A PARALLEL ROW-BASED ALGORITHM WITH ERROR CONTROL FOR STANDARD-CELL PLACEMENT ON A HYPERCUBE MULTIPROCESSOR

BY

JEFF SCOTT SARGENT

B.S., University of Illinois, 1987

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Electrical Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1988

Urbana, Illinois

ABSTRACT

A new row-based parallel algorithm for standard-cell placement targeted for execution on a hypercube multiprocessor is presented. Key features of this implementation include a dynamic simulated-annealing schedule, row-partitioning of the VLSI chip image, and two novel new approaches to controlling error in parallel cell-placement algorithms: **Heuristic Cell-Coloring** and **Adaptive (Parallel Move) Sequence Control.** Heuristic Cell-Coloring identifies sets of noninteracting cells that can be moved repeatedly, and in parallel, with no buildup of error in the placement cost. Adaptive Sequence Control allows multiple parallel cell moves to take place between global cell-position "updates." This feedback mechanism is based on an error bound we derive analytically from the traditional annealing move-acceptance profile.

We present placement results for real industry circuits and summarize the performance of an implementation on the Intel iPSC/2 Hypercube. The runtime of this algorithm is 5 to 16 times faster than a previous program developed for the Hypercube, while producing equivalent quality placement. An integrated place and route program for the Intel iPSC/2 Hypercube is currently being developed around this kernel algorithm.

ACKNOWLEDGEMENTS

I wish to especially thank my advisor Professor Prithviraj Banerjee for his keen guidance and encouragement. His ability to motivate me and his other students through personal attention and enthusiasm is without equal. I would also like to thank my fiancce and family for their support and understanding. Lastly I would have finished this thesis far more quickly without the constant interruptions and distractions from the other members of the Computer Systems Group, and for this I am very grateful.

I wish to acknowledge my corporate sponsor, Bell Communications Research, without whose support this work could not have begun.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

1.1. Motivation

The processing demands of software for VLSI (Very Large Scale Integration) chip design outstrip the power of conventional desktop workstations. This demand continues to rise due to increased device density. Conventional CAD (Computer-Aided Design) software is heavily stressed by custom-design styles in the lucrative ASIC market, where design turn-around time is critical. Though the performance of microprocessors has increased dramatically in the last several years, it is clear that the demands of VLSI CAD still render conventional uniprocessor algorithms inconvenient for industry-type problems. The introduction of low-cost multiprocessors in parallel-processing ensembles provides the low cost-to-performance ratio needed in this problem domain. However research into parallel approaches to standard VLSI CAD problems is still immature, and its progress slowed by lack of proper software development tools for multiprocessors. In this thesis we present a parallel algorithm to solve one particular VLSI CAD problem: standard-cell placement.

1.2. Standard Cell Placement

The standard-cell approach is a popular semi-custom design style in which the designer chooses functional building blocks from an existing library to construct a portion of, or an entire, VLSI chip. The building blocks (called cells from now on) provide a level of abstraction to the designer, relieving him/her of the necessity of "reinventing the wheel." Furthermore, the interior details of the cells can change (to improve performance for example) without invalidating an existing design.

Standard-cell designs are typically laid-out in multiple parallel rows or columns of cells. Placed in rows, cells are usually of the same height and of variable width. Figure 1.1 illustrates a sample layout with 3 rows of standard cells. Real layouts can have thousands of cells and many rows. I/O pads and intercell wiring are omitted for clarity. The gaps between rows are flexible in height and called *channels*. Gaps between cells in the same row, which allow signals to be routed between distant rows, are called *feed-throughs*. All cell-to-cell interconnect wiring between two rows must lie in the channel between them.

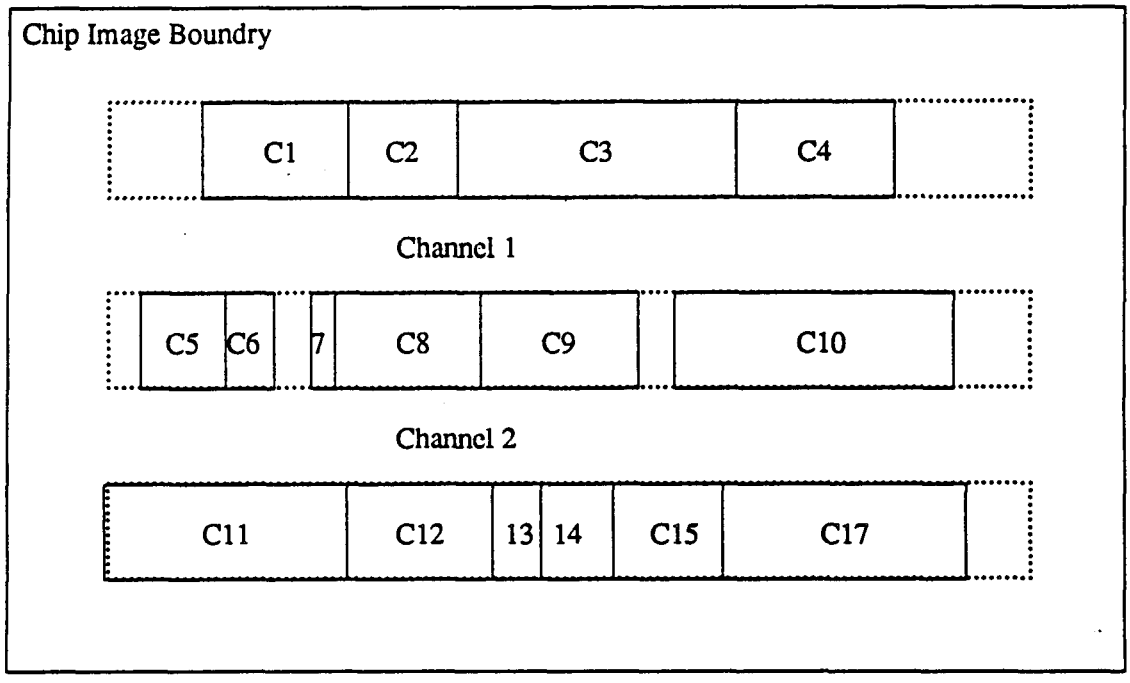


Figure 1.1. Sample Standard-Cell Layout

Given a set of standard cells and the interconnection net list, the objective of optimization is to place the cells in the rows so that the total interconnect wirelength is minimized. Sechen has shown [1] that the total wire length of a randomly-placed layout can be much larger than an optimized layout. A direct benefit of a short interconnect path is reduced signal-propagation delay, in general leading to better performance. By clustering highly-connected cells in close proximity to one another, channel heights can be reduced leading to a very compact layout. The placement of standard cells is then followed by interconnect routing. Routing assumes the cells are fixed in place, and interconnects them by laying conductive paths in the intervening channel. If the router is unable to lay a path in a channel, it merely increases the channel height to accommodate the new interconnect path. Good placement is essential for the router to operate efficiently. Ideally, place and route should occur simultaneously for optimal results, but since they are both NP-complete optimization problems, they are handled separately for simplicity.

Most approaches to placement can be categorized [2] as either **constructive** or **iterative**. Constructive methods generally take an incomplete placement (some subset of cells fixed in place) and add cells heuristically

to keep total interconnect at a minimum. Classes of constructive methods include cluster growth [3], min-cut partitioning [4, 5, 6], global methods such as quadratic assignment [7] and convex function optimization [8], and traditional branch and bound [3]. Iterative methods generally start with a complete placement and attempt to improve the configuration by generating small perturbations. Approaches that fall into this category include pairwise interchange [9], force-directed interchange [10] and unconnected sets [11]. Both the constructive and iterative methods tend to get stuck in local minima, and are thus unable to reach the global optima. Simulated annealing is an iterative approach that uses probabilistic hill climbing to avoid local optima.

1.3. Simulated Annealing

Kirkpatrick *et al.* [12] introduced *simulated annealing* in 1983 as a new technique for solving combinatorial optimization problems. Metals are annealed by first raising the temperature to its boiling point, then slowly cooling the molten mixture until frozen. By carefully controlling the cooling schedule (sequence of temperatures), the desired well-ordered low-energy crystalline structure will result. If the mixture is cooled too fast (quenched), then defects and imperfections in the crystalline lattice (high energy irregularities) will be frozen into the structure. For a constant temperature T, the system is in thermal equilibrium if the probability distribution of states S_i with corresponding energies $E(s_i)$ approximates the **Boltzmann distribution:**

$$P(s_i) = e^{-E(s_i)/k_b T}$$

with T being the absolute temperature and k_b the Boltzmann constant. Quenching occurs when the temperature is lowered before the distribution of state energies has converged to the Boltzmann distribution.

To accurately simulate physical annealing, the state of the system must be well-defined, and have an associated cost representative of the quantities to be minimized. Further, a process to generate new states from previous states must be defined that creates a rich set of perturbations. A temperature schedule sufficiently gentle to properly anneal without quenching, yet aggressive enough to limit CPU time, must be found. An outline of a generic simulated annealing algorithm is given in Figure 1.2.

```
Simulated Annealing Algorithm
 Set initial temperature T0 and state S0
 T = T0
 S = S0
 While ( Stopping Criteria Unsatisfied ) Do
   While ( Inner Loop Criteria Unsatisfied ) Do
     Generate new state S' = perturb (S)
     If CostAcceptable (Cost (S'),Cost (S))
       then S = S'.
   End While
 End While
End
```

Figure 1.2. Generic Simulated Annealing Algorithm

Metropolis [13] originally proposed a probabilistic method to determine whether a newly generated state is "acceptable." Let ΔC be the change in cost from the previous state S to the new state S'. Then the new state is acceptable if

$$P\{\text{New State Acceptable}\} = \begin{cases} 1, & \text{if } \Delta C < 0 \\ e^{-\Delta C/T}, & \text{if } \Delta C > 0 \end{cases}$$

New states with superior cost (lower cost) are accepted automatically, while those that increase cost will always have some small chance of acceptance. This "probabilistic hill-climbing" ability of simulated annealing allows escape from local minima by sometimes allowing poor (uphill) moves. As temperature falls to zero, the chance of accepting inferior permutations is reduced to zero. This function for acceptance has the consequence that the system evolves into the Boltzmann distribution.

1.4. Cell Placement and Simulated Annealing

Simulated annealing has proven to be a very successful method to optimize VLSI cell placement. The "state space" in cell-placement corresponds to all possible permutations of cell positions. The cost function is primarily aggregate wire length, though most algorithms include penalty costs so that a desired layout aspectratio is met. TimberWolf3.2 [14, 15] is a popular uniprocessor simulated-annealing algorithm that can produce near-optimal placement of standard-cell circuits. The primary deficiency of simulated annealing algorithms such as TimberWolf3.2 is the massive computation time required. Approaches to improving execution time fall into the broad areas of improved cooling schedules [16, 17], annealing hybrids [18, 19, 20], and parallel implementations [17, 21, 22, 23, 20, 24].

1.5. Thesis Outline

This thesis describes a new row-based parallel placement algorithm based on the simulated-annealing optimization technique. Chapter 2 will briefly review the fine points of the intended hardware platform, hypercube multiprocessors. Chapter 3 will review other work in parallel annealing algorithms for cell placement, including the performance results of an earlier program that inspired this new algorithm. Chapter 4 will outline the fundamentals of our row-based placement algorithm, including the area mapping, distributed data structures, communication patterns, and annealing schedule used. Chapter 5 will illustrate how error is created in parallel implementations, and then present two new methods for controlling error: **Heuristic Cell Coloring** and **Adaptive Sequence-Length Control**. The impact of error on the dynamic annealing schedule will be discussed. Finally we present results of an implementation on the Intel iPSC/2 Hypercube in Chapter 6. Our integrated error control coupled with dynamic scheduling yields an order of magnitude improvement in execution time over a previous parallel algorithm on the same machine, while preserving the final solution quality.

CHAPTER 2

HYPERCUBE MULTIPROCESSORS

The design and manufacture of VLSI "chips" have reached a level of manufacturing economy so high that the CPU chip (Central Processing Unit) is one of the cheapest components of a microcomputer - less expensive than the power supply, case and keyboard, hard disk drive, or miscellaneous chips. Interconnecting large numbers of relatively low-performance, low-cost microprocessors offers a cost-effective hardware platform with performance approaching a mainframe computer. This potential is often hard to reach because of the difficulties in writing parallel algorithms for such multiprocessors, assuming the problem is possible to decompose to begin with. The most popular interconnection topology for large-scale microprocessor ensembles to date is the hypercube [25].

The hypercube, or binary n-cube is a multiprocessor interconnection topology characterized completely by the hypercube "dimension," D. There are 2^D processors in a D-dimensional hypercube, and each processor is directly connected along a communication link to D other processors. If the 2^D processors are addressed from 0 to 2^D-1, then those processors with only one bit-position difference in their addresses are directly connected to one another. Figure 2.1 illustrates 8 processors interconnected in binary n-cube fashion, resulting in a three-dimensional hypercube. Processor addresses are represented in binary.

The hypercube topology offers a rich set of interprocessor connections, yet the number of physical "ports" at a processor only increases with the logarithm (base 2) of the number of processors. Also any two processors are separated by at most D communication link "hops."

The Mark I "Cosmic Cube" developed at the California Institute of Technology was the first hypercube built in this country [25]. Borrowing heavily from this academic effort, Intel introduced their Personal Super Computer (iPSC) line in about 1985. Each processing "node" in the iPSC contained an 8 MHz Intel 80286 microprocessor and associated 80287 floating-point co-processor, and 512Kbytes of RAM. The "extended memory" option allowed the substitution of 4.5 Mbyte RAM cards for half the processing nodes, resulting in a hypercube of half the dimension but a substantial amount of memory. Each node has eight 82586 Ethernet transceiver chips - seven for node-node communication in a seven-dimensional hypercube, and the eighth for a global channel shared by the other nodes and the cube manager "host." Internode message routing is implemented

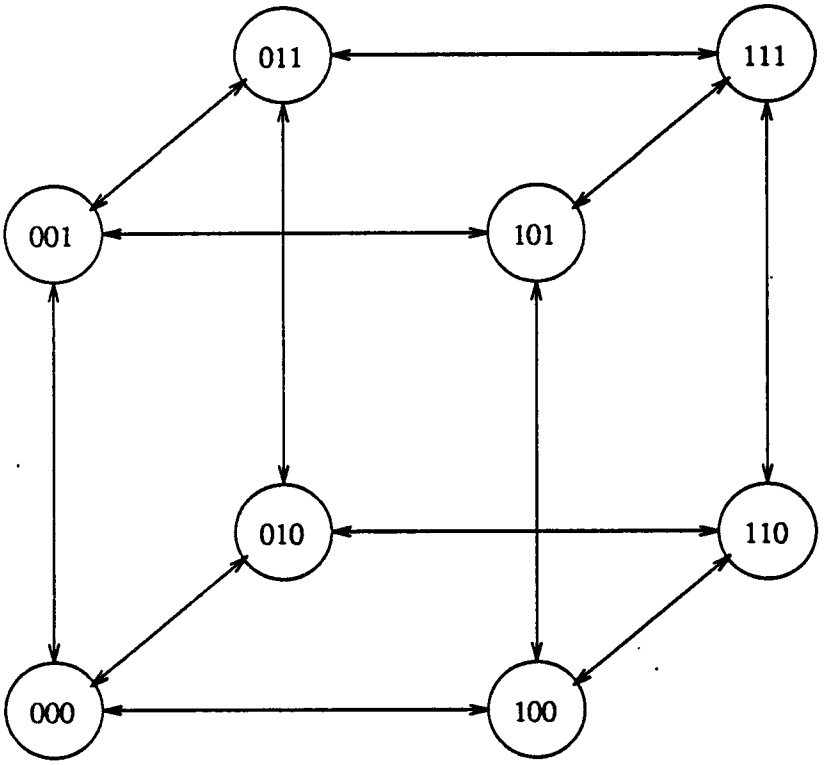


Figure 2.1. Three-Dimensional Hypercube

in a store-and-forward packet-switching technique so latency is proportional to the number of link hops traveled. The start-up cost of message transmission is very high (1.7 msec) making short messages inefficient with respect to the link bandwidth of 2.8 Mbits/sec. All message buffering and routing is handled in software, so that very little overlap in communication and computation can occur.

Recently Intel has released a new version, the iPSC/2, that has a new CPU and completely different message-routing hardware. In this new machine the 80286/80287 has been upgraded to the 80386/80387 processor pair for an increase in computational performance of about 4-6 times. Instead of store-and-forward packet switching, the new hardware implements circuit-switched techniques that greatly reduce latency for multihop messages. Single-hop message latency has been greatly reduced (< 400 μsec) thus improving the efficiency of short messages. Of a more practical nature, the new hardware and software combination allows multiple users to run their applications simultaneously on smaller "sub-cubes" allocated from the whole hyper-

cube.

Jones and Banerjee's placement algorithm was ported to both the iPSC/1 and iPSC/2 hypercubes here at Illinois; performance data were measured to determine the runtime characteristics of the algorithm. (A summary of these measurements can be found in Appendix A.) Though initial development began on the iPSC/1, **this new row-based algorithm was designed to take full advantage of the iPSC/2 architecture.** In particular, the circuit-switched nature of the interconnection network places a smaller penalty on nonlocal message traffic, which leads to improved flexibility in the design of the inter-node communication pattern. (See Section 4.6.)

CHAPTER 3

REVIEW OF RELATED WORK

Because of the tremendous amount of material published on simulated annealing, this review will only highlight new work that is particularly relevant to this thesis.

3.1. Uniprocessor Simulated Annealing Algorithms

The latest version of the TimberWolf program, TimberWolfSC version 4.1 [15], improves on the original program TimberWolf3.2 in several ways. The annealing temperature schedule is truncated so that only low-temperature annealing takes place. The initial acceptance rate is adjusted to 50% with appropriate cost scaling. In an attempt to reduce the number of rejected moves, only short-distance moves are generated in what is called "neighborhood relaxation." Furthermore, cell exchanges are proposed over cell displacements if possible. These improvements reduce the execution time substantially over that of TimberWolf3.2, while maintaining the same quality final placement.

To reduce overall execution time , Grover [18] clips the typical annealing schedule by starting at a cool temperature instead of a very hot "boiling" temperature. A good initial placement is first generated with a traditional min-cut algorithm [6] , and then annealed at a temperature so low that only a small fraction of the moves are accepted. This low temperature annealing does not perturb placement substantially but instead improves the initial configuration by about 10%. Runtime is improved because the min-cut algorithm produces a fairly good initial placement much faster than the annealing algorithm can at high temperatures. Grover [19] has implemented another approach that uses "approximate calculations" in cell position to avoid the high cost of exact cost calculations in simulated annealing. Error is introduced to the cost calculations because cells are shifted slightly to accommodate accepted cell moves into an already crowded neighborhood. He suggests that the magnitude of this error can approach the temperature and the solution will still converge to good final placement. Approximate calculations yield overall speedup of 3 to 5 times over the same algorithm with exact calculations.

Huang *et al.*[16] have developed an adaptive scheduling methodology independent of any particular annealing application. This statistical approach installed in a version of TimberWolf produced a savings in runtime of between 15 and 57% compared to that for the nonadaptive schedule, and resulted in equivalent quality

solutions. This general schedule was also applied to the traveling-salesman problem with good results. Because of the potential reduction in CPU time offered by a dynamic schedule, we have adapted Huang's schedule to our parallel algorithm.

3.2. Parallel Simulated Annealing Algorithms

Several groups have implemented parallel versions of simulated annealing. Parallel in the context of annealing can take two forms - functional parallelism or data parallelism. Functional parallelism provides limited speedup by using multiple processors to evaluate different phases of a single move. Data parallelism consists of proposing and evaluating moves independently at different processors (or groups of processors). Obviously the two forms can be mixed as well. Data parallelism has the advantage of easily scaling the algorithm to large ensembles of processors.

Jones and Banerjee [22] developed a parallel algorithm based on TimberWolf for the Intel iPSC/1 Hypercube. In this algorithm multiple cell moves are proposed and evaluated in parallel by pairs of processors. This early effort laid the foundation for our new row-based algorithm. Performance results of Jones and Banerjee's algorithm will be discussed in detail at the end of this chapter.

In a manner similar to Grover [18], Rose *et al.*[20] replace high-temperature annealing with a partitioning method called heuristic spanning that assigns cells to fixed sub-areas of the chip. These sub-areas are then annealed independently on separate processors via "section annealing." By risking possible nonoptimal placement they save considerable execution time.

Kravitz and Rutenbar [23] suggest that a hybrid approach is appropriate to deal with the dynamic characteristics of parallel annealing. At high temperature, functional partitioning of major annealing tasks improves performance while at low temperatures parallel moves may be proposed in parallel with little contention due to the high move-rejection rate. They present a tentative criterion to determine at what point their algorithm should switch from function to data partitioning.

Two groups, Casotto and A. Sangiovanni-Vincentelli [21] and Wong and Fiebrich [24], have developed algorithms for cell placement on the massively-parallel Connection Machine.

3.3. Key Performance Results of Jones and Banerjee

Jones and Banerjee's algorithm for the Intel Hypercube produced better quality placement than TimberWolf3.2 and was predicted to also have superior performance. At the time of publication, results were obtained from an implementation on a hypercube simulator because a physical hypercube was not available. Since that time the program has been ported to the Intel iPSC/1 and iPSC/2 and the performance discussed [26]. The absolute performance of the program was not as fast as expected. No program profiling tools are available for the iPSC/1 or /2 so detailed timings of important computational and communicational functions were measured *in situ* to determine execution bottlenecks. These performance measurements are included in Appendix A.

These measurements pointed out a significant bottleneck. A single parallel move can be decomposed into four primary steps: move proposal and evaluation, node-to-node message traffic, synchronizing broadcast and cell-position update. The timings for these four steps are listed below for an 800-cell circuit on a four-dimensional hypercube (iPSC/2).

Table 3.1. Individual Step Timings on iPSC/2

Step	Time(msec)
Move Evaluation	32.7
Node-Node Traffic	8.1
Broadcast	7.9
Update	161.1

It was clear the single sub-function dominating execution time was the *cell-position update* routine. This routine's high CPU cost is due to the structure and nature of the distributed-data structure it manipulates. We predicted that hand optimizing this routine, perhaps coding in assembly language, would only lead to a factor of 3 to 5 speed improvement over the original.

Instead we concentrated on developing a method to avoid this update step. Earlier, Jones and Banerjee [27] had reported the results of a uniprocessor placement algorithm that would update after multiple moves. They obtained the best results when 16 moves were made between updates. The rationale behind this behavior was that the misinformation accumulation allowed more uphill moves be accepted, thereby avoiding local minima. Rose *et al.* also allowed their parallel algorithm to perform multiple moves between updates. Experimentally they found that up to 10 parallel moves could take place between updates.

We have developed techniques to modulate update rate dynamically to achieve maximum performance without sacrificing placement quality.

CHAPTER 4

ROW-BASED PLACEMENT ALGORITHM

4.1. Motivation

The primary design goal of this algorithm was to provide high performance in terms of final placement quality and overall execution time. A secondary motivating factor was to implement the parallelism in a manner that would be convenient for a combined place and route tool, i.e., this placement program will be a subfunction in an integrated place and route program. Lastly we wanted to provide implicit error control as a means of improving performance without sacrificing solution quality.

4.2. Overview of Parallel Algorithm

This parallel algorithm is a parallel adaptation of the annealing methodology implemented in the TimberWolf [14] placement and routing package, with some important improvements. TimberWolf is a uniprocessor cell-placement optimization program that employs single-cell displacements, orientation "flips," and cell-pair exchanges in an effort to minimize aggregate interconnect wire length. The TimberWolf simulated annealing temperature schedule is fixed irregardless of circuit characteristics, and the number of attempted new states per temperature is a constant proportional to the size of the circuit (number of cells). (The latest release of TimberWolf, TimberWolfSC version 4.1, still has a fixed temperature schedule, but the number of attempted states increases slightly at low temperature). For simplicity our algorithm employs only single-cell displacements and cell-pair exchanges. Instead of a fixed annealing temperature schedule, a dynamic schedule very similar to that reported by Huang [16] is used, allowing considerable savings in overall execution time. The temperature decrement is controlled adaptively by the perceived variation in the cost of candidate placement states at the previous temperature. The temperature is reduced only after thermal equilibrium has been reached, i.e., the probability distribution of candidate states approximates the Boltzmann distribution. Since equilibrium detection is dynamic, the number of new attempted states per temperature varies across temperatures. Even with the overhead involved in adaptive scheduling, we find an improvement in overall runtime performance.

The basic theme of our parallel algorithm is to divide up the area of the VLSI chip image into equally-sized sub-areas, and allocate each sub-area to a separate processor. For a standard-cell circuit in row organization, each processor would be allocated one or more rows of chip image. Cells are assigned an initial position but are free to migrate across the layout. The cell's associated data structure passes from processor to processor as it crosses sub-area boundaries during placement evolution. Processors pair up to evaluate single move types (displacements or exchanges), drawing potential cells from the sub-areas assigned to both processors. On a P processor hypercube, a total of $P/2$ moves are evaluated in parallel at each "parallel move." The $P/2$ moves are independent - the geographic partition guarantees that a cell cannot be moved by more than one processor simultaneously. A brief algorithmic outline of the parallel algorithm is presented in Figure 4.1.

The Host assembles the initial placement configuration completely at random. The overhead of Host-to-Node and Node-to-Host communication at the onset and termination of the program is negligible compared to overall execution time. Determination of initial temperature, frozen condition, and the dynamic temperature schedule is explained in Section 4.9. A move sequence is a series of parallel moves made without updating cell-position data structures between moves. This updating is an expensive procedure - so maximal length

```
Node_Program
{
  Receive initial placement and annealing parameters from Host.
  Determine initial temperature T0.

  While placement not "frozen" do
  {
    For i=1 to SequenceLength do
    {
      If current color exhausted, then switch colors.
      Evaluate cell move with Neighbor processor.
      If move successful, record event in MoveQueue.
    }
    Broadcast all cell moves from MoveQueue to all processors.
    Parse move sequence - add to knowledge of cost distribution.
    Adjust SequenceLength based on perceived average error.
    If at thermal equilibrium for this temperature,
     then reduce temperature.
  }

  Send optimized cell-placement configuration to Host.
}
```

Figure 4.1. Parallel Placement Algorithm

sequences are desirable. Excessively long sequences produce error that will impair convergence. Chapter 5 discusses the control of sequence length via error sampling. All processors maintain an identical current "color." Cell coloring influences the choice of cells selected for moves - only cells of the current color are allowed to move. This eliminates all temporary error due to interacting cell moves. Coloring as an error-control mechanism is discussed in Chapter 5, and the coloring method itself is outlined in Section 5.2. Pairs of processors cooperate to perform cell moves. Detailed descriptions of possible move types are in Section 4.7.

4.3. Mapping Chip Area to Processors

Jones and Banerjee's cell-placement program written for a hypercube multiprocessor [22] partitioned the chip area into square blocks, or a grid. This new algorithm instead incorporates row, or "strip" partitioning. Symbolic examples of the two chip partitioning strategies are illustrated below - Figure 4.2a illustrates strip partitioning, while figure 4.2b illustrates grid partitioning. Four rows of standard cells have been mapped to four processors.

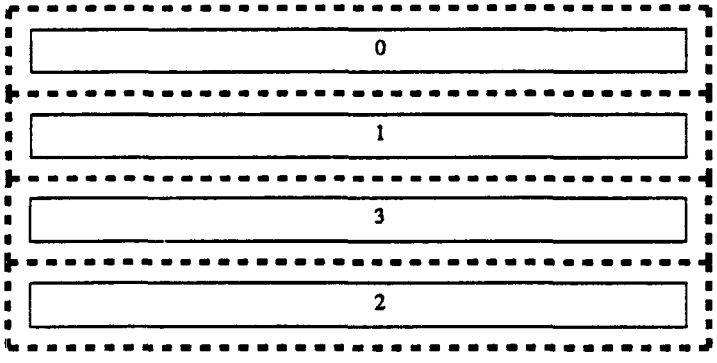


Figure 4.2a. Strip Partitioning of Rows to Processors

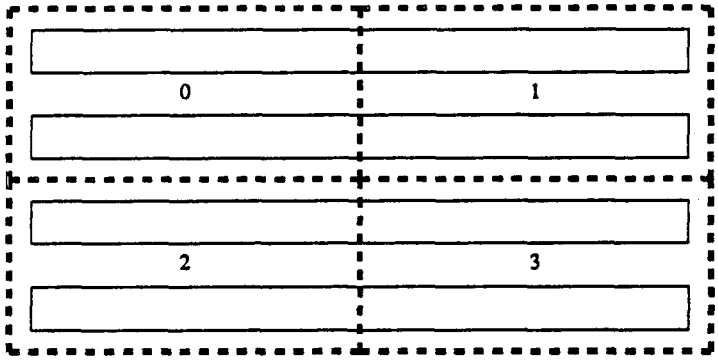


Figure 4.2b. Grid Partitioning of Rows to Processors

Strip partitioning cannot achieve as fine data granularity as square partitioning, because the sub-area corresponding to a row of cells can be allocated to only one processor. If the number of processors P in the hypercube assigned to the problem is larger than the number of rows R in the circuit, the remaining $P-R$ processors are drones - they do no work. If R exceeds P, then multiple physically adjacent rows are assigned to individual processors, i.e., $\left\lceil \frac{R}{P} \right\rceil$ rows are assigned to each processor.

The primary benefit of strip partitioning is that the entire cost of any move type can be computed solely on the basis of local information, along with its partner node. With grid partitioning, it was necessary to maintain information on the placement of cells at a node's east and west neighbors to assess the two penalty components of the cost function (see Section 4.5). The overhead of maintaining this "neighborhood" placement information can be very high. Furthermore, this neighborhood placement information must be exchanged between east-west neighbors after every parallel move to maintain state cost accuracy. By nature, strip partitioning has all such neighborhood placement information, eliminating this costly exchange. Most importantly, with exact knowledge of the two penalty cost components at each processor, the *only error that accumulates due to parallel moves is error in total wire length.* This partial elimination of error via strip partitioning not only reduces the overall magnitude of error (especially at high temperature) but also simplifies the theoretical justification behind our

methods of error control. Lastly, strip partitioning lends itself naturally to an integrated place and route strategy. Pairs of (logically adjacent) processors cooperating on moves have all the required cell-position knowledge to route the intervening channel simultaneously.

4.4. Distributed Data Structure

Each processor maintains a list of cells currently assigned to this processor. This list is implemented as a linked list of cell structures, where the structures contain net-list information necessary to compute the bounding-box portion of the cost function. Though timewise inefficient, this is the most memory-efficient method of storing the circuit description in a distributed manner. Cell structure information includes

1) The unique global cell ID#
2) The width of the cell
3) The x,y location at which the centroid of the cell is currently placed
4) A list of nets to which this cell is connected
5) For each net listed in 4), a list of other cells to which the net is connected, along with the x,y location(s) within these cells.

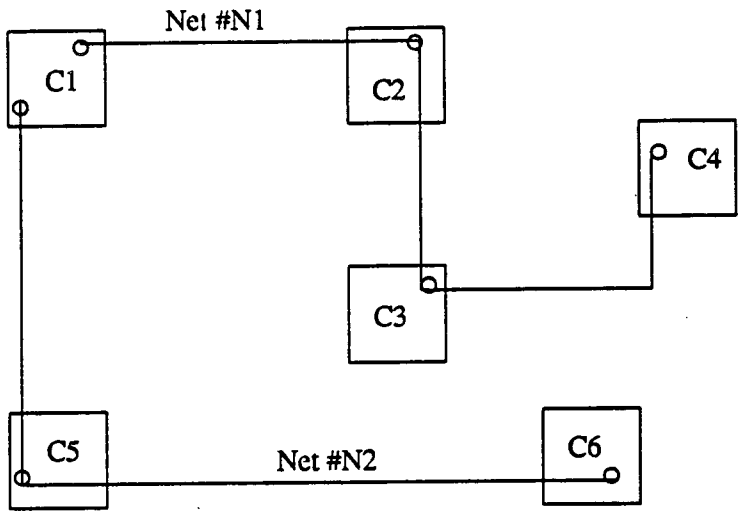
Figure 4.3 shows an example of several cells interconnected via nets, and the corresponding cell-structures as they would appear in a list at the processor. These cell structures are transferred between processors as cell-moves are made.

4.5. Cost Function

The cost function used in this algorithm is identical to that used in early versions of TimberWolf and several other simulated-annealing based cell-placement programs. The cost of a candidate placement is composed of three subcosts:

1] Estimate wiring length using half the perimeter of the bounding box rule.
2] Overshoot or undershoot of row lengths compared to ideal length (Penalty Cost).
3] Area overlap of cells in the same row (Penalty Cost).

At high temperatures cells are allowed to overshoot the maximum row length boundary, and can overlap one another. Ideally the penalty costs of 2] and 3] reduce to zero at the termination of annealing. Each processor



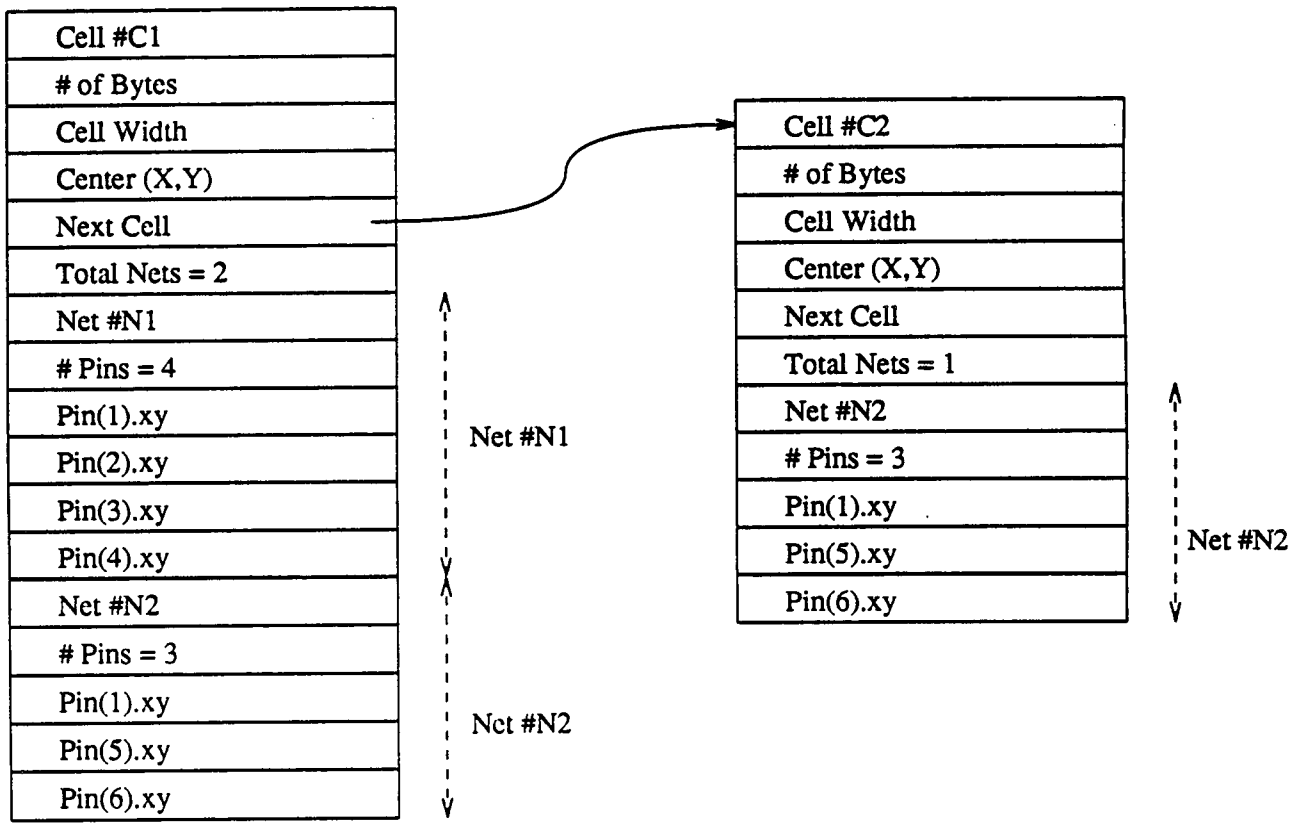


Figure 4.3. Example Cell-Cell Connectivity and Data Structure

can compute the exact cost of cells placed within its chip image sub-area. The exact cost of the current placement configuration for the entire chip is simply the sum of partial costs computed at each processor.

4.6. Interprocessor Communication Pattern

Processors pair up to evaluate a single move type - either a displacement or an exchange. Several inter-node messages are usually exchanged in the course of move resolution (see next section). Because minimizing inter-node message latency in this phase of the algorithm is very important to overall performance, the choice of node-pairings must made carefully. Uniprocessor algorithms can choose candidate cells from among the entire cell complement and can then displace them to any location within the confines of the chip image. Parallel annealing as implemented in this algorithm restricts the choice of cells to the rows mapped to the processor(s) performing the move.

The interprocessor communication pattern should satisfy two conflicting objectives: 1) short, medium, and long distance moves allowed to simulate the geographic uniformity of uniprocessor algorithms, and 2) processors chosen as neighbor pairs should communicate efficiently, i.e., the number of link "hops" necessary to communicate from processor to processor should be low. The algorithm developed by Jones and Banerjee [22] satisfied objective 2) by only allowing nearest-neighbor (1 link-hop) communication. With their square-grid chip partition, the communication pattern was a variant of the 5-point stencil [28]. If the algorithm was run on a hypercube of dimension 4 or above the higher-order links performed long-distance moves, satisfying objective 1).

We propose an improved neighbor mapping based on *hierarchical gray codes* [29] that satisfies the processor proximity goal while providing a more geographically uniform move selection. If physically consecutive rows are assigned to logically consecutive processors from a gray-code sequence, all physically adjacent rows will be mapped to adjacent processors in the hypercube. A mapping corresponding to a hierarchical gray-code sequence has the additional property that two processors with node numbers P_i and $P_{i\pm 2j}$ will be separated by at most two link hops for $0 < j < P/2$. An example of this mapping and the pairwise communication that is possible with such a mapping is shown in Figure 4.4. In the figure, a six-row chip image is mapped to a three-dimensional hypercube arranged in a broken ring arrangement with the hierarchical gray code. Arrows originating in the row allocated to node #2 indicate pairwise move resolution with other nodes $\pm 2j$, $0<j<2$.

Processors without rows (e.g. processors #4 and #5 do not take an active part in the algorithm, so the move performed with node #4 would be strictly internal to node #2. Given a layout with R rows executing on a hypercube with P processors where $P > R$, each processor will pair up with $2\log_2 R - 1$ other processors in this pattern. This provides geographic uniformity for hypercubes of any size.

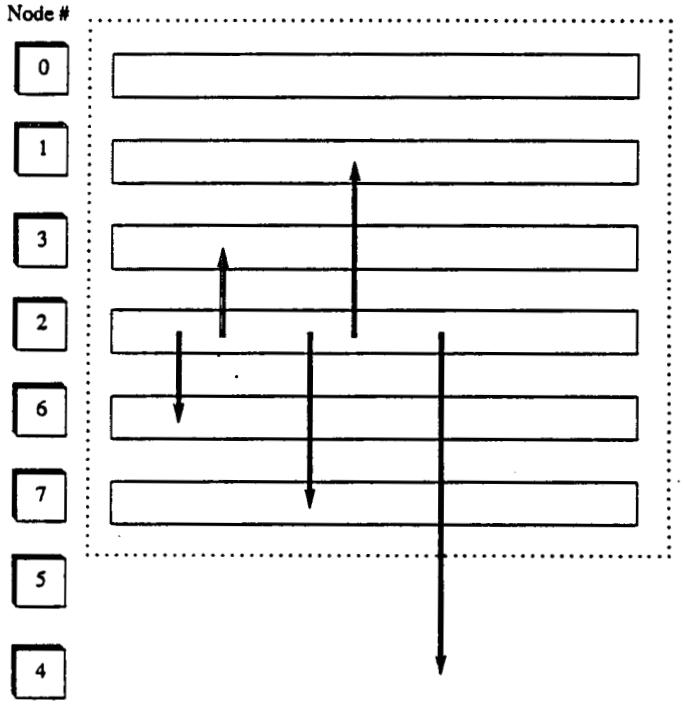


Figure 4.4. Processor-to-Processor Communication Pattern

4.7. Cell-Move Resolution

Pairs of processors cooperate to perform two kinds of cell moves - cell displacements and pairwise move exchanges. The ratio of displacements to exchanges is maintained at approximately 5:1 as used by TimberWolf. One processor assumes the role of master, the other of slave. The relative master/slave relationship between any two processors alternates in time to avoid load imbalance in cell complements. The master determines the type

of move that will be made and informs the slave. Theoretically, two processors cooperating can perform a cell move in half the time of a single processor; however, the precedence of computational steps in resolving a move does not always allow master and slave to operate concurrently. There are two sub-classes of moves for both displacements and exchanges, or four move types in total:

[1] Intraprocessor Cell Displacement

[2] Intraprocessor Cell Exchange

[3] Interprocessor Cell Displacement

[4] Interprocessor Cell Exchange

In an intraprocessor move, the master displaces or exchanges cells strictly within its own sub-area - the slave essentially is dormant. In an interprocessor move, the master either displaces a cell to the slave's sub-area, or the master and slave exchange cells. In the case of interprocessor cell movement the change in cost is evaluated partially at the master and partially at the slave. The ratio of interprocessor to intraprocessor moves is 1:1. We now present a brief algorithmic outline of each move type.

4.7.1. Intraprocessor cell displacement

The master displaces a single cell to another location within its allocated chip area. The candidate location is chosen randomly from within a range-limiting rectangle centered upon the cell's current location. The slave initially sends a cell structure packet which the master discards without inspection.

Table 4.1. Outline of Intraprocessor Cell Displacement

Master	Slave
Send null cell to slave.	Select cell at random. Send cell to master.
Receive cell from slave (discard). Select cell at random. Select random location. Compute displacement cost. If displacement cost acceptable, then update position, switch rows, and add move to move_queue.	Receive cell from master. (Determines move type)

4.7.2. Intraprocessor cell exchange

The master selects two candidate cells and exchanges their positions. Change in cost is calculated entirely by the master, as is the decision to accept the move. If the bounding box created by the two candidate cells exceeds the range-limiting window in either dimension, the exchange is rejected.

Table 4.2. Outline of Intraprocessor Cell Exchange

Master	Slave
Send null cell to slave.	Select cell at random. Send cell to master.
Receive cell from slave. (discard)	Receive cell from master. (Determines move type)
Select a cell at random. Select a second cell at random. Compute exchange cost of the two cells. If exchange cost acceptable, then modify cell list, and add move to MoveQueue.	

4.7.3. Interprocessor cell displacement

The master selects the candidate cell, computes the effect of its loss, and sends a copy of the cell to the slave. The slave picks a new location from the area created by the intersection of the slave's sub-area and the range-limiting box centered on the cell's previous location. Accounting for the master's loss, the slave computes the total cost of accepting the move, and decides accordingly.

Table 4.3. Outline of Interprocessor Cell Displacement

Master	Slave
Select cell at random.	Select cell at random. Send cell to master.
Receive cell from slave (discard). Compute displacement cost.	
Send cell to slave.	Receive cell from master. (Determines move type) Select random location for cell. If displacement cost acceptable, then update position, add cell. Add move to move queue. Inform Master of decision.
Receive move acceptance decision. If move accepted, remove cell from local area.	

4.7.4. Interprocessor cell exchange

Master and slave both select random cells, and both compute partial exchange costs. The slave informs the master of its partial cost-change calculations and the master then makes the decision to accept the move. As in intraprocessor exchanges, the move is rejected outright if the two cells are too far apart.

Table 4.4. Outline of Interprocessor Cell Exchange

Master	Slave
Select a cell at random.	Select cell at random.
Send cell to slave.	Send cell to master.
Receive cell from slave.	Receive cell from master.
Compute partial exchange cost P1.	Compute partial exchange cost P2.
	Send partial cost P2 to master.
Receive partial P2 cost from slave.	
If aggregate (P1+P2) cost acceptable,	
then modify cell list.	
Add move to move_queue.	
Inform Slave of decision.	Receive acceptance decision.
	If move was accepted,
	then update local placement.

4.8. Summary of Internode Traffic

Figure 4.5 summarizes the interprocessor message traffic that occurs in each move type. Message precedence is indicated by a number next to the arc.

4.9. Move Range Limiter

This algorithm incorporates a range-limiting window similar to TimberWolf3.2 to enhance convergence at the later stages of annealing. Displacement move types always restrict the destination within this box, and exchange move types are always rejected if the cells exceed either dimension in distance. The dimensions of this window are controlled by the following formula:

$$Width = 2*max(3, ChipWidth \cdot \frac{\log_{10}(T/5)}{\log_{10}(T_0/5)})$$

$$Height = 2*max(3, ChipHeight \cdot \frac{\log_{10}(T/5)}{\log_{10}(T_0/5)})$$

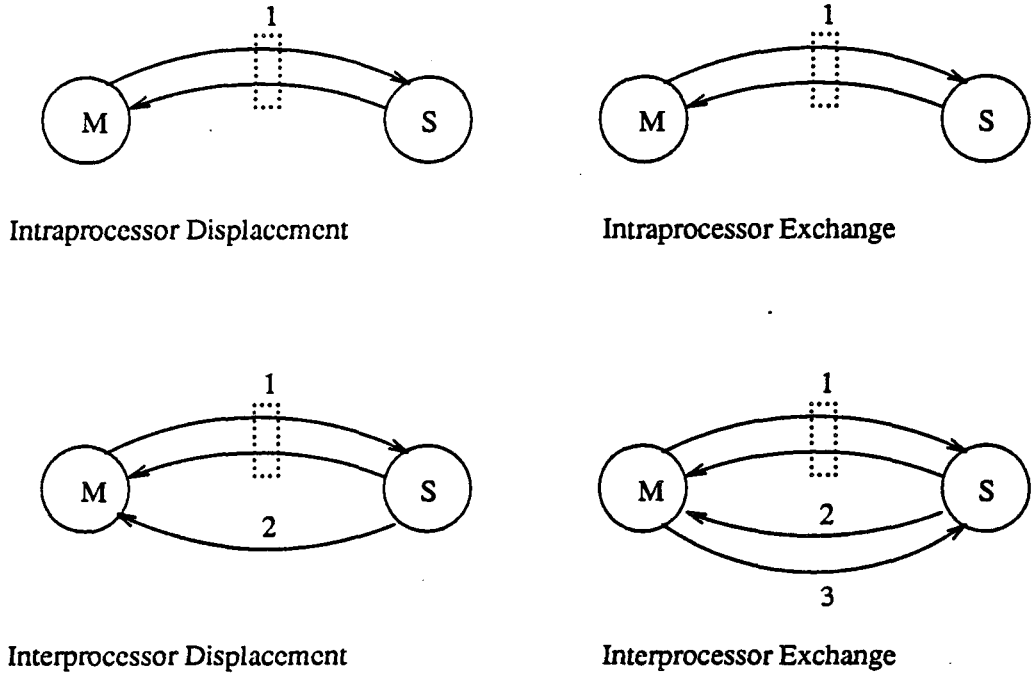


Figure 4.5. Interprocessor Message Traffic

As the window contracts within the boundaries of the layout, processor pairs that communicated previously to evaluate moves will now be separated by a distance exceeding the limiter. These processors will no longer pair-up. At the later stages of annealing, the range-limiting window only allows processors assigned to physically adjacent rows to pair-up (a consequence of gray-code mapping), thereby improving performance slightly over the general hierarchical gray-code mapping used at high temperature.

4.10. Annealing Schedule

The *annealing schedule* is the sequence of temperatures by which the placement problem is boiled, cooled and finally frozen. Most implementations of simulated annealing use a fixed sequence of temperatures derived empirically [12, 30, 15, 14, 27, 23, 21, 24, 18, 22]. Huang [16] has proposed an adaptive cooling schedule based on the characteristics of the cost distribution and the annealing curve itself (average cost vs. $\log_{10}$ temperature). Installation of Huang's schedule in a version of TimberWolf yielded substantial savings in overall execution

time with no significant change in final placement quality - for the small sample of test circuits listed. The overhead of adaptive scheduling is not large for a uniprocessor algorithm such as TimberWolf, but can be very substantial in an algorithm like ours that proposes cell moves in parallel. The system cost would have to be sampled after each accepted move to build an accurate cost distribution, requiring global synchronization and updating after every parallel move. This is exactly what we want to avoid in our new algorithm. However to validate Huang's schedule for parallel annealing we have implemented a version that only allows one parallel move between updates. The performance of this algorithm is discussed in Chapter 6.

In all the placement via annealing implementations mentioned above the number of attempted cell-moves per temperature is a linear function of the circuit size, and remains constant throughout the entire annealing schedule. TimberWolfSC4.2 increases the number of attempts at low temperature to compensate for the high move rejection rate. Huang instead relies on dynamic "equilibrium detection" to signal the appropriate point to lower the annealing temperature. The overhead in equilibrium detection is slight, and results in far fewer move attempts at high temperature than at low temperature, thereby reducing total CPU time. We have adapted Huang's schedule to our parallel algorithm. In addition to the aforementioned reduction in total move attempts, detection of equilibrium implicitly considers the impact of error on the the cost distribution, providing natural error control. This benefit is considered in the next chapter. The initial temperature, temperature decrement, equilibrium detection, and frozen condition will be discussed in the next four sections.

4.10.1. Initial temperature

Initial placement is "scrambled" randomly to determine the maximum possible variance in the placement cost distribution under a 100% move acceptance rate. To scramble the circuit, all randomly selected moves are accepted until the variance in the cost distribution stabilizes. The so-called **Hot Condition** [16] is reached when

$$\frac{|\sigma_{c_i} - \sigma_{c_0}|}{\sigma_{c_0}} < 0.03 \quad i=1...K$$

After each parallel move, the new state cost is sampled, and added to existing knowledge of the overall cost distribution. The standard deviation has stabilized when K successive samples of the standard deviation σ_{c_i} in sequence differ from the initial standard deviation of the sequence σ_{c_0} by 3% or less.

Having measured this maximal variation, temperature is set proportional to the standard deviation so that "bad" moves ($+3\sigma_c$) are initially accepted with high (75%) probability, i.e.,

$$0.75 = e^{+3\sigma_c/T}$$

therefore,

$$T_0 = k \cdot \sigma_c$$

Typically k is near 20 [16] , but as the temperature decline is very sharp at high temperatures the value of k is not critical.

4.10.2. Equilibrium detection

Equilibrium at a temperature means the probability distribution of placement configurations has reached a steady state. At high temperature this distribution will be approximately normal [16,31]. Huang suggests that equilibrium can be detected by sampling costs dynamically as the placement is perturbed. The ratio of cost samples within a closed interval about the mean cost to the total number of samples will reach a steady-state value if the system is at thermal equilibrium. For a normal distribution, this ratio is well-defined and easily computed via the error function *erf(x)*. Huang chose a small target interval $\delta = \pm\sigma/2$ as representative. For a standard normal distribution,

$$P\left\{C < \mu_c - \frac{\delta}{\sigma} \text{ and } \qquad C > \mu_c + \frac{\delta}{\sigma}\right\} = 0.38.$$

From this fraction a target count value and maximum count tolerance are established:

$$TargetCount = K(0.38)(Total\#of Cells)$$

$$MaxLimit = K(1-0.38)(Total\#of Cells)$$

where $K \approx 3$. While constructing the cost distribution, a counter J_1 increments each time a new sample cost lies within the target interval mentioned above. Another counter J_2 increments if the sample lies outside the target interval. Should J_1 reach the *TargetCount* before J_2 reaches the *MaxLimit*, equilibrium detection is signaled. Otherwise both counters are reset and counting resumes. To account for the multiple cell moves made in parallel, we weight the counter increment by the number of moves accepted at that parallel move. In our parallel implementation, the maximum number of new configurations generated is bounded from above by the same static limit used in a previous parallel implementation [22], and in TimberWolf3.2 [14].

4.10.3. Temperature decrement

The derivation of the function for temperature decrement from Huang [16] is too lengthy to repeat here. The idea is to reduce the current temperature so that the expected decrease in average cost is less than the standard deviation in the cost distribution at this temperature. To avoid sharp reductions at high temperatures, the new temperature is bounded from below by one-half the previous temperature, or

$$T_{i+1} = max\,(T_i e^{-\lambda T_i/\sigma}, T/2)$$

where λ is typically $\simeq 0.7$.

4.10.4. Frozen condition

A simple procedure detects the frozen condition. If average placement cost is unchanged for several consecutive temperatures, placement is essentially "frozen" and annealing terminates. Formally,

$$\frac{|\,C_1 - C_i\,|}{C_i} \leq 0.01 \quad i=1...4$$

if the average placement cost during four consecutive temperatures remains within 1% of the first measured cost in the sequence, the placement is "frozen."

CHAPTER 5

INTEGRATED ERROR CONTROL

5.1. Controlling Error

In algorithms such as ours that move multiple cells independently and simultaneously, error is the difference between the *real* change in cost from initial to final configurations and the *estimated* change in cost equal to the sum of locally perceived changes in cost at each processor. If C_i is the exact cost of the initial configuration, C_f the exact cost of the new configuration, and ΔC_j the perceived change in cost computed locally at the $1 \leq j \leq P/2$ processor pairs that evaluated moves in parallel, then

$$C_i + \sum_{j=1}^{P/2} \Delta C_j = C_f + Error \ .$$

When not written in the sum as above, consider ΔC_j to be the perceived cost change at an arbitrary processor pair. The ΔC_j of an unaccepted cell move is zero. Unaccepted moves may have also experienced error during evaluation, but this quantity is impossible to measure directly, so average error is only sampled from accepted moves. Clearly, error is due to inaccurate ΔC_j costs computed locally when evaluating potential moves. Of the three components making up the cost function, only estimated wire length contains error. The penalty components - cell overlap and row over/undershoot are computed accurately as a consequence of the strip-partitioning in row-based placement. We now examine a cause of error.

Consider the following typical move scenario. Figure 5.1 shows a net bounding box *BB* defined by extremally located pins in the cells named $C1$, $C2$ and $C3$. Because the three cells are in different rows, we can assume their associated data structures are located at three different processors.

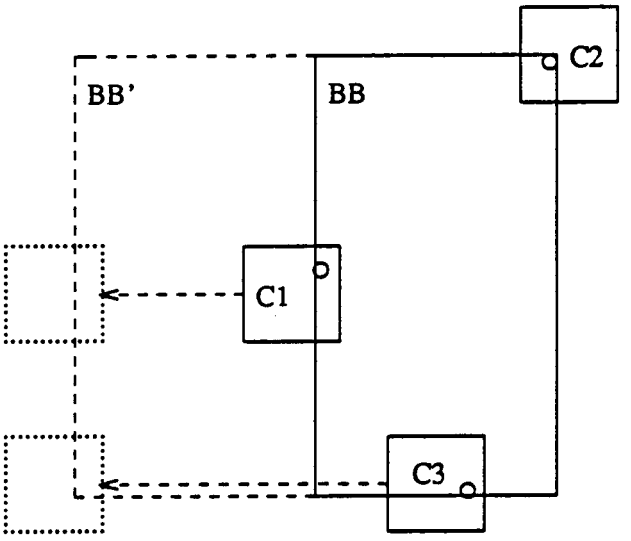


Figure 5.1. Independent Cell Displacements Altering Bounding Box

Suppose $C2$ and $C3$ are displaced independently (by different processors) in a move set to new positions indicated by the dotted boxes. The new dashed bounding box BB' results if either or both moves is accepted. Each cell has an associated data structure containing pin positions of all other cells that share at least one net with this original cell. The ΔC_j computed at both processors would account for the increase in bounding-box dimension from BB to BB' for all cells on that net. Both processors would compute a positive ΔC_j resulting from this new larger bounding box, but the sum of these partial cost changes is actually double the real change in cost. The magnitude of error would be on the same order as the change in bounding-box wire length. Thus error may be produced when cells that share nets are moved simultaneously, and both moves alter the dimension of one or more shared net bounding boxes. Obviously, the chance of error occurring increases with the number of cells moved simultaneously. At high temperatures, cells can move across the entire chip in a single displacement, whereas at low temperatures, the displacement is limited by the range-limiting window discussed previously. Long-distance moves create greater distortion in bounding-box dimensions, and thus create more error on the average. Though this entire scenario dealt with single-cell displacements, a similar argument can be made for the error created by simultaneous cell exchanges.

In summary, the error in ΔC_j due to distributed cell-location inconsistency for one parallel move is proportional to: 1) the number of cells in the parallel move set; 2) the extent to which these cells share common

nets; and 3) the distances ΔX and ΔY that the cells may be moved constrained by the range limiter. In his comparison of several parallel placement algorithms, Durand [32] classifies this error in parallel move evaluation "temporary error," the implied procedure to resynchronize and update distributed cell positioning after each parallel move. Large amounts of temporary error do not seem to impair convergence to a "good" solution. Even algorithms that move three-quarters of the total number of cells in one parallel move converge successfully [21,24].

Of more interest is the error accumulated after several parallel moves with no intervening cell position update. Clearly, the amount of misinformation will increase with each accepted move. However if this expensive synchronizing update could be reduced in frequency, overall execution time would be significantly improved, especially at low temperature. We call such a series of parallel moves a **parallel move sequence**, or simply a sequence. A move sequence is illustrated schematically in Figure 5.2. The initial placement configuration is S0, the final placement configuration SN. The exact costs of states S0 and SN are known because the algorithm synchronizes and updates at the start of every sequence. The exact costs of states S1 through S(N-1) are not known. However, a pseudocost for state S1 can be computed by adding the total perceived parallel move cost change (sum of ΔC_j's) to the original cost C_0. In this manner, pseudocosts are determined for states S2, S3 up to S(N-1). The pseudocosts will become increasingly inaccurate further along the sequence as a result of increasing error.

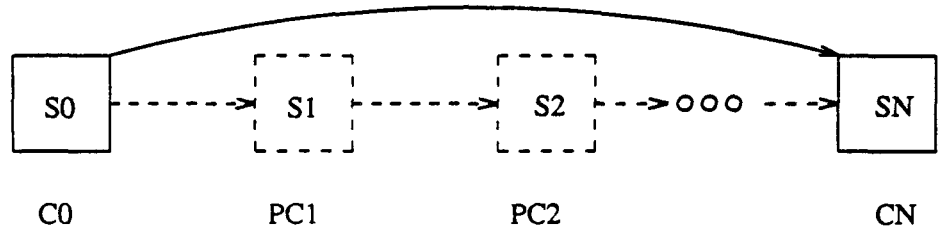


Figure 5.2. Sequence of States Permuted with Parallel Moves

Of course the total perceived cost change between pseudostates cannot be determined without synchronization, so pseudocosts are not computed until after synchronization at the end of the sequence. These pseudocosts are used as cost samples to build a cost distribution. In Huang's uniprocessor adaptive schedule, exact cost after every move is known (no error) implicitly, and an exact cost distribution can be built. What we are

creating is a cost distribution with error - the theoretical consequences of which will be discussed in Section 5.4.

The error of a sequence of length N is similar to the case for just one parallel move.

$$C_i + \sum_{k=1}^{N} \sum_{j=1}^{P/2} C_{kj} = C_f + Error.$$

Again average error is just the total error divided by the number of accepted moves. As the sequence length grows, so does the amount of misinformation on cell positioning. Hence, average error increases. Durand calls this error due to outdated information "cumulative error." While convergence is largely insensitive to temporary error, several groups report that cumulative error can impair or disable convergence entirely [20, 33, 19, 27]. Grover's algorithm with *approximate calculations* [19] restricts the error to a magnitude less than the current temperature, and maintains convergence. Due to the nature of our distributed data structure, we cannot provide rough analytic bounds with which to determine sequence length as he has.

Our placement algorithm guarantees that the error in system cost because of outdated cell-position information is due only to incorrect wire-length assessment. This benefit of strip partitioning not only reduces the magnitude of error, but also allows us to analyze the effect of error more closely. We have developed two new approaches for controlling error: **Heuristic Cell Coloring** to eliminate temporary error; and **Adaptive Sequence-Length Control** to constrain cumulative error.

5.2. Heuristic Cell Coloring

We have implemented an efficient circuit preprocessing algorithm called **Heuristic Cell Coloring**, which completely eliminates temporary error in our parallel placement algorithm by identifying sets of noninteracting cells. Noninteracting cells can be moved repeatedly, and in parallel, without any accumulation of cell position misinformation in the distributed database. Casotto [21] calls such a move-set *independent moves*. If each cell were only moved once between global cell-position updates (with coloring), then that set of moves would be a *Serializeable Subset* as defined by Kravitz and Rutenbar [30]. This cell-coloring method is extensible to other parallel-placement algorithms.

Finding sets of unconnected nodes in an arbitrary graph is analogous to graph coloring. Clearly the circuit description of a standard-cell circuit can be directly posed as a graph where cells correspond to nodes and nets correspond to edges. The graph is colored so that no two connected vertices are the same color. Now all ver-

tices (cells) of the same color are noninteracting, and can be moved repeatedly between updates without any error accumulation! Though optimal graphcoloring for arbitrary graphs is NP-complete, fast heuristic graph-coloring methods are available for graphs that are not "pathological cases." We will now outline a standard cell-coloring algorithm that can be used to heuristically color standard-cell circuits.

The graph-coloring problem, like optimal standard-cell placement, is NP-complete. Fortunately, heuristic algorithms for near-optimal graph coloring produce colorings sufficient to benefit parallel placement algorithms, in much less time than placement. The emphasis of this project was not to find an algorithm that produced the best coloring (fewest colors), but rather to determine what benefit approximate coloring can provide. Turner [34] suggests that most graphs are "easy" to color in j colors where $j \approx k$, the chromatic number of the graph. The chromatic number k is the minimum number of colors needed to color the graph. He further suggests that graphs that are extremely hard to color are pathological cases. We conjecture that standard-cell circuit descriptions are not pathological cases, and thus are easy to color in near-optimal number of colors. Graphs that exhibit low maximum connectivity (e.g., lack of very large cliques) are easier to color in k colors than those graphs with high maximum connectivity. The connectivity structure of several example circuits will be examined briefly, and its relation to overall colorability noted.

In keeping with Turner's suggestion, three variations of an algorithm originally proposed by Brelaz [35, 34] were developed to study the merits of different levels of heuristics in coloring graphs of typical circuits. In what follows, vertices are equivalent to cells and edges correspond to nets interconnecting cells. The "original" algorithm can be paraphrased with a vertex selection rule:

- **Select an uncolored vertex x from the heap of uncolored vertices such that the number of potential colors available to x is a minimum. If several vertices match this criterion, select that vertex with maximum degree in the as yet uncolored subgraph. Color this vertex x with the minimum color available.**

An outline of the original Brelaz algorithm is listed in Figure 5.3.

```
Brelaz()
{
  for all w ∈ V {                              ;For all cells
    w.color = NULL                             ;Set to uncolored
    w.avail = { Colors numbered 1 through |V| }  ;All colors available
    w.deg   = |w.neighbors|
  }
  M = makeheap(V)
  sortheap(M)                                  ;As per previous rule
  while heap_not_empty(M) {
    x = delete_heap_min(M)                     ;Pop top of heap
    x.color = min_avail_color(x.avail)         ;Smallest available color
    for z ∈ x.neighbors {                      ;For all connected cells
      if z.color == NULL {                     ;Reduce possible color
        z.avail ← z.avail - x.color            ;set.
        z.deg = z.deg - 1                      ;Reduce degree in
      }                                        ;uncolored subgraph
    }
    siftup(M)                                  ;Maintain heap order
  }
}
```

Figure 5.3. Brelaz' Graph-Coloring Algorithm

The original Brelaz algorithm will run in $O(m\log n)$ time (n vertices and m edges) if the set function routines (color availability set) are coded carefully with balanced binary-tree data structures [34]. The current implementation was not coded in this manner. The original Brelaz algorithm was the slowest of the three variations.

Variation no. 1, called "random," is identical to the original except the vertex heap is never sorted. Delete_heap_min then essentially removes a random element from the heap of remaining vertices. This variation is faster because sorting the heap is very expensive.

Variation no. 2, "greedy," is identical to the original, but no heap sorting is performed after the first sort. This orders the vertices in the heap by degree - those vertices with high degree will be removed first. This is similar to the "standard" greedy graph coloring algorithm. The execution time of this algorithm is somewhere between "original" and "random"[34].

The three graph-coloring algorithms were coded in the form of a program in the "C" Language. Input files for the program were cell-description files in TimberWolf format. (Actually TimberWolf format had to be converted slightly since it contained information superfluous to the coloring program.) Resulting colorings of several circuits are listed in Table 5.1. The runtime for the coloring algorithm on all circuits was under 10

minutes, with the exception of the 800-cell circuit, which took about an hour with the original Brelaz algorithm. This satisfies the constraint that coloring should take much less time than placement.

Table 5.1. K-Colorability of Sample Standard-Cell Circuits

Number of Cells	Connectivity			Colors Needed		
	Min	Max	Avg	Brelaz	Random	Greedy
32	3	10	6	6	6	6
64	3	16	6	11	11	11
183	1	23	8	15	15	15
286	1	123	40	29	24	30
800	1	123	22	21	24	24

The random and greedy algorithms performed as well (or better) than the original algorithms, except for the 800-cell circuit, where the original algorithm did slightly better. The 286-cell circuit had a very high average connectivity, and required a correspondingly high number of colors. Unfortunately no optimal coloring numbers were available for these circuits.

Of more interest is the distribution of the color-set sizes. For example, the 800-cell circuit colored by the original Brelaz algorithm can be colored in 21 distinct hues. The average color set would then include 800/21 = 38 cells. The actual distribution of set sizes for this circuit is plotted in Figure 5.4.

Casual inspection of the graph will show that the size of most of the color sets falls below the 38-cell average, with several large sets encompassing about 50% of the total cells. These large cell sets offer the greatest potential in terms of parallel cell moves, but those cells in the smaller sets must be allowed to move as well. By limiting the current color to a small color set, the overall acceptance rate drops with respect to the noncolored algorithm.

All three algorithm variations produced nearly the same set-size distribution. Certainly other algorithms could probably be developed that produce a more even distribution. It is doubtful that other algorithms could produce a substantially better coloring.

5.3. Adaptive Sequence-Length Control

We now describe a method used to control cumulative error. Our parallel algorithm dynamically extends and contracts the parallel move sequence length constrained by a bound on allowable error derived from temperature. Other researchers [20, 22, 23] have reported that fixed length sequences can be used and still maintain

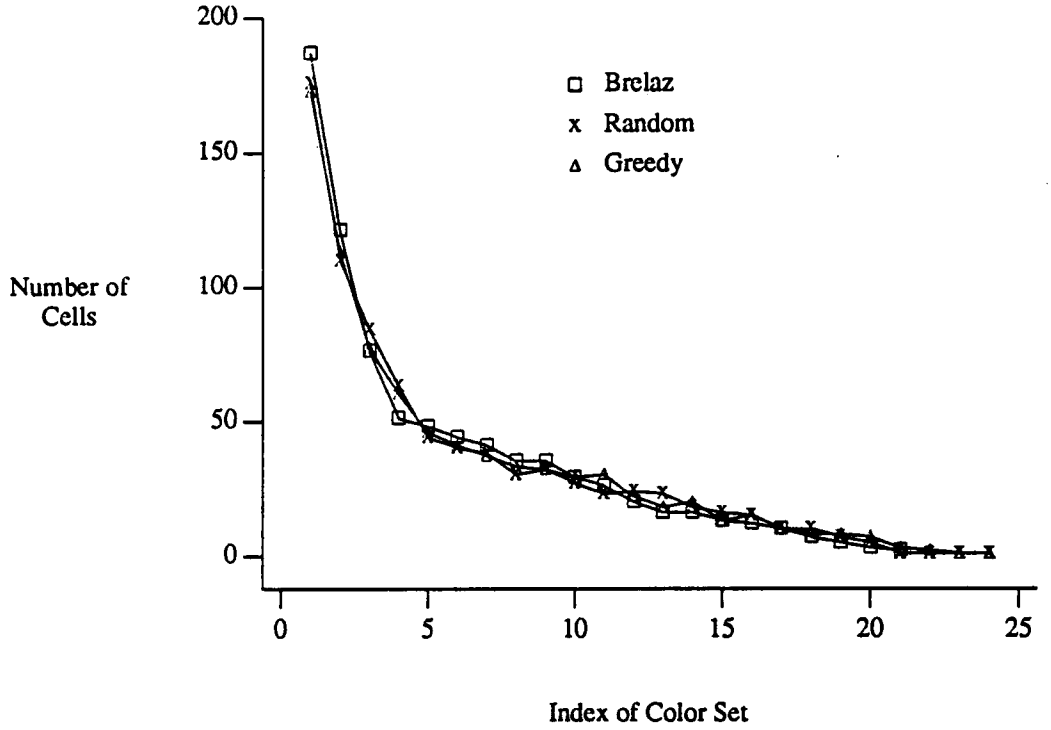


Figure 5.4. Distribution of Color-Set Sizes

convergence. Fixed sequence lengths are inappropriate due to the varying conditions found for different circuits at different temperatures. Maximal size sequence lengths are desirable because they limit the frequency of cell position updating thereby improving performance significantly.

5.3.1. Bounding error with temperature

We wish to find an upper bound on the maximum permissible average error at a particular temperature. By adjusting sequence length dynamically after each sequence, the average error can be limited to a specific range. We base our approach on the characteristics of the move-acceptance curve. The composite move-acceptance-rate curve is nearly continuous and similar to the so-called "annealing curve" of cost versus logarithm of temperature. We call the acceptance rate composite because it has two components:

$$P = P\{\textit{move accepted}\}$$

$$= P\{\textit{move accepted} \mid \Delta C > 0\} \cdot P\{\Delta C > 0\} + P\{\textit{move accepted} \mid \Delta C < 0\} \cdot P\{\Delta C < 0\}$$

where ΔC is the change in cost such a move would produce. The procedure developed by Metropolis [13] rewrites the probability of accepting a "good" move ($\Delta C < 0$) as unity and that of a "bad" move as ($e^{-\Delta C/T}$). Hence,

$$P = e^{-\Delta C/T} \cdot P\{\Delta C > 0\} + P\{\Delta C < 0\}$$

In the presence of error the composite acceptance rate changes slightly; however, the probability of generating good or bad moves is invariant with respect to error :

$$P_E = e^{-(\Delta C \pm E)/T} \cdot P\{\Delta C > 0\} + P\{\Delta C < 0\}$$

Our approach then is to bound the magnitude of allowable error so that the "normal" composite acceptance rate is not unduly affected. In similar fashion, TimberWolfSC version 4.1 [15] dynamically scales the acceptance rate of bad moves thereby forcing the composite rate to follow an empirically derived optimal curve. To bound the acceptance rate with error P_E to within 5% of normal, i.e.,

$$\frac{P - P_E}{P} \leq 0.05$$

we find two bounds on magnitude of error - one for pessimistic error E_+ and one for optimistic error E_-:

$$E_+ \leq -T \,/\, \ln(1-0.05) \approx T/20$$

$$E_- \leq -T \,/\, \ln(1+0.05) \approx T/21$$

The tighter bound (optimistic error) is used in the algorithm. If the average error measured at the end of a sequence is higher than $T/21$, the sequence length is reduced commensurate with that excess. If average error is very low ($T/42$) then sequence length is lengthened slowly. Plots of average sequence length vs. temperature are included in the next chapter. Experimentally we found that a 5% deviation in composite acceptance maintained convergence of all our test circuits - it is not clear what maximum variation in acceptance rate can be tolerated and still insure convergence.

5.4. Theoretical Justification

Our conjecture is that error constrained to wire length with our row-based algorithm will assume a normal distribution under a parallel move set. Furthermore the distribution of system cost with error will also retain normal form validating Huang's schedule [16] for our parallel move set.

We have demonstrated how error is created by evaluating interacting moves in parallel. Outdated cell-position information produces error that is proportional to the number of nets connected to the cell and the distance it was displaced from its original location. The error can be decomposed into X and Y components:

$$E_x = \alpha \cdot W_x$$

$$E_y = \beta \cdot W_y$$

where E_x is a random variable representing the system-wide error in the X-dimension at all nodes, and W_x is a random variable representing error in the bounding box. Now E_x is proportional to the W_x because all the cells are chosen independently without regard to connectivity. Therefore $E_x = \sum_{i=1}^{N} \alpha_i W_{x_i}$ will assume a normal distribution as the sum of independent uniformly random variables. The same argument can be made for E_y. Now the X and Y components of displacements and exchanges are completely independent. Therefore, the error components E_x and E_y can be combined into a single normal random variable E with mean,

$$\mu_E = \mu_{E_x} + \mu_{E_y} \ ,$$

and variance,

$$\sigma_E^2 = \sigma_{E_x}^2 + \sigma_{E_y}^2 \ .$$

Hence error introduced by a parallel move set is normal in distribution.

By treating simulated annealing as a Markov process Hajek [31] and others have shown that the cost distribution will assume a normal distribution at high temperature. Indeed Huang's statistical scheduling approach is based on this assumption. We have just shown that error in a parallel move set will assume a normal distribution. Assuming that the normal random variable error is independent with respect to the normal random variable cost, the sum of the two will assume a new distribution that is also normal:

$$\mu_{C_E} = \mu_C + \mu_E$$

$$\sigma^2_{\hat{c}_s} = \sigma^2_{\hat{c}} + \sigma^2_{\hat{\varepsilon}} \ .$$

Taking this procedure one step further, a sequence of parallel moves is equivalent to one very large move set except that cells may be moved more than during a sequence. Though cell moves are no longer independent, we conjecture that the average error experienced in a sequence of moves will also follow a normal distribution, with some minor dependency effects.

CHAPTER 6

IMPLEMENTATION AND RESULTS

6.1. Implementation

The performance of this row-based placement algorithm is at least 5 to 20 times as fast as a previous parallel algorithm for the hypercube [22]. We have implemented this algorithm on the Intel iPSC/1 and /2 Hypercubes, in about 6,000 lines of "C" language code. Three variants of the algorithm were developed with different levels of error control. The simplest algorithm, FIXEDSEQ, fixed the sequence length at one, and did not use heuristic cell coloring. This version basically verified that Huang's annealing schedule could be adapted to a parallel annealing algorithm. The second algorithm CELLCOL is identical to FIXEDSEQ, with the addition of heuristic cell coloring. We will present the impact of coloring on convergence and the annealing schedule. Lastly the third algorithm ADAPTIVE incorporates dynamic sequence-length adjustment, but does not include heuristic cell coloring. ADAPTIVE has performance an order of magnitude better than that for Jones' algorithm, while maintaining equivalent quality placement. In the future this program will form the core of an integrated VLSI placement and routing package tailored for the iPSC/2 Hypercube. We will first discuss general results of our error control methodologies, then the placement quality produced by all three algorithms, and finally the run-time performance.

6.2. FIXEDSEQ Algorithm

We stated that strip partitioning would eliminate error due to cell overlaps and edge overshoot/undershoot, and that error would be produced only in wire length. Figure 6.1 plots the average temporary error of a 64-cell test circuit for both the new row-based algorithm and the previous grid-based approach [22]. A four-processor hypercube was used in both cases, and the sequence length was limited to one. Clearly by eliminating error in cell overlap and row-over/undershoot the new algorithm experiences far less temporary error.

Several properties of the FIXEDSEQ algorithm are worth noting. Figure 6.2 is a plot of placement cost vs. temperature for a circuit run on a uniprocessor and a sixteen-processor hypercube. At high temperature the parallel algorithm appears to find some high-energy states not accessible to the uniprocessor algorithm. This

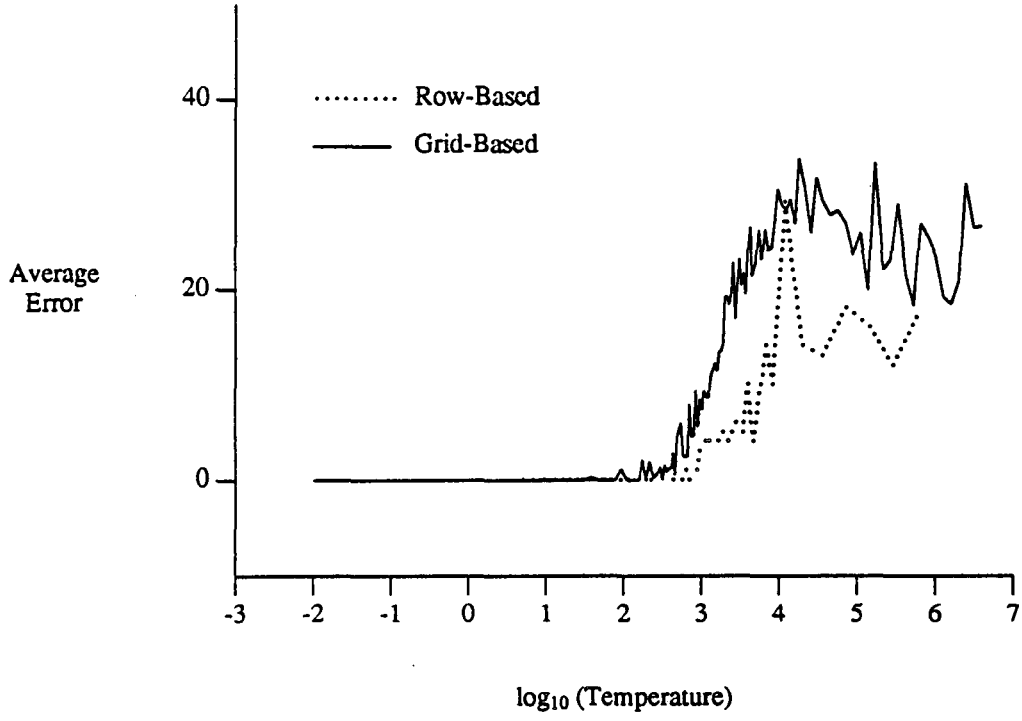


Figure 6.1. Average Temporary Error in Strip and Grid Partitioning

graphically depicts the parallel algorithm's ability to resist local minima in the presence of error, an idea first attributable to Jones and Banerjee [22]. At low temperature the cost curves converge as is expected because the behavior of the parallel algorithm converges to that of the uniprocessor under a high move-rejection rate. With this fuller exploration of the configuration space comes a corresponding higher variance in cost distribution. The variance of the placement cost vs. temperature is plotted in Figure 6.3 for the same circuit and hypercube configurations as above. Likewise the behavior of the parallel algorithm approaches that of the uniprocessor algorithm at low temperature. With the FIXEDSEQ algorithm we have shown that Huang's adaptive schedule is appropriate for our parallel annealing algorithm. Though FIXEDSEQ produces good quality results, the runtime is excessive due in part to the global synchronization that occurs after every parallel move (i.e., sequence length is one).

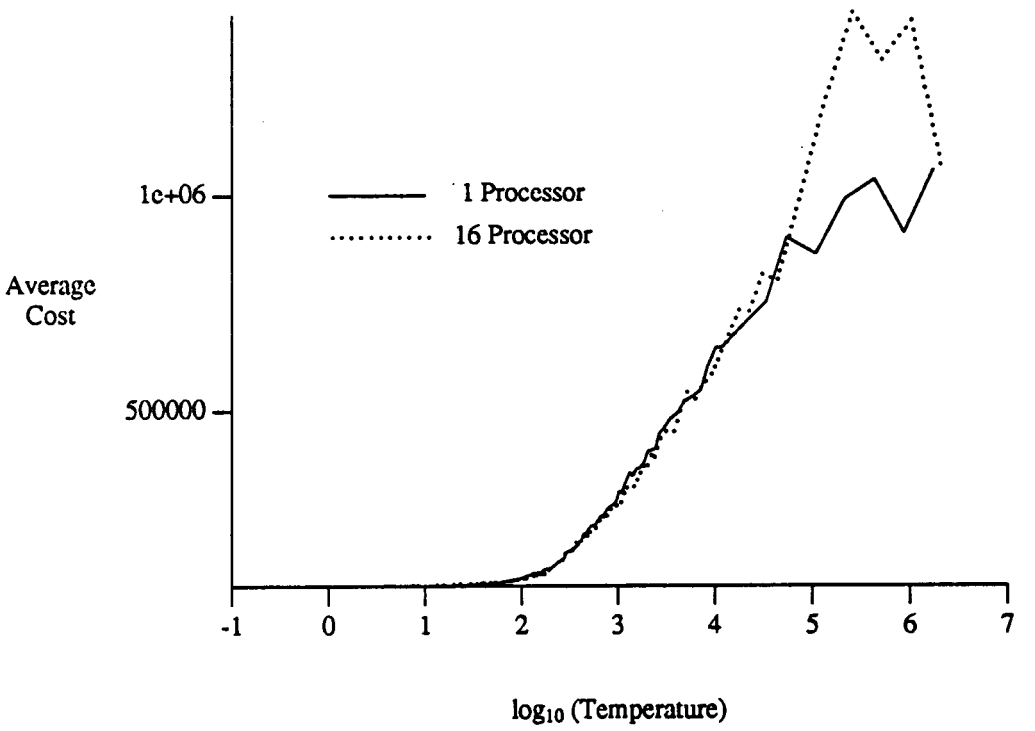


Figure 6.2. Average Cost vs. Temperature for 1 and 16 Processor Hypercubes

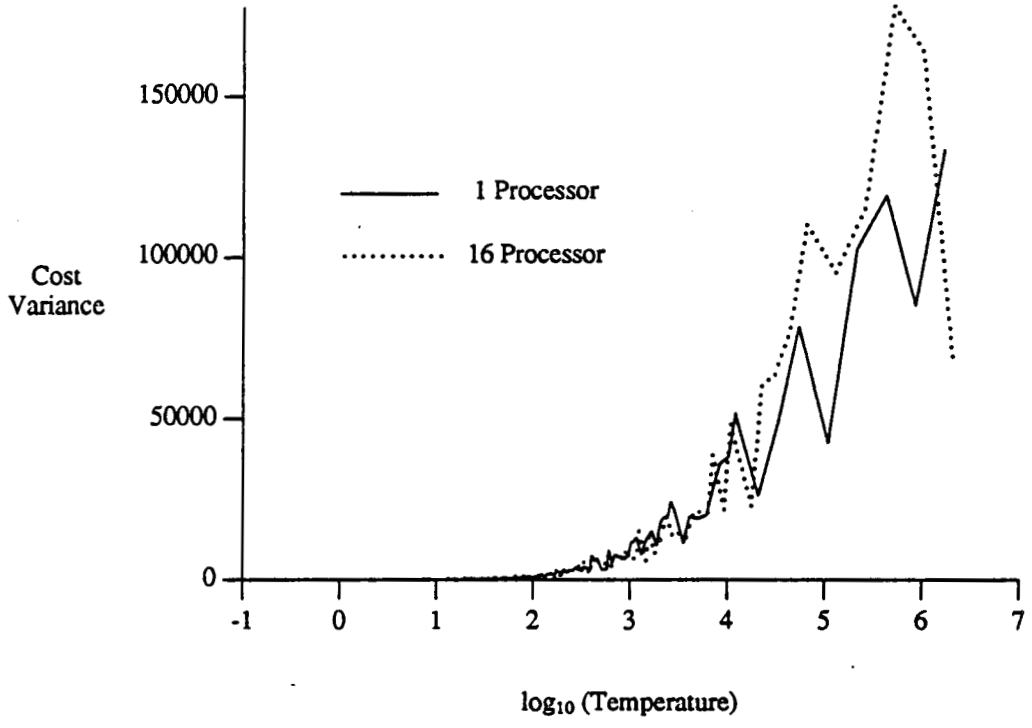


Figure 6.3. Cost Variance vs. Temperature for 1 and 16 Processor Hypercubes

6.3. CELLCOL Algorithm

The CELLCOL algorithm is identical to FIXEDSEQ but heuristic cell coloring influences the cells chosen at each parallel move. We conjectured that heuristic cell coloring could restrict the potential move sets enough to impair convergence. Figure 6.4 plots move-acceptance percentage vs. temperature for FIXEDSEQ and CELLCOL. At high temperature the acceptance rate is lower for CELLCOL than for FIXEDSEQ. If a processor pair fails to evaluate a move because of lack of cells of the proper color, this is tallied as nonacceptance. This graph then illustrates how restricting the pool of available cells in a parallel move set can damage acceptance rate, which in turn requires more parallel cell moves to reach final high-quality placement. For all the circuits tested, the final placement configuration found by CELLCOL was inferior to that for either FIXEDSEQ or ADAPTIVE.

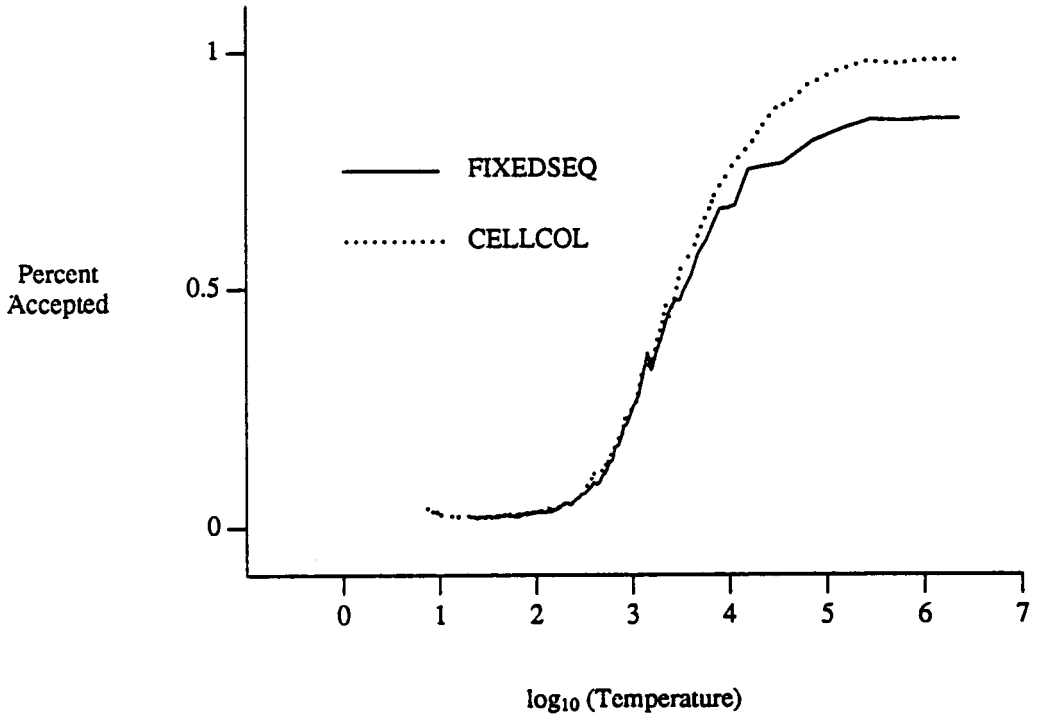


Figure 6.4. Move Acceptance Rate for FIXEDSEQ and CELLCOL

6.4. ADAPTIVE Algorithm

The ADAPTIVE program employs adaptive sequence-length control to achieve greater runtime performance through error monitoring. Our adaptive algorithm restricts sequence length to within a certain percentage of error so that the composite acceptance rate with error will stay very close to the composite acceptance rate without error. Figure 6.5 below compares the acceptance rate of the adaptive algorithm with 16 processors with that of the nonadaptive uniprocessor algorithm (i.e., no error implies unbiased acceptance rate). The adaptive acceptance rate closely follows the rate of the nonadaptive algorithm.

Figure 6.6 is a plot of sequence length vs. temperature for a 183-cell circuit placed on a 16-processor hypercube. Sequence length starts at 1 and climbs rapidly until the system reaches a particular temperature. The sequence length drops sharply to zero when the amount of temporary error in a single parallel move exceeds the allowable bound based on temperature. After this point, error at each parallel move is usually either zero or much larger than the allowable bound, which produces the wild oscillation in sequence length seen. At low tem-

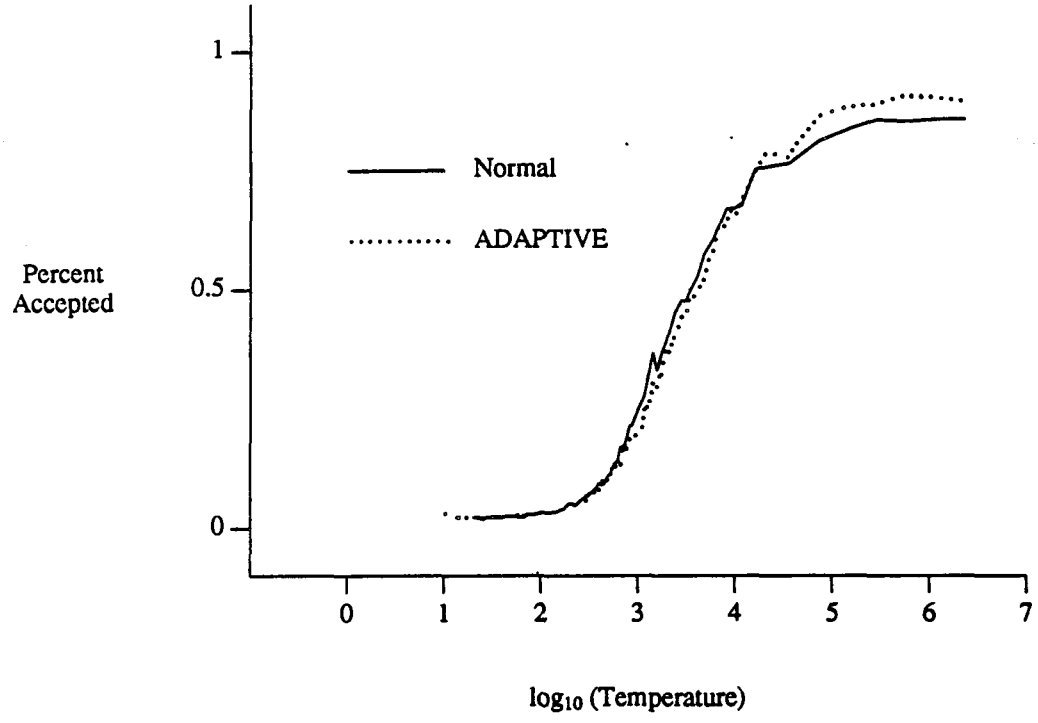


Figure 6.5. ADAPTIVE Acceptance Rate and Normal Acceptance Rate

perature a long sequence length is still appropriate because so few moves are accepted, but after the "threshold temperature" is reached a new method for modulating sequence length is needed. This is still a subject of our research.

Though fixed sequence lengths have been reported by other groups [20,22,23] **we find that the average ADAPTIVE sequence length is usually one or two orders of magnitude greater than these static limits.** Table 6.1 summarizes the average length of move sequences for all the test circuits. Allocating more processors to the task generates more error (i.e., larger parallel move set). Therefore, the sequences contract to restrict error to a constant fraction of temperature.

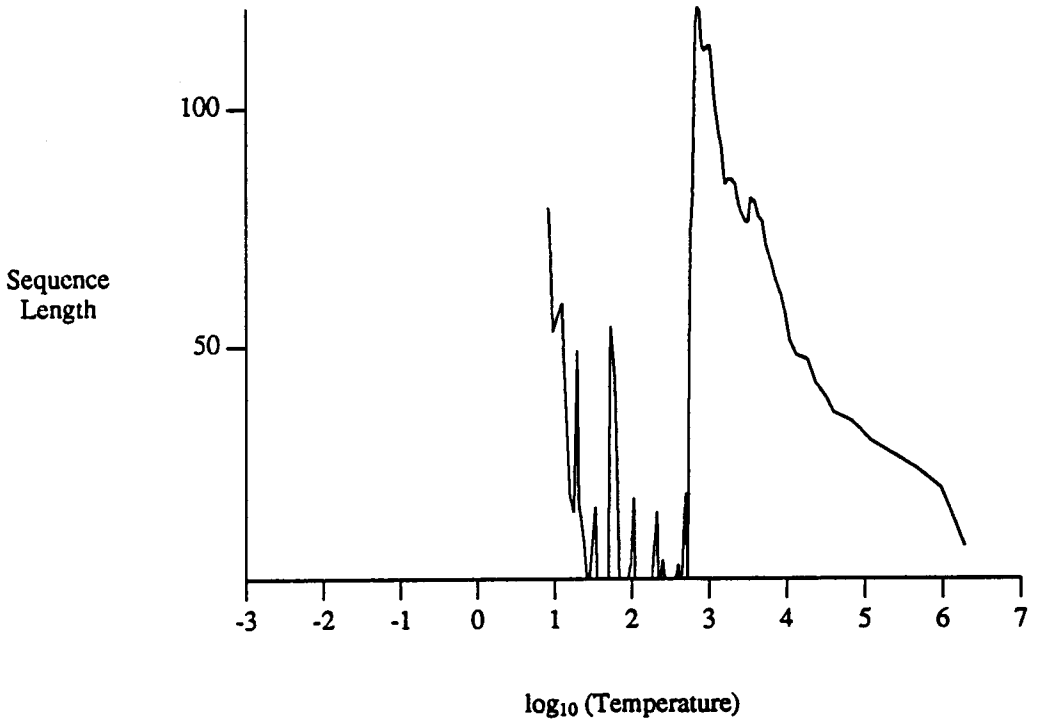


Figure 6.6. Sequence Length vs. Temperature for Sample Circuit

Table 6.1. Average ADAPTIVE Move-Sequence Length

Circuit Size	Number of Processors	Sequence Length
32	1	72
	2	60
	4	44
64	1	157
	2	115
	4	57
	8	33
183	1	70
	2	78
	4	60
	8	75
	16	40
286	1	329
	2	304
	4	275
	8	210
	16	105
469	1	na
	2	na
	4	287
	8	na
	16	145
800	1	na
	2	na
	4	316
	8	na
	16	174

6.5. Placement Results

Our algorithm produced placement results equivalent to TimberWolf3.2. Table 6.2 summarizes our placement results for various test circuits and hypercube sizes. FIXEDSEQ converged to a better or equivalent placement for the two circuits we could compare to TimberWolf. Larger circuits were not placed with FIXEDSEQ due to prohibitively long run time. The results for the 32-cell and 64-cell circuit are compared to placement values from Jones and Banerjee's program.

CELLCOL plainly did not converge for the three circuits attempted. This is probably a result of the restricted move set available to the annealing algorithm under heuristic cell coloring. Again larger circuits were not attempted due to excessive run times.

ADAPTIVE had excellent convergence for all circuits and hypercube sizes, with the exception of the 286-cell circuit which exhibited somewhat poorer placement.

Table 6.2. Optimized Placement Wire Length

Circuit Size	Number of Processors	FIXEDSEQ	CELLCOL	ADAPTIVE	TimberWolf
32	1	4666	6803	4930	5101
	2	4761	6485	4855	-
	4	5276	6663	4995	-
64	1	14604	25098	14448	25798
	2	14830	24385	14553	-
	4	14633	24497	14626	-
	8	14497	22320	14541	-
183	1	80588	102191	81422	97956
	2	80747	109021	75324	-
	4	86932	114025	77968	-
	8	82751	119648	89537	-
	16	82594	125545	83095	-
286	1	126478	na	144939	127788
	2	132140	na	135517	-
	4	131887	na	172017	-
	8	153923	na	149587	-
	16	143310	na	142849	-
469	1	na	na	na	258744
	2	na	na	na	-
	4	na	na	245808	-
	8	na	na	na	-
	16	na	na	249162	-
800	1	na	na	na	494948
	2	na	na	na	-
	4	na	na	544721	-
	8	na	na	na	-
	16	na	na	553875	-

6.6. Runtime and Speedup

With adaptive sequence-length control our algorithm ran between 5 and 16 times faster than a previous hypercube algorithm [22], and now has an overall execution time comparable to TimberWolf. The runtimes and speedups in minutes for three variants of the algorithm are listed in Table 6.3. Speedups are not as high as those reported by Jones [22], in which a fixed annealing schedule was used. The number of temperature decrements increased in the presence of error, and more total moves were evaluated to compensate for this effect, reducing speedup.

We expect with slight changes to the sequence-control feedback mechanism that we can improve this overall performance by a factor of 2 to 5 while preserving convergence.

Table 6.3. Execution Time and Speedup

Circuit Size	Number of Processors	FIXEDSEQ		CELLCOL		ADAPTIVE		Jones
		Time	Speedup	Time	Speedup	Time	Speedup	Time
32	1	8	1.0	7	1.0	7	1.0	36
	2	9	0.9	6	1.2	6	1.2	-
	4	7	1.1	5	1.4	3	2.1	30
64	1	32	1.0	38	1.0	28	1.0	102
	2	30	1.1	31	1.2	17	1.7	-
	4	24	1.3	22	1.7	10	2.8	66
	8	22	1.4	20	1.9	7	4.4	-
183	1	406	1.0	542	1.0	434	1.0	876
	2	451	0.9	465	1.2	433	1.0	-
	4	447	0.9	247	2.2	125	3.5	408
	8	123	3.3	128	4.2	118	3.7	-
	16	118	3.4	97	5.6	58	7.5	168
286	1	2940	1.0	na	na	1350	1.0	6840
	2	2415	1.2	na	na	1114	1.2	-
	4	1135	2.6	na	na	348	3.9	2448
	8	720	4.1	na	na	390	3.5	-
	16	482	6.1	na	na	170	7.9	840
469	1	na	na	na	na	na	na	136800
	2	na	na	na	na	na	na	-
	4	na	na	na	na	4149	na	45600
	8	na	na	na	na	na	na	-
	16	na	na	na	na	1018	na	10800
800	1	na	na	na	na	na	na	117720
	2	na	na	na	na	na	na	-
	4	na	na	na	na	6225	na	30780
	8	na	na	na	na	na	na	-
	16	na	na	na	na	1260	na	10260

CHAPTER 7

CONCLUSION

7.1. Summary of Results

We have presented a new row-based parallel cell-placement algorithm based on the simulated annealing technique, designed to run on a hypercube multiprocessor. The algorithm has been implemented on the Intel iPSC/2 Hypercube. The placement results of this algorithm are equivalent to TimberWolf - the standard for comparison in academia and industry - yet the execution time is much faster. Also the runtime is up to 16 times faster than a previous placement algorithm implemented on the hypercube [22].

Though all the parallel placement algorithms suffer from error as summarized by Durand [32], we are the first to implement error control in a systematic method with theoretical grounding. We have addressed the problem of error in parallel cell moves with two new techniques: **Heuristic Cell Coloring** and **Adaptive Sequence-Length Control**. Heuristic Cell Coloring eliminates all temporary error in a parallel move set by only choosing noninteracting cells at each parallel move. Cell coloring takes only a small fraction of the time necessary to place the circuit, and needs to be done only once as a preprocessing step. Adaptive Sequence-Length Control allows multiple parallel moves to occur between cell-position updates. We derive a bound on acceptable error based on the traditional move-acceptance curve, and the sequence length is modulated so that average measured error approaches this bound from below. Long sequence lengths provide high performance because all computation and interprocessor communication that take place during a sequence are asynchronous and nearly local. This locality of computation and communication enables the overall speedup to scale with the hypercube size.

7.2. Future Research

Having established a viable placement tool that can place industrial-sized circuits in a reasonable amount of time, the next task is to integrate channel-routing and placement into a single parallel algorithm. Merging a high-quality parallel router such as Brouwer's [36] with this placer should provide superior results over distinct place and route programs.

There are several improvements that can be made to this program to improve performance. First, adaptive sequence-length control provides tremendous improvements in execution time. It would be worthwhile to experiment with other (looser) bounds on error to maximize sequence length. Second, the move set of intraprocessor and interprocessor cell moves is not efficient in that the Slave processor is always dormant during an intraprocessor move. Perhaps a better move set would be along the lines of the latest version of TimberWolf [15] with mostly cell exchanges taking place over short distances. Third, as Huang points out [16] the statistical approach relies on a normal cost distribution. This is strictly the case only at high temperatures. The cost interval and target counts should be updated with decreasing temperature to account for the changing shape of the distribution.

APPENDIX A

PERFORMANCE TIMINGS

A.1. Introduction

Prior to development of the new row-based annealing algorithm, Jones and Banerjee's parallel algorithm was timed in some detail to determine performance bottlenecks. The new row-based algorithm was developed to surmount these bottlenecks in order to provide greater performance. No automatic profiling tools are available (a la 'gprof' under UNIX) for the hypercube, so code was inserted into the program to perform the timings. Each node processor in the iPSC/1 and iPSC/2 has a local real-time clock with precision in single milliseconds. Though accurate to milliseconds, on the iPSC/1 granularity is only available to 5 milliseconds. This clock value is accessible through the CLOCK() (iPSC/1) and MCLOCK() (iPSC/2) "C" language function calls. By bracketing important subroutines and code fragments with "clock" calls, timing estimates were made. To enhance accuracy the code fragments to be timed were repeated 1000-5000 times in a single timing. This reduced the effect of overhead interfering with timing, and made it possible to time routines with duration < 5 msec on the iPSC/1.

In the following sections, a brief algorithmic outline of each move type will be presented, with master and slave contributions listed side-by-side as computations occur simultaneously. Timings of important computational steps from the outline will follow. Pairwise node message traffic will be tabulated, as well as the synchronizing broadcast. Finally, absolute performance timings and relative speedups will be presented.

As in TimberWolf3.2, the ratio of displacements to exchanges is about 5:1. All processors synchronize at the end of a move cycle so performance will be limited by the slowest of the four move types listed above. This synchronization has been implemented as a combining tree broadcast, and with a simple ring broadcast.

A.2. Intraprocessor Cell Displacement

The master displaces one of its cells to another location within its allocated chip area. The slave initially sends a cell structure packet which the master discards without inspection. Though this is a wasted message for this move type, for other move types this initial cell transfer is useful data, and its presence provides a regular communication pattern. Table 2(a) shows the outline of the steps involved in the intraprocessor displacement in

the master and the slave processor and the average measured execution times for each step on the iPSC/2 abd iPSC/1 are shown in Table 2(b) and 2(c) respectively.

Table 2. Intraprocessor Cell Displacement

(a) Outline of move steps.

master	slave
Send null cell to slave.	Select cell at random. Send cell to master.
Receive cell from slave (discard). Select cell at random. Select random location. Compute displacement cost. If displacement cost acceptable, then update position, switch rows.	Receive cell from master. (Determines move type)
Broadcast changes.	Broadcast null.
Update affected cell structures.	Update affected cell structures.

(b) Move step timings in milliseconds for iPSC/2

Number of Cells	Number of Procs	master			Total Time
		Select Random Cell	Select Random Location	Compute Displacement Cost	
32	1	0.08	0.28	2.25	2.61
	4	0.06	0.28	2.19	2.53
64	1	0.13	0.28	2.51	2.91
	4	0.08	0.28	2.46	2.82
	16	0.07	0.28	1.79	2.14
183	1	0.29	0.28	3.77	4.33
	4	0.13	0.28	3.70	4.11
	16	0.09	0.28	2.76	3.13
286	1	0.41	0.28	8.43	9.12
	4	0.19	0.28	7.88	8.35
	16	0.11	0.28	5.89	6.28
469	1	na	na	na	na
	4	0.26	0.28	23.52	26.45
	16	0.12	0.28	24.81	25.21
800	1	1.05	0.28	11.93	13.26
	4	0.42	0.28	11.54	12.24
	16	0.18	0.28	8.53	8.99

(c) Move step timings in milliseconds for iPSC/1

Number of Cells	Number of Procs	master			Total Time
		Select Random Cell	Select Random Location	Compute Displacement Cost	
32	1	0.75	1.16	12.44	14.35
	4	0.41	1.16	12.35	13.92
64	1	1.21	1.16	13.90	16.27
	4	0.55	1.16	13.87	15.58
	16	0.36	1.16	10.56	12.08
183	1	2.76	1.16	20.87	24.79
	4	0.90	1.16	20.92	22.98
	16	0.49	1.16	16.31	17.96
286	1	3.94	1.16	46.69	51.79
	4	1.28	1.16	44.50	46.94
	16	0.55	1.16	34.81	36.52
469	1	na	na	na	na
	4	1.74	1.16	132.90	135.80
	16	0.64	1.16	146.44	148.24
800	1	10.17	1.16	66.07	77.4
	4	2.88	1.16	65.18	69.22
	16	0.95	1.16	50.32	52.43

A.3. Intraprocessor Cell Exchange

The master selects two candidate cells and exchanges their positions. Change in cost is calculated entirely by the master, as is the decision whether or not to accept the move. Table 3(a) shows the outline of the steps involved in the intraprocessor cell exchange move in the master and the slave processor and the average measured execution times for each step are shown in Table 3(b).

Table 3. Intraprocessor Cell Exchange

(a) Outline of move steps

master	slave
Send null cell to slave.	Select cell at random. Send cell to master.
Receive cell from slave. (discard)	Receive cell from master. (Determines move type)
Select a cell at random. Select a second cell at random. Compute exchange cost of the two cells. If exchange cost acceptable, then modify cell list.	
Broadcast changes.	Broadcast null.
Update affected cell structures.	Update affected cell structures.

(b) Move step timings in milliseconds

Number of Cells	Number of Procs	master		Total Time
		Select Two Random Cells	Compute Total Exchange Cost	
32	1	0.16	4.07	4.23
	4	0.12	3.99	4.11
64	1	0.26	4.70	4.96
	4	0.16	4.92	5.08
	16	0.14	3.84	3.98
183	1	0.58	6.94	7.52
	4	0.26	8.16	8.42
	16	0.18	5.60	5.78
286	1	0.82	18.14	18.96
	4	0.28	17.49	17.77
	16	0.22	11.99	12.21
469	1	na	na	na
	4	0.52	48.97	49.49
	16	0.24	45.56	45.80
800	1	2.10	22.31	24.41
	4	0.84	22.93	23.77
	16	0.36	17.66	18.02

A.4. Interprocessor Cell Displacement

The master selects the candidate cell, computes the effect of losing it from its local area, and sends the cell to the slave. The slave picks a new location and computes the cost of accepting the cell. The decision whether or not to accept the move is then made by the slave. Table 4(a) shows the outline of the steps involved in the interprocessor cell displacement move in the master and the slave processor and the average measured execution

times for each step are shown in Table 4(b).

Table 4. Interprocessor Cell Displacement

(a) Outline of move steps

master	slave
Select cell at random.	Select cell at random. Send cell to master.
Receive cell from slave (discard). Compute displacement cost.	
Send cell to slave.	Receive cell from master. (Determines move type) Select random location for cell. If displacement cost acceptable, then update position, add cell.
Broadcast null.	Broadcast changes.
Update affected cell structures.	Update affected cell structure.

(b) Move step timings in milliseconds

Number of Cells	Number of Procs	master		slave		Total Time
		Select Random Cell	Compute Displacement Cost	Select Random Cell	Select Random Location	
32	4	0.08	1.03	0.08	0.28	1.31
64	4	0.13	1.04	0.13	0.28	1.45
	16	0.08	0.71	0.08	0.28	1.07
183	4	0.29	1.82	0.29	0.28	2.39
	16	0.13	1.16	0.13	0.28	1.57
286	4	0.41	2.58	0.41	0.28	3.27
	16	0.19	1.63	0.19	0.28	2.10
469	4	0.26	7.83	0.26	0.28	8.37
	16	0.12	6.88	0.12	0.28	7.28
800	4	0.42	5.86	0.42	0.28	6.56
	16	0.18	3.94	0.18	0.28	4.40

A.5. Interprocessor Cell Exchange

master and slave both select random cells, and both compute partial exchange costs. The slave informs the master of its cost calculations and the master then makes the decision whether or not to accept the move. Table 5(a) shows the outline of the steps involved in the interprocessor cell exchange move in the master and the slave processors and the average measured execution times for each step are shown in Table 5(b).

Table 5. Interprocessor Cell Exchange

(a) Outline of move steps

master	slave
Select a cell at random.	Select cell at random.
Send cell to slave.	Send cell to master.
Receive cell from slave.	Receive cell from master.
Compute partial exchange cost P1.	Compute partial exchange cost P2.
	Send partial cost P2 to master.
Receive partial P2 cost from slave.	
If aggregate (P1+P2) cost acceptable, then modify cell list.	
Broadcast changes.	Broadcast null.
Update affected cell structures.	Update affected cell structures.

(b) Move step timings in milliseconds

Number of Cells	Number of Procs	master		slave		Total Time
		Select Random Cell	Compute P1 Partial Cost	Select Random Cell	Compute P2 Partial Cost	
32	4	0.06	2.11	0.06	2.11	2.17
64	4	0.08	2.57	0.08	2.57	2.65
	16	0.07	1.76	0.07	1.76	1.83
183	4	0.13	4.13	0.13	4.13	4.26
	16	0.09	2.72	0.09	2.72	2.81
286	4	0.19	6.18	0.19	6.18	6.37
	16	0.11	6.86	0.11	6.86	6.97
469	4	0.26	24.09	0.26	24.09	24.35
	16	0.12	20.83	0.12	20.83	20.95
800	4	0.42	11.10	0.42	11.10	11.53
	16	0.18	7.79	0.18	7.79	7.97

A.6. Master-Slave Message Traffic

The interprocessor exchange move type requires the master and slave exchange entire candidate cell data structures in order to calculate cost in a distributed fashion. Other move types require structure transmission from slave to master only, or transmission of an empty acknowledgement packet. The communication cost incurred by transmission of a single average cell data structure is given in Table 6. Cell data structures will vary in size depending on connectivity; a minimum, maximum, and average message length is presented.

Table 6. Message Latency (milliseconds) on Node Processor

Number	Length in Bytes			Average Latency	
of Cells	Min	Max	Avg	iPSC-1	iPSC-2
32	104	308	190	2.00	0.87
64	104	400	230	2.03	0.88
183	68	1024	359	2.20	0.91
286	68	3800	851	3.34	1.21
469	68	6124	4344	7.10	2.56
800	68	892	403	2.27	0.97

A.7. Broadcast Timing

Two varieties of N-way broadcast were implemented and timed. Ring broadcast maps a virtual ring onto the physical hypercube topology, and has a delay proportional to the number of processors. The combining tree broadcast has delay proportional to the dimension of the hypercube. Performance for four- and sixteen-node hypercubes is listed below in Table 7.

Broadcast packets sent from each node fall into 3 discrete sizes: 0 bytes, 28 bytes, and 56 bytes depending on the move type performed and whether or not the generated move was accepted. This packet size is independent of circuit size, and thus total broadcast time is independent of circuit size. Zero-length packets will tend to dominate broadcast traffic at low temperatures late into the annealing schedule, and this will reduce broadcast time slightly. The effect is not significant due to the high start-up cost for even empty packets on the iPSC/1.

Table 7. N-way Broadcast Time in Milliseconds

Hypercube Size	iPSC-1		iPSC-2	
	Ring Broadcast	Combining Tree Broadcast	Ring Broadcast	Combining Tree Broadcast
4	15.70	10.72	2.26	2.29
16	102.17	90.42	11.23	7.91

A.8. Cell Update Timings

After every set of of parallel moves, the new cell locations have to be updated in various processors. Cost of updating is directly related to the number of cell moves accepted and is therefore most expensive at high temperatures. This high-temperature update requirement for various circuits is shown below in Table 8.

Table 8. Cell Update Timings in Milliseconds

Number of Cells	Number of Procs	Average Cells Affected	iPSC-1		iPSC-2	
			Total Update Time	Per Cell-Move Update Time	Total Update Time	Per Cell-Move Update Time
32	1	1.0	25.6	25.6	1.8	1.8
	4	2.4	15.7	6.5	2.2	0.9
64	1	1.0	59.8	59.8	4.3	4.3
	4	2.4	39.5	16.5	5.5	2.3
	16	9.6	41.9	4.4	5.3	0.6
183	1	1.0	235.8	235.8	37.2	37.2
	4	2.4	147.3	61.4	27.2	11.3
	16	9.6	159.4	16.6	29.4	3.1
286	1	1.0	1186.1	1186.1	187.1	187.1
	4	2.4	690.3	287.6	127.4	53.1
	16	9.6	713.8	74.4	134.3	13.9
469	1	1.0	na	na	na	na
	4	2.4	4191.2	1746.3	773.2	322.2
	16	9.6	4146.7	431.9	780.9	81.3
800	1	1.0	1475.6	1475.6	232.7	232.7
	4	2.4	913.1	380.5	168.5	70.2
	16	9.6	855.5	89.1	161.1	16.8

A.9. Overall Runtime and Speedup

Execution time and speedup of the algorithm as a function of circuit size and hypercube size are listed in Tables A.9 and A.10. Execution time is given in real-time hours. Speedup as listed here is the ratio of the execution time on a single processor to execution time on a multiple processor configuration.

Table 9. Overall Execution Time and Speedup - iPSC/1

Number of Cells	Number of attempts per cell	1 Processor		4 Processor		16 Processors	
		Runtime(hrs)	Speedup	Runtime(hrs)	Speedup	Runtime(hrs)	Speedup
32	100	4.2	1.0	3.2	1.4	2.4	1.9
64	100	11.7	1.0	6.4	1.8	2.9	4.0
183	25	25.1	1.0	9.7	2.6	3.8	6.7
286	5	32.5	1.0	11.7	2.8	4.1	7.9
469	1	130.7	1.0	43.8	3.0	10.8	12.1
800	1	57.0	1.0	18.1	3.1	5.6	10.2

Table 10. Overall Execution Time and Speedup - iPSC/2.

Number of Cells	Number of attempts per cell	1 Processor		4 Processor		16 Processors	
		Runtime(hrs)	Speedup	Runtime(hrs)	Speedup	Runtime(hrs)	Speedup
32	100	0.6	1.0	0.5	1.2	0.2	2.4
64	100	1.7	1.0	1.1	1.5	0.6	3.0
183	100	14.6	1.0	6.8	2.2	2.8	5.3
286	25	28.5	1.0	10.2	2.8	3.5	8.2
469	2	22.8	1.0	7.6	3.0	1.8	12.5
800	5	21.8	1.0	5.7	3.8	1.9	11.8

A.10. Conclusions

For all circuits except the smallest (32- and 64-cell) the cell-position update time dwarfs the other components of cell-move evaluation. Prior to every update, the hypercube must perform the N-way broadcast to synchronize and share update information. Despite novel algorithms developed to minimize this cost , this broadcast overhead will grow larger proportional to the rest of the algorithm as the hypercube size increases. While the other individual routines have tremendous potential for optimization, broadcast time is dependent almost exclusively on the hardware and would eventually become a serious bottleneck, especially on larger hypercubes. It is for this reason we chose to develop adaptive sequence-length adjustment as an effective means of eliminating broadcast overhead.

APPENDIX B

PROGRAM USERS' GUIDE AND OVERVIEW OF ALGORITHM

B.1.1. Introduction

Physically this program has two parts - the "host" program that runs on the iPSC/2 Host, and the "node" program that runs on each Hypercube node. Both programs are physically divided across several "C" language source files. Each program is compiled separately and a "makefile" is included with the source to ease this process. After the program has been modified or retuned, type "touch *.c", then "make" to rebuild the host and node binary files. It is important to always recompile all the source files if the header file "anneal.h" is modified.

B.1.2. Program invocation

To start up the program on the Intel iPSC/2, simply type:

```
getcube -c test -t Nm4 > NODELOG
host > HOSTLOG &
```

The parameter N above should be the size (number of nodes) of the hypercube desired. The program will automatically adjust for the size of the hypercube allocated. Now preliminary output will be sent to the file HOSTLOG, and execution will start. Output from the nodes is collected separately in the file NODELOG. The NODELOG accumulates data as annealing progresses, recording temperature, cost averages, and other statistics. The HOSTLOG will contain beginning and ending placement costs, elapsed runtime, etc.

B.1.3. Program input

The host program reads the circuit description and system parameters from a file in the local directory called "data." This circuit description file is the same format as specified in M. Jones' thesis, "A Parallel Simulated Annealing Algorithm for Standard Cell Placement on a Hypercube Multiprocessor," with the exception that a new "centered" flag appears as the first line. The first seven lines in this file have the following meaning to the program:

1. Centered Bit. 1 if all pins are located at center of cell, else 0. (1 for data.32 and data.64).

2. Number of attempted moves per cell at each iteration/temperature

of the system. (not used.)

3. Standard height of each logic cell.

4. Bytes of memory required to hold all cell-specification structures.

5. Desired length of every row of cells in circuit.

6. Number of rows of cells in circuit.

7. Desired character prefix for output file.
(not used)

Following these parameters a variable number of cell-specification structures should follow. For each logic cell in the circuit the following format is required:

1. Unique global cell ID-number (zero-based.)

2. Cell width.

3. Total number of nets cell is a member of.

4. For each net specified in 3.

a) Unique global net ID number.

b) Total number of pins in net specified in a.

c) For each of the pins specified in b.

i) ID number of cell in which pin is located.

ii) X and Y location of pin relative to center of cell.
(OMIT if Centered Bit set above.)

In addition, if heuristic cell coloring mode is enabled, the file 'data.col' will be read for coloring details produced by the program -color-. See Section B.2 on coloring for details.

B.1.4. Program output

In addition to the output sent to stdout (UNIX standard output, redirect with a pipe '|' or '>') by the host and node programs, at program completion the host writes out two files:

1) "results" : a pic-format file that shows cell-placement
graphically. Useful for small (under 300 cells) circuits
to verify the program is not creating Frankenstein-like
placement.

2) "twout.pl1" : Optimized cell placement record in TimberWolf format. Actually after the host creates this file it needs to be reformatted slightly using the following commands:

```
cp twout.pl1 twout.tmp
sort -n +6 -8 +1 -3 twout.tmp > twout.pl1
rm twout.tmp
```

B.1.5. Program Tunables and Compiler Directives

Several runtime options are included in the source with conditional #if and #ifdef preprocessor statements. Some merely have to be defined (i.e., with -D*option* in the makefile) while others hold specific values and are defined in the header file 'anneal.h'.

DEBUGZ

Enables all the embedded debugging messages. These messages are sent to standard output along with normal output. This is useful if you don't have any idea where to start but slows the program down tremendously.

WEIGHTED

Causes cost calculations regarding wiring to be based on the formula

$$1/2 perimeter\ of\ bounding\ box \times min(1, sqrt(number\ pins\ in\ net - 2)$$

instead of just 1/2 the bounding box.

iPSC/2

Archaic flag to indicate which hypercube program will run on. Should always be defined.

SYNCH

Another anachronism. Keep this defined.

The following are explicitly defined in 'anneal.h' and change the behavior depending on whether they are set ON = 1, or OFF = 0.

COLORING

Enables Heuristic Cell Coloring.

FEEDBACK

Enables adaptive sequence length adjustment through error control. Obviously COLORING and FEEDBACK should not both be enabled simultaneously.

TWRANGE

Enables a move range limiter identical to TimberWolf3.2.

NORANGE

Disables range limiter. Only one of TWRANGE and NORANGE should be enabled simultaneously.

B.2. Cell-Coloring Program

The cell-coloring program "color" takes as input a circuit description file in the format described above, and generates an output file with lines of the the following format:

Unique_Cell_ID_# Cell_Color

i.e., one line for each cell. Program invocation is simple:

color *inputfile centered-flag outputfile scheme#*

where scheme# is 1 for Brelaz, 2 for Random, and 3 for greedy depth first. A sample invocation would be:

color data.32 1 32.col1 1

Now the file 32.col1 would be copied over to "data.col" for the annealing program if necessary.

B.3. Procedural Description of Parallel Algorithm

The parallel simulated annealing algorithm has been implemented in the C programming language. The software package has been divided into several modules, each of which controls a different aspect of program control. Each of the modules is contained in a separate file. The following sections give details and purpose of the procedures and functions contained in each module.

anneal.h

Header file containing all global structure and constant definitions along with external declarations of global variables. This file is used by all other modules through inclusion in the compilation process.

host.c

This file contains all source code which is loaded into and run by the host-processing node to initialize the system, distribute the work load to the hypercube processing nodes, and gather the final optimized cell placement. This file contains the following procedures and functions:

main - Main functional level procedure of host node which calls all required procedures and loads the processing nodes with executable code.

input_params - Reads from user file the initial setting of various system-wide parameters and allocates buffer space for holding the cell specification structures.

input_mods - Reads from user file the size and interconnectivity of the standard logic cells whose placement is to be optimized.

distribute_mods - Randomly performs the initial placement of cells and distributes the physical chip area among the node processors.

init_mod - Initializes the cell-specification structures at both the cell and net level as determined by the initial random placement.

send_mods - Transfers the cell-specification structure over the hypercube links to each processing node as determined by the **distribute_mods** procedure.

gather_mods - Retrieves the optimal placement of cells from the processing nodes of the hypercube.

print.c

This file contains the procedures run at the host node, which performs terminal and file output of circuit statistics. These procedures include:

network_cost - Calculates and outputs to the terminal the cost of a given cell placement in terms of edge overlap, cell overlap, and required wire routing.

print_mod_pos - Outputs the position of each of the standard logic cells and the total area required for the given placement of cells.

print_circuit - Graphically shows the relative position of each of the cells in a given placement. A file capable of being run using pic | roff -me to create an exact picture of the given placement is also created.

node.c

This file contains the **main** functional level procedure which is duplicated and run at each of the node processors of the hypercube to perform the parallel simulated annealing algorithm.

init.c

This file contains the node procedures and functions which initialize a hypercube node using system parameters and cell specification structures received from the host node. This file contains the following procedures and functions:

init_params - Initializes the system wide parameters received from the host node.

init_mod - Initializes the locally allocated cell specification structures received from the host node.

neighbors - Determines the identity of the node processors which correspond to the east and west logical neighbors of the physically mapped circuit.

init_borders - Interacts with logical east and west node processors to create a list of cells to be used in determining cell overlap attributed to cells in neighboring processors.

net.c

File containing communications-oriented procedures and functions used to transmit and receive information over the links of the hypercube using logical channels. This file contains the following procedures and functions:

send_mod - Transmits the cell-specification structure of a given cell to a neighboring node processor.

rec_mod - Receives a cell-specification structure transmitted using **send_mod**.

broadcast_cost - This function transmits the partial global cost associated with a node's locally allocated cells to all other nodes in the hypercube. It then receives and adds partial costs from all other nodes in order to determine the global cost of the present placement.

broadcast_update - Informs and receives from all other node processors information regarding changes in cell placement during the last iteration of the algorithm.

send_host - Transmits the final placement of all locally allocated cells to the host node.

utility.c

This file contains various computationally intensive procedures and functions used during the iterative phases of the algorithm. This file contains the following procedures and functions:

irandom - Produces a pseudorandom integer between given limits.

drandom - Produces a pseudorandom real valued number between given limits.

param_update - Updates temperature parameter and range limiter.

mod_sel - Randomly selects a cell from a list of locally allocated cells.

dist_ok - Determines if the distance of the movement of a cell is within the bounds set by the range limiter.

accept_change - Determines if a proposed move should be accepted based on the change in cost and an exponential function of temperature.

switch_list - Switches the row a cell is associated with.

insert_mod - Adds a cell to the present set of locally allocated cells.

remove_mod - Removes a cell from the present set of locally allocated cells.

find_cost - Determines the partial global cost associated with the present set of locally allocated cells.

find_my_ex_cost - Determines the change in cost for a proposed intraprocessor exchange of cells.

find_ex_cost - Determines the partial change in cost for a proposed interprocessor exchange of cells.

find_disp_cost - Determines the change in cost for a proposed intraprocessor displacement or the slave processor's partial cost for a proposed interprocessor displacement.

disp_loss_cost - Determines the master's change in cost for an interprocessor displacement.

wire_cost - Determines the change in wiring cost for a proposed move.

overlap_cost - Determines the change in cell overlap with cells within the same processor for a proposed move.

border_cost - Determines the change in cell overlap with cells in logical east and west neighboring processors for a proposed move.

update - Updates all locally allocated cell-specification structures for a change in a given cell's location.

REFERENCES

[1] Carl Sechen, "Average Interconnection Length Estimation for Random and Optimized Placement," *International Conference on Computer-Aided Design '87*, pp. 190-193, 1987.

[2] Bryan T. Preas and Patrick G. Karger, "Automatic Placement - A Review of Current Techniques," *Proceedings 23rd Design Automation Conference*, pp. 622-629, 1986.

[3] M. Hanan and J. M. Kurtzburg, "Placement Techniques," *Design Automation of Digital Systems: Theory and Techniques*, pp. 213-282, 1972.

[4] M. A. Breuer, "Min-cut placement," *Journal Design Automation and Fault Tolerant Computing*, vol. 1, pp. 343-382, 1977.

[5] B. W. Kernighan and S. Lin, "An efficient heuristic for partitioning graphs," *Bell System Technical Journal*, February 1970.

[6] A. E. Dunlop and B. W. Kernighan, "A procedure for placement of standard-cell VLSI circuits," *IEEE Transactions on Computer-Aided Design of Circuits and Systems*, vol. CAD-4, pp. 92-98, January 1985.

[7] M. J. Hanan and J. M. Kurtzburg, "A Review of Placement and the Quadratic Assignment Problem," *SIAM review*, pp. 602-615, June 1985.

[8] T. Blank, "A Survey of Hardware Accelerators Used in Computer-Aided Design," *IEEE Design and Test*, pp. 21-39, August 1984.

[9] D. G. Schweikert, "A 2-dimensional Placement Algorithm for the Layout of Electrical Circuits," *Proceedings 13th Design Automation Conference*, pp. 408-416, June 1976.

[10] M. Hanan, P. K. Wolff Sr., and B. J. Agule, "Some Experimental Results on Placement Techniques," *Proceedings 13th Design Automation Conference*, pp. 214-224, June 1976.

[11] L. Steinberg, "The Backboard Wiring Problem," *SIAM Review*, vol. 3, pp. 37-50, January 1961.

[12] S. Kirkpatrick, C. D. Gelatt, Jr., and M. P. Vecchi, "Optimization by Simulated Annealing," *Science*, vol. 220, pp. 671-680, May 13, 1983.

[13] N. Metropolis, A. Rosenbluth, M. Rosenbluth, A. Teller, and E. Teller, *Journal Chemical Physics*, vol. 21, p. 1087, 1953.

[14] Carl Sechen, Douglas Braun, and Alberto Sangiovanni-Vincentelli, "Thunderbird: a complete standard cell layout package," *IEEE Journal of Solid-State Circuits*, vol. 23, pp. 410-420, April 1988.

[15] Carl Sechen and Kai-Win Lee, "An Improved Simulated Annealing Algorithm for Row-Based Placement," *International Conference on Computer-Aided Design '87*, pp. 478-481, 1987.

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

[17] Emile H. L. Aarts, Frans M. J. de Bont, Erik H. A. Habers, and Peter J. M. van Laarhooven, "Parallel Implementations of the Statistical Cooling," *INTEGRATION*, vol. 4, pp. 209-238, 1986.

[18] Lov K. Grover, "Standard Cell Placement Using Simulated Sintering," *24th ACM/IEEE Design Automation Conference*, pp. 56-59, 1987.

[19] Lov K. Grover, "A New Simulated Annealing Algorithm for Standard Cell Placement," *Proceedings ICCAD '86*, pp. 378-380, 1986.

[20] Jonathan S. Rose, W. Martin Snelgrove, and Zvonko G. Vranesic, "Parallel standard cell placement algorithms with quality equivalent to simulated annealing," *IEEE Transactions on Computer-Aided Design*, vol. 7, pp. 387-396, March 1988.

[21] Andrea Casotto and Alberto Sangiovanni-Vincentelli, "Placement of Standard Cells Using Simulated Annealing on the Connection Machine," *Proceedings ICCAD*, pp. 350-352, 1987.

[22] M. Jones and P. Banerjee, "Performance of a Parallel Algorithm for Standard Cell Placement on the Intel Hypercube," *Proceedings 24th Design Automation Conference*, pp. 807-813, June 1987.

[23] Saul A. Kravitz and Rob A. Rutenbar, "Placement by simulated annealing on a multiprocessor," *IEEE Transactions on CAD*, vol. CAD-6, pp. 534-549, July 1987.

[24] Chi-Pong Wong and Rolf-Dieter Fiebrich, "Simulated Annealing-Based Circuit Placement on The Connection Machine System," *Proceedings International Conference on Computer Design (ICCD '87)*, pp. 78-82, October 1987.

[25] C. L. Seitz, "The Cosmic Cube," *Communications of the ACM*, vol. 28, pp. 22-33, January 1985.

[26] P. Banerjee, M. Jones, and J. Sargent, "A Parallel Simulated Annealing Algorithm for Standard Cell Placement on Hypercube Multiprocessors," *(To appear in IEEE Transactions on CAD)*.

[27] M. Jones and P. Banerjee, "An Improved Simulating Annealing Algorithm for Standard Cell Placement," *Proceedings International Conference on Computer Design*, pp. 83-86, October 1987.

[28] Daniel A. Reed and Richard M. Fujimoto, *Multicomputer Networks: Message-Based Parallel Processing*. Cambridge, Mass.: The MIT Press, 1987.

[29] Tony F. Chan and Youcef Saad, "Multigrid algorithms on the hypercube multiprocessor," *IEEE Transactions on Computers*, vol. C-35, pp. 969-977, November 11, 1986.

[30] Saul A. Kravitz and Rob A. Rutenbar, *Multiprocessor-Based Placement by Simulated Annealing*. 23rd IEEE Design Automation Conference, 1986. pp. 567-573.

[31] Bruce Hajek, "A Tutorial Survey of Theory and Applications of Simulated Annealing," *Proceedings 24th Conference on Decision and Control*, pp. 755-760, 1985.

[32] M. D. Durand, "Controlling error in paralell simulated annealing algorithms for VLSI placement," *(to appear in IEEE Transactions on CAD)*, January 31, 1988.

[33] R. Jayaraman and R. A. Rutenbar, "Floorplanning by Annealing on a Hypercube Multiprocessor," *Proceedings International Conference on Computer-Aided Design*, pp. 346-349, November 1987.

[34] Jonathan S. Turner, "Almost all k-Colorable graphs are easy to color," *Journal of Algorithms*, vol. 9, pp. 63-82, March 1988.

[35] D. Brelaz, "New methods to color the vertices of a graph," *Communications of the ACM*, vol. 22, pp. 251-256, 1979.

[36] Randall Jay Brouwer and Prithviraj Banerjee, "A Parallel Simulated Annealing Algorithm for Channel Routing on a Hypercube Multiprocessor," *to appear in 1988 International Conference on Computer Design*, 1988.

We present placement results for real industry circuits and summarize the performance of an implementation on the Intel iPSC/2 Hypercube. The runtime of this algorithm is 5 to 16 times faster than a previous program developed for the Hypercube, while producing equivalent quality placement. An integrated place and route program for the Intel iPSC/2 Hypercube is currently being developed around this kernel algorithm.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1a. REPORT SECURITY CLASSIFICATION Unclassified	1b. RESTRICTIVE MARKINGS None
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution unlimited
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) UILU-ENG-88-2259 (CSG 92)	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION Coordinated Science Lab University of Illinois 6b. OFFICE SYMBOL (If applicable) N/A	7a. NAME OF MONITORING ORGANIZATION NASA
6c. ADDRESS (City, State, and ZIP Code) 1101 W. Springfield Ave. Urbana, IL 61801	7b. ADDRESS (City, State, and ZIP Code) NASA Langley Research Center Hampton, VA 23665
8a. NAME OF FUNDING/SPONSORING ORGANIZATION NASA 8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER NASA NAG 1-613
8c. ADDRESS (City, State, and ZIP Code) See 7b.	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO.

11. TITLE (Include Security Classification)

A Parallel Row-Based Algorithm with Error Control For Standard-Cell Placement on a Hypercube Multiprocessor.

12. PERSONAL AUTHOR(S)

Jeff Scott Sargent

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Technical	FROM _____ TO _____	1988 August	68

16. SUPPLEMENTARY NOTATION

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	Cell placement, simulated annealing, parallel alorithm, hypercube, error control

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

A new row-based parallel algorithm for standard-cell placement targeted for execution on a hypercube multiprocessor is presented. Key features of this implementation include a dynamic simulated-annealing schedule, row-partitioning of the VLSI chip image, and two novel new approaches to controlling error in parallel cell-placement algorithms: Heuristic Cell-Coloring and Adaptive (Parallel Move) Sequence Control. Heuristic Cell-Coloring identifies sets of noninteracting cells that can be moved repeatedly, and in parallel, with no buildup of error in the placement cost. Adaptive Sequence Control allows multiple parallel cell moves to take place between global cell-position "updates." This feedback mechanism is based on an error bound we derive analytically from the traditional annealing move-acceptance profile.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT	21. ABSTRACT SECURITY CLASSIFICATION
☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	Unclassified
22a. NAME OF RESPONSIBLE INDIVIDUAL	22b. TELEPHONE (Include Area Code) / 22c. OFFICE SYMBOL