NASA Contractor Report 178336

ICASE REPORT NO. 87-22

ICASE

AUTOMATED PROBLEM SCHEDULING AND REDUCTION OF SYNCHRONIZATION DELAY EFFECTS

Joel H. Saltz

Contract No. NAS1-18107
July 1987

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

Operated by the Universities Space Research Association

National Aeronautics and
Space Administration

Langley Research Center
Hampton, Virginia 23665

Automated Problem Scheduling and Reduction of Synchronization Delay Effects

Joel H. Saltz
Yale University
and
Institute for Computer Applications in Science and Engineering

Abstract

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 now under development for the automated exploitation of parallelism.

In this paper we present a framework for partitioning very sparse triangular systems of linear equations that is designed to produce favorable performance results in a wide variety of parallel architectures. Efficient methods for solving these systems are of interest because (1) they provide a useful model problem for use in exploring heuristics for the aggregation, mapping and scheduling of relatively fine grained computations whose data dependencies are specified by directed acyclic graphs and (2) because such efficient methods can find direct application in the development of parallel algorithms for scientific computation.

Simple expressions are derived that describe how to schedule computational work with varying degrees of granularity. We use the Encore Multimax as a hardware simulator to investigate the performance effects of using the partitioning techniques presented here in shared memory architectures with varying relative synchronization costs.

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

1 Introduction

A set of techniques is proposed for producing parameterized mappings of the solution of a very sparse triangular system of linear equations onto a range of parallel architectures. The solution of systems in which matrices typically have just a few non-zero elements per row is a particularly interesting problem to investigate for a variety of reasons.

In solutions of such systems, the number of floating point operations that can be performed at any one time is typically rather limited, hence in this problem synchronization and communication overheads play a particularly crucial role in determining the performance that can be achieved. The problem thus provides a simple yet challenging context in which to develop practical methods for automated problem mapping and work aggregating techniques. Efficient methods for solving very sparse triangular systems, easily adapted to a variety of architectures, have direct application in the efficient parallelization of Conjugate Gradient type algorithms preconditioned with incomplete LU factorizations. In our discussions and experimental investigations, we will focus in particular on sparse triangular systems that are generated from incomplete factorizations of matrices arising from discretizations of two dimensional partial differential equations.

Methods for aggregating work represented by a directed acyclic graph are of particular interest for use in conjunction with techniques under development for automating the exploitation of parallelism. In the ongoing Crystal/ACRE [13] parallel programming environment development effort, we are developing simple sets of techniques that can be used for the automated mapping of a variety of problems onto both very tightly coupled systems as well as onto systems that are quite loosely coupled. The study of methods for aggregating the work involved in solving very sparse triangular systems provides a tractable model system for the exploration of methods for aggregating work represented by various types of directed acyclic graphs arising during the compilation and execution of Crystal/ACRE programs.

In the Crystal/ACRE system, 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.

In scientific applications, one frequently wishes to obtain multiple solutions with the same triangular system. Consequently it can be appropriate to spend a modest amount of time prescheduling the computations. Because of the small number of floating point computations required by each row, it is essential that very little time be spent during the algorithm's execution in scheduling row solutions. The approach taken here is to develop a set of methods requiring a set of standard preprocessing techniques that will

allow this problem to be mapped or scheduled onto architectures with widely varying characteristics. The techniques will involve choosing parameters that allow one to make a variety of tradeoffs in the schedule or mapping specification.

We will assign to a single processor all computations pertaining to a row of the matrix. All computations pertaining to a given row are performed during the first phase after the data required is known to be available. 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 determined by the dependencies between the rows of the triangular matrix. The computation is partitioned into phases and simple expressions are derived that describe the scheduling of computational work. This paper will for the most part discuss these methods in the context of architectures that support shared memory and will consider the tradeoffs between synchronization delays and load balance. When appropriate, we will also discuss the use and performance of these mappings in environments that support fast preferential access to a local memory. The mapping techniques discussed here are currently being implemented on a message passing machine, the experimental results will be presented elsewhere.

In the execution of a fine grained problem on a shared memory machine such as the Encore Multimax, primary impediments to the achievement of ideal multiprocessor performance are (1) load imbalance, (2) synchronization delays and (3) programming techniques that introduce computations not found in a corresponding sequential program intended to coordinate the parallel execution of a problem. Our techniques provide a reduction in (1) and (2). Since a prescheduled approach is used, it will be shown that it is possible to keep (3) from becoming a serious problem. The strategies developed here for dealing with these overheads are to: (A) reduce the number of synchronizations required during the solution of the problem, (B) make synchronizations less expensive and (C) improve the balance of load in between synchronizations. Due to the rather slow computation and the rapid synchronization, the Multimax presents a rather benign parallel environment. We will consequently also make use of this machine to provide hardware simulations of algorithm performance in architectures with relatively larger synchronization times.

In message passing environments (such as the Intel iPSC), (1) and (3) remain crucial impediments to the achievement of ideal multiprocessor performance. In current message passing machines communication startups are quite expensive ([12]); techniques that minimize the number of such startups are consequently of crucial importance. The techniques discussed here that reduce the number of synchronizations clearly also reduce the number of startups required. In message passing machines the amount of information communicated also can play an important role in the determination of performance. As we shall see from the analysis of a model problem below, the specification of the parameters in the parameterized mapping also plays an important role in determining the amount of information that must be communicated.

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 [14]. 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. Experimental work has been reported on the NYU ultracomputer prototype involving the use of a wavefront method, where the work involved in solving for rows of sparse triangular linear systems was allocated in a self-scheduled manner [6].

A related body of literature also exists on the solution of triangular systems that are less sparse than the ones described here, these systems are generally obtained from matrix factorizations used in direct methods for solving sparse or non-sparse systems of linear equations. In these problems, the data dependencies between rows of the triangular matrix preclude the efficient use of methods that schedule all work pertaining to a given row at a time when all required data is available.

George [4] 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 [7] 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 [7], 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.

In section 2 we discuss methods for generating a parameterized problem decomposition that allows for considerable flexibility in determining the granularity of parallelism and facilitates inexpensive forms of synchronization. This defines a kind of coordinate system that can be used to advantage in specifying how the problem is to be solved. We derive in section 3, expressions using the parameters arising from the decomposition defined in section 2 that allow the specification of the decomposition of the triangular system in a way that guarantees that the data dependencies in the problem will be respected. The expressions describing the parametrized schedules depend on, among other things, the type of synchronization utilized.

In section 4, through the analysis of a model problem, an analysis is made of the the tradeoffs between load imbalance and synchronization costs, as well as the tradeoffs between load imbalance and communication costs. An inexpensive method for explicitly balancing the load during each a phase of computation is described in section 5. In section 6 the results are reported of experimental investigations on the Encore Multimax multiprocessor that (1) explore the effect of parametric variations of granularity on performance, (2) evaluate the value of explicitly balancing load during each computational phase and (3) compare the performance obtained through the use of different synchronization techniques.

2 Problem Partitioning

2.1 Overview

The methods of problem partitioning described here have a strong geometrical motivation and will consequently first be introduced in a geometrical context. 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.

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 [14]. 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 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

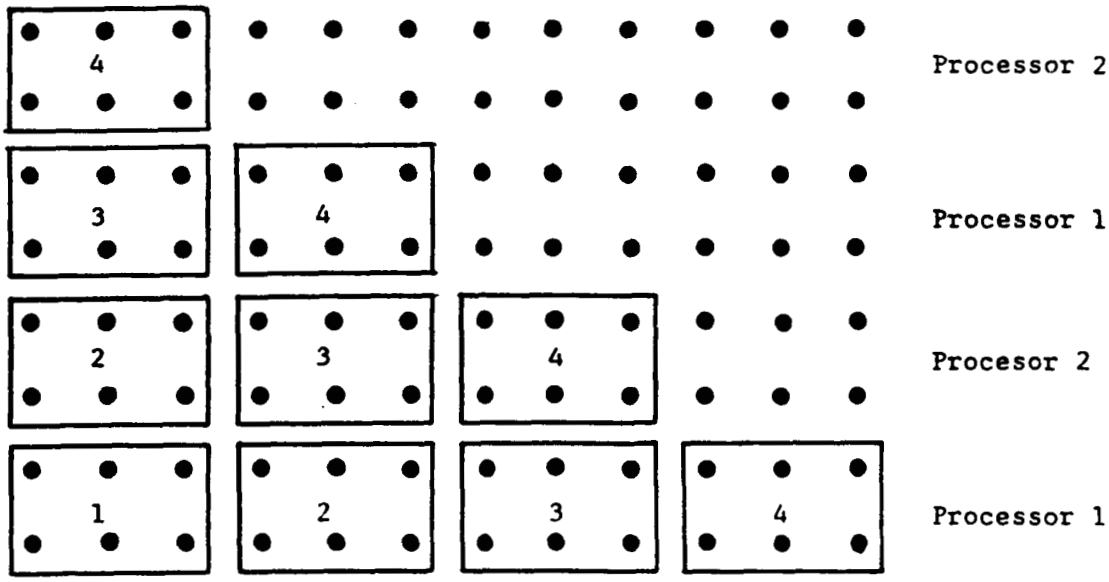

Figure 1: Two processors, five point stencil, block size = 2, window = 3. Numbers designate computational phases.

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 [14] 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 [14] 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 analytically and experimentally in the following in order to explore 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 a window of w 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 determined by dataflow considerations. If one wishes to aggregate points in blocks into larger units, with each unit 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

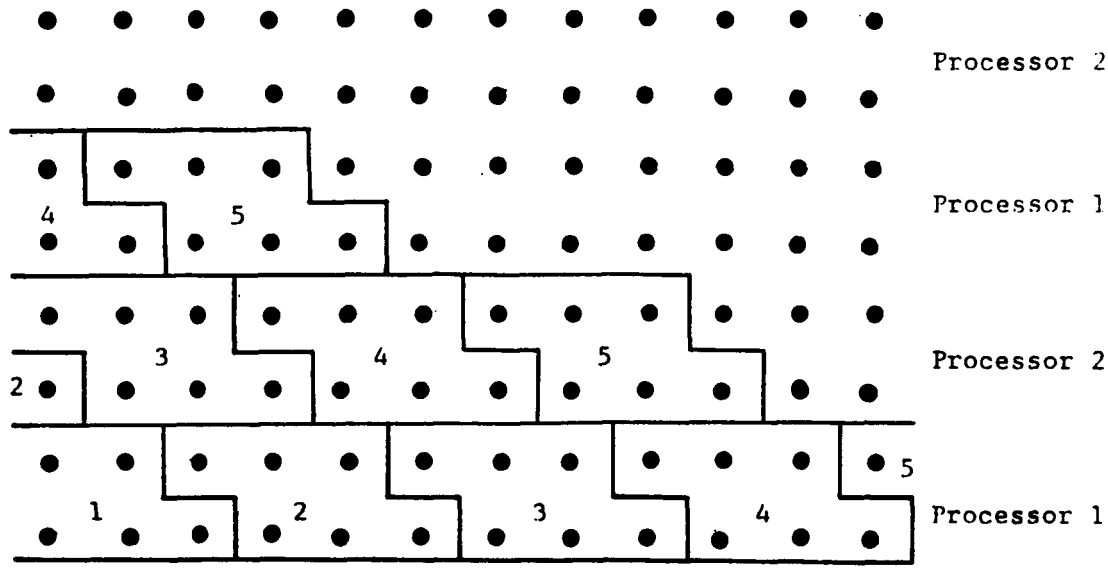

Figure 2: Two processors, nine point stencil, block size = 2, window = 3. Numbers designate computational phases.

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 have developed and tested 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], [10]. 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 [9] allows the wavefronts of a DAG 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

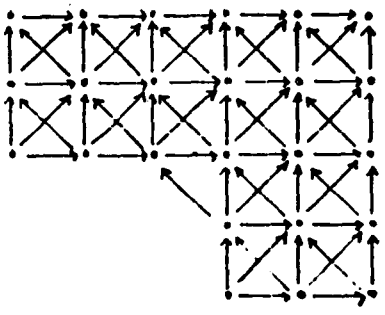

Figure 3: Data Dependencies

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 substantial, as will be shown in the experimental results below. On many message passing machines, (e.g. the Intel iPSC [12]), 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 by 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*.

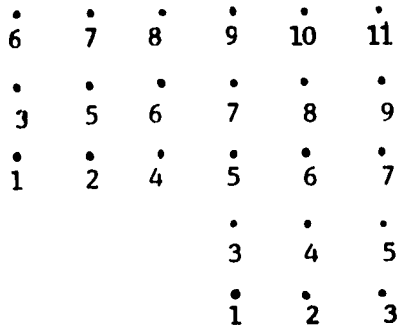

Figure 4: Wavefronts

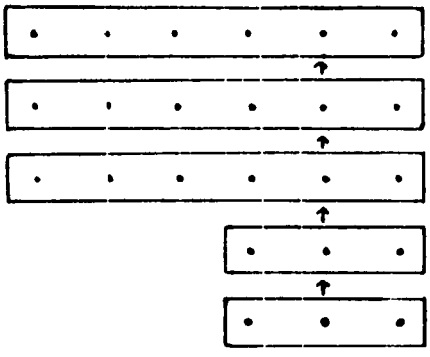

Figure 5: Strings and String DAG

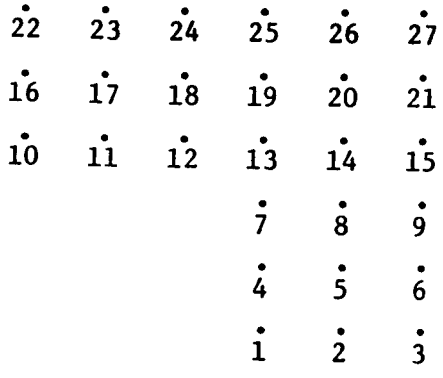

Figure 6: Mesh Point Ordering

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.

When a DAG originates from the incomplete decomposition of a two dimensional domain, it is possible to make sure that the strings are chosen in a much more controlled manner. The decomposition can be determined by the way in which the mesh points are ordered in the formation of the matrix. It is simple to arrange for the algorithm to give preference to lower numbered rows when forming new strings from start vertices D, and to attempt to incorporate rows into a growing string in order of increasing row number. For example, figure 6 depicts a simple way of numbering the grid points from figures 3 and 4 to ensure the production of the string decomposition depicted in 5.

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 will be shown below 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.

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 Overview

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 is 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 executions during the execution of the algorithm. We consequently have chosen to use a prescheduled approach in which the rows to be computed by a string at a given phase in a computation are chosen implicitly 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 and are assigned to consecutively indexed 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.

For barrier and dataflow synchronization methods, 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.

The phase during which one can assure data availability for a given computation depends on (1) the synchronization mechanism used, (2) the data dependency relationships between the strings of the string DAG and (3) the data dependencies between the blocks into which the string DAG is partitioned.

3.2 Barrier Synchronization

We will now discuss scheduling when barrier synchronization methods are employed; these methods assure that all processors have finished phase $p-1$ before any processor is allowed to begin phase p.

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 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). A wavefront of this coarse grained DAG will be called a *block wavefront.* It should be noted that one may assign any work scheduled during a phase to any processor one desires. In problems described by irregular DAGs it will be shown below that explicitly balancing the processor load in a block wavefront during each phase can be quite 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 consecutively indexed 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.

□

3.3 Barrier Synchronization - 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 consecutively indexed 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 $1 \leq q < i+1$.

□

3.4 Dataflow Synchronization

When dataflow synchronization is used, at any given time processors may be performing computations specified by different phases. To ensure that needed data is available when a processor performs its computations during a phase, we must construct a schedule that makes adequate allowance for the weak interprocessor synchronization. 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. The schedule to be presented below also takes into account the data dependencies between blocks of strings in the string DAG.

To illustrate the role of inter-block data dependencies in determining schedules for performing work when dataflow synchronization is used, consider an important special case that arises when inter-block data dependencies are restricted to nearest neighbors, i.e. block $i+1$ requires data only from block i. In this case the constraints for scheduling wavefront execution during a given phase p are identical for dataflow and barrier synchronization mechanisms. In either of these synchronization mechanisms, a processor P beginning phase $p+1$ has to know that its predecessor has finished phase p. Block i assigned to P requires data only from block $i-1$ so that no provision need be made for the possibility that blocks upon which i depends have not yet computed values for wavefronts corresponding to phase p.

When dataflow synchronization is employed, proposition 3 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 (1) the first block computes exactly w wavefronts per phase, and (2) block i can require data only from blocks j, $\max(i-d,1) \leq j < i$. Note that when $d = 1$ we have the previously discussed case of nearest neighbor data dependencies. The string DAGs that arise from many problems obtained from incomplete factorizations of matrices arising from partial differential equations are frequently characterized by small values of d.

The expression is obtained by mapping a chain of logical processes in a wrapped manner onto the P processors of the machine. For the sake of tractability, the expressions are derived under the assumption that the logical processes assigned to a given processor are assumed to be independent of one another. In the actual system, all processes assigned to the same processor must complete a given phase before beginning the next. Creating schedules using the assumption that processes assigned to a processor are independent assures us that computations will not be undertaken until the required data are available, since such a schedule allows for the possibility that all processes on a given processor could be computing the same phase at a given time. When $d < P$, the expressions derived yield the largest wavefront that could be scheduled by a given block at a particular time, when $d \geq P$ this is no longer necessarily the case.

Proposition 3 *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 consecutively indexed processors in a wrapped manner. Assume that each block constitutes a process that executes its computations in phases subject to the constraint that at any time, if block i has finished phase p, block $i+1$ can complete all phases with numbers less than or equal to $p+1$. Furthermore assume that each block i requires data only from blocks* $\max(i-d,1)$ *through* $i-1$.

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.

For $i \geq 2$, W_p^i is given by the expression

$$W_p^i = \max(\lceil p/d \rceil, w(p-i+1) + \lceil (i-1)/d \rceil) \tag{1}$$

In scheduling work for block $i+1$ during phase p, we must take into account the numbers of the wavefronts corresponding to the latest available results from blocks $\max(1, i-d+1) \leq j < i+1$, since block $i+1$ 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 $p \leq 0$. The number of the smallest wavefront corresponding to any result that might be needed by block $i+1$ at the beginning of phase p may be expressed for $i \geq 1$ as

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

Consequently,

$$W_p^{i+1} = \min_{\max(1,i-d+1) \leq j < i+1} W_{p-i+j-1}^j + 1 \tag{2}$$

for $p \geq 1$, $i \geq 1$.

We now use the above to prove by induction on $i \geq 2$ that for all $p \geq 1$, if

$$\hat{W}_p^i = \max(\lceil p/d \rceil, w(p-i+1) + \lceil (i-1)/d \rceil), \tag{3}$$

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

We first establish the base of the induction. By (2), $W_p^2 = W_{p-1}^1 + 1$; as $W_p^1 = wp$ by assumption it follows that $W_p^2 = w(p-1)+1$. From (3), $\hat{W}_p^2 = max(\lceil p/d \rceil, w(p-1)+\lceil 1/d \rceil)$. For $p \geq 1$, the above expression is equal to $w(p-1)+1$. Hence $W_p^2 = \hat{W}_p^2$.

Assume that $W_p^j = \hat{W}_p^j$ for $2 \leq j \leq i$, $p \geq 1$, we will show that $W_p^{i+1} = \hat{W}_p^{i+1}$. We first consider the case when

$$p - i + \max(1, i-d+1) - 1 \leq 0. \tag{4}$$

From (3), for all $1 \leq j \leq i$ and $p \geq 1$, $W_p^j \geq 1$. Since $W_p^j = 0$ for $p \leq 0$, from (2) and (4) it follows that $W_p^{i+1} = 1$.

We will show that when (4) is satisfied, it is also the case that $\hat{W}_p^{i+1} = 1$. By (3) $\hat{W}_p^{i+1}$ may be expressed as

$$\hat{W}_p^{i+1} = \max(\lceil p/d \rceil, w(p-i) + \lceil i/d \rceil). \tag{5}$$

When $i \leq d$, from (4) we have $p \leq i$ as well as $p \leq d$. Thus we have $0 < p/d \leq 1$, $w(p-i) \leq 0$ and $0 < i/d \leq 1$ and hence by (5), $\hat{W}_p^{i+1} = 1$.

When $i > d$, from (4), $p \leq d$. Thus $0 < p/d \leq 1$ and $\lceil p/d \rceil = 1$. $i > d$ also implies that

$$w(p-i) + \lceil i/d \rceil \leq w(d-i) + \lceil i/d \rceil.$$

and

$$w(d-i) + \lceil i/d \rceil \leq w(d-i) + i/d + 1.$$

Now $w(d-i)+i/d+1 \leq 1$ if and only if $wd(d-i) \leq -1$. Since $w \geq 1$, $d \geq 1$ and $d-i \leq -1$, we ascertain that $w(p-i) + \lceil i/d \rceil \leq 1$. Thus when (4) is satisfied, $\hat{W}_p^{i+1} = 1$, and hence in this situation we have shown that $W_p^{i+1} = \hat{W}_p^{i+1}$.

We shall now prove the induction hypothesis when (4) is not satisfied, i.e. when

$$p - i + \max(1, i-d+1) - 1 \geq 1. \tag{6}$$

When $i - d + 1 \geq 2$ we obtain from (2) and (5)

$$W_p^{i+1} = \min_{i-d+1 \leq j < i+1} \max\left(\lceil \frac{p-i+j-1}{d} \rceil, w(p-i) + \lceil \frac{j-1}{d} \rceil \right) + 1$$

and hence

$$W_p^{i+1} = \max\left(\lceil \frac{p}{d} \rceil, w(p-(i+1)-1) + \lceil \frac{(i+1)-1}{d} \rceil \right).$$

When $i - d + 1 \leq 1$,

$$W_p^{i+1} = \min\left[\min_{2 \leq j < i+1} \max\left(\lceil \frac{p-i+j-1}{d} \rceil, w(p-i) + \lceil \frac{j-1}{d} \rceil\right), \max\left(\lceil \frac{p-i}{d} \rceil, w(p-i)\right)\right] + 1$$

hence

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

By (6), $p - i \geq 1$ and consequently $w(p-i) \geq \lceil (p-i)/d \rceil$, and thus

$$\max(\lceil \frac{p-i}{d} \rceil, w(p-i)) + 1 = w(p-i).$$

Since $1 \leq i \leq d$, $\lceil i/d \rceil = 1$ and thus

$$W_p^{i+1} = w(p-i) + \lceil \frac{i}{d} \rceil = w(p-(i+1)-1) + \lceil \frac{(i+1)-1}{d} \rceil$$

We have thus shown that $W_p^{i+1} = \hat{W}_p^{i+1}$ and the proposition is proved.

□

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 are 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 $n = r_1P$ and $m = r_2P$, 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 is given by:

$$\frac{SP(P-1)\min(r_1,r_2)}{r_1r_2P^2}+T_SP(r_1P+r_2P)$$

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

$$\frac{SP(P-1)}{r_1P^2}+T_SP(r_1P+r_2P). \tag{7}$$

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 Wavefront Longest Processing Time Scheduling

Propositions (1) and (2) describe a parametric method of constructing work schedules for performing the calculations required to solve the problem in question, when a form of barrier synchronization is used between phases. As stated previously, these propositions may also be regarded as a way of parametrically describing the wavefronts of a coarse grained DAG, each vertex of which represents the solution of a number of rows. It is consequently natural to consider balancing the processor load for the block wavefronts of this DAG, i.e. balancing the load during each phase of computation. We choose to utilize a prescheduled approach for allocating work although it is also possible to allocate work represented in these wavefronts in a self scheduled manner.

The scheduling of independent tasks to obtain a minimum finishing time is known to be NP-hard. There exist a variety of methods for obtaining approximate solutions to this problem [5], [8]. One method that has been extensively studied is the *Longest Processing Time* or the *LPT* schedule. An LPT schedule is one that is the result of an algorithm which, whenever a processor becomes free, assigns to that processor a task which requires a run time longer than that of any task not yet assigned.

While the relative difference in performance between results obtained with LPT and optimal schedules in the worst case is $1/3 - 1/(3P)$ where P is as usual the number of processors [5] , in simulations investigating the error that might be expected given a variety of randomly generated datasets, it was found that the difference between the generated solution and the optimal solution was only a few percent [2]. Because the prescheduled work assignment must use approximate work estimates based on calculations involving the number of floating point operations required to solve for a row of the triangular system, a heuristic such as LPT is likely to perform as well as a more expensive scheme to find a closer approximation to the optimal schedule.

The LPT rule requires time $r \log r$ to schedule the execution of r tasks. For a fixed choice of window w and block size b in the work schedule, the amount of computation in a wavefront of a triangular solve arising from a zero fill incomplete factorization of a matrix generated by a regular n by n mesh increases with order n. In an asymptotic sense then, even the rather inexpensive LPT scheduling algorithm has somewhat unfavorable properties. The performance obtained through the use of the LPT scheduling algorithm is compared with that obtained through the use of a wrapped assignment of strings in the

following section.

6 Experimental Results

6.1 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 [3].

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 for the length of each computation. This programming methodology is further described in [11]. 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 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.

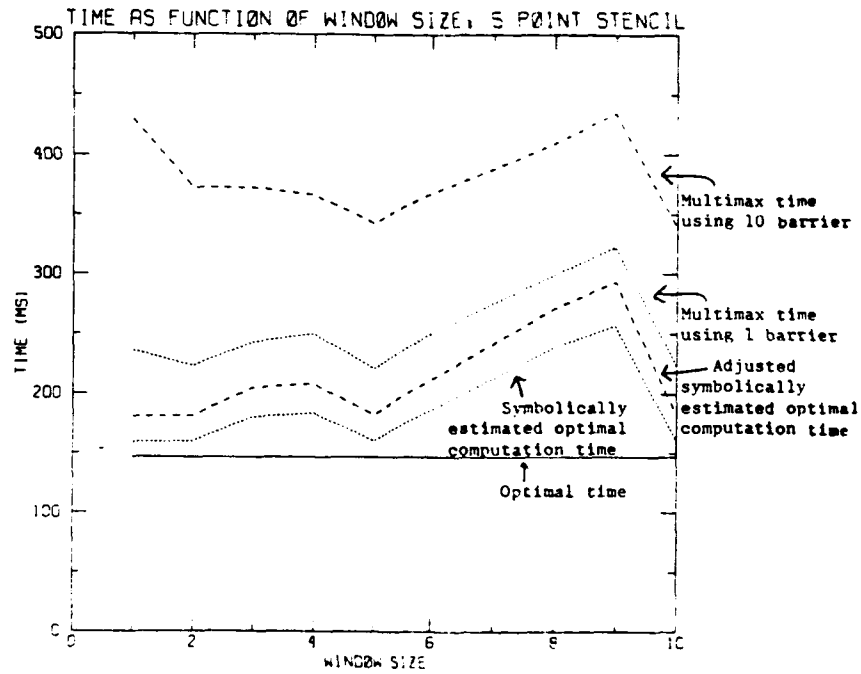

Figure 7: Effect of Window Size on Execution Time. Matrix from a 100 by 100 mesh, 5 point template. 10 processors used, timings for 25 consecutive trials averaged.

6.2 Effect of Window and Block Size on Performance

The effect of window size on execution time was investigated. The data depicted in figure 7 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. Barriers were used for synchronization. Note that this matrix is extremely sparse, there are no more than two non-zero off diagonal elements in any matrix row. This matrix, along with the others described here, has unit diagonal elements. The forward solve consequently does not involve divisions. The parallelism encountered here is consequently quite fine grained. The strings in this problem partition the domain into horizontal slices as was described in the analysis of the model problem previously discussed. Horizontally oriented strings were used in all experimental results reported here unless another orientation is explicitly specified. Ten processors were used to solve this problem, and a block size of one was employed.

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 7 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 7 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 (7), 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 7, 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.

Finally, the time required to solve the the problem using the sequential code was divided by the number of processors used; this is denoted as optimal time.

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 8 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

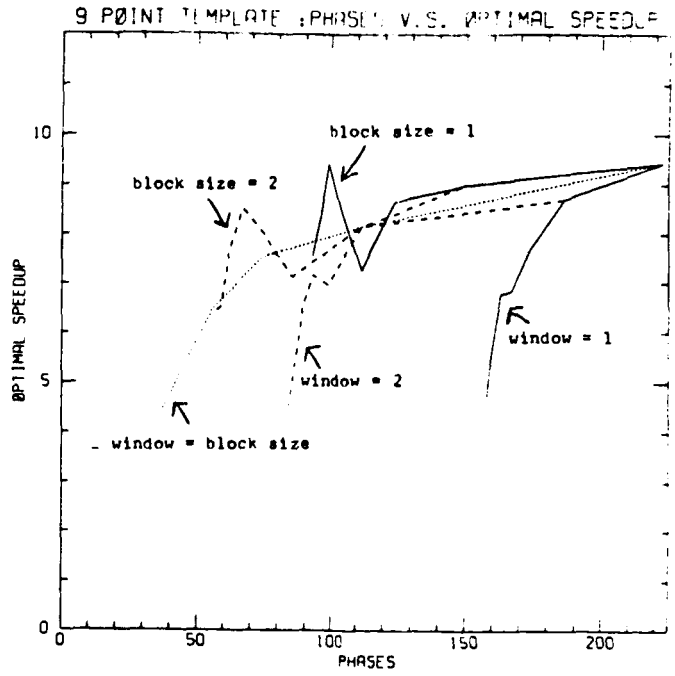

Figure 8: Symbolically estimated optimal speedup versus phases required to solve problem on 12 processors. Matrix from a 75 by 75 point mesh, 9 point template. Horizontal strings used.

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 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 9. 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.

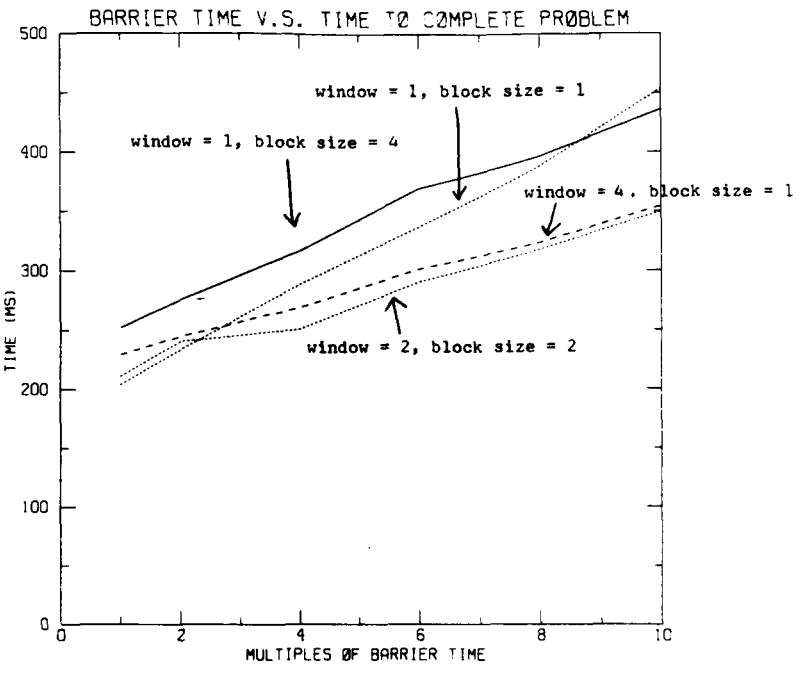

Figure 9: Effect of window, block size on execution time. Matrix from a 75 by 75 point mesh, 9 point template. 12 processors used, timings for 25 consecutive trials averaged.

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 computation. In figure 9, 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.

6.3 String Orientation Effects

The relative merits of using horizontal versus diagonal strings in partitioning a mesh with a 9 point template were investigated. In figure 10 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,

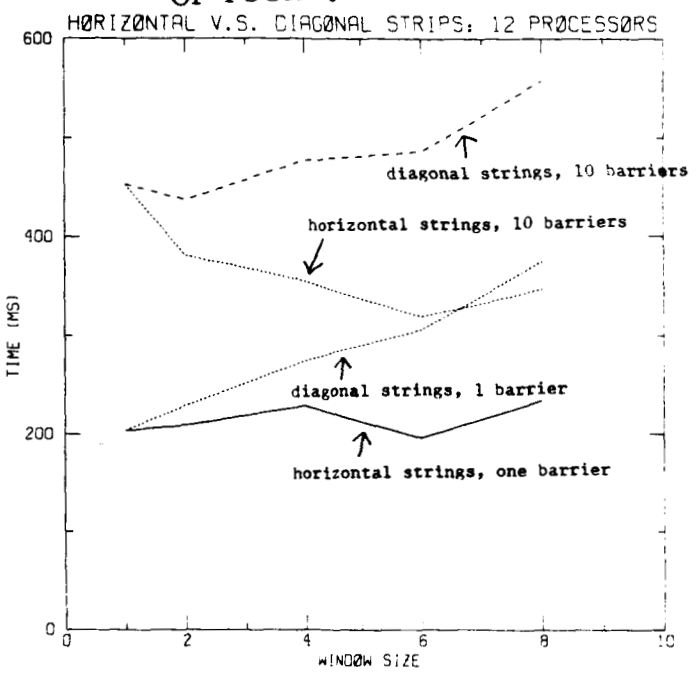

Figure 10: Effect of string orientation on execution time. Matrix from a 75 by 75 mesh, 9 point template. 12 processors, block size = 1, timings for 25 consecutive trials averaged.

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 11 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.

6.4 Comparison Between LPT and Wrapped Scheduling

When barrier synchronization is utilized, the work required to compute the blocks during a phase can be scheduled in an explicit manner. We have discussed how one might use an inexpensive heuristic such as LPT to do this scheduling. Experimental comparisons will now be made between the performance that can be achieved through the use of LPT and that obtained by assigning the workload to the processors in a wrapped fashion.

The difference in performance between these scheduling methods is only expected to

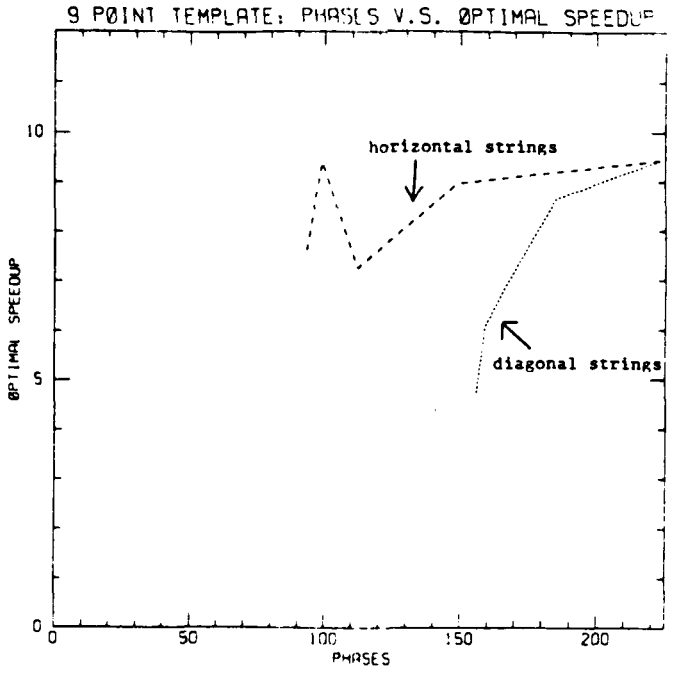

Figure 11: Symbolically estimated optimal speedup versus phases required in solving problem on 12 processors. Matrix from a 75 by 75 mesh, 9 point template. Size of block = 1.

be noticeable in problems with some degree of irregularity. If during each phase a number of blocks with identical computational requirements had to be executed, a wrapped assignment should lead to an optimal balance of load during that phase. Note that this does not mean that the load will be balanced, since the number of blocks assigned to processors can differ by one.

Of course, the computational requirements of blocks to be computed during a phase are not identical, even in problems derived from rectangular meshes in which a uniform template was utilized. In such problems, the blocks derived from mesh points near the boundaries of the domain will generally require smaller amounts of computation than those derived from points further away from the boundaries. Two sets of experiments were performed to compare LPT and wrapped scheduling using a matrix generated from a 80 by 80 point mesh with a 5 point template and a matrix generated from the same mesh using a 13 point template. Block and window size were varied and 10 processors were used. The performance obtained using the two scheduling methods were compared using both symbolically estimated optimal speedups and measured runtimes on the Encore Multimax. The symbolically estimated optimal speedups were not effected by the scheduling mechanism used, and the Multimax runtimes measured showed minimal differences in no consistent direction.

More substantial differences in run times were noted in problems possessing irregular-

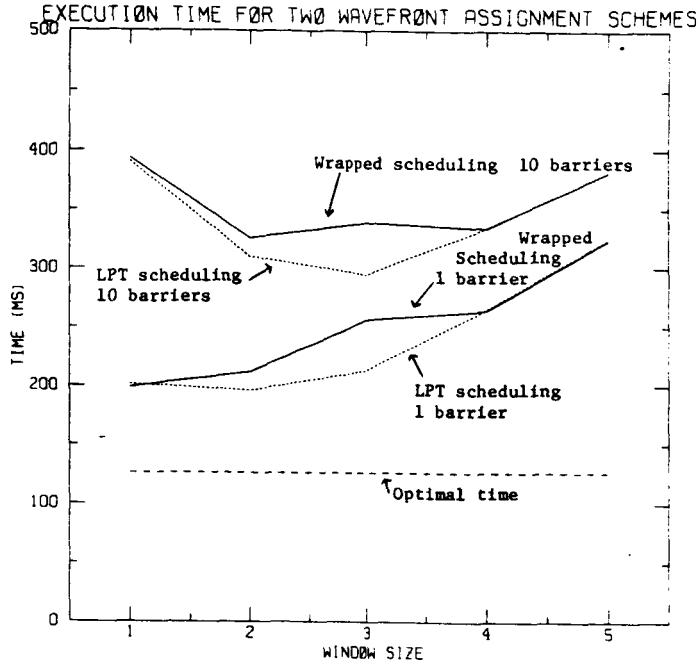

Figure 12: Run times from wrapped and LPT scheduling. Matrix from 80 by 80 mesh, mesh rows 1 to 29, 51 to 80: 5 point template; rows 30 to 50: 13 point template. 10 processors, block size equal window size, timings for 25 trials averaged.

ities that would lead to more substantial differences in the computational requirements of blocks during each phase. The data depicted in figure 12 was obtained through a forward solve on 10 processors of the zero fill factorization of a matrix generated using a 80 by 80 point square mesh. Points in mesh rows 1 through 29 and rows 51 through 80 employed a 5 point template; points in rows 50 through 80 employed a 13 point template. The block size was set equal to the window size and both were varied from 1 to 5. The time required to solve the problem was measured when LPT or wrapped scheduling was used to balance load in each block wavefront. These measurements were made when one barrier was used; to simulate the effects of these manipulations on an architecture requiring more expensive synchronization, measurements were also made using 10 barriers. For window sizes of 2 and 3 LPT scheduling led to shorter run times than did wrapped scheduling. When a window of size one was used in this problem, no significant difference between the scheduling mechanisms was measured.

When only one barrier is used, there is no advantage to using a window of size greater than one in any event. Consequently on the Encore, for this problem, there appears to be nothing to be gained from using either LPT scheduling or from using methods to increase the granularity of parallelism. On architectures where the costs of synchronization are larger compared to the costs of computation, both LPT scheduling and the use of these granularity increasing methods will be advantageous.

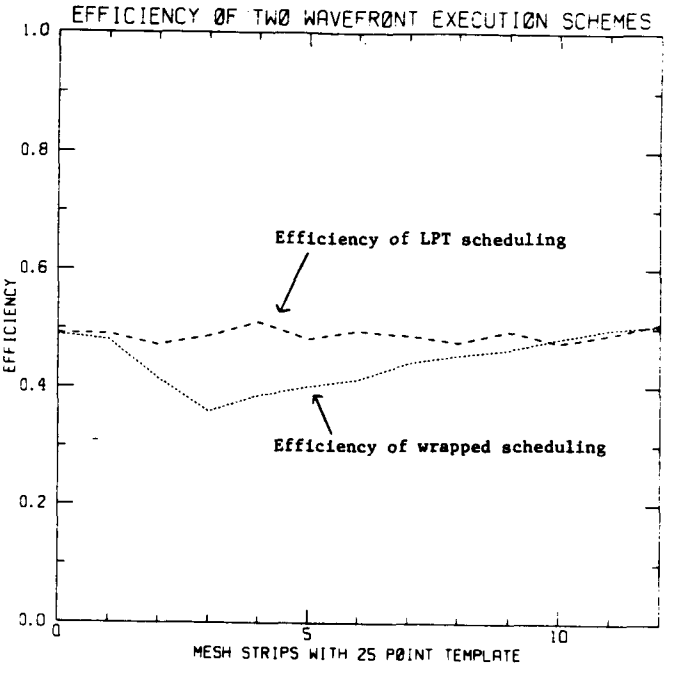

Figure 13: Efficiency of wrapped vs LPT scheduling. From Multimax times. Matrix from 100 by 40 mesh, bottom strips have 25 point template, rest have 5 point template. 12 processors, block,window equal 1, timings for 25 trials averaged.

A set of problems exhibiting more dramatic load imbalances was also examined. A lower triangular system was produced by the incomplete factorization of a matrix generated from a 100 by 40 mesh point matrix in which the bottom s strips had a 25 point template, and the $40 - s$ upper strips had a 5 point template. Both the block size and the window size were set equal to one and a single barrier was used for synchronization. Figure (13) depicts the efficiency with which 12 processors of the Multimax solves the system as s is varied from 0 to 12. Efficiency is defined here as the ratio of the time required to solve the problem using a separate sequential code on one processor to the product of the measured time to solve the problem and the number of processors used.

The efficiency obtained through the use of LPT scheduling does not vary much with s, remaining approximately 0.50. The efficiency exhibited by the wrapped scheduling decreases to a low of 0.36 for s equal to 3, but is comparable to the efficiency obtained through the use of LPT when s is close to either 0 or 12. The reasons for this appear to be quite straightforward. When one has, during each phase, a number of very time consuming blocks that is small compared to the number of processors used, one risks a serious load imbalance when a wrapped assignment strategy is used. As the number of time consuming blocks encountered during each phase increases to approach the number of processors, the amount of wasted processor capacity decreases.

Further insight into the situation is provided by figure 14, in which the problem de-

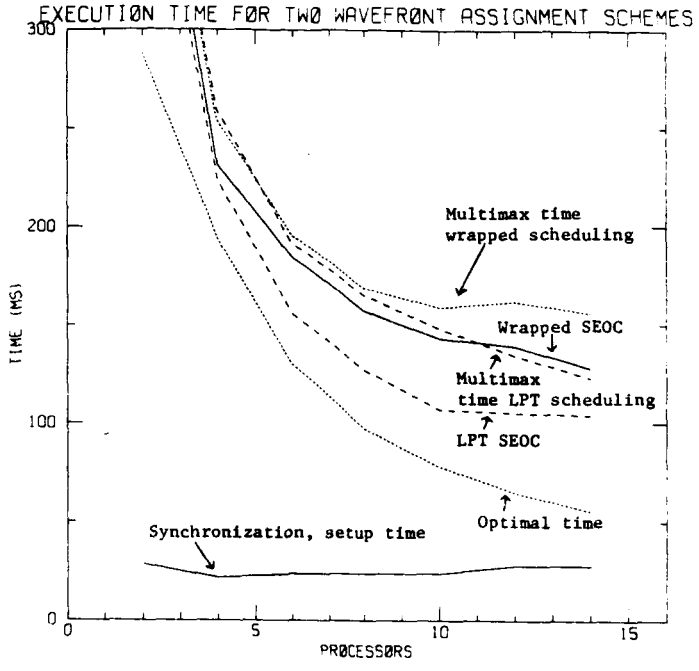

Figure 14: Run time of wrapped and LPT scheduling. Matrix from 100 by 40 mesh, bottom 5 strips have 25 point template, other strips have 5 point template. 12 processors, block, window equal 1. Timings for 25 trials averaged.

scribed above is solved when $s = 5$ for varying numbers of processors. In this figure we display the symbolically estimated optimal computation times (abbreviated as SEOC in the figure) obtained from both the wrapped and the LPT schedules, along with the measured Multimax execution times. As in previous figures, the optimal time is defined as the sequential time divided by the number of processors. The time required for both synchronization and for executing the program control structure was also measured by making program measurements with the floating point calculations commented out.

Insight into the sources of inefficiency in this problem may be obtained by examining this figure. There is a substantial difference between the optimal time and both symbolically estimated optimal computation times, suggesting that load imbalance makes a significant contribution to the departure from the optimal run time observed here. This conclusion is reinforced by observing the measured synchronization and setup time.

It may be noted in this figure that the symbolically estimated optimal computation time calculated for the wrapped assignment added to the synchronization and setup time produce numbers that are quite close to the measured multimax time for wrapped scheduling. This correspondence has been noted in the case of the wrapped assignment in a variety of other problems not presented here and gives confidence in the accuracy of the measurements. When LPT scheduling is used, the symbolically estimated optimal computation time appears to have less predictive value, as seen in figure 14, and has been noted in other

measurements. For instance, while the efficiencies obtained through using LPT in figure 13 vary little with s, the symbolically estimated optimal computation times vary with s to a substantial extent. For instance for s equal to 2, the measured time taken to solve the problem using the LPT algorithm was 112.16, while the SEOC was 69.4; for s equal to 3, the measured time was 118.3 but the SEOC was 99.31. The difference in synchronization time between the two cases was, however, quite minimal.

It should be remembered however, that the LPT scheduling method uses symbolically estimated computation times to perform its load balancing. SEOC estimates are clearly not completely accurate. If one measures the SEOC obtained, after rescheduling computations using a method that produces a near optimal processor schedule based on operation counts, one will tend to obtain overly optimistic estimates of the execution time. This observation points to the obvious importance of accurate run time estimates when performing LPT scheduling.

6.5 A Comparison between Barrier and Dataflow Synchronization

While barrier synchronization is relatively inexpensive on the Encore Multimax, nearest neighbor synchronization is less expensive still as it can be implemented in a way that requires, each processor, only one shared variable increment followed by a busy wait. Figure 15 depicts a comparison between execution times measured when a barrier was utilized and execution time measured when dataflow synchronization was employed. Both barrier and dataflow synchronization - setup time are also measured and depicted. The problem solved here originates from a 75 by 75 point mesh with a 9 point template, the window and block size are 1. It is evident from this figure that dataflow synchronization is less expensive than barrier synchronization.

7 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 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 granu-

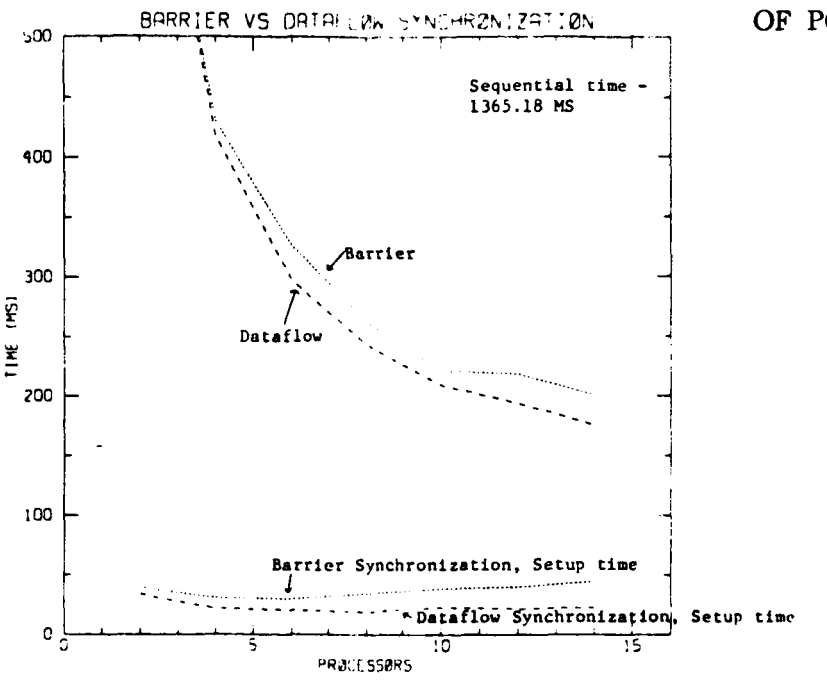

Figure 15: Execution time of barrier and dataflow synchronization. Matrix from 75 by 75 mesh, 9 point template. Window, block equal 1. Timings from 25 consecutive trials averaged.

larity 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 load imbalance and computational granularity. Studies of this tradeoff obtained through comparing operation counts and the number of computational phases required to solve a problem suggest that the load balance granularity tradeoff becomes smoother when window 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.

The Longest Processing Time scheduling heuristic was used to explicitly schedule the work performed during each phase. The performance obtained through the use of this method was compared to the time required for assigning blocks to processors in a wrapped manner. LPT scheduling was found in some cases to decrease the run time in problems with irregular work demands during a typical phase; it had no measurable effect in very uniform problems. Through the use of symbolically estimated optimal computation times obtained through operation counts, along with direct measurements of synchronization times, for both LPT and wrapped scheduling, the origins of the measured run times were traced.

An experiment was conducted comparing the execution costs incurred through the use of barrier and dataflow synchronizations on the Multimax. Despite the use of a rather efficient barrier, time was clearly saved though the use of the even less expensive dataflow synchronization method. Dataflow synchronization has, however, a number of drawbacks. When non local data dependences occur between blocks, we see from proposition 3 that the number of phases required to complete a problem increases. Furthermore it does not appear that one can easily make use of methods for balancing the load between phases when dataflow synchronization is employed.

To sum up, in this paper we present a framework for partitioning very sparse triangular systems of linear equations that appears to be flexible enough to produce favorable performance results in a wide variety of parallel architectures. In this paper we have used the Multimax as a hardware simulator to investigate the performance effects of using the partitioning techniques presented here in shared memory architectures with varying relative synchronization costs.

A few comments are in order on which of these techniques we can recommend for use on the current Multimax. On the Encore Multimax, due to it's low ratio of synchronization costs to costs of floating point operations, there does not appear to be an advantage in aggregating work to increase the computational granularity. Balancing load within each phase of computation does appear to be advantageous in this architecture, although further practical experience is required to discover when the overhead required for this extra stage of scheduling is worthwhile. The use of dataflow synchronization on the Multimax also appears to be advantageous although it's use precludes that of wavefront LPT balancing. One can limit the problem decomposition process to the identification of wavefronts if one has no need to increase granularity through the use of windows or to use strings to implement dataflow synchronization. Hence, on the Encore there should be no reason to pay both the overhead for string decomposition and for LPT balancing.

8 Acknowledgements

I wish to thank Stan Eisenstat for making available an inexpensive barrier he has developed on the Multimax and to thank Stan Eisenstat, Dave Nicol and Martin Schultz for useful

discussions. Bob Voigt's careful reading of the manuscript and editorial suggestions are also gratefully acknowledged.

References

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

[2] E. Coffman, M. Garey, and D. Johnson. An application of bin packing to multiprocessor scheduling. *SIAM Computing*, 7(1):1–17, 1978.

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

[4] 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.

[5] R. Graham. Bounds on multiprocessor timing anomalies. *SIAM Jr. on Appl. Math*, 17(2):416–429, 1969.

[6] A. Greenbaum. *Solving Sparse Triangular Linear Systems Using Fortran with Paralllel Extensions on the NYU Ultracomputer Prototype.* Report 99, NYU Ultracomputer Note, April 1986.

[7] 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.

[8] E. Horowitz and S. Sahni. *Fundamentals of Computer Algorithms.* Computer Science Press, Rockville Maryland, 1978.

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

[10] 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.

[11] 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.

[12] 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.

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

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

Standard Bibliographic Page

1. Report No. NASA CR-178336 ICASE Report No. 87-22	2. Government Accession No.	3. Recipient's Catalog No.
4. Title and Subtitle AUTOMATED PROBLEM SCHEDULING AND REDUCTION OF SYNCHRONIZATION DELAY EFFECTS		5. Report Date July 1987 6. Performing Organization Code
7. Author(s) Joel H. Saltz		8. Performing Organization Report No. 87-22
9. Performing Organization Name and Address Institute for Computer Applications in Science and Engineering Mail Stop 132C, NASA Langley Research Center Hampton, VA 23665-5225		10. Work Unit No. 505-90-21-01 11. Contract or Grant No. NAS1-18107
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

15. Supplementary Notes

Langley Technical Monitor:
J. C. South

Submitted to SIAM J. Sci. Statist. Comput.

Final Report

16. Abstract

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 now under development for the automated exploitation of parallelism.

In this paper we present a framework for partitioning very sparse triangular systems of linear equations that is designed to produce favorable performance results in a wide variety of parallel architectures. Efficient methods for solving these systems are of interest because (1) they provide a useful model problem for use in exploring heuristics for the aggregation, mapping and scheduling of relatively fine grained computations whose data dependencies are specified by directed acyclic graphs and (2) because such efficient methods can find direct application in the development of parallel algorithms for scientific computation.

Simple expressions are derived that describe how to schedule computational work with varying degrees of granularity. We use the Encore Multimax as a hardware simulator to investigate the performance effects of using the partitioning techniques presented here in shared memory architectures with varying relative synchronization costs.

17. Key Words (Suggested by Authors(s))	18. Distribution Statement
triangular solve, load balancing, mapping, aggregation, parallel algorithm	62 - Computer Systems 64 - Numerical Analysis Unclassified - unlimited

19. Security Classif.(of this report)	20. Security Classif.(of this page)	21. No. of Pages	22. Price
Unclassified	Unclassified	37	A03

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

NASA Langley Form 63 (June 1985)