**NASA Contractor Report** 182039
**ICASE Report No. 90–34**

# ICASE

## RUN-TIME PARALLELIZATION AND SCHEDULING OF LOOPS

**Joel H. Saltz**
**Ravi Mirchandaney**
**Kay Crowley**

Contract No. NAS1–18605
May 1990

Revised version of ICASE Report No. 88–70

Institute for Computer Applications in Science and Engineering
NASA Langley Research Center
Hampton, Virginia 23665–5225

Operated by the Universities Space Research Association

National Aeronautics and
Space Administration

**Langley Research Center**
Hampton, Virginia 23665-5225

# RUN-TIME PARALLELIZATION AND SCHEDULING OF LOOPS[1]

*Joel H. Saltz*
Institute for Computer Applications in Science and Engineering
NASA Langley Research Center
Hampton, VA 23665

*Ravi Mirchandaney*
*Kay Crowley*
Department of Computer Science
Yale University
New Haven, CT 06520

## ABSTRACT

In this paper, we study run-time methods to automatically parallelize and schedule iterations of a do loop in certain cases, where compile-time information is inadequate. The methods we present in this paper involve execution time preprocessing of the loop. At compile-time, these methods set up the framework for performing a loop dependency analysis. At run-time, wavefronts of concurrently executable loop iterations are identified. Using this wavefront information, loop iterations are reordered for increased parallelism.

We utilize symbolic transformation rules to produce: (1) *inspector* procedures that perform execution time preprocessing and (2) *executors* or transformed versions of source code loop structures. These transformed loop structures carry out the calculations planned in the *inspector* procedures. We present performance results from experiments conducted on the Encore Multimax. These results illustrate that run-time reordering of loop indices can have a significant impact on performance. Furthermore, the overheads associated with this type of reordering are amortized when the loop is executed several times with the same dependency structure.

[1]Research supported by the U.S. Naval Office under Grant N00014-86-K-0310, by the NSF under Grant ASC-8819374, as well as by the National Aeronautics and Space Administration under NASA Contract No. NAS1-18605 while the authors were in residence at the Institute for Computer Applications in Science and Engineering (ICASE), NASA Langley Research Center, Hampton, VA 23665.

# 1 Introduction

Dependencies between various iterations of a do loop cannot always be characterized during program compilation. This inability to characterize dependencies can inhibit exploitation of potential parallelism by means of the usual types of parallel loop constructs, the *doall* or *doacross* loops [15] [4]. *Doall* loops do not impose any ordering on loop iterations while *doacross* loops impose a partial execution order so that some of the iterations are forced to wait for the partial or complete execution of some previous iterations. Typically *doacross* loops make use of a-priori knowledge of inter-iteration dependencies to carry out required synchronizations.

Parallelization carried out during compilation is necessarily conservative; if it cannot be shown that loop iterations can be performed concurrently, the loop iterations are sequentialized. Many loop nests defy compile-time parallelization because dependency patterns are determined by variables or arrays initialized *during program execution.* Compile-time analysis and parallelization may also fail when non-linear expressions characterize the dependencies.

The methods we present in this paper involve execution time preprocessing of loops. We describe a form of execution time preprocessing that is able to transform a loop having inter-iteration dependencies into a sequence of *doall* loops. At compile-time, these methods set up the framework for performing a loop dependency analysis. At run-time, *wavefronts* of concurrently executable loop iterations are identified. The original loop L is transformed into two loops, $L_1$ and $L_2$. The new outer loop $L_1$ is sequential, and the new inner loop $L_2$ involves all indices assigned to each wavefront. The transformation we present is analogous to *loop skewing*, except that the identification of parallelism is carried out during program execution [25]. For some loops, reordering iterations can reduce the deleterious effects of inter-iteration dependencies on performance.

In all of the situations described above, we use symbolic transformations to produce: (1) *inspector* procedures that perform execution time preprocessing and (2) *executors* or transformed versions of source code loop structures. These transformed loop structures carry out the calculations planned in the *inspector* procedures.

Characterizing the cost of execution time preprocessing is a critical aspect of this research. One clear requirement is that the execution time preprocessing itself be parallelizable. The wavefront scheduling mechanisms are based on a topological sort, which can be parallelized using a version of a *doacross* loop. One method called *global scheduling*, performs a topological sort of the 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.

Some of the related work in this field is described next. Lusk and Overbeek [12] implement a self-scheduled mechanism to dynamically allocate work to processors, however this work is restricted to *doall* loops. Polychronopoulos and Kuck [17] 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, loops that are not *doall* are not addressed. Tang and Yew[24] 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 prescheduling using static assignment of loop iterations to processors. Krothapalli and Sadayappan[11] describe a method which is able to remove anti- and output-dependencies, by performing an analysis of the reference pattern generated and using multiple copies of variables in order to simulate a single assignment language. Cytron[4] discusses the problem of how to schedule *doacross* loops with lexically backward dependencies 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 [15], [16], [1], [5]. Generalization of some of the examples described in this paper relies on a type of doacross construct that for some kinds of loops itself requires execution time preprocessing. Details and experimental results pertaining to this *preprocessed doacross* can be found in [21]. Methods of execution time loop analysis for distributed machines that are in some respects analogous are described in [13], [10], [20]. Some numerical algorithms have been modified using schemes that reorder operations to increase available parallelism, [2], [18], [3], [7], [19].

As in [16], we define the following types of dependencies between two statements $S_1$ and $S_2$. In these definitions, we assume that $S_1$ is executed before $S_2$. A *true* or *data* dependency exists between statements $S_1$ and $S_2$, if statement $S_2$ uses a variable assigned by statement $S_1$. An *antidependency* exists between the statements when $S_1$ uses a variable that is assigned by $S_2$. An *output* dependency exists when non adjacent statements $S_1$ and $S_2$ assign the same variable.

The remainder of this paper is organized as follows: In Section 2, we present the *doconsider* loop and the transformations required to integrate this loop in a compiler front-end. A mathematical model which compares the performance of some scheduling strategies is presented in Section 3. Results from experiments conducted on the Encore Multimax are given in Section 4 and we summarize our results in Section 5.

```
1:  doconsider i=1,n
2:      y(i) = rhs(i)
3:      do j=low(i),high(i)
4:        y(i) = y(i) - a(j)*y(column(j))
5:      end do
6:   end doconsider
```

Figure 1: An annotated loop

# 2 The Doconsider loop

## 2.1 Motivation

In many programs, e.g. iterative linear system solvers, a nest of loops may execute many times. The results of a single run time analysis can be reused, as long as data structures that determine dependency patterns are not overwritten. When dependency patterns in a nest of loops do not depend on the *values* of variables or array elements computed within the loop nest, we say that the nest of loops is *start-time schedulable*. In many cases, when compile-time analysis fails, parallelization of *start-time* schedulable loops can be performed during program execution.

Determination of when data structures can be written to during program execution can be reasonably straightforward when the data structures are local to a procedure but is a rather complex task when the data structures are passed to procedures or reside within common blocks. Inter-procedural analysis is concerned with the study of such problems.

In our current implementation, the user will need to indicate that an array which determines dependency patterns has been modified, by setting a flag. This would normally entail a re-evaluation of the dependency structure. If mechanisms of the type we describe in this paper are integrated into a parallelizing compiler with effective inter-procedural analysis, it should be possible to avoid the need to set this flag. Next, we describe the transformations that produce the inspector and executor procedures from user code.

### 2.1.1 Transformations Required for the Doconsider Loop

A compiler front end transforms an annotated loop into *two* separate code segments. The first code segment, the *inspector*, reorders and partitions loop indices, the second code segment, the *executor*, carries out the scheduled work. We provide a sketch of the required transformations, using as an example a code segment that carries out a sparse triangular solve, (Figure 1). A loop of the form shown in Figure 1, may be executed many times during the running of a given program. Note that the data dependencies between the

```
1:  do  i=1,n
2:      mywf = 0
3:      do j=low(i),high(i)
4:        mywf = max(maxwfy(y(column(j))),mywf)
5:      end do
6     maxwfy(i) = mywf + 1
7:    end do
```

Figure 2: A topological sort

```
1    do i=1,nlocal
1a      isched = schedule(i)
2       y(isched) = rhs(isched)
2a      if(isched. eq. BARRIER)
2b             call barrier()
2c       else
3              do j=low(isched),high(isched)
3a             index = column(j)
4              y(sched) = y(sched) - a(j)*y(index)
5         end do
6     end do
```

Figure 3: A Prescheduled loop

elements of $y$ are determined by the values assigned to the data structure column during program execution. 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 $y(i_2)$. The example code shown in Figure 1 has been chosen for ease of explanation of the transformations we will present shortly. In practice, we can handle multiple-nested loops and much more complex right hand side expressions.

To parallelize such loops, the method we use is as follows: 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*.

A loop L in the source code is transformed into a new loop L' designed to assign a wavefront number to each iteration of L.. Since the wavefront number for each index is one plus the maximum of the wavefront numbers of the indices on which it depends, L can simply sweep sequentially through the indices and calculate the wavefront for each index. Figure 2 illustrates the code segment used to calculate the wavefronts corresponding to each index in line 1 of Figure 1. The wavefront corresponding to index $i$ is stored in `maxwfy(i)`. The wavefront information must then be used to create a schedule of outer loop indices to be executed by each processor.

The code segment in Figure 3 is a simplified version of an *executor* corresponding to Figure 1. This code segment runs on each processor and carries out a sequence of *doall* loops in which each *doall* loop is separated from the next by a synchronization barrier. The assignment of loop indices to processors is determined by the local array schedule (line 1a). This array also determines the order in which loop iterations are carried out. In addition, schedule also contains synchronization information; when an element of schedule is equal to a constant BARRIER, a global synchronization is carried out (lines 2a,2b).

While the executor in Figure 3 will produce the correct solution if Figure 1 represents a sparse triangular solve, the code segment does not take into account the possibility that in the course of computing $y(i_1)$ we might use $y(i_2)$ before it has been computed by this nest of loops. We need to make sure that $y(i_2)$ is not updated before the data stored in $y(i_2)$ is consumed. The dependence relation between $y(i_2)$ and $y(i_1)$ is an example of an antidependency [15].

One way we can deal with antidependencies is to modify the inspector (in this case the topological sort) so that the computation of wavefronts takes them into account. Alternately, we can modify the executor so that we use both old and new versions of arrays; in Figure 1 we would replace y with two arrays, yold and ynew. The executor would then read from yold and write to ynew. Note that if the executor does not modify

```
1    call barrier()
C    Begin loop nest
2    do i=1,nlocal
3       isched = schedule(i)
4       y(isched) = rhs(isched)
5             do j=low(isched),high(isched)
6             index = column(j)
7             while ( ready(index) .ne. completed) end while
8             y(sched) = y(sched) - a(j)*y(index)
9      end do
10    ready(isched) = completed
11    end do
```

Figure 4: A Self-Executing loop

all elements of y, portions of array ynew must be copied into yold.

Instead of producing an executor that carries out a sequence of *doall* loops on consecutive wavefronts of topologically sorted indices, we can compute the sorted indices using a modified doacross. We illustrate this transformation in Figure 4. The outer loop in Figure 4 goes over the indices assigned to this processor (line 2). The shared array ready is used to communicate the availability of array elements. When outer loop iteration $i$ is completed, y(schedule(i)) is available and ready(schedule(i)) is marked (line 10). In line 7 of Figure 4, a busy-wait is carried out until the array element required for the computation is available. Note that during an invocation of the nest of loops, each element of ready is written to only once (line 10) but can be read many times. This form of synchronization is particularly effective in bus-based multiprocessors with snooping caches since in such architectures the busy-waiting generates no bus traffic. Synchronization mechanisms of this kind should also be quite effective in a pipelined multiple instruction stream machines such as the (now defunct) Denelcor HEP [6], [8]. In [21] we present details of how the modified doacross used here is implemented. In many cases, the modified doacross loop itself can require a separate stage of execution time preprocessing; an outline of this preprocessing and a study of associated overheads can be found in this reference.

### 2.1.2 Efficient Calculation of the Topological Sort

The schedule of outer loop indices for each processor can be obtained by *global scheduling*, i.e. assigning indices to processors in a way that evenly partitions the work in each wavefront. For the running example in Figure 1, we would employ maxwfy from Figure 2 to produce a new array bywf that would list the indices in order of increasing wavefront

```
1   doconsider i1 =1, N1
2      do i2 = 1, N2
3         do i3 = 1, N3
              .
                .
4                do im = 1, Nm
5                   z = f(i1,i2,....im)
6                   A(z) = ik*B(i1,..im) + A(C(..))*ij + D(..)*A(E(..))
7                end do
                .
              .
8              end do
9       end do
10  end doconsider
```

Figure 5: More Complex Loops

number. After this, we would evenly partition the indices in each wavefront between the processors.

Alternately we could begin with a fixed assignment of indices to processors and use the wavefront information to simply rearrange the local ordering of indices in each processor in an attempt to increase parallelism.

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 assigning 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.

### 2.1.3 More Complex Loops

In Section 2.1.1, (Figure 1) we presented a simple version of the *doconsider* loop. We have implemented transformations that allow us to handle more complicated nested loop structures. The *doconsider* loop in Figure 5 has multiple nested do loops and a complex right hand side expression in statement 6. Currently, some restrictions are placed on the left hand side index variable z. For each index z, A(z) can be written to multiple times. However, partially computed values of A(z) cannot be used to update other elements of A. Allowing partially computed values of A on the right hand side of an assignment statement

such as 6 adds significant complexity to the topological sort. Each index element z, must be clustered with all other indices that use partially computed values of A(z) and each such cluster must be scheduled as a unit on a single processor.

The transformation system uses a source to source compiler that employs a set of data structures called the Blaze Intermediate Form or BIF graph to represent the parse tree [9]. On recognizing a *doconsider* loop, the system searches for an assignment statement of the form shown in line 6, Figure 5. For each such statement that is encountered, an inspector and an executor is generated. Each of these routines has $n$ nested do loops, where $n$ is the nesting level of the assignment statement. The executor is analogous to that shown in Figure 4.

### 2.1.4 Summary of Doconsider Transformations

We now provide a short stepwise description of the procedure which takes as input a code of the type shown in Figure 1 and restructures it into a suitable parallel version. Steps 1 and 2 are performed at compile-time, while steps 3 and 4 are performed at run-time. Notice that an outer loop iteration can depend upon several other iterations, as is seen in (statement 4) of the code in Figure 1.

1. From the original loop, the code for the inspector is generated by the compiler. *During program execution* this code determines the wavefront number of each index.
2. The loop in Figure 1 is transformed into a self-executing or prescheduled executor.
3. Immediately prior to the actual execution of the loop, the inspector computes the wavefront numbers and the indices are sorted on the basis of these wavefronts.
4. The actual computation is now performed by each processor on its assigned subset of indices, using one of the executors that have been generated, in step 2.

In the next section, we describe our model problems and present mathematical expressions that illustrate the load balance obtainable with the different executors.

# 3 Description and Analysis of Model Problems

## 3.1 Model Problems

We now present analytic results obtained through the study of two model problems. In the following section, we will present empirical results from experiments on a shared memory multiprocessor. The loop that we use to evaluate the different scheduling and

synchronization strategies is depicted in Figure 1. The patterns of dependency between loop iterations are controlled by the integer array `column`. By varying the definition of `column` we are able to explore the performance achieved by our different strategies over a wide range of dependency patterns.

Many of the sparse triangular systems that we use as model problems arise from incompletely factored matrices that were obtained from a variety of discretized partial differential equations. The solution of these sparse triangular systems accounts for a large fraction of the sequential execution time of linear solvers that use Krylov methods[3]. The dependencies encountered in solving these systems inhibit the parallelization of the outer loop of row substitutions (line 1, Figure 1). A full description of the structure of the triangular systems used in our experiments is found in [3], a brief definition of the problems may be found in the appendix. We also employ a simple parameterized workload that will be used to illustrate the tradeoffs between use of prescheduled and self-executing doconsider loops. This workload generator functions by initializing data structure `column` used in the loops depicted in Figure 1.

A dependency graph is generated as follows: We begin with a domain consisting of a 2-dimensional $n \times m$ mesh of points whose connections have yet to be established. Each point $(i, j)$ in the mesh is assigned an index number $i \times n + j$. We generate graphs in which the point $(i, j)$ is linked to all points $(q, r)$ located within a manhattan metric distance $D$; i.e. where $\mid i - q \mid + \mid j - r \mid \leq D$. The array `column` is initialized so that it represents the dependencies between the indices in a generated graph. This workload generator generates a dependency graph in which both dependencies and antidependencies occur because the value assigned to `column(j)` can be either less than or greater than $j$.

## 3.2 Analysis of a Model Problem

We use two model problems to compare the quality of load balance that can be obtained using the prescheduled and the self-executing doconsider loops. We consider the first model problem in some detail, this problem involves the solution of 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 $n$ by $m$ domain and $p \leq \min(m, n)$ processors. This problem is equivalent to solving the recurrence equation

$$z_{i,j} = rhs_{i,j} - f_{i-1,j} \times z_{i-1,j} - g_{i,j-1} \times z_{i,j-1} \tag{1}$$

for $1 \leq i \leq m$ and $1 \leq j \leq n$, where $f_{0,j} = g_{0,j} = f_{i,0} = g_{i,0} = 0$. We can choose `column`, `low`, `high` and a in Figure 1 so that $y(i \times n + j)$ in that loop nest is equal to $z_{i,j}$ in Equation 1.

We assume that all computations required to solve the problem would require time $S$ on a single processor, and that computation of each iteration of the outer loop (statement 1) in Figure 1 takes time $T_{iter} = S/(mn)$ for this model problem. This ignores the

relatively minor disparities caused by the matrix rows represented by indices on the lower and the left boundary of the domain. For each type of loop, we will now calculate an estimate $E_{lb}$ of the efficiency that would be achieved if the only source of inefficiency were due to imperfect load balance. At the end of this section, we will briefly consider another model problem. This problem consists of solving an $n$ by $n$ dense triangular matrix with unit diagonals.

### 3.2.1 Load Balance with Prescheduled Doconsider

To understand the relative performance of the two types of doconsider loops for the first model problem, we need to specify the mapping of indices onto the processors. We will describe this mapping using the recurrence equation problem formulation in Equation 1. The global topological sort produces a list of indices sorted by wavefront. When we define the arrays `low`, `high`, `column` and `a` in Figure 1 so that the solutions obtained are equivalent to those in Equation 1, the topological sort produces a list L that corresponds to arranging the indices in each wavefront in order of increasing index number. The indices in L are assigned in a wrapped manner, i.e. the $k$th index in wavefront $w$ is assigned to processor $k$ mod $p$.

When prescheduling is used, the computation is divided into wavefronts separated by global synchronizations. Define $MC(j)$ as the maximum number of indices computed by any processor during wavefront $j$. It is clear that $n+m-1$ wavefronts are required to complete the computation. The computation time required to complete wavefront $j$ is equal to $T_{iter}MC(j)$. The computation time required to complete the problem is consequently

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

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

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

The computational time needed to solve the problem $T_C$ is given by

$$T_C = T_{iter} \sum_{j=1}^{n+m-1} MC(j)$$
$$= \frac{S}{mn} \sum_{j=1}^{m+n-1} \left\lceil \frac{min(j, m, n, m+n-j)}{p} \right\rceil$$

By assumption, the sequential time to solve the problem is $S = mnT_{iter}$. The estimated efficiency $E_{lb}$, if load imbalance were the only source of inefficiency would be $\frac{S}{pT_C}$.

### 3.2.2 Load Balance: Self-Executing Doconsider

Much of the load imbalance we saw in the prescheduled doconsider loop can be corrected. During any wavefront $j \leq \min(m, n) - 1$, where $j$ is not a multiple of $p$, there are $p - j \ mod \ p$ idle processors. When $j$ is a multiple of $p$, no processors are idle. The failure to balance is essentially an end-effect; e.g., a wavefront has $p + 1$ work units with equal computational demands, but only $p$ processors are available. In [14] 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 concurrently compute row substitutions in consecutive wavefronts provided that we observe the dependencies. This is taken care of naturally since the busy-wait synchronization mechanism ensures that dependencies are in fact observed.

We can derive an expression for $E_{lb}$ 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. To see this, assume that solution values are available for indices in list L through the index that corresponds to wavefront $w$, $i = C$, (i.e. $i = C, j = w - C + 1$). All indices in L up to the index corresponding to wavefront $w + 1$, $i = C$ (i.e. $i = C$ , $j = w - C + 2$) will have their dependencies satisfied, and can be calculated simultaneously. The cumulative processor idle time is consequently $p(p - 1)$. $E_{lb}$ is thus given by

$$\frac{mn}{mn + p(p-1)} \tag{2}$$

### 3.2.3 Load Balance: Doacross Loop

We now derive an expression $E_{lb}$ when a doacross loop is used to solve the problem. In a doacross loop, each index $k$ of y is assigned to processor $k$ mod $p$; this corresponds to assigning $z_{i,j}$ to processor $(i \times n + j)$ mod $p$. In this derivation, we assume that the time required to multiply and subtract the terms $f_{i-1,j} \times z_{i-1,j}$ and $g_{i,j-1} \times z_{i,j-1}$ is equal to half the time $T_{iter}$ required to perform a loop iteration.

The processor $q$ designated to compute $z_{i,k}$ is the same as the processor designated to compute $z_{i-1,n-p+1-k}$, $1 \leq k \leq p$. As soon as a processor $q$ finishes $z_{i-1,n-p+k}$, it can begin to work on $z_{i,k}$. For $p \geq 3$, a pipeline effect is created so that only $T_{iter}/2$ time is needed to calculate each $z_{i,j}$. Let $t_{i,j}$ represent the time at which $z_{i,j}$ is computed. It is convenient to assume that $t_{0,j} = j/2$, and straightforward to establish that $t_{i,0} = t_{i-1,n-p+3}$. Because of the pipeline mentioned above, $t_{i,j} = t_{i,j+1} + 1/2$. From this observation, we can deduce that $t_{n-p+2,m} = \frac{m(n-p+2)}{2}$, and hence that $t_{n,m} = \frac{m(n-p+2)}{2} + \frac{p-2}{2}$. Thus for the doacross, $E_{lb}$ is equal to

$$\frac{2mn}{p((n-p+2)m+p-2)}$$

Table 1 depicts the values of $E_{lb}$ for the model problem for domains of varying dimensions. We note that for values of $m$ and $n$ that are much larger than $p$, both the prescheduled and self-executing doconsider will have a perfect load balance. The doacross loop on the other hand exhibits an efficiency that decreases with increasing numbers of processors with $E_{lb}$ equal to $2/p$. On the other hand, if $m = p+1$, and $n$ is large relative to $m$, the self-executing doconsider achieves an $E_{lb}$ of 1.0 while the $E_{lb}$ in the prescheduled doconsider case is only $1/2 + 1/2p$. In the prescheduled doconsider, during most computational wavefronts, one processor has to perform twice as much work as all the rest. The global synchronizations required for the prescheduled doconsider force all processors to wait for the busiest processor to finish. Because the doacross loop does not perform index set reordering, we see a marked efficiency difference between the case $m = p+1$, $n$ large and $n = p+1$, $m$ large. For the $n = p+1$ case, $E_{lb}$ is equal to $\frac{2}{3} + \frac{2}{3p}$ , while for $m = p+1$, $E_{lb}$ is equal to $2/p$. We obtain a better load balance using the doacross than we do using the prescheduled doconsider when $n$ is equal to $p+1$, and $m$ is large. When both $m$ and $n$ are equal to $p$, we obtain a better load balance using the doacross than we do with *either* type of doconsider.

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 prescheduled 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

Table 1: Model Problem Load Balance Efficiencies

| Domain Dimensions | Prescheduled | Self-Executing | Doacross |
|---|---|---|---|
| m=n, both large | 1 | 1 | 2/p |
| m=n=p | p/(2p-1) | p/(2p-1) | 2p/(3p-2) |
| m=p, n large | 1 | 1 | 2/p |
| m=p+1, n large | 1/2 + 1/(2p) | 1 | 2/p |
| n=p, m large | 1 | 1 | 1 |
| n=p+1, m large | 1/2 + 1/(2p) | 1 | 2/3 + 2/(3p) |
| n by n full triangular system p = n-1 | 1/p | (p+1)/(2p) | (p+1)/(2p) |

by global synchronizations; each row substitution forms its own wavefront. The sequential computation time and the prescheduled computation time are both $T_{saxpy}\frac{n(n-1)}{2}$. Calculated only on the basis of load balance, both the self-executing efficiency and the doacross efficiencies $E_{lb}$ are $n/2(n-1)$ while the prescheduled $E_{lb}$ is $1/(n-1)$.

# 4 Experimental Results

In this section, we provide experimental results for the performance of the inspectors and executors described in Section 2. The experiments described in this section also highlight the overheads associated with the inspectors and the synchronization mechanisms employed by the executors. The following timings were done on an Encore Multimax/320 with 13 megahertz APC/02 boards and version 2.1 of the FORTRAN compiler. Parallel efficiency is defined as the ratio between the time required to solve a problem using an optimized sequential version and the product of the time required on the same problem by the multiprocessor code multiplied by the number of processors.

The inspectors and executors used in our experiments were coded by hand (the experiments discussed here were performed to help us decide precisely how the *doconsider* transformations should be implemented). The executors used were similar to those depicted in Figures 3 and 4. Antidependencies were handled by writing to a new version of array y and by evaluating a conditional to determine whether references to array y should refer to the old or the new version of y (this was described in Section 2.1.1).

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

| Test Problem | Prescheduled Time | Self-Executing Time | Sort Time | Doacross Time | Sequential Time |
|---|---|---|---|---|---|
| SPE2 | 33 | 21 | 49 | 34 | 223 |
| SPE5 | 29 | 23 | 100 | 45 | 241 |
| 5-PT | 31 | 19 | 72 | 37 | 192 |
| 7-PT | 56 | 56 | 204 | 84 | 615 |
| 9-PT | 80 | 58 | 154 | 98 | 698 |

## 4.1 Multiprocessor Timings

### 4.1.1 Where Does the Time Go

In Table 2 we present multiprocessor timings on 16 processors for lower triangular solves arising from the incompletely factored test problem matrices. In this table we present the time required to solve the triangular systems using self-executing and prescheduled executors. The schedule of loop iterations was produced using global scheduling. The execution time required by the prescheduled executor was consistently either greater than or roughly equal to the time required by the self-executing code. The same pattern of results was observed for a wide variety of test matrices [22]. Table 2 also depicts the the results of timings of an optimized sequential version of the program for each of the lower triangular systems. For most of the problems tested, the parallel efficiencies obtained were between 50 and 70 %. This table also depicts the time required to carry out a parallelized global scheduling, this time ranged from 20 to 60 % of the sequential execution time. Finally in Table 2 we depict the time required to carry out a *doacross* version of these triangular solves. We see that the *doacross* loop is consistently less efficient than either the prescheduled or the self-executing loops. For example, the self-executing form of the executor for the SPE5 problem required 23 milliseconds, the prescheduled executor required 29 milliseconds, while the *doacross* version of the solve required 45 milliseconds.

Recall that the self-executing loop is essentially a busy-wait *doacross* loop with a reordered index set. We consequently expect that the self-executing executor will exhibit more concurrency than the busy-wait *doacross* 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 prescheduled executor performs work corresponding to only one wavefront at a time. We fully expect the busy-wait *doacross* loop to out perform the prescheduled executor in some problems, (e.g. a dense lower triangular solve).

We performed an operation-count based analysis of the parallelism that could be obtained given a particular assignment of indices to processors when either type of executor was employed using global scheduling. The analysis made the assumption that the load balance could be characterized solely by the distribution and scheduling of the floating

Table 3: Operation Count Efficiencies and Performance Predictions

| Test Problem | P.S. Op. Count Efficiency | S.E. Op. Count Efficiency | P.S. Time | P.S. Predict Time | S.E. Time | S.E. Predict Time |
|---|---|---|---|---|---|---|
| SPE2 | 0.52 | 0.89 | 33 | 33 | 21 | 20 |
| SPE5 | 0.70 | 0.96 | 29 | 30 | 23 | 22 |
| 5-PT | 0.61 | 0.95 | 31 | 31 | 19 | 18 |
| 7-PT | 0.94 | 0.98 | 56 | 56 | 56 | 57 |
| 9-PT | 0.78 | 0.97 | 80 | 84 | 58 | 58 |

point operations. The efficiency estimated in this manner was computed for several test problems on 16 processors and is depicted in Table 3 as operation count efficiency. From this analysis we were able to quantify the deleterious effects of prescheduling on load balance. To assure ourselves of the meaningfulness of the operation count efficiency metric, we used operation count efficiencies along with estimates of other overheads to predict parallel execution times. The methods for estimating other overheads were discussed in some detail in [23]. We accounted for extra operations that had to be executed by the parallel code, costs for logical and arithmetic operations required for synchronization as well as contention for resources such as shared memory and bus access. We note from Table 3 that self-executing methods yield superior operation count efficiencies for some problems and that we are able to predict parallel execution times to a reasonable degree of accuracy.

### 4.1.2 Results from synthetic workloads

In Figure 6, we compare the parallel efficiencies on 16 processors of the Encore Multimax obtained using the doacross and prescheduled and self-executing doconsider executors. We choose domain size $n = m = 25$ and vary the parameter $D$.

Recall that $D = 1$ produces a pattern of true dependencies identical to that in the model problem described in Section 3.2. As $D$ increases, we obtain increasing numbers of true dependencies. In the limit of $D = 25$, we obtain a fully connected graph in which every loop iteration depends on all earlier loop iterations. For $D = 2$ and above the prescheduled executor exhibits monotonically decreasing parallel efficiencies. This decrease is to be expected since as $D$ increases, there is decreasing amounts of parallelism for the prescheduled executor to exploit. When $D = 1$, the computation is small and the sequential time for this problem was only 64 milliseconds, while for $D = 12$ the sequential computation time was 2734 milliseconds. The relative contribution of synchronization costs and other sources of overhead play an increasing role for small values of $D$. We will examine in detail the effects of non-load balance related overheads in other experiments to be presented later in this section.

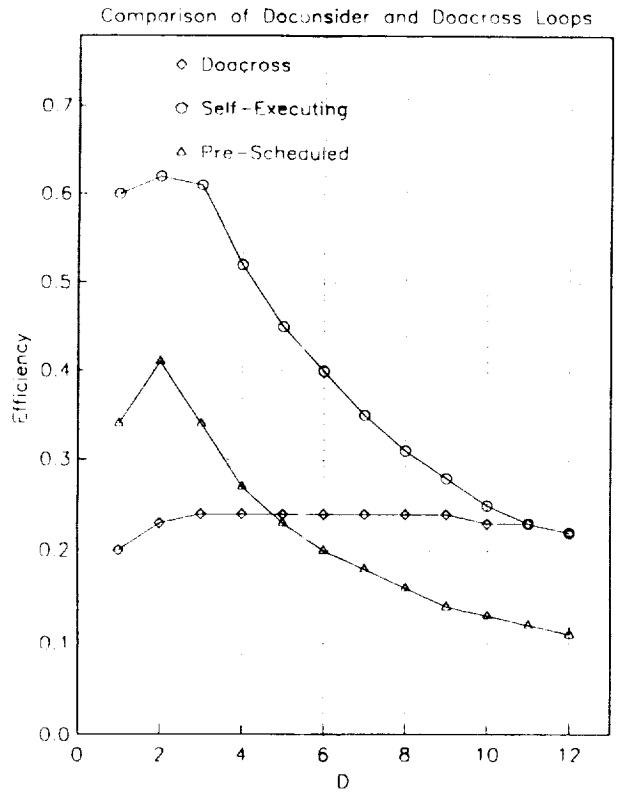


Figure 6: Results from Synthetic Workload

The self-executing doconsider executor exhibited substantially higher parallel efficiencies than the prescheduled doconsider for all values of $D$. For $D = 2$, the parallel efficiency for the self-executing doconsider was 0.62 while for the prescheduled doconsider the parallel efficiency was 0.41. For larger values of $D$ the parallel efficiency of the self-executing doconsider declines; but for $D = 5$ and above self-executing efficiencies are consistently approximately twice prescheduled efficiencies. The efficiency obtained using the doacross executor for $D \geq 2$ varied in a narrow range between 0.22 and 0.24. For values of $D$ above 11, the efficiencies obtained by using the doacross executor did not vary significantly from those obtained using the self-executing doconsider executor.

### 4.1.3 Effects of Local Reordering

We are interested in evaluating the role played by the synchronization mechanism in determining performance, when indices are not repartitioned after a topological sort. We compared the estimated efficiency of the same partition and schedule using global synchronization and self-executing synchronization in a matrix. Indices were assigned to processors in a wrapped 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 wavefront in order of increasing index number. From Figure 7, we can see that the results obtained through the use of global synchronization can vary wildly with

Table 4: Global versus local sorting (self-executing)

| Test Problem | Self-Executing Global Sort | Self-Executing Local Sort | Global Sort Time | Local Sort Time | Sequential Time |
|---|---|---|---|---|---|
| SPE2 | 21 | 30 | 49 | 30 | 223 |
| SPE5 | 23 | 24 | 100 | 46 | 241 |
| 5-PT | 24 | 28 | 72 | 39 | 192 |
| 7-PT | 56 | 55 | 204 | 78 | 615 |
| 9-PT | 58 | 63 | 154 | 101 | 698 |

the number of processors used. Often, many, if not all of the indices in a wavefront get assigned to a single processor, resulting in sequential execution for that wavefront.

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. Prescheduling on the other hand, appears to be much less robust.

### 4.1.4 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 Section 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 wrapped manner. For the *local* sorting method, the initial partition of indices is maintained, but their ordering is changed based upon wavefront numbers. In [22] 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. 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 241 ms. Because we pay for the sorting only once, subsequent iterations of the code will show a great advantage for the parallel code (23 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.

Thus, we conclude that local index set scheduling overhead does turn out to be much less than global index set scheduling overhead, as is to be expected. Global scheduling frequently but not invariably leads to slightly lower execution times than did local scheduling. For example, in the case of SPE2, the global run time was 21 ms and the local execution time was 30 ms but for SPE3 (not shown in Table). Global scheduling gave a run time of 25 ms while local scheduling required 22 ms.

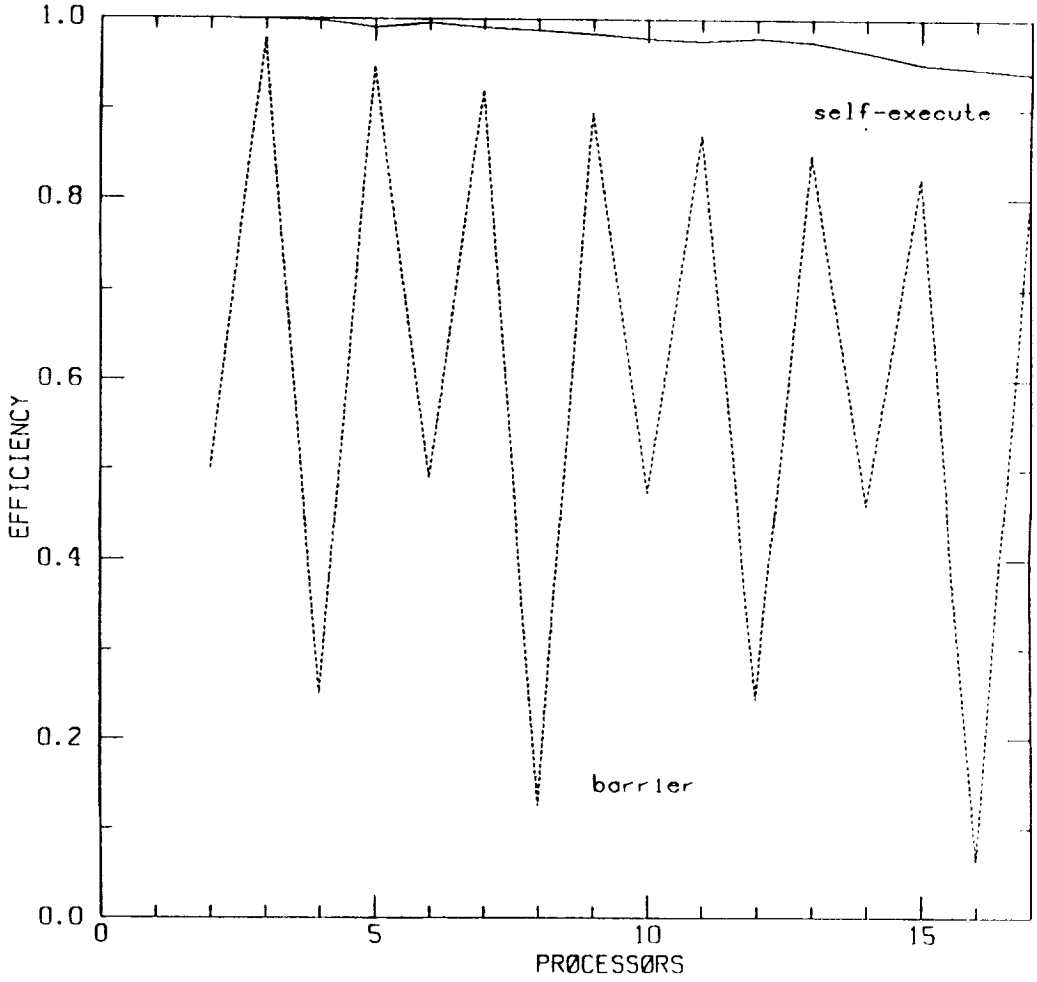


Figure 7: Effect Of Local Ordering

# 5 Conclusions

When the data dependencies of the problem are known at compile-time, detection of parallelism and task decomposition can automatically be performed by the compiler. However, there are problems where the parallelism cannot be fully characterized during compilation due to data dependencies that become manifest during run-time. In this paper, we presented the *doconsider* construct and transformation rules which would allow compilers to effectively parallelize such problems.

In this paper, we have reached the conclusion that for the types of workloads we have investigated, loop iteration reordering can have a positive impact on performance. The doacross related self-execution method almost always gives better results than does 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 8, which pictorially summarizes the findings reported in this paper.

# 6 Appendix: Definition of Test Triangular Systems

The the triangular systems refered to in Section 3.1 were derived from the following partial differential equation descretizations:

SPE2 This problem arises from the thermal simulation of a steam injection processes. The grid is 6x6x5 with 6 unknowns per grid point, this yields a system with 1080 equations. The matrix is a block seven point operator with 6x6 blocks.

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 16x23x3 grid with 3x3 blocks yielding 3312 equations.

5-PT The problem is a five point central difference discretization on a 63 x 63 grid; this yields a system with 3969 equations.

7-PT The problem is a seven point central difference discretization on a 20 x 20 x 20 grid; this yields a system with 8000 equations.

9-PT The problem is a nine point box scheme discretization on a 63 x 63 grid; this yields a system with 3969 equations.

**Acknowledgements:** The authors would like to thank Dennis Gannon and Piyush Mehrotra for supplying us with a Fortran parser linked to Blaze Intermediate Form data structures. We would also like to thank Mike Wolfe, Doug Baxter, Martin Schultz and

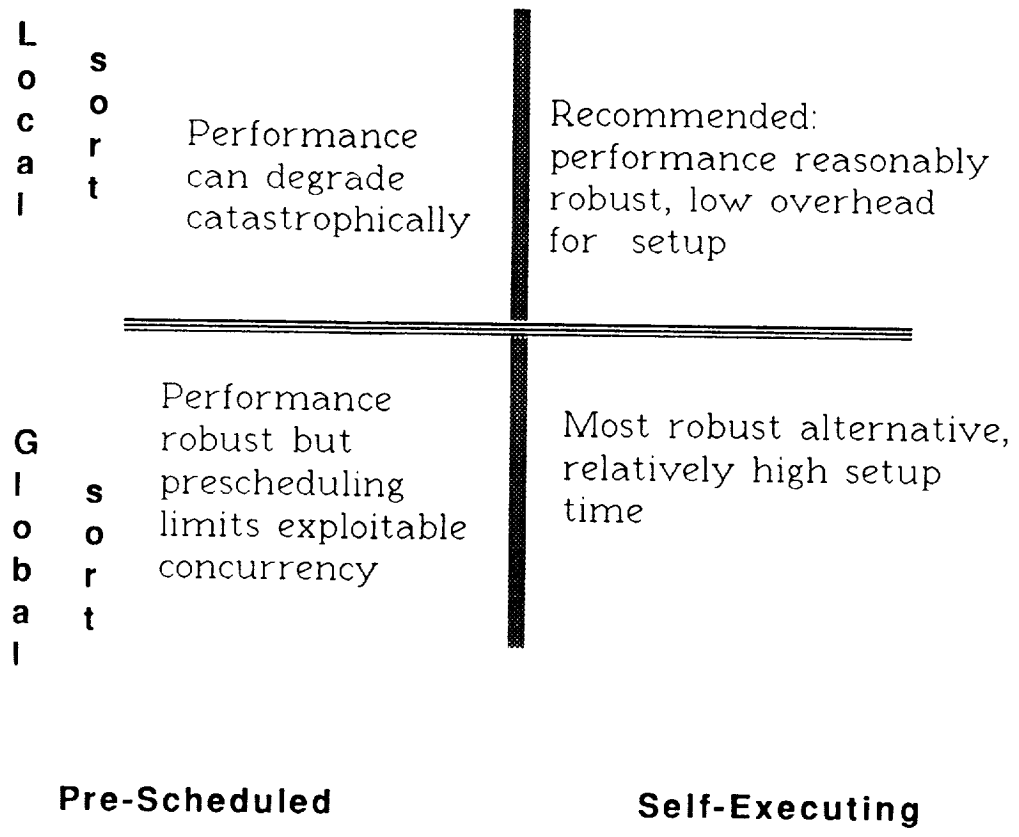


Figure 8: Summary of Results

Stan Eisenstat for helpful discussions and Bob Voigt for his careful editing of various versions of this manuscript.

# References

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

[2] E. Anderson. Solving sparse triangular linear systems on parallel computers. Report 794, UIUC, June 1988.

[3] 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*, pages 1698,1711, January 1988.

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

[5] 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*, pages 238,253, July 1988.

[6] M. C. Gilliland and Burton J. Smith. Hep: a semaphore-synchronized multiprocessor with central control. In *Proc. 1976 Summer Computer Simulation Conf.*, pages 57–62, July 1976.

[7] A. Greenbaum. Solving sparse triangular linear systems using fortran with paralllel extensions on the nyu ultracomputer prototype. Report 99, NYU Ultracomputer Note, April 1986.

[8] Harry F. Jordan. Performance measurements on hep, a pipelined mimd computer. In *Proceedings of the 10th Annual International Symposium on Computer Architecture, SIGARCH Newsletter*, volume 11, pages 207–212, 1983.

[9] C. Koelbel. The BIF data structures user's manual. in preparation, Purdue University, West Lafayette, IN, 1987.

[10] C. Koelbel, P. Mehrotra, and J. Van Rosendale. Supporting shared data structures on distributed memory architectures (to appear in 2nd ACM SIGPLAN Symposium on Principles Practice of Parallel Programming, March 1990),. Report 90-7, ICASE, January 1990.

[11] 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.

[12] Ewing L. Lusk and Ross A. Overbeek. A minimalist approach to portable, parallel programming. In *The Characteristics of Parallel Algorithms, Leah Jamieson, Dennis Gannon, and Robert Douglass, Editors*, pages 351–362, Cambridge Mass, 1987. MIT Press.

[13] R. Mirchandaney, J. H. Saltz, R. M. Smith, D. M. Nicol, and Kay Crowley. Principles of runtime support for parallel processors. In *Proceedings of the 1988 ACM International Conference on Supercomputing , St. Malo France*, pages 140–152, July 1988.

[14] D. M. Nicol and J. H. Saltz. Delay point schedules for irregular parallel computations. *International Journal of Parallel Programming*, 18(1), Feb 1989.

[15] 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.

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

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

[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] J. Saltz. Aggregation methods for solving sparse triangular systems on multiprocessors. *SIAM J. Sci. and Stat. Computation.*, 11(1):123–144, 1990.

[20] J. Saltz, K. Crowley, R. Mirchandaney, and Harry Berryman. Run-time scheduling and execution of loops on message passing machines, (to appear in Journal Parallel and Distributed Computing, April 1990). Report 89-7, ICASE, January 1989.

[21] J. Saltz and R. Mirchandaney. The preprocessed doacross loop. Report 90-11, ICASE Interim Report, 1990.

[22] J. Saltz, R. Mirchandaney, and D. Baxter. Run-time parallelization and scheduling of loops. Report 88-70, ICASE, December 1988.

[23] J. Saltz, R. Mirchandaney, and D. Baxter. Runtime parallelization and scheduling of loops. In *The Proceedings of the Symposium of Parallel Algorithms and Architectures, Santa Fe, NM*, June 1989.

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

[25] Michael Wolfe. *Optimizing Supercompilers for Supercomputers.* The MIT Press, Cambridge Mass, 1989.

NASA National Aeronautics and Space Administration

# Report Documentation Page

| 1. Report No. | 2. Government Accession No. | 3. Recipient's Catalog No. |
|---|---|---|
| NASA CR-182039<br>ICASE Report No. 90-34 | | |

| 4. Title and Subtitle | 5. Report Date |
|---|---|
| RUN-TIME PARALLELIZATION AND SCHEDULING OF LOOPS | May 1990 |
| | 6. Performing Organization Code |

| 7. Author(s) | 8. Performing Organization Report No. |
|---|---|
| Joel H. Saltz<br>Ravi Mirchandaney<br>Kay Crowley | 90-34 |
| | 10. Work Unit No. |
| | 505-90-21-01 |

| 9. Performing Organization Name and Address | 11. Contract or Grant No. |
|---|---|
| Institute for Computer Applications in Science and Engineering<br>Mail Stop 132C, NASA Langley Research Center<br>Hampton, VA 23665-5225 | NAS1-18605 |
| | 13. Type of Report and Period Covered |
| | Contractor Report |

| 12. Sponsoring Agency Name and Address | 14. Sponsoring Agency Code |
|---|---|
| National Aeronautics and Space Administration<br>Langley Research Center<br>Hampton, VA 23665-5225 | |

| 15. Supplementary Notes | |
|---|---|
| Langley Technical Monitor:<br>Richard W. Barnwell<br><br>Final Report | Revised version of 88-70, submitted to Transactions on Computers; Excerpts in Proc. of the Third Int. Conf. on Supercomputing, Crete Greece, June 1989 and in Proc. of the First Int. Symposium on Parallel Algorithms and Architectures, Santa Fe, NM. |


16. Abstract

In this paper, we study run-time methods to automatically parallelize and schedule iterations of a do loop in certain cases, where compile-time information is inadequate. The methods we present in this paper involve execution time preprocessing of the loop. At compile-time, these methods set up the framework for performing a loop dependency analysis. At run-time, wave fronts of concurrently executable loop iterations are identified. Using this wavefront information, loop iterations are reordered for increased parallelism.

We utilize sybolic transformation rules to produde: (1) inspector procedures that perform execution time preprocessing and (2) executors or transformed versions of source code loop structures. These transformed loop structures carry out the calculations planned in the inspector procedures. We present performance results from experiments conducted on the Encore Multmax. These results illustrate that run-time reordering of loop indices can have a significant impact on performance. Furthermore, the overheads associated with this type of reordering are amortized when the loop is executed several times with the same dependency structure.


| 17. Key Words (Suggested by Author(s)) | 18. Distribution Statement |
|---|---|
| loop parallelization, shared memory, wavefront, level set, inspector, executor | 59 - Mathematical and Computer Sciences (General)<br>61 - Computer Programming and Software<br>Unclassified - Unlimited |

| 19. Security Classif. (of this report) | 20. Security Classif. (of this page) | 21. No. of pages | 22. Price |
|---|---|---|---|
| Unclassified | Unclassified | 24 | A03 |

NASA FORM 1626 OCT 86