UMIACS-TR-88-81 October 1988
CS-TR-2135 Revised Jan. 1990

Communication and Matrix Computations on Large Message Passing Systems

G. W. Stewart*

ABSTRACT

This paper is concerned with the consequences for matrix computations of having a rather large number of general purpose processors, say ten or twenty thousand, connected in a network in such a way that a processor can communicate only with its immediate neighbors. Certain communication tasks associated with most matrix algorithms are defined and formulas developed for the time required to perform them under several communication regimes. The results are compared with the times for a nominal n^3 floating point operations. The results suggest that it is possible to use a large number of processors to solve matrix problems at a relatively fine granularity, provided fine grain communication is available.

*Department of Computer Science and Institute for Advanced Computer Studies, University of Maryland, College Park, MD 20742. This work was supported in part by Air Force Office of Sponsored Research under grant AFOSR-82-0078

Communication and Matrix Computations on Large Message Passing Systems

G. W. Stewart*

1. Introduction

This paper is concerned with the consequences for matrix computations of having a rather large number of general purpose processors, say ten or twenty thousand, connected in a network in such a way that a processor can communicate only with its immediate neighbors. We will assume that each processor can perform floating-point arithmetic at around a million operations per second (1 Mflop/sec), which gives the network a potential speed of ten to twenty Gflop/sec. Although no such system has been built, there is no reason in principle why it could not. It is therefore appropriate to consider what such a large number of processors means for matrix computations.

One immediate consequence is that we can pose very large problems, so large that we may not be able to solve them in a reasonable time. Suppose, for example, that each processor has two Mwords of memory. Then on a system of ten thousand processors, we can store a matrix of order, say, $n = 40,000$. Let us assume that a computation with this matrix generates n^3 flops, which is the typical order of many matrix computations. Then if each processor can be run in parallel on the problem at the rate of 1 Mflop/sec, the time required for the computation will be

$$\frac{40{,}000^3}{10{,}000 \times 10^6}\,\text{sec} = 6{,}400\,\text{sec} = 107\,\text{min}.$$

Thus we must wait over a hour and three quarters for an answer. Speeding up the processors to 10 Mflop/sec still leaves us waiting for over ten minutes. Although ten minutes or even two hours may not sound like a long time to wait for the solution to a problem, many $O(n^3)$ matrix algorithms have order constants of one-hundred or more, and the resulting times can stretch into days. Moreover,

*Department of Computer Science and Institute for Advanced Computer Studies, University of Maryland, College Park, MD 20742. This work was supported in part by Air Force Office of Sponsored Research under grant AFOSR-82-0078

matrix problems are frequently solved not just once, but again and again as a part of a larger computation.[1]

It should come as no surprise that we must ultimately be faced with matrix problems that are too large to solve. A dense matrix problem of order n is determined by $k = n^2$ units of data and requires time $O(n^3) = O(k^{\frac{3}{2}})$. Thus to keep computation times constant each increase in memory requires a disproportionate increase in computation speed. The only surprise is that we have had to wait so long for this phenomenon to become manifest. Until recently the rule of thumb — if you can fit it in memory, you can afford to solve it — has applied to dense matrix computations.

All this suggests that the next generation of parallel computers will have to be used to solve smaller matrix problems faster rather than to solve the largest problems that can be put on the system. However, this creates difficulties of its own. In addition to being computational intensive, parallel matrix algorithms must communicate a lot. It is true that the ratio of computation to communication decreases with the problem size, so that if the problem is large enough we can attain high efficiency (for an example of this phenomena in another context, see [9]). But will a problem that is large enough to be solved efficiently be too large to be solved in a reasonable time?

In this paper we shall investigate this question empirically by what amounts to an elaborate back-of-the-envelope calculation. Briefly, we shall assume that the matrix in question has been partitioned into square submatrices, which are assigned to the individual processors of the system. We will then consider some typical communication tasks and their implementations on a grid and a hypercube. For each case we will compute the communication times for problems of various sizes and compare them with the computation costs to determine break-even points.

Such calculations are notoriously subject to the biases of the person who controls the underlying assumptions, and I will discuss mine in the appropriate places. However, this is the place to answer one natural objection: namely, that it is unrealistic to treat large dense matrix problems, since most large problems are sparse and can be handled by special techniques. The answer is twofold. In the first place in some applications it is necessary to compute all the eigenvalues of a matrix [17], and the only known way to do this in general is to use dense matrix techniques. Second, matrices can be sparse without having an easily exploitable structure;

[1] One of the referees has pointed out that unless care is taken the time to input the matrix can become a significant part of the computation.

e.g., matrices corresponding to multi-dimensional meshes. The matrices arising from queuing networks are of this kind [3]. A third, perhaps less cogent answer is that dense matrix problems have become an important benchmark for commercial systems; those that do not perform well on them are at a disadvantage in the marketplace.

In the next section we will describe the assumptions on which our calculations will be based. In §3 we will derive formulas for the communication times. In §4 we will fix the values of the parameters in the model and describe the calculation. The final section is devoted to a discussion of the results. To anticipate the principle result, we will find that the efficiency of certain important matrix algorithms at fine granularity is restricted by the granularity of the communication. Fortunately, this problem can be ameliorated by an easily implementable mode of fine grain communication which we call simple streaming.

2. Assumptions

In this section, we will fix the assumptions that underlie our calculations. They may be divided into four categories: the geometry of the system, the distribution of the matrix, communication tasks, and communication modes. A discussion of the limitations of these assumptions and alternatives will be found at the end of this section.

2.1. The geometry of the system

The processors in most message passing systems are connected in a network of fixed geometry. In this paper, we will compare the performance of two geometries: the grid and the hypercube.

Grids. Here we assume that the number of processors p is a square and that they are arranged in a $\sqrt{p} \times \sqrt{p}$ grid. Each processor can communicate with the processors immediately to the north, south, east, and west. The greatest distance between two processors in the network is $2\sqrt{p} - 2$.

Hypercubes. Here we assume that the number of processors is a power of two. Each processor is assigned a binary number from zero to $p - 1$. Two processors are neighbors if their numbers differ in exactly one bit. Thus each processor has $\log_2 p$ neighbors. The greatest distance between two processors in the network is $\log_2 p$.

2.2. The distribution of the matrix

The example given in the introduction shows that for large systems assigning even a few columns of a matrix to each processor can result in prohibitively large problems. This suggests that we shall have to deal with problems in which the order of the matrix n is less than the number of processors p, which precludes assigning entire rows or columns to the processors.

As an alternative, we shall assume that the number of processors is a square and partition the matrix A in the form

$$A = \begin{pmatrix} A_{11} & A_{12} & \cdots & A_{1,\sqrt{p}} \\ A_{21} & A_{22} & \cdots & A_{2,\sqrt{p}} \\ \vdots & \vdots & & \vdots \\ A_{\sqrt{p},1} & A_{\sqrt{p},2} & \cdots & A_{\sqrt{p},\sqrt{p}} \end{pmatrix}, \tag{2.1}$$

where each submatrix A_{ij} is (approximately) of order $n/\sqrt{p}$. Each processor will then be responsible for calculations involving its own submatrix.

The order of assignment is important. When the processors form a grid, we shall use the natural assignment, in which the (i,j)-submatrix in (2.1) is assigned to the corresponding (i,j)-processor in the grid. When the processors form a hypercube, we embed a grid in the hypercube in such a way that each row and each column lies on a sub-hypercube [12, 11]. We then assign the matrix to the embedded grid as described above. This insures that the distance between two processors in any row or column is not greater than $\log_2 p/2$.

2.3. Communication tasks

It is beyond the scope of this paper to consider the communication requirements of all possible matrix algorithms. Fortunately, many of the most important algorithms — those that reduce a matrix to simpler form by transformations that introduce zeros into the matrix — have similar requirements. These algorithms proceed by generating a vector u that defines the transformation and then broadcasting u across the system, where the processors use it to apply the transformation of the matrix. These steps are then repeated, usually about n times, until the matrix has the required form. Let us examine these two steps in greater detail.

A typical example of a generation step is the computation of multipliers in Gaussian elimination with partial pivoting (for a description of Gaussian elimination and related reductions see [8]). Here the maximum element of the current

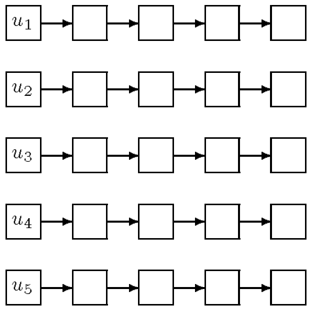

Figure 2.1: Broadcasting of u

pivot column must be determined. This is then distributed to to processors responsible for the pivot column, who use it to compute the multipliers. To compute the maximum, all the processors responsible for the pivot column must communicate, and the same is true for the broadcasting the maximum. Thus the total communication time is at least twice the communication diameter of the system times the time it takes to transmit a single number. We will use this fact in deriving our formulas.

Once the transformation vector u has been computed, it must be distributed to the processors. A little reflection on Gaussian elimination shows that a processor needs to know the ith component of u only if it is responsible some part of row i of the matrix. This means that if the processors are arranged in a grid according to the the blocks for which they are responsible, then the corresponding pieces of u have to be distributed as in Figure 2.1.

In most algorithms the generation and reduction must be repeated about n times. To give the process a name, we will call it SIMPLE REDUCTION.

Simple reduction is typically found in calculations, such as eigenvalue algorithms, in which the matrix is transformed from both sides. When the matrix is transformed from only one side, there is an important variation. In these algorithms, once the vector u starts moving across the rows, a second can follow behind it, then a third, etc., so that the communication occurs in parallel waves passing through the processors. Clearly this parallelization of the communication process has the potential to save a great deal of time. We will call this mode WAVEFRONT REDUCTION.

Unfortunately, at the fine granularities that we are interested in, communication in the generation step can take almost as long as the subsequent broadcasting of the reduction vector, so that closely packed waves never have a chance to form. In Gaussian elimination with partial pivoting or in the similar Householder reduction, nothing can be done about this. However, in the Cholesky reduction and certain reductions based on orthogonal transformations, the generation of the transformation vector can be pipelined with the other communication. We will call this mode PIPELINED REDUCTION and model it by leaving the generation time out of our formulas.

2.4. Communication modes

In asynchronous message passing systems of the kind we are considering there are two distinct communication problems. The first is to move data between two neighboring processors; the second is to move data between two programs (i.e., processes) running asynchronously on neighboring processors. The first problem can in principle be solved by appropriate hardware: handshaking and queue management can be done by coprocessors. The second problem is complicated by the fact that the data, even if it is on the receiving processor, must not be delivered to the receiving program until it has been explicitly requested, at which point the receiving program is blocked while the request is processed and the message delivered. In all message passing systems currently available, the overhead for this is large; the best require about 50–100 μsec to pass a message between processes (e.g., see [6]).

Since in parallel algorithms for matrix computations data must be passed between programs, we will assume that any communication has a startup time σ, which is about 100 μsec. Specifically we will assume that it requires time

$$\sigma + k\tau \tag{2.2}$$

to transmit a message of length k between programs. Here τ^{-1} represents a transmission rate. This model has been widely used and seems to describe the behavior of real-life systems.

When a message must be relayed down a line of processors, it is not necessarily efficient to send the entire message from processor to processor. For example, if the startup time σ is zero, it will clearly be most efficient to stream the message item by item. Consequently, in our calculations we shall suppose that a message of length n is divided into m packets of length n/m, which are then passed one after another through the system. When we come to evaluate formulas for communication time, we will choose m to minimize the time. We shall call this mode of communication PACKET STREAMING.

There is an alternative, in which all the programs in a stream drop what they are doing and cooperate in passing data one unit at a time. This SIMPLE STREAMING, as we will call it, amounts to packet streaming with a negligible startup time and a packet size of one. Simple streaming requires special hardware for its implementation. The WARP [1] and the TRANSPUTER, which provide hardware for synchronizing the delivery of small messages, can perform simple streaming. Moreover, a class of architectures in which neighboring processors can access each other's memories has been proposed by the author and his colleagues [14]. Here simple streaming can be done under program control with copy loops synchronized by shared variables.

2.5. Discussion

Since we cannot compute everything, we have had to be selective in making assumptions. In this subsection, we will discuss some of the alternatives.

Most dense matrix algorithms run nicely on a grid of processors. In fact, people who code matrix algorithms for a hypercube find that they use the higher connectivity simply to reduce the communication diameter. Thus a square grid and a hypercube are reasonable choices of geometry. We have already commented that the use of a linear array would place a lower bound on the size of of the problems we could treat efficiently; hence its exclusion. However, at very coarse granularity, a linear array with row assignment may actually be more efficient, since the computations in the generation step may be done in parallel.

From the point of view of arithmetic efficiency, there are better ways to distribute a matrix than blocks. The problem is that as the reduction proceeds the processors whose rows and columns have all been eliminated become idle. One cure is to wrap the rows and columns around the processor grid modulo $\sqrt{p}$. Although this keeps all processors running — at least until the tail end of the reduction — it does not significantly change the communication requirements, which is why we do not consider this kind of assignment here.

In describing the communication tasks we have restricted ourselves to conventional matrix algorithms. For our purposes, the most important alternative is block algorithms, which are currently under investigation in connection with hierarchical shared memories [2, 4]. These certainly deserve serious consideration, but at present block algorithms for the full range of matrix computations do not exist. Moreover, it may turn out that the communication costs for these algorithms will be comparable to the conventional algorithms, though this is by no means certain.

Another alternative is "fast" matrix algorithms that run in time less that $O(n^3)$ (for a survey see [15]). Unfortunately, the numerical properties of these algorithms is so little understood, that at present they are not serious contenders (however, see [10]).

We have also understated the amount of broadcasting involved in typical algorithms. For example, in Gaussian elimination, the pivot row must also be broadcast. But this represents factors of two or three in an application where we are concerned with orders of magnitude. Moreover, it is counterbalanced by the fact that we will compare these times with the time it takes to perform n^3 operations, which is also an underestimate for many algorithms.

3. Formulas

In this section we shall derive formulas for communication times under the various conditions outlined in the last section. We will use the following notation:

System	grid	gr
	hypercube	hc
Task	simple reduction	sr
	wavefront reduction	wr
	pipeline reduction	pr
Method	packet streaming	ps
	simple streaming	ss

In principle we have twelve possible combinations. However, since simple streaming requires all processors to shake hands and pass data, there is no need to consider simple streaming with wavefront or pipeline reduction. Consequently we will consider only the cases

1. gr/sr/ps
2. gr/sr/ss
3. gr/wr/ps
4. gr/pr/ps

and the corresponding cases for the hypercube.

The derivation of the time for gr/wr/ps is typical. Recall that under packet streaming, a message is divided into m packets, which are streamed across the processor grid. The time required for the first processor to compute the first transformation vector is

$$2\sqrt{p}(\sigma + \tau),$$

where σ is the startup time and τ^{-1} is the transmission rate. Since the number of items in a packet is $\frac{n}{m\sqrt{p}}$, the time for a message to pass from processor to processor is

$$m\left(\sigma + \frac{n}{m\sqrt{p}}\tau\right). \tag{3.1}$$

We begin by looking at the first column of processors. These processors must generate a transformation vector and ship it to the next processor, after which they can begin to generate another transformation vector. This must be repeated n times to give a total time of

$$2n\sqrt{p}(\sigma + \tau) + mn\left(\sigma + \frac{n}{m\sqrt{p}}\tau\right).$$

Once the first column of processors is finished, last transformation vector must reach the last processor. The first packet requires approximately time

$$\sqrt{p}\left(\sigma + \frac{n}{m\sqrt{p}}\tau\right)$$

to reach the last processor. It then takes time (3.1) for the last processor to receive the message. Thus the total communication time is approximately

$$2n\sqrt{p}(\sigma + \tau) + (\sqrt{p} + mn)\left(\sigma + \frac{n}{m\sqrt{p}}\tau\right).$$

An approximation to the optimal value of m can be derived be setting the derivative with respect to m of the above formula to zero. This gives

$$m_{\mathrm{opt}} = \sqrt{\frac{\tau}{\sigma}}.$$

Since $\tau < \sigma$ this gives a value for m that is less than one, an impossibility. Consequently, we will always take $m_{\mathrm{opt}} = 1$ in this case.

The other cases are derived similarly. The results are contained in Table 3.1.

The times for a hypercube are given in Table 3.2. They may be derived from the formulas for a grid by replacing $\sqrt{p}$ by $\log_2 p$ whenever the former refers to the

Case	Time	m_{opt}
gr/sr/ps	$n\left[2\sqrt{p}(\sigma+\tau)+(\sqrt{p}+m)\left(\sigma+\frac{n}{m\sqrt{p}}\tau\right)\right]$	$\sqrt{n\frac{\tau}{\sigma}}$
gr/sr/ss	$n\left(3\sqrt{p}+\frac{n}{\sqrt{p}}\right)\tau$	—
gr/wr/ps	$2n\sqrt{p}(\sigma+\tau)+(\sqrt{p}+nm)\left(\sigma+\frac{n}{m\sqrt{p}}\tau\right)$	1
gr/pr/ps	$(\sqrt{p}+nm)\left(\sigma+\frac{n}{m\sqrt{p}}\tau\right)$	1

Table 3.1: Communication Times for a Grid

Case	Time	m_{opt}
hc/sr/ps	$n\left[2\log_2 p\,(\sigma+\tau)+(\log_2 p+m)\left(\sigma+\frac{n}{m\sqrt{p}}\tau\right)\right]$	$\sqrt{n\frac{\log_2 p}{\sqrt{p}}\frac{\tau}{\sigma}}$
hc/sr/ss	$n\left(3\log_2 p+\frac{n}{\sqrt{p}}\right)\tau$	—
hc/wr/ps	$2n\log_2 p\,(\sigma+\tau)+(\log_2 p+nm)\left(\sigma+\frac{n}{m\sqrt{p}}\tau\right)$	1
hc/pr/ps	$(\log_2 p+nm)\left(\sigma+\frac{n}{m\sqrt{p}}\tau\right)$	1

Table 3.2: Communication Times for a Hypercube

communication diameter as opposed to a message length. However, it should be noted that in deriving formulas in this manner, we have made the tacit assumption that the items in a message to be broadcast can be sent from all $\log_2 p$ connections at the same time, something we will call SIMULTANEOUS FAN-OUT. If items can be sent to only one connection at a time — SEQUENTIAL FAN-OUT — then the time for the first processor to start a packet is essentially the same as the time for the packet to be broadcast, which inhibits the formation of tightly packed waves. In particular, with sequential fan-out the formula for the case hc/wr/ps reduces to that of the case hc/sr/ps, and hc/pr/ps reduces to hc/sr/ps without the generation term. The consequences of this are quite unfortunate.

The reader will have noted that in deriving these formulas, we have treated quantities like the message length $n/\sqrt{p}$ as if they were continuous, whereas in fact they can take only integer values. We have also ignored the restriction that p be a square for the grid or a power of two for the hypercube. However, by the continuity of the formulas, this will make little difference in the results we will obtain from them.

4. Calculations and Comments

In this section we will present the results of evaluating the formulas derived in the last section. Table 4.1 gives the values we will use for the various parameters. The particular values were selected because they seem reasonable to the author. It might be objected that α^{-1} is too small, given that supercomputers perform beyond the 100 Mflop/sec range. However, for microprocessors of the kind we can afford to use in a large parallel system, a rate of one Mflop/sec is quite respectable. The value of τ was choses to balance communication and arithmetic. The value of σ is an approximation to the current best value. The extremes

Table 4.1: Parameters in the evaluation

Parameter	Time
Arithmetic (α^{-1})	10^6 flops/sec
Startup (σ)	10^{-4} sec
Transmission rate (τ^{-1})	10^6 words/sec
Order of matrix (n)	500–30,000
Number of processors (p)	1,000–20,000

Figure 4.1: Arithmetic Times

$n = 500$ and $p = 20000$ allow us to explore fine granularity: the block size for such a configuration would be about 3×3.

Moreover, the way we present the results makes them more widely applicable than the specific values of the parameters might suggest. For example, the arithmetic times are presented as contours of the times as a function of n and p. Thus, the same contours with different values attached will serve for all arithmetic speeds. The communication times are normalized by the arithmetic times, so that the same contours serve when the arithmetic and communication are speeded up proportionally. Finally, the contrast between packet streaming and simple streaming gives some idea of the effect of the startup time. Nonetheless, for the reader who may wish to vary the parameters, fragments of MATLAB code are given in an appendix.

Figure 4.1 gives the contours of the common logarithm of the computation time, calculated from the formula

$$\alpha\frac{n^3}{p}.$$

The heavy line curving out from the origin is the contour corresponding to 1 sec. Each line to the right represents an order of magnitude increase in the time, and the line fragment to the left an order of magnitude decrease. The results confirm what we hinted at in the introduction: it is very easy to pose problems that are too large to solve. For matrices of order 10,000 we may expect times in the hundreds of seconds. For matrices of order 30,000 the times are in the thousands or even tens of thousands of seconds.

The raw communication times are not very informative. Instead, for each of the cases for which we derived formulas, we plot the contours of the common logarithm of the ratio ρ of the communication time to the arithmetic time. If we define the efficiency of the calculation as

$$\frac{\text{arithmetic}}{\text{arithmetic} + \text{communication}}$$

then

$$\text{efficiency} \cong \begin{cases} \rho^{-1} & \text{if } \rho \text{ is large} \\ 1-\rho & \text{if } \rho \text{ is small} \end{cases}$$

Figure 4.2 gives the contours for the grid. The heavy line is the zero contour, where the efficiency is one-half. The efficiency decreases by roughly an order of magnitude for each line as we go from this line to the left. It quickly approaches one as we go to the right. The nearer the heavy line is to the y-axis, the greater the efficiency at fine granularity.

The combination of packet streaming with either simple reduction or wavefront reduction is quite inefficient unless n is large or p is small. This is confirmed by what people have observed experimentally on message passing systems (e.g., see [7, 6, 5]). For wavefront reduction the generation step is almost entirely responsible for the large communication time; for when it is removed ($\mathsf{gr/pr/ps}$) the bold line moves very near the p-axis. The combination of simple reduction and simple streaming is almost as good. To summarize, without simple streaming, the system is good for simple matrix tasks, such as solving linear equations or least squares problems, provided the algorithms are carefully tailored. But simple streaming is required to make the system effective for more complicated tasks, such as eigenvalue problems.

The contours for the hypercube in Figure 4.3 are better, but still not good for simple and wavefront reduction with packet streaming. The combination of simple

Figure 4.2: Grid Ratios

Figure 4.3: Hypercube ratios

reduction and simple streaming is now the winner, with pipelined reduction and packet streaming a close second. However, without simultaneous fan-out the bold line for the latter will move significantly to the right, leaving simple streaming as the only efficient mode at low granularity.

It is important not to make a fetish of efficiency. A glance at the formulas shows that both arithmetic time and communication time are monotonic increasing in n. Thus if with a fixed number of processors we can solve a problem of a given size in a satisfactory amount of time we can solve all smaller problems in at least the same amount of time. In many applications — for example those in which the matrix problem is solved only once — this may be sufficient. However, in applications where we wish to solve many small problems, efficiency makes a real difference. For example, on the grid the 10^2 contour for sr/ps is approximately the same as the 10^0 contour for sr/ss. Thus in a problem dominated by computations in this range the switch from packet streaming to simple streaming will reduce the time by a factor of about one hundred.

5. Conclusions

In this section we will draw some conclusions from the calculations presented in the preceding section.

The least controversial conclusion is that the raw arithmetic power needed to solve dense matrix problems severly limits the size of the problems we can reasonably consider. We had a hint of this in the introduction, and the plot in Figure 4.1 confirms it. The difficulty is that if p is roughly proportional to n, the time grows as the square of either. To make matters worse, some important $O(n^3)$ matrix algorithms have order constants that are well over one hundred.[2] Increasing the speed of the processors will of course help; but this is a linear effect bucking a quadratic trend, and it must ultimately loose out.

The encouraging conclusion is that communication costs need not restrict the granularity of the calculations unduly, so that the problems we can afford to solve at all, we can up to a point solve faster by using more processors. Even without simple streaming, we can implement some important matrix algorithms efficiently. With simple streaming, we can implement virtually all.

The plots show that there is not much to decide between a grid and a hypercube, at least if the latter has simultaneous fan-out. The ratios for the hypercube

[2]Jack Dongarra (personal communication) has used MATLAB [13] to estimate the order constants for a number of matrix algorithms.

are better, but not markedly so. However, this is in part due to the grid-like nature of dense matrix computations, and these plots should not be used to argue for grids against hypercubes.

Finally, the calculations show that simple streaming is a good thing. In fact simple streaming is a special case of a more general mode of communication and computation in which items are actually manipulated as they pass from processor to processor. This permits the efficient implementation on MIMD systems of algorithms that are essentially systolic (for an example see [16]). It should be remarked that this kind of communication is not incompatible with other forms of data routing; on the contrary they should be regarded as supplementary. Since simple streaming can be easily implemented in a number of ways, there seems to be no reason not to include it in the next generation of message passing systems.

References

[1] M. Annoratone, E. Arnould, T. Gross, H. T. Kung, O. Menzicioglu, and J. A. Webb (1987). "The WARP Computer: Architecture, Implementation, and Performance." *IEEE Transaction on Computers*, **C-36**, 1523–1538.

[2] C. Bischof and C. Van Loan (1987). "The *WY* Representation for Products of Householder Transformations." *SIAM Journal on Scientific and Statistical Computing*, **8**, s2–s13.

[3] S. C. Bruell and G. Balbo (1980). *Computational Algorithms for Closed Queueing Networks*. North Holand, New York.

[4] J. J. Dongarra, J. Du Croz, I. Duff, and S. Hammarling (1987). "A Proposal for a Set of Level 3 Basic Linear Algebra Subprograms." *SIGNUM Newsletter*, **22**, 2–14.

[5] S. C. Eisenstat, M. T. Heath, C. S. Henkel, and C. H. Romine (1988). "Modified Cyclic Algorithms for Solving Triangular Systems on Distributed-Memory Multiprocessors." *SIAM Journal on Scientific and Statistical Computing*, **9**, 589–600.

[6] G. C. Fox, S. W. Otto, and A. J. G. Hey (1987). "Matrix Algorithms on a Hypercube I: Matrix Multiplication." *Parallel Computing*, **4**, 17–31.

[7] A. George, M. T. Heath, and J. Liu (1986). "Parallel Cholesky Factorization on a Shared-Memory Multiprocessor." *Linear Algebra and Its Applications*, **77**, 165–187.

[8] G. H. Golub and C. F. Van Loan (1983). *Matrix Computations.* Johns Hopkins University Press, Baltimore, Maryland.

[9] J. L. Gustafson, G. R. Montry, and R. E. Benner (1988). "Development of Parallel Methods for a 1024-Processor Hypercube." *SIAM Journal on Scientific and Statistical Computing*, **9**, 609–638.

[10] N. J. Higham (1989). "Exploiting Fast Matrix Multiplication within the Level 4 BLAS." Technical Report 89-984, Department of Computer Science, Cornell University. To appear in *ACM Tran. Math. Soft.*.

[11] S. Lennart Johnsson and Ching-Tien Ho (1986). "Spanning Graphs for Optimum Broadcasting and Personalized Communication in Hypercubes." Research Report YALE/DCS/RR-500, Department of Computer Science, Yale University.

[12] S. Lennart Johnsson and Ching-Tien Ho (1987). "Matrix Multiplication on Boolean Cubes Using Generic Communication Primitives." Research Report YALE/DCS/RR-530, Department of Computer Science, Yale University.

[13] Cleve Moler (1982). "MATLAB Users' Guide." Technical Report CS81-1 (Revised), Department of Computer Science, University of New Mexico.

[14] D. P. O'Leary, Roger Pierson, G. W. Stewart, and Mark Wieser (1986). "The Maryland CRAB: A Module for Building Parallel Computers." Technical Report TR-1660, Department of Computer Science, University of Maryland.

[15] V. Pan (1984). *How to Multiply Matrices Faster.* Springer, New York.

[16] G. W. Stewart (1987). "A Parallel Implementation of the QR Algorithm." *Parallel Computing*, **5**, 187–196.

[17] J. E. Van Ness (1986). "Examples of Eigenvalue/vector Use in Electric Power System Problems." In J. Cullum and R. A. Willoughby, editors, *Large Scale Eigenvalue Problems*, pages 181–192, Amsterdam. Elsiveir.

Appendix: MATLAB code

.

The following are fragments of the MATLAB code used to generate the results. The quantities `alpha`, `sigma`, and `tau` must be initialized to their desired values. In addition the vectors `nn` and `pp` must contain the points at which the times are to be evaluated. In this paper

```
pp = 1000 : 1000 : 20000
nn = 500 : 500 : 30000
```

The code for generating the grid times is

```
nproc = length(pp);
nmat = length(nn);
result = zeros(nproc, nmat);

% arithmetic time

for i=1:nproc
   p = pp(i);
   for j=1:nmat
     n = nn(j);
     alpha*n^3/p;
     result(i,j) = ans;
   end
end
arith = result;
ariths = log10(result);

% grid/ simple reduction/ packet streaming

for i=1:nproc
   p = pp(i);
   sqrp = sqrt(p);
   for j=1:nmat
     n = nn(j);
     m = min([max([sqrt(n*tau/sig), 1]),n/sqrp]);
     result(i,j) = n*(2*sqrp*(sig + tau) + (sqrp + m)*(sig + n*tau/(m*sqrp)));
   end
end
grsrps = log10(result./arith);

% grid/ simple reduction/ simple streaming

for i=1:nproc
   p = pp(i);
   sqrp = sqrt(p);
   for j=1:nmat
     n = nn(j);
     result(i,j) = n*(3*sqrp + n/sqrp)*tau;
   end
end
```

```
grsrss = log10(result./arith);

% grid/ wavefront reduction/ packet streaming

for i=1:nproc
   p = pp(i);
   sqrp = sqrt(p);
   g = 2*sqrp*(sig+tau);
   for j=1:nmat
     n = nn(j);
     result(i,j) = n*2*sqrp*(sig + tau) + (sqrp + n)*(sig + n*tau/sqrp);
     end
   end
end
grwrps = log10(result./arith);

% grid/ pipelined reduction/ packet streaming

for i=1:nproc
   p = pp(i);
   sqrp = sqrt(p);
   for j=1:nmat
     n = nn(j);
     result(i,j) = (sqrp + n)*(sig + n*tau/sqrp);
   end
end
grprps = log10(result./arith);
```

The contour plots were produced by the following code

```
v(5) = -.005;
v(6) = .005;
subplot(221),contour(grsrps,v,nn,ppp)
title('gr/sr/ps')
xlabel('order')
ylabel('processors')
v(5) = -.01;
v(6) = .01;
subplot(222),contour(grsrss,v,nn,ppp)
title('gr/sr/ss')
xlabel('order')
ylabel('processors')
v(5) = -.005;
v(6) = .005;
subplot(223),contour(grwrps,v,nn,ppp)
title('gr/wr/ps')
xlabel('order')
ylabel('processors')
v(5) = -.05;
v(6) = .05;
subplot(224),contour(grprps,v,nn,ppp)
title('gr/pr/ps')
xlabel('order')
ylabel('processors')
```

Here

```
v = [-4 -3 -2 -1 -.01 .01 1 2 3 4]
ppp = 20000: -1000 : 1000
```

The code for the hypercubes is similar.