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 almost half the storage. Communication overhead is shown to be virtually eliminated by the Static and Dynamic variants, both of which take advantage of hardware parallelism to hide communication costs. The Basic variant is shown to yield comparable or slightly better performance than full storage ScaLAPACK routine `PDPOTRF` while clearly outperformed by both Static and Dynamic. Models of execution assuming zero communication costs and overhead are developed. For medium and larger sized problems the Static schedule is near-optimal on our target machine based on comparisons with these models and measurements of synchronization overhead.

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 for symmetric positive definite matrices. 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

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.
Permission to make digital/hard copy of all or part of this material without fee for personal or classroom use provided that the copies are not made or distributed for profit or commercial advantage, the ACM copyright/server notice, the title of the publication, and its date appear, and notice is given that copying is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists requires prior specific permission and/or a fee.
© 20YY ACM 0098-3500/20YY/1200-0001 $5.00

here we only consider dense matrices. Although not as ubiquitous as large sparse Cholesky factorization, there are applications of large dense Cholesky factorization, for example when solving linear least squares problems with the method of normal equations [Baboulin et al. 2005a].

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]. In regard to scheduling, [Gerasoulis and Nelken 1989] considers parallel Cholesky factorization on MIMD architectures. 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 cases and 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 [Gustavson et al. 2007a]. This format allows a unification of packed and full storage routines into a single high performance implementation.

In [Gustavson et al. 2007b], we first examined the feasibility of the DSBP format. We demonstrated that performance better than the ScaLAPACK full storage Cholesky factorization routine `PDPOTRF` ([Choi et al. 1996]) is achievable for a packed storage routine. In this contribution, three DSBP algorithm variants of parallel packed Cholesky factorization are developed. The first variant (Basic) is a right-looking Cholesky factorization algorithm and it is comparable with `PDPOTRF` but operates on a matrix in DSBP format. The second variant (Static) takes advantage of hardware parallelism to overlap communication with computation via use of look-ahead and non-blocking communication primitives. The third variant (Dynamic) uses a more flexible scheduling on DM nodes and is important on hybrid systems with SMP or multi-core nodes. Our goal is to reach an optimal task schedule on the nodes and perfect overlap between communication and computation (see [Agarwal et al. 1994] for earlier such results for parallel matrix multiplication). We do this by a technique called algorithmic look-ahead [Agarwal and Gustavson 1988; 1989; Dackland et al. 1992; Dackland et al. 1993; Strazdins 1998] which reorders the outer loop to perform the next panel factorization before the current trailing matrix update is complete.

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 standard column major 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.

In [Baboulin et al. 2005b], a secondary blocking level is introduced. An *elemen-*

tary block corresponds to a distribution block of a two-dimensional square block-cyclic distribution. A *grid block* is a $P_r \times P_c$ block matrix of elementary blocks. A *distributed block* is a square matrix of elementary blocks. Each dimension of a distributed block consists of an integral multiple of $\mathrm{lcm}(P_r, P_c)$ elementary blocks. Thus, a distributed block is made up of a set of complete grid blocks and as a result the elementary blocks are perfectly distributed and all distributed blocks have the same distribution. The packed Cholesky implementation reuses ScaLAPACK and PBLAS on the level of distributed blocks. Performance is for the most part slightly worse than for the full storage ScaLAPACK routine, possibly due to extra communication, library overhead, and a reduced degree of concurrency. 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 DMC implementations. In Section 4, the three Cholesky algorithm variants are described along with a brief discussion of the details of the communication algorithms. The MPI interface provides the possibility to express overlap of communication with computation at the application level, but to what extent overlap is actually exploited is highly machine and software specific. In Section 5, we therefore present an evaluation of the target machine's overlap capabilities. This is crucial for interpreting the performance results given in Section 6. Scalability is 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 partitioned into

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

submatrices of order n_b with padding of the possibly incomplete last block row and column. Padding simplifies the DSBP addressing scheme; see Section 3 below.

The following LAPACK/BLAS names are used in the text:

—`POTRF`: computes the Cholesky factorization $A = LL^T$.

—`TRSM`: solves a triangular system of equations with multiple right-hand sides.

—`GEMM`: performs a matrix multiply and add update.

—`SYRK`: performs a symmetric rank-k update.

It is important to distinguish the different notions of blocking in DMC. A block cyclic layout (BCL) has a *distribution block size*, which, in our case, is square. A blocked algorithm usually has one, and sometimes more, *algorithmic block sizes*. For the combination of a BCL and a blocked algorithm it is common to use the same block size for both the distribution and the algorithm (typified by ScaLAPACK) and referred to as *distribution blocking*. However, a more general approach decouples the two block sizes and this is often called *algorithmic blocking*. This allows for better computational load balance since the distribution block size can be reduced. Processor interactions are often more frequent when using algorithmic blocking. For

example, the diagonal block factorization (see Section 4) is a local operation when using distribution blocking while it is a collective operation when using algorithmic blocking. We do not consider algorithmic blocking in conjunction with DSBP since they may have contradicting goals (improved load balance versus reduced data movement).

3. DISTRIBUTED SQUARE BLOCK PACKED FORMAT

In this section, we give a self-contained discussion on the Square Block Packed (SBP) and Distributed Square Block Packed (DSBP) storage formats. Much of the discussion on SBP have appeared in earlier publications but is summarized here for completeness.

The motivation for looking at other packed storage formats than the standard stacked column format used in BLAS and LAPACK is that there is no efficient way to use level-3 BLAS in combination with the latter format. High performance implementations of the BLAS at IBM and elsewhere have for a long time internally transformed the input into architecture-aware formats, for example the Square Block (SB) format. A block in SB format is stored contiguously and therefore it maps optimally into all levels of the memory hierarchy. The SBP format for packed storage is a convention on how to store a symmetric or triangular matrix as a set of contiguous square blocks.

Researchers at IBM have demonstrated that implementing Cholesky factorization on SBP input on a sequential architecture may not only be faster than standard packed Cholesky but also faster than standard full storage Cholesky. The difference between Cholesky on full storage and on SBP is claimed to be due to a better utilization of the memory hierarchy brought about by the contiguous block storage in SBP. Another benefit of SBP is that the Cholesky code can directly call BLAS kernels that do less data copying and internal transformations since the data is already in a suitable format. Further improvements can be made by storing the blocks themselves in some architecture-aware format, e.g., to better match memory streams or contiguous SIMD register loads. A block stored in a non-canonical format is referred to as a *non-simple block* [Gustavson et al. 2007a].

DSBP is a distributed memory generalization of SBP that maps the blocks to processors by a 2D block-cyclic distribution. Implementations using the DSBP format may also reduce memory traffic when sending and receiving messages by avoiding message packing.

3.1 Description of the DSBP Format

An $N_b \times N_b$ block matrix (with square blocks of order n_b) is distributed according to a two-dimensional BCL with the first block stored on processor $(0, 0)$. The last local block column index on processor column q is

$$j_{\text{last}}(q) = \left\lfloor \frac{(N_b - 1) - q}{P_c} \right\rfloor .$$

Similarly for the last local block row index on processor row p:

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

On each processor we construct an integer array cp (short for Column Pointer) with one component per local block column to speed up address calculation to a constant-time operation. The jth component of cp is defined by

$$\mathrm{cp}[j] = \sum_{k=0}^{j} \left(i_{\mathrm{last}}(p) - \left\lceil \frac{(q + kP_c) - p}{P_r} \right\rceil + 1 \right) - 1.$$

The expression in the sum calculates the number of local blocks on local block column k.

To each block we associate a block offset to index the block vector which stores the blocks. The block offset of local block (i, j) is

$$\mathrm{blockoffset}(i, j) = \mathrm{cp}[j] - (i_{\mathrm{last}}(p) - i).$$

Figure 1 shows a detailed example of a block matrix in DSBP format.

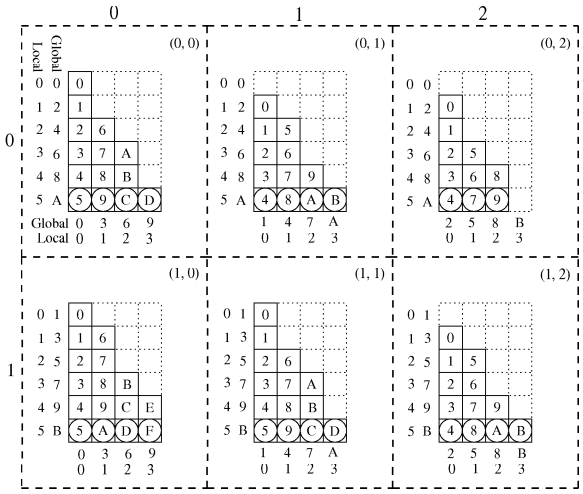

Fig. 1. Detailed example of a 12×12 block matrix distributed on a 2×3 mesh from the processors' viewpoint. Hexadecimal numbers indicate the local block offsets and circled block offsets correspond to the values in the column pointer array. Dotted blocks emphasize the typical full storage requirements.

Notice how the addressing scheme is based on the last block of a column and a negative offset depending on the local block row index. Thus, the addressing scheme is not dependent on whether the blocks are stored in a block packed or block full storage matrix. For full storage the column pointer array on all processors in Figure 1 contain $(5, 11, 17, 23)$ and the same addressing scheme and Cholesky algorithms can be used. In this case the column pointer array reduces to

$$\mathrm{cp}[j] = \sum_{k=0}^{j} (i_{\mathrm{last}}(p) + 1) - 1 = (j + 1) i_{\mathrm{last}}(p) + j.$$

There is a strong connection between the local storage format of DSBP and the Block Compressed Column Storage (BCCS) used for sparse blocked matrices. The

column pointer array plays a similar role as the column pointer array in BCCS (hence the same name). Because there is a regular pattern when the matrix is dense the row index array in BCCS does not need to be stored explicitly and the block offset is instead calculated directly from the column pointer array and the row index.

3.2 Properties of the DSBP Format

3.2.1 *Reduced Memory Requirements.* The storage required by the DSBP format is roughly half that of full storage. In case there are incomplete blocks, the last block row and column are padded. The storage requirement of the DSBP format in number of words is thus

$$\frac{N_b n_b^2 (N_b + 1)}{2} = \frac{\left\lceil \frac{N}{n_b} \right\rceil n_b^2 \left(\left\lceil \frac{N}{n_b} \right\rceil + 1 \right)}{2}.$$

3.2.2 *Less Data Movement in BLAS Operations.* Memory streams, vector registers, and other hardware features typically require register blocking. In order to effectively utilize such advanced hardware features, state-of-the-art kernels for BLAS routines such as `GEMM` usually reformat all or some of their operands [Goto and van de Geijn 2007; Gustavson et al. 2007a]. In [Gustavson et al. 2007a] 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 special kernel routines that operate on the non-simple blocks.

The four kernels in our Cholesky variants (`POTRF`, `TRSM`, `GEMM`, and `SYRK`) take as input one (`POTRF`), two (`TRSM`, `SYRK`) or three (`GEMM`) blocks, all of which are contiguous on account of the DSBP format. All operands will thus map into all levels of the memory hierarchy without conflict misses (assuming the cache capacity is sufficient to hold the operands and that the caches are at least three-way set associative). Combined with non-simple formats and special kernels this would provide an ideal situation for optimal kernel performance.

3.2.3 *Less Data Movement in Communication.* The message passing library (in this case an implementation of MPI) must pack and unpack messages when sending and receiving. This 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 and receive contiguous square blocks.

4. DSBP ALGORITHM VARIANTS OF THE CHOLESKY FACTORIZATION

There are various ways to implement Cholesky factorization, such as left/right-looking and a symmetric version of Crout's method. We have chosen the variant commonly called blocked right-looking Cholesky factorization [Dongarra et al. 1998]. Algorithm 1 describes this variant on a high level.

4.1 Buffers

We begin by describing the use of buffers in our parallel implementations. Because some data is remote we use explicit communication into local buffers. In the

Algorithm 1 High-Level Right-Looking Cholesky

1: **while** N, order of $A \neq 0$ **do**
2: Choose 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} has size $b \times b$.
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**

algorithms that we present there are three types of buffers:

—R (short for Reciprocal) contains a factored diagonal block used as input for scaling (`TRSM`) operations. Each R is generated by the pivot process and communicated to a separate replicated R residing on processor column.

—W (short for West) stores the scaled blocks of a panel used as input for updates (`SYRK` and `GEMM`). The West buffer is distributed along mesh columns and replicated along mesh rows.

—S (short for South) stores the block transpose of W (i.e., it is a row block vector and each block remains untransposed) used as input for `GEMM` updates. The South buffer is distributed along mesh rows and communicated along mesh columns to each processor.

In Basic, we only need one set of local buffers so $W(j)$ and $S(j)$ refer to these local West and South buffers, respectively. In Static and Dynamic, where two iterations are active at the same time, we use two sets of local buffers to reduce data dependencies and enhance performance. In these algorithms, $W(j)$ and $S(j)$ are instead to be read as the local West buffer $j \bmod 2$ and the local South buffer $j \bmod 2$, respectively. Subscripts i and k in $W_i(j-1)$ and $S_k(j-1)$, see Algorithm 3, refer to blocks i and k, respectively.

4.2 Basic Variant

The high-level blocked right-looking Cholesky factorization is adapted to DSBP in Algorithm 2. The distribution of computation follows the owner-computes rule, which states that the owner of an affected block is also the process that performs the computation. The details of the communication are left until Section 4.5 since they are common to all three variants. Also, communication is intermixed with the computation in both Static and Dynamic variants.

4.3 Static Variant

Basic (Algorithm 2) is divided into distinct communication and computation phases and is therefore unable to effectively overlap communication with computation. By algorithmic look-ahead we mean that a processor begins to factor a panel before all of its preceeding trailing matrix updates have completed on that processor. The number of extra iterations that can be active in this way on any processor is the

Algorithm 2 Basic

```
1: for j = 0, N_b − 1 do
2:    % Panel factorization
3:    Cholesky(A_jj), (POTRF)
4:    R ← A_jj
5:    Start to replicate R (Algorithm 5)
6:    Wait for R
7:    for i = j + 1, N_b − 1 do
8:       A_ij ← A_ij R^{−T}, (TRSM)
9:       W_i(j) ← A_ij
10:      Start to replicate W_i(j) and S_i(j) (Algorithm 4)
11:   end for
12:   % Trailing matrix update
13:   Wait for all W_*(j) and S_*(j)
14:   for k = j + 1, N_b − 1 do
15:      A_kk ← A_kk − W_k(j)W_k(j)^T, (SYRK)
16:      for i = k + 1, N_b − 1 do
17:         A_ik ← A_ik − W_i(j)S_k(j)^T, (GEMM)
18:      end for
19:   end for
20: end for
```

depth of the look-ahead. With this definition, Basic has look-ahead depth zero (no look-ahead), Static and Dynamic both have look-ahead depth one.

Static (Algorithm 3) is our look-ahead depth one variant with aggressively early scheduling of panel factorization suboperations. This scheduling is derived by following the critical path of the Cholesky factorization algorithm.

Figure 2 is a pictorial representation of parts of Algorithm 3. The left part illustrates a fused panel factorization (lines 14–28) consisting of a diagonal block update and factorization (bottom left) and a panel update and scaling (bottom right). The right part illustrates a trailing matrix update (lines 30–36) with diagonal block updates (bottom left) and full block updates (bottom right).

4.4 Dynamic Variant

A careful examination of Algorithm 3 using a Gantt-chart of its execution reveals two situations where a processor becomes idle although it still has work to do. We call these situations *spurious synchronizations* since they could have been avoided, at least in the short term, by scheduling another available operation. These are:

(1) *Updates become available in random order* (see Figure 3) due to the non-deterministic order in which messages arrive. The static schedule enforces a strict order on the independent kernel operations that form a trailing matrix update.

(2) *Scaling is scheduled early* (line 25) in order to maximize the overlap possibilities. However, the updates on lines 24 and 30–36 might be possible to partially perform prior to the scaling. The static schedule waits for R (line 21) before any of the updates are performed and thus enforces a strict order on the operations.

Algorithm 3 Static

```
1: % First panel factorization
2: Cholesky(A_00), (POTRF)
3: R ← A_00
4: Start to replicate R (Algorithm 5)
5: Wait for R
6: for i = 1, N_b − 1 do
7:    A_i0 ← A_i0 R^{−T}, (TRSM)
8:    W_i(0) ← A_i0
9:    Start to replicate W_i(0) and S_i(0) (Algorithm 4)
10: end for
11: % Loop over remaining panels
12: for j = 1, N_b − 1 do
13:    % Update and factor diagonal block
14:    Wait for W_j(j − 1)
15:    A_jj ← A_jj − W_j(j − 1)W_j(j − 1)^T, (SYRK)
16:    Cholesky(A_jj), (POTRF)
17:    R ← A_jj
18:    Start to replicate R (Algorithm 5)
19:    % Update and scale panel
20:    Wait for S_j(j − 1)
21:    Wait for R
22:    for i = j + 1, N_b − 1 do
23:       Wait for W_i(j − 1)
24:       A_ij ← A_ij − W_i(j − 1)S_j(j − 1)^T, (GEMM)
25:       A_ij ← A_ij R^{−T}, (TRSM)
26:       W_i(j) ← A_ij
27:       Start to replicate W_i(j) and S_i(j) (Algorithm 4)
28:    end for
29:    % Update non-panel part of trailing matrix
30:    for k = j + 1, N_b − 1 do
31:       Wait for S_k(j − 1)
32:       A_kk ← A_kk − W_k(j − 1)W_k(j − 1)^T, (SYRK)
33:       for i = k + 1, N_b − 1 do
34:          A_ik ← A_ik − W_i(j − 1)S_k(j − 1)^T, (GEMM)
35:       end for
36:    end for
37: end for
```

In both cases, the static schedule causes spurious synchronizations to occur and it should be clear that any static schedule would have similar problems.

The first type of synchronization is described in Figure 3. It depicts a 12×12 block matrix distributed on a 2×3 mesh with details for processor $(0, 1)$. To the left is a West buffer and below is a South buffer. Remote blocks and buffers have a dotted outline. For a block update to be available the corresponding West and South buffer blocks must both have received their data. The order in which buffer

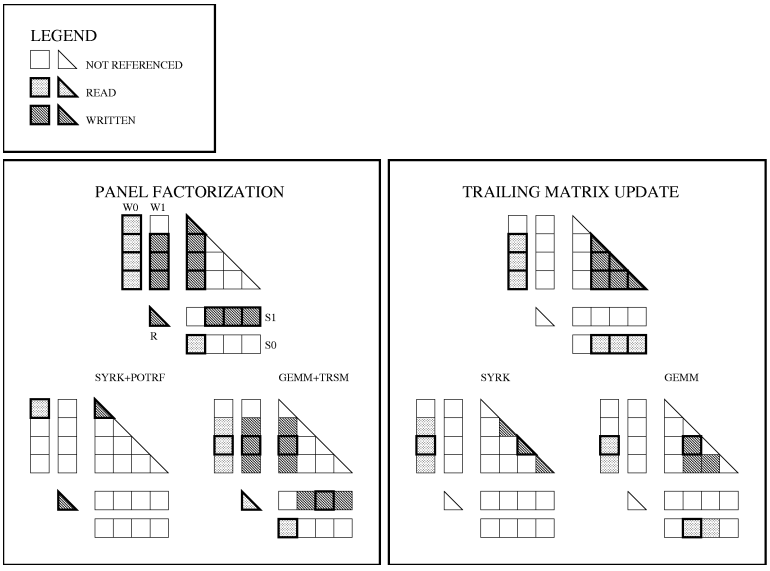

Fig. 2. The two tasks of fused panel factorization and trailing matrix update broken down into series of kernel operations with information on how the buffers are used.

blocks become available is random and therefore it is impossible to optimally match by any static schedule.

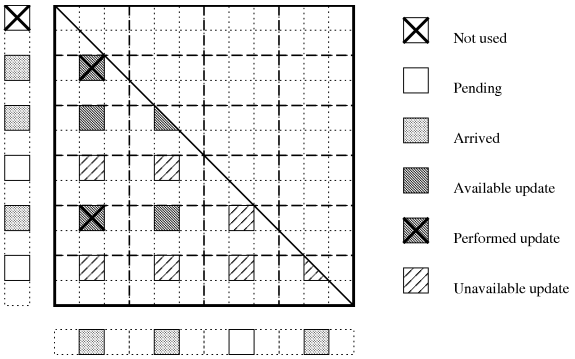

Fig. 3. Availability of updates is determined by the arrival of blocks in both West and South buffers.

An example of the second type of synchronization (obtained from an execution of Static) is illustrated in Figure 4. 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 only on operations prior to the ones labeled B and can therefore be executed before B. The second group of updates labeled B are the updates following the panel factorization partially performed in the first group labeled B. The gap before B could be filled by operations in A although the

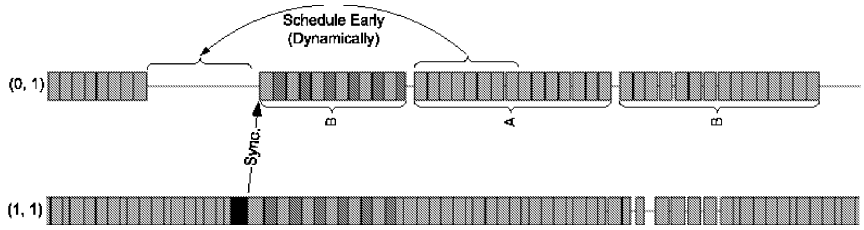

Fig. 4. A partial view of the timelines for the second processor column of a 2×2 mesh executing the Static variant on a 16×16 block matrix. Black boxes are diagonal block factorizations (`POTRF`), dark gray boxes are `TRSM` operations, and the light gray boxes are the `SYRK` and `GEMM` operations. The Static schedule introduces a gap on the $(0, 1)$ processor although any of the updates labeled A (rotated) could be scheduled there.

optimal number and selection of operations will depend on many factors, including the iteration number. It is impractical, if not impossible, to construct a static schedule which would fill such gaps.

A dynamic scheduling of tasks on the nodes would be more flexible and allow local avoidance of spurious synchronizations. In order to investigate if removing these spurious synchronizations reduces overall execution time, we designed and implemented a scheduling mechanism which addresses these issues. We give a brief description of the dynamic scheduling mechanism which we refer to as the 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 that 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 unexecuted task is kept. The list is scanned sequentially for the first ready task, starting from the first unexecuted task. Figure 5 visualizes an example task list (acronyms used in the figure: **T**RSM, **S**YRK, and **G**EMM). One sees that the next task to execute would be the first ready `GEMM` task

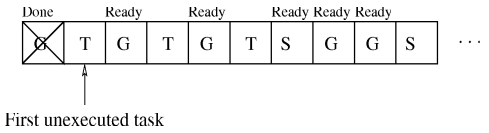

Fig. 5. Data structure for efficient dynamic scheduling.

since the data for the first `TRSM` task is unavailable.

4.5 Node Communication

Since all blocks are square they can be communicated as atomic units. Algorithm 4 shows how a single panel block (scaled block A_{ij}) is communicated to all its replicas in the remote West and South buffers. Communication starts at the root process (p_i, q_j) that owns the scaled panel block A_{ij}. The block is passed on from west to east until it reaches process (p_i, q_i) that owns the diagonal block A_{ii}. This process then splits the communication into a north to south transfer to fill the replicas of the South buffer. Both communications continue in parallel. At the end of this

Algorithm 4 Communicate $W_i(j)$ and $S_i(j)$ from iteration j

```
1: Let processor (p_i, q_j) be the processor that holds block A_ij
2: if I am on row p_i and need W_i(j) for some update then
3:    if I am processor (p_i, q_j) then
4:       W_i(j) ← A_ij
5:    else
6:       receive(W_i(j), WEST)
7:    end if
8:    send(W_i(j), EAST) if needed for some update on EAST neighbour
9: end if
10: Let processor (p_i, q_i) be the processor that holds block A_ii
11: if I am on column q_i and need S_i(j) for some update then
12:    if I am processor (p_i, q_i) then
13:       S_i(j) ← W_i(j)
14:    else
15:       receive(S_i(j), NORTH)
16:    end if
17:    send(S_i(j), SOUTH) if needed for some update on SOUTH neighbour
18: end if
```

procedure, processors $(p_i, *)$ have their copy of $W_i(j)$ and processors $(*, q_i)$ have their copy of $S_i(j)$.

All communication is performed using non-blocking MPI routines. The algorithm is executed for all the blocks of W and S concurrently and asynchronously. MPI polling is used to discover transfer completion and to resume the corresponding algorithm instance.

Algorithm 5 Communicate R from iteration j

```
1: Let processor (p_j, q_j) be the processor that holds block A_jj
2: if I am on column q_j and need R for some scaling operation then
3:    if I am processor (p_j, q_j) then
4:       R ← A_jj
5:    else
6:       receive(R, NORTH)
7:    end if
8:    send(R, SOUTH) if needed for some scaling operation on SOUTH neighbour
9: end if
```

Communication of the R buffer is similar (Algorithm 5). The root process (p_j, q_j) that owns the factored diagonal block A_{jj} starts a north to south transfer to build the replicas of R on processors $(*, q_j)$.

4.6 Considerations for Hybrid Systems

If the nodes of the distributed memory system are shared memory (e.g. SMP or multi-core), an additional level of scheduling is required if message passing within a node is to be avoided. We use the term *node-level scheduling* to refer to the

scheduling of the tasks of an MPI process onto the processor cores of a DM node. One technique to solve the node-level scheduling problem is to use multi-threaded BLAS on each node and map only one MPI process to each node. This is one way the node-level scheduling problem can be addressed in LAPACK and ScaLAPACK.

For the variants presented here it is probably not efficient to parallelize the kernels, especially if n_b is small. Dynamic scheduling of atomic kernel operations is one way to expose parallelism to many threads. This strategy has been successfully tested recently, by both the PLASMA and FLAME projects [Buttari et al. 2007; Chan et al. 2007] in pure shared memory environments.

Even though several threads are computing there need not be more than one thread that calls MPI routines. In fact, for the Cell BE it might be advisable to let the PPE handle all MPI calls and delegate all computations to the SPEs. Therefore, it is not necessary to have a thread-safe implementation of MPI in order to use algorithms and methods discussed here.

5. COMPUTATION-COMMUNICATION OVERLAP EVALUATION

In what follows, we assume the reader is familiar with concepts such as blocking/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 separate hardware for communication and computation that execute concurrently. The extent to which this parallelism can be exploited is highly dependent on the machine and system software. We therefore present an evaluation of the overlap capabilities of our target machine in this section.

With MPI, overlap is enabled by using *non-blocking primitives* for point-to-point communication (the MPI standard interface does not define any non-blocking collective operations). When the network controller detects an incoming message it must know where to store it. If a process has posted a receive the MPI library could instruct the network controller to place the message directly into the buffer supplied by the user. If the process has not yet posted a receive when an incoming message is detected then either the message transfer must be delayed or some temporary buffer must be allocated. These two options result in two different types of message transfer protocols:

(1) The *eager* protocol: 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. The eager protocol is used to improve latency of small messages at the cost of reduced bandwidth due to the extra memory copy operations.

(2) The *rendezvous* protocol: the sender and receiver handshakes to make sure a receive buffer is available. Transfer of data directly into the receive buffer can thus be guaranteed and this protocol is used for large, bandwidth limited messages.

The eager protocol allows for hardware parallelism but costs at least one extra memory copy. The rendezvous protocol allows overlap on the sending side but not on the receiving side unless the receive is non-blocking too. We therefore conclude that the use of 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 gives an MPI implementation the capability of performing communication while the user process is not executing an MPI routine. An MPI implementation that supports independent progress can actually overlap communication with computation and such an implementation is thus preferred.

5.1 Micro-Benchmarks

We designed two micro-benchmarks on our target machine to investigate which type of protocol is used in what situation and also to measure the amount of speedup that we can realistically achieve.

In both benchmarks we combine the execution of one or several `GEMM` updates with the transmission of a message. In all cases we used two processes on separate nodes and the computation of a `GEMM` update with the chosen parameters took approximately $T_{\text{compute}} = 544\mu s$. We denote the time to send a message of any particular size $T_{\text{communicate}}$.

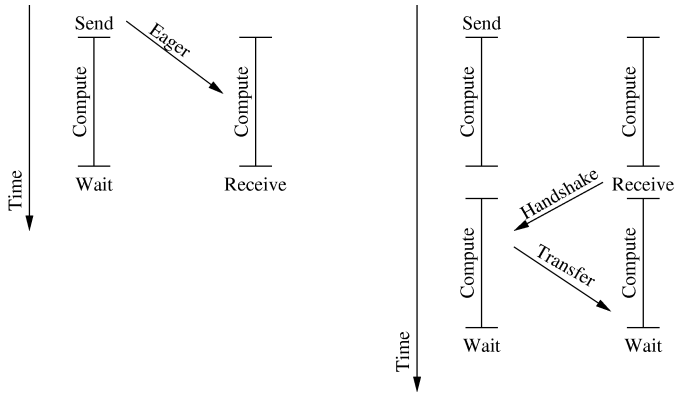

Fig. 6. Two micro-benchmarks to evaluate the MPI library overlap capabilities. Left: a test for the use of an eager protocol. Right: a test for independent progress.

5.1.1 *Benchmark A: Eager versus Rendezvous.* This benchmark is designed to identify which type of protocol is used depending on the message size and it is visually described in Figure 6 (left). Two processes, sender and receiver, each compute one `GEMM` update. The sender starts a non-blocking send prior to starting a `GEMM` computation and waits for it to complete after its `GEMM` computation completes. The receiver starts a blocking receive at the end of its `GEMM` computation. If an eager protocol is used, we would expect the time for both processes to be approximately

$$\max(T_{\text{compute}}, T_{\text{communicate}})$$

since the communication would be concurrent with the computation. This expected execution time is marked with a dashed curve in Figure 7. In the same figure, the solid curve marks the measured time. Clearly, up to about 32 KB the total time is just above the time to compute, whereas for larger messages it is closer to the sum of the compute and communicate times. We can be sure, based only on these

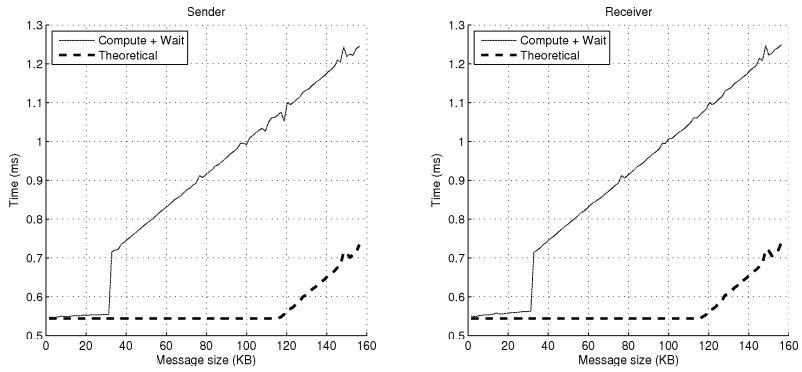

Fig. 7. Results for Benchmark A.

results, that for messages up to 32 KB the system uses an eager protocol. For larger messages a rendezvous protocol is probably used. However, it could also be explained by a lack of independent progress.

5.1.2 *Benchmark B: Independent Progress and Speedup.* This benchmark is designed to discover if the MPI implementation supports independent progress and also allows us to assess the practically achievable speedup of overlapping communication with computation (see Figure 6 (right) for description). Both processes now compute a `GEMM` twice and the receiver starts the receive between the two computations and waits for it to complete after the second computation. If a rendezvous type protocol is used, then the time for both processes in the presence of independent progress is expected to be

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

This is marked with a dashed curve in Figure 8. As with the previous benchmark,

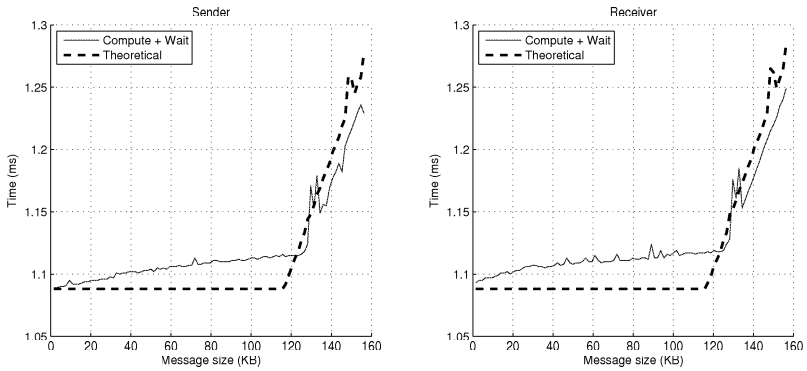

Fig. 8. Results for Benchmark B.

the measured times are marked with a solid curve. There are no major deviations

from the expected times and the difference on the expected flat part indicates that overlap slightly affects computation time. This indicates that $90 - 95\%$ of the communication overhead is eliminated on both sides of the communication.

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 with a Myrinet 2000 high speed interconnect with the MPI library MPICH-MX version 1.2.7 capable of point-to-point communication with roughly 230 MB/s bandwidth. Each node has 8 GB of memory and the BLAS library we used was GotoBLAS version r1.12. All the tests were performed with a distribution and algorithmic block size of $n_b = 100$. The Dynamic variant executed nearly identically as fast as the Static variant. For this reason, we omit reporting on the performance and scalability of the Dynamic variant.

For large problems the time spent in `GEMM` operations completely dominate the time spent in all other operations as well as the idle time and communication overhead. Therefore, if two algorithms have different `GEMM` performance the one with the highest performance will eventually outperform the other. Hence, we focus our attention on small and medium sized problems to highlight the differences amongst our DSBP algorithms and their relation to `PDPOTRF` in ScaLAPACK. However, large scale problems are also tested to make sure performance does not degrade.

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 has a near-optimal scheduling of tasks on the nodes for medium and larger 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 Basic was measured and averaged over the total number of their invocations. The results of these tests are reported in Table I. Note the relatively poor performance of `POTRF`. The reason for this is 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.

We have not optimized the kernels to take advantage of non-simple block storage formats. In all measurements we used LAPACK and BLAS routines. It is therefore important for us to investigate the performance of each of the these four employed routines (see Figure 9). It also justifies the choice of $n_b = 100$ for the other tests,

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 ($n_b = 100$).

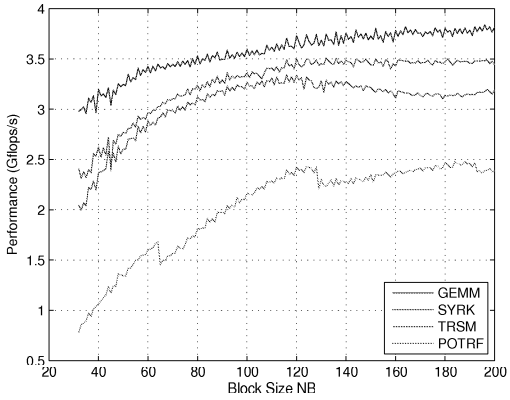

Fig. 9. Performance of kernels with respect to block size.

because at this block size the performance of the `GEMM` routine is almost flat.

6.2 Communication Performance

An important characteristic of a distributed memory machine is its message passing performance. In our case, it is the MPI implementation MPICH-MX that we benchmark. We use the communication model

$$t_s + t_w m$$

for an m-word message and estimate the parameters t_s and t_w by experimentation. The startup cost (t_s) and inverse bandwidth (t_w) were determined by fitting this model to a ping-pong benchmark (see Table II for results). Note that the latency

t_s	29.6 μs
t_w	34.6 ns

Table II. The communication parameters for the Sarek cluster.

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.

6.3 Absolute Performance

We tested Basic and Static as well as the ScaLAPACK full storage `PDPOTRF` routine on various mesh and problem sizes. The largest test was on a 12×12 mesh and a

matrix of order 110000. In Figure 10, we report the performance per processor on the 12×12 mesh.

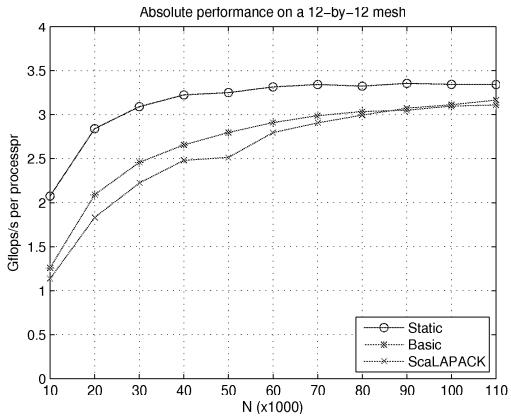

Fig. 10. Absolute performance per processor of Basic, Static, and `PDPOTRF` on 12×12 mesh for various problem sizes.

6.4 Modeled Versus Measured Performance

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

6.4.1 *Models of Basic and Static.* 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 kernels are assumed to be optimized, a parallel algorithm should minimize the impact of the communication and other overheads. Therefore, we model the performance of Basic and Static by simulations to see if our algorithms meet this expectation. We do not model communication overhead or other overheads besides computation since we wish to compare our performance with an ideal machine where overhead is not an issue. This is one way to quantify the otherwise so elusive overhead component in parallel algorithms.

6.4.2 *Comparisons.* In Figure 11, a comparison on 36 processors (6×6 mesh) is presented. The measured performance for Static is very close to the simulated performance for medium to large problems. However, Basic is not close to its simulated performance showing that communication overhead is a significantly limiting factor for this less efficient algorithm.

Let T denote the parallel execution time (assumed equal for all $P_r \cdot P_c$ processors due to synchronization). The execution time can be partitioned into four components for each processor k ($0 \leq k < P_r \cdot P_c$):

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

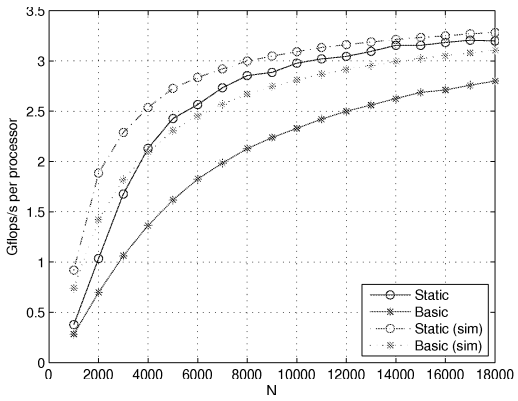

Fig. 11. Comparisons between the simulated performance and the measured performance for Basic and Static on 36 processors arranged in a 6×6 mesh.

Typically, the components of T differ for separate processors (hence the superscript k). All three variants (Basic, Static, and Dynamic) use the same kernels and the same distribution of computational work. They are instances of a bigger class of algorithm variants with the same kernels and distribution of work. If $T^k_{\text{communication}} = T^k_{\text{overhead}} = 0$, a lower bound for the execution time is

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

This quantity can be accurately estimated by the Static model since these conditions hold by design.

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	$\min_k T^k_{\text{idle}}$	T	$\min_k T^k_{\text{idle}}$
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 in theory. We now discuss our empirical evidence.

6.4.3 *Empirical Evidence of Negligible Idle Times.* Simulations of execution strongly indicated that with no communication or other overhead the minimum idle time observed over all processors is close to zero. The comparisons presented in Section 6.4.2 predict that this should be observable in practice. The code was

instrumented to accumulate the time spent in synchronizing MPI calls. Table IV shows that at least one processor has a small idle time component, i.e., it is active almost all the time. A further reduction in parallel execution time would either require faster kernel execution, moving work between nodes, or reducing the number of nodes and re-balancing the data layout. Such efficiencies are beyond the scope of this paper.

$P_r \times P_c$	4×4		8×8		12×12	
N	T	$\min T_{\text{wait}}$	T	$\min T_{\text{wait}}$	T	$\min T_{\text{wait}}$
10000	6.554	0.076	1.956	0.116	1.116	0.195
20000	49.719	0.289	13.354	0.092	6.520	0.155
30000	164.880	0.343	43.053	0.107	20.221	0.161
40000	392.612	0.923	100.500	0.109	45.948	0.160
50000	763.795	0.276	194.252	0.311	89.033	0.172
60000	1303.342	0.142	334.360	0.130	150.797	0.211
70000			521.126	0.138	237.592	0.266
80000			783.428	0.147	356.522	1.080
90000			1123.918	0.119	503.045	1.117
100000					692.181	0.400
110000					921.571	0.199

Table IV. Measured time in synchronizing code (e.g., MPI wait routines). The columns labeled $\min T_{\text{wait}}$ contain the smallest measured waiting time over all the processors.

7. SCALABILITY

Fixed size scaling (strong scalability) examines how performance degrades when more processors are used to solve a problem of fixed size. The performance per processor should ideally remain constant but in practice it will decrease as a consequence of increased overhead. Based on Figure 12 we conclude that Static is more

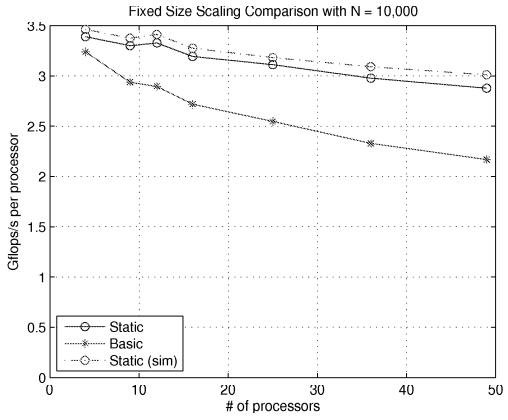

Fig. 12. Performance on various mesh sizes for a fixed problem size.

scalable than Basic. The advantage of overlapping iterations is more apparent when a large number of processors solve a small problem.

Under memory constrained scaling the amount of available memory is assumed to scale linearly with the number of processors and the problem size is scaled so that the consumed memory is kept constant per processor. An algorithm has a good memory constrained scalability if it can maintain a constant performance per processor. For Cholesky factorization, which operates on $\mathcal{O}\left(N^2\right)$ memory, the

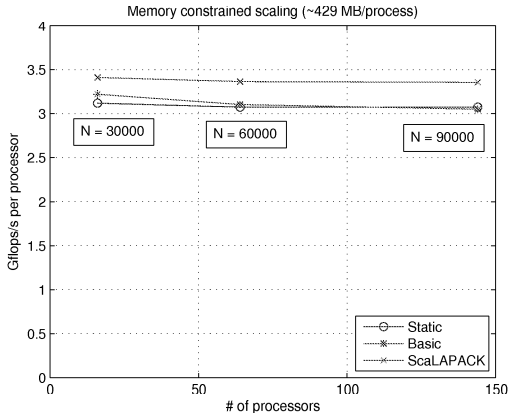

Fig. 13. Memory constrained scaling (with full storage requiring $\approx$ 429 MB per process).

problem size N must be scaled with a factor $\sqrt{p_2/p_1}$ when the number of processors scales from p_1 to p_2 (see Figure 13). The memory constrained scalability of all tested algorithms was excellent.

The isoefficiency function [Grama et al. 1993] maps the number of processors to problem size. It tells how much the problem size must be scaled up to maintain a constant efficiency. In Figure 14, we show curves which are similar to the isoefficiency curves but instead of comparing parallel execution time with the best serial implementation we measure efficiency as performance per processor. With this metric, we conclude that Static is far more scalable than Basic.

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 has 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 our target DM machine. Based on these models we also conjecture that the Static variant is capable of nearly completely overlapping the communication with computation. Important

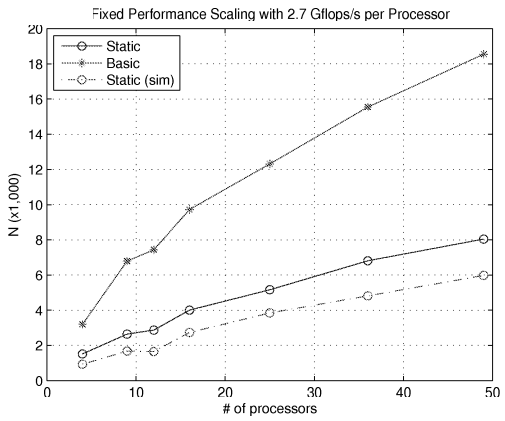

Fig. 14. Problem size required to achieve a fixed performance per processor of 2.7 Gflops/s.

characteristics of our target machine include its ability to reach near peak performance on small Level 3 BLAS operations and a low overhead MPI implementation capable of asynchronous communication with independent progress. The presented algorithms should perform well on other machines with similar characteristics.

Since the Static and Dynamic variants give similar performance, the simpler Static variant is sufficient. A dynamic scheduling similar to the Dynamic variant could provide efficient scheduling on hybrid systems with SMP or 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 supported callbacks when a request completes.

The scheduling of tasks in the Basic variant is described by an outer control loop and inner loops to traverse the blocks of the matrix. The Static variant is more complicated with a necessary preamble (see lines 1–10 of Algorithm 3) before the main control loop. The Dynamic variant is yet more complicated, with a dynamic rearrangement of the control loop body of the Static variant. An optimal schedule is likely to require a rearrangement across several iterations of the control loop. It is difficult to imagine there is any practical way of coding such an optimal schedule without using dynamic scheduling.

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., and Gratton, S. 2005a. A Parallel Distributed Solver for Large

Dense Symmetric Systems: Applications to Geodesy and Electromagnetism Problems. *Internation Journal of High Performance Computing Applications 19*, 353–363.

Baboulin, M., Giraud, L., Gratton, S., and Langou, J. 2005b. 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.

Buttari, A., Langou, J., Kurzak, J., and Dongarra, J. 2007. A Class of Parallel Tiled Linear Algebra Algorithms for Multicore Architectures. Tech. Rep. UT-CS-07-600.

Chan, E., Quintana-Ortí, E. S., Quintana-Ortí, G., and van de Geijn, R. 2007. Super-Matrix Out-of-Order Scheduling of Matrix Operations for SMP and Multi-Core Architectures. In *SPAA '07: Proceedings of the Nineteenth ACM Symposium on Parallelism in Algorithms and Architectures*. San Diego, CA, USA, 116–125.

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., Duff, I. S., Sorenson, D. C., and van der Vorst, H. A. 1998. *Numerical Linear Algebra on High-Performance Computers*. SIAM.

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.

Gerasoulis, A. and Nelken, I. 1989. Scheduling Linear Algebra Parallel Algorithms on MIMD Architectures. In *Parallel Processing for Scientific Computing*. 68–95.

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.

Grama, A. Y., Gupta, A., and Kumar, V. 1993. Isoefficiency: measuring the scalability of parallel algorithms and architectures. *IEEE Parallel and Distributed Technology: Systems and Applications 1,* 3, 12–21.

Gustavson, F. G., Gunnels, J. A., and Sexton, J. C. 2007a. Minimal Data Copy for Dense Linear Algebra Factorization. In *PARA 2006: State of the Art in Scientific and Parallel Computing*, B. Kågström et al., Eds. Lecture Notes in Computer Science, LNCS 4699. Springer, 540–549.

Gustavson, F. G., Karlsson, L., and Kågström, B. 2007b. Three Algorithms for Cholesky Factorization on Distributed Memory Using Packed Storage. In *PARA 2006: State of the Art in Scientific and Parallel Computing*, B. Kågström et al., Eds. Lecture Notes in Computer Science, LNCS 4699. Springer, 550–559. 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.