TR 661

Run-Time Parallelization and Scheduling of Loops *

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

January 23, 1989 661

1 Introduction

There exist many problems in which substantial parallelism is available but where the parallelism cannot be exploited using *doall* or *doacross* loops [12] [5]. *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, i.e., *doconsider*. 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.

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

*This work was supported by the U.S. Office of Naval Research under Grant N00014-86-K-0310 and NASA grants NAS1-18107 and NAS1-18605

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 [3]) used to solve a range of linear systems, a synthetic workload is also employed. 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 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 extended abstract.

The rest of this extended abstract 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 or language extensions. 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. Finally, we summarize our findings in Section 6.

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[13]. 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 [10], we present self-execution primitives that aid greatly in the on-the-fly detection of parallelism in such problems. In this present extended abstract, we will only be concerned with *start-time* schedulable problems.

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.

Figure 1: Summary of Results

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

Figure 2: 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 3: A Self-Executing loop

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. A loop of the form shown in Figure 2, may be executed many times during the running of a given program. 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 $x(i_2)$. 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.

```
S1:  doconsider 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 4: A nested loop (Triangular Solve)

The main loop in Figure 3 corresponds to the indices assigned to this processor (line 1). The key point in Figure 3 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 the case of the pre-scheduled code, 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 global synchronization is performed before going on to the work in the subsequent phase.

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. 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. The array used to store the wavefront numbers must then be sorted to generate a schedule.

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. We now provide a short stepwise description of the automated procedure which takes as input a code of the type shown in Figure 4 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 determines the wavefront number of each index.

3. The loop in Figure 4 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 [9] 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 [14] 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[18] 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[8] 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[5] 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 [12], [13], [1], [6]. 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, [2], [16], [4], [7], [15].

4 Description and Analysis of Model Problems

4.1 Model Problems

In our experiments, many of our model problems come from the solutions of sparse linear systems arising from a variety of partial differential equations using preconditioned Krylov methods. A description of the problems solved are found in [17]. The solution of 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 4). Typically the number of non-zero elements in a row is too small to allow efficient parallelization of the inner loop (S2 in Figure 4).

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 [17], a much more detailed account of the PCGPAK results is presented in [3]. 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.

4.2 Analysis of a Model Problem

In [17], we presented the analysis of a model problem to illustrate the performance difference between using pre-scheduing and self-execution. In that analysis we estimated 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. The pattern of dependencies characterizing that computation correspond to those found in the recursion relation: $z_{i,j} = a_{i,j}z_{i,j-1} + b_{i,j}z_{i-1,j}$. We used a m by n domain and $p \leq \min(m,n)$ processors. We explicitly took 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. In the following paragraphs, we only present the final results of our analysis. For further details on the analysis, the reader is referred to [17].

For large n and $m = p + 1$, we expect to find that slightly under half of the processors are idle due to load imbalance. In the limit of large m, the ratio of the time to solve the problem using pre-scheduling versus self-execution is

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

The quantities R_{synch}, R_{inc} and R_{check} represent the ratios of the various overhead times to the computation time. For example, $R_{synch} = T_{synch}/T_p$, where T_{synch} is the time for one global synchronization and T_p is the time to compute one index. R_{inc} and R_{check} represent overheads arising from the operations in the self-executing case depicted in lines 3a and 3c in Figure 3. 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

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

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.

5 Experimental Results

The following timings were done on an Encore Multimax/320 with 13 megahertz APC/02 boards and version 2.1 of the FORTRAN compiler.

5.1 Multiprocessor Timings

5.1.1 Pre-scheduled vs. self-execution

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 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.

Two versions of parallel PCGPAK, a Krylov space solver [3], 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. In [17], we presented the 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.

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. 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.

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 1 and 2 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 Sec-

tion 5.1.1. This is not surprising since the symbolically estimated efficiencies do not take into account a number of important sources of overhead.

In Table 1 and 2 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 1 and 2).

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 1 and 2 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 of floating point calculations. In effect, we are assuming that the effect of the extra operations required in the multiprocessor program could be explained by simply adding a fixed overhead to each floating point operation.

Contention for resources such as shared memory and bus access can cause inefficiencies that are not accounted for by the above estimates. We ran a version of the multiprocessor code designed to simulate the memory and communications access patterns of the actual program. This version of the code is designed to have a perfect load balance. When executed on P processors, this program executes the schedules a total of P times. Each processor ends up executing the schedules assigned to all processors so that each processor ends up computing the work associated with all of the indices in the problem. The time required for this program to complete is called the *rotating processor* time because each processor takes on the work assigned to each other processor with control being shifted in a rotating fashion. No synchronization takes place in this version of the codes. 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 Table 1 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 2) 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.

Table 1: 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 2: 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

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 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 3 would be obtained.

The estimate of non load balance related loss (*Best* in table 3) 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 extended abstract but some discussion of that issue can be found in [11].

In Table 3, 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 3 reflect the varying relative costs of global synchronizations and array writes in problems with different structures, this issue was discussed in Section 5.1.2.

Table 3: 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

5.1.4 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 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. In [17], we saw that the results obtained through the use of global synchronization can vary wildly with the number of processors used. Often, many, if not all the indices in a phase get assigned to a single processor, resulting in sequential execution for that phase.

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.

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 [17] 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 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.

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. 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

In this research, 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. 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. Thus, we are left with a 2-dimensional solution space, as depicted in Figure 1, which pictorially summarizes the findings reported here.

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] 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, and S. Eisenstat. *Preconditioned Krylov solvers and methods for runtime loop paralleization.* Technical Report 655, Yale University, 1988.

[4] 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.

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

[6] 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.

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

[8] 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.

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

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

[11] 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.

[12] 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.

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

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

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

[16] 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.

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

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

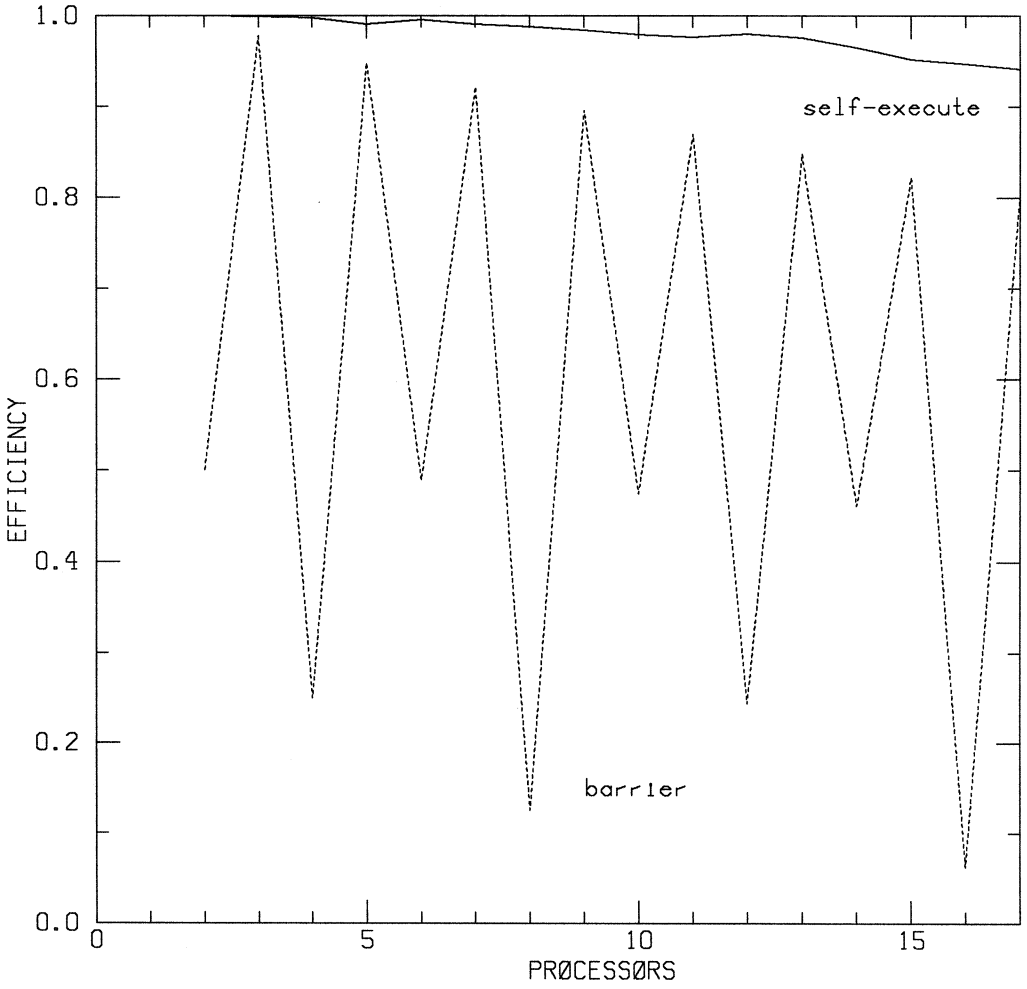

Figure 12: Effect Of Local Ordering