A PARALLEL ALGORITHM FOR HIERARCHICAL CIRCUIT EXTRACTION

Krishna P. Belkhale and Prithviraj Banerjee

Center for Reliable and High-Performance Computing,
Coordinated Science Laboratory,
University of Illinois
Urbana, IL- 61801

ABSTRACT

In *ICCAD88, 89*, we proposed efficient parallel algorithms to speed up the task of VLSI circuit extraction on distributed memory multiprocessors. The input to the algorithms is a flat description of the circuit, in terms of the rectangles in different layers. In this paper, we describe an algorithm that combines the benefits of hierarchical analysis and parallelism. The input is a hierarchical description of the circuit. Towards this goal, we formulate and solve a general problem in scheduling. The parallel algorithm for hierarchical circuit extraction has been implemented on an Encore shared memory multiprocessor.

1. INTRODUCTION

Given a set of rectangles in different mask levels for a VLSI layout as the input, a circuit extractor determines the circuit connectivity and estimates various electrical parameters such as the resistances of lines, capacitances of nodes, and dimensions of transistors. There are two disjoint, but orthogonal, approaches to speed up circuit extraction. The first approach is to use the information available in a hierarchical representation of a circuit. Each cell consists of some subcells and base level geometry. Hierarchical extractors exploit the repetitive information in this representation by extracting a cell that is used many times only once and appropriately modifying it wherever it is used [1, 2]. The main problem in hierarchical extraction, which is responsible for the overhead, is that the properties of a cell will change if it overlaps another cell, or some base level geometry. Some hierarchical extractors [2] forbid any overlapping of cells. However, this is inconvenient and unnecessarily restrictive. Other extractors like [1] transform the circuit with overlapping cells to a circuit in which there are no overlapping cells.

Another approach to speed up the task of circuit extraction is to use multiprocessors [3-5]. A parallel circuit extractor basically divides the circuit into smaller parts and performs various tasks in circuit extraction on each of the parts in parallel, results of which are combined to complete the extraction of the whole circuit.

The organization of this paper is as follows. In Section 2, we discuss the changes in the parallel flat algorithm, for the case of shared memory multiprocessors, as compared to the hypercube multiprocessor implementation described in [4]. In Section 3, we describe an algorithm that combines the benefits of hierarchical analysis and parallelism. In Section 4, we present the implementation results for the Encore shared memory multiprocessor.

This research was supported in part by the National Science Foundation Award under Grant NSF MIP 87-14646, and in part by the Argonne National Laboratory Computing Facility

2. SHARED MEMORY IMPLEMENTATION

The circuit partitioning described in [4] is for the case when the number of processors is a power of 2. In the shared memory implementation, this has been extended for the general case. A t tuple $(n_1, n_2, ..., n_t)$, represents a partitioning scheme that involves t divisions along x axis, the i^{th} subregion being divided into n_i subregions in the y dimension. For example, the partitioning scheme shown in Fig. 1(a) is represented by the tuple $(4, 3, 3)$. Let lx and ly be the x and y dimensions of the circuit. The expected maximum perimeter among all the regions is $\max_{1 \le i \le t}((lx/p)n_i + (ly/n_i))$. This has to be minimized, as this represents the communicational requirements in the algorithm. From the concavity of the above function, it can be proved that there is an optimal partitioning scheme where $|n_i - n_j| \le 1$, for all i, j. This observation implies that we need to evaluate only p solutions to obtain the best partitioning scheme.

Once the partitioning scheme is determined, the partitioning lines are obtained using an approximate point based partitioning strategy. The circuit region is divided into different bins in the x dimension. The number of rectangles that fall in each bin is determined and stored in an array. The entries in the array can then be used to determine the partitioning lines along the x dimension. The process can then be repeated in each of the resulting subregions, by the appropriate processors, to determine the partitioning lines along the y dimension.

An important step in a parallel extractor involves assigning global labels to locally connected sets (*LCS* sets), so that connected *LCS* sets at different processors get the same label. This problem, called the merge problem, is solved in the shared memory implementation using a parallel union find algorithm and locking. Each processor computes the border segments for the four borders, with their associated *LCS* set labels. Each processor writes this information onto the shared memory. In the merge step, each processor looks at its top and right border segments. For each border segment a, it determines the corresponding border segment b in the processor at the other end of the border. It then does an union operation on the *LCS* sets of a and b. The union operations performed at different processors interfere, and hence locks are used to ensure correctness.

Consider the operation *union*(a, b), where a and b are two border segments. The data structure used by the algorithm consists of a union find data structure as shown in Fig. 1(b). The first step in the algorithm is to determine the current roots x, y

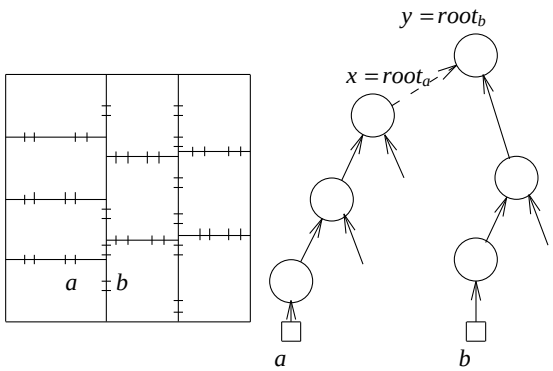

Fig. 1. (a) Example Partitioning (b) Merge Algorithm

of $LCS(a)$ and $LCS(b)$ respectively. If x is same as y, the union algorithm returns. If $x \neq y$, the processor locks the roots in increasing order of their labels, in order to avoid deadlock. Once the locks are obtained, the processor checks if x and y are still the roots. If x or y is not a root the whole operation is continued from x, y, after releasing the locks. If x and y are still the roots, a directed edge is added from x to y (or y to x) and the locks are released. The complete algorithm is shown in Fig. 2.

3. PARALLEL HIERARCHICAL ALGORITHM

Hierarchical Circuit Extraction has two steps. The first step is the analysis and transformation step. The second step is the actual extraction step. The first step involves transforming a circuit that has overlapping cells into one that does not have any overlapping cells. One of the ways of achieving this involves looking at the subcells of each cell, and instantiating a subcell A if it overlaps another subcell or geometry. The instantiation process involves replacing the cell A with its subcells and geometry. The whole process is repeated until there are no overlapping subcells in each of the cells. At the end of this step, some cells that are not useful are discarded. A cell is useful if is the root cell, or it occurs as a subcell of an useful cell.

At the end of the first step, the job of extraction consists of extracting each of the cells that remain and appropriately combining the results using common node names. For each cell, the information available consists of its ports, the geometry in terms of rectangles, and ports of its subcells. The ports are represented as rectangles which have a span of zero along one dimension. With this representation, the extraction of a cell is done in a similar manner as flat circuit extraction.

The first step is parallelized by assigning cells to different processors dynamically from a list. The cells are ordered so that subcells of a cell occur before a cell. The amount of parallelism in this step is generally not more than a factor of 2. Fortunately, the second step is the most time consuming part of hierarchical circuit extraction. There are two approaches we could take for parallelizing this step.

```
union(a, b)
    x ← LCS(a), y ← LCS(b), fin ← 1
    while ( fin ≠ 1 ) do
        x ← findroot(x), y ← findroot(y)
        if ( x = y ) then return;
        lock(x), lock(y)  /* In increasing order of labels */
        if ( x and y are still roots ) {
            x.father ← y, fin ← 1
          }
        unlock(x), unlock(y)
    end while
end union
```

Fig. 2: The union algorithm.

4.1. Algorithm LPT

One approach to parallelizing the second step involves assigning the extraction of cells to processors dynamically from a list. The list of cells could be obtained by sorting the cells in decreasing order of the number of rectangles it contains. This algorithm corresponds to the *LPT* algorithm (largest processing time first) [6] for scheduling independent tasks on multiprocessors. The disadvantage with the above approach is that some of cells could be very large. A large cell assigned to a single processor will be a bottleneck for the whole computation, reducing the parallelism that can be obtained.

4.2. Algorithm PITS

From the previous discussion, it is clear that we need to look at solutions that assign more processors to the extraction of a large cell. The assigned processors can extract the cell using the parallel algorithm described in the previous sections. However, since the speedups obtained will be less than linear, assigning many processors to the extraction of a cell could result in an increase in the overall time. This problem is formulated as a general problem called *partitionable independent task scheduling (PITS)* problem [7]. It consists of scheduling n independent tasks on p processors. Associated with each task T_i, $1 \leq i \leq n$, we have an estimate of the time taken by the task $\tau(T_i)$ on a single processor, and values $\sigma_j(T_i)$, $1 \leq j \leq p$, that give the speedups that can be obtained by running the task on j processors. The problem then is to schedule the collection of tasks on p processors so that the maximum completion time is minimized. For our application, the different tasks represent the different cells to be extracted. The time taken for extraction of a cell is estimated from the number of rectangles in the cell. The speedup function is assumed to be same for all the cells, and is estimated by the speedups obtained by the parallel flat algorithm of the previous section for different number of processors.

The *partitionable independent task scheduling* problem can be shown to be NP-hard. We describe a heuristic that is used to solve the problem. The heuristic is guaranteed to produce results that are within twice the optimal [7], if a condition called the efficiency condition is true. The efficiency condition is necessarily true if the speedup curve is concave, which is true for most applications. The main idea in the algorithm is to construct a schedule and iteratively modify it by assigning some tasks to more processors. At any stage, for every task T_i, we

have a number s_i which denotes the number of processors that has been currently allocated to T_i. During the course of the algorithm, the number of processors allocated to a task may be increased iteratively. Each increase is allowed only if it results in an "immediate" decrease in the schedule time. Instead of actually constructing the schedule at every stage, an upper bound on the schedule time is used to guide the solution process to arrive at a solution faster. A task T_i assigned to $j = s_i > 1$ processors can be looked upon as a collection of j subtasks each having execution time $\frac{\tau(T_i)}{\sigma_j(T_i)}$. These subtasks will be referred to as *split* tasks. The other tasks T_i, with $s_i = 1$, will be referred to as *unsplit* tasks. The complete algorithm is described below.

procedure *fast_part_schedule*

[1] Set s_i to 1, for all $1 \le i \le n$, and set a to p ($p-a$ gives the current number of *split* tasks). Set *avg* to be $\frac{1}{p}\sum_{j=1}^{n} \tau(T_j)$, and *est* to be $\max\{h, avg + y(1-1/p)\}$, where h, y are the highest and $(p+1)^{th}$ largest execution time of a task respectively. Let T_i be a task with the highest execution time.

[2] If $a = 0$ go to 5, else continue. If $a = 1$ and $s_i = 1$ then go to 5, else continue.

[3] If $est \neq h$ then go to 5, else do the following.

[4] If $s_i = 1$ set a to $a-2$, else set a to $a-1$. Increment s_i. Now let $T_{i'}$ be a task such that for all j, $\frac{\tau(T_{i'})}{\sigma_{s_{i'}}(T_{i'})} \ge \frac{\tau(T_j)}{\sigma_{s_j}(T_j)}$, and let $h' = \frac{\tau(T_{i'})}{\sigma_{s_{i'}}(T_{i'})}$. Also, let y' be the $(a+1)^{th}$ largest execution time of an *unsplit* task. Set avg' to $avg + \frac{1}{p}\left(\frac{\tau(T_i)s_i}{\sigma_{s_i}(T_i)} - \frac{\tau(T_i)(s_i-1)}{\sigma_{s_i-1}(T_i)}\right)$ and est' to $\max\{h', avg' + y'(1-1/p)\}$. If $h < est'$ then decrement s_i, restore a (i.e if $s_i = 1$, set a to $a+2$, else set a to $a+1$), and go to 5. Else, set est to est', avg to avg', h to h' and i to i', and go to 2.

[5] Assign the jobs with $s_i > 1$ to $p-a$ processors. The *unsplit* tasks are scheduled dynamically in an *LPT* fashion.

end *fast_part_schedule*.

During the course of the algorithm, $p-a$ gives the total number of *split* tasks in the current schedule, which never exceeds p in the algorithm. The variable h gives the maximum execution time of a *split* or an *unsplit* task. The variable *avg* gives the sum of the computations times of all the *split* and *unsplit* tasks divided by p. The variable *est* gives an upper bound on the current schedule time. This is determined as the maximum of h and $avg + y(1-1/p)$, where y is the $(a+1)^{th}$ largest *unsplit* task.

At every step in the algorithm, we consider allocating one more processor to the task T_i with the current highest execution time. Since the algorithm admits only a maximum of p *split* tasks this step cannot be done if $a = 0$ or $a = 1$ and $s_i = 1$. This check is performed in Step 2. In Step 3, a check is performed to see if T_i is a bottleneck with the current number of allocated processors. If it is, Step 4 considers allocating one more processor to T_i. The upper bound est' on the new schedule is then computed. The above move is accepted only if there is no increase in the schedule time. The proof of the bounded performance of the above algorithm is given in [7].

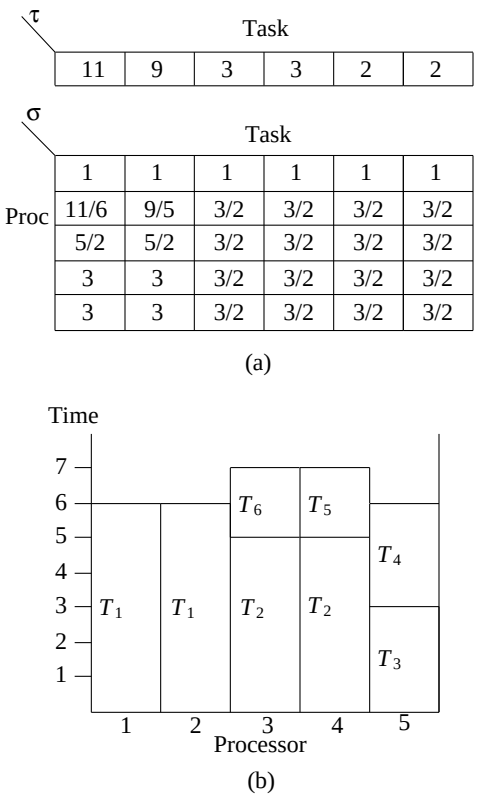

Fig. 3(a) Example Problem (b) Schedule

Example 1: Let us consider the example situation shown in Fig. 3(a). There are 6 tasks to be scheduled on 5 processors. Following the algorithm, the variables i, h, avg, y, est are respectively set to 1, 11, 6, 2, and 11 in Step 1. In Step 3, the check $est = h$ succeeds. Thus one more processor is temporarily allocated to T_1. In Step 4, the variables i', h', avg', y', est' are respectively set to 2, 9, 31/5, 2, and 9. Since $est' \le h$, T_1 is currently allocated 2 processors. The variables i, h, avg, y, est are respectively set to 2, 9, 31/5, 2, and 9. In Step 3, the check $est = h$ succeeds. Thus one more processor is temporarily allocated to T_2. In Step 4, the variables i', h', avg', y', est' are respectively set to 1, 6, 32/5, 3, and 44/5. Since $est' \le h$, T_2 is currently allocated 2 processors. The variables i, h, avg, y, est are respectively set to 1, 6, 32/5, 3, and 44/5. The check in Step 3 fails, as $est \neq h$, and thus the algorithm exits form the loop. The schedule shown in Fig. 3(b) is then constructed.

4. RESULTS

The parallel algorithms described in this paper have been implemented on an Encore shared memory multiprocessor consisting of 8 processors. In Table 1, we give the time taken in seconds on some example layouts, for five algorithms. These are the sequential flat algorithm, parallel flat algorithm, sequential hierarchical algorithm, parallel hierarchical algorithm based on *LPT* algorithm and parallel hierarchical algorithm based on *PITS* algorithm. The times reported exclude the time for input and output. For each circuit, the total number of rectangles in the circuit is given in brackets. As can be observed, for circuits 1, 2, 5 the use of hierarchy did not result in any improvement

Circuit	Flat Seq	Flat Par (8 procs)	Hier Seq	Hier Par (*LPT*) (8 procs)	Hier Par (*PITS*) (8 procs)
Hypercube Router Chip (54800)	612	164	586	588	172
Multiplier Array (66000)	662	151	657	665	159
Static Ram (130700)	927	223	450	323	91
Placement Coprocessor (258600)	1865	495	490	325	91
Associative Cache (258700)	3142	714	3126	3294	626

TABLE1: Time taken in seconds with *HPEX* model.

Circuit	Hierarchical Sequential		Hierarchical Parallel(*PITS*) (8 processors)	
	Step 1	Step 2	Step 1	Step 2
Hypercube Router Chip (54800)	79	507	59	113
Multiplier Array (66000)	51	606	51	108
Static Ram (130700)	26	424	14	77
Placement Coprocessor (258600)	28	462	15	76
Associative Cache (258700)	111	3015	107	519

TABLE 2: Sequential and parallel hierarchical extraction.

over the flat case. However, for circuits 4, 5 the use of hierarchy resulted in a good improvement. In each of cases, the *PITS* based algorithm performed the best (or close to the best). The parallel algorithm, with *LPT* scheduling performs poorly in most of the cases.

In Fig. 4, we graphically illustrate the working of the *LPT* based algorithm and the *PITS* based algorithm for the fourth circuit. We show the time taken by each of the processors. We also indicate the important cells that are extracted by each of the processors.

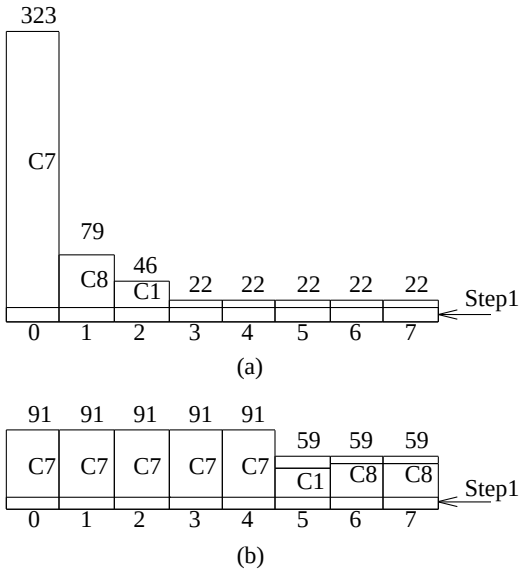

Fig. 4. (a) *LPT* based (b) *PITS* based.

6. CONCLUSIONS

We have presented a parallel algorithm for hierarchical circuit extraction for a shared memory multiprocessor. As a part of this problem, we have derived algorithms for the *partitionable independent task* scheduling problem. This problem could have many other applications. In future, we will explore the method of combining hierarchy and parallelism for other VLSI CAD applications.

REFERENCES

[1] R. Hon and A. Gupta, *HEXT: A Hierchical Circuit Extractor*. Computer Science Press, 1983. pp. 18-34.

[2] G.M. Tarolli and W.J. Herman, ‘‘Hierarchical circuit extraction with detailed parasitic capacitance,’’ *Proc. 20th Design Automation Conf*, pp. 337-345, 1983.

[3] S. Levitin, *A Multiprocessing Approach to Circuit Extraction*. MIT, 1986, Master’s Thesis.

[4] K. P. Belkhale and P. Banerjee, ‘‘PACE2: An Improved Parallel VLSI Extractor with Parameter Extraction,’’ *Proc. Int. Conf. on Computer-Aided Design*, pp. 526-529, Nov. 1989.

[5] B. A. Tonkin, ‘‘Circuit Extraction on a Message-Based Multiprocessor,’’ *27th Design Automation Conf.*, pp. 260-265, June 1990.

[6] R. L. Graham, ‘‘Bounds on Multiprocessing Timing Anomalies,’’ *SIAM J. Appl. Math.*, vol. 17, pp. 263-269, 1969.

[7] K. P. Belkhale and P. Banerjee, ‘‘Approximate Algorithms for the Partitionable Independent Task Scheduling Problem,’’ *Proc. Int. Conf. on Parallel Processing*, pp. 72-75, Aug. 1990.