Instruction Reordering for Fork-Join Parallelism

Vivek Sarkar
IBM Research
Thomas. J. Watson Research Center
P.O. Box 704, Yorktown Heights, NY 10598

1 Introduction

Any execution of a parallel program must satisfy the program's *control dependences* [FOW87] and *data dependences* [KKP*81]. Broadly speaking, the mechanisms available for satisfying dependences fall into two classes—*control sequencing* and *data synchronization*. In *control sequencing*, dependences are satisfied by using parallel or sequential control structures like `fork-join`, `doall`, **cobegin-coend**, and even sequential **begin-end**. In *data synchronization*, dependences are satisfied by using synchronization objects like binary semaphores, counting seamphores and I-structures. It is desirable for the compiler to automatically generate parallel code that correctly and efficiently satisfies all the program's dependences, using whatever control sequencing and data synchronization mechanisms are available for the target architecture. In this paper, we investigate the problem of generating maximally parallel code using only *fork* and *join* operations to correctly satisfy all the control and data dependences in the program. This problem is of interest when compiling for multiprocessor architectures and runtime systems where *fork-join* is the only mechanism, or the most efficient mechanism, available for satisfying dependences. We do not address the problem of trading off parallelism and overhead in this paper. Our approach is to use the algorithms described in this paper to expose the maximum amount of fork-join parallelism, and then use our previous work on automatic partitioning of parallel programs [SH86, Sar89b] to select the useful fork-join parallelism for a given multiprocessor system.

A *program dependence graph* [FOW87] consists of computation nodes connected by control dependence edges and data dependence edges. For each data dependence edge in the program dependence graph, we assume that its source and destination nodes are identically control dependent. The program dependence graph can then be viewed as a hierarchy of dags, so that the hierarchy reflects the program's control dependences, and for each set of identically control dependent nodes, there is a dag

Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation. White Plains, New York, June 20-22, 1990.

that reflects all the data dependences among those nodes. This assumption is valid for IF1 program graphs (IF1 [SG85] is a graphical intermediate form for strict applicative languages and is used as an intermediate form for programs written in the single-assignment language, SISAL [MSA*85]). For program graphs derived from imperative languages like Fortran, it is possible for the source and destination nodes of a data dependence edge to have different control conditions. However, it has been shown that all data dependences in such a program graph can be satisfied by inserting "sequencing edges" between identically control dependent nodes [CHH89]. In this case, we can also view the program dependence graph as a hierarchy of dags, except that the edges of the dag should correspond to the sequencing edges defined in [CHH89], rather than the original data dependence edges defined in the program dependence graph.

In a fork-join model, the only operations available for expressing parallelism are *fork* (spawn a node's execution as a new thread of control) and *join* (wait for all previously forked threads to complete). Let us consider the problem of generating maximally parallel fork-join code for a single dag, where the nodes are labelled with execution times. Once we know how to solve the problem for a single dag, we can apply the algorithm separately to each dag in the hierarchy defined by the program dependence graph. Since we are interested in generating maximal parallelism, we assume that every node in the dag is forked as a new thread of control. The problem then becomes one of selecting a topological node ordering of the dag, and deciding where the *join* operations should be placed.

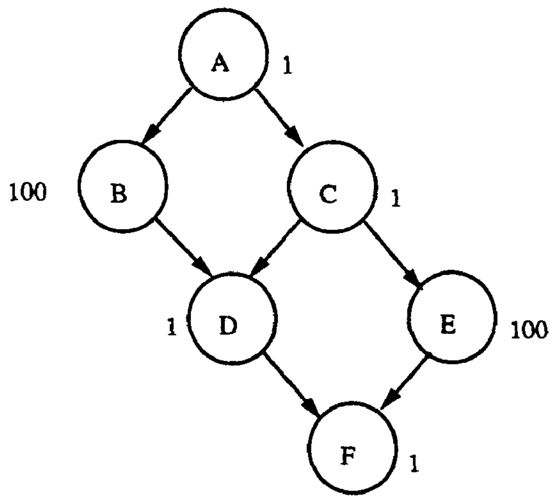

Figure 1. Example dag

For example, consider the dag shown in Figure 1, with nodes $A, \ldots, F$ labelled with their execution times. The maximally parallel fork-join code for this dag is

fork A ; *join* ; *fork* C ; *join* ; *fork* B ; *fork* E ;
join ; *fork* D ; *join* ; *fork* F ; *join* ;

Remember that a *join* operation waits for all previous *fork* operations to complete. The ideal parallel execution time of the above fork-join code on a multiprocessor with zero overhead and at least 2 processors is $1 + 1 + 100 + 1 + 1 = 104$. If we performed no reordering on the original node sequence, $A, \ldots, F$, the best possible fork-join code would be

fork A ; *join* ; *fork* B ; *fork* C ; *join* ; *fork* D ;
fork E ; *join* ; *fork* F ; *join* ;

with ideal parallel execution time $= 1+100+100+1 = 202$, which is close to the sequential execution time ($= 204$). This illustrates the importance of instruction reordering for fork-join parallelism.

At a more abstract level, the problem can be viewed as partitioning the dag into a sequence of blocks, such that

1. if there is an edge in the dag from node i to node j, then node i must belong to an earlier block than node j, and

2. the sum, $\sum_{\text{block } k}$ (Execution time of the largest node in block k), which gives the ideal parallel execution time, is minimized.

Each block in the block sequence corresponds to a set of *fork* operations, followed by a *join* operation. For example, the block sequences for the two fork-join code fragments shown above are $<\{A\},\{C\},\{B,E\},\{D\},\{F\}>$ and $<\{A\},\{B,C\},\{D,E\},\{F\}>$ respectively.

Note that the critical path length of the dag in Figure 1 is 103, which is less than the cost of the optimal block sequence (= 104). The critical path length is the ideal parallel execution time using general synchronization (rather than just fork-join) on a multiprocessor with zero overhead and an unbounded number of processors. This illustrates that, in the worst case, we may be forced to give up some parallelism because of using only fork-join synchronization. However, our experimental results on real program graphs show that the ideal fork-join parallel execution time is usually very close to the critical path length of the dag.

Our interest in the reordering problem for fork-join parallelism stems from its application in the parallel execution of SISAL [MSA*85] programs, using the Optimizing Sisal Compiler (OSC) developed at Colorado State University and Livermore [OC88, Can89]. OSC provides a portable, fork-join implementation of SISAL [CLOS87, Can89], based on its own microtasking runtime system. The current implementation of OSC performs no instruction reordering to enhance fork-join parallelism. Instead, it just uses the (arbitrary) topological ordering produced by the SISAL front-end and IF1 optimizations [SW85]. A *join* is simply placed before each node that depends on any earlier *fork* node located after the previous *join*. So, for the dag in Figure 1, OSC's approach would choose the block sequence, $<\{A\},\{B,C\},\{D,E\},\{F\}>$, which has a cost of 202, as discussed earlier. It is therefore necessary *to perform node reordering to expose the maximum* fork-join parallelism in the SISAL program. The reordering algorithms described in this paper have been implemented in an extended version of the OSC compiler, called POSC (Partitioning and Optimizing Sisal Compiler) [SC90].

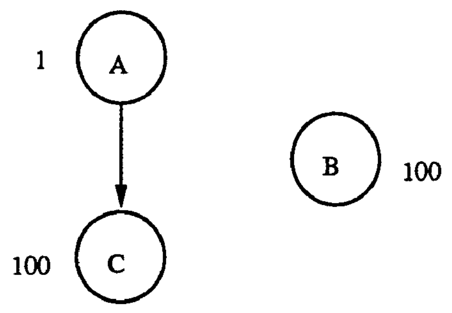

Figure 2. Another example dag

Even if we were not allowed to perform any node reordering, an interesting observation is that the selection of *join* locations can still impact the ideal fork-join parallel execution time. Consider the dag shown in Figure 2 with nodes A, B, C labelled with their execution times. If we use OSC's approach for placing *join* operations, we obtain the block sequence, $<\{A,B\},\{C\}>$, which has a cost of $100 + 100 = 200$. However, we could use the same node ordering with a different placement of the *join* operation to obtain a better block sequence, $< \{A\}, \{B, C\} >$, which has a cost of $1+100 = 101$.

Having made these observations about the effect of node reordering and the placement of *join* operations on the amount of fork-join parallelism, we summarize the major results of this work:

1. An optimal algorithm for inserting *join* operations in a given node ordering (Section 3). This algorithm takes $O(m \times (m + e))$ time and is based on a dynamic programming algorithm by Kernighan [Ker71] (m and e are the number of nodes and edges in the dag).

2. An approximation algorithm for the general (NP-complete) problem of node reordering and placement of *join* operations (Section 4). This algorithm takes $O(m \times (m + e))$ time and is observed to be optimal or nearly optimal in practice. It is also provably optimal for some interesting special cases.

3. An implementation of the approximation algorithm on IF1, a graphical intermediate representation for SISAL (Section 5).

4. Experimental results based on ideal, simulated parallel execution times of real SISAL programs, to compare OSC's current strategy with our approximation algorithm (Section 6).

The rest of the paper is organized as follows. Section 2 formally describes the problem of partitioning a dag to maximize fork-join parallelism. Sections 3, 4, 5 and 6, contain the major results of this work outlined above. Section 7 wraps up with conclusions and a discussion of future work.

2 Problem Statement

Given a directed acyclic graph (dag), $DG \subseteq N \times N$, on a set of nodes, $N = \{1, ..., m\}$, where node i has execution time $t_i \geq 0$, we are interested in finding a sequence of blocks, $< B_1, ..., B_k >$, such that

1. the blocks completely partition the set of nodes i.e. $\bigcup_i B_i = N \wedge B_i \cap B_j = \emptyset \, \forall \, i \neq j$, and
2. $\exists \, (x, y) \in DG \Rightarrow \exists \, i, j$ such that $x \in B_i \wedge y \in B_j \wedge i < j$, and
3. $COST(< B_1, ..., B_k >) = \sum_i \max_{j \in B_i}(t_j)$, is minimized.

3 Optimal Partition for a given Node Ordering

In the example from Figure 2, we observed that the selection of *join* locations can impact the ideal fork-join parallel execution time, even if we are not allowed to perform any node reordering. In particular, the simple approach used by OSC in selecting *join* locations is sub-optimal. In this section, we present an optimal algorithm for selecting *join* locations, for a given ordering of the nodes in the dag. This algorithm can be useful in a compiling system where node reordering is not permitted (perhaps due to some external constraint), but where it is important to expose the the maximum possible fork-join parallelism. This algorithm can also be used as a post-pass to the approximation algorithm described later in Section 4, by attempting to improve the placement of *join* operations, if possible, for the selected node ordering.

Figure 3 outlines a dynamic programming algorithm that finds an optimal block sequence for

```
$C_{opt}[0] := 0$ ;
for $j := 1$ to $m$ do
/* INVARIANT: we have already determined optimal partitions for the $j-1$ subgraphs, $1\ldots 1$, $1\ldots 2$, $\ldots$,
$1\ldots j-1$. These partitions are encoded in $PREVBREAK[2\ldots j]$ and $C_{opt}[0\ldots j-1]$. */
  1. $C_{min} := +\infty$ ; $t_{max} := 0$ ; $pred_{max} := 0$ ;
  2. $i := j$ ;
     repeat
     /* Evaluate the partition of $1\ldots j$, using the previously determined optimal partition of $1\ldots i-1$
     along with the block $\{i, \ldots, j\}$. */
      (a) $t_{max} := max(t_{max}, t_i)$ ; /* Now $t_{max} = max(\{t_x \mid i \leq x \leq j\})$ */
      (b) if $t_{max} + C_{opt}[i-1] < C_{min}$ then
            i. $C_{min} := t_{max} + C_{opt}[i-1]$ ; $BESTINDEX := i$ ;
          end if ;
      (c) for each predecessor $x$ of node $i$ in the dag do
            i. $pred_{max} := max(pred_{max}, x)$ ;
          end for ; /* Now $pred_{max} = max(\{x \mid \exists \, \text{edge} \, x \rightarrow k \, \text{in the dag and} \, i \leq k \leq j\})$ */
      (d) $i := i - 1$ ;
     until $(i = pred_{max})$ ;
  3. $C_{opt}[j] := C_{min}$ ; $PREVBREAK[j+1] := BESTINDEX$ ;
end for ;

/* Print out the sequence of blocks in reverse order. */
$j := m + 1$ ;
repeat
  1. $prev := PREVBREAK[j]$ ; print "Start of block" ;
  2. repeat  $j := j - 1$ ; print $j$ ; until $j = prev$ ;
  3. print "End of block" ;
until $j = 1$ ;
```

Figure 3: Optimal algorithm for a given node ordering

a given node ordering (say $1 \ldots m$) of nodes in the dag. This is a modified and extended version of the algorithm presented by Kernighan in [Ker71]. The proof of optimality hinges on the loop invariant for the outer loop (on index variable j). The loop invariant states that, at the start of the j iteration, we already have optimal block sequences for the $j-1$ subgraphs of the dag defined by the subranges $1 \ldots 1$, $1 \ldots 2$, ..., $1 \ldots j-1$. Since we are given a fixed node ordering, we can find an optimal block sequence for the subgraph $1 \ldots j$ by examining all locations $i \leq j$ as possible *join* locations.

The sub-array $PREVBREAK[2 \ldots j]$ encodes $j-1$ optimal partitions of the $j-1$ subgraphs, $1 \ldots 1$, $1 \ldots 2$, ..., $1 \ldots j-1$ as follows. Consider the subgraph $1 \ldots x$, $1 \leq x \leq j-1$ (with edges given by $DG \cap (\{1, \ldots, x\} \times \{1, \ldots, x\})$). Let the optimal partition of $1 \ldots x$ contain k blocks, so that it can be represented by the block sequence, $< B_1, \ldots, B_k >$. The blocks can then be defined by using the following recurrence to identify the *join* locations,

$$J_k = x+1 \; ; \; J_i = PREVBREAK[J_{i+1}] \; \forall \; 0 \leq i \leq k-1$$

so that block $B_i = \{J_{i-1}, \ldots, J_i - 1\}$. Thus, $PREVBREAK[2 \ldots j]$ contains backward pointers which define the *join* locations for all $j-1$ optimal partitions, ending at $J_0 = 1$. Further, the cost of the optimal partition of subgraph $1 \ldots x \; \forall \; 1 \leq x \leq j-1$ is stored in $C_{opt}[x]$. Note that steps 2.a to 2.d in the first **repeat-until** loop will only be executed for values of $i > pred_{max}$. This condition ensures that only independent (concurrent) nodes will be placed in the same block.

Theorem 3.1 *The invariant described in the outer* **for** $j := 1$ **to** m **do** *loop (Figure 3) is always true at the start of each iteration: for all* i *such that* $1 \leq i < j$, *an optimal partition for subgraph* $1 \ldots i$ *is given by the chain of reverse pointers starting with* $PREVBREAK[i+1]$, *and has cost* $= C_{opt}[i]$.

Proof: By induction on j. Details have been omitted due to space limitations. A similar proof was presented in [Ker71]. □

Theorem 3.1 implies that an optimal block sequence for the entire dag is computed at the end of the outer **for** loop, where the invariant should be evaluated with $j = m+1$. The optimal solution has cost $= C_{opt}[m]$, and is obtained by following the chain of backward pointers starting at $PREVBREAK[m+1]$, as illustrated by the **repeat-until** loops at the end of Figure 3.

Kernighan's algorithm was originally designed to partition the instructions in a computer program into virtual memory pages, so as to minimize the number of transitions between pages. Kernighan's algorithm takes $O(m+e)$ time, where m is the number of instructions and e is the number of edges in the instruction-level transition graph (or control flow graph). However, the page size contributes to the constant factor in the $O(m+e)$ term, and is significantly large in practice. Our algorithm presented in Figure 3 takes $O(m \times (m+e))$ time in the worst case, because a block in the optimal partition may be arbitrarily large.

4 Approximation Algorithm for the General Problem

In this section, we address the problem of finding an optimal block sequence, when reordering is permitted. This is the general problem discussed in the Introduction, and formally stated in Section 2. Like many other sequencing and scheduling problems, this problem can be shown to be NP-complete. However, the NP-completeness proof is beyond the scope of this paper.

Since the optimal block sequencing problem is NP-complete, we have developed an approximation algorithm (outlined in Figure 4), that is observed to be optimal or close to optimal in practice. This is a greedy algorithm which forms a block by first selecting the node with the largest execution time (= node i in step 1), and then iteratively placing other concurrent nodes (= node j in step 4.a) in the same block so as to keep the ideal parallel execution time as small as possible. When there are no other concurrent nodes (i.e. when $ParallelSet = \emptyset$ in step 4), the algorithm repeats this process by starting a new block with a new node i in step 1. The algorithm terminates when all nodes have been assigned to blocks (i.e. when $RemainingNodes = \emptyset$). Looking at the dag in Figure 1, we see that this algorithm will first form a block with nodes B and E, which then leads to the optimal block sequence discussed in the Introduction.

Let us now discuss the algorithm in Figure 4 in some more detail. It first initializes $RemainingNodes$ to the set $\{1, \ldots, m\}$, and DG and $INVDG$ to forward and reverse adjacency list representations for the edges in the dag. Step 1 in the outer **while** loop sets i to the node with the largest execution time (of all nodes that have not been assigned as yet). Step 2 calls procedure DetermineTimes (outlined in Figure 5) to compute $DAGCP$ (the critical path length of dag DG), $EST[*]$ (the earliest starting times) and $LCT[*]$ (the latest completion times). $T_{PRED} = EST[i]$ and $T_{SUCC} = DAGCP - LCT[i]$ then give the ideal parallel execution times of node i's (immediate and non-immediate) predecessors and successors respectively. Step 3 computes $ParallelSet$, the set of nodes that can execute concurrently with node i. The notation DG^* and $INVDG^*$ is used in steps 3 and 4.c to represent transitive closures of DG and $INVDG$. However, the transitive closures of DG and $INVDG$ are not explicitly computed in the algorithm, otherwise they would have to be recomputed each time DG and $INVDG$ are updated in step 5. Instead, $DG^*[i]$ represents a depth-first search of dag DG to enumerate all nodes reachable from node i. Similarly for $INVDG^*[i]$.

Node i represents a new block, and each execution of step 4.a selects a new node j that should be placed in the same block as i. Node j is chosen so as to minimize the expression $\max(T_{PRED}, EST[j]) + \max(T_{SUCC}, DAGCP - LCT[j])$, which in turn minimizes the parallel execution time of all nodes in $CurBlock \cup \{j\}$ along with their predecessors and successors. Note that node i must have the largest execution time of all nodes in the set $CurBlock \cup \{j\}$, so the contribution from this set remains constant as we select a new node j in each execution of step 4.a. When no more nodes can be added to i's block, step 5 updates the data structures so that DG and $INVDG$ contain a reduced dag obtained by collapsing all nodes in i's block into node i.

```
$RemainingNodes := \{ 1, \ldots, m \}$ ;
for $i := 1$ to $m$ do
  $DG[i] := \{ x \mid i \rightarrow x$ is an edge in the input dag $\}$ ;
  $INVDG[i] := \{ x \mid x \rightarrow i$ is an edge in the input dag $\}$ ;
end for

while $RemainingNodes \neq \emptyset$ do
   1. $i :=$ node in $RemainingNodes$ with the largest value of $t_i$ ;
      $Block[i] := i$ ; $CurBlock := \{ i \}$ ; /* Start a new block with node $i$ */
   2. /* Compute $DAGCP$, $EST[*]$, $LCT[*]$, $T_{PRED}$ and $T_{SUCC}$ */
      call DetermineTimes($m, t, DG, INVDG, DAGCP, EST, LCT$) ;
      $T_{PRED} := EST[i]$ ; $T_{SUCC} := DAGCP - LCT[i]$ ;
   3. $ParallelSet := RemainingNodes - (DG^*[i] \cup INVDG^*[i])$ ;
   4. while $ParallelSet \neq \emptyset$ do
        (a) $j :=$ node in $ParallelSet$ with the smallest value of
            $\max(T_{PRED}, EST[j]) + \max(T_{SUCC}, DAGCP - LCT[j])$ ;
        (b) $Block[j] := i$ ; $CurBlock := CurBlock \cup \{ j \}$ ; /* Put $j$ in the same block as $i$ */
            $T_{PRED} := \max(T_{PRED}, EST[j])$ ; $T_{SUCC} := \max(T_{SUCC}, DAGCP - LCT[j])$ ;
        (c) $ParallelSet := ParallelSet - (DG^*[j] \cup INVDG^*[j])$ ;
      end while
   5. /* Update $DG[*]$ and $INVDG[*]$ */
      $DG[i] := \bigcup_{j \in CurBlock} DG[j]$ ; $INVDG[i] := \bigcup_{j \in CurBlock} INVDG[j]$ ;
      for $x \in RemainingNodes$ do
        (a) if $DG[x] \cap CurBlock \neq \emptyset$ then $DG[x] := (DG[x] - CurBlock) \cup \{ i \}$ ;
        (b) if $INVDG[x] \cap CurBlock \neq \emptyset$ then $INVDG[x] := (INVDG[x] - CurBlock) \cup \{ i \}$ ;
      end for
   6. $RemainingNodes := RemainingNodes - CurBlock$ ;
end while
```

Figure 4: Approximation algorithm for selecting a good block sequence

Then, the next execution of step 1 chooses a new value of i to start a new block.

Each of steps 1, 2, 3, 4.a, 4.b, 4.c, 5 and 6, take at most $O(m+e)$ execution time. This is true for the call to procedure DetermineTimes in step 2, as well as the depth-first searches required to compute $DG^*[i]$, $INVDG^*[i]$ in step 3, and $DG^*[j]$, $INVDG^*[j]$ in step 4.c. Since each of the aforementioned steps are performed $O(m)$ times, the entire algorithm takes $O(m \times (m+e))$ time.

The algorithm in Figure 4 is an approximation algorithm, designed to work well on real program graphs. So far, we have not been able to prove that it has a worst-case constant performance bound. However, the experimental results presented in Section 6 clearly demonstrate that this greedy algorithm works well on real program graphs. To further justify the algorithm, we prove that it is optimal in two special cases that represent interesting boundary conditions.

First, we consider a dag in which each node has execution time $=0$ or 1, and all nodes with execution time $=1$ can be executed concurrently. In this case, the greedy algorithm will place all nodes with execution time $=1$ in the same block, so as to yield the optimal parallel execution time ($=1$). The algorithm must start by choosing one of the nodes with execution time $=1$ as node i in step 1. It must also eventually choose all the remaining nodes with execution time $=1$ as node j in step 4.a, though it may place some nodes with execution time $=0$ in the same block as well. This case can model the dags in Figures 1 and 2, if we observe that an execution time $=1$ is negligible compared to an execution time $=100$, and replace the old execution times by the new (relative) execution times of 0 and 1 respectively.

Next, we consider a dag in which all nodes have the same execution time ($=1$, say). In this case, it is well known that a simple level decomposition provides an optimal solution, with a parallel execution time that is equal to the critical path length of the dag. However, a level decomposition is a poor strategy to use in general, as can be seen from the example dags in Figures 1 and 2. We have already seen that our algorithm is optimal for those examples, and we now show that it is also optimal for the unit execution time case. In this case, we can establish the following invariant each time the **while** condition in step 4 is evaluated: $T_{PRED} + T_{SUCC} + 1 \leq CP$, where CP is the critical path length of the original dag. The invariant must be true when T_{PRED} and T_{SUCC} are first initialized in step 2 (also note that node i may be any node from the set $RemainingNodes$, since all execution times are equal). For proof by contradiction, assume that some j is chosen in step 4.a such that the invariant is violated after T_{PRED} and T_{SUCC} are updated in step 4.b. Then, node j must have some predecessor or successor node $k \in ParallelSet$, which would have yielded $T_{PRED} + T_{SUCC} + 1 \leq CP$ if k had been chosen instead of j in step 4.a. Therefore $\max(T_{PRED}, EST[j]) + \max(T_{SUCC}, DAGCP - LCT[j])$ must have been greater than $\max(T_{PRED}, EST[k]) + \max(T_{SUCC}, DAGCP - LCT[k])$ in step 4.a, and node k should have been chosen instead of node j, which gives the desired contradiction.

Inputs:

1. A set of nodes $1 \ldots m$.
2. An execution time mapping $t[1 \ldots m]$.
3. DG, an array of precedence edges representing a dag. We assume that the sequence $1 \ldots m$ is a valid topological sort of DG.
4. $INVDG$, the inverse of DG.

Outputs:

1. $DAGCP$, the critical path length of dag DG.
2. An *earliest starting time* mapping, $EST[*]$.
3. A *latest completion time* mapping, $LCT[*]$.

```
procedure DetermineTimes($m, t, DG, INVDG, DAGCP, EST, LCT$)
  1. /* Compute $EST[*]$ */
     (a) for $i := 1$ to $m$ do $EST[i] := 0$ end for ;
     (b) for $j := 1$ to $m$ do
         for each $i \in INVDG[j]$ do
           i. $EST[j] := \max(EST[j], EST[i] + t[i])$ ;
         end for
  2. /* Compute $DAGCP$ */
     (a) $DAGCP := 0$ ;
     (b) for $i := 1$ to $m$ do $DAGCP := \max(DAGCP, EST[i] + t[i])$ end for
  3. /* Compute $LCT[*]$ */
     (a) for $i := 1$ to $m$ do $LCT[i] := DAGCP$ end for ;
     (b) for $i := m$ downto 1 do
         for each $j \in DG[i]$ do
           i. $LCT[i] := \max(LCT[i], LCT[j] - t[j])$ ;
         end for
end procedure
```

Figure 5: Procedure DetermineTimes

5 The SISAL Implementation

In this section, we discuss a specific implementation of the algorithm in Section 4 for reordering nodes in IF1 [SG85], a graphical intermediate representation for SISAL [MSA*85] programs. The reordering algorithm is performed as a pass in the POSC compiler [SC90]. An important prerequisite for the reordering algorithm is that the IF1 nodes be labelled with execution times. In previous work [Sar89a, Sar89b], we developed a framework for determining average execution times in a program. The approach is based on automatic execution profiling: the runtime system automatically gathers frequency information from previous executions of the program, and the frequency information is used by the compiler to derive average execution times. In the original SISAL implementation [Sar89b], execution profiling was implemented as an extension to the IF1 debugger, DI [SYO87]. Recently, the OSC compiler has been extended to optionally perform execution profiling in the compiled code [SC90].

The frequency information obtained from automatic execution profiling is stored as node pragmas in IF1. Each subgraph of a compound node is labelled with a frequency value, which gives the average number of times the subgraph is executed in a single execution of the parent compound node. These frequency values are then combined with execution time values of simple nodes, to determine average execution times of all compound nodes. The algorithm for determining average execution times is inter-procedural, so that the execution time determined for a function is passed on to all its call sites. Recursive functions are also handled in this framework [Sar89b]. The average execution times are also stored as IF1 node pragmas.

Given an IF1 program, annotated with pragmas for average frequencies and execution times, we are now ready to apply the algorithm from Section 4. IF1 is a hierarchical graphical representation (like a "tree of dags"), since a dag may contain both compound nodes and simple nodes, and a compound node itself contains subgraphs which are dags. The algorithm from Section 4 is applied separately to each dag in the hierarchy. For simplicity in the initial implementation, we just set a node's cost to its average *sequential* execution time. A more accurate approach would be to process the graphs in a bottom-up traversal of the hierarchy, and set a node's cost to be its average, ideal, fork-join parallel execution time. In future work, we intend to compare the results obtained by using sequential execution times as node costs, with those obtained by using ideal fork-join parallel execution times.

6 Experimental Results

In this section, we present experimental results based on the implementation described in Section 5. For these experiments, each SISAL program was processed by the following steps (outlined below as Unix commands on a SISAL program named `foo.sis`):

1. `osc -vtrace foo.sis ; s.out ;`
   `osc -FREQ foo.if1`
   Generate a sequential executable module, `s.out`, with extensions for execution profiling. Execute `s.out` and incorporate the frequency information into the IF1 file [Can89, SC90].

2. `if1costs foo.if1`

   Determine average execution times, and incorporate the information into the IF1 file [Sar89b].

3. `if1seq foo.if1`

   Perform node reordering and placement of *join operations*, based on the algorithm in Section 4. `if1seq` also has options to generate the block sequences that would be chosen by OSC's current approach in two cases:

   (a) using the topological ordering actually produced by the SISAL front-end and IF1 optimizations

   (b) using the "worst possible" topological ordering (chosen automatically by a heuristic algorithm); this is a valid "adversary" approach, since the SISAL front-end is free to choose any topological ordering of the nodes when generating IF1.

4. `doideal foo`

   Perform an ideal fork-join simulation, using the fork-join structure chosen by `if1seq`. The ideal simulation is for a multiprocessor with zero overhead and an unbounded number of processors, and is performed by using an extended version of the IF1 debugger, DI [Sar89b, SYO87].

For the purpose of this paper, we are interested in comparing simulated ideal parallel execution times obtained for different node reorderings on real SISAL programs. So, for each program, we present four different parallel execution times:

1. CPL = the critical path length of the program i.e. the ideal parallel execution time assuming general (rather than just fork-join) synchronization.

2. FJ_{new} = the ideal fork-join parallel execution time obtained by using the reordering algorithm in Section 4.

3. FJ_{orig} = the ideal fork-join parallel execution time obtained by using OSC's current approach on the ordering produced by the SISAL system (see option (a) above for `if1seq`).

4. FJ_{bad} = the ideal fork-join parallel execution time obtained by using OSC's current approach on the heuristically chosen "worst possible" reordering (see option (b) above for `if1seq`).

Note that CPL is based on general synchronization, and will never be larger than FJ_{new}, FJ_{orig}, or FJ_{bad}, all of which are based on the more restrictive fork-join synchronization. The amount by which FJ_{new} is smaller than FJ_{orig} and FJ_{bad} indicates the improvement due to the reordering algorithm in Section 4. It is intractable to execute an exponential-time optimal algorithm for comparison with results from our approximation algorithm. However, CPL is a lower bound on the optimal block sequence cost, so we can judge how well the approximation algorithm is doing by observing if FJ_{new} and CPL are close to each other or not.

Table 1 shows the ideal parallel execution times defined above (CPL, FJ_{new}, FJ_{orig}, FJ_{bad}) for five different SISAL programs. For comparison, we also included the total sequential execution time, SEQ. We see that FJ_{new} is always equal or nearly equal to CPL, indicating that our approximation algorithm is essentially optimal for these programs. Further, FJ_{new} is significantly smaller than both FJ_{bad} and FJ_{orig}, which indicates that there is a substantial

Program	Inputs	CPL	FJ_{new}	FJ_{orig}	FJ_{bad}	SEQ
Cocke-Younger-Kasami	Input string with 20 symbols	796	798	950	952	57732
Fast Fourier Transform	Array of 128 real numbers	1250	1269	1448	2627	176270
Fibonacci	Solve $f_n = f_{n-1} + f_{n-2}$ for $n = 15$	40	40	40	2437	3046
Matrix Multiplication	Two 20×20 real matrices	104	104	108	108	256048
Merge-exchange Sort	Array of 50 integers	767	771	1173	1761	82803

Table 1. Parallelism Measurements from ideal simulations of SISAL programs

benefit to be gained by reordering nodes for fork-join parallelism.

The largest benefit due to reordering was observed for the Fibonacci program in Table 1. This is because the only parallelism in the Fibonacci program is between the two recursive calls, unlike the other programs which also have loop-based parallelism (from *Forall* loops). In general, a bad node ordering for a recursive program with divide-and-conquer-style parallelism could force the two recursive calls to be sequentialized, whereas our reordering optimization would allow them to execute concurrently in a fork-join model.

7 Conclusions and Future Work

In this paper, we have studied the problem of generating fork-join code with the maximum amount of parallelism, for a given dag. The results of our work have been applied to SISAL programs, so as to expose more parallelism for the fork-join microtasking model supported by OSC. The reordering optimization was found to be crucial for programs which rely solely on non-loop parallelism (e.g. Fibonacci).

This paper did not address the issue of trading off overhead and parallelism and thus determining what parallelism is useful for the target multiprocessor. In previous work [SH86, Sar89b], we addressed the problem of automatically partitioning SISAL programs for a general macro-dataflow execution model. In the future, we plan to modify the macro-dataflow partitioner so that it can be targetted to OSC's fork-join execution model.

Another area of future work is to allow *nested blocks* in the block sequence, so that we are not forced to give up any parallelism in dags that are structured like nested series-parallel graphs. Nested blocks can be easily supported by OSC's microtasking system, which already supports nested fork-join parallelism due to compound nodes and function calls. One level of nesting will be automatically supported by the partitioner, which (for efficiency reasons) may require that some nodes be executed in the parent task, rather than being forked as separate tasks. We plan to study the optimization problem for arbitrary levels of block nesting, and assess its usefulness for real programs.

Acknowledgements

The author thanks Jeanne Ferrante for her comments and suggestions on this paper, and especially thanks Dave Cann for his assistance with the OSC implementation, which included extensions to perform automatic execution profiling. The author is also grateful for the computer system resources provided by the Computing Research Group at Livermore, which were used to obtain the experimental results presented in this paper.

References

[Can89] D. C. Cann. *Compilation Techniques for High Performance Applicative Computation.* PhD thesis, Colorado State University, Computer Science Department, Fort Collins, CO, 1989.

[CHH89] Ron Cytron, Michael Hind, and Wilson Hsieh. Automatic generation of dag parallelism. *Proceedings of the 1989 SIGPLAN Conference on Programming Language Design and Implementation*, 24(7):54–68, July 1989.

[CLOS87] David C. Cann, Ching-Cheng Lee, R. R. Oldehoeft, and S. K. Skedzielewski. SISAL *Multiprocessing Support.* Technical Report UCID-21115, Lawrence Livermore National Laboratory, Livermore, CA, 1987.

[FOW87] J. Ferrante, K. Ottenstein, and J. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 319–349, July 1987.

[Ker71] Brian W. Kernighan. Optimal sequential partitions of graphs. *JACM*, 18(1):34–40, January 1971.

[KKP*81] D. J. Kuck, R. H. Kuhn, D. A. Padua, B. Leasure, and M. Wolfe. Dependence graphs and compiler optimizations. *Conference Record of 8th ACM Symposium on Principles of Programming Languages*, 1981.

[MSA*85] J. McGraw, S. Skedzielewski, S. Allan, R. Oldehoeft, J. Glauert, C. Kirkham, B. Noyce, and R. Thomas. *SISAL: Streams and Iteration in a Single Assignment Language Reference Manual Version 1.2.* Manual M-146, Rev. 1, Lawrence Livermore National Laboratory, Livermore, CA, March 1985. No. M-146, Rev. 1.

[OC88] R. R. Oldehoeft and D. C. Cann. Applicative parallelism on a shared memory multiprocessor. *IEEE Software*, 5(1):62–70, January 1988.

[Sar89a] Vivek Sarkar. Determining average program execution times and their variance. *Proceedings of the 1989 SIGPLAN Conference on Programming Language Design and Implementation*, 24(7):298–312, July 1989.

[Sar89b] Vivek Sarkar. *Partitioning and Scheduling Parallel Programs for Multiprocessors. Research Monographs in Parallel and Distributed Computing*, Pitman, London and The MIT Press, Cambridge, Massachusetts, 1989. In the series, Research Monographs in Parallel and

Distributed Computing.

[SC90] Vivek Sarkar and David Cann. Posc — a partitioning and optimizing sisal compiler. *To appear in the Proceedings of the ACM 1990 International Conference on Supercomputing*, June 1990. Amsterdam, the Netherlands.

[SG85] S. Skedzielewski and J. Glauert. *IF1 – An Intermediate Form for Applicative Languages.* Manual M-170, Lawrence Livermore National Laboratory, Livermore, CA, July 1985. No. M-170.

[SH86] V. Sarkar and J. Hennessy. Partitioning parallel programs for macro-dataflow. In *Proceedings of the ACM Conference on Lisp and functional programming*, pages 202–211, August 1986.

[SW85] S. K. Skedzielewski and M. L. Welcome. Data flow graph optimization in IF1. In Jean-Pierre Jouannaud, editor, *Functional Programming Languages and Computer Architecture*, pages 17–34, Springer-Verlag, New York, NY, September 1985.

[SYO87] S. K. Skedzielewski, R. K. Yates, and R. R. Oldehoeft. DI: an interactive debugging interpreter for applicative languages. In *Proceedings of the ACM SIGPLAN 87 Symposium on Interpreters and Interpretive Techniques*, pages 102–109, June 1987.