Parallel Algorithms for Force Directed Scheduling of Flattened and Hierarchical Signal Flow Graphs

Pradeep Prabhakaran **Prithviraj Banerjee**

Center for Reliable and High-Performance Computing
Coordinated Science Laboratory
University of Illinois
1308 West Main St.
Urbana, Illinois 61801
(217) 333 6564
(pradeep,banerjee)@crhc.uiuc.edu

Abstract

In this paper, we present some novel algorithms for scheduling hierarchical signal flow graphs in the domain of high-level synthesis. With complex chips that need to be designed in the future, it is expected that the runtimes of these scheduling algorithms will be quite large. There are several key contributions of this paper. First, we develop a novel extension of the force-directed scheduling problem which naturally handles loops and conditionals by coming up with a scheme of scheduling *hierarchical* signal flow graphs. Second, we develop three new parallel algorithms for the scheduling problem. Third, our parallel algorithms are portable across a wide range of parallel platforms. We report results on a set of high-level synthesis benchmarks on 8-processor SGI Challenge, a 16 processor Intel Paragon, and a network of 4 SUN SPARCstation5 work stations. Finally, while some parallel algorithms for VLSI CAD reported by earlier researchers have reported a loss of qualities of results, our parallel algorithms produce exactly the same results as the sequential algorithms on which they are based.

KEYWORDS: High-level synthesis, force-directed scheduling, hierarchical graphs, parallel algorithms, multiprocessors, networks of workstations.

This research was supported in part by the National Science Foundation under grant MIP-9320854, the Semiconductor Research Corporation under grant SRC 95-DP-109, and the Advanced Research Projects Agency under contract DAA-H04-94-G-0273 administered by the Army Research Office. We would also like to thank Intel Corporation for the donation of an Intel Paragon to the University of Illinois.

1 Introduction

With the rapid improvement in VLSI technology, circuit design is becoming extremely complex and is placing increasing demands on CAD tools. Parallel processing is fast becoming an attractive solution to reduce the inordinate amount of time spent in VLSI circuit design [1]. Many workstation vendors have announced products based on multiprocessors. Furthermore, most CAD environements have large networks of workstations that can be used as parallel machines. It is clear that many CAD sofware vendors will soon be announcing products that will exploit the parallelism in the hardware. This has been recognized by several researchers in VLSI CAD as is evident in the recent literature for cell placement [16], [19], [20], circuit extraction [18], test generation [21], [22] and logic synthesis [24], [30], [31].

In this paper, we present some novel algorithms for scheduling hierarchical signal flow graphs in the domain of high-level synthesis. High level synthesis translates behavioral description of a circuit, specified as a signal flow graph to register transfer level description. With the increasing complexity of systems being designed, automated high level synthesis is becoming increasingly important.

Scheduling is a major step in high level synthesis. It assigns each node in the graph to a specific time step. Numerous algorithms have been proposed for scheduling [2], [4], [5], [7], [8]. An excellent overview of existing scheduling algorithms appears in [6]. An integer linear programming approach to scheduling is described in [7] and [8]. Integer linear programming is suitable for small systems, but may result in very large run times for larger problems. List scheduling [4], [5] is a fast algorithm which tries to find a minimal schedule, given the resource constraints. Force directed scheduling [2], introduced by Paulin and Knight tries to minimize the number of resources for a given latency by trying to smooth out the resource requirement for different time steps within a given latency. Force directed scheduling have been found to be extremely effective and has gained interest since its introduction among other research groups [9], [10], [11], [12], [13]. Force directed list scheduling (FDLS) [2],which is a variation of force directed scheduling in which the user specifies the resources available and the signal flow graph is scheduled with minimum latency given the resource constraints.

Scheduling algorithms for data flow graphs corresponding to small designs containing about 50-100 nodes can take several minutes to execute on conventional workstations. With complex chips that need to be designed in the future containing thousands of nodes, it is expected that the runtimes of these algorithms will reach hours. Designers of high level synthesis tools wish to explore a very large design space of solutions with various architectural alternatives (e.g. number of adders, multipliers, registers, buses, memory ports,etc.) for various values of expected latencies. The runtimes for exploring all the different choices can easily take several hours. This task is very interactive in nature, and a designer typically would want to have the solutions in less than a minute,and not hours. In addition, researchers are proposing more complex cost functions to produce designs with higher speeds, lower power and increased testability during the scheduling process. Hence, it is important to investigate efficient parallel algorithms for the problem which can reduce the runtimes from hours to minutes. The work described in this paper is part of an ongoing project called ProperCAD [15] which is aimed at developing an integrated suite of parallel applications for VLSI CAD that run on a variety of parallel platforms. In the past, various tools have been developed to solve the problems at lower levels in the VLSI CAD hierarchy, namely, placement [16], routing [17], circuit extraction [18], design rule checking [26], logic synthesis [27], [27], test generation [25], [29], fault simulation [28], and behavioral simulation [23]. In this paper, we address the problem of scheduling in high-level synthesis.

There are several key contributions of this paper. First, we develop a novel extension of the force-directed scheduling problem which naturally handles loops and conditionals by coming up with a scheme of scheduling *hierarchical* signal flow graphs. Second, we develop three novel parallel algorithms for the scheduling problem, one based on node-based partitioning, another based on time-step partitioning, and a final one based on hierarchical partitioning. Third, our parallel algorithms are portable across a wide range of parallel platforms which includes shared memory multiprocessors distributed memory message-passing multicomputers and networks of workstations. We report results on an 8-processor SGI Challenge shared memory multiprocessor, a 16 processor Intel Paragon distributed memory multicomputer, and a network of 4 SUN SPARCstation5 workstations. Fourth, we report actual implementations of these algorithms on real parallel machines, and report results of qualities and runtimes for various benchmark circuits on a variety of parallel machines. We show in the paper that we have obtained speedups of about 8 to 10 on 16 processors of an Intel Paragon. Finally, while some parallel algorithms for VLSI CAD reported by earlier researchers have reported on loss of qualities of results, our parallel algorithms produce exactly the same results as the sequential algorithms on which they are based.

In some related work, some researchers [14] have proposed a parallel system for distributed high level synthesis which uses coarse-grained parallelism to explore and evaluate many alternative VLSI designs efficiently. They propose a distributed version of force directed list scheduling in which each processor executes the FDLS algorithm on a separate module set. Force directed scheduling algorithm for minimizing resource requirement for a given latency assumes that the user specifies the latency for each loop body. In comparison to that work, we propose an extension of force directed scheduling algorithm which naturally handles loops and conditionals, and which automatically selects the latencies of the loop bodies. We assume that the overall latency of the signal flow graph is given, and attempt to minimize the resource requirement.

In Section 2 the basic force directed scheduling algorithm is summarized. In Section 3 we propose two approaches to implementing parallel force directed scheduling algorithm for non-hierarchical graphs. In Section 4 we propose a sequential force directed scheduling algorithm for hierarchical graphs. A parallel implementation of the above appears in Section 5. In Section 6 we give some experimental results, and conclude in Section 7.

2 Basic force directed scheduling

The basic force directed scheduling algorithm is an iterative algorithm which schedules one operation each iteration. The operation to be scheduled is selected based on a quantity termed $force$ defined for each step. It is a measure of concurrency at that step. The force directed scheduling tends to balance the concurrency at each step without lengthening the execution time. First, it calculates the time frame of each operation, namely the time interval from the earliest start time to the latest start time for that operation. It is done by calculating the *as soon as possible*$(ASAP)$ and *as late as possible*$(ALAP)$ schedule of the graph. At each step, the algorithm determines the force of each node to each step in its time frame. It then selects the node with the least force and schedules it to the corresponding step. The force consists of two components, $self$ and $predecessor\_successor(ps)$ forces. Self force of a node n to a step s is a measure of the increase in concurrency due to scheduling that node to that step. Predecessor and successor forces are measures of increase in concurrency of the predecessors and successors, respectively as a result of scheduling n to s.

The concurrency of operations can be captured by a *distribution graph* (DG) for each operation type. For each operation type k, the DG in step i is given by

$$DG(i) = \sum_{k} Prob(op, i)$$

the sum being taken over all operations of type k.The mechanical analog for DG is the spring constant.

The self force associated with the assignment of an operation with time frame from t to b to time step j $(t \leq j \leq b)$ is given by

$$self\_force(j) = \sum_{i=t}^{b} [DG(i) * x(i)]$$

where $x(i)$ is the change in operation probability (the mechanical analog being the displacement of spring), given by

$$x(j) = (h-1)/h$$

and

$$x(i) = -1/h$$

where $h = (b - t + 1)$.

The predecessor force for the assignment of a node to a step j is the sum of the forces of the predecessors of the node arising out of change in time frames and the resulting change in concurrency. For a particular predecessor, it is quantified as:

$$Force(nt, nb) = \sum_{i=nt}^{nb} [DG(i)/(nb - nt + 1)] - \sum_{i=t}^{b} [DG(i)/(b - t + 1)] \tag{1}$$

where the interval from t to b is the old time frame and that from nt to nb is the new time frame. The successor force is defined analogously.

The basic force directed scheduling algorithm is summarized below:

FORCE-DIRECTED()

repeat
    Evaluate time frames :
        2.1.*Find ASAP schedule*
        2.2.*Find ALAP schedule*
    Update distribution graphs
    calculate self, predecessor and successor forces,
    Add them to get the total force for each feasible control step
    Schedule operation with lowest force;
    Set its time frame equal to the selected step.
until *all operations scheduled*

2.1 Scheduling of conditionals and loops in the basic force directed scheduling scheme

The alternates of a conditional are mutually exclusive. Thus, for a step in which mutually exclusive operations intersect, the probability of only the operation with the highest probability is added to the corresponding DG. The force calculation and scheduling proceeds exactly the same way as described earlier for an acyclic graph without conditional structures.

When a loop is part of the behavioral description, the user have to specify a constraint on the loop iteration time or, alternately a constraint of the number of structural units available. For multiple embedded loops, the operations of the innermost loop are scheduled first, relative to the local timing constraint. After that, the entire loop is taken as a single operation with execution time equal to the loop's local time constraint.

Further details of the basic force directed scheduling can be found in [2].

3 Parallel force directed scheduling on non-hierarchical graphs

In this section, we will look at the force directed scheduling problem of simple acyclic graphs, corresponding to straight line code sequences in a behavioral description. We describe algorithms for graphs with conditionals and loops in sections 4 and 5. For parallel implementation of the basic force directed scheduling, we have taken two problem partitioning approaches, node-based and step-based. In the node based method, we partition the nodes among the processors, and in step-based method, we partition the time steps among the processors.

3.1 Node based problem partitioning

The nodes are partitioned among the processors.The partitioning is done by first calculating the total available work, which is approximated by

$$W = \sum_{i=0}^{numnodes} (ALAP(i) - ASAP(i) + 1)$$

and work per processor, given by $W/numprocs$. Each processor selects a set of nodes S such that $\sum_{s \in S}(ALAP(s) - ASAP(s))$ is approximately equal to the work per processor.

We can define a $local\_DG$ for each processor which is the DG calculated using only the nodes owned by that processor.The $global\_DG$ is obtained by combining the $local\_DGs$ of all the processors.

At each step, each node calculates the forces of each owned node to each feasible time step, and sends the information to a $master$ processor. The $master$ determines the node with the least force and schedules it to the step with the least force. It then broadcasts that information to all the processors. Each processor, upon receiving this information, updates the time frames of other nodes, and recalculates the $local\_DG$. The processors then perform a global reduction operation by which the $local_DG$s are combined together to get the $global\_DG$. Since the time frames of the nodes could change as a result of the schedule operation, the load function is re-computed, and nodes re-assigned to processors, to minimize load imbalance.

The node based parallel force directed scheduling algorithm is as follows:

PARALLEL-FORCE-DIRECTED-NODE()

```
repeat
        Compute the work per processor
```

Determine the list of nodes on current processor

Evaluate time frames :

4.1.*Find ASAP schedule*

4.2.*Find ALAP schedule*

Compute local distribution graph for the assigned nodes

Perform global reduction to get the overall distribution graph

Calculate self, predecessor and successor forces for the assigned nodes,

Add them to get the total force for each feasible control step

Send the operation with lowest force to the master

Receive the global best node and step;

Set its time frame equal to the selected step.

until *all operations scheduled*

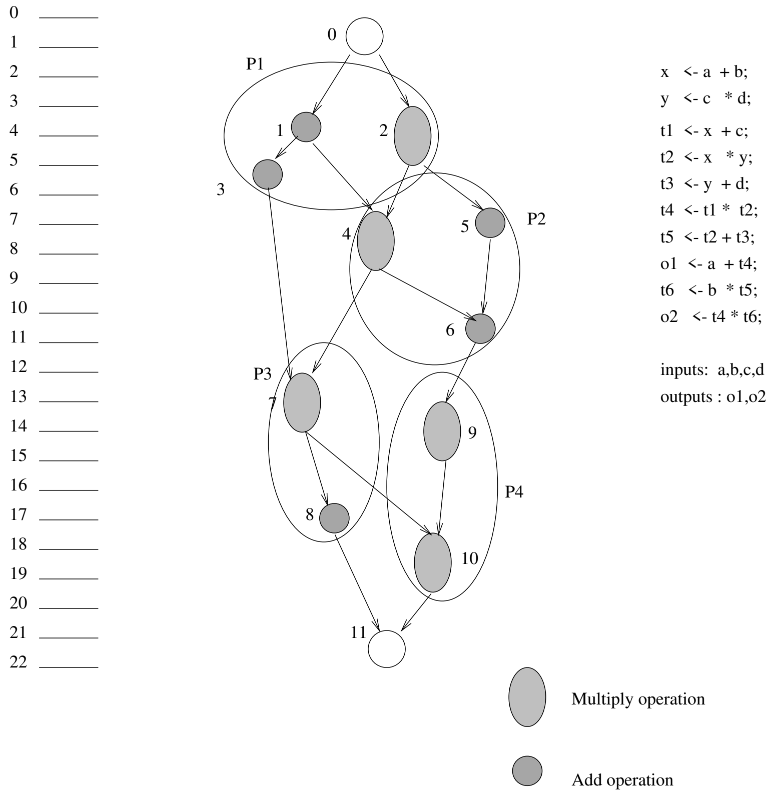


Figure 1: Node based decomposition for a non-hierarchical graph for 4 processors

Figure 1 shows an example showing an non-hiearchical graph and a sample node-based decomposition among four processors. We report experimental results on this algorithm in Section 6.

3.2 Step based problem partitioning

In this approach, the time steps are partitioned among the processors rather than the nodes. The partitioning is done by first calculating the total available work, which, as in the node based scheme, is approximated by

$$W = \sum_{i=0}^{numnodes} (ALAP(i) - ASAP(i) + 1)$$

and work per processor, given by $W/numprocs$. Each processor selects a set of steps T such that $\sum_s (ALAP(s) - ASAP(s) + 1)$, where the time frame of s intersects T, is approximately equal to the work per processor. Let S be the set of nodes whose timeframes intersect the set T of processor p.It calculates the forces of each node in S to each time step in T it intersects. It then sends the information to a $master$ processor which selects the least force node.

The master then schedules that node to the step with the minimum force. It then broadcasts the information. All other processors, on receiving the information, updates the time frames of other nodes, and recalculates the DGs of the steps in T. In contrast to the node-based approach, the step based approach does not have to perform a global reduction of DGs since it calculats the overall DGs of the steps it owns. As in the case of node-based approach, the steps are reassigned to processors after each iteration so that they get approximately equal amount of work.

The step based parallel force directed scheduling algorithm is as follows:

```
PARALLEL-FORCE-DIRECTED-STEP()
repeat
        Compute work - per - processor
        Determine the list of steps for the current processor
        Evaluate time frames :
            4.1.Find ASAP schedule
            4.2.Find ALAP schedule
        Compute distribution graph for the assigned steps
        Calculate self, predecessor and successor forces for the
        nodes which intersect the assigned steps,
        Add them to get the total force for each feasible control step
        Send the operation with lowest force to the master
        Receive the global best node and step; set its time frame
        equal to the selected step.
  until all operations scheduled
```

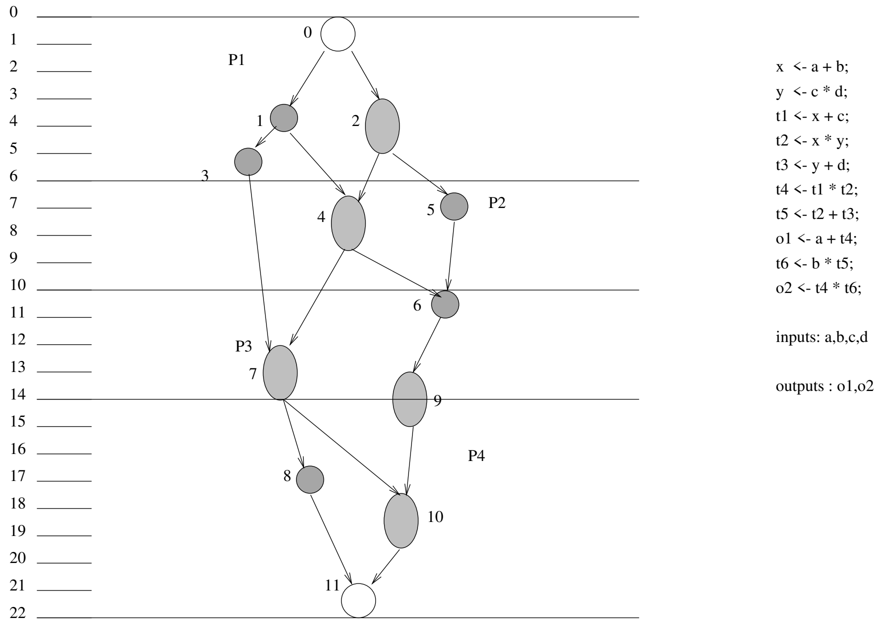


Figure 2: Step-based decomposition for a non-hierarchical graph for 4 processors

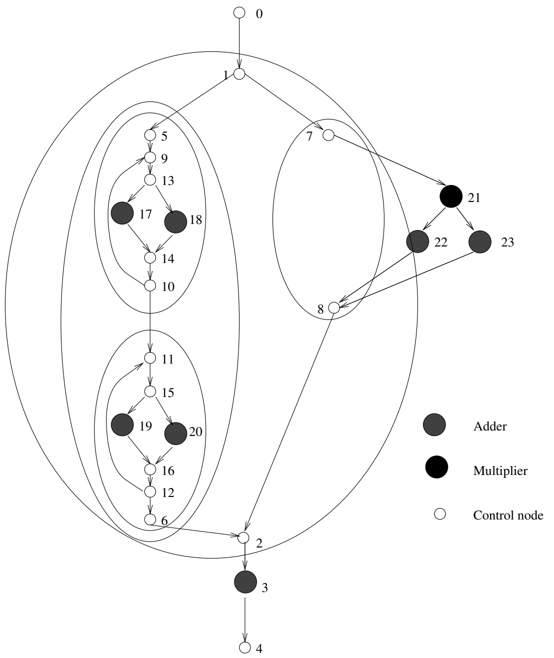
Figure 2 shows an example showing an non-hiearchical graph and a sample step-based decomposition among four processors. We show experimental results of this parallel algorithm in Section 6.

4 Hierarchical force directed scheduling

Previous approaches for force directed scheduling work for hierarchical signal flow graphs assume that the user has to specify the latency of each loop. We extend the basic force directed scheduling scheme to naturally handle hierarchical signal flow graphs, so that the latencies of loops are selected automatically to minimize the total resource requirement.

Definition 1 A hierarchical graph *is a signal flow graph in which each node can be*

- *a block, which is a set of nodes enclosed by a source and a sink*
- *atomic operation (eg an addition or multiplication) or*
- *a loop which encloses a block*
- *a conditional structure which encloses one or more $alternates$ each of which are blocks.*




```
if (x > y) then
 {
    for i := 1 to n do
      a[i] := c + d;
      b[i] := a + b;
    end for;
    for i := 1 to n do
      x[i] := a[i] + k;
      y[i] := b[i] + l;
    end for;
 }
else
 {
    t1 := c * d;
    o1 := t1 + a;
    o2 := t1 + b;
 }
```

Figure 3: A hierarchical signal flow graph

Figure 3 shows an example hierarchical signal flow graph. Node 0 is the source and node 4 is the sink. Node 1 is a conditional. Nodes 5 and 7 are the source nodes of the alternates for the conditional. Node 9 and 11 are source nodes for loops. Nodes 9 and 10 are the source and the sink, respectively of the loop body of the loop whose source is 5. The nodes of a hierarchical graph can be broadly classified as $control$ nodes and $non\text{-}control$ nodes. Control nodes include $source$ and $sink$ nodes which enclose hierarchical entities, $loop\_source$ and $loop\_sink$ which enclose a loop, and if_source and if_sink which enclose a conditional. Control nodes do not require resource allocation. The $loop\_source$ node contains information about the loop such as the number of iterations.The $non-control$ nodes are the regular nodes (eg adder) which require resource allocation. We assign a level number to each node. The level number of a node corresponds to the depth of nesting of that node in the signal flow graph. The nodes enclosed by the outermost $source - sink$ pair are assigned a level number 0. The next inner level nodes (those enclosed by hierarchical nodes having level number 0) are assigned a level number 1 and so on.

4.1 Distribution graph of a hierarchical entity

For each hierarchical entity h, we can associate a corresponding sub_DG, given by the tuple $sub\_DG_h^k = \prec P, lb, ub, k \succ$ for each operation type k where lb and ub are the lower and upper limits for the earliest and latest start times respectively for all nodes enclosed by h, P is a vector such that $P(i)$, $lb \leq i \leq ub$ is the sum of the probabilities over all nodes $n \in h$ that n will be executed during time step i, and k is the type of the operation (eg adder, multiplier). The sub_DG of a hierarchical entity depends on the type of the entity. The calculation of sub_DGs for the various types of hierarchical DGs are explained below. In the following, $sub\_DG_n^k(i)$ means the same as $P(i)$, the i^{th} element of the member P of the tuple $sub\_DG_h^k$.

4.1.1 Calculation of sub_DG for an atomic node

The sub_DG for an atomic node a of type k is given by $sub\_DG_a^k = \prec ADG_a, lb_a, ub_a, k \succ$, where $lb_a = ASAP(a)$, $ub_a = ALAP(a) + l_k$ and

$$ADG_a(i) = \frac{(\sum_{j=ASAP(a)}^{ALAP(a)} \alpha(i,j))}{(ALAP(a) - ASAP(a) + 1)}$$

where l_k is the latency of operation of type k, and

$$\alpha(i,j) = \begin{cases} 1 \text{ if } j \leq i < j + l \\ 0 \text{ otherwise.} \end{cases}$$

4.1.2 Calculation of sub_DG for a block

For a straight line block of code B, we calculate the sub_DG by recursively calculating the sub_DGs of each component node and adding them up for each step. Let $\prec P_j, lb_j, ub_j, k \succ$ be the sub_DG for node j, where $j \in B$. The overall sub_DG of the block for each operation type k is given by $sub\_DG_B^k = \prec BDG_B, lb_B, ub_B, k \succ$ where

$$lb_B = \min_{j \in B} lb_j$$

$$ub_B = \max_{j \in B} ub_j$$

and

$$BDG_B(i) = \sum_{j \in B} P_j(i)$$

4.1.3 Calculation of sub_DG for a loop

For a loop structure, we determine the maximum and minimum latencies of the loop body, recursively calculate the sub_DG of the loop body for each start time of the loop source and for each corresponding latency and compute their weighted sum, the weights being the probability that the loop body is assigned that latency and the loop source has that particular start time. We assume that all the possible start times of the source and all latencies for that particular start time are equally probable.

Let L be a loop structure which executes for n iterations.Let e_{src} and l_{src} be the earliest and the latest start time of the source node of L and e_{snk} and l_{snk} be the earliest and latest start times of the sink of L respectively. For a particular start time s, $s \leq (l_{snk} - n * (e_{snk} - e_{src}))$ for the loop source, the maximum latency of the loop body is given by

$$l_{max} = (l_{snk} - s)/n$$

and the minimum latency by

$$l_{min} = (e_{snk} - e_{src})/(n-1)$$

.

Let $\prec BDG^l_B, lb_B, ub_B, k \succ$ be the sub_DG for the body of the loop for a given latency l and for operation type k. Then the sub_DG for L, denoted by $sub\_DG^{s,l}_L$ for start time s for the loop source and latency l and is given by

$$sub\_DG^{s,l}_L = \prec LDG^{s,l}_L, lb^{s,l}_L, ub^{s,l}_L, k \succ$$

where $LDG^{s,l}_L(i) = BDG_B(i \bmod l)$, $lb^{s,l}_L = s$ and $ub^{s,l}_L = s + n \times l$. The overall sub_DG for L, is given by the tuple $sub\_DG_L = \prec LDG_L, lb_L, ub_L, k \succ$, where $lb_L = e_{src}$, $ub_L = l_{snk}$ and

$$LDG(i) = \sum_{s=e_{src}}^{l_{src}} \frac{1}{(l_{src} - e_{src} + 1)} \sum_{l=l_{min}}^{l_{max}} \frac{LDG^{s,l}_L(i)}{(l_{max} - l_{min} + 1)}$$

4.1.4 Calculation of sub_DG for conditional structure

For a conditional structure C, we recursively calculate the sub_DGs of each alternate for the conditional. The overall sub_DG is the DG formed by taking the stepwise maximum of the above sub_DGs at each step. Let the conditional structure have m alternates. Let $\prec P_j, lb_j, ub_j, k \succ$ be the sub_DG for the j^{th} alternate. Then, the overall DG of the conditional structure is given by $sub\_DG_C = \prec CDG_C, lb_C, ub_C, k \succ$ where

$$lb_C = \min_{j=1}^{m} lb_j$$

$$ub_C = \max_{j=1}^{m} ub_j$$

and

$$CDG_C(i) = \max_{j=1}^{m} P_j(i)$$

for each operation type k.

4.2 The scheduling step

The hierarchical force directed algorithm proceeds level by level, scheduling the nodes at each higher level before proceeding to the next lower level. Consider a node n. Let $Succ(n,k)$ be the set of successors of n of type k and let $Prec(n,k)$ be the set of predecessors of n of type k which are in the same level as n in the signal flow graph. Define $Succ\_DG(n,k,s)$ to be the aggregate DG of all nodes of type k in $Succ(n,k)$ when n is scheduled to step c. Similarly, $Pred\_DG(n,k,s)$ is the aggregate DG of all nodes in $Pred(n,k)$ when n is scheduled to step c. Let $\prec sub\_DG_j, lb_j, ub_j, k_j \succ$ be the sub_DG of node j. Then, $Succ\_DG(n,k,c)$ is given by

$$Succ\_DG(n,k,c) = \prec SDG_n, lb_n^s, ub_n^s, k \succ$$

where $lb_n^s = \min_{t \in Succ(n,k)} lb_t$, $ub_n^s = \max_{t \in Succ(n,k)} ub_t$, and $SDG_n(j) = \sum_{t \in Succ(n,k)} sub\_DG_t(j)$. $sub\_DG_t(j)$ is assumed to be zero if $j > ub_t$ or $j < lb_t$.

Similarly,

$$Pred\_DG(n,k,c) = \prec PDG_n, lb_n^p, ub_n^p, k_n \succ$$

where $lb_n^s = \min_{t \in Pred(n,k)} lb_t$, $ub_n^s = \max_{t \in Pred(n,k)} ub_t$, and $PDG_n(j) = \sum_{t \in Pred(n,k)} sub\_DG_t(j)$.

The self force of a control node is defined to be zero. The self force for a non_control node is the same as that in the case of the basic force directed scheduling algorithm.

The predecessor force of n for time step c is given by

$$\sum_{k=1}^{numtypes} \sum_{i=lb}^{ub} (P[i] + DG[k][i]/3) \times DG[k][i] \tag{2}$$

and successor force by

$$\sum_{k=1}^{numtypes} \sum_{i=lb}^{ub} (P[i] + DG[k][i]/3) \times DG[k][i] \tag{3}$$

where the lb,ub and P in the first and second formula equals the corresponding lb,ub and P fields of $Pred\_DG(n,k,c)$ and $Succ\_DG(n,k,c)$ respectively. The term $DG[k][i]/3$ is the look-ahead factor. The total force is given by the sum of self, predecessor and successor forces. The total force of all nodes to all feasible time steps are determined and the node with the least force is scheduled.

4.3 Speeding up the DG calculation by reuse of sub_DGs

For hierarchical entities, it can be observed that the sub_DG depends only on the *relative* positions of the source and the sink, not the absolute values of their earliest or latest start times, as long as the scheduled status of the interior nodes doesnt change. Thus, the sub_DGs are invariant under translation. Since the calculation of sub_DG of a hierarchical entity usually involves calculation of the sub_DGs of its components repeatedly with different values for the earliest start time of the source or sink, but with the same latency, the sub_DG for those components can be calculated once, and then reused with suitable translation. This results in an order of magnitude improvement in run time. In practice, we maintain a table which stores the sub_DGs of hierarchical entities for each latency encountered so far. Each table entry has a $valid$ bit which indicates whether the corresponding sub_DG is correct. Initially, all the entries in the table are invalid. When the algorithm needs the sub_DG for a node for a given latency, it first checks the table. If it encounters a valid entry, it can directly use the stored result. If the table entry is invalid, it calculates the sub_DG, stores it in the table and sets the valid bit. As long as none of the nodes with in the hierarchical node is scheduled, the table entry remains valid, and can be reused. After a schedule step, all entries in the table corresponding to the hierarchical entities enclosing the nodes scheduled are invalidated.

The overall sequential algorithm is:

```
FORCE-DIRECTED-HIERARCHY()
for level ← 0 to number of levels
    do repeat
            Evaluate time frames :
                3.1.Find ASAP schedule
                3.2.Find ALAP schedule
            Calculate self, predecessor and successor forces for the nodes
            Add them to get the total force for each feasible control step
            Determine the best node and step; set its time frame equal
            to the selected step.
            Invalidate the stored sub_DGs on the path from the scheduled node to the root
            Update the overall DG
        until all operations scheduled
```

.

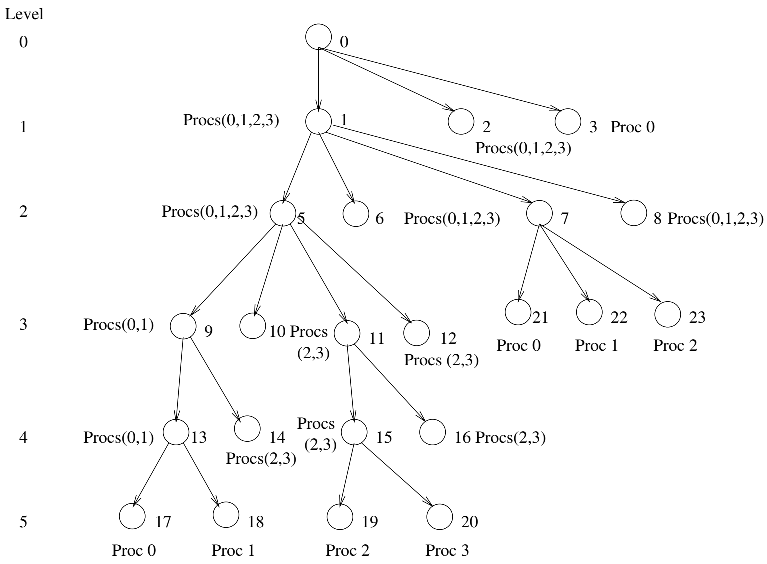


Figure 4: Syntax tree of example hierarchical graph with processor assignments

Figure 4 shows the syntax tree of example hierarchical signal flow graph of Figure 3, and the levels of various nodes.

5 Parallel hierarchical force directed scheduling

In this section, we describe a parallel algorithm for the hierarchical scheduling approach explained in the previous section. As in the sequential case, the scheduling is done one level at a time starting from level 0, scheduling all nodes at a particular level before proceeding to the next. The processors are assigned to the hierarchical nodes based on a cost function (specified in the next section) based on the number of nodes enclosed by the node. Call the set of processors assigned to a node n the *process group* of n. For nodes enclosed by a hierarchical node n, processors in the process group of n are assigned based on the cost function. For nodes at the lowest level(the atomic nodes), the processors in the process group of the immediately enclosing hierarchical node are assigned cyclically. For the graph in Figure 4, the process group of node 9 is $\{0,1\}$. As in the case of the sequential algorithm, we store sub_DGs in a table and reuse them as long as they are valid. However, since the sub_DGs calculated by a particular processor are stored in its local memory, it is not immediately accessible to other processors.

The processors in the process group of each hierarchical node collectively calculates the sub_DGs of that node. It is done in a bottom up manner, since calculation of sub_DG for a hierarchical entity requires the sub_DGs of the nodes it enclose. The processors calculating the sub_DGs of nodes at a particular level communicates them with the processors in the process group of the hierarchical node enclosing those nodes by a group-level *gather* operation. Once this process completes with the calculation of sub_DGs of the root node, all processors have, in their local memories all the sub_DGs needed to calculate forces for the nodes assigned to them. This is due to the fact that scheduling is done in a top down manner. At a particular level l, each processor needs only the sub_DGs of the nodes enclosed by the hierarchical node immediately enclosing it (which is at level $l-1$). Since all nodes at level $l-1$ has been already scheduled, changes in time frames of the children of a particular node h in level $l-1$ does not affect the time-frames of the children of another node h' at level $l-1$, and hence does not have to be included in the list of predecessors or successors used in calculating the ps force (Equations 2, 3) for a child of h. Thus, a processor calculating the force for a child of h needs only the sub_DGs of other children of h, which have been calculated and distributed among the processors in the process group of h.

Consider for example node 15 in Figure 3. When its force is calculated, node 11 has already been scheduled, since it is at a lower level. Thus, scheduling node 15 does not affect the time-frames of nodes 13 or 14, and hence those nodes does not contribute to the predecessor force of 15. Only the sub_DGs of node 16 is required which was calculated earlier by processors 2 and 3. Only those processors assigned to a particular hierarchical node performs the calculation of forces and sub_DGs for the nodes enclosed by the hierarchical node. One advantage of this static division of work is that we can exploit locality in reusing the sub_DGs stored in each processor's local memory.

For each hierarchical entity, there is possibly more than one assigned processor. One of them is assigned ownership for that node. The role of the owner of a node is to update the sub_DG of the node during scheduling when one of the nodes enclosed by the node is scheduled. Since the sub_DGs of all nodes at a level are known, the calculation of forces at any given level closely resembles that in the case of the basic force directed scheduling algorithm. The parallel algorithm has three major steps:

- Node partitioning step
- The pre-processing step, and
- The force calculation and scheduling step

5.1 Node partitioning step

In the node partitioning step, the nodes are distributed among the processors. Associated with each hierarchical entity is a $height$, which denotes the depth of the syntax tree corresponding to the nodes in that entity. Consider for example, the nodes in level 0. It contains hierarchical entities having possibly different heights. Consider the set of nodes with a given height (say h). The processors are partitioned across all the nodes in that set so that the number of processors assigned to node i is proportional to a cost funtion which is an approximation of the work involved for that node, given by :

$$C(i) = ALAP(i) - ASAP(i) + 1 + \sum_{j \in N(i)} C(j)$$

if i is a hierarchical node and

$$C(i) = ALAP(i) - ASAP(i) + 1$$

otherwise. where $N(i)$ is the set of nodes enclosed by i. A particular processor among the processors assigned to each hierarchical node is assigned ownership for that node. In level 1, the processors assigned to the parent hierarchical entity are distributed among the nodes contained in that entity and so on until we reach the leaves.

5.2 Pre-processing step

In the pre-processing step, the sub_DGs for each hierarchical entity for all feasible latencies for that entity that arise during the force calculation and scheduling step are calculated and stored. This is achieved by setting the latency of the immediately enclosing hierarchical entity to its maximum value within the limits of its $ASAP$ and $ALAP$ schedule, (since that results in the maximum possible flexibility for the nodes within). The sub_DGs are calculated level by level, starting at the deepest. As the computation proceeds from one level to the next, the sub_DGs of each node in the current level for the feasible latencies calculated in the current step are distributed to all processors in the process group of the node immediately enclosing the node in the hierarchy, by a gather operation. Once the pre-processing step is complete, all the processors have the sub_DGs required for calculating the forces.

5.3 Force Calculation and Scheduling

After the pre-processing step, the processors calculate the forces of the nodes to each feasible step and sends the best node to a master processor. The master selects the node with the least force and schedules it to the corresponding step. It then broadcasts the information to all processors. Due to this scheduling, the pre-calculated sub_DG of the hierarchical nodes enclosing the scheduled node (and all hierarchical nodes in the path from the node to the root node in the hierarchy) are no longer valid. Thus, they are recalculated. The sub_DG of a node is recalculated by the processor owning the node and sent to the processor owning the parent of the node, starting at the level in which the nodes are being scheduled currently. Since the scheduling proceeds from the lowest to the highest level, all the control nodes enclosing the scheduled node have been scheduled, and the new sub_DG doesnt have to be distributed to all processors in the group, since the sub_DGs are needed only for force calculation. However, the sub_DG of the root node (which is the overall DG of the graph) is required for calculation of forces and hence is broadcast to all processors by the owner processor of the root node. Consider for example the situation when node 15 is scheduled. The sub_DGs of nodes 11,5,1 and 0 become invalid. Processor 2, the owner of node 11 recalculates the sub_DG of 11 and sends it to processor 0. Processor 0 receives that and recalculates the sub_DGs of nodes 5,1 and 0 since it owns all those nodes. Once processor 0 calculates the sub_DG of node 0 (which is the overall DG of the graph), it broadcasts it to other processors.

5.4 An Example

We consider the hierarchical graph in Figure 3 in the previous section. At level 0, it has a conditional node (node 1) and an add operation(node 3). The conditional operation has two alternatives whose sources are nodes 5 and 7 respectively. Nodes 5 and 7 are at level 1. The second alternative has a multiply and two add operations, which are at level 2. Assuming that we have 4 processors,they are assigned to the various nodes as shown in the figure. Assume that we have to schedule the graph for latency 10. Assume, for simplicity that each loop executes once. Thus, the conditional statement can have latencies ranging from 3 to 9. The first alternative can have latencies ranging from 2 to 9, and the second, from 3 to 9. Consider the node 9.In the preprocessing step, processors 0 and 1 recursively calculate the sub_DGs for node 9 for latencies 1 to 4, and processors 2 and 3 calculate the sub_DGs for node 11 for latencies 1 to 4. After that, processors 0,1,2,and 3 perform an *allgather* operation (which is defined in MPI [32], after which processors 0,1,2 and 3 have the sub_DGs for nodes 9 and 11 for feasible latencies. However, processors 0 and 1 does not have the sub_DGs of nodes 15 or 16. But, they are not required for the DG calculation for node 0 or for the calculation of forces for nodes 9 and 10. In the next step, processors 0,1,2 and 3 calculate and store the sub_DGs for nodes 5,6, 7 and 8 after which they perform an allgather operation which results in processors 0,1,2 and 3 having the sub_DGs for feasible latencies for nodes 5,6,7 and 8.

Consider now the force calculation phase. Processors 0,1,2 and 3 calculate the forces for nodes 1,2 and 3 and schedules them. The sub_DGs of node 0 is invalidated and should be recalculated. However, the sub_DG of node 1 can still be reused, since the sub_DG of a hierarchical node depends only on the latency of the node, provided that none of the interior nodes are scheduled. After 1 and 2 are scheduled, processors 0,1,2 and 3 calculate the forces for nodes 5,6,7 and 8 and schedules them. After they are scheduled, processors 0 and 1 calculate the forces for nodes 9 and 10 and processors 2 and 3 calculate the forces of nodes 11 and 12.

5.5 Load Balancing

As explained above, each hierarchical entity has a list of processors assigned to it in the first step. Due to the way the sub_DGs were calculated in the pre_processing step, each of the processor associated with a hierarchical node has the sub_DGs for all feasible latencies of the component nodes of the hierarchial entity in their local memories, and hence can calculate the forces of any of those nodes. The force calculation of a particular node to a particular step represents a unit of work. The total work involved in calculating the forces of the component nodes of a hierarchical entity is distributed among the processors in the process group of that entity. This distribution is done cyclically among the processors with the grain size ranging from 1 to $total\_work/(nprocs)$. In the figure, during the force calculation for nodes 5,6,7 and 8, the $< node, step >$ pairs are cyclically distributed among processors 0,1,2 and 3.

6 Analysis

6.1 Pre-processing step

Assume that the given hierarchical graph has N nodes and has depth d. Let L be the overall latency. In the worst case, each level has one hierarchical node except one of them, having $N - d + 1$ nodes. Thus, all p processors could be performing the all-gather operation for $d - 1$ levels in the worst case. In the last stage, each node could be allocated k processors, where k ranges from 1 to p. All-gather operation is equivalent to all nodes sending some message to one processor (gather) followed by that node broadcasting all the received messages to all the processors. Thus is takes at most the amount of communication equivalent to a gather operation followed by a broadcast. The gather operation and broadcast can be implemented in $2t_s(\sqrt{p}-1)+t_w m_1(p-1)$ time for a $2-D$ mesh. Thus, the all-gather operation can be perfomed in less than $4t_s(\sqrt{p}-1) + 2t_w m_1(p-1)$ units of time, where t_s is the start-up time, t_w is the per-word transfer time.

The computation required for the pre-processing step for a particular level t for each hierarchical node at that level is at most $c_1 Ls$ where s is the number of nodes contained within the hierarchical node at level $t+1$ and c_1 is some constant. Thus, the overall computation required for the pre-processing stage is $\sum_{t=1}^{d} c_1 LN$ which is $O(LN^2)$. Thus in the worst case, the time for the pre-processing step is $O(LN^2) + 4dt_s(\sqrt{p}-1) + 2t_w m_1(p-1)$, where m_1 is the maximum length of a message whose upper bound is $c_2 NL^2$ where c_2 is some constant.

6.2 The force-calculation and scheduling phase

In this phase, after each scheduling step, each processor sends its best node, along with the force to a master processor, and the master processor selects the globally best node and broadcasts it to all processors. Each of the above can be performed in time $2(t_s + t_w m_3)logp + 2t_h(\sqrt{p}-1)$ where m_3 is the length of the messages which is a constant (since the message contains only the node with the least force, the corresponding time step, and the force). There are at most N schedule steps (since at a given step, more than one node could be scheduled). Each force calculation step takes $O(LN^2/p)$ time assuming good load balance. Thus, the force calculation for all steps takes $O(LN^3/p)$ since there are at most N steps. Thus the overall time for the force calculation and scheduling is $O(LN^3/p) + 2(t_s + t_w m_3)logp + 2t_h(\sqrt{p}-1)$.

7 Experimental Results

The parallel hierarchical force directed scheduling algorithm was implemented using the portable Message Passing Interface(MPI) [32], which has been ported on a variety of parallel machines. We have tested the algorithm on a set of high level synthesis benchmarks.The resulting schedules were optimal for most cases and very close to optimal for others.The run times for some benchmarks for different latencies are as shown below. The results are shown for a 16-processor Intel Paragon distributed memory multicomputer, 8-processor SGI Power Challenge shared memory multiprocessor and a network of SUN Sparc station 5 work-stations.

7.1 Experimental results for non-hierarchical graphs

Tables 1,2,3 shows the runtimes and speedups for node-based approach for non-hierarchical graphs and Tables 4,5 and 6 show the runtimes and speedups for step-based approach for non-hierarchical graphs. From the results we can make the following observations. The qualities of the results (in terms of the hardware resources used) for each latency case are identical in the sequential and parallel executions. If the latency is close to the minimum, then the time frames of the nodes are small, in which case the total work involved in calculating the forces is relatively small. This results in low overall running time and poor speedup. For the node based partitioning scheme, we are getting speedups of 5 to 9 on 16 processors on an Intel Paragon. The speedups on the Intel Paragon are comparable to that of the network of work stations, which has significant costs for communication. Clearly, the problem has been parallelized such that the grain size of computation between communications is quite large. Hence, we have effectively parallelized the problem.

Both partitioning schemes are comparable in their runtimes and speedups for a given parallel environment. Clearly we have been successful in balancing the load equally in both cases.

The results on the SGI Challenge shared memory multiprocessor are worse than that of the Intel Paragon which is a distributed memory multicomputer. The result is counter-intuitive since most people think that it is possible to get better speedups on shared memory multiprocessors. We attribute the result due to cache coherence effects, contention and multitasking. In addition, for MPI, it is possible to get a large communication bandwidth on message passing machines during global operations such as $allgather()$ which creates more bus traffic in a shared memory multiprocessor.

Table 1: Run times(seconds) and speedups for node-based approach for non-hierarchical graphs on a network of work stations

time(speedup)				
circuit	latency	1 proc	2 proc	4 proc
Random(60)	60	336.1(1.0)	245.9(1.37)	177.3(1.89)
Random(60)	80	586.1(1.0)	336.4(1.74)	281.1(2.08)
elliptic	17	1.725(1.0)	0.998(1.73)	0.595(2.89)
elliptic	27	5.01(1.0)	2.78(1.80)	1.498(3.34)

Table 2: Run times(seconds) and speedups for node-based approach for non-hierarchical graphs on an SGI Power Challenge multiprocessor

time(speedup)					
circuit	latency	1 proc	2 proc	4 proc	8 proc
Random(60)	60	384(1.0)	305(1.26)	242.5(1.58)	148.1(2.59)
Random(60)	80	704(1.0)	713.8(0.98)	548.8(1.28)	358.0(1.96)
elliptic	17	1.37(1.0)	0.96(1.43)	0.71(1.93)	1.80(0.76)
elliptic	27	5.12(1.0)	2.69(1.9)	2.18(2.35)	3.75(1.36)

Table 3: Run times (seconds) and speedups for node-based approach for non-hierarchical graphs on Intel Paragon multicomputer

time(speedup)						
circuit	latency	1 proc	2 proc	4 proc	8 proc	16 proc
Random(60)	60	666.5(1.0)	472.1(1.4)	332.5(2.0)	203.8(3.27)	118.0(5.64)
Random(60)	80	1252(1.0)	706.1(1.77)	441.8(2.83)	286.2(4.37)	188.2(6.65)
elliptic	17	3.125(1.0)	1.843(1.69)	1.218(2.57)	0.8437(3.7)	0.703(4.44)
elliptic	27	11.20(1.0)	6.17(1.81)	3.39(3.3)	1.89(5.92)	1.25(8.96)

Table 4: Run times(seconds) and speedups for step-based approach for non-hierarchical graphs on a network of work stations

time(speedup)				
circuit	latency	1 proc	2 proc	4 proc
Random(60)	60	336.1(1.0)	334.9(1.0)	173.3(1.94)
Random(60)	80	586.1(1.0)	337.5(1.74)	228.1(2.57)
elliptic	17	1.86(1.0)	1.73(1.07)	0.996(1.87)
elliptic	27	4.81(1.0)	2.60(1.85)	1.392(3.45)

Table 5: Run times(seconds) and speedups for step-based approach for non-hierarchical graphs on an SGI Power Challenge multiprocessor

time(speedup)					
circuit	latency	1 proc	2 proc	4 proc	8 proc
Random(60)	60	384(1.0)	319.2(1.20)	251.1(1.53)	139.8(2.75)
Random(60)	80	704(1.0)	308.7(2.28)	229.2(3.07)	128.9(5.46)
elliptic	17	1.43(1.0)	0.96(1.49)	0.83(1.72)	2.01(0.71)
elliptic	27	4.83(1.0)	2.63(1.84)	2.47(1.95)	2.71(1.78)

Table 6: Run times for step-based approach for non-hierarchical graphs on Intel Paragon multicomputer

time(speedup)						
circuit	latency	1 proc	2 proc	4 proc	8 proc	16 proc
Random(60)	60	666.5(1.0)	458.9(1.45)	316.9(2.1)	195.1(3.42)	110.0(6.06)
Random(60)	80	1252(1.0)	702.5(1.78)	444.3(2.82)	293.2(4.27)	193.5(6.47)
elliptic	17	3.125(1.0)	1.797(1.74)	1.172(2.67)	0.828(3.77)	0.672(4.65)
elliptic	27	11.20(1.0)	6.03(1.86)	3.29(3.4)	1.90(5.89)	1.27(8.82)

7.2 Experimental results for hierarchical graphs

Tables 7,8 and 9 show the runtimes and speedups for hierarchical graphs. For the parallel algorithms for hierarchical graphs, the speedup results are similar to that of non-hierarchical graphs. Again the qualities of the results (in terms of the hardware resources used) for each latency case are identical in the sequential and parallel executions. As in the case of non-hierarchical graphs, if the latency is close to the minimum, then the time frames of the nodes are small, resulting in low overall running time and poor speedup. For Kalman, there are only few nodes in each level which results in low parallelism. The higher latencies in the above table correspond to the latencies which results in close to the minimum resource requirement for the corresponding circuit.

Table 7: Runtime(seconds) and speedups for parallel algorithm on a network of workstations for hierarchical graphs

time(speedup)				
circuit	latency	1 proc	2 proc	4 proc
kalman	3700	205.56(1.0)	133.36(1.54)	92.80(2.2)
elliptic	17	6.85(1.0)	3.32(2.1)	2.61(2.62)
elliptic	27	20.21(1.0)	10.87(1.85)	6.65(3.03)
synth	100	98.61(1.0)	77.78(1.26)	44.60(2.21)
synth	200	533.9(1.0)	419.9(1.27)	175.3(3.04)

Table 8: Runtime(seconds) and speedup of parallel algorithm on Intel Paragon message passing multicomputer for hierarchical graphs

Runtime(speedup)						
circuit	latency	1 proc	2 proc	4 proc	8 proc	16 proc
kalman	3700	107.89(1.0)	69.14(1.56)	38.51(2.8)	23.76(4.6)	17.22(6.3)
elliptic	17	3.23(1.0)	1.95(1.6)	1.45(2.3)	1.03(3.1)	0.804(4.01)
elliptic	27	11.35(1.0)	6.14(1.8)	3.45(3.3)	2.10(5.4)	1.45(7.8)
synth	100	49.67(1.0)	37.01(1.34)	21.01(2.4)	12.69(3.9)	8.57(5.8)
synth	200	253.47(1.0)	191.00(1.3)	82.06(3.1)	43.43(5.8)	24.68(10.3)

Table 9: Runtime(seconds) and speedups for parallel algorithm on SGI Power Challenge multiprocessor for hierarchical graphs

time(speedup)				
circuit	latency	1 proc	2 proc	4 proc
kalman	3700	76.09(1.0)	47.0(1.62)	28.97(2.63)
elliptic	17	1.634(1.0)	0.914(1.78)	0.629(2.59)
elliptic	27	5.064(1.0)	2.669(1.89)	1.459(3.47)
synth	100	27.94(1.0)	20.12(1.39)	13.62(2.05)
synth	200	158.2(1.0)	121.3(1.30)	55.25(2.86)

8 Conclusions

In this paper, we presented some novel algorithms for scheduling hierarchical signal flow graphs in the domain of high-level synthesis. There were several key contributions of this paper. First, we developed a novel extension of the force-directed scheduling problem which naturally handles loops and conditionals by coming up with a scheme of scheduling *hierarchical* signal flow graphs. Second, we developed three novel parallel algorithms for the scheduling problem. Third, our parallel algorithms are portable across a wide range of parallel platforms. We reported results on an 8-processor SGI Challenge, a 16 processor Intel Paragon, and a network of 4 SUN SPARCstation5 workstations. Fourth, we reported actual implementations of these algorithms in real parallel machines, and report results of qualities and runtimes for various benchmark circuits on a variety of parallel machines. Finally, while some parallel algorithms for VLSI CAD reported by earlier researchers have reported on loss of qualities of results, our parallel algorithms produced exactly the same results as the sequential algorithms on which they are based.

In some related work, we are developing parallel algorithms for behavioral simulation [23]. In the future, we will pursue parallel algorithms for other tasks in high-level synthesis, such as allocation, and in hardware/software codesign and cosimulation. We will integrate all these tools with the parallel tools developed for solving the CAD problems at the lower level such as placement, routing, logic synthesis, and testing. All these tools will run in a portable manner in a large variety of parallel and distributed platforms.

References

[1] P. Banerjee, "Parallel Algorithms for VLSI Computer-aided Design Applications," *Englewood-Cliffs, NJ:Prentice Hall*, 1994.

[2] P. G. Paulin and J. P. Knight, "Force-directed scheduling for the behavioral synthesis of ASICS," *IEEE Trans. Computer-Aided Design*, vol. 8, no. 6, pp. 661-679, 1989.

[3] W. Gropp, E. Lusk, and A. Skellum, "Using MPI: Portable Parallel Programming with the Message Passing Interface", *MIT Press*, 1994.

[4] R. Haupt, "A survey of priority-rule based scheduling," *OR Spektrum,* vol. 11, no. 1, pp. 3-16, 1989.

[5] M. C. McFarland, "Using bottom-up design techniques in the synthesis of digital hardware from abstract behavioral descriptions," *Proc. 23rd DAC*, Las Vegas, NV, June 1986, pp. 474-480.

[6] M. C. McFarland, A. C. Parker, and R. Camposano, "The high-level synthesis of digital systems," *Proc. IEEE*, vol. 78, no. 2, pp. 301-318, 1990.

[7] C. H. Gebotys and M. I. Elmasry, "A global optimization approach for architectural synthesis," in Proc. ICCAD, Santa Clara, CA, Nov. 1990, pp. 258-261.

[8] C. T. Hwang, T. H. Lee, and Y. C. Hsu, "A formal approach to the scheduling problem in high level synthesis," IEEE Trans. CAD, vol. 10, no. 4, pp. 464-475, 1991.

[9] R. J. Cloutier and D. E. Thomas, "The combination of scheduling, allocation, and mapping in a single algorithm," Carnegie Mellon Univ Res. Rep CMUCAD-90-17, May 1990.

[10] T. Kim, J. W. S. Liu, and C. L. Liu, "A scheduling algorithm for conditional resource sharing," in Proc ICCAD, Santa Clara, CA, Nov 1991, pp. 84-87.

[11] A. Oláh, S. H. Gerez, and S. M. Heemstra de Groot, "Scheduling and allocation for the high-level synthesis of DSP algorithms by exploitation of transfer mobility", in *Proc.CompEuro*, Delft, The Netherlands, 1992, pp. 145-150.

[12] L. Stok, "Architectural synthesis and optimization of digital systems," Ph. D dissertation, Eindenhoven Univ. of Technol., Eindenhoven, The Netherlands.Apr. 1990

[13] Wim F. J. Verhaegh, Paul E. R. Lippens, Emile H. L. Aarts, Jan H. M. Korst, Jef L. van Meerbergen, Albert van der Werf, "Improved force-directed scheduling in high-throughput digital signal processing", in *IEEE trans on CAD*, vol. 14, no. 8, August 1995.

[14] J. Roy, N. Kumar, R. Dutta, R. Vemuri,"DSS: A distributed high-level synthesis system", in *IEEE design and test of computers*, June 1992.

[15] B. Ramkumar and P. Banerjee, "ProperCAD: A portable object-oriented parallel environment for VLSI CAD," *IEEE Trans. Computer-Aided Design*, vol. 13, pp. 829–842, July 1994.

[16] S. Kim, B. Ramkumar, J. Chandy, S. Parkes, and P. Banerjee,"ProperPLACE: A portable parallel algorithm for standard cell placement. *Proc. 8th Int. Parallel Processing Symp. (IPPS-94)*, Apr. 1994.

[17] R. Brouwer and P. Banerjee, "PHIGURE: A parallel hierarchical global router," in *Proc. 27th Design Automation Conference.*, pp. 591-597, June 1990.

[18] K. P. Belkhale and P. Banerjee. "Parallel algorithms for VLSI circuit extraction," IEEE Trans. CAD, 10(2):604-618, May 1991.

[19] J. S. Rose, W. M. Snelgrove, and Z. G. Vranesic, "Parallel cell placement algorithms with quality equivalent to simulated annealing," *IEEE Trans. Computer-Aided Design*, vol. 7, pp. 387–396, Mar. 1988.

[20] A. Casotto, F. Romeo, and A. Sangiovanni-Vincentelli, "A Parallel Simulated Annealing Algorithm for the Placement of Macro-Cells", in *IEEE Trans. Computer-aided Design Integrated Circuits Systems*, pages 838-847, Sept. 1987.

A. Casotto and A. Sangiovanni-Vincentelli, "Placement of standard cells using simulated annealing on the Connection Machine," in *Digest of Papers, International Conference on Computer-Aided Design*, (Santa Clara, CA), pp. 350–353, Nov. 1987.

[21] S. Patil, P. Banerjee, and T. Patel," Parallel Test Generation for Sequential Circuits on General Purpose Multiprocessors," in *Proc. 28th Design Automation Conf (DAC-91)*, June. 1991.

[22] P. Agrawal, V. D. Agrawal, J. Villodo, "Sequential Circuit Test Generation on a Distributed System," in *Proc 30th Design Automation Conf, (DAC-93)*, June. 1993.

[23] V. Krishnaswamy, P. Banerjee, "Actor basec parallel VHDL simulation using Time Warp," in *Proc. Int. Conf. on VLSI Design*, 1996.

[24] K. De, B. Ramkumar and P. Banerjee, "ProperSYN: A portable parallel algorithm for logic synthesis," in *IEEE Trans. Computer Aided Design(ICCAD-92)*, Nov. 1992.

[25] B. Ramkumar and P. Banerjee, "ProperTEST: A portable parallel test generator for sequential circuits," to appear in *IEEE Trans. on CAD*, 1996.

[26] K. MacPherson, "Parallel algorithms for layout verification," *MS thesis, Department of Electrical and Computer Engineering,University of Illinois, Urbana-Champaign, Technical report*, 1995.

[27] K. De, J. A. Chandy,S. Roy, S. Parkes and P Banerjee, "Parallel algorithms for logic synthesis using the MIS approach," *Proceedings of IPPS*, Santa Barbara, April 1995.

[28] S. Parkes, P. Banerjee, and J. Patel, "A parallel algorithm for fault simulation based on PROOFS," *Proceedings of the ICCD*, Austin, TX, Oct. 1995.

[29] S. Parkes, P. Banerjee, and J. Patel "ProperHITEC: A portable, parallel, object-oriented approach to sequential test generation," in *Proc. 31st Design Automation Conf.,* June 1994.

[30] R. Galivanche and S. M. Reddy, "A Parallel PLA Minimization Program," in *Proc. Design Automation Conf.*, pages 600-607, 1987.

[31] H. D. Hachtel and P. H. Moceyunas, "Parallel Algorithms for Boolean Tautology Checking," in *Proc. Int. Conf. Computer-aided Design*, pp. 422-425, Nov. 1987.

[32] W. Gropp, E. Lusk, and A. Skellum, "Using MPI: Portable Parallel Programming with the Message Passing Interface", *MIT Press*, 1994.