PARALLEL ALGORITHMS FOR STANDARD CELL PLACEMENT USING SIMULATED ANNEALING

BY

JOHN ATTUPURATHU CHANDY

S.B., Massachusetts Institute of Technology, 1989
M.S., University of Illinois at Urbana-Champaign, 1993

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Electrical Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1996

Urbana, Illinois

PARALLEL ALGORITHMS FOR STANDARD CELL PLACEMENT
USING SIMULATED ANNEALING

John Attupurathu Chandy, Ph.D.
Department of Electrical and Computer Engineering
University of Illinois at Urbana-Champaign, 1996
Prithviraj Banerjee, Advisor

As modern VLSI designs have become larger and more complicated, the computational requirements for design automation tools have also increased. As a result, the parallelization of these tools is of great importance. One of the more computationally intensive parts of the entire VLSI design process is the placement process. Simulated-annealing-based approaches have been the most popular and effective methods for cell placement. In this thesis, parallelization approaches to simulated-annealing-based standard cell placement are presented.

In this work, four parallel algorithms have been investigated, with two that provide scalable behavior as well as acceptable quality. The first is the parallel moves approach based on work by Kim [1, 2]. The second algorithm is a multiple Markov chains approach that gives nearly linear speedups with very little loss of quality. This approach is suitable for small scale multiprocessors and for circuits that are small enough to fit in the memory of a single node. The next algorithm is known as speculative computation and is not as effective. The final algorithm addresses the memory scalability problems by partitioning the circuit across the nodes. This circuit-partitioned approach provides speedups to larger numbers of processors with little loss of quality. All of the algorithms have been implemented using the ProperCAD II environment [3], and the circuit-partitioned work has also been implemented using the Message Passing Interface (MPI) [4].

The placement algorithms discussed above dealt only with minimization of the wirelength and indirectly area minimization. For current high density circuits, this approach is no longer appropriate, and more performance driven techniques are needed. We have, therefore, also developed a new algorithm for sequential timing driven cell placement. Because the addition of timing driven features to standard cell placement adds significant overhead to the computation, time, we have also developed an algorithm for its parallelization.

To my family

ACKNOWLEDGMENTS

I am grateful and indebted to my advisor, Professor Prithviraj Banerjee, for his continuous encouragement and guidance. His ready support and accessibility have made it easy to work with him. I would also like to thank the members of my committee, Professors W. Kent Fuchs, Ibrahim Hajj, and C. L. Liu, for their valuable insights and comments in guiding this thesis.

I would like to thank all of the members of the ProperCAD and PARADIGM research groups for their friendship as well as their technical help. John Holm, Antonio Lain, Dan Palermo, Amber Roy-Chowdhury, and Shankar Ramaswamy, who have been here for most of my five years at Illinois, have been invaluable in my survival here. I would especially like to thank Steven Parkes for our countless discussions and for his assistance with ProperCAD. I would also like to acknowledge Sungho Kim for his guidance in the specific area of placement.

Lastly and most importantly, I would like to thank my family, who have always supported me in all my endeavors and continue to offer their constant encouragement and understanding.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Figure **Page**

CHAPTER 1

INTRODUCTION

1.1 Motivation

With the rapid advances in VLSI process technology, circuit design is becoming increasingly complex and in turn is placing ever higher demands on CAD tools. Designs containing millions of transistors are typical, and it is expected that designs may approach 100 million transistors by the end of the decade. The computational resources needed to effectively design these circuits are enormous. Each of the different phases in the VLSI design process can take several hours to several days using existing CAD algorithms in current processing technology. With the sizes of these designs growing, the CAD tools become increasingly taxing on the memory resources of computers. As a result, with many modern designs, it is not possible to effectively use existing CAD tools on the entire design because of memory shortage.

Parallel processing is fast becoming an attractive solution to reduce the inordinate amount of time spent in VLSI circuit design. This fact has been recognized by several researchers in VLSI CAD as evidenced in the recent literature for cell placement, floor planning, circuit extraction, test generation, fault simulation, logic synthesis, etc [5]. Parallel processing can also address the memory issue by using the distributed memory resources on a multiprocessor.

In this thesis, we examine one phase of the design process in detail, namely, standard cell placement. Placement of standard cells is particularly expensive because of the inherent compute intensive nature of simulated annealing, the most popular approach used for cell placement. There have been several attempts to parallelize this algorithm, usually with quality results that do not compare to the best available sequential algorithm, or with speedups that are not acceptable.

In this work, four parallel algorithms have been investigated, with two that provide scalable behavior as well as acceptable quality. The first is the parallel moves approach based on work by Kim [1, 2]. The second algorithm is a multiple Markov chains approach that gives nearly linear speedups with very little loss of quality. This approach is suitable for small scale multiprocessors and for circuits that are small enough to fit in the memory of a single node. The next algorithm, known as speculative computation, is not as effective. The final algorithm addresses the memory scalability problems by partitioning the circuit across the nodes. This circuit-partitioned approach provides speedups to larger numbers of processors with little loss of quality. All the algorithms have been implemented using the ProperCAD II environment [3], and the circuit-partitioned work has also been implemented using the Message Passing Interface (MPI) [4].

The placement algorithms discussed so far have dealt only with minimization of the wirelength and indirectly area minimization. For current high density circuits, this cost function is no longer appropriate. The delays associated with the wiring elements are more critical to the performance of the circuit; thus, steps must be taken to minimize these delays. Timing driven placement is the process of simultaneously minimizing the circuit area as well as minimizing the critical path delays. In this thesis we also introduce a new algorithm for timing driven placement as well as methods of parallelization.

Before describing the parallel algorithms, we will provide a brief overview of the ProperCAD II environment.

1.2 ProperCAD II

The use of parallel platforms, in spite of increasing availability, remains largely restricted to well-structured numeric codes. Irregular applications in terms of data access patterns as well as control flow are difficult to effectively and efficiently parallelize. The use of object-oriented design techniques and the actor model of computation can address the use of parallel platforms

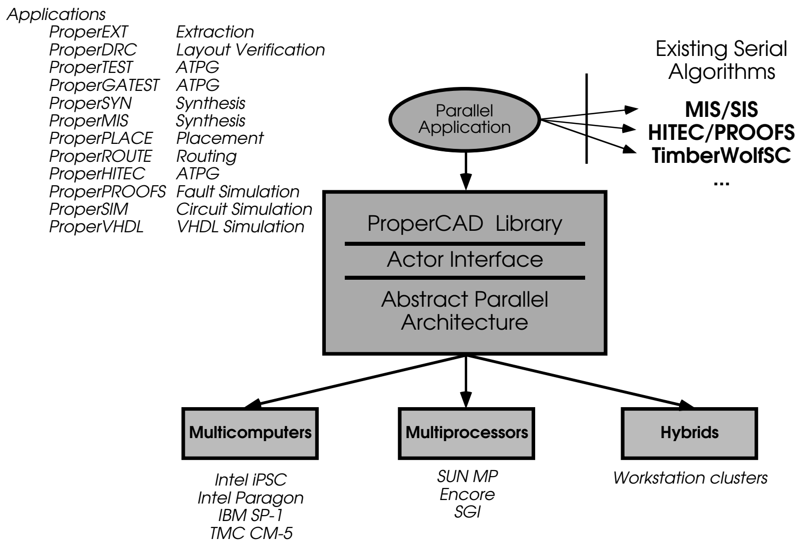


Figure 1.1: An overview of the ProperCAD project.

for unstructured problems. ProperCAD II is an object-oriented library supporting the design of actor-based parallel programs [3, 6]. The library easily allows the design of data structures with parallel semantics for use in irregular applications. Because the foundation is based on C++, inheritance mechanisms allow creation of the distributed data structures from standard C++ objects.

The major goal of the ProperCAD project [7] is to develop portable parallel algorithms for VLSI CAD applications that will run on a range of parallel machines including shared memory multiprocessors such as the Sun SparcServer 1000E and the SGI Challenge, distributed memory multicomputers such as the Intel Paragon, IBM SP-2, and Thinking Machines CM-5, and networks of workstations (Figure 1.1).

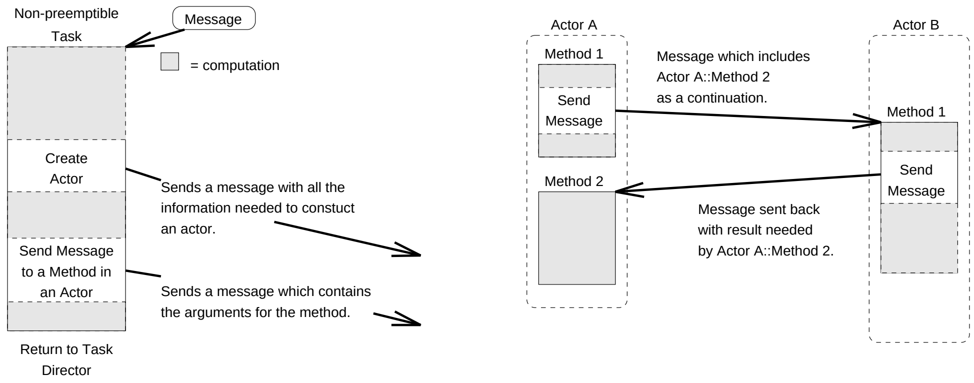
The domain of VLSI CAD provides a rich class of irregular problems. With the rapid advances in VLSI process technology, increasingly complex circuit designs are placing ever higher demands on CAD tools. The computational intensity of these tools makes parallel processing an attractive solution [5]. However, most applications in this area are characterized by complex interrelated data structures as well as irregular access patterns across these objects. These properties make VLSI CAD applications particularly difficult to efficiently parallelize. The use of the ProperCAD II library as well as C++ design techniques help to alleviate this problem. The approach has been used on a wide variety of VLSI CAD problems including test generation [8], fault simulation [9], logic synthesis [10, 11], state assignment [12, 13], layout verification [14, 15], and VHDL simulation [16]. In this thesis, we demonstrate the use of these techniques in a specific VLSI CAD problem, standard cell placement.

1.2.1 Actor basics

The ProperCAD II library expresses parallelism with a statically typed high level C++ actor based interface. The library is class library-based and allows multiple levels of abstraction as well as incremental parallelization. Through the use of a fundamental object called an actor [17], the library provides mechanisms necessary for achieving concurrency. An actor object consists of a thread of control that communicates with other actors by sending messages, and all actor actions are in response to these messages. Specific actor methods are invoked to process each type of message.

Figure 1.2(a) shows the three basic actions that a method in an actor can take: create new actors, send messages to actors, and perform computations that change its state. In formal actor terminology, the changing state is actually a subset of the *become* operation as defined by Agha [17]. When a method creates an actor, a message is sent to the run-time system with all the information needed to construct an actor. When a method sends a message, a message containing the arguments and the identity of the method to be invoked is sent to the run-time system for later execution. Both actor creation and message sends are non-blocking calls. The model only specifies that the actor be created or the task be run sometime in the future. Once a task starts, it runs to completion and cannot be preempted.

The actor model lacks explicit sequencing primitives. Synchronization is implicit and arises because of the single-threaded nature of individual actors. The return executed at the completion of an actor method is an implicit wait; the actor automatically becomes available for any pending method invocations. Since an actor cannot suspend execution implicitly in the middle



(a) Actor Behaviors

(b) Continuation Passing Style of execution.

Figure 1.2: Actor model with continuation passing.

of a computation, *continuation-passing style* (CPS) [18] is used to express control and data dependencies. Figure 1.2(b) shows an example of continuation passing style. The actor model is a message-driven model in which the method name is in the message and the method is the code invoked upon message reception.

1.2.2 ProperCAD II interface

Applications created with the ProperCAD II library use five basic classes provided by the library: `Actor`, `ActorName`, `ActorMethod`, `Continuation`, and `Aggregate`.

1.2.2.1 Actor

All actor types are derived from the library supplied class, `Actor`. Adding the `Actor` base to a class in a sequential object-oriented program enables the creation of actor methods and continuations as described below. These features allow the expression of parallelism. For example, a user class may be created as follows.

```
class Foo : public Actor { ... };
```

1.2.2.2 ActorName

Actor names serve the role of pointers and references for instances of actor classes. Because normal pointers are not valid across processor boundaries, actor names provide the mechanism for access of actors in a global namespace.

```
Foo* actorPtr = ...;
ActorName<Foo> fooName = actorPtr;
```

1.2.2.3 ActorMethod and Continuation

`ActorMethods` are member functions which may be invoked asynchronously and remotely. `ActorMethods` are executed via `Continuations`, the concurrent equivalent of member function pointers. An example of the definition and use of these constructs is shown below.

```
class Foo : public Actor {
   Foo( fooArgs& );
   void bar( barArgs& );
   class New : public NewActorMethod<fooArgs> {};
   class bar : public ActorMethod<barArgs> {};
}
```

The `Foo` actor has a constructor which takes `fooArgs` as an argument as well as a method `bar()` which takes `barArgs` as an argument. In order to allow creation of this actor remotely, we need to add to the class a special nested class, New, which is derived from a templated `NewActorMethod`. Likewise, we can designate `bar()` as an actor method by creating a new nested class `bar` derived from a templated `ActorMethod`. The code below shows the use of these constructs.

```
{
...
fooArgs &fargs
ActorName<Foo> fooName = ActorName<Foo>::newName();
Foo::New::Continuation cont1 ( fooName );
cont1( fargs );
barArgs &bargs;
Foo::bar::Continuation cont2 ( fooName );
cont2( bargs );
...
}
```

To create the actor, we first optionally assign an `ActorName` to it. We then create a `Continuation` `cont1` bound to that `ActorName`. Execution of the continuation will schedule construction of the actor. Note, that the actor is not created at this point but it is deferred until some point in the future. We also have specified where the actor should be created, though this may optionally be indicated when the continuation is created. In order to invoke `bar()` asynchronously, we similarly create a `Continuation` `cont2` bound to `fooName`. We can now treat `cont` as member function pointer and execute it directly causing a message send, or pass it to another method. In this particular example, we execute it and because the actor has not been created but simply scheduled for creation, the message becomes pending, waiting for construction of the actor.

1.2.2.4 `Aggregate`

Individual actors express neither internal parallelism nor data distribution. Collection types, based on aggregates with explicit distributions, allow both object-internal concurrency as well as data distribution. An aggregate is simply a collection or group of actors which share a common name [19]. An example of an aggregate would be a distributed array where different elements are stored on different actors. The use of aggregate representations removes the serialization step that would be required because of a gateway actor. The interface is similar to that of the `Actor` class, and the creation and use of names and actor methods are accomplished similarly.

```
class FooAggr : public Aggregate { ... };
```

Aggregates provide the necessary mechanisms for distributed data structures. Because of the standard C++ interface, access to these distributed data structures is efficient. The benefit of aggregates is apparent particularly in the circuit partitioned algorithm for parallel cell placement described in Chapter 6.

1.2.3 Related work in parallel software environments

Several other researchers have produced work in environments to support irregular applications in object-oriented environments, such as Charm++ [20], CC++ [21], Concurrent Aggregates/Concert [19, 22, 23], pC++ [24], and SDDG/DAGH [25]. Charm++ provides similar run-time support for message driven applications to ProperCAD II. The primary differences are Charm++'s lack of support for static message typing, as represented by first-class continuations, and composability. Concurrent Aggregates is a pure actor functional language with support for aggregates. pC++ is a language extension of C++ with support for data parallel semantics in much the same manner as HPF [26]. Since it presents a data parallel view of the world, it is difficult to express irregular problems such as VLSI CAD in this framework. CC++ achieves concurrency through parallel constructs which allow particular code fragments to be performed on different processing threads. This task parallelism approach can mimic many of the features in an actor model. However, the ProperCAD II library does provide extra meta programmability features that allow the program designer to change the behavior of the run time system such as queuing policies, memory usage, and load balancing. SDDG/DAGH provides a collection of C++ distributed data structures to support parallel adaptive finite difference codes based on hierarchical adaptive mesh-refinement methods.

Other work to support irregular applications but are not targeted toward object-oriented environments include Multipol [27] and PARTI/CHAOS [28]. Multipol provides a library of distributed data structures for use with a message driven run-time system. The basic block of computation is called an atomic thread, which is essentially the functional equivalent of an actor method. The PARTI/CHAOS library offers irregular run-time support for iterative irregular computation in which the communication pattern is unchanged and predictable, but not resolvable at compile time. The library is most appropriate for finite element computations. Neither Multipol nor PARTI/CHAOS allows parallelism via derivation as available in ProperCAD II.

1.3 Thesis Outline

This thesis is organized as follows. Chapter 2 covers the background and related work in parallel standard cell placement. Chapter 3 describes the parallel moves approach based on work by Kim [1, 2]. Chapters 4 and 5 describe the multiple Markov chains and speculative computation algorithms, respectively. The final parallel area driven placement algorithm, circuit partitioned, is discussed in Chapter 6. The algorithms for timing driven placement are introduced in Chapter 7. The last chapter contains the conclusions and directions for future research.

CHAPTER 2

BACKGROUND AND RELATED WORK

Standard cell based design methodology allows a designer to build his or her design from a library of predefined modules or cells. The placement problem involves placing these cells on a VLSI layout, given a netlist that provides the connectivity between each cell and a library containing layout information for each type of cell. This layout information includes the width and height of the cell, the location of each pin, the presence of equivalent pins, and the possible presence of feed through paths within the cell. The primary goal of cell placement is to determine the best location of each cell so as to minimize the total area of the layout and the length of the nets connecting the cells together. With standard cell design, the layout is organized into equal height rows, and the desired placement should have equal length rows, as shown in Figure 2.1.

2.1 Simulated Annealing

One of the more powerful algorithms for standard cell placement has been simulated annealing. It is a suitable approach to problems like VLSI cell placement because they lack good heuristic algorithms. Simulated annealing is analogous to the materials science problem of annealing of solids. When annealing solids, the goal is to bring the solid into a low energy state, for example, to generate crystal lattices in silicon or to soften glass and metals. The physical process involves heating the material to allow atomic motion. Gradually, the material is cooled carefully until the material freezes into the desired state. Briefly then, simulated annealing is an iterative optimization software strategy that starts with a system in a disordered state, and

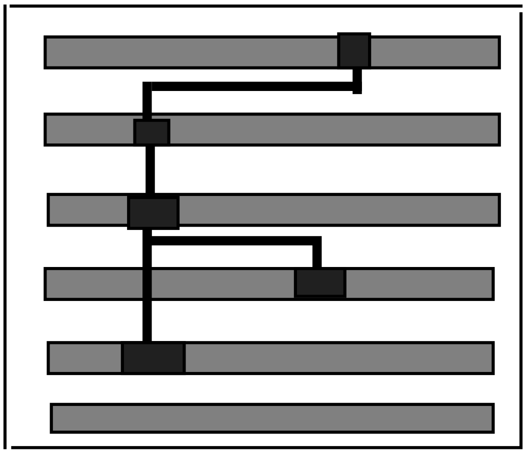


Figure 2.1: Standard cell placement.

through perturbations of the state, brings the system gradually to a low energy, and thus optimal, state [29, 30]. The energy is a cost function of the system that is to be minimized. In the context of cell placement, perturbations are simply moves of the cells to different locations on the layout, and the energy is an approximated layout cost function.

As moves are made, any move that reduces the cost function is accepted. However, simulated annealing, unlike greedy algorithms, will also allow moves that increase the cost. The effect of this change is to allow the solution to escape from local minima. In cases where the cost is increased, the new state is accepted with probability

$$p = e^{-\frac{\Delta C}{T}} \tag{2.1}$$

where ΔC is the change in the cost or energy and T is the temperature of the system. The temperature is an analog of the effect of temperature in crystal annealing. We start with an extremely high temperature to allow nearly all moves to be accepted. Gradually, the temperature is reduced until a termination condition is reached.

Theoretical studies show that simulated annealing is guaranteed to reach an optimal solution given enough time and proper monitoring of the temperature or annealing schedule. To achieve this, at each temperature, the system must be at equilibrium before the temperature is lowered again. However, it is impossible to guarantee equilibrium in finite time. In computing

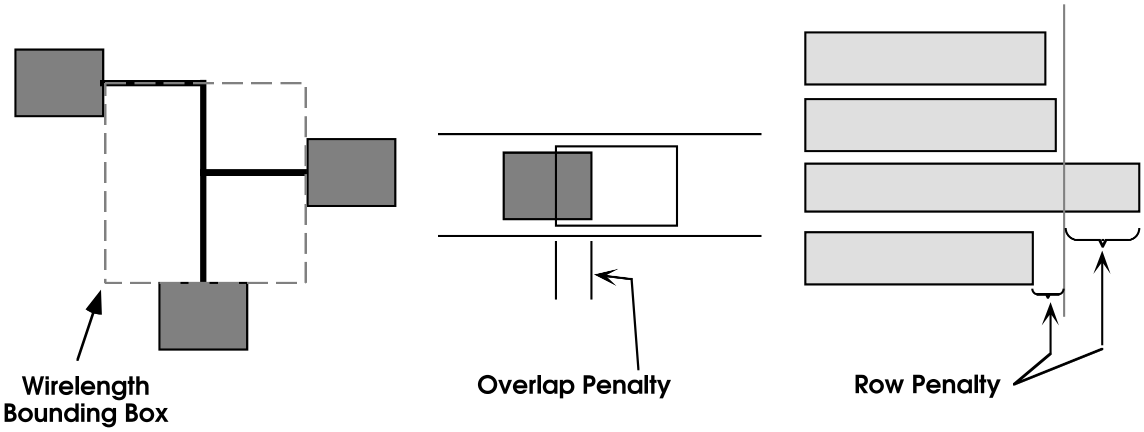


Figure 2.2: `TimberWolfSC` cost function components.

applications, it is impractical to wait for the system to achieve equilibrium before changing the temperature, so heuristics are used to develop a fast and near optimal schedule [31–34].

2.2 `TimberWolfSC`

One of the more popular uses of simulated annealing for placement has been the `TimberWolfSC` cell placement tool [35–37]. The `TimberWolfSC` cost function is defined in Eq. (2.2) where W is the bounding box half perimeter estimate of the wirelength of all nets, P_O is the penalty for the area overlap between cells in the same row, and P_R is the penalty for the difference between the actual row length and the desired row length. These cost parameters are shown graphically in Figure 2.2. The coefficient terms μ and λ are adjusted using a feedback control mechanism to arrive at optimal values.

$$C = W + \mu P_O + \lambda P_R. \tag{2.2}$$

Moves are generated by choosing a random cell and then displacing it to a random location on the layout. If a cell is already present at the new location, the two cells are exchanged. A temperature dependent range limiter is used to limit the distance over which a cell can move. Initially, the span of the range limiter is set such that a cell can move anywhere on the layout. Subsequently, the span is decreased logarithmically with temperature. These range limiter updates are made at the end of each of the 160 iterations into which `TimberWolfSC` segments the simulated annealing procedure. As the algorithm progresses, the temperature is gradually decreased by forcing the acceptance rate to follow a theoretically derived schedule that attempts to keep the acceptance rate close to 44% during the middle region of annealing [33]. `TimberWolfSC 6.0` also uses row bins to aid in the computation of overlap and row penalties, and early rejection methods are used to speed up the decision process [38].

2.3 Parallel Annealing Algorithms for Placement

Because of the inherent computational costs associated with simulated annealing, several methods have been proposed for the parallelization of the procedure [39]. Using the taxonomy defined in [40], there are three major classes of parallel simulated annealing algorithms: *serial-like*, *asynchronous*, and *altered generation*.

Serial-like algorithms preserve the convergence characteristics of the sequential algorithm through the use of single move acceleration or serializable subsets. Kravitz and Rutenbar have investigated both approaches and found that these algorithms have limited parallelism and are more appropriate for shared memory architectures [41].

The second class of parallel simulated annealing techniques, altered generation, is distinguished from serial-like algorithms in that they do not follow the exact search space laid out by the sequential algorithm. This is usually accomplished with a processor or group of processors either exploring a restricted state space or using a restricted search on the entire state space. To ensure proper global convergence, the global state is kept up to date through periodic solution exchanges or with a shared memory architecture. Parallel placement algorithms using this strategy for shared memory machines include work by Darema et al. [42] and Natarajan and Kirkpatrick [43]. Sun and Sechen have recently shown results achieving near linear speedup on a network of workstations [44]. This method shows great promise for a few processors, but it is unlikely that the convergence properties will hold as more processors are used.

The final class of parallel simulated annealing algorithms is the asynchronous or "parallel moves" algorithm where each processor generates and evaluates moves independently. This differs from altered generation methods in that the state space is not restricted. In other words, each processor contains information on the entire circuit regardless of whether the global layout information is accurate in the local processor. Obviously, the cost function calculations may be incorrect because of the moves made by the other processors. There are various methods to address the effect of error, but all involve some form of periodic updates. The number of updates is directly related to the average acceptance rate of the particular annealing scheduling chosen. Banerjee, Jones and Sargent [45] implemented a parallel placement algorithm using the parallel move approach on an Intel hypercube multiprocessor and proposed several partitioning strategies for the problem specific to the hypercube topology. Speedups of up to 12 on 16 processors were reported. Rose et al. [46] proposed a parallel algorithm on an experimental distributed memory multiprocessor. In that algorithm, they replaced the high temperature portion of the parallel simulated annealing placer with a placement program based on a min-cut algorithm and used a parallel moves strategy for lower temperatures. Speedups of 4 on five processors were reported.

The only reported instance of large scale parallelism being applied to cell placement is the use of parallel moves for SIMD machines. Both Casotto and Sangiovanni-Vincentelli [47] and Wong and Fiebrich [48] have presented similar parallel simulated annealing placement algorithms for the SIMD Connection Machine. These methods fall in between a completely asynchronous approach and the altered generation methods. By completely distributing the circuit state, the necessity for global updates is removed, while still allowing for asynchronous parallel moves.

CHAPTER 3

PARALLEL MOVES APPROACH

3.1 Algorithm

In this chapter, we describe `ProperPLACE-PM`, an implementation of parallel moves based cell placement derived from work by Kim et al. [1, 2]. In the parallel moves algorithm, each processor generates and evaluates moves independently. This leads to cost function inaccuracies that must be adequately addressed to arrive at good solutions. `ProperPLACE-PM` does so through the use of periodic updates with specific dynamic error control mechanisms.

The application begins with a random input placement that is replicated on each available physical processor. Using the aggregate class provided by the ProperCAD II library, an aggregate named `CircuitAggr` is constructed to manage access to the circuit structure and maintain a coherent state of the current placement. Each processor will have one representative of the aggregate responsible for its local copy of the circuit. In addition, an `AnnealActor` actor is created per physical processor to perform the annealing steps, i.e. move, evaluate, and decide. The placement is divided up topographically by rows, with the rows and its cells assigned to separate `AnnealActor` actors. Each actor is responsible for one row, and thus is only allowed to attempt moves on cells in that row. If a cell is moved to a region owned by another actor, the ownership of the cell is transferred to the new actor and the original actor is no longer responsible for moving that cell. Because an entire row, not a sub part, is owned by an actor, there will be no error in the calculation of cell overlaps and row lengths during the simultaneous evaluation of multiple moves. Note that this approach assumes that the number of rows is greater than or equal to the number of actors. If not, the rows must be split into a number of subrows, in which case, some overlap penalties may be calculated erroneously.

After partitioning, each `AnnealActor` actor proceeds with the annealing algorithm outlined in Figure 3.1. A valid cell is selected for perturbation, and then a displacement or exchange is performed on that cell. As detailed below, there are two subclasses of moves for both displacement and exchange, or four move types in total. The move type is determined by the intended location of the selected cell A.

M1. Intra-actor Cell Displacement. Cell A moves to new location owned by the same actor.

M2. Intra-actor Cell Exchange. Two cells A and B owned by the same actor exchange their locations.

M3. Inter-actor Cell Displacement. Cell A moves to new location owned by a different actor.

M4. Inter-actor Cell Exchange. Two cells A and B owned by different actors are exchanged.

An example of each type of move is shown in Figure 3.2. In the figure, assume that each row is owned by different `AnnealActor` actors. Note that the three moves (M1, M2, M3) can be done alone by actor 0, the owner of cell A. For the move M4, however, actor 0 needs permission from actor 1 which owns cell B, as it is possible that cell B may have already been moved to another location or is frozen because of some pending move. Because the information about cell B may be out of date in the database of actor 0, it locks (or freezes) cell A and cell B and sends an `AskPermission` message to actor 1. After receiving the `AskPermission` message, actor 1 examines the state of cell B and determines whether to allow the exchange. The decision is sent back to actor 0 by sending the `ReturnAnswer` message. Upon receipt of the `ReturnAnswer` message, actor 0 unlocks cells A and B, and the move is attempted if the returned answer is yes. Actor 0 does not wait idly until the `ReturnAnswer` message is received - instead, it continues annealing by making other moves with unfrozen cells that it owns.

If a move is accepted, the accepting actor must send the move to the `CircuitAggr` aggregate so that a consistent cell position database can be maintained. To amortize the startup cost

```
ProperPLACE-PM()
while termination not reached
    do while equilibrium not reached
        do select cell A in this actor's region
           select new location for cell A
           if cell B at new location
             then move is an exchange
           determine move type (M1, M2, M3 or M4)
           evaluate move cost
           if ACCEPT(A, B, move) == yes
             then if move == M4
                    then FREEZE(A, B)
                         send AskPermission msg to Circuit
                    else accept the move
                         update local database
                         send Update msg to Circuit

       lower T

```

Figure 3.1: Outline of the `ProperPLACE-PM` algorithm.

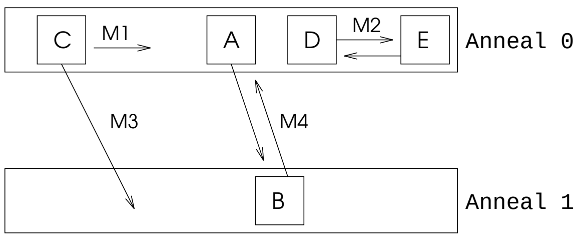


Figure 3.2: Moves in `ProperPLACE-PM`.

of sending a message, position update messages are held until a number of moves have been accepted. Although this reduces the total number of update messages sent among processors, there is a drawback in this approach. As the frequency of update messages is reduced, the cell position database on each `CircuitAggr` representative becomes increasingly inaccurate, thereby causing the cost function calculation error to increase as well. This error, if too large, may prevent the algorithm from converging to an optimal solution. Previous researchers [45, 47, 49] have shown that simulated annealing is tolerant to some error in cost function calculations.

Since actor methods are non-blocking, the actor's annealing process must give up control every so often to allow the aggregate to gain computation time to respond to the update and `AskPermission` messages. Therefore, a limit is placed on the number of moves that may be performed in succession without interruption. The `CircuitAggr` aggregate can then process any waiting messages. `AnnealActor` will have rescheduled itself by sending itself a message that will enable control to come back and the next set of moves can then be proposed and evaluated.

`ProperPLACE-PM` includes a dynamic error control mechanism that adaptively controls the frequency of the update messages in order to keep the error in the cost function relatively small. `ProperPLACE-PM` also takes full advantage of the prioritized messages provided by the ProperCAD II library to guide the run-time system to select critical messages. Load balancing is achieved through inter-actor move suppression. These features are discussed in more detail in [1, 2].

3.2 Results and Analysis

Results are presented for a set of benchmark circuits (Table 3.1) in Tables 3.2 and 3.3 for a Sun SparcServer 1000 and Intel Paragon, respectively. W indicates the normalized wirelength cost of the resultant placement, and S is the speedup compared to that for a single processor. The algorithm provides moderate speedup with some loss of quality. The parallel placement algorithms based on parallel moves are hindered because they must limit the cumulative error

Table 3.1: Circuit Information
ISCAS, Physical Design Workshop 91, and Other Industry Benchmarks

Circuit	Number of cells	Number of nets	Initial wirelength
fract	125	163	12839
s298	133	138	126984
s420	212	233	274056
primary1	760	1172	1202241
struct	1888	1920	388707
industry1	2271	2583	5301571
primary2	3014	3817	7292946
biomed	6417	5742	18844676
circuit	5812	7574	83198647
industry2	12142	13419	90099851
avq.large	25114	25384	129120230

Table 3.2: `ProperPLACE-PM` Results (Sun SparcServer 1000E)

	1 PE		4 PE		8 PE	
Circuit	W	S	W	S	W	S
primary1	1.00	1.00	1.03	2.35	1.07	3.06
industry1	1.00	1.00	1.05	1.88	1.04	1.99
primary2	1.00	1.00	1.02	2.22	1.09	3.05
biomed	1.00	1.00	1.01	2.49	1.18	3.22
industry2	1.00	1.00	1.18	2.22	1.33	2.33
avq.large	1.00	1.00	1.44	1.84	1.14	2.90

Table 3.3: `ProperPLACE-PM` Results (Intel Paragon)

	1 PE		4 PE		8 PE	
Circuit	W	S	W	S	W	S
primary1	1.00	1.00	1.04	2.75	1.04	3.95
industry1	1.00	1.00	0.98	2.70	0.99	3.95
primary2	1.00	1.00	1.09	2.68	1.10	3.85
biomed	1.00	1.00	1.12	2.62	1.21	3.86
industry2	1.00	1.00	1.07	2.49	1.05	3.44

effects on the global cell database. This can cause a significant degradation in quality as the number of processors increases.

Also, because of the necessary global updates, the speedups will likely not scale linearly with the number of processors. Since every processor must perform these updates, this becomes essentially a serial bottleneck. In fact, because updates are done only for accepted moves, speedups are intimately related to the acceptance rate. Consider if t_{me} is the time to propose and evaluate a move, and t_d and t_u are the times to decide and update a move, respectively, the total run time of a serial execution is

$$t_{ser} = I(t_{me} + t_d + t_u) \tag{3.1}$$

where I is the total number of moves attempted. In a parallel moves implementation the run time is as follows:

$$t_{par} = \frac{I}{N}(t_{me} + t_d + t_u + \alpha(N-1)t_{mu}) \tag{3.2}$$

where N is the number of processors, α is the acceptance rate, and t_{mu} is the time required to perform another processor's accepted move. Note because of the way `TimberWolfSC` does incremental wirelength cost estimates, t_{mu} is not equivalent to t_u. The speedup can be characterized as follows:

$$S = \frac{t_{ser}}{t_{par}} = \frac{I(t_{me} + t_d + t_u)}{\frac{I}{N}(t_{me} + t_d + t_u + \alpha(N-1)t_{mu})} \tag{3.3}$$

Measurements have shown that t_{mu} is close to $t_{me} + t_u$, and $t_{me} + t_u \gg t_d$.[1] Using these assumptions, S simplifies to $\frac{N}{1+\alpha(N-1)}$. Essentially, this limits the speedup to $\frac{1}{\alpha}$ regardless of the number of processors.

Recent work by Sun and Sechen [44] used a parallel moves approach that addresses this problem with less frequent updates. The updates are made at specific intervals instead of for every accepted move. In spite of the fewer updates, there is no reduction in quality, due possibly, in part, to using a different serial algorithm at the core of their parallel implementation. `Tim-`

[1]Measurement details are shown in Table 5.2 in Chapter 5.

`berWolfSC 7.0` [50] uses clustering of cells and has removed cell overlap, which appears to make the update frequency less critical.

3.3 Summary

In this chapter, the parallel moves approach to parallel cell placement approach was described. Parallel moves offers limited speedup potential with severe quality degradation problems. The main problem is that update frequency limits the speedup regardless of the number of processors available. In addition, this algorithm is not memory scalable, since the circuit must be replicated on all processors. These problems are addressed with other parallel algorithms described in Chapters 4 and 6.

CHAPTER 4

MULTIPLE MARKOV CHAINS APPROACH

In this chapter we introduce the concept of multiple Markov chains, first presented as parallel clustered statistical cooling by Aarts et al. [51–53]. It was further refined by Lee and Lee, who introduced an asynchronous approach to this methodology and, in particular, applied the algorithm to the graph partitioning problem for shared memory multiprocessors [54, 55]. The algorithm can be understood if the sequential simulated annealing procedure is considered as a search path where moves are proposed and either accepted or rejected depending on particular cost evaluations and also a starting random seed. Each search path is essentially a Markov chain, and parallelization is accomplished by initiating different chains (using different seeds) on each processor. Each chain then explores the entire search space by independently performing the annealing perturbation, evaluation and decision steps. After each processor has completed the annealing schedule, the solutions are compared and the best is selected. Rose et al. used a similar approach with the min-cut algorithm in the high-temperature region of the simulated annealing schedule [46]. Note that unlike the parallel moves approaches, each chain is allowed to perform moves on the entire set of cells and not just a subset.

Of course, there is no speedup in this approach since each processor is individually performing the same amount of work as the sequential algorithm. There is, however, the opportunity for better quality solutions. To achieve speedup, we must reduce the number of moves evaluated in each chain by a factor of $\frac{1}{N}$ where N is the number of processors. Since the number of moves determines the run time of the program, a reduction by a factor of $\frac{1}{N}$ will cause a speedup of N. Of course, such a reduction alone is not appropriate since the quality will decrease accordingly. To take advantage of the use of multiple processors, some means of interaction between the various chains are necessary.

4.1 Synchronous Multiple Markov Chains

One possible interaction scheme, called *synchronous MMC with periodic exchange* by Lee and Lee, is to stop annealing and compare solutions at fixed intervals. This method allows each Markov chain to update its local database with the best solution, and then continue. This exchange point serves as the end of a *segment* of computation and behaves as a barrier synchronization point. According to the algorithm proposed by Aarts et al., the exchange point occurs after every move. At the barrier, various application specific metrics can then be used to determine the best solution.

In an actor framework, each chain or search path is represented by a separate actor or thread of control. Since the actor model can not assume a shared memory architecture, solution updates must be done with message sends. The barrier at the end of each segment is implicitly achieved through the use of these messages, as shown in Figure 4.1. When an actor has reached the end of its segment, it propagates a solution metric up to a *master* actor through a reduction tree. This metric is only a cost measurement of the solution and is not the entire global state. The *master* thread will determine the best solution, and then directs the actor with the best solution to broadcast its state to all other actors. In the example in Figure 4.1, actor 3 is determined to have the best solution. The barrier could be implemented in a single phase manner if each actor propagates its entire state to the *master* rather than just the cost metric. The *master* could then broadcast the state itself rather than request the winning actor to broadcast it. However, because of the size of the state in cell placement, the two-phase method is more efficient. The implementation proposed by Lee and Lee can use a single phase by transferring the entire state through shared memory.

When applied to the `TimberWolfSC` placement tool, a natural point for solution exchanges is the end of each `TimberWolfSC` iteration. Figure 4.2 shows a summary of the algorithm for synchronous multiple Markov chains. As mentioned above, the entire cell database state is quite

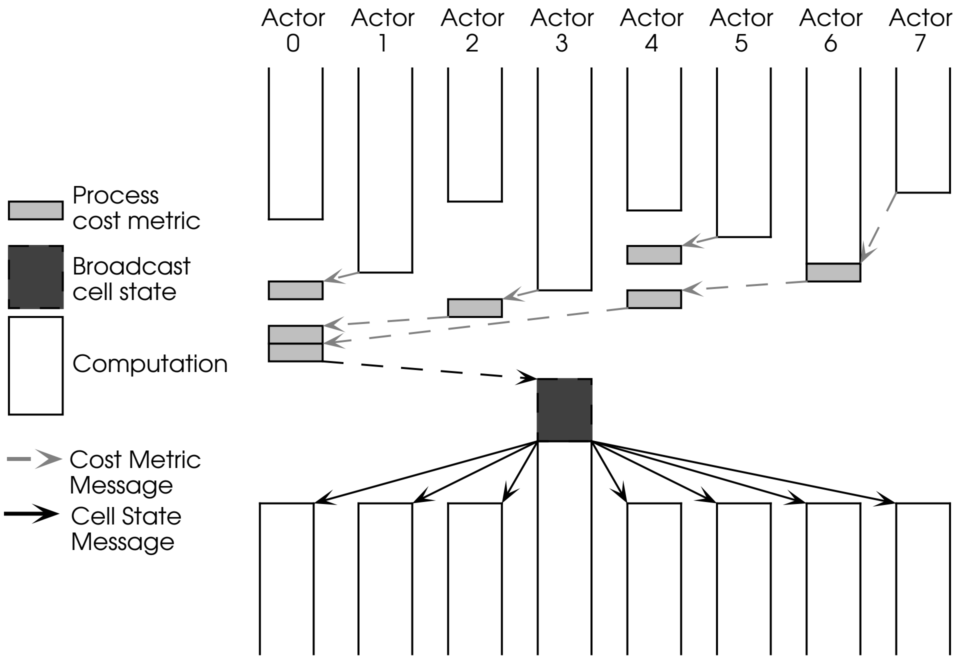


Figure 4.1: Message flow in actor based synchronous MMC.

large. Table 4.1 shows for a sampling of circuits the cell database size needed for message transfer.

4.2 Asynchronous Multiple Markov Chains

From examining Figure 4.1, it is obvious that barriers can be costly operations; an asynchronous approach is preferred. Figure 4.3 shows a pseudocode for such an implementation. Note that very few modifications have to be made to support the asynchronous method. In this approach, the *master* actor does not perform any computation. Instead, it serves as a location for the best available solution at any particular time. When an actor has completed an iteration, it sends its solution metric to the *master* actor, and requests the best solution available. The master thread on receipt of this request will determine if the received solution is better than the local “best” solution. If it is better, the *master* will ask the requestor to send its state back, by calling the SendCellState() method. Remember that the requestor had only sent the metric initially. The requestor will then send its local state back to the master and continue with the

```
MMCSync::DoIteration( cell_state )
{
   local_state = cell_state;
   I = I + 1; // update TimberWolfSC iteration number
   while ( N <= Nmax ) { // number of moves attempted per iteration
      N = N + 1;
      update temperature if necessary
      propose a move and evaluate new cost
      if move is accepted, update local_state
   }
   // send my solution metric and my actor name to master
   master->SendSolutionMetric ( local_state.metric, myName );
}

MMCSync::SendCellState ()
{
   // the master actor is asking us to broadcast our
   // state, since we have the current best state
   broadcast->DoIteration ( local_state );
}

MMCSyncMaster::SendSolutionMetric ( metric, name )
{
   // if the incoming metric is better than the
   // the current best state, update our best values
   // when all metrics have been received and thus
   // synchronized, ask the winner to broadcast its
   // cell state.
   if (metric better than current best_metric) {
      best_metric = metric;
      best_actor = name;
   }
   if (all metrics received)
      best_actor->SendCellState ();
}
```

Figure 4.2: Outline of the multiple Markov chains synchronous actor interface.

Table 4.1: Circuit Size Information
ISCAS and Physical Design Workshop 91 Benchmarks

Circuit	Cell state size (bytes)
fract	9636
s298	8884
s420	13844
primary1	65484
struct	119636
industry1	219220
primary2	1901814

next iteration with its own local state. If the *master* has determined that the received solution is worse than the best solution, the *master* will simply send the current best state to the requestor. At the cost of dedicating an extra processor for "master" usage, this asynchronous approach can eliminate much of the idle time that was present earlier.

Graphically, this algorithm is illustrated in Figure 4.4. For example, Actor 2 has finished its segment and sends its solution metric to the *master* that determines that its solution is the best and sends a message back requesting Actor 2's local state. While waiting for Actor 2's state, the *master* has in the meantime received a solution metric from Actor 1. Since it hasn't yet received Actor 2's state, the *master* must compare Actor 1's metric with the previous state. It then determines that Actor 1 has an inferior solution, and thus sends the previous state back to Actor 1. Note, because of the asynchronous behavior, Actor 1 was not able to receive the best solution at that current point. This type of erroneous update is acceptable, since the actors can correct themselves at future iterations, and it also provides an opportunity to escape from local minima.

From Eq. (3.1), we can see that the parallel run time in an asynchronous MMC implementation becomes

$$t_{par} = \frac{I}{N}(t_{me} + t_d + t_u) + t_c \tag{4.1}$$

where t_c is the communication time. Since the time to send the solution metric is small, t_c is essentially the time to send the state, which is related to the size of the state. If t_c is small, it is

```
MMCAsync::DoIteration( cell_state )
{
   local_state = cell_state;
   I = I + 1; // update TimberWolfSC iteration number
   while ( N <= Nmax ) { // number of moves attempted per iteration
      N = N + 1;
      update temperature if necessary
      propose a move and evaluate new cost
      if move is accepted, update local_state
   }
   // send my solution metric and my actor name to the master actor
   master->SendSolutionMetric ( local_state.metric, myName );
}

MMCAsync::SendCellState ()
{
   // the master actor is asking us for our state
   // since we have the current best state
   master->ReceiveCellState ( local_state );
   DoIteration ( local_state );
}

MMCAsyncMaster::SendSolutionMetric ( metric, name )
{
   // if the incoming metric is better than the current
   // best state, get the state from that actor,
   // otherwise, just start the next iteration
   // with the current best state on the actor.
   if (metric better than local_state.metric)
      name->SendCellState ();
   else
      name->DoIteration ( local_state );
}

MMCAsyncMaster::ReceiveCellState ( cell_state )
{
   local_state = cell_sate; // update local best state
}
```

Figure 4.3: Outline of the multiple Markov chains asynchronous actor interface.

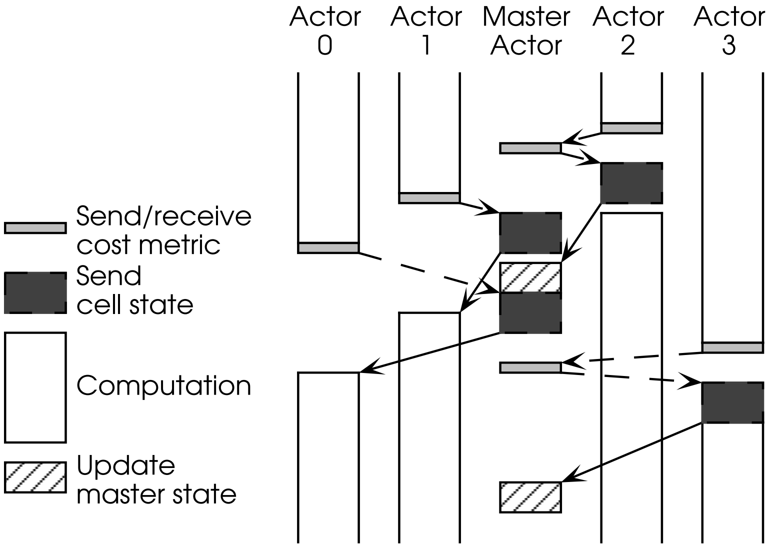


Figure 4.4: Message flow in actor based asynchronous MMC.

obvious that the speedup for asynchronous MMC is perfectly linear, i.e., $S = N$. However, t_c is not necessarily small, and we must determine its effect. Comparing Tables 4.1 and 3.1 yields 80 as an approximate ratio of state size to number of cells. Since the state is communicated at every one of `TimberWolfSC`'s 160 iterations, $T_c = 12800Ct_b$, where C is the number of cells and t_b is the per byte communication cost. Also, for `TimberWolfSC`, I, the number of total moves, is set to $4000C(\frac{C}{500})^{\frac{1}{3}}$. Thus the speedup can be expressed as

$$S = \frac{5(\frac{C}{500})^{\frac{1}{3}}(t_{me} + t_d + t_u)}{\frac{5}{N}(\frac{C}{500})^{\frac{1}{3}}(t_{me} + t_d + t_u) + 16t_b} \tag{4.2}$$

On an Intel Paragon, $t_{me} + t_d + t_u$ is on the order of 50 to 100 μs, while t_b is only 0.018 μs. Thus it can be seen that t_c is negligible, and the expected speedup should approach N.

4.3 Experimental Results

We examined the effectiveness of the asynchronous MMC algorithm on the Sun SparcServer 1000E as well as for the Intel Paragon as shown in Tables 4.2 and 4.3. The implementation of the algorithm is called `ProperPLACE-MMC`. We start at two processors because of the need for a

Table 4.2: `ProperPLACE-MMC` Results (Sun SparcServer 1000E)

	2 procs		4 procs		8 procs	
Circuit	W	S	W	S	W	S
s298	1.00	1.00	1.04	2.74	1.07	6.51
s420	1.00	1.00	1.00	2.57	1.06	7.22
primary1	1.00	1.00	1.00	2.78	1.06	4.80
industry1	1.00	1.00	1.02	2.79	1.04	7.06
primary2	1.00	1.00	1.07	2.64	1.05	6.44
circuit	1.00	1.00	1.01	2.92	1.10	6.88
industry2	1.00	1.00	1.02	3.17	1.04	7.15

master processor. The quality of the solutions shows no degradation as the number of processors increases - in fact, they sometimes show improvements because of the periodic exchange of solutions. The algorithm scales much better than the parallel moves approach. As expected, communication time is not significant since the number of messages sent is few. Note that the speedup and quality are much better than can be achieved with a pure parallel moves strategy as presented in the previous chapter.

Sun and Sechen have used a modified parallel moves approach to achieve similar speedup results [44]. Though they use parallel moves and partitioned circuits, at each iteration, as in multiple Markov chains, solutions are exchanged among processors. They do not have to process frequent update messages.

If improving the run time is not the goal, `ProperPLACE-MMC` can be used in another mode to provide better solutions for the same run time. Table 4.4 compares `TimberWolfSC` on one processor with `ProperPLACE-MMC` on four processors. The solution quality is improved by an average of 9% at a cost of only 3% in run time. To achieve that type of quality improvement in `TimberWolfSC` would require significantly more computation time.

4.4 Summary

In this chapter, we discussed using multiple Markov chains for parallelizing simulated annealing based cell placement. Conceptually a very simple algorithm, MMC, is a very promising

Table 4.3: `ProperPLACE-MMC` Results (Intel Paragon)

	2 procs		4 procs		8 procs	
Circuit	W	S	W	S	W	S
s298	1.00	1.00	1.02	2.28	1.03	2.80
s420	1.00	1.00	1.01	2.84	1.09	5.91
primary1	1.00	1.00	1.01	2.78	1.06	5.67
industry1	1.00	1.00	1.01	2.99	1.03	6.76
primary2	1.00	1.00	1.02	2.93	1.09	6.75
circuit	1.00	1.00	1.07	2.97	1.08	6.90
industry2	1.00	1.00	0.95	3.13	1.09	6.79

Table 4.4: `ProperPLACE-MMC` Quality Improvement (Sun SparcServer 1000E)

	`TimberWolfSC` 1 procs		`ProperPLACE-MMC` 4 procs	
Circuit	W	S	W	S
biomed	1.00	1.00	0.82	0.95
circuit	1.00	1.00	0.91	0.98
industry2	1.00	1.00	0.97	0.94
avq.large	1.00	1.00	0.94	1.02

approach to parallelizing the cell placement problem. It offers excellent speedups with modest degradations in quality. Multiple Markov chains can also be used to improve the quality of the resultant placement at very little extra computation cost. The main disadvantage of this algorithm is that it is not memory scalable, since the circuit must be replicated on all processors. In addition, as the number of processors increases, the placement quality degrades, but not as severely as with parallel moves.

CHAPTER 5

SPECULATIVE COMPUTATION APPROACH

Another approach recently suggested for generalized parallel simulated annealing is speculative computation [56]. Witte et al. applied this approach to the task assignment problem and found speedups approaching $log_2\ P$, where P is the number of processors. In this chapter, we apply the concept of speculative computation to cell placement and determine the applicability of such an approach. We first give a brief description of the algorithm.

5.1 Generalized Speculative Computation

A sequential simulated annealing schedule is simply a series of move proposals intended to reduce some cost function as related to the particular problem. Each move consists of three parts - the proposal or perturbation, evaluation, and decision. Only after these three parts are completed is the next move started. Since the decision made by the next move is dependent on the current state as determined by prior moves, simulated annealing is almost inherently serial in nature. However, note that if none of the moves are accepted, the state does not change and the opportunity exists for parallelizing moves. This is the approach taken by [41] and was recently suggested again by [57]. Parallelization through speculative computation is similar except that it also attempts to accelerate accepted moves as well.

Consider the decision tree of moves in Figure 5.1(a). The top node represents a move attempted in a simulated annealing process. There are two possible decisions as a result of this move - acceptance or rejection. Speculative computation will assign two different processors

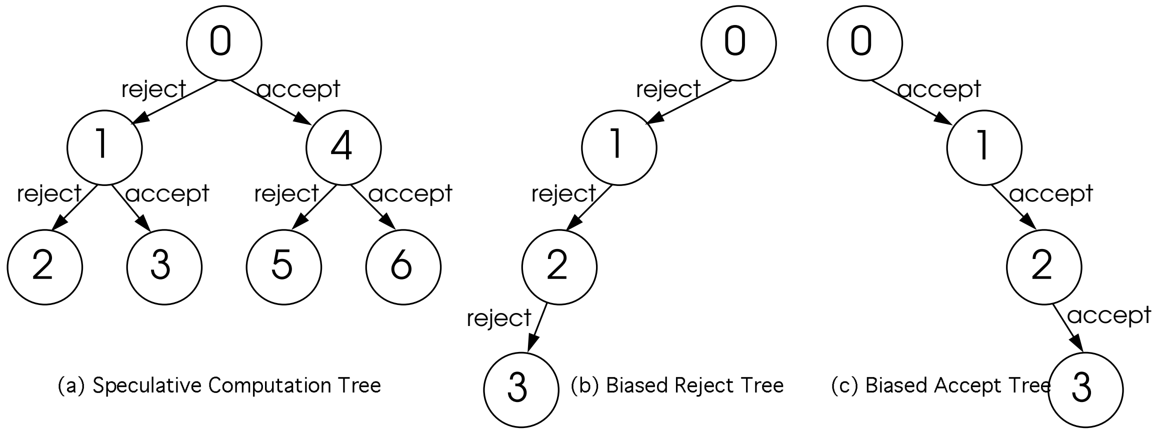


Figure 5.1: Speculative computation trees.

to speculatively work on the two possibilities before the parent move has completed. The rejection child can start at the same time as the parent, since it will assume that the state has not changed. After the parent has completed the move proposal, it can then relay the new state to the accept processor. The communication behavior is illustrated on the timeline in Figure 5.2. In the figure, t_m is the time to propose and perform a move, t_e is the evaluation time, and t_d is the decision time. In this example, the decision tree has gone from node 0 (accept) to node 4 (reject) to node 5. Comparing the parallel time with the sequential time on the same graph shows the potential speedups of speculative computation.

As the acceptance characteristics of the procedure vary, the shape and bias of the tree can also change. For example, if the acceptance rate is high, it would make sense to generate a linear tree of only acceptance nodes, and on the other hand, a very low acceptance rate would imply the creation of only rejection nodes (Figures 5.1(b) and (c)). This latter mode is essentially the mode in which the algorithms proposed by [41] and [57] operate. Witte et al. constructed an analytical model that placed the speedup approximately at $\frac{P - log\ P}{ln\ P - ln(log\ P)}$. In the extreme, it is also limited by the inverse of the annealing acceptance rate in the low temperature region and the inverse of the rejection rate in the high temperature region. This latter limit is essentially the serial subset length called $|E_{ss}|$ by Kravitz [41]. It was ignored by Witte et al. but it is necessary since it accounts for the maximum expected length of each linear tree used at the extreme temperature regions.

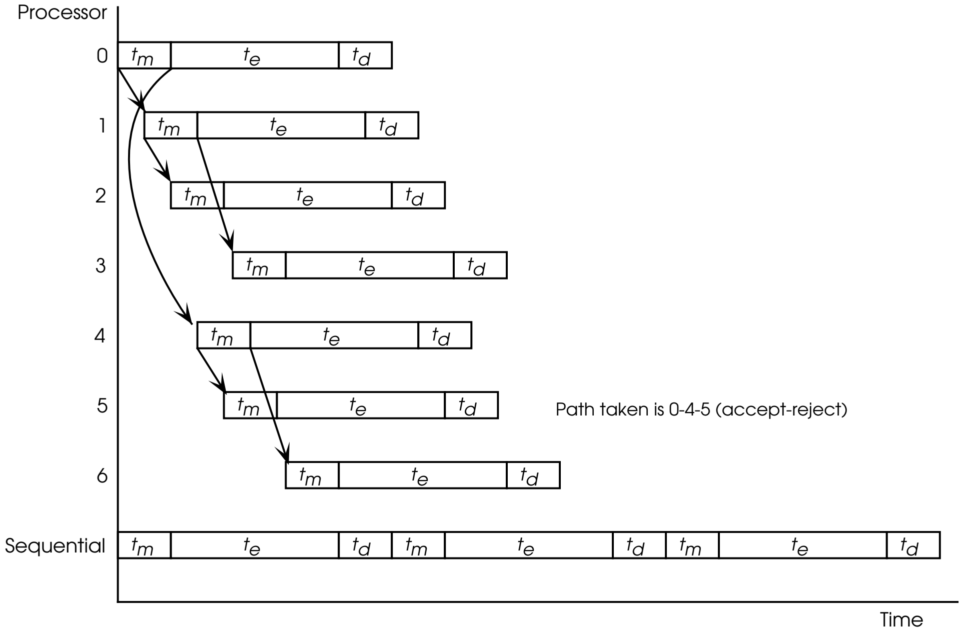


Figure 5.2: Seven-processor speculative computation timeline.

5.2 Speculative Computation for Placement

Since speculative computation seems to be a promising avenue to achieve at least some speedup in the high temperature region, we decided to investigate the applicability of such an approach to the cell placement problem. The problem fits naturally into an actor-based framework, in that each speculated move can be represented by an actor. One of the major changes we made to the algorithm was to add some asynchronous behavior by removing the need for a centralized root processor that was required to start off each tree. Eliminating this synchronization point allows multiple speculative trees to be active at once. Indexing was used to properly order the execution of trees.

Another major modification made to Witte's algorithm was to transfer only the move proposal to the accept child rather than the entire state after the move. That is, if the root node were to propose moving a cell to a new location, it would convey the cell number and new location to the accept child, and the child would be responsible for duplicating the move. As with multiple Markov chains, this decision was made because of the potentially large size of the cell state. Once a speculative move has been determined to be false, the actor responsible for the move

Table 5.1: `ProperPLACE-SC` Results (Sun 4/690MP)

	1 procs		2 procs		4 procs	
Circuit	W	S	W	S	W	S
fract	1.00	1.00	1.00	0.30	1.00	0.22
s298	1.00	1.00	1.00	0.25	1.00	0.18
primary1	1.00	1.00	1.00	0.23	1.00	0.20

must then abort its move as well as all its parent's moves. It must then update its cell database with the moves from the correct path.

An outline of the algorithm is given in the code fragment in Figure 5.3. For simplicity, the details of the asynchronous algorithm are not shown. The `MoveActor` constructor creates the child actors. If the actor is the root of the tree, it continues and evaluates the move. If it is accepted, the accept child is told to continue while the reject child is aborted, and vice-versa if the move is rejected. Once the state has been updated, a new tree can be started.

5.3 Experimental Results and Analysis

After the modifications were made, the implementation, called `ProperPLACE-SC`, was run on a variety of circuits and machines as shown in Table 5.1. As can be seen from the table, the wirelengths are identical, as expected. The speedups, however, are disappointingly poor, as indicated by the drastic slowdowns. The primary reason for this behavior is the faultiness of Witte's model when applied to cell placement - particularly `TimberWolfSC`. Witte makes two main assumptions - the time to perform and propose a move is small compared to the evaluation of the move, and second, the outcome of that move or the resultant state is easily communicable.

Consider the following analysis. The serial time for `TimberWolfSC` as stated before is

$$t_{ser} = I(t_{me} + t_d + t_u) \tag{5.1}$$

```
main()
{
// ... initialize .. and then start off the tree
MoveActor::New::Continuation () ( 0 );
}

MoveActor::MoveActor( MoveList &moves )
{
   doParentMoves( moves )
   rejectChild = ActorName<MoveActor>::newName();
   New::Continuation ( rejectName ) ( moves );

   // propose new move and add to moves list ...

   acceptChild = ActorName<MoveActor>::newName();
   New::Continuation ( acceptName ) ( moves );
   if ( isRoot() ) Continue();
}

void MoveActor::Continue()
{
   if ( accept( newmove ) ) {
      Continue::Continuation ( acceptChild ) ();
      Abort::Continuation ( rejectChild ) ();
      update_state();
      updateAllOtherThreads();
      if ( isRoot() && annealing termination not reached )
 MoveActor::New::Continuation () ( 0 ); // start off new tree
   } else {
      Continue::Continuation ( rejectChild ) ();
      Abort::Continuation ( acceptChild ) ();
   }
}

void MoveActor::Abort()
{
   undoParentMoves();
   Abort::Continuation ( acceptChild ) ();
   Abort::Continuation ( rejectChild ) ();
}
```

Figure 5.3: Outline of the `ProperPLACE-SC` algorithm.

Table 5.2: Task Time Measurements

Task	time (μs)
Proposal and evaluation	
cell exchange	87.8
cell move	43.2
Accept	5.4
Update after accept	
cell exchange	30.6
cell move	14.8
Speculative move	
cell exchange	116.2
cell move	48.3

In terms of Witte's model, $t_{me} = t_m + t_e$, and t_u is not incorporated into Witte's model. In a parallel implementation, the execution time is

$$t_{par} = \frac{I}{L}((L-1)(t_c + \alpha(t_m + t_{sm})) + t_d + t_u) \tag{5.2}$$

where L is the expected length of the tree, t_c is the time required for communicating a proposal, and t_{sm} is the time required to perform a parent's move. The speedup is

$$S = \frac{L(t_{me} + t_d + t_u)}{(L-1)(t_c + \alpha(t_m + t_{sm})) + t_{me} + t_d + t_u} \tag{5.3}$$

The term $(L-1)(t_c + \alpha(t_m + t_{sm}))$ is the cost due to performing the parent's updates. If this term is small, then the speedup is simply L, which in the extreme temperature regions is N, and at worst is $logN$. However, our measurements have shown that this is not the case. In `TimberWolfSC`, the move and evaluation steps are combined into one for efficiency, since the process of changing data structures to reflect the move also easily allows the incremental calculation of the cost. The time to decide whether to accept is therefore minimal. After the move has been accepted, the appropriate data structures must then be updated. We were able to measure the times of various moves to gauge their impact on the algorithm (Table 5.2). Note that the proposal and evaluation time is the largest portion of a full move.

Since our implementation requires the child node to redo the parent's moves, we also measured the speculative move time(t_{sm}), that is, the time that a child node requires to perform its parent's move. This time is essentially the proposal and evaluation time plus some time to update associated data structures. Note that even though the accept stage is not performed, the times are comparable with a non-speculative move. With such times dominating the move time, it is clear why cell placement parallelism through speculative computation is inefficient. For a two-processor system, we found that 48.9% of an actor's time was spent waiting for the other actor to finish. The primary reason for this idle time, which drastically hinders parallelism, is that, because of the expense of performing speculative moves, parent nodes must wait for the child to perform its moves as well as its parent's moves. For a two-level tree, the child is performing essentially twice as much work. The amount of idle time will increase with the number of levels in the tree, though our asynchronous implementation attempts to address that by starting future trees ahead of time.

Another problem that affects parallelization of `TimberWolfSC` is its relatively high acceptance rate. The steady region attempts to maintain an acceptance rate of 44%. Naturally, this leads to poor speculative computation behavior, since a lot of work must be duplicated. On a two-processor system, we found that 23.4% of the actor time was spent either performing parent's moves, aborting moves, or updating the other actor's moves. In light of these problems and those mentioned above, it is clear that speculative computation is not a feasible approach to parallelization of cell placement.

5.4 Summary

In this chapter, we have described the speculative computation approach to parallel simulated annealing and applied it to the cell placement problem. The characteristics of cell placement make speculative computation an inadequate means of parallelization. Though it is excellent in terms of retaining solution quality, there is no opportunity for parallelism using this method.

CHAPTER 6

CIRCUIT PARTITIONED APPROACH

As circuit sizes increase, it becomes more and more infeasible to adequately address VLSI CAD problems on a single processor as a result of inordinate memory requirements. This is especially true in the area of cell placement where design sizes are approaching 100,000 cells and more. In the past, designers have partitioned the circuit into more manageable subcircuits and then suffered a loss of quality on the placement of the subdivided circuit. In this chapter, a method is presented that takes advantage of the large memory resources spread across multiple processors in a parallel processing machine and still achieves faster placement times without loss of quality. The implementation of this circuit partitioned parallel algorithm developed using ProperCAD II is called `ProperPLACE-PART`. We also present an implementation of the algorithm using MPI.

6.1 An Object-Oriented Cell Placement Algorithm

As with the work described in earlier chapters, `ProperPLACE-PART` is based on `TimberWolfSC` 6.0. The current version of `TimberWolfSC`, 7.0 could not be used because the code was not available. The concepts of parallelization, though, will hold for the newer version as well. To understand the parallelization procedure, some further explanation of the data structures used in `TimberWolfSC` 6.0 is necessary. The circuit information is described primarily with the use of three arrays - the list of cells, the list of nets, and finally an array describing row information. Each cell data structure contains positional information as well a linked list of pins that belong

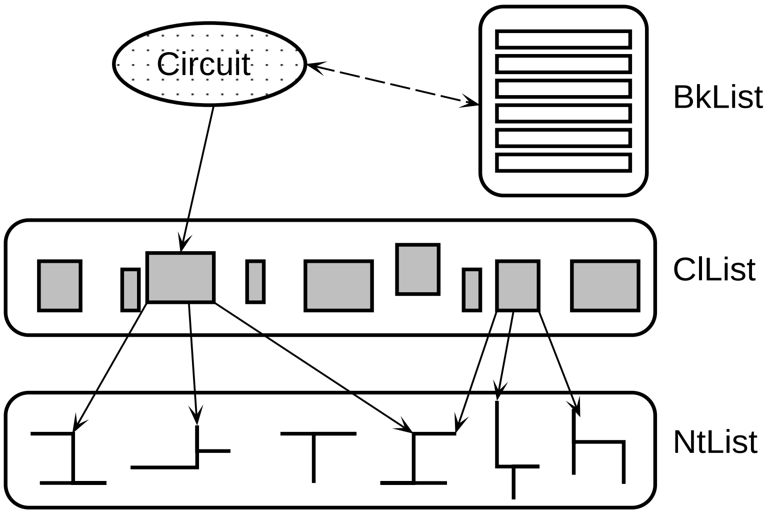


Figure 6.1: Relationships between objects.

to the cell. Likewise, each net data structure has bounding box information as well as a linked list of pins that belong to the net. The pin data structures are shared by both the cell and net linked lists.

In an object-oriented framework such as ProperCAD II, these data structures must be converted to C++ objects. The cell and net structures are converted into `Cell` and `Net` C++ classes, respectively. Of particular interest are the cell and net arrays, which now become the `ClList` and `NtList` C++ objects. A `BkList` object is used to keep track of row information as well as the bins used for overlap penalty calculation. All the objects are integrated together in an object called `Circuit` which manages the core annealing algorithm. In addition, the `Circuit` object also contains many of the global parameters and flags that are relevant during annealing. In this new formulation, the `Circuit` object makes requests to the `ClList` object to pick a `Cell` to move. A new location and possible cell to exchange with are chosen by making a request to the `BkList` data structure. The `Circuit` object then asks the chosen `Cell` to evaluate the net cost of the move. This is done by making requests of the `NtList` on the appropriate `Nets`. Similar mechanisms are taken to calculate the row and overlap penalties. The relationships between these objects are shown graphically in Figure 6.1.

The core algorithm is shown in the `Circuit::anneal()` code fragment in Figure 6.2. The `Circuit` object asks the `ClList` object through the `pickACell()` method to pick a `Cell` to move. A new location for the cell is chosen by making a request to the `BkList` data structure. The `Circuit` object then asks the `ClList` to evaluate the delta cost of the move of the chosen `Cell`. If the move is accepted, then the data structures are updated through `updateCell()` which updates each net attached to `Cell`, through the `updatePin()` method.

6.2 Parallelism Through Inheritance

To remain compatible with the arrays in the sequential code, we need arrays with distributed semantics. This is accomplished easily with the use of aggregates [19]. An aggregate in its simplest form is a collection of actors distributed across the available processors in the system. In the case of an array, an aggregate acts as a single name for the array that has been decomposed across many processors.

For example, we can simply create a new distributed class, `CellAggr`, that is derived from the `ClList` class as well as the aggregate class, as below.

```
class CellAggr : public ClList, public Aggregate {
    Cell &pickACell();
    ...
};
```

By doing so, the `CellAggr` class has representative actors on each processor that are responsible for the cells allocated to that particular processor. Thus, a request made to access a particular cell can be made to the local representative actor which will then forward it on to the actor on the thread where the cell is actually present. A similar transformation is done for `NtList`. In total, there are aggregates for the cells (`CellAggr`), nets (`NetAggr`), and rows (`BlockAggr`), as well as the `CircuitAggr` that is derived from `Circuit`. The dashed line in Figure 6.3 indicates the separation between two threads. Note that the individual cells and nets are not replicated.

How does this data distribution affect the transformation of the serial program to a parallel version? The main operations in the `Circuit::anneal()` method are picking a cell to move, evaluating the move, deciding whether to accept, and updating the cell structure. The evaluation

```
class Circuit {
    ClList &carray;
    NtList &netarray;
    BkList &barray;
    ...
};

Circuit::anneal()
{
while ( terminationNotReached() )
    tryCellMove();
finishUp();
}

Circuit::tryCellMove()
{
Cell &cell = carray.pickACell();
newLocation = barray.newLocation();
deltaCost = carray.evaluateMove( cell, newLocation );
if ( acceptMove( deltaCost ) )
    carray.updateCell( cell, newLocation );

}

ClList::updateCell( Cell &cell, Position &newLocation )
{
pinIterator next(cell);
Pin *ptr;
while ( ptr = next() ) {
    ptr->updateLocation( newLocation );
    netarray.updatePin( *ptr );
}
}
```

Figure 6.2: Core code for serial algorithm.

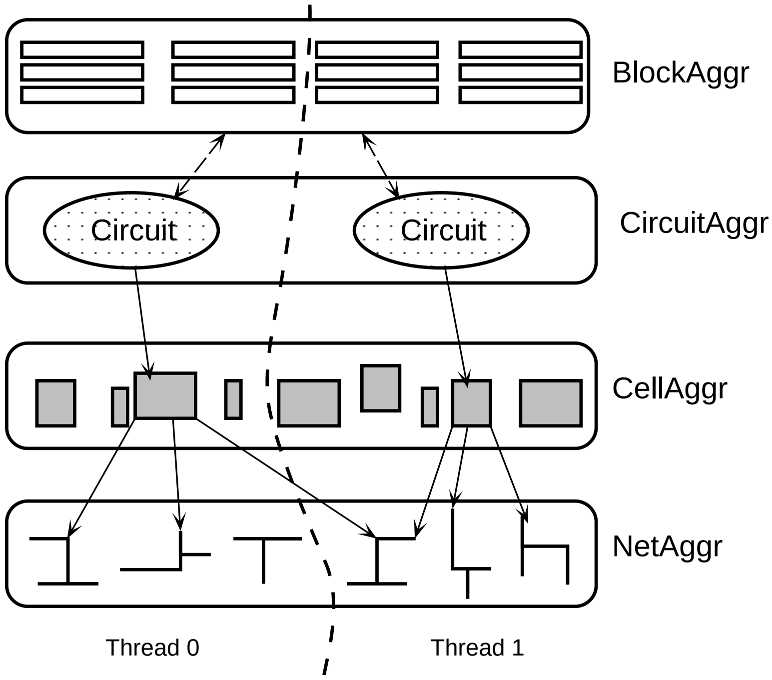


Figure 6.3: Relationships between distributed objects.

and decision phases are independent of location of the cell. Picking a cell should be made only from the local pool of cells. To accomplish this, we make `ClList::pickACell()` a virtual base function and rewrite the implementation as shown in Figure 6.4.

Likewise, although updating cells is a local operation, the updating of nets may not be local because the adjacent nets are distributed. Therefore, `NtList::updatePin()` is now made virtual and rewritten as in Figure 6.4. We use the continuation invocation mechanism to invoke a remote `NetAggr::updatePin()`. The invocation of the remote method is asynchronous, so the update is in effect a "lazy" update.

The presence of the updates causes an interesting problem. Since actor methods are run to completion without preemption, if `Circuit::anneal()` is left as it is, the while loop will run to completion before any update messages can be serviced by the run-time system. In order to give these messages an opportunity to arrive, the while loop must be transformed into message-driven code. As seen in Figure 6.5, through the use of virtual functions again, the `anneal()` method is modified to try only one move, and then re-enable itself by executing an asynchronous continuation to itself. Similar transformations have been discussed in [58].

```
Cell &CellAggr::pickACell()
{
Cell cell;
do {
    cell = ClList::pickACell();
} while ( !isLocalCell( cell ) );
}

class NetAggr : public NtList, public Aggregate {
    void updatePin( Pin& );
    class updatePin : public ActorMethod<Pin> {};
};

void NetAggr::updatePin( Pin &p, Position &location )
{
if ( isLocalNet( p ) {
    NtList::updatePin( p, location );
} else {
    updatePin::Continuation cont ( locationOf( p ) );
    cont( p, location );
}
}
```

Figure 6.4: Code for `NetAggr` and `CellAggr`.

```
class CircuitAggr : public Circuit, public Aggregate {
    void anneal();
    class anneal : public ActorMethodVoid {};
};

void CircuitAggr::anneal()
{
tryCellMove();
if ( terminationNotReached() ) {
    CircuitAggr::anneal::Continuation cont( *this );
    cont();
} else {
    finishUp();
}
}
```

Figure 6.5: Code for `CircuitAggr`.

6.2.1 Data distribution

The circuit is read in on a single thread, and as each cell and net are read in, the associated data structures are distributed to the other threads. The process of determining those cells which are assigned to a processor is done using a prepartitioning phase. There are two primary concerns in our partitioning. First, the load balance must be maintained, i.e., each partition should have roughly the same number of cells. Second, the number of nets cut should be minimized to decrease the interaction between partitions. The necessity of these requirements will become clear in the following section on parallelization. Ratio cut methods have long been used in the CAD community because of their effectiveness at reducing the cut size. However, these methods are inappropriate for our use because they do not provide well-balanced partitions. We have instead used a partitioning algorithm based on the Sanchis modification of the Fiduccia-Mattheyses algorithm [59, 60]. Graph partitioning methods such as recursive spectral bisection or METIS may alternatively be used [61, 62].

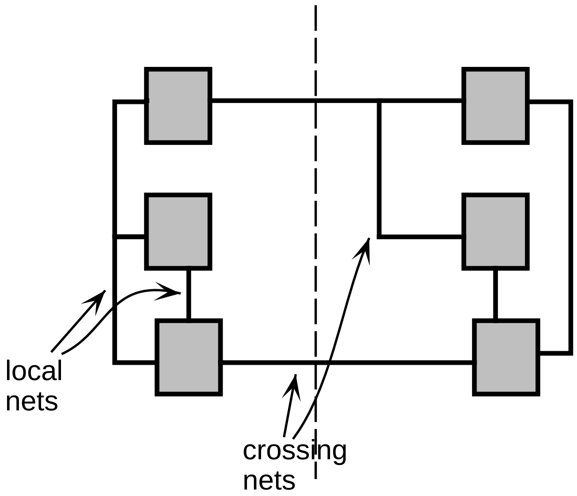


Figure 6.6: Crossing nets.

6.3 Parallel Algorithm

Once the circuit has been read in and distributed, the annealing procedure can begin. The `CircuitAggr` representative actor on each thread will then perform simulated annealing on its partition of the entire circuit, by making the same requests as the serial `Circuit` object. Now, these requests are handled by the `CellAggr` class, so that when a request is made to move a cell, the local `CellAggr` representative picks one from the pool of cells available on its thread. It does not supply cells from other threads. When a request is made to evaluate the cost of a cell move, the connected `Net` objects may be located on another thread. One option would be for the `NetAggr` to send a message to the appropriate representative to request a calculation of the cost. The overhead involved in the communication makes this prohibitive. Therefore, copies of these nets that span multiple threads are replicated locally. An example of these "crossing" nets is shown in Figure 6.6.

6.4 Error Control

As described above, each actor independently places its partition of cells without concern for the remainder of cells. Obviously, this can cause inaccuracy in the calculation of cost. As there are three components to the `TimberWolfSC` cost function, there are likewise three main cost errors - wirelength cost, overlap penalty cost, and row penalty cost.

6.4.1 Wirelength error

Though the effect of the wirelength error is decreased because of the partitioning to minimize net cut, at high temperatures with frequent movement of cells, it is clear that the error will be significant at the partition borders. The "border" in this context is the cutline and may bear no resemblance to the actual geographical placement border. In order to keep the threads up to date with respect to wirelength cost, at fixed intervals, updates of this border information are made in a two-stage process. In the first stage, each thread sends its pins that are on "crossing" nets to the owner of the net. Once all the foreign pins for a net have been received, the net information is distributed to all the threads that have a copy of that net.

6.4.2 Overlap penalty error

In addition to the wirelength error, even with a proper partitioning, overlap penalty error is still a serious problem. Without knowing the overlap penalties due to cells on other processors, each partition will tend to collapse to the center of the layout. Therefore, each thread must keep the penalties associated with the other cells but will assume that these "foreign" cells are fixed. The bin structure is shown in Figure 6.7. The positions of the fixed cells are obtained by performing cell position updates. Because of the cost of these updates, these are done very infrequently, at each `TimberWolfSC` iteration. To further reduce the overlap penalty error, when the cells from all the other threads have been received, each thread individually removes the overlaps by shifting cells appropriately.

6.4.3 Row penalty error

The pin and cell updates address the wirelength and overlap errors, but they do not adequately address the row penalty error. This error is particularly severe, because of a peculiar "ping-pong" effect. The row penalty is used to force the final placement to have equal length

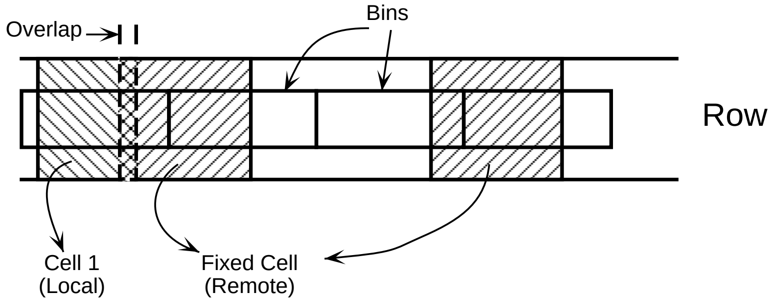


Figure 6.7: Fixed cells in bins.

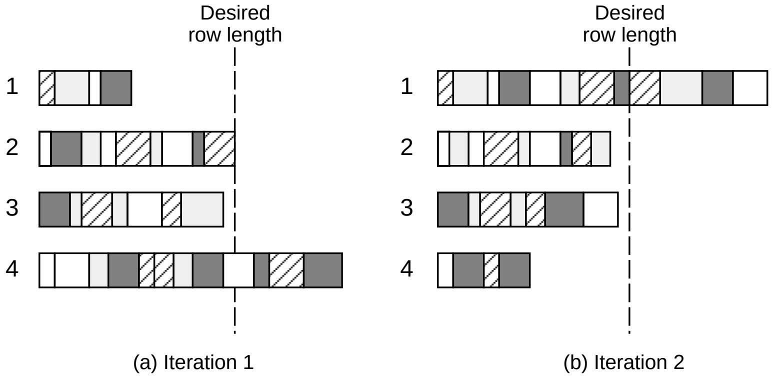


Figure 6.8: Effect of row penalty error.

rows, and in a parallel environment this can cause problems. Take for example, the situation in Figure 6.8(a). Row 1 is too short and row 4 is too long; thus, all the threads will try to move cells from row 4 to row 1. By the next iteration, row 1 has become too long and row 4 is now too short (Figure 6.8(b)). It is clear that this type of row shifting will continue without making any real progress in improving the placement.

We address this problem with three methods. The first is based on the observation that each thread is trying to satisfy a short row without realizing that other threads are doing the same thing. Therefore, we decrease the desired row length to take account of this. Depending on the range limiter, each thread is expected to contribute only part of the cells required to equalize a short row. For example, the placement from Figure 6.8(a) is redrawn in Figure 6.9 with a shorter desired row length.

The second method of addressing the row penalty error is to update the actual row sizes at distinct intervals. Since the amount of data sent is minimal, these updates can be done frequently

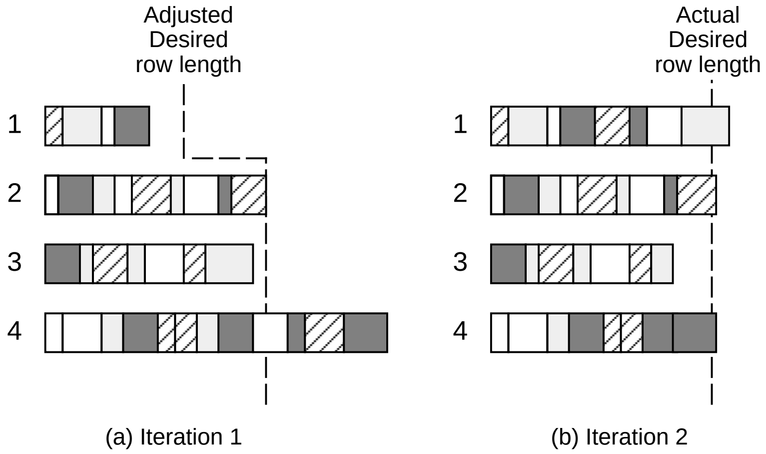


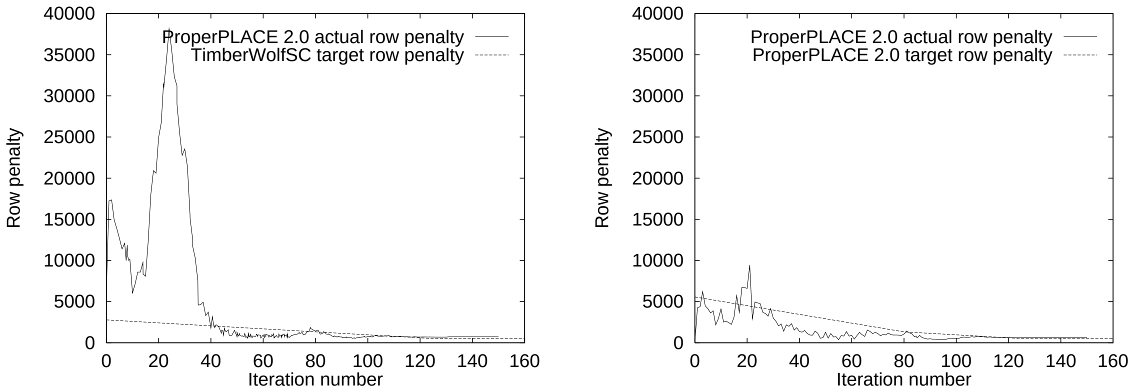
Figure 6.9: Example of desired row length adjustment.

without a loss of performance. These row updates are done using a lazy propagation update method.

The final heuristic to reduce the row penalty error takes advantage of the penalty feedback mechanism built into `TimberWolfSC`. Recall from Eq. (2.2) that the weight of the row penalty in the cost function is adjusted with a sophisticated feedback mechanism. Using experimental observations, the authors of `TimberWolfSC` have determined the optimal row penalty for each iteration, and then they adjusted the feedback coefficient so that the annealing schedule was close to this target penalty. Equation (6.1) shows the target penalty calculation. L_R is the total row length.

$$P_R^T = \begin{cases} .008L_R\left(6 - \frac{5I}{125}\right) & I \leq 125 \\ .008L_R & I > 125 \end{cases} \tag{6.1}$$

In a parallel setting, our experiments have shown that this target penalty is not sufficient. For example, Figure 6.10(a) shows the row penalty for the *primary2* circuit for four processors plotted against the iteration number. For comparison, the `TimberWolfSC` target penalty is plotted. As can be seen, the target penalty is off considerably in the earlier iterations. This deviation affects the λ coefficient considerably and thus the cost function in Eq. (2.2) is biased towards reducing the row penalty. For this reason, we use a modified target row penalty schedule as shown in Eq. (6.2). P is the number of processors being used, and B is the break point. We



(a) `TimberWolfSC` (b) adjusted for `ProperPLACE-PART`

Figure 6.10: Target row length penalty.

have found that this adjustment makes a tremendous positive effect on the quality of the circuit placement. Note from Figure 6.10(b) that with this modification, the row penalty error is much better controlled.

$$P_R^T = \begin{cases} .008L_R P\left(3P - \frac{3P-2.8}{80}I\right) & I \leq 80 \\ .008L_R\left(6 - \frac{5I}{125}\right) & 80 < I \leq 125 \\ .008L_R & I > 125 \end{cases} \qquad (6.2)$$

6.4.4 Dynamic error control

The error control mechanisms described above rely heavily on the use of updates. The cell updates are performed at fixed intervals; however, the absolute frequency of the row and pin updates was not specified. We use a mechanism called dynamic error control, where the frequency of the updates is adjusted according to the amount of error present. Several researchers have determined that bounding the accumulated error to a constant factor of the temperature will still guarantee convergence [34, 45].

6.5 Dynamic Redistribution

The final element of our parallel algorithm is the dynamic redistribution. As the annealing schedule proceeds, the initial partition becomes more irrelevant since it corresponds very little to the geographic partitioning of the rows. While the initial partitioning does reduce the amount of communication in terms of pin updates, the partition being spread across many rows can affect the row penalty calculations as well as cell mobility. For this reason, it is a good idea to repartition the cells so that the partition actually reflects the geographical row-based partition. The nets are assigned to the partition that contains the most cells from the particular net. The repartitioning is started only after the cells have settled within some proximity to their final destinations. This is done so that the net cut set will be reduced. Through empirical evidence, we have determined that repartitioning should begin at the 40th `TimberWolfSC` iteration, and every four iterations thereafter until iteration 120. After this point, the range limiter is so small that cells no longer move out of a row, so repartitioning is no longer necessary. Casotto et al. use a repartitioning scheme as well, but they are forced to repartition much more often and use a secondary simulated annealing procedure to perform the repartitioning [63]. Our method does not require frequent repartitioning because of the use of other error control methods, and the use of geographical row-based partitioning simplifies the partitioning procedure.

Redistribution is a synchronizing operation that has to be supported in an asynchronous message-driven environment. Enforcing synchronization in such an environment can be done by ensuring in the program flow that all threads arrive at the same point and process the synchronization step. In the sample code in Figure 6.11, `CircuitAggr::anneal()` has been modified to start up repartitioning when necessary as explained above. All the processors will find new locations for their cells and send them to the appropriate place using the `moveCells` continuation. Once a processor has received all the `MoveCellMsgs` from all other processors, it can re-enable the `anneal()` continuation once again. The waiting for `MoveCellMsg` becomes the synchronization point, but once the `anneal() method` has been re-enabled, the asynchronous behavior can start again.

```
void CircuitAggr::anneal()
{
tryCellMove();
if ( timeToRepartition() {
    // add cells to outgoing message
    MoveCellMsg mcmsg[numthreads] ;
    for ( i=0;i<numcells_i_own;i++) {
        newloc = find_new_location( carray[i] ) ;
        mcmsg[newloc].add( carray[i] );
    }
    for ( i=0;i<numthreads;i++) {
        CellAggr::moveCells::Continuation cont( nameOfRepresentative( i ) );
        cont( mcmsg[i] ) ;
    }
} else if ( terminationNotReached() ) {
    CircuitAggr::anneal::Continuation cont( *this );
    cont();
} else {
    finishUp();
}
}

void CellAggr::moveCells( MoveCellMsg &mcmsg )
{
count++;
// add mcmsg cells to local list
...
if ( count == numthreads ) {
    // re-enable anneal()
    CircuitAggr::anneal::Continuation cont( cktName );
    cont();
    count = 0;
}
}
```

Figure 6.11: Code for repartitioning.

```
ProperPLACE-PART()
if actor on thread 0
  then read circuit and distribute cells and nets to different threads
for each thread
    do while termination not reached
              do while TimberWolfSC iteration not complete
                        do ATTEMPT-MOVE()
                           attemptsCount++
                           if attemptsCount%U_p == 0
                             then UPDATE-PINS()
                           if attemptsCount%U_r == 0
                             then UPDATE-ROWS()
                                  FIX-ROW-DESIRES()
                 adjust U_p and U_r
                 UPDATE-FIXED-CELLS()
                 REMOVE-OVERLAPS()
                 if time to repartition
                   then REPARTITION()
```

Figure 6.12: Outline of the ProperPLACE-PART algorithm.

6.6 Algorithm Analysis

The ProperPLACE-PART algorithm is summarized in Figure 6.12, where U_p is the number of moves attempted between pin updates and U_r is the number of moves attempted between row updates. These values are adjusted dynamically.

The major overhead contributions in ProperPLACE-PART are due to the communication involved in the updates of the pins, rows, and cells. Assuming a balanced distribution of cells, each thread will attempt $\frac{I}{N}$ moves, where I is the number of moves attempted by TimberWolfSC, and N is the number of threads. Therefore, there are $\frac{I}{NU_p}$ pin updates and $\frac{I}{NU_r}$ row updates. TimberWolfSC typically completes 160 iterations, which means that there are 160 cell updates. If we define speedup, S, as $\frac{t_{ser}}{t_{par}}$, then we arrive at the following:

$$t_{ser} = It_m \tag{6.3}$$

$$t_{par} = \frac{I}{N}t_m + \frac{I}{NU_p}t_{up} + \frac{I}{NU_r}t_{ur} + 160t_{uc} \tag{6.4}$$

$$S = \frac{It_m}{\frac{I}{N}\left(t_m + \frac{t_{up}}{U_p} + \frac{t_{ur}}{U_r}\right) + 160t_{uc}} \tag{6.5}$$

where t_m, t_{up}, t_{ur}, and t_{uc} are the move attempt, pin update, row update, and cell update times, respectively. Since repartitioning is done infrequently we can ignore this component. Assuming a communication cost model of $t_c = t_o + bt_b$ where t_o is the overhead cost and t_b is the per-byte cost, an analysis of the communication pattern and typical circuit characteristics shows

$$t_{up} \approx 2N\left(t_o + 16\frac{C}{N}t_b\right) \tag{6.6}$$

$$t_{ur} \approx N\left(t_o + 4Rt_b\right) \tag{6.7}$$

$$t_{uc} \approx 2N\left(t_o + 20\frac{C}{N^2}t_b\right) \tag{6.8}$$

where C is the number of cells and R is the number of rows. `TimberWolfSC` statically sets the number of moves, I, to $4000C(\frac{C}{500})^{\frac{1}{3}}$. Substituting for I in Eq. (6.5) along with Eq. (6.8), we can see that the t_{uc} term can be ignored in S. Likewise, if we simplify the communication model and ignore t_o, we arrive at

$$S = \frac{Nt_m}{t_m + 32\frac{Ct_b}{U_p} + \frac{4RNt_b}{U_r}} \tag{6.9}$$

It is clear that U_p and U_r are the keys to determining the expected speedup. From experiments, we have determined that setting U_r to 30 and U_p to $2C$ yield the best tradeoff between quality and speedup.

Table 6.1: `ProperPLACE-PART` Results (Sun SparcServer 1000E)

	`TimberWolfSC` version 6.0	`ProperPLACE-PART` 1 proc		4 procs		8 procs	
Circuit	Run time(s)	W	S	W	S	W	S
primary1	340.4	1.02	1.16	1.01	2.79	1.03	5.83
industry1	1996.3	1.01	1.23	1.06	3.20	1.08	4.36
primary2	3230.2	1.01	1.11	1.02	3.20	1.03	6.05
biomed	9356.7	0.92	1.16	1.08	2.20	1.10	5.04
industry2	21556	0.95	0.97	1.13	2.20	1.11	4.98
avq.large	52221	1.00	1.22	1.05	2.43	1.09	4.77
Average		0.98	1.14	1.06	2.67	1.07	5.17

6.7 Experimental Results

6.7.1 Speedup and quality

We ran `ProperPLACE-PART` on a Sun SparcServer 1000E, a shared memory multiprocessor, as well as two distributed memory machines, the Thinking Machines CM-5 and the Intel Paragon. Results are presented for a set of benchmark circuits in Tables 6.1, 6.2, and 6.3. The first column indicates the run time in seconds for the serial `TimberWolfSC` 6.0 code running on one processor. The following columns are the results for our parallel implementation. W represents the normalized wirelength cost of the resultant placement, and S is the speedup relative to the serial `TimberWolfSC`. The dashed lines indicated that the circuit could not be run because of excessive memory requirements. For these larger circuits (biomed, industry2, avq.large), the reported `TimberWolfSC` times are extrapolated from a Sun 4/690MP, a machine with comparable uniprocessor performance.

These circuits show the advantage of using a circuit-partitioned approach to exploit the large memory resources on distributed processors. As circuits get larger and larger, such an approach is essential to place these circuits on memory starved machines. In addition, the algorithm provides speedups of 10 to 12 on 32 processors.

Table 6.2: `ProperPLACE-PART` Results (Thinking Machines CM-5)

	`TimberWolfSC` version 6.0	`ProperPLACE-PART`							
		1 proc		8 procs		16 procs		32 procs	
Circuit	Run time(s)	W	S	W	S	W	S	W	S
primary1	1266.0	1.01	0.71	1.06	5.34	1.08	6.30	‡	‡
industry1	5664.7	1.00	0.88	1.03	6.93	1.01	9.78	1.09	10.57
primary2	8931.1	1.03	1.00	1.06	4.07	1.09	7.43	1.13	9.89
biomed	22621.†	-	-	1.05	4.54	1.13	6.20	1.10	9.89
industry2	56058.†	-	-	1.04	5.04	1.08	7.44	1.10	10.14
Average		1.01	0.86	1.05	5.18	1.08	7.43	1.10	10.12

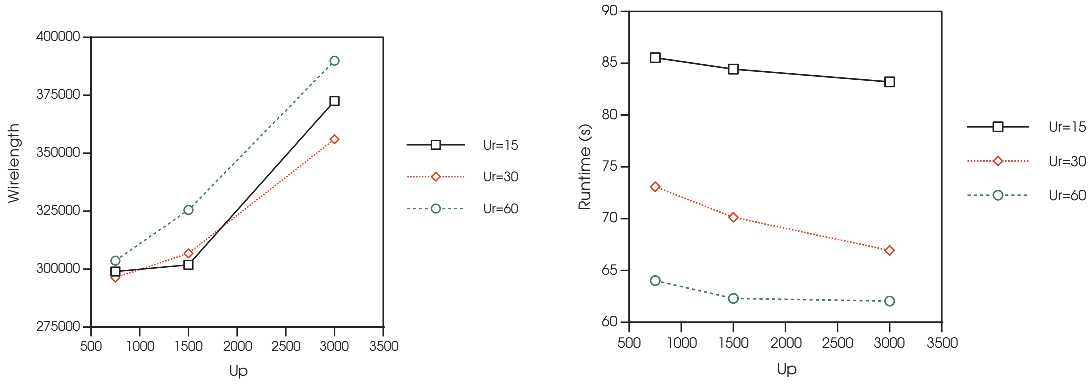
Table 6.3: `ProperPLACE-PART` Results (Intel Paragon)

	`TimberWolfSC` version 6.0	`ProperPLACE-PART`							
		1 proc		8 procs		16 procs		32 procs	
Circuit	Run time(s)	W	S	W	S	W	S	W	S
primary1	786.68	1.01	0.93	1.00	4.41	1.08	6.42	‡	‡
industry1	3004.8	0.95	0.90	1.03	3.85	1.13	6.37	1.12	11.19
primary2	5108.7	1.00	0.94	1.04	5.08	1.13	7.13	1.09	12.53
biomed	16136.†	-	-	1.03	2.73	1.06	6.62	1.07	12.06
industry2	31777.†	-	-	1.06	2.64	1.08	4.29	1.13	9.12
avq.large	62477.†	-	-	-	-	1.05	6.23	1.08	11.06
Average		0.99	0.92	1.02	3.74	1.09	6.18	1.10	11.19

6.7.2 Error control

We also compared the effect of U_r on the final quality and speedups. Figure 6.13 shows the results for *primary1* on an eight-processor SparcServer 1000. It can be seen that both U_r and U_p have significant effects on the quality and speedups. We have selected a U_r of 30 as an optimal value. U_p is more circuit dependent because the size of the pin update messages grows as the circuit grows. Setting U_p to $2C$ provides the best tradeoff between performance and quality.

†Estimated times extrapolated from Sun4 690/MP.
‡Circuit was not large enough to be effectively partitioned over 32 processors.



(a) Effect on wirelength (b) Effect on run time

Figure 6.13: Effect of U_r and U_p (*primary1*, SS1000, 8 processors).

Table 6.4: Effect of Error Control

	`TimberWolfSC` 1 proc		`ProperPLACE-PART` (8 procs) No error control		With error control	
Circuit	Wirelength	Time(s)	Wirelength	Time(s)	Wirelength	Time(s)
primary1	294304	340.40	540142	50.35	302239	58.36
industry1	661644	1353.28	1052296	181.23	713913	312.65
primary2	1766174	3230.18	2499589	352.90	1864454	534.43

As a measure of the effectiveness of all the error control mechanisms, we turned off all these mechanisms and compared the results. These are shown in Table 6.4. It can be seen that the error control methods that we used contributed greatly to reducing the error, at a cost of some speedup.

6.7.3 Comparison

We now compare this partitioned algorithm for parallel cell placement with the algorithms detailed in earlier chapters, parallel moves (`ProperPLACE-PM`), multiple Markov chains (`ProperPLACE-MMC`), and speculative computation (`ProperPLACE-SC`). Table 6.5 shows the relative wirelengths (W) and speedups (S) for a few circuits on an eight-processor Intel Paragon. The partitioned algorithm does not offer speedups that are as good as `ProperPLACE-MMC`. However, the key advantage is that the circuit has been partitioned allowing a very large circuit. Remem-

Table 6.5: Comparison with Previous Algorithms (Intel Paragon, 8 processors)

Circuit	ProperPLACE PART		ProperPLACE MMC		ProperPLACE PM		ProperPLACE SC	
	W	S	W	S	W	S	W	S
primary1	1.06	4.99	1.06	5.67	1.04	3.95	1.00	0.20
primary2	1.09	4.79	1.09	6.75	1.10	3.85	1.00	0.20
avq.large	1.08	3.50	-	-	-	-	-	-

ber that all previous parallel placement algorithms require the circuit to be duplicated, thereby preventing very large circuits from being run. This is clear for the *avq.large* circuit which can not be run with any of the algorithms because of inadequate memory. Moreover, the wirelengths in `ProperPLACE-PART` are comparable with those in `ProperPLACE-MMC`.

6.8 An MPI Implementation

Because of the wide variety of parallel architectures and thus programming models, there has been a significant amount of work in creating a standard interface for writing message passing software. Among these are the Message Passing Interface [4], Parallel Virtual Machine (PVM) [64], and p4 [65]. MPI seems to be gaining support as the de facto standard for an accepted message passing interface. It is an inclusive standard that supports virtually all styles of send-receive communication, group protocols, reductions, and noncontiguous data structures.

In light of the popularity of the MPI standard, a version of the circuit partitioned algorithm was implemented using the MPI protocol. This implementation will be compared with the ProperCAD II version as a measurement of the efficiency of the ProperCAD II platform. Using an actor model such as ProperCAD II offers certain programming conveniences and portability across architectures, but it is not clear at what cost. This investigation should help answer that.

We implemented the algorithm using the Message Passing Interface (MPI) [66]. Using the MPICH [67] library implementation of MPI, we were able to build an implementation of the circuit partitioned algorithm called `mpiPLACE`. We ran `mpiPLACE` on a Sun SparcServer 1000E as well as the Intel Paragon. The speedups and wirelengths for a set of benchmark circuits are

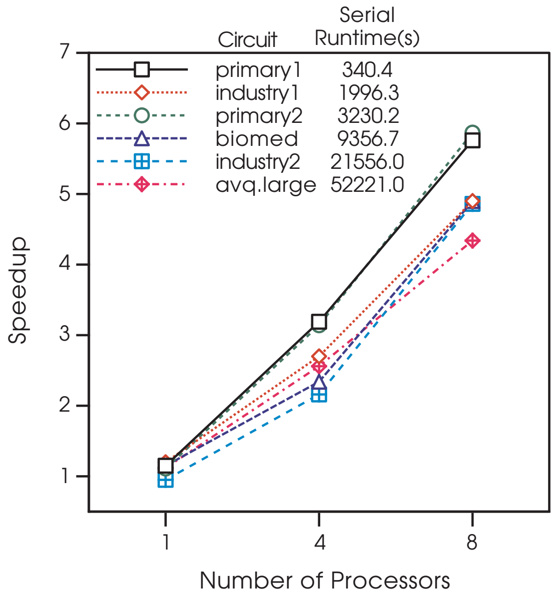


Figure 6.14: `mpiPLACE` speedups (Sun SparcServer 1000E).

shown in Figures 6.14 and 6.15, and Tables 6.6 and 6.7. The speedup graphs show an average speedup of over 5 on 8 processors on the SparcServer and an average speedup of over 11 on 32 processors of the Paragon. The tables give the normalized wirelength cost of the resultant placement, and the dashed lines indicate that the circuit could not be run because of excessive memory requirements.

These results of `mpiPLACE` are very similar to those for the `ProperPLACE-PART` version. It is clear that the ProperCAD II environment does not present any significant overheads com-

Table 6.6: `mpiPLACE` Wirelengths (Sun SparcServer 1000E)

Circuit	1 proc	4 procs	8 procs
primary1	1.02	1.02	1.02
industry1	1.01	1.05	1.09
primary2	1.01	1.01	1.05
biomed	0.92	1.05	1.08
industry2	0.95	1.10	1.09
avq.large	1.00	1.07	1.07
Average	0.98	1.05	1.07

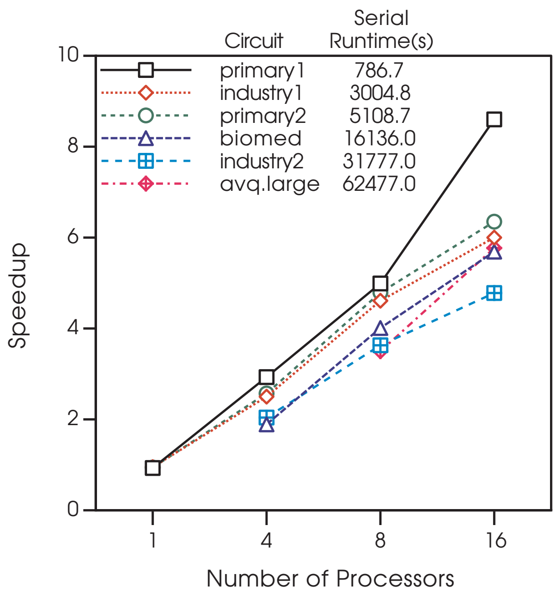


Figure 6.15: `mpiPLACE` speedups (Intel Paragon).

Table 6.7: `mpiPLACE` Wirelengths (Intel Paragon)

Circuit	1 proc	4 procs	8 procs	16 procs
primary1	1.00	1.02	1.06	1.09
industry1	0.98	1.03	1.06	1.13
primary2	1.00	1.05	1.09	1.14
biomed	-	1.04	1.04	1.08
industry2	-	1.07	1.06	1.09
avq.large	-	-	1.08	1.07
Average	0.99	1.04	1.07	1.10

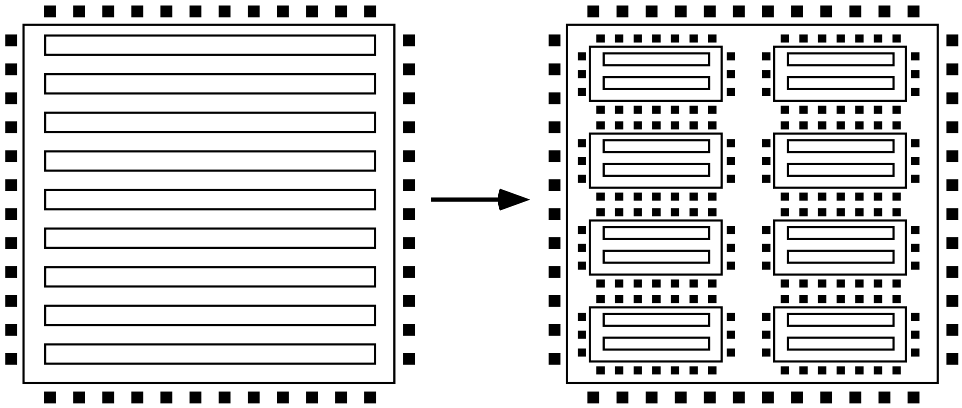


Figure 6.16: Example of eight-way circuit partitioning.

pared to a normal message passing library such as MPI. The circuit partitioned algorithm exhibits characteristics of an SPMD style of programming for which MPI is the natural interface. Yet, the `ProperPLACE-PART` implementation is just as efficient, showing that ProperCAD II can be used to write efficient SPMD code in spite of the message driven underlying model.

Using `mpiPLACE` we performed two more experiments to further show the usefulness of the algorithm in situations where the circuit is too large to be placed in the memory of a single machine. The traditional approach has been to take these circuits, partition them, place each partition separately and then merge the separate placements. As an example, Figure 6.16 shows a large circuit that has been divided into eight smaller circuits. Each of these individual circuits can be placed independently and then finally combined into the larger placement. Using a similar procedure, we took the largest circuit available to us, avq.large, and partitioned it into eight subcircuits using the Fiduccia-Mattheyses method. Each partition is placed individually using `TimberWolfSC` and then merged back together. The results are shown in Table 6.8 along with comparisons with `TimberWolfSC` and `mpiPLACE`. The run time reported for the partitioned `TimberWolfSC` approach is the maximum of the run times for the eight partitions. Note that `mpiPLACE` has much better quality for roughly equivalent run times. Bear in mind, also, that the time reported for the partitioned `TimberWolfSC` approach includes only the placement time and not the time for merging or partitioning the circuit.

In addition, we wanted to find how effective `mpiPLACE` is when run on a large number of processors. Again using the avq.large circuit, we ran `mpiPLACE` on an Intel Paragon with 128

Table 6.8: Comparison with Partitioned Placement (Sun SparcServer 1000E)

avq.large		
	Wirelength	Run Time (s)
`TimberWolfSC` (1 processor)	11545887	79941
`TimberWolfSC` 8-way partitioned (1 processor)	16642589	9699
`mpiPLACE`(8 processors)	12406356	14975

Table 6.9: `mpiPLACE` Results (Intel Paragon, 128 processors)

avq.large		
	Wirelength	Speedup
1 processor	11545887	-
16 processors	12406356	6.23
128 processors	28969401	55.1

processors. The results are shown in Table 6.9. Note that the quality suffers severely, so it is clear that such an approach is not appropriate for a large number of processors unless the circuit is sufficiently large. Current circuit sizes are not large enough to take advantage of such a large number of processors, but in the future it is anticipated that circuits will be large enough.

6.9 Summary

In this chapter, we describe `ProperPLACE-PART`, a circuit partitioned approach to parallel cell placement. Using sophisticated error control mechanisms to improve solution quality, the algorithm is able to achieve reasonable speedups with moderate degradation in quality. Though it does not provide the speedups seen in MMC, the primary advantage of partitioning the circuit is that it helps memory scalability. We are able to run circuits that are too large to fit on one processor by distributing it across the nodes of a multiprocessor. In addition, we also describe an MPI implementation of this algorithm. The results show that the overheads in ProperCAD II to support a message driven style of programming are no worse than for any other portable message passing library.

CHAPTER 7

TIMING DRIVEN APPROACH

The placement algorithms discussed so far have dealt only with minimization of the wirelength and indirectly area minimization. For current high density circuits, this cost function is no longer appropriate. As VLSI designs grow and feature sizes shrink, overall circuit performance becomes more closely related to the interconnect timing characteristics. Timing driven placement is the process of simultaneously minimizing the circuit area as well as minimizing the critical path delays. During the physical design process, a common heuristic used is to minimize total net wirelength as an approximation of the area. Minimizing the wirelength of a net would seem to minimize the effect of interconnect delays. However, timing is not determined solely by the delays of individual nets, but instead by a sequence of nets or a signal path. Moreover, only nets on the longest or critical paths in the circuit are of concern.

There have been two general approaches to timing driven placement - net based [68–70] and path based [71–75]. Net based algorithms identify critical paths a priori and assign criticality weights or upper bounds for each net in the path, and then guide the placement process based on these bounds. The pre-timing analysis may not be able to effectively select measures for each net; thus, the placement quality may suffer. Path based approaches address this problem by doing complete path delay analysis during placement. This is, of course, at the cost of an increase in computational complexity.

The other critical component of timing driven placement is the choice of the delay model to approximate interconnect behavior. To simplify calculation, most previous approaches have used a basic delay model. The simplest models are based on the assumption that pin to pin delay is proportional to the net wirelength [69, 76]. Other more detailed models use a simplified RC model where wire capacitance can be computed either as a function of fanout or proportional to wire length [71–74]. These models do not take into consideration wire resistance, which is becoming more and more important as designs scale down to submicron features. Additionally, these models do not account for driver pin location or the delay characteristics of a distributed RC tree.

In this chapter, we describe a new placement algorithm that incorporates a detailed interconnect model based on the Elmore delay model. It uses complete path delay information to derive a placement that minimizes the longest path delay without sacrificing area quality. The chapter is organized as follows. Section 7.1 describes the timing analysis methodology used in this chapter. Section 7.2 details the algorithm used for placement, and the results are shown for a few MCNC benchmark circuits in Section 7.3. Section 7.4 describes the parallel algorithm and the results of the speedups on a Sun SparcServer 1000E and Intel Paragon.

7.1 Timing Analysis

7.1.1 Delay model

The best method available for accurate timing analysis of circuits is the SPICE circuit simulation tool. Because the computation requirements of SPICE are prohibitive, it is impractical to use it during placement. Linear delays proportional to net length are more tractable but also more inaccurate. An intermediate model such as the Elmore delay [77] is of more interest because of its accuracy and reasonable computation times. The Elmore delay model has been shown to have very good correlation to SPICE computed timing analysis [78].

The Elmore delay model is useful for approximating delays of distributed RC trees. Given a net tree rooted at a source pin p_0, the delay from p_0 to a sink p_j can be expressed as follows:

$$d_j = r_0 C_0 + \sum_{\epsilon_i \in path(p_0, p_j)} r_{\epsilon_i} \left(c_{\epsilon_i}/2 + C_i \right) \tag{7.1}$$

where ϵ_i is the edge from pin p_i to its parent and c_{ϵ_i} and r_{ϵ_i} are the capacitance and resistance, respectively, along that edge. C_i is the tree capacitance at pin p_i, in other words, the sum of all edge and sink capacitances on the tree rooted at pin p_i. The d_i's for all sink pins can be calculated in a two-phase process. In the first step, the delays for each edge are calculated in a depth first search of the tree; likewise, in the second traversal of the tree, the edge delays can be summed up for each pin. Each step is an O(n) process where n is the number of pins on the net.

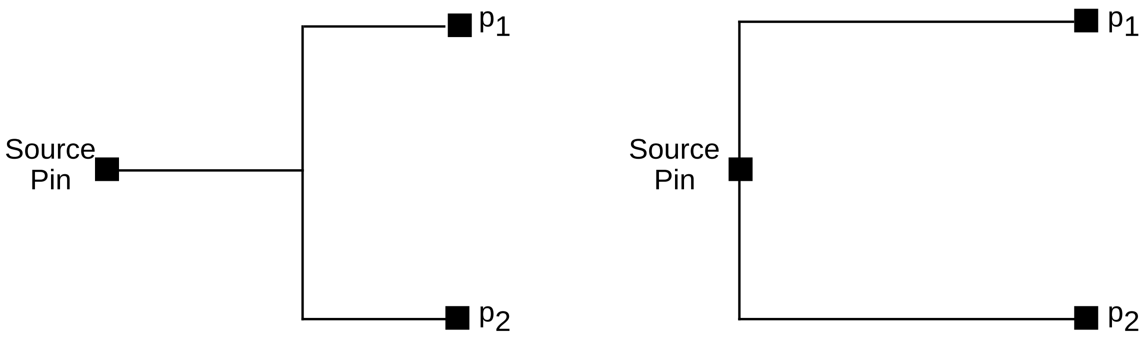
It is clear that the routing structure of the net significantly affects the computation of delays. For example, the net in Figure 7.1(a) can be routed alternatively as in Figure 7.1(b), and the equivalent RC models are shown in Figure 7.2. At node 3, there is an implicit Steiner pin that has no sink capacitance. The delays from source to sink p_2 for the two routes are shown below.

$$d_{2_a} = (r_0 + r_{\epsilon 3})(c_{\epsilon_1} + c_1 + c_{\epsilon_2} + c_2) + c_{\epsilon 3}\left(r_0 + \frac{r_{\epsilon 3}}{2}\right) + r_{\epsilon 2}\left(\frac{c_{\epsilon 2}}{2} + c_2\right) \quad (7.2)$$

$$d_{2_b} = r_0\left(c'_{\epsilon_1} + c_1 + c'_{\epsilon_2} + c_2\right) + r'_{\epsilon 2}\left(\frac{c'_{\epsilon 2}}{2} + c_2\right) \quad (7.3)$$

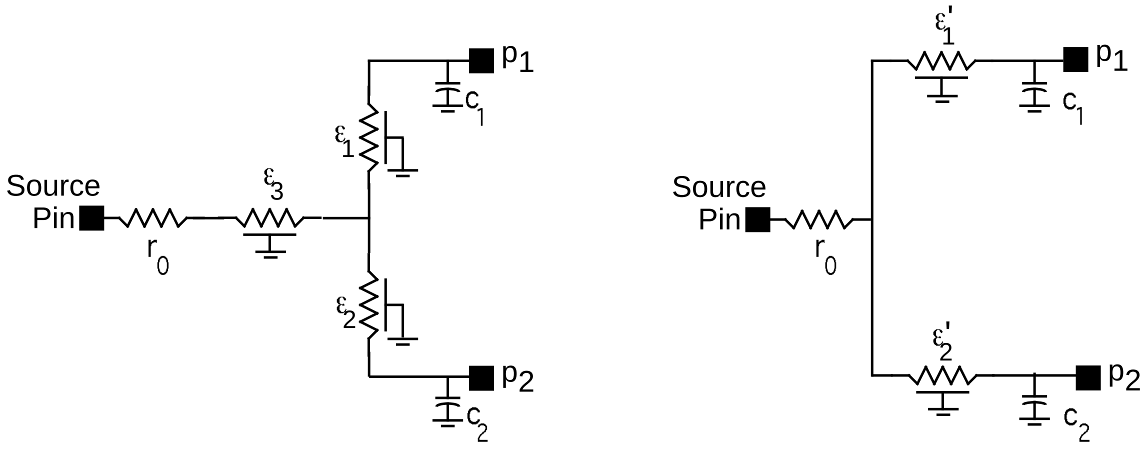
The example makes it clear that it is important that the routing be known before an accurate delay can be computed. Several strategies exist to construct near-optimal Steiner trees for improved performance during the routing process [79–81]. However, during placement, it is impractical to use any of these algorithms to optimally route each net because of the computation time required to do so.

Instead, we quickly approximate the Steiner tree, by building a trunk based tree rooted on the source node. The bounding box of each net is partitioned into a 4x1 grid as shown in Figure 7.3(a). If the source node is in the shaded region, a horizontal trunk is created and segments are built off the trunk to connect sink pins to the trunk. Implicit Steiner pin points are created where the segments meet the trunk. Likewise, if the source node is in the hatched region, segments are built off a vertical trunk. This Steiner tree construction methodology is very quick and can be used effectively during the placement. The majority of nets in most designs have two or three terminals. For two terminal nets, the trees generated by this heuristic are obviously optimal. For three terminal nets, however, the approximate Steiner tree may not be optimal.



(a) Alternative 1 with Steiner Pin (b) Alternative 2 without Steiner pin

Figure 7.1: Routing alternatives for a net.



(a) Alternative 1 with Steiner pin (b) Alternative 2 without Steiner pin

Figure 7.2: Distributed RC models for net.

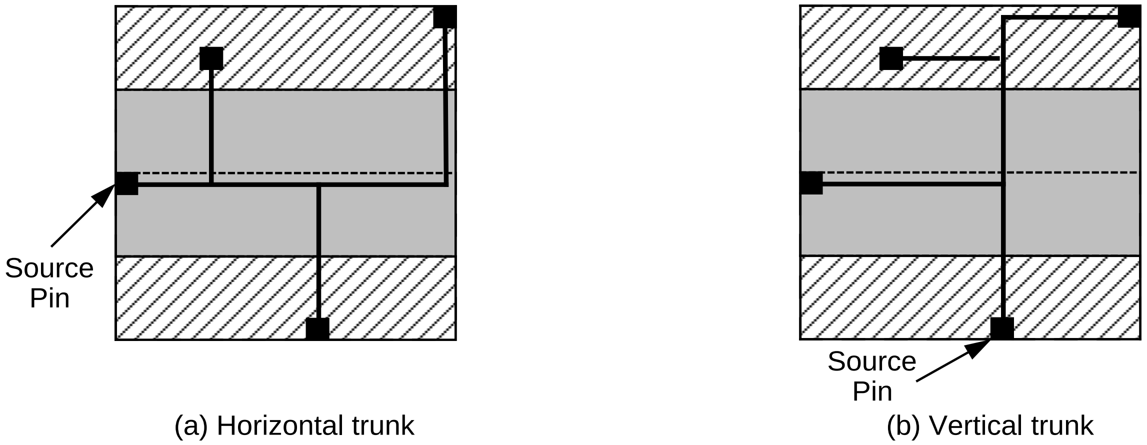


Figure 7.3: Steiner approximation for nets.

Since optimal Steiner trees can be quickly created for three terminal nets, we treat these nets as a special case and generate optimal trees. By doing so, we can ensure that 75% to 95% of nets in a typical design will have optimal delay trees.

7.1.2 Path delay analysis

The previous section showed how to calculate the pin to pin delay on a particular net. In this section, we describe the analysis methodology to compute path delays. As described in [82,83], we use a block oriented technique in which all the cells are levelized, and then cell and net delays are processed in block order. Once all delays have been computed, the maximum output delay, T_d, can be determined by examining delay times at each output pin. The longest path can easily be arrived at by tracing back from the output pin with the maximum delay. This algorithm is also O(n) where n is the number of cells.

The algorithms for path delay analysis are summarized in Figure 7.4. LEVELIZE-CIRCUIT is based on a simple breadth-first topological sort algorithm. Each cell, v, is a node in the graph G. From the primary inputs, a breadth-first search is initiated such that each edge is traversed once. As each node is visited, the maximum level is assigned to that node, and then when all edges incident on that node have been traversed, every fanout from that node is then explored. COMPUTE-DELAYS processes the cells in levelized order and calculates the delays on each pin, by first calculating the cell delay and then the net delay.

LEVELIZE-CIRCUIT(G)
for each $v \in G$
  do $level[v] \leftarrow -1$
for each $v \in PrimaryInput[G]$
  do BREADTH-FIRST-VISIT$(v, 1)$

BREADTH FIRST VISIT(v, l)
if $l > level[v]$
  then $level[v] \leftarrow l$
if all input edges traversed
  then $\Lambda_l \leftarrow \Lambda_l \cup v$
    if v is not a primary output
      then for $u \in fanout(v)$
        do BREADTH-FIRST-VISIT$(u, l + 1)$

COMPUTE-DELAYS$()$
for $l \leftarrow 1$ **to** $maxLevel$
  do for each $v \in \Lambda_l$
    do CALCULATE-CELL-DELAY(v)
      for each $\eta \in fanout[v]$
        do CALCULATE-NET-DELAY(η)

Figure 7.4: Delay analysis algorithms.

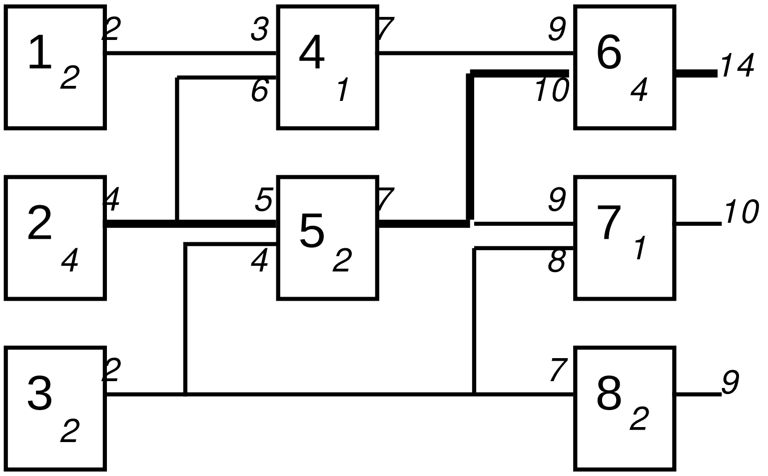
The path construction algorithms are summarized in Figure 7.5. CONSTRUCT-ALL-PATHS finds the longest paths terminating at all primary outputs including all flip-flops. The list Π keeps track of all accumulated paths. After initially calling CONSTRUCT-PATH on all primary outputs, the paths can then be constructed by tracing back towards the primary inputs calling CONSTRUCT-PATH in reverse levelized order. CONSTRUCT-PATH builds a path back from a cell by determining the critical input net, i.e., the net that has the longest delay. Then we create a path segment, ψ, which is simply a 4-tuple identifying the critical net, the source and sink cells, as well as the path to which the path segment belongs. This path segment is then added to all paths, π, that the cell v is on. The path segment is then added to the source cell as well as the critical net.

An example of the longest path is shown in Figure 7.6. The cells are processed in block order, which in this example is 1,2,3,4,5,6,7,8. The nominal cell delay is shown in italic within

```
CONSTRUCT-ALL-PATHS()
Π ← ∅
for each primary output cell v
    do π ← new path
       Π ← Π ∪ π
       ADD-PATH-TO-CELL(v, π)
       CONSTRUCT-PATH(v)
for l ← maxLevel to 2
    do for each v ∈ Λ_l
           do CONSTRUCT-PATH(v)

CONSTRUCT-PATH(v)
maxDelay ← 0
for each input net η adjacent to v
    do s ← source(η)
       if maxDelay < delay[s]
         then maxDelay ← delay[s]
              maxSource ← s
              maxInputNet ← η
for each path π that v is on
    do ψ ← (v, maxSource, maxInputNet, π)
       ADD-SEGMENT-TO-PATH(π, ψ)
       ADD-PATH-TO-CELL(maxSource, π)
       ADD-PATH-TO-NET(maxInputNet, ψ)
```

Figure 7.5: Path construction algorithms.



Path	Delay
2 5 6	14
2 4 6	13
3 5 6	13
1 4 6	10
2 5 7	10
3 5 7	9
3 7	9
3 8	9

Figure 7.6: Longest path analysis.

the cell, and the arrival times are shown in italic at the cell boundaries. The actual interconnect delays are not shown but are readily calculated by subtracting appropriate arrival times. The longest path is highlighted in bold, and all other viable paths are listed by the side.

Special care must be taken for sequential circuits. We assume that the circuit can be represented as a Moore model finite state machine as shown in Figure 7.7. To transform such circuits when constructing delay paths, each latch element output must be treated as a normal primary input and, likewise, each latch element input is treated as a primary output as shown in Figure 7.8. There are now four types of delays that appear, PI to PO, PI to latch, latch to PO, and latch to latch. Each of these delays must be accounted for separately, as the minimization objective may only be one or more of these specific delays.

7.2 Timing Driven Placement

Our timing driven placement algorithm is based on the algorithm used in TimberWolfSC 6.0 as described in Section 2.2. The cost function has been modified as shown in Eq. (7.4) where W is the estimate of wirelength of all nets as the half perimeter of the bounding box, P_O is the penalty for area overlap between cells in the same row, and P_R is the penalty for the difference between the actual row length and the desired row length. The coefficient terms, μ and λ, are

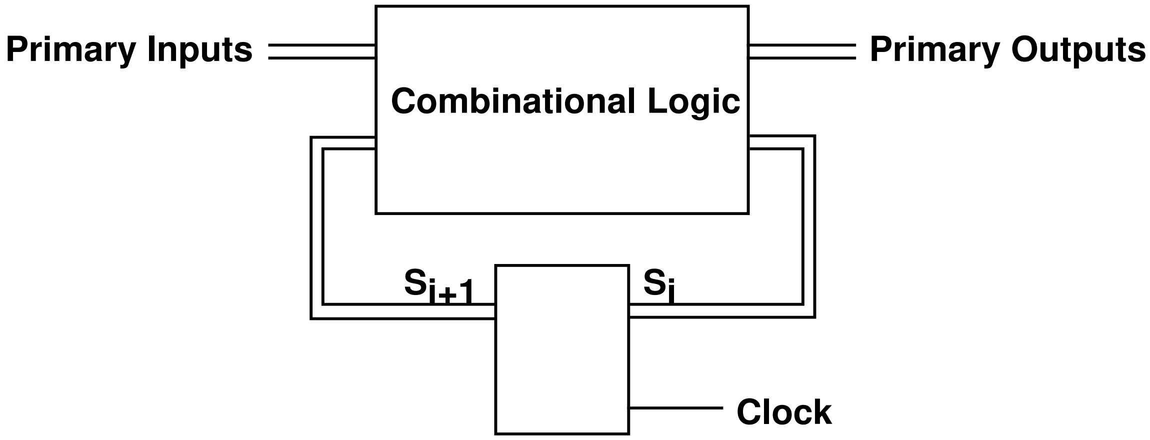


Figure 7.7: Moore model finite state machine.

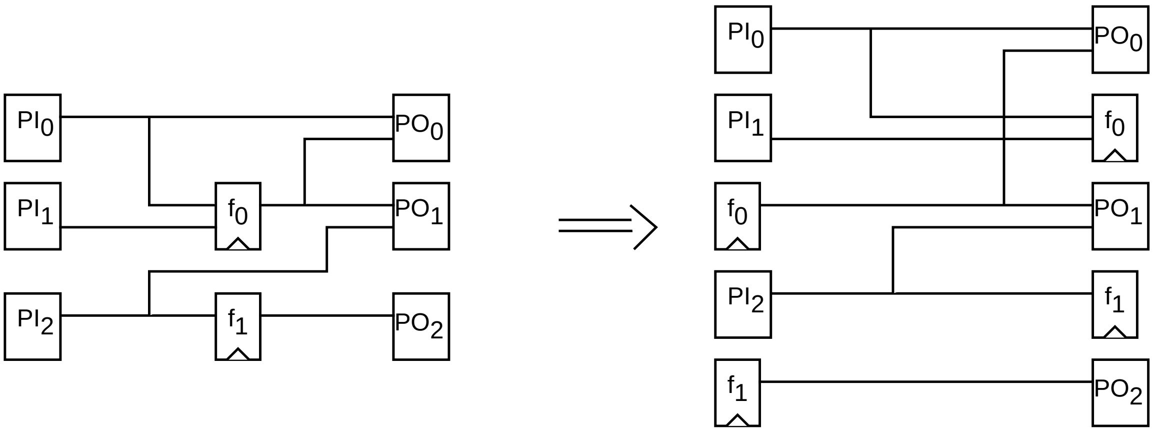


Figure 7.8: Sequential circuit transformation.

adjusted using a feedback control mechanism to arrive at optimal values [38]. Note that these terms are similar to the cost function used in TimberWolf (Eq. (2.2)).

$$C = W + \mu P_O + \lambda P_R + \gamma T_d \tag{7.4}$$

More importantly, we have added a cost term that is used to minimize the longest path delay, T_d. The γ weight controls the tradeoff between delay and area minimization. Through experimentation, in order to achieve the minimal path delay with the least area expense, we have set

$$\gamma = \frac{W_0}{T_{d_0}} \tag{7.5}$$

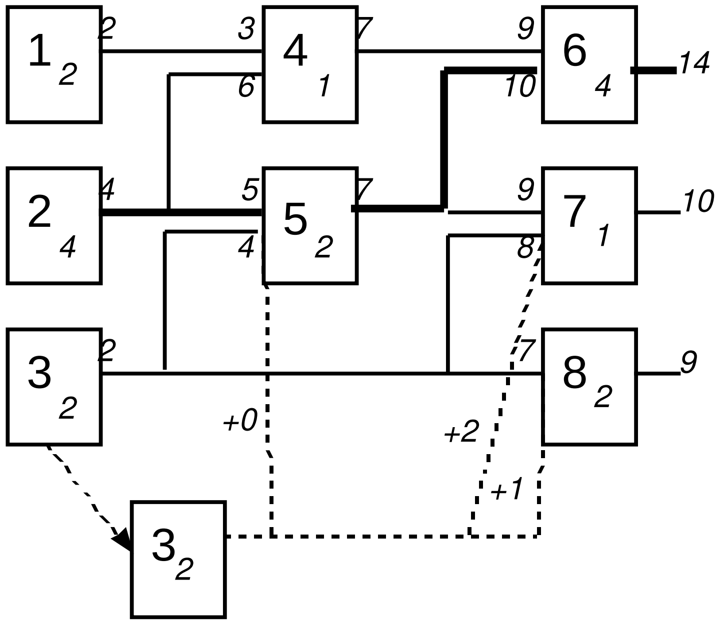
W_0 and T_{d_0} are the initial wirelength and path delay.

7.2.1 Path delay cost calculation

In this section, we describe how to efficiently keep track of longest path delays during placement. In the context of simulated annealing, every time a move is made, it can possibly affect the longest path delay. Using the algorithm in Section 7.1.2, we can determine the longest path, and then whenever a move is attempted, if a net on the critical path is perturbed by the move, the ΔT_d is easy to calculate. Keeping track of only one critical path can lead to problems because moves that may not affect the pre-determined critical path may create new critical paths. It is not practical to recalculate the longest path for each move attempt so it is necessary to monitor several possible paths.

One solution is, as in previous work [37, 76], to have the designer provide a set of paths or critical nets that the placement algorithm would use in path delay minimization. However, in large designs, it is very difficult for a user to identify these critical paths beforehand. Especially in light of complicated interconnect delays, this task is more difficult than ever.

Instead, as was done by Swartz and Sechen [74, 84], our algorithm identifies these critical paths for the user. However, as the number of elements in a design increases, the number of possible paths increases exponentially. For very large circuits, keeping track of all paths becomes very intractable. Therefore, we identify only longest paths between all pairs of inputs and out-



Path	Delay	
2 5 6	14	
2 4 6	13	
3 5 6	13	
1 4 6	10	
2 5 7	10	
3 5 7	9	+0
3 7	9	+2
3 8	9	+1

Figure 7.9: Longest path cost determination.

puts. For sequential circuits, this includes all latch inputs and outputs as well. This reduces the number of paths considerably. In addition, only paths that have delays within 10% of the longest path delay are kept. With these restrictions, only paths (2,5,6) and (3,5,6) from Figure 7.6 will be kept. As more moves become accepted and our list of longest paths is no longer applicable, it is necessary to periodically recalculate the longest paths. We have found that is sufficient to perform this calculation every C accepted moves where C is the number of cells in the circuit.

Note that when a move is proposed the entire path need not be traced from source to sink to determine the change. Instead, each net keeps a list of paths that it is part of and applies its Δ to all these paths. In Figure 7.9, cell 3 has been moved causing three nets to change. This move affects three paths (3,5,7), (3,7), (3,8). Instead of tracing the effect of the move all the way to the outputs on all paths, since each affected net has a link to the path, we can apply the Δ directly to the path. We then process the list of paths, to determine the new longest path. In this case, the longest path will not change because of this move. The intermediate arrival times will be inaccurate of course, but that is acceptable, since our only concern during annealing is the change of the longest path delay.

```
PATH BASED TIMING DRIVEN PLACEMENT()
LEVELIZE-CIRCUIT()
COMPUTE-DELAYS()
CONSTRUCT-ALL-PATHS()
while termination not reached
        do while iteration not complete
                do attempt move
                    if any of the affected nets on a critical path
                        then calculate Δ of longest path delay
                                accept or reject move
                    update temperature
                    if time to recompute path delays
                        then COMPUTE-DELAYS()
                             CONSTRUCT-ALL-PATHS()
```

Figure 7.10: Timing driven placement algorithm.

7.3 Experimental Results

We have implemented the new placement algorithm in C++ on a Sun SparcServer 1000E and compared the results with TimberWolfSC 6.0 (Figure 7.3). Our results use four of the MCNC benchmarks which include timing information (Table 7.1. We use technology parameters from the MOSIS 2.0 design rules as shown in Table 7.2. The area and delay numbers are taken after the circuit has been globally routed, but the wirelength and execution times are only for the placement procedure. Note that since the global router is not timing driven, the delay does vary somewhat from that predicted by the placement process. The results show an average of 12% improvement in the longest path delay at the cost of about 5% increase in area. The execution time is roughly four times worse, which is comparable to the results presented in [74], in spite of a more accurate delay model. By using a more accurate delay model, we are able to select critical paths that may not be apparent in a less detailed model. Also, the accurate delay model allows us to be more confident in the final longest path timing characteristics.

Table 7.1: Circuit Information
MCNC Benchmarks

Circuit	Number of cells	Number of PIs	Number of POs	Number of flip flops	Number of nets
fract	125	22	2	16	147
struct	1888	32	32	0	1920
biomed	6417	25	72	655	5742
avq.small	21854	30	34	4041	22114

Table 7.2: Technology Parameters
MOSIS 2.0 Design Rules and MCNC Benchmark Parameters

metal width	3 μm
sheet resistance (metal1)	.108 $\Omega/\Box$
substrate capacitance (metal1)	.027 fF/μm
sheet resistance (metal2)	.045 $\Omega/\Box$
substrate capacitance (metal2)	.021 fF/μm
sink capacitance	from 9.47 to 165 fF
source resistance	from 1.18 to 5.44 kΩ

Table 7.3: Timing Driven Placement Results

Circuit	TimberWolfSC 6.0				Timing Driven			
	Longest Delay(ns)	Area (mm^2)	Wire-length	Run Time(s)	Longest Delay(ns)	Area (mm^2)	Wire-length	Run Time(s)
fract	20.3	0.53	39636	74.1	20.0	0.54	42863	118.2
struct	102.3	11.9	956328	916.6	96.8	12.3	997781	1220
biomed	56.7	120.5	5599231	8571	52.4	111.2	5231329	123835
avq.small	866.3	1220	41194758	16631	561.3	1500	44035597	81667
Pct. change					-12.4%	+5.1%	+3.2%	+457%

```
PARALLEL-COMPUTE-DELAYS()
for all processors
    do for l ← 1 to maxLevel
            do for each v ∈ Λ_l
                    do if this processor owns cell v
                          then CALCULATE-CELL-DELAY(v)
                                for each η ∈ fanout[v]
                                    do CALCULATE-NET-DELAY(η)
                                        mark changed pins
                send new pin delays of marked pins
                receive new pin delays
```

Figure 7.11: Parallel delay calculation algorithm.

7.4 Parallel Timing Driven Placement

As can be seen in the results from the previous section, providing timing driven placement adds significant overhead to the normal run time of cell placement. In this section we describe an algorithm for parallelization of timing driven placement. The algorithm is based on the approach described in Chapter 6.

7.4.1 Path delay analysis

As with the serial algorithm, there are two phases to the delay analysis: delay calculation and path construction. Calculating delays is done as before, by processing the cells in levelized order. However, since the cells are distributed, we must now send all the updated delays to the appropriate location. This causes a synchronization point at each level. The algorithm is summarized in Figure 7.11.

Likewise, path construction must be done with synchronization points at each level. The algorithm is summarized in Figures 7.12 and 7.13. As we proceed back from the primary outputs, each processor identifies path segments using the CONSTRUCT-PATH algorithm described in Figure 7.5. At the first level, each path, π, contains only one path segment, ψ. If the critical net, η, on ψ is owned by another processor, ψ is added to Ψ_b, a list of backward updates. Otherwise, ψ is added to Ψ_f. After the CONSTRUCT-PATH has been called on all primary outputs, the path segments in Ψ_b are sent to the owners of the relevant nets. This allows the owners of the nets to accumulate all the path segments that belong to a particular net.

At this point, as in the serial case, we trace backwards in the circuit by going through the cells in reverse levelized order. There is an implicit synchronization at each level because of the updates. When a processor receives a set of backward updates, the updates are then transferred to Ψ_f, the set of forward updates. Ψ_f updates are sent so that all copies of a net have all the path segments belonging to that net.

7.4.2 Parallel placement algorithm

The algorithm for parallel timing driven placement is summarized in Figure 7.14. In structure, it is very similar to the algorithm for `ProperPLACE-PART` shown in Figure 6.12. The only major modification is to insert the call to perform delay analysis as described above. One minor change is also applied to the repartitioning algorithm. In the non-timing driven placement algorithm, the nets are assigned to the partition containing the most cells attached to the net. Because of the parallel delay analysis approach, this heuristic is no longer appropriate. Instead, we assign nets to the partition containing the source pin. This approach limits us to circuits with only a single source pin per net. This is not a severe limitation, as it is easy to transform a multisource net into a single source net through insertion of intermediary buffers.

7.4.3 Experimental results

The parallel timing driven placement algorithm, called `mpiPLACE-TIME`, has been implemented using the Message Passing Interface (MPI). We present experimental results on a Sun SparcServer 1000E as well as the Intel Paragon in Tables 7.4 and 7.5. As was evident with the original `ProperPLACE-PART`, we get reasonable speedups with moderate wirelength degradation. There is little degradation of the delay as well.

PARALLEL-CONSTRUCT-ALL-PATHS()

for all processors
  do $\Pi \leftarrow \emptyset$
    $\Psi_b \leftarrow \emptyset$
    $\Psi_f \leftarrow \emptyset$
    for each $v \in PrimaryOutput[G]$
      do if this processor owns cell v
        then $\pi \leftarrow$ **new** $path$
          $\Pi \leftarrow \Pi \cup \pi$
          ADD-PATH-TO-CELL(v, π)
          CONSTRUCT-PATH(v)
          $\psi \leftarrow$ path segment $\in \pi$
          $\eta \leftarrow \eta(\psi) \triangleleft$ critical net component of ψ
          if η owned by another processor
            then $\Psi_b \leftarrow \Psi_b \cup \psi$
            else $\Psi_f \leftarrow \Psi_f \cup \psi$
    send Ψ_b
    for $l \leftarrow maxLevel$ **to** 2
      do RECEIVE-BACKWARD-UPDATES()
        send Ψ_f
        RECEIVE-FORWARD-UPDATES()
        for each $v \in \Lambda_l$
          do CONSTRUCT-PATH(v)
            for each new path segment $\psi \in \pi \in \Pi$
              do if $\eta(\psi)$ owned by another processor
                then $\Psi_b \leftarrow \Psi_b \cup \psi$
                else $\Psi_f \leftarrow \Psi_f \cup \psi$
        send Ψ_b
    RECEIVE-BACKWARD-UPDATES()
    send Ψ_f
    RECEIVE-FORWARD-UPDATES()

Figure 7.12: Parallel path construction algorithm.

RECEIVE-FORWARD-UPDATES()

$\Psi_f \leftarrow \emptyset$
receive Ψ_f
for each $\psi \in \Psi_f$
    do ADD-PATH-TO-NET$(\eta(\psi), \psi)$
$\Psi_f \leftarrow \emptyset$

RECEIVE-BACKWARD-UPDATES()

$\Psi_b \leftarrow \emptyset$
receive Ψ_b
for each $\psi \in \Psi_b$
    do ADD-PATH-TO-NET$(\eta(\psi), \psi)$
        $\Psi_f \leftarrow \Psi_f \cup \psi$
$\Psi_b \leftarrow \emptyset$

Figure 7.13: Parallel path construction algorithm (cont.).

PARALLEL TIMING DRIVEN PLACEMENT - `mpiPLACE-TIME()`

if actor on processor 0
  then read circuit and distribute cells and nets to different processors
for each processor
    do while termination not reached
        do while `TimberWolfSC` iteration not complete
            do ATTEMPT-MOVE()
                $attemptsCount++$
                if $attemptsCount \% U_p == 0$
                  then UPDATE-PINS()
                if $attemptsCount \% U_r == 0$
                  then UPDATE-ROWS()
                      FIX-ROW-DESIRES()
          adjust U_p and U_r
          UPDATE-FIXED-CELLS()
          REMOVE-OVERLAPS()
          if time to repartition
            then REPARTITION()
          PARALLEL-COMPUTE-DELAYS()
          PARALLEL-CONSTRUCT-ALL-PATHS()

Figure 7.14: Outline of the `mpiPLACE-TIME` algorithm.

Table 7.4: `mpiPLACE-TIME` Results (Sun SparcServer 1000E)

	`TimberWolfSC`	`mpiPLACE-TIME`	
		1 processor	8 processors
fract			
Wirelength	39636	42863	46038
Area	0.53	0.54	0.55
Delay (ns)	20.3	20.0	20.0
Run time(s)	74.1	118.2	26.3
struct			
Wirelength	956328	997781	1078901
Area	11.9	12.3	12.4
Delay (ns)	102.3	96.8	100.0
Run time(s)	916.6	1220	249.0
biomed			
Wirelength	5599231	5231329	5666327
Area	120.5	111.2	120.9
Delay (ns)	56.7	52.4	52.7
Run time(s)	8571	123835	20938
avq.small			
Wirelength	41194758	44035597	46312331
Area	1220	1500	1452
Delay (ns)	866.3	561.3	580.2
Run time(s)	16631	81667	14868

Table 7.5: `mpiPLACE-TIME` Results (Intel Paragon)

	`TimberWolfSC`	`mpiPLACE-TIME`	
		1 processor	16 processors
struct			
Wirelength	956328	997781	1071662
Area	11.9	12.3	12.4
Delay (ns)	102.3	96.8	101.0
Run time(s)	2557	5589	628.0
biomed			
Wirelength	-	-	5781491
Area	-	-	121.0
Delay (ns)	-	-	52.9
Run time(s)	-	-	64620
avq.small			
Wirelength	-	-	46943623
Area	-	-	1527
Delay (ns)	-	-	600.2
Run time(s)	-	-	33822

7.5 Summary

In this chapter, we described a new timing driven algorithm for placement as well as a parallel implementation based on `ProperPLACE-PART`. A new timing driven algorithm has been presented that is able to achieve significant reductions in longest path delays, while at the same time causing little area degradation. Previous approaches have used fairly inaccurate models of interconnect delay, but this work is the first to present a timing driven approach to use an Elmore delay model. The use of a more accurate model ensures that the algorithm is correctly identifying longest paths and critical nets. The parallel algorithm uses the same procedures described in Chapter 6 to achieve good speedups with moderate loss in quality.

CHAPTER 8

CONCLUSIONS

In this thesis, we have investigated algorithms for parallelizing simulated annealing based standard cell placement. The main contributions are in identifying efficient algorithms for parallelization of area driven and timing driven placement algorithms.

In the domain of area driven placement, four different parallel simulated annealing strategies were studied. The first strategy, parallel moves, delivers consistent speedups for few processors with some degradation in wire length. Multiple Markov chains appears to be promising as a means to achieve moderate speedup without losing quality and, in fact, in some cases improve quality. Speculative computation, however, is shown to be inadequate as a means of parallelization of cell placement. A combination of the parallel moves approach with intermediate exchanges as in multiple Markov chains may offer benefits in terms of reducing the error present in the parallel moves approach alone. These first three approaches are only suitable in situations where the circuit is small enough that it can be replicated on all nodes of the multiprocessor. The final approach to parallel cell placement presented in this thesis is the circuit partitioned algorithm. Through the use of various error control mechanisms, we are able to provide reasonable speedups with moderate loss of quality.

The most useful algorithms are the multiple Markov chains and circuit partitioned algorithms. Both are useful in different circumstances. For smaller circuits or in systems where memory usage is not a concern, the multiple Markov chains approach is clearly the best alternative. It provides excellent speedups with very little degradation. However, in distributed memory environments where the circuit may not fit on a single node, the circuit partitioned approach is preferred. While it does not offer the speedups seen with multiple Markov chains, it does give consistent speedups with very little degradation in quality.

The other area of research of this thesis is the investigation of timing driven placement strategies and their parallel algorithms. We have presented a sequential algorithm that uses a very detailed timing model to drive the placement process and it is able to achieve very good results. The parallel algorithm is based on the circuit partitioned algorithm for standard cell placement and achieves consistent speedups with little quality degradation.

The results of this thesis are limited to simulated annealing based placement algorithms. Recently, researchers have proposed alternate forms of placement algorithms such as simulated evolution [85, 86], force directed placement [87, 88], linear optimization based [89–91]. and min-cut based placement [92–94]. Very little work has been done in developing parallel algorithms for these alternate approaches [95–97]. New and more efficient parallel algorithms for these approaches must be investigated.

The algorithms in this thesis have been targeted for portable parallel environments such as ProperCAD II and MPI. However, algorithms targeted specifically to shared memory multiprocessors have not been addressed fully. Other possible future extensions to this thesis include parallel algorithms for the other phases of the layout process, namely, global and detailed routing.

REFERENCES

[1] S. Kim, J. A. Chandy, S. Parkes, B. Ramkumar, and P. Banerjee, "ProperPLACE: A portable parallel algorithm for cell placement," in *Proceedings of the International Parallel Processing Symposium*, Cancun, Mexico, Apr. 1994, pp. 932–941.

[2] S. Kim, "Improved algorithms for cell placement and their parallel implementations," Ph.D. dissertation, Department of Electrical and Computer Engineering, University of Illinois at Urbana-Champaign, Urbana, IL, July 1993, Tech. Rep. CRHC-93-18/UILU-ENG-93-2231.

[3] S. Parkes, J. A. Chandy, and P. Banerjee, "A library-based approach to portable, parallel, object-oriented programming: Interface, implementation, and application," in *Proceedings of Supercomputing '94*, Washington, DC, Nov. 1994, pp. 69–78.

[4] Message-Passing Interface Forum, "Document for a standard message-passing interface," University of Tennessee, Knoxville, TN, Tech. Rep. CS-93-214, 1993.

[5] P. Banerjee, *Parallel Algorithms for VLSI Computer Aided Design Applications*. Englewoods Cliffs, NJ: Prentice Hall, 1994.

[6] S. M. Parkes, "A class library approach to concurrent object-oriented programming with applications to VLSI CAD," Ph.D. dissertation, University of Illinois at Urbana-Champaign, Sept. 1994, Tech. Rep. CRHC-94-20/UILU-ENG-94-2235.

[7] B. Ramkumar and P. Banerjee, "ProperCAD: A portable object-oriented parallel environment for VLSI CAD," *IEEE Transactions on Computer-Aided Design*, vol. 13, no. 7, pp. 829–842, July 1994.

[8] S. Parkes, P. Banerjee, and J. H. Patel, "ProperHITEC: A portable, parallel, object–oriented approach to sequential test generation," in *Proceedings of the Design Automation Conference*, San Diego, CA, June 1994, pp. 717–721.

[9] S. Parkes, P. Banerjee, and J. Patel, "A parallel algorithm for fault simulation based on PROOFS," in *Proceedings of the International Conference on Computer Design*, Austin, TX, Oct. 1995.

[10] K. De, J. A. Chandy, S. Roy, S. Parkes, and P. Banerjee, "Portable parallel algorithms for logic synthesis using the MIS approach," in *Proceedings of the International Parallel Processing Symposium*, Santa Barbara, CA, Apr. 1995, pp. 579–585.

[11] S. Roy, "Parallel algorithms for algebraic factorization in logic synthesis," M.S. thesis, University of Illinois at Urbana-Champaign, May 1996, Tech. Rep. CRHC–96–07/UILU–ENG–96–2212.

[12] G. Hasteer, "Parallel algorithms for state assignment of finite state machines," M.S. thesis, University of Illinois at Urbana-Champaign, Jan. 1996, Tech. Rep. CRHC-96-02/UILU–ENG–96–2202.

[13] G. Hasteer and P. Banerjee, "A parallel algorithm for state assignment in finite state machines," in *Proceedings of the International Conference on Parallel Processing*, Bloomingdale, IL, Aug. 1996, To appear.

[14] K. MacPherson, "Parallel algorithms for layout verification," M.S. thesis, University of Illinois at Urbana-Champaign, Aug. 1995, Tech. Rep. CRHC–95–18/UILU–ENG–95–2229.

[15] K. MacPherson and P. Banerjee, "Integrating task and data parallelism in an irregular application: A case study," in *Proceedings of the IEEE Symposium on Parallel and Distributed Processing*, New Orleans, LA, Oct. 1996, To appear.

[16] V. Krishnaswamy and P. Banerjee, "Actor based parallel VHDL simulation using Time Warp," in *Proceedings of the 1996 Workshop on Parallel and Distributed Simulation*, Philadelphia, PA, May 1996.

[17] G. A. Agha, *Actors: A Model of Concurrent Computation in Distributed Systems*. Cambridge, MA: The MIT Press, 1986.

[18] A. W. Appel, *Compiling with Continuations*. Cambridge, England: Cambridge University Press, 1992.

[19] A. A. Chien, *Concurrent Aggregates: Supporting Modularity in Massively Parallel Programs*. Cambridge, MA: The MIT Press, 1993.

[20] L. V. Kalé and S. Krishnan, "CHARM++: A portable concurrent object oriented system based on C++," in *Proceedings of the Conference on Object Oriented Programming Systems, Languages and Applications*, Sept. 1993, pp. 91–108.

[21] K. M. Chandy and C. Kesselman, "Compositional C++: Compositional parallel programming," in *Proceedings of Workshop on Compilers and Languages for Parallel Computing*, 1992, pp. 79–93.

[22] V. Karamcheti and A. Chien, "Concert - Efficient runtime support for concurrent object-oriented programming languages on stock hardware," in *Proceedings of Supercomputing '93*, Portland, OR, Nov. 1993, pp. 598–607.

[23] A. Chien, V. Karamcheti, and J. Plevyak, "The Concert System: Compiler and runtime support for fine-grained concurrent object-oriented languages," University of Illinois, Department of Computer Science, Urbana, Illinois, Tech. Rep. UIUC DCS TR R-93-1815, 1993.

[24] D. Gannon and J. K. Lee, "Object-oriented parallelism: pC++ ideas and experiments," *Proceedings of the Japan Society for Parallel Processing*, pp. 315–339, 1993.

[25] M. Parashar and J. C. Browne, "Distributed dynamic data-structures for parallel adaptive mesh-refinement," in *Proceedings of International Conference on High Performance Computing*, New Delhi, India, Dec. 1995, pp. 22–23.

[26] High Performance Fortran Forum, *High Performance Fortran Language Specification, version 1.1*, 1994.

[27] C.-P. Wen, S. Chakrabarti, E. Deprit, A. Krishnamurthy, and K. Yelick, "Runtime support for portable distributed data structures," in *Workshop on Languages, Compilers and Runtime Systems for Scalable Computers*, May 1995.

[28] R. Ponnusamy, J. Saltz, and A. Choudhary, "Runtime-compilation techniques for data partitioning and communication schedule reuse," in *Proceedings of Supercomputing '93*, Portland, OR, Nov. 1993, pp. 361–370.

[29] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, "Optimization by simulated annealing," *Science*, vol. 220, no. 4598, pp. 671–680, May 1983.

[30] R. A. Rutenbar, "Simulated annealing algorithms: An overview," *IEEE Circuits & Devices*, vol. 5, no. 1, pp. 19–26, Jan. 1989.

[31] S. Kirkpatrick, "Optimization by simulated annealing: Quantitative study," *Journal of Statistical Physics*, vol. 34, pp. 975–986, 1984.

[32] E. H. L. Aarts and P. J. M. van Laarhoven, "Statistical cooling: A general approach to combinatorial optimization problems," *Philips Journal of Research*, vol. 40, no. 4, pp. 193–226, 1985.

[33] J. Lam and J.-M. Delosme, "Performance of a new annealing schedule," in *Proceedings of the Design Automation Conference*, 1988, pp. 306–311.

[34] D. R. Greening, "Simulated annealing with errors," Ph.D. dissertation, University of California at Los Angeles, 1995.

[35] C. Sechen and A. Sangiovanni-Vincentelli, "The TimberWolf placement and routing package," *IEEE Journal for Solid State Circuits*, vol. SC-20, no. 2, pp. 510–522, Apr. 1985.

[36] C. Sechen, *VLSI Placement and Global Routing Using Simulated Annealing*. VLSI, Computer Architecture and Digital Signal Processing. Boston, MA: Kluwer Academic Publishers, 1988.

[37] Yale University, *TimberWolf: Mixed Macro / Standard Cell Floorplanning Placement and Routing Package*, New Haven, CT, Sept. 1991.

[38] 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, CA, Nov. 1987, pp. 478–481.

[39] M. D. Durand, "Accuracy vs. speed in placement," *IEEE Design & Test of Computers*, pp. 8–34, June 1989.

[40] D. R. Greening, "Parallel simulated annealing techniques," *Physica*, vol. D42, pp. 293–306, 1990.

[41] S. A. Kravitz and R. A. Rutenbar, "Placement by simulated annealing on a multiprocessor," *IEEE Transactions on Computer-Aided Design*, vol. CAD-6, no. 4, pp. 534–549, July 1987.

[42] F. Darema, S. Kirkpatrick, and V. A. Norton, "Parallel algorithms for chip placement by simulated annealing," *IBM Journal of Research and Development*, vol. 31, no. 3, pp. 391–402, May 1987.

[43] K. Natarajan and S. Kirkpatrick, "Evaluation of parallel placement by simulated annealing: Part I - The decomposition approach," IBM, Tech. Rep. RC 15246, Nov. 1989.

[44] 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, CA, Nov. 1994, pp. 137–144.

[45] P. Banerjee, M. H. Jones, and J. S. Sargent, "Parallel simulated annealing algorithms for standard cell placement on hypercube multiprocessors," *IEEE Transactions on Parallel and Distributed Systems*, vol. 1, pp. 91–106, Jan. 1990.

[46] J. S. Rose, W. M. Snelgrove, and Z. G. Vranesic, "Parallel cell placement algorithms with quality equivalent to simulated annealing," *IEEE Transactions on Computer-Aided Design*, vol. 7, no. 3, pp. 387–396, Mar. 1988.

[47] 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, Nov. 1987, pp. 350–353.

[48] C.-P. Wong and R.-D. Fiebrich, "Simulated annealing-based circuit placement algorithm on the Connection Machine system," in *Proceedings of the International Conference on Computer Design*, Rye Brook, NY, Oct. 1987, pp. 78–82.

[49] R. Jayaraman and R. A. Rutenbar, "Floorplanning by annealing on a hypercube multiprocessor," in *Digest of Papers, International Conference on Computer-Aided Design*, Santa Clara, CA, Nov. 1987, pp. 346–349.

[50] W.-J. Sun and C. Sechen, "Efficient and effective placement for very large circuits," *IEEE Transactions on Computer-Aided Design*, vol. 14, no. 3, pp. 349–359, Mar. 1995.

[51] E. H. L. Aarts, F. M. J. de Bont, E. H. A. Habers, and P. J. M. van Laarhoven, "Parallel implementations of the statistical cooling algorithm," *Integration, the VLSI Journal*, vol. 4, pp. 209–238, Sept. 1986.

[52] E. H. L. Aarts and P. J. M. van Laarhoven, *Simulated Annealing: Theory and Applications*. Mathematics and Its Applications. Boston, MA: Kluwer Academic Publishers, 1987.

[53] E. H. L. Aarts and J. H. M. Korst, "Boltzmann machines as a model for parallel annealing," *Algorithmica*, vol. 6, pp. 437–465, 1991.

[54] S.-Y. Lee and K.-G. Lee, "Asynchronous communication of multiple Markov chains in parallel simulated annealing," in *Proceedings of the International Conference on Parallel Processing*, St. Charles, IL, Aug. 1992, pp. III:169–176.

[55] K.-G. Lee and S.-Y. Lee, "Efficient parallelization of simulated annealing using multiple Markov chains: An application to graph partitioning," in *Proceedings of the International Conference on Parallel Processing*, St. Charles, IL, Aug. 1992, pp. III:177–180.

[56] E. E. Witte, R. D. Chamberlain, and M. A. Franklin, "Parallel simulated annealing using speculative computation," *IEEE Transactions on Parallel and Distributed Systems*, vol. 2, no. 4, pp. 483–494, Oct. 1991.

[57] A. Sohn, "Parallel speculative computation of simulated annealing," in *Proceedings of the International Conference on Parallel Processing*, St. Charles, IL, Aug. 1994, pp. III:8–11.

[58] J. G. Holm, A. Lain, and P. Banerjee, "Compilation of scientific programs into multi-threaded and message driven computation," in *Proceedings of the Scalable High Performance Computing Conference*, Knoxville, TN, May 1994, pp. 518–525.

[59] L. A. Sanchis, "Multiple-way network partitioning," *IEEE Transactions on Computers*, vol. 38, pp. 62–81, 1989.

[60] C. M. Fiduccia and R. M. Mattheyses, "A linear-time heuristic for improving network partitiongs," in *Proceedings of the Design Automation Conference*, June 1982, pp. 175–181.

[61] S. T. Barnard and H. D. Simon, "Fast multilevel implementation of recursive spectral bisection for partitioning unstructured problems," *Concurrency: Practice and Experience*, vol. 6, no. 2, pp. 101–117, Apr. 1994.

[62] G. Karypis and V. Kumar, "A fast and high quality multilevel scheme for partitioning irregular graphs," Department of Computer Science, University of Minnesota, Minneapolis, MN, Tech. Rep. 95-035, June 1995.

[63] A. Casotto, F. Romeo, and A. Sangiovanni-Vincentelli, "A parallel simulated annealing algorithm for the placement of macro-cells," *IEEE Transactions on Computer-Aided Design*, vol. CAD-6, no. 5, pp. 838–847, Sept. 1987.

[64] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam, *PVM 3.0 User's Guide and Reference Manual*, Feb. 1993.

[65] R. Butler and E. Lusk, *User's Guide to the p4 Parallel Programming System*, Argonne, IL, June 1992.

[66] Message-Passing Interface Forum, *Message-Passing Interface Standard 1.1*, June 1995.

[67] P. Bridges, N. Doss, W. Gropp, E. Karrels, E. Lusk, and A. Skjellum, *Users' Guide to mpich, a Portable Implementation of MPI*, Sept. 1995.

[68] H. Youssef, R.-B. Lin, and E. Shragowitz, "Bounds on net delays for VLSI circuits," *IEEE Transactions on Circuits and Systems - II*, vol. 39, no. 11, pp. 815–824, Nov. 1992.

[69] T. Gao, P. M. Vaidya, and C. L. Liu, "A new performance driven placement algorithm," in *Digest of Papers, International Conference on Computer-Aided Design*, Santa Clara, CA, Nov. 1991, pp. 44–47.

[70] M. Marek-Sadowska and S. P. Lin, "Timing driven placement," in *Digest of Papers, International Conference on Computer-Aided Design*, Santa Clara, CA, Nov. 1989, pp. 94–97.

[71] W. E. Donath, R. J. Norman, B. K. Agrawal, S. E. Bello, S. Y. Han, J. M. Kurtzberg, P. Lowy, and R. I. McMillan, "Timing driven placement using complete path delays," in *Proceedings of the Design Automation Conference*, Orlando, FL, June 1990, pp. 84–89.

[72] M. A. B. Jackson, A. Srinivasan, and E. S. Kuh, "A fast algorithm for performance-driven placement," in *Digest of Papers, International Conference on Computer-Aided Design*, Santa Clara, CA, Nov. 1990, pp. 328–331.

[73] T. Hasegawa, "A new placement algorithm minimizing path delays," in *Digest of Papers, International Conference on Computer-Aided Design*, Santa Clara, CA, Nov. 1991, pp. 2052–2055.

[74] W. Swartz and C. Sechen, "Timing driven placement for large standard cell circuits," in *Proceedings of the Design Automation Conference*, San Francisco, CA, June 1995, pp. 211–215.

[75] A. Srinivasan, K. Chaudhary, and E. S. Kuh, "RITUAL: A performance-driven placement algorithm," *IEEE Transactions on Circuits and Systems - II: Analog and Digital Signal Processing*, vol. 39, no. 11, pp. 825–840, Nov. 1992.

[76] W. Swartz and C. Sechen, "New algorithms for the placement and routing of macro cells," in *Digest of Papers, International Conference on Computer-Aided Design*, Santa Clara, CA, Nov. 1990, pp. 336–339.

[77] W. C. Elmore, "The transient response of damped linear network with particular regard to wideband amplifiers," *Journal of Applied Physics*, vol. 19, pp. 55–63, 1948.

[78] K. D. Boese, A. B. Kahng, B. A. McCoy, and G. Robins, "Fidelity and near-optimality of Elmore-based routing constructions," in *Proceedings of the International Conference on Computer Design*, Cambridge, MA, Oct. 1993, pp. 81–84.

[79] C. J. Alpert, T. C. Hu, J. H. Huang, A. B. Kahng, and D. Karger, "Prim-Dijkstra tradeoffs for improved performance-driven routing tree design," *IEEE Transactions on Computer-Aided Design*, vol. 14, no. 7, pp. 890–896, July 1995.

[80] K. D. Boese, A. B. Kahng, B. A. McCoy, and G. Robins, "Near-optimal critical sink routing tree constructions," *IEEE Transactions on Computer-Aided Design*, vol. 14, no. 12, pp. 1417–1436, Dec. 1995.

[81] J. Cong, A. B. Kahng, C.-K. Koh, and C.-W. A. Tsao, "Bounded-skew clock and Steiner routing under Elmore delay," in *Digest of Papers, International Conference on Computer-Aided Design*, Santa Clara, CA, Nov. 1995.

[82] R. B. Hitchcock, Sr., "Timing verification and the timing analysis program," in *Proceedings of the Design Automation Conference*, 1982, pp. 594–604.

[83] T. I. Kirkpatrick and N. R. Clark, "PERT as an aid to logic design," *IBM Journal of Research and Development*, vol. 10, no. 2, pp. 135–141, Mar. 1966.

[84] W. P. Swartz, Jr., "Automatic layout of analog and digital mixed macro/standard cell integrated circuits," Ph.D. dissertation, Yale University, May 1993.

[85] R. M. Kling, "Optimization by simulated evolution and its application to cell placement," Ph.D. dissertation, University of Illinois at Urbana-Champaign, Aug. 1990, Tech. Rep. CRHC–90–07/UILU–ENG–90–2237.

[86] R. M. Kling and P. Banerjee, "Empirical and theoretical studies of the simulated evolution method applied to standard cell placement," *IEEE Transactions on Computer-Aided Design*, vol. 10, no. 10, pp. 1303–1315, Oct. 1991.

[87] N. R. Quinn and M. A. Breuer, "A force directed component placement procedure for printed circuit boards," *IEEE Transactions on Circuits and Systems*, pp. 377–388, June 1979.

[88] K. J. Antreich, F. M. Johannes, and F. H. Kirsch, "A new approach for solving the placement problem using force models," in *Proceedings of the International Symposium on Circuits and Systems*, 1982, pp. 481–486.

[89] J. Frankle and R. M. Karp, "Circuit placements and cost bounds by eigenvector decomposition," in *Digest of Papers, International Conference on Computer-Aided Design*, Santa Clara, CA, Nov. 1986, pp. 414–417.

[90] J. M. Kleinhans, G. Sigl, F. M. Johannes, and K. J. Antreich, "GORDIAN: VLSI placement by quadratic programming and slicing optimization," *IEEE Transactions on Computer-Aided Design*, vol. 10, no. 3, pp. 356–365, Mar. 1991.

[91] R.-S. Tsay, E. S. Kuh, and C.-P. Hsu, "Proud: A sea-of-gates placement algorithm," *IEEE Design & Test of Computers*, vol. 5, no. 6, pp. 44–56, Dec. 1988.

[92] M. A. Breuer, "Min-cut placement," *Design Automation and Fault-Tolerant Computing*, vol. I, no. 4, pp. 343–362, Oct. 1977.

[93] U. Lauther, "A min-cut placement algorithm for general cell assemblies based on a graph representation," in *Proceedings of the Design Automation Conference*, San Diego, CA, June 1979, pp. 1–10.

[94] A. E. Dunlop and B. W. Kernighan, "A procedure for placement of standard-cell VLSI circuits," *IEEE Transactions on Computer-Aided Design*, vol. CAD-4, no. 1, pp. 92–98, Jan. 1985.

[95] R. M. Kling and P. Banerjee, "Concurrent ESP: A placement algorithm for execution on distributed processors," in *Digest of Papers, International Conference on Computer-Aided Design*, Santa Clara, CA, Nov. 1987, pp. 354–357.

[96] E. I. Horvath, "A parallel force direct based VLSI standard cell placement algorithm," in *Proceedings of the International Symposium on Circuits and Systems*, Chicago, IL, May 1993, pp. III:2071–2074.

[97] Z. Xing and P. Banerjee, "A parallel hierarchical algorithm for module placement based on sparse linear equations," in *Proceedings of the International Symposium on Circuits and Systems*, Atlanta, GA, May 1996, pp. IV:691–694.

VITA

John Attupurathu Chandy received the S.B. in Electrical Engineering from the Massachusetts Institute of Technology in 1989. He then joined the Data General Corporation in Westborough, MA, as a hardware engineer where he was responsible for firmware design for open systems workstations and multiprocessor servers. In 1991, he enrolled at the University of Illinois at Urbana-Champaign, where he was supported by the Semiconductor Research Corporation as a research assistant in the Center for Reliable and High-Performance Computing. He received the M.S. degree in 1993 and will receive the Ph.D. degree in Electrical Engineering in 1996 from the University of Illinois at Urbana-Champaign. He will be joining Cadence Design Systems in Chelmsford, MA. His research interests are in parallel software and high performance I/O systems.