Implementation in ScaLAPACK of Divide-and-Conquer Algorithms for Banded and Tridiagonal Linear Systems

A. Cleary
Department of Computer Science
University of Tennessee

J. Dongarra
Department of Computer Science
University of Tennessee
Mathematical Sciences Section
Oak Ridge National Laboratory

Abstract

Described here are the design and implementation of a family of algorithms for a variety of classes of narrowly banded linear systems. The classes of matrices include symmetric and positive definite, nonsymmetric but diagonally dominant, and general nonsymmetric; and, all these types are addressed for both general band and tridiagonal matrices. The family of algorithms captures the general flavor of existing divide-and-conquer algorithms for banded matrices in that they have three distinct phases, the first and last of which are completely parallel, and the second of which is the parallel bottleneck. The algorithms have been modified so that they have the desirable property that they are the same mathematically as existing factorizations (Cholesky, Gaussian elimination) of suitably reordered matrices. This approach represents a departure in the nonsymmetric case from existing methods, but has the practical benefits of a smaller and more easily handled reduced system. All codes implement a block odd-even reduction for the reduced system that allows the algorithm to scale far better than existing codes that use variants of sequential solution methods for the reduced system. A cross section of results is displayed that supports the predicted performance results for the algorithms. Comparison with existing dense-type methods shows that for areas of the problem parameter space with low bandwidth and/or high number of processors, the family of algorithms described here is superior.

1 Introduction

We are concerned in this work with the solution of banded linear systems of equations

$$Ax = b.$$

The matrix A is $n \times n$. In general, x and b are $n \times nrhs$ matrices, but it is sufficient to consider $nrhs = 1$ for explanation of the algorithms. Here, $nrhs$ is the number of right-hand sides in the system of equations.

The matrix A is *banded,* with lower bandwidth β_l and upper bandwidth β_u. This has the following meaning:

If $i \geq j$, then $i - j > \beta_l \Rightarrow A_{i,j} \equiv 0$; if $i \leq j$, then $j - i > \beta_u \Rightarrow A_{i,j} \equiv 0$.

Thus, the matrix entries are identically zero outside of a certain distance from the main diagonal. We are concerned here with *narrowly banded* matrices, that is, $n \gg \beta_l, n \gg \beta_u$. Note that while, in general, entries within the band may also be numerically zero, it is assumed in the code that all entries within the band are nonzero. As a special case of narrowly banded matrices, we are also concerned with tridiagonal matrices, specifically, $\beta_l = \beta_u = 1$.

Three classes of matrix are of interest: general nonsymmetric, symmetric and positive definite, and nonsymmetric with conditions that allow for stable computations without pivoting, such as *diagonal dominance*. In this article we focus on the last two cases and indicate how the family of algorithms we present can apply to other classes of matrix. The first class, general nonsymmetric, is the subject of a forthcoming report.

This article concentrates on issues related to the inclusion in ScaLAPACK [CDPW93, BCC+97] of library-quality implementations of the algorithms discussed here. ScaLAPACK is a public-domain portable software library that provides broad functionality in linear algebra mathematical software to a wide variety of distributed-memory parallel systems. Details on retrieving the codes discussed here, as well as the rest of the ScaLAPACK package, are given at the end of this article.

Algorithms for factoring a single banded matrix A fall into two distinct classes.

- **Dense-type**: Fine-grained parallelism such as is exploited in dense matrix algorithms is used. The banded structure is used to reduce operation counts but not to provide parallelism.
- **Divide and Conquer**: Medium/large-grained parallelism resulting from the structure of banded matrices is used to divide the matrix into large chunks that, for the most part, can be dealt with independently.

The key parameter for choosing between the two classes of algorithms is the bandwidth. Parallelism of the dense methods depends on the bandwidth in the exact same way that parallelism in dense matrix algorithms depends on the matrix size. For small bandwidths, dense methods are very inefficient. Reordering methods are the opposite: the smaller the bandwidth, the greater the potential level of parallelism. However, reordering methods have an automatic penalty that reduces the spectrum over which they are the methods of choice: because of fill-in, they incur an operation count penalty of approximately four compared with sequential algorithms. They are therefore limited to an efficiency of no greater than 25%. Nonetheless, these algorithms scale well and must be included in a parallel library.

Many studies of parallel computing reported in the literature [Wri91, AG95, GGJT96, BCD+94, Joh87, LS84] address either the general banded problem or special cases such as the tridiagonal problem. Wright [Wri91] presented an ambitious attempt at incorporating pivoting, both row and column, throughout the entire calculation. His single-width separator algorithm without pivoting is similar to the work described here. However, Wright targeted relatively small parallel systems and, in particular, used a sequential method for the reduced system. Arbenz and Gander [AG95] present experimental results demonstrating that sequential solution of the reduced system seriously impacts scalability. They discuss the basic divide-and-conquer algorithms used here, but do not give the level of detail on implementation that we present. Gustavson et al. [GGJT96] tackle the wide-banded case with a systolic approach that involves an initial remapping of the data. Cleary [BCD+94] presents an in-place wide-banded algorithm that is much closer to the ScaLAPACK-style of algorithm [CDPW93], utilizing fine-grained parallelism within BLAS-3 for parallel performance.

2 Divide-and-Conquer Algorithms

In this article, we are concerned only with divide-and-conquer methods that are appropriate for narrowly banded matrices. The family of divide-and-conquer algorithms used in ScaLAPACK perform the following algebraic steps. Here, P is an $n \times n$ permutation matrix (specified in Section 3) that reorders A to allow exploitation of parallelism.

First, $Ax = b$ is multiplied on the left by P to produce the equation

$$PA(P^{-1}P)x = Pb.$$

The reordered matrix PAP^{-1} is factored via Gaussian elimination as (if A is symmetric and positive definite, Cholesky decomposition is used and $U = L^T$)

$$PAP^{-1} = LU.$$

Substituting this factorization and the following definitions,

$$x' = Px, b' = Pb,$$

we are left with the system

$$LUx' = b'.$$

This is solved in the traditional fashion by using triangular solutions:

$$Lz = b', Ux' = z.$$

The final step is recovery of x from x':

$$x = P^{-1}x'.$$

3 The Symmetric Positive Definite Case

Dongarra and Johnsson [DJ87] showed that their divide-and-conquer algorithm, when applied correctly to a symmetric positive definite matrix, can take advantage of symmetry throughout, including the solution of the reduced system. In fact, the reduced system is itself symmetric positive definite. This fact can be explained easily in terms of sparse matrix theory: *the algorithm is equivalent to applying a symmetric permutation to the original matrix and performing Cholesky factorization on the reordered matrix.* This follows from the fact that the matrix PAP^T resulting from a symmetric permutation P applied to a symmetric positive definite matrix A is also symmetric positive definite and thus has a unique Cholesky factorization L. The reordering is applied to the right-hand sides as well, and the solution must be permuted back to the original ordering to give the final solution. The key point is that, mathematically, the whole process can be analyzed and described in terms of the Cholesky factorization of a specially structured sparse matrix.

We take this approach in the sequel. Figure 1 pictorially illustrates the user matrix upon input, assuming the user chooses to input the matrix in lower triangular form (analogous to banded routines in LAPACK [ABB$^+$95], we provide the option of entering the matrix in either lower or upper form). Each processor stores a contiguous set of columns of the matrix, denoted by the thicker lines in the figure. We partition each processor's matrix into blocks, as shown. The sizes of the A_i are given respectively by O_i. The matrices B_i, C_i, D_i are all $\beta \times \beta$. Note that the last processor has only A_i.

The specific reordering can be described as follows:

- Number the equations in the A_i first, keeping the same relative order of these equations.

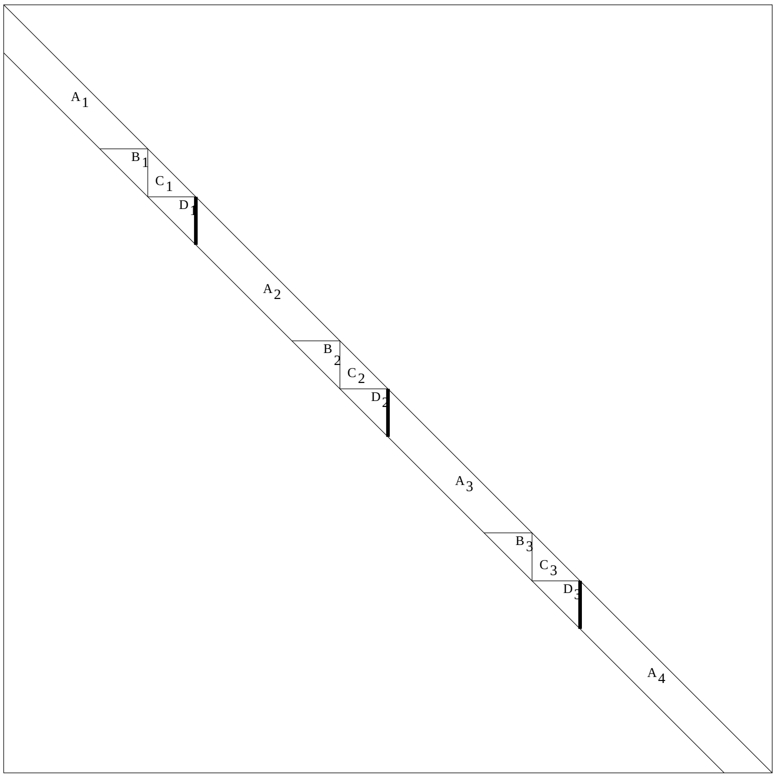

Figure 1: Divide-and-conquer partitioning of lower triangular portion of a symmetric matrix

- Number the equations in the C_i next, again keeping the same relative order of these equations.

This results in the lower triangular matrix shown in Figure 2. We stress that we do not physically reorder the matrix but, rather, base our block operations on this mathematical reordering.

The Cholesky factorization of the matrix in Figure 2 can be computed largely with sequential block operations, as can the solution of a linear system with these factors. The resultant mathematical Cholesky factorization is illustrated in Figure 3. Communication between processors is needed only for a small portion of the total computation.

Figure 3 illustrates the major weakness of divide-and-conquer algorithms for banded matrices: *fill-in*. The blocks G_i and H_i represent fill-in that approximately doubles the number of nonzeros in the Cholesky factorization when compared with the factor produced by the sequential algorithm applied to the original matrix. While fill-in itself is not very expensive, the operation count of the factorization is approximately four times that of the sequential algorithm. This can be seen in block terms (which we give in more detail later): the factorization of the A_i sum to $N\beta^2 + O(N\beta)$, which is the same as the sequential operation count in the higher-order term. However, forming the G_i has an operation count of $2N\beta^2 + O(N\beta)$, and using the G_i to modify the C_i sums to $N\beta^2 + O(N\beta)$ (the other block operations are of order less than $N\beta^2$). Summing these terms gives the factor of four degradation.

The computational process is generally regarded as occurring in three phases:

Phase 1 : Formation of the reduced system. Each processor does computations independently (for the most part) with local parts and then combines to form the Schur complement system

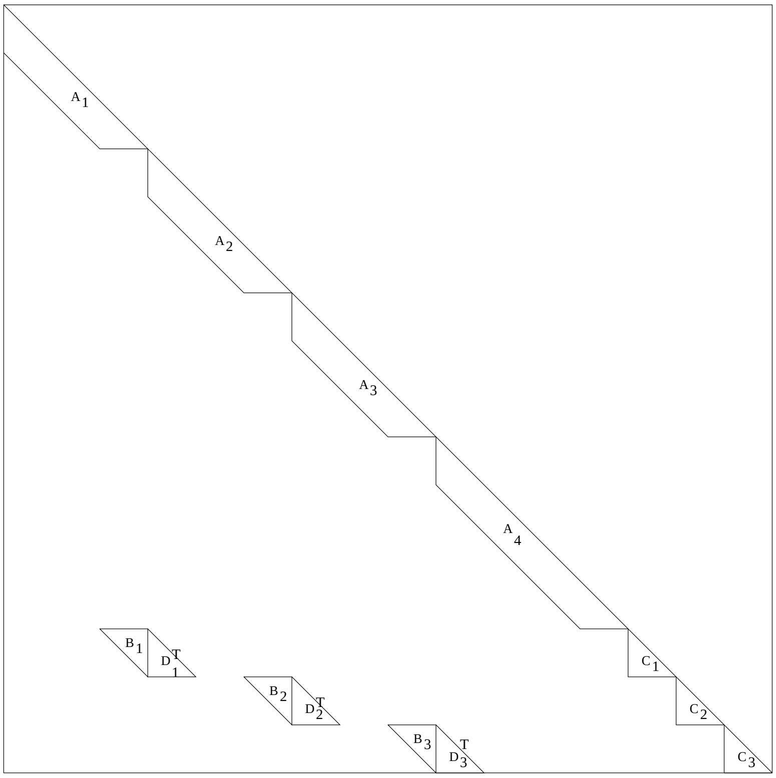

Figure 2: Matrix after divide-and-conquer reordering

corresponding to the parts already factored. The Schur complement is often called the *reduced system*.

Phase 2 : The reduced system is solved, and the answers are communicated back to all of the processors.

Phase 3 : The solutions from Phase 2 are applied in a backsolution process.

3.1 Phase 1

We concentrate now on the local computations in Phase 1. For illustration we will look at the ith processor. Note that the first and last processors do not have to perform all of these steps.

The first step is a communication step: D_i is sent to processor $i+1$. This is a small communication and is completely overlapped with the subsequent computation steps.

At this point, the portions of the matrix are stored locally, as illustrated by Figure 4. In this figure, we view the local computations as a frontal computation. Hence, in this case, we take O_i factorization steps and apply them to the remaining submatrix of size 2β. This submatrix is then subsequently used in Phase 2 to form the reduced system. Mathematically, this is exactly what the local operations equate to. The frontal calculations have specific and unique structure dictated by the way in which the frontal matrix was derived, but this remains a frontal calculation. The "divide" in the algorithm's name is a result of the fact that the reordering allows each of the fronts so defined to be independent. Only the 2β update equations at the end of each front need be coordinated with other processors.

It is relatively easy to derive a block Cholesky formulation, as well as a partial factorization as

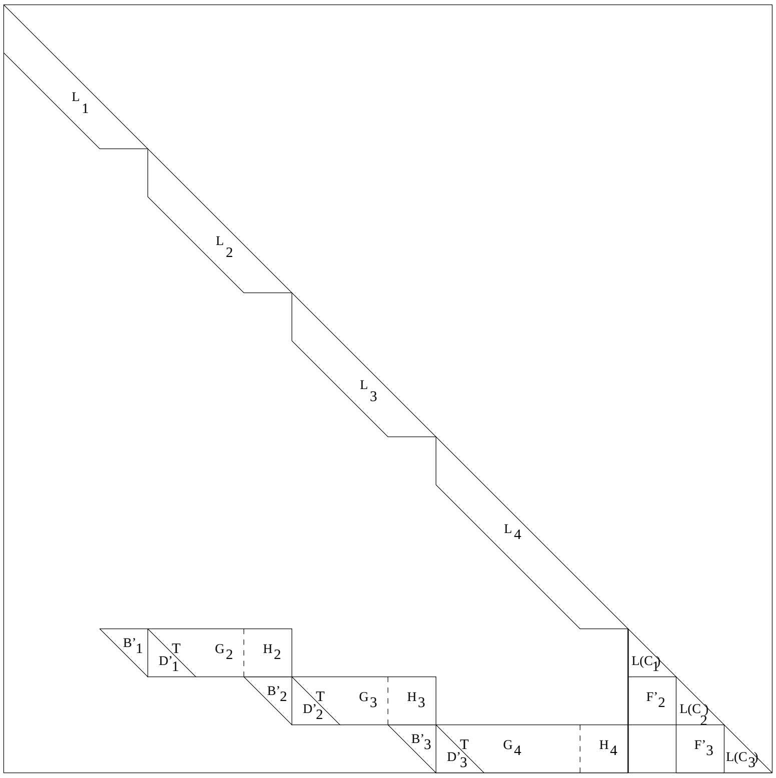

Figure 3: Cholesky factor of reordered divide-and-conquer matrix

is required in this case, by merely equating the blocks of the original matrix with the corresponding blocks of the product of the desired Cholesky factors and then solving for the blocks of the Cholesky factors. We will not reproduce this derivation here. Ultimately, the local frontal calculations will result in the matrix illustrated in Figure 5. In this figure, A_i is of size $O_i \times O_i$, and G_i is of size $\beta \times O_i$. All other blocks are of size $\beta \times \beta$, including D_i and H_i, which are actually subblocks of G_i. We now illustrate the sequence of steps necessary to arrive at this state.

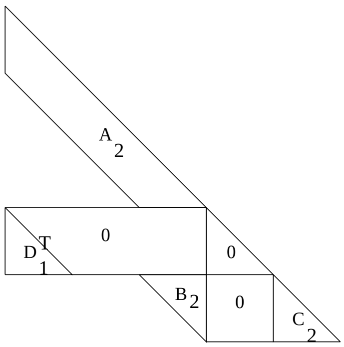

Figure 4: Local matrix on processor 2 after initial communication of D_i

Because the reordering allows the Cholesky factorization to begin simultaneously with each of the A_i, the first computational step in processor i is the factorization

$$A_i = L_i {L_i}^T.$$

This is easily done via a single call to the LAPACK [ABB+95] routine `DPBTRF`. The matrix resulting after this step is illustrated in Figure 6. We allow the factors to overwrite the storage for A_i.

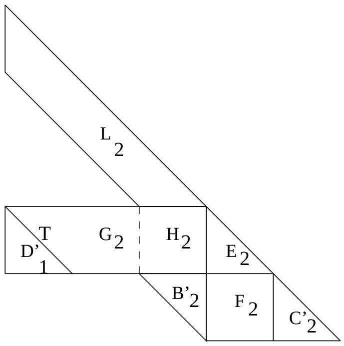

Figure 5: Local storage after completion of Phase 1

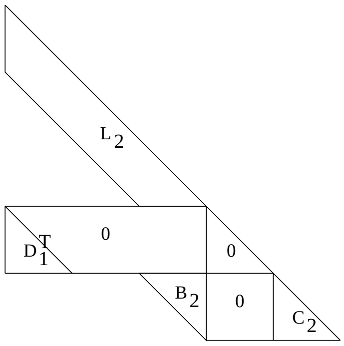

Figure 6: Local matrix on processor 2 after factorization of L_i

The factors just computed are used to complete the factorization of the B_i by solving

$$L_i {B'_i}^T = {B_i}^T.$$

The matrix B_i is transposed and copied into the workspace, since the transpose is needed in the next step. Once the matrix is in the workspace, the BLAS [DDHH84] subroutine `DTRTRS` is used to compute ${B'_i}^T$, which is copied back into the space held by B_i. Mathematically, the local matrix is given by Figure 7.

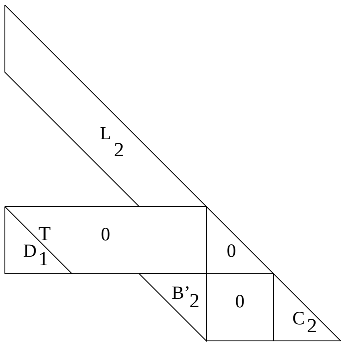

Figure 7: Local matrix on processor 2 after modification of B_i

The matrix ${B'_i}^T$ is used to modify C_i according to

$$C'_i = C_i - B'_i {B'_i}^T.$$

Only the lower half of C_i is to be modified (because of symmetry). The BLAS do not specify routines for multiplying *two* triangular matrices, so the choice is to either write a new BLAS-like routine or use a BLAS routine designed for a dense-by-triangular matrix multiplication. We chose the latter route to maintain full dependence on the BLAS, although this uses more operations than necessary. However, the assumption that $\beta << N$ and the fact that these matrices are $\beta \times \beta$ make the impact of the extra operations negligible.

Figure 8 shows the state after this computation.

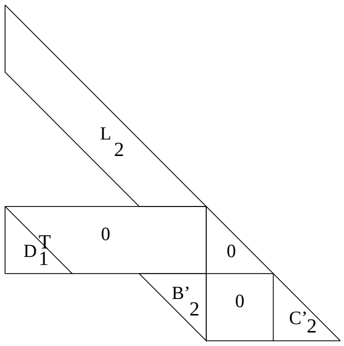

Figure 8: Local matrix on processor 2 after modification of C_i

Only at this point do the processors need to execute the receive operation for D_{i-1} transmitted from the previous processor. This is received into auxiliary space because the entire matrix G_i represents fill-in and cannot overwrite the original matrix. Since subsequent operations operate with G_i^T, D_{i-1} is actually stored in G_i^T. This fill-in is often referred to as the *spike* in this algorithm.

The calculation of G_i^T is accomplished as

$$L_i {G_i}^T = D_i$$

(some liberty has been taken with the sizes of matrices in this expression, but it easy to interpret properly), by using the LAPACK routine `DTBTRS`. We note that in terms of operation count, this step is the most costly, and thus its serial efficiency is key to the efficiency of the entire algorithm. Figure 9 shows the state after this computation.

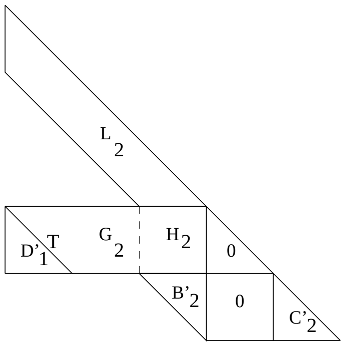

Figure 9: Local matrix on processor 2 after calculation of the spike fill-in G_i

The matrix E_i represents the contribution from processor i to the diagonal block of the reduced system stored on processor $i-1$, that is, C'_{i-1}. It is calculated by the BLAS routine `DSYRK` according to the formula

$$E_i = G_i {G_i}^T.$$

Figure 10 shows the state after this computation.

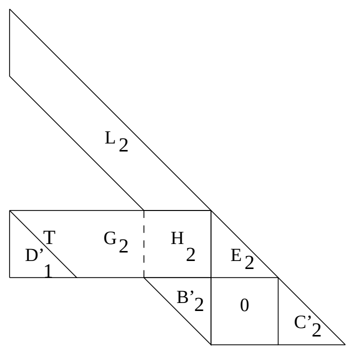

Figure 10: Local matrix on processor 2 using the spike fill-in G_i to calculate E_i

The local computation phase is completed by computing F_i, using B_i' and the last β columns of G_i, which we label as H_i. The BLAS routine DTRMM accomplishes this task, although it requires the use of a data copy because of the way it is defined. However, this data copy comes for free because F_i is also fill-in and thus must be stored in work storage and cannot be calculated in-place. The computation is

$$F_i = B'_i {H_i}^T.$$

However, for ease of use in the subsequent reduced system factorization, we actually compute and store the transpose,

$$F_i^T = H_i {B'_i}^T.$$

3.2 Phase 2

Phase 2 consists of the forming and factorization of the Schur complement matrix. Each processor contributes three blocks of size $\beta \times \beta$ to this system: E_i, F_i, C_i'. Each C_i' is added to E_{i+1} to form the diagonal blocks of the matrix, and the F_i form the off-diagonal blocks. The resultant system is block tridiagonal, with $P-1$ blocks.

Several methods for factoring the reduced system have been proposed and implemented in the past. For small P or small β, an efficient algorithm is to perform an all-to-all broadcast of each processor's portion of the reduced system, leaving the entire reduced system on each processor [GGJT96]. Each processor then solves this system locally. The advantage of this scheme is that there is only one communication step, albeit an expensive one whose cost grows quickly with P. The disadvantage is that since each processor is performing redundant computation, the algorithm is essentially serial and will not scale.

An algorithm with similar performance characteristics involves gathering the reduced system on a single processor, solving it sequentially, and broadcasting the results. Again, the principal disadvantage is the lack of scalability due to the sequential solution. Arbenz and Gander [AG95] show that the best time for this algorithm occurs at $P = 20$, which is a disaster for scalability. Theoretically and practically they show the need for a parallel reduced system algorithm, which is what we have implemented for ScaLAPACK.

We use a block formulation of odd-even (or cyclic) reduction. This algorithm has $\log_2 P$ stages. At each stage, the odd-numbered blocks are used to "eliminate" the even-numbered blocks, with the process decreasing the number of blocks left by a factor of two at each stage. Similar to Phase 1, symmetry is maintained throughout, since the actual mathematical calculation is a Cholesky factorization of a symmetric permutation of the blocks of the reduced system. In this case, the blocks are ordered so that the even-numbered blocks in Step 1 are ordered first, the even-numbered blocks in Step 2 are numbered second, and so on. Such a reordering results in an elimination tree of minimal height over all reorderings ([Cle89]).

The implementation of this algorithm requires that additional space for fill-in be allocated, since the odd-even reordering creates fill-in in the reduced system (although this is of a much lower order than the fill-in created in Phase 1).

3.3 Phase 3

Phase 3 exists only in the solution of a linear system, whereas up to now we have discussed only the factorization. When one is solving a linear system, the operations are performed in Phase 1 and Phase 2 to the right-hand sides that mirror the factorization steps. We do not list these in detail.

At the end of Phase 2 of solving a linear system, each processor contains portions of the solution to the reduced system. Each processor then distributes 2β elements of this solution to neighboring processors to begin Phase 3. These partial solutions are easily backsubstituted into the locally stored factors in a completely local computation stage to finish the solution process. For brevity, we will not detail the steps in the triangular solution process. Suffice it to say that they have a similar but simpler structure than the factorization process. Block operations in LAPACK and the BLAS are used for the various operations. Multiple right-hand sides are as easily handled in this context as a single right-hand side, and thus our code addresses this more general case.

3.3.1 Experimental Results

Results from a typical parallel system are included in Figure 11. This figure shows computational speeds from running the code on the IBM SP2 parallel supercomputer located at the Cornell Theory Center, although results from other systems are qualitatively similar. The problem size has been scaled with the number of processors so that the submatrix stored on each processor is constant; that is, the bandwidth is fixed, but n scales as P. Two curves are given in this figure: one is the reciprocal of time and is the computational rate relative to the sequential operation count, while the other is the computational rate relative to the divide-and-conquer operation count.

The results fit very well with the predicted times when considering the details of the algorithm. The time for $P = 1$ reflects the speed of the underlying LAPACK banded factorization routine modulo minor additions for the parallel setup. The times for $P = 2$ and $P = 4$ show *increases*, reflecting the penalty of the fill-in, which causes an operation count of approximately four times that of the sequential algorithm. The actual megaflop rating shows an almost linear increase, with a deviation when going from $P = 1$ to $P = 2$ and $P = 4$ caused by the introduction of communication.

For $P > 4$, both graphs show almost linear performance. A closer look at the actual times shows that the time can easily be divided into two components: Phase 1 and the reduced system. The time for Phase 1 stays constant, since the work is essentially the same. The time for the reduced system increases gradually as $\log P$, and this slight increase causes the deviation from linearity in the performance graph.

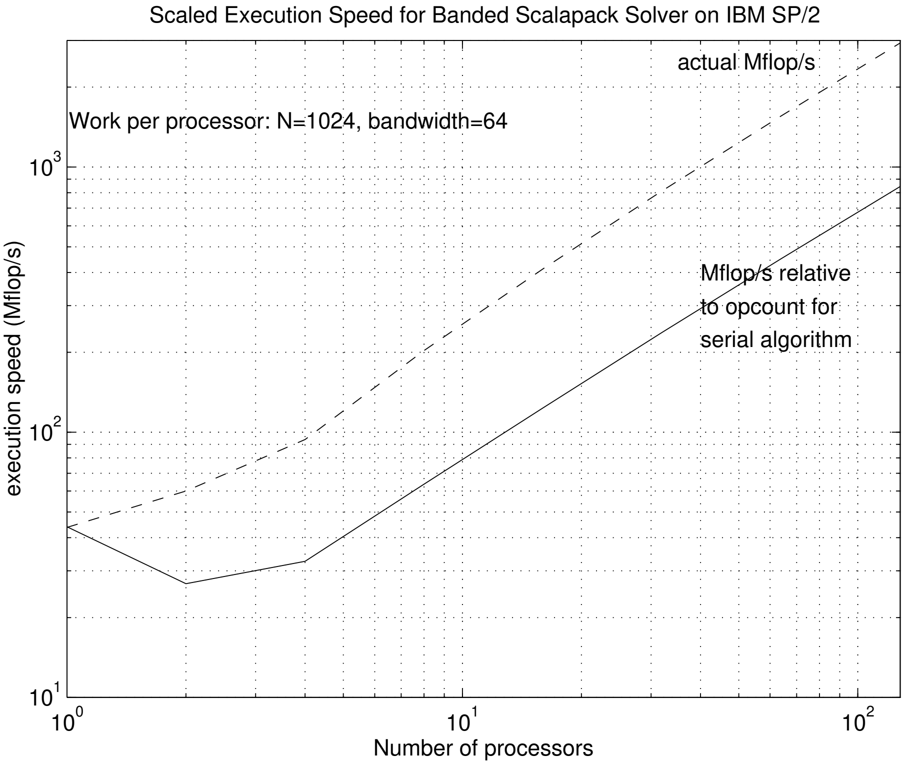

Figure 11: Scaled problem results for symmetric divide-and-conquer algorithm on IBM SP2

4 The Unsymmetric but Stable Case

An important class of banded matrices comprises those that are unsymmetric but that have numerical properties such that Gaussian elimination without pivoting is stable. Diagonally dominant matrices are the prototypical example in this class. ScaLAPACK provides special code to solve these unsymmetric matrices. This is an addition to the LAPACK standard, and as such, an extension to the subroutine naming scheme has been adopted. In addition to the two standard classes of banded matrices, PB for positive definite banded and GB for general banded, the letters DB indicate matrices and routines for handling them that are stable without interchanging (the mnemonic is Diagonally-dominant-like Banded).

The literature has several divide-and-conquer algorithms for unsymmetric matrices (see, for example, that of Lawrie and Sameh [LS84]). However, the majority of these have an unsymmetric aspect that has two practical drawbacks: it results in unnecessarily large and complicated reduced systems, and it complicates reusing the algorithmic structure of the symmetric positive definite codes.

Our approach is to have our algorithm mirror the symmetric code as much as possible by treating the transpose of the upper triangle of the matrix and factor in the same fashion as the lower triangle. This approach gives our algorithm a characteristic that the algorithm of Lawrie and Sameh (as well as others) does not: *it is equivalent to performing Gaussian elimination on a suitably reordered matrix*. Thus, we can recapture established properties of Gaussian elimination for implementation and analysis. For instance, if a symmetric positive definite matrix is input into the unsymmetric code, the Cholesky factors from the symmetric positive definite algorithm are reproduced (modulo diagonal scaling) by our unsymmetric code. Algorithms such as Lawrie and Sameh's do not share this property. Our approach also has the desirable practical consequence that code structure can be reused, allowing us to maintain the code for both cases in the same file, using source preprocessing techniques. We elaborate on this feature later.

We now give the algorithm in more detail. Figure 12 pictorially illustrates the user matrix of size n, lower bandwidth β_l, and upper bandwidth β_u, upon input. Consistent with the symmetric case, each processor stores a contiguous set of columns of the matrix, demarcated by the thicker lines in the figure. We partition each processor's matrix into blocks, as shown. The sizes of the A_i are given respectively by O_i. The sizes of the smaller matrices reflect the fact that two different bandwidths must be taken into account. A key algorithmic choice is that the separator matrices C_i are of size β_m, where $\beta_m = \max(\beta_l, \beta_u)$. This choice induces sizes for the matrices B_i, D_i of $\beta_m \times \beta_m$. Note that the last processor has only A_p.

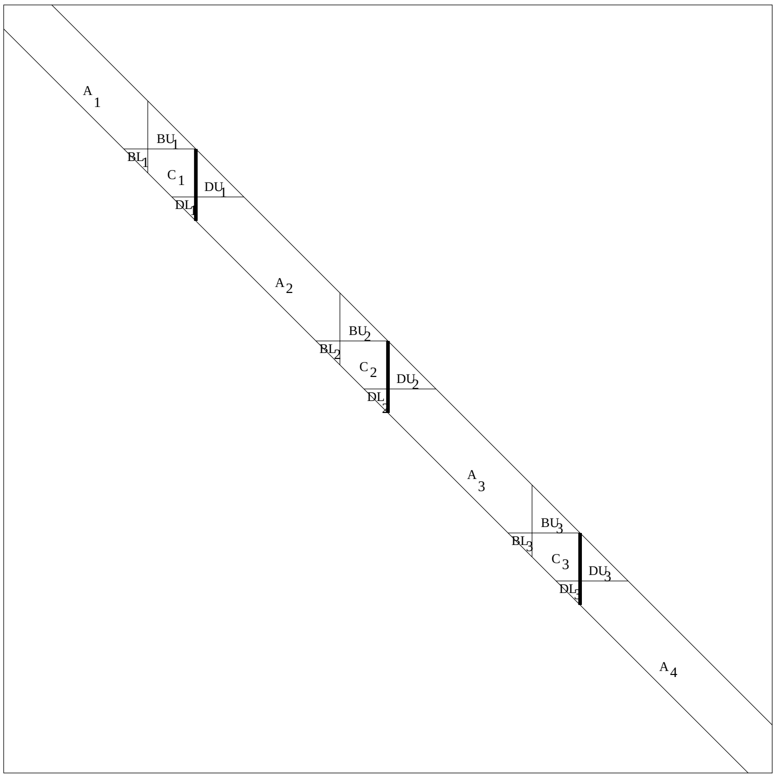

Figure 12: Divide-and-conquer partitioning of an unsymmetric matrix

The matrix is reordered by using the same algorithm as is used in the symmetric case, with β_m playing the role of β for the unsymmetric case. The resulting matrix is given in Figure 13. The matrix is then factored in the same stages as in the symmetric case, suitably adjusted.

4.1 Phase 1

As in the symmetric case, we concentrate on computations local to a processor, and for illustration we follow processor 2. After a preliminary communication stage in which DL_i is sent to processor $i+1$, each processor's storage is illustrated in Figure 15. The local computations are viewed as a frontal computation, with O_i factorization steps being taken and then applied to the remaining submatrix of size $2\beta_m$. This remaining part of the front is used in the formulation and solution of the reduced system. Ultimately, the local frontal calculations will result in the matrix labeled by

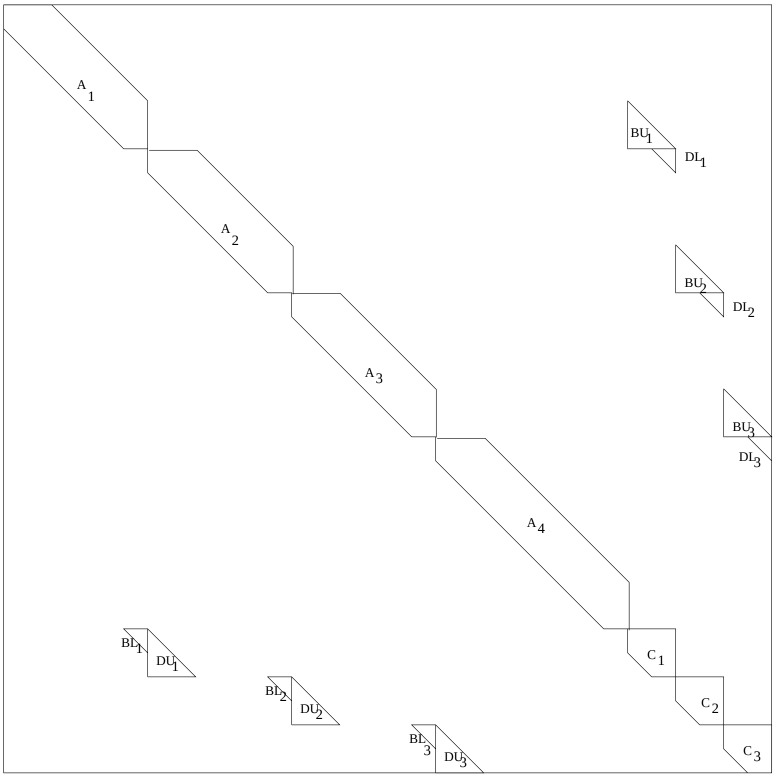

Figure 13: Matrix after divide-and-conquer reordering

Figure 14. We now illustrate the sequence of steps necessary to arrive at this state.

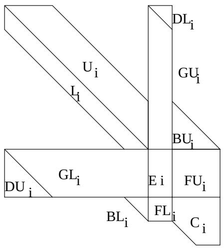

Figure 14: Factored local matrix

Each processor performs an LU factorization of A_i in parallel with the other processors. The factorizations are done without pivoting. Since LAPACK does not currently provide this functionality, we have submitted a routine with this functionality for future inclusion.

The major difference between our parallel algorithm and that of Lawrie and Sameh occurs at this point. Laurie and Sameh's algorithm applies both ${L_i}^{-1}$ and ${U_i}^{-1}$ (via triangular solution with L and U) to A *from the left side*. As we mentioned earlier, this is an asymmetric process, and it is this that causes the complicated structure of the reduced system. The ScaLAPACK algorithm applies ${L_i}^{-1}$ from the left, as is done in the symmetric code; however, we apply ${U_i}^{-1}$ *from the right*, in contrast to the algorithm of Lawrie and Sameh, as well as most others, though this is not the first paper to discuss the idea, see e.g. citeArbGan95.

Note that this has the desired effect on the A_i: once ${L_i}^{-1}$ has been applied, since $A_i = L_i U_i$,

$${L_i}^{-1} A_i = {L_i}^{-1} L_i U_i = U_i,$$

and thus multiplying by ${U_i}^{-1}$ on either side reduces the main blocks to the identity.

Computationally, ${L_i}^{-1}$ is applied from the left by solving the following systems:

$$L_i BU'_i = BU_i,$$

$$L_i GU_i = DL_i.$$

Analogously, ${U_i}^{-1}$ is applied from the right by solving

$$U_i {BL'_i}^T = {BL_i}^T,$$

$$U_i {GL_i}^T = {DU_i}^T.$$

At this point, the factors stored on each processor have been computed. They now must be used to form the reduced system. The local matrix after these steps is illustrated in Figure 15.

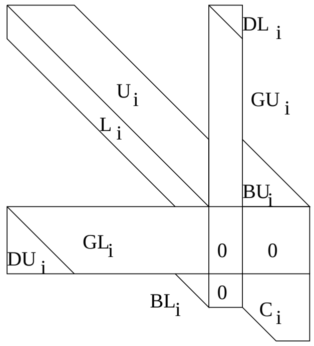

Figure 15: Local matrix on processor 2 after initial communication of DL_i

The matrices ${B(L,U)'_i}^T$ are used to modify C_i according to

$$C'_i = C_i - BL'_i BU'_i,$$

where all matrices in the formula are interpreted as being $\beta_m \times \beta_m$ (the code uses the minimal sizes possible which are smaller in the general case in which $\beta_l \neq \beta_u$).

Each processor i computes a modification to C_{i-1} via

$$E_i = GL_i GU_i.$$

Here, E_i is either $\beta_m \times \beta_l$ if $\beta_m = \beta_u$, or $\beta_u \times \beta_m$ otherwise.

Finally, the two off-diagonal blocks FL_i and FU_i are computed:

$$FL_i^T = HU_i^T BL'_i,$$

$$FU_i = HL_i {BU'_i}^T.$$

The matrices FL_i and FU_i are of size $\beta_l \times \beta_l$ and $\beta_u \times \beta_u$, respectively.

4.2 Phase 2

Phase 2 is the forming and factorization of the reduced system. Like the symmetric case, the reduced system is block tridiagonal and of size $P - 1$ blocks. The diagonal blocks have the same bandwidth as A, and the off-diagonal blocks are of size $\beta_m \times \beta_m$ and have nonzero structure determined by the two bandwidths.

An odd-even block algorithm with the exact same structure as that in the symmetric code is used, although the code is modified to take into account the asymmetry. Each processor has one diagonal block and two off-diagonal blocks, as well as a contribution to the previous processor's diagonal block. Extreme care must be taken in the code, however, to operate on blocks of the correct size, since reordering and the mismatched bandwidths create blocks of different sizes and orientations.

4.3 Phase 3

Phase 3 has the exact same structure as in the symmetric case, where again the code has been modified to reflect the unsymmetric matrix and the slightly varying blocksizes. Pieces of the solution are communicated in the reverse of the factorization structure, and then a purely local backsubstitution finishes the solution process.

4.4 Experimental Results

Figure 16 presents results for this code on the IBM SP/2, similar to the result for the symmetric positive definite code shown in Figure 11. As predicted based on the fact that both codes use the same basic algorithm, the results for the nonsymmetric code have the same qualitative behavior as those for the symmetric code. Actual computing rates are higher for the nonsymmetric code because the underlying sequential kernels run faster for nonsymmetric matrices than for symmetric matrices. The dropoff in performance as the number of processors is slightly worse for the nonsymmetric case, partially because the on-processor computing speed is higher since that increases the communication to computation ratio, and partially because the communication in the reduced system phase is doubled.

5 Tridiagonal Matrices

Tridiagonal matrices form a very important practical subset of banded matrices, since many applications involve tridiagonal matrices. At the simplest, banded codes may be used to solve tridiagonal matrices by setting $\beta_l = \beta_u = 1$, but this approach is inefficient because subroutines are called in the banded codes to perform operations that in the tridiagonal case are a single operation. A better approach is to use the same algorithms we have described for banded matrices, but to specialize the code implementation to tridiagonal matrices. An easily maintainable way to do this is via source preprocessing that replaces the block operations in the banded code with the appropriate operations for tridiagonal matrices, leaving the algorithmic sections of the code unchanged. This is the strategy we have used in ScaLAPACK.

As mentioned earlier, for very small bandwidths, the reduced system is solved relatively efficiently by a sequential-type algorithm, since such an algorithm trades off parallelism in the computation for fewer communication startups, and the computation is almost trivial for small bandwidths. Thus, an option for implementation is to keep the bulk of the algorithm the same as for banded matrices, but to solve the reduced system on a single processor. For a number of processors below a cutoff number that is dependent on many performance factors, the sequential algorithm will outperform the parallel algorithm. However, to ensure scalability, a parallel algorithm must be used for a system with more processors than this cutoff. While dynamically choosing a reduced system algorithm

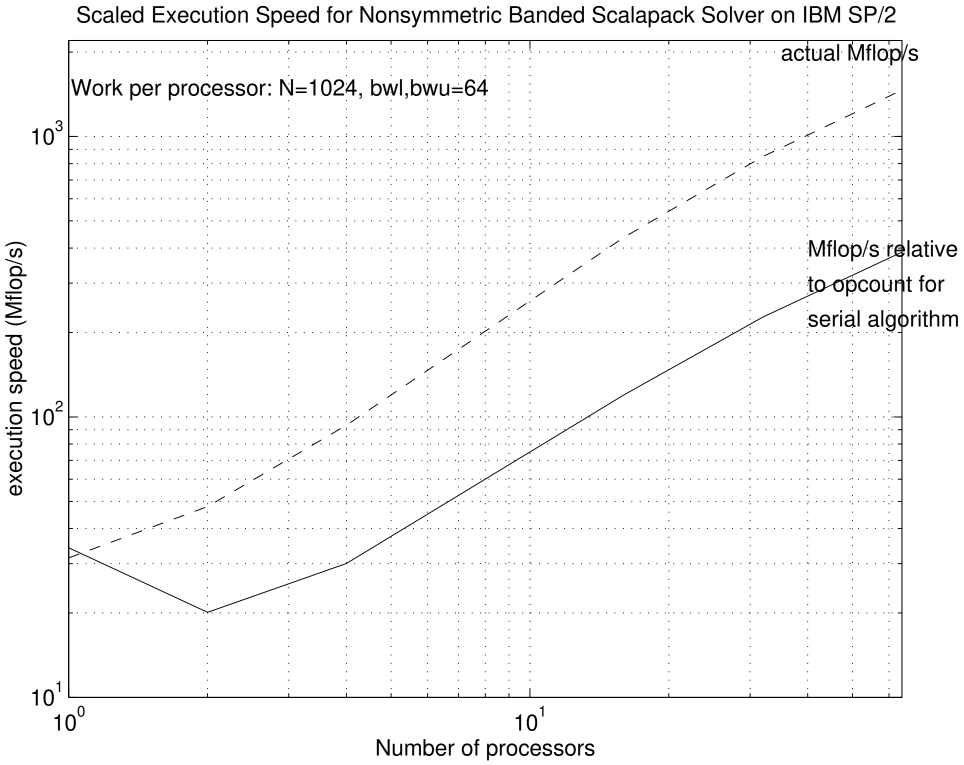

Figure 16: Scaled problem results for diagonally dominant divide-and-conquer algorithm on IBM SP2

based on various parameters is possible, we have opted to retain the parallel algorithm to guarantee scalability for large systems.

6 Development and Maintenance Using Source Preprocessing

Although the different combinations of matrix bands, matrix types, precisions (four precisions are supported in ScaLAPACK), and parameters (such as the UPLO parameter in the symmetric positive definite codes and the TRANS parameter in the triangular solve codes) lead to a large quantity of individual subroutines, the development above shows that mathematically they are very similar. We have used this similarity to manage the development of all of these combinations in as uniform a way as possible using source preprocessing techniques. Using a macro substitution package similar to the C preprocessor, we maintain the bulk of the computational code in two meta-source files, one for factorization and another for triangular system solution. Each of these is in turn divided into two subfiles, one for Phase 1 and a second for the reduced system, or Phase 2. This strategy greatly facilitates maintenance, in the sense that if an algorithmic improvement or a bug fix is made to one code, it is simultaneously made to all of them. The meta-source file developed this way is only fractionally larger than the individual source files created from it.

7 Summary

Release 1.2 of ScaLAPACK included the first codes in ScaLAPACK that address banded linear systems: the symmetric positive definite codes. Release 1.3 in November 1996 included three more categories of software: the diagonally dominant unsymmetric banded codes, and the tridiagonal codes for both symmetric positive definite matrices and diagonally dominant unsymmetric matrices. All of this software has been discussed in this article.

In the near future we will release code based on new algorithms from the same family of algorithms discussed here for the difficult problem of general nonsymmetric matrices, incorporating partial pivoting at all stages of the factorization and maintaining a parallel reduced system solution. A followup to this article will detail the new algorithm and its implementation.

This article has shown the family of algorithms used to solve these systems in considerable detail, giving not only the mathematical algorithm but also the implementational details. Performance characteristics were predicted based on the general structure of the algorithm family, and these predictions were confirmed by experimental results.

These codes add considerable capability to the already comprehensive ScaLAPACK package and fill the largest remaining hole in the functionality of the package compared to LAPACK, the community standard sequential linear algebra package. We anticipate that these codes, like the rest of the ScaLAPACK package, will be adopted in parallel application codes. They have already been included in commercially available software libraries such as IBM's PESSL and NAG's Numerical PVM Library.

To retrieve the software described in this document, point your web browser at a netlib repository site and follow the links to ScaLAPACK. For instance,

`http://www.netlib.org/scalapack/index.html`

8 Acknowledgments

The authors thank R. Clint Whaley, L. Susan Blackford, and Antoine Petitet for helpful discussions and support with various aspects of the software development.

References

[ABB+95] E. Anderson, Z. Bai, C. H. Bischof, J. Demmel, J. J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. C. Sorensen. *LAPACK Users' Guide*. SIAM, 2nd edition, 1995. (Also available in Japanese, published by Maruzen, Tokyo, translated by Dr Oguni).

[AG95] P. Arbenz and W. Gander. A survey of direct parallel algorithms for banded linear systems. Technical report, Swiss Federal Institute of Technology, Zurich, Switzerland, 1995.

[BCC+97] S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. Whaley. Scalapack: A linear algebra library for message-passing computers. In *Proceedings of 1997 SIAM Conference on Parallel Processing*, May 1997.

[BCD$^+$94] R. Brent, A. Cleary, M. Dow, M. Hegland, J. Jenkinson, Z. Leyk, M. Nakanishi, M. Osborne, P. Price, S. Roberts, and D. Singleton. Implementation and performance of scalable scientific library subroutines on Fujitsu's VPP500 parallel-vector supercomputer. In *Proceedings of the 1994 Scalable High Performance Computing Conference*, 1994.

[CDPW93] J. Choi, J. Dongarra, R. Pozo, and D. Walker. ScaLAPACK: A scalable linear algebra library for distributed memory concurrent computers. Technical Report 53, LAPACK Working Note, 1993.

[Cle89] A. Cleary. *Algorithms for Solving Narrowly Banded Linear Systems on Parallel Computers by Direct Methods.* PhD thesis, The University of Virginia, Department of Applied Mathematics, 1989.

[DDHH84] J. Dongarra, J. DuCroz, S. Hammarling, and R. Hanson. A proposal for an extended set of Fortran basic linear algebra subprograms. Technical Memo 41, Mathematics and Computer Science Division, Argonne National Laboratory, December 1984.

[DJ87] J. Dongarra and L. Johnsson. Solving banded systems on a parallel processor. *Parallel Computing*, 5:219–246, 1987.

[GGJT96] A. Gupta, F. Gustavson, M. Joshi, and S. Toledo. The design, implementation, and evaluation of a banded linear solver for distributed-memory parallel computers. Research Report RC 20481, IBM, june 1996.

[Joh87] L. Johnsson. Solving tridiagonal systems on ensemble architectures. *SIAM J. Sci. Statist. Comput.*, 8:354–392, 1987.

[LS84] D. Lawrie and A. Sameh. The computation and communication complexity of a parallel banded system solver. *ACM Trans. Math. Softw.*, 10:185–195, 1984.

[Wri91] S. Wright. Parallel algorithms for banded linear systems. *SIAM J. Sci. Stat. Comput.*, 12(4):824–843, 1991.