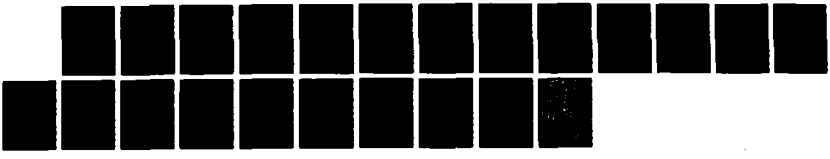
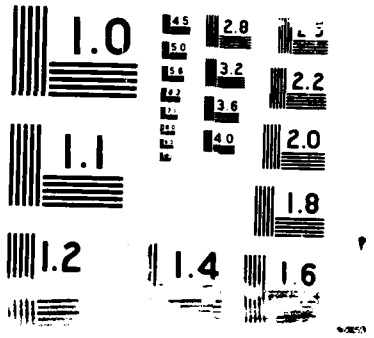
AD-A182 520 ANALYSIS OF PARAMETERIZED METHODS FOR PROBLEM PARTITIONING(U) YALE UNIV NEW HAVEN CT DEPT OF COMPUTER SCIENCE J H SALTZ MAY 87 YALEU/DCS/RR-537 1/1

UNCLASSIFIED N00014-86-K-0310 F/G 12/5 NL




1.0
1.1
1.2
1.4
1.6
1.8
2.0
2.2
2.8
3.2
3.6
4.0
4.5
5.0
5.6

Analysis of Parameterized Methods for Problem Partitioning

Joel H. Saltz

Research Report YALEU/DCS/RR-537
May 1987

YALE UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE

It is anticipated that in order to make effective use of many future high performance architectures, programs will have to exhibit at least a medium grained parallelism. Methods for aggregating work represented by a directed acyclic graph are of particular interest for use in conjunction with techniques under development for the automated exploitation of parallelism.

In this paper we carry out an investigation into methods appropriate for the aggregation, mapping and scheduling of relatively fine grained computations specified by a directed acyclic graph. The solution of very sparse triangular linear systems provides a useful model problem for use in exploring these heuristics.

A number of questions that relate to partitioning the work required to solve sparse triangular linear systems are consequently explored. A method is described for using the triangular matrix to generate a parameterized assignment of work to processors and simple expressions are derived that specify the scheduling of computational work. The tradeoffs between load imbalance and synchronization costs as a function of two orthogonal measures of granularity, block size and window size are examined experimentally on a shared memory machine, as well as analytically in the context of a model problem.

Analysis of Parameterized Methods for Problem Partitioning

Joel H. Saltz

Research Report YALEU/DCS/RR-537
May 1987

Work supported in part by the Office of Naval Research under Contract No. N00014-86-K-0310, NSF grant DCR 8106181 and NASA contract NAS 1-18107

1 Introduction

In the Crystal Runtime system [9], a very high level algorithm specification is supplied by the user. In this specification, the detailed interactions among processes in space and time are suppressed. The Crystal compiler and runtime system are being designed to allow the generation of instructions to direct an assemblage of communicating processes in the efficient execution of the specified algorithm. The compiler generates as many logical processes as possible and then the compiler or the runtime system combines clusters of logical processes to produce a problem decomposition that possesses a degree of granularity that is appropriate for the target machine. We carry out an investigation into methods appropriate for the aggregation, mapping and scheduling of relatively fine grained computations specified by a directed acyclic graph. The solution of very sparse triangular linear systems provides a useful model problem for use in exploring these heuristics.

The sparse triangular systems discussed here may be obtained from incomplete matrix factorizations used in Congugate Gradient type iterative algorithms employed in the solution of linear systems. A method is described for using the triangular matrix to generate a parameterized assignment of work to processors. The computation is partitioned into phases and simple expressions are derived that describe the scheduling of computational work. Each schedulable unit of work consists of the solution of a number of matrix rows that is determined by the block and window size specified; the computations in each phase are partitioned into a number of these independent work units. This scheduling ensures that all required data are computed before the first phase during which the data are required.

The tradeoffs between load imbalance and synchronization costs as a function of block and window size are examined, as are effects of the choice of strings on performance. These investigation are carried out in the following ways: (1) through analysis of a model problem, (2) through examination of the relationship between number of phases and operation counts, obtained for specific problems in the course of generating partitionings and schedules, and (3) through experimental timings on a Encore Multimax shared memory multiprocessor. A more extensive presentation of the work described here is to be found in [8].

We will assume that all computations pertaining to a row of the matrix will be assigned to a single processor. Note that this implies a potentially fine degree of granularity as we have a stated interest in matrices having few non zero elements in a row. The concurrency achievable through the use of this algorithm is largely determined by the dependencies between the rows of the lower triangular matrix. The approach to be taken here is to utilize an analysis of the data dependencies to produce a parameterized mapping. This partitioning process will take place in two stages. The rows of L will be partitioned into a number of disjoint sets to be called strings. So as not to compromise potential concurrency, strings contain only rows that could not under any circumstances be evaluated concurrently. The strings will then be distributed between processors with all computations pertaining to a given string assigned to a single processor.

The partitioning of the rows into strings and the distribution of the strings are performed so as to attempt to satisfy the objectives of maximizing potential concurrency and of allowing for the minimization of synchronization costs. The partitioning process described here is also suitable for generating mappings of work onto message passing architectures [9]. On message passing architectures, one seeks a good tradeoff between increasing con-

currency and the balance of load, and reducing communication costs. On these loosely coupled architectures, there is often particular motivation to attempt to limit the amount of information that must be communicated between processors and to reduce the number of communication startups. The above mentioned increase in computational granularity can be achieved by a judicious pattern of assignment of strings to processors and by scheduling work to be performed within strings in a course grained manner.

The problem partitions and work schedules that result from the process described above may be viewed as a generalization of the work described by Saad [11]. In that report, a wavefront method was proposed for scheduling work involved in forward and backsolves of matrices arising from incomplete factorizations of matrices generated by 5 point discretizations of two dimensional elliptic partial differential equations. The work described by Saad as well as the results presented here assume a row oriented matrix storage scheme. George [3] presents algorithms for a column oriented sparse cholesky factorization, along with algorithms for column oriented forward and backsolves. These algorithms utilize the notion of a pool of tasks whose parallel execution is controlled by a self-scheduling discipline. Heath [4] presents algorithms for parallel solution of triangular systems in distributed memory multiprocessors; these algorithms utilize a type of adjustable parameter for controlling algorithm granularity quite different from the ones discussed here. In the algorithms described in [4], the work required to calculate the inner products involved in solving for each row is shared among the processors. In very sparse triangular systems considered in this paper, there are very few computations involved in solving for a given variable. In these systems, parallelism can be obtained because the data dependencies between rows can allow one to solve for many variables simultaneously.

2 Problem Partitioning

2.1 Overview

In the simplest form of incomplete LU preconditioning, the factors L and U have the same sparsity structure as the lower and upper portions of A respectively. A prior knowledge of the sparsity structure will be used to advantage in the generation of the following parameterized problem mapping. Note that this prior knowledge is not needed when the automated version of the problem mapping is used. This automated version of problem mapping will be described in the following section.

We will assume that we have a rectangular array of grid points, all points are connected with the same stencil. The stencil is assumed to link a given point with it's left, right, upper and lower neighbors in the grid. The matrix is formed by using the so called natural ordering in which grid points are numbered in a row-wise fashion beginning with the first column of the first row of the domain. We assume the same stencil is utilized for all mesh points in the problem.

The data dependency pattern between unknowns in the lower triangular solution may be best understood by referring back to the stencil and the grid utilized in the formulation of the problem [11]. Let $x_{i,j}$ be the location of a mesh point in the two dimensional domain, where $1 \leq i \leq n$ and $1 \leq j \leq n$. In the definition of the problem, a function value at a point $x_{i,j}$ is linearly dependent on function values at a given set of surrounding points. When a system involving a lower triangular matrix with the same sparsity structure as A is solved, the only interactions that need be considered are with variables in the grid that

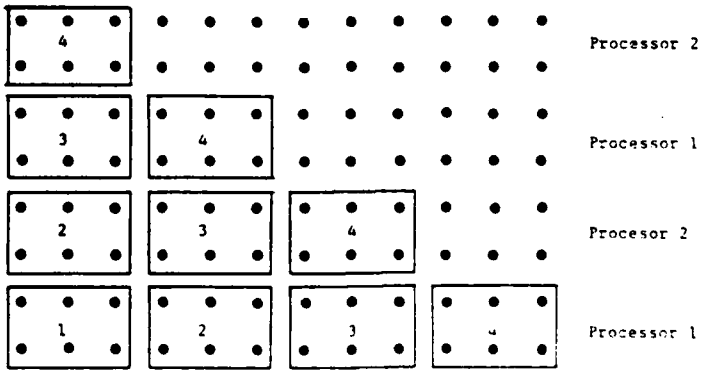


Figure 1: Two processors, five point stencil, block size = 2, window = 3. Numbers designate computational phases.

are in rows before i, as well as variables in row i that are before column j.

The grid points in a given row must be solved for sequentially, due to the coupling of each point to it's immediate neighbors. We assume that the stencil is rather small, so that relatively few calculations are involved in obtaining the value for a single grid point of the domain. In these mappings, the smallest unit of work that may be assigned to a particular processor consists of the computations pertaining to a particular row of grid points. The computations in a given row i depend only on results from row $j < i$. Depending on the relative size and properties of the problem and of the machine, better performance may be obtained by using a coarser grained assignment of work in which contiguous blocks of several rows are assigned to each of k processors. When there are more blocks of rows than there are processors, a wrapped assignment is used in which blocks are assigned to processors modulo k.

Given a fixed assignment of grid points to processors, one may be free to schedule the work associated with calculating values at mesh points in a variety of different ways. This processor scheduling has a marked effect on the frequency with which processors must interact to exchange information. When a five point stencil is utilized, a convenient method of scheduling is to partition each block into windows of w columns each. Because of the use of the five point stencil, values for all points in a given window of a block may be computed before any work on the next window is begun. If one numbers the windows in each block from left to right, block i may commence work on window j when block $i-1$ has finished work on window j. This leads to a pattern of computation [11] in which a wavefront of computation is seen to propagate from the lower left portion of the domain (Figure 1). The block size and the size chosen for the window both determine the coarseness of the computation's granularity. In [11] is found a quantitative analysis of this tradeoff in the case where the block size is equal to the window size and the grid is square. This analysis is extended both through analytically and experimentally in the following in order to explore

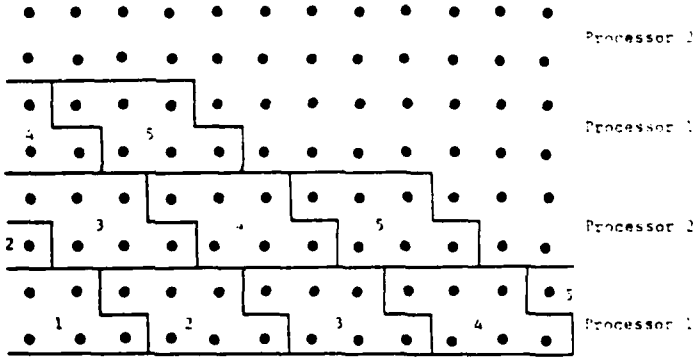


Figure 2: Two processors, nine point stencil, block size = 2, window = 3. Numbers designate computational phases.

the effects of independently varying block and window size in a rectangular grid.

For a grid whose points are connected by an arbitrary stencil, the definition of work schedules that maintain data dependency relations yet allow for varying degrees of granularity is somewhat more subtle. Work is begun in the first row of the first block, and in this row the values for w window of grid points are calculated. Following this, values are found for all mesh points in the block for which data dependencies allow calculation. The computation proceeds after this in stages, with the computations that may proceed in a block at a given time being determined by dataflow considerations. If one wishes to aggregate points in blocks into larger units, with each unit to be calculated sequentially, the partitioning will take on a zig-zag form. Figure 2 depicts the pattern of wavefronts that results from partitioning a domain with a nine point stencil into blocks of size two, and scheduling computation using a window size of two.

2.2 Automated Problem Partitioning

In order to automate problem partitioning and work scheduling, it is essential to be able to dispense with as much application dependent information as possible. We are in the process of developing and testing a method for generating a work partition in problems possessing data dependencies given by a directed acyclic graph (DAG). This method bears a strong relationship to methods proposed for systolic array generation [1], [6]. The order in which variables, described by rows in L, can be solved may be depicted by a directed acyclic graph D. The evaluation of rows in L are represented by the vertices of D, and the data dependencies between the rows by D's edges. The dependence of matrix row a on matrix row b is represented by an edge going from vertex b to vertex a. A topological sort may be performed which partitions the DAG into wavefronts. A stage of this sort is performed by alternately removing all vertices that are not pointed to by edges, and then removing all edges that emanated from the removed vertices. All vertices removed during a given stage constitute a wavefront; the wavefronts are numbered by consecutive integers. An adaptation of a common topological sort algorithm [5] allows the wavefronts of a DAG

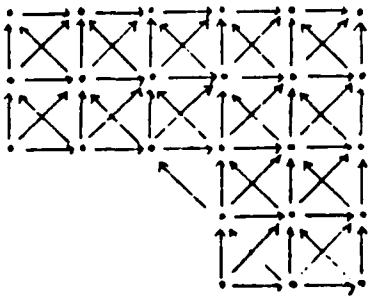


Figure 3: Data Dependencies

to be calculated efficiently.

The wavefronts calculated through this process can be utilized directly in implementing a very general method for scheduling the row substitutions required for the solution of the equations. The row substitutions in any wavefront may be executed simultaneously. A very straightforward method for solving the problem is consequently to partition the problem's solution into phases, each of which is dedicated to a given wavefront. On shared memory machines, the straightforward application of this technique requires a global synchronization between phases. Because in many cases there is only a relatively modest amount of computation required for a given phase, the relative cost of the global synchronization can be quite substantial, as will be shown in the experimental results below. On many message passing machines, (e.g. the Intel iPSC [10]), the communication latency makes this kind of medium grained parallelism particularly prohibitive. Similarly, in message passing machines, it is of considerable importance to map problems in a way that reduces interprocessor communication requirements.

For many problems possessing relatively regular patterns of data dependency, one can obtain a variety of benefits on both shared memory and message passing machines by carrying the run time analysis a step further through partitioning the DAG in a particular way. The points of a DAG are partitioned into disjoint sets called *strings*. A string partition of a problem is generated through the following sequence of depth first traversals in DAG D.

We define a start vertex of D as a vertex not pointed to by any edge. The vertices making up a string S are chosen in the following way. A start vertex V of D is chosen, all edges emanating from V are removed; if a new start vertex V' is created through the removal of edges, V' is included in the string. The process is continued to recursively remove as many vertices as possible from D, and assign them to S. Note that when, during the creation of string S, the removal of a vertex exposes multiple start vertices, only one of these start vertices are included in S. As vertices V' are assigned to S, we mark the vertices W remaining in D that had edges arising from V'. New strings are begun using available start vertices. In choosing vertices to incorporate in all strings after the first, preference is given to vertices previously marked by other strings.

Strings have the following properties: (1) The points in each string are connected, (2) There is no more than one point belonging to a given wavefront in a string, (3) The graph describing the *inter-string* dependencies is a directed acyclic graph. The DAG describing the inter-string dependencies will be called the *string DAG*.

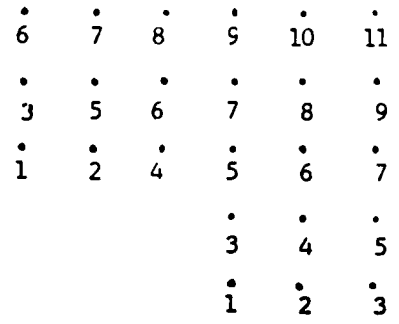


Figure 4: Wavefronts

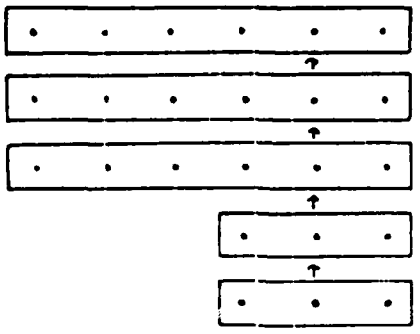


Figure 5: Strings and String DAG

Figure 3 depicts a DAG which could be obtained from a zero fill incomplete factorization of a matrix arising from the discretization of an elliptic partial differential equation using a nine point star template. Figure 4 depicts the wavefronts in the computation, and figure 5 depicts a string decomposition and illustrates the string DAG corresponding to this problem.

It should be noted that it may be possible to partition a DAG into strings in several different ways. For example, the triangular system arising from the zero fill factorization of the matrix generated by a rectangular grid with a 9 point template can be partitioned in two ways. In one partitioning, matrix rows originating from horizontal strips of domain form strings, in the other matrix rows originating from diagonal strips of domain form strings. Performance implications of these different methods of decomposition will be touched on in the presentation of experimental timings below.

In a rough sense, the strings of a DAG D are sets of points orthogonal to the wavefronts of D. This type of decomposition allows for considerable flexibility in determining the granularity of parallelism, as discussed below. The decomposition of the DAG D into strings is shown in [8] to facilitate particularly inexpensive forms of synchronization in shared memory architectures.

The data dependency relationships in the problems discussed in this paper are quite regular and are easily handled by the mechanism described above and in fact could be handled by methods described in [1] if the data dependencies were given in a symbolic fashion. An discussion of methods for generating strings for more irregular problems is currently underway.

2.3 Mapping Strings onto Processors

The string DAG may be distributed among processors in a variety of ways. On message passing machines, mapping large contiguous sections of the string DAG onto each processor will tend to minimize communication costs, but will also tend to lead to poor load distributions. Scattering or wrapping strings that are contiguous in the DAG may lead to a much better load distribution at the price of increased communication costs.

The work associated with each cluster of strings may be scheduled with varying degrees of granularity. The string DAG defines a partial ordering among the strings. The starting strings may be defined as the strings that precede all others in this partial ordering. Computations of rows in these strings are not dependent on information from any other strings in the string DAG.

The partial ordering of the data dependencies between the strings allows for the straightforward implementation of dataflow synchronization methods. The granularity of parallelism may be determined by fixing the amount of work starting strings can perform before communicating their data to other strings in the string DAG. Simple relationships involving the wavefronts of *rows* allows the calculation of which rows may be solved for by a processor assigned to a cluster of strings.

3 Construction of Work Schedules

3.1 General String DAG

The parallelism involved in solving a sparse triangular system of equations is inherently rather fine grained, the work required to compute the value of a variable corresponding to a given row generally amounts to only a few floating point operations. In scientific applications, one frequently wishes to obtain multiple solutions with either the same triangular system or triangular systems with the same non zero structure. In these cases, it seems to be appropriate to spend a modest amount of time to calculate a work schedule. It is essential, however, that very little time be required to schedule row solutions during the execution of the algorithm. We consequently have chosen to preschedule the assignment of work to processors. The rows to be computed by a string at a given phase in a computation are chosen by determining the wavefronts that should be computed during that phase.

We will now consider methods for scheduling the execution of work given a string DAG. We will assume that the strings making up the string DAG have been linearly ordered and that contiguous blocks of b strings are demarcated. In this paper we will assume that the strings are assigned assigned to consecutive processors in a wrapped manner. In the following, we will say that we have computed wavefront q in some block of strings when we compute values for all matrix rows belonging to wavefront q.

We will calculate the largest wavefront that the strings in a block must compute during a particular phase. Obviously computations cannot be undertaken until the required data is available. The calculation of the wavefronts that are to be computed during each phase takes into account the data that is guaranteed to be available when a processor reaches a given phase. In this paper we will limit our discussion and experimentation to the use of barrier synchronization methods; these methods assure that all processors have finished phase $p-1$ before any processor is allowed to begin phase p. We allow for the existence of an arbitrary pattern of data dependencies between the strings of the string DAG.

The proposition below presents expressions that give the maximum wavefront number that is to be computed by a given block i during phase p, under the assumption that the first block computes exactly w wavefronts per phase; i.e. during phase p the first block computes wavefronts $w(p-1)+1$ to wp. This proposition may therefore be regarded as a method of parametrically describing the wavefronts of a coarse grained DAG, each vertex of which represents the solution of a number of rows (note: this coarse grained DAG is *not* the string DAG). It should be noted that one may assign any work scheduled during a phase to any processor one desires, although in message passing machines this strategy is likely to lead to higher communication costs. In problems described by irregular DAGs it is expected that explicitly balancing the processor load during each phase will often be advantageous.

Proposition 1 *Assume that strings making up the string DAG have been linearly ordered, that contiguous blocks of strings are demarcated, and that these blocks are assigned to consecutive processors in a wrapped manner. Let W_p^i represent the largest wavefront that can be scheduled during phase p by block i under the following conditions: (1) the first block advances w wavefronts per phase, i.e. $W_p^1 = wp$, and (2) all required data is computed before the system reaches phase p.*

W_p^i is given by the expression $W_p^i = \max(p, w(p-i+1)+i-1)$.

In scheduling work for block i during phase p, we must take into account the numbers of the wavefronts corresponding to the latest available results from blocks $1 \leq j < i$, since block i may require results from any of these blocks. Since no work can be performed before the first phase, we set $W_p^i = 0$ for $p = 0$. The number of the smallest wavefront corresponding to any result that might be needed by block i at the beginning of phase p may be expressed as

$$\min_{1 \leq j < i} W_{p-1}^j$$

Consequently,

$$W_p^i = \min_{1 \leq j < i} W_{p-1}^j + 1$$

for $p \geq 1$.

We now use the above to prove that for all $p \geq 1$, if $\hat{W}_p^i = \max(p, w(p-i+1)+i-1)$ then $W_p^i = \hat{W}_p^i$. This proof proceeds by induction on block number i.

For $i = 1$, by assumption $W_p^1 = wp$. Since $\hat{W}_p^1 = \max(p, wp)$, $W_p^1 = \hat{W}_p^1$.

We will now use the induction hypothesis for $j \leq i$ to show $W_p^{i+1} = \hat{W}_p^{i+1}$ for $p \geq 1$ and $i \geq 2$. We are assuming that for $j \leq i$ and $p \geq 1$,

$$W_p^j = \max(p, w(p-j+1)+j-1).$$

For $p \geq 2$, $j \leq i$ we thus have

$$W_{p-1}^j = \max(p-1, w(p-j)+j+1) = \max(p-1, w(p-1)-(w-1)(j-1))$$

Now

$$W_p^{i+1} = \min_{1 \leq j < i+1} W_{p-1}^j + 1,$$

so because

$$\min_{1 \leq j < i+1} W_{p-1}^j = \max(p-1, w(p-i)-(w-1)(i-1)),$$

it follows that

$$W_p^{i+1} = \max(p, w(p-(i+1)+1)+(i+1)-1).$$

Thus $\hat{W}_p^{i+1} = W_p^{i+1}$ and the induction is complete for $p \geq 2$.

For $p = 1$, since $W_0^j = 0$, $W_1^{i+1} = \min_{1 \leq j < i+1} W_0^j + 1 = 1$. As it is easily verified that $\hat{W}_1^{i+1} = 1$, $W_1^{i+1} = \hat{W}_1^{i+1}$.

Thus we have shown that for $p \geq 1$, $W_p^{i+1} = \hat{W}_p^{i+1}$ and the proposition is proved.

□

3.2 Nearest Neighbor String DAG

When it is known that data dependencies occur only between adjacent strings, a more aggressive scheduling policy can be used.

Proposition 2 *Assume that strings making up the string DAG have been linearly ordered so that data dependencies occur only between adjacent strings, that contiguous blocks of strings are demarcated, and that these blocks are assigned to consecutive processors in a wrapped manner. Let W_p^i represent the largest wavefront that can be scheduled during phase p by block i under the following conditions: (1) the first block advances w wavefronts per phase, i.e. $W_p^1 = wp+b-1$, and (2) all required data is computed before the system reaches phase p.*

W_p^i is given by the expression

$$W_p^i = \begin{cases} w(p-i+1)+ib-1 & \text{if } p \geq i \\ bp & \text{if } 0 \leq p < i. \end{cases}$$

Assume that block B has assigned to it strings $v+r$, $1 \leq r \leq b$ and that string v has advanced it's calculations up to phase p. Due to the nearest neighbor data dependency relations, string $v+r$ may be advanced to wavefront $p+r$. Note that were we not to assume nearest neighbor inter-string data dependencies, it is possible that string $v+r$ could have a direct data dependence on string v. In this general case, string $v+r$ could not proceed beyond phase $p+1$. We are thus able to conclude that when we use continuous blocks of b strings each,

$$W_p^i = W_{p-1}^{i-1} + b$$

Using the above relationship, we will show by induction on block number i that for all $p \geq 1$, if

$$\hat{W}_p^i = \begin{cases} w(p-i+1)+ib-1 & \text{if } p \geq i \\ bp & \text{if } 0 \leq p < i \end{cases}$$

then $W_p^i = \hat{W}_p^i$.

For $i = 1$, $\hat{W}_p^1 = wp+b-1$ for $p \geq 1$ so $W_p^1 = \hat{W}_p^1$.

Assume that $W_p^i = \hat{W}_p^i$ for $p \geq 1$. We will show that $W_q^{i+1} = \hat{W}_q^{i+1}$ for $q \geq 1$. We first consider the situation that occurs when $q \geq i+1$. In this case we have

$$W_{p+1}^{i+1} = W_p^i + b = w((p+1)-(i+1)+1)+(i+1)b-1.$$

Since $p+1 \geq i+1$, the above expression is equal to $\hat{W}_{p+1}^{i+1}$, and consequently $W_q^{i+1} = \hat{W}_q^{i+1}$ for $q \geq i+1$.

For $0 \leq p < i$,

$$W_{p+1}^{i+1} = W_p^i + b = b(p+1).$$

Since $p+1 < i+1$, $\hat{W}_{p+1}^{i+1} = b(p+1)$ and hence $W_q^{i+1} = \hat{W}_q^{i+1}$ for $1 \leq q < i+1$.

□

4 Load Balance - Synchronization Cost Tradeoffs

4.1 Analysis of a Model Problem

For a given problem, the tradeoffs between load imbalance and synchronization costs will vary with choice of window and block size. We will examine this tradeoff in the context of solving a lower triangular system generated by the zero fill factorization of the matrix arising from a rectangular mesh with a five point template. We will utilize P processors and partition the domain into n horizontal strips where each strip is divided into m blocks, as is depicted in figure 1. We will assume that the problem is obtained from a domain with N by M mesh points, and that all computations required to solve the problem would require time S on a single processor. We will also assume that computation of each block takes time $T_B = S/(mn)$; this ignores the relatively minor disparities caused by the matrix rows represented by points on the lower and the left boundary of the domain. Horizontal strips of blocks are assigned to each of P processors in a wrapped manner. The computation is divided into phases; during phase p the processor assigned to strip i computes block $p - i + 1$ in the strip, as long as $1 \leq p - i + 1 \leq n$.

A brief inspection of figure 1 makes it clear that $n + m - 1$ phases required to complete the computation. Define $MC(j)$ as the maximum number of blocks computed by any processor during phase j. The computation time required to complete phase j is equal to $T_B MC(j)$, the computation time required to complete the problem is consequently

$$\sum_{j=1}^{n+m-1} T_B MC(j).$$

We now proceed to calculate $MC(j)$. During phase j, a total of j blocks must be computed when $1 \leq j < \min(m,n)$. Since the blocks are assigned in a wrapped manner,

$$MC(j) = \lceil \frac{j}{P} \rceil.$$

When $\min(m,n) \leq j \leq n + m - \min(m,n)$, a total of $\min(m,n)$ blocks must be completed during phase j. Due to the wrapped assignment of blocks to processors,

$$MC(j) = \lceil \frac{\min(m,n)}{P} \rceil.$$

Finally when $n + m - \min(m,n) < j \leq n + m - 1$, a total of $n + m - j$ blocks must be computed during phase j so

$$MC(j) = \lceil \frac{n+m-j}{P} \rceil.$$

The computation time required to complete the problem is consequently

$$T_B \sum_{j=1}^{n+m-1} MC(j) =$$
$$\frac{S}{mn}\Big(\sum_{j=1}^{\min(m,n)-1} \lceil \frac{j}{P} \rceil + (n+m-2\min(m,n)+1)\lceil \frac{\min(m,n)}{P} \rceil +$$
$$\sum_{j=m+n-\min(m,n)+1}^{n+m-1} \lceil \frac{n+m-j}{P} \rceil\Big)$$

In a shared memory environment we must synchronize between phases. Assume that each synchronization has cost T_S. The total time spent synchronizing is then given simply by $T_S(n+m-1)$. Assume the problem is mapped to a message passing machine so that processors assigned consecutive strips of blocks directly communicate and where links between processors can operate in parallel. The cost of sending a B word message between two processors can be approximated as $\alpha + \beta B$. We will make the further approximation concerning the cost of requiring *each* processor to send a message to its neighbor following phase j. That cost is equal to the time required to communicate the largest message sent between two processors following phase j. The maximum amount of information that must be sent from one processor to another after phase j is $(M/n)MC(j)$. The cost of communications that follow phase j may be expressed as

$$\alpha + \beta \frac{M}{n} MC(j).$$

The total cost of communications is hence given by

$$\alpha(n+m-1) + \beta \frac{M}{n}$$
$$\left(\sum_{j=1}^{n+m-1} \lceil \frac{j}{P} \rceil + (n+m-2\min(m,n)+1)\lceil \frac{\min(m,n)}{P} \rceil + \right.$$
$$\left. \sum_{j=m+n-\min(m,n)+1}^{n+m-1} \lceil \frac{n+m-j}{P} \rceil \right)$$

Using the above considerations, we will now calculate a simple expression for the amount of work forgone during a computation when the number of blocks in a strip n as well as the number of strips in a problem m are both integer multiples of the number of processors P utilized. We thus assume that $n = r_1 P$ and $m = r_2 P$, for r_1, r_2 positive integers. ¿From the discussion above, during the first $\min(m,n) - 1$ phases, the computation requires time

$$T_B \sum_{j=1}^{\min(m,n)-1} \lceil \frac{j}{P} \rceil.$$

During phase $j \leq \min(m,n) - 1$ when j is not a multiple of P, there are $P - j \bmod P$ processors idle; when j is a multiple of P, no processors are idle. Thus the sum of the processor idle time for $j \leq \min(m,n) - 1$ is $T_B \ \min(r_1, r_2) \sum_{l=1}^{P} (l-1)$, or

$$T_B \frac{\min(r_1, r_2) P(P-1)}{2}.$$

Through identical arguments, the sum of the processor idle time for the last $\min(m,n)-1$ phases is the same as that above. During the intermediate phases the load is balanced with $\min(m,n)$ blocks assigned to each processor.

In a shared memory environment, using the above expressions for the processor time wasted due to load imbalance and the time spent in synchronization we find that the total processor time wasted from both causes may be given by:

$$\frac{SP(P-1)\min(r_1,r_2)}{r_1 r_2 P^2} + T_S P(r_1 P + r_2 P)$$

Note that the above expression is symmetric with respect to r_1 and r_2. When the synchronization cost is the dominant overhead, it consequently does not matter whether one uses small windows and assigns large blocks of variables to each processor, or whether one uses large windows and assigns small blocks of variables to each processor. Assume without loss of generality that $r_2 \leq r_1$, i.e. that the window size is at least as large as the number of rows of grid points in a block. In this case the total processor time wasted is

$$\frac{SP(P-1)}{r_1 P^2} + T_S P(r_1 P + r_2 P). \tag{1}$$

For any fixed r_1 reducing r_2 to 1 decreases the time spent by processors in synchronization without impacting adversely on the balance of computational load. Thus when the number of blocks in a strip n as well as the number of strips in a problem m are both integer multiples of the number of processors P utilized, the window size can be profitably increased to M/P. This increase in window size does not affect the distribution of load and reduces the number of phases required to solve a problem.

5 Experimental Results

5.1 Preliminaries

The figures discussed in the current section depict the results of measurements made of the amount of time required to perform a forward substitution utilizing the three forms of synchronization discussed above. The matrix utilized was generated through the zero fill incomplete factorization of square meshes of various sizes, in which one of a number of templates were employed.

Before discussing the experimental results in detail, the architecture of the Encore Multimax will be briefly described. The Encore Multimax is a bus based shared memory machine that utilizes 10 MHz NS32032 processors and NS32081 floating point coprocessors. Processors, shared memory, and i/o interfaces communicate using a 12.5 MHz bus with separate 64 bit data paths and 32 bit address paths.

Associated with each pair of processors is a 32K-byte cache of fast static RAM. Memory data is stored in this cache whenever either of the two processors associated with the cache reads or writes to main memory locations. Each cache is kept current with the relevant changes in main memory by continuous monitoring of traffic on the bus [2].

All tests reported were performed on a configuration with 16 processors and 16 Mbytes memory at times when the only active processes were due to the author and to the operating system. On the Encore the user has no direct control over processor allocation. Tests were performed by spawning a fixed number of processes and keeping the processes in existence

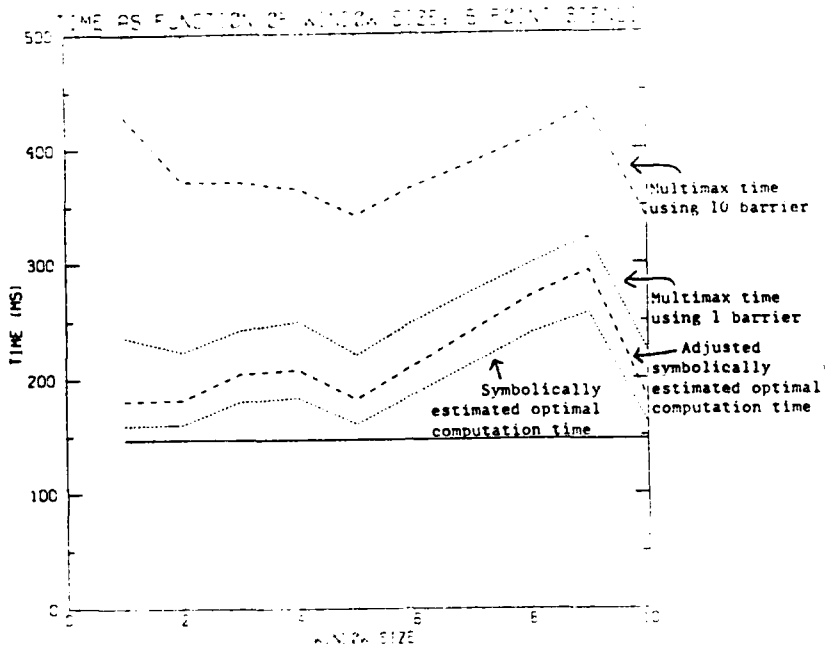


Figure 6: Execution time required to solve a lower triangular system generated by a zero fill incomplete factorization of a matrix arising from a 100 by 100 mesh, 5 point template. 10 processors used, timings for 25 consecutive trials averaged.

for the length of each computation. This programming methodology is further described in [7]. The processes spawned are scheduled by the operating system, and for this otherwise empty system, throughout the following discussions we make the tacit assumption that there is a processor available at all times to execute each process. In order to reduce the effect of system overhead on our timings, tests were performed using no more than 14 processes; this left two processors available to handle the intermittent resource demands presented by processes generated by the operating system.

It should be noted that the bus connecting processors to memory does not appear to to cause significant performance degradation in problems with the mix of computations and memory references that characterize the problems described here. In a set of experiments using a variety of sparse lower triangular matrices, multiple identical sequential forward solves were run on separate processors at the same time. Timings of this experiment exhibited performance degradations of less than one percent as one increased the number of processors utilized from one to 14.

5.2 Effect of Window and Block Size on Performance

The effect of window size on execution time was investigated. The data depicted in figure 6 was obtained through a forward solve of the zero fill factorization of a matrix generated using a 100 by 100 point square mesh, in which a 5 point template was employed. Note that this matrix is extremely sparse, there are no more than two non-zero off diagonal elements in any matrix row. The strings in this problem partition the domain into horizontal slices as was described in the analysis of the model problem previously discussed. Ten processors were used to solve this problem, and a block size of one was utilized.

A symbolic estimate was made of the optimal speedup that could be obtained in the

absence of synchronization delays, given the assignment of work to processors characterizing a particular window and block size. For each window size, the time required for a separate sequential code to solve the problem was divided by the estimated optimal speedup. This yields the amount of time that would be required to solve the problem in the absence of any sources of inefficiency other than load imbalance. The results of these calculations are plotted in figure 6 where they are denoted as the symbolically estimated optimal computation time.

Dividing the execution time of the one processor version of the parallel code by the estimated optimal speedup yields a further refined estimate of the shortest amount of time in which the problem could be solved in the absence of synchronization delays. In figure 6 these results are plotted and are denoted as the adjusted symbolically estimated optimal computation time. It is of interest to note that, as predicted by equation (1), the two estimates of the optimal computation time predict close to identical computation times for windows of size one, two, five and ten. The computation times estimated for windows of other sizes are larger.

Timings were obtained by solving the problem using ten processors on the Multimax, timings were averaged over 25 consecutive runs. Barrier synchronization between phases was utilized. When timed separately, this synchronization was found to require 75 microseconds; this compares to approximately 20 microseconds required for a single precision floating point multiply and add. It is not clear that future architectures utilizing much faster processors and more general interconnection networks will allow for synchronization costs that are as small relative to the costs of floating point computation. In a separate set of measurements also depicted in figure 6, the effects of varying window size in an environment characterized by higher relative synchronization costs were explored though the use of ten 75 microsecond barriers between phases of the computation.

Tradeoffs between load imbalance and synchronization costs were examined in a different manner by comparing the symbolically estimated optimal speedup against the number of phases required to complete a problem. The symbolically estimated optimal speedup takes into account the degree to which a given assignment of work to processors balances the workload. The number of phases required to solve a problem has a strong bearing on the synchronization overhead encountered in solving a problem.

In figure 7 the symbolically estimated optimal speedup was compared with the phases required for solving a lower triangular system generated by zero fill factorization of a matrix arising from a 75 by 75 point mesh, utilizing a nine point template. The strings chosen were those partitioning the domain into horizontal strips.

The estimated speedups resulting from the use of blocks of sizes one and two, with the size of windows varying from one to eight are depicted, along with the speedups resulting from the use of windows of sizes one and two, with the size of blocks varying from one to eight. Also depicted are speedups resulting from using a block size that is equal to the window size; both are varied from one to six.

The tradeoff between speedup and number of phases used, appears to be generally more advantageous when large windows and small block sizes are used than when the situation is reversed. As was observed in the examination of the performance obtained using the five point template, the number of phases declines with increasing window and or block size, while the load balance exhibits substantial fluctuations. The tradeoff between load imbalance and number of phases required appears to be much smoother when the size of the window used is set equal to the size of the block than in the other cases discussed

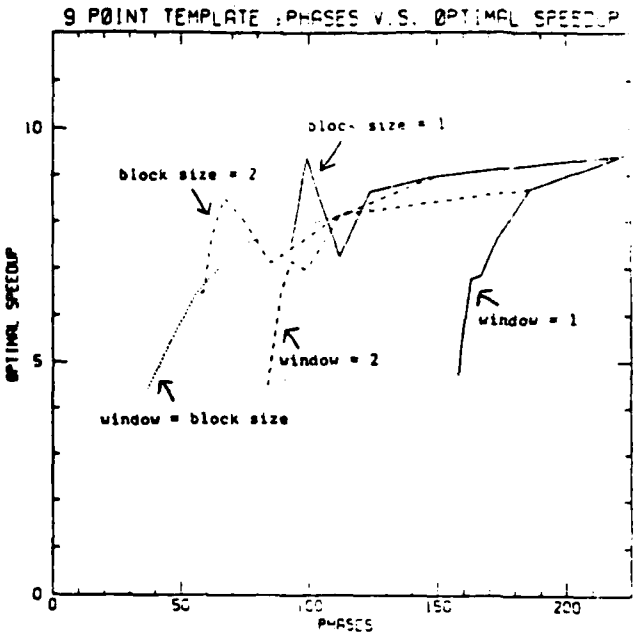


Figure 7: Solution of lower triangular system generated by zero fill incomplete factorization of a matrix arising from a 75 by 75 point mesh, 9 point template. Symbolically estimated optimal speedup versus phases required in solving problem on 12 processors. Horizontal strings used.

above; the estimated speedup is in this case a decreasing function of the block and window size used. For any given number of phases, the load balance when window size is equal to block size is superior to that obtained when the window size is set equal to one or two and the block size is varied.

As synchronization costs increase, it becomes more advantageous to reduce the number of phases required to solve a problem even at the cost of increased load imbalances.

The relative performance of four combinations of window and block size in the face of increasing synchronization costs are depicted in figure 7. The execution time required to solve the lower triangular system described above was measured when the following combinations of window and block size were employed: (1) window size = 1, block size = 1, (2) window size = 2, block size = 1, (3) window size = 1, block size = 2 and (4) window size = 2 , block size = 2. Between phases, we employed from one to ten 75 microsecond barrier synchronizations.

The numbers of phases and the symbolically estimated optimal speedup for each of these cases are listed below.

block size	window	phases	est. speedup
1	1	223	9.43
1	4	112	7.26
4	1	167	6.84
2	2	112	8.14

As one can observe from the above table, block size four, window one and block size one, window four in this problem require at least as many phases as does block size two, window two and the later achieves a superior load balance. The use of block size one, window one allows one to achieve a load balance that is even better, but at the cost of added phases of

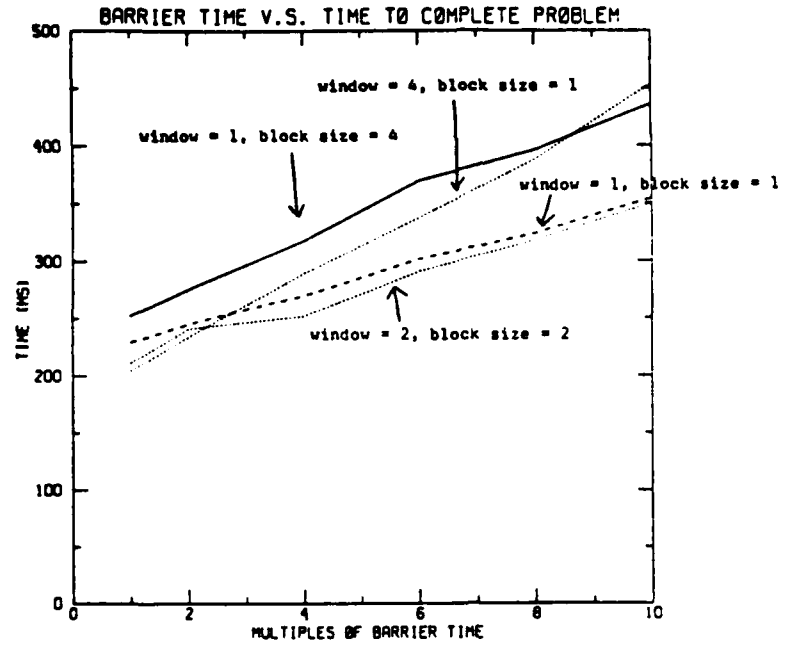


Figure 8: Execution time required to solve a lower triangular system generated by a zero fill incomplete factorization of a matrix arising from a 75 by 75 point mesh, 9 point template. 12 processors used, timings for 25 consecutive trials averaged.

computation. In figure 8, for barrier times between 75 and 150 microseconds, the shortest run times were obtained using block size and window size both equal to one. When barriers were utilized that required more than 150 microseconds the use of block and window sizes both equal to two lead to the shortest run times.

5.3 String Orientation Effects

The relative merits were investigated of using horizontal versus diagonal strings in partitioning a mesh with a 9 point template. In figure 9 is plotted the time required for 12 processors to solve a lower triangular system generated by a zero fill factorization of a matrix arising from a 75 by 75 point mesh. The block size was kept constant at one, and the window size was varied from one to eight. Tests were carried out using both single 75 microsecond barriers between computational phases and using ten 75 microsecond barriers between phases. For each synchronization cost and window size investigated, the time required for solving the problem using diagonal strings was greater than that required when horizontal strings were utilized. A substantial reduction in execution time occurred with increasing window size when horizontal strips were employed and inter phase synchronization was expensive.

In figure 10 for the problem described immediately above, the symbolically estimated optimal speedup is plotted against the number of phases utilized. The use of horizontal strings leads to a substantially more favorable tradeoff between the speedup obtained and the phases required for solving the problem.

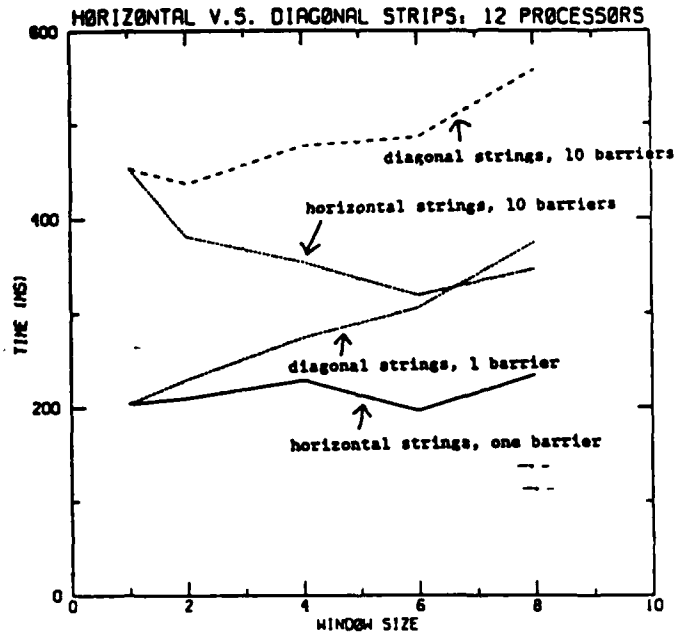


Figure 9: Execution time required to solve a lower triangular system generated by a zero fill incomplete factorization of a matrix arising from a 75 by 75 mesh, 9 point template. 12 processors, block size = 1, timings for 25 consecutive trials averaged.

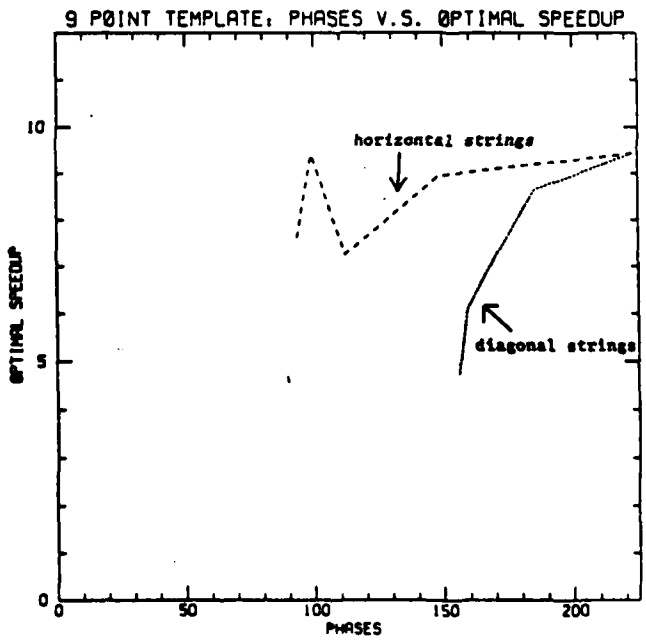


Figure 10: Solution of lower triangular system generated by zero fill incomplete factorization of a matrix arising from a 75 by 75 mesh, 9 point template. Symbolically estimated optimal speedup versus phases required in solving problem on 12 processors. size of block = 1.

6 Conclusion

We have carried out an investigation into methods appropriate for the aggregation, mapping and scheduling of relatively fine grained computations specified by a directed acyclic graph. The solution of very sparse triangular linear systems provides a useful model problem for use in exploring these heuristics. A method for using the triangular matrix to generate a parameterized assignment of work to processors was described along with simple expressions that describe how to schedule computational work with varying degrees of granularity. These expressions are of considerable practical importance because they allow one to easily determine what computations need to be performed during a given phase to ensure that all required data are computed before they are required. The tradeoffs between load imbalance and synchronization costs as a function of block and window size were examined in the context of a model problem and it was demonstrated that increases in the granularity of parallelism can, in some circumstances, be obtained without any increase in load imbalances.

Experimental timings on an Encore Multimax shared memory multiprocessor confirmed the above observation in the case of the model problem and went on to explore the effects of block size and window size on multiprocessor performance in a wider variety of settings. The ratio between the costs of synchronizing and the costs of performing computations on the current Multimax is low enough that rather fine grained parallelism can be profitably used. Examination of load imbalance / synchronization cost tradeoffs in architectures requiring coarser grained parallelism was performed by experimentally varying the cost of synchronization.

As was to be expected from the model problem analysis, there is often not a particularly smooth tradeoff between between load imbalance and computational granularity. Studies of this tradeoff obtained through comparing operation counts and number of computational phases required to solve a problem suggest that the load balance / granularity tradeoff becomes smoother when window size and block size are roughly equal.

The effects of the choice of strings on performance was also examined. It was found that partitioning a lower triangular system obtained from a rectangular domain using strings with a horizontal orientation yielded a much more favorable tradeoff between load imbalance and computational granularity.

7 Acknowledgements

I wish to thank Stan Eisenstat and Dave Nicol for useful discussions.

References

[1] M. C. Chen. A design methodology for synthesizing parallel algorithms and architectures. *Journal of Parallel and Distributed Computing*, 116–121, 1986.

[2] *Multimax Technical Survey.* Technical Report 726-01759 Rev A, Encore Computer Corporation, 1986.

[3] A. George, M.T. Heath, J. Liu, and E. Ng. *Solution of Sparse Positive Definite Systems on a Shared-Memory Multiprocessor.* Technical Report ORNL/TM-10260, Oak Ridge National Laboratory, January 1987.

[4] M. T. Heath and C. H. Romine. *Parallel Solution of Triangular Systems on Distributed Memory Multiprocessors.* Technical Report ORNL/TM-10384, Oak Ridge National Laboratory, March 1987.

[5] E. Horowitz and S. Sahni. *Fundamentals of Data Structures.* Computer Science Press, Rockville Maryland, 1983.

[6] Delosme J-M and Ilse Ipsen. An illustration of a methodology for the construction of efficient systolic architecture in vlsi. In *Proceedings of the Second International Symposium on VLSI Technology, Systems, and Applications*, pages 268–273, May 1985.

[7] J. F. Jordan, M. S. Benten, and N. S. Arenstorf. *Force User's Manual.* Department of Electrical and Computer Engineering 80309-0425, University of Colorado, October 1986.

[8] J. H. Saltz. *Automated Problem Scheduling and Reduction of Communication Delay Effects; submitted for publication.* Report 87-22, ICASE, May 1987.

[9] J. H. Saltz and M. C. Chen. Automated problem mapping: the crystal runtime system. In *The Proceedings of the Hypercube Microprocessors Conf., Knoxville, TN*, September 1986.

[10] J.H. Saltz, V. K. Naik, and D.M. Nicol. Reduction of the effects of the communication delays in scientific algorithms on message passing mimd architectures. *SIAM J. Sci. Stat. Comput*, 8(1):s118, 1987.

[11] M. Schultz Y. Saad. *Parallel Implementations of Preconditioned Conjugate Gradient Methods.* Department of Computer Science YALEU/DCS/TR-425, Yale University, October 1985.

END

DATE

FILMED

DTIC

10-88