Basic Techniques for Numerical Linear Algebra on Bulk Synchronous Parallel Computers*

Rob H. Bisseling
Department of Mathematics, Utrecht University
P. O. Box 80010, 3508 TA Utrecht, the Netherlands
`http://www.math.ruu.nl/people/bisseling`

Abstract

The bulk synchronous parallel (BSP) model promises scalable and portable software for a wide range of applications. A BSP computer consists of several processors, each with private memory, and a communication network that delivers access to remote memory in uniform time.

Numerical linear algebra computations can benefit from the BSP model, both in terms of simplicity and efficiency. Dense LU decomposition and other computations can be made more efficient by using the new technique of two-phase randomised broadcasting, which is motivated by a cost analysis in the BSP model. For LU decomposition with partial pivoting, this technique reduces the communication time by a factor of $(\sqrt{p}+1)/3$, where p is the number of processors.

Theoretical analysis, together with benchmark values for machine parameters, can be used to predict execution time. Such predictions are verified by numerical experiments on a 64-processor Cray T3D. The experimental results confirm the advantage of two-phase randomised broadcasting.

1 Introduction

The field of parallel numerical linear algebra has rapidly evolved over the last decade. A major development has been the acceptance of the two-dimensional cyclic data distribution as a standard for dense matrix computations on parallel computers with distributed memory. The two-dimensional cyclic distribution of an $m \times n$ matrix A over a parallel computer with $p = M \cdot N$ processors is given by

$$a_{ij} \longmapsto P(i \bmod M, j \bmod N), \text{ for } 0 \leq i < m \text{ and } 0 \leq j < n \ , \qquad (1)$$

*To appear in: Proc. First Workshop on Numerical Analysis and Applications, Rousse, Bulgaria, June 1996, Lecture Notes in Computer Science, Vol. 1196, Springer-Verlag, Berlin 1997.

where $P(s,t)$, with $0 \leq s < M$ and $0 \leq t < N$, is a two-dimensional processor number. To the best of our knowledge, this distribution was first used by O'Leary and Stewart [12] in 1985. It has been the basis of parallel linear algebra libraries such as PARPACK [1, 3], a prototype library for dense and sparse matrix computations which was developed for use on transputer meshes, and ScaLaPack [4], a public domain library for dense matrix computations which is available for many different parallel computers. (ScaLaPack uses a generalisation of (1), where each matrix element a_{ij} is replaced by a submatrix A_{ij}.) Distribution (1) has acquired a variety of names, such as 'torus-wrap mapping', which is used in [8], and 'scattered square decomposition' [5]. Following PARPACK, we use the term $M \times N$ *grid distribution*. In many cases, it is best to choose $M \approx N \approx \sqrt{p}$. For an optimality proof of the grid distribution with respect to load balancing and a discussion of its communication properties, see [3]. Application of this distribution in a wide range of numerical linear algebra computations (LU decomposition, QR factorisation, and Householder tridiagonalisation) is discussed in [8].

The chosen data distribution need not have any relation with the physical architecture of a parallel computer. Even though terms as 'processor row $P(s,*)$', or 'processor column $P(*,t)$' are used in this paper and elsewhere, these terms just describe a collection of processors with particular processor identities and not a physical submachine. Although many parallel linear algebra algorithms were originally developed for rectangular meshes or hypercubes, today it is recognised that these algorithms can often be used on other architectures as well, simply by taking the processor numbering as a logical numbering.

Parallel algorithms are always developed within a given programming model, whether it is explicitly specified or not. A simple parallel programming model that leads to portable and scalable software can clearly be of great benefit in applications, including numerical linear algebra. The Bulk Synchronous Parallel (BSP) model by Valiant [15] is such a simple model; it will be explained briefly in Sect. 2. The BSP model allows us to analyse the time complexity of parallel algorithms using only a few parameters. On the basis of such an analysis, algorithms can be better understood and possibly improved. One improvement is the technique of two-phase randomised broadcasting, which is presented in Sect. 3 and tested in Sect. 4. The technique is based on the idea of using intermediate randomly chosen processors, which originates in routing algorithms [14].

Within the BSP framework, there have been a few studies of broadcasting as part of parallel numerical linear algebra. Gerbessiotis and Valiant [6] present and analyse an algorithm for Gauss-Jordan elimination with partial pivoting. Their algorithm broadcasts matrix rows and columns in $\log_2 \sqrt{p}$ phases. In preliminary work with Timmers [13], we implemented the technique of two-phase randomised broadcasting as part of a parallel Cholesky factorisation. Although our theoretical analysis revealed major benefits, we did not observe them in practice, for the simple reason that we used a parallel computer with only four processors. To reap the benefits of this technique, more processors must be used. This is done in Sect. 4 of the present work, where 64 processors are used to study two-phase randomised broadcasting as part of a parallel LU decomposition. Recent theoretical work on communication primitives for the BSP model such as broadcast and parallel prefix can be found in [9].

2 BSP model

The BSP model was proposed by Valiant in 1990 [15]. It defines an architecture, a type of algorithm, and a function for charging costs to algorithms. We use the variant of the cost function proposed in [2]. For a recent survey of BSP computing, see [10].

A *BSP computer* consists of p processors, each with private memory, and a communication network that allows processors to access private memories of other processors. Each processor can read from or write to any memory cell in the entire machine. If the cell is local, the read or write operation is relatively fast. If the cell belongs to another processor, a message must be sent through the communication network, and this takes more time. The access time for different non-local memories is the same.

A *BSP algorithm* consists of a sequence of supersteps, each ended by a global barrier synchronisation. In a *computation superstep*, each processor performs a sequence of operations on locally held data. In numerical linear algebra, these operations are mainly floating point operations (flops). In a *communication superstep*, each processor sends and receives a number of messages. The messages do not synchronise the sender with the receiver and they do not block progress. Synchronisation takes place only at the end of a superstep.

The *BSP cost function* is defined as follows. An *h-relation* is a communication superstep where each processor sends at most h data words to other processors and receives at most h data words. We denote the maximum number of words sent by any processor by $h_{\rm s}$, and the maximum number received by $h_{\rm r}$. Therefore,

$$h = \max\{h_{\rm s}, h_{\rm r}\} \ . \tag{2}$$

This equation reflects the assumption that a processor can send and receive data simultaneously. Charging costs on the basis of h is motivated by the assumption that the bottleneck of communication lies at the entry or exit of the communication network, so that simply counting the maximum number of sends and receives per processor gives a good indication of communication time.

The cost of an h-relation is

$$T_{\rm comm}(h) = hg + l \ , \tag{3}$$

where g and l are machine-dependent parameters. The cost unit is the time of one flop. Cost function (3) is chosen because of the expected linear increase of communication time with h. The processor that sends or receives the maximum number of data words determines h and hence the communication cost. Since $g = \lim_{h\to\infty} T_{\rm comm}(h)/h$, the value of g can be viewed as the time (in flops) needed to send one word into the communication network, or to receive one word from it, in a situation of continuous message traffic. The linear cost function includes a nonzero constant l because each h-relation incurs a fixed cost. This fixed cost includes: the cost of global synchronisation; the cost of ensuring that all communicated data have arrived at their destination; and startup costs of sending messages.

The cost of a computation superstep with an amount of work w is

$$T_{\rm comp}(w) = w + l \ . \tag{4}$$

The amount of work w is defined as the maximum number of flops performed in the superstep by any processor. The value of l is taken to be the same as that of a communication superstep, despite the fact that the fixed cost is less: global synchronisation is still necessary, but the other costs disappear. The advantage of having one parameter l is simplicity; the total synchronisation cost of an algorithm can be determined by simply counting the supersteps. As a consequence of (3) and (4), the total cost of a BSP algorithm becomes an expression of the form $a + bg + cl$.

A BSP computer can be characterised by four parameter: p is the number of processors; s is the single-processor speed measured in flop/s; g is the communication cost per data word; and l is the synchronisation cost of a superstep. (We slightly abuse the language, because l also includes other costs.) We call a computer with these four parameters a BSPC(p, s, g, l). The execution time of an algorithm with cost $a+bg+cl$ on a BSPC(p, s, g, l) is $(a+bg+cl)/s$ seconds.

Estimates for g and l of a particular machine can be obtained by benchmarking a range of *full* h-relations, i.e., h-relations where each processor sends and receives exactly h data. In the field of numerical linear algebra, it is appropriate to use 64-bit reals as data words. The measured cost of a full h-relation will be an upper bound on the cost of an arbitrary h-relation. For a good benchmark, a representative full h-relation must be chosen which is unrelated to the specific architectural characteristics of the machine. A cyclic pattern will often suffice: processor $P(i)$, $0 \leq i < p$, sends its first data word to $P((i+1) \bmod p)$, the next to $P((i+2) \bmod p)$, and so on. The source processor $P(i)$ is skipped as the destination of messages, since local assignments do not require communication. (As an alternative, an average for a set of random full h-relations can be measured.)

Figure 1 shows the results of benchmarking cyclic full h-relations on 64 processors of a Cray T3D. The nodes of the Cray T3D used in this experiment consist of a 150 MHz Dec Alpha processor and a memory of 64 Mbyte. The communication network is a three-dimensional torus. The Cray T3D is transformed into a BSP computer by using the Cray T3D implementation of the

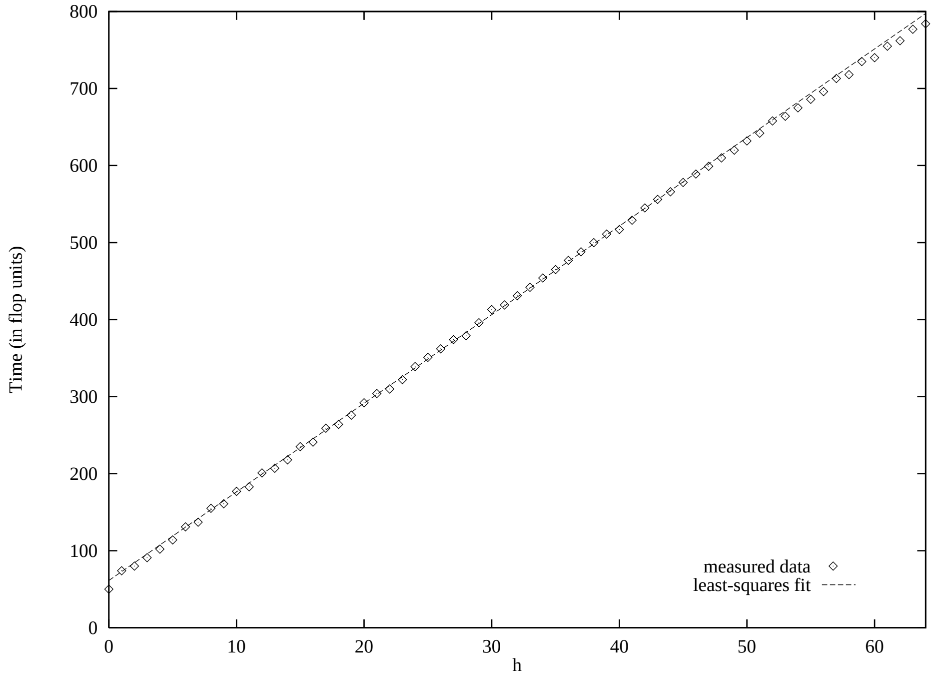

Figure 1: Time of an h-relation on a 64-processor Cray T3D

Oxford BSP library [11], version 1.1. The measured data points of the figure lie close to the straight line of a least-squares fit, so that the behaviour is indeed linear as modelled by (3). The fitted BSP parameters are $g = 11.5$ and $l = 61.4$. The sequential computing speed $s = 6.0\,\mathrm{Mflop/s}$ is obtained by measuring the time of a DAXPY operation $\mathbf{y} := \alpha\mathbf{x} + \mathbf{y}$, where $\mathbf{x}$ and $\mathbf{y}$ are vectors of length 1024 and α is a scalar.

Communication in the BSP model does not require any form of synchronisation between the sender and the receiver. Global barrier synchronisations guarantee memory integrity at the start of every superstep. The absence of sender/receiver synchronisation makes it possible to view the communications as one-sided: the processor that initiates the communication of a message is active and the other processor is passive. If the initiator is the sender, we call the send operation a *put*; if it is the receiver, we call the receive operation a *get*. Conceptually, puts and gets are single-processor operations. More and more, puts and gets are supported in hardware, which makes them very efficient. An early example of such support is the SHMEM facility of the Cray T3D. The use of one-sided communications makes programs simpler: the program text only includes a put or get statement for the initiator. This is in contrast to traditional message-passing programs where both parts of a matching send/receive pair must be included. In dense matrix computations, puts are usually sufficient. In sparse matrix computations, gets are sometimes needed because the receiver of the data knows which data it needs, for instance because of its local sparsity pattern, but the sender does not have this knowledge. Note that we use the terms 'send' and 'receive', even if we only perform puts. Furthermore, we still analyse algorithms counting both sent and received data words. The advantage of one-sided communications lies mainly in simpler program texts of algorithms and their implementations.

3 Two-Phase Randomised Broadcasting

An important communication operation in parallel algorithms for numerical linear algebra is the broadcast of a matrix row or column. Let us examine a column broadcast in detail. The $M \times N$ grid distribution assigns matrix element a_{ij} to processor $P(i \bmod M, j \bmod N)$. Suppose we have to broadcast column k, i.e., each matrix element a_{ik} must be communicated to all processors that contain elements from matrix row i. For the grid distribution, these processors are contained in processor row $P(i \bmod M, *)$. Often, a column broadcast and a similar row broadcast prepare the ground for a subsequent rank-1 update of the matrix. The broadcasts communicate the elements a_{ik} and a_{kj} that are required in assignments of the form $a_{ij} := a_{ij} - a_{ik}a_{kj}$.

Usually, only part of a column k must be broadcast, such as e.g. the elements a_{ik} with $i > k$ in stage k of an LU decomposition. In our explanation, however, we assume without loss of generality that all elements a_{ik}, $0 \leq i < m$, must be broadcast, where $m \geq 1$. A simple algorithm that performs this column broadcast is given by Fig. 2.

if $k \bmod N = t$ **then**
    for all $i : 0 \leq i < m \wedge i \bmod M = s$ **do**
        put a_{ik} in $P(s, *)$;
sync;

Figure 2: Program text for processor $P(s,t)$ of one-phase broadcast of column k

This broadcast consists of one phase, which is a communication superstep. The cost of this superstep can be analysed as follows. The only processors that send data are the M processors from $P(*, k \bmod N)$. These processors send $N-1$ copies of at most $R = \lceil m/M \rceil$ matrix elements, so that the maximum number of elements sent per processor is $h_{\mathrm{s}} = R(N-1)$. In general, all processors except the senders receive elements from column k. The maximum number of elements received per processor is $h_{\rm r} = R$. It follows that $h = h_{\rm s} = R(N-1)$. (Note that this count also holds in the special case $N = 1$, where $h = 0$.) The resulting cost of the one-phase broadcast is

$$T(\text{one-phase broadcast}) = \left\lceil \frac{m}{M} \right\rceil (N-1)g + l \ . \tag{5}$$

In the common case $M = N = \sqrt{p}$ with $m \gg \sqrt{p} \gg 1$, the cost becomes $T \approx mg + l$.

This cost analysis reveals a major disadvantage of the straightforward algorithm. The maximum number of sends is $N-1$ times larger than the maximum number of receives. In the ideal situation, $h_{\rm s}$ and $h_{\rm r}$ would both be equal to the average number of data communicated per processor, which is the communication volume divided by p. In the one-phase broadcast, however, the communication work is badly balanced. The senders have to copy the column elements and send all these copies out, whereas the receivers only receive one copy of each column element. For $M = N = \sqrt{p}$ with $m \gg \sqrt{p} \gg 1$, we are far from the ideal situation: $h_{\rm s} \approx m$, $h_{\rm r} \approx m/\sqrt{p}$, and the average number of communicated data is about $m/\sqrt{p}$.

The communication imbalance can be eliminated by first sending one copy of each data element to a randomly chosen intermediate processor and making this processor responsible for copying and for sending the copies to the destination. This method is similar to two-phase randomised routing [14], which sends packets from source to destination through a randomly chosen intermediate location, to avoid congestion in the routing network. The new broadcasting method splits the original communication superstep into two communication supersteps: an unbalanced h-relation which randomises the location of the data elements to be broadcast; and a balanced h-relation which performs the broadcast itself. In analogy with the routing case, we call the resulting pair of h-relations a *two-phase randomised broadcast*.

In general, the intermediate processor in a two-phase randomised broadcast is chosen randomly. However, in certain situations with regular communication patterns, for instance in dense matrix computations, the intermediate processor can also be chosen deterministically. (For sparse matrix computations, which are often irregular, a random choice may be more appropriate.) Processor $P(s, k \bmod N)$, $0 \leq s < M$, is the source of the matrix elements a_{ik} with $i \bmod M = s$. The row index i of a local element a_{ik} of this processor can be written as $i = \mathtt{i}M + s$, where $\mathtt{i}$ is a local index. Note that $\mathtt{i} = i \operatorname{div} M$, and that the local indices of the local elements are consecutive. The intermediate processor for an element a_{ik} of $P(s, k \bmod N)$ can be chosen within processor row $P(s, *)$. This is a natural choice, because the broadcast involves a nearly equal number of elements for each processor row and because source and destination are always in the same processor row. The consecutive local indices $\mathtt{i}$ can be used to assign intermediate addresses in a cyclic fashion (similar to the cyclic method of the grid distribution itself). This can be done by assigning elements with local index `i` to processor $P(s, \texttt{i} \bmod N)$. The resulting two-phase broadcast is given by Fig. 3.

> **if** $k \bmod N = t$ **then**
>   **for all** $i : 0 \leq i < m \wedge i \bmod M = s$ **do**
>   put a_{ik} in $P(s, (i \text{ div } M) \bmod N)$;
> sync;
> **for all** $i : 0 \leq i < m \wedge i \bmod M = s \wedge (i \text{ div } M) \bmod N = t$ **do**
>   put a_{ik} in $P(s, *)$;
> sync;

Figure 3: Program text for processor $P(s, t)$ of two-phase broadcast of column k

The cost analysis of the two-phase broadcast is as follows. The broadcast consists of two communication supersteps. Let $R = \lceil m/M \rceil$. The first superstep has $h_\mathrm{s} = R - \lfloor R/N \rfloor$, since at least $\lfloor R/N \rfloor$ puts are local, and $h_\mathrm{r} = \lceil R/N \rceil$. For simplicity, we write $h \leq R$. The second superstep has $h_\mathrm{s} = \lceil R/N \rceil (N-1)$ and $h_\mathrm{r} = R - \lfloor R/N \rfloor$. An upper bound for h_s can be obtained from $h_\mathrm{s} = \lceil R/N \rceil (N-1) \leq (R/N + 1)N = R + N$. We write $h \leq R + N$. The resulting cost of the two-phase broadcast is

$$T(\text{two-phase broadcast}) \leq (2 \left\lceil \frac{m}{M} \right\rceil + N)g + 2l \ . \tag{6}$$

For $M = N = \sqrt{p}$ with $m \gg p$, the cost becomes $T \approx 2(m/\sqrt{p})g + 2l$. Compared to the one-phase broadcast, the communication cost decreases by the considerable factor of $\sqrt{p}/2$, at the modest expense of doubling the synchronisation time.

The cost of the two-phase broadcast is close to minimal because a lower bound for every broadcast is

$$T(\text{broadcast}) \geq \left\lceil \frac{m}{M} \right\rceil g + l \ . \tag{7}$$

The lower bound follows from the fact that the processor with most words to broadcast has to send $\lceil m/M \rceil$ data words at least once and this takes at least one superstep.

Figure 4 illustrates the implementation of a two-phase broadcast as part of an LU decomposition program. The program fragment performs the broadcast of elements a_{ik} with $k < i < n$, for an $n \times n$ matrix A. The program is written in ANSI C extended with primitives from the proposed BSP Worldwide standard library [7]. The program uses two-dimensional processor coordinates `(s,t)` and the corresponding one-dimensional identity `s+t*M`. The first `bsp_put` primitive writes the double a_{ik} = a[i][kc] into ak[i] of processor s+(i mod N)*M. The value of i0 is the smallest integer i $\geq$ kr1 with i mod $N = t$.

```
#define SZD (sizeof(double))
/* nr =  number of local rows of matrix A
   kr1 = first local row with global index > k */

if (k%N==t){
    /* kc = local column with global index = k */
    for(i=kr1; i<nr; i++){
         bsp_put(s+(i%N)*M,&a[i][kc],ak,i*SZD,SZD);
    }
}
bsp_sync();
for(i=i0; i<nr; i +=N){
    for(t1=0; t1<N; t1++){
        bsp_put(s+t1*M,&ak[i],ak,i*SZD,SZD);
    }
}
bsp_sync();
```

Figure 4: Implementation of two-phase broadcast from LU decomposition in ANSI C with BSP extensions

4 Experimental Results

The theoretical analysis of Sect. 3 shows that two-phase randomised broadcasting is advantageous. In this section, we put this claim to the test by performing numerical experiments on a parallel computer. We choose LU decomposition with partial pivoting as our test problem and we perform our tests on 64 nodes of a Cray T3D. This parallel computer with parameters $p = 64$, $s = 6.0$ Mflop/s, $g = 11.5$, and $l = 61.4$ has good BSP characteristics, because of the low values of g and l and because of its scalable and predictable behaviour, see Sect. 2. For the experiments, we implemented LU decomposition with one-phase and two-phase broadcasts. Except for the broadcasts, the two implementations are identical. The $n \times n$ matrix A which is decomposed into $PA = LU$ is distributed according to the 8×8 grid distribution. The test matrix is chosen such that in each stage k, $0 \leq k < n$, of the LU decomposition, row k is explicitly swapped with the pivot row r, forcing communication. This is done to generate the worst-case communication behaviour.

The communication operations of LU decomposition in stage k are: a broadcast of column k which involves elements a_{ik} with $k < i < n$; a broadcast of row k which involves elements a_{kj} with $k < j < n$; and a swap of rows k and r which involves elements a_{kj} and a_{rj} with $0 \le j < n$. The time complexity of the broadcasts is analysed in Sect. 3; the row swap costs about $(n/\sqrt{p})g + l$ flop units. Each stage of the parallel LU decomposition algorithm with two-phase randomised broadcasting contains six supersteps. (This small number can be obtained in an implementation by combining supersteps and by performing computation and communication in the same superstep.) The total cost of the algorithms can be shown to be equal to

$$T(\text{LU with one-phase broadcast}) \approx \frac{2n^3}{3p} + \left(n^2 + \frac{n^2}{\sqrt{p}}\right) g + 5nl \ , \tag{8}$$

and

$$T(\text{LU with two-phase broadcast}) \approx \frac{2n^3}{3p} + \frac{3n^2}{\sqrt{p}} g + 6nl \ . \tag{9}$$

Figure 5 shows the measured time of LU decomposition with partial pivoting for matrices of order n=250–5000. Two-phase randomised broadcasting clearly improves performance for all problem sizes. The largest gain factor achieved is 2.53, for $n = 500$. Theoretically, the total communication cost of the algorithm decreases from $(n^2 + n^2/\sqrt{p})g$ to $(3n^2/\sqrt{p})g$. This is a decrease by a factor of $(\sqrt{p}+1)/3$, which equals three for $p = 64$. Synchronisation is dominant for small problems and computation is dominant for large problems; in both asymptotic cases the decrease in communication time is relatively small compared to the total solution time. In the intermediate range of problem size, however, the decrease is significant. The experiments show that in the range of $n = 250$–1500 the decrease of the total time is more than a factor of two, but even for $n = 5000$ it is substantial: the two-phase version takes only 143 s, whereas the one-phase version takes 202 s.

The theoretical timing formula (9) and the measured machine characteristics p, s, g, and l can be used to predict the total execution time and its three components. Table 1 compares predicted and measured execution times. The table displays reasonable agreement for small n, but significant discrepancy for larger n. This discrepancy is mainly due to a sequential effect: it is difficult to predict sequential computing time, because the actual computing rate may differ from one application to another. Simply substituting a benchmark result in a theoretical time formula gives what may be termed an *ab initio prediction*, i.e. a prediction from basic principles, which may be useful as an indication of expected performance, but not as an accurate estimate. Of course, the prediction can always be improved by using the measured computing rate for the particular application, instead of the benchmark rate s. (For LU decomposition, we measured a sequential rate of 9.35 Mflop/s for $n = 1000$, which is considerably faster than the benchmark rate of 6.0 Mflop/s.)

A breakdown of the predicted total time in computation time, communication time, and synchronisation time gives additional insight. Synchronisation

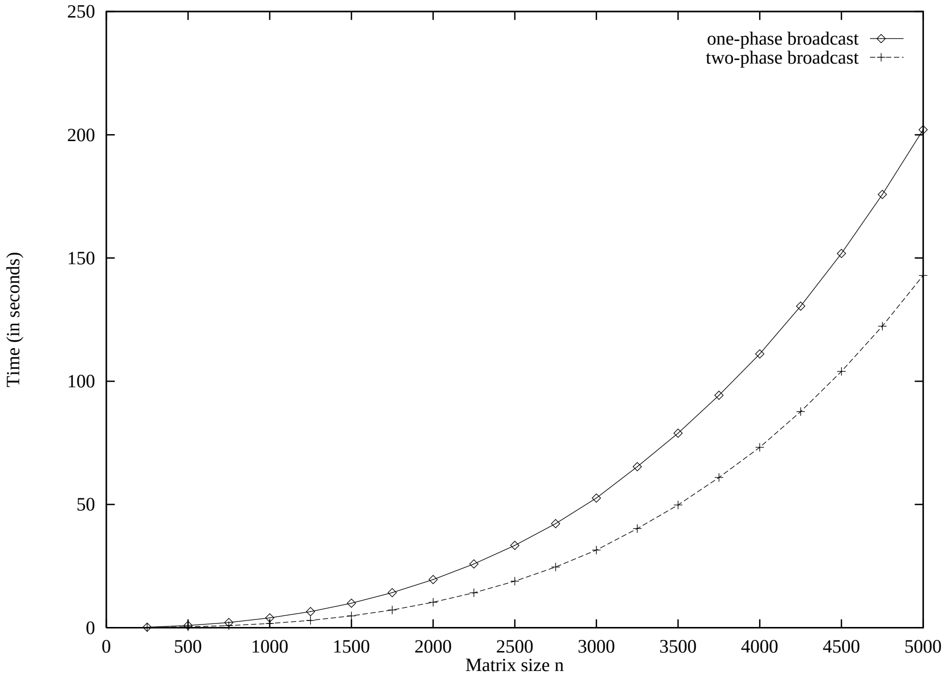

Figure 5: Time of LU decomposition with partial pivoting on a 64-processor Cray T3D

time is negligible for this machine and this problem, except in the case of very small n. This implies that the increase in synchronisation time caused by two-phase randomised broadcasting is irrelevant. The savings in communication time, however, is significant. For example, for $n = 5000$, the predicted communication time is brought down from 53.91 s to 17.97 s, and this predicted decrease accounts for much of the total measured decrease of 59 s. (As in the case of computation, we cannot expect an ab initio prediction to give an exact account of all communication effects.)

Our implementation of LU decomposition was developed to study broadcasts in a typical parallel matrix computation. The program is a straightforward implementation in ANSI C of parallel LU decomposition with the grid distribution. The program does not use Basic Linear Algebra Subprograms (BLAS) or submatrix blocking, which are crucial in obtaining high computing rates. As expected, our implementation is far from optimal with respect to computing speed; it achieves 0.56 Gflop/s for $n = 8000$ and $p = 64$, whereas ScaLaPack attains 5.3 Gflop/s [4] for the same problem on the same machine. Absolute performance can of course be improved by using BLAS wherever possible and reorganising the algorithm to use matrix-matrix multiplication in the core com-

Table 1: Predicted and measured time (in s) of parallel LU decomposition

	Predicted				Measured
n	T_{comp}	T_{comm}	T_{sync}	T_{total}	T_{total}
250	0.03	0.04	0.02	0.09	0.10
500	0.22	0.18	0.03	0.43	0.36
750	0.73	0.40	0.05	1.18	0.87
1000	1.74	0.72	0.06	2.52	1.72
2000	13.89	2.88	0.12	16.89	10.33
3000	46.88	6.47	0.18	53.53	31.53
4000	111.11	11.50	0.25	122.86	73.20
5000	217.01	17.97	0.31	235.29	142.91

putation. The data distribution need not be changed. Note that for higher computing rates the issue of communication will become relatively more important and hence the benefits of two-phase randomised broadcasting will grow in significance.

5 Conclusion

The main result of this work is that significant performance improvement can be obtained by using the BSP model in parallel numerical linear algebra. The new technique of two-phase randomised broadcasting has been introduced to balance the communication work in a parallel computer. This technique can be used to accelerate row and column broadcasts in LU decomposition, Cholesky factorisation, QR factorisation, and Householder tridiagonalisation. Numerical experiments on a Cray T3D show that the improvement can be observed in practice. Application is not restricted to a BSP context; the technique may be used wherever row or column broadcasts occur in parallel numerical linear algebra. The derivation of the technique is natural in the BSP model. One may speculate that it is much more difficult to obtain such techniques without the guidance of the BSP model.

Acknowledgements

This work is partially supported by the university grants programme of NCF in the Netherlands and Cray Research. Numerical experiments were performed on the Cray T3D of the Ecole Polytechnique Fédéral de Lausanne.

References

[1] R. H. Bisseling and L. D. J. C. Loyens. Towards peak parallel LINPACK performance on 400 transputers. *Supercomputer*, 45:20–27, 1991.

[2] R. H. Bisseling and W. F. McColl. Scientific computing on bulk synchronous parallel architectures. In B. Pehrson and I. Simon, editors, *Technology and Foundations: Information Processing '94, Vol. I*, volume 51 of *IFIP Transactions A*, pages 509–514. Elsevier Science Publishers, Amsterdam, 1994.

[3] R. H. Bisseling and J. G. G. van de Vorst. Parallel LU decomposition on a transputer network. In G. A. van Zee and J. G. G. van de Vorst, editors, *Parallel Computing 1988*, volume 384 of *Lecture Notes in Computer Science*, pages 61–77. Springer-Verlag, Berlin, 1989.

[4] J. J. Dongarra and D. W. Walker. Software libraries for linear algebra computations on high performance computers. *SIAM Review*, 37(2):151–180, 1995.

[5] G. C. Fox, M. A. Johnson, G. A. Lyzenga, S. W. Otto, J. K. Salmon, and D. W. Walker. *Solving Problems on Concurrent Processors: Vol. I, General Techniques and Regular Problems.* Prentice Hall, Englewood Cliffs, NJ, 1988.

[6] A. V. Gerbessiotis and L. G. Valiant. Direct bulk-synchronous parallel algorithms. *Journal of Parallel and Distributed Computing*, 22(2):251–267, 1994.

[7] M. W. Goudreau, J. M. D. Hill, K. Lang, B. McColl, S. B. Rao, D. C. Stefanescu, T. Suel, and T. Tsantilas. A proposal for the BSP Worldwide standard library. Technical report, Oxford Parallel, Oxford, UK, July 1996.

[8] B. A. Hendrickson and D. E. Womble. The torus-wrap mapping for dense matrix calculations on massively parallel computers. *SIAM Journal on Scientific Computing*, 15(5):1201–1226, 1994.

[9] B. H. H. Juurlink and H. A. G. Wijshoff. Communication primitives for BSP computers. *Information Processing Letters*, 58:303–310, 1996.

[10] W. F. McColl. Scalable computing. In J. van Leeuwen, editor, *Computer Science Today: Recent Trends and Developments*, volume 1000 of *Lecture Notes in Computer Science*, pages 46–61. Springer-Verlag, Berlin, 1995.

[11] R. Miller. A library for bulk synchronous parallel programming. In *General Purpose Parallel Computing*, pages 100–108. British Computer Society Parallel Processing Specialist Group, 1993.

[12] D. P. O'Leary and G. W. Stewart. Data-flow algorithms for parallel matrix computations. *Communications of the ACM*, 28(8):840–853, 1985.

[13] P. Timmers. Implementing dense Cholesky factorization on a BSP computer. Master's thesis, Department of Mathematics, Utrecht University, Utrecht, the Netherlands, June 1994.

[14] L. G. Valiant. A scheme for fast parallel communication. *SIAM Journal on Computing*, 11:350–361, 1982.

[15] L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.