# An Empirical Comparison of Loop Scheduling Algorithms on a Shared Memory Multiprocessor

Matthias A. Blumrich* Carrie J. Brownhill† Kai Li* Alexandru Nicolau†

## Abstract

This paper studies several methods of instruction level parallelization applied at the statement level on a shared memory multiprocessor, and reports the results of an empirical evaluation to determine which of the methods yields the best results. Using sequential code as a base case, we compared doacross, list scheduling, greedy software pipelining (a variant of perfect pipelining), and top down scheduling. The experiments were performed on loops both with and without loop carried dependencies. We find that statement level parallelism does yield speedups on the shared memory multiprocessor. In addition, we observed an interesting superlinearity effect for fully vectorized loops.

## 1 Introduction

The purpose of this study was to determine whether instruction level parallelism is a viable method for obtaining speedups on a multiprocessor and if so, which methods yield the best results. To answer these questions, we empirically compared various popular fine grain loop scheduling algorithms on an shared memory multiprocessor. Most of these algorithms were developed for synchronous architectures with relatively low synchronization times such as Very Long Instruction Word (VLIW) machines, but have been adapted to take into account the asynchronicity and higher cost of synchronization on multiprocessors. The data presented should be interpreted as a *relative* comparison between methods, using the serial time as a base case. We did not have the luxury of implementing a full optimizing compiler for each method which probably would have improved the absolute execution times considerably.

The comparisons were performed on a DEC Firefly, a shared memory multiprocessor workstation. The configuration we used consists of six CVAX processors sharing a global memory of 32 Mbytes on a common bus. Each processor has a 64 Kbyte direct mapped snooping data cache and a 1 Kbyte direct mapped instruction cache. A complete description of the Firefly architecture and system software can be found in [14].

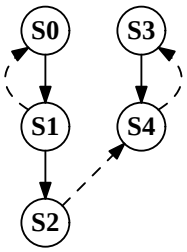
In addition to sequential code, the scheduling methods we compared were doacross, list scheduling, greedy software pipelining, and top down scheduling. For each method, we generated several schedules for several loops, varying the number of processors and the degree of loop unrolling. This produced an enormous amount of code, so a decision was made early on to manipulate instructions at a high level in C rather than assembly language. That is, we consider a C statement to be the atomic scheduling unit and will henceforth use the term "statement level parallelism" instead of instruction level parallelism.

Our suite of benchmarks consists of four inner loops from scientific code, chosen to exhibit various behavior in the schedulers. All of these loops have only one basic block as the loop body, and loop carried dependencies (LCDs) span only one iteration. This seemingly limited subset of loops is actually representative of "real" scientific code. It is an interesting subject to study because such fine grain parallelizing algorithms are relatively easy to implement for these types of loops and can yield significant speedups, as will be shown. Two of the loops we scheduled were modified to allow for the extraction of parallelism as will be described below.

Section 2 provides a description of each of the scheduling algorithms in turn, pointing out the specifics of our implementations. Section 3 is a description of the experiments we performed and a discussion of the results. Finally, section 4 concludes.

*Department of Computer Science, Princeton University, Princeton, New Jersey 08544

†Department of Computer Science, University of California at Irvine, Irvine, California 92717




```
for(i=1; i<=N; i=i+1) {
   A[i] = B[i-1];       /* S0 */
   B[i] = A[i];         /* S1 */
   C[i] = B[i];         /* S2 */
   D[i] = E[i-1];       /* S3 */
   E[i] = D[i]+C[i-1];  /* S4 */
}
```

Figure 1: *example DAG*

# 2 The Scheduling Algorithms

The four algorithms we chose were selected partly because they have been widely studied, and partly because we could either generate schedules or had access to schedulers. Each of the schedulers takes as input a directed acyclic graph (DAG) of the loop body, where nodes represent instructions and arcs represent dependencies between the instructions. For example, Figure 1 shows a loop and its associated DAG. Note that dependencies within the loop are represented with solid arrows and loop carried dependencies are represented with dashed arrows. For our experiments, we constructed the DAGs manually.

We use event count synchronization in all the schedules. For each send/receive synchronization we allocate a global variable initialized to some value and incremented by the sender each iteration. The synchronization is then effected by the receiver spinning on the variable until it reaches the loop index. On the Firefly, the send requires one instruction to increment the variable and a minimum 400ns bus transfer time:

```
addl2  $delta,_e0
```

The receive requires two instructions:

```
L0: cmpl   _e0,r1
    jlssu  L0
```

Because the snooping cache architecture implements a coherent memory protocol by allowing only one writer at a time per cache line, the synchronizations do not require interlocked instructions. In addition, the receiving processor polls the event count variable in its own cache without generating bus transactions. Therefore, this method is as efficient as can be expected on the Firefly.

## 2.1 Serial Scheduling

This is no scheduling algorithm at all and serves as a base case. The loop unrolling is applied and then the code is run sequentially on one processor.

## 2.2 Doacross Scheduling

The idea behind the doacross algorithm is to schedule individual iterations of the loop onto separate virtual processors [5]. In order to get maximum performance from this algorithm, we assign one virtual processor to each physical processor for this experiment. Unlike software pipelining [11], the order of operations in the loop body is not altered. However, code must be added to synchronize LCDs. For example, consider the hypothetical loop in Figure 1. We can schedule this fragment onto two processors (assuming `N` is even) as follows:

Processor 1:

```
for(i=1; i<=N; i=i+2) {
     while(f0 < i) ;
     A[i] = B[i-1];
     B[i] = A[i];
     e0 = e0 + 2;
     C[i] = B[i];
     while(f1 < i) ;
     D[i] = E[i-1];
     E[i] = D[i] + C[i-1];
     e1 = e1 + 2;
}
```

Processor 2:

```
for(i=2; i<=N; i=i+2) {
     while(e0 < i) ;
     A[i] = B[i-1];
     B[i] = A[i];
     f0 = f0 + 2;
     C[i] = B[i];
     while(e1 < i) ;
     D[i] = E[i-1];
     E[i] = D[i] + C[i-1];
     f1 = f1 + 2;
}
```

The event count variables `e0` and `e1` are initialized to 0 and the variables `f0` and `f1` are initialized to 1. Doacross as described in [4] would actually estimate when the sender would have data ready and insert a time delay at the receiver before the while loop to avoid unnecessary bus cycles. We have not included this delay since the snooping cache architecture of the Firefly keeps the test local and does not generate memory bus transactions.

Clearly, doacross is one of the most straightforward loop optimizations to perform. For loops with no LCDs, it produces a completely vectorized and perfectly balanced schedule.

## 2.3 List Scheduling

The term "list scheduling" is actually a bit of a misnomer here. For our purposes, list scheduling includes both the scheduler and the code generator. This method was originally developed for resources scheduling and was modified by John Ellis [6] for use on VLIWs and other synchronous machines. To our knowledge this algorithm has never been used on a multiprocessor, so we wrote our own scheduler following Ellis' thesis. It is an attractive method because it is guaranteed to complete in linear time, and has been successful as a code generator for trace scheduling.

Let the roots of the input DAG be those operations with no successors, and the leaves be those with no predecessors. That is, the roots are those operations which execute last. The algorithm recursively traverses the DAG from the roots to the leaves, propagating estimates of which processor would best suit each operation. As the recursion unwinds from the leaves to the roots, the actual assignments are performed. LCDs are not taken into account when scheduling the DAG. Consider the example in Figure 1. The chart below gives some hypothetical instruction execution times and corresponding best case start times. That is, the earliest times the instructions can begin on any processor taking the dependencies into account.

| Node | Execution Time | Start Time |
|---|---|---|
| S0 | 500 | 0 |
| S1 | 1 | 500 |
| S2 | 1 | 501 |
| S3 | 1 | 0 |
| S4 | 1 | 1 |

The roots of the DAG are S2 and S4. With a communication delay of 1, the recursive list scheduling algorithm will proceed as shown in Figure 2, resulting in the final schedule:

| Node | Processor | Cycle |
|---|---|---|
| S0 | 0 | 0 |
| S1 | 0 | 500 |
| S2 | 0 | 501 |
| S3 | 1 | 0 |
| S4 | 1 | 1 |

This schedule requires one synchronization for the LCD from S2 to S4. For a concise, yet thorough description of list scheduling see [7].

## 2.4 Greedy Software Pipelining

Perfect pipelining is an algorithm originally developed for use on VLIWs and other synchronous machines [1][2] to expose optimal parallelism across loop boundaries. Greedy Software Pipelining (GSP) utilizes perfect pipelining and a collection of mapping heuristics to generate a real schedule which takes the cost of synchronization into account. That is, the perfect pipelining schedule is first generated for a hypothetical synchronous machine with an infinite number of resources and no communication delay between processors. The GSP mapping heuristics are then applied to create a schedule for the real machine.

The first step in the GSP algorithm is to divide the operations in the loop into three groups, each of which is scheduled separately. The *flow-in* nodes are those with either no predecessors, or whose predecessors are all other flow-in nodes. The *flow-out* nodes have no successors, or only successors that are other flow-out nodes. The rest of the operations in the DAG are the *cyclic* nodes.

The cyclic nodes are scheduled using a greedy algorithm. First, the perfect pipelining algorithm is applied. This algorithm achieves the effect of infinite loop unrolling combined with fine-grain "as soon as possible" scheduling. The basic idea is to ignore iteration boundaries and simply schedule each operation as early as possible on one of the processors. Eventually a repeating instruction pattern will emerge across

```
call ASSIGN(S2)
    estimate processor 1 at time 501
    estimate processor 2 at time 501

    call ASSIGN(S1)
        estimate processor 1 at time 500
        estimate processor 2 at time 500

        call ASSIGN(S0)
            processor 1 at time 0
            processor 2 at time 0
            Assigning node S0 to P1 at time 0

        processor 1 at time 500
        processor 2 at time 501
        Assigning node S1 to P1 at time 500

    processor 1 at time 501
    processor 2 at time 502
    Assigning node S2 to P1 at time 501

call ASSIGN(S4)
    estimate processor 1 at time 1
    estimate processor 2 at time 1

    call ASSIGN(S3)
        processor 2 at time 0
        processor 1 at time 502
        Assigning node S3 to P2 at time 0

    processor 2 at time 1
    processor 1 at time 502
    Assigning node S4 to P2 at time 1
```

Figure 2: *list scheduling example*

all the processors, and this becomes the new loop body. Note that perfect pipelining could require a run time that is exponential in the number of nodes in the DAG, but seems to run in low order polynomial time in practice [1].

Once the "ideal" pattern has been determined, it is used as a guide for the mapping heuristics. The cyclic nodes are scheduled in the same topological order as the pattern, without backtracking. As each node is considered, it is assigned to the processor which generates the earliest estimated finishing time for that operation. Estimates of the synchronization and instruction execution times are used when calculating the estimated finishing time.

Since the flow-in and flow-out groups have dependencies in only one direction, they are not on the critical path of the loop. Therefore, they are easily scheduled by allocating whole iterations to each processor. As long as sufficient resources are available, they will not affect the execution time of the loop. If resources are limited, processors are allocated proportionally according to the expected execution times of the three groups, so that they will finish at approximately the same time.

Consider the example in Figure 1, which has no flow-in or flow-out nodes. One perfect pipelining schedule begins:

| Step | Processor 1 | Processor 2 |
|---|---|---|
| 0 | $S0_1$ | $S3_1$ |
| 1 | $S1_1$ | $S4_1$ |
| 2 | $S2_1$ | $S3_2$ |
| **3** | $\mathbf{S0}_2$ | $\mathbf{S4}_2$ |
| **4** | $\mathbf{S1}_2$ | $\mathbf{S3}_3$ |
| **5** | $\mathbf{S2}_2$ | |
| 6 | $S0_3$ | $S4_3$ |
| 7 | $S1_3$ | $S3_4$ |
| ⋮ | ⋮ | ⋮ |

The pattern of steps 3 through 5 will repeat until all iterations have been exhausted. Therefore, we can create a loop which achieves the effect of infinitely unrolling the DAG and greedy scheduling with the following code (for `N` even and `N` $> 1$):

Processor 1:

```
for (i=1; i<=N; i=i+1) {
{    A[i] = B[i-1];
     B[i] = A[i];
     C[i] = B[i];
     f0 = f0 + 1;
}
```

Processor 2:

```
     D[i] = E[i-1];
     for (i=1; i<N; i=i+1) {
          while(f0 < i) ;
          E[i] = D[i] + C[i-1];
          D[i+1] = E[i];
     }
     while(f0 < N) ;
     E[N] = D[N] + C[N-1];
```

Again, `f0` is initialized to 1 rather than 0, satisfying the while condition on the first iteration. Note that the perfect pipelining schedule requires only two processors, so it can be used as is.

For a more complex example and a thorough description of the GSP heuristics used for our experiments, see [3].

## 2.5 Top Down Scheduling

The idea behind top down scheduling is to group interdependent operations together on the same processor to avoid synchronization costs, while scheduling with no backtracking [9]. The schedule is generated in reverse-execution order by proceeding top down through the DAG. That is, the operations to be executed last are scheduled first and those to be executed first are scheduled last. The basic algorithm for $P$ processors is:

1. If all the nodes have been scheduled, then exit the algorithm.
2. Create a set, $S$, of nodes whose parents have all been scheduled.
3. Schedule up to $P$ of these nodes in decreasing depth order, trying to schedule nodes onto the same processor as one of their parents.
4. If less than $P$ nodes were scheduled, go to step 1.
5. Schedule subtrees of these $P$ nodes until at least one of the subtrees has been exhausted.
6. Go to step 1.

Consider the example in Figure 1. After step 2, $S = \{S2, S4\}$, so S2 is scheduled to P1 and S4 is scheduled to P2. Since 2 nodes have been scheduled, the subtree S1 of S2 and the subtree S3 of S4 are scheduled. Now the subtree of S4 is exhausted so we return to step 1 and create $S = \{S0\}$. Since the parent of S0 is scheduled onto P1, the final schedule is:

| Processor 1 | Processor2 |
|---|---|
| S2 | S4 |
| S1 | S3 |
| S0 | |

Because this is a top down schedule, it must be reversed for execution. The final code for this example is the same as the list schedule:

Processor 1:

```
for (i=1; i<=N; i=i+1) {
     A[i] = B[i-1];
     B[i] = A[i];
     C[i] = B[i];
     f0 = f0 + 1;
}
```

Processor 2:

```
for (i=1; i<=N; i=i+1) {
     D[i] = E[i-1];
     while(f0 < i) ;
     E[i] = D[i] + C[i-1];
}
```

Note that we have not taken instruction execution time or LCDs into account but we still get excellent results, as will be discussed below. Since only one pass is required, top down scheduling is linear in the number of nodes in the DAG.

# 3 The Experiments

It is difficult to objectively compare the scheduling methods, even when one considers only loops consisting of one basic block, with LCDs spanning only one iteration. Besides the choice of loops, there are the issues of machine models, unrolling factor, scheduling granularity, interprocessor communication cost, runtime variations due to nondeterministic contention, and array size.

We selected a representative loop for each of four cases: a loop with no LCDs, a loop with temporary variable reuse LCDs only, a loop with one LCD, and a reasonably complicated loop with many LCDs. Each of these experiments is discussed individually in the following sections.

We considered two machine models: 2 processors and 4 processors. In order to compare with doacross, we selected one unrolling factor to match the number of processors. The other was chosen assuming that the loop was unrolled once before doacross scheduling. Therefore, the second factor is twice the first. We did not consider any other cases because the schedules become too long to generate by hand in a reasonable amount of time. The schedulers were given the number of processors available as an input, though they were not required to use them all.

As discussed in section 1, the scheduling granularity was chosen to be a single C statement. The execution times of each of the statements was obtained through experimentation, as was the cost of synchronization. These times were used as estimates by greedy software pipelining and list scheduling.

Because we performed actual empirical experiments on the Firefly, we attempted to reduce the architectural effects by performing the experiments as follows. There were eight separate programs; one for each of the loops and for each of the machine models (two and four processors). Each of these programs iterated many times through all the scheduling algorithms on a particular model *without* changing the data placement in virtual memory or the mapping between physical and virtual addresses. Array size was chosen so that approximately 4 megabytes of memory was occupied, thereby obviating the need for page swapping. To complete, the results of all the iterations were averaged for each algorithm. Performing the experiments in this manner served to reduce the effect of data placement in memory and in the cache since it remained the same for all of the algorithms.

The data is presented in tables subdivided by number of processors and unrolling factors. Each column of the tables is based upon schedules generated from the same DAG. The fact that this DAG is created by unrolling the loop DAG some number of times is only significant when comparing numbers from different columns. This is important because GSP only generates one schedule for a given loop and finds the ideal amount of unrolling. By unrolling before scheduling, we cannot in truth say this is a GSP schedule for that loop. It is, however, a GSP schedule for the unrolled loop *DAG*. Therefore, comparisons between GSP schedules for, say, unrolled once and unrolled thrice DAGs can only be made by keeping this in mind. We would still expect the unrolled thrice schedule to outperform the unrolled once schedule because there are half as many iterations of the code, although this need not be the case since the source DAGs are different. We have the same situation with doacross. Technically there is only one doacross schedule per loop, but we can also say there is one doacross schedule per DAG. It is only informative to compare different columns when we keep this in mind.

We have not included any of the actual schedules in this paper because they are quite lengthy. The tables in the appendix give various code statistics and actual schedules can be obtained from the authors.

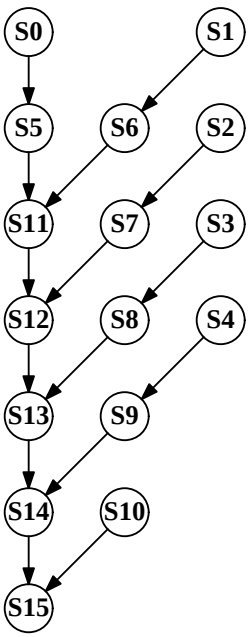



Figure 3: *FIR Filter*

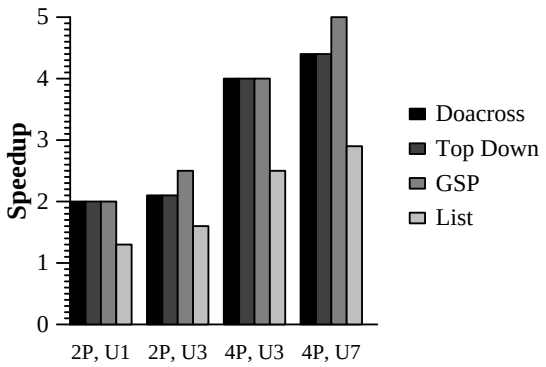



Figure 4: *FIR filter speedups*

### 3.1 Loop Without LCDs

This experiment involves a Linear Phase B-Spline Interpolated FIR filter used for digital signal processing applications [13]. The DAG is shown in Figure 3 and the code can be found in the appendix. The loop has no LCDs, so the most efficient parallelization involves scheduling entire iterations onto processors and balancing the workload between processors. The purpose of this experiment was to see if the scheduling methods would find this obvious parallelization. Note that this loop is representative of the fully vectorizable code found in many scientific applications. The experiment was performed with an array size of 29400 and all variables were of type double. The execution times are shown in table 1.

Speedups were calculated relative to the serial execution time with the equation:

$$\text{SPEEDUP(alg)} = \frac{\text{TIME(serial)}}{\text{TIME(alg)}}$$

Note that TIME(serial) is that from the same column (same machine model and unrolling factor) as TIME(alg). The speedups for the FIR filter are shown in Figure 4. The most notable feature of the graph is that there are quite a few superlinear speedups. These speedups were achieved by the algorithms which successfully found the parallelization described above. The superlinearity is due to the fact that the loop overhead is parallelized and that by dividing the code among processors, we are effectively increasing the cache size and lowering the collision rate. This applies to both the instruction and data caches. The effect is interesting because there are, and will continue to be, many machines with direct mapped caches. If a parallelization technique distributes the work to more than one cache in such a way as to reduce cache line conflicts, we get a certain degree of speedup for free, or in addition to that gained by the overlap of operation execution.

The second most notable feature of Figure 4 is that list scheduling does quite a bit worse than the other algorithms. This is because it did not schedule entire iterations to processors, but split the iterations between the processors, adding quite a few unnecessary synchronizations (see the appendix). It is a relatively simple matter to add a heuristic to any loop scheduling method to test for the presence of LCDs. If none are found, doacross should be used.

Serial shows a slowing down trend as the loop is unrolled and processors are added. Since this is run on only one processor, the effect must be due to cache line collisions. Loop unrolling does not significantly effect the data access pattern, but it does increase the code size. Since the instruction cache is only 1 Kbyte in size, the code pages are probably being swapped out more frequently leading to a greater number of instruction cache misses. Notice that these architectural effects are non-trivial indicating that *small variations between numbers for all the experiments are not significant.* The analysis for the other algorithms is similar.

Now consider comparisons between algorithms for the same DAG (columns of the table). Doacross generates the optimal schedule by default. List scheduling, GSP, and top down scheduling all have the option of rearranging the order of operations. GSP and top down scheduling both scheduled entire iterations to processors, though they both found different orders for the operations. Variation in the execution times

| Algorithm | 2 processors | | 4 processors | |
|---|---|---|---|---|
| | Unrolled 1 | Unrolled 3 | Unrolled 3 | Unrolled 7 |
| Serial | 3.000 | 3.237 | 3.228 | 3.612 |
| Doacross | 1.530 | 1.508 | 0.809 | 0.813 |
| Top Down | 1.522 | 1.524 | 0.805 | 0.814 |
| GSP | 1.526 | 1.288 | 0.803 | 0.716 |
| List | 2.256 | 2.070 | 1.285 | 1.254 |

Note: all times are in seconds.

Table 1: *FIR Filter execution times.*

for these two algorithms and doacross is due to the different data access patterns. Note that Top down scheduling viewed each iteration as a subtree of S15 (Figure 3).

### 3.2 Loop With Intra-Iteration LCDs

Livermore Loops Kernel 7 is an equation of state fragment [12]. We modified the inner loop to allow for the extraction of parallelism by adding temporary variables as follows:

```
for(i=0; i<N; i=i+1) {
 t0=T*(u[i+6]+R*(u[i+5]+R*u[i+4])); /* S0 */
 t1=u[i+3]+R*(u[i+2]+R*u[i+1]);     /* S1 */
 t2=u[i]+R*(z[i]+R*y[i]);           /* S2 */
 t0=T*(t0+t1);                      /* S3 */
 x[i]=t0+t2;                        /* S4 */
}
```

The DAG is shown in Figure 5(a) and the original code can be found in the appendix. When the code is unrolled, we allocate a new set of temporaries for each instance of the loop body, assuming a compiler would do this. Therefore, the only LCDs are for reuse of the temporaries within an iteration as shown in the once unrolled DAG of Figure 5(b). Therefore, this experiment is similar to the FIR filter experiment, except the presence of LCDs could influence the schedulers. The optimal schedule, again, involves scheduling entire iterations onto processors and balancing the workload. The experiment was performed with an array size of 125000, and all variables were defined as type double. The speedups are shown in Figure 6 and the execution times are shown in table 2.

Again, we see superlinear speedups for doacross and top down scheduling, and the explanation is the same as that for the FIR filter: doacross has no choice and top down scheduling views entire iterations as subtrees of S4. GSP, however, is thrown by the presence of the LCDs and does not schedule entire iterations onto processors. The iterations are

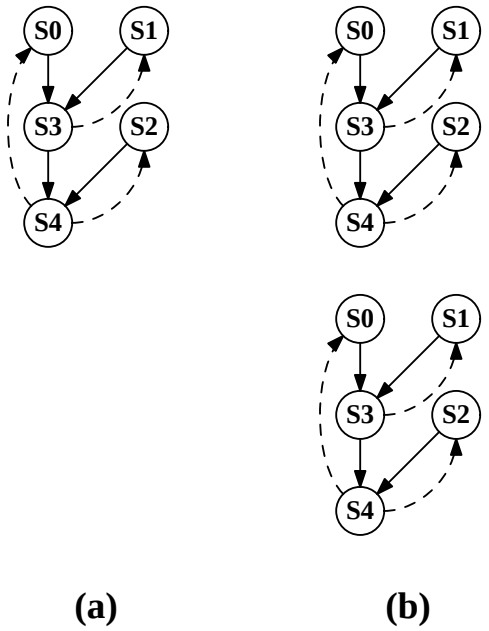



Figure 5: *Livermore Loops Kernel 7: (a) not unrolled, and (b) unrolled once*

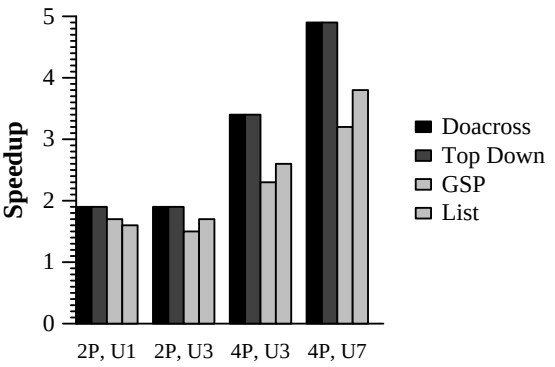



Figure 6: *Livermore Loops Kernel 7 speedups*

| Algorithm | 2 processors | | 4 processors | |
|---|---|---|---|---|
| | Unrolled 1 | Unrolled 3 | Unrolled 3 | Unrolled 7 |
| Serial | 6.588 | 6.522 | 6.578 | 9.764 |
| Doacross | 3.476 | 3.417 | 1.934 | 1.974 |
| Top Down | 3.476 | 3.417 | 1.934 | 1.974 |
| GSP | 3.874 | 4.231 | 2.845 | 3.030 |
| List | 4.009 | 3.877 | 2.553 | 2.578 |

Note: all times are in seconds.

Table 2: *Kernel 7 execution times.*

split and synchronizations added resulting in somewhat lower performance. List scheduling continues to divide the iterations among processors, as with the FIR filter. Note that the doacross and top down schedules turned out to be identical.

### 3.3 Loop With One LCD

This experiment deals with the inner loop of the Gauss-Seidel step [8]. That is, one step of the Gauss-Seidel iteration for the solution of linear equations. The purpose of this experiment was to see how the schedulers performed with a loop that is typical of the inner loops from a large number of "real" matrix oriented scientific applications: it is short and has one LCD. The original code can be found in the appendix. Again, it can easily be modified to allow for the extraction of parallelism but there are two obvious ways to do this. The first will be called the "balanced tree" method and produces the DAG shown in Figure 7(a) corresponding to the following code:

```
for (i=1; i<N; i=i+1) {
   t0 = new[i-1][j] + new[i][j-1];  /* S0 */
   t1 = old[i+1][j] + old[i][j+1];  /* S1 */
   new[i][j] = C * (t0 + t1);       /* S2 */
}
```

The balanced tree is interesting because it is the typical way a compiler might parallelize this type of code. The somewhat less obvious parallelization will be called the "optimal" method and produces the DAG shown in Figure 7(b) corresponding to the following code:

```
for (i=1; i<N; i=i+1) {
   t0 = old[i+1][j] + old[i][j+1];  /* S0 */
   t0 = t0 + new[i][j-1];           /* S1 */
   new[i][j] = C*(t0+new[i-1][j]);  /* S2 */
}
```

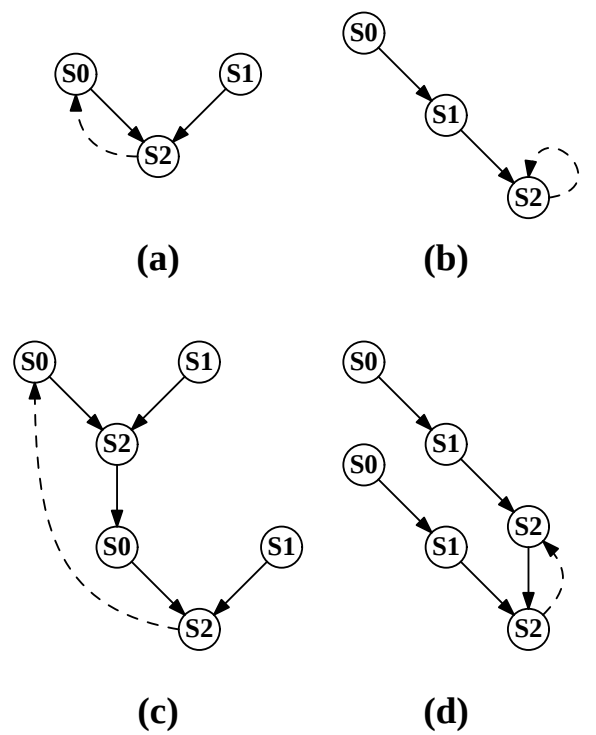



Figure 7: *Gauss-Seidel Inner Loop: (a) "tree" DAG, (b) "optimal" DAG, (c) tree DAG unrolled once, and (d) optimal DAG unrolled once*

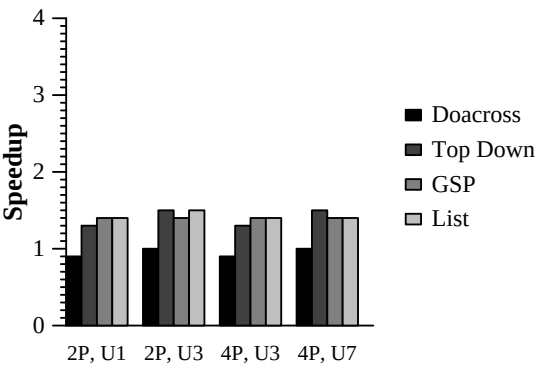



Figure 8: *Gauss-Seidel balanced tree DAG speedups*

| Algorithm | 2 processors | | 4 processors | |
|---|---|---|---|---|
| | Unrolled 1 | Unrolled 3 | Unrolled 3 | Unrolled 7 |
| Serial | 5.825 | 5.780 | 5.792 | 6.131 |
| Doacross | 6.256 | 5.942 | 6.304 | 5.913 |
| Top Down | 4.350 | 3.917 | 4.354 | 4.166 |
| GSP | 4.066 | 3.998 | 4.022 | 4.443 |
| List | 4.119 | 3.936 | 4.038 | 4.482 |

Note: all times are in seconds.

Table 3: *Gauss-Seidel Balanced Tree execution times.*

"Optimal" simply refers to the fact that this parallelization has the shortest possible path length between iterations when the loop is unrolled (Figures 7(c) and 7(d)) allowing for the greatest degree of parallelism.

First, consider the balanced tree method. The experiment was performed with an array size of 170040 and all variables were of type double. The speedups are shown in Figure 8 and the execution times are shown in table 3.

Notice that in this parallelization almost all the operations are on the critical path so very little parallelism can be extracted. Adding processors does not significantly help any of the algorithms since the code is almost completely serialized and any speed gained through parallelism is lost to synchronization overhead. Therefore, we do not see any significant speedups at all. In fact, we see some slowdowns caused by the additional communication costs. This would indicate that loops such as this with a completely dominant critical path might not be worth parallelizing because the small amount of time gained from parallelism could easily be exceeded by the time lost to communication, and the risk is too great.

Unrolling seems to help a bit in most cases since the loop overhead is decreased and there is a bit more opportunity for parallelism because of the additional S1 nodes which are off the critical path. Doacross, which cannot creatively schedule these nodes, does worse than the algorithms which can in all cases. Doacross exhibits an almost uniform slowdown as compared with serial because the doacross code is almost completely serialized by the LCD from S2 to S0.

The three algorithms which can rearrange the order of operations make various tradeoffs between increasing parallelism and decreasing interprocessor communication so their run times differ somewhat. One interesting case is the 4 processor unrolled seven times DAG where top down with 16 synchronizations (see appendix) surpasses both GSP and list scheduling with fewer. Top down only used two processors scheduling the entire critical path onto one and all the other operations onto the other. Since there are fewer operations off the critical path than on, they could be "precomputed" by the one processor and always be available to the other. GSP and list scheduling both distributed the critical path operations between processors (2 for GSP and 4 for list scheduling).

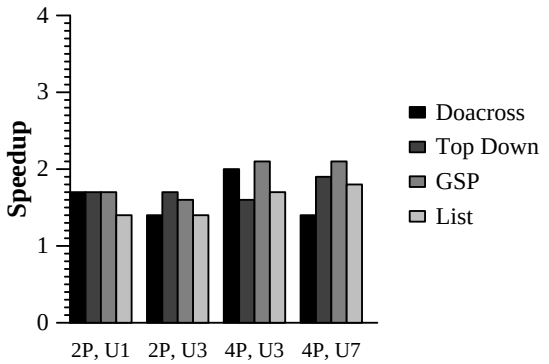



Figure 9: *Gauss-Seidel optimal DAG speedups*

Top down scheduling yields the fastest schedule in two out of the four cases tested. In the other two it does quite well, running less than 10% slower than the fastest schedule. Considering the simplicity of the method, this is quite impressive.

Now consider the optimal method. The experiment was also performed with an array size of 170040 and all variables were of type double. The speedups are shown in Figure 9 and the execution times are shown in table 4.

The first thing to notice is that the speedups are better than those obtained with the balanced tree method. The reason for this, as hypothesised above, is that the critical path length of this loop when unrolled is only 1 between iterations. The second thing to notice is that the speedups remain rather flat as the loop is unrolled and processors are added. This is because the loop is so short that although there is more parallelism than with the tree method in the unrolled loop, synchronization time masks the decrease in work time for the most part.

| Algorithm | 2 processors | | 4 processors | |
|---|---|---|---|---|
| | Unrolled 1 | Unrolled 3 | Unrolled 3 | Unrolled 7 |
| Serial | 5.561 | 5.520 | 5.517 | 5.827 |
| Doacross | 3.222 | 4.031 | 2.728 | 4.030 |
| Top Down | 3.226 | 3.184 | 3.369 | 3.070 |
| GSP | 3.259 | 3.352 | 2.606 | 2.755 |
| List | 4.103 | 4.047 | 3.323 | 3.291 |

Note: all times are in seconds.

Table 4: *Gauss-Seidel Optimal execution times.*

When the unrolling factor matches the number of processors, doacross would seem to be the best algorithm to use. However, GSP actually exceeds doacross on 4 processors (though not necessarily significantly). In the doacross schedule, the four processors communicate in a ring-like fashion as they share the value of `new[i][j]`, so the critical path of the unrolled loop is spread uniformly among the 4 processors and each dependency requires an interprocessor synchronization. Therefore, the processors spend a good deal of time simply waiting since the loop is so short. It turns out that the GSP schedule uses only three processors and does less synchronization on the critical path.

Doacross does significantly worse as the loop is unrolled because the parallelism actually decreases. This can be seen for the 2 processor case by considering Figure 7. When unrolled once, each processor has a loop similar to Figure 7(b) with a critical path length of 1. When unrolled thrice, each processor has a loop similar to Figure 7(d) with a critical path length of 4, so the amount of work being done in parallel is less.

Variations in the run times of the three algorithms which can rearrange the order of operations can be attributed to the various tradeoffs between increasing parallelism and decreasing interprocessor communication. Notice that top down scheduling comes in a close second behind GSP overall for this example.

### 3.4 Loop With Complex LCDs

The elliptical filter is a well known benchmark for the study of high level circuit synthesis [10]. The DAG is shown in Figure 10 and the code can be found in the appendix. The purpose of this experiment was to see how the scheduling algorithms would deal with a complicated loop having many LCDs. The experiment was performed with an array size of 21000, and all variables were defined as type double. The speedups are shown in Figure 11 and the execution times are shown in table 5.

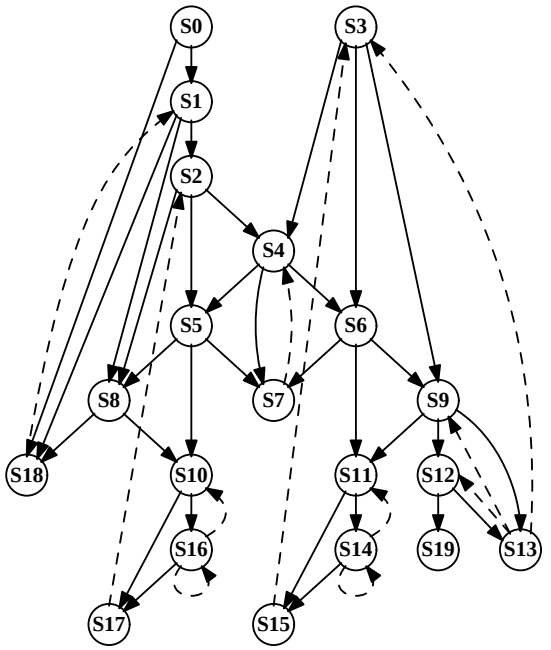



Figure 10: *Elliptical Filter*

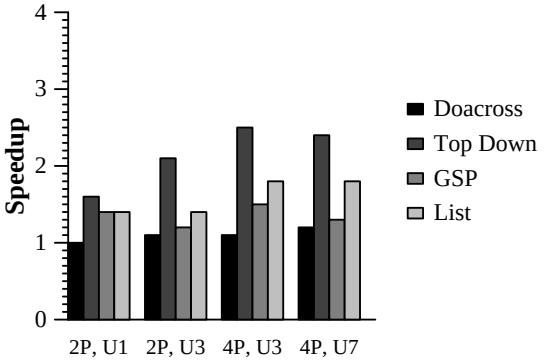



Figure 11: *Elliptical filter speedups*

| Algorithm | 2 processors | | 4 processors | |
|---|---|---|---|---|
| | Unrolled 1 | Unrolled 3 | Unrolled 3 | Unrolled 7 |
| Serial | 3.384 | 3.655 | 3.608 | 4.052 |
| Doacross | 3.315 | 3.368 | 3.332 | 3.387 |
| Top Down | 2.073 | 1.711 | 1.425 | 1.686 |
| GSP | 2.463 | 3.069 | 2.482 | 3.108 |
| list | 2.458 | 2.663 | 1.986 | 2.284 |

Note: all times are in seconds.

Table 5: *Elliptical Filter execution times.*

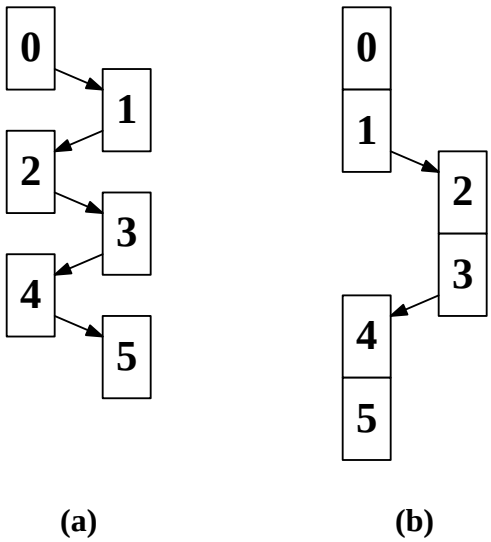



Figure 12: *Doacross: (a) unrolled once, and (b) unrolled thrice (Elliptical Filter)*

The most obvious feature of the graph is that top down scheduling did significantly better than any other algorithm overall. This is impressive considering that operation execution times, which range from 1.5 $\mu$sec to 9.6 $\mu$sec, and LCDs were not taken into account. This would suggest that grouping operations together is very important when scheduling for a machine with high communication cost, even if that requires concentrating on only one iteration at a time.

The second most notable feature of the graph is that the speedups remain quite flat as the loops are unrolled and processors are added. The complexity of the dependencies seems to prevent all of the algorithms from extracting more than the obvious initial amount of parallelism. The numbers for doacross represent almost serialized code because of the LCD from S18 to S1. Notice that GSP and list scheduling cannot seem to do much better than the serialized code.

In all but one case (top down on 2 processors), the execution time increases as the loops are unrolled. Again, if we consider this to be significant, there is a possible explanation. In the case of doacross, there is a slight overlap of execution between the processors as shown in Figure 12(a). When the loop is unrolled, there is no overlap between successive iterations on the same processor so the degree of parallelism actually decreases. The same sort of overlapping effect may contribute to the slight speedup for the thrice unrolled loop as we go from 2 to 4 processors.

For the three algorithms which can rearrange the order of operations, there is a reverse correlation between execution time and code size (see appendix). In addition, there is a reverse correlation between execution time and synchronization number in all but the "unrolled once on 2 processors" case. This would reinforce the intuitive notion that minimizing interprocessor communication while still exploiting parallelism can maximize performance for algorithms which can rearrange the order of operations. Finding the balance point is, of course, non-trivial.

# 4 Conclusions

The experiments presented in this study confirm that fine-grain parallelism is a viable method for obtaining speedups on a shared memory multiprocessor. In fact, the speedups presented were quite impressive when one considers the architecture of the Firefly, which was certainly not designed to support fast interprocessor synchronization. Parallelizing at the statement level rather than the instruction level provided a situation analogous to instruction level parallelism on a machine with fast synchronization. Current and next generation processors such as the Intel i860 and the IBM RS/6000 which allow for very fast synchronization between multiple on-board functional units, and large machines such as the Alliant supercomputers which provide a dedicated synchronization bus should benefit from these techniques.

While our results were obtained with particular ratios of statement execution time to synchronization time, we would expect the methods to apply to other situations since the schedulers (for the most part) take this into account.

Top down scheduling and GSP achieved the best results, with top down scheduling doing the best overall. This is somewhat surprising considering that top down scheduling is the most straightforward algorithm we used besides doacross. The idea of grouping operations from the top down is intuitively appealing since most mathematical DAGs are narrow at the top (where the results are produced) and wide at the bottom. Though this does not appear to be the case for the elliptical DAG, top down scheduling still yields the best results. It should be noted that the execution times for GSP were largely dependent upon the heuristics applied to the perfect pipelining schedules. While perfect pipelining finds an optimal schedule for the loop, the heuristics are forced to make the hard choices and fit this schedule onto the real machine. Changing the heuristics would change the results.

The loops with no LCDs displayed an interesting superlinear speedup effect for the fully vectorized parallelizations. This effect was obviously not due to the decreased run time of the parallelized code, but to the inflated run time of the serial code due to cache line conflicts. The parallelization effectively multiplied the cache size and lead to a lower overall miss rate. It would be interesting to measure this effect with various degrees of cache associativity. The direct mapped cache should exhibit the best overall speedups because of the high degree of line sharing. Since many multiprocessors are built with direct mapped caches for economy and simplicity, this type of speedup should be widely applicable.

Fine-grain parallelization can also be applied to distributed memory machines with shared memory multiprocessor nodes. For physical and economic reasons, these architectures are becoming more and more prevalent since the physical size of components is shrinking much faster than the size of interconnects. Machines with shared memory multiprocessor nodes can take advantage of instruction level parallelism, allowing the programmer to use a simple distributed machine model while realizing the power of the additional processors.

It can be argued that our sample size is quite small. Although this is true, the results certainly indicate that these techniques are useful and should be explored further. They are efficient to use since they all produce schedules in low order polynomial time and are applied only once during compilation. The speedups are certainly acceptable. For instance, both loops with LCDs showed speedups greater than a factor of two on four processors, with top down scheduling achieving a speedup of 2.5 for the elliptical filter with the complex dependencies. We would expect to see similar speedups for loops composed of more than one basic block where techniques such as trace scheduling could be applied. Finally, we have demonstrated that these techniques can be used with virtually any shared memory multiprocessor having a small number of processors.

Finally, it should be mentioned that we did not select the experiments to include in this paper from a larger group of results. We selected the four loops based upon their inherent characteristics and then performed the experiments which have been documented here.

## Acknowledgements

Matthias Blumrich was supported by a DARPA Assistantship in Parallel Processing. Carrie Brownhill was supported in part by NSF grant CCR8704367 and ONR grant N0001486K0215. Kai Li was supported in part by NSF grant CCR-9020893, by DARPA and ONR under contract N00014-91-J-4039, and by Digital Systems Research Center and Digital External Research Program. Alexandru Nicolau was supported by ONR grant N00014-91-J-1406.

## References

[1] A. Aiken and A. Nicolau. Optimal loop parallelization. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 308–317, June 1988.

[2] A. Aiken and A. Nicolau. Perfect pipelining: A new loop parallelization technique. In *Proceedings of the 1988 European Symposium on Programming*. Springer-Verlag Lecture Notes in Computer Science No. 300, March 1988.

[3] C. J. Brownhill, K. Kim, and A. Nicolau. Fine-grain loop scheduling for MIMD machines. Technical Report 90-34, University of California at Irvine, October 1990.

[4] R. Cytron. *Compile-time Scheduling and Optimization for Asynchronous Machines*. PhD thesis, University of Illinois at Urbana-Champaign, October 1984.

[5] R. Cytron. Doacross: Beyond vectorization for multiprocessors. In *Proceedings of the 1986 Int'l*

*Conference on Parallel Processing*, pages 836–844, August 1986.

[6] J. R. Ellis. *Bulldog: A Compiler for VLIW Architectures*. PhD thesis, Yale University, 1985.

[7] J. A. Fischer, J. R. Ellis, J. C. Ruttenberg, and A. Nicolau. Parallel processing: A smart compiler and a dumb machine. In *Proceedings of the SIGPLAN '84 Symposium on Compiler Construction*, pages 37–47, 1984. Published as SIGPLAN Notices 19 (6), June, 1984.

[8] G. H. Golub and C. F. Van Loan. *Matrix Computations*. The Johns Hopkins University Press, 1983.

[9] R. Gupta. Employing register channels for the exploitation of instruction level parallelism. In *Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 118–127, March 1990. Published as SIGPLAN Notices 25 (3), March, 1990.

[10] S. Y. Kung, H. J. Whitehouse, and T. Kailath. *VLSI and Modern Signal Processing*. Prentice-Hall Information and Systems Sciences Series, 1985.

[11] M. Lam. Software pipelining: An effective scheduling technique for VLIW machines. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 318–328, July 1988. Published as SIGPLAN Notices 23 (7).

[12] F. H. McMahon. Lawrence livermore national laboratory fortran kernels: Mflops. Livermore, CA, 1983.

[13] Y. Neuvo, D. Chengyu, and S. K. Mitra. Interpolated finite impulse response filters. *IEEE Transactions ASSP*, ASSP-32(3), June 1984.

[14] C. P. Thacker, L. C. Stewart, and E. H. Satterthwaite, Jr. Firefly: A multiprocessor workstation. *IEEE Transactions on Computers*, 37(8), August 1988.

# Appendix

The following sections contain the original loop code and tables with various code statistics. Note that GSP can generate a long prologue and epilogue to the loop body so comparisons between code size and execution time can be misleading. Again, the experimental code is available from the authors.

## FIR Filter

This is the loop with no LCDs.

```
for(i=0; i<N; i=i+2) {
   a[0][i] = b[i] + a10;              /* S0  */
   a[1][i] = a1 + a9;                 /* S1  */
   a[2][i] = a2 + a8;                 /* S2  */
   a[3][i] = a3 + a7;                 /* S3  */
   a[4][i] = a4 + a6;                 /* S4  */
   a[5][i] = m0 * a[0][i];            /* S5  */
   a[6][i] = m1 * a[1][i];            /* S6  */
   a[7][i] = m2 * a[2][i];            /* S7  */
   a[8][i] = m3 * a[3][i];            /* S8  */
   a[9][i] = m4 * a[4][i];            /* S9  */
   a[10][i] = m5 * a5;                /* S10 */
   a[11][i] = a[5][i] + a[6][i];      /* S11 */
   a[12][i] = a[11][i] + a[7][i];     /* S12 */
   a[13][i] = a[12][i] + a[8][i];     /* S13 */
   a[14][i] = a[13][i] + a[9][i];     /* S14 */
   a[15][i] = a[14][i] + a[10][i];    /* S15 */
}
```

FIR Filter Code Size

| Algorithm | 2 processors Unrolled 1 | 2 processors Unrolled 3 | 4 processors Unrolled 3 | 4 processors Unrolled 7 |
|---|---|---|---|---|
| serial | 35 | 70 | 70 | 139 |
| doacross | 36 | 70 | 72 | 140 |
| top down | 36 | 70 | 72 | 140 |
| GSP | 36 | 70 | 72 | 140 |
| list | 52 | 125 | 115 | 238 |

Note: Numbers represent lines of code in the loop only.

FIR Filter Synchronization Number

| Algorithm | 2 processors Unrolled 1 | 2 processors Unrolled 3 | 4 processors Unrolled 3 | 4 processors Unrolled 7 |
|---|---|---|---|---|
| serial | 0 | 0 | 0 | 0 |
| doacross | 0 | 0 | 0 | 0 |
| top down | 0 | 0 | 0 | 0 |
| GSP | 0 | 0 | 0 | 0 |
| list | 7 | 11 | 17 | 37 |

Note: Number of event count variables used.

## Kernel 7

This is the loop with intra-iteration LCDs only.

```
for (l=1; l<=loop; l=l+1) {
  for (i=0; i<N; i=i+1) {
    x[i]=u[i]+R*(z[i]+R*y[i])
         + T*(u[i+3]+R*(u[i+2]+R*u[i+1])
         + T*(u[i+6]+R*(u[i+5]+R*u[i+4])));
  }
}
```

We modified the inner loop as follows:

```
for(i=0; i<N; i=i+1) {
  t0=T*(u[i+6]+R*(u[i+5]+R*u[i+4]));/* S0 */
  t1=u[i+3]+R*(u[i+2]+R*u[i+1]);    /* S1 */
  t2=u[i]+R*(z[i]+R*y[i]);          /* S2 */
  t0=T*(t0+t1);                     /* S3 */
  x[i]=t0+t2;                       /* S4 */
}
```

Kernel 7 Code Size

| Algorithm | 2 processors Unrolled 1 | 2 processors Unrolled 3 | 4 processors Unrolled 3 | 4 processors Unrolled 7 |
|---|---|---|---|---|
| serial | 13 | 25 | 25 | 49 |
| doacross | 14 | 26 | 28 | 52 |
| top down | 14 | 26 | 28 | 52 |
| GSP | 44 | 86 | 120 | 236 |
| list | 24 | 46 | 59 | 104 |

Note: Numbers represent lines of code in the loop only.

Kernel 7 Synchronization Number

| Algorithm | 2 processors Unrolled 1 | 2 processors Unrolled 3 | 4 processors Unrolled 3 | 4 processors Unrolled 7 |
|---|---|---|---|---|
| serial | 0 | 0 | 0 | 0 |
| doacross | 0 | 0 | 0 | 0 |
| top down | 0 | 0 | 0 | 0 |
| GSP | 4 | 8 | 16 | 32 |
| list | 4 | 8 | 11 | 18 |

Note: Number of event count variables used.

## Gauss-Seidel Step

This is the loop with one LCD.

```
for (j=1; j<M; j=j+1) {
   for (i=1; i<N; i=i+1) {
      new[i,j] = C*(new[i-1,j] + new[i,j-1]
                  + old[i+1,j] + old[i,j+1];
   }
}
```

Code parallelized with the "balanced tree" method:

```
for (i=1; i<N; i=i+1) {
   t0 = new[i-1][j] + new[i][j-1];  /* S0 */
   t1 = old[i+1][j] + old[i][j+1];  /* S1 */
   new[i][j] = C * (t0 + t1);       /* S2 */
}
```

G-S Balanced Tree DAG Code Size

| Algorithm | 2 processors Unrolled 1 | 2 processors Unrolled 3 | 4 processors Unrolled 3 | 4 processors Unrolled 7 |
|---|---|---|---|---|
| serial | 9 | 17 | 17 | 33 |
| doacross | 14 | 22 | 28 | 44 |
| top down | 20 | 30 | 42 | 74 |
| GSP | 28 | 48 | 48 | 102 |
| list | 18 | 32 | 40 | 79 |

Note: Numbers represent lines of code in the loop only.

G-S Balanced Tree DAG Synchronization Number

| Algorithm | 2 processors Unrolled 1 | 2 processors Unrolled 3 | 4 processors Unrolled 3 | 4 processors Unrolled 7 |
|---|---|---|---|---|
| serial | 0 | 0 | 0 | 0 |
| doacross | 2 | 2 | 4 | 4 |
| top down | 4 | 4 | 7 | 16 |
| GSP | 4 | 4 | 4 | 10 |
| list | 3 | 5 | 5 | 13 |

Note: Number of event count variables used.

Code parallelized with the "optimal" method:

```
for (i=1; i<N; i=i+1) {
   t0 = old[i+1][j] + old[i][j+1];  /* S0 */
   t0 = t0 + new[i][j-1];           /* S1 */
   new[i][j] = C*(t0+new[i-1][j]);  /* S2 */
}
```

G-S Optimal DAG Code Size

| Algorithm | 2 processors | | 4 processors | |
|---|---|---|---|---|
| | Unrolled 1 | Unrolled 3 | Unrolled 3 | Unrolled 7 |
| serial | 9 | 17 | 17 | 33 |
| doacross | 14 | 22 | 28 | 44 |
| top down | 16 | 30 | 34 | 62 |
| GSP | 21 | 64 | 58 | 112 |
| list | 16 | 30 | 38 | 60 |

Note: Numbers represent lines of code in the loop only.

G-S Optimal DAG Synchronization Number

| Algorithm | 2 processors | | 4 processors | |
|---|---|---|---|---|
| | Unrolled 1 | Unrolled 3 | Unrolled 3 | Unrolled 7 |
| serial | 0 | 0 | 0 | 0 |
| doacross | 2 | 2 | 4 | 4 |
| top down | 2 | 4 | 3 | 7 |
| GSP | 2 | 8 | 6 | 10 |
| list | 2 | 4 | 4 | 4 |

Note: Number of event count variables used.

## Elliptical Filter

This is the loop with complex dependencies.

```
for (i=1; i<=N; i=i+1) {
  a[0][i] = b[i];                        /* S0  */
  a[1][i] = a[0][i]+a[18][i-1];          /* S1  */
  a[2][i] = a[1][i]+a[17][i-1];          /* S2  */
  a[3][i] = a[15][i-1]+a[13][i-1];       /* S3  */
  a[4][i] = a[3][i]+a[7][i-1]+a[2][i];   /* S4  */
  a[5][i] = (m21*a[4][i])+a[2][i];       /* S5  */
  a[6][i] = (m24*a[4][i])+a[3][i];       /* S6  */
  a[7][i] = a[6][i]+a[5][i]+a[4][i];     /* S7  */
  a[8][i] = (m9*(a[2][i]+a[5][i]))
             +a[1][i]+a[1][i];           /* S8  */
  a[9][i] = (m30*(a[6][i]+a[3][i]))
             +a[13][i-1];                /* S9  */
  a[10][i] = a[16][i-1]+a[8][i]
              +a[5][i];                  /* S10 */
  a[11][i] = a[14][i-1]+a[6][i]
              +a[9][i];                  /* S11 */
  a[12][i] = m40*(a[9][i]+a[13][i-1]);   /* S12 */
  a[13][i] = a[12][i]+a[9][i];           /* S13 */
  a[14][i] = a[14][i-1]+(m36*a[11][i]);  /* S14 */
  a[15][i] = a[14][i]+a[11][i];          /* S15 */
  a[16][i] = a[16][i-1]+(m16*a[10][i]);  /* S16 */
  a[17][i] = a[16][i]+a[10][i];          /* S17 */
  a[18][i] = a[8][i]+a[0][i]\\
              +(m6*(a[1][i]+a[8][i]));   /* S18 */
  a[19][i] = a[12][i];                   /* S19 */
}
```

Elliptical Filter Code Size

| Algorithm | 2 processors | | 4 processors | |
|---|---|---|---|---|
| | Unrolled 1 | Unrolled 3 | Unrolled 3 | Unrolled 7 |
| serial | 43 | 85 | 85 | 169 |
| doacross | 48 | 90 | 96 | 180 |
| top down | 63 | 116 | 168 | 353 |
| GSP | 67 | 351 | 503 | 986 |
| list | 73 | 140 | 173 | 335 |

Note: Numbers represent lines of code in the loop only.

Elliptical Filter Synchronization Number

| Algorithm | 2 processors | | 4 processors | |
|---|---|---|---|---|
| | Unrolled 1 | Unrolled 3 | Unrolled 3 | Unrolled 7 |
| serial | 0 | 0 | 0 | 0 |
| doacross | 2 | 2 | 4 | 4 |
| top down | 8 | 13 | 34 | 66 |
| GSP | 6 | 45 | 62 | 123 |
| list | 14 | 25 | 36 | 68 |

Note: Number of event count variables used.