Designing a BSP version of ScaLAPACK*

Guy Horvitz† Rob H. Bisseling‡

Abstract

The ScaLAPACK library for parallel dense matrix computations is built on top of the BLACS communications layer. In this work, we investigate the use of BSPlib as the basis for a communications layer. We examine the LU decomposition from ScaLAPACK and develop a bulk synchronous parallel (BSP) version. For small problems, where communication dominates, the BSP version is about 10% faster compared to the native BLACS version and 50% compared to the MPI BLACS version. For large problems, where computation dominates, the differences are less pronounced, but the BSP version is still slightly faster. We present the main features of a new library, BSP2D, which we propose to develop for porting the whole of ScaLAPACK.

1 Introduction

To obtain the highest performance in parallel computation both computation and communication must be optimised. LAPACK [1] has provided us with highly optimised implementations of state-of-the-art algorithms in the field of numerical linear algebra, in particular for the solution of dense linear systems and eigensystems. Many years of effort have gone into optimising LAPACK, and much of its success is due to the encapsulation of system-dependent optimisations into the Basic Linear Algebra Subprograms (BLAS). LAPACK is available for sequential computers, vector supercomputers, and parallel computers with shared memory.

The ScaLAPACK [3] project aims to provide a scalable version of LAPACK for parallel computers with distributed memory. Portability is ensured by building ScaLAPACK on top of the Basic Linear Algebra Communication Subprograms (BLACS). The parallel efficiency depends critically on the communication performance achieved by this library and thus it is natural to ask whether the performance can be further improved.

The bulk synchronous parallel (BSP) model [9] views a parallel algorithm as a sequence of supersteps, each containing computation and/or communication, followed by a global synchronisation of all the processors. This imposes a discipline on the user, thus making parallel programming simpler, but it also provides possibilities for system-level optimisation such as combining and rescheduling of messages. This can be done because the superstep provides a natural context for communication optimisation by the system; the user need not be concerned about this.

*This work was supported in part by NCF, which provided computer time on the Cray T3E of the High Performance Applied Computing centre at the Technical University of Delft.

†Fritz Haber Research Center for Molecular Dynamics, Hebrew University, Jerusalem 91904, Israel, guyh@fh.huji.ac.il

‡Department of Mathematics, Utrecht University, P.O. Box 80010, 3508 TA Utrecht, The Netherlands, Rob.Bisseling@math.uu.nl

A BSP computer can be characterised by four global parameters: p, the number of processors; s, the computing speed in flop/s; g, the communication time per data element sent or received, measured in flop time units; and l, the synchronisation time, also measured in flops. Algorithms can be analysed by using the parameters p, g, and l; the parameter s just scales the time. The time of a superstep with both computation and communication is $w + hg + l$, where w denotes the maximum amount of work (in flops) of a processor, and h is the maximum number of data elements sent or received by a processor. The total execution time of an algorithm (in flops) can be obtained by adding the times of the separate supersteps. This yields an expression of the form $a + bg + cl$. In the following presentation, we consider the architecture as an abstract BSP computer, and therefore we use the term 'processes' instead of 'processors'. In our experiments, only one process executes on each processor, so these terms may be used interchangeably.

BSPlib [7] is a proposed standard which makes it possible to program directly in BSP style. BSPlib is an alternative to PVM [8] and MPI [5]. It provides both direct remote memory access (i.e., one-sided communications such as put and get), and bulk synchronous message passing.

BSPPACK [2] is a prototype application package built on top of BSPlib. It is a research and educational library which contains parallel implementations of algorithms for sparse and dense linear system solution, fast Fourier transforms, and other scientific computations.

The aim of the present work is to answer the question: can ScaLAPACK be ported to BSPlib and does this improve performance? This may indeed be the case, because we expect ScaLAPACK to benefit from ideas developed within the context of BSPPACK and from the excellent implementation of BSPlib available as the Oxford BSP toolset [6]. Here, we limit ourselves to investigating the ScaLAPACK LU decomposition subroutine `PSGETRF`.

The design philosophy of ScaLAPACK is to use a hierarchy of software layers. The top of the pyramid is ScaLAPACK itself, which calls the Parallel BLAS (PBLAS). The PBLAS use the BLAS for single-process linear algebra computations and the BLACS for communication. The BLACS can be built on top of a basic communications layer such as MPI or PVM. The BLACS perform communication at a higher level: they send complete matrices of all types and they allow us to view the processes as a two-dimensional grid and to perform operations within the scope of a process row or column, or the complete grid.

The data distribution of ScaLAPACK is the two-dimensional block-cyclic distribution with a user determined block size nb. Another parameter is the algorithmic block size nb'. The algorithms in the sequential package LAPACK handle complete blocks of size nb'. ScaLAPACK structures its algorithms in the same way, but it imposes $nb' = nb$. We make the same choice for reasons of convenience, but in our case it is straightforward to relax this constraint to $nb' \geq nb$; we shall discuss this later.

Since the communication in ScaLAPACK is isolated in the BLACS it would be the most natural choice to construct a BLACS version based on BSPlib. A straightforward BSPlib implementation of the BLACS, however, would be impossible for different reasons; one important reason is the following. The BLACS include pair-wise message passing for communication where the receiver has to wait for the data to arrive in order to continue. The sender can continue as soon as the message is sent off. In BSPlib, a message transfer is completed only after the next global synchronisation. Suppose there is exactly one message to be communicated and hence in the program there is one call to a BLACS send and one to a BLACS receive. The processes that do not send or receive are not aware of this communication and hence do not synchronise, thus violating the principle of global synchronisation.

Forcing the user to synchronise globally between a send and a receive requires drastic changes in both the syntax and the semantics of the BLACS subroutines. This would turn the BLACS into a different library, which could be called BSP2D; section 4 outlines how such a library could be constructed in the future. The present work simply removes the BLACS and adapts ScaLAPACK and the PBLAS using direct calls to BSPlib. This alone is not sufficient: it is also necessary to restructure ScaLAPACK and the PBLAS on the basis of supersteps.

2 BSP version of ScaLAPACK LU decomposition

Programming in BSPlib requires global synchronisation. For this reason, every process should know when a global synchronisation is needed to perform a certain task. Sometimes, a process also needs to know about resources (such as buffers) provided by remote processes. Such knowledge can be transferred by communication, but this would be inefficient.

Another approach would be to let all the processes call subroutines together and with the same values for the scalar input parameters. This way, each process can deduce the behaviour of the other processes. We adopted this approach for the PBLAS. For example, consider the PBLAS subroutine `PSSWAP` which swaps two rows or columns of distributed matrices. If the swap is local and no communication is needed, the processes do not synchronise. Otherwise, all the processes perform one synchronisation, even if they do not hold any of the related data and do not actively participate in the operation. All processes can distinguish between the two situations, because they have the necessary information.

2.1 Unblocked LU decomposition and pivot search subroutines

An example of how a ScaLAPACK subroutine and a PBLAS should be altered, is shown in the case of the ScaLAPACK subroutine `PSGETF2`, which performs an unblocked parallel LU decomposition on a block of consecutive columns; it is called by the main LU decomposition subroutine `PSGETRF`. The main part of the `PSGETF2` code is given in Fig. 1.

In the original subroutine, the main loop (`DO 10` ... `10 CONTINUE`) is executed only by the process column `IACOL` that holds the block to be decomposed. After the decomposition, the pivot indices `IPIV(IIA..IIA+MN-1)` of that block are broadcast to the other process columns by the sending subroutine `IGEBS2D` and the receiving subroutine `IGEBR2D`. This structure is inherited from the PBLAS. Since the PBLAS subroutine `PSAMAX`, which finds the pivot of matrix column `J`, returns the result only to the processes of the process column `IACOL` that holds `J`, the other processes cannot evaluate the singularity test `GMAX.NE.ZERO`.

We mentioned earlier that BSP based PBLAS should be called by all processes with the same values for the scalar input parameters. The example of `PSGETF2` makes it clear that scalar output parameters must be returned to all processes too. This way, `GMAX` and the pivot index become available to all the processes so they can participate in the main loop, and can call subsequent PBLAS together, as required. Inevitably, sending the output scalars to all processes costs extra communication and synchronisation time.

An advantage of the changes in `PSAMAX` and `PSGETF2` is the ability to choose an algorithmic block size that differs from the distribution block size. This is impossible in the current version of ScaLAPACK; e.g. if $nb' = 2nb$, then two process columns should participate in the decomposition of one algorithmic block of columns. The subroutine `PSAMAX`, however, returns its results only to the process column that holds matrix column `J`.

```
      IF( MYCOL.EQ.IACOL ) THEN                                         DEL
         DO 10 J = JA, JA+MN-1
            I = IA + J - JA
*
*           Find pivot and test for singularity.
            CALL PSAMAX( M-J+JA, GMAX, IPIV( IIA+J-JA ), A, I, J,
     $                   DESCA, 1 )
            IF( GMAX.NE.ZERO ) THEN
*
*              Apply the row interchanges to columns JA:JA+N-1
               CALL PSSWAP( N, A, I, JA, DESCA, DESCA( M_ ), A,
     $                      IPIV( IIA+J-JA ), JA, DESCA, DESCA( M_ ) )
*
*              Compute elements I+1:IA+M-1 of J-th column.
               IF( J-JA+1.LT.M )
     $            CALL PSSCAL( M-J+JA-1, ONE / GMAX, A, I+1, J,
     $                         DESCA, 1 )
            ELSE IF( INFO.EQ.0 ) THEN
               INFO = J - JA + 1
            END IF
*
*           Update trailing submatrix
            IF( J-JA+1.LT.MN ) THEN
               CALL PSGER( M-J+JA-1, N-J+JA-1, -ONE, A, I+1, J, DESCA,
     $                     1, A, I, J+1, DESCA, DESCA( M_ ), A, I+1,
     $                     J+1, DESCA )
            END IF
   10    CONTINUE
*
         CALL IGEBS2D( ICTXT, 'Rowwise', ROWBTOP, MN, 1, IPIV( IIA ),   DEL
     $                 MN )                                             DEL
      ELSE                                                              DEL
         CALL IGEBR2D( ICTXT, 'Rowwise', ROWBTOP, MN, 1, IPIV( IIA ),   DEL
     $                 MN, MYROW, IACOL )                               DEL
      END IF                                                            DEL
```

FIG. 1. *Main part of* **PSGETF2** *source code. Lines marked by* **DEL** *are deleted in the BSP version.*

2.2 Collective communication subroutines

Sometimes, we need subroutines to perform collective communications such as broadcasts or reductions. In our case, we need to broadcast data within a process row (or column), and perform this operation for all process rows simultaneously. The method adopted for the PBLAS, global replication of scalar parameters, is not suitable here. The reason is that the size of the broadcast may differ between the process rows. We must allow different sizes, but the number of synchronisations should not depend on them.

The simplest solution is always to use a broadcast with two synchronisations, except when the broadcast is in the scope of one or two processes. For one process no synchronisation is needed, and for two processes a single synchronisation suffices. All processes can take the same decision because the number of participants in the broadcast is the same and known to all of them. The choice of performing two synchronisations in the general case is based on the efficiency of the so-called *two-phase broadcast*, which first scatters the elements of a data vector across all the processes, and then lets each process broadcast the data it received. This was shown to be efficient in the LU decomposition program from BSPPACK, see [2].

2.3 Multiple row swap subroutine

The ScaLAPACK subroutine `PSLASWP` applies a series of row exchanges in a matrix prescribed by a given vector of pivoting indices. This is originally done by pairwise row swaps, each time using the PBLAS subroutine `PSSWAP`. A direct translation into BSP would imply one superstep for each swap. We change the method so that all the swaps are done in one superstep, in good BSP style. The changes are as follows.

First we translate the representation of the permutation from swaps into cycles. For example, suppose the swap vector is (4, 10) (3, 10) (2, 10) (1, 10), which means: first swap rows 1 and 10, then 2 and 10, etc. In this example, rows 1, 2, 3, 4 are on the same process A and row 10 resides on a different process B. The cycle representation of this permutation is (10, 4, 3, 2, 1), which means: 1 goes to 2, 2 goes to 3, ..., 10 goes to 1. The operations performed by A and B in this case are:

Process A	**Process B**
Put row 4 in buffer on process B	Put row 10 in buffer on process A
For $i = 3$ to 1 step -1	
copy row i into row $i+1$	
Sync	Sync
Copy buffer into row 1	Copy buffer into row 10

In this way, only one row is exchanged between A and B. In the original algorithm, which performs the swaps sequentially, four rows are exchanged. In the general case, the different cycles are handled separately, but with one global synchronisation for all of them.

2.4 Registered buffers

Often, we have to communicate noncontiguous data like e.g. a matrix row, which in ScaLAPACK is stored as a strided subarray. The data elements can of course be sent separately, but even though BSPlib automatically combines small messages, there is still a notable overhead for extremely small messages such as single words. If the access pattern is based on a stride, the overhead can be avoided by packing messages in buffers.

Put operations are the most efficient means of communication on many architectures, including our test machine. When we use puts for communications, the locations of the buffers in which we put the packed data must have been registered previously. The purpose of registration is to link the name of a remote variable with its local address; this enables putting into dynamically allocated memory. Since registration incurs communication and synchronisation costs, it is more efficient to register the locations only once, at the beginning of the computation. The locations should then be passed to the PBLAS.

For this purpose, we implemented a management system for registered buffers. At the start of the program, we allocate and register buffers of appropriate sizes. When a PBLAS requests a buffer of a certain size, it calls a subroutine which returns a pointer to the smallest buffer of at least the requested size. Similar to the registration procedure of BSPlib, buffers are requested in lock step. All processes participate in all requests, and they ask for a buffer of the same size.

To achieve the ultimate in efficiency, we use the high performance put primitive `bsp_hpput` which is unbuffered on source and unbuffered on destination, instead of `bsp_put`

TABLE 1

Computing rate in Gflop/s of LU decomposition on a CRAY T3E for three different communication layers. The process grid has size 8×8; the block size is 32.

Size	BSPlib	native BLACS	MPI BLACS
500	0.37	0.33	0.22
1000	1.22	1.11	0.81
2000	3.19	3.04	2.42
3000	5.34	4.96	4.15
4000	6.79	6.57	5.61
6000	9.63	9.24	8.56
8000	10.47	10.55	10.04
10000	12.73	12.48	12.20

which is doubly buffered. In the case of the high performance primitives, responsibility for buffering rests on the user instead of on the BSPlib system. On our test machine, we found that the improvement in performance was significant.

3 Experimental results

We performed numerical experiments on a CRAY T3E computer with $p = 64$ processors, each with a theoretical peak performance of 600 Mflop/s. We measured a sequential speed of $s = 250$ Mflop/s for the matrix multiplication part of the LU decomposition. Normalised for this value of s, we found $g \approx 14$ and $l \approx$16000–80000. (We measured these values within the context of the program, not in a separate benchmark. This explains the variation in l.) The aim was to compare the ScaLAPACK performance of three communication layers: BSPlib, a Cray-specific native version of the BLACS, and an MPI version. We ran tests for three different process grids (with size 8×8, 16×4, 4×16) and four different block sizes ($nb = 16$, 32, 48, 64). The optimal grid size for all three communication layers was 8×8, and the optimal block size was 32. We used single precision, which is 64 bits on this machine. We ran a test program which generates a square matrix with random elements.

The measured computing rate is given in Table 1. The rate is based on the overall execution time, including all overheads. For small problems, where communication dominates, the table shows a considerable gain in speed obtained by using BSPlib: about 10% compared to the native BLACS and 50% compared to the MPI BLACS, for $n = 1000$. For large problems, where computation dominates, the differences are less pronounced: about 2% compared to the native BLACS and 4% compared to the MPI BLACS, for $n = 10000$. The BSPlib version is faster than the others, except for $n = 8000$.

To understand the savings in execution time, we measured the time used by each part of the program. Using BSPlib we can measure the communication/synchronisation time separately from the local computation time. We then separated the communication time from the synchronisation time by using a BSP-provided estimate of the synchronisation time. We also measured the packing time, which includes the time spent in packing and unpacking data, and in local swaps. Finally, we could estimate the idle time of each process, which we define as the average time a process waits for the others to finish their computation. The resulting breakdown of the costs is presented in Fig. 2. As expected,

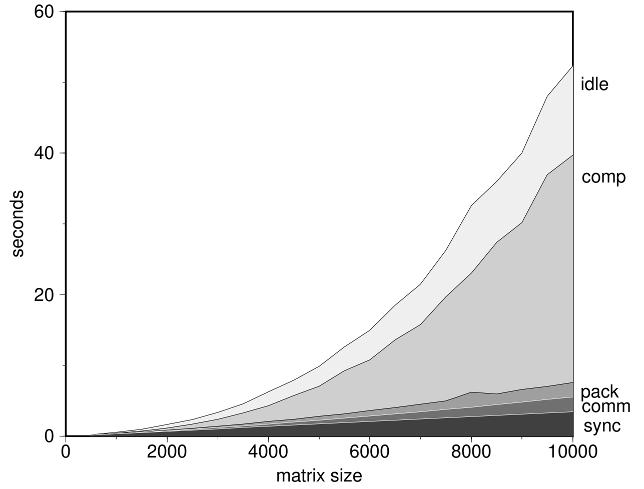

FIG. 2. *Breakdown of the total execution time for BSP based LU decomposition. The components are: synchronisation, communication, packing, computation, and idling. The process grid has size* 8×8*; the block size is 32.*

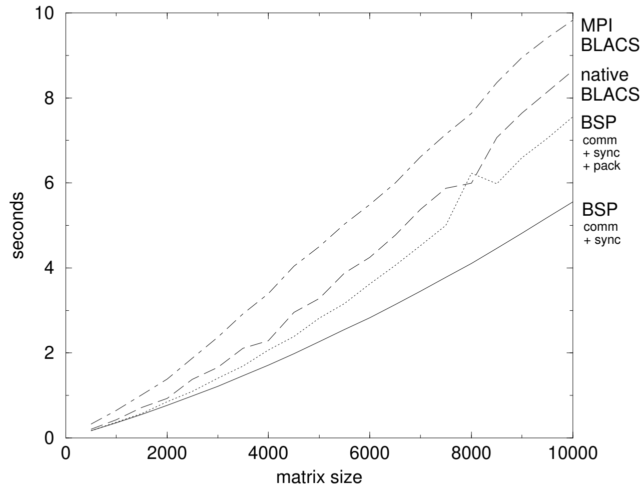

FIG. 3. *Communication time (including packing and synchronisation) during LU decomposition for three communication layers: BSPlib, native BLACS and MPI BLACS. For BSPlib, the time without packing is also given. The process grid has size* 8×8*; the block size is 32.*

the computation time of $2n^3/3p$ flops dominates for large n. Note that the synchronisation time, although only linear in n, is still significant compared to the computation time.

The computation and idling time is identical for all three versions, because they differ only in the communication part. By subtracting the computation and idling time from the measured total time, we can obtain the time of the communication part, which is presented in Fig. 3. It is clear that the communication time for BSP is significantly less than for the

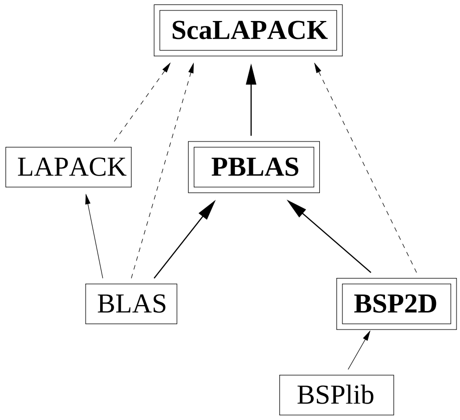

FIG. 4. *Hierarchical view of a possible BSP based ScaLAPACK, adapted from [4]. The double boxes contain the parts affected by moving to BSP. Solid arrows represent major dependencies and dashed ones minor dependencies. The solid bold arrows represent the main structure of ScaLAPACK.*

other two versions. For large n, the typical savings compared to the native BLACS are 10–15%. The exception is again the case $n = 8000$, for which the native BLACS are slightly faster. We found that the cause of this exception is the poor performance on the Cray T3E of the vendor-provided BLAS `SCOPY` and `SSWAP`, which we used for packing and local swaps in `PSSWAP` and `PSLASWP`. They reduced the performance of our program considerably, and more than that of the native and MPI BLACS versions. Improving the copying would reduce the communication time for BSPlib to that depicted in the lower line of Fig. 3.

4 Proposal for a BSP2D library

When developing a BSP implementation of the whole ScaLAPACK, it would be most convenient to have available a high level BSP based communication layer, called BSP2D. This would save much effort and would also improve modularity. The position of the BSP2D layer in the ScaLAPACK hierarchy is shown in Fig. 4. BSP2D has the functionality of the BLACS, i.e., communicating complete matrices (or vectors) of different types. Like the BLACS it views the processes as a two-dimensional grid. It can be built on top of BSPlib or another suitable BSP library.

There are two types of communication operation in BSP2D: pair-wise communications and collective communications. Pair-wise communications should be done by bulk synchronous message passing (using `bsp_send`) and not by direct remote memory access (using `bsp_put`, `bsp_get`, or their high performance equivalents).

Direct remote memory access cannot be used for the following reason. The communication of noncontiguous data structures involves packing of the data in buffers. Communicating by direct remote memory access requires previous registration of these buffers. Since the size of data each process sends is not always known to the other processes we cannot use the global management system for registered buffers described in Subsection 2.4. (For a general library such as BSP2D we cannot adopt the same solution as for the PBLAS, namely calling each subroutine with the same global parameters. This would render the library hard to use.) An alternative would be to register a buffer for each put operation, but this would be inefficient. A third possibility would be to use static pre-registered buffers, where each process makes $p-1$ buffers available for use by the other processes; this wastes too much memory. Therefore, none of these methods is satisfactory.

As a consequence, pair-wise communication should be done by bulk synchronous message passing. This means that data are sent, and after global synchronisation the destination process moves the data from its receive queue. Messages consist of a payload and a tag. The payload contains the matrix to be communicated, packed in a suitable form. The tag consists of type information, the identity of the sending process, and the number of messages that were already sent by that process to the receiving process in the current superstep. This number represents the order in which the send operations occur in the program text, and not the actual order in which BSPlib sends them. BSPlib is still allowed to optimise communication by rescheduling messages within a superstep. (This is the main advantage over traditional message passing.)

In BSP2D, messages originating in the same process must be moved in the order those messages were sent; this is similar to the requirement for receives in the BLACS. The messages of the receive queue of BSPlib, however, are in arbitrary order and the queue can only be accessed in this order. Still, this poses no problem since the high performance move operation `bsp_hpmove` can be used to create a list of the message positions in the queue. This operation is done as part of the BSP2D synchronisation subroutine. In an implementation, the list can be sorted in linear time by source process and message number. The use of `bsp_hpmove` instead of `bsp_move` also enables BSP2D to unpack data straight from the receive buffer, thus saving the time of local copying. (Performance could be improved even more if a high performance send `bsp_hpsend` were available, so the data could be sent straight from the source memory.)

Collective communications such as broadcasts and reductions involve synchronisation, so they should be called by all processes at the same time. They can be performed in the scope of a process row, a process column, or the whole process grid. To ensure that all the processes perform the same number of synchronisations, these subroutines always have two supersteps, except when the number of processes in the scope is one or two. As already observed in our study of LU decomposition, the decision on the number of synchronisations cannot rely on the number of data to be communicated, since it may vary between different process rows or columns.

We already described the two-phase broadcast in Subsection 2.2. Two-phase reduction is similar. Suppose the scope of the operation has q processes. In the reduction, each process has a vector of the same size n. Associative and commutative component-wise operations such as additions have to be performed on these q vectors. This is done as follows. The data on each process are divided into q blocks of size n/q, numbered $0, \ldots, q-1$, and each block is sent to a different process, so that process i gets all the blocks numbered i. Then each process performs a local reduction of the blocks, and sends the result to all the other processes. The total communication/synchronisation cost is about $2ng + 2l$.

In summary, BSP2D will include subroutines for pair-wise and collective communications, for global synchronisation with additional housekeeping, for the creation, initialisation, and destruction of the process grid, and for retrieving the grid dimensions and process coordinates.

5 Conclusions and future work

In this work, we have demonstrated that it is feasible to produce, with a relatively minor effort, an efficient bulk synchronous parallel version of an important ScaLAPACK subroutine. We expect that the same can be done for most subroutines from ScaLAPACK. The BSP version outperforms two other versions, one based on a vendor-built BLACS communication layer, and the other on MPI BLACS. The performance gains were entirely due to a reduction of the communication time; the computation part was left unchanged.

For large problems, e.g. $n = 10000$, communication time was reduced by up to 15% compared to the vendor-built BLACS version and even more compared to the MPI version. Because our test machine has relatively fast communications the reduction in total execution time is less pronounced. For machines with slower communication relative to computation, the influence of communication on the total time will be larger, and hence the gain we expect to achieve by using BSPlib would be proportionally larger. Of course, in future work this prediction should be tested in practice. For small problems, communication is dominant and the savings in total time are considerable. These results demonstrate that a public-domain software layer such as BSPlib can outperform a vendor-supplied layer. (We would expect a vendor-supplied version of BSPlib to improve performance even more.)

Our practical experience in porting one major ScaLAPACK subroutine led to the formulation of the BSP2D library. Whereas we could build one single routine (and the required PBLAS) directly on top of BSPlib and we could manage the registered buffers within the subroutine, this would not be a feasible solution for the whole of ScaLAPACK. Instead, using an intermediate BSP2D layer would increase modularity and software reuse, at only a slight increase in cost due to copying and other overheads.

The approach of BSPlib, based on global synchronisation, can be carried over to the PBLAS, and this gives the additional advantage that the algorithmic and distribution block sizes can be decoupled. This enables a better trade-off between load balance, speed of BLAS operations in the unblocked part of the algorithm, and speed in the blocked part, thus providing further opportunities for improving the performance of ScaLAPACK.

References

[1] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen, *LAPACK Users' Guide, Release 2.0*, SIAM, Philadelphia, PA, second ed., 1995.

[2] R. H. Bisseling, *Basic techniques for numerical linear algebra on bulk synchronous parallel computers*, in vol. 1196 of Lecture Notes in Computer Science, Springer-Verlag, Berlin, 1997, pp. 46–57.

[3] L. 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. C. Whaley, *ScaLAPACK Users' Guide*, SIAM, Philadelphia, PA, 1997.

[4] J. Choi, J. J. Dongarra, L. S. Ostrouchov, A. P. Petitet, D. W. Walker, and R. C. Whaley, *The design and implementation of the ScaLAPACK LU, QR, and Cholesky factorization routines*, Scientific Programming, 5 (1996), pp. 173–184.

[5] W. Gropp, E. Lusk, and A. Skjellum, *Using MPI: Portable programming with the Message-Passing Interface*, MIT Press, Cambridge, MA, 1994.

[6] J. M. D. Hill, S. R. Donaldson, and A. McEwan, *Installation and user guide for the Oxford BSP toolset (v1.3) implementation of BSPlib*, technical report, Oxford University Computing Laboratory, Oxford, UK, Nov. 1997.

[7] J. M. D. Hill, B. McColl, D. C. Stefanescu, M. W. Goudreau, K. Lang, S. B. Rao, T. Suel, T. Tsantilas, and R. H. Bisseling, *BSPlib: The BSP programming library*, Parallel Computing, 24 (1998), pp. 1947–1980.

[8] V. S. Sunderam, *PVM: A framework for parallel distributed computing*, Concurrency: Practice and Experience, 2 (1990), pp. 315–339.

[9] L. G. Valiant, *A bridging model for parallel computation*, Communications of the ACM, 33 (1990), pp. 103–111.