Exploring Partial Synchrony in an Asynchronous Environment Using Dense LU

Jonathan Lifflander, Phil Miller, Ramprasad Venkataraman, Anshu Arya
University of Illinois
Urbana, IL 61801
{jliffl2, mille121, ramv, arya3}@illinois.edu

Terry Jones
Oak Ridge National Laboratory
Mailstop 5164
Oak Ridge, TN 37831
trj@ornl.gov

Laxmikant V. Kalé
University of Illinois
Urbana, IL 61801
kale@illinois.edu

ABSTRACT

Recent forecasts in high-performance computing predict that programming models of the future will be asynchronous in nature. However, opportunistic execution of available work can lead to interference with segments of the computation that should execute synchronously.

This paper describes a scheduling methodology that tightly synchronizes parts of an otherwise asynchronous parallel algorithm to obtain higher performance. Specifically, we apply *exclusive scheduling classes* to both asynchronous collectives and application-specific work units.

Our exploration of exclusive scheduling classes and other techniques arises from implementing a dense LU solver in a message-driven programming model and scaling it on modern supercomputers. The other techniques include mapping schemes beyond the traditional block-cyclic distribution and a method for decreasing network contention by ad-hoc agglomeration of data requests. Our findings suggest that future programming models will be hybrid models: asynchrony is beneficial, but these models must incorporate mechanisms that allow highly synchronous operations to perform efficiently.

1. INTRODUCTION

It is widely posited within high-performance computing that the predominant parallel programming models of today need to evolve to perform well in the peta/exascale era. Leaders in the field have suggested that new techniques beyond the bulk synchronous model are needed to scale applications to tomorrow's supercomputers.

Many have predicted that upcoming parallel models will use asynchrony and possibly message-driven computation to obtain high performance. Murphy, Sterling, and Dekate [23] assert that new system software needs to incorporate the following for the exacale era: message-driven computation, lightweight control objects for eliminating global synchronization, dynamic scheduling, allocation, and resource management. Brightwell, et. al., [6] predict that future exascale systems will need to be asynchronous and dynamic; they explain that the current bulk synchronous approach is susceptible to scaling limitations due to synchronization. Keyes, in the context of PDE solvers [19], suggests that new programming models are needed that postpone synchronizations, but that exploiting asynchronous algorithms requires prioritization much like in operating systems.

In many asynchronous and message-driven programming models, work is decomposed into work units that are scheduled on a processor from a local queue of available work units to maximize processor utilization. Some examples of systems that use this method are TBB [25], Charm++ [17], ParalleX [14], and Concurrent Collections [9]. To obtain higher performance, work units are often prioritized by the application, suggesting a preferred execution order to the runtime system in the presence of a choice between multiple pending work units. Execution decisions are then based solely on the work units pending in each processor's local queue.

This opportunistic execution of work units hides communication latencies in many circumstances. However, if the disparity in the size of the work units becomes too large, a greedy approach to maximizing processor utilization can lead to an overall degradation of performance. For instance, if the highest priority work unit in the queue is a large grain but has lower priority compared to the critical path work that has not arrived, executing the large grain can delay the critical path work that may become available soon. Hence, it may be beneficial to delay execution of an unsuitable work unit if a more fitting work unit, in terms of grain size or priority, will arrive soon.

Given that scheduling is complex in an asynchronous model but required in some contexts to achieve high performance, we implement a dense LU solver, a traditionally bulk synchronous application that relies on synchrony to obtain high performance, in a message-driven, asynchronous programming model. This allows us to evaluate the message-driven, asynchronous model for an application that can range between highly synchronous and asynchronous behavior.

Dense LU factorization is well understood and widely established as a supercomputer benchmark used to rank systems for the top500 list [1]. Its performance is primarily computation bound, but it presents distinct challenges in memory usage, scheduling, and communication to obtain high efficiency.

In developing an efficient dense LU implementation in a message-driven context, this paper makes the following contributions:

- *Exclusive scheduling classes*: This paper describes a work-unit classification scheme that enables tight synchronization for a subset of processors in a message-driven, asynchronous environment. The concept of exclusive scheduling classes can be applied to general scheduling policies for asynchronous collectives, or application specific scheduling to stratify work units into scheduling classes to improve performance.
- *Mapping*: This paper describes variations to the traditional block-cyclic mapping scheme that improve performance. Several variants of this traditional mapping scheme, such as row rotation or striding within a tile, are described and empirically tested.
- *Limiting network contention*: Although dense LU is not typically communication bound, when using a message driven model, the 'bursty' communication pattern found in dense LU may saturate network resources. When using a pull-based scheme to constrain memory, which uses point-to-point communication to transfer L or U block data, blocks in high demand introduce this problem. To ameliorate these effects, we limit the injection rate of blocks onto the network and dynamically agglomerate requesters of the same block into multicasts.
- *Trailing submatrix updates*: This paper expands on the previous theoretical work in describing a deadlock-free scheduling order for the trailing submatrix updates, which are the bulk of the computation in a dense LU solver. By scheduling work beyond step-order, the critical path computation in dense LU can be accelerated in a message-driven context.

We present a high-performance dense LU implementation[1], benchmarked on modern supercomputers, and demonstrate scalability to thousands of processors. The efficacy of our techniques is empirically tested and shown to substantially improve performance, making this LU implementation competitive with similar libraries. Our findings suggest that programming models of the future will be hybrid models: asynchrony seems beneficial in many contexts, but these models need to incorporate mechanisms that allow highly synchronous operations to perform efficiently.

2. ALGORITHM AND DESIGN

This paper describes a solver for a set of n linear equations in n variables: it performs LU factorization, solves the system of equations, and validates the solution by calculating the scaled residual (as described by the HPC Challenge specifications [11] and implemented in HPL). The LU factorization phase uses Crout's algorithm, which performs in-place factorization with partial pivoting for numerical stability.

To parallelize the process, the matrix is decomposed into square blocks of size b^2 (shown in figure 1) distributed across a set of processors. Blocks are assigned to processors at startup according to a *mapping* scheme, which is discussed in section 5, and are not reassigned during the course of execution. Typically, there will be tens to hundreds of blocks assigned to each processor core.

[1]The source code for our implementation is available from `git://charm.cs.illinois.edu/charmlu.git`

The factorization process can be described as follows:

for *step* in $0..n/b-1$:

Active panel blocks are those at/below diagonal block *step*

1. for *column* in $0..b$: (on each active panel block)
   (a) Each block identifies its maximum value below the diagonal in the current *column* within that block and contributes to a reduction among the active panel blocks.
   (b) The result of the reduction identifies the *pivot row*, which is swapped to the diagonal position and broadcast to all of the active panel blocks.
   (c) Each active panel block performs a rank-1 update of the section after *column* with multipliers from *column* and the pivot row.
2. The sequence of pivot exchanges is broadcast to the blocks of U and the *trailing submatrix*, which communicate to apply the same swaps as the active panel.
3. Active panel blocks send their contents, each a portion of L, to the blocks to their right.
4. U blocks to the right of the diagonal each perform a *triangular solve*, and send the result to the blocks below them.
5. Blocks in the trailing submatrix each compute a *trailing update* as the product of the L and U blocks they have received.

2.1 Grain Size

The trailing updates comprise the bulk of the computation that is performed in a dense LU solver. Each trailing update is a $\mathcal{O}(b^3)$ matrix-matrix multiplication (i.e. a call to the `dgemm()` level-3 BLAS routine). The triangular solves (via `dtrsm()`) are similarly computationally intensive. Each trailing update or triangular solve takes a few tens of milliseconds on the block sizes and processors used here. In contrast, the active panel is communication intensive, with b reductions and broadcasts occurring in rapid succession, each nominally taking hundreds of microseconds to single digit milliseconds.

2.2 Prioritization

On each processor, the work units for which input data has arrived are placed in a priority queue. The priorities are set by the type of work a unit represents and the index of its target block in the matrix. The basic priority scheme gives high priority to active panel work and U triangular solves (to generate work quickly), and lower priority to trailing updates.

2.3 Decomposition

In our implementation, each block is placed in a message-driven object, driven by coordination code written in Structured Dagger [18]. The coordination code describes the message dependencies and control flow for each block. Thus, every block can independently work its way through these steps, advancing as input data arrives in messages. With many blocks per processor, the Charm++ [17] runtime system inherently provides dynamic overlap of communication and computation by allowing blocks that have received their input to do work while blocks lacking data wait. In general, this arrangement ensures high utilization, since some block

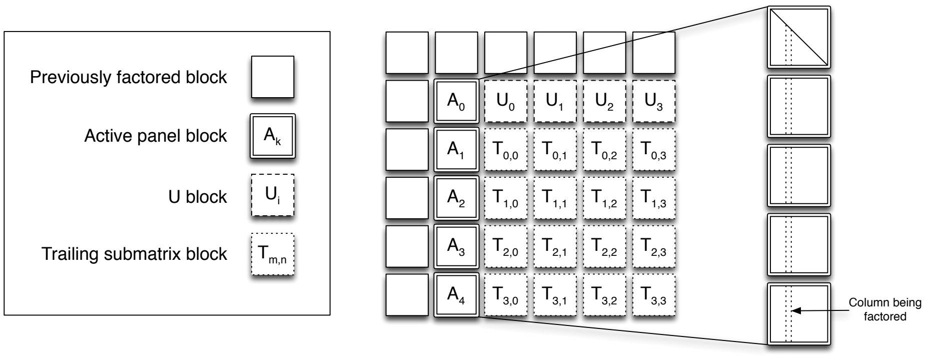
Previously factored block
Active panel block
A_k
U block
U_i
Trailing submatrix block
$T_{m,n}$
A_0 U_0 U_1 U_2 U_3
A_1 $T_{0,0}$ $T_{0,1}$ $T_{0,2}$ $T_{0,3}$
A_2 $T_{1,0}$ $T_{1,1}$ $T_{1,2}$ $T_{1,3}$
A_3 $T_{2,0}$ $T_{2,1}$ $T_{2,2}$ $T_{2,3}$
A_4 $T_{3,0}$ $T_{3,1}$ $T_{3,2}$ $T_{3,3}$
Column being factored

Figure 1: The matrix is decomposed into square blocks, which take on different roles as the factorization proceeds.

on each processor should always have work. The flexibility of a block-object decomposition is illustrated by the experiments described in sections 4 and 5.

This style of message-driven programming allows a clear and concise representation of the algorithm without explicit buffering of messages. When a message arrives, the Charm++ runtime system invokes a method on an object or buffers it if the object is not ready to execute the message.

By representing each matrix block as a separate object, the description of the parallel algorithm is separated from the particular details of its execution. Additionally, the control flow executed for each block is directly visible in the code; it is linear and effectively independent of other activity on its host processor.

Due to the simplicity of expression in the locally message-driven style, our code is approximately 2,550 lines long.[2] This is shorter than HPL, which is around 12,000 lines and the UPC implementation [16], which is around 4,000 lines of code.

2.4 Overlap

Ideally, every processor would remain busy during the entire factorization process. However, in each step, only a subset of processors will own blocks that are participating in the active panel. Thus, to avoid idling processors, work from multiple steps should be overlapped. The extent of the overlap (specifically, the number of steps the active panel runs ahead of trailing updates) in an implementation of dense LU is known as its *lookahead depth* [26].

Bulk synchronous implementations, such as HPL [24], require a fixed lookahead depth and restrict the overlap of steps to that amount. This restriction is due to memory limits of the machine; delaying the computation by increasing the lookahead depth means that memory for input blocks accumulates and then must be controlled. Due to implementation complexity and performance portability issues, the ScaLAPACK library [10] does no lookahead (i.e. its lookahead depth is 0).

In an asynchronous, dataflow parallel programming model, the availability of input data immediately triggers the next steps in the algorithm that depend on it. For typical, iterative, scientific algorithms, the amount of parallelism in the computations remains more or less steady as the algorithm progresses. Such algorithms can be expressed in pure dataflow semantics and can exploit asynchronous execution models without other concerns. However, the LU factorization has varying amounts of parallelism at different stages of the computation. When expressed in the dataflow model, it can cause unbridled spikes in memory usage because early steps in the algorithm trigger large amounts of data movement to feed the subsequent steps. For factorizations involving large matrices in limited memory environments, this can cause premature and unsuccessful termination of the execution. Hence, although lookahead is a natural consequence of using the dataflow model, it still needs to be moderated by a continuous awareness of memory and bandwidth utilization. This leads to a reality where the dataflow semantics are adaptively throttled by a system that monitors memory usage and other system parameters.

In a message-driven, asynchronous environment, LU can be implemented to allow dynamic lookahead: the diagonal can progress without a bound before the rest of the matrix finishes updating. Our solver implements dynamic lookahead, using a dynamic *pull-based* scheme to constrain memory consumption below a given threshold.

To implement the pull-based scheme, each processor has a distinguished *scheduler object* in addition to its assigned blocks. The scheduler maintains a list of the blocks assigned to its processor, and tracks what step they have reached. Within the bounds of the memory threshold, it requests blocks from remote processors that are needed for local triangular solves and trailing updates. To eliminate the possibility of deadlock, the order in which operations are executed, and hence remote blocks requested, must be carefully selected. Husbands and Yelick point out [16] that selecting updates in step order is deadlock-free, but suggest that there may be a general solution for finding a deadlock-free selection order of trailing updates using the dependencies between blocks. Section 7 describes the dependencies between the blocks and uses this to safely reorder the selection of trailing updates to execute.

3. OVERALL RESULTS

3.1 Performance

The percentage of peak obtained by an LU solver is correlated to the performance of the DGEMM implementation that it invokes. The performance of a DGEMM often varies with the size of the matrix that it operates on; a larger DGEMM can normally execute more efficiently than a smaller one. Therefore, when decomposing the matrix being factorized, there is a tradeoff between coarse grains that aid in higher DGEMM efficiency and smaller ones that may

[2]As counted by David Wheeler's SLOCcount.

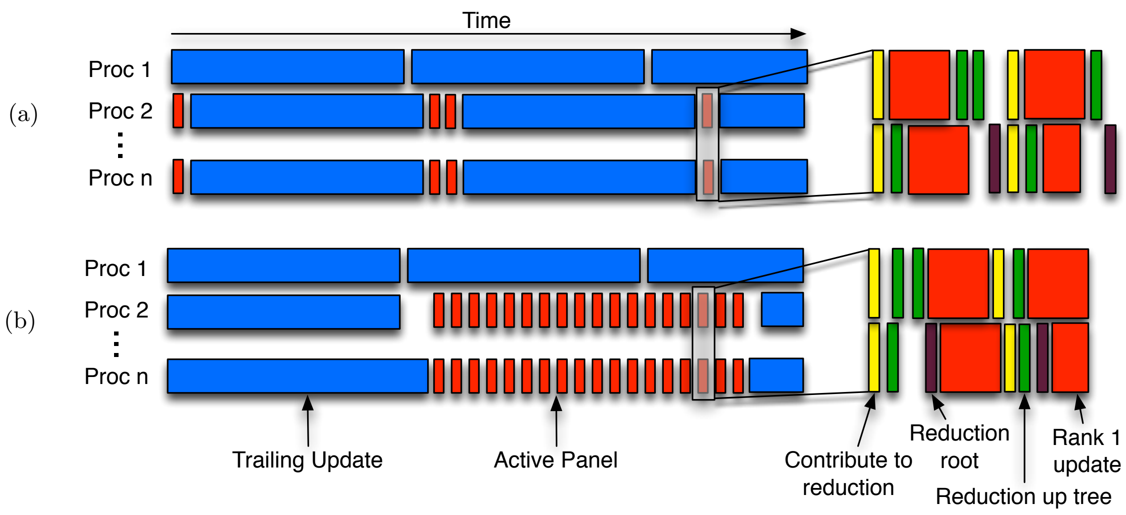

Figure 3: Two different time progressions of dense LU: (a) displays execution with interleaving of various grain sizes; (b) shows execution with isolation. If the smaller grains are interleaved with larger grains, the critical path is prolonged.

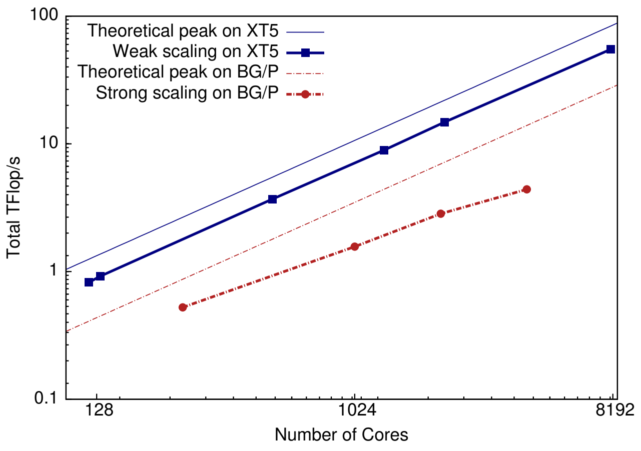

Figure 2: Weak scaling (memory usage of matrix is constant around 75%) from 120 to 8064 processors on Jaguar, a Cray XT5 machine with 12 cores per node. Strong scaling ($n = 96,000$) from 256 to 4096 processors on Intrepid, an IBM BG/P machine with 4 cores per node.

Block size	450	500	504	525	560	700
DGEMM (%)	78.2	81.9	82.3	81.8	81.6	83.6
LU (%)	65.5	66.6	67.0	66.5	65.5	65.0

Table 1: Percent of peak achieved by DGEMM and LU factorization on Cray XT5 with 120 cores and $n = 126000$

Library	Peak	Cores	n	Architecture
UPC [16]	76.6	512	229K	XT3
DPLASMA [4]	58.3	3072	454K	XT5
ScalaPack [10]	59	3072	454K	XT5
HPCC [3] HPL	65.8	224220	3936K	XT5
Jaguar top500 [1]	75.5	224162	5474K	XT5
CharmLU	67.4	2112	528K	XT5

Table 2: Percent of peak achieved by various linear algebra libraries. CharmLU is the implementation presented in this paper.

allow more overlap of communication and computation. Table 1 shows this tradeoff.

Table 2 compares our implementation with other dense LU solvers. Note that the architectures and matrix sizes vary, so it is difficult to provide an exact comparison, but the values imply that our implementation is competitive.

3.2 Scaling

To demonstrate the scalability of the dense LU solver described in this paper, we weak scale this solver to over 8000 processors. Using approximately 75% of memory[3] our LU solver obtains over 67% of peak on Jaguar, a Cray XT5 supercomputer. In addition, we demonstrate that our LU solver also strong scales up to 2048 processors[4] on Intrepid, an IBM Bluegene/P supercomputer, with over 50% parallel efficiency. Figure 2 shows both sets of results.

4. EXCLUSIVE SCHEDULING CLASSES

[3]This represents about 530 blocks of 500 $\times$ 500 doubles for each processor, yielding a matrix size of 132000 on 132 processors, up to 985000 on 8064 processors.

[4]With a matrix size of 96000, in blocks ranging from 300 to 150 doubles.

In applications that mix large grains of sequential execution with latency-sensitive communication operations, there is a tension between computational throughput and responsiveness: a single processor's work tends to execute most efficiently when presented in large chunks; however, when such compute kernels are running, reacting to incoming messages is difficult or impossible.

In many asynchronous programming models, work is decomposed into units and each processor draws from a local queue of available work units. When a processor finishes executing a work unit, it will select the highest priority work unit available in its queue as the next. In general, as long as work units are available it is beneficial to execute them to avoid idle time and maintain high utilization. However, if the highest priority work unit available is not on the critical path and is relatively long, it may delay execution of a critical work unit that will arrive soon. Therefore, it may be beneficial for that processor to idle briefly, waiting for the higher priority in-flight work unit, rather than opportunistically executing the already available work unit.

In any asynchronous execution model that is opportunistic, ensuring that specific classes of work execute uninterrupted is a challenging problem. The problem is exacerbated if there are large grain size variations across these classes. Decreasing the interleaving of a critical class with grains from other classes may be important for ensuring that the critical path computation or communication proceeds quickly.

Existing applications and runtime environments resolve this tension using a variety of methods:

- *Interrupts/Preemption*: Long stretches of execution can be interrupted when a latency-sensitive event occurs, with the reaction preempting the ongoing computation. This method can achieve excellent responsiveness, but requires low-level hardware or runtime support, may be overhead prone, and is difficult to program.
- *Polling*: The code for a long stretch of work can be adapted to explicitly poll for the arrival of a critical message and respond to it before resuming execution. This in-line interruption introduces overhead, but it also presents deeper issues of determining polling frequency. Moreover, it is not always desirable or possible to poll from within optimized compute kernels like those found in BLAS libraries.
- *RDMA*: If the critical operation is purely a data transfer operation on precomputed data, this problem can be resolved using remote direct memory access. With hardware support, this can be very efficient, since the ongoing computation can continue executing unaffected. However, only very simple operations are possible. Hardware and programming environment support are also necessary, limiting its portability

A straightforward message-driven implementation of dense LU factorization exhibits this problem because it carries a mix of latency-sensitive messages on the active panel, and mostly latency-insensitive work in the trailing submatrix. The former take microseconds to single-digit milliseconds per matrix column, whereas the latter take tens of milliseconds each.

When work on the active panel is available on a processor, it is given priority over all other parts of the factorization process. However, because new active panel work only arrives after the previous one has been completed, the intervening gap between these units gives the processor an opportunity to schedule large grain trailing updates or triangular solves. If such large grains are scheduled, the processor's participation in the next unit of active panel work is delayed, affecting all the processors involved in the panel factorization. This considerably slows down this class of work which lies on the critical path. With sufficient delays, processors will exhaust their backlog of trailing updates before the current panel is factorized and data for the next batch becomes available.

A possible method to decrease this interference is to separate work units into *exclusive scheduling classes*. During execution, the scheduler is set to some exclusive scheduling class. Work units of lower classes in the local queue will be held back in favor of higher class work units. Such stratification of work units allows the scheduler to selectively choose only the work units that are suitable for execution, depending on the currently active scheduling class. The active scheduling class is determined by the application; it instructs the scheduler to transition to a different scheduling class when appropriate.

This methodology has the advantage of maintaining the desired variation in grain sizes while using a general scheduling methodology to solve the problem, thereby improving performance. Moreover, the intricacies of using application-specific polling or interruption/preemption can be avoided by segmenting work into scheduling classes.

To achieve high overall performance in dense LU, we simulate a scheduling-class scheme on top of the Charm++ runtime's priority-based scheduler. When work of one class is selected for execution on a processor, other work in lower scheduling classes is held back to avoid introducing unnecessary latency. This technique is analogous to scheduling classes in realtime systems and microprocessor interrupt levels: the delay or preemption of the latency-sensitive factorization is prevented by temporarily disabling execution of lower-class large grain work. Figure 3 shows two different possible executions, both with and without isolation using exclusive scheduling classes enabled.

4.1 Isolation of Active Panel

The most apparent class distinction in dense LU is between the active panel factorization and the bulk work (triangular solves and trailing updates). This separation is enforced by keeping a processor-local counter of the blocks currently participating in the active panel. When this counter is non-zero, bulk work is not enqueued into the runtime scheduler's queue. Instead, it is placed into an application-level queue, to be re-scheduled when the active panel completes. Bulk work units that are waiting in the runtime's queue are removed and placed in the same application-level queue. To maintain this counter, each block on the active panel increments this counter after contributing to the first column's pivot reduction and receiving the broadcast that results.[5] They decrement the counter when active panel work is com-

[5]The increment must wait for the first column to finish to prevent deadlock: some other block on a processor may need to perform a trailing update before it can participate in the active panel.

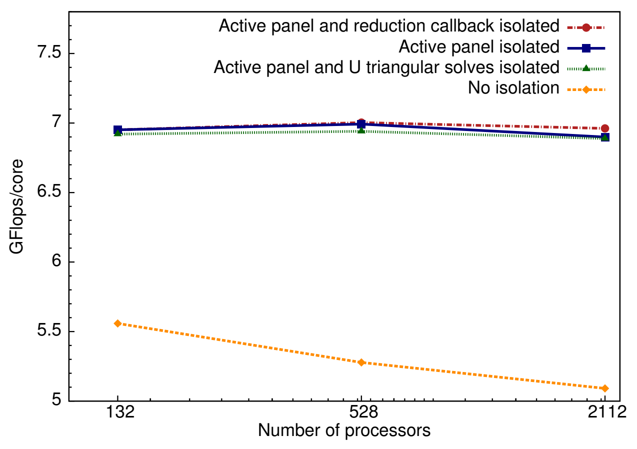

Figure 4: Performance effects of enforcing various exclusive scheduling classes on XT5 with weak scaling.

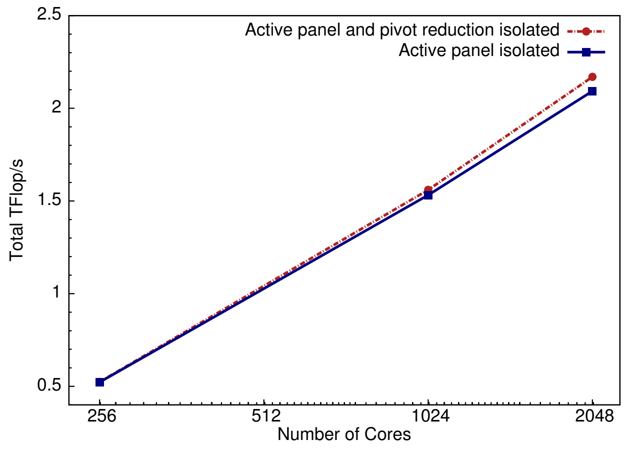

Figure 5: Performance effects of enforcing the pivot reduction exclusive scheduling class on BG/P with strong scaling from 256 to 2048 processors. As we scale, isolation has a greater impact on performance.

plete.

The benefits of isolating the active panel from the bulk work can be seen in figure 4. As the application weak scales with the active panel isolated, performance remains consistently high. However, without isolation, performance drops sharply.

4.2 Isolation of Triangular Solves

Among the larger work units, there are two different tasks: triangular solves on U blocks and trailing updates. Because triangular solves generate additional concurrent work, we generally prefer to perform triangular solves before trailing updates. Thus, we have also considered delaying trailing updates when the data to perform triangular solves is expected to be available. This occurs when an active panel is completed, and the diagonal block and all pivoting instructions have been broadcast. As figure 4 shows, this separation is actually slightly detrimental to performance. Performance degrades in this configuration because each triangular solve depends on pivot data from one or more blocks in the trailing submatrix below it, some of which may not have completed their updates for the previous step. Thus, the triangular solves wait longer than the execution time of several trailing updates before becoming ready to execute, and the processor idles.

Instead of a class separation, simple prioritization of ready-to-execute triangular solves ahead of any trailing updates provides the best performance. A more elaborate prioritization scheme might still prefer some trailing updates, such as to blocks that are in the next active panel, over triangular solves, especially those far to the right in the matrix.

4.3 Isolation of Asynchronous Reductions

The final work class distinction considered in this paper lies within each active panel process. Our steps for the factorization of each column of the matrix include: pivot identification via asynchronous reduction amongst all the participants in the active panel factorization; broadcast of a fragment of the pivot row to all participants; and a rank-1 update of the remaining unfactorized sub-blocks that are on the active panel. Performance gains were realized by splitting the rank-1 update into two separate updates: one for the immediate next matrix column and the other for the remaining sub-block. This allows earlier participation in the next pivot identification which is critical to progress and overlaps this communication with the rank-1 update computations.

The runtime performs the pivot reductions by constructing a spanning tree amongst the participant processors. These reduction operations along the spanning tree are fine-grain, while the rank-1 updates are large in comparison. When these rank-1 updates were scheduled on a processor before the reduction moved past it along the spanning tree, the overall progress was impaired by the delay in the reduction (inset of figure 3(a)). Thus, we place the reductions in a higher class than the rank-1 updates.

We modified CHARM++'s reduction mechanism to signal a callback on each processor after a reduction has propagated past that processor's position in the tree. This signals a transition out of the pivot identification work class, and pending rank-1 updates can then be executed (inset of figure 3(b)). Figure 5 shows that this yields an increasing performance improvement as we strong-scale. This gulf appears because strong scaling LU leads to a growing proportion of execution time spent in active panel factorizations.

We believe such a notification mechanism can be a general technique for scheduling around asynchronous sender-driven collectives. This directly aids in transitioning between exclusive scheduling classes.

4.4 Synchrony Amidst Asynchrony

By partitioning work into exclusive scheduling classes, we demonstrate that ideally highly synchronous workflows can run without interference from large-grain latency-insensitive asynchronous computation. Moreover, by placing an asynchronous collective in a separate scheduling class, fine-grained critical path work runs unaffected by larger grains, which are deferred by the scheduler's transition into a higher scheduling class. For dense LU, we describe an application-specific implementation of such a scheme and show that it substantially improves performance.

Our methodology attempts to increase the efficiency of synchronous operations in an asynchronous programming model. This suggests that for some parallel algorithms, purely asynchronous programming models may have disad-

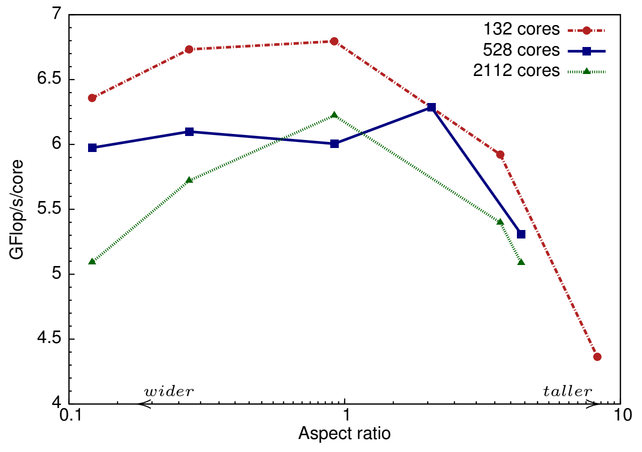

Figure 6: Various aspect ratios for the tile of a row-major block-cyclic distribution. Approximately square tiles perform the best. Tall tiles decrease performance because they lack parallelism in U. Wide tiles decrease performance because they slow down the active panel. Results are from weak scaling on XT5.

vantages. For instance, if highly synchronous work is on the critical path, ensuring that it executes early, uninterrupted by other work, may be essential to obtaining high performance. Hence, it seems that while asynchrony may be required to effectively program on the next generation of supercomputers, methodologies and runtime tools that increase the efficiency of synchronous operations, allowing them to execute without interruption, will also be necessary for the programming models of the future.

5. MAPPING

The traditional mapping scheme used in dense LU factorization is a block-cyclic distribution of blocks to processors. Figure 7(a) shows a typical block-cyclic layout with a processor tile of dimension 6×4. This paper describes variations to the traditional block-cyclic mapping, *rotation* and *striding*, that improve performance and may be generally beneficial to other implementations. The variations arise from addressing the following considerations:

- Computational parallelism in the active panel
- Locality among processors computing the active panel
- Memory contention between multiple active-panel processors on a node
- Communication parallelism in distributing blocks of U

These concerns are closely interrelated, and hence there are tradeoffs between them.

Within the block-cyclic mapping scheme the tile dimensions may be adjusted to vary the aspect ratio of the tile. There is a tradeoff between a tall tile, which increases the number of processors working on the active panel, and a wide tile, which allows for more U parallelism. Although the triangular solve work is slight compared to the trailing update work, parallelizing them further spreads the requests for that block of data over more processors. In addition, there is a tradeoff between a row- or column-major tile: a row-major tile spreads the processors working on the active panel across many nodes, reducing network locality but minimizing contention within each included node; whereas a column-major tile improves locality, but may introduce memory bandwidth contention.

Formulas for the traditional block-cyclic mapping of a block (x, y) to a processor with tiles of length l and width w are as follows:[6]

$$\begin{aligned} m_1 &= x \bmod l && \text{(x in tile)} \\ n_1 &= y \bmod w && \text{(y in tile)} \\ f_{row}(x,y,l,w) &= m_1 w + n_1 && (1) \\ f_{col}(x,y,l,w) &= n_1 l + m_1 && (2) \end{aligned}$$

If all the processors are to be used, lw must equal the number of processors. Our testing and existing benchmarks show that a row-major tile tends to perform better. Hence, all the performance measurements presented use a row-major tile.

Figure 6 shows that a tile that is overly tall or wide decreases performance. From this figure it is apparent that an approximately square tile yields the best performance. However, performance analysis reveals that while a square tile yields the best performance, a tall tile (which increases the number of processors on the active panel) causes the active panel to compute faster, which might enable better overall performance with further modifications. To combine the benefits of a fast active panel and sufficient U parallelism, we use a tall tile but apply *rotation* to each tile depending on its location.

The rotation parameter r cycles the tile in the x direction by r rows in each new tile across the matrix in the y direction. Intuitively, the rotate parameter has the following effect: a relatively large r that is a factor of l causes a small increase in U parallelism; a relatively small r that is a factor of l causes a large increase in U parallelism; r as a co-prime of l or $r = 1$ causes the maximum amount of U parallelism.

$$\begin{aligned} p &= \left\lfloor \frac{y}{w} \right\rfloor && \text{(tile y index)} \\ m_2 &= (x + pr) \bmod l && \text{(x in tile)} \\ n_2 &= y \bmod w && \text{(y in tile)} \\ f_{rotRow}(x,y,l,w,r) &= m_2 w + n_2 && (3) \\ f_{rotCol}(x,y,l,w,r) &= n_2 l + m_2 && (4) \end{aligned}$$

Figure 7(b) shows a tall tile with a rotation of 2, the result of $f_{rotRow}(x, y, 6, 4, 2)$.

The performance effects of applying rotation to a tall tile are shown in figure 8. In figure 6 performance degrades below 6 GFlops/core when a relatively tall tile is used (right side of the graph). However, by applying an adequate amount of rotation (obtaining sufficient U parallelism) the performance for tall tiles increases to over 6 GFlops/core as shown in figure 8. If rotation is used excessively, it can create an abundance of U parallelism, which increases network traffic beyond its capability. Therefore, it is important to tune this parameter to the system configuration and network that is being used.

The choice of row- or column-major tiling represents a tradeoff between network locality from keeping the active panel on fewer nodes (favoring column-major) and contention within each node among the processors performing memory-

[6] The *mod* operator used in the following formulas is the typical C % operator.

0	1	2	3	0	1	2	3	0
4	5	6	7	4	5	6	7	4
8	9	10	11	8	9	10	11	8
12	13	14	15	12	13	14	15	12
16	17	18	19	16	17	18	19	16
20	21	22	23	20	21	22	23	20
0	1	2	3	0	1	2	3	0
4	5	6	7	4	5	6	7	4
8	9	10	11	8	9	10	11	8

(a) 32 PEs, 6x4 tall tile. Uses equation 1, $f_{row}(x, y, 6, 4)$.

0	1	2	3	8	9	10	11	16
4	5	6	7	12	13	14	15	20
8	9	10	11	16	17	18	19	0
12	13	14	15	20	21	22	23	4
16	17	18	19	0	1	2	3	8
20	21	22	23	4	5	6	7	12
0	1	2	3	8	9	10	11	16
4	5	6	7	12	13	14	15	20
8	9	10	11	16	17	18	19	0

(b) 32 PEs, 6x4 tall tile, rotate 2. Uses equation 3, $f_{rotRow}(x, y, 6, 4, 2)$.

0	1	12	13	4	5	16	17	8
2	3	14	15	6	7	18	19	10
4	5	16	17	8	9	20	21	0
6	7	18	19	10	11	22	23	2
8	9	20	21	0	1	12	13	4
10	11	22	23	2	3	14	15	6
0	1	12	13	4	5	16	17	8
2	3	14	15	6	7	18	19	10
4	5	16	17	8	9	20	21	0

(c) 32 PEs, 6x4 tall tile, rotate 2, stride 2. Uses equation 5, $f_{stride}(x, y, 6, 4, 2, 2)$.

Figure 7: Block-cyclic mapping scheme using a tall tile and applying rotation and stride to increase performance. Rotation increases the amount of U parallelism and striding increases active panel locality.

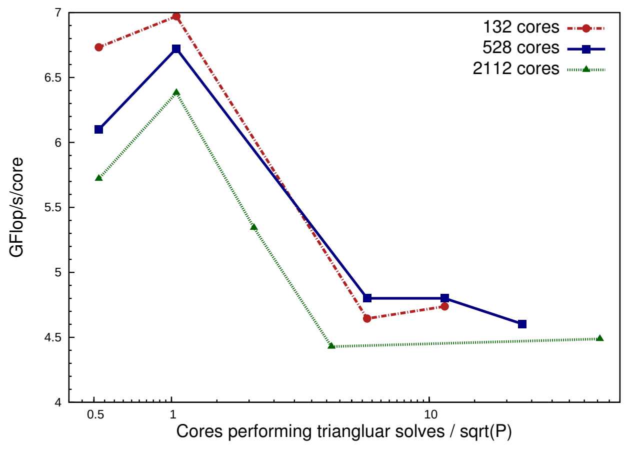

Figure 8: The effect of adding rotation to a tall tile to gain the benefits of a fast active panel, which is provided by a tall tile, and U parallelism which is recovered by rotating the tile in the x direction. The x-axis is scaled by $\sqrt{P}$ to illustrate that the ideal U parallelism matches the amount given by a square tile. Results are from weak scaling on XT5.

access intensive rank-1 updates as part of the active panel (favoring row-major).

In order to explore intermediate points between these extremes, we introduce the notion of a stride in the y direction. The stride parameter s is a generalization of controlling whether the tile is column- or row-major. When it is between column- and row-major, it changes the number of processors per node that work on the active panel. When $s = 1$ the tiling is effectively column-major; when $s = w$ the tiling is row-major. The domain of s is $1 \leq s \leq w$ where s is a factor of w.

$$p = \left\lfloor \frac{y}{w} \right\rfloor \qquad \text{(tile y index)}$$

$$m_3 = (x + pr) \bmod l \qquad \text{(x in tile)}$$

$$n_3 = y \bmod s \qquad \text{(y in tile)}$$

$$q = \left\lfloor \frac{y \bmod w}{s} \right\rfloor \qquad \text{(subtile y)}$$

$$f_{stride}(x, y, l, w, r, s) = m_3 s + n_3 + lsq \qquad (5)$$

Figure 7(c) shows the a tall tile with a rotation of 2 and a stride of 2, the result of $f_{stride}(x, y, 6, 4, 2, 2)$.

The optimal stride parameter will vary depending on the memory bandwidth of a node, the width of the node, and the prevalence of noise on the machine. If the machine is noisy, spreading the active panel over more nodes may decrease performance due to a potential loss of synchronization.

Figure 9 demonstrates the trade-off between active panel locality and memory bandwidth. With many active panel processors per node (a low stride, to the right on the graph) performance is low due to memory bandwidth limits. This maps the active panel on the smallest set of nodes possible (the same as a column-major tiling). A larger stride (the left side of the graph) approximates a row-major tile which spreads the active panel across many nodes and provides locality to the U blocks. By mapping U blocks close together, this causes contention for network bandwidth along with non-locality for the active panel.

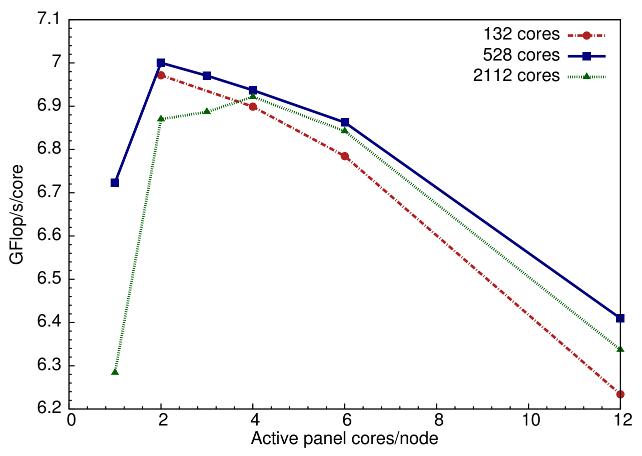

Figure 9: Performance effects of striding the block-cyclic mapping with rotation and a tall tile. 1 PE per node is effectively row-major (stride = tile width), while 12 PEs per node is effectively column major (stride = 1). Results are from weak scaling on XT5.

By varying aspect ratio, rotation, and striding we explicitly consider the degree of parallelism, locality, and memory contention. These considerations yield performance beyond that observed by using the originally optimal square tile.

6. LIMITING NETWORK CONTENTION

Dense LU factorization is not generally considered a network-intensive parallel operation, since its computation asymptotically dominates its communication. However, it presents communication patterns that involve moving large volumes of data (matrix blocks) in a 'bursty' fashion from a few source processors to many recipients. In a synchronous implementation, these bursts of communication can be implemented as efficient collective broadcasts to statically known subsets of processors (e.g. the 'process rows' and 'process columns' in HPL). In a pull-based implementation, however, recipient processors may request blocks at any time, and the owner of a block will need to respond quickly enough that the requester does not run out of work and idle.

Testing shows that responding to these requests one-by-one as they arrive leads to network saturation on processors owning blocks that are in high demand. This saturation stretches the time the sender spends responding, and delays arrival of the response on requesting processors.

To address network saturation, we dynamically batch block requests to efficiently broadcast blocks and spread the network load. Requests for a block arriving before that block is ready are batched and sent in a single broadcast when the block's computation is complete. However, requests arriving later have no inherent method for batching into broadcast groups. Thus, we limit the number of large outgoing messages that each processor may have in flight at a time. When a request for a block arrives, the requesting processor is put on a list of requesters for the block, and the block puts itself in a send queue. Eventually, as sends complete, each queued block will reach the head of the queue.

When a block reaches the head of the send queue, it will have accumulated a list of several processors that have requested the block since the last time that block was sent. The list of requesting processors participating in a broadcast is transmitted by constructing a binary spanning tree on the fly. This enables dynamic, asynchronous collective communication with negligible additional latency and little message size overhead.

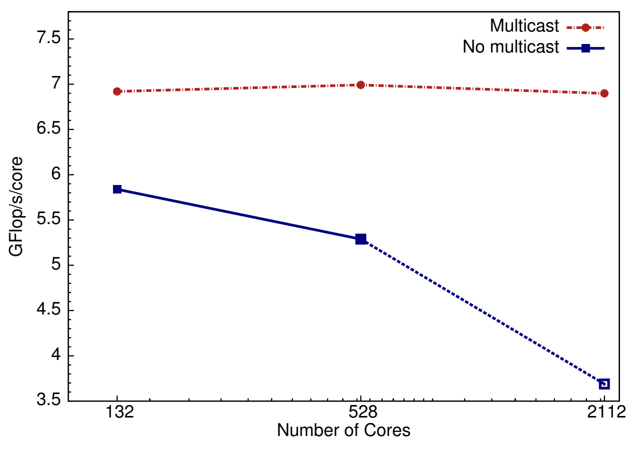

Figure 10: Performance effects of agglomerating work and multicasting it on-the-fly to destinations. The 2112 point of the no multicast curve did not finish in allotted time; hence it represents the maximum performance that configuration could have achieved.

Figure 10 shows that our multicast scheme substantially outperforms point-to-point responses to each request.

7. TRAILING SUBMATRIX UPDATES

To achieve high machine utilization, and hence good performance, the active panel and trailing update calculations must be overlapped. Specifically, the active panel for a step t should finish early enough before the trailing updates from step $t-1$ such that no processor idles while waiting for input data for step t's trailing updates. In strong scaling scenarios and in the large weak-scaled runs, each active panel may take longer to factor than all of the trailing updates it generates. Thus, to maintain overlap throughout the factorization, active panels should be executed as eagerly as possible while staying within memory limits.

In a matrix decomposed into $N \times N$ blocks, the factorization of active panel t enables $(N-t)^2$ trailing updates. However, only $N-t$ of those updates must complete before the factorization of active panel $t+1$ can start. Despite this, the UPC implementation allocates memory for these updates in strict step order, as a conservative means to avoid deadlock. Thus, with a matrix that is large relative to available memory, it must execute most of each step's updates before making space for the next step, and lookahead is very limited until late in the factorization, when little of the matrix remains to be updated.

In order to explore less conservative scheduling policies, we formalize the dependence structure in terms of *planned operations*, those for which memory has been reserved. These include both triangular solves and trailing updates, but not pivoting, since it consumes a minimal amount of memory.

For each block (x, y), major operations on it are denoted as a triple (x, y, t). Every block will go through trailing updates

$$(x, y, t) \quad | \quad 0 \leq t < min(x, y).$$

Blocks below the diagonal, $x > y$, become part of the active panel after their last trailing update and so have no more operations to plan. Blocks above the diagonal, $x < y$, complete their trailing updates and then perform a triangular solve, whose triple will always be of the form (x, y, x). For simplicity, this formulation conservatively subsumes pivoting operations into whatever major operation follows them, since they consume little additional memory (obviating the need to plan them explicitly).

A triangular solve (x, y, x) depends on its final trailing update

$$(x, y, x-1) \prec (x, y, x)$$

the final trailing updates to its associated active panel

$$(i, x, x-1) \prec (x, y, x) \quad | \quad x \le i < N$$

and (due to pivoting) the previous step's trailing updates on the column below it

$$(i, y, x-1) \prec (x, y, x) \quad | \quad x \le i < N.$$

A trailing update (x, y, t) directly depends on the previous update to that block

$$(x, y, t-1) \prec (x, y, t) \quad | \quad t > 0$$

and the triangular solve of its U input

$$(t, y, t) \prec (x, y, t).$$

The dependence on U creates a transitive dependence on the corresponding block of L, and so need not be considered explicitly.

If operations are planned strictly in step order, with triangular solves preceding trailing updates, these dependencies are effectively expanded to include the entire trailing submatrix at step $t-1$ for every step t triangular solve. Under that policy, it is clear that all dependencies will be planned before their dependents, and this will create a deadlock-free schedule. Moreover, this is a policy that every processor can follow independently, without communication to coordinate decision-making. This is the policy followed by the UPC implementation.

We have two desiderata for a less conservative scheduling policy. First, it should enable overlap to the greatest extent possible. Second, it should require little or no non-local information to operate correctly, because coordinating multiple processors operating asynchronously can be expensive, error-prone and difficult to reason about.

Given a set of operations S that can be considered done or planned at some point in time, the operations $E(S)$ eligible for planning can be determined by which dependencies are satisfied:

$$\begin{aligned} E(S) = \quad & \{(x,y,t) | \quad t < min(x,y) \quad \wedge \\ & \qquad ((x,y,t-1) \in S \vee t = 0) \quad \wedge \\ & \qquad (t,y,t) \in S\} \quad \cup \\ & \{(x,y,x) | \quad x < y \wedge (x,y,x-1) \in S \quad \wedge \\ & \qquad (\{(i,x,x-1),(i,y,x-1) | x \le i < N\} \\ & \qquad \subseteq S \vee x = 0)\} \end{aligned} \tag{6}$$

If these precise dependencies are applied on a local, per-processor basis, deadlock can result, as shown in figure 11. This occurs because the trailing updates necessary to pivot data for some triangular solve can be mutually blocked by other trailing updates across two or more processors. Step-order planning does not give rise to cases like this; specifically, it guarantees that the trailing updates on one processor needed to generate pivots for a triangular solve would be planned before any trailing updates that may depend on that triangular solve's output.

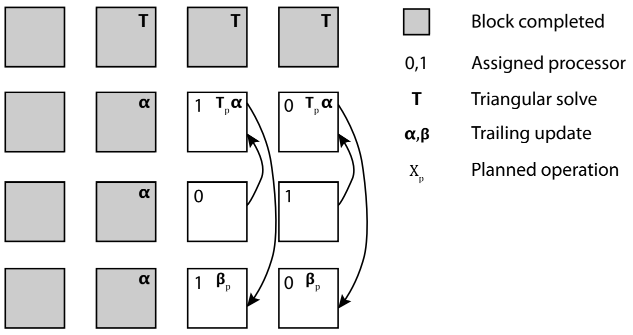

Figure 11: Possible deadlock situation if only local dependencies are considered. This is an example with two processors and an allowed planning depth of two. The two processors that try to execute β_p are dependent on their T_p triangular solves. These two solves are dependent for pivoting on the two non-local blocks that have not completed α. Since these blocks are not planned, deadlock ensues.

Suppose that processor has the information that the triangular solve $(t+1, y, t+1)$ has completed, despite some of its blocks in column y not having been updated to step t, and thus unable to pivot with $(t+1, y)$. This means that those blocks contained no pivot rows for that step. The completion of the triangular solve with no contribution from those blocks lets that processor delay planning updates to those blocks, in favor of step-wise later updates that might be closer to critical for the active panel.

Our baseline implementation follows the conservative step order that avoids the possibility of deadlock. However, we deviate from step order by exploiting the information about finished triangular solves in a limited fashion. When a U block on the first block super-diagonal does its triangular solve, it broadcasts a notice of this progress to the scheduler objects on all of the processors. That broadcast is used to release conservatively set dependences that would hold back the next active panel. In its limited form, the benefits of this are small: about a 0.5% increase in peak.

A more complete implementation would make the same release notification from every triangular solve, allowing columns that are a few steps away from being on the active panel to run further ahead of other columns further to the right in the matrix. The challenge, then, would be for each scheduler to determine how much memory to allocate to updates on which part of the matrix, given a need to balance fastest immediate progress with having work to do when otherwise idle. We plan to explore this space in future work.

8. RELATED WORK

HPL [24], a high performance implementation of LU factorization, uses a bulk synchronous approach along with fixed lookahead, obviating the need for scheduling classes. Its communication and computational patterns are described in a paper by Dongarra, Luszczek, and Petitet [12]. ScaLAPACK [10] uses static scheduling with no lookahead. Both implement parallel algorithms for LU that differ greatly from our implementation.

UPC [16] uses a similar parallel algorithm, namely the "dataflow" variant of the algorithm. They use a pull-based scheme to constrain memory, but do not agglomerate requests. Instead of using local coordination with prioritization for work unit control-flow, they use user-level threads which are co-operatively scheduled. They explain that preemption is not viable in this context, but threads explicitly yield when they start a long-latency operation or synchronization dependence. It is possible that applying the idea of exclusive scheduling classes to threads could be beneficial.

Lewis and Richards [22] use a dataflow parallel algorithm, but only apply it to shared memory up 24 processors. Kurzak and Dongarra [21] investigate the dataflow algorithm with pipelining and lookahead (both techniques we used) on two dual-core processors.

Krishnan, Lewis, and Vishnu [20] describe methods using ARMCI for optimizing communication efficiency in LU for smaller problem sizes using RDMA.

Amestoy, Guermouche, and Pralet [2] use a master/slave model with dynamic scheduling for LU and constrain memory within this scheme.

Bosilca, et. al. describe DPLASMA [4], a library for distributed dense numerical linear algebra that uses the DAGuE framework [5], which prioritizes and schedules tasks using the application DAG. Chan, et. al. use SuperMatrix [8] to perform LU factorization without pivoting and schedule tasks using the data dependencies.

A theoretical understanding of when synchrony is effective and ineffective has garnered interest with the increased concurrency found in high-performance computing. Hoefler, et al. [15] developed a simulation toolchain that injects noise delays from traces gathered on common large-scale architectures into a simulator. They noted that the scale at which noise becomes a bottleneck for synchronous operations is system specific and depends on the structure of the noise, and that co-scheduling is able to effectively remove the negative impact on synchronous collectives. Ferreira, et al. [13] used noise injection techniques to assess synchronous behavior on several applications at scale and noted the importance of noise frequency and duration. Co-scheduling in our case would help synchronize the processors as they start non-active panel work, but it would still interleave grains that should ideally be delayed.

9. CONCLUSION

In this paper, we describe an asynchronous, message-driven implementation of the common dense LU factorization benchmark, and show that it successfully scales to thousands of processors on multiple, leading-edge supercomputer architectures. In doing so, we explore several challenges and opportunities presented by fully asynchronous parallel programming models. We show how opportunistic execution can negatively impact performance in certain circumstances, and describe a mechanism to mitigate that loss. Performing work out of lock-step synchronization also introduces new considerations for parallelism and resource contention, which we address through variations on traditional mapping schemes and adaptive collective communication mechanisms. Given a well-tuned asynchronous implementation to use as a baseline, we have begun to explore the more flexible deep-lookahead schedules for dense LU that are possible but untested.

Acknowledgements

Grants: Blue Waters (Phil), Simdemics, IACAT? Hecura? (Jonathan), OpenAtom? (Anshu, Ram) Machines:

Development runs on Kraken were done under the TeraGrid [7] allocation grant TG-ASC050040N supported by the NSF.

This research used Surveyor and Intrepid of the Argonne Leadership Computing Facility at Argonne National Laboratory, under the support of DOE contract DE-AC02-06CH11357.

Running time on Jaguar, a resource of the National Center for Computational Sciences at Oak Ridge National Laboratory, was supported by DOE contract DE-AC05-00OR22725. Accounts on Jaguar were made available via the Performance Evaluation and Analysis Consortium End Station, a DOE INCITE project.

10. REFERENCES

[1] Top500 supercomputing sites. http://top500.org.

[2] P. R. Amestoy, A. Guermouche, and S. Pralet. Hybrid scheduling for the parallel solution of linear systems. *Parallel Computing*, 32:136–156, 2006.

[3] B. Bland. Hpc challenge class i award g-hpl winning submission, 2010.

[4] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, A. Haidar, T. Herault, J. Kurzak, J. Langou, P. Lemarinier, H. Ltaief, P. Luszczek, A. Yarkhan, and J. Dongarra. Distibuted [sic.] dense numerical linear algebra algorithms on massively parallel architectures: Dplasma. Technical Report UT-CS-10-660, University of Tennessee, September 2010.

[5] G. Bosilca, A. Bouteiller, A. Danalis, T. Herault, P. Lemarinier, and J. Dongarra. DAGuE: A generic distributed DAG engine for high performance computing. Technical Report ICL-UT-10-01, Innovative Computing Laboratory, University of Tennessee, April 2011.

[6] R. Brightwell, B. Barrett, K. Hemmert, and K. Underwood. Challenges for high-performance networking for exascale computing. In *Computer Communications and Networks (ICCCN), 2010 Proceedings of 19th International Conference on*, pages 1 –6, 2010.

[7] C. Catlett et al. TeraGrid: Analysis of Organization, System Architecture, and Middleware Enabling New Types of Applications. In L. Grandinetti, editor, *HPC and Grids in Action*, volume 16, pages 225–249, Amsterdam, 2007. IOS Press.

[8] E. Chan, F. Van Zee, E. Quintana-Orti, G. Quintana-Orti, and R. van de Geijn. Satisfying your dependencies with supermatrix. In *Cluster Computing, 2007 IEEE International Conference on*, pages 91 –99, September 2007.

[9] A. Chandramowlishwaran, K. Knobe, and R. Vuduc. Performance evaluation of concurrent collections on high-performance multicore computing systems. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS)*, 2010.

[10] J. Choi, J. Dongarra, and D. Walker. The Design of Scalable Software Libraries for Distributed Memory Concurrent Computers. In H. Siegel, editor, *Proc. Eighth International Parallel Processing Symposium.* IEEE Computer Society Press, April 1994.

[11] J. Dongarra and P. Luszczek. Introduction to the HPC Challenge Benchmark Suite. Technical Report UT-CS-05-544, University of Tennessee, Dept. of

Computer Science, 2005.

[12] J. J. Dongarra, P. Luszczek, and A. Petitet. The linpack benchmark: Past, present, and future. concurrency and computation: Practice and experience. *Concurrency and Computation: Practice and Experience*, 15:2003, 2003.

[13] K. B. Ferreira, P. Bridges, and R. Brightwell. Characterizing application sensitivity to os interference using kernel-level noise injection. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, SC '08, pages 19:1–19:12, Piscataway, NJ, USA, 2008. IEEE Press.

[14] G. R. Gao, T. L. Sterling, R. Stevens, M. Hereld, and W. Zhu. Parallex: A study of a new parallel computation model. In *IPDPS*, pages 1–6, 2007.

[15] T. Hoefler, T. Schneider, and A. Lumsdaine. Characterizing the influence of system noise on large-scale applications by simulation. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–11, Washington, DC, USA, 2010. IEEE Computer Society.

[16] 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, New York, NY, USA, 2007. ACM.

[17] L. Kalé and S. Krishnan. CHARM++: A Portable Concurrent Object Oriented System Based on C++. In A. Paepcke, editor, *Proceedings of OOPSLA'93*, pages 91–108. ACM Press, September 1993.

[18] L. V. Kale and M. Bhandarkar. Structured Dagger: A Coordination Language for Message-Driven Programming. In *Proceedings of Second International Euro-Par Conference*, volume 1123-1124 of *Lecture Notes in Computer Science*, pages 646–653, September 1996.

[19] D. Keyes. Partial differential equation-based applications and solvers at extreme scale. *Int. J. High Perform. Comput. Appl.*, 23:366–368, November 2009.

[20] M. Krishnan, R. Lewis, and A. Vishnu. Scaling linear algebra kernels using remote memory access. In *Parallel Processing Workshops (ICPPW), 2010 39th International Conference on*, pages 369 –376, September 2010.

[21] J. Kurzak and J. Dongarra. Implementing linear algebra routines on multi-core processors with pipelining and a look ahead. In *Proceedings of the 8th international conference on Applied parallel computing: state of the art in scientific computing*, PARA'06, pages 147–156, Berlin, Heidelberg, 2007. Springer-Verlag.

[22] B. Lewis and K. Richards. Lu factorization case study using fast: Dataflow parallelism with the forte application scalability tool, 2003.

[23] R. Murphy, T. Sterling, and C. Dekate. Advanced architectures and execution models to support green computing. *Computing in Science Engineering*, 12(6):38 –47, 2010.

[24] A. Petitet, R. C. Whaley, J. Dongarra, and A. Cleary. HPL - a portable implementation of the high-performance linpack benchmark for distributed-memory computers.

[25] J. Reinders. *Intel Threading Building Blocks: Outfitting C++ for Multi-Core Processor Parallelism.* O'Reilly Media, 2007.

[26] Y. Saada. Communication complexity of the gaussian elimination algorithm on multiprocessors. *Linear Algebra and its Applications*, 77:315–340, May 1986.