2

NASA Contractor Report 181763

ICASE REPORT NO. 88-70

ICASE

RUN-TIME PARALLELIZATION AND SCHEDULING OF LOOPS

Joel H. Saltz

Ravi Mirchandaney

Doug Baxter

Contract Nos. NAS1-18107 and AFOSR 88-0117
December 1988

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

Operated by the Universities Space Research Association

NASA
National Aeronautics and
Space Administration
Langley Research Center
Hampton, Virginia 23665

Run-Time Parallelization and Scheduling of Loops *

Joel H. Saltz
Ravi Mirchandaney
Doug Baxter
Department of Computer Science
Yale University
New Haven, CT 06520

A-1

Abstract

~~In~~ this paper, ~~we~~ extend the class of problems that can be effectively compiled by parallelizing compilers. This is accomplished with the **doconsider** construct which would allow these compilers to parallelize many problems in which substantial loop-level parallelism is available but cannot be detected by standard compile-time analysis. ~~We~~ describe and experimentally analyze mechanisms used to parallelize the work required for these types of loops. In each of these methods, a new loop structure is produced by modifying the loop to be parallelized. ~~We~~ also ~~present~~ the rules by which these loop transformations may be automated in order that they be included in language compilers. The main application area of our research involves problems in scientific computations and engineering. The workload used in ~~our~~ experiments includes a mixture of real problems as well as synthetically generated inputs. From ~~our~~ extensive tests on the Encore Multimax/320, we have reached the conclusion that for the types of workloads we have investigated, self-execution almost always performs better than pre-scheduling. Further, the improvement in performance that accrues as a result of global topological sorting of indices as opposed to the less expensive local sorting, is not very significant in the case of self-execution.

[1] This research was supported by the U.S. Office of Naval Research under Grant N00014-86-K-0310, the United States Air Force Office of Scientific Research under Contract No. AFOSR Grant No. AFOSR 88-0117, and the National Aeronautics and Space Administration under NASA Contract No. NAS1-18107 while the first author was in residence at the Institute for Computer Applications in Science and Engineering (ICASE), NASA Langley Research Center, Hampton, VA 23665.

1 Introduction

There exist many problems in which substantial parallelism is available but where the parallelism cannot be exploited using the two principal concurrent loops described in the literature: ***doall*** and ***doacross*** [14] [6]. ***doall*** loops do not impose any ordering on loop iterations while ***doacross*** loops impose a partial execution order in the sense that some of the iterations are forced to wait for the partial or complete execution of some previous iterations. We propose a new type of loop, the ***doconsider*** construct. The ***doconsider*** loop allows loop iterations to be ordered in new ways that preserve dependency relations and increase concurrency. Often, these sorts of index reorderings can be done at very low cost and can have substantial benefits.

A variety of systems for restructuring loops and reordering indices have been developed in the functional language and systolic array generation communities. These methods rely on being able to detect the existence of uniform or quasi-uniform recurrence relations at compile-time. The dependency vectors characterizing these recurrence relations are examined and a new, hopefully more efficient way of traversing the dependency graph is found. We are able to handle loops whose inter-iteration dependency may be complex or where the dependences may be determined by variables whose values are not available until program execution begins. The methods we present here set up the framework, at compile-time, for performing a loop dependency analysis and produce a restructured loop that is reorderd on the basis of the information obtained from the dependency analysis. The actual dependency analysis is performed at the start of program execution. We will show that this kind of analysis can be performed very quickly and has very substantial payoffs.

Symbolic transformations are used to produce: (1) ***scheduling*** procedures that reorder and repartition index sets of loops and (2) ***executors*** or transformed versions of source code loop structures. These transformed loop structures carry out the calculations planned in the scheduling procedures. An executor may be regarded as a ***doacross*** loop that executes loop iterations in a modified order.

The scheduling mechanisms we explore are based on a topological sort. The index set is partitioned into disjoint subsets of indices or ***wavefronts***, such that work pertaining to all indices in a wavefront may be carried out in parallel. One method called ***global scheduling***, performs a topological sort of index set and assigns indices to processors in a way that evenly partitions the work in each wavefront. In each processor, indices are scheduled in order of increasing wavefront number. The other method called ***local scheduling***, starts out with a fixed assignment of indices to processors and simply rearranges the local ordering of those indices to improve parallelism.

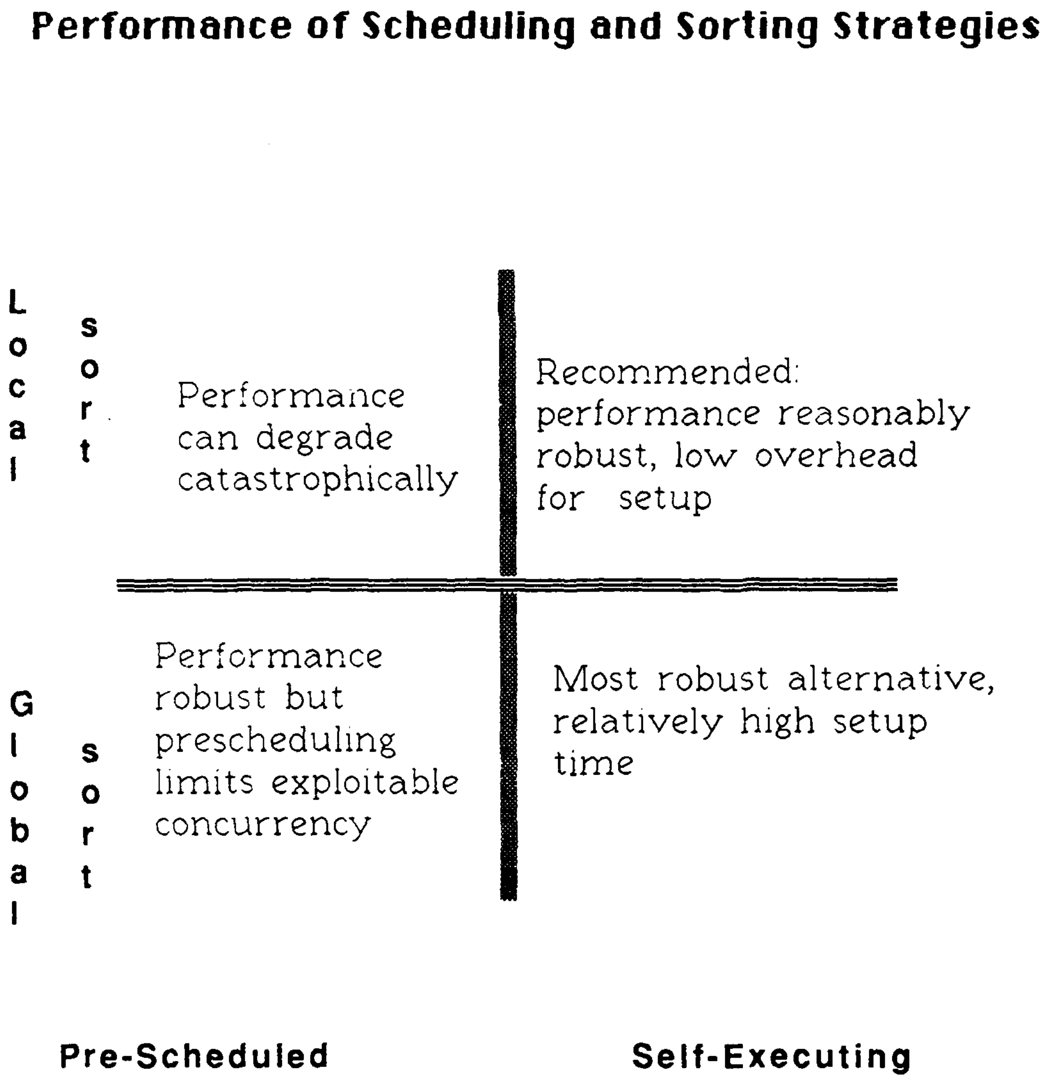
We investigate two types of executors in which indices belonging to each wavefront are partitioned among the processors. In the first executor, based upon ***pre-scheduling***, global synchronizations separate consecutive wavefronts. In the second executor, whcih we call ***self-executing***, a shared array is used to indicate whether a solution variable has been calculated. Global synchronizations are replaced by *busy waits* that ensure that needed values have been produced before those values are used.

We investigate the performance tradeoffs that characterize the different scheduling and execution methods we propose. The investigation uses a complete, commercial sparse matrix solver (PCGPAK [4]) used to solve a range of linear systems, a synthetic workload is also employed. We first clearly delineate the performance tradeoffs between pre-scheduled and self-executing loops. To fully explain the performance tradeoffs between these types of loops, 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 kinds of loops.

The method used to rearrange the index set of the loop to be parallelized will determine both the potential performance benefits that can be gained and the overhead that must be paid. We study the tradeoffs between *local* and *global* index set scheduling and conclude that for self-executing loops, local scheduling appears to lead to multiprocessor performance that is comparable to global scheduling in problems of interest at a significantly lower overhead cost.

From the results of experiments, we have reached the conclusion that for the types of workloads we have investigated, self-execution almost always performs better than prescheduling. Further, the improvement in performance that accrues as a result of global topological sorting of indices as opposed to the less expensive local sorting, is not very significant in the case of self-execution. Thus, we are left with a 2-dimensional solution space, as depicted in Figure 1, which pictorially summarizes the findings reported in this paper.

The rest of this paper is organized as follows: In Section 2, we provide simple rules that allow the transformation of certain types of loops into different parallel forms. These rules can be inserted into parallelizing compilers, extending the class of problems that can be effectively compiled for parallel machines. We describe some of the related research in Section 3. A simple mathematical model which captures the tradeoff between load balance and synchronization costs is described in Section 4. The results of multiprocessor experiments are presented in Section 5. These experiments provide a quantitative performance study of the schedulers and executors under consideration. Finally, we summarize our findings in Section 6.



Performance of Scheduling and Sorting Strategies

	Pre-Scheduled	Self-Executing
Local sort	Performance can degrade catastrophically	Recommended: performance reasonably robust, low overhead for setup
Global sort	Performance robust but prescheduling limits exploitable concurrency	Most robust alternative, relatively high setup time

Figure 1: Summary of Results

```
do i=1,n
     x(i) = x(i) + b(i)*x(ia(i))
end do
```

Figure 2: A simple loop

2 The Automated Execution System

2.1 Motivation

In a broad sense, modules of code in parallel programs are either compile-time or run-time schedulable. In order that a code be compile-time schedulable, it needs to possess sufficient information so that the compiler is able to extract the parallelism and map and schedule the code, e.g., *doall* type loops in Fortran[15]. In certain other types of codes, examination of run-time data is absolutely critical in order to detect hidden parallelism. We have been interested in the study of such problems. Within this class of run-time schedulable codes, there are two main categories, i.e., those that are *start-time* schedulable and those that are not.

Codes are *start-time* schedulable if all data dependences are resolved before the program begins execution and if these dependences do not change during the course of the computation. For codes that are not *start-time* schedulable, the data dependences may be determined by functions whose parameters are other functions, the values of which are only computed at some unknown point during the computation. In [11], we present self-execution primitives that aid greatly in the on-the-fly detection of parallelism in such problems. In this present paper, we will only be concerned with *start-time* schedulable problems.

Standard techniques developed by researchers in the field of parallel imperative compilers can determine when the data structure that describes the dependency relations is not changed during the course of the computation [1].

2.2 Transformation rules for automated system

In this section, we describe the rules by which an automated symbolic manipulator performs source to source transformation of a sequential user code into a suitable parallel version. These rules can be included in a conventional parallelizing compiler so that the class of problems that can be handled by these compilers is extended to include those that are start-time schedulable.

A loop of the form shown in Figure 2, may be executed many times during the run-

```
1:  doconsider i=1,n
2:      x(i) = x(i) +  b(i)*x(ia(i))
     }
```

Figure 3: An annotated loop

```
1:  do i=1,nlocal

1a:    isched = schedule(i)
1b:    needed_index = ia(isched)
2a:    if (needed_index >= isched) then
2b:        x(isched) =  xold(isched) +
                              b(isched)*xold(needed_index);

         else
3a:        while (ready(needed_index) .ne. COMPLETED)) end while
3b:        x(isched) = xold(isched) +
                                b(isched)*x(needed_index);
3c:        ready(isched) = ready(isched)+1;
         endif
      enddo
```

Figure 4: A Self-Executing loop

```
1a:  do i=1,nlocal
1b:      isched = schedule(i)

1c:      if (isched .eq. NEWPHASE) then
1d:        call global synchronization

          else
2a:        needed_index = ia(isched)
2b:        if (needed_index .ge. isched) then
2c:            x(isched) = xold(isched) +
                            b(isched)*xold(needed_index)
            else
2d:            x(isched) = xold(isched) +
                                  b(isched)*x(needed_index)
            endif
          endif
       enddo
```

Figure 5: A Pre-scheduled Loop

ning of a given program. We refer to this program as `simple`. The data dependences between the elements of x are determined by the values assigned during program execution to the data structure *ia*. 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 $x(i_1)$ requires $x(i_2)$.

In the first version of our system, user programs will need simple annotations which will direct the compiler to invoke its run-time parallelization modules. This will apply to complete parallel languages as well as language extensions which have explicit parallel sections. We propose to provide language extensions using constructs such as **`doconsider`** and **`forconsider`**, depending upon the language being extended. An annotated user code corresponding to Figure 2 is shown in Figure 3. These constructs will be used in addition to the `doall` and `doacross` type loops already provided in such systems. Details of these language extensions are currently being finalized. The mechanical process by which the run-time modules are invoked is described in [12]. Briefly however, an annotation of the type **`forconsider`** will generate code that is able to sort the indices on a processor in order of increasing wavefront number (details of this sorting procedure are provided in Section 2.3). Next, an appropriate transformation will be invoked to produce an *executor* to actually run the code using the newly created index ordering.

The example code shown in Figure 2 has been chosen for ease of explanation of the transformations we will present shortly. In the system that we are designing, realistic codes that tend to be much more complex in structure can and will be handled.

To parallelize such loops, the method we use is as follows: We first partition the indices of the outer loop of Figure 2 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 dependences 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*. A single program multiple data method of problem decomposition is used; the wavefront information is used to prepare a schedule of outer loop indices to be executed by each processor.

The main loop in Figure 4 corresponds to the indices assigned to this processor (line 1). The key point in Figure 4 has to do with line 3a and the while loop which ensures that an index is never used until it has been computed. Finally, the array **`ready`** is used to maintain the status of all the indices. In Figure 5, we depict the code transformed into one that uses barrier synchronization at the end of each phase. Before this code is executed, it is assumed that a topological sort of the data dependences is performed and the end of a phase is marked by a special flag with the appropriate index on every processor. A check is made to see if the end of phase is reached and if so, a call is made to **`global synchronization`**. The rest of the code is self-explanatory. It should be noted here that we first partition the index set. Given this static partition each

```
doconsider i=1,n
    temp = f(i)
    do j=1,m
        y(i) = y(i)+ temp*y(g(i,j))
    enddo
enddo
```

Figure 6: A nested loop

processor is informed when it should perform work associated with each of its assigned indices.

2.3 Efficient Calculation of the Topological Sort

The schedule of outer loop indices for each processor can be obtained by *global scheduling*, assigning indices to processors in a way that evenly partitions the work in each wavefront. In each processor, indices are scheduled in order of increasing wavefront number. Alternately using *local scheduling*, one begins with a fixed assignment of indices to processors and uses the wavefront information to simply rearrange the local ordering of those indices to improve parallelism.

The loops in the source code can be transformed to assign a wavefront number to each loop index. For instance, a loop of the form depicted in Figure 6 is converted to the transformed loop in Figure 7. Since the wavefront number for each index is one plus the maximum of the wavefront numbers of the indices on which it depends, one can simply sweep sequentially through the indices and calculate the wavefront for each index. Figure 7 depicts a version of the topological sorting procedure. This process produces an array `maxwfy`, as shown in Figure 7. Array `maxwfy` 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.

While local scheduling is almost completely parallelizable, it is not clear how one would efficiently parallelize global scheduling. The interprocessor coordination required for this rather fine grained computation appears to be prohibitive in the absence of a fetch and add primitive.

```
do i=1,n
   temp = f(i)
   mywf = 0
   for do j=1,m
       mywf = max(maxwfy(g(i,j)),mywf)
   enddo
maxwfy(i) = mywf;
enddo
```

Figure 7: Computation of Wavefronts

We now provide a short stepwise description of the automated procedure which takes as input a code of the type shown in Figure 6 and restructures it into a suitable parallel version. Steps 1 through 3 are performed at compile-time, while steps 4 and 5 are performed at run-time.

1. The indices of the computation are logically distributed among the processors in some *specified manner*.
2. A topological sort code is then generated by the compiler, ***during program execution*** this code which determines the wavefront number of each index (Figure 7).
3. The loop in Figure 6 is transformed into a self-executing or pre-scheduled version, with the optional insertion of the code that repartitions indices among the processors.
4. At start of execution, the wavefront numbers are computed and the indices are sorted on the basis of these wavefronts. The indices may or may not be repartitioned.
5. The actual computation is now performed by each processor on its assigned subset of indices, using one of the executors that have been generated, as in step 3.

3 Related Work

The execution of parallel tasks using self-scheduling has received considerable attention. Lusk and Overbeek [10] implement a self-scheduled mechanism to dynamically allocate work to processors. While this method has the advantage of simplicity, many of the more complex dynamic problems that we are interested in solving do not seem to be easily formulated in this framework. Polychronopoulos and Kuck [16] are concerned with the efficient execution of *doall* type loops using run-time self scheduling. While the efficacy of self-scheduling for certain classes of problems on shared memory machines is demonstrated in that paper, more complex problems which cannot be formulated in a *doall* setting are not studied. Tang and Yew[19] describe a mechanism to execute multiple nested *doall* loops, using self-scheduling. It is shown that for certain types of problems, self-scheduling is more efficient than pre-scheduling using static assignment of loop iterations to processors. Krothapalli and Sadayappan[9] describe a method which is able to remove anti- and output-dependences, by performing an analysis of the reference pattern generated and using multiple copies of variables in order to simulate a single assignment language. Cytron[6] discusses the problem of how to schedule *doacross* loops with lexically backward dependences by introducing delays in appropriate places in the code to ensure correctness. A linear programming problem is formulated and solved in order to calculate the minimum delays.

Loop restructuring has been used successfully to allow parallelizing compilers to improve parallelism and enhance performance in memory hierarchies [14], [15],[2],[7]. To our knowledge, there has been no work in the automatic detection of run-time parallelism along with the restructuring of such loops for efficient scheduling.

Numerical methods for solving sparse triangular systems have however employed closely related schemes to reorder operations to increase available parallelism, [3],[18],[5],[8],[17].

As far as performance improvement is concerned, we show the efficacy of our techniques. From a programming language standpoint, we believe that user codes for parallel machines ought not to include the details of scheduling and mapping. This has several advantages: program portability will certainly become more feasible and program development time will decrease. We believe that robust transformations which automatically restructure programs to exploit parallelism will aid in reducing the effort required to program parallel machines.

4 Description and Analysis of Model Problems

4.1 Model Problems

There are several ways to generate the workload needed to test the various aspects of the system. In our experiments, these model problems come from two main sources, i.e., the solutions of sparse linear systems arising from a variety of partial differential equations using preconditioned Krylov methods and from parameterized synthetic workload generators. We examine in particular detail the solution of sparse triangular systems obtained through incomplete factorizations of matrices arising from discretizations of the partial differential equations in question on a variety of two and three dimensional meshes. A description of the problems solved are found in Appendix 1. The solution of

```
S1:  do i=1,n
        y(i) = rhs(i)
S2:     do j=ija(i),ija(i+1)-1
            y(i) = y(i) - a(j)*y(ija(j))
        end do
     end do
```

Figure 8: Triangular Solve

these sparse triangular systems accounts for a large fraction of the sequential execution time of these linear solvers. The dependences encountered in solving these systems inhibit the parallelization of the outer loop of row substitutions (S1 in Figure 8). Typically the number of non-zero elements in a row is too small to allow efficient parallelization of the inner loop (S2 in Figure 8).

We also present overall performance results for a commercial preconditioned Krylov solver PCGPAK which was *completely parallelized*. Parallelization was carried out using either the pre-scheduled or self-executing constructs presented here. Details of how the parallelization was carried out are presented in Appendix 2, a much more detailed account of the PCGPAK results is presented in [4].

For a more general source of matrices, we utilize a a simple workload generator which is able to incorporate the important parameters such as locality of communications, volume of communication between nodes etc, in the generation of matrices. The synthetic workload generator should have the following properties:

- The output of the generator should approximately be able to describe approximately some of the real problems we encounter, implying that the workload is not completely random.
- It should be easy to vary the input parameters of the workload generator to test certain canonical features of the sofware system.

Clearly, having such a generator will provide faster turnaround time for performance testing and because it will be easy to vary the parameters, the testing of the software modules will be more robust.

Most of the problems that we have been interested in solving have the following charactertistics:

- The computation is defined over a reasonably large index set of values.
- There exists a phase structure inplicit in the computation such that not all indices

can be executed at the same instant because of certain data dependences that must be satisfied.

- Usually, indices interact with other indices that are close by, where closeness is a feature of the physical problem being solved.

In the first implementation of the workload generator, we have made the following approximations: The input domain consists of a 2-dimensional mesh of points whose connections have yet to be established. Each point in the mesh is a unique index of the computation to be performed using that mesh. The points are numbered using their natural ordering. We use two probability distributions to model the workload; one determines the total number of dependency links between an index and other indices in the domain, the other is used to determine the locality of the links to be forged.

The number of indices that any given index needs to communicate its output with is given by a Poisson density function, with parameter λ. The Poisson approximation is reasonable because several physical phenomena can be modeled using this random variable. The density function for this random variable is defined as follows:

$p_i = \mathbf{Pr}[X = i] = \lambda^i \epsilon^{-\lambda}/i!, \ i \geq 0$

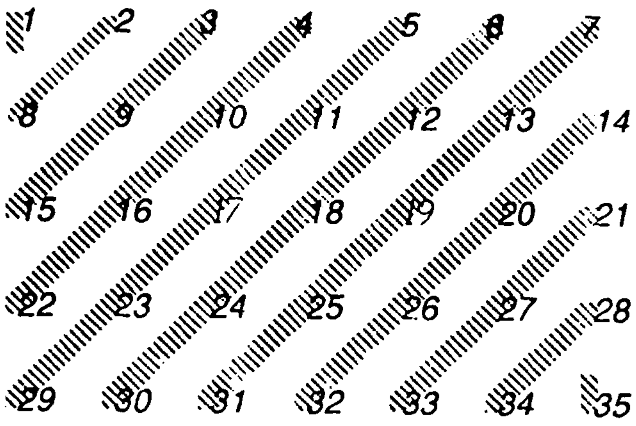
Depending upon the value of λ, the probability density can be varied to suit the problem at hand. Further, the Poisson density function is often used as an approximation to the Binomial density function, which is normally much more expensive to compute.

It is known that many problems in nature have the property that spatial regions of the problem domain tend to interact more intensely with adjacent or close-by regions. A probability density function that possesses such a property is the geometric density function. A random variable X has a range 1, 2,... and density function

$\mathbf{Pr}[X = i] = (1 - p)p^i, \ \forall i \geq 0$

The interpretation that we ascribe to this density function is as follows: If an index value k is to communicate with an index that is i distant from itself, the set of indices that are i units away (using the Manhattan metric) from index k is determined. One of these indices (if any) is selected in order to make a connection with k. This process of making connections is continued until all the links are exhausted for each index. Thus, we can generate a data-dependency matrix using the mesh generated by the above procedure.

In the following subsection, we present the results of experiments by which we determine the performance of the schedulers and executors under consideration.



**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 9: Assignment of Indices to Wavefronts

4.2 Analysis of a Model Problem

We will use a model problem to illustrate the performance difference between using prescheduing 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 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 $T_p = S/(mn)$; This ignores the relatively minor disparities caused by the matrix rows represented by points on the lower and the left boundary of the domain.

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 anti-diagonal strip of the domain. For instance, in Figure 9, we depict a five by seven domain with the points in each wavefront linked

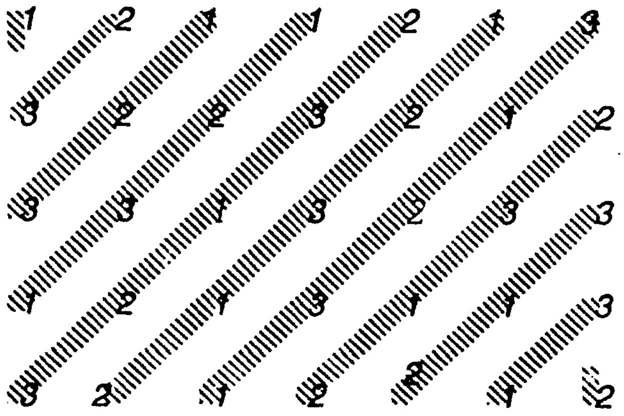


Figure 10: Assignment of Indices to Processors

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 the example problem in Figure 10. When pre-scheduling is used, the computation is divided into phases separated by global synchronizations.

A brief inspection of Figure 9 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 $T_p MC(j)$. The computation time required to complete the problem is consequently

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

We now proceed to calculate $MC(j)$. During phase j, a total of j strips 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 $\min(m,n)$ strips 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 $n + m - j$ strips 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

$$\begin{aligned} T_C &= T_p \sum_{j=1}^{n+m-1} MC(j) \\ &= \frac{S}{mn} (\sum_{j=1}^{\min(m,n)-1} \lceil \frac{j}{p} \rceil + (n + m - 2\min(m,n) + 1) \lceil \frac{\min(m,n)}{p} \rceil + \\ & \sum_{j=m+n-\min(m,n)+1}^{n+m-1} \lceil \frac{n+m-j}{p} \rceil) \end{aligned}$$

By assumption, the sequential time to solve the problem is $S = mnT_p$. 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

$$\begin{aligned} mn(& \sum_{j=1}^{\min(m,n)-1} \lceil \frac{j}{p} \rceil + (n + m - 2\min(m,n) + 1) \lceil \frac{\min(m,n)}{p} \rceil + \\ & \sum_{j=m+n-\min(m,n)+1}^{n+m-1} \lceil \frac{n+m-j}{p} \rceil)^{-1} \end{aligned} \tag{2}$$

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 $j \leq \min(\hat{m}, \hat{n}) - 1$ when j is not a multiple of p, there are $p - j \textit{ mod } p$ 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:

$$L_{tr} = \frac{T_p \min(\hat{m}, \hat{n}) \sum_{l=1}^{p} (l-1)}{p} = \frac{T_p \min(\hat{m}, \hat{n})(p-1)}{2} \quad . \tag{3}$$

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

$$L_{ss} = T_p(p - \min(m, n) \bmod p)$$

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 $E_{opt} =$

$$\frac{mn}{mn + \min(\hat{m}, \hat{n})(p-1) \quad + \quad (m + n + 1 - 2\min(\hat{m}, \hat{n}))((p - \min(m, n)) \bmod p)} \tag{4}$$

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 [13] we rearrange the global synchronizations in a way that obtains a tradeoff between improved load balance and the costs of the global synchronizations. While that mechanism is 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 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 11 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. 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

$$\frac{mn}{mn + p(p-1)} \tag{5}$$

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,

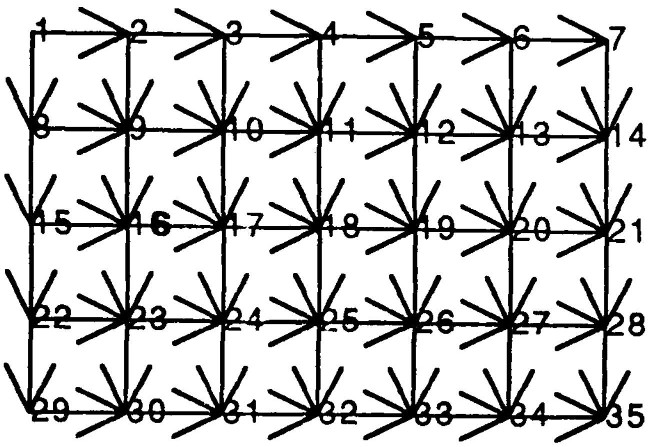


Figure 11: Data Dependences between Indices

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 expression below, $R_{synch} = \frac{T_{synch}}{T_p}$, $R_{inc} = \frac{T_{inc}}{T_p}$ and $R_{check} = \frac{T_{check}}{T_p}$.

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

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

$$\frac{2p + R_{synch}}{(p+1)(1 + R_{inc} + 2R_{check})} \tag{6}$$

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. 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

$$\frac{1}{1 + R_{inc} + 2R_{check}}. \tag{7}$$

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 Multiprocessor Timings

The experimental results in this section are organized in the following manner: We describe the performance of PCGPAK using the self-executing and pre-scheduled executors. Next, we perform a detailed analysis of the various timing losses that occur in

Table 1: Self-Execution vs Pre-Scheduling for PCGPAK 16 Processors of the Encore Multimax

Test Problem	Pre-Scheduled		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

the code. This detailed analysis does not use PCGPAK, instead we use a separate set of programs written to study the issues we are investigating. The pre-scheduled executor's performance is compared using local and global sorting of the indices based upon their wavefronts. Because we see that the performance of the pre-scheduled executor is almost always worse than that of the self-executing version, we restrict some of our later studies to the self-execution system.

In the case of the synthetic workload, a matrix represented as 65-4-3 implies the discretization of a 65*65 mesh where the average number of edges leaving a mesh point equals 4, with a Poisson distribution, and the average distance between connections being 3, with a geometric distribution.

5.1.1 Pre-scheduled vs. self-execution

Two versions of parallel PCGPAK, a Krylov space solver [4], were produced. In the first version. the triangular solves and the numeric factorization were implemented using self-scheduling; in the second 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 in a wrapped manner. The timings were done on an Encore Multimax/320 with 13 megahertz APC/02 boards and version 2.1 of the FORTRAN compiler.

In Table 1 we present time required to solve the test problems for the pre-scheduled and self-executing versions of PCGPAK, along 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 pre-scheduled version.

Overheads in the self-executing version of the program arise from the need to check and update the shared array which indicates whether needed solution variables or pivot rows have been computed. In the pre-scheduled version of the program, overheads arise from the cost of global synchronizations. Overheads aside, it is possible to show that the parallelism available from the self-executing version of the program is always better than in the pre-scheduled version. Measured efficiencies for all problems except 7-PT and L7-PT favor the self-executing version of the program.

In section 5.1.2, we will explain the differing relative performance between the pre-scheduled 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 1, the times required to perform the topological sort for each of the test problems. In each of these test problems, the time required to perform the topological sort required for global index scheduling was quite small, compared to the total execution time. Since the scheduling had only to be performed once and was amortized over a substantial number of iterations, even the relatively expensive global scheduling did not represent a troublesome overhead. The cost of performing both global and local scheduling will be examined in much more detail in the following sections.

5.1.2 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 2 and 3 respectively, are depicted symbolically estimated efficiencies for self-executing and pre-scheduled 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 5.1.1. 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 2 and 3 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 *1 PE seq.* in tables 2 and 3).

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 2 and 3 show the single processor parallel code timing divided by the product of the number of processors used and the symbolically estimated efficiencies (*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 ofg loatinripoint 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 codes. 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 pre-scheduled 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 pre-scheduled 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 pre-scheduled and self-executing programs becomes more apparent in comparing tables 2 and 3. For the 5-PT and SPE2 test problems, the difference in the load balance obtainable through the use of pre-scheduled and self-executing codes is large enough that the *1 PE Seq* time for the pre-scheduled code is greater than the *Parallel Time* for the self-executing program. Even if we had a hypothetical pre-scheduled code with no overheads except for load imbalance, that code would still be less efficient than the self-executing program. Recall that the pre-scheduled 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 pre-scheduling is relatively small. Since little loss due to load imbalance is seen for 7-PT, we are able to see that pre-scheduling gives a slightly faster timing.

In Table 2 we depict the time required for a *doacross* loop to execute each triangular solve. We see that the doacross loop is consistently less efficient than either the prescheduled or self-executing loops. For example in the SPE5 problem, the self-executing solve requires 23.4 milliseconds, the prescheduled solve (in Table 3) required 29.0 milliseconds and the doacross version of the solve took 45.0 milliseconds.

Recall that the self-executing loop is a doacross loop with a reordered index set. We expect that the doacross loop will exhibit less concurrency than the self-executing loop. Since the doacross loop does not have to perform array references to access the reordered index set, we expect that the doacross will also be accompanied by smaller overheads. The results of measurements not presented here confirm that while the concurrency obtained from doacross loops was quite limited, doacross loop execution was accompanied by less overhead. On the Multimax/320 measurments indicate that accessing the reordered index set is relatively expensive and hence the performance differences between the doacross loops and the reordered loops is attenuated to some degree.

5.1.3 Timing Projections

Since we can accurately account for the execution time in the Encore Multimax/320, it is reasonable to make some timing projections. These projections make the assumption

Table 2: Parallel Time and Estimates for Self-Executing Triangular Solves

Test Problem	Phases	Symbolic Efficiency	Parallel Time	Rotating Estimate	1 PE Parallel	1 PE Seq.	Doacross Time
SPE2	60	0.89	20.7	20.0	17.9	15.0	33.9
SPE5	66	0.96	23.4	21.6	18.5	15.3	45.0
5-PT	124	0.95	18.7	17.6	14.5	12.2	37.1
9-PT	311	0.97	57.9	57.1	51.7	43.2	97.5
7-PT	58	0.98	56.3	57.6	45.1	38.1	84.1

Table 3: Parallel Time and Estimates for Pre-Scheduled Triangular Solves

Test Problem	Phases	Symbolic Efficiency	Parallel Time	Rotating Estimate + Barrier	Rotating Estimate	1 PE Parallel	1 PE 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

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 balance were perfect, the *Best* efficiencies in Table 4 would be obtained.

The estimate of non load balance related loss (*Best* in table 4) 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 such projections 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 4, we present efficiencies for 16 processors and projected efficiencies for 32 and 64 processors. The projected performance of the pre-scheduled 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 4 reflect the varying relative costs of global synchronizations and array writes in problems with

Table 4: Estimated Efficiencies for Larger Machines

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
7-PT	66	70	65	66	64	62	60	55
9-PT	76	64	73	52	68	26	39	12

different structures, this issue was discussed in Section 5.1.2.

5.1.4 Effects of Local Reordering

In Figure 12, we demonstrate the crucial role played by the synchronization mechanism in determining performance, when indices are not repartitioned after a topological sort. We compare the estimated efficiency of the same partition and schedule using global synchronization and self-executing synchronization in a matrix generated by a 65 by 65 point mesh using a 5 point stencil. Indices were assigned to processors in a striped manner, i.e. for P processors index i was assigned to processor i modulo P. The schedule was produced by performing a topological sort and scheduling indices in each phase in order of increasing index number. We can see that the results obtained through the use of global synchronization vary wildly with the number of processors used. This is understandable when we realize that the poor performance arises from the poor distribution of indices among processors in any given phase. All work assigned to a phase must be completed before any work corresponding to the next phase can commence. Often, many, if not all the indices in a phase get assigned to a single processor, resulting in sequential execution for that phase. We saw this effect to a significant degree in all of the problems we examined although we carefully selected the 65 by 65 point mesh as the source of the dramatic performance fluctuations are particularly evident from the structure of the problem.

In Figure 12, we also depict the performance obtained on the model problem when self-executing synchronization is employed. In a great many cases, data from all indices in a given wavefront are not actually required by each index in the next wavefront. When self-executing synchronization is employed, a pipeline sort of effect may be generated and we see substantial performance benefits. Pre-scheduling on the other hand, appears to be much less robust.

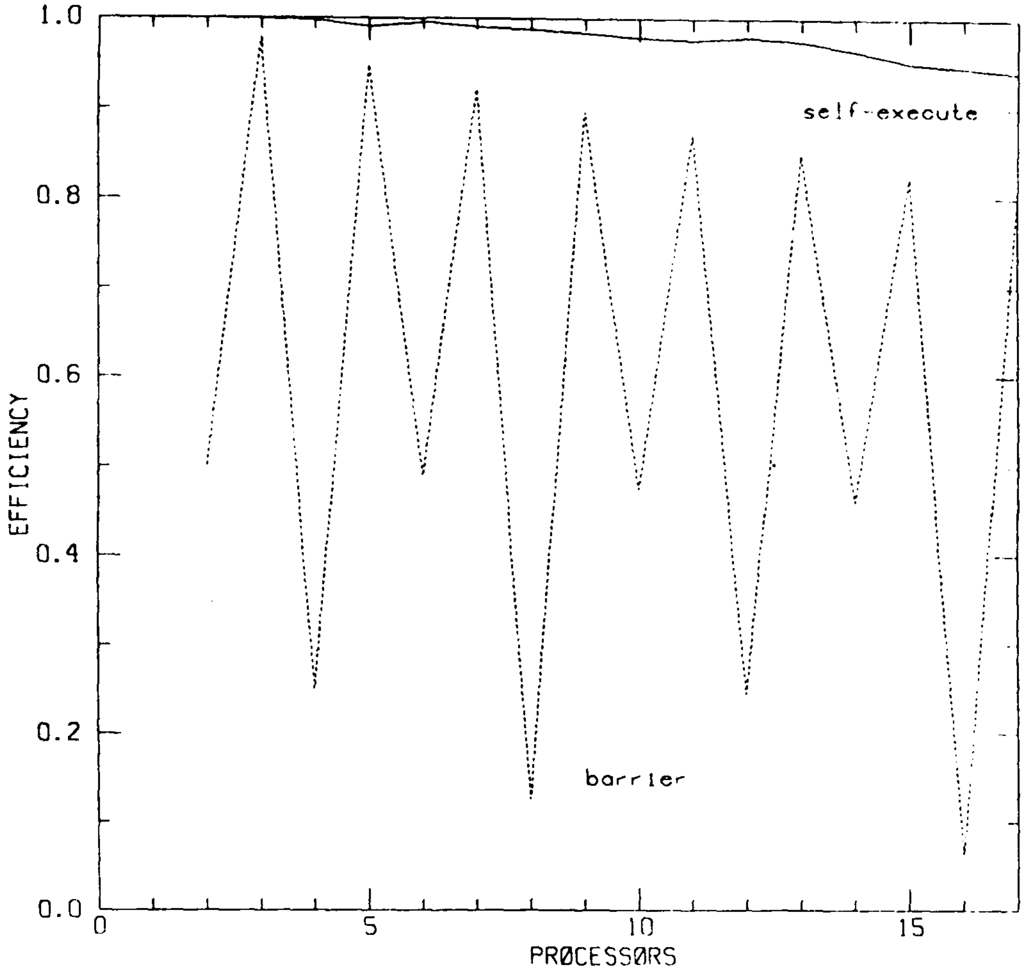


Figure 12: Effect Of Local Ordering

5.1.5 Local v.s. Global Index Set Scheduling

We performed a set of experiments to examine the performance tradeoffs between local and global index set scheduling defined in sections 1. We used only the self-executing loop structures in the experiments in this section. Recall that when *global* index set scheduling is used, the index set is sorted in increasing wavefront order. The index set is then partitioned between processors in a striped manner. For the *local* sorting method is used, the initial partition of indices is maintained, but their ordering is changed based upon wavefront numbers. In Table 5 we present the sequential time required to solve each test problem, the times required to perform a sequential and a parallel version of the sort and the time required to rearrange indices globally. All times in this table are in milliseconds. We also depict the time required to perform local index set scheduling as well as the 16 processor Multimax/320 timings obtained using these schedules. The time required to perform the sequential scheduling is slightly lower than the time needed for performing a sequential iteration. For example, in the case of SPE5, the time required to perform the sequential sort plus the triangular solve adds up to 220 ms, while a completely sequential execution takes 240 ms. Because we pay for the sorting only once, subsequent iterations of the code will show a great advantage for the parallel code (30 ms vs. 240 ms on 16 processors). The time required to produce a parallelized global schedule ranged from 17 percent to 61 percent of the time needed for a sequential iteration.

From Table 5, we can see that local index set scheduling overhead does turn out to be much less than global index set scheduling overhead, as is to be expected. However, as far as run times were concerned, local and global scheduling each yielded better results than the other for some test problems. For example, in the case of SPE2, global run time was 21.3 ms and local was 29.6 ms and for SPE3, global gave a run time of 25.1 while local was 22.3 ms.

6 Conclusions and Future Work

There is a hierarchy of problems with different levels of scheduling complexity that are of interest to researchers in the field of parallel programming. When the data dependences of the problem are known at compile-time, task decomposition can automatically be performed by the compiler. However, there are problems where workloads cannot be fully characterized during compilation due to data dependences that become manifest at run-time. In [12], we presented our initial results from applying these ideas to pre-schedulable problems. In this paper, we have extended the class of problems that can be effectively compiled by parallelizing compilers. We presented the `doconsider` construct which would allow these compilers to effectively parallelize such problems.

In this paper, we have reached the conclusion that for the types of workloads we have

Table 5: Global v.s. Local Scheduling

Test	Global			Local		Sequential
Problem	seq sort	parallel sort	solve	parallel sort	solve	
SPE1	43.9	31.0	6.2	14.9	6.0	50.5
SPE2	135.1	48.8	21.3	29.9	29.6	223.2
SPE3	245.0	135.0	25.1	56.0	22.3	245.0
SPE4	46.1	29.8	6.3	7.1	6.0	47.6
SPE5	191.3	100.0	30.0	46.0	23.6	240.9
seven.l	466.5	203.5	57.5	78.0	54.7	615.7
nine.l	465.0	153.6	58.3	100.6	62.6	698.3
fivers.l	148.7	72.1	24.0	38.8	28.8	192.0
65-10-1.5	384.8	173.6	58.8	109.6	66.8	633.6
65-10-3	423.5	131.7	58.5	63.1	79.8	767.5
65-4-1.5	284.4	106.4	34.4	38.8	44.2	394.9
65-4-3	277.4	101.1	44.4	35.4	44.3	386.1
65mesh	213.0	149.3	30.6	82.0	22.8	241.7

investigated, self-execution almost always performs better than pre-scheduling. Further, the improvement in performance that accrues as a result of global topological sorting of indices as opposed to the less expensive local sorting, is not very significant in the case of self-execution. Thus, we are left with a 2-dimensional solution space, as depicted in Figure 1, which pictorially summarizes the findings reported in this paper. As regards program transformations are concerned, we have shown how simple annotations might be included in parallel languages in order to aid the compiler to create the appropriate scheduler and executor, given a shared memory architecture.

Acknowledgements: The authors would like to thank Martin Schultz and Stan Eisenstat for helpful discussions and Scientific Computing Associates for use of PCGPAK related data.

References

[1] F. Allen, M. Burke, P. Charles, R. Cytron, and J. Ferrante. An overview of the ptran analysis system for multiprocessing. In *The Proceedings of the 1987 conference on supercomputing, Athens, 1987*, 1987.

[2] J. R. Allen, D. Callahan, and K. Kennedy. Automatic decomposition of scientific programs for parallel execution. In *Conf. Record, 14th POPL*, January 1987.

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

[4] D. Baxter, J. Saltz, M. Schultz, and S. Eisenstat. *Preconditioned Krylov solvers and methods for runtime loop parallelization.* Technical Report 655, Yale University, 1988.

[5] D. Baxter, J. Saltz, M. Schultz, S. Eisentstat, and K. Crowley. An experimental study of methods for parallel preconditioned krylov methods. In *Proceedings of the 1988 Hypercube Multiprocessor Conference, Pasadena CA*, January 1988.

[6] R. Cytron. Doacross: beyond vectorization for multiprocessors. In *The Proceedings of the ICPP, 1986*, pages 836–844, 1986.

[7] K. Gallivan, W. Jalby, and D. Gannon. On the problem of optimizing data transfers for complex memory systems. In *Proceedings of the 1988 ACM International Conference on Supercomputing , St. Malo France*, July 1988.

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

[9] V. Krothapalli and P. Sadayappan. An approach to synchronization for parallel computing. In *The Proceedings of the 1988 conference on supercomputing, St. Malo, 1988*, pages 573–581, 1988.

[10] E. Lusk. R. Overbeek, and et. al. *Portable Programs for Parallel Processors.* Holt, Rinehart and Winston Inc., 1987.

[11] R. Mirchandaney and J. H. Saltz. *Dodynamic: A construct for on-the-fly parallelization of loops.* Technical Report 650, Yale University, 1988. in preparation.

[12] R. Mirchandaney, J. H. Saltz, R. M. Smith, D. M. Nicol, and Kay Crowley. Principals of runtime support for parallel processors. In *Proceedings of the 1988 ACM International Conference on Supercomputing , St. Malo France*, July 1988.

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

[14] D. A. Padua, D. J. Kuck, and D. H. Lawrie. High-speed multiprocessors and compilation techniques. *IEEE Trans. on Computers*, 29(9):763–776, September 1980.

[15] D. A. Padua and M. J. Wolfe. Advanced compiler optimizations for supercomputers. *CACM*, Dec 1986.

[16] C. Polychronopoulos and D. Kuck. Guided self-scheduling: a practical scheduling scheme for parallel supercomputers. *IEEE Transactions on Computers*, 1987.

[17] J. Saltz. Aggregation methods for solving sparse triangular systems on multiprocessors. *SIAM J. Sci. and Stat. Computation.*, to appear, 1989.

[18] J. Saltz. Methods for automated problem mapping. In *The IMA Volumes in Mathematics and its Applications. Volume 13: Numerical Algorithms for Modern Parallel Computer Architectures Martin Schultz Editor*, Springer-Verlag, 1988.

[19] P. Tang and P. Yew. Processor self-scheduling for multiple nested parallel loops. In *The Proceedings of the ICPP, 1986*, pages 528–535, 1986.

1. Appendix I

1.1. Preconditioned Krylov Methods Background

We briefly present the basics of Krylov methods such as are found in PCGPAK.

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

$$Mx = b \tag{1.1}$$

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

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

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

The basic tasks involved 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 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 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*. 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.

1.2. The Test Problems

We now present the eight test problems used in our experiments.

Problem 1 (SPE1) This problem models the pressure equation in a sequential black oil simulation. The grid is 10 x 10 x 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 x 6 x 5 with 6 unknowns per grid point giving 1080 unknowns. The matrix is a block seven point operator with 6 x 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 x 11 x 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 x 23 x 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 x 23 x 3 grid with 3 x 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 x 63 giving 3969 unknowns. The L5-pt problem is the same problem with a 200 x 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 x 63 giving 3969 equations. The L9-pt problem is the same problem with a 127 x 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 x 20 x 20 yielding 8000 equations. The L7-pt problem is the same problem with a 30 x 30 x 30 grid.

2. Appendix II: Parallel Implementations of the Basic Krylov Method

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

The easily parallelizable procedures 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.

2.2. Parallel Triangular Solves and Sparse Numeric Factorizations

2.2.1. Triangular Solves

The triangular solve and the sparse numeric factorization can often be efficiently parallelized once the matrix dependent data dependencies are known. Refer to Figure 8 for a description of the triangular solve code.

2.2.2. 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 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 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. On the other hand, when row 2 is stabilized, non zeros in that row may arise directly from a non-zero present in the original matrix *or* may arise as a result from fill from row 1. 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.

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 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 13). 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 13: Schematic Sparse Factorization

2.3. Sparse Symbolic Factorizations

Because the pattern of fill is not known, 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 dealing with incomplete factorization 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 are omitted.

NASA

Report Documentation Page

1. Report No. NASA CR-181763 ICASE Report No. 88-70	2. Government Accession No.	3. Recipient's Catalog No.
4. Title and Subtitle RUN-TIME PARALLELIZATION AND SCHEDULING OF LOOPS		5. Report Date December 1988 6. Performing Organization Code
7. Author(s) Joel H. Saltz; Ravi Mirchandaney and Doug Baxter		8. Performing Organization Report No. 88-70 10. Work Unit No. 505-90-21-01
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		11. Contract or Grant No. NAS1-18107 13. Type of Report and Period Covered Contractor Report
12. Sponsoring Agency Name and Address National Aeronautics and Space Administration Langley Research Center Hampton, VA 23665-5225		14. Sponsoring Agency Code

15. Supplementary Notes

Langley Technical Monitor:
Richard W. Barnwell

Submitted to Trans. Comput.

Final Report

16. Abstract

In this paper, we extend the class of problems that can be effectively compiled by parallelizing compilers. This is accomplished with the doconsider construct which would allow these compilers to parallelize many problems in which substantial loop-level parallelism is available but cannot be detected by standard compile-time analysis. We describe and experimentally analyze mechanisms used to parallelize the work required for these types of loops. In each of these methods, a new loop structure is produced by modifying the loop to be parallelized. We also present the rules by which these loop transformations may be automated in order that they be included in language compilers. The main application area of our research involves problems in scientific computations and engineering. The workload used in our experiments includes a mixture of real problems as well as synthetically generated inputs. From our extensive tests on the Encore Multimax/320, we have reached the conclusion that for the types of workloads we have investigated, self-execution almost always performs better than pre-scheduling. Further, the improvement in performance that accrues as a result of global topological sorting of indices as opposed to the less expensive local sorting, is not very significant in the case of self-execution.

17. Key Words (Suggested by Author(s)) parallelizing compilers, languages, Kryloff methods, doall loops	18. Distribution Statement 61 - Computer Programming and Software Unclassified - unlimited

19. Security Classif. (of this report) Unclassified	20. Security Classif. (of this page) Unclassified	21. No. of pages 34	22. Price A03