       FLA_Repart_2x2_to_3x3(
16          ATL, /**/ ATR,       &A00, /**/ &A01, &A02,
17       /* ************* */   /* ******************** */
18                               &A10, /**/ &A11, &A12,
19          ABL, /**/ ABR,       &A20, /**/ &A21, &A22,
20          1, 1, FLA_BR );
21       FLA_Repart_2x1_to_3x1( pT,       &p0,
22                           /* ** */   /* ** */
23                                      &p1,
24                              pB,      &p2,
25                              1, FLA_BOTTOM );
26       /*-----------------------------------------------*/
27       FLA_Merge_2x1( A11,
28                      A21,     &AB1 );
29       FLASH_LU_piv( AB1, p1 );
30       FLA_Merge_2x1( A10,
31                      A20,     &AB0 );
32       FLASH_Apply_pivots( FLA_LEFT, FLA_NO_TRANSPOSE,
33                           p1, AB0 );
34       FLA_Merge_2x1( A12,
35                      A22,     &AB2 );
36       FLASH_Apply_pivots( FLA_LEFT, FLA_NO_TRANSPOSE,
37                           p1, AB2 );
38       FLASH_Trsm( FLA_LEFT, FLA_LOWER_TRIANGULAR,
39                   FLA_NO_TRANSPOSE, FLA_UNIT_DIAG,
40                   FLA_ONE, A11, A12 );
41       FLASH_Gemm( FLA_NO_TRANSPOSE, FLA_NO_TRANSPOSE,
42                   FLA_MINUS_ONE, A21, A12, FLA_ONE, A22 );
43       /*-----------------------------------------------*/
44       FLA_Cont_with_3x3_to_2x2(
45          &ATL, /**/ &ATR,         A00, A01, /**/ A02,
46                                   A10, A11, /**/ A12,
47        /* ************** */   /* ****************** */
48          &ABL, /**/ &ABR,         A20, A21, /**/ A22,
49          FLA_TL );
50       FLA_Cont_with_3x1_to_2x1( &pT,        p0,
51                                 p1,
52                               /* ** */   /* ** */
53                                 &pB,       p2,
54                                 FLA_TOP );
55    }
56    return FLA_SUCCESS;
57 }
```

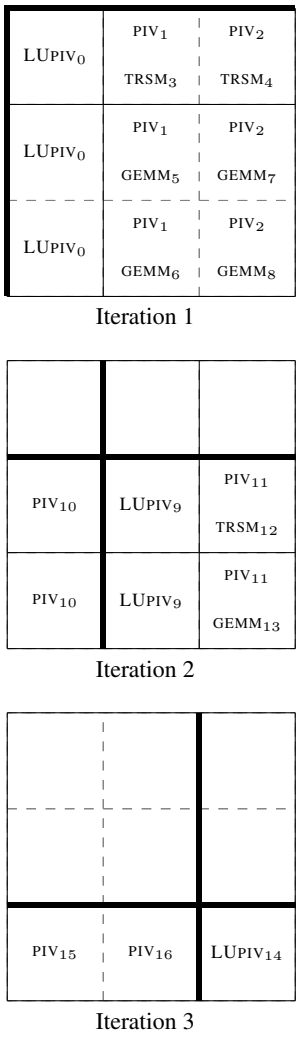

Figure 2: Left: The FLASH implementation of LU factorization with partial pivoting. Right: The tasks that overwrite each block in every iteration within LU factorization with partial pivoting on a 3×3 matrix of blocks. The subscripts denote the order in which the tasks can be sequentially executed within the algorithm-by-blocks. Certain blocks have two tasks overwriting it where the task pictured on top must be executed first.

pivot may occur within $A_{2,0}$ in one iteration and then within $A_{1,0}$ within the next iteration.

The application of the pivots is partitioned into separate tasks updating each of the two column panels to the right of the current one with the tasks PIV_1 and PIV_2. Just like the LU factorization subproblem, the application of the pivots to a column panel is done atomically.

The TRSM update of A_{12} is partitioned into two independent tasks. TRSM_3 overwrites $A_{0,1}$, which was previously updated by PIV_1, so PIV_1 must complete execution before TRSM_3 can begin. This situation is an example of a flow dependency (read-after-write) between these two tasks where one task reads the output of another task, which also occurs for PIV_2 and TRSM_4 on $A_{0,2}$. The update of $A_{2,2}$ is decomposed into four independent tasks overwriting $A_{1,1}$, $A_{1,2}$, $A_{2,1}$, and $A_{2,2}$. The same flow dependencies occur between the GEMM and PIV tasks.

Clearly, this approach generalizes to matrices partitioned with a large number of blocks.

2.2 Lookahead

The difficulty in exploiting a coarser granularity of parallelism from LU factorization with partial pivoting lies with each LUPIV task and the resulting pivoting across the entire matrix, which become bottlenecks within each iteration. In order to alleviate this problem, lookahead is used, which is also called compute-ahead in the literature [1, 27]. An early study of look-ahead for distributed-memory parallelization of LU factorization can already be found in [13]. For this operation, the update of A_{22} is subdivided and partially computed so that the LUPIV from the next iteration can be performed ahead of the current iteration in parallel with the rest of the update to A_{22}.

For example in Figure 2 (right), PIV_1 and PIV_2 are independent and thus can be executed in parallel along with TRSM_3 and TRSM_4. GEMM_5, GEMM_6, GEMM_7, and GEMM_8 are also all independent of each other. In order to apply lookahead, we first schedule PIV_1, TRSM_3, and then GEMM_5 and GEMM_6 to execute first. Once those tasks are complete, we can then schedule LUPIV_9 to execute alongside PIV_2, TRSM_5, and then GEMM_7 and GEMM_8.

The difficulty with traditional approaches for implementing lookahead is that the code becomes obfuscated and quite complex. Applying this technique to a wide range of different linear algebra algorithms has not been done because this solution does not inherently address programmability. First, finding the inherent bottleneck to computation is nontrivial. Second, partitioning the rest of the computation in order to allow the next iteration to start is often quite difficult. Moreover, lookahead can be applied to more than one iteration, which further complicates the code.

3. SUPERMATRIX RUNTIME SYSTEM

Instead of exposing the details of parallelization, like lookahead, within the code that implements the linear algebra operation, 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 for which only the input and output matrix operands of each task are needed to perform the dependence analysis. Tasks represent the nodes of the graph and data dependencies between tasks represent the edges [7, 8]. Once the analyzer is done, the dispatcher phase is invoked which dispatches and schedules tasks to threads.

3.1 Analyzer

The FLASH code in Figure 2 (left) invokes the analyzer phase that generates the DAG of tasks. Only flow dependencies occur between tasks from LU factorization with partial pivoting. For example, the analyzer stores tasks as depicted in Figure 2 (right) given a 3×3 matrix of blocks and constructs the resulting DAG shown in Figure 3 (left).

In nearly all the linear algebra operation we have studied thus far [26], each operation is decomposed into tasks where each matrix operand consists of a single block, such as TRSM and GEMM. By contrast, LUPIV and PIV require that a matrix operand be a column panel of blocks, which was not previously supported by the SuperMatrix runtime system. We introduce the concept of *macroblocks*, which is a matrix partition representing several blocks, so a task's matrix operand can either be a single block or a macroblock. For instance, LUPIV_0 updates the macroblock consisting of $A_{0,0}$, $A_{1,0}$, and $A_{2,0}$. This macroblock mechanism is needed in order to properly detect data dependencies between tasks and thus correctly construct the DAG. Once the DAG is built, the dispatcher does not need knowledge of a task updating a single block or macroblock and hence a further separation of concerns.

3.2 Dispatcher

Once the DAG is constructed, the dispatcher is invoked in order to dynamically schedule the tasks in parallel. In [6], we discussed several different scheduling algorithms and heuristics. In this paper we will only use a single queue implementation from which all threads enqueue and dequeue ready tasks. We define ready tasks as ones where all of its dependent tasks have been executed, so every task residing on the queue can be executed in parallel.

LUPIV_0 is the only initial ready task within the DAG in Figure 3 (left) and thus is enqueued. Once a thread dequeues that task from the single queue and executes it, then PIV_1 and PIV_2, both of which are the dependent tasks of LUPIV_0, become ready and get enqueued. The dispatcher continues this process until all tasks have been executed.

A typical scheduling of tasks is to use a simple first-in first-out (FIFO) queue ordering. Since a thread can choose any task on the queue to dequeue instead of strictly the one at the head of the queue, we can apply different heuristics to either improve the load balance or data locality of tasks executed by each thread.

One particular heuristic is sorting the queue according to the heights of each task within the DAG. The height of a task is the distance between itself and its farthest leaf where a leaf is a task in the DAG without any dependent tasks. Conversely, a root is a task that does not depend upon any other tasks. In Figure 3 (left), LUPIV_0 is the only root, and PIV_{15} and PIV_{16} are leaves. The height of a root in the DAG represents the critical path of execution. By sorting the queue with this heuristic and having each thread dequeue from the head of queue, we attempt to schedule tasks on the critical path of execution and thus reduce the time to execute all tasks.

This height sorting heuristic mimics the concept of lookahead. In Figure 3 (left), GEMM_5 and GEMM_6 both have a height of six while GEMM_7 and GEMM_8 have a height of five. By sorting the tasks, threads will execute GEMM_5 and GEMM_6 first and then potentially execute LUPIV_9 in parallel with GEMM_7 and GEMM_8 since all three of those tasks have the same height.

An algorithm-by-tiles similar to our algorithm-by-blocks had already been proposed for a parallel out-of-core LU factorization in [28]. In that implementation, parallelism is extracted within operations with tiles that are brought in from disk rather than across operations with tiles, and a DAG of tasks is neither built nor scheduled. Their approach is also similar to ours in that it also implements classical LU factorization with partial pivoting.

4. ALTERNATIVES

We present two alternatives to using LU factorization with partial pivoting as a solution for solving linear systems.

4.1 LU factorization with incremental pivoting

We now review an approach, incremental pivoting, that avoids many of the dependencies exhibited in the classical LU factorization with partial pivoting [23, 24, 26].

While we refer the reader to the aforementioned papers for details, we very briefly give a flavor of how incremental pivoting works. Recall that Gaussian elimination is one formulation of LU factorization. As elements below the diagonal are eliminated in Gaussian elimination, we can choose to swap the current row where a zero is to be introduced with the row being used to eliminate that row. Thus, at each step, the row among such a pair of rows that has the largest magnitude pivot element could be swapped to be used to introduce a zero in the other row after swapping. Incremental pivoting works similarly but with blocks.

In Figure 3 (right), we present the DAG for LU factorization with incremental pivoting on a 3×3 matrix of blocks. Here, each LUPIV task updates the symbolic partition A_{11} which only consists of a single block as opposed to a macroblock. The TRSM tasks update the blocks composing A_{12}, which is similar to TRSM within LU factorization with partial pivoting, but these tasks are prepended with the application of the pivots. LUSA and FSSA are "structurally-aware" kernels that update A_{21} and A_{22}, respectively, which update the matrix according to the LUPIV task performed within each iteration. Notice that the DAG produced from incremental has a shorter critical path of execution and more opportunities for exploiting parallelism within the DAG.

Strictly speaking, LU factorization with partial pivoting is numerically unstable due to the potential for 2^n in element growth where n is the matrix dimension. In practice, it is considered to be stable based on decades of experience. Incremental pivoting is inherently less stable than partial pivoting [23] and has not been used for decades, so it is not known if it is stable in practice.

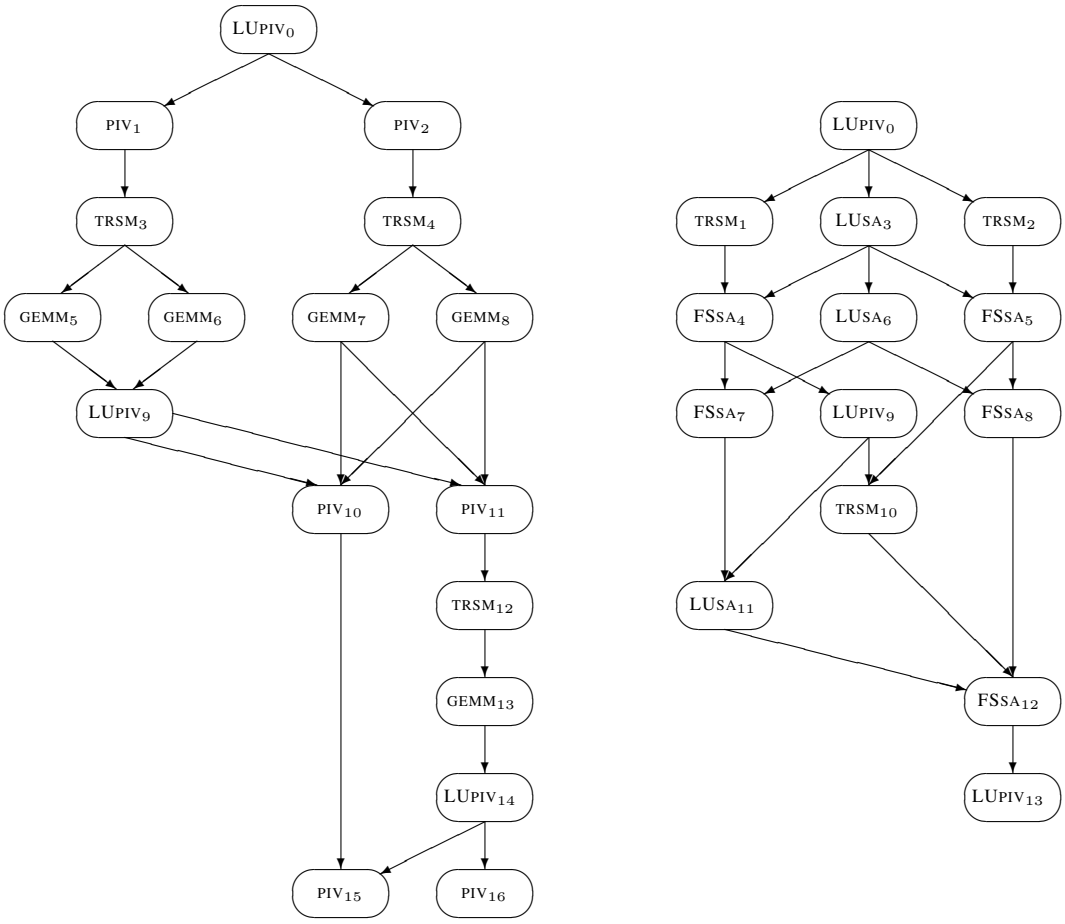

Figure 3: The directed acyclic graphs for LU factorization with partial (left) and incremental (right) pivoting on a 3×3 matrix of blocks.

4.2 QR factorization

QR factorization based on orthogonal transformations, such as Householder transformations, is a numerically stable alternative but requires twice the number of floating point operations than LU factorization.

In [25], two Householder-based algorithms-by-blocks for QR factorization were studied. One was implemented using a classical high-performance QR factorization while the other implemented an incremental scheme, first proposed for out-of-core computation [17], not unlike LU with incremental pivoting. As with LU factorization, the incremental scheme was shown to exhibit more parallelism. What is different is that this incremental QR factorization approach is guaranteed to be stable. Thus, if stability is a concern, this alternative solution method can be employed despite the extra computational cost.

5. PERFORMANCE

We compare the performance of LU factorization with partial and incremental pivoting within the SuperMatrix runtime system with other high-performance libraries and show their performance on two different platforms.

5.1 Target architectures

We performed experiments on a single SMP node of a large distributed-memory cluster `ranger.tacc.utexas.edu` containing 3,936 nodes for which it uses a 2.6.18.8 Linux kernel. 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 C Compiler 10.1 served as the underlying threading mechanism used by SuperMatrix. We linked to GotoBLAS2 1.00 and Intel MKL 10.0.

We also gathered results on a 16 core UMA machine running Windows Server 2008 R2 Enterprise which consists of four sockets with 2.4 GHz Intel Xeon E7330 Quad-Core processors providing a theoretical peak performance of 153.6 GFLOPS. This machine contains 8 GB of memory, and each socket has two 3 MB L2 caches shared between each pair of cores. We compiled using Microsoft Visual C++ 2010 and Intel Visual Fortran 11.1. We linked to GotoBLAS2 1.00 and Intel MKL 10.2.

5.2 Implementations

We report the performance (in GFLOPS, one billion floating point operations per second) for several implementations of LU

factorization using double precision floating point arithmetic. We used an operation count of $\frac{2}{3}n^3$ of useful computation for each implementation presented to calculate the rate of execution. We tuned the storage and algorithmic block size for each problem size when possible, yet we mapped 16 threads to each of the 16 cores on both machines for all experiments.

SuperMatrix:.

We used the SuperMatrix runtime system embedded within the open source library `libflame` [29]. Both LU factorization with partial and incremental pivoting are implemented using SuperMatrix. For the partial pivoting implementation, we used the scheduling heuristic for sorting tasks according to their heights within the DAG as described in Section 3. For the incremental pivoting implementation, we used the cache affinity scheduling algorithm described in [6] which attempts to balance between both data locality and load balance simultaneously. SuperMatrix requires that the matrices are stored hierarchically. We call a serial BLAS library for the execution of tasks on a single thread.

LAPACK + Multithreaded BLAS:.

We linked the sequential implementation of `dgetrf` provided by LAPACK 3.0 to multithreaded BLAS routines from GotoBLAS and MKL. Parallelism is only exploited within each call to a multithreaded BLAS routine. `dgetrf` assumes that the matrices are stored in the traditional "flat" column-major order storage.

Multithreaded GotoBLAS/MKL:.

We linked to the highly optimized multithreaded implementations of `dgetrf` within GotoBLAS and MKL. These implementations exploit parallelism internally.

5.3 Results

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

- For smaller problem sizes with SuperMatrix, LU factorization with incremental pivoting ramps up in performance more quickly than partial pivoting because there are more opportunities for parallelism within the DAG as shown in Figure 3.

  Despite having many more bottlenecks to parallelism, the SuperMatrix implementation of LU factorization with partial pivoting achieves much better performance for asymptotically large problem sizes because the kernels invoked by partial pivoting are much more efficient than the ones for incremental pivoting. The bulk of the computation in partial pivoting lies with calls to GEMM which is a highly-tuned kernel whereas incremental pivoting predominantly calls the structurally-aware tasks LUSA and FSSA, which we have hand coded. In order to implement these tasks, we use an inner algorithmic block size to stride through the matrix operands. For instance the storage block size is typically 192×192, yet we use an inner block size of around 48. As we stride through the matrix operands of each task, we perform computation on submatrix partitions as in a typical FLAME algorithm. BLAS operations are highly tuned for larger problem sizes, such as GEMM performed with 192×192 blocks, as opposed to multiple calls to GEMM and TRSM on much smaller submatrices [14].

  Another difference in asymptotic performance of LU with partial pivoting versus incremental pivoting is due to the fact that the former amortizes $O(n^2)$ operations related to pivoting over $O(n^3)$ operations while the latter amortizes $O(b^2)$ operations related to pivoting over $O(b^3)$ operations where n is the total matrix size and b is the storage block size. Since b is typically fixed as n increases, asymptotically the performance of the algorithm that uses incremental pivoting is slower than that of partial pivoting.

  As the problem sizes grow asymptotically large, the use of sorting to provide lookahead achieves good load balance between threads. Since all the computational kernels invoked by partial pivoting are significantly faster than incremental pivoting, partial pivoting outperforms incremental pivoting despite incremental pivoting having better parallel efficiency. Unfortunately, partial pivoting does not scale as well as incremental pivoting when using more threads because of the inherent bottlenecks within LU factorization with partial pivoting.

- In order to perform the subproblem for LU factorization with partial pivoting, we copy the macroblock from storage-by-blocks to a flat matrix, execute the subproblem using `dgetrf`, and then copy the result back into the original macroblock. We made a small optimization not to copy the matrix for the LU subproblem if there is only one block within the macroblock. To apply the pivots to a macroblock, we hand coded a kernel that is structurally-aware of the storage-by-blocks instead of copying into a flat matrix and calling the optimized LAPACK implementation `dlaswp`.

  Performing the copy for the LU subproblem does not incur a significant performance penalty because that task performs $O(n^2b)$ operations on $O(nb)$ data, so the copying is essentially amortized across the computational cost of the task. Also, the LU subproblem is only invoked once per iteration of the loop on a single column panel in the hierarchically stored matrix A. The application of the pivots is performed on every other column panel within every iteration of the loop. The relative cost of copying a column panel into a flat matrix would be too high due to the small amount of operations done by the application of the pivots.

  This hand coded implementation for applying the pivots is not as highly optimized as the implementation of `dlaswp` provided by both GotoBLAS and MKL, so a performance penalty is incurred when compared to the multithreaded implementations of `dgetrf` provided by both. The SuperMatrix implementations can attain better performance if we develop optimized structurally-aware kernels.

- The sequential implementation of `dgetrf` linked to multithreaded BLAS implementations does not perform as well as the others because of the many lost opportunities for parallelism.

If numerical stability is not an issue, then we can employ incremental pivoting for smaller problem sizes and then switch to partial pivoting for large problem sizes to provide the best performance with SuperMatrix.

In Figure 4 (top left), we also compare the performance of the SuperMatrix implementation of LU factorization with partial pivoting where use a simple FIFO ordering of tasks as opposed to sorting the tasks according to their heights within the DAG. As we can clearly see, sorting tasks, which mimics lookahead, nearly doubles the performance for SuperMatrix. This scheduling heuristic is completely subsumed within the runtime system and is not exposed within the code that implements this operation.

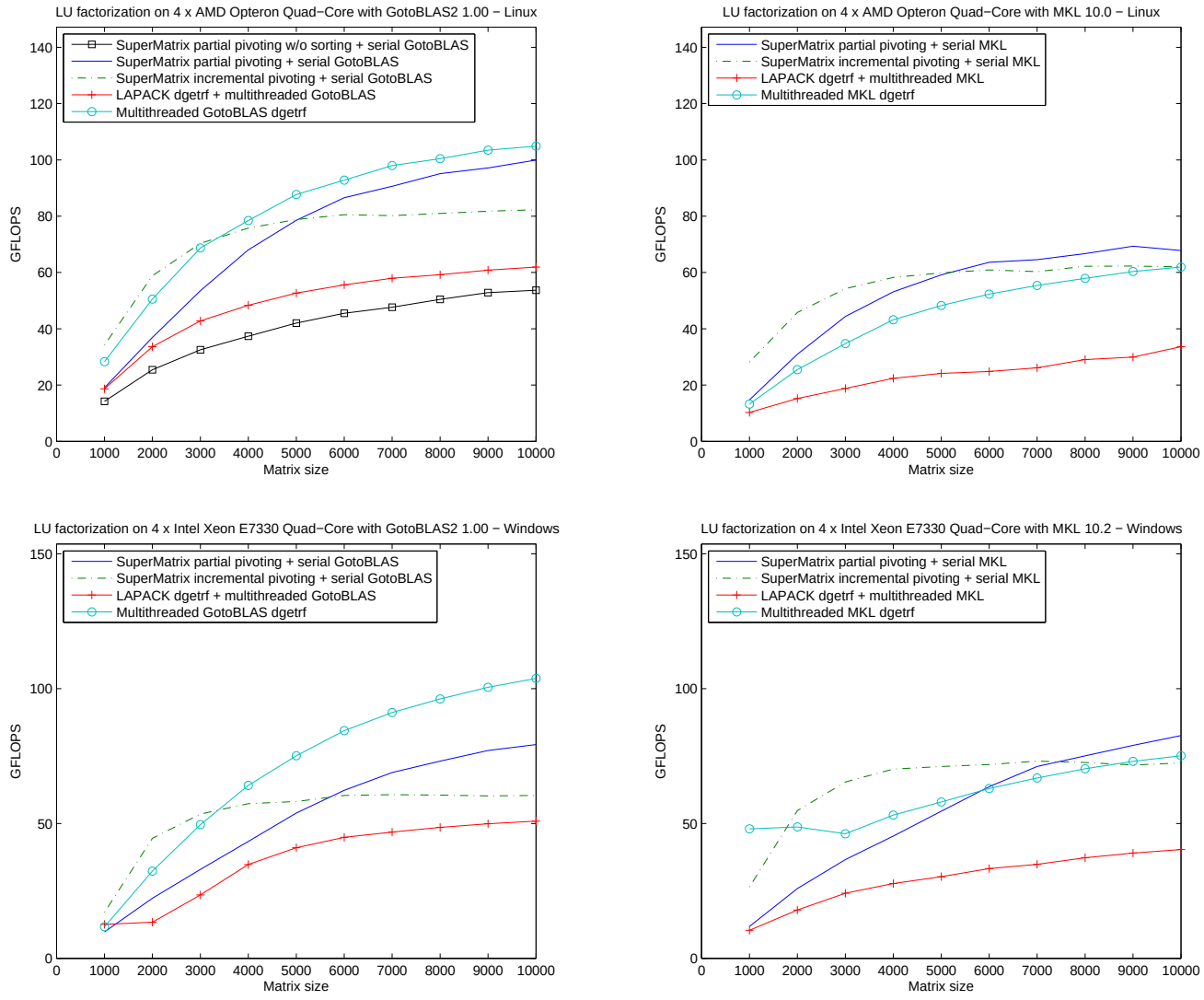

Figure 4: Performance of different implementations of LU factorization on two different platforms.

6. CONCLUSION

In this paper, we have shown a solution for parallelizing LU factorization with partial pivoting that also addresses programmability by separating the runtime system from the code that implements the operation. By using this separation of concerns, we are able to subsume the idea of lookahead seamlessly. As such, this strategy is highly competitive with finely tuned, high-performance implementations provided by commercial libraries.

We believe the SuperMatrix runtime system is the only solution for parallelizing matrix computations that addresses programmability. Hierarchically Tiled Arrays (HTA) [18] and Unified Parallel C (UPC) [20] both provide programming language support for blocked computation, but those two do not perform dependence analysis in order to exploit parallelism between operations within algorithms from which UPC uses lookahead embedded within its code to parallelize LU factorization with partial pivoting. SMP Superscalar (SMPSs) [22] is a general-purpose runtime system that also constructs a DAG using the input and output operands of tasks, but they do not focus on developing algorithms-by-blocks in order to parallelize matrix computations. Parallel Linear Algebra for Scalable Multi-core Architectures (PLASMA) [4, 5] uses a similar DAG scheduling methodology to parallelize matrix computations, but the details of parallelization are not separated from the code from whom LU factorization with incremental pivoting has been implemented thus far and not partial pivoting. SuperLU [9] and High-Performance LINPACK (HPL) [10] both use lookahead in order to parallelize LU factorization with partial pivoting for distributed-memory computer architectures, yet neither addresses programmability since the lookahead strategy is embedded directly within the code that implements this operation. Communication avoiding LU factorization [15] is a fundamentally different pivoting strategy from partial and incremental pivoting since it was designed to limit communication between nodes of distributed-memory architectures.

Additional information

For additional information on FLAME visit

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

7. ACKNOWLEDGMENTS

This research is sponsored by Microsoft Corporation and 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).* We thank the Texas Advanced Computing Center (TACC) for access to their equipment.

8. REFERENCES

[1] C. Addison, Y. Ren, and M. van Waveren. OpenMP issues arising in the development of parallel BLAS and LAPACK libraries. *Scientific Programming*, 11(2):95–104, April 2003.

[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] 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.

[4] 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.

[5] 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.

[6] E. Chan. Runtime data flow scheduling of matrix computations. Technical Report TR-09-27, The University of Texas at Austin, Department of Computer Sciences, August 2009.

[7] 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.

[8] 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.

[9] J. W. Demmel, J. R. Gilbert, and X. S. Li. An asynchronous parallel supernodal algorithm for sparse Gaussian elimination. *SIAM Journal on Matrix Analysis and Applications*, 20(4):915–952, October 1999.

[10] J. J. Dongarra, J. R. Bunch, C. B. Moler, and G. W. Stewart. *LINPACK Users' Guide*. SIAM, Philadelphia, 1979.

[11] 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.

[12] 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.

[13] A. Gerasoulis and I. Nelken. Scheduling linear algebra parallel algorithms on MIMD architectures. In *Proceedings of the Fourth SIAM Conference on Parallel Processing for Scientific Computing*, pages 68–95, Philadelphia, PA, USA, 1990.

[14] 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.

[15] L. Grigori, J. W. Demmel, and H. Xiang. Communication avoiding Gaussian elimination. In *SC '08: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, pages 1–12, Austin, TX, USA, November 2008.

[16] 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.

[17] 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.

[18] J. Guo, G. Bikshandi, B. B. Fraguela, M. J. Garzaran, and D. Padua. Programming with tiles. In *PPoPP '08: Proceedings of the Thirteenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 111–122, Salt Lake City, UT, USA, February 2008.

[19] F. G. Gustavson. New generalized matrix data structures lead to a variety of high-performance algorithms. In *Proceedings of the IFIP TC2/WG2.5 Working Conference on Software Architectures for Scientific Computing Applications*, pages 211–234, Ottawa, ON, Canada, October 2000.

[20] P. Husbands and K. Yelick. Multi-threading and one-sided communication in parallel LU factorization. In *SC '07: Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*, pages 1–10, Reno, NV, USA, November 2007.

[21] T. Meng 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.

[22] 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.

[23] E. S. 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.

[24] 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.

[25] 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.

[26] 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.

[27] P. Strazdins. A comparison of lookahead and algorithmic blocking techniques for parallel matrix factorization. *International Journal of Parallel and Distributed Systems and Networks*, 4(1):26–35, June 2001.

[28] S. Toledo. Locality of reference in LU decomposition with partial pivoting. *SIAM Journal on Matrix Analysis and Applications*, 18(4):1065–1081, October 1997.

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