Multi-Threading and One-Sided Communication in Parallel LU Factorization

Parry Husbands
Lawrence Berkeley National Laboratory

1 Cyclotron Road
Berkeley, CA 94720 USA
+1 510 486 4378
pjrhusbands@lbl.gov

Katherine Yelick
Lawrence Berkeley National Laboratory and
University of California at Berkeley
777 Soda Hall
Berkeley, CA 94720 USA
+1 510 642 8900
yelick@eecs.berkeley.edu

ABSTRACT
Dense LU factorization has a high ratio of computation to communication and, as evidenced by the High Performance Linpack (HPL) benchmark, this property makes it scale well on most parallel machines. Nevertheless, the standard algorithm for this problem has non-trivial dependence patterns which limit parallelism, and local computations require large matrices in order to achieve good single processor performance. We present an alternative programming model for this type of problem, which combines UPC's global address space with lightweight multithreading. We introduce the concept of memory-constrained lookahead where the amount of concurrency managed by each processor is controlled by the amount of memory available. We implement novel techniques for steering the computation to optimize for high performance and demonstrate the scalability and portability of UPC with Teraflop level performance on some machines, comparing favourably to other state-of-the-art MPI codes.

Categories and Subject Descriptors
D.1.3 [**Parallel Programming**]

General Terms
Algorithms, Performance, Languages.

Keywords
Multithreading, latency tolerance, dense linear algebra.

SC07 November 10-16, 2007, Reno, Nevada, USA
(c) 2007 ACM 978-1-59593-764-3/07/0011…$5.00

1. INTRODUCTION
Much of scientific computation is organized into a bulk-synchronous model having distinct phases of communication and computation. In this paper we describe a parallel execution model for a problem that is not naturally bulk-synchronous, namely matrix factorization. We start with the UPC language [14][11][34], which has one-sided communication via its global address space, locality control through the partitioning of the address space, and a static parallelism model with barrier synchronization which lends itself well to a bulk-synchronous style. We then explore extensions of the basic UPC execution model to better support problems such as matrix factorization with interesting dependence patterns.

The long term goal of our project is to develop highly optimized matrix factorization routines for both dense and sparse matrices. In addition, the UPC community is exploring possible extensions to UPC to improve productivity and performance. In this paper we use dense LU factorization, which is simpler than the sparse case, but still has nontrivial dependences that lead to many possible parallel schedules and some challenges for Partitioned Global Address Space (PGAS) languages such as UPC. In addition, the high computational intensity of dense LU factorization means that arithmetic unit utilization should be very high, and the prevalence of LU performance data across machines from benchmark implementations of LU [20][25][33] sets a high standard for success. For an arbitrarily large dense matrix, the high computation to communication ratio leads to a computation that scales with machine size and processor performance, as long as the input matrix is large enough to mask communication, memory, and synchronization costs. For small matrices or for sparse ones, the dependencies inherent in the algorithm can result in poor scaling due to memory costs, communication overhead, synchronization, and load imbalance.

Two of the most common parallel LU factorization codes for distributed memory machines are from the ScaLAPACK library [12] and the High Performance Linpack (HPL) benchmark used in determining the Top 500 list [33]. Both of these codes are written for portability and scalability using the two-sided message passing model in MPI [32], and are written to keep the processors somewhat synchronized in order to manage the matching of sends and receives and the associated buffer space for messages. The ScaLAPACK code synchronizes for each distinct phase of the algorithm, while the HPL code allows for a statically determined amount of communication and algorithmic overlap. In this paper we present an alternative parallelization strategy for LU based on

the PGAS model in UPC augmented with an event-driven multithreaded execution model. The global address space provides for one-sided communication, which decouples data transfer from synchronization; the partitioning gives application control over data layout; and the multithreading relaxes the static (SPMD) model used in UPC. Our code is designed with latency hiding as primary goal, and we explore the programmability and performance benefits of UPC's one-sided communication model, coupled with a dynamic parallelism model. Our experience highlights some of the subtle pitfalls of dynamic threading and the need for application-level control for thread management, which is relevant to the HPCS languages (X10 [13], Fortress [2] and Chapel [10]) as well as existing libraries like Charm++ [21] or (if augmented with locality control) Cilk [5]. Scheduling decisions must be made to balance the needs of parallel progress, memory utilization, and cache performance.

Latency tolerance for both memory and network latencies has become increasingly important in high performance algorithm design, as latencies have remained relatively stagnant over years of tremendous gains in clock speed and bandwidth scaling. Our implementation is designed to respect Little's Law, which quantifies the need for parallelism to mask latency: to run at bandwidth rather than latency speeds, sufficient parallelism (*bandwidth*latency*) in the memory and network streams is required to keep pipelines full [3]. Current high performance implementations, such as HPL and ScaLAPACK, view each task's execution as a single thread of control, which unnecessarily serializes the computation, and results in ad hoc and restrictive solutions to latency hiding. Rather than designing an algorithm for a specific degree of parallelism, we will use a dataflow interpretation of the algorithm with a multithreaded implementation [28][30] that exposes all available parallelism at runtime. Lewis and Richards [23] used this dataflow approach successfully for LU in a shared memory parallel setting on up to 24 processors, but not in a scalable distributed memory context. This work was continued by Kurzak and Dongarra [22] and colleagues [8][9] for additional matrix factorizations on multi-core processors. A mixed shared/distributed memory code in this style was also run on the NASA Columbia machine, using the data driven approach within shared memory [29].

Several challenges arise in using a highly parallel dataflow view of the algorithm as we do. First, because we want to run on hundreds or even thousands of processors and across clusters, locality is critical. We use UPC's global address space to statically distribute blocks of the input matrix and build scheduling queues for the tasks associated with each block; both the matrix blocks and queues are remotely accessed through the global address space. Second, the multithreading support that is needed to expose available parallelism can have a significant runtime cost; we explore several different strategies for implementing fast user-level threads. Third, while the algorithm is highly dynamic, control over task scheduling is critical and non-obvious. For highest performance we use an application-specific scheduling policy to ensure proper prioritization. Fourth, as with any attempt to expose all available parallelism to the runtime layer, memory resources can easily be strained, and deadlock may result in a constrained memory environment, because tasks that have been allocated may not be able to run until other unallocated tasks complete. We use a novel dependence-constrained task allocation mechanism to avoid deadlock. Finally, we incorporate some of the best-practice optimizations from prior work, including recursive algorithms to increase granularity and the combining certain tasks to improve local task size and thereby boost serial performance. We found each of these optimizations necessary to high performance.

2. A MULTI-THREADED VIEW OF LU

2.1 The LU Factorization Algorithm

The LU factorization algorithm that is most commonly used on parallel machines is simply a reorganization of classic Gaussian Elimination. The basic algorithm proceeds row by row, attempting to "eliminate" entries below the main diagonal. Multiples of row *i* are subtracted from rows below *i* in order to ensure that the part of column *i* below the main diagonal becomes zero. To enhance numerical stability *pivoting*, the swapping of rows to place a large value on the diagonal, is performed prior to each elimination step. The steps for a matrix *A* are[1]:

for *i = 1* to *n-1*

1. find maximum absolute element in column *i* below the diagonal
2. swap the row of maximum element with row *i*
3. scale column *i* below diagonal by *1/A(i,i)*
   L(i,i)=1
   for *j = i+1* to *n*
   L(j,i)=A(j,i)/A(i,i)
4. Set row *i* of *U*
   for *j = i* to *n*
   U(i,j)=A(i,j)
5. Perform a "trailing matrix update", i.e. update the part of the matrix below and to the right of *A(i,i)*
   for *j=i+1* to *n*
   for *k = i+1* to *n*
   *A(j,k) = A(j,k)-L(j,i)*U(i,k)*

   This step can equivalently be expressed as a "rank-one update":
   A(i+1:n,i+1:n) = A(i+1:n,i+1:n) -
   *L(i+1:n,i)*U(i,i+1:n)*

After this process is completed, the solution of *Ax=b* can be obtained by forward and back substitution with *L* and *U*. For benchmarking purposes both HPL and our code append *b* to *A* as an extra column and perform the above operations on this augmented matrix. In this way, only substitution with *U* is needed to find *x*.

One can view this algorithm from a dataflow perspective and see that there are several dependencies between the first few steps, but the update in the last step is both parallel in itself and, if it is broken into smaller tasks, the subsequent elimination steps may be started as soon as column *i+1* is computed. This view exposes extremely fine-grained parallelism, but uses only inefficient matrix-vector and vector-vector operations.

[1] Note that the lower triangular part of the input matrix is commonly overwritten with *L* and the upper triangular part with *U*

The blocked algorithm, schematically depicted in Figure 1, is critical to high performance on cache-based and distributed memory machines, and emerges when we note that mathematically the result stays the same if we replace matrix elements by blocks, division by linear solves, and scalar multiplication with matrix multiplication. Pivots are still computed (maximums) within a single column, and individual rows, not blocks, are swapped during pivoting. In addition, a slight modification is required in the computation of the blocks of *U*. In this algorithm, blocks of columns referred to as "panels", are first factored (steps 1-3), a number of rows of *U* are computed (step 4), then the trailing matrix is updated (step 5). A more detailed treatment of the algorithmic issues can be found in Golub and van Loan [16].

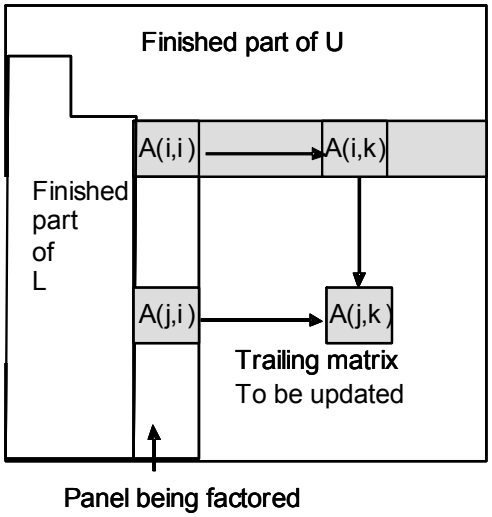

Figure 1. Dataflow in LU Factorization. The computation proceeds from left to right, factoring panels then updating higher number numbered panels.

2.2 The Parallel Case

Load balancing considerations motivate the use of a blocked cyclic layout for the parallel algorithm, which is standard in distributed memory implementations. Returning to Figure 1, we note that if, for example, the matrix is distributed by block columns, as the computation proceeds fewer and fewer processors perform useful work (because the factorization proceeds from left to right). In order to keep all the processors busy for as long as possible we use a layout such as the one shown in Figure 2 where 4 processors are logically viewed as a 2x2 processor grid and matrix blocks distributed to the processors.

The bulk of the computation comes from the matrix multiplications in the blocked version of step 5, which multiply thin rectangular matrices to update the trailing submatrices of the original. Each processor is responsible for updating the blocks it owns, including the multiplication of blocks for this update, which it generally does not own. The blocked cyclic layout spreads out the work of the multiplication across most of the processors, except possibly at the end of the computation, where there may not be enough matrix blocks to distribute across the entire machine. Figure 2 also reveals the importance of choosing the block size, since a block size that it too large will result in significant time spent in the beginning or ending phases (the left and right edges of the Figure) when some processors are idle.

0	2	0	2	0	2	0	2
1	3	1	3	1	3	1	3
0	2	0	2	0	2	0	2
1	3	1	3	1	3	1	3
0	2	0	2	0	2	0	2
1	3	1	3	1	3	1	3
0	2	0	2	0	2	0	2
1	3	1	3	1	3	1	3

Figure 2. Block cyclic decomposition with a 2x2 processor grid. The numbers denote the processor owning the respective block.

2.2.1 The Major Operations

The major operations of the algorithm are blocked versions of the steps detailed in Section 2.1. They sweep through the matrix performing:

1. Panel factorizations encompassing algorithm steps 1-3.
2. Updates to U (step 4). In this step the pivots are applied to the rest of the matrix, for pivots are only applied to the panel when it is factored.
3. The trailing matrix updates.

Note that these operations are carried out in parallel on the processors owning the corresponding blocks of the matrix. They can execute concurrently but are, however subject to a few constraints, discussed below.

2.2.2 The Constraints

The constraints on the operations come from the algorithm's dependencies:

- A panel can be factorized after all previous trailing updates to its rows and columns have been performed
- The pivot sequence can be applied to a region of the matrix when all previous trailing updates to it are complete
- U is updated after the pivot sequence is applied
- The trailing submatrix updates can be performed after the pivot sequence is applied

For the triangular solve using the upper triangular factor, *U*:

- A block triangular solve on a block of the solution vector (for computing *x* via back-substitution) can be performed only after all the updates from previous block

solves (with higher block numbers) have been accumulated

In our event-driven execution model, the dependencies are handled in part by dynamically creating threads for dependent operations:

- The panel factorization creates *U* update and trailing sub-matrix update threads
- Block triangular solves in back-substitution create update threads for lower numbered vector blocks.

The task graph for LU factorization, shown schematically in Figure 3 below, is not a tree, because update operations have two input dependencies. The first is from the thread created by panel factorization above, and the second is a result of updating *U*:

- Updates to *U* provide the other argument to the trailing update

The factorization process is initiated by noting that the constraints on the leftmost panel of the matrix are trivially satisfied and so a panel factorization can commence immediately.

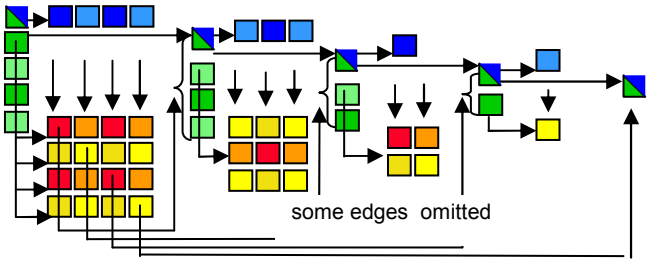

Figure 3. Dependencies in LU factorization

The constraints are handled in two ways. Operations that are dependent on remote events, such as the completion of the factorization of a panel, are triggered by the receipt of a notification. In addition, because we can perform operations on matrix blocks in any legal order, an additional synchronization method is needed. For example, a panel cannot be factored until all trailing updates and pivots are performed. In order to keep track of this, a count of the number of remaining updates is kept for each matrix block. When this count reaches zero, it is then safe to continue.

3. IMPLEMENTING THE ALGORITHM IN UPC

Our implementation is written entirely in UPC, except for the single node matrix kernels, such as matrix multiplication, which are performed using calls to an optimized Basic Linear Algebra Subroutine (BLAS) library [4], and a heap data structure taken from the C++ Standard Template Library. Our code was written in approximately 4,000 lines of code (not counting the threading system) compared to about 12,000 lines for HPL. Comparisons to ScaLAPACK LU are difficult because it depends on a large amount of supporting infrastructure that is used in other linear algebra routines. In this Section, we detail how our code dealt with many of the challenges of implementing the algorithm using distributed multithreading.

3.1 The Data Layout

As mentioned in Section 2.2, the input matrix is distributed in 2-D block cyclic fashion among the processors. All the blocks owned by a particular processor are stored contiguously in its local shared memory region. This allows update operations to be aggregated across blocks by viewing several matrix multiplications as one larger one, which can improve individual processor efficiency. Although UPC supports distributed arrays, including blocked and cyclic layouts, we chose to avoid them for a few reasons. First, they do not directly support blocking in multiple dimensions, which is necessary for the layout in Figure 2. Second, the block size must be a compile-time constant, whereas we would like it to be an input parameter. Finally, our prior experience has shown that use of blocked layouts can incur significant overheads in some compilers, although this is an active area in which optimizations are under development.

Although we do not use UPC's distributed arrays, we do take advantage of the global address space to build a distributed data structure that uses global pointers, which can refer to memory associated with other UPC processes.

3.2 Managing the concurrency

We use a multithreaded implementation of the algorithm with threads running on each processor for the major operations of Section 2.2.1. These threads are built over the fixed set of UPC threads and are scheduled co-operatively, which means they are never preempted, but instead explicitly yield when they reach a long-latency operation or synchronization dependence. Because there are many threads sharing a single processor and performance of the dense matrix operations suffers if cache context is lost, mid-execution, pre-emption is not viable. In addition, co-operative threads simplify maintenance of data structure consistency (no critical regions are necessary) as the threads give up control of the processor by explicit yielding only when the data structures are in a consistent state. Co-operative threads also provide a convenient abstraction for managing the multiple dependent operations that each processor may have in flight at any given time [1].

For managing these co-operative threads, we experimented with a number of threading packages, from GNU Pth [15] to POSIX Threads where only one thread executes at any time. For reasons of portability (to machines such as the Cray X1) and performance (context switches in our package cost only a function call or two) we eventually settled on a home-grown package that uses only function calls combined with the use of a variant of Duff's Device [31].

3.3 Enforcing locality

As discussed before, our implementation spawns threads on remote processors to perform certain matrix operations. Because we have a global address space a natural question is why we choose to spawn a thread on a particular processor. The answer, of course, is locality. Although not implemented this way, this can be viewed as giving a locality hint to a "global" thread scheduler: execute, for example, a trailing update on the processor that owns the block being updated.

3.4 Coalescing Updates

Each trailing update operates on a single block of the matrix (see Figure 3). These blocks are of size b x b, where b is the block size used to partition that matrix. We can obtain higher

performance if we observe that we can update an entire *block column* with a single matrix-matrix multiplication:

$$\left.\begin{array}{c} A_{i,j} \leftarrow A_{i,j} - L_{i,k} * U_{n,j} \\ \vdots \\ A_{i+m,j} \leftarrow A_{i+m,j} - L_{i+m,k} * U_{n,j} \end{array}\right\} \Rightarrow A_{i:i+m,j} \leftarrow A_{i:i+m,j} - L_{i:i+m,k} * U_{n,j}$$

In our implementation, this optimization was manually performed, but it is conceivable that in the future, some high level language constructs can be provided for coalescing the threads that implement a trailing update on a single block. Our coalescing strategy departs from the standard HPL code. HPL also coalesces multiple block columns to produce even larger trailing update matrices. Because these matrices are larger, the matrix multiplications can potentially run at a higher fraction of peak. The trade-off is that this approach exposes less concurrency because a panel factorization can start when the update to the panel (and not the whole submatrix stored on the processors) is completed. In future work we will consider a compromise hybrid scheme that coalesces larger blocks for panels that are not about to be used for panel factorizations. Note that our scheme requires that we use larger block sizes than HPL because of the need for large matrices for multiplication.

3.5 Memory-Constrained Lookahead

In matrix factorization, lookahead refers to the technique of overlapping panel factorizations with trailing matrix updates. In HPL lookahead is static in the sense that, at program startup time, the user must define the number of panel factorizations that can be overlapped. In our case, lookahead is dynamic. As long as the dependencies are satisfied, any amount of overlap is permissible.

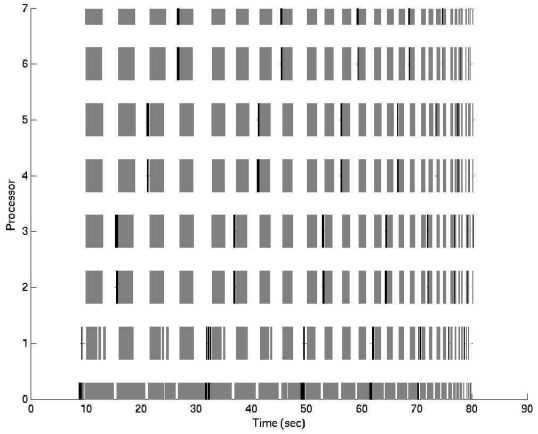

Figure 4. Timeline for an LU run with no lookahead (SGI Altix Itanium 2 1.4GHz, n=12,800, process grid = 2x4, block size = 400). Grey blocks represent matrix multiplication and black blocks denote panel factorization.

Figures 4 and 5 show timelines for processors on a small problem with and without lookahead. They show how the dynamic lookahead version can overlap panel factorizations with matrix multiplications, filling "holes" in the execution schedule of the more synchronous code.

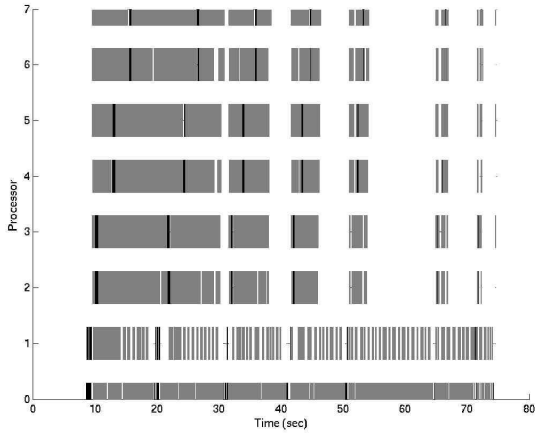

Figure 5. Timeline for the same problem as Figure 4 with dynamic lookahead (a buffer size of 512MB). As a very small problem was used for illustrative purposes, the processors were not very busy towards the end of the computation.

One of the ways we create opportunities for overlap is by coding the operations that require remote blocks in a split-phase manner. For example, an update operation on a trailing block requires two other blocks (one to the left and one above) to be multiplied. One or both of these input blocks may be remote, and we use bulk UPC operations such as upc_memget. On most machines we use a Berkeley compiler extension to these bulk operations, which provide non-blocking functionality [6][7]. In this way we can hide the latency of the transfer by yielding and only resuming once the transfer is complete.

The remote blocks used during updates and in the computation of *U* require storage on the processor where the computation is to be performed. In the current implementation we decide at startup time on the amount of memory we wish to devote for latency hiding. Memory allocated for buffering trailing matrix updates is tracked. When initiating an operation we reserve a buffer from this memory pool, start the transfer(s), then return to other work. When all the transfers for a particular update are complete, the operation is performed, then the memory returned to the pool. In this way we can hide the latency of as many transfers as we have memory. In addition we have a stored "reservoir" of work that can be performed when the processor would otherwise be idle. If we find that we cannot allocate enough memory to hold a transfer, we simply defer the operation and return to it at a later (and hopefully more favourable) time.

It is important to note that there is the potential for deadlock. If the notification for the completion of the factorization of panel 3, for example, arrives before panel 2's notification and the last bit of memory on a processor is allocated for buffering panel 3's updates, progress is stopped because panel 2's updates can't be run and these must come before panel 3's updates release the required memory. This is avoided by always allocating memory in the order of the panel number that spawned the update with no "holes" in the sequence. In the previous example, no memory would be allocated for panel 3 until panel 2 is taken care of. This strategy ensures that all the trailing updates to a panel can be buffered and so complete before dealing with any higher panels. While application specific, this suggests a possible general

solution that looks at dependency information before making memory management and scheduling decisions.

3.6 Panel Factorization Issues

The panel factorization is managed by the processor owning the corresponding diagonal block. It sends requests to the other participants in its process column (that own the part of the matrix below its diagonal block) asking them to:

- Return the absolute maximum element of a particular column. The managing processor can then perform the pivots.
- Scale a column
- Perform a trailing matrix update to other columns in the block column.

Note that at this point the processor controlling the factorization can choose to either yield or wait for completion of the operations it initiates on other processors. Because the panel factorization is a critical operation (as it produces work for the other processors) the controlling processor only performs its portion of the panel factorization and handles event notifications while waiting for completion of the operations. It suspends the threads that perform trailing updates and updates to *U*.

The algorithm implemented by this master processor is either a simple blocked version of Gaussian Elimination or a recursive version in the style of Gustavson [19]. After some experimentation we determined that the recursive version provides better performance and is used in all the experiments below.

After the panel factorization is complete, the processors that own blocks of *U* in the same block row as the diagonal block are notified of its completion. The pivots that were needed for the panel factorization are then applied to each block column of the trailing submatrix, but only after all updates for all previous factorizations have been applied. The corresponding blocks of *U* can then be computed. Because we use one-sided communication, only the processor involved in the update to U needs to actively participate in applying the pivots. It is notified when previous trailing matrix updates in its block column are complete to indicate when it is safe to begin pivoting. These notifications are sent by the processors in its process column that perform trailing submatrix computations.

4. TUNING

Tuning the code for performance involves finding good values of the following parameters; the block size, process grid, base case of the panel recursion, and amount of memory to use for buffering. The goal is simple: to keep the processors as busy as possible.

In order to accomplish this task, the local thread scheduler needs to make some decisions about what operations to perform. As such, we have the following prioritization of operations:

- Panel factorizations run as soon as possible. This is motivated by the fact that the panel factorizations drive the whole process and expose the concurrency in the algorithm. In fact, when panel factorizations have started, the participating processors suspend all other activities (including trailing updates) in order to focus on the task. In a small experiment on 32 2.2GHz Opteron processors on a 40,000 x 40,000 matrix, block size 200, performance dropped from 107.87 GFlop/s to 89.64 GFlop/s when other threads weren't suspended for panel factorizations.
- Trailing updates (matrix multiplications) to panels with lower column numbers are performed before all others. This is accomplished by organizing all ready updates (those for which all remote data has been transferred) in a priority queue. This is of use primarily at large scales when lots of panels are buffered. To simulate this on a small number of processors we ran a 25,600 x 25,600 matrix on 16 Opteron processors with a small block size (50). Absolute performance naturally suffered, but we observed an additional 20% decrease in performance with a poorer schedule that always used the panel with the highest block number.

These rules ensure that panel factorizations (the generators of work) start as soon as all their dependencies are satisfied.

The rate at which the dependencies are satisfied, however, depends on the amount of memory available. If memory isn't available to buffer panels of *L* and strips of *U*, the trailing updates cannot be executed. Thus there is a dependency between the amount of memory available and the degree of utilization of the processors.

The block size used plays a very interesting role. It must be large enough to enable the trailing updates to run at a high fraction of peak performance. However, it must not be so large that the panel factorizations (a very communication-intensive process due to pivoting) take a long time to complete, resulting in idle processors. In addition, the larger the panels, the fewer of them that can be buffered for a fixed amount of memory.

Table 1 shows how performance varies when all other parameters are fixed and the block size is varied. For this configuration, a block size of 400 gives the best performance, but the matrix multiplications (using the Goto BLAS [17]) run slower compared to a large block size such as 1600. The combination of longer panel factorization times and less space for buffering dooms this parameter choice as it results in an execution that cannot hide latency sufficiently well to achieve high performance.

Table 1. Variation of performance with block size on SGI Altix Itanium 2 1.4GHz, n=25,600, process grid = 2x4, buffer size = 512MB

Block Size	Max dgemm() performance (Gflop/s)	Total Performance (Gflop/s)
200	5.03	33.77
400	5.26	35.52
800	5.40	32.75
1600	5.44	24.03

In addition to the optimizations described above that are specific to our particular implementation strategy, we also implemented the common "tricks" that Linpack benchmark writers have used over the years:

- Each block is stored in row major order so that, when rows are exchanged in the pivot operation, contiguous portions of memory are transferred.
- The blocks are arranged in memory in column major order so that the trailing matrix update on a block column can be performed with a single BLAS-3 call, obviating the need for memory copies.

5. PERFORMANCE RESULTS

Our experimental setup is similar to any HPL experiment. A random dense matrix, A, and random vector, b, are created (using the "Mersenne Twister" code of Matsumoto and Nishimura [27]), and the code solves for x where $Ax=b$. In order to keep the code as clean as possible there are some restrictions on the matrix size and block size. The block size must evenly divide into the matrix size and the total number of row and column blocks must be an exact multiple of the process row and process column size respectively. These ensure that each task contains the same number of blocks. A future version will relax this restriction, primarily by upgrading the 2-D distribution and global to local address computation routines.

Our UPC code is compiled with the Berkeley UPC compiler, except on the Cray X1 where Cray's UPC compiler is used. The Berkeley compiler supports some non-blocking extensions of the bulk memory operations which are important on some machines.

5.1 Single Processor Performance

In order to evaluate the overheads incurred by our multithreaded implementation, we first collect performance numbers for single processor runs. These are reported in Table 2. The data shows that the overall performance of the code is within 10% of the performance of the matrix multiplications on their own, which are entirely BLAS 3 calls. Here, only the block size needs to be tuned as there isn't any parallel execution. Threads are, however, created and managed as in the parallel case.

Table 2. Single Processor Performance of UPC-LU code

Processor	Performance GFlop/sec	% peak	matrix mult. %peak in run
Opteron 2.2 GHz	3.6	81.9	89.2
Itanium 2 1.5GHz	6.0	91.8	95.2

5.2 Parallel Performance

In this Section, we present our results for parallel execution. Fair performance comparisons across implementations are difficult, because each one has to be tuned using parameters such as block size, and in the case of the HPL code, the amount of overlap. We primarily provide comparisons to HPL because our UPC-LU code greatly outperforms synchronous codes such as ScaLAPACK. For example, we observe a 62% increase in performance on a sample 16 processor, n=32,000 problem run on the SGI Altix.

Comparisons to HPL numbers from the HPC Challenge survey are shown in Tables 3 and 4. Because of the difficulty of gaining access to large amounts of time at supercomputing centers, we attempted to match machine configurations as closely as possible, always reporting HPL results on a machine as least as powerful as we used for our code. Although the random matrices used are also different and this affects the time spent in pivoting, we later note that pivoting only accounts for a negligible fraction of our running time.

Table 3. Parallel Performance of UPC-LU

Machine	# proc.	Perf. (Gflop/s)	% peak	n
Cray X1 (800 MHz)	64	576.7	70.4	128,000
Cray X1 (800 MHz)	128	1215.8	74.2	230,400
Opteron (2.2 GHz) Infiniband	64	216.3	76.8	76,800
SGI Altix (Itanium 2 1.5GHz)	32	149.8	78.0	64,000
Itanium 2 1.4GHz (Elan4)	512	2249.4	78.5	230,400
Cray XT3 (2.4 GHz)	512	2041.1	76.6	229,376

Table 4. Parallel Performance of HPL. Source: HPC Challenge [20][26].

Machine	# proc.	Perf. (Gflop/s)	% peak	n
Cray X1 (800 MHz)	64	521.6	63.7	160,000
Cray X1 (800 MHz)	60	578.8	75.4	135,555
Cray XD1 (2.2 GHz Opteron)	64	223.4	79.3	84,000
Cray XD1 (2.4 GHz Opteron)	128	502.1	81.7	110,000
SGI Altix (Itanium 2 1.6GHz)	32	147.3	71.9	49,152
SGI Altix (Itanium 2 1.6GHz)	64	297.0	72.5	98,304

The data in the Tables 3 and 4 show very good performance for our UPC implementation. On a 512 processor Itanium/Quadrics machine (the Thunder machine at LLNL), the UPC code achieves over 2 TFlop/s, which validates the scalability of the Berkeley UPC implementation and multi-threaded approach to LU. Our historical data from the development of our LU code indicates that non-blocking communication, the use of a very lightweight thread layer, and careful task prioritization are all critical.

The UPC code also performs very well in comparison to the HPL implementation written in MPI, but in approximately 1/3rd the program size. The following graph summarizes the data for a few machine configurations. As noted, the HPL/MPI numbers are taken by other researchers, since the effort of tuning these codes to a particular machine and problem size can be substantial. We were unable to access identical machines for the Opteron cluster and Altix comparisons, but chose machines that should favor the HPL/MPI code. For the Opteron cluster comparison (denoted "Opt" in Figure 6), we used numbers from a Cray XD1 for the HPL code and an Infiniband cluster at NERSC for the UPC code, both with the same Opteron clock rate. This should give a substantial advantage to the HPL implementation because of the more tightly integrated network, but we see that the UPC performance is quite close. Similarly, the clock rate on the Altix machine is slightly faster on the machine used for the HPL/MPI code, but the UPC code slightly outperforms the MPI.

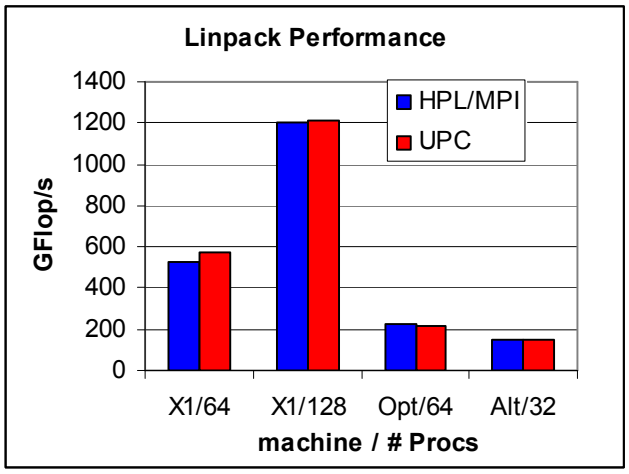

Figure 6. Performance Summary

5.3 Other Performance Factors

To conclude our performance discussion, we present some performance aspects of our code that, while not the most important, shed some light on how the use of multi-threading and one-sided communication influence the final Gflop/s number.

There is some conventional wisdom in the HPC community that the application of the pivots is an intricate, time consuming process. However, because we use one-sided communication (along with the row-major layout) we find that this is not the case. For example, on the n=76,800 run from Table 3, on a loosely coupled cluster, the application of the pivot sequences to the trailing sub-matrices only accounted for a maximum (over all UPC tasks) of 0.36% of the execution time. By comparison, the overhead of managing the queues took up 5.04% of the total time. In addition, UPC Task 0 spawned 46,193 threads locally. This validates our decision to implement a low-overhead threading system.

6. CONCLUSION

This paper presents a set of parallelism primitives for the UPC language, including non-blocking bulk communication, lightweight user-level threading to mask communication and synchronization latency, and a remote thread spawning mechanism. We demonstrate this in a running example, which results in a new implementation of LU factorization and the High Performance Linpack benchmark written from the ground up using UPC extended with user-level threading. The performance meets and sometimes exceeds that of the MPI-based HPL and ScaLAPACK codes, including runs that have been tuned by others for benchmark reporting. The code was run on up to 512 processors so far, with performance exceeding 2 TFlop/s. The UPC code uses non-blocking remote memory operations (one-sided communication), remote task creation, and dynamic threading to allow computations to suspend mid-execution at statically determined points. The code is roughly 1/3rd the size of the HPL and requires less manual tuning, since the degree of lookahead in the HPL is handled automatically and dynamically through our memory-constrained approach.

Our experience touches on several areas of relevance to language designers and users of parallel programming systems:

- We demonstrated the effectiveness of combining multithreading for communication overlap with user-controlled data layout for locality. While both multithreading and locality control exist in isolation in previous systems, and are proposed for the HPCS languages, there is little experience with this combination at scale. Control over data and task placement are essential to the performance of our code and many other applications.
- We demonstrated the use of latency tolerance mechanisms including non-blocking one-sided communication and multi-threading with threads that automatically deschedule themselves when they reach a long-latency operation. It is our contention (see Gürsoy and Kale [18] for a supporting viewpoint) that parallel systems of the future must include some facilities for handling this issue.
- We presented a technique for memory deadlock avoidance in latency tolerant programs. This is a critical problem with the kinds of scheduling flexibility that can arise in a distributed multithreaded environment. Combined with constrained resources (such as limited memory) a system can easily deadlock by allocating all available resources to tasks that are unable to run to completion due to dependencies. Our solution so far is application-dependent, but we believe it can be generalized.
- We leveraged UPC's partitioned global address space to ensure locality in multi-threaded computations. The global address space was used for updating remote information about dependencies, for access to remote blocks of the matrix, and for scheduling remote tasks.
- We identified three of the major issues concerning scheduling and prioritization of multiple local threads in a distributed memory environment. We observed that using application-dependent information was critical. In our case:
- Parallel progress was ensured by prioritizing panel factorizations and lower numbered trailing updates

- Memory was controlled using our dependency based allocation scheme
- Cache performance was maximized by increasing the size of the matrix multiplications. While not strictly a scheduling concern in this case, this issue will become more important in, say, sparse matrix factorization algorithms [24] where very large BLAS 3 operations may not always be available.

We intend to continue research in these matters, gaining experience from implementing a wide range of applications in this style on a wide range of machines (including those with explicitly managed memory hierarchies such as Cell), in the hopes of discovering a concise yet complete set of primitives that are useful for building high performance parallel applications.

7. ACKNOWLEDGMENTS

This work was supported by the Director, Office of Science, Office of Advanced Scientific Computing Research, of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231.

8. REFERENCES

[1] Adya A., Howell J., Theimer M., Bolosky W. J., and Douceur J. R. Cooperative Task Management without Manual Stack Management or, Event-driven Programming is not the Opposite of Threaded Programming. In *Proceedings of the 2002 USENIX Annual Technical Conference*, 2002.

[2] Allen E., Chase D., Hallett J., Luchangco V., Maessen J.-W., Ryu S., Steele G. L., and Tobin-Hochstadt S. *The Fortress Language Specification*, 2007. Available at http://research.sun.com/projects/plrg/Publications/index.html

[3] Bailey D. *Little's Law and High Performance Computing*, 1997. Available at: http://crd.lbl.gov/~dhbailey/dhbpapers/little.pdf

[4] Blackford S., Demmel J., Dongarra J., Duff I., Hammarling S., Henry G., Heroux M., Kaufman L., Lumsdaine A., Petitet A., Pozo R., Remington K., and Whaley R.C. An Updated Set of Basic Linear Algebra Subprograms (BLAS), *ACM Trans. Math. Soft., 28*, 2 (2002), 135-151.

[5] Blumofe R. and Leiserson C. Space-Efficient Scheduling of Multithreaded Computations. *SIAM J. on Computing, 27*, 1 (1998), 202-229.

[6] Bonachea D. *GASNet Specification*, v1.1. U.C. Berkeley Technical Report CSD-02-1207, 2001.

[7] Bonachea D. *Proposal for Extending the UPC Memory Copy Library Functions and Supporting Extensions to GASNet, v1.0*. Lawrence Berkeley National Laboratory Technical Report LBNL-54983, 2004.

[8] Buttari A., Dongarra J., Kurzak J., Langou J., Luszczek P., and Tomov S. The Impact of Multicore on Math Software. In *Proceedings of PARA 2006*, Umeå, Sweden, June 2006.

[9] Buttari A., Langou J., Kurzak J., and Dongarra J. Parallel *Tiled QR Factorization for Multicore Architectures*. Technical Report UT-CS-07-598, University of Tennessee, Computer Science Department, July 2007. Also published as LAPACK Working Note 190.

[10] Callahan D., Chamberlain B. L., and Zima, H,P. The Cascade High Productivity Language. In *Proceedings of the 9th International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS 2004)*, 52-60, IEEE Computer Society, 2004.

[11] Chen W., Bonachea D., Duell J., Husbands P., Iancu C., and Yelick K. A Performance Analysis of the Berkeley UPC Compiler. In *Proceedings of the 17th Annual International Conference on Supercomputing (ICS)*, 2003.

[12] Choi J., Dongarra J., Ostrouchov S., Petitet A., Walker D., and Whaley, R.C. The Design and Implementation of the ScaLAPACK LU, QR, and Cholesky Factorization Routines. *Scientific Programming, 5*, (1996), 173-184.

[13] Ebcioglu K., Saraswat V., and Sarkar, V. X10: an Experimental Language for High Productivity Programming of Scalable Systems. In *Proceedings of the P-PHEC 2005 Workshop*, held in conjunction with HPCA 2005, 2005.

[14] El-Ghazawi T., Carlson W., Sterling T., and Yelick K. *UPC: Distributed Shared-Memory Programming*. Wiley-Interscience, 2005.

[15] Engelschall R. Portable Multithreading: The Signal Stack Trick for User-Space Thread Creation. In *Proceedings of the 2000 USENIX Annual Technical Conference*, 2001.

[16] Golub G. and Van Loan, C. *Matrix Computations*. Johns Hopkins University Press, 1996.

[17] Goto K. and van de Geijn R. *On reducing TLB misses in matrix multiplication*. Technical Report TR-2002-55, The University of Texas at Austin, Department of Computer Sciences, 2002. Also published as FLAME Working Note #9.

[18] Gürsoy A. and Kale L. V. Performance and modularity benefits of message-driven execution. *Journal of Parallel and Distributed Computing*, *64,* (2004), 461-480.

[19] Gustavson F. 1997. Recursion Leads to Automatic Variable Blocking for Dense Linear-Algebra Algorithms. *IBM Journal of Research and Development, 41*, 6 (1997), 737-755.

[20] HPC Challenge Benchmark Page. Available at: http://icl.cs.utk.edu/hpcc/

[21] Kale L. V. and Krishnan S. CHARM++ : A Portable Concurrent Object Oriented System Based On C++, *ACM Sigplan Notes*, *28*, 10 (1993), 91-108.

[22] Kurzak J. and Dongarra J. *Implementing Linear Algebra Routines on Multi-Core Processors with Pipelining and a Look Ahead.* Technical Report UT-CS-06-581, University of Tennessee, Computer Science Department, 2006. Also published as LAPACK Working Note 178.

[23] Lewis B. and Richards K. *LU Factorization Case Study Using FAST: Dataflow Parallelism with the Forte Application Scalability Tool.* 2003. Available at: http://developers.sun.com/prodtech/cc/articles/FAST/lu_content.html

[24] Li X. and Demmel J. SuperLU_DIST: A Scalable Distributed-Memory Sparse Direct Solver for Unsymmetric Linear Systems. *ACM TOMS, 31*, 3 (2003), 110-140.

[25] Lin H.-Y. and Luszczek P. Tuning LINPACK N*N for PA-RISC Platforms. Presented at the *2001 High Performance Computing on Hewlett-Packard Systems Conference*, 2001.

[26] Luszczek P., Dongarra J., Koester D., Rabenseifner R., Lucas B., Kepner J., McCalpin J., Bailey D., and Takahashi D. Introduction to the HPC Challenge Benchmark Suite. SC2005 (submitted), Seattle, WA, 2005.

[27] Matsumoto M. and Nishimura T. Mersenne Twister: A 623-dimensionally equidistributed uniform pseudorandom number generator. *ACM Transactions on Modeling and Computer Simulation, 8*, 1 (1998), 3-30.

[28] Nikhil R. and Arvind. Can dataflow subsume von Neumann computing? In *Proceedings of the 16th Annual International Symposium on Computer Architecture*, 1989.

[29] Panziera J.-P. and Baron J. A Highly Efficient Linpack Implementation Based on Shared-Memory Parallelism. In *Proceedings of the 2005 International Supercomputer Conference*, 2005.

[30] Papadopoulos G. and Traub K. Multithreading: A revisionist view of dataflow architectures. In *Proceedings of the 18th Annual International Symposium on Computer Architecture*, 1991.

[31] Raymond E. *The New Hacker's Dictionary*. The MIT Press. ISBN 0-262-68092-0, 1996.

[32] Snir M., Otto S., Huss-Lederman S., Walker D., and Dongarra J. *MPI: The Complete Reference - 2nd Edition: Volume 1*. The MIT Press. ISBN 0-262-57123-4, 1998.

[33] The Top 500 Supercomputer Sites. Available at: http://www.top500.org, 2007.

[34] UPC Consortium. UPC Language Specifications, v1.2. Available at: http://upc.lbl.gov/docs/user/upc_spec_1.2.pdf, 2005.