Three Algorithms for Cholesky Factorization on Distributed Memory Using Packed Storage

Fred G. Gustavson[1,2], Lars Karlsson[2], and Bo Kågström[2]

[1] IBM T. J. Watson Research Center,
Yorktown Heights, NY 10598, USA
fg2@us.ibm.com
[2] Department of Computing Science and HPC2N, Umeå University,
SE-901 87 Umeå, Sweden
{larsk, bokg}@cs.umu.se

Abstract. We present three algorithms for Cholesky factorization using minimum block storage for a distributed memory (DM) environment. One of the distributed square block packed (SBP) format algorithms performs similar to ScaLAPACK `PDPOTRF`, and our algorithm with iteration overlapping typically outperforms it by 15–50% for small and medium sized matrices. By storing the blocks contiguously, we get better performing BLAS operations. Our DM algorithms are not sensitive to cache conflicts and thus give smooth and predictable performance. We also investigate the intricacies of using rectangular full packed (RFP) format with ScaLAPACK routines and point out some advantages and drawbacks.

1 Introduction

Dense linear algebra routines that are implemented in a distributed memory environment typically use a 2D block cyclic layout (BCL), with ScaLAPACK being one example of a library that uses BCL for all routines [3]. A BCL can provide effective load balance for many algorithms. The mapping of matrix elements to processors does not prescribe how they are later stored on each processor. The approach taken by the ScaLAPACK library is to store each elementary block as a submatrix of a column major 2D array (standard Fortran array) [3]. Another approach is to store each elementary block contiguously, for example as a column major block 2D array.

Storing elementary blocks contiguously has at least three advantages. They will map very well into L1 cache and level 3 operations involving such blocks will therefore tend to achieve high performance and minimize memory traffic. Another benefit is that moving a block can be done by one contiguous memory transfer. In this contribution we use *square elementary blocks* (called a square block, or SB) to store the local matrix. Furthermore, we store only the triangular part of the block matrix to achieve minimum block storage for symmetric matrices. We call this *square block packed* (SBP) format.

We identify an inefficiency in straightforward data parallel implementations, e.g., the implementation of the Cholesky factorization in ScaLAPACK (routine `PDPOTRF`) and develop an iteration overlapping data parallel implementation which removes much of the idling and thus decreases execution time.

2 Near Minimal Storage in a Serial Environment

A recently proposed format for storing triangular or symmetric matrices is called *rectangular full packed* (RFP) (see [8] for details). This format takes many slightly different forms. Figure 1 illustrates a lower triangular matrix. The matrix is

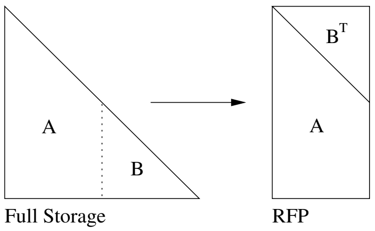

Fig. 1. Illustration of rectangular full packed format

partitioned into two submatrices A and B. The triangular matrix B^T is merged along the diagonal with A. As can be seen, this new matrix can be stored as a standard full format rectangular array with no waste of memory.

Another format for near minimal storage is a generalization of a standard column major format. The matrix is divided into square blocks and the format is based on storing each such block in a contiguous memory area. The blocks can then be stored for example in either a row or column major ordering. Figure 2

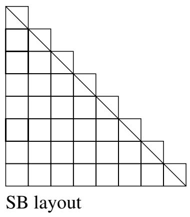

Fig. 2. Illustration of square block packed format

illustrates the square blocks. The elements above the diagonal of the diagonal blocks are wasted storage. By picking the block size to one, we see that we get either the standard row or column major format. For details on this format see [7].

3 Minimum Block Storage in a Distributed Environment

In this section we describe how RFP and SBP can be used in a distributed memory environment. We show how both approaches give a nearly minimum block storage.

3.1 A Distributed SBP Algorithm for Cholesky Factorization

In this contribution we consider the blocked algorithmic variant of Cholesky factorization described in Algorithm 1. We note that this algorithmic variant is used in ScaLAPACK [4]. In a distributed environment with a 2D block cyclic

Algorithm 1. Standard blocked Cholesky factorization

```
1: for each panel left to right do
2:     Partition $A = \begin{bmatrix} A_{11} & \\ A_{21} & A_{22} \end{bmatrix}$, where $A_{11}$ is NB×NB
3:     Factorize $A_{11} = LL^T$ using unblocked algorithm
4:     Update panel $A_{21} := A_{21}L^{-T}$ using triangular solver
5:     Update trailing matrix $A_{22} := A_{22} - A_{21}A_{21}^T$ using symmetric rank-$k$ update
6:     Continue with $A = A_{22}$
7: end for
```

layout with block size NB×NB, block A_{11} resides on one processor, block A_{21} on one processor column, and A_{22} generally resides on all processors. By using parallel triangular solve and symmetric rank-k update routines Algorithm 1 will achieve scalable performance due to good load balance and because most of the computation is in step 5 which is easy to parallelize. However, steps 3 and 4 do not utilize all processors effectively. One variant of this algorithm is to start the next iteration before the current iteration has finished step 5 (see [7] for more details). This is possible by noting that the first updated column panel of the new pivot from step 5 will be used as the only input for step 3 and 4 of the next iteration.

A major problem with a straightforward parallel implementation of Algorithm 1 is the idle time introduced when processors implicitly synchronize after each iteration. This idle time is caused both by slight load imbalances and the work in steps 3 and 4 that are not performed on all processors. By using the iteration overlapping algorithm this idle time will be eliminated if the communication of data between steps 3 and 4 can be carried out while still doing useful work in updates.

The data dependencies in Algorithm 1 are simple. Output from step 3 is input for step 4 whose output in turn is input for step 5. As for the first dependency a column broadcast is all that is needed. The second dependency requires a somewhat more complicated communication pattern and is now described briefly. All subblocks of A_{21} are broadcasted along the processor rows. Once a subblock of

A_{21} reaches the processor holding the diagonal block of that row it is broadcasted along its processor column. One can show that after this, each processor holds the blocks of A_{21} and A_{21}^T that it needs for step 5. In our implementation these blocks are stored in two block buffer vectors W and S, where W (for West border vector) holds blocks of A_{21} and S (for South border vector) holds blocks of A_{21}^T.

We have studied how overlapping two successive pivot steps can affect the performance of our parallel implementation. Our implementation is described in Algorithm 2. The overlapping in Algorithm 2 happens during the execution

Algorithm 2. Cholesky with iteration overlap

```
 1: for each panel left to right do
 2:     Partition global $A = \begin{bmatrix} A_{11} & \\ A_{21} & A_{22} \end{bmatrix}$, where $A_{11}$ is NB×NB
 3:     if process holds $A_{11}$ then
 4:         Factorize $A_{11} = LL^T$ using serial algorithm
 5:         Column broadcast the block $L$
 6:     end if
 7:     if process column holds $A_{21}$ then
 8:         Receive the block $L$
 9:         Partition $S_1 = \begin{bmatrix} S_A & S_B \end{bmatrix}$, where $S_A$ is NB×NB
10:         Update $A_{21} := A_{21} - W_1 S_A$
11:         Scale $A_{21} := A_{21} L^{-T}$
12:         Start communication of $A_{21}$ using buffers $W_2$ and $S_2$ (sender)
13:         Update $A_{22} := A_{22} - W_1 S_B$
14:     else {all other process columns}
15:         Start communication of $A_{21}$ using buffers $W_2$ and $S_2$ (receiver)
16:         Update $A := A - W_1 S_1$
17:     end if
18:     Move (symbolically) $W_1 := W_2$ and $S_1 := S_2$ {there is no data movement}
19: end for
```

of steps 10 to 13. Taken together, steps 10 and 13 perform a complete update. The execution order of the straightforward algorithm would put steps 10 and 13 together, and step 11 after both. Executing step 11 before 13 allows the communication needed for the subsequent update to take place during the update in step 13 (and while the other processors execute step 16).

In Figure 3, we illustrate by example how our local matrices are stored in practice. The blocks are stored columnwise in a one-dimensional block vector indexed by a column pointer (CP) array. The entries in CP are pointers to the first block of each block column.

Figure 4 shows how the two sets of buffers are used in Algorithm 2. The light shaded blocks are those used for the update of iteration i. The darker shaded blocks are those computed during iteration i for use in iteration $i+1$. After the panel factorization the communication algorithm is started and it will broadcast the panel and its transpose to all processors with this data stored in the second

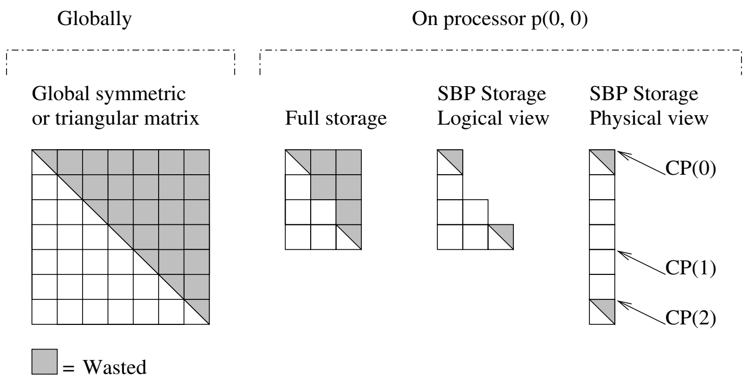

Fig. 3. Illustration of how a 7×7 block global matrix is laid out on a 2 × 3 mesh in SBP format and addressed with its column pointer (CP) array. The full size of the global matrix is `7NB`×`7NB`.

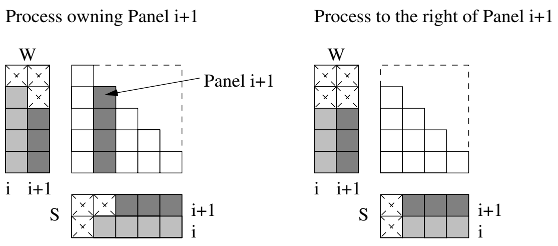

Fig. 4. Data layout for the SBP with double sets of W and S border vectors

set of buffers. While this communication takes place, the first set of buffers is used to finish the update of iteration i.

3.2 A Distributed RFP Algorithm for Cholesky Factorization

Because of the good performance achievable with RFP format in a serial environment (see [9]) we investigated its extension to parallel environments via using ScaLAPACK and PBLAS. Algorithm 3 gives the details of the RFP Cholesky algorithm. The limitations of PBLAS and ScaLAPACK do not generally allow matrices to begin inside an elementary block; each submatrix must be block aligned. Therefore, we use the RFP format on the block level, introducing some wasted storage and thus achieve minimum block storage while still being able to use RFP with existing routines.

The RFP format could be used with an algorithm similar to the one we used with SBP. Such an RFP algorithm would probably achieve similar performance to the SBP algorithm so we did not develop any implementation of it.

Algorithm 3. RFP Cholesky with ScaLAPACK/PBLAS routines

```
1: Matrix $A$ is in RFP format: $A = \begin{bmatrix} A_{11} \backslash A_{22}^T \\ A_{21} \end{bmatrix}$
2: Factor $A_{11} = LL^T$ using ScaLAPACK routine PDPOTRF
3: Update panel $A_{21} := A_{21} L^{-T}$ using PBLAS routine PDTRSM
4: Update trailing matrix $A_{22} := A_{22} - A_{21} A_{21}^T$ using PBLAS routine PDSYRK
5: Factor $A_{22} = LL^T$ using ScaLAPACK routine PDPOTRF
```

4 Related Work on DM Cholesky Factorization

We briefly discuss other packed storage schemes for DM environments.

D'Azevedo and Dongarra suggested in 1997 a storage scheme where the elementary blocks are mapped to the same processor as in the full storage case, but only the non-redundant blocks are stored [6]. Each block column is stored as a submatrix the same way as it would in full storage. The result is that each block column is a regular ScaLAPACK matrix and can be used as such. Note that the blocks will be mapped to the same processors as the SBP format, but the local processor storage layout is different. Benefits include routine reuse via PBLAS and ScaLAPACK routines. However, some new PBLAS routines seem to be required to handle the packed storage [6]. Furthermore, their results indicate that the performance varies wildly with input, making performance extrapolation difficult.

Recently, Marc Baboulin et al. presented a storage scheme which uses relatively large square blocks consisting of at least LCM(p, q)[1] elementary blocks [2]. This format also supports code reuse via PBLAS and ScaLAPACK. The granularity is limited to the distributed block size, which means less possibility to save memory. For the Cholesky factorization routines, the chosen block sizes for performance measurements were between 1024 and 10240. This resulted in a departure from their minimum storage by as much as 7–13%. Using their minimum allowed distributed block size would bring this percentage down to about 1–3% but at the cost of longer execution times.

5 Performance Results and Comparison

In this section we give some performance related results. We compare RFP, SBP and ScaLAPACK routines and analyze the differences that we observed.

All tests were performed on the *Sarek* cluster at HPC2N. It consists of 190 HP DL145 nodes, with dual AMD Opteron 248 (2.2GHz) processor and 8 GB memory per node. The AMD Opteron 248 processor has a 64 kB instruction and 64 kB data L1 Cache (2-way associative) and a 1024 kB unified L2 Cache (16-way associative). The cluster's operating system is Debian GNU/Linux 3.1 and we used Goto BLAS 0.94 throughout.

[1] The least common multiple of the integers a and b (written LCM(a, b)) is the smallest integer that is a multiple of both a and b.

Table 1. Execution times for PDPOTRF and the SBP algorithm with iteration overlap for various square grid sizes. The block size NB is set to 100.

N	2x2	3x3	4x4	5x5	6x6	7x7
4000	2.13/0.86	1.48/0.63	1.04/0.66	0.79/0.68	0.63/0.64	0.57/0.65
8000	14.80/0.92	8.29/0.80	5.33/0.79	3.97/0.77	3.15/0.71	2.64/0.73
12000		25.20/0.83	16.30/0.80	10.90/0.84	8.27/0.80	7.11/0.78
16000		57.30/0.84	34.50/0.85	24.00/0.85	18.30/0.80	13.90/0.85
20000			65.00/0.85	43.90/0.86	33.00/0.81	25.90/0.84
24000					53.90/0.84	42.30/0.85

Table 1 shows selected times for both PDPOTRF and the SBP algorithm with iteration overlap. Each cell has the form X/y, where X is the time (in seconds) of the PDPOTRF routine and y = Y/X, where Y is the time for the SBP algorithm. The same block size was used for both implementations. We identify two trends. First of all, the relative gain by overlapping increases with the number of processors since the idle time is introduced on the entire mesh. The bigger the mesh the more idle time we can remove by overlapping. Second, the relative gain decreases with increasing problem sizes. This is expected because the dominant operation is the trailing matrix update (with $\mathcal{O}\left(N^3\right)$ flops) whereas the operations causing idle time (the panel factorization) make up for only $\mathcal{O}\left(N^2\right)$ flops.

Table 2. Execution time for PDPOTRF and the RFP algorithm using ScaLAPACK routines for various grid sizes

N	2x2	3x3	4x4	5x5	6x6	7x7
4000	2.13/1.26	1.48/1.16	1.04/1.18	0.79/1.44	0.63/1.42	0.58/1.34
8000	14.80/*1.48*	8.29/1.25	5.33/1.23	3.97/1.37	3.15/1.33	2.64/1.33
12000		25.20/*1.41*	16.30/1.14	10.90/1.34	8.27/1.33	7.11/1.29
16000		57.30/1.16	34.50/*1.34*	24.00/1.28	18.30/1.20	13.90/1.34
20000			65.00/1.13	43.90/*1.40*	33.00/1.22	25.90/1.25

Table 2 is similar to Table 1 but shows selected times for PDPOTRF and our RFP algorithm which uses four calls to ScaLAPACK/PBLAS routines. Each cell has the form X/y, where X is the time (in seconds) of the PDPOTRF routine and y = Y/X, where Y is the time for the RFP algorithm. As can be seen from this table the RFP algorithm has typically a 10–30% longer execution time. By tracing the execution of the algorithm we found two substantial causes for this overhead. The performance of the BLAS operations issued by the RFP algorithm was less efficient than was typical for the other algorithms we tested. Moreover, there are more synchronization points in the RFP algorithm due to the two ScaLAPACK and two PBLAS calls on problems half the size. This amplifies the communication overhead and load imbalance. Taken together, this would probably explain most of the time differences we observed. One interesting detail to note in Table 2 is that when the local matrix dimension is 4000 the RFP algorithm experienced a dramatic loss in performance (emphasized by italics in Table 2). This is caused by a cache effect because the leading dimension is actually 4100 which is close to $2^{12} = 4096$; also the L1 cache on Sarek is only 2-way set associative.

The block size mainly affects performance of the BLAS operations and the load balance. Larger blocks tend to give good BLAS performance but less load balance. For the SBP algorithm the block size is intimately related to BLAS performance because then all `GEMM` calls are on matrices of order `NB`. The ScaLAPACK algorithm is less dependent on the block size because of the fewer and larger PBLAS operations. Table 3 gives an idea of how the block size relates to

Table 3. Impact of block size on performance (measured in Gflops/s per processor) for ScaLAPACK `PDPOTRF` and our overlapping SBP algorithm

NB	**PDPOTRF**	**Overlapping**
25	2.08	1.91
50	2.12	2.57
75	2.09	2.75
100	2.15	3.04
125	2.24	3.06
150	2.13	3.04

performance for both of these algorithms. The processor mesh was 2×3 and the order of the matrix was `N=6000`. On Sarek we see that when we approach a block size of 100 we get close to optimal performance, whereas the block size does not matter much for the ScaLAPACK routine. The gap in performance between the two routines is mainly due to less idling in the overlapping routine.

Finally, we note that our overlapping SBP algorithm could be modified so that it updates first and factorizes the next panel afterwards. This makes the algorithm essentially equal to the straightforward implementation but with a different data format. We implemented this variant too and found that as expected it gave performance nearly identical to the ScaLAPACK algorithm.

6 Future Work

We outline some future directions of development. Our overlapping algorithm relies on the idea that the task of trailing matrix update can be divided into two tasks: the first panel on the column of processors holding the pivot and the rest of the panels on all processors. This allows us to have two iterations on the same processor, but three is not possible. A solution is to further divide the tasks. The trailing matrix update could for example be divided into one task for each block column. Instead of waiting for data it now becomes attractive to do smaller tasks instead. The order of the tasks thus becomes non-deterministic because it would depend on processor interactions. To get a clean implementation it might be necessary to use a style reminiscent of a work pool.

The overlapping algorithm relies heavily on the interleaving of communication and updates. One consequence of the overlapping is that more workspace is needed. In general each ongoing iteration will require its own W and S buffer. It is preferable to have many iterations ongoing because in that way more work is kept at each processor and chances for idling will get reduced. The concept of lookahead in factorization algorithms has been addressed several times (cf. [1,5,7]) and recently in [10]. The emphasis of the latter contribution is that a dynamic lookahead is most appropriate. A large lookahead is not feasible in a DM environment because of the large workspace required. Setting a fixed cap (or dynamic relative to a fixed workspace) on the number of iterations may be a feasible solution.

Our work provides an argument for the inclusion of *nonblocking collective* communication routines in communication libraries. The de-facto industry standard MPI has substantial support for nonblocking point-to-point communication but collectives are all blocking. Our implementation emulates nonblocking collectives by repeatedly testing for individual completion of nonblocking point-to-point operations. This complicates the code and probably comes at a higher cost than would have been the case if nonblocking collectives existed as part of the library.

7 Conclusion

We have implemented and compared three algorithms and data formats for minimum block storage in distributed memory environments using a 2D block cyclic data layout.

In a serial environment, the RFP format is an attractive choice [9]. However, the straightforward generalization of serial RFP algorithms has some weaknesses.

The SBP format was implemented and tested with two algorithm variants. One resembles ScaLAPACK's `PDPOTRF` but makes no use of PBLAS or ScaLAPACK routines, and one overlaps iterations. We have demonstrated that performance at least as good as the ScaLAPACK algorithm is attainable, and for the overlapping variant far better performance, especially for small and medium sized matrices, was achieved.

The ideas that we explored in this work can be applied to many other algorithms as well. Two examples very similar to the Cholesky factorization are the LU and QR factorizations.

Acknowledgements. This research was conducted using the resources of the High Performance Computing Center North (HPC2N). Financial support has been 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.

References

1. Agarwal, R.C., Gustavson, F.G.: A parallel implementation of matrix multiplication and LU factorization on the IBM 3090. In: Wright, M. (ed.) Aspects of Computation on Asynchronous and Parallel Processors, pp. 217–221. IFIP, North-Holland, Amsterdam (1989)
2. Baboulin, M., Giraud, L., Gratton, S., Langou, J.: A distributed packed storage for large parallel calculations. Technical Report TR/PA/05/30, CERFACS, Toulouse, France (2005)
3. Blackford, L.S., et al.: ScaLAPACK user's guide. SIAM Publications (1997)
4. Choi, J., Dongarra, J.J., Ostrouchov, S., Petitet, A.P., Walker, D.W., Whaley, R.C.: Design and implementation of the ScaLAPACK LU, QR, and Cholesky factorization routines. Scientific Programming 5(3), 173–184 (1996)
5. Dackland, K., Elmroth, E., Kågström, B.: A ring–oriented approach for block matrix factorizations on shared and distributed memory architectures. In: Sincovec, R.F., et al. (eds.) SIAM Conference on Parallel Processing for Scientific Computing, pp. 330–338. SIAM Publications (1993)
6. D'Azevedo, E., Dongarra, J.: Packed storage extension for ScaLAPACK. Technical Report UT-CS-98-385 (1998)
7. Gustavson, F.: Algorithm compiler architecture interaction relative to dense linear algebra. Technical Report RC 23715, IBM Thomas J. Watson Research Center (September 2005)
8. Gustavson, F.: New generalized data structures for matrices lead to a variety of high performance dense linear algebra algorithms. In: Dongarra, J.J., Madsen, K., Waśniewski, J. (eds.) PARA 2004. LNCS, vol. 3732, pp. 11–20. Springer, Heidelberg (2006)
9. Gustavson, F.G., Wasniewski, J.: Rectangular Full Packed Format for LAPACK Algorithms Timings on Several Computers. In: Kågström, B., Elmroth, E., Dongarra, J., Wasniewski, J. (eds.) PARA 2006. LNCS, vol. 4699, pp. 570–579. Springer, Heidelberg (2007)
10. Kurzak, J., Dongarra, J.: Implementing Linear Algebra Routines on Multi-core Processors with Pipelining and a Look Ahead. In: Kågström, B., Elmroth, E., Dongarra, J., Wasniewski, J. (eds.) PARA 2006. LNCS, vol. 4699, pp. 147–156. Springer, Heidelberg (2007)