4

LUX ET VERITAS

YALE UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE

4

Yale University
Department of Computer Science

Preconditioned Krylov Solvers and Methods for Runtime Loop Parallelization

Doug Baxter Joel Saltz Martin Schultz and Stan Eisenstat

YALEU/DCS/TR- 655
October 1988

This work has been supported in part by the U.S. Office of Naval Research under Grant N00014-86-K-0310

Table of Contents

Abstract

We make a detailed examination ~~was made~~ of the performance achieved by a Krylov space sparse linear system solver that uses incompletely factored matrices for preconditioners. We compared two related mechanisms for parallelizing the computationally critical sparse triangular solves and sparse numeric incomplete factorizations on a range of test problems. From these comparisions we drew several interesting conclusions about methods that can be used to parallelize loops of the type found here.

The performance we obtain is brought into perspective by comparisons with timing results from a Cray X/MP supercomputer. Performance on an Encore Multimax/320 with relatively modest computational capabilities comes within a small factor of the performance on a comparable code run on a Cray X/MP.

1. Introduction

We report the results of a detailed study of the performance obtained in parallelizing a Krylov-space linear system solver, preconditioned using incompletely factored matrices (PCGPAK) . We feel that an understanding of how one proceeds to parallelize such a sparse solver is of intrinsic interest to researchers and practitioners in scientific computing. We also have found this problem to be ideally suited for investigating issues that arise in the development of systems for automated runtime parallelization.

The structure of the matrix describing the sparse linear system to be solved plays a key role in determining the data dependencies between variables in many loops. Since this matrix is generally input at runtime or generated during program execution, information concerning data dependencies found in critical inner loops cannot be ascertained at compile time. A two step method is used to parallelize the loops in question. In the first step, index sets of the loops are reordered using a topological sort type algorithm. A new loop structure is produced by modifying the loop to be parallelized. This new loop structure is designed to make use of the reordered index set. One type of loop structure partitions the loop into a sequence of fully parallelized do loops that are separated from one another by global synchronizations. The other loop structure is a generalization of the doacross loop [3]. Instead of using global synchronizations, we require processors to write into a specified location in an array in shared memory when work pertaining to a given index is completed. Before a variable value can be used, a processor makes sure that the appropriately updated value has been calculated by repeatedly checking a designated location in shared memory. This is commonly called *busy waiting.*

Topological sorts have been used by a number of researchers to reorder row substitutions and increase available parallelism [7],[9],[4],[1],[10].

We will present and compare in detail the performance that can be obtained in the sparse linear system solver through the use of these two different types of loops that use the reordered index set. To fully explain the performance tradeoffs between these loop structures, we need to be able to quantitatively explain the performance we are observing. We present a set of experiments and analysis able to account for how time is spent in the two different loop constructions.

These experimental results are put into perspective by showing that the performance we obtain on a multiprocessor with relatively modest computational capabilities comes within a small factor of the performance that can be obtained by a comparable code on a Cray X/MP supercomputer.

We will also sketch how the methods used to parallelize loops in these sparse matrix problems can be generalized. Symbolic transformations can be employed to produce program segments that reorder index sets and program segments that are able to use the reordered index sets to increase concurrency. These transformations can be used as the basis for a variety of language extensions, these extensions can be used to guide a preprocessor in restructuring source codes in ways that increase parallelism.

2. Preconditioned Krylov Methods Background

We briefly present the basics of Krylov methods such as are found in PCGPAK. The interested reader may find the details of GMRES(k) in [16] and of the other methods in [2, 8]

Consider a large, sparse, system of linear equations of the form

$$Mx = b \tag{2.1}$$

where M is a real matrix of order N, b is a given vector of length N and x is unknown vector to be computed.

Given an initial guess x_0, Krylov methods generate an approximate solution x_i from the translated Krylov space $x_0 + K_i$ where $r_0 = b - Mx_0$ and

$$K_i \subset span\{r_0, Mr_0, ..., M^{i-1}r_0\}.$$

x_i is usually chosen to minimize some norm of its residual $b - Mx_i$ [15].

The basic computational subtasks in Krylov methods are sparse matrix-vector multiplies with matrix M, additions of scalar multiples of vectors to other vectors (SAXPYs), and vector inner-products. The latter are used in determining the linear combination of Krylov vectors to add to the initial guess so as to minimize the norm of the residual.

Preconditioned Krylov methods consist of using an auxiliary matrix $Q = Q_l Q_r$ to first implicity generate the preconditioned system

$$(Q_l^{-1} M Q_r^{-1}) Q_r x = Q_l^{-1} b$$

. The matrix Q is chosen to be an approximation to M for which it is easy to compute $Q_l^{-1} v$ and $Q_r^{-1} v$ for a vector v.

Approximate LU factorization preconditioners [8, 6, 11] have been found to have very favorable convergence properties. Here we take Q to be LU where L is lower triangular and U is upper triangular. Typically, we form L and U by a process of incomplete factorization in which M is approximately factored in a way that allows only limited fill to occur.

The preconditioned matrix-vector multiply in the resulting Krylov method consists of doing a forward and backward sparse triangular solves using L and U as well as the sparse matrix multiplies by M. The cost of performing this incomplete factorization and the costs of solving the resulting triangular systems tends to be much smaller than the costs associated with an exact factorization because of the enforced sparsity of the matrices involved.

The computation in PCGPAK is carried out by (1) performing a symbolic incomplete factorization to determine the sparsity structure of L and U, (2) numeric calculation of the incomplete factorization using the previously calculated sparsity structures and (3) matrix vector multiplies, SAXPYs, vector inner products and sparse triangular solves.

2.1. The Test Problems

We now present our eight test problems used in our experiments in this paper.

Problem 1 (SPE1)	This problem models the pressure equation in a sequential black oil simulation. The grid is $10 \times 10 \times 10$ with one unknown per gridpoint for a total of 1000 unknowns.
Problem 2 (SPE2)	This problem arises from the thermal simulation of a steam injection process. The grid is $6 \times 6 \times 5$ with 6 unknowns per grid point giving 1080 unknowns. The matrix is a block seven point operator with 6×6 blocks.
Problem 3 (SPE3)	This problem comes from an IMPES simulation of a black oil model. The matrix is a seven point operator on a $35 \times 11 \times 13$ grid yielding 5005 equations.
Problem 4 (SPE4)	This problem also comes from an IMPES simulation of a black oil model. The matrix is a seven point operator on a $16 \times 23 \times 3$ grid giving 1104 equations.
Problem 5 (SPE5)	This problem arises from a fully-implicit, simultaneous solution simulation of a black oil model. It is a block seven point operator on a $16 \times 23 \times 3$ grid with 3×3 blocks yielding 3312 equations.

Problem 6 (5-Pt) This problem is a five point central difference discretization of the following equation on the unit square:

$$-\frac{\partial}{\partial x}(e^{-xy}\frac{\partial}{\partial x}u) - \frac{\partial}{\partial y}(e^{xy}\frac{\partial}{\partial y}u) + 2(x+y)(\frac{\partial}{\partial x}u + \frac{\partial}{\partial y}u) + (2 + \frac{1}{1+x+y})u = f$$

with Dirichlet boundary conditions and f chosen so that the exact solution is

$$u = x\,e^{xy}\sin(\pi x)\sin(\pi y).$$

The discretization grid is 63 × 63 giving 3969 unknowns. The L5-pt problem is the same problem with a 200 × 200 grid.

Problem 7 (9-pt) This problem is a nine point box scheme discretization for the following equation on the unit square:

$$-(\frac{\partial^2}{\partial x^2}u + \frac{\partial^2}{\partial y^2}u) + 2\frac{\partial}{\partial x}u + 2\frac{\partial}{\partial y}u = f$$

with Dirichlet boundary conditions and f chosen so that the exact solution is

$$u = x\,e^{xy}\sin(\pi x)\sin(\pi y).$$

The discretization grid is 63 × 63 giving 3969 equations. The L9-pt problem is the same problem with a 127 × 127 grid.

Problem 8 (7-pt) This problem is a seven point central difference discretization of the following equation on the unit cube:

$$-\frac{\partial}{\partial x}(e^{xy}\frac{\partial}{\partial x}u) - \frac{\partial}{\partial y}(e^{xy}\frac{\partial}{\partial y}u) - \frac{\partial}{\partial z}(e^{xy}\frac{\partial}{\partial z}u) + 80(x+y+z)\frac{\partial}{\partial x}u + (40 + \frac{1}{1+x+y+z})u = f$$

with Dirichlet boundary conditions and f chosen so that the exact solution is

$$u = (1-x)(1-y)(1-z)(1-e^{-x})(1-e^{-y})(1-e^{-z}).$$

The discretization grid is 20 × 20 × 20 yielding 8000 equations. The L7-pt problem is the same problem with a 30 × 30 × 30 grid.

3. Parallel Implementations of the Basic Krylov Method

3.1. SAXPY operations, Vector inner-products, and Sparse matrix-vector

The easily parallelizable subprocedures in the preconditioned Krylov methods implemented here are the SAXPY operations, the vector inner-products and the sparse matrix-vector products. For p processors and a linear system of order n, the indices from 1 to n are divided into p contiguous groups of roughly equal size. The i^{th} group is assigned to the i^{th} processor. and the obvious computations are carried out.

3.2. Parallel Triangular Solves and Sparse Numeric Factorizations

3.2.1. Triangular Solves

The triangular solves and the sparse numeric factorization can often be efficiently parallelized once the matrix dependent data dependencies are known. Consider the triangular solve in Figure 1. In this and the following figures we use a standard sparse matrix data structure where the non zeros in a matrix A are stored in a one dimensional array a. For each row i, ija(i) and ija(i+1)-1 represent the locations in array a of the left and rightmost non-zero columns of the row in matrix A. The column of A corresponding to element j of a is given by ija(j). Rather than sweeping over a two dimensional array, we sweep over a one dimensional array where dependences are given by the integer array ija.

The data dependencies between row substitutions indexed by the variable i in that Figure are determined by the values assigned during program execution to the data structure ija. A value of the outer loop index i, i_1 has a dependence on another value of the outer loop index i_2 if the computation of $y(i_1)$ requires $y(i_2)$.

To parallelize, we first partition the indices of the outer loop of Figure 1 into disjoint sets S_i, such that row substitutions in a set S_i may be carried out independently. To obtain the sets S_i, we perform a topological sort of the directed acyclic dependence graph G that describes the dependencies between the outer loop indices. Stage k of this sort is performed by placing into set S_k all indices of G not pointed to by graph edges. Following this all edges that emanated from the indices in S_k are removed. The elements of S_k are said to belong to *wavefront* k. Later, we will outline an efficient method that can be used to perform this topological sort.

We investigate two related methods of using wavefront information to parallelize the computations. In both cases, indices belonging to each wavefront are partitioned among the processors. In the first case, which we call *pre-scheduling*, global synchronizations separate consecutive wavefronts. In the other method which we call *self-executing*, a shared array is used to keep track of whether a solution variable has been calculated. Global synchronizations are replaced by *busy waits* that make sure that needed values have been produced before those values are used.

A pre-scheduled parallelized triangular solve is shown in Figure 2. The array *bywf* contains a reordering of the index set from S1 resulting from the sort. The array *wf* contains the wavefront number from the sort; a global synchronization is carried out between consecutive wavefronts.

Figure 3 depicts a self-executing triangular solve, with *bywf* and *wf* defined as before. Note that when processors attempt to simultaneously solve for independent work units, no interprocessor delays are generated. In both Figures 2 and 3, indices are assigned to processors by assembling the topologically sorted indices into *bywf*. Consecutive indices in *bywf* are assigned to consecutively numbered processors in a wrapped or striped fashion, i.e., index i is assigned to processor i modulo P where P represents the number of processors in the machine.

```
S1 do i=1,n
  y(i) = rhs(i)
        do j=ija(i), ija(i+1)-1
        y(i) = y(i) - a(j)*y(ija(j))
        end do
  end do
```

Figure 1: Triangular Solve

3.2.2. Sparse Factorizations

In a straightforward sequential version of Gaussian elimination without pivoting, consecutive *pivot* rows i are used to eliminate any non-zeros in column i of all rows $i + 1$ to N. All non-zeros to the left of row i's diagonal are eliminated before a i becomes a pivot row. When all non-zeros to the left of i's diagonal are eliminated, we say that row i has been *stabilized*.

The elimination or factorization process tends to introduce new non-zeros or *fill* into the factored matrix. An approximate factorization can be carried out by selectively suppressing the creation of many of the non-zeros created during the factorization process. The suppression is

```
do i=proc , n,numproc
row = bywf(i)
y(row) = rhs(row)
  do j=ija(row),ija(row+1)-1
        column = ija(j)
        while(ready(column).ne.iter) endwhile
        y(row) = y(row) - vals(j)*y(column)
 end do
 ready(row) = ready(row) + 1
 end do
```

Figure 2: Self-Executing Triangular Solve

```
do i=proc , n,numproc
row = bywf(i)
        while(wf(row).gt.newphase)
        newphase= newphase+1
        call global synchronization
        end while
y(row) = rhs(row)
        do j=ija(row), ija(row+1)-1
        column = ija(j)
        y(row) = y(row) - vals(j)*y(column)

        end do
end do
```

Figure 3: Pre-Scheduled Triangular Solve

performed on the basis of determining how *indirect* the fill was. For instance, all fill created by eliminations using the first matrix row as a pivot row arise directly from non-zeros present in the original matrix. However, when row 2 is stabilized, non zeros in row 3 may arise directly from a non-zero present in the original matrix *or* may arise as a result from fill from row 2. There are a variety of methods used to quantify the indirectness of fill; only fill that is sufficiently direct is retained and is capable of generating further fill. The specifics of the algorithm used here to determine which elements are to be retained is described in [8, 11].

During the course of the computation, each row i undergoes a number of transformations as non-zero elements in consecutive columns $j < i$ are eliminated by stabilized pivot rows j. When all the non zeros in columns $j < i$ have been eliminated, row i itself is stabilized and may be used as a pivot row in other eliminations.

The incomplete factorization procedure consists of a symbolic and a numeric factorization. The symbolic factorization calculates the non-zero structure of the factored matrix, and the numeric factorization computes the numeric values for the incompletely factored matrix.

The numeric factorization is parallelized in a way that is analogous to the triangular solve. Elimination in each row i requires the use of a sequence of stabilized pivot rows identified as before by the sparse data structure ija. (Figure 4). In parallelizing the numeric factorization, a topological sort of the dependencies pertaining to the outer loop indices is performed. As was shown explicitly for the triangular solve, prescheduled and self-executing versions of the numeric factorization algorithm can be formulated.

```
S1 do i=1,n
      do j=ija(i),ija(i+1)-1
      Use pivot row ija(j) to perform elimination on row i
      end do
   end do
```

Figure 4: Schematic Sparse Factorization

3.3. Sparse Symbolic Factorizations

Because the pattern of fill is not known a priori, the data dependencies in symbolic factorization cannot be analyzed before the algorithm executes. In our implementation of the algorithm, we distribute the rows of the matrix over processors in a wrapped manner and execute in a self-scheduled fashion.

Since we are computing the incomplete factorizations of sparse matrices, the fill pattern will be sparse. The columns of row i that are filled in at any given stage of the algorithm are kept sorted in increasing order in a linked list. Operations on row i with pivot row j require that the list of non-zeros pertaining to row i be merged with the list of non-zeros pertaining to pivot row j. Note that because this is an incomplete factorization, some of the non-zero elements in the newly created merged list may be omitted.

3.4. Efficient Calculation of the Topological Sort

It is possible to perform a particularly efficient topological sort of index sets arising from some sequential programs such as the triangular solve and the sparse numeric factorization. For instance, in the sparse triangular solve, computation that pertains to row i can only require information produced by rows having index values less than i. Since the wavefront for each row is one plus the maximum of the wavefronts of the rows on which it depends, one can simply sweep sequentially through the rows and calculate the wavefront for each row. This process produces an array wf defined in Section 3.2. The array wf must then be sorted to produce an execution schedule for the processors.

On the Multimax/320, the sequential execution time required for both these operations tends to be slightly less than the cost of a single triangular solve using the same matrix. The topological sort can be parallelized to a degree by striping consecutive indices across the processors and by using busy waits to assure that variable values have been produced before being used.

3.5. A Robust Transformation

In the linear solver, we program making the implicit assumption that the sparse data structures will be initialized in particular ways. In Figure 1, we make the tacit assumption that sparse data structures a and ija must represent a lower triangular matrix. If ija were any integer array, when the computation of y(i) requires y(j), j could be greater than or less than i. When $j > i$, the value of y(j) that is needed corresponds to the one existing before the loop commences execution. When one parallelizes these loops, it is vital that the old value of y(j) not be overwritten before it is needed. To avoid this overwritting, we use separate arrays to store the old and new values of y.

This type of data dependence is called an *anti-dependence.* [13].

Automated transformations must deal with issues such as this one in order to produce correct code. Issues related to the production of these transformations are addressed in [17]. The self-executing program analogous to Figure 3, designed to handle this kind of situation is shown in Figure 5.

```
do i=proc , n,numproc
row = bywf(i)
ynew(row) = rhs(row)
    do j=ija(row), ija(row+1)-1
    column = ija(j)
        if(column .gt. row) then
        ynew(row) = ynew(row) - vals(j)*yold(column)
        else
        while(ready(column).ne.iter) endwhile
        ynew(row) = ynew(row) - vals(j)*ynew(column)
        endif
    end do
ready(row) = ready(row) + 1
end do
```

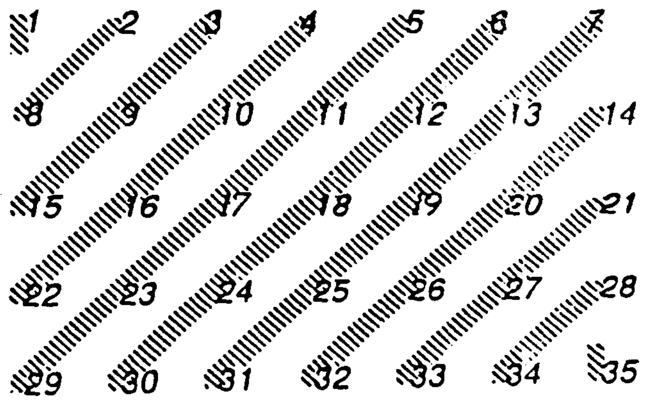
Figure 5: Generalized Triangular Solve

Programs which reordered index sets of loops can also be produced from source codes by symbolic transformations. These reordered index sets are executed by transformed source code loop structures. In [18] we describe in some detail the properties of loops amenable to various kinds of runtime scheduling and the transformations required to parallelize those loops.

4. Analysis of a Model Problem

We will use a model problem to illustrate the difference in performance of pre-scheduling and self-execution. We will examine this by estimating the time that would be required to solve 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 use a m by n domain and $p \leq \min(m, n)$ processors. We will explicitly take into account only floating point and synchronization related computations. In Section 5.2 we demonstrate experimentally that these assumptions can be used to predict multiprocessor timings rather accurately.

We assume that all computations required to solve the problem would require time S on a single processor, and that computation of each point takes time $T_{point} = S/(mn)$; This ignores the relatively minor disparities caused by the matrix rows represented by points on the lower and the left boundary of the domain.



**Sorted List = (1,2,8,3,9,15,4,10,16,22,5,11,17,23,29,
6,12,18,24,30,7,13,19,25,31,14,20,26,32
21,27,33,28,34,35)**

Figure 6: Assignment of Indices to Wavefronts

To understand the relative performance of the two synchronization mechanisms on this problem, we need to make clear how the indices are mapped onto the machine's processors. The global topological sort produces a list of indices sorted by wavefront. The points in a wavefront arise from an antidiagonal strip of the domain. For instance, in Figure 6, we depict a five by seven domain with the points in each wavefront linked by an antidiagonal stripe. When the points in the domain are naturally ordered the topological sort produces a list L that picks points on the anti-diagonal strip going from the upper right point in the strip to the lower left point. This corresponds to arranging the points in each wavefront in order of increasing index number.

The indices in L are assigned in a wrapped manner, as depicted for using 3 processors in the example problem in Figure 7. When prescheduling is used, the computation is divided into phases separated by global synchronizations.

A brief inspection of Figure 6 makes it clear that $n + m - 1$ phases are required to complete the computation. Define $MC(j)$ as the maximum number of points computed by any processor during phase j. The computation time required to complete phase j is equal to $T_{point}MC(j)$. The

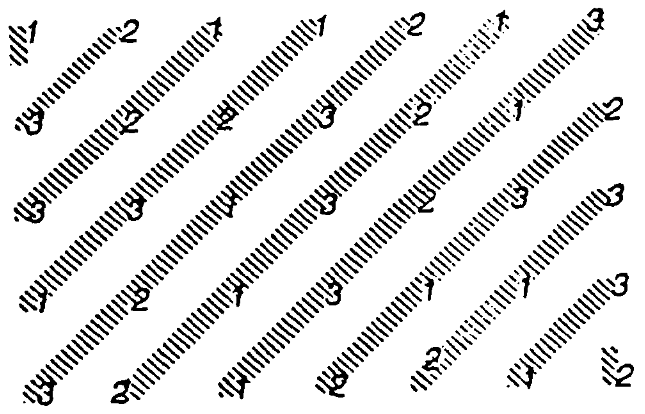


Figure 7: Assignment of Indices to Processors

computation time required to complete the problem is consequently

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

We now proceed to calculate $MC(j)$. During phase j, a total of j points must be computed when $1 \leq j < \min(m, n)$. Since the strips 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 $\min(m, n)$ points must be completed during phase j. Due to the wrapped assignment of strips 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 $n + m - j$ points 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

$$T_C = T_{point} \sum_{j=1}^{n+m-1} MC(j) =$$

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

$$\sum_{j=m+n-\min(m,n)+1}^{n+m-1} \lceil \frac{n+m-j}{p} \rceil \Big)$$

By assumption, the sequential time to solve the problem is $S = mnT_{point}$. The estimated efficiency E_{opt} we could achieve in the absence of any source of inefficiency unrelated to load imbalance would be $\frac{S}{T_C}$ or

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

$$\sum_{j=m+n-\min(m,n)+1}^{n+m-1} \lceil \frac{n+m-j}{p} \rceil \Big)^{-1}$$

We can derive a simpler expression that approximates E_{opt} by estimating the total amount of time all processors spend idle due to load imbalance. Let $\hat{m}$ and $\hat{n}$ be equal to the largest multiples of p that are smaller than m and n respectively. During any phase $j \leq \min(m,n) - 1$ when j is not a multiple of p, there are $p - j \ mod \ p$ processors idle. When j is a multiple of p, no processors are idle. Thus the cumulative processor idle time for $j \leq \min(\hat{m}, \hat{n}) - 1$ is:

$$L_{tr} = \frac{T_{point} \min(\hat{m},\hat{n}) \sum_{l=1}^{p}(l-1)}{p} = \frac{T_{point} \min(\hat{m},\hat{n})(p-1)}{2} \quad .$$

Through similar reasoning, the sum of the processor idle time for the last $\min(\hat{m},\hat{n}) - 1$ phases is the same.

Between the first and last $\min(m,n) - 1$ phases if $\min(m,n)$ is equal to p, no time is wasted, otherwise the time lost per phase is

$$L_{ss} = T_{point}(p - \min(m,n) mod \, p)$$

We can use the above considerations to estimate the cumulative time wasted by all processors, and use this estimate to calculate the following approximate expression which gives $E_{opt} =$

$$\frac{mn}{mn + \min(\hat{m},\hat{n})(p-1) \quad + \quad (m+n+1-2\min(\hat{m},\hat{n}))((p-\min(m,n)) \bmod p)}$$

Much of the load imbalance we observe above can be corrected. The failure to balance is essentially an end effect; e.g., the phase has $p+1$ work units with equal computational demands, but only p processors are available. In [5] we rearranged the global synchronizations in a way that obtained a tradeoff between improved load balance and the costs of the global synchronizations. While that mechanism was shown to be advantageous for some problems, rearrangement of the global synchronizations does require an extra stage of preprocessing.

Self-execution also eliminates these end effects. In the model problem we are presenting here, we can see that any given row substitution in a wavefront requires only *two* solution values from the previous wavefront. It is possible to concurrently compute row substitutions in consecutive wavefronts provided that we observe dependences. This is taken care of naturally since the self-execution *busy wait* synchronization mechanism ensures that dependences are in fact observed.

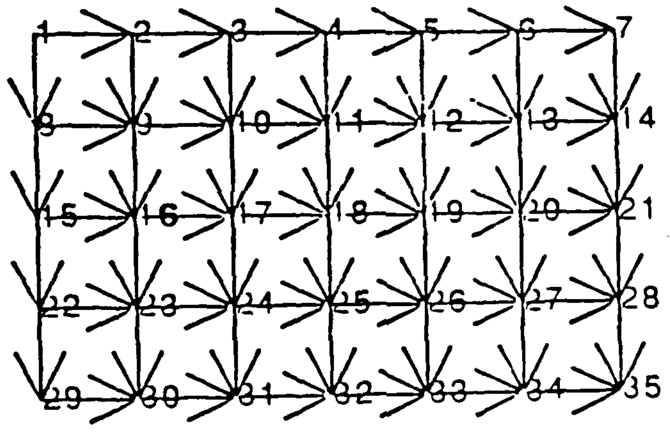


Figure 8: Data Dependencies Between Indices

Figure 8 depicts the data dependences between row substitutions in the model problem. Assume that solution values are available for indices in list L through the index corresponding to wavefront w, domain strip s. All indices in L up to the index corresponding to wavefront $w+1$, domain strip s will have their dependences satisfied and can be concurrently calculated.

We can derive an expression for E_{opt} for the self-executing case, here again we assume that we employ $p \leq \min(m,n)$ processors to solve the model problem. Assuming again that the time required to compute the solutions is identical for all indices, only the first and last $p-1$ wavefronts contribute to load imbalance. By arguments similar to those made for the pre-scheduling case, the cumulative processor idle time is $p(p-1)$. E_{opt} is thus given by

$$\frac{mn}{mn+p(p-1)}$$

If T_{synch} is the cost of a single global synchronization, the time required to synchronize the pre-scheduled computation is T_{synch} times the number of synchronization needed, i.e. $T_{synch}(n+m-1)$. The self-executing program ensures global synchronization by incrementing elements of a shared array when variables are calculated. As described earlier, the shared array is checked to find out which variables have been solved for at any given time. The cost of incrementing the array elements is given by $T_{inc}mn$, where T_{inc} is the cost of incrementing a single array element. Since computing each solution value is assumed to need two other solution values, the cost of checking the array elements is estimated by $2T_{check}mn$, where T_{check} is the cost of checking a shared memory location. Note that we have accounted separately for idle time due to load imbalance; we assume here that we only have to verify that a required solution value is available.

By modifying the above expressions for E_{opt} to include the synchronization overheads, we derive an expression for the ratio between the time required to solve the model problem using pre-scheduling to that required for solving the problem using self-execution, R_{p-s}. In the following expression, $R_{synch} = \frac{T_{synch}}{T_{point}}$, $R_{inc} = \frac{T_{inc}}{T_{point}}$ and $R_{check} = \frac{T_{check}}{T_{point}}$.

$$R_{p-s} = [mn + (L_{ss} + 2L_{tr})/T_p + R_{synch}(n+m-1)][mn(1+R_{inc}+2R_{check}) + p(p-1)]^{-1}$$

For large n and $m = p+1$, we expect to find that slightly under half of the processors are idle due to load imbalance. The above ratio in the limit of large m becomes

$$\frac{2p + R_{synch}}{(p+1)(1+R_{inc}+2R_{check})}.$$

The above expression suggests that the self-executing program might be expected to perform substantially better than the pre-scheduled program as long as it is relatively inexpensive to check and to increment shared memory. In practice, one often obtains triangular systems that have a relatively large number of phases with modest amounts of work to be performed in each phase, as we will see in Section 5.2. The limit derived above sheds some insight into these cases.

For $m = n$ the situation is quite different; as n increases we obtain the ratio

$$\frac{1}{1+R_{inc}+2R_{check}}.$$

If the problem size increases in both dimensions, the relative contribution of the end effect load imbalances diminish. The amount of computation to be performed grows as mn while the number of global synchronizations needed grow as $n + m - 1$. In this case, pre-scheduling is preferable to self-execution. In shared memory machines with fast access to shared memory, there will be only a small difference between the pre-scheduled and self-executing times.

Many problems of practical interest are somewhat less sparse than the model problem analyzed here. When such a problem is to be solved using many processors, we may expect dramatic performance differences between pre-scheduled and self-executing programs. To illustrate this, we present the rather extreme (from our point of view) example of solving a n by n dense triangular matrix having unit diagonals using $n-1$ processors. Assume T_{saxpy} is the time required for a floating point multiply and add. The computation time required to solve this system using self-execution is $T_{saxpy}(n-1)$. No parallelism at all is obtained when one attempts to solve such a system when row substitutions are separated by global synchronizations; each row substitution forms its own wavefront. The sequential computation time and the pre-scheduled computation time are both $T_{saxpy}\frac{n(n-1)}{2}$. Calculated only on the basis of load balance, the self-executing efficiency E_{opt} is $\frac{n}{2(n-1)}$ while the pre-scheduled E_{opt} is $\frac{1}{n-1}$.

5. Experimental Results

5.1. Sequential Timings

In this section, we present benchmarks for solutions of the test problems discussed in Section 2.1. There are a number of parameters that can be specified when a Krylov space solver such as PCGPAK is employed. Two principal parameters are the degree of fill used in the incomplete factorization and whether the original matrix is used in the iterations or whether an equivalent smaller, denser system, i.e., a *reduced system* is employed.

We present results for the parameters that have been found to be best for solving systems with sequential PCGPAK in Table 1. The results presented in this paper are for the Encore Multimax/320 with 13 megahertz APC/02 boards and version 2.1 of the FORTRAN compiler. This table is is divided into five broad categories, showing the method and preconditioner used, the iteration count needed to reach the stopping criteria; the percentage of CPU time spent in the reduced system computation when that methodology was used; the percentage of time spent forming the preconditioner; and the percentage of time spent in the various subtasks of the basic preconditioned algorithms. The column labeled "SAXPY and SDOT" corresponds to item 2 of the basic tasks. These computations involve adding a scalar multiple of one vector to another vector (SAXPY operation) and computing vector inner-products (SDOT operation). The stopping criteria used was to reduce the Euclidean norm of the residual by six orders of magnitude, starting with an initial guess of 0.

Test Problem	GM RES (K)	Precond-itioner	Number Iter-ations	Reduced System	Factor-ization	MVP	Solves	SAXPY and SDOT	Total Time (seconds)
SPE1	1	ILU(0)	26	17	6	26	33	17	3.6
SPE2	1	ILU(0)	11	-	23	31	37	9	10.1
SPE3	10	MILU(0)	24	15	11	25	28	20	24.1
SPE4	10	ILU(0)	17	21	12	22	26	19	3.1
SPE5	20	ILU(0)	43	-	2	17	22	59	64.0
5-PT	5	MILU(0)	20	17	6	23	30	23	17.6
9-PT	10	ILU(2)	20	-	20	18	38	24	47.9
7-PT	1	ILU(0)	20	-	6	28	38	27	46.4
L5-PT	4	MILU(0)	54	8	3	27	34	28	410.7
L9-PT	10	ILU(1)	80	-	5	22	37	36	603.3
L7-PT	1	ILU(0)	31	-	4	29	39	28	239.9

Table 1: Breakdown of Times for single processor PCGPAK on the Encore Multimax/320 with APC-02s

These results show that the sparse matrix-vector products occupy roughly 20% to 30% of the CPU time, the sparse triangular solves occupy roughly 20% to 40% of the time, and the SAXPY and SDOT operations take from 25% to 40% (with the exceptions of SPE2 (10%) and SPE5 (59%)). When reduced system preprocessing is used, it takes from 7% to 18% of the CPU time. The formation of the preconditioner takes from 1% to 12% of the time (except for SPE2 (23%)). Thus, matrix-vector products, sparse triangular solves, and SAXPY and SDOT operations dominate the total CPU time.

All these experiments used the "right-oriented" preconditioning, *i.e.* the preconditioned problem was

$$MQ^{-1}\hat{x} = b, \qquad x = Q^{-1}\hat{x}$$

where M is the coefficient matrix and Q is the preconditioner. The initial guess was $x_0 \equiv 0$.

5.2. Multiprocessor Timings

Two versions of parallel PCGPAK were produced. In the first version, the triangular solves and the numeric factorization were implemented using self-scheduling; while in the second version the triangular solves and numeric factorization were pre-scheduled. In both cases, the index set of the outer loop of the appropriate procedure was partitioned as described in Section 3.2.

Test Problem	Prescheduled		Self-executing		Sort Time
	Time	Efficiency	Time	Efficiency	Time
SPE1	1.48	14	0.83	25	0.03
SPE2	2.49	24	1.63	37	0.26
SPE3	3.84	35	3.11	44	0.11
SPE4	1.04	17	0.66	26	0.03
SPE5	6.18	62	5.89	65	0.10
5-PT	3.11	33	2.50	41	0.14
9-PT	6.31	42	4.76	56	0.25
7-PT	4.90	57	5.41	52	0.19
L5-PT	41.76	50	37.93	56	1.40
L9-PT	64.01	54	54.74	63	0.80
L7-PT	23.20	62	23.51	61	0.79

Table 2: Times and Efficiencies for PCGPAK 16 Processors of the Encore Multimax

In Table 2 we present time required to solve the test problems for the pre-scheduled and self-executing versions of PCGPAK, along with the with the parallel efficiencies achieved. Parallel efficiency is defined as the ratio between the time required to solve a problem by an optimized sequential version of PCGPAK and the product of the time required on the same problem by the multiprocessor code multiplied by the number of processors. The self-executing version of the program yields the highest efficiencies and the lowest times for all test problems except the small and large problems using the seven point operator (7-PT and L7-PT). For many of the problems, the timing differences in favor of the self-executing version of the code are quite substantial. In the SPE problems 1, 2 and 4 the self-executing version PCGPAK completes in less than 70 percent of the time required by the prescheduled version.

Overheads in the self-executing version of the program arise from the need to check and

Test	Time	Prescheduled			Self-executing		
Problems	Ratio	Solves	Iterative	Numeric	Solves	Iterative	Numeric
SPE1	1.78	0.56	1.02	1.27	0.22	0.47	0.52
SPE4	1.58	0.32	0.53	0.60	0.14	0.34	0.40
SPE2	1.52	0.60	1.02	1.27	0.40	0.83	1.12
9PT	1.33	3.07	4.66	5.10	1.66	3.15	3.58
5PT	1.24	1.17	1.86	2.08	0.60	1.26	1.48
SPE3	1.23	1.48	2.39	2.73	0.78	1.67	2.01
L9PT	1.17	35.20	58.19	59.91	25.93	48.87	50.20
L5PT	1.1	16.90	30.11	32.05	13.00	26.31	28.24
SPE5	1.05	2.20	5.72	5.86	1.66	5.16	5.31
L7PT	0.99	11.02	20.55	21.57	11.09	20.94	21.93
7-PT	0.90	2.24	4.16	4.46	2.20	4.24	4.53

Table 3: Self Execution vs Prescheduling for PCGPAK 16 Processors of the Encore Multimax

update the shared array which indicates whether needed solution variables or pivot rows have been computed. In the prescheduled version of the program, overheads arise from the cost of global synchronizations. Overheads aside, the parallelism available from the self-executing version of the program is always better than in the pre-scheduled version.

In Section 6, we will explain the differing relative performance between the prescheduled and self-executing versions of PCGPAK. This will be done by showing that for the test problems, we can account in a quantitative manner for the timing differences between pre-scheduled and self-executing versions of the triangular solves. We also present in Table 2, the times required to perform the topological sort for each of the test problems, which turn out to be extremely small, compared to the total execution time.

In Table 3, we depict the times required for different portions of the linear solver for both prescheduled and self-executing programs. We present the times required for performing the triangular solves, the *iterative portion* of the calculations and all non-symbolic portions of the calculations. We also calculated the ratio of time for the self-executing calculation to the time required to perform the prescheduled computation. That ratio is presented in Table 3 and the test problems are listed in descending order of that ratio. In Table 4 we present a detailed breakdown of efficiencies for the self executing version of the program.

The iterative and numeric efficiencies are of interest since one expects to see performance close to the iterative efficiencies in a slowly converging problem with structure similar to the various test problems. One would expect to see performance close to the numeric efficiencies in systems arising from a non-linear or time dependent problem where the systems to be solved during any given step of the algorithm do not change in symbolic structure.

Test Problem	Triangular Solves	Iterative Efficiency	Numeric Efficiency	Symbolic Efficiency	Total Efficiency
SPE1	31	36	37	09	25
SPE2	44	53	50	23	37
SPE3	47	63	61	12	44
SPE4	30	37	37	10	26
SPE5	47	72	71	30	65
5-PT	46	62	62	12	41
9-PT	55	69	68	25	56
7-PT	43	60	60	31	52
L5-PT	52	71	71	12	56
L9-PT	47	68	67	16	63
L7-PT	45	65	64	32	61

Table 4: Self-Executing Efficiencies on the Multimax/320

The efficiencies for the triangular solves were in the neighborhood of 50 percent for all but SPE1, SPE2 and SPE4. For all problems besides for those three, the efficiencies for the iterative and numeric portions of the solver were at least 60 percent. The problems SPE1, SPE2 and SPE4 are quite small. From Table 1 we see that the sequential execution times required for these problems are markedly smaller than for any of the other problems tested.

5.3. Vector Processor Comparisons

Timings were performed on a single head of a Cray/XMP using a version of PCGPAK written in FORTRAN77 [14] compiled with version 1.3 of the CFT77 compiler, no compiler directives were used. In Table 5 we present the timings for various portions of PCGPAK on the Cray X/MP. We

Test Problem	Solves		Iterative		Numeric		Total	
	Time	Ratio	Time	Ratio	Time	Ratio	Time	Ratio
SPE1	0.063	3.5	0.098	4.8	0.114	4.6	0.127	6.4
SPE2	0.085	4.7	0.138	6.0	0.186	6.0	0.195	8.2
SPE3	0.252	3.1	0.390	4.3	0.516	3.9	0.600	4.7
SPE4	0.034	4.1	0.055	6.2	0.073	5.5	0.088	7.3
SPE5	0.600	2.8	0.940	5.5	0.976	5.4	0.984	5.9
5-PT	0.302	2.0	0.467	2.7	0.557	2.7	0.622	4.0
9-PT	0.603	2.8	0.935	3.4	1.067	3.4	1.365	3.5
7-PT	1.191	1.8	1.823	2.3	1.895	2.4	1.918	2.8
L5-PT	6.897	1.9	10.405	2.5	11.248	2.5	11.856	3.2
L9-PT	9.011	2.9	13.608	3.6	13.954	3.6	14.749	3.5
L7-PT	5.968	1.9	9.037	2.3	9.280	2.4	9.358	2.4

Table 5: Cray X/MP Times and Ratios of Multimax/320 to X/MP Times

also present ratios of the Multimax time using the self-executing multiprocessor code to the time on the Cray.

The timings and ratios are presented for the triangular solves, the iterative portion of the program, for the numeric portion of the program and for the entire program. As expected, the Multimax/320 program performed comparatively well on the triangular solves, with the Cray ranging from 1.8 to 4.7 times as fast as the Multimax on this portion of the code. While sparse triangular solves do not vectorize well, they parallelized relatively efficiently (Table 4). Matrix vector multiplies, SAXPYs and inner products vectorize and parallelize extrememly well. Since the peak computation rate of a single Cray X/MP processor is substantially greater than the combined computational capabilities of the 16 Multimax/320 processors, the ratios between the Multimax and Cray times are higher for the iterative times than for the triangular solves. The ratios of the numeric times were comparable to the ratios for the iterative times. The ratio of the total times were in some cases markedly higher than any of the other ratios. The symbolic computation parallelized quite poorly (Table 4). While the symbolic computations also vectorize poorly, the high scalar speed of the Cray X/MP was apparently decisive here.

The Cray times obtained using compiler directives were smaller than those obtained without the use of directives by a factor of two to three. It seems evident that a shared memory machine built from relatively simple processors is able to achieve performance that is within a reasonable factor of a supercomputer.

6. A Detailed Performance Analysis

The multiprocessor results presented in Section 3 indicate that self-executing programs appear to run more efficiently on most of the test problems we have examined. In this section we will examine in detail the sources of inefficiency occurring in both prescheduled and the self-executing programs. This analysis will shed insight into the performance tradeoffs encountered in self-executing and prescheduled methods. This information will also be used to make projections concerning the efficiency that could be obtained by the use of these methods on a larger shared memory multiprocessor.

6.1. Where Does the Time Go

We performed an operation-count based analysis of the parallelism that could be obtained given a particular assignment of indices to processors. The analysis made the assumption that the load balance could be characterized solely by the distribution and scheduling of the floating point operations. The efficiency estimated on this basis will be called the *symbolically estimated efficiency*. In Tables 6 and 7 respectively, are depicted *symbolically estimated efficiencies* for self-executing and prescheduled triangular solves. The estimates presented are for some of the previously discussed test problems on 16 processors. The parallelism we anticipate obtaining through the use of self executing code is better, frequently by a wide margin.

The efficiencies predicted by operation count based analysis are substantially higher than those we saw in Section 3. This is not surprising since the symbolically estimated efficiencies do not take into account a number of important sources of overhead. We will demonstrate that we can account for these overhead sources in a systematic way and use these overhead values to accurately predict the multiprocessor timings in both self-executing and pre-scheduled versions of a standalone program for paralleling a sparse lower triangular solve.

In Table 6 and 7 we have the actual multiprocessor timings on 16 processors for lower triangular solves arising from the incompletely factored test problem matrices. An optimized sequential version of the program was also timed for each of the lower triangular systems. We depict sequential times divided by the product of the number of processors used and the symbolically estimated efficiencies (timings are denoted by *Ideal SU: best seq.* in Tables 6 and 7).

The estimates of multiprocessor times obtained in the estimate above are quite optimistic.

To take into account the extra operations that had to be executed by the parallel version of the program, we timed the multiprocessor program on a single processor. Tables 6 and 7 show the single processor parallel code timing divided by the product of the number of processors used and the symbolically estimated efficiencies (*Ideal SU:1 PE Par.*). In performing this calculation, we tacitly assume that load balance effects of the distribution of work in the multiprocessor program can still be estimated by taking into account only the distribution of floating point calculations. In effect, we are assuming that the effect of the extra operations required in the multiprocessor program could be explained by simply adding a fixed overhead to each floating point operation.

Contention for resources such as shared memory and bus access can cause inefficiencies that are not accounted for by the above estimates. We ran a version of the multiprocessor code designed to simulate the memory and communications access patterns of the actual program. This version of the code is designed to have a perfect load balance. When executed on P processors this program, executes the schedules a total of P times. Each processor ends up executing the schedules assigned to all processors so that each processor ends up computing the work associated with all of the indices in the problem. The time required for this program to complete is called the *rotating processor* time because each processor takes on the work assigned to each other processor with control being shifted in a rotating fashion.

No synchronization takes place in this version of the code. The shared array reads and writes used in the busy wait coordination in the self executing code still take place but the program is modified so that no waiting actually has to occur. In the prescheduled version of the program, global synchronizations are not employed. In the absence of resource contention, we would expect that the time required for the above computation would be very close to the time spent running the parallel version of the codes on a single processor.

In the self-executing case, the time estimate obtained from dividing the rotating processor time by the product of the number of processors and the symbolically estimated efficiency gives a very close estimate of the actually observed multiprocessor time (*Rotating Estimate*). For the prescheduled case, we must include the time required for the global synchronizations to obtain an accurate prediction of the actual multiprocessor time (*Rotating Estimate + Barrier*). When this is done, we get a very good estimate of the pre-scheduled multiprocessor timings. In using the symbolically estimated efficiencies, we again make the tacit assumption that the extra overhead (except the global synchronizations) could be explained by adding a fixed overhead to all floating point operations. Note that while more sophisticated models of overhead are certainly possible and may be desirable in some cases; we find here that these simple techniques and assumptions adequately explain the timings we observe.

The sources of the timing differences between prescheduled and self-executing programs becomes more apparent in comparing Tables 6 and 7. For the 5-PT and SPE2 test problems, the difference in the load balance obtainable through the use of prescheduled and self executing codes is large enough that the *1 PE Seq* time for the prescheduled code is greater than the *Parallel Time* for the self executing program. Even if we had a hypothetical prescheduled code with no overheads except for load imbalance, that code would still be less efficient than the self executing program.

Recall that the prescheduled program uses global synchronizations in between each phase but does not need to write into a shared array to keep track of which variables have been calculated. In a reasonably large problem such as 7-PT where there are relatively few global synchronizations, the overhead required for prescheduling is relatively small. Since little loss due to load imbalance is seen for 7-PT, we are able to see that prescheduling gives a slightly better timing.

Test Problem	Phases	Symbolic Efficiency	Parallel Time	Rotating Estimate	Ideal SU: 1 PE Parallel	Ideal SU: best Seq.	Sort Sort Time
SPE2	60	0.89	20.7	20.0	17.9	15.0	41.9
SPE5	66	0.96	23.4	21.6	18.5	15.3	94.6
5-PT	124	0.95	18.7	17.6	14.5	12.2	71.1
9-PT	311	0.97	57.9	57.1	51.7	43.2	153.1
7-PT	58	0.98	56.3	57.6	45.1	38.1	237.1

Table 6: Parallel Time and Estimates for Self Executing Triangular Solves

6.2. Timing Projections

Since we can accurately account for the execution time in the Encore Multimax, it is reasonable to make some timing projections. These projections make the assumption that the costs of synchronization, the costs from the extra operations required to run the parallel versions of the codes and the costs due to contention do not change with the number of processors. If the load

Test Problem	Phases	Symbolic Efficiency	Parallel Time	Rotating Estimate + Barrier	Rotating Estimate	Ideal SU: 1 PE Parallel	Ideal SU: best Seq.
SPE2	60	0.52	32.7	32.8	30.0	26.6	25.6
SPE5	66	0.70	29.0	29.5	26.4	22.6	20.8
5-PT	124	0.61	31.1	31.0	25.2	20.2	18.8
9-PT	311	0.78	80.3	83.9	63.5	56.7	53.9
7-PT	58	0.94	56.2	56.3	53.7	44.0	39.8

Table 7: Parallel Time and Estimates for Prescheduled Triangular Solves

balance were perfect, the *Best* efficiencies in Table 8 would be obtained. (Note that in Table 8, S.E. is used to indicate self executing synchronization and P.S. is used to represent prescheduled synchronization).

The estimate of non-load balance related loss (*Best* in Table 8) obtained from timings on 16 processors is clearly not valid for larger machines if we simply add more processors to the current machine. The estimate is reasonable if we assume that the capabilities of the shared resources such as interprocessor communication are engineered to scale with the size of the machine.

It is clearly easier to assure performance characteristics that scale with the number of processors if one designs machines with distributed memory or a hierarchical shared memory. We are currently extending projections such as these to those types of machines, that work is beyond the scope of this paper but some discussion of that issue can be found in [12].

In Table 8, we present efficiencies for 16 processors and projected efficiencies for 32 and 64 processors. The projected performance of the prescheduled programs deteriorates much more rapidly as one increases the number of processors. This difference is driven by the increasing disparity between symbolically estimated efficiencies in the two scheduling methods. The differences seen in the *Best* efficiencies in Table 8 reflect the varying relative costs of global synchronizations and array writes in problems with different structures, this issue was discussed in Section 6.2.

Test	Best		16 Processors		32 Processors		64 Processors	
Problem	S.E.	P.S.	S.E.	P.S.	S.E.	P.S.	S.E.	P.S.
SPE2	75	78	67	40	58	25	45	12
SPE5	65	71	62	49	56	39	46	23
5-PT	65	61	52	27	55	30	34	15
9-PT	76	64	73	52	68	26	39	12
7-PT	66	70	65	66	64	62	60	55

Table 8: Estimated Efficiencies for Larger Machines

7. Conclusion

We have performed a detailed performance study of a powerful commercially available linear system solver. Two mechanisms were presented for parallelizing the critical inner loops of the computation, the method of *pre-scheduling* and of *self-executing.* Both mechanisms parallelization require reordering of the the index set of the loop. The methods differ in how to carry out the computation that will be performed using the reordered index set. The sources of time losses on the multiprocessor were examined in considerable detail so that the effects of the different parallelization mechanisms could be clearly understood.

The self-executing mechanism allows for more parallelism than the pre-scheduling method. Overhead in the self-executing mechanism arises from the busy wait synchronizations used, in the pre-scheduling mechanism overhead comes from global synchronizations. Both methods required extra array references to access the reordered index sets. On balance, the self executing method exhibited superior performance on most test problems. On problems with few global synchronizations and ample potential concurrency, the performance of the prescheduled method was comparable or slightly superior to the self-executing method.

The performance we obtain was brought into perspective by comparisons with timing results from a Cray X/MP supercomputer. Performance on an Encore Multimax/320 with relatively modest computational capabilities came within a small factor of the performance of a comparable code run on a Cray X/MP.

8. Acknowledgements

Thanks to Ravi Mirchandaney for careful readings of the manuscript, to Peter Farkas for running the Cray timings.

References

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

[2] Rati Chandra, *Conjugate Gradient Methods for Partial Differential Equations*, Ph.D. Thesis, Department of Computer Science, Yale University, 1978. Also available as Technical Report 129.

[3] R. Cytron, Doacross: Beyond Vectorization for Multiprocessors, *Proceedings of the1986 International Conference on Parallel Processing*, 1986, pp. 836–844.

[4] D. Baxter and J. Saltz and M. Schultz and S. Eisentstat and K. Crowley, An Experimental Study of Methods for Parallel Preconditioned Krylov Methods, *Proceedings of the 1988 Hypercube Multiprocessor Conference, Pasadena CA,* January 1988.

[5] D. Nicol and J. Saltz, *Optimal Pre-Scheduling of Problem Remappings*, Technical Report 87-52, ICASE, 1987.

[6] Todd Dupont, Richard P. Kendall and H. H. Rachford Jr., *An approximate factorization procedure for solving self-adjoint elliptic difference equations*, SIAM Journal on Numerical Analysis, 5(1968), pp. 559–573.

[7] E. Anderson, *Solving Sparse Triangular Linear Systems on Parallel Computers*, Report 794, UIUC, June 1988.

[8] Howard C. Elman, *Iterative Methods for Large, Sparse, Nonsymmetric Systems of Linear Equations*, Ph.D. Thesis, Department of Computer Science, Yale University, 1982. Also available as Technical Report 229.

[9] J. Saltz, Methods for Automated Problem Mapping, *The IMA Volumes in Mathematics and its Applications. Volume 13: Numerical Algorithms for Modern Parallel Computer Architectures Martin Schultz Editor*, Springer-Verlag, 1988.

[10] ———, *Aggregation Methods for Solving Sparse Triangular Systems on Multiprocessors*, SIAM J. Sci. and Stat. Computation., to appear (1989).

[11] J. A. Meijerink and H. A. van der Vorst, *Guidelines for the usage of incomplete decompositions in solving sets of linear equations as occur in practical problems*, Journal of Computational Physics, 44(1981), pp. 134–155.

[12] Mirchandaney, Ravi and Saltz, Joel and Smith, Roger and Nicol, Dave and Crowley, Kay, Principles of Runtime Support for Parallel Processors, *1988 International Conference on Supercomputing, St. Malo, France,* July 1988, pp. 140–152.

[13] D. A. Padua and M. J. Wolfe, *Advanced Compiler Optimizations for Supercomputers*, CACM, 29(1986), pp. 1184–1201.

[14] Scientific Computing Associates, *PCGPAK: Benchmarks for the FPS and CRAY/XMP*, Technical Report 111, Scientific Computing Associates, New Haven Connecticut, 1987.

[15] ———, *PCGPAK: User's Guide*, Technical Report 106, Scientific Computing Associates, New Haven Connecticut, 1984.

[16] Youcef Saad and Martin H. Schultz, *GMRES: A Generalized Minimal Residual Algorithm for Solving Nonsymmetric Linear Systems*, SIAM Journal on Scientific and Statistical Computing, 7(1986), pp. 856–869.

[17] J. Saltz and R. Mirchandaney and D. Baxter, *Run-time Parallelization and Scheduling of Loops, submitted to IEEE Transactions on Computers*, Report 88-70, ICASE, December 1988.

[18] J. H. Saltz and R. Mirchandaney, *Runtime Parallelization and Scheduling of Loops*, Technical Report, ICASE report 88-70 Submitted to Transactions on Computers, 1988.