Mapping Dense LU Factorization on Multicore Supercomputer Nodes

Jonathan Lifflander, Phil Miller,
Ramprasad Venkataraman, Anshu Arya, Laxmikant Kale
University of Illinois Urbana-Champaign
Email: {jliffl2, mille121, ramv, arya3, kale}@illinois.edu

Terry Jones
Oak Ridge National Laboratory
Email: trj@ornl.gov

***Abstract*—Dense LU factorization is a prominent benchmark used to rank the performance of supercomputers. Many implementations use block-cyclic distributions of matrix blocks onto a two-dimensional process grid. The process grid dimensions drive a trade-off between communication and computation and are architecture- and implementation-sensitive. The critical panel factorization steps can be made less communication-bound by overlapping asynchronous collectives for pivoting with the computation of rank-k updates. By shifting the computation-communication trade-off, a modified block-cyclic distribution can beneficially exploit more available parallelism on the critical path, and reduce panel factorization's memory hierarchy contention on now-ubiquitous multicore architectures.**

During active panel factorization, rank-1 updates stream through memory with minimal reuse. In a column-major process grid, the performance of this access pattern degrades as too many streaming processors contend for access to memory. A block-cyclic mapping in the row-major order does not encounter this problem, but consequently sacrifices node and network locality in the critical pivoting steps. We introduce striding to vary between the two extremes of row- and column-major process grids.

The maximum available parallelism in the critical path work (active panel factorization, triangular solves, and subsequent broadcasts) is bounded by the length or width of the process grid. Increasing one dimension of the process grid decreases the number of distinct processes and nodes in the other dimension. To increase the harnessed parallelism in both dimensions, we start with a tall process grid. We then apply periodic rotation to this grid to restore exploited parallelism along the row to previous levels.

As a test-bed for further mapping experiments, we describe a dense LU implementation that allows a block distribution to be defined as a general function of block to processor. Other mappings can be tested with only small, local changes to the code.

I. Introduction

Dense LU factorization is well understood and widely established as a supercomputer benchmark used to rank systems for the top500 list [31]. It is an important parallel kernel due to its use in scientific applications and is representative of other important algorithms. Its performance is primarily computation bound, but it presents distinct challenges in memory allocation and access, communication, and scheduling to obtain high efficiency.

The many challenges in implementing LU efficiently present tradeoffs influenced by the interplay of processor microarchitecture, node design, and interconnection network. Recent architectural trends have engendered supercomputers with an increasing numbers of cores per node: successive generations of Blue Gene have 2, 4, and 16 compute cores [17]; Cray's XT3 had 1–2 cores [32], while the XE6 has 24 cores [8]; the bullx S6010 nodes in today's 9th-ranked Tera-100 cluster can support up to 40 cores [4]. The computational capacity of each core has increased greatly in that time, but per-core cache and memory bandwidth have not kept up. Networks have seen relatively incremental improvements in bandwidth and latency. In the push from petascale to exascale computing, new techniques will be required to optimally exploit the capabilities of the available systems.

To illustrate new techniques that improve performance, this paper presents a high-performance dense LU implementation, benchmarked on modern supercomputers, and demonstrates scalability to thousands of processors. These techniques are derived from the reconsideration of old assumptions, architectural analysis, microbenchmark studies, and efforts to improve overall performance. These techniques are empirically tested and shown to substantially improve performance, making this LU implementation competitive with other libraries.

The contributions of this paper include:

- § III a compact implementation of dense LU that allows easy experimentation with new forms of overlap and arbitrary mappings of blocks to processors[1];
- § IV *striding* to allow an explicit tradeoff between memory contention effects and network locality during panel factorization by mapping blocks to the process grid in a hybrid of row- and column-major order;
- § V *rotation* in the mapping of blocks to adjust relative degrees of parallelism between panel factorization and triangular solves to match architectural parameters.

The novel mappings are illustrated in Figure 1. These new techniques work well together, and only manifest minimal potential overhead (§ VI). The efficiency and scalability of our implementation is demonstrated on thousands of cores of the Jaguar Cray XT5 and Intrepid Blue Gene/P supercomputers (§ VII).

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

II. Background

Dense LU factorization is commonly used as a means to solve dense linear systems. A set of n linear equations in n variables is solved by performing LU factorization and solving the resulting triangular systems. The algorithm has a few different variants, one of which is Crout's algorithm which performs an in-place factorization. Numerical stability is achieved via partial pivoting.

Most parallel formulations of LU are blocked algorithms with underlying sequential operations delegated to a high performance linear algebra library (e.g. an implementation of BLAS). The matrix is typically decomposed into square blocks of size b^2 (shown in Figure 2) and distributed across a set of processors.

A. Algorithm

The parallel factorization process can be described as follows:

```
1) for step in $0..\frac{n}{b} - 1$:
   Active panel blocks are those at/below diagonal block step
   a) for column in $0..b$: (on each active panel block)
      i) 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.
      ii) 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.
      iii) Each active panel block performs a rank-1 update of the section after column with multipliers from column and the pivot row.
   b) 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.
   c) Active panel blocks send their contents, each a portion of L, to the blocks to their right.
   d) U blocks to the right of the diagonal each perform a triangular solve, and send the result to the blocks below them.
   e) Blocks in the trailing submatrix each compute a trailing update as the product of the L and U blocks they have received.
```

B. Data Distribution

There are several considerations to take into account when distributing the matrix blocks onto the available processors. The amount of computation that has to be performed on each block of data to achieve the final factorization differs based on the location of the block in the overall matrix. A two dimensional block-cyclic data distribution achieves satisfactory load balance while also presenting good communication behavior and scalability, and being easy to implement and understand [12].

For such a data distribution, the available processes (N) are first arranged into a two-dimensional process grid of dimensions $P \times Q$ such that $N = P \times Q$. The aspect ratio of the process grid and how the process ranks are arranged within this grid depend on system architecture.

The two dimensional block cyclic pattern can be obtained by tiling the process grid onto the matrix as shown in Figure 1b. The figure depicts a matrix decomposed into blocks. The bold lines are the boundaries of the the process grid and the label in each block is the process rank to which it is *mapped*.

The process rank for each block is computed as a function of the block's coordinates:

$$\text{process rank} = f(x_{block}, y_{block}). \tag{1}$$

The following are formulæ[2] for the traditional block-cyclic mapping of a block, whose 0-based coordinates in the blocked view of the input matrix are (x, y), onto a process grid of dimensions $P \times Q$:

$$\begin{aligned} m_1 &= x \bmod P && \text{(x in grid)} \\ n_1 &= y \bmod Q && \text{(y in grid)} \\ f_{row}(x, y, P, Q) &= m_1 Q + n_1 && (2) \\ f_{col}(x, y, P, Q) &= n_1 P + m_1 && (3) \end{aligned}$$

Equation 2 describes the process rank for a row-major process grid; Equation 3 describes the process rank for a column-major process grid.

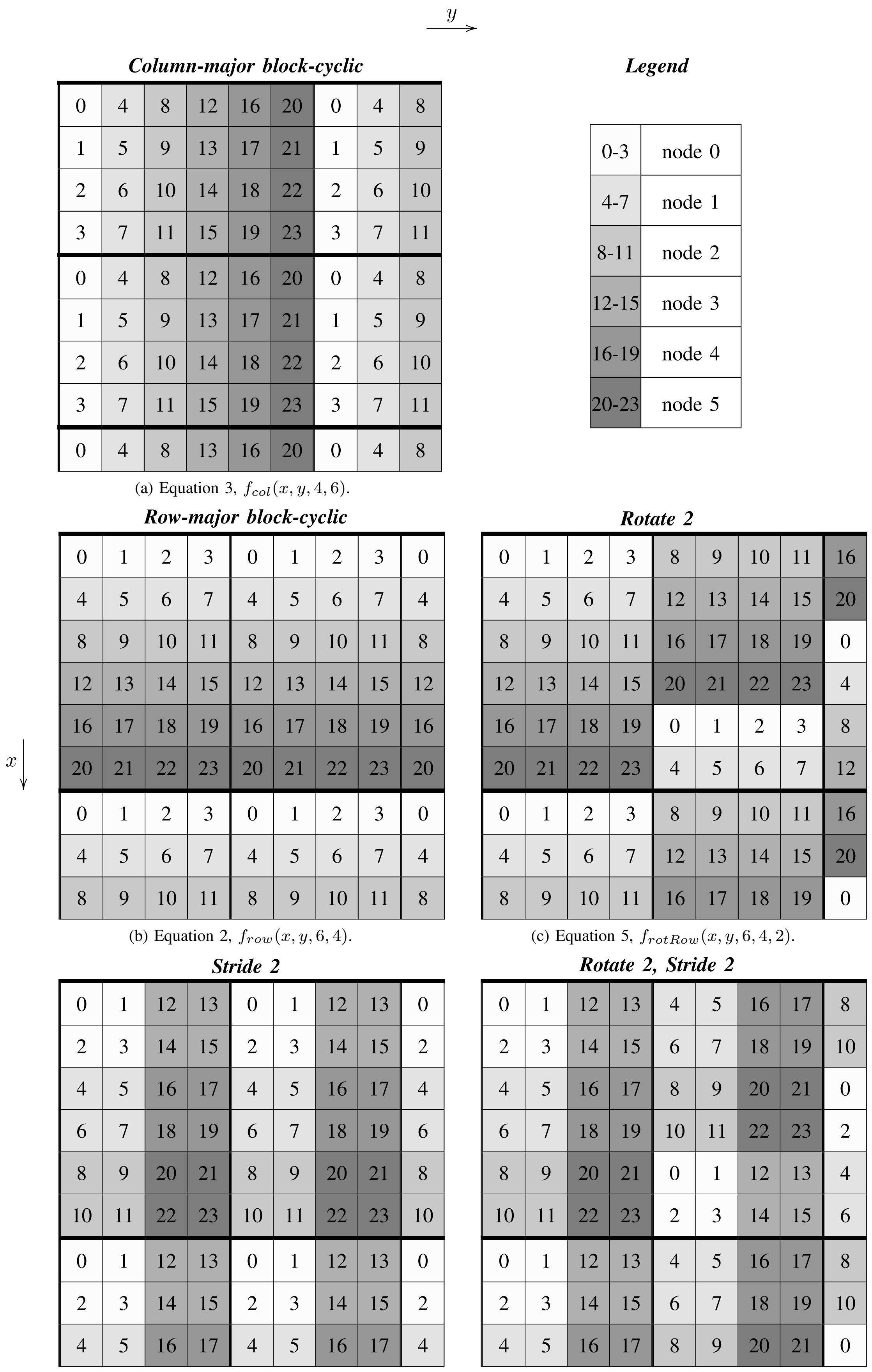
C. Granularity Spectrum

The factorization presents a challenging spectrum of computation and communication grain sizes. The trailing updates comprise the bulk of the computation in a dense LU solver. Each trailing update is an $\mathcal{O}(b^3)$ matrix-matrix multiplication (i.e. a call to the `dgemm()` level-3 BLAS routine). The triangular solves (via `dtrsm()`) are of similar computational cost. Each trailing update or triangular solve takes tens of milliseconds for the block sizes common on today's architectures. Large messages drive the heavy computation kernels. In contrast, the active panel is communication intensive, with b small-message pivot reductions and broadcasts occurring in rapid succession, interspersed with smaller computations. Each of these fine-grained steps on the active panel nominally take hundreds of microseconds to single digit milliseconds.

D. Lookahead

Ideally, every processor would remain busy during the entire factorization process. However, in each step, only a subset of processors own blocks that participate in the active panel. Thus, to avoid idling processors, work from multiple steps must be overlapped. The extent of the overlap (specifically, the number of steps that the active panel runs ahead of trailing updates) in an implementation of dense LU is known as its *lookahead depth* [28].

[2]The *mod* operator used in the formulæ is the typical C `%` operator.



Column-major block-cyclic

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

(a) Equation 3, $f_{col}(x, y, 4, 6)$.

Legend

0-3	node 0
4-7	node 1
8-11	node 2
12-15	node 3
16-19	node 4
20-23	node 5

Row-major block-cyclic

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

(b) Equation 2, $f_{row}(x, y, 6, 4)$.

Rotate 2

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

(c) Equation 5, $f_{rotRow}(x, y, 6, 4, 2)$.

Stride 2

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

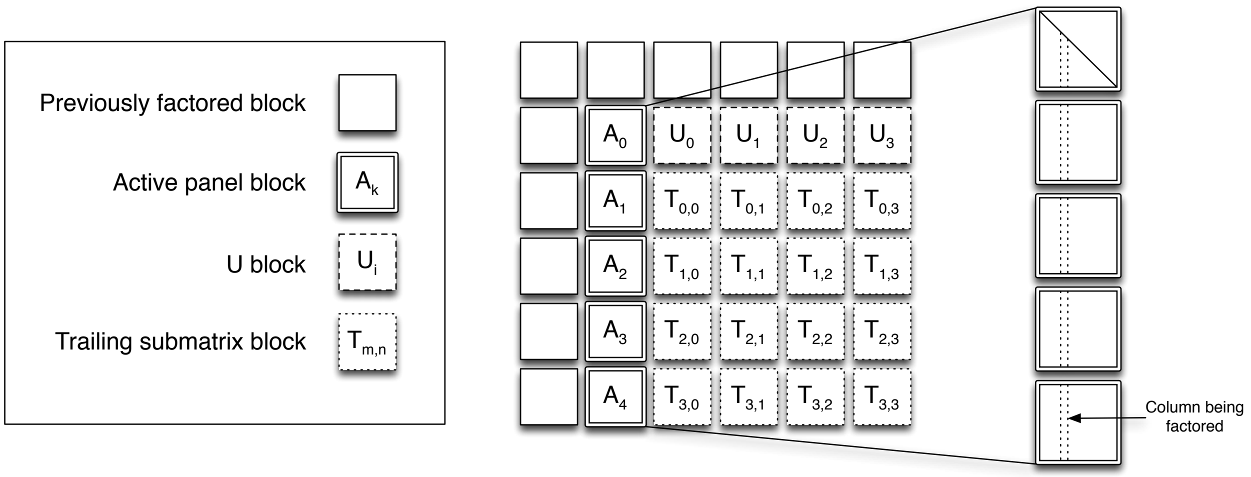
(d) Equation 4, $f_{stride}(x, y, 6, 4, 2)$.

Rotate 2, Stride 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

(e) Equation 7, $f_{strideRot}(x, y, 6, 4, 2, 2)$.

Fig. 1: Block-cyclic mapping of each logical matrix block to a processor (with 24 processors), applying *rotation* and *striding* to increase performance. Rotation increases the amount of row parallelism and striding increases active panel locality.



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

Fig. 2: The matrix is decomposed into square blocks, which assume different roles as the factorization proceeds.

III. Design

This section describes our implementation of a library for solving dense linear systems that follows the guidelines described by the HPC Challenge specifications [11]. A test driver, as required by the specification, demonstrates the use of the library. The driver feeds the library with distributed input in the form of randomly-generated matrix blocks and validates the solution by calculating the scaled residual.

A. Allowing Data Redistribution

Matrix blocks are assigned to processors when the library starts up, according to a *mapping* scheme, and are not reassigned during the course of execution. Typically, there will be tens to hundreds of blocks assigned to each processor core. The expression and modification of the data distribution scheme is simplified by encapsulating the logic into a simple sequential function call that uses the block's coordinates (shown in Figure 2) to compute the process rank it should be placed on.

This allows library users to evaluate data distribution schemes that may differ from the traditional two-dimensional block-cyclic format. This capability is used for many of the experiments outlined in this paper.

B. Programming Paradigm

In the new implementation, each block is placed in a message-driven object, driven by coordination code written in Structured Dagger [20]. The coordination code describes the message dependencies and control flow from the perspective of a block. Thus, every block can independently advance as it receives data and bulk synchrony is avoided by allowing progress in the factorization when dependencies have been met. With many blocks per processor, the Charm++ [19] runtime system inherently provides dynamic overlap of communication and computation by scheduling blocks that have received the necessary data. In general, the system ensures high utilization, since some blocks on each processor should always have work. Others implementations dynamically interleave the work performed on various blocks, either by introducing task parallelism to HPL [25] or by spawning many light-weight threads in UPC [16].

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 process 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, the source code for our implementation of the factorization library is approximately 1,650 lines long [18].[3] This is shorter than HPL, which is around 12,000 lines and the UPC implementation [16], which is around 4,000 lines of code. Furthermore, the distribution of blocks to processors is not embedded in the expression of the parallel factorization algorithm, but is instead localized to discrete mapping functions. The flexibility this provides was previously used to study some atypical mappings in an earlier non-pivoting version of this code [13].

C. 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.

D. Dynamic Lookahead

Bulk synchronous implementations, such as HPL [26], 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

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

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 [7] does no lookahead (i.e. its lookahead depth is 0).

In a message-driven, asynchronous environment, LU can be implemented to allow dynamic lookahead: the diagonal can progress without 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 object 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. An earlier technical report [23] describes the dependencies between the blocks and uses this to safely reorder the selection of trailing updates to execute.

IV. Memory Hierarchy Contention in Panel Factorization

The performance of one complete panel factorization step is influenced by the latency of its iterated substeps: collectives among a process column to identify and distribute the pivot row, and updates of the remainder of the panel by each process in the process column. Our implementation partially overlaps the collectives with the updates from the previous iteration. Regardless of the degree to which these steps can be overlapped, each must be optimized to minimize the time to solution.

Generally, the collective for column k of a b-column wide panel should take

$$\Theta((b-k)\log P)$$

time on a process grid of height P (a reduction followed by a broadcast, or a single combination of the two). More realistically, the collective cost looks like

$$\Theta((b-k)\log(P/u)\mathrm{Coll}(u))$$

where u is the number of processors per node in a given process column, and 'Coll' is the time for a node-local collective among u cores.

The rank-1 update of a single block of size $b \times b$ from column k involves updating $\Theta(b(b-k))$ words, each read and written once. In a given step, these updates will take time $\mathrm{Update}(b,k,u)$ for a given number of blocks per core. This was benchmarked on several microarchitectures by having each of the u cores generate a fixed number of blocks of

Microarchitecture	Cores/socket performing updates 1	2	3	4	5	6	Efficiency
Intel Nehalem-EP	16	22	30	38			42%
AMD Istanbul	17	27	38	50	63	76	22%
IBM Blue Gene/P	19	20	20	22			86%

TABLE I: Rank-1 update times (ms) for varying numbers of active cores

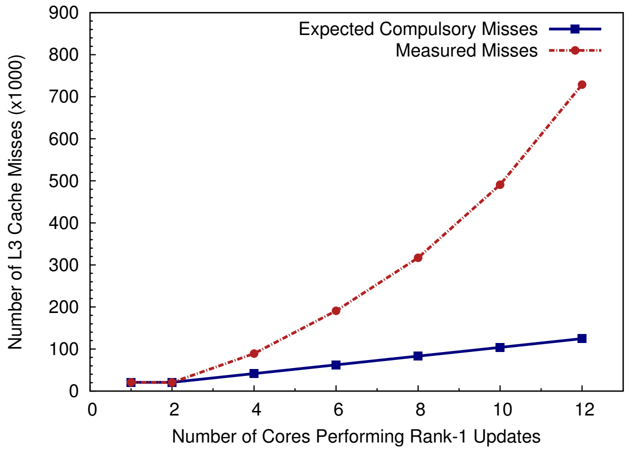


Fig. 3: Number of L3 cache misses on a node of Jaguar (dual Istanbul processors), as increasing numbers of cores are used to perform rank-1 updates

doubles randomly, along with a random input vector, and timing the application of rank-1 updates to all of the blocks. The block sizes and counts were selected to match real runs of the factorization process; specifically, the blocks were large enough to attain good utilization in DGEMM, and there were enough blocks to ensure that in each iteration, each block was re-read from memory rather than remaining in cache. The time varies with b and k as one would expect, but varying u produces striking results. Table I shows that time-per-update and throughput suffer with increasing u across a range of architectures. The same effect appears whether the other cores are idle or are performing out-of-cache DGEMMs; the idle case is shown for simplicity. Given v blocks to be updated in an entire panel, column update k will take

$$\max_{\text{nodes}} \lceil v/P \rceil \mathrm{Update}(b,k,u)$$

time.

The reason for the poor efficiency of many cores per node performing panel updates is that the update computation is essentially a memory-bound stream. Figure 3 shows that on the Istanbul microarchitecture, increasing u increases level-3 cache misses super-linearly! Various studies have shown that this effect is no surprise [24], [30]. These mainstream and supercomputer architectures simply cannot serve full stream bandwidth to all cores, limiting the performance of the few kernels with such poor computational intensity and reuse.

In order to obtain optimal performance for panel factorization, the sum (serialized) or maximum (overlapped) of pivoting and updating time must be minimized, with respect to the available parameters.

Present implementations offer direct control over P, and two choices of u. In a sufficiently-large row-major process grid, u is 1. In a column-major grid, u will equal the number of cores in a node. These possibilities are shown in Figure 1a. This choice represents a binary tradeoff among several factors, including network locality within process columns (favoring column-major) and improved panel update throughput and locality within process rows (favoring row-major). Either arrangement requires that increasing P correspondingly increases the inter-node communication of panel factorization. Thus, shorter process grids are considered essential to achieving optimal performance, even though they sacrifice parallelism in panel factorization's update steps.

The minimum time is unlikely to be found at either extreme, but rather at an intermediate point where the active panel blocks are placed on a few cores of each of the nodes involved. Block placement corresponding to these intermediate points could be achieved by rank remapping in the job launcher or runtime environment, but such a change is apt to interfere with other parts of a parallel application, in which LU factorization is just one kernel among many. Instead, P cores in a column can be selected with a *stride*, s (in ranks) between successive entries in a process column. Viewed differently, this is equivalent to constructing row-major sub-grids of width s into which ranks are mapped sequentially (Figure 1d).

The domain of s is $1 \leq s \leq Q$, where s is a factor of Q, the width of the process grid. The block-to-rank mapping this produces is given by the following formalæ.

$$\begin{aligned} p &= \left\lfloor \frac{y}{Q} \right\rfloor && \text{(grid y index)} \\ m_3 &= x \bmod P && \text{(x in grid)} \\ n_3 &= y \bmod s && \text{(y in subgrid)} \\ q &= \left\lfloor \frac{y \bmod Q}{s} \right\rfloor && \text{(subgrid y)} \\ f_{stride}(x, y, P, Q, s) &= m_3 s + n_3 + Psq && (4) \end{aligned}$$

We can now express u as t/s, where t is the total number of cores per node, and thus, the time for one collective in this arrangement is

$$\Theta((b-k)\log(Ps/t)\mathrm{Coll}(t/s)).$$

If a stride of 2 (as shown in Figure 1d) were used on a system composed of dual-socket hexa-core nodes (e.g. Cray XT5 such as Jaguar or Kraken), it would place $12/2 = 6$ cores per node in each process column. Assuming NUMA-appropriate mapping of cores in the node, 3 cores will stream from the memory system available to the processor in each socket. This arrangement gives those cores much less contended access to memory than a column-major layout, while halving the number of nodes in each process column relative to a row-major layout. In exchange for decreasing the number of nodes involved in the latency-sensitive panel factorization steps, striding increases the node count of each process row by a corresponding factor. Note that a stride of $s = Q$ degenerates to a row-major grid, $s = t$ is similar to row-major but with a different ordering among nodes, and $s = 1$ is column-major. Striding may allow an increase in P to exploit more parallelism in each panel factorization, where communication costs or memory contention may have previously made this impractical.

Figure 4 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 process grid). A larger stride (the left side of the graph) approximates a row-major process grid 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.

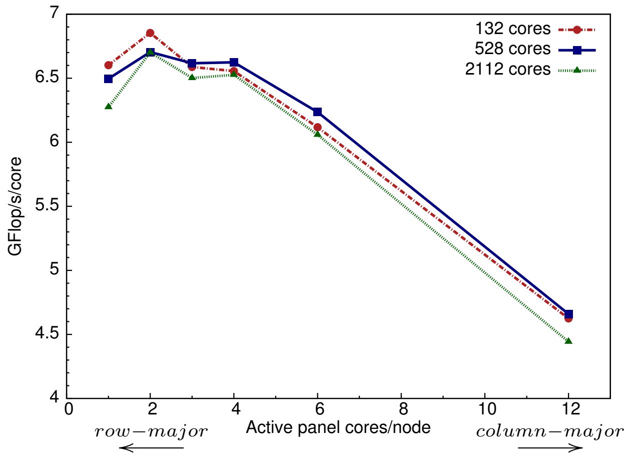
The optimal stride parameter will depend not only on the width and memory system of each node and the network connecting them, but also the prevalence of noise on the machine. The collective communication in panel factorization can be delayed by any node involved [27]. If the machine is noisy, spreading the active panel over more nodes may decrease performance due to a potential loss of synchronization among nodes performing panel factorization.

Performance and scaling results from applying various strides can be seen in Figure 4. Running on Jaguar with a nearly-square process grid, a stride of 6 (2 cores per node on a given process column, or 1 per socket) gives slightly better performance (4% of peak improvement on 2,112 cores) than row-major. With a tall process grid on Jaguar, a stride of 3 (4 cores per node) is the best of a broad sweet spot, beating the column-major layout by 5% of peak. On 256 cores in a 16×16 grid on Blue Gene/P, a column-major process grid attains 54% of peak, while row-major or any stride attains 55% of peak. This minuscule difference is explained by the machine's balanced arithmetic and memory throughput and is confirmed by the microbenchmark results shown in Table I.

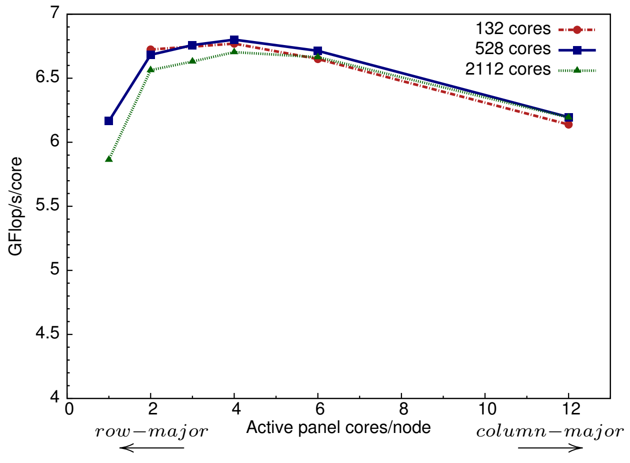
There are other possible means to address the memory contention described in this section. Different top-level panel factorization algorithms may improve the low-level reuse patterns [15], usage of multiple cores [10], and even the parallel communication structure [14]. At a lower level, one could potentially construct a customized `DGER` kernel that offers greater memory access concurrency and latency tolerance than current implementations. Such an implementation could be parameterized by the expected degree of contention and autotuned to the characteristics of each architecture [22], [33], [34]. On architectures that have larger queues and buffers but still suffer conflict misses in prefetching, cache partitioning [9] could also be helpful.

V. Increasing Parallelism Along Both Dimensions

Blocked LU with partial pivoting has substantial asymmetries in both computation and communication activities



(a) Square process grid



(b) Tall process grid

Fig. 4: Performance effects of striding the block-cyclic mapping with square and tall process grids. Results are from weak scaling on XT5.

between process rows and process columns. Each row participates in large-message broadcasts, and shares the work of sets of triangular solves on $b \times b$ blocks. In contrast, each column performs many small collectives interleaved with low-rank panel updates. Thus, the aspect ratio of a block-cyclic distribution's process grid determines a balance of parallelism and locality in each of these sections.

A wide process grid (more distinct processes in each row than column) allows more triangular solves to proceed concurrently on the process row, whereas a tall process grid realizes greater parallelism along the column in the active panel factorization, and in the subsequent broadcast that feeds the matrix-matrix multiplications. Traditionally, in HPL, more parallelism along the column tends to decrease performance because the fine-grained communication in the active panel dominates the computation (the rank-k updates). In this situation, a wide process grid typically yields the highest performance.

Using asynchronous collectives to overlap the communication in the active panel factorization with the computation allows the panel factorizations to complete faster. This also hides the latencies of the communication process and permits spreading the panel blocks across more processes, thereby speeding up the panel factorization. This effect is shown in Figure 5 where the amount of time to factorize each panel of matrix blocks is graphed for wide, square, and tall process grids. Going from a wide process grid to a square process grid yields an almost perfect speedup, reducing time from 6 seconds to 3 seconds for the first panel. Stepping from square to tall, however, only provides a 1.5x speedup, due to communication in the active panel starting to dominate. Even though efficiency is lower in this case, executing work more aggressively along the critical path is crucial to keeping processors busy because active panel work generates the inputs for the matrix-matrix multiplies, which comprise the bulk of the computation.

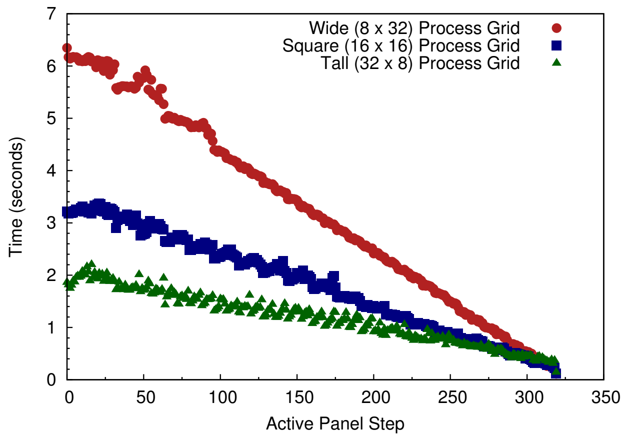


Fig. 5: The time to factorize each panel of matrix blocks for wide, square, and tall process grids. The tall process grid reduces the time due to increased exploitation of computational parallelism along the column of the process grid. Experiments were performed on BG/P on 256 cores using a 96000×96000 matrix with a block size of 300×300 in a row-major process grid.

The tall process grid reduces the active panel time, but the overall performance is worse than the square process grid because fewer processes are available for the triangular solves along each row. Using a wide process grid also decreases overall performance from a square process grid because compute resources along the process grid columns are reduced. Figure 6 shows the impact that various process grid aspect ratios have on overall performance. These experiments were conducted with a row-major process grid.

To increase the parallelism exploited in both dimensions, a tall process grid is chosen (which minimizes active panel time), but instead of repeating the same grid over the entire

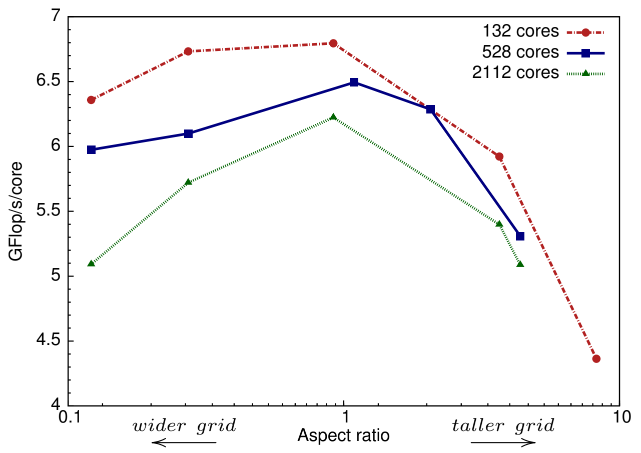


Fig. 6: Various aspect ratios for the process grid of a row-major block-cyclic distribution. Approximately square process grids perform the best. Tall process grids decrease performance because they lack parallelism in U. Wide process grids decrease performance because they slow down the active panel.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

blocked matrix, the processor rows are *rotated* by shifting them a variable amount depending on their location in the matrix. The effect that this parameter has on the process grid is visualized in Figure 1c. The tall grid allots more compute resources to the panel factorizations, while the rotation compensates for this and restores the number of compute resources available along the process grid rows.

The following formulæ formalize the notion of rotation. An additional parameter r augments the traditional block-cyclic distribution, determining the amount of rotation that is applied to the process grid. The grid is cycled in the x direction by r rows in each subsequent overlay of the process grid across the matrix in the y direction.

$$p = \left\lfloor \frac{y}{Q} \right\rfloor \qquad \text{(grid y index)}$$
$$m_2 = (x + pr) \bmod P \qquad \text{(x in grid)}$$
$$n_2 = y \bmod Q \qquad \text{(y in grid)}$$
$$f_{rotRow}(x, y, P, Q, r) = m_2 Q + n_2 \tag{5}$$
$$f_{rotCol}(x, y, P, Q, r) = n_2 P + m_2 \tag{6}$$

Intuitively, the rotate parameter has the following effect: a relatively large r that is a factor of P causes a small increase in row parallelism; a relatively small r that is a factor of P causes a large increase in row parallelism; r as a co-prime of P or $r = 1$ causes the maximum amount of row parallelism. In Figure 1c, a rotation of 2 is applied to the tall process grid, the result of $f_{rotRow}(x, y, 6, 4, 2)$.

The effects of applying rotation to a tall process grid are shown in Figure 7a. In Figure 6, performance degrades below 6 GFlops/core when a tall process grid is chosen (right side of the graph). However, Figure 7a shows that applying an adequate amount of rotation (obtaining sufficient row parallelism) increases the performance for tall process grids to over 6 GFlops/core.

Just as the dimensions of a process grid are architecture-dependent, the amount of rotation must also be tuned to the system. Excessive rotation may increase network traffic beyond its capability (the right-hand side of Figure 7a) because it increases the number of processors in each row broadcast. However, insufficient rotation may not expose enough concurrency for the triangular solves.

In Figure 7b, the x-axis is scaled from 7a by the square root of of the total number of cores. The optimal ratio is 1: when the amount of rotation increases the row parallelism until it is equal to that provided by a square process grid. This demonstrates that the performance that is lost due to a tall process grid is recovered by rotating.

This suggests a heuristic for tuning this parameter on new architectures: first, find the process grid aspect ratio that performs best on that architecture. Then, experiment with dimensions to find the appropriate tall process grid that minimizes the active panel factorization time (assuming that it is computation-bound). Using this tall process grid, compute the amount of rotation required to bring the amount of row parallelism back to the best-performing aspect ratio.

VI. Rotation and Striding Combined

The following formulæ describe the application of both rotation and striding to a process grid using the parameters r and s.

$$p = \left\lfloor \frac{y}{Q} \right\rfloor \qquad \text{(grid y index)}$$
$$m_4 = (x + pr) \bmod P \qquad \text{(x in grid)}$$
$$n_4 = y \bmod s \qquad \text{(y in grid)}$$
$$q = \left\lfloor \frac{y \bmod Q}{s} \right\rfloor \qquad \text{(subgrid y)}$$
$$f_{strideRot}(x, y, P, Q, r, s) = m_4 s + n_4 + Psq \tag{7}$$

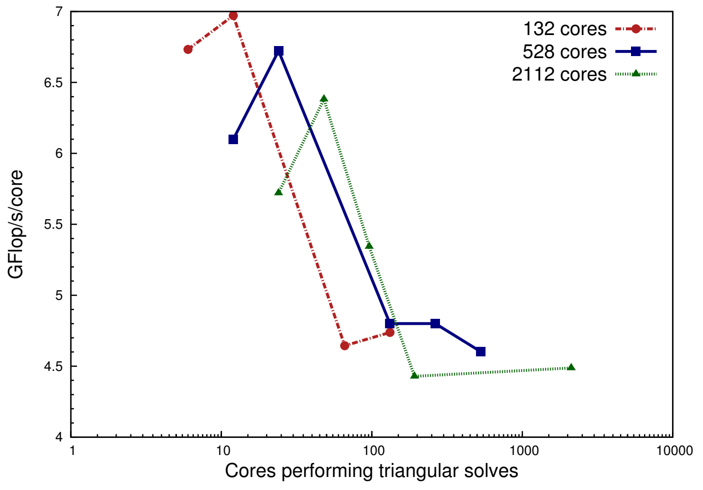
Figure 1d shows a tall process grid with a rotation of 2 and a stride of 2, the result of $f_{strideRot}(x, y, 6, 4, 2, 2)$.

Figure 8 demonstrates the effect of striding on a tall process grid that is rotated to the optimal configuration. Striding still has a substantial impact on performance, increasing performance by approximately 9% on 2112 cores compared to the standard row- or column-major configuration.

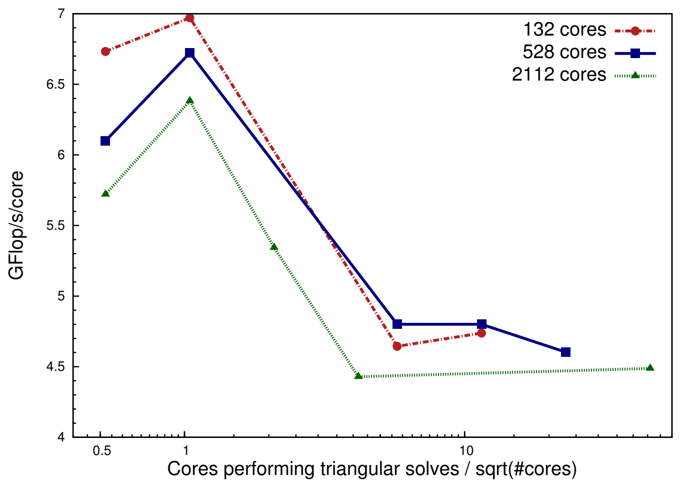
By varying aspect ratio, rotation, and striding, the degree of parallelism, locality, and memory contention in a process grid are explicitly considered. These considerations yield performance beyond that observed by using the originally optimal square process grid.

A. Data Movement Overhead

The largest overhead of using rotation and striding is the data redistribution that may be required if the matrix is provided in the traditional block-cyclic form. To measure this overhead, the data is migrated from the traditional distribution to a rotated and strided process grid. A breakdown of the



(a)



(b)

Fig. 7: The effect of adding rotation to a tall process grid to gain the benefits of a fast panel factorization, which is provided by the tall grid, and row parallelism which is recovered by rotating the process grid in the x direction. The same data is graphed in (b), but with the x-axis scaled by $\sqrt{\#\text{cores}}$ to illustrate that the ideal U parallelism matches the amount given by a square process grid. Results are from weak scaling on XT5.

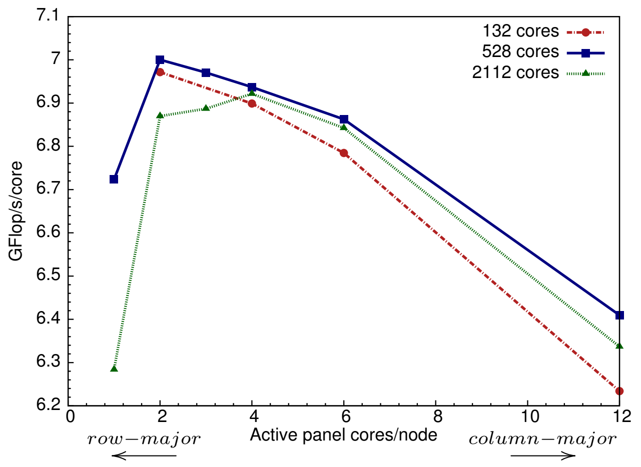


Fig. 8: Performance effects of striding the block-cyclic mapping with rotation and a tall process grid. 1 PE per node is effectively row-major (stride = process grid width), while 12 PEs per node is effectively column major (stride = 1). Results are from weak scaling on XT5.

Number of Processors	528	2112
Factorization Time (seconds)	653.3	1427.4
Data Movement Time (seconds)	12.4	14.1
Total Time (seconds)	665.7	1441.5
Data Movement Percentage of Total	1.9%	1.0%

TABLE II: Measurement of the data movement overhead of switching from a traditional block-cyclic distribution to a rotated and strided distribution for a tall process grid.

Arch.	Cores	N	b	P	Q	r	s	Peak
XT5	120	126K	504	24	5	6	5	67
XT5	132	132K	500	22	6	11	6	67.1
XT5	528	264K	500	44	12	22	6	67.4
XT5	1296	420K	500	72	18	24	3	66.2
XT5	2112	528K	500	88	24	44	3	67.4
XT5	8064	1048K	500	192	42	64	3	65.7
BG/P	256	96K	300	64	4	8	3	60.2
BG/P	1024	96K	300	128	8	8	3	45
BG/P	2048	96K	150	128	16	16	2	40.7
BG/P	4096	96K	150	256	16	16	2	31.6

TABLE III: Highest performing runs plotted on Figure 9.

times for the data movement phase and the factorization are presented in Table II. For 528 and 2112 processors, the data movement overheads on 140000×140000 and 280000×280000 matrices are 1.9% and 1%, respectively. On 2112 processors, striding improves performance by approximately 8% from the best square process grid. Rotation on 2112 processors improves performance by about 3% from the same comparison point. The combination of these techniques improves performance by 11%, corresponding to a 7% increase in percentage of peak achieved on Cray XT5.

VII. Peak Performance

A. *DGEMM Performance*

The peak performance obtained by an LU solver is bounded by the performance of the DGEMM implementation that it invokes. The performance of a DGEMM often varies with the size of the matrix on which it operates; a larger DGEMM generally executes more efficiently than a smaller one. The

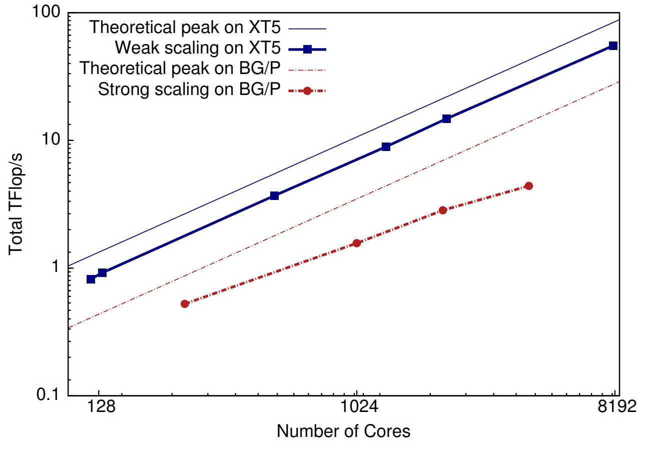


Fig. 9: Weak scaling (memory usage of matrix is constant around 75%) from 120 to 8064 processors on Jaguar, a Cray XT5 machine with 12 cores per node. Strong scaling ($n = 96,000$) from 256 to 4096 processors on Intrepid, an IBM BG/P machine with 4 cores per node.

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

Library	Peak	Cores	n	Arch.
UPC [16]	76.6	512	229K	XT3
DPLASMA [2]	58.3	3072	454K	XT5
ScalaPack [7]	59	3072	454K	XT5
HPCC [1] HPL	65.8	224220	3936K	XT5
Jaguar top500 [31]	75.5	224162	5474K	XT5
CharmLU	67.4	2112	528K	XT5

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

tradeoff between coarse grain sizes that aid in higher DGEMM efficiency and fine grain sizes that allow greater overlap of communication and computation is shown in Table IV.

Table V 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.

B. Scaling

To demonstrate the scalability of the dense LU solver described in this paper, it was weak scaled to 8064 processors on Cray XT5 using approximately 75% of memory. This represents about 530 blocks of 500×500 doubles for each processor. The solver obtains over 67% of peak on XT5 machines. Additionally, we also demonstrate the capabilities of the solver in the strong scaling regime up to 2048 processors on IBM Blue Gene/P, achieving over 50% parallel efficiency. Figure 9 and Table III show both sets of results.

VIII. Related Work

Many implementations of dense LU factorization use the traditional block-cyclic distribution. These include the widespread ScaLAPACK [7] library, the reference HPL benchmark implementation [26] and some variants [25], the UPC implementation by Husbands and Yelick [16], and the ARMCI implementation by Krishnan, Lewis, and Vishnu [21].

Implementations that dynamically schedule an explicit task graph have little reason to maintain a fixed mapping from block to processor. These include DPLASMA, by Bosilca, et. al. [2], based on the DAGuE framework [3] and SuperMatrix by Chan, et. al. [6]. Up to the implementation limits of those runtimes, any processor that receives the data of a block can perform subsequent steps involving that block.

Implementations of very different algorithms for LU factorization may go even further in varying block distributions. For instance, the recent communication-optimal LU implementation by Solomonik et al. [29] obtains its efficiency by replicating the matrix and using efficient collective operations to consolidate its results. As mentioned earlier, the distribution flexibility that the present implementation provides has been explored in an earlier paper [13] using a version of the code before pivoting was implemented.

IX. Conclusion

This paper describes two 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:

- locality among processors computing the active panel,
- memory contention between multiple active-panel processors on a node,
- computational parallelism in the active panel,
- and increased concurrency for the triangular solves along the row.

These concerns are closely related, and hence there are trade-offs between them.

Striding the process grid provides intermediary alternatives to a row- or column-major configuration. The grid can then be tuned to the architecture to ameliorate memory hierarchy contention. As nodes become wider on supercomputer architectures, such techniques are required to take advantage of node-locality without introducing contention. In the benchmarks considered, an 8% improvement in performance is demonstrated from this optimization.

Rotation enables a larger degree of parallelism in both process rows and process columns compared to the traditional block-cyclic distribution. For the problem scales considered, this provides a 3% improvement in performance, independent of striding. The two techniques compose to provide an overall 11% performance increase.

ACKNOWLEDGMENTS

This work was supported in part by the National Science Foundation (OCI-0725070, ITR-HECURA-0833188, NSF-0904844) and the Department of Energy (DE-SC0001845, DE-FG-08OR23332).

This research used Intrepid and Surveyor of the Argonne Leadership Computing Facility at Argonne National Laboratory, under the support of DOE contract DE-AC02-06CH11357. Runs on Kraken were done under the TeraGrid [5] allocation grant TG-ASC050040N supported by the NSF. 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.

REFERENCES

[1] B. Bland. HPC challenge class I award G-HPL winning submission, 2010.

[2] 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. Distributed dense numerical linear algebra algorithms on massively parallel architectures: DPLASMA. Technical Report UT-CS-10-660, University of Tennessee, September 2010.

[3] 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.

[4] Bull SAS. bullx Supernode Specification Sheet. http://www.bull.com/extreme-computing/download/S-BullxSuper-en4Web.pdf, 2011.

[5] 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.

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

[7] 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.

[8] Cray Inc. Cray XE6 Specifications. http://www.cray.com/Assets/PDF/products/xe/CrayXE6Brochure.pdf, 2010.

[9] X. Ding, K. Wang, and X. Zhang. ULCC: a user-level facility for optimizing shared cache performance on multicores. In *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*, PPoPP '11, pages 103–112, New York, NY, USA, 2011. ACM.

[10] J. Dongarra, M. Faverge, H. Ltaief, and P. Luszczek. Exploiting fine-grain parallelism in recursive LU factorization. In *Proceedings of ParCo 2011*, 2011.

[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. Dongarra, R. van de Geijn, and D. Walker. A look at scalable dense linear algebra libraries. In *Scalable High Performance Computing Conference, 1992. SHPCC-92. Proceedings.*, pages 372 –379, apr 1992.

[13] I. Dooley, C. Mei, J. Lifflander, and L. Kale. A study of memory-aware scheduling in message driven parallel programs. In *Proceedings of 17th Annual International Conference on High Performance Computing*, 2010.

[14] L. Grigori, J. W. Demmel, and H. Xiang. Communication avoiding Gaussian elimination. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, SC '08, pages 29:1–29:12, Piscataway, NJ, USA, 2008. IEEE Press.

[15] F. G. Gustavson. Recursion leads to automatic variable blocking for dense linear-algebra algorithms. *IBM Journal of Research and Development*, 41:737–755, November 1997.

[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] IBM. Blue Gene/P Specification Sheet. http://www-03.ibm.com/systems/resources/bgpspecsheet4.pdf, 2007.

[18] L. Kale, A. Arya, A. Bhatele, A. Gupta, N. Jain, P. Jetley, J. Lifflander, P. Miller, Y. Sun, R. Venkataraman, L. Wesolowski, and G. Zheng. Charm++ for productivity and performance: A submission to the 2011 HPC class II challenge. Technical Report 11-49, Parallel Programming Laboratory, November 2011.

[19] 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.

[20] 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.

[21] 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.

[22] B. C. Lee, R. W. Vuduc, J. W. Demmel, and K. A. Yelick. Performance models for evaluation and automatic tuning of symmetric sparse matrix-vector multiply. In *Proceedings of the 2004 International Conference on Parallel Processing*, ICPP '04, pages 169–176, Washington, DC, USA, 2004. IEEE Computer Society.

[23] J. Lifflander, P. Miller, R. Venkataraman, A. Arya, T. Jones, and L. Kale. Exploring partial synchrony in an asynchronous environment using dense lu. Technical Report 11-34, Parallel Programming Laboratory, August 2011.

[24] A. Mandal, R. Fowler, and A. Porterfield. Modeling memory concurrency for multi-socket multi-core systems. In *Performance Analysis of Systems & Software, 2010 IEEE International Symposium on*, ISPASS, pages 66–75, White Plains, NY, USA, 2010. IEEE.

[25] V. Marjanović, J. Labarta, E. Ayguadé, and M. Valero. Overlapping communication and computation by using a hybrid MPI/SMPSs approach. In *Proceedings of the 24th ACM International Conference on Supercomputing*, ICS '10, pages 5–16, New York, NY, USA, 2010. ACM.

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

[27] F. Petrini, D. Kerbyson, and S. Pakin. The Case of the Missing Supercomputer Performance: Achieving Optimal Performance on the 8,192 Processors of ASCI Q. In *ACM/IEEE SC2003*, Phoenix, Arizona, Nov. 10–16, 2003.

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

[29] E. Solomonik, A. Bhatele, and J. Demmel. Improving communication performance in dense linear algebra via topology aware collectives. In *International Conference for High Performance Computing, Networking, Storage and Analysis*, Supercomputing, 2011.

[30] M. Thomadakis. The architecture of the Nehalem processor and Nehalem-EP SMP platforms. Technical report, Texas A&M University, March 2011.

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

[32] J. S. Vetter, S. R. Alam, T. H. D. Jr., M. R. Fahey, P. C. Roth, and P. H. Worley. Early evaluation of the Cray XT3. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2006.

[33] R. C. Whaley, A. Petitet, and J. J. Dongarra. Automated empirical optimizations of software and the ATLAS project. *Parallel Computing*, 27(1-2):3 – 35, 2001.

[34] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel. Optimization of sparse matrix-vector multiplication on emerging multicore platforms. *Parallel Computing*, 35, March 2009.