Parallel Algorithms For FPGA Placement *

Malay Haldar, Anshuman Nayak, Alok Choudhary and Prith Banerjee
Center for Parallel and Distributed Computing
Northwestern University
Evanston, IL 60208-3118
{malay,nayak,choudhar,banerjee}@ece.nwu.edu

Fast FPGA CAD tools that produce high quality results has been one of the most important research issues in the FPGA domain. Simulated annealing has been the method of choice for placement. However, simulated annealing is a very compute-intensive method. In our present work we investigate a range of parallelization strategies to speedup simulated annealing with application to placement for FPGA. We present experimental results obtained by applying the different parallelization strategies to the Versatile Place and Route (VPR) Tool, implemented on an SGI Origin shared memory multiprocessor and an IBM-SP2 distributed memory multiprocessor. The results show the tradeoff between execution time and quality of result for the different parallelization strategies.

1 Introduction

The popularity of Field-Programmable Gate Arrays to implement digital circuitry has seen significant increase in recent times. The prime advantages provided by FPGAs are their fast manufacturing turnaround time, low start-up costs and ease of design that involves less financial risks [11]. With increasing device densities new challenges emerge as one-million gate FPGAs become feasible. One of the concerns in such a scenario is the compile time for FPGAs that includes synthesis, placement and routing time. The challenge is to reduce the compile time without compromising on the quality of solution. The utility of FPGAs suffer from large compile times as design turn around time is crucial. In fact most users desire the compile time to be as low as compile time for C programs [12].

Placement time forms a large part of the compile time. The most popular method for placement is simulated annealing. The Versatile Place and Route (VPR) tool [13], one of the leading tools in academia uses simulated annealing for placement and can be used to place a wide range of FPGA architectures. Simulated annealing however is time-consuming. For the next generation of CAD tools for FPGAs, fast placement methods are critical. Parallelization is an appealing solution for providing fast placements. In our present work we investigate a range of parallelization techniques for FPGA placement using simulated annealing. We modified VPR's placement routines to implement our parallel simulated annealing techniques. Our modifications reuse the VPR code and the changes made are fully compatible to the VPR router. Hence sequential enhancements to the VPR tool in future can easily be incorporated in our version with parallel placement. We present the experimental results and the involving tradeoffs for each of the parallelization strategies. Our approach is similar to the parallelization techniques by Banerjee et al for standard cell placement [2] [1]. To our knowledge, there has been no previous work in parallel placement for FPGAs.

The contributions of the paper are :

- While there have been many previous parallel algorithms for cell placement for ASIC design, our work is the first on FPGA placement.
- We have taken one of the most widely used publicly available placement tools for FPGAs and parallelized it.
- We have evaluated a wide range of parallel algorithms on both shared memory and distributed memory multiprocessor.
- We have performed detailed experimental evaluation of each algorithm presented using real benchmarks.

The organization of the paper is as follows. Section 2 describes the placement problem for FPGAs and the placement algorithm used in VPR. Section 3 describes the different parallelization strategies implemented and the results obtained. Section 4 summarizes related work and Section 5 concludes the paper.

2 The Placement Problem

Figure 1 shows the generic architecture of an FPGA. The generic structure consists of an array of logic blocks

*This research was supported in part by DARPA under contract DABT-63-97-0035.

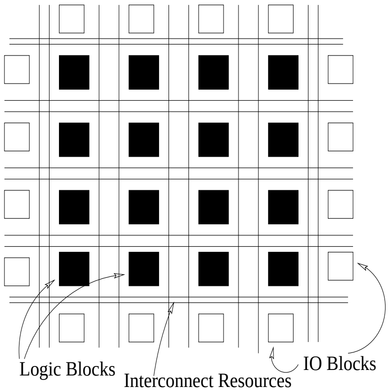

Figure 1: Generic Architecture of an FPGA

that can be configured to realize simple combinational or sequential logic. User configurable IO blocks provide the interface between the external package pins and internal logic. In addition to logic and IO blocks, there are interconnect resources which may be configured to connect logic/IO blocks together. The circuit to be realized in the FPGA is first decomposed into smaller sub-circuits that can each be mapped into a logic block. The placement problem is to map these sub-circuits to the logic blocks of the FPGA so that the placement cost function is minimized. The placement cost function is typically designed to produce a compact placement that facilitates routing. The routing problem is to find a possible way to connect the sub-circuits using the available interconnect resources.

VPR [13] uses simulated annealing for placement. The cost function employed is

$$cost = \sum_{n=1}^{N_{nets}} q(n) \left[\frac{bb_x(n)}{C_{av,x}(n)} + \frac{bb_y(n)}{C_{av,y}(n)} \right]$$

The summation is over all the nets. bb_x and bb_y denote the horizontal and vertical bounding box for each net. $q(n)$ is a compensating factor for the discrepancy between the bounding box wire length model and the actual wire length needed to connect four or more terminals. $C_{av,x}(n)$ and $C_{av,y}(n)$ are average channel capacities in the x and y direction over the bounding box of net n.

The initial temperature is calculated in a manner similar to [14]. VPR uses an innovative annealing schedule where the new temperature is computed from the old temperature as $T_{new} = \alpha \cdot T_{old}$. α is dependent on the fraction of accepted moves (R_{accept}) at T_{old} in a way that lowers the annealing temperature slowly when R_{accept} is high and lowers the annealing temperature quickly in case R_{accept} is low. At each temperature $10 \cdot (N_{blocks})^{1.33}$ moves are evaluated, where N_{blocks} is the number of blocks (logic + IO). The algorithm also uses the range limiter concept that sets a limit on the distance between the two blocks that can be swapped by a move of simulated annealing. At high temperatures the limit is large enabling almost any block to be swapped with any other. As simulated annealing proceeds, the limit is decreased, dependent on R_{accept}. At lower temperatures, the limit becomes small enabling only nearby blocks to be swapped.

More details about VPR placement can be found in [13]. Depending on the size of the circuit, the number of moves evaluated per temperature ($10 \cdot (N_{blocks})^{1.33}$) can be large. For the benchmarks presented, the average number of moves is 4.4×10^7. Such a large number of moves evaluation present a significant amount of computation.

3 Parallel Algorithms for Placement

3.1 Parallel Moves Approach

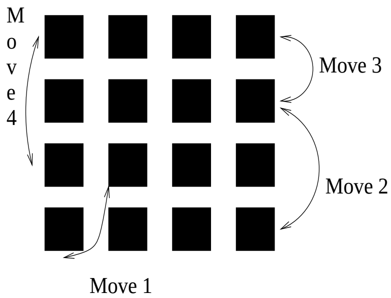

Figure 2: Parallel Moves: Moves 1 and 4 can be done in parallel but not moves 2 and 3

The number of moves evaluated by simulated annealing at each temperature is quite large. The evaluation of a move may result in three cases - **(i)** two blocks are swapped **(ii)** a block is moved to a new(empty) position **(iii)** the move is rejected. A block refers to either a logic block or an IO block to be mapped to an appropriate block of the FPGA. Two moves can be done in parallel provided they do not move the same block(s). Also, while moving a block to an empty position in parallel with other moves, care must be taken so that another block is not moved to the same empty position (Figure 2). However, ensuring the above two conditions does not guarantee the results to be equivalent to sequential execution. Parallel evaluation of moves may incur error while calculating the cost function as it is dependent on the bounding box of the nets containing the blocks. Two moves that move blocks of the same net may evaluate the bounding box incorrectly as each one of the moves can not take into account the fact that the other move is changing the bounding box (Figure 3). There are two approaches one can take - **(1)** Ignore the error in cost function **(2)** Avoid inaccurate computation of bounding boxes by evaluating parallel moves that not only move different blocks, but also blocks that belong to different nets.

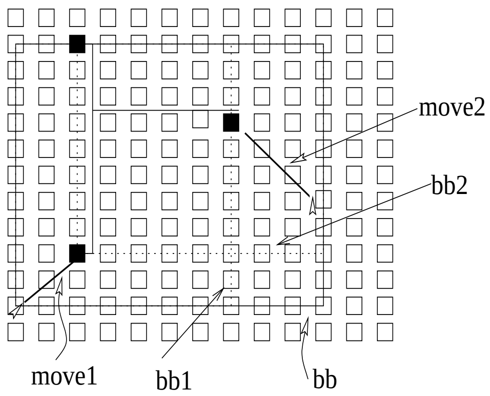

Figure 3: Error in Bounding Box Calculation: If move1 and move2 are done in parallel, they calculate the bounding box as bb1 and bb2 respectively, whereas the actual bounding box is bb.

Both approaches effect the quality of result adversely. The first approach has negative effects because of the error in cost function, which interferes with the acceptance of moves. The second approach restricts the moves of simulated annealing and thus evaluates a smaller search space. We present the results for the second approach. Even the initial results for the first approach showed substantial quality degradation.

Computation of correct bounding boxes can be done in two ways - **(i)** Generate and evaluate moves that move blocks belonging to different nets. **(ii)** Evaluate all moves, but accept only those moves that move blocks belonging to different nets. The second approach has the drawback that the percentage of moves accepted would be low when compared to sequential simulated annealing, provided same number of moves are evaluated. While this problem can be solved by evaluating more moves per temperature in the parallel version, it is not clear exactly how many more moves should be evaluated. Hence we take the first approach. The strategy was implemented for a shared memory machine as the parallelism at moves level is quite fine-grained and is shown in Figure 4. In our design, given N processors, one processor (P_0) generates moves that move blocks belonging to different nets. Other processors evaluate these moves in parallel and either accept or reject them. The generation and evaluation of moves is overlapped to save time, i.e, P_0 generates moves for step $n+1$, while other processors evaluate moves for step n.

```
begin Parallel Moves SA
   if( my_rank == 0){
      do an initial random placement
      find initial temperature
      generate n-1 independent moves }
   while( t > end-temperature ){
      for( number of moves per temperature ){
         if(my_rank==0)
            generate n-1 independent moves
         else
            evaluate move number my_rank generated
            in the previous step}
      update temperature}
end Parallel Moves SA
```

Figure 4: Parallel Placement Algorithm using Parallel Moves

3.1.1 Experimental Results

The parallel moves approach was implemented on an SGI Origin shared memory multiprocessor. Figure 5 and 6 show the variation of execution time and cost with number of processors, respectively. As seen, the parallel implementations show negative speedups. This is due to the fact the overhead of synchronization outweighs the advantages of parallelization. The cost is also affected significantly for the *des* and *bigkey* benchmarks.

3.2 Area Based Partitioning

The problem in the previous approach was that the moves were very restricted. In our next approach we try to alleviate this problem by partitioning the area of the FPGA and assigning the partitioned areas to different processors. Each processor is free to move blocks within its own area. This helps us in two ways -

- There is much less synchronization involved as compared to the previous approach. In the previous approach, the generation of moves by P_0 and evaluation of moves by $P_{1,\ldots,n-1}$ has to be synchronized. In our current approach, each processor can carry out simulated annealing on the area it owns and the point of synchronization is flexible.

- The moves evaluated are much less restricted than the previous approach. In our current approach, the moves evaluated by a processor on its area are done sequentially, and hence the restrictions that arise in the previous approach do not affect us here.

Note that for nets that span two or more partitioned areas, we may still incur error in computation of bounding box (Figure 7).

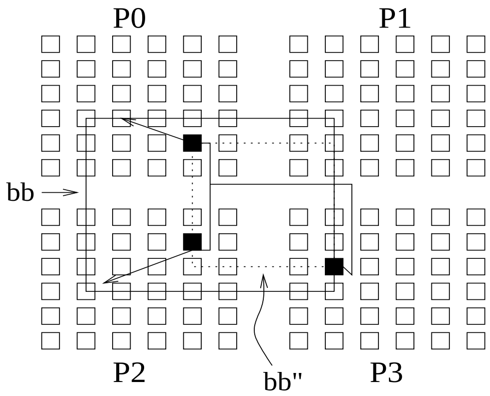

Figure 7: Error in Bounding Box for Area Based Partitioning: For nets that span across partition, bounding boxes may be computed erroneously as the move made by other processors are not communicated immediately.

This is because two or more processors may move blocks of the same net simultaneously. However, we expect this error to be small as compared to the error ignoring approach mentioned in the previous section. This is because in the current approach errors occur only for those nets that span two or more partitioned areas. Moreover, with falling temperature the distance covered by moves are reduced and most of the moves are expected to happen between nearby blocks.

Finally, after periodic intervals (typically after each temperature), all the processors update their data structures to reflect the current placements of the blocks and the bounding boxes of the nets. Thus, each processor gets to know of the moves done by other processors at the end of a temperature. This information can be made more recent to each processor at the expense of more synchronization.

In area based partitioning, a processor can move a block within its area only. Therefore the movement of a block is confined to the partitioned area of FPGA it currently belongs. To allow blocks to move all over the FPGA, the partitioning of area must be changed. Moreover, the sequence of different partitioning schemes should ensure that a block placed in any arbitrary location has the freedom to move to any other arbitrary location in the FPGA. We adopt the following two partitioning schemes. We alternate between the two schemes for successive temperature (Figure 8). An overview of the parallel algorithm is shown in Figure 9.

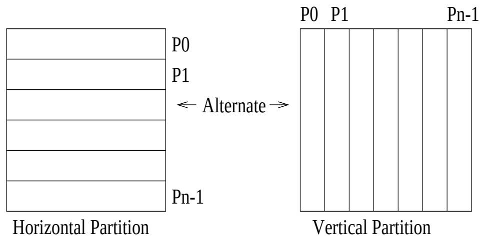

Figure 8: Partitioning schemes for Area Based Partitioning

```
begin Area Partitioned SA
   if( my_rank == 0){
      do an initial random placement
      find initial temperature
      choose a partition(horizontal/vertical) }
   while( t > end-temperature ){
      for( number of moves per temperature )
         generate and evaluate moves in own area
      update data-structures of all processors
      change partition
      update temperature}
end Area Partitioned SA
```

Figure 9: Parallel Placement algorithm using Area Based Partitioning

3.2.1 Experimental Results

The area based partitioning approach was implemented on an SGI Origin. Figure 10 and 11 show the variation of execution time and cost with number of processors, respectively. The timing results show marked improvement when compared to the parallel moves approach. The speedups are due to less synchronization requirements. However, the speedups are not linear. The positive thing is that the cost does not degrade with increasing processors.

3.3 Synchronous Markov Chains

The shortcomings of the approaches in the previous two sections is that they fail to maintain the quality of the solution. Also the quality is quite unpredictable. This is due to two reasons - **(i)** restricted moves **(ii)** error in cost function. In area based partitioning, we made the moves less restricted by assigning parts of the FPGA to individual processors and giving them the freedom to move blocks within their area. However, the error in cost function affected the quality of results negatively. As our next step, we remove the restriction on moves altogether by assigning the whole FPGA to each processor. Each processor carries out simulated annealing on the whole FPGA, starting with a different random seed. To avoid concurrent updates to data structures, each processor does simulated annealing on a local copy of the FPGA (Figure 12). At periodic intervals, the results from all the processors are combined. The result of the processors can be combined in different ways -

- Take the best placement of each net and combine them together to form the new combined placement. In case of conflicts, find a new placement for the conflicting nets iteratively.
- Among all the processors, take the best placement obtained by a processor as the new combined placement.

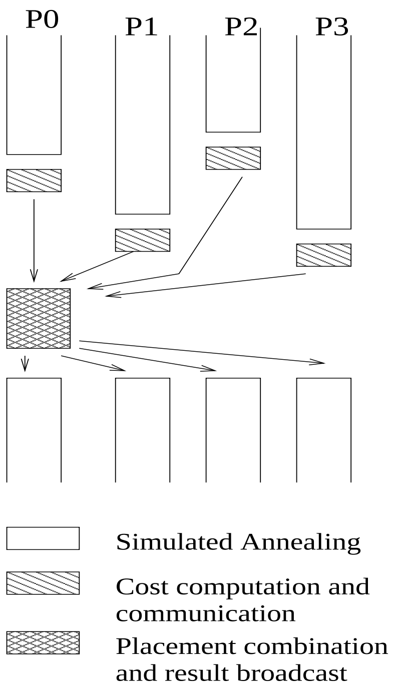

Figure 12: Synchronous Markov Chains

The first approach of combining best configurations of individual nets is non trivial and may consume significant computation time (even when net placements are non conflicting). The second approach is very efficient in terms of computation time, and as we found from experimental results, does quite well in terms of preserving the quality of solution. If we consider simulated annealing as a search path where moves are proposed and either accepted or rejected depending on a particular cost evaluation and a random seed. Each search path can be viewed as a Markov Chain. Our current approach then essentially implements parallel Markov Chains. Our approach is similar to that presented in [2]. To achieve speedup we reduce the number

```
begin Synchronous Markov Chain SA
   generate initial random placement
   find initial temperature
   initialize counter to 0
   while( t > end_temperature ){
      increment counter
      for( number of moves per temperature )
         generate and evaluate moves
      if( counter% update_frequency == 0){
         best_rank= rank of the processor
         with best placement
         if( my_rank ==best_rank)
            broadcast placement to all other
            processors}}
end Synchronous Markov Chain SA
```

Figure 13: Parallel Placement Algorithm based on Synchronous Markov Chains

of moves evaluated at each temperature by $\frac{1}{N}$, where N is the number of processors. Note that we could have reduced the number of moves being evaluated at each temperature by an arbitrary factor, thus obtaining any desired speedup. Of course, an arbitrary reduction in the number of moves will degrade the quality of result. The quality of result also depends on the periodicity with which the results of the different processors are combined. The number of moves after which the results of the processors are combined is referred to as *update_frequency*. An overview of the parallel algorithm is shown in Figure 13.

3.3.1 Experimental Results

The synchronous Markov chain approach was implemented on an IBM-SP2 distributed memory multiprocessor. Figures 14 and 15 show the variation of execution time and cost with number of processors, respectively. The timing graph shows near linear speedups. This is due to the fact that synchronization is minimal. Also due to the nature of the problem, the load is quite evenly distributed. There is a gradual decrease in the quality of solution with increasing processors, except for *des* which shows a substantial degradation. *des* also shows significant degradation for the parallel moves approach which suggests that the benchmark is very sensitive to any alteration from traditional simulated annealing.

3.4 Asynchronous Markov Chains

Our final approach is conceptually similar to the previous approach. In our present approach we improve upon the synchronization requirement of the approach in the previous section. Our approach is similar to the approach presented in [5] [2]. In the previous approach, the synchronization requirement was quite strong, as each processor had to complete a pre-determined number of iterations before the results could be combined. The combination of result involves determining the best placement and distributing it to all the processors. In our current approach, instead of combining the results of the processors synchronously, we make the combination asynchronous. A server maintains the best cost and placement. At periodic intervals, processors query the server. If their current placement is better than the server's best placement, they export their placement to the server. Otherwise they import the server's placement. Thus the synchronization across all the processors is removed. There are two design choices regarding the server. The server may also carry out simulated annealing or may just service queries. The tradeoff is between more work done by assigning work to the server or servicing the queries faster, thereby giving more time to the other processors to work. For a very small number of processors, the server may also do simulated annealing. But in a scalable design, the server is better off servicing queries only. In our design the server services queries only (Figure 16). An overview of the algorithm is given in Figure 17.

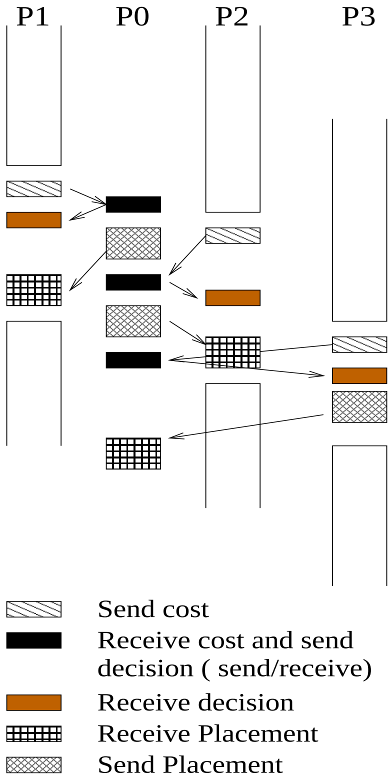

Figure 16: Asynchronous Markov Chains

```
begin Asynchronous Markov Chain SA
   if( my_rank == 0){
      initialize best_cost to infinity
      initialize best_placement to null
      while( other processors so annealing){
         receive cost from processor P
         if( cost of P < best_cost){
            best_cost= cost of P
            best_placement=receive placement from P}
         else{
            send best_cost to P
            send best_placement to P}}}
   else{
      generate initial random placement
      find initial temperature
      initialize counter to 0
      while( t > end_temperature ){
         increment counter
         for( number of moves per temperature )
            generate and evaluate moves
         if( counter%update_frequency == 0){
            send P0 current cost
            send/receive placement from P0}}}
end Asynchronous Markov Chain SA
```

Figure 17: Parallel Placement Algorithm based on Asynchronous Markov Chains

3.4.1 Experimental Results

The asynchronous Markov chains approach was implemented on an IBM-SP2. Figure 18 and 19 show the variation of execution time and cost with number of processors, respectively. The characteristics of the curves are similar to the synchronous Markov chain implementation. The speedups are more close to linear and the cost degradation is much more gradual.

4 Related Work

Several approaches to parallelize simulated annealing have been proposed in the domain of cell placement and can be broadly classified into two categories :

1. *Move Acceleration* In this approach the evaluation of individual moves is parallelized by doing the different tasks involved in evaluating a move in parallel. The available parallelism in this approach is limited and implementation is restricted to shared memory model.

2. *Parallel Moves* In this approach multiple moves are evaluated concurrently. The concurrent evaluation of moves may suffer from inaccurate evaluation of the cost function. A range of alternatives have

been proposed to counter the inaccuracy in cost function evaluation. These alternatives again can be classified into two categories -

(a) *Avoiding Error* These methods involve generating and evaluating moves that do not interact so that there may be inaccuracy in the evaluated cross function. Deciding which moves are not interacting, however is not trivial.

(b) *Tolerate Error* These methods ignore the error in cost function evaluation for parts of the annealing. Errors are corrected after certain moves by synchronizing with other processors. A large spectrum of algorithms exist that differ in the way the problem is partitioned and the frequency and mechanism of synchronization.

Kravitz and Rutenbar [9] report a speedup of 2 on 4 processors for the move acceleration approach and a speedup of 3.5 on 4 processor for the avoiding error approach on a shared memory multiprocessor. Banerjee, Jones and Sargent [3] present a variety of partitioning methods for the parallel moves approach on a hypercube. They obtained a speedup of 12 for 16 processors. Several other works on parallelizing simulated annealing have been reported for different applications and on different architectures. Casotto et al. [10] achieved speedup of 6 on 8 processors for placement of macrocells on a shared memory multiprocessor. Rose et al. [8] propose a hybrid algorithm of min-cut algorithm and simulated annealing that achieve a speedup of 4 on 5 processors. Sun and Sechen [6] show near linear speedups for the parallel moves approach on a network of workstations. Banerjee, Kim, Ramkumar, Parkes and Chandy [1] present a range of algorithms based on parallel simulated annealing for standard cell placement.

5 Conclusion

Our work in this paper is the first one to evaluate parallel placement algorithms for the FPGA placement application. We have investigated a range of parallel simulated annealing algorithms for FPGA placement. The parallel moves approach does not seem very promising due to loss of speedup tight by synchronization requirements and degradation in quality of result because of restricted moves. The second approach of area based partitioning provides better speedups and quality of solution. The speedup obtained is mainly due to reduction in synchronization. In the same direction the Markov chains approach reduces the synchronization requirement significantly and we observe near linear speedup. Markov chains also prove promising in terms of quality of result.

References

[1] J. A. Chandy, S. Kim, B. Ramkumar, S. Parkes and P. Banerjee. An Evaluation of Parallel Simulated Annealing Strategies with Applications to Standard Cell Placement. In *IEEE Trans. on Computer Aided Design*, Vol.16, April 1997, pp.398-410.

[2] J. A. Chandy and P. Banerjee. Parallel Simulated Annealing Strategies for VLSI Cell Placement. In *Proc. 9th International Conference on VLSI Design*, Bangalore - India, Jan. 1996.

[3] P. Banerjee, M. H. Jones and J. S. Sargent. Parallel simulated annealing algorithms for standard cell placement on hypercube multiprocessors. In *IEEE Trans. on Parallel and Distributed Systems*, Vol.1, Jan. 1990, pp.91-106.

[4] S. Kim, J. A. Chandy, S. Parkes, B. Ramkumar and P. Banerjee. ProperPLACE: A portable parallel algorithm for cell placement. In *Proc. of International Parallel Processing Symposium*, Cancun - Mexico, Apr.1994, pp.932-941.

[5] S. Y. Lee and K. G. Lee. Asynchronous communication of multiple Markov Chains in parallel Simulated Annealing. In *Proc. International Conference on Parallel Processing*,Aug. 1992,Vol.III,pp.169-176.

[6] W. J. Sun and C. Sechen. A loosely coupled parallel algorithm for standard cell placement. In *Digest of papers, International Conference on Computer Aided Design*,San Jose,Nov. 1994,pp.137-144.

[7] C. Sechen and K. W. Lee. An improved simulated annealing algorithm for row-based placement. In *Digest of papers, International Conference on Computer Aided Design*,Santa Clara,Nov. 1987,pp.478-481.

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

[9] S. A. Kravitz and R. A. Rutenbar. Placement by simulated annealing on a multiprocessor.In *IEEE Trans. Computer Aided Design*,Vol.CAD-6,July 1987,pp.534-549.

[10] 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*,Vol.CAD-6,Sept.1987,pp.838-847.

[11] S. Brown and J. Rose. FPGA and CPLD Architectures: A Tutorial. In *IEEE Design and Test of Computers*, Vol.12, Summer 1996, pp.42-57.

[12] J. Rose and D. Hill. Architectural and Physical Design Challenges for One-Million Gate FPGAs and Beyond. In *FPGA'97, ACM Symp. on FPGAs*, Feb. 1997, pp.129-132.

[13] V. Betz and J. Rose. VPR: A New Packing, Placement and Routing Tool for FPGA Research. In *7th International Workshop on Field-Programmable Logic*, Londan, August 1997, pp. 213-222.

[14] M. Huang, F. Romeo and A. Sangiovanni-Vincentelli. An Efficient General Cooling Schedule for Simulated Annealing. In *ICCAD* , 1986,pp.381-384.

Parallel Non-interacting Moves

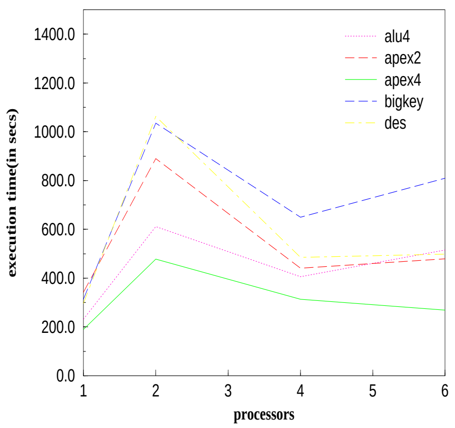

Figure 5: Variation of execution time for Parallel Moves algorithm on an SGI shared memory multiprocessor. Time shown for 1 processor corresponds to the serial algorithm.

Parallel Non-Interacting Moves

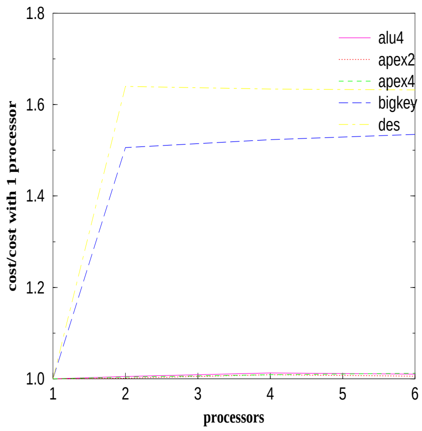

Figure 6: Variation of Normalized Placement Cost for Parallel Moves algorithm on an SGI shared memory multiprocessor. Cost shown for 1 processor corresponds to the serial algorithm.

Area Based Partioning

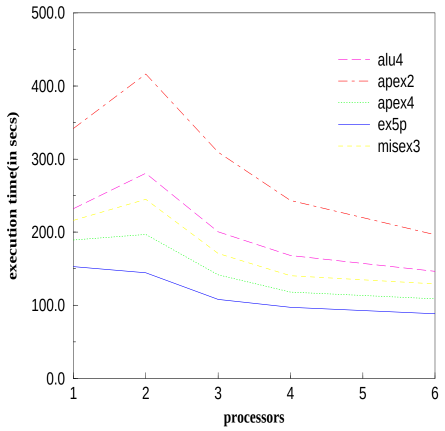

Figure 10: Variation of execution time for Area based Partitioning algorithm on an SGI shared memory multiprocessor. Time shown for 1 processor corresponds to the serial algorithm.

Area Based Partitioning

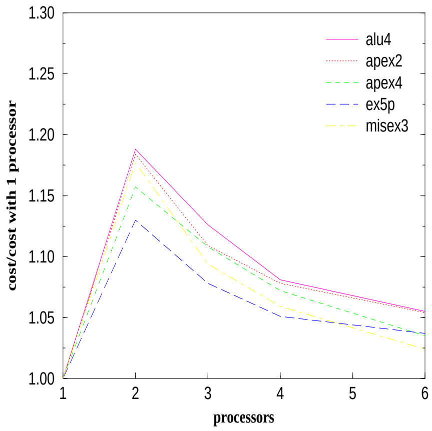

Figure 11: Variation of Normalized Placement Cost for Area based Partitioning algorithm on as SGI shared memory multiprocessor. Cost shown for 1 processor corresponds to the serial algorithm.

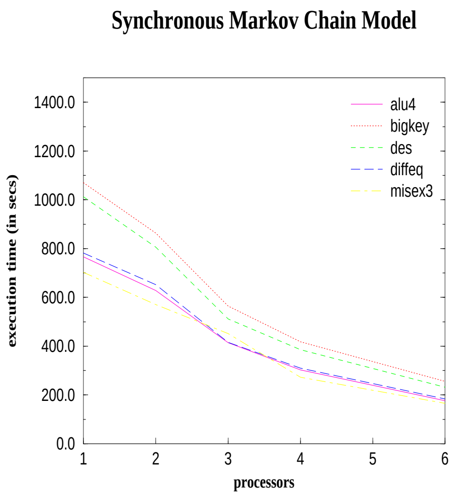

Figure 14: Variation of execution time for Synchronous Markov Chains algorithm on an IBM-SP2 distributed memory multiprocessor. Time shown for 1 processor corresponds to the serial algorithm.

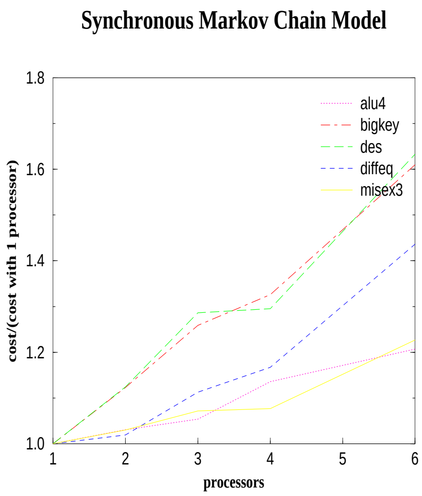

Figure 15: Variation of Normalized Placement Cost for Synchronous Markov Chains algorithm on an IBM-SP2 distributed memory multiprocessor. Cost shown for 1 processor corresponds to the serial algorithm.

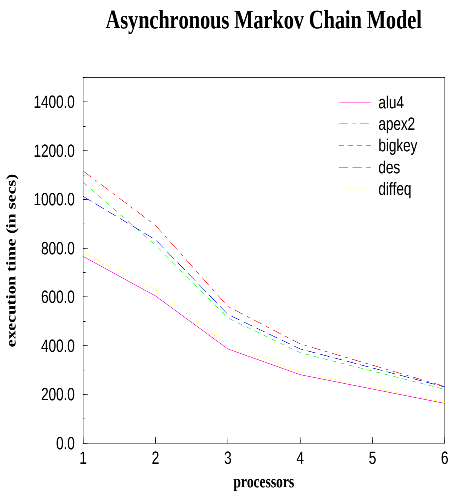

Figure 18: Variation of execution time for Asynchronous Markov Chains algorithm on as IBM-SP2 distributed memory multiprocessor. Time shown for 1 processor correspond to the serial algorithm.

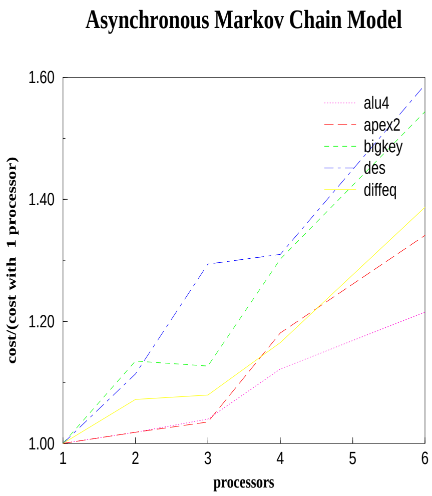

Figure 19: Variation of Normalized Placement Cost for Synchronous Markov Chains algorithm on as IBM-SP2 distributed memory multiprocessor. Cost shown for 1 processor correspond to the serial algorithm.