RC24342 (W0709-011) September 5, 2007
Mathematics

IBM Research Report

Distributed SBP Cholesky Factorization Algorithms with Near-Optimal Scheduling

Fred Gustavson
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598
USA

Lars Karlsson, Bo Kågström
Department of Computing Science and HPC2N
Umeå University
SE-901 87
Sweden

Research Division
Almaden - Austin - Beijing - Haifa - India - T. J. Watson - Tokyo - Zurich

Distributed SBP Cholesky Factorization Algorithms with Near-Optimal Scheduling

FRED GUSTAVSON
IBM T.J. Watson Research Center and Umeå University
LARS KARLSSON and BO KÅGSTRÖM
Umeå University

The minimal block storage Distributed Square Block Packed (DSBP) format for distributed memory computing on symmetric and triangular matrices is presented. Three algorithm variants (Basic, Static, and Dynamic) of the blocked right-looking Cholesky factorization are designed for the DSBP format, implemented, and evaluated. On our target machine, all variants outperform standard full storage implementations while saving half the storage. Communication overhead is shown to be virtually eliminated by our Static and Dynamic variants which take advantage of hardware parallelism. The Static and Dynamic variants give nearly the same performance while clearly outperforming the Basic variant which in turn is comparable with standard implementations such as the one found in ScaLAPACK. Models of execution assuming zero communication costs are developed. The Static schedule is near-optimal on our target machine for medium to large problems based on comparisons with these models.

Categories and Subject Descriptors: F.2.1 [**Analysis of Algorithms and Problem Complexity**]: Numerical Algorithm and Problems—*Computation on matrices*; G1.3 [**Numerical Analysis**]: Numerical Linear Algebra—*Linear Systems (direct methods)*; G.4 [**Mathematical Software**]: Algorithm Design and Analysis, Reliability and robustness, Performance

General Terms: Parallel Computing, Parallel Algorithms

Additional Key Words and Phrases: Real symmetric matrices, positive definite matrices, Cholesky factorization, distributed square block format, packed storage

1. INTRODUCTION

Cholesky factorization is a special case of Gaussian elimination when the matrix is symmetric positive definite. Algorithms for Cholesky factorization can save roughly half of the floating point operations, use half of the memory, and since A is positive definite there is no need for pivoting [Golub and van Loan 1996]. In the literature there has been a great deal of interest in sparse parallel Cholesky algorithms but

Technical Report UMINF 07.19. Author's addresses: F. G. Gustavson, IBM T.J. Watson Research Center, Yorktown Heights, NY 10598, USA; email: `fg2@us.ibm.com`; L. Karlsson and B. Kågström, Department of Computing Science and HPC2N, Umeå University, SE-901 87, UMEÅ; email: `{larsk,bokg}@cs.umu.se`.
The research was conducted using the resources of the High Performance Computing Center North (HPC2N). Financial support was provided by the *Swedish Research Council* under grant VR 621-2001-3284 and by the *Swedish Foundation for Strategic Research* under grant A3 02:128.

here we only consider dense matrices.

A number of different parallel dense Cholesky factorization algorithms have been designed and implemented on a wide range of computer architectures. Three of the earliest ones are on use of systolic arrays [Brent and Luk 1982], data-flow [O'Leary and Stewart 1985] and distributed memory [Geist and Heath 1985]. Today, distributed memory computing (DMC) with message passing through the MPI interface is the de-facto standard for parallel large-scale computations. Scalable, portable routines for Cholesky factorization adapted to DM architectures can be found in ScaLAPACK [Choi et al. 1996] and PLAPACK [van de Geijn 1997] to name but two.

However, few attempts have been made at storing the symmetric matrix in a packed format. Previous approaches have often looked at packed routines as special and these have not been able to deliver the same level of performance as full storage routines. Here, we present the Distributed Square Block Packed (DSBP) format which generalizes both standard packed and full storage. Parallel algorithms using this format will work in both modes, resulting in performance for the packed storage being at least as good as for the full storage.

In [Gustavson et al. 2006b], we first examined the feasibility of the DSBP format which resulted in performance better than the ScaLAPACK full storage Cholesky factorization routine `PDPOTRF` (see [Choi et al. 1996]). In this contribution, three DSBP algorithm variants of parallel packed Cholesky factorization are developed. The first variant is a basic right-looking Cholesky factorization algorithm and it is comparable with `PDPOTRF` but operates on a matrix in DSBP format. The second variant takes advantage of hardware parallelism to overlap communication with computation via use of look-ahead and non-blocking communication primitives. The third, a dynamic approach, uses a more flexible scheduling and is important on hybrid systems with multi-core nodes. Our goal is to reach an almost optimal task schedule and overlap between communication and computation (see [Agarwal et al. 1994] for earlier such results for parallel matrix multiplication).

Earlier contributions on packed distributed storage of symmetric matrices include work by [D'Azevedo and Dongarra 1998] where a packed lower triangular matrix is represented as a collection of block columns, each using a standard storage format. Their main focus is to encapsulate the packed storage within the abstraction framework of the ScaLAPACK building blocks such as the PBLAS. Performance figures are presented which show respectable but slightly worse performance compared with full storage ScaLAPACK routines.

A different approach is taken in [Baboulin et al. 2005] where a secondary blocking level is introduced. An elementary block corresponds to a square block of a two-dimensional block-cyclic distribution and a grid block is a $P_r \times P_c$ block matrix of elementary blocks (i.e. each processor gets exactly one elementary block). Finally, a distributed block is a square matrix of grid blocks. A matrix is divided into distributed blocks and a blocked packed implementation with parallelization within the block operations is presented. This is a form of algorithmic blocking but on a macro level. Performance is for the most part slightly worse than for the full storage ScaLAPACK routine, possibly due to extra communication and library overhead. We remark that the storage requirement is larger than that of the DSBP format.

2. ORGANIZATION AND NOTATION

The rest of the paper is organized as follows. Section 3 describes the DSBP format for memory efficient storage and high-performance implementations in distributed memory. In Section 4, the three DSBP algorithm variants are described along with a brief discussion of the details of the communication algorithms. The MPI interface provides the possibility to overlap communication with computation, but to what extent is highly machine and software specific. In Section 5, we present an evaluation of the target machine's overlap capabilities. The performance of the DSBP algorithm variants are evaluated in Section 6 and their scalability characteristics, using different metrics, are examined in Section 7. Finally, we conclude with a summary of our major findings and outline future work in Section 8.

Processors are arranged in a logical $P_r \times P_c$ mesh with each processor having 2-dimensional coordinates (p, q) with $p \in \{0, \ldots, P_r - 1\}$ and $q \in \{0, \ldots, P_c - 1\}$. The matrix A being factored is of size $N \times N$. It is partioned into

$$N_b^2 = \left\lceil \frac{N}{n_b} \right\rceil^2$$

blocks of size $n_b \times n_b$ with padding of the possibly incomplete last block row and column. Due to symmetry only $N_b\,(N_b + 1)/2$ blocks need to be stored.

The following LAPACK/BLAS mnemonics are used in the text[1]:

—`POTRF`: computes a Cholesky factorization $A = LL^T$ of a symmetric positive definite matrix stored in lower triangular format and the factor overwrites the input.

—`TRSM`: solves one of the matrix equations

$$\mathrm{op}(A)X = \alpha B \quad \text{or} \quad X\,\mathrm{op}(A) = \alpha B,$$

where A is triangular and the solution X overwrites B.

—`GEMM`: computes one of the matrix multiply updates

$$C \leftarrow \beta C + \alpha\,\mathrm{op}(A)\,\mathrm{op}(B).$$

—`SYRK`: computes one of the symmetric rank-k updates

$$C \leftarrow \beta C + \alpha A A^T \quad \text{or} \quad \beta C + \alpha A^T A,$$

where C is symmetric and stored in triangular format.

3. DISTRIBUTED SQUARE BLOCK PACKED FORMAT

Globally the blocks are distributed with a square two-dimensional block-cyclic distribution. Locally the blocks are stored in Square Block Packed (SBP) format [Gustavson 2003]. The elements of the blocks are stored by column, but other data formats could be more efficient and the generally best solution is to adapt the data format to suit the characteristics of the kernels. For brevity we only discuss the storage of a lower triangular matrix or the lower triangular part of a symmetric matrix. The upper triangular case is the transposed form of the lower triangular case with the blocks stored *by row*.

[1]The α and β are scalars and $\mathrm{op}(X) = X$ or X^T.

The number of blocks of block column j stored on processor (p, q) with $q = j \bmod P_c$ is

$$\text{colsize}_{pq}(j) = \left\lfloor \frac{(N_b - 1) - p}{P_r} \right\rfloor - \left\lceil \frac{j - p}{P_r} \right\rceil + 1.$$

We assign to the $\text{colsize}_{pq}(q)$ blocks of block column q on processor (p, q) these block offsets:

$$0, \ldots, \text{colsize}_{pq}(q) - 1.$$

The second local block column (global block column $q + P_c$) has $\text{colsize}_{pq}(q + P_c)$ blocks and we assign the block offsets

$$\text{colsize}_{pq}(q), \ldots, \text{colsize}_{pq}(q) + \text{colsize}_{pq}(q + P_c) - 1.$$

A detailed example of this numbering scheme is illustrated in Figure 1 and conceptually it corresponds to label the local blocks consecutively from left to right and from top to bottom.

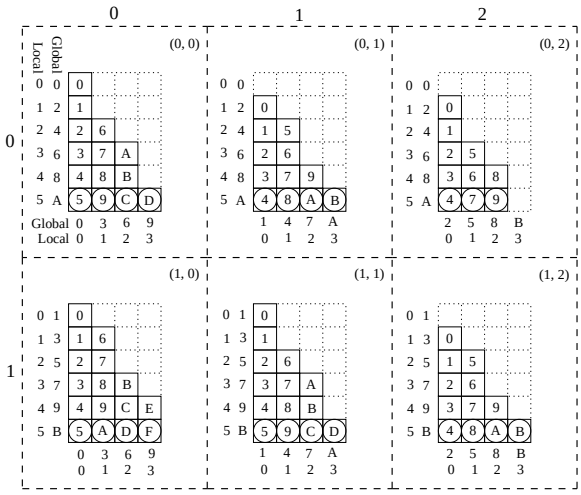

Fig. 1. Detailed example of a 12×12 block matrix distributed on a 2×3 mesh from the processors' viewpoint. Hexadecimal numbers indicate the local block offsets and circled block offsets are the elements of the auxiliary vector used in the addressing scheme. Dotted blocks emphasize the typical full storage requirements.

A block's address in local memory is determined by its block offset. We remark that for $P_r = P_c$ the $\text{colsize}_{pq}(\cdot)$ numbers are easy to calculate in closed form, but for $P_r \neq P_c$ these values get complicated because of modular arithmetic. Efficient use and calculation of a block offset is implemented via an auxiliary integer vector holding the block offsets of the *last* block in each column. On the $(1, 0)$ processor in Figure 1 this vector contains the offsets $5, 10, 14, 15$ (circled numbers in Figure 1). The last block row on processor (p, q) has (if it is non-empty) the local row coordinate

$$i_{\text{last}} = \left\lfloor \frac{(N_b - 1) - p}{P_r} \right\rfloor.$$

The block offset of a particular block with local coordinates (i, j) is calculated by

$$j_{\text{offset}} - (i_{\text{last}} - i),$$

where j_{offset} is the offset for column j (as stored in the auxiliary integer vector).

Note that the addressing scheme is based on the last block of a column and a negative offset depending on the local row coordinate. Thus, the addressing scheme is not dependent on whether the blocks are stored packed or are just blocks of a larger matrix in full storage. For full storage the auxiliary integer vector of the $(1, 0)$ processor in Figure 1 contains $5, 11, 17, 23$ and the same addressing scheme can be used.

The storage required by the DSBP format is roughly half that of full storage. In case there are incomplete blocks, one can pad the last block row and column with zeros and with ones on the diagonal. The storage requirement of the DSBP format in words is thus

$$\frac{N_b n_b (N_b n_b + n_b)}{2} = \frac{\left\lceil \frac{N}{n_b} \right\rceil n_b \left(\left\lceil \frac{N}{n_b} \right\rceil n_b + n_b \right)}{2}.$$

State-of-the-art kernels for routines such as `GEMM` in implementations of the BLAS usually reformat all or parts of their operands [Goto and van de Geijn 2007; Gustavson et al. 2006a]. Memory streams, vector registers, and other features typically require register blocking. In [Gustavson et al. 2006a] it is shown that the amount of data copying performed in dense matrix factorization is $\mathcal{O}\left(N^3\right)$ but could potentially be reduced to $\mathcal{O}\left(N^2\right)$ by using SBP with non-simple storage formats for the blocks together with kernel routines.

The four kernels (`POTRF`, `TRSM`, `GEMM`, and `SYRK`) used in our implementations take as input one (`POTRF`), two (`TRSM`, `SYRK`) or three (`GEMM`) blocks, all of which are contiguous on account of the DSBP format. Each operand will thus map into all levels of the memory hierarchy without conflict misses (assuming the capacity is sufficient to hold the operand).

The message passing library (in this case an implementation of MPI) must marshal (unmarshal) messages when sending (receiving), a process which is more expensive for non-contiguous messages. An $m \times n$ submatrix in column-major format generally consists of n contiguous vectors of length m, each separated by LDA $\geq m$ elements. On the other hand, our Cholesky variants send/receive single contiguous blocks.

4. DSBP ALGORITHM VARIANTS OF THE CHOLESKY FACTORIZATION

There are various ways to implement Cholesky factorization, such as left/right-looking and Crout's method. We have chosen the variant commonly called blocked right-looking Cholesky factorization [Dongarra et al. 1984]. Algorithm 1 describes this variant on a high level.

The DSBP format partitions the matrix into square blocks with padding for any incomplete blocks. To limit the communication overhead and simplify the code, we also use n_b as the algorithmic block size.

Algorithm 1 High-Level Right-Looking Cholesky

1: **while** N, order of $A \neq 0$ **do**
2: Block size $b = \min(n_b, N)$.
3: Partition $A = \begin{pmatrix} A_{11} & A_{21}^T \\ A_{21} & A_{22} \end{pmatrix}$ where $A_{11} \in \mathbb{R}^{b \times b}$.
4: Compute the Cholesky factorization $A_{11} = L_{11}L_{11}^T$ in-place.
5: Scale $A_{21} \leftarrow A_{21}L_{11}^{-T}$.
6: Update the trailing matrix $A_{22} \leftarrow A_{22} - A_{21}A_{21}^T$.
7: Continue with $A = A_{22}$.
8: **end while**

4.1 Abstraction of the Parallel Algorithms

A description of a DMC algorithm quickly becomes very technical and detailed. Our algorithms are implemented using the Single Program Multiple Data (SPMD) model, i.e., each processor is running the same program. Processors own pieces of the global matrix A in accordance with the DSBP format and the block cyclic layout. An SPMD implementation accesses local data and decisions on the actual operations to be performed are part of the node program logic.

However, we keep to a global view when presenting the parallel algorithms, which means that all operations, computations as well as communications, are expressed as global operations. Indeed, in practice such approaches already exist and we anticipate that they will be more common in future programming languages. By knowing the data layout and the required communication it would be possible to (semi-)automatically transform a global program view to a corresponding local SPMD program.

It is important to keep in mind that the global view is purely notational and that each processor only performs operations relevant to its local data and only loops over indices relevant to these operations.

Communication is indicated in the algorithms (*in italics*) and the details are presented in Section 4.7. Of course, communication operations are only performed on the relevant subset of processors.

4.2 Kernels

There are four computational kernels in each variant. In the presentation of the algorithm variants, we use a simplified notation based on parameterized routines. Each kernel routine is purely local with no communication involved. The details of these routines are given below.

—`factor(`s`)`: compute a Cholesky factorization of the (s, s) diagonal block,

$$A_{ss} = LL^T,\ A_{ss} \leftarrow L \qquad (\texttt{POTRF}).$$

—`scale(`s`, `i`)`: the (i, s) off-diagonal block is scaled with the contents of the R-buffer,

$$XR^T = A_{is},\ A_{is} \leftarrow X \qquad (\texttt{TRSM}).$$

—`updateDiagonal(`s`, `j`)`: the symmetric (j, j) diagonal block is updated via a symmetric rank-k update with the jth block of the west buffer,

$$A_{jj} \leftarrow A_{jj} - W_j W_j^T \qquad (\texttt{SYRK}).$$

—`updateInterior(`s`, `i`, `j`)`: updates the (i, j) interior block with the ith block of the west buffer and the transpose of the jth block of the south buffer,

$$A_{ij} \leftarrow A_{ij} - W_i S_j^T \qquad (\texttt{GEMM}).$$

4.3 Buffers

The buffers relate to the matrix A as follows:

—R holds factored A_{ss},

—W_i holds scaled A_{is}, and

—S_i holds scaled A_{is}.

Note that the Static and Dynamic variants described below use two sets of W- and S-buffers in order to work on two iterations simultaneously. One set each for even and odd iterations.

The three buffers are used for replicating data needed for local operations. The W_i block is replicated on processors $(i \bmod P_r, *)$ and the S_i block is replicated on processors $(*, i \bmod P_c)$. Since communication and computation are concurrent in both the Static and Dynamic variants we have chosen to separately describe the communication (see Section 4.7). The W- and S-buffers are called "west" and "south", respectively. This is logical considering that the W-buffer is replicated in the west to east direction and the S-buffer is replicated in the north to south direction.

4.4 Basic Variant

The high-level blocked right-looking Cholesky factorization is adapted to DSBP in Algorithm 2. The factored diagonal block is copied to the R-buffer for communication. Each `scale` kernel produces a block which corresponds to a block in the W- and S-buffers. These blocks are replicated following Algorithm 4 and used by the `update*` kernels. All `factor` and `scale` kernel operations in an iteration are collectively called a *panel factorization* since they operate on a block column (panel) of A. The `updateDiagonal` and `updateInterior` kernel operations in an iteration are called a *trailing matrix update* since they update the A_{22} trailing part of A.

4.5 Static Variant

The Basic variant (Algorithm 2) is divided into distinct communication and computation phases and is therefore unable to overlap communication with computation. A technique called look-ahead [Agarwal and Gustavson 1988; 1989; Dackland et al. 1992; Dackland et al. 1993; Strazdins 1998] reorders the loop body to perform panel factorization before all of the previous trailing matrix updates have completed. Algorithm 3 outlines this approach when it is applied to the DSBP format. Note that this algorithm needs two sets of W and S buffers to avoid artificial data dependencies.

Algorithm 2 Basic

```
1: for s = 0, N_b − 1 do
2:   factor( s ) and copy to R
3:   Start to replicate R
4:   Wait for R
5:   for i = s + 1, N_b − 1 do
6:     scale( s, i ) and copy to W_i
7:     Start to replicate W(i) and S(i)
8:   end for
9:   Wait for all W and S
10:  for j = s + 1, N_b − 1 do
11:    updateDiagonal( s, j )
12:    for i = j + 1, N_b − 1 do
13:      updateInterior( s, i, j )
14:    end for
15:  end for
16: end for
```

Updates are dependent on scaling which in turn depends on the factoring. Performing the `factor` and `scale` operations early allows initiation of the communication and consequently updates become available to other processors more quickly. That is why, in the Static variant, scaling is interleaved with the update of the previous iteration (see lines 19 and 20).

4.6 Dynamic Variant

A careful examination of Algorithm 3 or a Gantt-chart of its execution reveals two situations where a processor becomes idle although in principle it still has work to do.

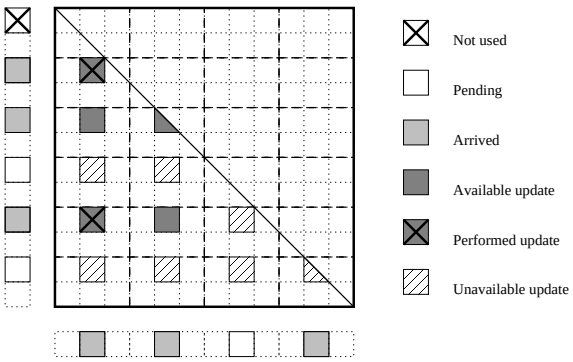

Fig. 2. Availability of updates is determined by the arrival of blocks in both the west and south buffers.

(1) *Updates become available in random order* (see Figure 2) due to messages arriving randomly. The static schedule enforces a strict order on the independent

Algorithm 3 Static

```
1: if N_b > 0 then
2:   factor( 0 )
3:   Start to replicate R
4:   Wait for R
5:   for i = 1, N_b − 1 do
6:     scale( 0, i )
7:     Start to replicate W_i and S_i
8:   end for
9: end if
10: for s = 1, N_b − 1 do
11:   Wait for W_s from iteration s − 1
12:   Wait for S_s from iteration s − 1
13:   updateDiagonal( s − 1, s )
14:   factor( s )
15:   Start to replicate R
16:   Wait for R
17:   for i = s + 1, N_b − 1 do
18:     Wait for W_i from iteration s − 1
19:     updateInterior( s − 1, i, s )
20:     scale( s, i )
21:     Start to replicate W_i and S_i
22:   end for
23:   for j = s + 1, N_b − 1 do
24:     Wait for S_j from iteration s − 1
25:     updateDiagonal( s − 1, j )
26:     for i = j + 1, N_b − 1 do
27:       updateInterior( s − 1, i, j )
28:     end for
29:   end for
30: end for
```

kernel operations that form a trailing matrix update.

(2) *Scaling is done early* (line 20) in order to maximize the overlap possibilities. However, data for the independent updates (lines 19 and 23–29) might be available before scaling can be performed (due to waiting for the arrival of R).

In both cases, the static schedule ensures that some processor idling will occur.

An example of the second type of idling obtained from an execution of the Static variant is illustrated in Figure 3. Note that the figure provides only *partial* timelines for *two* of the four processors and that time flows from left to right. The updates on processor $(0, 1)$ labeled A depend on operations prior to the ones labeled B. The second group of updates labeled B are the updates following the panel factorization partially performed in the first group labeled B. Scheduling any of the updates labeled A before B is allowed but since the idle time is dependent on several factors, including the iteration number, static scheduling to fill this gap is not practical.

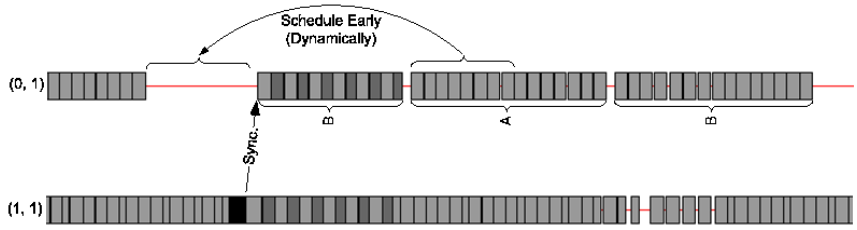

Fig. 3. A partial view of the timelines for the second processor column of a 2×2 mesh executing the Static variant on a 16×16 block matrix. Black boxes are `factor` invocations, dark gray boxes are `scale` invocations, and the light gray boxes are the `update*` invocations. The Static schedule introduces a gap on the $(0, 1)$ processor although any of the updates labeled A (rotated) could be scheduled.

A more flexible dynamic local scheduling would not suffer from these problems, but, as we will see in Section 6, it may have little impact on the overall running time. In order to investigate if removing these inefficiencies reduces overall execution time, we designed and implemented a scheduling mechanism which addresses these issues. Here, we give a brief description of the dynamic scheduling mechanism which we refer to as our Dynamic variant. At the start of each iteration a list is created with information about all the kernel operations (the tasks) that will execute in this iteration. The tasks are ordered in the list in exactly the same order as they would be executed by the Static variant. A pointer to the first not yet executed task is kept. The list is scanned sequentially for the first ready task, starting from the first not yet executed task. Figure 4 visualizes an example task list (acronyms used in the figure: Scale, Update Interior, Update Diagonal). One sees that the

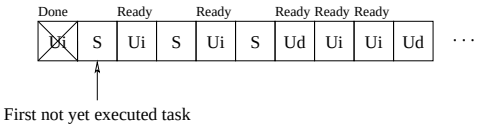

Fig. 4. Data structure for efficient dynamic scheduling.

next task to fetch would be the first ready `updateInterior` since the data for the `scale` task has not arrived.

4.7 Node Communication

Replication of W_i and S_i for iteration s (see Algorithms 2 and 3) is conceptually controlled by the global Algorithm 4. Blocks of W flow west to east in the mesh from the root processor (p_w, q_w). At the processor (p_w, q_s) the communication of a W-buffer block is additionally divided into a communication of the corresponding S-buffer block.

This implementation uses ring broadcasts to transfer the blocks of W and S. This is based on an effort to minimize idle time but other schemes are of course possible, e.g., a bi-directional ring broadcast or tree-based algorithms on switched networks. All communication operations are non-blocking and multiple invocations

Algorithm 4 Communicate W_i and S_i for iteration s

```
1: Let processor (p_w, q_w) be the processor that holds block A_is.
2: if I am on row p_w and need W_i then
3:   if I am processor (p_w, q_w) then
4:     W_i ← A_is.
5:   else
6:     receive( W_i, WEST ).
7:   end if
8:   send( W_i, EAST ) if needed.
9: end if
10: Let processor (p_w, q_s) be the processor that holds block A_ii.
11: if I am on column q_s and need S_i then
12:   if I am processor (p_w, q_s) then
13:     S_i ← W_i.
14:   else
15:     receive( S_i, NORTH ).
16:   end if
17:   send( S_i, SOUTH ) if needed.
18: end if
```

of this algorithm are concurrently running by keeping information about the state of progress for each invocation. When a request completes (tested by polling the MPI library) the corresponding algorithm is resumed. Ring broadcasts end early when a processor does not need a particular block (hence the condition "if needed" in Algorithm 4).

We remark that communication of the R-buffer proceeds in an analogous way and is therefore not discussed further.

4.8 Multi-Core Considerations

If the nodes of the distributed memory system are multi-core, an additional level of scheduling is required. The work mapped to a node must be partitioned onto the different cores and this secondary scheduling problem is referred to here as node-level scheduling. A common technique used to address the node-level scheduling problem is to structure the computation into BLAS operations. Parallel versions of these operations are then applied, resulting in what is typically called fork-join parallelism. This is how LAPACK extracts performance from SMP machines and one way in which ScaLAPACK can be used on hybrid systems (by linking the PBLAS to a parallel BLAS implementation).

For the variants presented here it is obviously not efficient to parallelize the kernels, especially if n_b is small. So a dynamic scheduling approach appears to be a good choice since it exposes parallelism and removes the artificial synchronizations inherent in an operation-abstraction such as the parallel BLAS.

5. COMPUTATION-COMMUNICATION OVERLAP EVALUATION

In what follows, we assume the reader is familiar with concepts such as (non-)blocking, send/receive requests, and other basic MPI terminology. For definitions

see the MPI standard documents ([MPI Forum 1995]).

On many systems there is different hardware for communication and computation, meaning the two can concurrently execute. The extent to which this parallelism can be exploited is highly dependent on the machine and system software. Therefore, we evaluate the overlap capabilities of our target machine in this section.

With MPI, overlap is primarily controlled by using *non-blocking primitives* for point-to-point communication since the interface does not define any non-blocking collective operation. One key issue when implementing message passing is where to put incoming messages. If a receive request has been posted the system has access to the final buffer space. On the other hand, if a receive request has *not* been posted by the time a message arrives, the MPI implementation has only two options:

(1) *Allocate a temporary buffer* into which the message is received, and copy from the temporary buffer to the receive buffer when the receive is posted. This technique is often called an *eager* protocol, and it is used to improve latency at the cost of reduced bandwidth due to the extra memory copy operations. It is typically used for small messages and/or only for a few messages at a time.

(2) *Force the sender and receiver to handshake when the receive is posted.* After this handshake transfer of data directly into the receive buffer can start, circumventing any need for a temporary buffer. It is often called a *rendezvous* protocol and it is used for long messages where high bandwidth is important.

The eager protocol allows for hardware parallelism but costs at least an extra memory copy. The rendezvous protocol, however, does not allow for overlap on the receiving side if the receive is blocking. We therefore conclude that the use of both non-blocking send and receive is critical as only then is overlap practically possible on both sides of the communication.

An important feature of an MPI implementation is *independent progress* [Brightwell and Underwood 2004], which is the capability of an MPI implementation to perform communication while the user process is not executing an MPI routine.

We designed two micro-benchmarks to investigate which protocol, eager or rendezvous, was used and when it was used and also to measure the amount of speedup that we could realistically achieve on our target machine.

A micro-benchmark to decide if and when the MPI implementation uses an eager protocol is graphically described in Figure 5 (left). The computation performed is a matrix multiply of two 100×100 matrices and this takes approximately $544\mu s$. The time for the processors to both compute and synchronize is theoretically, in the presence of an eager protocol, the maximum of the compute time and the communication time. This theoretical expectation is marked with a dashed graph in Figure 6 together with the measured times for both sender and receiver. It is evident that an eager protocol was used for messages up to approximately 32 KB, or a 64×64 double precision matrix. For larger messages the transfer is evidently postponed until the receive is posted, so a rendezvous-type protocol is used.

Our second micro-benchmark was designed to test for independent progress as well as to measure the effect on computational speed for both sender and receiver. This micro-benchmark is described in Figure 5 (right). If a rendezvous-type proto-

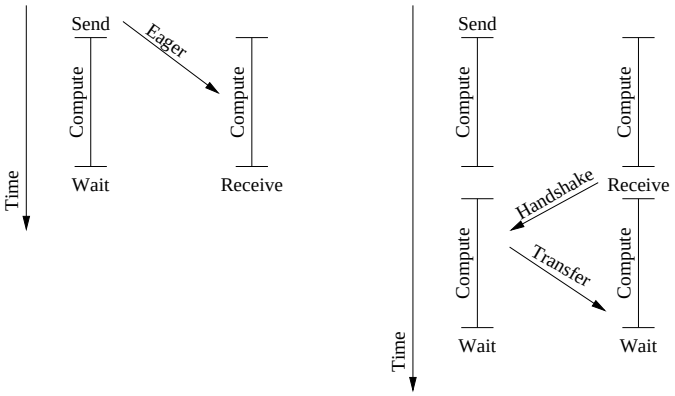

Fig. 5. Two micro-benchmarks to evaluate the MPI library overlap capabilities. Left: a test for the use of an eager protocol. Right: a test for independent progress.

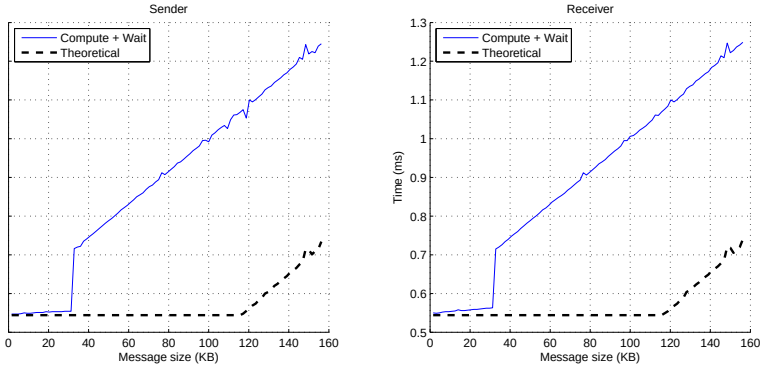

Fig. 6. Results for the eager micro-benchmark.

col is used the theoretical time for both sender and receiver should be

$$T_{\text{rendezvous}} = \max(2T_{\text{compute}}, T_{\text{compute}} + T_{\text{communicate}}).$$

In Figure 7, $T_{\text{rendezvous}}$ is marked with a dashed graph together with measured results. Figure 7 shows clearly that both sender and receiver are able to take advantage of this form of hardware parallelism and hence eliminate most of the communication overhead. This elimination is not perfect since the compute time has increased slightly. The data suggests that $90 - 95\%$ of the communication overhead is eliminated (for both sender and receiver) when the rendezvous protocol is used.

6. PERFORMANCE RESULTS

The performance and scalability of the three algorithm variants were evaluated on the Sarek cluster at the High-Performance Computing Center North (HPC2N) in Umeå, Sweden. The Sarek cluster has 192 nodes with dual AMD Opteron 248 (2.2 GHz) processors. The nodes are connected via a Myrinet 2000 high speed

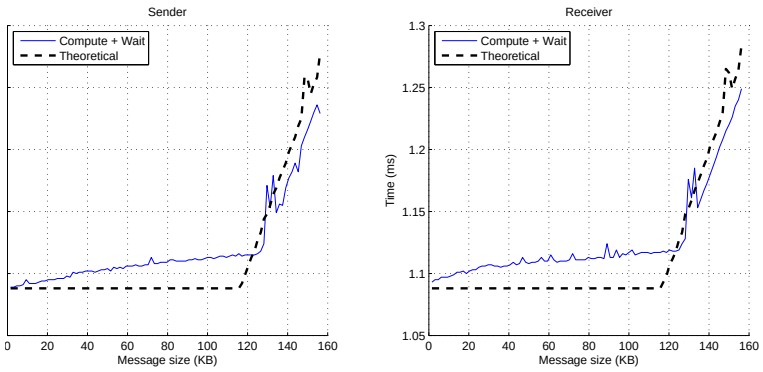

Fig. 7. Results for the independent progress micro-benchmark.

interconnect with the MPI library MPICH-MX version 1.2.5..12 capable of point-to-point communication at a speed of roughly 230 MB/s. Each node has 8 GB of memory and the BLAS library we used was GotoBLAS version r0.94. All the tests were performed with a distribution and algorithmic block size of $n_b = 100$ and only one processor per node was used. The Dynamic variant resulted in execution times that were very close to the Static variant. Therefore, the Dynamic variant's performance and scalability results are not presented.

For very large problems the `GEMM` operations dominate the communication and idle times introduced by panel factorization. Thus these two combined times can be safely neglected. However, as the ratio between node performance and inter-node bandwidth continues to increase, ever larger problems are needed for this effect to come into play. Since our DSBP algorithms are mainly right-looking variants they roughly have the same performance for large-scale problems. However, as we will see, the performance of our Basic and Static variants differ greatly for these very small to medium sized problems. Therefore, we consider such problems to highlight these differences.

Below, the performance of each kernel is examined. In addition, the communication system's characteristics are determined and discussed. The section ends by comparing our measured performance to our models of execution showing that the Static variant already is near to optimal for medium sized problems.

6.1 Kernel Performance

Since the operands to each of the four kernels are stored as contiguous blocks of a known size, the execution time of these kernel routines can be accurately approximated by a simple benchmark. There is little variability in the performance of a kernel routine except for where in the memory hierarchy the operands are at the time of the invocation.

To get a fair estimate of the performance of each kernel under a realistic scenario (e.g., the operands are not optimally placed in the memory hierarchy), the time spent in each kernel during the execution of the Basic variant was measured and averaged over the total number of invocations. Results of these tests are reported in Table I. Note the relatively poor performance of `POTRF`. The reason for this is

Kernel	flops	Time (μs)	Gflops/s
POTRF	$n_b^3/3$	192	1.74
TRSM	n_b^3	350	2.86
SYRK	$n_b^3 + n_b^2$	318	3.18
GEMM	$2n_b^3$	565	3.54

Table I. Kernel performance figures.

that LAPACK uses a level 2 kernel factorization routine POTF2. Furthermore, the early flattening of the performance is indicative of a mismatch between the blocksize and/or algorithm with the BLAS implementation.

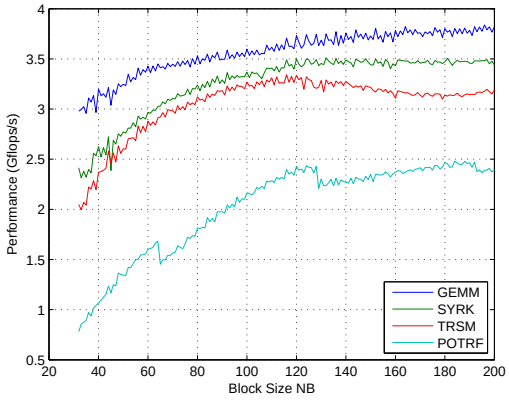

Fig. 8. Performance of kernels with respect to block size.

As of yet we have not optimized the kernels to take advantage of the memory layout. This would include using a full level 3 algorithm for POTRF. Instead we use standard LAPACK and BLAS routines and it is therefore important for us to investigate the performance of each of the four kernel routines. Figure 8 shows a detailed view of the performance for each kernel. It also justifies the choice of $n_b = 100$ for the other tests, because at this block size the performance of the major kernels is almost flat. Our results here show that the data format and algorithms are practically useful even without optimized kernels.

6.2 Communication Performance

We use the common communication model

$$t_s + t_w m$$

for an m-word message. The startup cost (t_s) and inverse bandwidth (t_w) were determined by fitting this model to a simple ping-pong benchmark (see Table II for results). Note that the latency is almost 1000 times higher than the inverse bandwidth. Even so, for the chosen block size of $n_b = 100$ the block transmission time will be approximately 10 times the latency.

t_s	29.6 μs
t_w	34.6 ns

Table II. The communication parameters for the Sarek cluster.

6.3 Modeled Versus Measured Performance

In this section, we use our models of execution of the Basic and Static variants to support our claim of the near-optimality of the Static schedule.

6.3.1 *Models of the Basic and Static Variants.* Inter-node data dependencies are handled via message passing and intra-node data dependencies by the ordering of the operations. The execution time is determined by the speed of the kernels, the communication overhead, and the inter-node dependencies. Since the speed of the kernels are usually optimized, the parallel algorithm should minimize the impact of the communication overhead and inter-node dependencies. Therefore, we model the performance of the Basic and Static variants by simulating them. The model assumes infinite bandwidth and zero latency to remove all communication overhead. Our model is relevant as we want to compare the implemented parallel algorithms with an ideal situation (and hence provide an upper bound on best possible performance).

6.3.2 *Comparisons.* In Figure 9, a comparison on 36 processors (6×6 mesh) is presented. The measured performance for the Static variant is very close to the

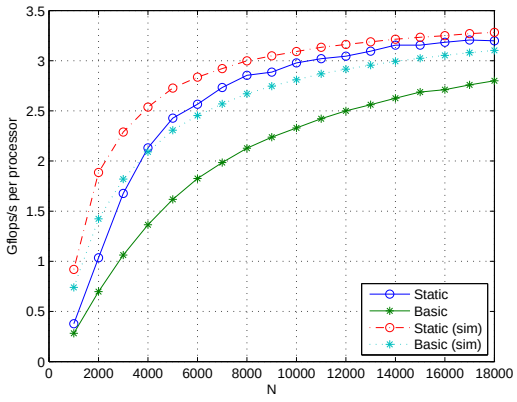

Fig. 9. Comparisons between the simulated performance and the measured performance for the Static and Basic variants on 36 processors arranged in a 6×6 mesh.

simulated performance for medium to large problems. However, the Basic variant is not close to its simulated performance showing that communication overhead is not reduced to the same extent as it is in the Static variant.

Let T denote the parallel execution time (assumed equal for p processors due to synchronization) which can be partitioned into four components for each processor

k ($1 \leq k \leq p$):

$$T = T^k_{\text{computation}} + T^k_{\text{communication}} + T^k_{\text{idle}} + T^k_{\text{overhead}}.$$

Typically, the components of T differ for different processors. All three variants (Basic, Static, and Dynamic) use the same kernels and the same distribution of computational work, controlled by the DSBP format and the block cyclic layout. They are instances of a bigger class of algorithm variants with the same kernels and distribution of work. We argue that within this class, the Static variant is near-optimal on our target machine and we conjecture that it will be near-optimal on other machines as well. Among all the variants of this class, the $T^k_{\text{computation}}$ times are constant (but different for different processors). Assuming no overhead (i.e., $T^k_{\text{communication}} = T^k_{\text{overhead}} = 0$) implies that the execution time for *any* variant in the class is bounded below by

$$\max_k T^k_{\text{computation}} = T - \min_k T^k_{\text{idle}}.$$

The distance to optimality of the Static schedule is therefore accurately estimated by measuring the minimum idle time, which can be done using our Static model.

In Table III, we illustrate the minimum simulated idle times on a 4×4 mesh. Similar negligible idle times for the Static model have been observed for other meshes as well. These negligible simulated idle times show that the Static schedule

	Basic		**Static**	
N	T	$\min_k T^k_{idle}$	T	$\min_k T^k_{idle}$
5000	1.0	0.1135	0.9	0.0012
10000	6.8	0.4409	6.4	0.0015
15000	21.9	0.9822	20.9	0.0012
20000	50.7	1.7380	48.9	0.0015
25000	97.5	2.7110	94.8	0.0012
30000	166.9	3.9000	163.0	0.0015

Table III. Simulated execution time and minimum simulated idle time (both in seconds) for the Basic and Static variants.

is near-optimal.

7. VARIOUS SCALABILITY MEASURES

For some applications the main limitation is the size of the problem that can be solved. For these applications a suitable characteristic is the *memory constrained* scalability. A problem size N_1 on a uni-processor is fixed and memory is assumed to scale linearly with the number of processors p. Since the memory required per processor is $\mathcal{O}\left(N_1^2\right)$ the memory constrained problem size on p processors is

$$N_p = \sqrt{p} N_1.$$

Ideally, the performance per processor should be constant while scaling. In Figure 10, the uni-processor size N_1 is fixed at 2500. The Static variant is close to constant and the Basic variant is close too, but scales slightly worse.

Another view of scalability is imposed by deadline-driven applications. Here, time-to-solution is a critical factor and the problem size should be increased to

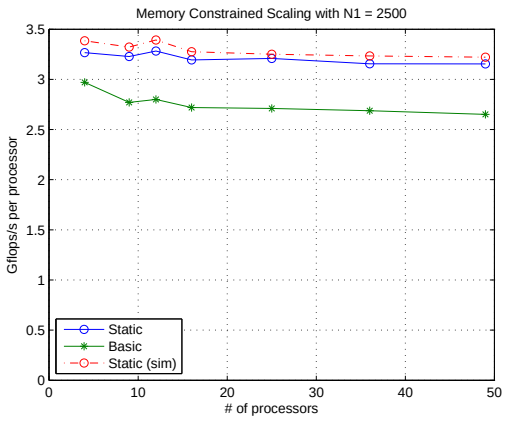

Fig. 10. Performance on problem sizes scaled with a memory constraint.

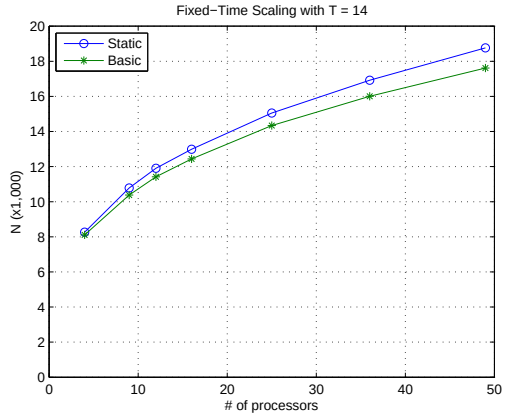

Fig. 11. Problem size solvable within a fixed amount of time.

consume the available time. This type of scalability is often referred to as *time-constrained* scalability. Figure 11 shows graphs of the largest possible problem size that can be solved within the fixed time of 14 seconds to give an idea of how the two algorithms compare. Because of the cubic nature of the arithmetic complexity of the Cholesky factorization, only a modestly larger problem can be solved within a given fixed amount of time. The relative improvement of Static over Basic at 49 processors is approximately 6% and tends to increase with increasing number of processors.

The performance point of view applies when a given problem must be solved as quickly as possible. It is also called "strong scalability" (as opposed to "weak scalability" where the problem size is allowed to grow). Results for this type of scalability are presented in Figure 12 with a fixed size of $N = 10000$. The performance per processor is used as a measure and, ideally, it should be close to constant. From

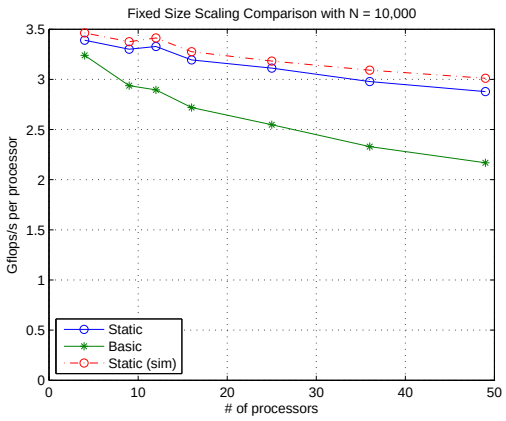

Fig. 12. Performance on various processors for a fixed problem size.

Figure 12 it appears as if there is a qualitative difference between the Static and Basic variants. The smaller slope of the graph for Static together with the absolute difference in performance indicates an advantage for the Static variant.

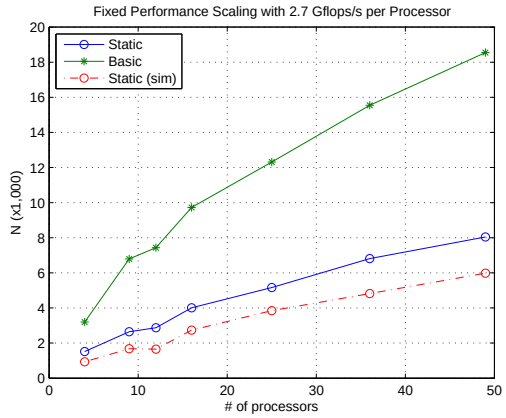

Fig. 13. Problem size required to achieve a fixed performance per processor.

While the three scalability metrics examined above focus on particular needs of different applications there is another view that focuses on the efficient use of a machine's resources. A good metric in this case is the performance per processor since it directly relates to the average efficiency of the floating point units. A machine centric view could ask for the required problem size on p processors to achieve a fixed performance per processor (FPU efficiency). In Figure 13, a performance of 2.7 Gflops/s is fixed, which corresponds to roughly 68% of the practical peak. A clear difference is seen between the two variants, with the Basic variant requiring consistently more than twice the problem size compared to the Static variant.

8. CONCLUSIONS AND FUTURE WORK

We presented the Distributed SBP format and showed that it is possible to achieve the same or better performance for packed storage Cholesky factorization compared to full storage. The Static variant had a $5-55\%$ higher performance than the Basic variant for matrices of size N between 5000 and 10000 on $4-49$ processors. In addition, the Static variant is significantly more scalable than the Basic algorithm for fixed problem sizes and is also much better to maintain a constant FPU efficiency as the number of processors increase.

Models of execution that assume no parallel overhead except processor idling support that the Static variant is close to the optimum schedule on the target DM machine. Based on these models we also conjecture that the Static variant is capable of nearly completely overlapping the communication with computation.

Since the Static and Dynamic variants give similar performance, the much simpler Static variant was sufficient and is recommended. A dynamic scheduling similar to the Dynamic variant would provide efficient scheduling on multi-core nodes.

There is frequent polling of the MPI layer in order for the communication algorithm to detect the completion of requests. This overhead would be avoided if the MPI interface and various MPI implementations support callbacks when a request completes.

The scheduling of tasks (kernel invocations) in the Basic variant is described by an outer control loop and some inner loops to traverse the blocks of the matrix. The Static variant is more complicated with a necessary preamble (see lines 1–9 of Algorithm 3) leading to the main control loop. The Dynamic variant is yet more complicated, with a dynamic rearrangement of the control loop body of the Static variant. The theoretically *optimal* schedule would probably need a rearrangement across several iterations of the control loop, basically forcing a complete loop unrolling without producing any substantial performance improvements.

Parallelism on multi-core nodes of a distributed memory machine have so far mainly been taken advantage of in one of the following two ways:

(1) Consider every core as individual nodes in a distribued memory machine. Communication can be optimized between two cores on the same physical node but is still expensive and data is often duplicated unnecessarily.

(2) Parallelize the basic building blocks, e.g., the BLAS, with lightweight threads or OpenMP. This SMP-BLAS approach is an example of fork-join parallelism and causes significant load unbalance issues and synchronization overhead.

We conjecture that a dynamic approach, similar to the one presented here, will prove to be superior to both of these two common paradigms since it overcomes their major drawbacks (unnecessary data duplications and spurious synchronizations). We will therefore pursue the development and evaluation of such dynamic approaches.

REFERENCES

Agarwal, R. C. and Gustavson, F. G. 1988. A parallel implementation of matrix multiplication and LU factorization on the IBM 3090. In *Aspects of Computation on Asynchronous and Parallel Processors*, M. Wright, Ed. IFIP, North-Holland, Amsterdam, 217–221.

Agarwal, R. C. and Gustavson, F. G. 1989. Vector and Parallel Algorithms for Cholesky Factorization on IBM 3090. In *Supercomputing '89: Proceedings of the 1989 ACM/IEEE conference on Supercomputing*. ACM Press, New York, NY, USA, 225–233.

Agarwal, R. C., Gustavson, F. G., and Zubair, M. 1994. A High Performance Matrix Multiplication Algorithm on a Distributed Memory Parallel Machine Using Overlapped Communication. *IBM Journal of Research and Development 38,* 6 (November), 673–681.

Baboulin, M., Giraud, L., Gratton, S., and Langou, J. 2005. A distributed packed storage for large parallel calculations. Tech. Rep. TR/PA/05/30, CERFACS, Toulouse, France.

Brent, R. P. and Luk, F. T. 1982. Computing the Cholesky Factorization Using a Systolic Architecture. Tech. Rep. TR 82–H521, Department of Computer Science, Cornell University, Upson Hall, Cornell University, Ithaca, New York 14853. September.

Brightwell, R. and Underwood, K. D. 2004. An analysis of the impact of MPI overlap and independent progress. In *ICS '04: Proceedings of the 18th annual international conference on Supercomputing*. ACM Press, New York, NY, USA, 298–305.

Choi, J., Dongarra, J. J., Ostrouchov, S., Petitet, A. P., Walker, D. W., and Whaley, R. C. 1996. Design and implementation of the ScaLAPACK LU, QR, and Cholesky factorization routines. *Scientific Programming 5,* 3 (Fall), 173–184.

Dackland, K., Elmroth, E., Kågström, B., and Van Loan, C. 1992. Parallel Block Factorizations on the Shared Memory Multiprocessor IBM 3090 VF/600J. *Int. J. Supercomputer Applications 6.1*, 69–97.

Dackland, K., Elmroth, E., and Kågström, B. 1993. A ring–oriented approach for block matrix factorizations on shared and distributed memory architectures. In *SIAM Conference on Parallel Processing for Scientific Computing*, R. S. et al, Ed. SIAM Publications, 330–338.

D'Azevedo, E. and Dongarra, J. 1998. Packed storage extension for ScaLAPACK. Tech. Rep. UT-CS-98-385.

Dongarra, J. J., Gustavson, F. G., and Karp, A. 1984. Implementing Linear Algebra Algorithms for Dense Matrices on a Vector Pipeline Machine. *SIAM Review 26,* 1 (January).

Geist, G. A. and Heath, M. T. 1985. Parallel Cholesky factorization on a hypercube multiprocessor. Tech. Rep. ORNL–6190, Oak Ridge National Lab., TN (USA). August.

Golub, G. H. and van Loan, C. F. 1996. *Matrix Computations*, third ed. Johns Hopkins University Press.

Goto, K. and van de Geijn, R. A. 2007. Anatomy of High-Performance Matrix Multiplication. Accepted for publication in ACM Transactions on Mathematical Software.

Gustavson, F. G. 2003. High-Performance Linear Algebra Algorithms Using new Generalized Data Structures for Matrices. *IBM Journal of Research and Development 47,* 1, 31–55.

Gustavson, F. G., Gunnels, J. A., and Sexton, J. C. 2006a. Minimal Data Copy for Dense Linear Algebra Factorization. In *PARA'06: State-of-the-Art in Scientific and Parallel Computing* . Lecture Notes in Computer Science, LNCS 4699. Springer. (To appear 2007).

Gustavson, F. G., Karlsson, L., and Kågström, B. 2006b. Three Algorithms for Cholesky Factorization on Distributed Memory Using Packed Storage. In *PARA'06: State-of-the-art in Scientific and Parallel Computing*. Lecture Notes in Computer Science, LNCS 4699. Springer. (To appear 2007). Also as IBM Technical Report RC24137.

MPI Forum 1995. MPI: A Message Passing Interface Standard. `http://www.mpi-forum.org/`.

O'Leary, D. P. and Stewart, G. W. 1985. Data-flow algorithms for parallel matrix computation. *Communications of the ACM 28*, 840–853.

Strazdins, P. 1998. A Comparison of Lookahead and Algorithmic Blocking Techniques for Parallel Matrix Factorization. Tech. Rep. TR-CS-98-07, Canberra 0200 ACT, Australia.

van de Geijn, R. A. 1997. *Using PLAPACK*. MIT Press.