NASA-CR-172,436

NASA Contractor Report 172436

ICASE REPORT NO. 84-41

ICASE

ON THE IMPACT OF COMMUNICATION COMPLEXITY IN THE DESIGN OF PARALLEL NUMERICAL ALGORITHMS

Dennis Gannon

and

John Van Rosendale

Contracts No. NAS1-17070 and NAS1-17130
August 1984

INSTITUTE FOR COMPUTER APPLICATIONS IN SCIENCE AND ENGINEERING
NASA Langley Research Center, Hampton, Virginia 23665

Operated by the Universities Space Research Association

NASA
National Aeronautics and
Space Administration

Langley Research Center
Hampton, Virginia 23665

On the Impact of Communication Complexity in the Design of Parallel Numerical Algorithms.

Dennis Gannon [1]

Dept. of Computer Sciences
Purdue University,
West Lafayette, Indiana

John Van Rosendale [1]

ICASE, NASA Langley Research Center
Hampton, VA.

ABSTRACT

This paper describes two models of the cost of data movement in parallel numerical algorithms. One model is a generalization of an approach due to Hockney, and is suitable for shared memory multiprocessors where each processor has vector capabilities. The other model is applicable to highly parallel nonshared memory MIMD systems. In this second model, algorithm performance is characterized in terms of the communication network design. Techniques used in VLSI complexity theory are also brought in, and algorithm independent upper bounds on system performance are derived for several problems that are important to scientific computation.

1. Introduction

The traditional model of parallel algorithm analysis was motivated by a desire to explore the potential of parallelism. Thus the question was asked: given an unlimited number of processing elements and an infinite capacity to move and permute data, what is the fastest method to solve the problem under consideration? This has proven to be a fruitful area of research and much has been learned. However, with the appearance of the Illiac IV, the Cray I and the CDC Cyber 205, it was quickly realized that the design of data structures and the cost of processor to processor and processor to memory communication are critical ingredients in the design and analysis of practical algorithms. The goal of this paper is to highlight the role played by communication cost in the analysis of numerical algorithms.

Because the spectrum of parallel architectures suitable for scientific computation is so broad, it is difficult to derive one analytical model of computation

[1] Research supported by the National Aeronautics and Space Administration under NASA Contract Nos. NAS1-17070 and NAS1-17130 while the authors were in residence at the Institute for Computer Applications in Science and Engineering, NASA Langley Research Center, Hampton, VA 23665. Primary support for the first author was provided by an IBM Faculty Development Grant.

that characterizes the performance of every machine. In this paper we restrict our attention to three families of machine architectures and describe an analytical model of performance that is reasonably suited to each. In particular, our goal for each model is to characterize the effect of communication costs on system performance. In each model we give an estimate of effective efficiency or speed-up of a computation as a function the latency and bandwidth of the processor communication medium.

The first model is that of a "medium scale" shared-memory multiprocessor, having perhaps 2 to 32 processors, with each processor capable of exploiting substantial local vector parallelism. Section 2 of this paper gives a formal description of the shared memory model and illustrates a method of analyzing algorithms for machines of this type. Several standard, but important, numerical problems are studied and a number of alternate implementations are analyzed. In particular, it is shown that for machines which have two levels of parallelism the performance of algorithms depends strongly on the way in which the problem is partitioned to fit on the architecture. The performance of the algorithms is given as a function of global and local memory latencies, the speed of arithmetic operations, the number of processors, and the size of the problem.

The second model is that of a highly parallel MIMD system where processors communicate through a large network and there is no shared memory. We assume here a number of processors ranging from perhaps 32 to a few thousand, but with processors of lesser power than in the shared memory model. Analysis and design of algorithms for such systems turns out to be significantly different than it is for the shared memory machines. In section 3 it is shown that the techniques used in VLSI complexity analysis can be used to derive reasonable upper bounds on speed-up and efficiency. The appropriate parameters for this analysis turn out to be the ratio of message transmission times to arithmetic speed, and the relation of the problem being solved to the topology of the communication network. By looking at specific algorithms it is shown that many of the derived upper bounds are exact.

As a variant of this second architecture model, in section 4 we consider machines interconnected by packet switched communications networks. Analysis of algorithms for such machines is similar to analysis of algorithms for other non shared memory machines, except communication delays play a central role. The paper concludes with a discussion of the shortcomings of the approaches described here and suggests several directions where more work needs to be done.

2. Shared Memory Machines

One of the clearest trends in commercial systems is the trend toward multiprocessor shared memory architectures (see Figure 2.1), where each processor has either a pipelined multitasking or vector capability. This family of multiprocessors includes the Cray X-MP, Cray-II, the HEP-I and HEP-*II*[1] and the ETA GF10. The proposed Cedar multiprocessor [GKLS83] may be viewed as a machine in this class, where each "processor" is in fact a cluster of smaller processors. In this section we consider the design and analysis of algorithms for such machines. We begin with a list of properties shared by many machines in this class. (Of course, no list will characterize every machine and the set of specification below should be considered only as an approximation to a family of

[1] The Hep I and II machines belong in this class, though the model of analysis given below does not fully describe their performance.

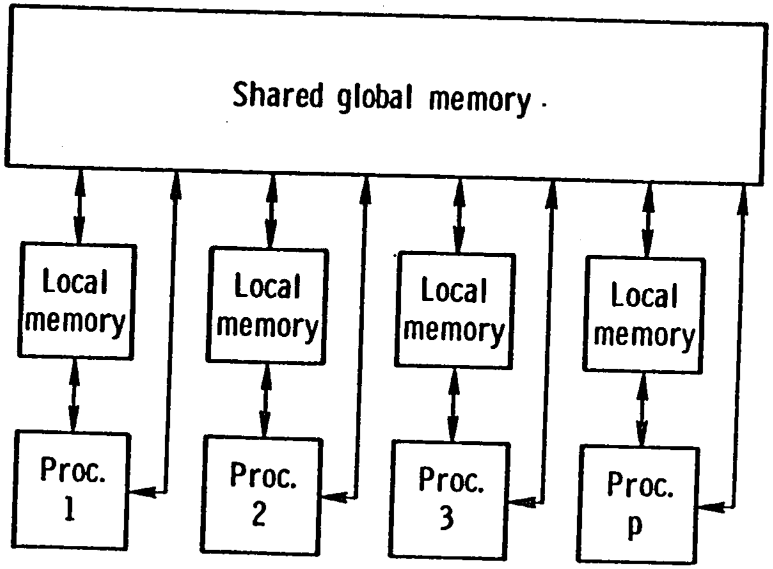


Figure 2.1. Shared Memory Multiprocessor

architectures.)

1. There are p processors, with p roughly in the range $2 \le p \le 32$.
2. All processors have equal access to shared memory and vectors may be of arbitrary length and stride [2].
3. Each processor also has a sizable local memory from which it can fetch vectors of arbitrary length and stride.
4. Each processor can perform vector diadic operations (or vector triadic operations where one operand is a scalar) using operands in from either the local memory or global shared memory. The execution time for a vector operation of length n is

   $$r_\infty^{-1}(n + n_{1/2}^{G})$$

   if either operand is in global memory, while it is

   $$r_\infty^{-1}(n + n_{1/2}^{L})$$

   if both operands are in local memory. Here r_∞^{-1} is the asymptotic performance rate for one processor, and $n_{1/2}$ is the vector length required to achieve half the asymptotic performance rate, an idea due to Hockney and Jesshope [HoJe81]. Here we assume local memory accesses have much less latency than global memory accesses, and thus

   $$n_{1/2}^{L} \ll n_{1/2}^{G}.$$

   The scale of this inequality depends on the machine. For a system that uses a network of $log(p)$ stages to bring data from shared memory into local (cache) memory, one may have $n_{1/2}^{G} = n_{1/2}^{L} + C{*}log(p)$ for some constant C. In general, the ratio $\frac{n_{1/2}^{G}}{n_{1/2}^{L}}$ might vary between 10 and 1000 .[3]
5. The parallel execution of p tasks on p processors is denoted by:

   ```
   pardo(i = 1,p)
        task(i);
   endpar;
   ```

   Here task(i) is a procedure, block, or statement that is executed on the i-th processor. No assumptions will be made about the processor synchronization or task scheduling mechanisms other than that the execution order will be consistent with the serial data dependencies.

Algorithm Design

The most natural way to design algorithms for these systems is to employ what the mechanical and structural engineering community has called problem substructuring. In this approach, the problem is divided into a set of independent tasks, each operating on its own portion of the data structure.

[2] In some systems performance may be severely degraded if two processors access the same vector or an access has non-unit stride. The algorithms that follow avoid multiple accesses, but to simplify the analysis, non-unit stride performance problems have been ignored.

[3] An alternate assumption would be that the latencies $n_{1/2}^{L}$ and $n_{1/2}^{G}$ are equal but that the computation rates for operand from global and local memory differ, or that all vector operations on global data must be "cached" to local memory before execution. An analytical model can be built from any such set of assumptions.

To illustrate this idea we consider an example studied many times in the literature [HoJe81],[BrKa81],[LaVo75],[Ston75],[Hell77]: that of solving T systems of tridiagonal matrix equations each of size n. Our notation is as follows. Vectors and arrays will be denoted with capitol letters (X, Y) and problem instances will be denoted with a superscript. Scalars (and vector components) will be denoted by lower case letters (with subscripted positions). A range of superscripts or subscripts will be denoted by [i;j] where i is the first element and j is the last element. In general, we shall use superscripts to denote equation numbers and subscripts to denote the row within the matrix of an equation. Let A be the tridiagonal matrix whose i^{th} row has nonzero elements (b_i, a_i, c_i). We seek the solutions of T tridiagonal systems.

$$A^j X^j = Y^j, \quad j = 1,T$$

We assume here, and through out this paper, that the matricies A are all symmetric positive definite and can be factored without partial pivoting.

The simplest algorithm is to divide the problem into p sets of problems each containing T/p subproblems. The standard vector algorithm is then run on each processor.

```
simple( A, Y, X)
     Pardo(i = 1, p)
               begin (* on processor i do *)
                    r = (i-1)*T/p+1; s = i*T/p;
                    for j = 2 to n do begin
                         $m^{[r:s]} = - b_{j+1}^{[r:s]} / a_j^{[r:s]}$
                         $a_{j+1}^{[r:s]} = a_{j+1}^{[r:s]} + m * c_j^{[r:s]}$;
                         $y_{j+1}^{[r:s]} = y_{j+1}^{[r:s]} + m^{[r:s]} * y_j^{[r:s]}$;
                         end;
                    $x_n^{[r:s]} = x_n^{[r:s]} / a_n^{[r:s]}$;
                    for j = n-1 downto 1 do
                         $x_j^{[r:s]} = (y_j^{[r:s]} - c_j^{[r:s]} * x_{j+1}^{[r:s]}) / a_j^{[r:s]}$;
               end;
          endpar;
```

The indices r,s,j and vector m within each block are assumed to be local to the executing processor. If all data is stored and fetched from shared (global) memory (local memory used only for m) then the cost of this algorithm is

$$T_p^{simp.GM} = r_\infty^{-1}(8n - 7)(\frac{T}{p} + n_{\frac{1}{2}}^{G})$$

On the other hand, if one first brings the matrix and data vectors down to the local memory and then solves the $\frac{T}{p}$ problems there and copies the solution vector back to global memory the cost is in terms of $n_{\frac{1}{2}}^{L}$ for the expression above, but it also includes the movement of 5 vectors ($a_{[1:n]}^{[r:s]}$, $b_{[1:n]}^{[r:s]}$, $c_{[1:n]}^{[r:s]}$, $Y^{[r:s]}$ and $X^{[r:s]}$) each of length $\frac{Tn}{p}$ to and from shared memory.

$$T_p^{simp.LM} = r_\infty^{-1}(8n - 7)(\frac{T}{p} + n_{\frac{1}{2}}^{L}) + 5*(\frac{Tn}{p} + n_{\frac{1}{2}}^{G})$$

One useful method for comparing these two implementations of this algorithm is to compute the "effective efficiency" of each. Observe that the asymptotic speed of our machine is $r_\infty p$ operations per second. The set of T tridiagonals requires $(8n-7)T$ operations. If the machine could be programmed to operate at 100% efficiency the execution time would be

$$T^{opt} = r_{\infty}^{-1} \frac{(8n-7)T}{p}$$

The *effective efficiency* of the simple algorithm operating from shared memory is defined by

$$E^{simp,GM} = \frac{T^{opt}}{T^{simp,GM}} = \frac{1}{1 + \frac{pn_h^G}{T}} \qquad (2.1)$$

For $n \gg p$, the algorithmic effective efficiency for the simple algorithm operating from local memory is approximately

$$E^{simp,LM} = \frac{8/13}{1 + \frac{8pn_h^L}{13T} + \frac{5pn_h^G}{13nT}} \qquad (2.2)$$

Thus, if T is large in relation to pn_h^G, the latency cost for global memory access gets masked by the arithmetic, and the global memory scheme is superior. On the other hand if T is near n_h^G, then the local memory method is superior.

An alternative solution is to substructure the problem so that processor i eliminates variables $(i-1)n/p+1$ through $in/p-1$ from each of the T systems. The result is a set of T problems of size $2p$. (This is a variation on a parallel algorithm of Sameh and Kuck [Saku78].)

```
Substructure(A,Y,X);

eliminate: pardo(i = 1, p)
                (* in processor i do *)
                for j = (i-1)*n/p + 2 to i*n/p - 1 do begin
                     $m^{[1:T]} = -b_{j+1}^{[1:T]} / a_j^{[1:T]}$;
                     $a_{j+1}^{[1:T]} = a_{j+1}^{[1:T]} + m^{[1:T]} * c_j^{[1:T]}$;
                     $b_{j+1}^{[1:T]} = m^{[1:T]} * b_j^{[1:T]}$;
                     $y_{j+1}^{[1:T]} = y_{j+1}^{[1:T]} + m^{[1:T]} * y_j^{[1:T]}$;
                end;
          endpar;
          pado(i = 1, p)
                (* in processor i do *)
                for j = i*n/p - 1 downto (i-1)*n/p + 2 do begin
                     $m^{[1:T]} = -c_{j-1}^{[1:T]} / a_j^{[1:T]}$;
                     $b_{j-1}^{[1:T]} = b_{j-1}^{[1:T]} + m^{[1:T]} * b_j^{[1:T]}$;
                     $c_{j-1}^{[1:T]} = m^{[1:T]} * c_j^{[1:T]}$;
                     $y_{j-1}^{[1:T]} = y_{j-1}^{[1:T]} + m^{[1:T]} * y_j^{[1:T]}$;
                end;
          endpar;
```

The resulting system is shown in Figure 2.2. The subsystem corresponding to rows $(i-1)*n/p$ and $i*n/p - 1$ for $i = 1,p$ can be solved by the "simple" method described above and the remaining variables can be solved by the "back solve" process

```
bk-solve: pardo(i = 1,p)
                for j = (i-1)*p/n+1 to i*p/n-1 do begin
                     $y_j^{[1:T]} = y_j^{[1:T]} - b_j^{[1:T]} * x_{(i-1)p/n}^{[1:T]} + c_j^{[1:T]} * x_{i*n/p}^{[1:T]}$;
                     $x_j^{[1:T]} = y_j^{[1:T]} / a_j^{[1:T]}$;
                end;
          endpar;
```

$$\begin{array}{l|cccccccccccccccccccc|l}
1 & a & & & & c & & & & & & & & & & & & & & & & \\
 & b & a & & & c & & & & & & & & & & & & & & & & \\
 & b & & a & & c & & & & & & & & & & & & & & & & \\
 & b & & & a & c & & & & & & & & & & & & & & & & \text{processor 1} \\
n/p & b & & & & a & c & & & & & & & & & & & & & & \\
\cline{2-21}
n/p+1 & & & & & b & a & & & & c & & & & & & & & & & \\
 & & & & & & b & a & & & c & & & & & & & & & & \\
 & & & & & & b & & a & & c & & & & & & & & & & \text{processor 2} \\
 & & & & & & b & & & a & c & & & & & & & & & & \\
2n/p & & & & & & b & & & & a & c & & & & & & & & & \\
\cline{2-21}
2n/p+1 & & & & & & & & & & b & a & & & & c & & & & & \\
 & & & & & & & & & & & b & a & & & c & & & & & \\
 & & & & & & & & & & & b & & a & & c & & & & & \text{processor 3} \\
 & & & & & & & & & & & b & & & a & c & & & & & \\
3n/p & & & & & & & & & & & b & & & & a & c & & & & \\
\cline{2-21}
3n/p+1 & & & & & & & & & & & & & & & b & a & & & & c & \\
 & & & & & & & & & & & & & & & & b & a & & & c & \\
 & & & & & & & & & & & & & & & & b & & a & & c & \text{processor 4} \\
 & & & & & & & & & & & & & & & & b & & & a & c & \\
4n/p & & & & & & & & & & & & & & & & b & & & & a &
\end{array}$$

Substructured Elimination to reduced system involving equations 1, n/p, 2n/p, 2n/p+1, 3n/p, 3n/p+1,4n/p for p=4 processors.

Figure 2.2 Substructured Elimination

As uming all vectors are fetched and stored in shared memory, the algorithm given above requires

$$T_p^{subst,GM}(n,T) \;=\; 17(\frac{n}{p}-2)*r_\infty^{-1}(T+n^G_{1/2}) + (16p-7)*r_\infty^{-1}(\frac{T}{p}+n^G_{1/2}) + 4S$$

time steps where S is the cost of processor synchronization. To do the same algorithm from local memory requires the downloading of 4 vectors of length nT/p for the original problem and 4 vectors of length T for the T subproblems of size p divided among p processors. The uploading of the subproblems requires one vector move of length T for each processor and uploading the final solution is a vector move of length nT/p. The penalty for doing the substructured algorithm in local memories is then

$$5*(\frac{nT}{p}+T+2n^G_{\frac{1}{2}}).$$

Define the multiprocessor speed-up for the simple algorithm from shared memory by the relation

$$S_p^{simple}(n,T) \;=\; \frac{T_1(n,T)}{T_p^{simple}(n,T)}$$

For the shared memory version, this value is approximated by

$$\frac{p(T+n^G_{1/2})}{T+pn^G_{1/2}}$$

For $T = n^G_{1/2}$ this gives a speed-up of less than $\frac{2p}{(p+1)}$. For large n and ignoring S, the substructured algorithm that uses only shared memory has a speed-up of approximately

$$S_p^{substr} \;=\; \frac{\frac{8}{17}}{1+\frac{16}{17n}\frac{(T+pn^G)}{(T+n^G)}}$$

over the single processor simple method. In the range $T = n^G_{1/2}$ the substructured algorithm yields a speed-up of approximately $p/2$. In fact, the reader can verify that the point at which the simple algorithm is superior to the substructured algorithm when both use only shared memory is

$$T \;>\; \frac{1}{9}(8p-17)n^G_{\frac{1}{2}} \;-\; \frac{4r_\infty p}{9(n-2p)}S.$$

If we again consider effective efficiency, one finds that for $n \gg p$ the approximate performance is

$$E^{subst,GM} \;=\; \frac{8/17}{1+\frac{n^G}{T}+\frac{16p^2n^G}{17nT}} \tag{2.3}$$

$$E^{subst,LM} \;=\; \frac{8/22}{1+\frac{17n^L}{22T}+\frac{(10p+16p^2)n^G_{\frac{1}{2}}}{8nT}} \tag{2.4}$$

Figure 2.3 plots the effective efficiencies (equations 2.1-2.4) for the four methods described above as a function of T for n = 64,000, p=32, n^L = 10, n^G = 1000. Observe that as T becomes large the simple shared memory method has the best asymptotic performance. On the other hand, for small T the local memory, substructured algorithm is clearly superior. Consequently, we find the choice of optimal algorithm depends heavily on the relation of problem parameters to

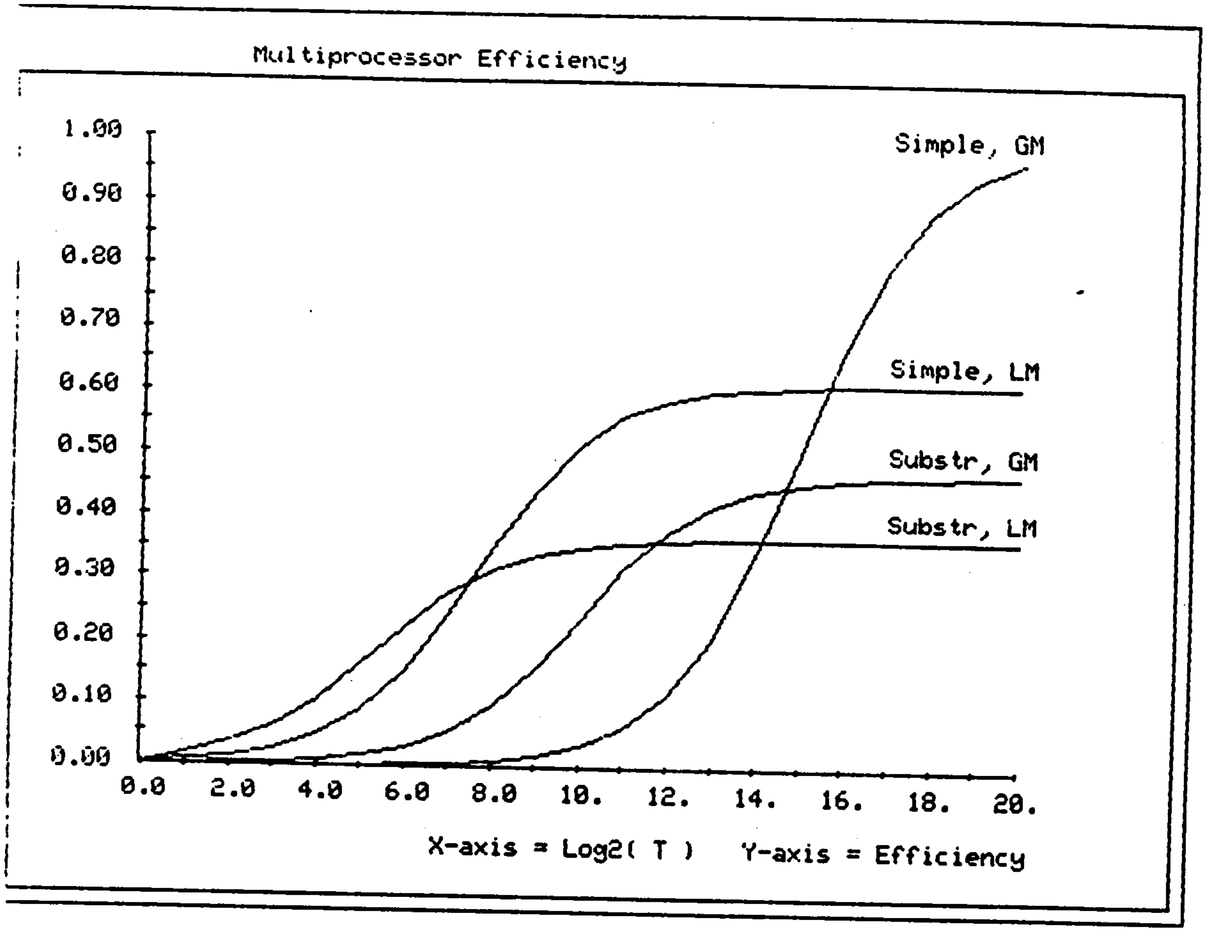


Figure 2.3. Effective Efficiencies $E^{simp.GM}$, $E^{simp.LM}$, $E^{subst.GM}$, $E^{subst.LM}$ for n =64000, p = 32, $n_{1/2}^{L}$ = 10, $n_{1/2}^{G}$ = 1000, plotted as a function of T.

Hockney-Jessope parameters, $n_{1/2}$ and the multiprocessor parameters p and S.

ADI Iterations

As an application of these results consider the solution of a system of finite difference equation arising from the solution of a partial differential equation on a two dimension square domain. The region is discretized as an n by n grid and the differential operator is approximated by a sparse matrix M of size n^2 by n^2. To find an approximate solution to the partial differential equation requires that we solve the equation $MX=Y$ where Y is a given n by n array of values. A common technique used to solve for X array is to view M as approximately factored into a product $A*B$ where the matrix A is a system of tridiagonal matrices linking the rows of the X array and B has the same structure but it links the columns of the X array. To find $X = B^{-1}A^{-1}Y$ requires us to first solve n tridiagonal systems

$$A^j Z^j = Y^j \qquad \text{for } 1 \le j \le n$$

where the superscript j refers to the jth column of the gird. Then the set of solution columns Z^j is viewed as a set of vectors in the solution of another n tridiagonal systems

$$B_j X_j = Z_j \qquad \text{for } 1 \le j \le n$$

This method is known as the Alternating Direction Implicit (ADI) method and is used in many applications [PeRa55]. We examine two solution schemes.

Assume the components of the arrays are stored by rows. The most natural partitioning of the algorithm is to substructure the system (A matrix , B matrix , and Y array) into blocks of size $\frac{n}{p}$ by n. Let $r_i = \frac{in}{p}$ and $s_i = \frac{(i+1)n}{p}$. The block $A_{[1:n]}^{[r_i:s_i]}$ is the set of coefficients corresponding to $\frac{n}{p}$ column equations and the block $B_{[1:n]}^{[r_i:s_i]}$ corresponds to components r_i through s_i of all n row equations. By "downloading" the data

$$(A_{[1:n]}^{[r_i:s_i]},\ B_{[1:n]}^{[r_i:s_i]},\ Y_{[1:n]}^{[r_i:s_i]}\ )$$

into the local memory of processor i, one may use the "simple" algorithm to compute

$$Z^{[r:s]} = (A^{[r:s]})^{-1} Y^{[r:s]}$$

in each processor using only local memory. Then, without "uploading" the Z array back to shared memory, it is possible to use the substructured method to solve the row equations

$$X^{[r:s]} = (B_{[1:n]})^{-1} Z.$$

The only use of the shared memory is to solve n reduced systems of size $2p$ required to complete the substructured elimination. The total time to complete this is

Step 1. Download data and solve column equations

$$7r_\infty^{-1}(\frac{n^2}{p} + n_{1/2}^{G}) + r_\infty^{-1}(8n-7)(\frac{n}{p} + n_{1/2}^{L})$$

Step 2. Do the substructured elimination and upload final results

$$17(\frac{n}{p} - 2)r_\infty^{-1}(n + n_{1/2}^{L}) + (16p-7)(\frac{n}{p} + n_{1/2}^{L}) + 8(n + n_{1/2}^{G}) + (\frac{n^2}{p} + n_{1/2}^{G})$$

The alternative is to use the simple algorithm for both column and row equations. Unfortunately, this requires that the partial solution vector Z be moved back to shared memory and read back to local memory in transposed order. Based on our memory addressing assumptions this step requires a minimum of $r_\infty^{-1}\frac{n}{p}(n + n_{1/2}^{G})$ seconds.

Step 2'. Transpose Z and use the simple method and upload final results.

$$r_\infty^{-1}(8n-7)(\frac{n}{p} + n_{1/2}^{\ell}) + 2r_\infty^{-1}(\frac{n^2}{p} + n_{1/2}^{G}) + r_\infty^{-1}\frac{n}{p}(n + n_{1/2}^{G})$$

To determine the effective efficiency observe that to compute $B^{-1}A^{-1}Y$ requires at least $n(16n-14)$ operations. If the machine can be programmed to run at 100% efficiency the execution time would be

$$r_\infty^{-1}\frac{n}{p}(16n-14)$$

The asymptotic efficiency for the substructured algorithm is 50% (computed in the limit as n goes to infinity) and 62% for the transposed simple method. In the case that n is small (near or below $n_{1/2}^{G}$) the substructured algorithm is superior. To illustrate this, consider the special case of p=32 processors, $n_{1/2}^{G} = 1000$ and $n_{1/2}^{\ell} = 10$. Figure 2.4 depicts the efficiency as a function of n. The cross point at which both methods are equal is when n is approximately 12000. In general, one can show that the substructured algorithm is superior to the simple scheme when

$$n \le p(\frac{1}{3}(1- \frac{4}{n})n_{1/2}^{G} + \frac{4}{3}(1- \frac{p}{n})n_{1/2}^{\ell})$$

III. Algorithm Analysis for Machines Based on Large Networks

In this section we consider systems built from a large number of simple processors interconnected by a communication network. Each processor contains local memory, but there is no global shared memory. These architectures can be based on a variety of types of communications networks. These networks can be of fixed topology, such as a ring or mesh, or can be packet switched or circuit switched networks. Numerous examples exist. The Finite Element Machine [Jord78] is a mesh connected lattice of 36 processors. The Cal-Tech Cosmic Cube contains 64 processors connected as a binary 6-cube. The Non-Von (Columbia Univ.) is a tree of processors. The CHiP architecture is lattice of processors [Snyd82] interconnected via a circuit switching network which can be configured as any member of a large family of graphs. The Boolean Vector Machine (Duke University) is an implementation of the Cube Connected Cycle network [PrVi82]. These machines are all non-shared memory MIMD architectures based on large communications networks.

In all of the above machines, no shared memory is used, and all interprocessor communication takes place via explicit interprocessor communication steps. In other cases a processor connection network is used to emulate a crossbar switch. Examples in this category the CDC Cyber plus which is a network of four rings with 16 processors in each ring. Zmob (Univ. of Maryland) and Crystal (Univ of Wisconsin) are based on ring architectures. A host of other systems share properties that are similar to the machines cited, but have a global address space and share data through a processor to memory permutation network. These systems include the TRAC system [LiTr77], PASM [Sieg79], Cedar[GKLS82], and the ULTRA Computer [GoSc82]. These systems are considered in section 4.

Our principal focus in this section is on nonshared memory machines where data movement is explicitly controlled by the processors and limited by the

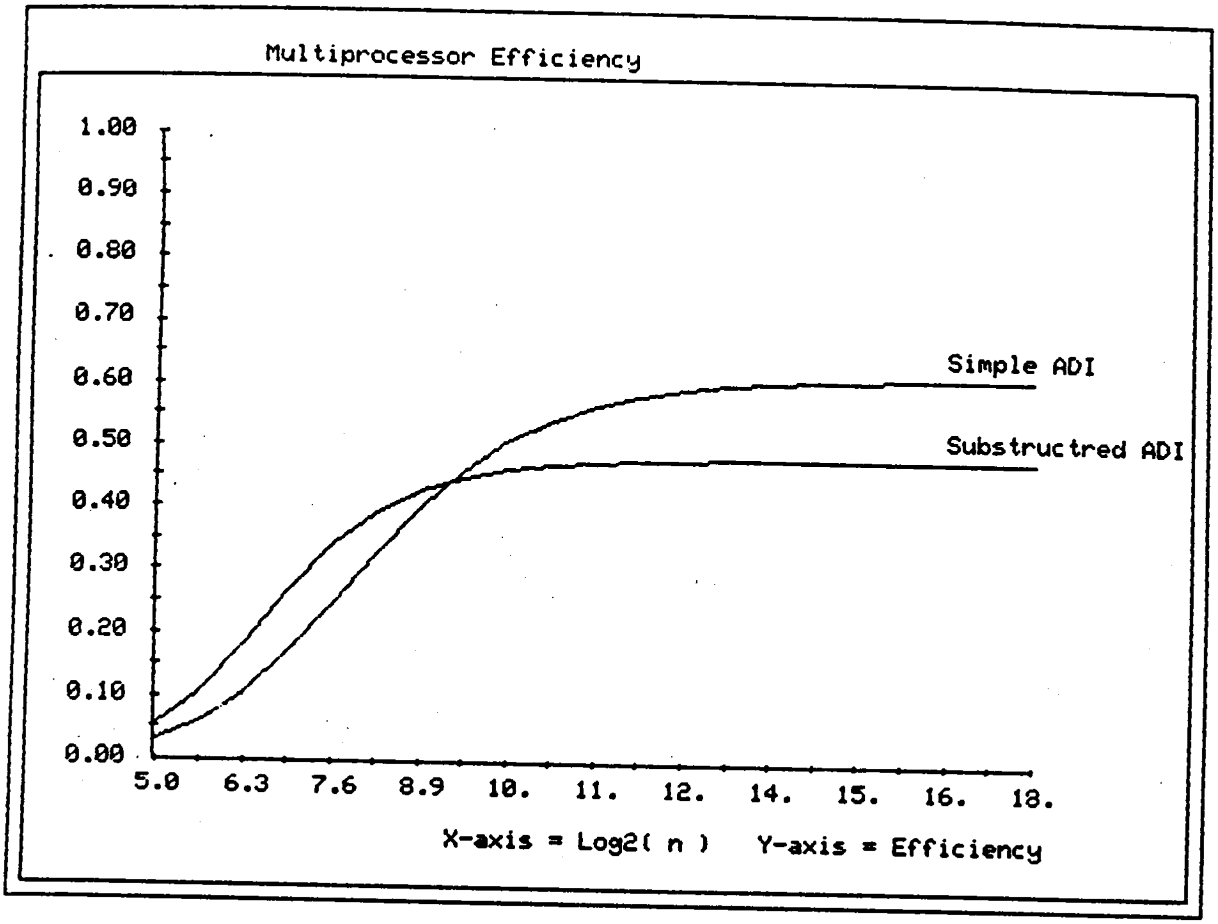


Figure 2.4. Algorithmic Efficiency as a function of n for $n_{1/2}^{G} = 1000$, $n_{1/2}^{L} = 10$ and $p = 32$.

topology of the network. The primary result presented here is that techniques developed for VLSI complexity analysis can be used to derive algorithm independent upper bounds on speed-up and effective efficiency that depend only on the problem being solved and the network topology. The upper bounds are derived for three problems: Fourier Transforms, Tridiagonal systems of Equations, and two dimensional elliptic boundary value problems. To prove that many of the bounds are exact, we describe the optimal algorithms. The system hardware will be modeled by the following rules:

1. The system is composed of p processors where p could be very large, $p \geq 32$.
2. Each processor has a sizable local memory and there is no shared memory.
3. Each arithmetic operation takes α seconds. Initiation or receipt of a data transmission requires β seconds per word of data, and receipt of a message can be done immediately after transmission, or at any time thereafter. Processors do not "overlap" communication with arithmetic. (This is the primary focus of section 4.)
4. The processors communicate with each other along paths that correspond to edges of a fixed connection graph. If the graph is complete, a crossbar network is modeled. If the graph is not complete, communication between processors not connected by an edge must be broken down into a sequence of message transmissions between processors along a path connecting the origin and the destination processors.

In the paragraphs that follow we examine the performance of algorithms that have been implemented on this class of architectures, paying particular attention to the structure and cost of communication.

The Cost of Communication.

Let A be a parallel algorithm and let M be a machine with p processors. We can describe the interconnection topology of the machine M as a graph $G(M)$. Similarly, the data flow graph of the algorithm A can be defined as the graph $G(A)$ which is the directed acyclic graph whose nodes represent the operations in A, and whose arcs represent operand data dependencies. By an implementation of A on M we mean a mapping

$$im : G(A) \rightarrow G(M)$$

where the operations of G(A) are mapped to processors and the communication arcs map to $G(M)$ in one of three possible ways. Let a and b be operators in $G(A)$ that are connected by an arc c. Assume a and b are mapped to processors p_a and p_b respectively. The three possible mappings of the arc c are

1. Processors p_a and p_b coincide. In this case arc c becomes a self-loop and no interprocessor communication is need to implement this arc.
2. Processors p_a and p_b are connected by an arc in the graph $G(M)$. In this case p_a transmits a value to p_b to implement arc c.
3. Processors p_a and p_b are not connected in the graph $G(M)$. In this case the arc c must be mapped into a path

$$p_a = p_1, p_2, \cdots, p_k = p_b$$

where p_i is connected by an arc in $G(M)$ to p_{i+1}. That is the communication along arc c is mapped into a sequence of interprocessor communications need to relay the message.

The time required to perform the communication represented by arc c will vary depending on which of these cases applies. In the first case, no communication occurs. In the second case, the message must be sent and received, requiring time 2β. In the third case, where the message is relayed k times, the communication time becomes $2k\beta$. In this third case, performance of all processors along the path is degraded by the time spent forwarding messages.

Let $T_M^A(\alpha,\beta)$ be the time required to execute algorithm A on machine M. It is often the case that algorithms designed for this class of architecture take the form of a loop with two steps

```
Repeat
    1. Permute the data via the Communication network.
    2. Execute a set of arithmetic functions in parallel.
until done;
```

(Though all parallel algorithms can be put in this form, many have optimal implementations that violate this structure. This case is briefly considered in the next section.) $T_M^A(\alpha,0)$ is the total amount of time required to execute step 2 for all passes through the loop and $T_M^A(0,\beta)$ is the total amount of time required to complete the data routing. In this case we have

$$T_M^A(\alpha,\beta) \;=\; T_M^A(\alpha,0) + T_M^A(0,\beta)$$

In the general case, I/O can be generated in one processor while another is engaged in arithmetic and messages are moving through the wires. In this case it is possible (but not trivial!) to show that the equality above becomes a $\leq$. Hence, given a parallel program it is possible for us to put an upper-bound on execution time. For a given problem, we can ask what is the lower bound on the execution time for any algorithm running on machine M. Clearly, if we know the optimal serial algorithm SER, we have the bound

$$\frac{1}{p}T_1^{SER}(\alpha,0) \;\leq\; T_M^A(\alpha,0).$$

This bound is tight only if there is enough parallelism to exploit p processors. A technique devised by Thompson [Thom80] (and extended by many others [Agga83], [BrGo82], [CaMo81], [Leig81], [Sava81], [Viul80]) to study the area-time trade-off in VLSI design can be used to find lower bounds on $T_M^A(0,\beta)$. Define a bisector B of a graph G of n nodes to be a set of arcs of G whose removal separates G into two graphs of equal size (i.e. if n is odd, one subgraph is size $\frac{n}{2}+\frac{1}{2}$ and the other is of size $\frac{n}{2}-\frac{1}{2}$). Let b_G be the number of arcs in the minimal bisector. The bisection bandwidth,

$$\left\lceil \frac{b_{G(M)}}{\beta} \right\rceil$$

is the number of words per second that can cross any minimal bisector of $G(M)$. Let B be a minimal bisector of $G(M)$ and $im{:}G(A)\rightarrow G(M)$ be an implementation of A on M with the following property

> If A has n inputs and n outputs then im maps $\frac{n}{p}$ inputs and $\frac{n}{p}$ outputs to each of the p processors.

In other words, we assume the implementation "uniformly distributes" the problem. This condition implies the set $\mathrm{Im}^{-1}(B)$ is a bisector of the input and output nodes of $G(A)$. Let b_{Pr} be the size of the minimal bisector of $G(A)$ for all algorithms that solve the given problem Pr. We then have

$$\beta \left\lceil \frac{b_{Pr}}{b_{G(M)}} \right\rceil \leq T_M^A(0,\beta).$$

This inequality characterizes the communication "bottleneck" imposed by the network topology induced bandwidth constraints. Network delay can also play an important role. A function with n inputs and n outputs is said to be transitive if every output is a nontrivial function of every input. For any transitive funçtion that has been uniformly distributed over p processors, the minimal time that information about each input can be propagated to each output is $2\beta log(p)$. In this case we have

$$\alpha \frac{T_1^{SEQ}(1,0)}{p} + \beta \max \left[\left\lceil \frac{b_{Pr}}{b_{G(M)}} \right\rceil , \ 2log(p) \right] \leq T_M^A(\alpha,\beta)$$

Values of b_{Pr} have been derived for a wide variety of problems. We consider three problems important for scientific computation.

1. FFT_n: an FFT on n complex numbers $X[0;n-1]$.
2. TRi_n: The direct solution of a tridiagonal matrix of size n.
3. Ell_n: The direct solution of the n linear equations obtained by a simple approximation (5 or 9 point star) of a second order elliptic boundary value problem on a unit square discretized as a $\sqrt{n}$ by $\sqrt{n}$ grid.

The first problem here is well known ([Thom80]), but the second two have not been studied in this context and it is interesting to note that the same proof applies to all three problems.

Lemma 3.1.

Assume that n is even and each of the problems three FFT_n , TRi_n, and Ell_n is solved by algorithms where the inputs $X[0;m]$ and outputs $Y[0;m]$ are equally distributed over all processors. Then all three problems are transitive functions of their inputs and

$$b_{FFT_n} = n; \quad b_{TRi_n} = 2; \quad b_{Ell_n} = 2\sqrt{n};$$

Proof:

Each of the problems above can be viewed as the problem of solving a matrix equation of the form

$$Ax = y$$

for a given n by n invertible matrix A. The basic idea is as follows: any bisector will divide the flow graph of an algorithm into two "machines" where one machine has one half the input vector, call it y_1, and the other machine has the other half, y_2. The bisection width of the problem is defined to be the minimal amount of "information" about y_1 that machine 1 must send to machine 2 plus the "information" about y_2 that machine 2 must send to machine 1. The theorem states that, for example, no algorithm exists for directly solving elliptic boundary value problems for which the the information flow between the two halves of the system falls below $2\sqrt{n}$ words for all input vectors y. The formal proof requires a formal definition of information and algorithm. Assume, without much loss of generality, that A and y have values that are rational numbers. Define an algorithm to be any finite sequence of tests and branches and rational arithemetic operations (+,-,*,/). In other words, an algorithm is assumed to be a piecewise rational function of the inputs. The basic unit of information will be a rational number.

Let $\bar{A}$ denote the matrix obtained from A by permuting the rows and columns so that the components of x and y corresponding to machine 1 are the first $\frac{n}{2}$ rows and the components of x and y corresonding to machine 2 form the bottom half of the system. The linear equation can now be written in the form

$$\bar{A}\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} A_1 & B_1 \\ B_2 & A_2 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \end{bmatrix}$$

The inverse of $\bar{A}$ exists and, can be decomposed into blocks of size $\frac{n}{2}$ by $\frac{n}{2}$ as follows:

$$\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} C_1 & D_1 \\ D_2 & C_2 \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \end{bmatrix}$$

or

$$x_i = C_i y_1 + D_i y_2 \quad i = 1,2.$$

We pose the question: How much information about y_2 must be known to compute x_1? Let $G(y_2)$ be the part of the algorithmin in machine 2 that encodes the minimal amount of information about the vector y_2 needed to compute x_1 given y_1 and let F be algorithm in machine 1 used to compute x_1 given $G(y_2)$.

$$x_1 = F(y_1, G(y_2))$$

Setting $y_1 = 0$, we have a new function $\bar{F}$ defined by

$$x_1 = \bar{F}(G(y_2)) = F(0, G(y_2)).$$

Bu then

$$D_1 y_2 = \bar{F}(G(y_2)).$$

If G returns k values it follows that

$$k \geq rank(D_1)$$

The same argument shows that the minimum amount of information about y_1 that one needs to compute x_2 is $rank(D_2)$. The minimal bisection width b_P is then given by

$$min(rank(D_1) + rank(D_2))$$

over all row and column permutations of the matrix A. Because the inverse of A exists and the blocks are of equal size, it can be shown that $rank(D_i) = rank(B_i)$ for $i=1,2$. To complete the proof we make the following observations

1 For a tridiagonal system B_i contains only one element and, hence has rank 1

2 The block tridiagonal system obtained by the natural order of a finite difference operator has $rank(B_i) = n^{1/2}$ which, the reader can verify, can not be reduced by any row or column permutations.

3 The FFT matrix is composed of n column vectors that are of the form

$$A^i = [1, \vartheta^i, \vartheta^{2i}, \cdots \vartheta^{(n-1)i}]^T.$$

where ϑ is a primative root of unity. Selecting any $\frac{n}{2}$ rows from any $\frac{n}{2}$ columns is easily shown to have rank $\frac{n}{2}$. **QED**

Upper Bounds on Efficiency.

Asymptotic lower bounds on the operation counts are well known for each of these problems.

$$T_1^{FFT} = 2n\ log(n), \quad T_1^{TRi} = 8n-7, \quad T_1^{Ell} = Cn\ log(n)$$

The constant 2 in the FFT algorithm assumes a complex operation requires only 1 time unit. In terms of operations on real numbers only, the formula is $5n\ log(n)$. The constant C depends on the special properties of the equation. In the case Ell_n we restrict our attention to the family of implementations of Fast Poisson Solvers [BuGo70, SCKu76]. In this case the lower bound $b_{FPS_n} = b_{Ell_n}$ and the arithmetic serial complexity is

$$T_1^{FPS} = 2n(log(n)+4) - 7$$

The resulting lower bounds on parallel complexity can be expressed as upper bounds on effective efficiency. Letting $r = \frac{\beta}{\alpha}$ the bound are

$$E_M^{FFT} \le \frac{1}{1+\frac{rp}{2nlog(n)}\max\left[\left\lceil\frac{n}{b_{G(M)}}\right\rceil, log(p)\right]}$$

$$E_M^{TRi} \le \frac{1}{1+\frac{rplog(p)}{4n}}$$

$$E_M^{FPS} \le \frac{1}{1+\frac{rp}{2nlog(2n)}\max\left[\left\lceil\frac{2n^{1/2}}{b_{G(M)}}\right\rceil, log(p)\right]}$$

To apply these bound to specific processor connections, we need only specify G(M) and determine $b_{G(M)}$. Figure 3.1 illustrates 5 graphs:

Com, the complete graph on p processors; $b_{G(Com)} = p$.

Shuf, a reduction of the shuffle-exchange graph on $2p$ processors obtained by identifying pairs of adjacent processors connected by exchange edges; $b_{G(Shuf)} = p$.

Ring, a ring of p processors; $b_{G(Ring)} = 2$.

Tree, a tree of $p-1$ processors; $b_{G(Tree)} = 1$.

Mesh, a $p^{1/2}$ by $p^{1/2}$ octagonally connected grid of processors; $b_{G(Mesh)} = p^{1/2}$.

Given the Efficiencies, an upper bound on the speed-up S_M^A can be derived using the relation $S_M^A = E_M^A$. Using the relation $SP_M^A = pE_M^A$, one can derive an upper bound on speed-up performance for each of these network topologies. This information is given in Table 3.1. These bounds are exact, up to constant terms, as functions of n. Their primary short comings are that they often underestimate communication costs by a factor of $log(p)$ for networks of large bandwidth. A better model of the delay seems to be needed. In the case of the FFT, the communication delay term is, in fact, $\frac{n}{p}log(p)$ not just $log(p)$ as we have indicated.) To show that some of these upper bounds are tight, we now construct algorithms and consider each case in turn.

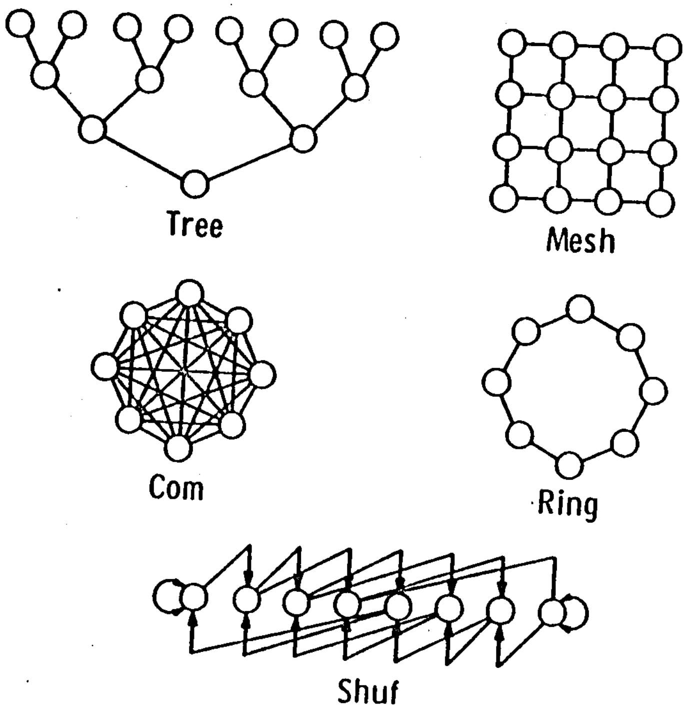


Figure 3.1 Network Connection Graphs.

	FFT	Tridiagonal System	Elliptic PDE
Tree	$= \dfrac{p}{1 + cr \dfrac{p}{log(n)}}$	$= \dfrac{p}{1 + cr \dfrac{plog(p)}{n}}$	$= \dfrac{\bar{c}\, p}{1 + cr \dfrac{p}{n^{1/2} \log n}}$
Ring	$= \dfrac{p}{1 + cr \dfrac{p}{log(n)}}$	$<< \dfrac{p}{1 + cr \dfrac{plog(p)}{n}}$	$< \dfrac{\bar{c}\, p}{1 + cr \dfrac{p}{n^{1/2} \log n}}$
Mesh	$= \dfrac{p}{1 + cr \dfrac{p^{1/2}}{\log n}}$	$<< \dfrac{p}{1 + cr \dfrac{plog(p)}{n}}$	$= \dfrac{\bar{c}\, p}{1 + cr \dfrac{p^{1/2}}{n^{1/2} \log n}}$
Shuf	$< \dfrac{p}{1 + \dfrac{cr}{log(n)}}$	$= \dfrac{p}{1 + cr \dfrac{plog(p)}{n}}$	$<< \dfrac{\bar{c}\, p}{1 + cr \dfrac{1}{n^{1/2} \log n}}$

Table 3.1 Bisection Width Speed-up Upper Bounds.

The value $r = \frac{\beta}{\alpha}$ and the constants c and $\bar{c}$ are both less than 1. Terms preceded by < are exact if r is replaced by $r\ log(p)$. The prefix ≪ indicates the best known methods are substantially slower and = means that the bound is exact for the appropriate choice of the constants.

The Fast Fourier Transform.

The FFT algorithm on a problem of size n can be defined in many ways. Here we follow [AHU174]. The algorithm is expressed in terms of a sequence of $log(n)$ permutations. The k^{th} term in this sequence is defined on a vector of length n, $X[0;n-1]$ by the expression,

$$Bfly_k(X) = X_{p_k(0)}, X_{p_k(1)}, \cdots, X_{p_k(n-1)}$$

where

$$p_k(j) = 2^k \left\lfloor \frac{j}{2^k} \right\rfloor + (j + 2^{k-1}\ mod\ 2^k).$$

These "butterfly" permutations are concatenated to form the flow graph of the FFT algorithm as shown in Figure 3.2. This butterfly graph has an interesting property. If p divides n we can group the columns into p blocks of $\frac{n}{p}$ adjacent columns each. Then the resulting "quotient graph" of the butterfly graph $Bfly_k$ is the butterfly graph $Bfly_{k-log(p)}$. (See [FiFi82], [Degr83] for many other useful quotient graph relations). Assuming that inputs $X_{(j-1)r}$ to X_{jr-1} all reside on processor j for $j=1,p$, we can write the FFT algorithm as

```
FFT( X[0;n-1] );
    for i = log(n) downto 1 do begin
```

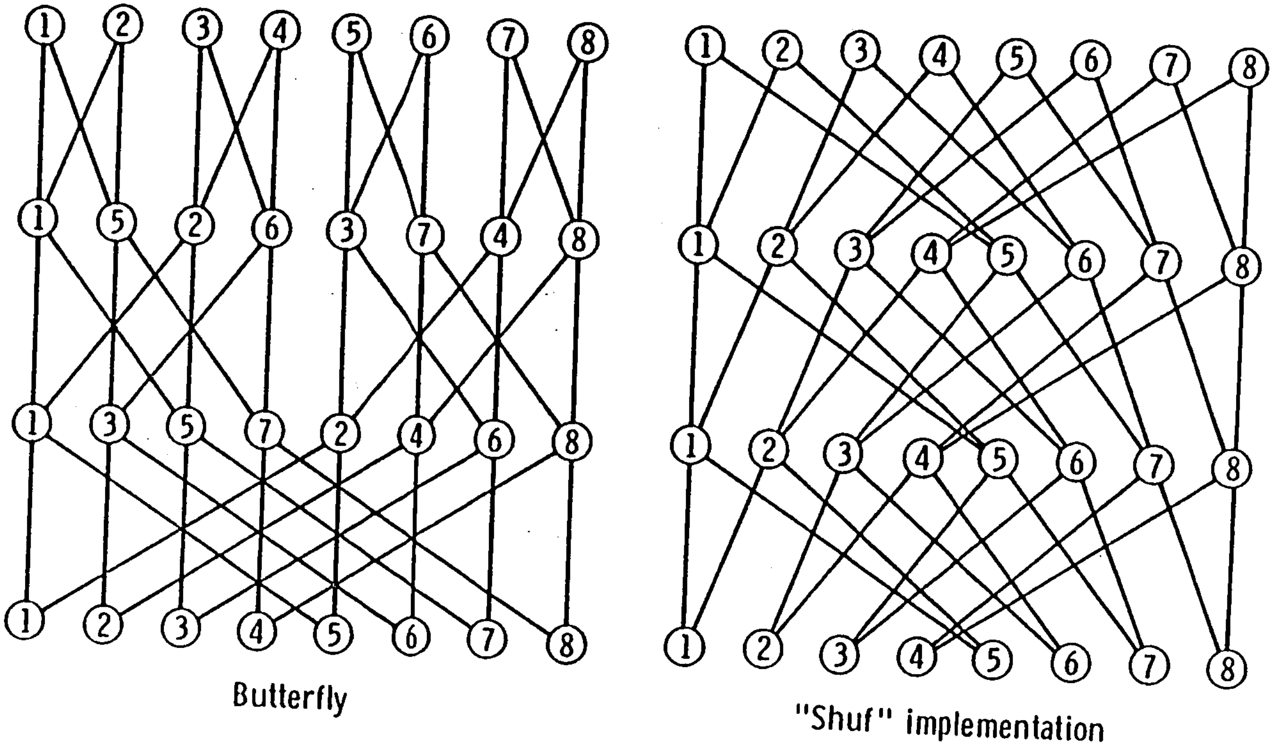


Figure 3.2 Butterfly based FFT flow graph and equivilent Shuffle implementation.

```
          Y = $Bfly_i$(X);
          pardo(j = 1,p )
               for k = (j-1)n/p to jn/p - 1 do
                    $X_k = X_k + \vartheta_{i,k} * Y_k$;
          endpar;
     end;
end ;[1]
```

The values $\vartheta_{i,k}$ are powers of primative roots of unity that are given in [AHU174] and are not of concern here. As index variable i runs from $log(n)$ down to 1, the permutation $Bfly_i$ can be accomplished using the interprocessor connection $Blfy_{i-log(n/p)}$. For values of $i \leq log(n/p)$ no interprocessor communication is involved. For the other values of i, there are $\frac{n}{p}$ data items to be sent and received by each processor, so $\frac{n}{p}$ communication steps are required. This requires time $\beta\frac{2n}{p}$on the *Com* network.

It can be shown that the $log(p)$ stage butterfly network is topologically equivalent to the $log(p)$ passes of the *Shuf* permutation. (The rearrangement is shown in Figure 3.2 and a formal proof is given in [Park80]). Consequently, on machines with *Com* and *Shuf* interconnection topologies, the execution time is approximately,

$$T^{FFT}(\alpha,\beta) \;=\; \alpha\frac{2n}{p}log(n) \;+\; (\beta\frac{2n}{p})log(p).$$

with speed-up

$$SP^{FFT} \;=\; \frac{p}{1 + r\frac{log(p)}{log(n)}}$$

To emulate the permutation $Bfly_k$ on a ring network requires uniform shifts of distance $\pm 2^{k-1}$. With each such permutation requiring $2\beta 2^k$ seconds the speed-up is found to be

$$SP^{FFT}_{ring} \;=\; \frac{p}{1 + r\frac{p}{log(n)}}$$

On a mesh connected computer the uniform shifts of distance 2^{k-1} can be done in time $\beta 2^{k-\frac{1}{2}log(p)}$ (see [ThKu77]) and the speed-up is

$$SP^{FFT}_{Mesh} \;=\; \frac{p}{1 + r\frac{p^{\frac{1}{2}}}{log(n)}}$$

In the case of the *Ring* and the *Mesh*, the speed-up agrees with the theoretical upper bound and must be optimal. In the other cases, we feel the algorithm is optimal and the speed-up bound is too generous.

Figure 3.3 depicts the relative efficiencies of the FFT algorithm on the three networks described above in the case that p = 512 and r=1.0.

[1] We have omitted the usual terminating "bit reversal" permutation to simplify the discussion. Inclusion of this permutation would have only a minor effect on the results here.

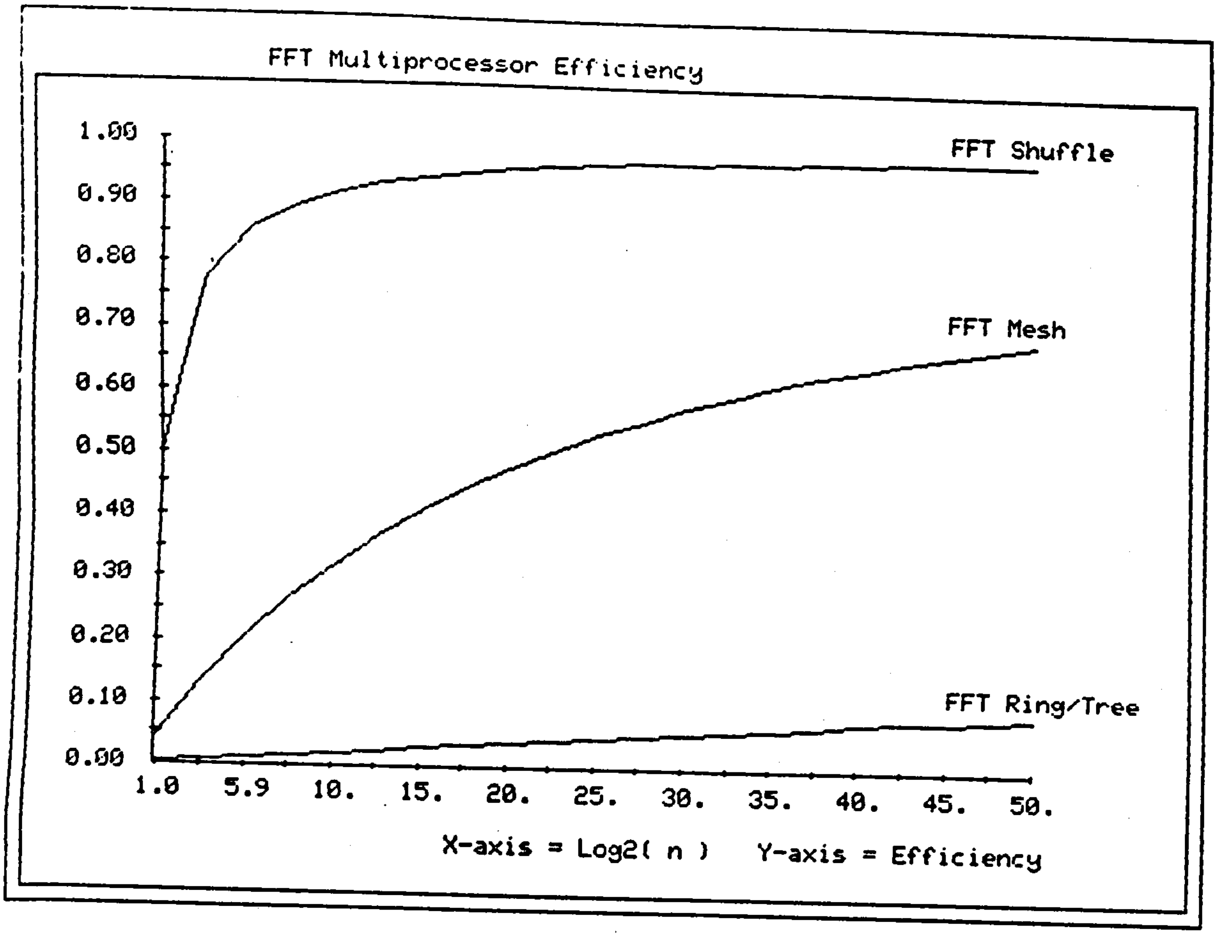


Figure 3.3 Multiprocessor Algorithm Efficiency for FFT on three networks, Shuffle connection, Mesh, and Ring each with $p = 512$, $\tau = 1.0$.

Tridiagonal Systems

To solve the tridiagonal system $AX = Y$ we assume that the inputs $Y[(i-1)\frac{n}{p}; i\frac{n}{p}-1]$ are located in processor i for $i \in [1..p]$. We employ the same solution strategy as used in section 2. By substructured elimination we reduce the $\frac{n}{p}$ equations in each processor to 2. This takes time

$$12\alpha(\frac{n}{p}-1)$$

and involves no interprocessor communication. To solve the reduced system of size $2p$ let M be a tree of $p-1$ processors with the root processor numbered 1 and the children of the i^{th} processor numbered $2i$ and $2i+1$. Assume the p pairs of equations are represented as p records $e[0;p-1]$ of the form

```
eqn = record
          a,b,c,y: array[0,1] of real;
      end;
```

After an operation of $log(p)$ communication steps we may assume the elements of the array e have been stored in the leaves of M with equation-pairs $e[2i]$ and $e[2i+1]$ $i \in [0, \frac{p}{2}-1]$ stored in processor $\frac{p}{2}+i$. To solve the system of tridiagonal equation on a tree of processors we use the following basic idea. Each internal node of the tree receives a pair of equations from its two decendants giving it 4 equations

$$b_i x_{j_{i-1}} + a_i x_{j_i} + c_i x_{j_{i+1}} = y_i \quad i = 1,4$$

in 6 variables

$$x_{j_0}, x_{j_1}, \cdots, x_{j_5}$$

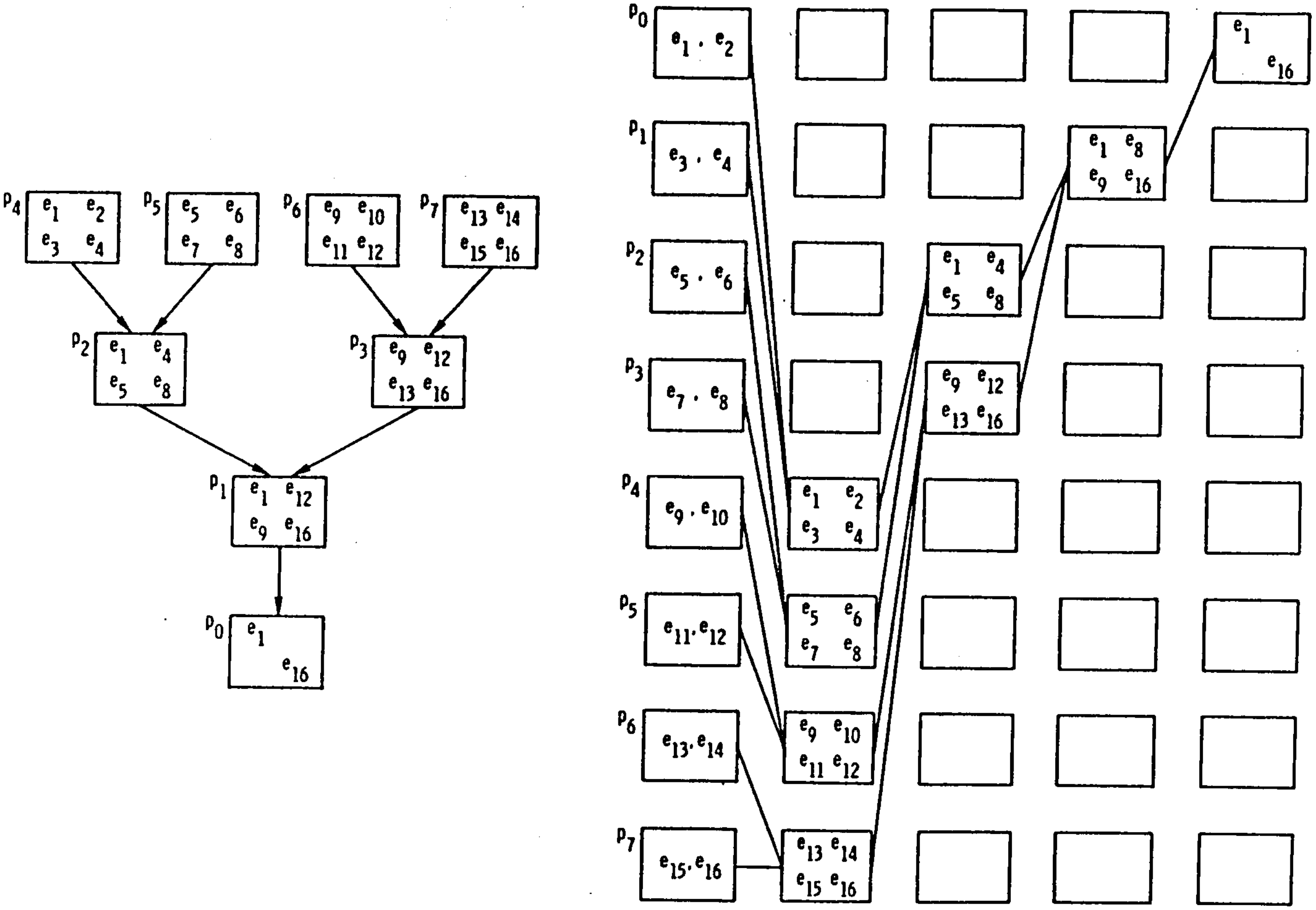
Let $elim()$ be a function called by the tree node that applies the substructured elimination computation to a set of four such equations and sends first and last of the new set

$$b_1 x_{j_0} + a_1 x_{j_1} + c_1 x_{j_6} = y_1$$
$$b_4 x_{j_0} + a_4 x_{j_4} + c_4 x_{j_5} = y_4$$

to its parent node and leaves the other 2 equations stored in the executing processor. To recursively reduce the $2p$ equations to 2 on the Tree connection we apply the function

```
function reduce(i: integer): eqn;
begin
    on processor(i) do
        reduce := if (i >= p/2 )
                    elim(e[2i-p], e[2i-p+1])
                else
                    elim(reduce(2i), reduce(2i+1))
end;
```

Figure 3.4 illustrates the movement of the variables through the tree. Once the 2 by 2 system has been solved a "backsolve" procedure can be used to move the solutions back to the leaf processors. The backsolve process requires 5 arithmetic steps per equation, and somewhat less communication than the forward elimination. Observe that this algorithm works with 4 equations per processor



p_4 e_1 e_2 e_3 e_4
p_5 e_5 e_6 e_7 e_8
p_6 e_9 e_{10} e_{11} e_{12}
p_7 e_{13} e_{14} e_{15} e_{16}
p_2 e_1 e_4 e_5 e_8
p_3 e_9 e_{12} e_{13} e_{16}
p_1 e_1 e_{12} e_9 e_{16}
p_0 e_1 e_{16}
p_0 $e_1 \cdot e_2$
p_1 $e_3 \cdot e_4$
p_2 $e_5 \cdot e_6$
p_3 $e_7 \cdot e_8$
p_4 $e_9 \cdot e_{10}$
p_5 $e_{11} \cdot e_{12}$
p_6 $e_{13} \cdot e_{14}$
p_7 $e_{15} \cdot e_{16}$
e_1 e_2 e_3 e_4
e_5 e_6 e_7 e_8
e_9 e_{10} e_{11} e_{12}
e_{13} e_{14} e_{15} e_{16}
e_1 e_4 e_5 e_8
e_9 e_{12} e_{13} e_{16}
e_1 e_8 e_9 e_{16}
e_1 e_{16}

Figure 3.4 Recursive reduction of 8 equations to 2
and *Shuf* implementation

while other parallel tridiagonal system solvers need only three equations in each process. The advantage of this algorithm is that we can do the communication on the *Tree* while the standard methods require a more elaborate connection network.

Taking both the elimination and the backsolve into account, the execution time is found to be:

$$T_{tree}^{Tri\_n}(\alpha,\beta) \; = \; \alpha(17\frac{n}{p} + 34\,log\,(p) + 9) + 17\,\beta\,log\,(p)$$

and the speed-up is approximately

$$Sp_{tree}^{Tri\_n} \; = \; \frac{\frac{8}{17}p}{1 + (2 + r)\,\frac{p log\,(p)}{n}}$$

This differs from the theoretical upper bound for this problem by the factor $\frac{8}{17}$ in the numerator, and by the factor 2 in the communication term. The later is due to extra arithmetic caused by "matrix fill" associated with the the substructed elimination and the constant 2 comes from redundant arithmetic used to solve the reduced system.

To execute this algorithm on the *Shuf* network we need only embed the tree structure in the target graph (as shown in Figure 3.4) and the execution time and speed-up will be the same as on the tree. The tree has the property that any node can be reached in at most $2log\,(p)$ steps from any other node, consequently no direct embedding of the tree is possible in the ring or mesh. We do not know of an algorithm for these networks that is within a factor of $log\,(p)$ of the correct communication complexity.

The tree computation above proceeds as a wave from the leaf nodes to the root. If we consider the problem of solving m tridiagonal systems of size n (m_Tri_n), we can pipeline the method above. The execution time is

$$T_{tree}^{m\_Tri\_n}(\alpha,\beta) \; = \; 17\alpha(\frac{nm}{p} + 2(n - m - 1) + 3log\,(p)) \; + \; 9\alpha \; + \; 30\beta(n \, + log\,(p))$$

This result will be used below.

Fast Poisson Solvers

There are a wide variety of interesting direct solvers for the poisson equation

$$\nabla^2 x \; = \; y$$

(see [HoJe81], [Gros79], [SCKu76]). The method here is the easiest to explain. Consider a grid of $n^{1/2}$ by $n^{1/2}$ points. The basic fast poisson solver operates in three steps on an array of values y.

1. Apply an FFT to each row of the grid of y values. ($\bar{y} = Row\_FFTs\,(y)$).
2. Solve n systems of tridiagonal equations of size n using the columns of the grid of $\bar{y}$ values as the right hand sides.
3. Apply an inverse FFT to each row of the solution of the previous step.

With Dirichlet boundary conditions, the FFTs used here are actually real sin transforms which can be computed with the complex FFT algorithm (see [HoJe82] for details). Assume we have p processors to execute this algorithm. Let k be a divisor of p and partition the problem so that the columns are divided into k groups and the rows are divided into $\frac{p}{k}$ groups. We can then assign each processor to a block with $\frac{n^{½}}{k}$ columns and $n^{½}\frac{k}{p}$ rows. In steps 1 and 3 each row of k processors must compute $n^{½}\frac{k}{p}$ FFTs of size $n^{½}$. Using the algorithms above, the minimal execution time for both steps is

$$T^{FFTs} = 2\alpha\frac{nlog(n)}{p} + 4\beta\frac{n}{p}log(k).$$

In step 2, each column of $\frac{p}{k}$ processors must solve $\frac{n^{½}}{k}$ tridiagonal systems of size $n½$. As we have seen, this step takes

$$T^{Tri} = 17\alpha(\frac{n}{p} + 2log(\frac{p}{k})) - 25\alpha + 30\beta(\frac{n^{½}}{k} + log(\frac{p}{k}) - 1)$$

To choose the optimal partitioning of the problem we minimize $T^{FFTs} + T^{Tri}$ as a function of $k \in [\frac{p}{n^{½}}, min(p,n^{½})]$. The function takes the form

$$R + 30\beta\frac{n^{½}}{k} + (4\beta\frac{n}{p} - 34\alpha - 30\beta)\ log(k) \qquad (3.2)$$

where R is independent of k. Minimizing 3.2 as a function of k is, in general, not easy. There are two interesting cases to consider.

Case 1. $\frac{n}{p} \le \frac{17}{½}\frac{\alpha}{\beta} + \frac{15}{2}$.

In this case the last term in 3.2 is negative, thus we pick k as large as possible. If $p < n^{½}$ then we set $k = p$ which implies that it is best to distribute the FFTs across all p processors and to solve the columns of tridiagonal systems without communication ($\frac{n^{½}}{p}$ columns per processor.) If $n^{½}$ or p is greater than $\frac{17}{2}\frac{\alpha}{\beta} + \frac{15}{2}$ we find $n^{½} < p$. Setting $k = n^{½}$ the FFTs are again distributed and the execution time is

$$T^{FPS} = \alpha(\frac{2nlog(n)}{p} + \frac{17n}{p} + 34log(\frac{p}{n^{½}}) + \beta(\frac{2nlog(n)}{p} + 30log(\frac{p}{n^{½}}))$$

Case 2. $\frac{n}{p} > \frac{17}{2}\frac{\alpha}{\beta} + \frac{15}{2}$.

The optimal value for k is found to be

$$k = max(\ 1,\ (\frac{2}{15}\frac{n^{½}}{p} - n^{-½}(1 + \frac{34}{30}\frac{\alpha}{\beta}))^{-1}\ )$$

In the case that $n^{½} \gg \frac{15p}{2}$ we have $k = 1$ and the execution time is

$$T^{FPS} = \alpha(\frac{2n}{p}(log(p) + \frac{15}{2}) + 34log(p)) + 30\beta(n^{½} + log(p))$$

Setting $k = 1$ implies that each row of the grid is stored completely in one processor and the FFTs involve no communication. Consequently, only the solution of the tridiagonal systems involve communication and the time bound above is valid for a *tree* network. The resulting speed-up is of optimal complexity. On the other hand, the theoretical lower bound for communication cost for an Elliptic problem is $\beta C\frac{n^{½}}{p^{½}}$ for a mesh network and $\beta C\frac{n^{½}}{p}$ for the *Shuf* network. The time estimate above suggests that this form of the Fast Poisson Solver is suboptimal for these networks.

Based on the upper bounds for speed-up in table 3.1, Figure 3.5 illustrates the relative performance of optimal Fast Poisson Solvers as a function of

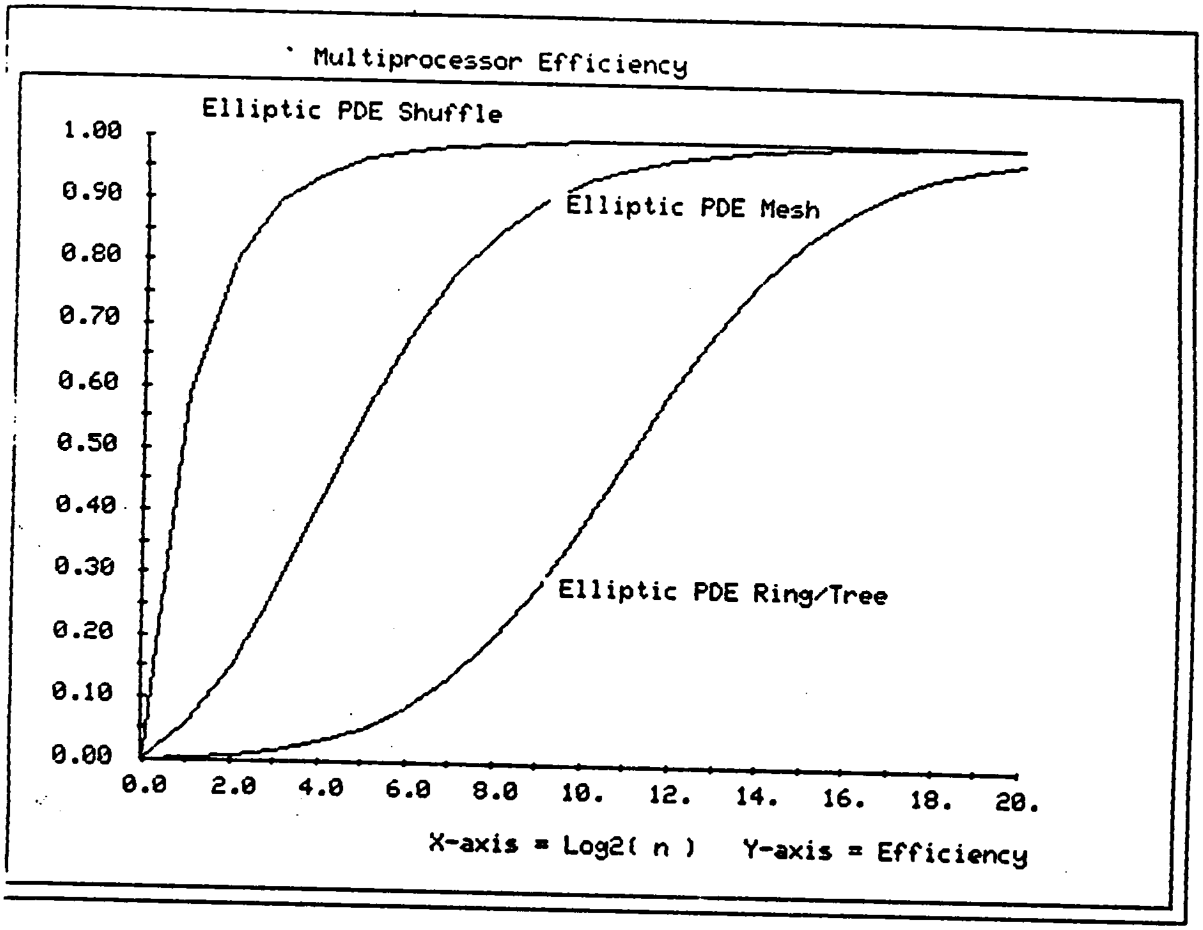


Figure 3.5 Efficiency upperbounds based on table 3.1 for Fast Poisson Solvers. $p = 512$, $r = 1.0$.

problem size (n) when p = 512 and r=1 for the three network topologies Ring, Mesh and Shuf.

Multi-Grid iterative methods [Bran81, GaVr82] are structured so that each stage of the iteration reduces a $n^{1/2}$ by $n^{1/2}$ problem to a problem of smaller size which is solved by a direct method. Let the initial problem be distributed such that square subgrid of size $(\frac{n}{p})^{1/2}$ by $(\frac{n}{p})^{1/2}$ are mapped to each processor in a mesh. The communication cost to reduce the original problem to one of size $p^{1/2}$ by $p^{1/2}$ is $\beta C_1(\frac{n}{p})^{1/2}$ for some constant C_1. Using the FPS method (case 1) to solve the reduced problem on the mesh requires an additional $\beta C_2 p^{1/2}$ communication steps. Applying K such iterations will reduce the error to a fixed level and cost

$$\beta K(C_1 \frac{n^{1/2}}{p^{1/2}} + C_2 p^{1/2})$$

which is of the optimal complexity. Other methods, such as the preconditioned conjugate gradient can also be shown to have this communication bound. On the other hand, the authors know of no method that achieves the lower bound of $\beta C\frac{n^{1/2}}{p}$ for the networks with high bandwidth.

4. Multistage Packet Switched Networks

The analysis of non-shared memory multiprocessors to this point assumed a machine M with a fixed or switchable connection topology $G(M)$. The only communication cost in this model was the time β taken by the processors to perform sends and receives. For most non-shared memory multiprocessors, this model is probably quite reasonable.

However, this model corresponds poorly to machines with multistage packet switched networks. An example of such a network is the Omega network of Lawrie [Lawr75]. With this type of network, messages can be broken into packets of uniform size where each packet consisits of a destination tag and a data field. The switches in the network read the destination tag and forward the packet along its route. An Omega network interconnection of p processors contains log(p) stages, each having $\frac{p}{2}$ switches. In the best case, a packet can be routed through the network in log(p) steps. However, when the network has heavy traffic, contention occurs and contending packets must be queued at the switches. Simulation results suggest packets are delayed by an average amount

$$C_0 log(p)$$

and that the number of packets transmitted per clock cycle is about

$$C_1 p$$

for any number of processors p. Thus the Omega network behaves much like a crossbar switch, except message propagation delays are relatively long [KrSn82], [GoSc82]. Performance of other log(p) stage packet networks, such as the Banyan network, baseline network, and so on, is comparable to that of the Omega network.

In principal, packet switched networks can be accommodated in our multiprocessor model by treating switches as specialized processors. Though feasible, this approach is difficult, since the communications patterns in packet networks are complex. A more illuminating approach is to view the packet switched network as a close emulation of a crossbar switch, and modify our multiprocessor model accordingly. The model given at the beginning of this section needs to be modified only in the two provisions governing communication:

3' Each arithmetic operation takes α seconds. Transmission or receipt of a word of data takes β seconds. Receipt of a message can be done γ seconds after its transmission or any time thereafter.

4' The connection topology is a complete graph.

The delay parameter γ here is designed to model the time taken to route and forward messages in the network, and the time packets spend queued at switches when there is contention. With this model, sending a one word message between two processors takes total time $2\beta + \gamma$. In sending a k word message, k sends and receives are required. But the propagation delays can be overlapped, so after receiving the first word, a new word can be received every β seconds, giving a total message delay of:

$$(k + 1)\beta + \gamma.$$

With this model of a multiprocessor interconnected by a packet switching network, it is possible to look at any of the algorithms already considered. We look here at fast Fourier transforms, since there are interesting aspects of this algorithm not yet treated. The communications required in an FFT can all be viewed as permuting data between processors. Suppose we have n words of data, with $\frac{n}{p}$ words per processor. Then we can ask, how much time is required to simultaneously move the data on every processor to some other processor. Let the time taken to perform this operation be denoted by $X_p(n)$.

To compute the value of $X_p(n)$, note that to move $\frac{n}{p}$ data words from one processor to another should take time

$$(\frac{n}{p} + 1)\beta + \gamma.$$

as discussed above. But this will be so only if the target processor is ready to receive the data as soon as it arrives there. In permuting n words of data, each processor must send $\frac{n}{p}$ and receive $\frac{n}{p}$ words, so the execution time for this permutation cannot be less than:

$$\frac{2n}{p}\beta$$

In fact, the time perform this permutation is

$$X_p(n) = \max[(\frac{2n}{p})\beta, (\frac{n}{p} + 1)\beta + \gamma] \qquad (4.1)$$

as one can easily verify.

Now consider the problem of performing an FFT on a vector of length n, with this multiprocessor model. The execution time will be:

$$\begin{aligned} T^{FFT\_n} &= \alpha(\frac{2n}{p})log(n) + X_p(n)log(p) \\ &= \alpha(\frac{2n}{p})log(n) + \max[(\frac{2n}{p})\beta, (\frac{n}{p} + 1)\beta + \gamma]\log(p) \end{aligned}$$

This is exactly the same as the execution time derived previously, except for the new term involving γ. Notice here that the vector length n does not multiply γ, so the impact of a large γ, caused perhaps by packet contention, is not as severe as one might expect.

Next consider the problem of performing multiple fast Fourier transforms. In treating fast Poisson solvers, it was implicitly assumed that to take fast Fourier transforms of m vectors, one would just repeat the parallel algorithm for a single FFT m times. This is not necessarily the best approach, especially on packet switched networks where one needs to contend with propagation delays. At least four reasonable approaches to performing m fast Fourier transforms can be found:

1 Repeat the parallel algorithm FFT_n, for a single data vector, m times.

2 Combine m invocations of FFT_n to overlap communication. Each step of the FFT would be performed on all m data vectors at once before proceeding to the next step.

3 If $m \le p$ and $m \mid p$, the data can be permuted so data vector i resides on processors $(i-1)\frac{p}{m}+1$ through $i\frac{p}{m}$. Then the algorithm FFT_n for a single data vector can be performed on each block of $\frac{p}{m}$ processors. Finally the results must be permuted back to their correct locations.

4 If $p \le m$ and $p \mid m$, the data can be permuted so each data vector resides on only one processor. Each processor then performs sequential FFTs on the $\frac{m}{p}$ data vectors it has, and finally the results are permuted back to their correct locations.

Now looking in detail at each of these, for the first approach, the execution time is just m times the execution time of FFT_n. That is:

$$T_1^{m\_FFT\_n} = \alpha(\frac{2mn}{p})log(n) + mX_p(n)log(p)$$

With the second approach, one will have only log(p) communication steps rather than $mlog(p)$ as in the first approach. However, each step is now a permutation on mn words of data rather than on n words as in the first approach. The execution time is thus:

$$T_2^{m\_FFT\_n} = \alpha(\frac{2mn}{p})log(n) + X_p(mn)log(p)$$

At first sight there appears to be little difference between the two approaches. However, the function $X_p(n)$ satisfies the inequality

$$X_p(mn) \le mX_p(n)$$

for all m, so the second approach is always at least as good as the first approach.

The third approach here is somewhat more complex. Two operations are involved, permuting the data, so each of the m vectors is distributed over a block of $\frac{p}{m}$ processors, and then performing FFTs on these processor blocks. The FFT algorithm needed here is just the FFT for a single data vector, FFT_n, already studied, except only $\frac{p}{m}$ processors are used now. The execution time to perform these FFTs on processor blocks is:

$$T_p^{FFT} = \alpha(\frac{2n}{p/m})log(n) + X_p(nm)log(\frac{p}{m})$$

where we have used the identity $X_{p/m}(n) = X_p(mn)$ which is easily derived from equation 4.1.

The other operation needed is permuting the m data vectors. Each data vector is originally distributed evenly over the p processors, and must be moved so it is distributed over a block of $\frac{p}{m}$ processors. The cost of this is:

$$T_{data} = \max\left[\frac{2mn}{p}(\frac{p-1}{p})\beta,\ (\frac{mn}{p}(\frac{p-1}{p}) + 1)\beta + \gamma\right]$$

The factors $\frac{p-1}{p}$ arise since a fraction of each vector is already in the proper processor memory and does not need to be transmitted. Assuming p is large, $\frac{p-1}{p}$ is close to unity and we can set:

$$T_{data} \sim X_p(mn)$$

The data needs to be permuted before and after performing the FFTs, so the total execution time becomes

$$T_3^{m\_FFT\_n} = \alpha(\frac{2mn}{p})log(n) + X_p(mn)(log(p) - log(m) + 2)$$

Analysis of the fourth algorithm is similar. No communication is involved in the FFTs in this case, but data permutations are required before and after the FFTs. The execution time is thus:

$$T_4^{m\_FFT\_n} = \alpha(\frac{2mn}{p})log(n) + 2X_p(mn)$$

One way to compare these four algorithms for computing m FFTs is to compute their speed-ups. The results are:

$$S_1 = \frac{p}{1+\max\left[\frac{\beta}{\alpha},\ \frac{\beta}{2\alpha} + \frac{p}{2n\alpha}(\beta+\gamma)\right]\frac{log(p)}{log(n)}}$$

$$S_2 = \frac{p}{1+\max\left[\frac{\beta}{\alpha},\ \frac{\beta}{2\alpha} + \frac{p}{2mn\alpha}(\beta+\gamma)\right]\frac{log(p)}{log(n)}}$$

$$S_3 = \frac{p}{1+\max\left[\frac{\beta}{\alpha},\ \frac{\beta}{2\alpha} + \frac{p}{2mn\alpha}(\beta+\gamma)\right]\frac{log(p) - log(m) + 2}{log(n)}}$$

$$S_4 = \frac{p}{1+\max\left[\frac{\beta}{\alpha},\ \frac{\beta}{2\alpha} + \frac{p}{2mn\alpha}(\beta+\gamma)\right]\frac{2}{log(n)}}$$

Comparing these equations it is clear that the first algorithm is never better than the second, as already mentioned, since the impact of γ is smaller in the second. Note that this conclusion applies only for the packet switched networks under consideration. For networks with fixed or circuit switched topology these two algorithms perform identically.

Figure 4.1 illustrates the efficiency in the case that $\alpha = \beta = 1.0$ and $\gamma = 50.0$, $p = 512$ and $n = 1024$. In this case we have plotted the preformance as a function of the number of equations, m. Observe that the third algorithm becomes better than the second when $m \geq 4$. Had we included the cost of the "bit reversal" permutation in all algorithms, the third algorithm would have become better even earlier. Between the third and fourth algorithms there is nothing to decide, since the third applies only to the case $m \leq p$ and the fourth to the case $m \geq p$. Figure 4.1 depicts these two methods as one with a transition at $m = p$.

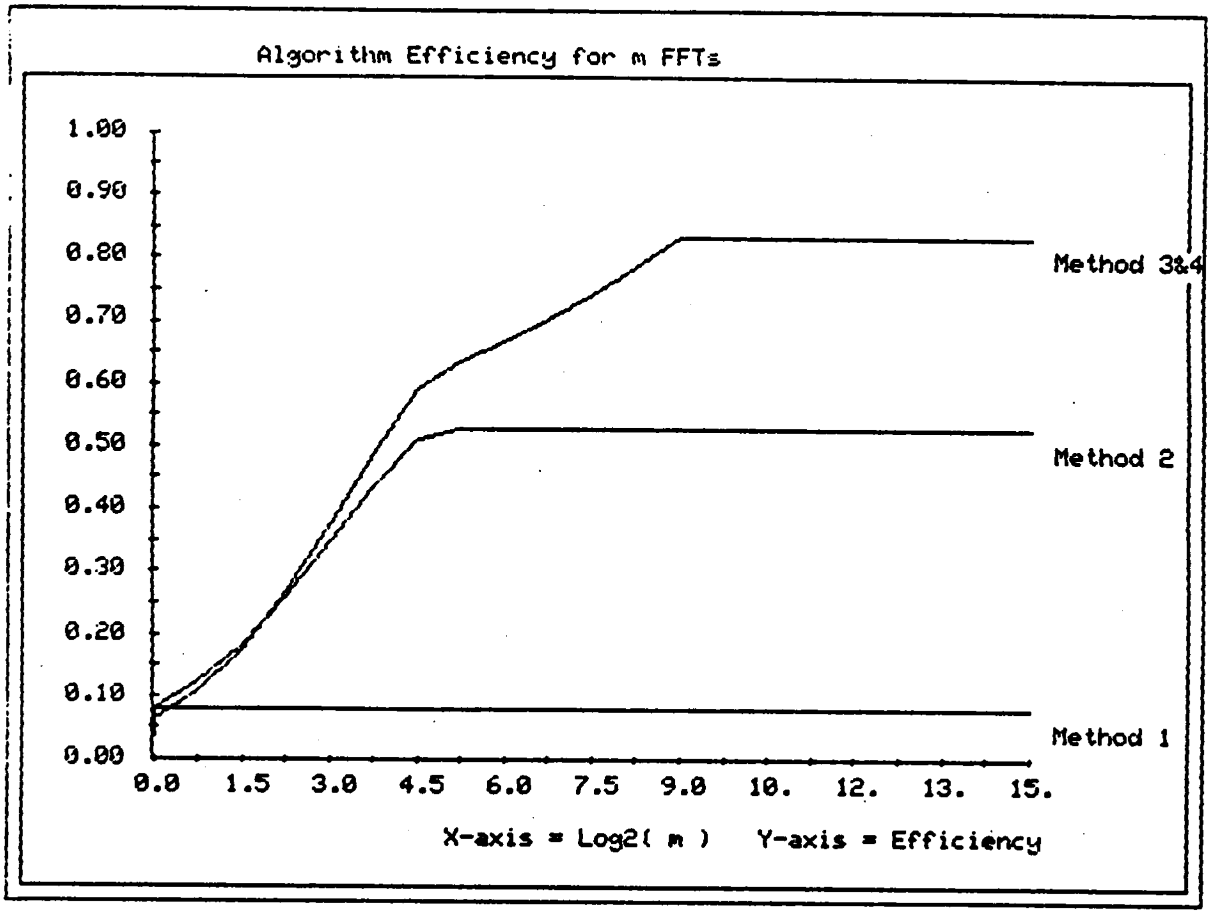


Figure 4.1 Efficiency as a function m for 4 methods to implement algorithms for doing m FFTs of size n. $p = 512$, $n = 1024$, $\alpha = \beta = 1.0$ and $\gamma = 50.0$.

Though searching for optimal algorithms is interesting, the real issue here is the impact of the delay γ caused by the use of a packet switching network. The effect of γ depends on the ratio of problem size to the number of processors; on problems with a great deal of computation, γ is well masked. In fact in the three FFT algorithms for multiple data vectors found to be best, γ always enters the execution time in the ratio:

$$\frac{\gamma}{2mn\alpha}$$

Though this analysis was performed only for FFT algorithms, experience suggests that the delays caused by packet switched networks are relatively unimportant on most compute bound problems.

5. Conclusion

This paper has considered three basic families of multiprocessors and the analysis of communication complexity in the algorithms for these architecture classes. The principal goal here was to look at communication and its impact on algorithm performance. For large shared memory multiprocessors analyzing communication turns out to be relatively straight forward. The main issues are memory latency and finding ways to organize or substructure problems to minimize its effect.

Studying algorithms on non-shared memory machines is more difficult, since the topology of the communication network is a central issue. Our analysis of non-shared memory network based machines was divided into two parts, the first covering machines with a fixed or circuit switched topology, the second covering machines based on packet switched networks.

On circuit switched machines, techniques borrowed from VLSI complexity theory provide a nice tool for obtaining lower bounds on algorithm complexity. Given an interconnection topology, one can with relative ease compute upper bounds on efficiency of the problem solution. An important point here is that these are upper bounds on the problem, (e.g. FFT, Fast Poisson Solve, direct solution of tridiagonal systems) not on any particular implementation of an algorithm for solving the problem. In the cases studied, these upper bounds are apparently quite tight; in two of the three cases studied these upper bounds are actually attained.

By contrast, analysis of algorithms on machines interconnected by a large packet switching network is far easier, given our simple model of the behavior of packet switching networks. Here the propagation delay parameter γ, modeling the impact of packet contention, is quite important. But on most large problems it seems to be possible to substructure the problem so that the effect of γ is minor. With our model of a packet switched network, in which such a network is treated as a crossbar switch with delay, analysis of algorithms is no more difficult than for shared memory multiprocessors. (In fact, the delay γ is closely related to the value $n\frac{G}{h}$.) At the moment this model rests only on heuristic considerations and simulation results, so it would be valuable to establish the precise circumstances under which it holds.

Many important problems remain to be solved. In particular, improved techniques are needed for lower bounds on communication in multiprocessors. In the case of specific algorithms, we do not know of better lower bounds (or better algorithms) in the case of elliptic PDEs on high bandwidth networks like the *Shuf* connection.

Of particular importance are issues that were not considered at all in this paper. This includes a systematic approach to algorithms with dynamic data structures, such as adaptive grid algorithms for PDEs. Do these problems have a reasonably nice solution on nonshared memory systems? If so, what is the structure of the communication? A closely related problem is the analysis of complexity of communication in Data Flow machines. How does it differ from the models we have surveyed in this paper?

6. References

[Agga83] Aggarwal, A., "Tradeoffs for Transitive Functions in VLSI with Delay," Proc. 23rd IEEE Symp on Found. of Comp. Sc., 1983, Tucson.

[AHU174] Aho, A. V., J. E. Hopcroft and J. D. Ullman, *The Design and Analysis of Computer Algorihtms, Addison-Wesley, (1974)*

[Bran80] Brandt A., "Multigrid Solvers on Parallel Computers", ICASE NASA Langley Research Center, Hampton, Va. Report No. 80-23, 1980.

[BrGo82] Brent, R., Goldshlager, L. "Some Area-time Tradeoffs for VLSI," SIAM J. of Computing. vol. 11, no. 4, nov 1982, pp. 737-747.

[BrKa81] Browne, J. and Kapur, R., "Block Tridiagonal Systems on Reconfigurable Array Computers," Proc. 1981 Intl. Conf. on Parallel Proc., Liu and Rothstein ed., pp.92-97, 1981.

[Buzb73] Buzbee, B. "a Fast Poisson Solver Amenable to Parallel Computation," IEEE Trans. on Computers Vol-22, pp. 793-796 (1973).

[BuGN70]Buzbee, B., Golub, G. and Neilson, "On direct Methods for Solving Poisson's Equation," SIAM J. Num. Anal. 7. 627-656.

[CaMo81]Chazelle, B. and Monier, L., "A model of Computation for VLSI with Related Complexity Results," Proc. 13th An. Symp. on Theory of Comp. 1981, pp. 318-325.

[DeGr83] DeGroot, D., "Expanding and Contracting SW-Banyan Networks," Proceedings of the 1983 International Conference on Parallel Processing. pp.19-24.

[FiFi82] Fishburn, J. and R. Finkel, "Quotient Networks," IEEE Transactions on Computers, Vol. C-31, #4, 1982.

[GaVR82]Gannon D., Van Rosendale J., "Highly Parallel Multigrid Solvers Elliptic P.D.E.s: An Experimental Analysis", ICASE NASA Langley Research Center, Hampton, Va. Tec. Report 1982.

[GKLS83]Gajski, D., Kuch, D., Lawrie d., Sameh, A., (1983) "CEDAR, A Large Scale Multiprocessor;" Proc. 1983 Intl. Conf. on Parallel Proc., pp.524-529, 1983.

[GeSa66] Gentleman, M., Sande, G., "Fast Fourier Transforms - for fun and profit. 1966 Fall Joint Com. Conf. AFIPS Proc. vol. 29. pp.563-578, 1966.

[GGKMRS]Gottlieb, A., Grishman, R., Kruskal, C. P., McAullite, K. P., Rudulf, L., and Snir, M. "The NYU Ultracomputer - Designing an MIMD Shared Memory Parallel Computer," IEEE Trans. on Computers Vol. 32, pp. 175-189 (Feb. 1983).

[GoSc82] Gottlieb, A., and J. T. Schwartz, "Networks and Algorithms for Very-Large-Scale Parallel Computation," IEEE Computer, (Jan. 1982).

[Gros79] Grosch, C., "Performance Analysis of Poisson Solvers on Array Computers," Supercomputers vol. 2, Hockney, Jesshope eds., pp. 149-81., Infotech, 1979.

[Hell78] Heller, D., "A Survey of Parallel Algorithms in Numerical Linear Algebra," SIAM Rev. vol.20, 740-777., 1978.

[HoJe81] Hockney R. W. and C. R. Jesshope, *Parallel Computers*, Adam Hilger Ltd., Bristol, Great Britain (1981).

[Jord78] Jordan, H., "A Special Purpose Architecture for Finite Element Analysis," Proc. of the 1978 International Conference on Parallel Processing, IEEE, 1978, pp. 263-266.

[KrSn82] Kruskal, C. P. and M. Snir "Analysis of Omega-Type Networks for Parallel Processing," Ultracomputer Note, Courant Institute, New York University. (1982)

[LaVo75] Lambiotte, J., Voigt, R. "The Solution of Tridiagonal Linear Systems on the CDC Star-100 Computer," ACM TOMS, vol. 1. pp.308-329, 1975.

[Lawr75] Lawrie, D. H., "Access and Aligment of Data in an Array Processor," IEEE Trans. on Computers Vol. 24, pp. 1145-1155 (Dec. 1975).

[Leig81] Lieghton, F. T., "New Lower Bound Techniques for VLSI," Proc. 22nd IEEE Symp. on the Found. of Comp., Oct. 1981, p.1-12.

[LiTr77] Lipovski, J., Tripathi, A., "A Reconfigurable Varistructured Array Processor," Proc. Intnl. Conf. on Parallel Processing., 1977, pp. 165-174.

[NoHu78]Noor, A., Hussein K., and Fulton, R., "Substructuring Techniques - Status and Projections," *Computers and Structures*, 1978, vol. 8, p.621-632.

[Park80] Parker, D. S. Jr., "Notes on Shuffle/exchange-Type Switching Networks," IEEE Trans. on Comp. C-29, pp213-222, 1980.

[PeRa55] Peaceman, D. W., and Rachford, H. H., Jr. "The Numerical Solution of Parabolic and Elliptic Differential Equations," *J. Soc. Indust. Applied Mathematics, Vol. 3, No. 1, pp. 28-41 (Mar. 1955).*

[PrVi82] Preparata F. and J. Vuillemin, "The Cube-Connected-Cycles: A Versatile Network for Parallel Computation," Comm. ACM, Vol. 24, pp. 300-319 (1981).

[SCKu76]Sameh, A., Chen S., and Kuck D., "Parallel Poisson and Biharmonic Solvers", Computing 17 (1976), 219-230.

[SaKu78] Sameh, A. and Kuck, D., "On Stable Parallel Linear System Solvers," *JACM* 25, Jan. 1978, pp.81-91.

[Sava81] Savage, J. E. "Planar Circuit Complexity and the Performance of VLSI Algorithms," in *VLSI Systems and Computations*, ed. Kung, H., Sproull, B., Steele, G., CS press, pp.61-68. 1981.

[SiMc81] Siegel, H. J. and McMillen, R. J., "Using the Augmented Data Manipulator Ntwork in PASM," Computer 14(2). pp. 25-33 (Feb. 1981).

[Snyd81] Snyder, L. "Introduction to the Configurable Highly Parallel Computer," *Computer*, vol 15 pp.47-56 (Jan. 1981)

[Ston71] Stone, H. S., "Parallel Processing with Perfect Shuffle," IEEE Trans. on Computers, Vol. 20, pp. 153-161 (Feb. 1971).

[Ston75] Stone, H. S., "Parallel Tridiagonal Solvers," ACM TOMS 1, 289-307, 1975.

[Thom80][Thom80] Thompson, C., "A Complexity Theory for VLSI," Ph.D. Thesis Carnegie-Mellon University, 1980.

[ThKu77] C. D. Thompson and H. T. Kung, "Sorting on a Mesh-Connected Parallel Computer", CACM v.20, no 4, pp.263-270, 1977.

[Vuil80] Vuillemin, J. E., "A Combinatorial Limit to the Computing Power of VLSI Curcuits," Proc. 21st. Ann. Symp. on the Found. of Comp. Sc. pp. 240-300, 1980.

1. Report No. NASA CR-172436
ICASE Report No. 84-41

2. Government Accession No.

3. Recipient's Catalog No.

4. Title and Subtitle
On the impact of communication complexity in the design of parallel numerical algorithms

5. Report Date
August 1984

6. Performing Organization Code

7. Author(s)
Dennis Gannon and John Van Rosendale

8. Performing Organization Report No.
84-41

9. Performing Organization Name and Address
Institute for Computer Applications in Science and Engineering
Mail Stop 132C, NASA Langley Research Center
Hampton, VA 23665

10. Work Unit No.

11. Contract or Grant No.
NAS1-17070
NAS1-17130

12. Sponsoring Agency Name and Address
National Aeronautics and Space Administration
Washington, D.C. 20546

13. Type of Report and Period Covered
Contractor Report

14. Sponsoring Agency Code
505-31-83-01

15. Supplementary Notes
Langley Technical Monitor: Jerry C. South, Jr.
Final Report

16. Abstract

This paper describes two models of the cost of data movement in parallel numerical algorithms. One model is a generalization of an approach due to Hockney, and is suitable for shared memory multiprocessors where each processor has vector capabilities. The other model is applicable to highly parallel nonshared memory MIMD systems. In the second model, algorithm performance is characterized in terms of the communication network design. Techniques used in VLSI complexity theory are also brought in, and algorithm independent upper bounds on system performance are derived for several problems that are important to scientific computation.

17. Key Words (Suggested by Author(s))
multiprocessor, shared memory,
parallel algorithms,
communication networks

18. Distribution Statement
60 Computer Operations & Hardware
62 Computer Systems

Unclassified - Unlimited

19. Security Classif. (of this report)
Unclassified

20. Security Classif. (of this page)
Unclassified

21. No. of Pages
37

22. Price
A03

For sale by the National Technical Information Service, Springfield, Virginia 22161