January 1987

UILU-ENG-87-2209
CSG-60

COORDINATED SCIENCE LABORATORY
College of Engineering

A PARALLEL SIMULATED ANNEALING ALGORITHM FOR STANDARD CELL PLACEMENT ON A HYPERCUBE COMPUTER

Mark Howard Jones

UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

A PARALLEL SIMULATED ANNEALING ALGORITHM FOR STANDARD CELL PLACEMENT ON A HYPERCUBE COMPUTER

BY

MARK HOWARD JONES

B.S., Michigan State University, 1985

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, 1987

Urbana, Illinois

ABSTRACT

A parallel processing algorithm for standard cell placement suitable for execution on a hypercube computer is presented. In the past there have been proposed several parallel algorithms for performing module placement that are suitable for execution on a two-dimensional array of processors. These algorithms had several limitations; namely, they got stuck at local minima, were susceptible to oscillation, could not handle variable size modules (standard cells), and allowed only nearest neighbor exchanges. Recently, simulated annealing, a general purpose method of multivariate optimization, has been applied to solve the standard cell placement problem on conventional uniprocessor computers. These algorithms do not get stuck at local minima and can handle modules of various sizes, but take an enormous amount of time to execute. In this thesis, a parallel version of the simulated annealing algorithm is presented which is targeted to run on a hypercube computer. A strategy for mapping the cells in a two-dimensional area of a chip onto processors in an n-dimensional hypercube is proposed such that both small and large distance moves can be applied. Two types of moves are allowed: cell exchanges and cell displacements. The computation of the cost function in parallel among all the processors in the hypercube is described along with a distributed data structure that needs to be stored in the hypercube to support parallel cost evaluation. A novel tree broadcasting strategy is used extensively in the algorithm for updating cell locations in the parallel environment. Studies on the performance of the algorithm on example industrial circuits show that it is faster and gives better final placement results than the uniprocessor simulated annealing algorithms. An improved uniprocessor algorithm is proposed which is based on the improved results obtained from parallelization of the simulated annealing algorithm. This enhanced algorithm, through the use of nonuniformly distributed moves and slightly outdated placement data, is found to be less likely to get stuck at local minima, and is found to converge to a better final placement for a variety of industry standard circuits.

ACKNOWLEDGEMENT

I wish to acknowledge the help of my advisor, Prithviraj Banerjee, in critiquing and providing helpful insight and information in the development of this thesis.

TABLE OF CONTENTS

CHAPTER PAGE

LIST OF TABLES

Table Page

LIST OF FIGURES

Figure Page

CHAPTER 1

INTRODUCTION

1.1. Motivation

As the complexity of digital systems implemented in VLSI increases, there is a greater need for automating the design of the layout for these systems. One of the areas of VLSI design automation which has received substantial attention in recent years is in researching algorithms for determining the placement of simple cells or modules in a VLSI design. The placement problem consists of finding an optimum assignment of N modules on a board with respect to some criterion prescribed on the interconnections of these modules, such as minimal wire length or signal propagation delays. The terms "module" and "board" are used as generic terms and apply equally well to all circuit levels. The physical design of computers includes several distinct categories of placement problems, depending on the type of packages involved.

The simplest placement problems arise in designing chips with structured layout rules. In these "gate array" chips, standard logic circuits, such as three- or four-input NOR's, are preplaced in a regular grid arrangement [1, 2]. The designer specifies only the signal wiring, which occupies the final, highest layers of the chip. In more general VLSI design, the standard cell layout is such that a set of standard cells of constant height and variable width are arranged in horizontal rows with pads placed around the periphery of the chip. These standard circuits may all be identical, or they may be described in terms of a few standard groupings of two or more adjacent cells. Furthermore, macro blocks may also be present on the chip. An example typical standard cell layout is shown in Figure 1.1.

Given a set of standard cells and a net list which describes the interconnections among the cells, the objective is to place the cells so as to minimize the total length of wires interconnecting the cells and to minimize the total area of the chip. Manual placement generally results in area and

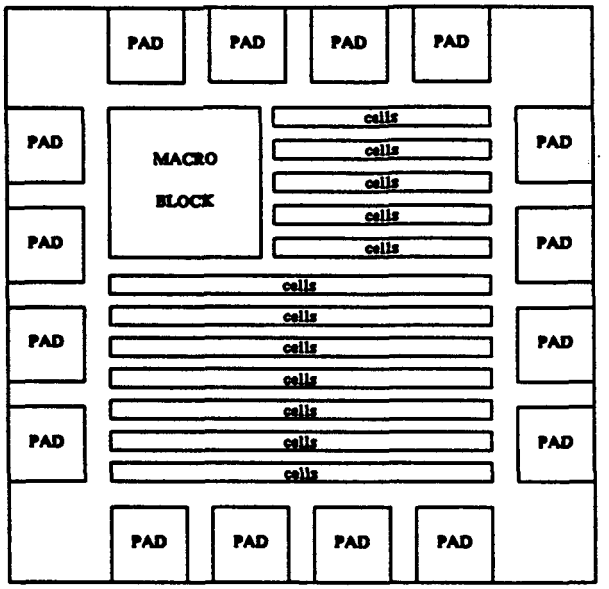


Figure 1.1. Example standard cell VLSI layout.

performance efficiency for small circuits. However, for very large circuits, not only is the design time prohibitively long, but the area and performance suffer. The problem that arises in automating this process is that like many combinatorial optimization problems this problem is NP-complete [3]. The time required to perform an algorithmic solution, which surveys all possible solutions of a given placement problem, grows exponentially with the number of cells. Fortunately, in practice one needs merely a good solution and some sort of assurance that the absolute minimum solution is not significantly better than the one found. Several heuristic methods which attempt to accomplish this have been developed that find good solutions with acceptable computational cost.

1.2. Previous Research

There are two basic strategies for heuristics: "divide-and-conquer" and "iterative improvement." In divide-and-conquer algorithms such as min cut [4], one recursively divides the problem

into subproblems of manageable size, then solves these subproblems individually. The solutions to these subproblems must then be patched back together. For this method to produce very good solutions, the subproblems must be naturally disjoint, and the divisions made must be appropriate ones so that errors made in patching do not offset gains obtained in applying more powerful methods to the subspaces.

Iterative improvement algorithms such as force-directed interchange, pairwise interchange, neighborhood interchange, and forced-directed pairwise relaxation [5, 6, 7] start with the system in a known configuration. A standard rearrangement operation is applied to all parts of the system in turn, until a rearranged configuration is observed which improves the cost function. The rearranged configuration then becomes the new configuration of the system, and the process is continued until no further improvement can be found. Iterative improvement consists of a search in the coordinate work space for rearrangement steps which lead "down hill," i.e., reduce the prescribed cost function. Since this search usually has a tendency to get stuck at a local and not the global minima of the objective cost function, the process normally has to be carried out several times, starting from several different randomly generated initial configurations, and then the best placement obtained is used. In addition to these problems, conventional heuristic algorithms usually do not allow for the amount of flexibility and extensibility desired by users.

To avoid the problems associated with conventional heuristic placement algorithms, a family of heuristic optimization algorithms have been devised based on simulated annealing [8]. These algorithms generate the next placement configuration randomly and can climb hills, i.e., changes that generate configurations of higher cost than the present configuration are sometimes accepted. These "hill climbing" changes are only accepted according to a certain criterion which takes the state-of-the-search process into consideration.

The simulated annealing technique has been proposed and applied to the standard cell placement problem in a program called TimberWolf [9, 10], which by applying all displacements, exchanges, and orientations of cells randomly, avoids getting stuck at local minima and thus

achieves near-optimal placement.

Recently, some researchers have started to investigate speeding up simulated annealing algorithms by running them on parallel processor systems. Aarts et al. have proposed schemes for parallelizing simulated annealing algorithms for several general classes of problems and have discussed theoretical convergence characteristics [11]. A parallel algorithm for the Traveling Salesman Problem based on simulated annealing has been reported for the hypercube [12]. Parallel algorithms for partitioning and routing have been proposed by Chung and Rao [13].

Two multiprocessor-based simulated annealing algorithms for the standard cell problem have been reported by Rutenbar and Kravitz [14, 15]. The first scheme, called Move Decomposition, partitions the computations of the individual move across processors and thus allows the cooperating parallel subtasks to evaluate the effects of this move more rapidly. The second scheme, called the Parallel Moves strategy, allows multiple-move evaluation in parallel but accepts only one of the moves. In this thesis, we propose a parallel simulated annealing algorithm that is targeted to run in a local memory message-passing parallel processing environment, namely the hypercube computer.

There are a number of basic differences in the three approaches to parallelize simulated annealing. In the first two cases, the parallel algorithms are based on a shared memory model, whereas the third uses a local memory model. The first is basically simulating a serial-simulated annealing environment, but evaluating each individual move faster. The second algorithm evaluates multiple moves in parallel but accepts only one move. Hence, its convergence characteristics are identical to the uniprocessor algorithm. In the third case proposed in this thesis, the moves are evaluated in parallel and accepted/rejected in parallel on the basis of changes in the cost function for each move, assuming that the other moves are not made. The theoretical considerations of whether the annealing properties are still preserved when the cost calculations are based on slightly outdated information and when only a restricted set of moves are allowed, is a subject of future research. Experimentally, we have verified that our algorithm works.

1.3. Thesis Outline

In this this, we present a parallel algorithm using simulated annealing on the hypercube computer. The basic idea used in the algorithm involves parallel exchange and displacement moves in different dimensions of the hypercube, and acceptance/rejection of the moves on the basis of changes in cost functions, ignoring the effects of other moves.

In Chapter 2, a detailed description of the hypercube architecture and an overview of the Intel hypercube simulator, which was used for program development, will be presented. In Chapter 3, we will briefly describe conventional simulated annealing, and then discuss a parallel version of the algorithm. We will describe the data structures that are necessary to support various parallel move evaluations and discuss how the subtasks for evaluating the acceptability of parallel moves are assigned. We will present a novel tree broadcasting strategy for the hypercube that is used extensively in our algorithm for updating cell locations in the parallel environment. In Chapter 4, we will describe the implementation of the algorithm on an Intel hypercube simulator. We will report on the performance of the proposed algorithm for several actual standard circuits used in industry and present some accurate estimates of the execution time for the algorithm. We will show that the parallel algorithm gives about 10-20% better final placements than conventional uniprocessor simulated annealing algorithms. Finally, in Chapter 5, an improved uniprocessor simulated annealing algorithm, based on the benefits observed from parallelizing the conventional simulated annealing algorithm, will be presented. We will demonstrate that this improved algorithm is less likely to get stuck at the local minima of the objective function, and thus converges to a final placement which is better than the final placement generated by the conventional uniprocessor algorithm.

CHAPTER 2

HYPERCUBE CONCURRENT PROCESSORS

2.1. Introduction

Supercomputers such as the IBM 3081/3084, CRAY-2, and Burroughs D-825 normally achieve their high performance by increasing the raw speed of the electronic components and logic circuits. For these mammoth computers, the switching and propagation delays are measured in nanoseconds, and data are propagated at speeds close to the speed of light. Unfortunately, these uniprocessors are nearing the limits imposed by physical and electrical constraints. Electronically, uniprocessor computers are reaching their speed limit. To increase the computing speed further, pipelining and parallelizing of operations must be exploited at the circuit level, making these supercomputers very large and very expensive.

An alternative approach to supercomputing is through parallelism at the processor level. We are on the verge of a revolution in computing spawned by advances in computer technology. Progress in very large-scale integration (VLSI) is leading not so much to faster computers, but to much less expensive and much smaller computers, i.e., computers contained on a few chips. These chips make it practical to build very high-performance computers, or supercomputers, consisting of a large number of smaller computers combined to form a single concurrent processor.

The concept of interconnecting multiple, small, inexpensive microcomputers is not new. A number of multiprocessing systems of differing configurations are in existence. Multiple processors communicating with each other via single or time shared bus architecture, such as DCS [16], are very common. In this architecture, several computers are connected to the bus and communicate with each other through token messages. A time shared bus is easy to construct, but the processor-to-processor communications are limited because only one information exchange is allowed at any one time. In another approach, STARAN [17] uses a complete point-to-point con-

nection between processors. This speeds up processor communications and allows simultaneous data transfer; however, the number of interconnection lines increases rapidly as the number of processors increases. C.mmp [18], a multi-miniprocessor at Carnegie-Mellon University, uses crossbar switches between a bank of memories and a bank of processors. This causes only slightly degraded simultaneous transfer ability; however, just like the STARAN, the crossbar network increases in complexity too fast as the number of functional units increases. More recently, Jordon [19] designed a FEM machine, which is a two-dimensional array, that allows any processor to communicate directly with its eight nearest neighbors. Tuazon [20] added more flexibility by providing a switching network that allows a processor to create a communication path to any other processor.

The advent of cost-effective VLSI components in the past few years has made feasible the commercial development of massively parallel computers with upwards of 1024 or more processors. Many different parallel architectures are under development, but the most commercially successful large-scale parallel architecture to date has been the Boolean hypercube, implementations of which are available from at least four different vendors. In the brief time since their introduction, these machines have already gone from experimental prototype status to near-commercial supercomputer performance and have done so at a relatively modest cost.

A significant difference between hypercubes and most other parallel processors is that these multiple-instruction, multiple-data machines (MIMD) use message-passing instead of shared variables for communication between concurrent processes. Each processor has only a small private local memory. Activities with other processors are coordinated by sending messages through an interconnection network. This type of architecture is more readily scaled up to very large numbers of processors than multiprocessor designs based on globally shared memory. The hypercube network is connected densely enough to support efficient communication between arbitrary sets of processors, yet sparsely enough to be relatively simple and inexpensive to build. Another virtue of the hypercube network is its flexibility; many other interconnection topologies (rings, grids, trees, etc.), are subnetworks of the hypercube; hence the hypercube is an ideal test bed for experimenta-

tion with parallel algorithms intended for many different types of distributed-memory, message-passing multiprocessors.

2.2. Hypercube Message-Passing Architecture

2.2.1. Hypercube interconnection network

A hypercube consists of 2^N processors that are connected by the binary N-cube interconnection. The processors are consecutively numbered or tagged by binary integers, i.e., bit strings of length N, from 0 through 2^N-1. In a hypercube interconnection network each processor is directly connected to N other processors whose binary tags differ from its own by exactly one bit. Topologically, this arrangement places the processors at the vertices of the N-dimensional cube. For example, in Figure 2.1, a 3-cube is pictured which has 2^3 processors placed at each of the vertices and communication links, which directly connect the processors, represented by the twelve edges of the 3-cube. Simultaneous communications between several pairs of nodes can therefore occur with

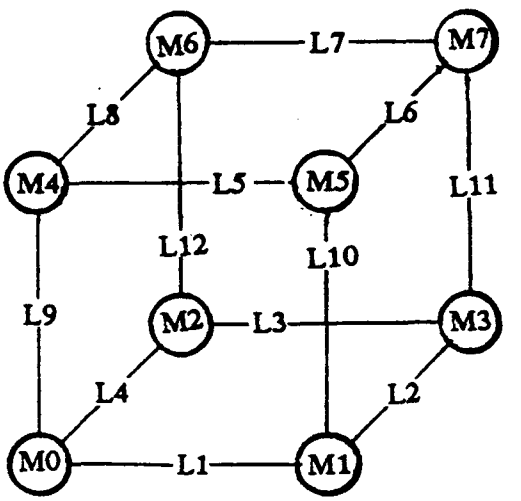


Figure 2.1. Three-dimensional hypercube.

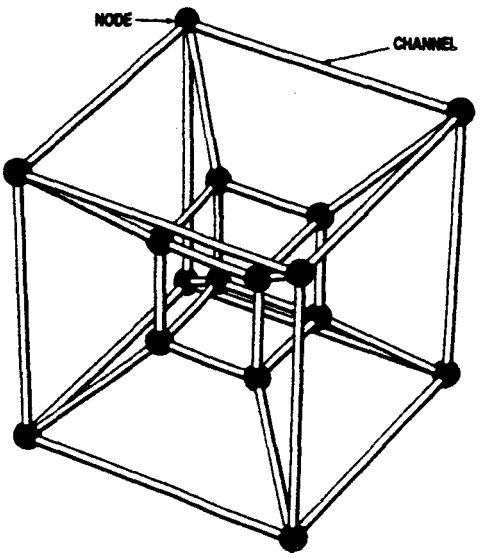


Figure 2.2. Four-dimensional hypercube (16 processing nodes).

this type of interconnection network.

Higher-order (hyper) cubes are more difficult to visualize. Figure 2.2 shows a four-dimensional hypercube which can be described as a cube within a larger cube with corresponding corner nodes connected. Hypercubes of arbitrary dimensions can be constructed by replicating the one of next-lower dimension, then connecting corresponding nodes. One of the advantages of the hypercube network is that as the number of processors increases, the number of connecting links per processor grows only logarithmically, so that very large numbers of processors connected in a hypercube network become both feasible and attractive.

In practice, the actual physical layout of the hypercube's processors is a linear arrangement in a card cage or a planar arrangement on a printed circuit board. Cube connections are then made by wires, conducting layers, or backplane. A planar view of a six-dimensional hypercube is shown in

Figure 2.3.

If a message needs to be sent between a pair of nodes that are not directly connected, then they are routed from node to node until they reach their destination. The routing path can be easily derived by inverting one bit at a time of the bits in the source address which differ from corresponding bits of the destination address until it exactly matches the destination address. For example, to route data from node 0101 to node 1010 in a four-dimensional hypercube, the intermediate nodes, 0100, 0110, and 0010, would be used. It can be easily verified that for an N-dimensional hypercube, the furthest node from any starting node is only $\log_2 N$ away. For every pair of nodes there are $(\log_2 N)!$ possible routes. This redundancy can be exploited to enhance communication bandwidth and fault tolerance of the hypercube network.

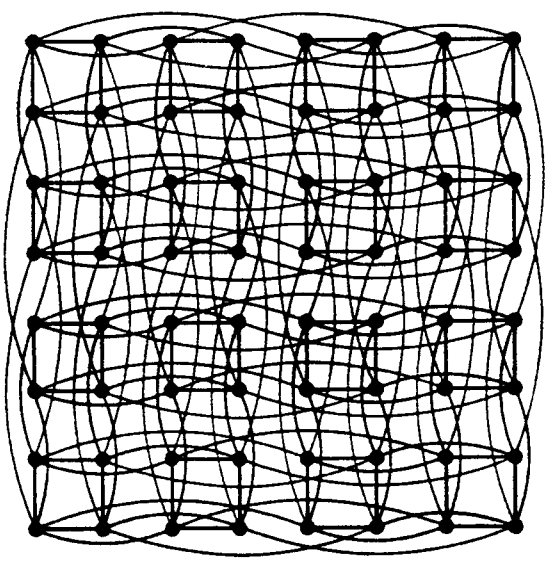
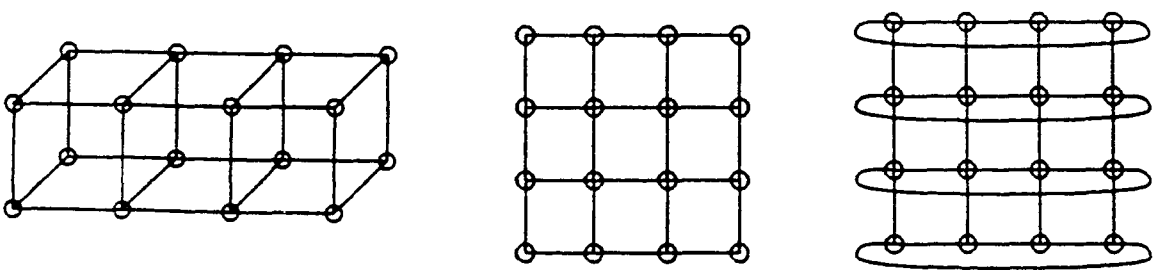


Figure 2.3. Six-dimensional hypercube (64 processing nodes).

Through software, a hypercube can be adapted to model other interconnection configurations by ignoring some of the interconnects. For example, in the four-dimensional hypercube, by ignoring some of the interconnects one can arrive at the variations shown in Figure 2.4 a, b, and c as the 3D cube, 2D plane, and toroidal mesh. The data routing requirements will affect the particular variation used. For example, problems which are normally represented in array form, such as matrix operations and sets of linear equations, etc., can be implemented using a 2D configuration. Analysis of three-dimensional structures can use the 3D topology.

2.2.2. Processing nodes

An attractive feature of the hypercube is its homogeneity. Because of this, all the processing nodes are normally designed to be identical. Nevertheless, with any distributed system, a need usually arises, either by necessity or by convenience, to have a separate processor that acts as master controller or manager of the remaining processors. This special processor, usually called the host, is generally not part of the main hypercube interconnection network, whose processors are



a) Three-dimensional cube b) Two-dimensional plane b) toroidal mesh

Figure 2.4. Subnetworks of four-dimensional hypercube.

referred to as node processors or simply nodes. The role of the host is to initiate a computation, collect results upon completion, and serve as the input/output (IO) link to the outside world. The host must be directly connected to at least a subset of the nodes in the hypercube and, preferably, to all of them, perhaps by a global bus that is used only for host/node communications as opposed to node/node communications.

Because the hosts need to do more powerful operations such as IO, program down loading, and system diagnostics, the architecture of the host is normally faster and more powerful. Because this processor is a critical link in the hypercube, i.e., its loss would disable all IO, the host is normally made to be more fault tolerant.

Each of the hypercube's processing nodes is composed of three separate components: the CPU, local memory, and communications circuitry. Some system designs have a separate communications coprocessor to handle node-to-node communications thus allowing for simultaneous computation and communication. Physically, each processing node is built from as few VLSI chips as possible in order to increase speed and to keep space requirements low.

2.2.3. Distributed software

The hardware structure of the hypercube when viewed at the level of nodes and channels is a difficult target for programming any but the most highly regular computing problems. Most hypercube resident operating systems create a more flexible and machine-independent environment for concurrent computation. Instead of formulating a problem to fit on the nodes and on the physical communication channels that exist only between certain pairs of nodes, the programmer can formulate problems in terms of processes and logical communication channels between processors. This process model of computation is quite similar to the hardware structure of the hypercube but is usually abstracted from it.

Processes are the basic unit of computations and can be described as a sequential program that sends and receives messages. A single node may contain many processes. All processes execute concurrently, whether by virtue of being in different nodes or by being interleaved in execution within

a single node. Multitasking in such an environment is quite feasible. Each process has a unique global identification that arises as an address for messages. All messages have headers containing the destination and the sender identification and a message type and length. Messages are queued in transit, but message order is preserved between any pair of processes.

Because it has only local memory, the hypercube needs to employ a distributed operating system. An operating kernel will reside in each node processor to supervise user processes running on the node and to handle message traffic. In particular, the kernel in a given node sends, receives, and queues messages for processes running on its node, and may also automatically forward incoming messages intended for processes running on other nodes, freeing the main node processor of much of the communication overhead. A variety of operating systems, compilers, and other parallel processing development tools have been designed and implemented for use on the hypercube architecture [21, 22, 23, 24]. Processor scheduling is an important area which has received substantial research in recent years [25, 26, 27].

The host is responsible for compiling application programs and loading the resulting object code into the appropriate node processors. Once the host has initiated a computation, the host and node processors all proceed asynchronously, coordinated only by the exchange of messages containing problem data or control information.

2.3. Current and Future Hypercube Systems

A hypercube of computers was often discussed in the mid 1970's as a practical means to implement a concurrent processing environment [28, 29]. The Russians [30] built a 32-node hypercube in the late 1970's with positive results. Many references have appeared in literature since then concerning the construction or use of hypercube computers [31, 32]. The pioneering work was finally brought to practical fruition in 1983 with the Mark I "cosmic cube" [33] at the California Institute of Technology, where it has since been in regular use for solving a wide variety of important scientific problems. Transfer of this new technology into the commercial sector has been relatively swift. Intel Scientific Computers Corp. announced the first commercially available

hypercube, the Intel Personal Super Computer (iPSC) [34], in early 1985. Several other commercial vendors soon followed: Ametek Computer Research's Ametek [35], NCUBE Corp. [36] with the NCUBE/ten, and Floating Point Systems Inc. with their T series. Further joint development by Caltech and the Jet Propulsion Laboratory has since created a new generation of hypercubes, the Mark II and Mark III.

The availability of these machines is making possible widespread experimentation of large-scale parallel computing for realistic applications. Moreover, these machines are moving quickly from experimental prototypes to genuine supercomputer performance and doing so at a relatively modest cost.

2.3.1. Commercial systems

Several commercially hypercube systems have become available recently. Even though they are all built around the same hypercube message-passing architecture, their actual hardware and software implementation and performance vary considerably from system to system. Three of the systems have already been delivered to customers. These systems include the Intel iPSC, the Ametek Computer Research's Ametek, and the NCUBE Corporation's NCUBE/ten.

Primarily to reduce development time, the first systems introduced used proven widely available VLSI circuits as the backbone of each of the node processors. The Intel and Ametek systems use the 16-bit Intel 80286 [37] to perform all general purpose computations. In addition to its high computational ability, the 80286 was selected for its built-in support of a custom coprocessor, the 80287. The coprocessor interface provides a very low overhead mechanism for a client program to invoke task management functions that are implemented concurrently. By using widely available technology, both systems were able to use existing hardware and software development tools and thus reduce system development time.

The communications hardware in both systems is rather simple and slow. Node-to-node communications are run over links controlled by an Intel ethernet chip at peak rates of 10 megabits/second. Because of this relatively simple communications hardware and the need to

perform a large part of the communications overhead in software, link delays for even very simple messages are in the milliseconds range for both systems [38]. This makes communication very expensive in comparison to computation time. This means that for an algorithm to be feasible for significant speedup on these systems, the ratio of computation to communication has to be rather large. Both systems also tend to have limited amounts of local memory, on the order of 512K bytes.

The Ametek system's communication system is small-packet based, which means that small packets take significantly less time to traverse the links than do larger ones. The Intel iPSC's use of ethernet with standard 1K byte packets enables it to have constant delay for packets of less than 1K bytes. For larger packets, multiple packets have to be sent.

Both of these systems were designed to allow for easy expansion with configurations from 16 to 128 nodes available in both systems. Additionally, recent developments in the Intel iPSC allows for up to 4 megabytes of RAM at each node and to have array processors and accelerators attached to node processors to enhance performance.

These initial systems were primarily designed as a quick implementation of the hypercube architecture for commercial use. Systems which followed these initial entries, primarily the NCUBE/ten, strived to develop special purpose hardware specifically targeted for hypercube use. In most of the parallel systems being proposed or manufactured, each node consists of many chips, often more than 100. In contrast, the NCUBE/ten node has only 7 chips, and 6 of them are memory. The NCUBE/ten uses state-of-the-art VLSI to integrate most of the system (except memory) at each node onto a single chip. Each node is designed to have 128K bytes of local memory with local groups of processors connected to a global 500M byte disc. The NCUBE/ten-node processor is a complex chip of about 160,000 transistors that integrates memory interface, communications links, and a high-speed 32-bit processor with 64-bit floating point. Each node is capable of performing at a peak rate of 0.5 megaflops. A broad range of error-correcting mechanisms in the data paths is incorporated to insure reliability. The NCUBE/ten is expandable from 16

to 1024 processors, and unlike the iPSC and Ametek allows for extremely high-speed IO at each of the processing nodes without the need to transfer information to the host processor first.

2.3.2. Experimental systems

The first experimental hypercube implementation to get significant recognition was the Mark I built at Caltech, commonly known as the cosmic cube [33]. This system consisted of at most 64 processing nodes based on the Intel 8086 microprocessor as data processor and 8087 coprocessor as the floating-point processor. Each node was equipped with 128K bytes of RAM. Full duplex communication channels running at a slow 2 megabits/second were utilized for node-to-node communications. Because of the slow processor and communications speeds and the limited 128K bytes of local memory, the system was definitely not in the supercomputer range, but even with these slow microelectronic technologies, the 64-node machine was found to be quite powerful for its cost and size. The performance of the Mark I encouraged Caltech to develop an enhanced model.

The follow-up to the Mark I was the Mark II system [39]. The Mark II was built in cooperation with the Jet Propulsion Laboratory. This system can be configured up to a 128-node network. Intel 8086 processors and 8087 coprocessors were again used, and RAM was increased to 256K bytes per node, with additional external IO incorporated into groups of node processors. Enhanced hardware and software have significantly increased the systems' performance over that of the Mark I. A follow-up system, the Mark III [40], will be a vastly more powerful machine, constructed from nodes, each of which has two MC68020s, floating point accelerator chips, and 4 megabytes of memory [41]. These powerful nodes, along with equally powerful node-to-node communications hardware, are expected to allow the Mark III to match or surpass the performance of most standard supercomputers available today.

Several other systems are in various stages of research in a number of universities throughout the country. The most ambitious system being developed is at the Los Alamos National Laboratory in connection with the University of New Mexico [42]. This system will be primarily hardware-oriented. Rather than approaching the hypercube problem by using nodes with minimal computing

resources, the engineers at Los Alamos have elected to implement the architecture, using nodes with sufficient computing resources to address interesting problems. A variety of off-the-shelf and specially designed VLSI circuits will be used in an attempt to allow for upwards of 20 megaflops of computational power at each node. A small local memory of between 16K and 64K bytes, along with a large disk with a capacity in excess of 300K bytes, is incorporated into each processing node. Fast node-to-node links with rates in excess of 40 megabits/second will also be incorporated. This system is expected to have a peak performance in excess of 20,000 megaflops

2.3.3. Comparison and benchmarks

The performance of a concurrent processing program depends on the hardware, architecture, and programming algorithm used. The maximum number of concurrent megaflops of computational power is a commonly used yardstick. This number as normally quoted is obviously only "potential" performance, which can only be achieved through efficient programming. Because of the nature of the hypercube architecture, several other factors have to be taken into account. The hardware factors effected by a particular hypercube implementation are

1) Memory Size

Invariably, as the node memory increases, the performance of the system also improves. Unfortunately, large memories can be very expensive. Secondary memory or dual port memories which allow simultaneous communication and computation may be used in some cases.

2) Processing Speed

Since scientific applications are the primary users of hypercubes, it is essential that the floating-point operational speed be as large as possible to solve these computationally intensive problems.

3) Communication Speed

High-speed communication is very important in a message-passing environment. Not only does the link transfer rate have to be high, but the time spent in doing the overhead associ-

ated with transmitting, routing, and receiving has to be kept low.

Table 2.1 gives a comparison of the hardware capabilities of the various systems discussed in the preceding two sections.

Table 2.1. Hardware capability comparison for various systems.

System	Max Number Processors	Type of CPU	Memory Size (bytes)	Computational Mflops (peak)	Communication Mbits/Second
Mark I	64	Intel 8086	128K	8	2
Mark II	128	Intel 8086	256K	15	8
Mark III	1024	MC68020	4M	>1000	-
iPSC	128	Intel 80286	512K-4M	20	10
Ametek	128	Intel 80286	512K	20	10
NCUBE/ten	1024	Custom VLSI	128K/500M	512	10
Los Alamos	1024	Custom VLSI	64K+300K	>20000	40

2.4. Hypercube Simulator

Due to the present unavailability of an actual Intel hypercube at the University of Illinois, initial testing of the algorithm to be presented in the next chapter has been completed using the Intel iPSC Simulator running on a SUN 3/50 work station system under UNIX 4.2 [43]. This simulator was chosen because of assurances that programs which executed properly under the simulator could be transferred to an actual iPSC system and operate with only minimum or no modification required.

The simulator package consists of a simulator program and a set of libraries which simulate hypercube operations in a sequential processing environment. This event-driven simulator provides an interactive interface to the user, which simulates a large portion of the iPSC's host node commands. These commands allow the user to load executable code into each of the nodes of the hypercube and to initiate execution. Nodal processes are simulated in the uniprocessor environment by forking off UNIX processes.

The major difference between hypercube algorithms and uniprocessor algorithms is the need to do message-passing between concurrently operating processors. The primary responsibility of the simulator is to model these message transfers in such a way that ordering of messages is preserved. In order to remove the programmer as far as possible from requiring an understanding of the exact communication routing requirements for a given message, a system of logical channels is adopted in the iPSC system and its simulator. A channel, as used in the iPSC system, is a 64-byte block of memory that contains information about a message to be sent or received. Typical information contained in this block of memory is the source node and process id, the destination node and process id, and the message length. A sending process needs to establish a channel to contain this information before a message can be sent. Likewise, a receiving process must also establish a channel to receive this information before the message can be received. Once an operation (send or receive) has been completed, the information is no longer needed, and the channel can be used again by another message. If a process needs to send/receive more than one message simultaneously, the process needs to open a channel for each of the simultaneous send/receive operations. Because of the nature of these logical channels, the programmer is relieved of determining the actual path over which a message travels. The operating kernel at each node of the hypercube will determine the optimal path between the two nodes connected by the logical channel.

A typical message transfer in software requires a call to a procedure send by the node processor wishing to send a message. Procedure send initiates the transmission of a message to another node processor. The caller can wait for this transmission to complete or simultaneous computation can be taking place. A typical call to send is of the form:

send(ci, type, buf, len, node)

where

ci : Channel identifier of channel over which message is to be transmitted

type : User specified integer value referring to type of message. The receiving node uses this value to distinguish multiple incoming messages.

buf : Pointer to the continuous block of memory (buffer) that contains

the message to be sent.

len : Number of bytes in buffer to be transmitted.

node : Physical address of node to receive message.

In a similar manner the node processor which is to receive the message calls a **receive** procedure. A typical **receive** call is of the form

receive(ci, type, buf, &cnt, &node)

where

ci : Channel identifier of channel over which message is to be received.

type : Integer value referring to the type of message wanting to receive.

buf : Pointer to the buffer where the received message is to be stored.

cnt : Upon reception of a message of the proper type, cnt will contain the number of message bytes received.

node : Upon reception of a message of the proper type, node will contain the identification of the processor which sent the message.

Through the exchange of data by sending and receiving of messages to and from other processors in this manner, nodes can exchange required data and coordinate activities.

CHAPTER 3

PARALLEL ALGORITHM FOR CELL PLACEMENT

3.1. Simulated Annealing Algorithm

Simulated annealing, as proposed by Kirkpatrick [8], is a popular Monte Carlo algorithm for combinatorial optimization. Simulated annealing is a variation on an algorithm introduced by Metropolis [44] for approximate computation of mean values of various statistical-mechanical quantities for a physical system in equilibrium at a given temperature. The Metropolis method, combined with Kirkpatrick's "several temperature" method, is collectively called simulated annealing.

The search for a minimum cost function in a simulated annealing algorithm has a close analogy to the physical process by which a material changes state while minimizing its energy. When a material is crystalized from the liquid phase, it must be cooled slowly if it is to assume its highly-ordered, lowest-energy state. At each temperature during the annealing process, the material is in equilibrium, i.e., the likelihood of its being in a given state is governed by the Boltzman distribution for that temperature. As the temperature decreases, the distribution becomes concentrated on the lower-energy states until, when the temperature finally reaches zero, only the minimum-energy state(s) have nonzero probability. However, if the cooling is too rapid, the material does not have time to reach equilibrium. Instead, various defects become frozen into the structure.

Because conventional iterative improvement algorithms forbid changes of state which increase the cost function, they are much like rapidly reducing a physical system to zero temperature in a very small period of time. Simulated annealing is thus a variation of the conventional iterative improvement algorithms in which uphill moves are permitted in the cost function under the control of a slowly reducing temperature parameter.

A simplified algorithmic structure of the simulated annealing algorithm is given below:

```
PROGRAM SIMULATED ANNEALING
    T = T0;
    X = X0;
    While (stopping criteria not satisfied)
        While (inner loop criteria not satisfied)
            X' = Generate(X);
            evaluate cost(X');
            If( accept(cost(X'), cost(X) ))
                X = X';
            ENDIF;
        ENDWHILE;
        update(T);
    ENDWHILE;
END PROGRAM
```

This algorithm is characterized by three main functions: **accept**, **generate**, and **update**. The function **accept** is used to determine if a proposed new configuration of the circuit should be accepted. While several accept functions can be used [45, 46, 47], a probabilistic exponential function is normally used for standard cell placement optimization because of its proven ability in other similar optimization problems. The **accept** function is given below:

```
FUNCTION accept( cost(X'), cost(X) )
    ΔC = cost(X') - cost(X);
    If( ΔC <= 0 )
        Return(TRUE);
    else
        y = exp(-ΔC /T);
        r = random(0,1);
        If( r < y )
            Return(TRUE);
        else
            Return(FALSE);
        ENDIF;
    ENDIF;
END FUNCTION;
```

New configurations characterized by a negative change in the cost function ($\Delta C \leq 0$) always satisfy the acceptance criterion. However, for new configurations characterized by $\Delta C > 0$, the

temperature parameter T and a random number generator play fundamental roles. If T is very large, then r is likely to be less than y, and a new state is almost always accepted irrespective of ΔC. If T is small, close to zero, then only new configurations which are characterized by very small $\Delta C > 0$ have any chance of being accepted. In general, all states with $\Delta C > 0$ have smaller chances of satisfying the test as the temperature decreases.

The generate function selects a new configuration of the circuit. This means randomly moving cells within the circuit. These moves can either be the exchange of cells, the displacement of a single cell, or an orientation or mirroring change in a cell. In the presented simulated annealing algorithm, the program variable X represents the present placement of cells and X' represents a new candidate cell configuration created by the generate function.

The update function, also called the annealing or cooling schedule, determines a new value for the temperature after completion of the inner loop. The update function is very important in determining the convergence properties of the simulated annealing algorithm. A broad range of update functions which return monotonically decreasing values of temperature have been found to guarantee convergence of the simulated annealing algorithm to an optimal or a near optimal solution [48, 49, 50, 51, 52, 53].

3.2. Overview of Parallel Algorithm

The simulated annealing technique has been proposed and applied to the placement problem in a program called TimberWolf [9, 10], which by applying displacements, exchanges, and orientation changes randomly, avoids getting stuck at local minima and thereby achieves near-optimal final placement results. TimberWolf has been shown to provide substantial chip area savings in comparison to existing standard cell layout methods. We now describe an algorithm for performing the standard cell placement using a variation of the TimberWolf algorithm on a hypercube of $log(P)$-dimensions connecting P processors. Let us suppose that we are given the problem of placing N standard cells where $N >> P$. An outline of this algorithm is shown below. Each of the steps in the algorithm will be described in the following subsections.

STEP 1. Perform initial cell assignments in P processors.

STEP 2. Determine initial temperature.

STEP 3. While "Stopping criteria" : temperature < 0.1 not reached

STEP 4. Generate new temperature

STEP 5. For inner_loop_count = 1 to NA
/* NA = (N × attempt_parameter) / (log(P) × P/2)*/

STEP 6. For each dimension i=0 to log(P)-1 do

STEP 7. Randomly select P/2 moves (exchange or displace) in parallel among pairs of PEs connected in dimension i.

STEP 8. Check "range-limiter" function in dimension i.

STEP 9. Evaluate change in cost for each move between pairs of PEs independently.

STEP 10. Accept/reject moves using exponential function independently.

STEP 11. Broadcast new cell locations to all other processors.

STEP 12. ENDFOR;

STEP 13. ENDFOR;

STEP 14. ENDWHILE;

3.3. Cell Assignment to Processors

We now describe a technique for mapping a $log(P)$-dimensional hypercube onto a two-dimensional area using an example six-dimensional hypercube. The results can be generalized to other dimensions. In the 64-processor hypercube, a processor having a binary address $p_5p_4 \cdots p_i \cdots p_0$ is connected to processor $p_5p_4 \cdots \overline{p_i} \cdots p_0$ via a link in dimension i. We propose that each processor be assigned an approximately equal area portion of the total chip area which can be viewed as a virtual 8 × 8 square grid. Each virtual grid corresponds to a horizontal portion of a number of rows. (For example, for a standard circuit with 16 rows of cells, each processor in a 64-processor hypercube will be in charge of one-eighth the horizontal length of two of the rows.)

The cells are initially assigned randomly to different processors such that each processor has an approximately equal number of cells assigned to it. The cells within each processor are also randomly placed with no regard to area overlaps. We also tested with a strategy of cell assignment such that the sums of areas of cells assigned to each processor is approximately equal to

$$A_{average} = \frac{1}{64} \sum_{m=1}^{N} A_m .$$

where A_m is the area of the m^{th} cell. But because of the large number of moves that are accepted at high temperatures in the initial stages of the annealing process, it does not make any difference which strategy is used since the cells get randomly dispersed anyway. Since all cells have constant height, each processor therefore is assigned a rectangular portion of the chip area. The correspondence between processor addresses and virtual grid regions on the physical chip area is shown in Figure 3.1. By choosing such a map, we guarantee that the processors that are adjacent in a predetermined set of four dimensions of the hypercube allow for all nearest North-South-East-West neighbor displace/exchanges. The other two dimensions of the hypercube are used for displace/exchanges across larger distances in the area map. For example, in Figure 3.1, processor 26, which controls grid location (3,4), has a a 4-link to processor 10, a 3-link to processor 18, a 2-link to processor 30, and a 0-link to processor 27, which correspond to the nearest neighbors in the North(2,4), South(4,4), East(3,5) and West(3,3) directions; in addition, the 1-link to processor 24 and the 5-link to processor 58 control grid locations (3,1) and (6,4), that are distance 3 away from (3,4).

3.4. Distributed Data Structure

We assume that each processor contains a list of cells currently assigned to this processor along with the following information for each cell to aid in the computation of the cost function in parallel among processors in the hypercube:

(1) The width of the cell

(2) The (x,y) coordinate location at which the center of the cell is currently placed

(3) A list of nets to which this cell is connected

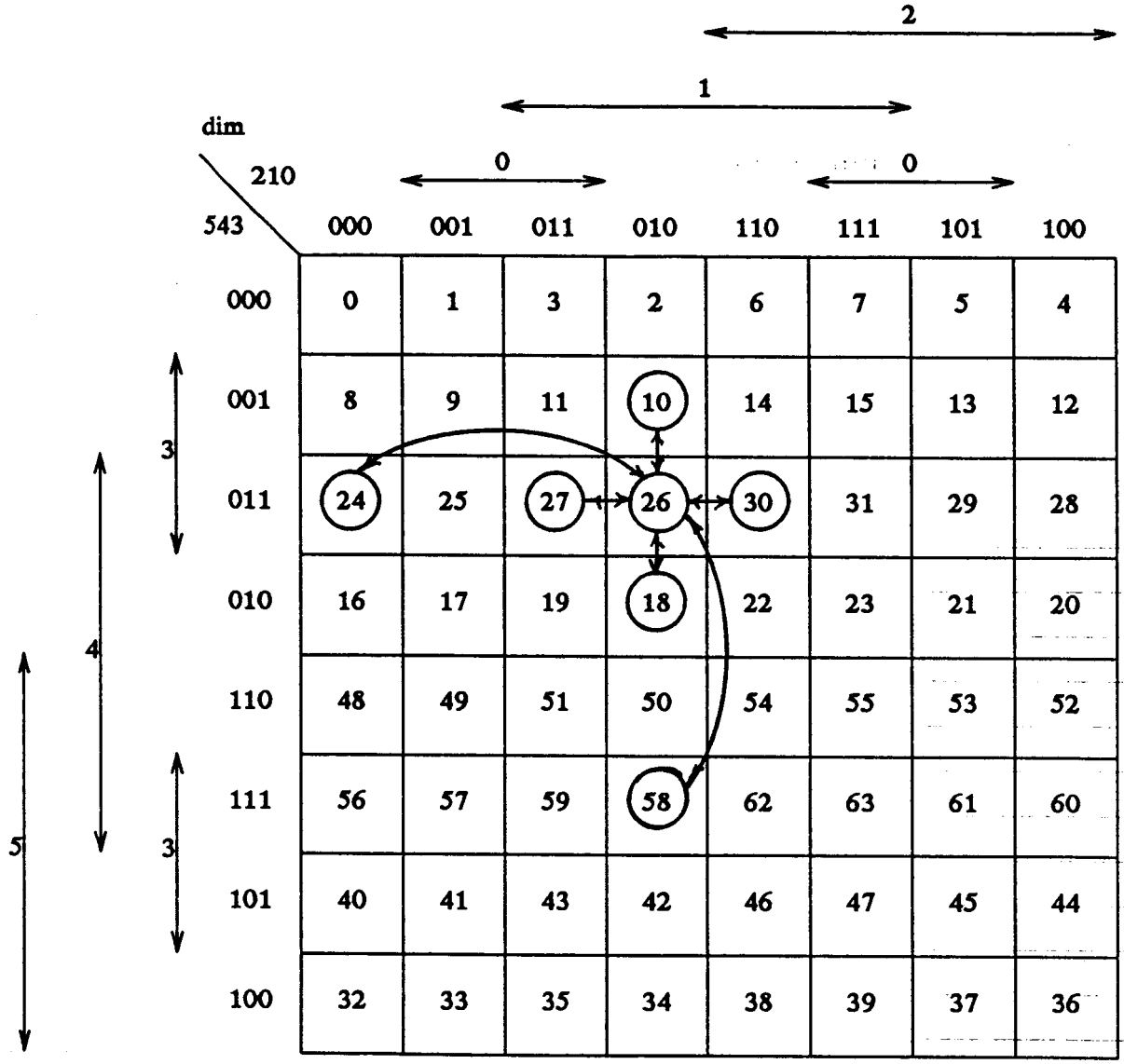


Figure 3.1 Area map of 64-processor hypercube

(4) For each net listed in (4), a list of other cells, to which the net is connected, along with the (x,y) pin location(s) within these cells

The state of any particular cell is composed of the information in (1) through (4) and is packed within a continuous block of memory to allow for easy packet transfer of information between nodes. Also, a list of (x,y) locations and widths of all cells that are assigned to processors that are adjacent in the two dimensions of the hypercube corresponding to the East-West nearest neighbors in the physical area map is also maintained in each processor. Figure 3.2 shows an example of the blocked memory data structure for typical cells.

3.5. Cost Function

Because of the nature of the simulated annealing algorithm, a very complex cost function can be used which takes into account many different aspects of a particular circuit configuration. The cost function for the standard cell placement problem consists of three parts:

(1) Estimated wire-length using half the perimeter of the bounding box rule

(2) Overshoot or undershoot of each row length over or under the desired row length

(3) Linear area overlap between cells in the same row

These are graphically shown in Figure 3.3 with corresponding cost functions. The horizontal work space length is calculated to be equal to 110% of the desired length of every row. The cells for a

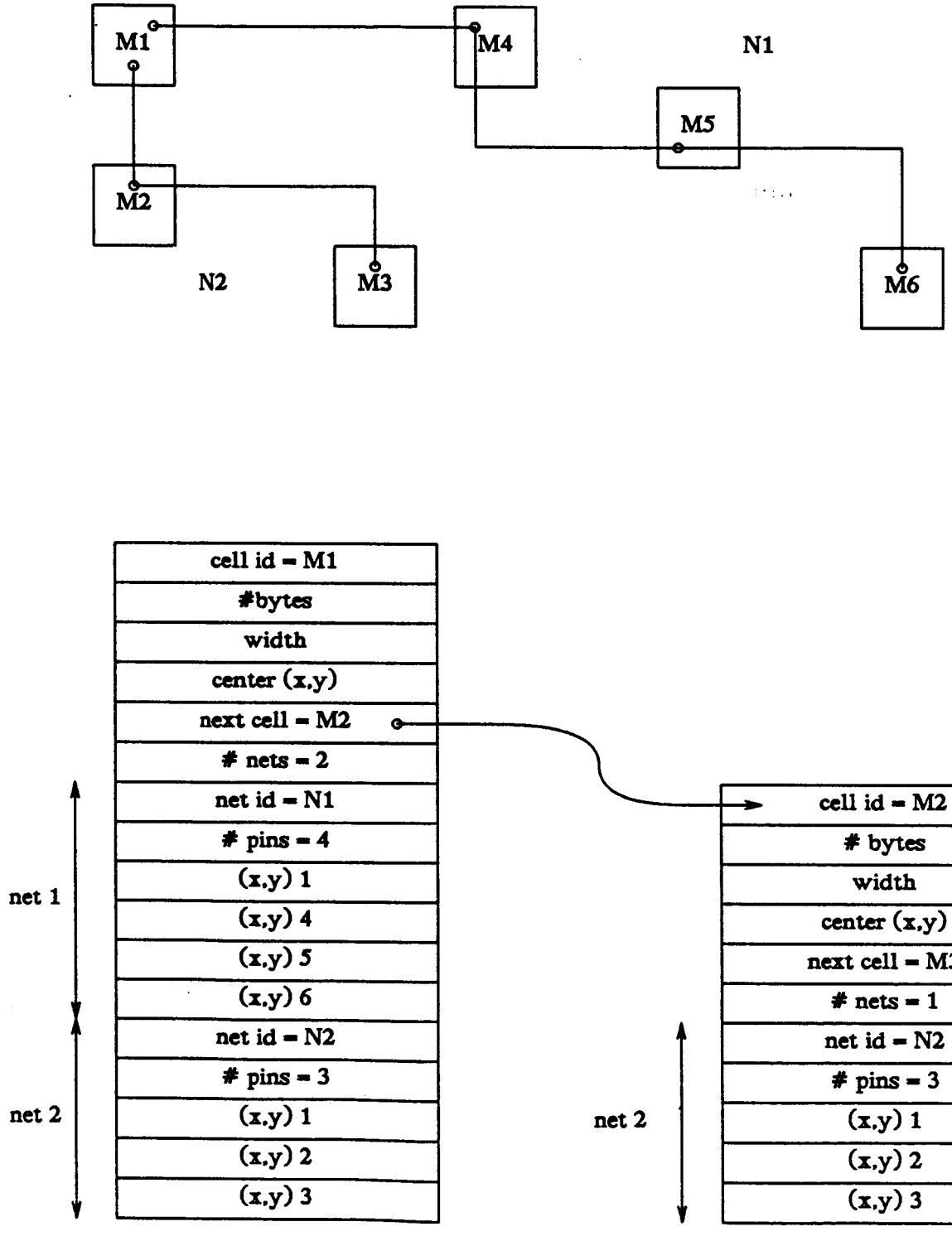


Figure 3.2. Example net and corresponding memory structure.

given row, therefore, have at least an additional 10% of length in which to move in each row. However, the cost penalty associated with going over or under the desired row length is calculated using the desired length and not the 110% length of each row.

3.6. Move Generation

After the cells have been distributed among the processors of the hypercube, each processor repeatedly interacts with its neighboring processors in each of the d dimensions of the hypercube. The set of steps involved in a parallel set of moves is outlined in Figure 3.4. At each time step, $P/2$ pairs of processors participate in the evaluating $P/2$ moves.

3.7. Discussion of Moves

3.7.1. Mastership selection

For each pair of processors (p,q) connected in dimension i, one of them is chosen to be the Master and the other to be the Slave using the criteria listed in STEP 1 of Figure 3.4 to ensure that the mastership of the pair alternates between processors in alternate iterations. The choice is not random as in [54] because it would then involve an extra synchronization message between the processors, and we wish to reduce the communication overhead as much as possible. We alternate mastership between iterations because otherwise, in a fixed scheme, we would bias the displacements of cells from the Master to the Slave processor resulting in the Master processor having no cells after several iterations.

3.7.2. Selection of move

At each iteration of the TimberWolf algorithm the generate function is performed with one of two types of cell movements randomly chosen to create a new circuit configuration for analysis. These moves are:

(1) Displacement of a single randomly selected standard cell from its present position to a randomly selected point anywhere within the physical work space

(2) Exchange in position of two randomly selected standard cells

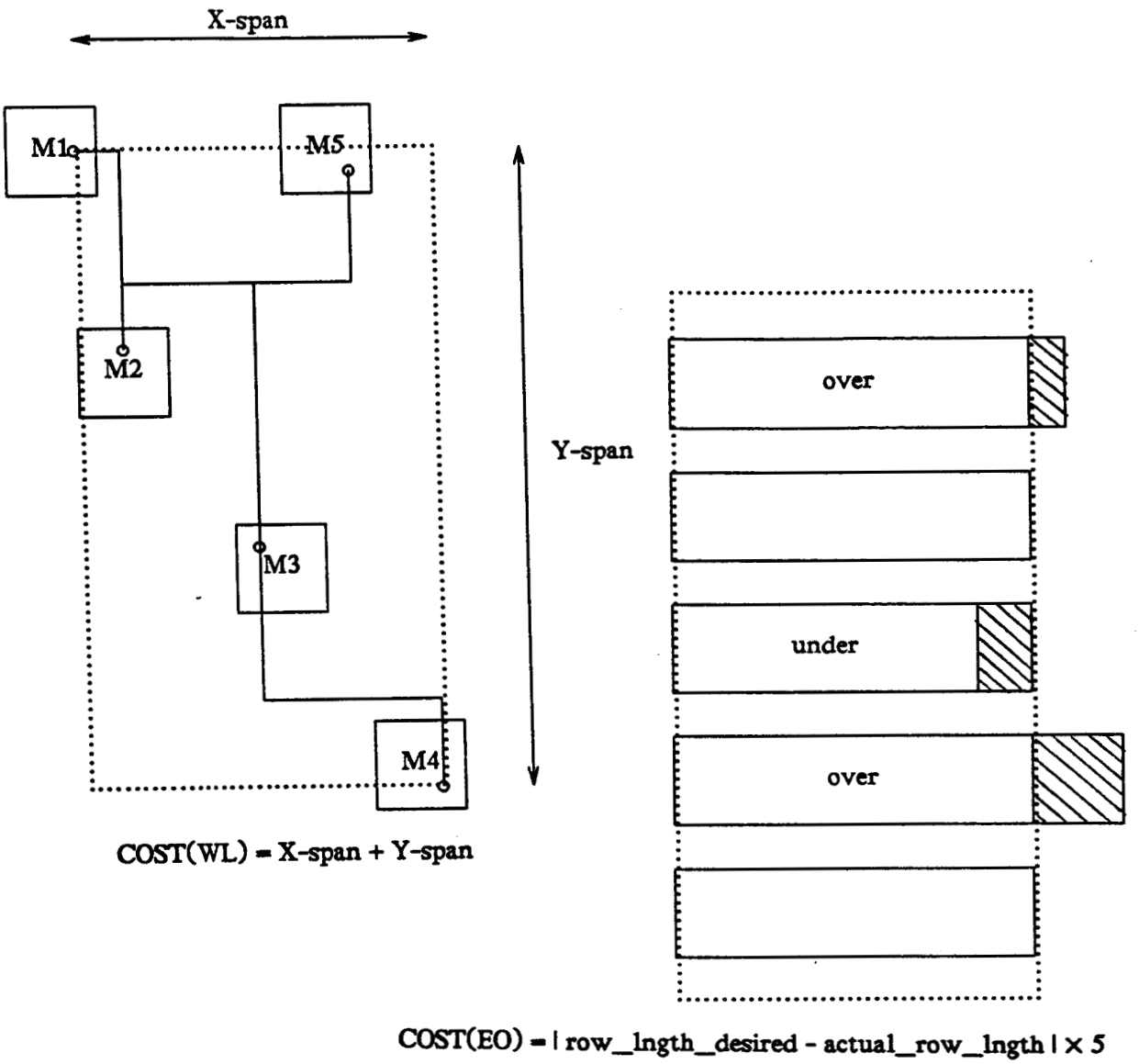


Figure 3.3. Cost function evaluation.

PROCEDURE PARALLEL MOVES:

STEP 1 For each pair of processors (p,q) connected in dimension i, if the inner_loop_count is even and if p < q, then p is chosen to be the Master, q to be the Slave, otherwise vice versa.

STEP 2 Master randomly decides if next move will be an exchange or a displacement, favoring the latter by a factor of 5 to 1. The Master also decides randomly with equal probability if the move will be an intraprocessor or interprocessor exchange/displace.

STEP 3.1 If MOVE = INTER-PROCESSOR EXCHANGE, processor p (Master) randomly selects a cell CELL(p) with (x,y)-position POS(p) within its allocated area map and sends the data structure of CELL(p) to processor q. Meanwhile processor q (Slave) also randomly selects a cell CELL(q) with (x,y)-position POS(q) within its allocated area map, and sends the data structure of CELL(q) to processor p.

STEP 4.1 Compute $\Delta_{exchange}(CELL(p),CELL(q)) =$
$\Delta_1(WL,CELL(p),POS(q),p) + \Delta_2(WL,CELL(q),POS(p),q)$
$+ \Delta_3(AO,CELL(p),POS(p),p) + \Delta_4(AO,CELL(p),POS(q),q)$
$+ \Delta_5(AO,CELL(q),POS(q),q) + \Delta_6(AO,CELL(q),POS(p),p)$
$+ \Delta_7(EO,CELL(p),POS(p),p) + \Delta_8(EO,CELL(p),POS(q),q)$
$+ \Delta_9(EO,CELL(q),POS(q),q) + \Delta_{10}(EO,CELL(q),POS(p),p)$

STEP 5.1 Processor q sends the portion of the cost function it computed to processor p.

STEP 6.1 Go to STEP 7

STEP 3.2 If MOVE = INTRA-PROCESSOR EXCHANGE, processor p (Master) randomly selects two cells, $CELL_1(p)$ and $CELL_2(p)$, both within its allocated area map.

STEP 4.2 Compute $\Delta_{exchange}(CELL_1(p),CELL_2(p)) = \Delta_1(WL,p) + \Delta_2(AO,p) + \Delta_3(EO,p)$

STEP 5.2 Go to STEP 7

STEP 3.3 If MOVE = INTER-PROCESSOR DISPLACEMENT, processor p (Master) selects a cell CELL(p) with position POS(p) within its allocated area map and sends the data structure for CELL(p) along with the portion of the cost function it has computed to processor q (Slave). Processor q selects a random position POS(q) within its area map and computes the remainder of the cost function.

STEP 4.3 Compute $\Delta_{displace}(CELL(p),POS(q)) = \Delta_1(WL,CELL(p),POS(q),q)$
$+ \Delta_2(AO,CELL(p),POS(p),p) + \Delta_3(AO,CELL(p),POS(q),q)$
$+ \Delta_4(EO,CELL(p),POS(p),p) + \Delta_5(EO,CELL(p),POS(q),q)$

STEP 5.3 Go to STEP 7.

STEP 3.4 If MOVE = INTRA-PROCESSOR DISPLACEMENT, processor p randomly selects a cell, $CELL(p)$, and a position, $POS(p)$, within its allocated area map.

STEP 4.4 Compute $\Delta_{displace}(CELL(p),POS(p)) = \Delta_1(WL,p) + \Delta_2(AO,p) + \Delta_3(EO,p)$

STEP 7 Master accepts/rejects move using exponential function $ACCEPT(DELTA,T)$
END PROCEDURE;

Figure 3.4. Parallel moves in the hypercube.

A move can be either an exchange or a displacement. Which of these is actually executed is randomly chosen by the Master in STEP 2 of Figure 3.4. The ratio of single-cell displacements to cell exchanges has a profound effect on the quality of the final placement. The best results were observed to occur when the random selection favors displacements in a ratio of approximately 5 to 1 similar to the result reported in [10]. In addition, the Master decides if the exchange or displacement move will be an intraprocessor (completely within the Master) or interprocessor (between the Master and the Slave). The best results were observed to occur when the number of intraprocessor moves is equal to the number of interprocessor moves. Orientation mirroring of cells was not implemented.

3.7.3. Cost calculation of exchange class move

We now discuss the cost function calculation for an interprocessor exchange, i.e., STEP 4.1. of Figure 3.4, which is the most complicated of all the move types. (The case of an intraprocessor exchange, STEP 4.2, is very simple.) We break up the task of calculating the cost of an interprocessor exchange move into 10 subtasks that are distributed equally among the Master and Slave processors. The first term, $\Delta_1(WL, CELL(p), POS(q), p)$ deals with the change in the wire length due to the movement of CELL(p) from POS(p) to POS(q). This is calculated by estimating the change in half the perimeter of the bounding box of each net. This term can be calculated by processor p alone, since it keeps information about all the nets to which CELL(p) is connected, along with all the (x,y) locations of cells that are on the same nets, and can read POS(q) (which is the new (x,y) location for CELL(p)) from the message sent by processor q. The term $\Delta_2(WL, CELL(q), POS(p), q)$ relates to the change in wire length due to the movement of CELL(q) from POS(q) to POS(p), and is computed in an identical manner by processor q. The term $\Delta_3(AO, CELL(p), POS(p), p)$ deals with the change in the area overlap of cells due to the movement of CELL(p) out of POS(p) and is calculated by processor p since it has information about all the cells that are near a given (x,y) location within processor p's area map. When CELL(p) is moved out of location POS(p), it may remove area overlapping of cells. The term

$\Delta_4(AO, CELL(p), POS(q), q)$ deals with the change in the cell area overlap due to the movement of CELL(p) into POS(q) and is calculated by processor q since it has information about all the cells that are near a given (x,y) location within processor p's area map. When CELL(p) is moved into location POS(p), it might create additional cell area overlap. The terms Δ_5 and Δ_6 are similar calculations for CELL(q). The term $\Delta_7(EO, CELL(p), POS(p), p)$ deals with the change in actual row length compared to desired row length (edge overshoot or undershoot) when CELL(p) is moved out of POS(p), and is calculated by processor p. The term $\Delta_8(EO, CELL(p), POS(q), q)$ deals with the change in edge overshoot/undershoot when CELL(p) is moved into POS(q), and is calculated by processor q. The terms Δ_9 and Δ_{10} are similar calculations for CELL(q).

3.7.4. Cost calculation for displacements

We now discuss how the cost function is calculated for an interprocessor displacement class move, STEP 4.3. (The intraprocessor displacement calculation in STEP 4.4 is relatively straightforward.) We break up the task into 5 subtasks that are shared between the Master and the Slave processors. The term $\Delta_1(WL, CELL(p), POS(q), q)$, computed by processor q, is the change in wire length due to the movement of CELL(p) from POS(p) to POS(q). The term $\Delta_2(AO, CELL(p), POS(p), p)$, computed by processor p, is the change in cell area overlap caused by the movement of CELL(p) out of POS(p). The term $\Delta_3(AO, CELL(p), POS(q), q)$, computed by processor q, is the change in cell area overlap caused by the movement of CELL(p) into POS(q). The term $\Delta_4(EO, CELL(p), POS(p), p)$, computed by processor p, is the change in edge overshoot/undershoot caused by the movement of CELL(p) out of POS(p). The term $\Delta_5(EO, CELL(p), POS(q), q)$, computed by processor p, is the change in edge overshoot/undershoot caused by the movement of CELL(p) into POS(q).

3.8. Annealing Schedule

In any simulated annealing algorithm, two important criteria are the choice of the initial temperature and the rate of decrease of the temperature. For the choice of the initial temperature, we

adopted the heuristic that at the initial temperatures, we should accept 95% of all moves for which there is an increase in the cost function. Hence, prior to starting the actual annealing algorithm, we calculate the change in cost functions for $10 \times N$ (N = number of standard cells in circuit) single moves within the hypercube. The average change, Δ, is calculated for those moves in which the change in cost is positive. This average cost is then used to find a proper initial temperature using the following formula:

$$T_{init} = -\frac{\Delta}{ln(0.95)}$$

The temperature of the system is then reduced after each stage of the algorithm according to the cooling schedule given by

$$T_{i+1} = \alpha(i)T_i$$

where α varies from 0.80 to 0.94 and decreases to 0.1 during the final stages of the algorithm. This variation is table-driven, as shown in Table 3.1. By using this strategy, during the initial stages of the algorithm virtually every new state is accepted and the temperature is reduced quite rapidly.

Table 3.1. Variation of alpha with temperature.

For Temperature Greater Than	α
40,000	0.80
20,000	0.84
10,000	0.88
5,000	0.91
200	0.94
100	0.90
50	0.85
5	0.80
1.5	0.70
0	0.10

During the intermediate stages of the algorithm, the temperature is reduced in such a way that the average change in cost ΔC for proposed moves is approximately equal from iteration to iteration. When the temperature is reduced below 1.5, rapid reduction in temperature is initiated in order for the system to firmly converge to a local minimum of the cost function. The final stopping criterion is satisfied when the temperature reaches a minimum value of 0.1.

In order to enhance convergence during the later stages of the algorithm, a range limiting mechanism is incorporated similar to TimberWolf [10]. For single intraprocessor displacements, a rectangular window is centered at the center of the cell to be displaced. A row is randomly selected which intersects the window and is within the locally allocated work space. A random position is then selected within that row and within the window or locally allocated work space, which ever is smaller. For proposed pairwise cell exchanges and interprocessor displacements, a move is attempted only if (1) the vertical distance between the change in movement of cell(s) is less than or equal to the vertical span of the range limiter window and (2) the horizontal distance between the change in movement of cell(s) is less than or equal to the horizontal span of the range limiter window.

Initially when the temperature is at its maximum value, the horizontal and vertical span of the range limiter window are equal to twice the span of the corresponding dimension of the physical work space. After the initial temperature is determined, the approximate number of decades, d, from zero is determined. Because it is desirable to have the window size shrink slowly, the horizontal and vertical window spans are made proportional to the logarithm base 10 of the value of the temperature. The actual formula controlling the respective window dimensions are shown below.

$$\frac{windowX}{2} = MAX[3, \frac{1}{d} xspan \log_{10}(0.2 temperature)]$$

$$\frac{windowY}{2} = MAX[3, \frac{1}{d} yspan \log_{10}(0.2 temperature)]$$

Regardless of row separation, the vertical range limiter is restricted from reducing below the distance needed for inter-row movement until the temperature of the system drops below 5.

At high temperatures during the simulated annealing process, we do not restrict the distance over which exchanges and displacements of cells can occur. Gradually, as the temperature is decreased for each processor, the range limit is also decreased accordingly until eventually certain dimensions of the hypercube are "frozen," i.e., changes between pairs of processors connected via those dimensions are effectively inhibited.

At each new temperature, the system is allowed to stabilize. This is accomplished by collectively attempting to generate a user-specified number of new states per cell at each stage/temperature of the system. For example, given a 1000 cell circuit for which a user wishes 300 attempts per cell, 300000 new states per stage/temperature will be attempted. The number of attempts per cell is directly proportional to the running time of the algorithm, and is the only user specified parameter which influences the run time. Large numbers of attempts per cell will give better placement but at the cost of excessive execution times. In general, to get the best performance to execution time ratio, Table 3.2 should be used as a guideline for various size circuits.

Table 3.2. Suggested attempts per cell for various size circuits.

Number of Cells in Circuit	Suggested Number of Attempts per Cell
<300	100
500	200
1000	300
1500	400
2000	500
2500	600
3000	700

3.9. Broadcasting New Cell Locations

Once the cells have been moved to new locations, these updated locations have to be sent to all processors so that they can update all net and pin information affected by the move. Two schemes for performing this task were investigated.

The first one uses uses the property of the existence of Hamiltonian circuits in the hypercube topology [55]. This scheme operated in the following manner. Each processor which has an updated cell location informs its Hamiltonian circuit successor of the updated value of the cell location. This processor would then inform its Hamiltonian circuit successor which would do the same. It can be easily seen that if all P processors contained updated cell locations, it will take P-1 time steps for all the updated cell locations to be available at all the processors. Figure 3.5 shows a three-dimensional hypercube with labels on processing nodes and links. Using this simple scheme, if processor 0, which is labeled M0, has an updated cell location to broadcast throughout the hypercube, a possible Hamiltonian circuit is M0, M1, M3, M2, M6, M4, M5, M7. This broadcast uses links L1, L2, L3, L12, L8, L5, and L6 requiring 7 time steps. Since each message transfer is extremely expensive, we decided to abandon this simple scheme and adopt a more complicated but extremely efficient one.

In the second scheme, each processor having a set of new cell locations broadcasts this information to all its log(P) neighbors in the first time step along its links in log(P) dimensions. In the next time step, the processors that have just received these messages from the first time step forward the messages to their own neighbors connected via links in the higher-most log(P)-i-1 dimensions where i equals the dimension of the link along which a message was received during the first time step. In the j^{th} time step, all processors receiving messages from the $j-1^{st}$ time step forward the messages to their neighbors in the higher most log(P)-i-1 dimensions where i again equals the dimension of the link along which a message was received during the $j-1^{st}$ time step. In the case of multiple initial processors wanting to broadcast modified cell locations, the messages are combined where needed at intermediate nodes before forwarding. This scheme guarantees that the

broadcasting is completed in log(P) time steps without conflicts for links. Figure 3.5 shows a three-dimensional hypercube with labels on processing nodes and links. Table 3.3 shows the steps involved in broadcasting updated cell locations from processors 1, 2, and 7 which are labeled as M1, M2, and M7 in Figure 3.5.

The entries in Table 3.3 are of the form Mi(j,k) which represents a message which originated from processor P_i during the first time step and moves from processor P_j to P_k during the current time step. For example, in time step 2, message M7(6,4), which has originated from P_7, is transmitted from processor P_6 to P_4 along a dimension 1 link. It can be verified that all messages reach all processors within 3 time steps. In case of conflicts for using a particular link at a particular time step, messages are combined. For example, in time step 2, link L9 has two messages M1(0,4) and M2(0,4) which represent messages originating from processors P_1 and P_2 but moving from P_0 to P_4 during time step 2.

A unique feature of our algorithm is that once messages are combined for transmission over a particular link, they need not be split up at intermediate nodes for transmission over separate links. The process of updating cell locations will take part at all nodes by extracting information from the received messages and using this information to modify locally affected cell structures.

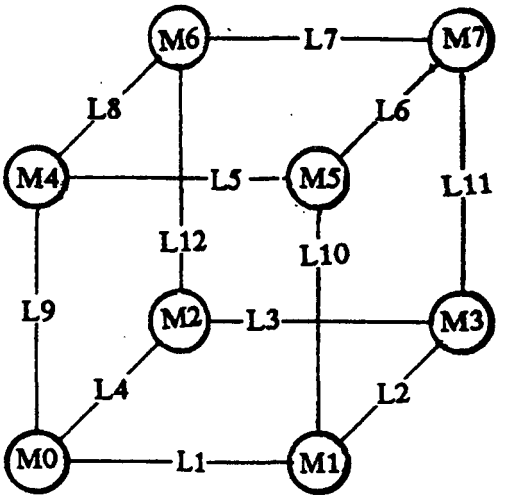


Figure 3.5. Three-dimensional hypercube.

Table 3.3. Broadcast steps for three-dimensional hypercube on a message from nodes 1, 2, and 7.

link number	STEP 1	STEP 2	STEP 3
L1	M1(1,0)		
L2	M1(1,3)	M2(3,1)	
L3	M2(2,3)		
L4	M2(2,0)	M1(0,2)	
L5			
L6	M7(7,5)		
L7	M7(7,6)		
L8		M7(6,4)	
L9		M1(0,4) M2(0,4)	M7(4,0)
L10	M1(1,5)	M7(5,1)	M2(1,5)
L11	M7(7,3)	M1(3,7) M2(3,7)	
L12	M2(2,6)	M7(6,2)	M1(2,6)

CHAPTER 4

ALGORITHM IMPLEMENTATION AND PERFORMANCE

4.1. Implementation

The advantage of our algorithm over TimberWolf is that it is much faster. We have implemented the algorithm in about 4,500 lines of C code. Due to the unavailability of an actual Intel hypercube at the present time at the University of Illinois, initial testing of this algorithm has been completed using the Intel iPSC Simulator running on a SUN 3/50 work station system under UNIX 4.2 [43]. Initial algorithm testing has only been attempted on a small scale due to excessive simulator execution times.

4.2. Placement Results

It should be noted that in the parallel annealing scheme, since we have deviated from the serial acceptance of moves, we cannot assume the convergence properties of the annealing algorithms to be valid. The theoretical convergence properties are still a subject of future research. However, we have experimented with positive results on a wide variety of standard cell circuits, some of which were randomly generated, others were obtained from industry and universities.

We will first report the the performance of our algorithm on a 16-processor hypercube using a small 64-standard cell circuit, which was randomly generated and has several clusters of cells with high connectivity. At each temperature of the annealing process, approximately 100 new states were attempted per cell. After 45 temperature reductions, the stopping criterion was satisfied with the final cell placement (Figure 4.1) showing excellent clustering characteristics. We have also implemented a uniprocessor version of the simulated annealing algorithm which is slightly simpler than TimberWolf in that the only moves that are allowed are exchanges and displacements and only standard cells are handled (no macro-cells or pads). Also as in the parallel algorithm

implementation, there is no hashing to enable fast search among cells for overlaps. The results of the final placements for that implementation are shown in Figure 4.2. Our parallel algorithm gives a final placement cost that is 10% better. The final placement cost for several standard circuits and percentage improvement in placement for the parallel algorithm over the uniprocessor algorithm are shown in Table 4.1.

The effect of the parallel simulated annealing was studied at each temperature. In Figure 4.3, it can be seen that the system cost (which can be calculated exactly in the hypercube only through additional message transfers) is a continuously decreasing function of temperature. This validates empirically that even though the acceptances/rejections of moves were performed on the basis of outdated information, our algorithm has the same general convergence property as the uniprocessor algorithm.

Figure 4.4 shows the variation of the percentage of exchanges and displacements that are accepted with temperature. It is clear that in the initial stages of the algorithm (higher temperatures), a large percentage of both types of moves are accepted. As the temperature is decreased, the percentage of acceptances of both types of moves decreases. However, at extremely low temperatures, the percentage of acceptances of displacements increases with practically no acceptance for exchanges. The increase in the acceptance of displacements is primarily due to only intraprocessor displacements being attempted as governed by the implemented range limiter.

4.3. Timing Estimates

Since we did not have access to an Intel hypercube at the University of Illinois to evaluate the speedup of our algorithm, we present here an estimate of the expected speedup. The Intel Simulator does not give any timing information for message communication, so timing has to be estimated from other sources. The running time of the algorithm depends on two separate components: Computation and Communication. We will present estimates of both in the following sections.

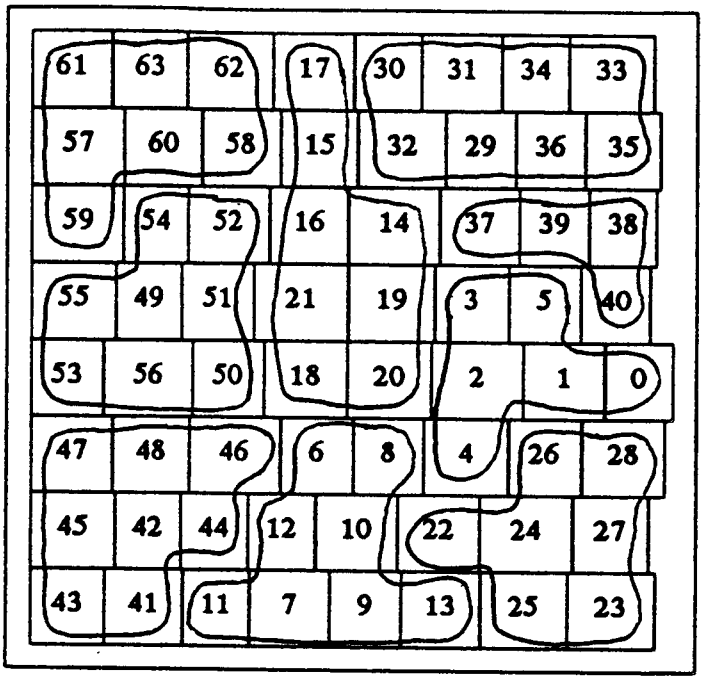


Figure 4.1. Cell placement with 16-processor hypercube

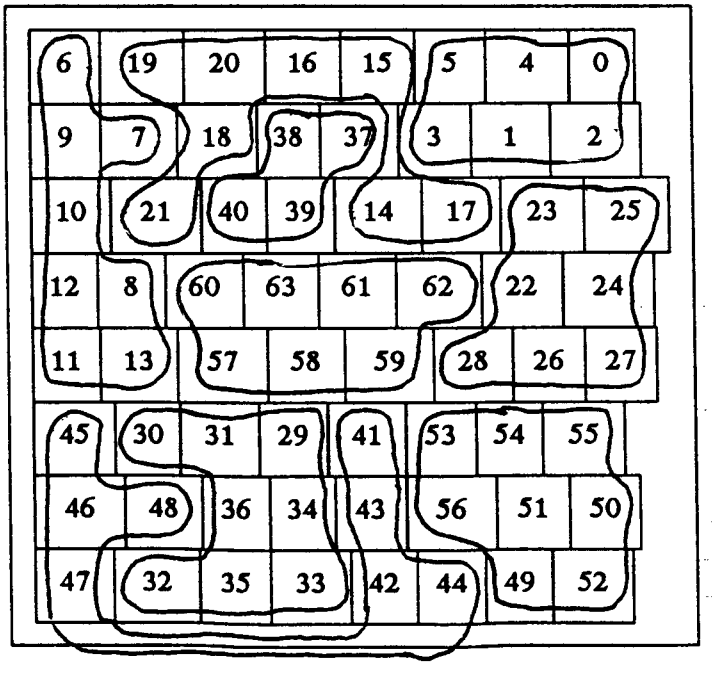


Figure 4.2. Cell placement with uniprocessor TimberWolf

Table 4.1. Placement wiring length comparison

Number Cells	4-dimensional Hypercube	TimberWolf	Percentage Change
64	29248	32135	10%
183	63094	76498	21%
286	96778	115359	19%
469	159759	195066	22%

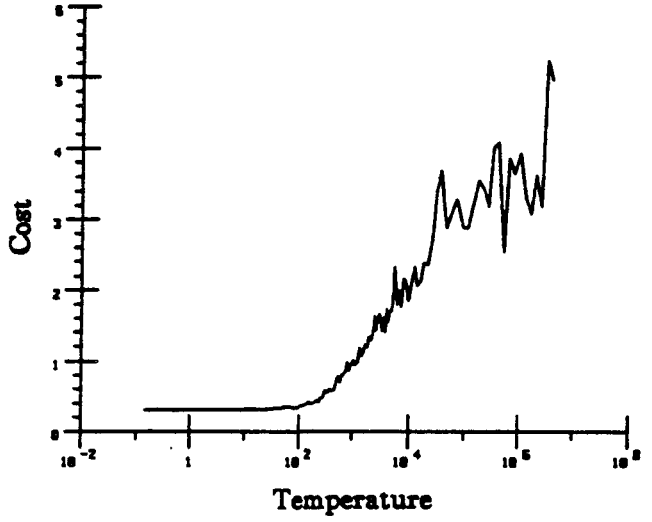


Figure 4.3. Temperature vs cost

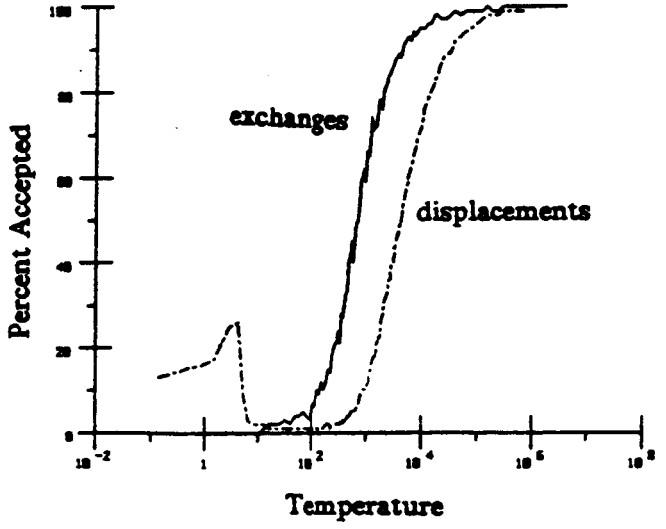


Figure 4.4. Temperature vs percentage accepted moves

4.3.1. Computation

To evaluate the computation cost per move (exchange and displacement), we implemented our algorithm on a single processor of the Intel hypercube simulator. We performed 1000 random moves of both the exchange and displacement classes and evaluated an average computation time. The results of these tests are given in Table 4.2. The CLOCK command in the simulator gives the running time on the machine on which the simulator is running, which was a SUN 3/50 work station using a Motorola 68020 CPU which is rated to be 2.7MIPs [56]. The Intel hypercube nodes consist of Intel 80286 CPUs which have been reported to be 0.78MIPs [37] or 3.5 times slower than the Motorola 68020 for the predominantly integer-oriented computation performed in our algorithm. Hence, the computation time per move on the Intel hypercube will be greater as given in Table 4.3.

4.3.2. Communication costs

We will use the results of some benchmark studies performed by Reed and Grunwald at the University of Illinois on communication costs on the Intel iPSC [38]. The results are summarized in Figure 4.5, which shows the delay in transfer of messages of varying size for simultaneous exchanges and unidirectional message transfers along a link in the Intel iPSC. We therefore need to

Table 4.2. Move timing requirements on MC68020 in milliseconds.

Number Cells	Intraprocessor Displace	Interprocessor Displace		Intraprocessor Exchange	Interprocessor Exchange	
		Master	Slave		Master	Slave
64	5	3	4	7	4	3
183	8	4	5	10	8	7
286	10	5	7	11	9	8
469	10	6	8	11	9	8
800	11	6	8	11	10	9
2357	11	7	9	13	10	9

Table 4.3. Move timing requirements on 80286 in milliseconds.

Number Cells	Intraprocessor Displace	Interprocessor Displace		Intraprocessor Exchange	Interprocessor Exchange	
		Master	Slave		Master	Slave
64	15	9	12	21	12	9
183	24	12	15	30	24	21
286	30	15	21	33	27	24
469	30	18	24	33	27	24
800	33	18	24	33	30	27
2357	33	21	27	39	30	27

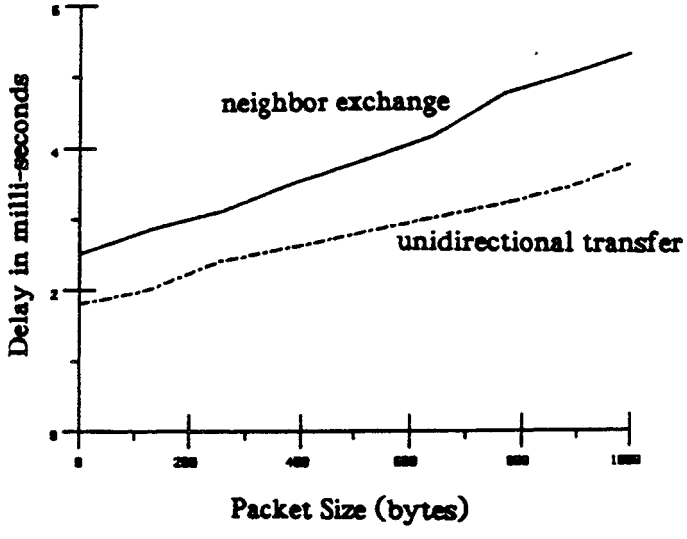


Figure 4.5. Link delay for various packet sizes.

estimate what the average packet size will be for different types of messages in order to determine communication costs. During the distributed cost calculation phase, the entire data structure for a candidate cell is sent to a neighboring processor over a single link in the hypercube. Table 4.4 shows the range of message sizes for various size standard cell circuits and corresponding required communication times derived from Figure 4.5.

4.3.3. Expected speedup

By combining these timing results and taking into account the parallelism involved in the calculation of the move cost, the time to complete each of the four types of moves was calculated as given in Table 4.5. The time required to broadcast updated cell information has been shown in Section 3.9 to require only log(P) communication steps. On the average, only 28 bytes of information are needed in each broadcast message for each individual change in a cells position. Combining of packets at intermediate nodes causes the intermediate time steps in the algorithm to be slower than the earlier and later stages. A complete broadcast cycle for a six-dimensional hypercube should require approximately 18.2 milliseconds if all nodes have a cell whose movement needs to be broadcast to the rest of the system. Unfortunately, each node in the Intel hypercube is not able to

Table 4.4. Memory usage for standard circuits.

Number Cells	Memory Usage (bytes) min	max	avg	Link Delay
64	99	688	448	2.8 ms
183	68	792	272	2.5 ms
286	36	844	214	2.4 ms
469	76	724	250	2.5 ms
800	68	1732	473	2.8 ms
2357	36	792	254	2.5 ms

Table 4.5. Estimate of time to complete the four types of. moves in milliseconds using Intel hypercube

Number Cells	Intraprocessor Displace	Interprocessor Displace	Intraprocessor Exchange	Interprocessor Exchange
64	15.0	15.6	21.0	15.6
183	24.0	18.1	30.0	27.1
286	30.0	24.0	33.0	30.0
469	30.0	27.0	33.0	30.0
800	33.0	27.6	33.0	33.6
2357	33.0	30.0	39.0	33.0

actively use all of its log(P) links at the same time due to architectural limitations. Thus, the actual number of simultaneous messages that can be transmitted/received will be somewhere between 2 and log(P). In the worst case only a single exchange of data between processing nodes can occur; hence, a complete broadcast cycle for a six-dimensional hypercube will require approximately 64.7 milliseconds.

Using these estimates, we can determine the expected speedup of our parallel algorithm over a similar uniprocessor version. If our algorithm were to be run on a six-dimensional hypercube using the 800-cell standard circuit, then at each iteration, 32 parallel moves will be attempted. It is to be expected that at least one of these moves will be an intraprocessor exchange which will be the bottleneck in terms of timing. The time to complete these 32 moves and update will be between 51.2 ms and 97.7 ms depending on update broadcast timing. For a uniprocessor version of this algorithm, the 32 moves will be distributed in a 5 to 1 ratio between displacements and exchanges. Computational time will thus be $(25.6 \times 33) + (6.4 \times 33 + 16) = 1072$ ms with the additional 16 ms added for time to complete updating of cell structures. In the hypercube, this updating is done while waiting for communications. Using these results, the estimated speedup of the Intel hypercube over the uniprocessor version will be somewhere between 11 and 21. Speedup estimates for other standard circuits are given in Table 4.6.

Table 4.6. Time to complete 32 moves in milliseconds.

Number Cells	Uniprocessor	six-dimensional Hypercube		Speedup	
		min	max	min	max
64	528	39.2	85.7	6.2	13.5
183	817	48.2	94.7	8.6	17.0
286	991	51.2	97.7	10.1	19.4
469	993	51.2	97.7	10.2	19.4
800	1072	51.2	97.7	11.0	20.9
2357	1102	57.2	103.7	10.6	19.3

CHAPTER 5

IMPROVED UNIPROCESSOR ALGORITHM

5.1. Introduction

Recently, many researchers have started to investigate speeding up simulated annealing algorithms by running them on parallel processor systems [11, 13, 14, 15, 54, 57]. Many of these parallelized placement algorithms have not only been found to be considerably faster but also to converge to a final placement which is more optimal than similar uniprocessor simulated annealing algorithms. For example, our parallel version of the simulated annealing algorithm shows an average final placement improvement of 19% over a similar uniprocessor algorithm for a variety of industry standard circuits as has been shown in Table 4.1.

The better performance of these parallel algorithms appears to be caused by the restrictions the parallel implementations place on the distances over which moves can occur and the use of slightly outdated cell placement information caused by multiple moves that interact with each other being accepted at each parallel iteration.

In the following sections of this chapter, we present an improved standard cell placement algorithm based on simulated annealing which incorporates several features inherent in a parallel processing environment. These features involve incorporating two techniques: (1) allowing multiple cell movements to be considered before updating cell placement data, thus making cost calculations based on slightly outdated placement data; (2) having the maximum range of cell movements controlled by a windowing technique which favors certain ranges.

5.2. Overview of New Algorithm

An improved algorithm can be derived which takes advantage of the performance enhancements that appear to come from parallelizing the uniprocessor simulated annealing algorithm. An algorithmic outline of this new algorithm is given in Figure 5.1.

STEP 1. Perform initial random placement of N standard cells

STEP 2. Determine initial temperature.

STEP 3. While "Stopping criteria" : temperature < 0.1 not reached

STEP 4. Generate new temperature

STEP 5. For inner_loop_count = 1 to (N × attempt_parameter)

STEP 6. Randomly select type of move (exchange or displacement)
with distance of cell movement probabilisticly determined

STEP 7. Check "range-limiter"

STEP 8. Evaluate change in cost for move

STEP 9. Accept/reject move using exponential function

STEP 10. IF the number of accepted moves is equal to limit (max_accepted)
THEN update all saved cell positions and zero number of accepted moves counter
ELSE increment accepted moves counter and save cell movements in temporary storage

STEP 11. ENDFOR;

STEP 12. ENDWHILE;

Figure 5.1. Improved simulated annealing algorithm.

The important difference between this algorithm and the previously discussed uniprocessor algorithm is that a condition has been added which controls when cell placement data are updated. Also, the generate function has been changed to allow for the distance over which moves take place not to necessarily be uniformly distributed throughout the work space. Although this algorithm appears to be identical to the parallel version, it should be noted that in the uniprocessor environment we have much more freedom in implementation over a parallel environment.

5.3. Use of Pseudoparallel Moves

In Figure 5.1, a conditional data update statement has been added which allows a multiple number of accepted moves to accumulate before an update of the circuits placement is done. This

amounts to having all moves after the first successful move to have outdated placement information on which to determine the cost function. For example, in Figure 5.2, if module M1 is successful in performing a displacement from (x_1, y_1) to (x_2, y_2) during the first iteration of the inner loop, then the circuit should be as shown in Figure 5.3, but because M1's position is not updated, the remainder of the cells still calculate cost functions which involve M1 as though it were still at position (x_1, y_1). Because of this, if module M2, which is connected to M1 via a net connection, is chosen for an attempted move during iteration two, then the half-perimeter wiring cost associated with the net will be computed using the old position of M1.

After each move acceptance, a counter is incremented to keep track of the number of successful moves, since the last cell position update and the new positions of the cells are placed in temporary storage for use later in updating the cell positions. Random cell selection for movement in subsequent iterations is not able to select cells which have made successful moves, but whose positions have not yet been updated. This amounts to freezing the cells' positions until the required

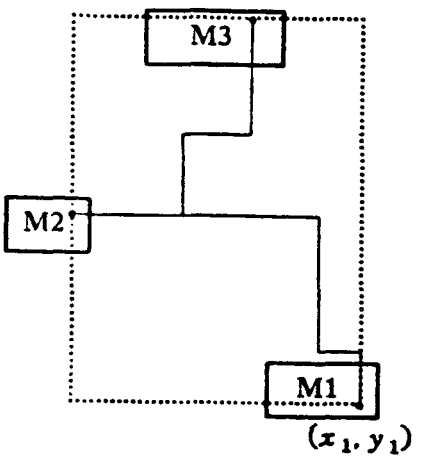
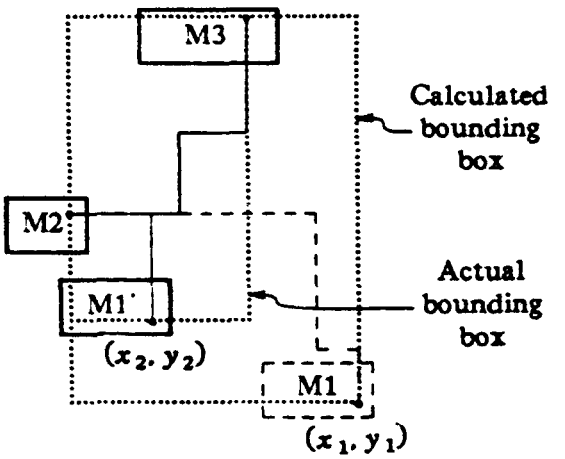


Figure 5.2. Original net placement.



5.3. Placement after initial acceptance.

number of moves has been accepted. After a specified number of successful moves, the conditional if statement criterion will be satisfied, and all cell positions will be updated using the information saved in temporary storage.

The effect of using slightly outdated information appears to give a higher probability of getting out of local minima, since this technique will accept a higher percentage of moves with uphill changes in the cost function. The **accept** function limits the magnitude of uphill moves but does not affect the total number. The number of uphill moves is affected by the random cell selection and the "observed" placement. By having the observed placement slightly different from the actual placement, it appears more uphill moves are accepted. By having greater numbers of uphill moves which are all limited in magnitude by the **accept** function, the probability of getting out of local minima is increased.

5.4. Use of Multiwindowing

Another way of increasing the number of uphill moves is to favor movement of cells over small distances. These types of moves will tend to have smaller changes in the cost function after initial clustering of cells in the first few iterations of the algorithm has been completed. In the parallel versions of the simulated annealing placement algorithm, it appears that the average distance a cell moves in the course of the algorithm has a profound affect on the final placement. Specifically, it appears that movement of cells should be biased so that movement of cells is restricted more to their local vicinity.

In most of the versions of the placement algorithm, a range limiter connected with the temperature of the system is incorporated which limits the distance any movement of cells can have. This means at high temperatures a cell will have uniform probability of moving anywhere in the physical circuit space. From observation of parallel algorithms it appears that this probability should not be uniform if optimal convergence is desired, but should favor certain distances.

Parallel simulated annealing algorithms running on message-passing architectures are constrained to certain probability distributions because of the way the cells are mapped to the

individual processors, i.e., the movement of cells from one section to certain other sections in the physical circuit space is not possible in a single move because processing nodes controlling those sections of the circuit space are not directly connected. A uniprocessor version of the simulated annealing algorithm is not constrained in this manner and thus can incorporate rather complex windowing techniques and distance probability distributions. For example, in Figure 5.4 and Table 5.1, if cell M was picked to perform a displacement, a simple triple windowing scheme could be used to determine where the cell will be displaced to. In Figure 5.4, the outermost window (W1) is always equal to the physical work space of the circuit. The inner windows, W2 and W3, have sizes

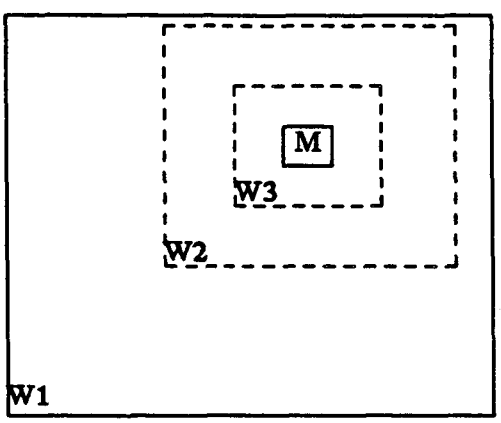


Figure 5.4. Example use of windowing in determining cell movement for cell M.

Table 5.1. Example window specifications.

Window	Fraction of Max Dimension	Example Size	Probability Within Window
W1	1	120 X 10	25%
W2	2/3	72 X 60	25%
W3	1/3	36 X 30	50%

proportional to 2/3 and 1/3 of the physical work space and are centered about cell M. In order to favor local movement, the probability of being in the innermost window (W3) is made greater than being in the outer windows. In Figure 5.4, cell M has a 50% probability that its proposed new position will be within the innermost window W3, a 25% probability of being within window W2 but not window W3, and a 25% probability of being in the physical work space but not within windows W2 or W3. Because of the affect of the continuously reducing range limiter, the affect of windowing disappears in the later stages of the algorithm when the range limiter limits movement of cells to within the confines of the innermost window.

5.5. Placement Results

The advantage of this algorithm over conventional, uniprocessor simulated annealing algorithms such as TimberWolf is that it converges to a better final placement in a given amount of time. We have implemented the algorithm in the C programming language on a Gould 9050 computer system, running under UNIX 4.2. Initial algorithm testing has only been attempted on a small scale due to excessive execution times.

The theoretical considerations of whether the annealing properties are still preserved when the cost calculations are based on slightly outdated information and when moves are not uniformly distributed, is a subject of future research. Experimentally, the improved algorithm has been shown to operate correctly for a wide range of standard industry circuits of varying sizes and complexities.

We will first report the performance of this algorithm using a small 64-standard cell circuit, which was randomly generated and has several known clusters of cells with high connectivity. At each temperature of the annealing process, approximately 100 new state moves were attempted per cell. A total of 45-temperature reductions was required before the stopping criterion of temperature <0.1 was satisfied.

Initial testing was performed to ascertain the effects of using multiwindowing and multiple moves before update. A variety of tests were run in order to derive the optimal combination of

these two techniques. The first set of tests was concerned with determining the optimal number of moves that should be accepted before cell updating is performed. Table 5.2 shows the final placement cost associated with waiting for various numbers of multiple moves before placement update. For these tests, no windowing was attempted, and thus movement of cells was unrestricted and uniformly distributed. Table 5.2 shows final placement costs generally decreasing as the number of multiple moves is increased. The optimal solution occurs when 16 moves have to be accepted before placement update will occur.

Simple testing of a few windowing schemes using 16 multiple moves showed consistent decreases in final placement cost over using windowing alone as seen in Table 5.3. Because of this, the remainder of the windowing scheme testing was performed using 16 multiple moves. A variety of windowing schemes were experimented with as shown in Tables 5.4 through 5.10. The number and size of windows for each test vary over a wide range. For example, in the ninth entry of Table 5.9 a triple windowing scheme is used with the largest window being equal to the physical work space; the second window being equal to 2/3 the size of the physical work space, and the third window equal to 1/3 the size of the physical work space. The two smaller windows are

Table 5.2. Cost vs number of multiple moves for 64-cell circuit.

Number of Multiple Moves	Final Placement Cost	Percentage Change
1	24125	+0.0%
2	24138	+0.1%
4	24003	-0.5%
8	23984	-0.6%
12	23924	-0.8%
16	23821	-1.3%
24	24173	+0.2%
32	24829	+2.9%

Table 5.3. Comparison of final cost for using or not using 16 multiple moves.

Number Windows	Window Sizes as Fraction of max	Distribution of Moves in Windows	Final Cost no multiple moves	Final Cost with multiple moves
1	1	100%	24125	23821
2	1 : ¼	20% : 80%	23872	22893
2	1 : ½	13% : 87%	24312	23654
2	1 : ¾	10% : 90%	24231	23823
3	1 : 2/3 : 1/3	10% : 30% : 60%	23890	22148
4	1 : ¾ : ½ : ¼	6% : 12% : 24% : 58%	23537	22813

Table 5.4. Comparison of cost vs distribution for (1 : ¼) double window.

Number Windows	Window Sizes as Fraction of max	Distribution of Moves in Windows	Final Cost	Percent Change
1	1	100%	23821	-1.3%
2	1 : ¼	50% : 50%	24017	-0.4%
2	1 : ¼	33% : 67%	23715	-1.7%
2	1 : ¼	25% : 75%	22783	-5.6%
2	1 : ¼	20% : 80%	22893	-5.1%
2	1 : ¼	0% : 100%	23990	-0.6%

Table 5.5. Comparison of cost vs distribution for (1 : ½) double window.

Number Windows	Window Sizes as Fraction of max	Distribution of Moves in Windows	Final Cost	Percent Change
1	1	100%	23821	-1.3%
2	1 : ½	33% : 67%	23935	-0.8%
2	1 : ½	20% : 80%	24123	-0.0%
2	1 : ½	17% : 83%	23543	-1.2%
2	1 : ½	13% : 87%	23654	-2.0%
2	1 : ½	0% : 100%	24143	+0.1%

Table 5.6. Comparison of cost vs distribution for (1 : ¾) double window.

Number Windows	Window Sizes as Fraction of max	Distribution of Moves in Windows	Final Cost	Percent Change
1	1	100%	23821	-1.3%
2	1 : ¾	20% : 80%	23911	-0.9%
2	1 : ¾	15% : 85%	23831	-1.2%
2	1 : ¾	12% : 88%	23784	-1.4%
2	1 : ¾	10% : 90%	23823	-1.3%
2	1 : ¾	0% : 100%	23741	-1.6%

Table 5.7. Comparison of cost vs distribution for (1 : 1/3) double window.

Number Windows	Window Sizes as Fraction of max	Distribution of Moves in Windows	Final Cost	Percent Change
1	1	100%	23821	-1.3%
2	1 : 1/3	50% : 50%	23798	-1.4%
2	1 : 1/3	33% : 67%	22417	-7.1%
2	1 : 1/3	20% : 80%	22098	-8.4%
2	1 : 1/3	10% : 90%	23531	-2.5%
2	1 : 1/3	0% : 100%	23784	-1.4%

Table 5.8. Comparison of cost vs distribution for (1 : 2/3) double window.

Number Windows	Window Sizes as Fraction of max	Distribution of Moves in Windows	Final Cost	Percent Change
1	1	100%	23821	-1.3%
2	1 : 2/3	25% : 75%	23431	-2.9%
2	1 : 2/3	20% : 80%	23627	-2.1%
2	1 : 2/3	13% : 87%	24015	-0.5%
2	1 : 2/3	10% : 90%	23923	-0.8%
2	1 : 2/3	0% : 100%	23815	-1.3%

Table 5.9. Comparison of cost vs distribution for (1 : 2/3 : 1/3) triple window.

Number Windows	Window Sizes as Fraction of max	Distribution of Moves in Windows	Final Cost	Percent Change
1	1	100%	23821	-1.3%
3	1 : 2/3 : 1/3	17% : 35% : 48%	23654	-2.0%
3	1 : 2/3 : 1/3	12% : 24% : 64%	23104	-4.2%
3	1 : 2/3 : 1/3	10% : 20% : 70%	23248	-3.6%
3	1 : 2/3 : 1/3	9% : 18% : 73%	23187	-3.9%
3	1 : 2/3 : 1/3	12% : 36% : 52%	22731	-5.8%
3	1 : 2/3 : 1/3	10% : 30% : 60%	22148	-8.2%
3	1 : 2/3 : 1/3	%9 : 27% : 64%	22314	-7.5%
3	**1 : 2/3 : 1/3**	**8% : 24% : 68%**	**21643**	**-10.2%**
3	1 : 2/3 : 1/3	10% : 40% : 50%	23007	-4.6%
3	1 : 2/3 : 1/3	8% : 40% : 52%	23721	-1.7%

Table 5.10. Comparison of cost vs distribution for (1 : ¾ : ½ : ¼) quadruple window.

Number Windows	Window Sizes as Fraction of max	Distribution of Moves in Windows	Final Cost	Percent Change
1	1	100%	23821	-1.3%
4	1 : ¾ : ½ : ¼	10% : 20% : 30% : 40%	23764	-1.5%
4	1 : ¾ : ½ : ¼	9% : 18% : 27% : 46%	23521	-2.5%
4	1 : ¾ : ½ : ¼	8% : 16% : 24% : 52%	23902	-0.9%
4	1 : ¾ : ½ : ¼	8% : 16% : 32% : 44%	23782	-1.4%
4	1 : ¾ : ½ : ¼	6% : 12% : 24% : 58%	22813	-5.4%
4	**1 : ¾ : ½ : ¼**	**5% : 10% : 20% : 65%**	**21032**	**-12.8%**
4	1 : ¾ : ½ : ¼	4% : 8% : 12% : 76%	22314	-7.5%
4	1 : ¾ : ½ : ¼	0% : 12% : 24% : 64%	23431	-2.9%
4	1 : ¾ : ½ : ¼	0% : 10% : 20% : 70%	22946	-4.9%
4	1 : ¾ : ½ : ¼	0% : 6% : 24% : 70%	23007	-4.6%

centered about the candidate cell for movement. The probability of a cell moving to within each of these windows but not smaller subwindows is distributed as 8%, 24%, and 68%, respectively. The final cost using this windowing scheme is 21643, which is 10.2% less than would be derived by an

algorithm which does not use multiple moves or windowing.

Several generalized results can be deduced from these tables. In Tables 5.5, 5.6, and 5.8, where the inner windows are not significantly smaller than the physical work space, the final placements tend to be decidedly inferior. This is in agreement with an earlier observation that the movement of cells should be localized to the area immediately surrounding the cell. This statement is reinforced by noting that in all the windowing schemes, better performance is generally obtained as the percentage of localized moves is increased. For example, in Table 5.7 if the percentage of moves into the innermost window is increased from 50% to 80%, a 7% decrease in the cost of the final placement results. It appears that a larger number of windows, such as in Tables 5.9 and 5.10, will give the best final placement results if the probability of movement farther away from the initial position decreases at least linearly with distance.

The effect of the modified simulated annealing algorithm was studied at each temperature. In Figure 5.5, it can be seen that the system cost is a continuously decreasing function of temperature. This validates empirically that even though we are performing the accepts/rejects on the basis of outdated information, our algorithm has the same general convergence property as the uniprocessor algorithm.

Figure 5.6 shows the variation of the percentage of exchanges and displacements that are accepted with temperature. It is clear that in the initial stages of the algorithm (higher temperatures), a large percentage of both types of moves are accepted. As the temperature is decreased, the percentage of acceptances of both types of moves decreases.

The best final placement was obtained using a quadruple windowing technique with 16 multiple moves before update. The final cell placement, as shown in Figure 5.7, shows excellent clustering characteristics. Figure 5.8 shows the final placement resulting from using no windowing or multiple moves. Even visually, the clustering in Figure 5.7 appears to be better than in Figure 5.8.

A few of the more promising windowing schemes were applied to two larger industry standard circuits of sizes 183 and 286 cells with promising results, as shown in Table 5.11.

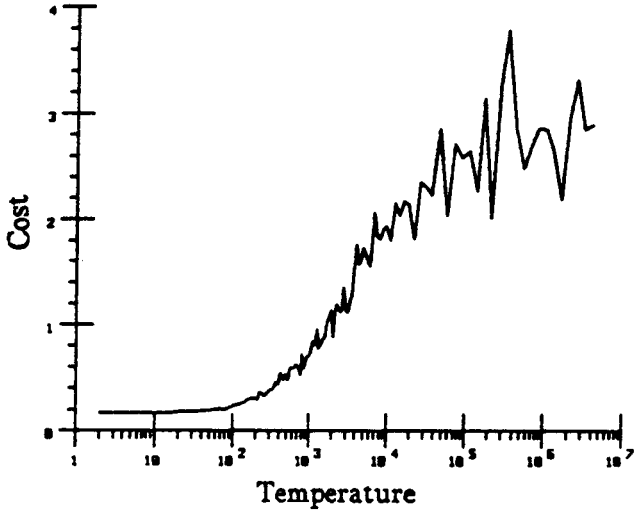


Figure 5.5. Temperature vs cost

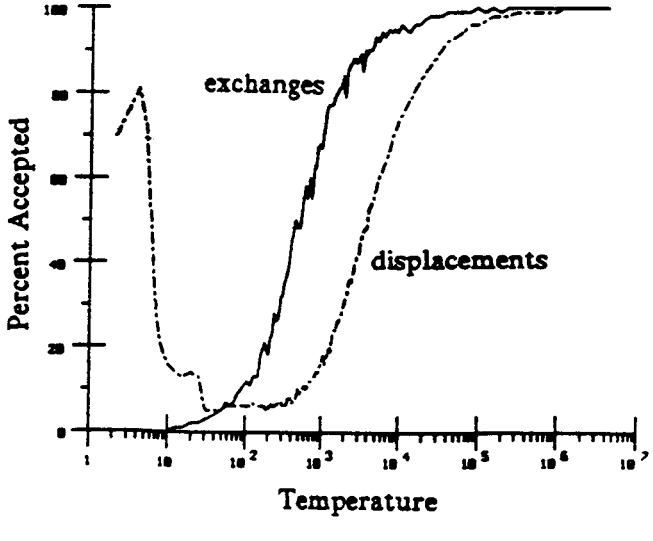


Figure 5.6. Temperature vs percentage accepted moves

Table 5.11. Comparison of cost vs windowing scheme for industry standard circuits.

Number Cells	Number Windows	Window Sizes as Fraction of max	Distribution of moves in windows	Final Cost	Percent Change
183	1	1	100%	76498	+0.0%
183	2	1 : 1/3	20% : 80%	67159	-12.2%
183	3	1 : 2/3 : 1/3	8% : 24% : 68%	69010	-9.8%
183	4	1 : ¾ : ½ : ¼	5% : 10% : 20% : 65%	650034	-15.0%
286	1	1	100%	115359	+0.0%
286	2	1 : 1/3	20% : 80%	102398	-11.2%
286	3	1 : 2/3 : 1/3	8% : 24% : 68%	102478	-11.2%
286	4	1 : ¾ : ½ : ¼	5% : 10% : 20% : 65%	98312	-14.8%

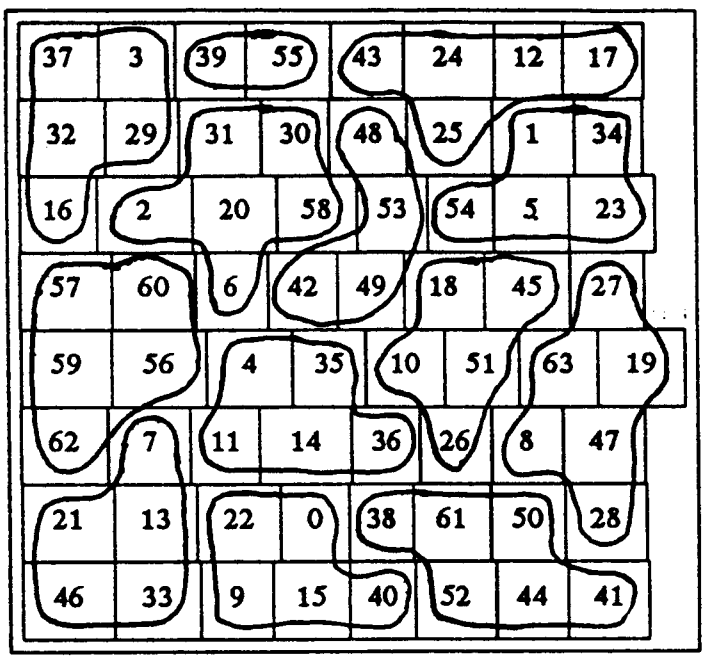


Figure 5.7. Cell placement with windowing and multiple moves

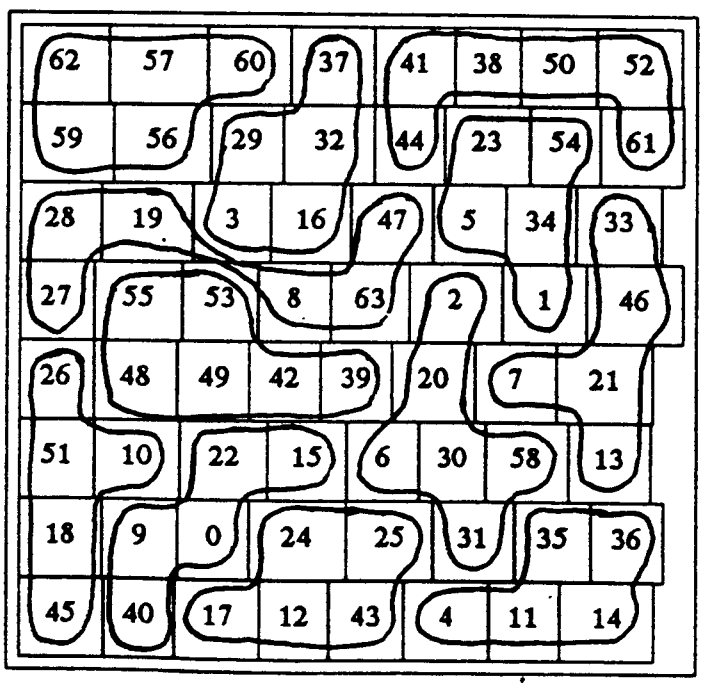


Figure 5.8. Cell placement with conventional simulated annealing algorithm

CHAPTER 6

CONCLUSIONS

6.1. Summary of Results

In this thesis, we have presented a parallel version of the simulated annealing technique for solving the standard-cell placement problem that is targeted to run in a local memory message-passing, parallel processing environment, namely the hypercube computer [54]. We have presented an improved algorithm that reduces the communication overhead, can handle more features of the placement problem, and is specifically targeted to run on the Intel hypercube. We have presented a novel tree broadcasting strategy for the hypercube that is used extensively in our algorithm for updating cell locations in the parallel environment. We have implemented the algorithm on an Intel hypercube simulator. We reported on the performance of our algorithm on actual standard cells used in industry [57]. We also presented some accurate estimates of the execution time for the algorithm. Our algorithm will not give rise to oscillations because we have a number of cells assigned to each processor, and cells are chosen randomly for possible exchange. Unlike the conventional array algorithms for module placement, our proposed algorithm will thus not get stuck at local minima. The possibility of choosing the same pair of cells for repeated exchange (oscillations) is very low. Cell exchanges can be performed among nearest neighbors through our novel area-mapping technique and also between cells that are large distances away. The results show that our parallel algorithm is not only faster but also gives better final placement results than the uniprocessor simulated annealing algorithms.

We also presented an improved uniprocessor simulated annealing algorithm based on results obtained from parallelization of the simulated annealing algorithm. We have implemented an algorithm which performs multiple moves before updating the circuit placement and uses a multiwindowing technique to generate new candidate circuit configurations. We have shown that this new

uniprocessor algorithm consistently converges to a final placement which is more optimal than conventional uniprocessor simulated annealing algorithms and does so for a variety of industry standard circuits.

6.2. Future Research

6.2.1. Hypercube algorithm

The ability to propose and evaluate moves efficiently is a requirement of any iterative algorithm. In the present version of the parallel hypercube algorithm the efficiency of computation has not been optimized as much as it can be. Future versions of the algorithm need to incorporate structures for the cells and nets which allow arrangement so that connectivity and spatial location are quickly available. A simple way to accomplish this for determining cell overlap is to sort cells into bins according to their location within a particular row. This will allow for the use of quick-sorting algorithms to isolate a smaller subset of cells which need to be checked for overlap instead of exhaustively checking all possible cells. Presently, in the updating of cell positions after a move has been accepted, an exhaustive search of all nets and pins is completed to find all references to modified cells. Additional information in the cell update broadcast packet containing the identity of all nets and cells which have references to the given modified cell and need to be updated, would decrease computation time considerable.

The present version of the parallel simulated annealing algorithm uses a simplified version of the standard cell placement problem. In particular, macro blocks, pads, and orientation mirroring movements have not yet been incorporated.

Because of the high cost of communication in comparison to computation, new strategies for reducing the amount of communication or performing simultaneous computations should be investigated. For example, since the initial exchange of full-cell specification structures is very expensive, maybe a smaller message only containing that information that is initially needed for cost comparison should be exchanged between cooperating node pairs. While this information is being

used to calculate the change in cost, the full-cell-specification structure can be transmitted simultaneously. Also, since approximately half of all moves are intraprocessor not requiring any communication overhead, the system may be able to be synchronized so that every other move is intraprocessor, and thus no communication will be required. This will reduce execution time, since on the average at least on pair of processing nodes in the system will be doing an interprocessor movement, and thus, the rest of the processing pairs will have to wait for it to complete before attempting the next set of moves.

The biggest area for future research is in attempting to execute this algorithm on an actual hypercube in order to get actual run time statistics. This information can be used to verify the expected performance of the algorithm and also may show areas for improvement that have not yet been identified.

6.2.2. Enhanced uniprocessor algorithm

The majority of the results presented for the improved uniprocessor algorithm are only based on experimentation with a small, 64-cell circuit. Although a few tests were run on larger circuits with promising results, future research needs to be done to determine the optimum number of multiple moves and windowing distribution to use in relation to the size and complexity of the candidate circuit. Performance of our uniprocessor algorithm on larger standard circuits is better than a conventional simulated annealing algorithm, but still a little less than the placements obtained when using the hypercube algorithm. It appears that a characteristic of the hypercube environment that we have not yet identified is favorably affecting the final placement results.

Even though both algorithms have been empirically shown to converge to a near-optimal final placement, further research is needed to determine if the annealing properties are still preserved when the cost calculations are based on slightly outdated information, and nonuniform distribution of moves is incorporated. More importantly, the increase in performance of our algorithms may only be due to conventional simulated annealing algorithms, such as TimberWolf, which our programs are largely based on, not using the most optimum cooling schedule or acceptance function.

APPENDIX A

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 moved 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.

APPENNDIX B

PROGRAM USERS GUIDE

The parallel simulated annealing algorithm has been implemented in the C programming language and divided into seven separate files. A UNIX shell script file named Makefile has been incorporated to aid in compilation of these files into an executable file that can be used on the Intel iPSC hypercube. This shell script can be invoked by typing the command : make *n*. The invocation of this command will result in the compilation and linking of the source code into two separate executable modules. These files will be named HOST*n* and NODE*n* and are the executable code run at the host and node processors, respectively. The parameter *n* used in the makefile invocation specifies the size of the hypercube one wants the final object code to execute on. At present only hypercubes of 4, 16, and 64 nodes have been implemented.

Compilation Parameters

Several options have been incorporated into the source code which are activated via the preprocessor #ifdef commands. The following options can be activated by enabling the definition parameter using -D*option* in the Makefile and recompiling the source code:

DEBUG: Allows additional information at each iteration regarding selection, cost, and acceptance of moves to be printed. (Simulator Only)

PRINT: Enables printing of additional placement and iteration statistics.

COST: After each temperature change the global cost of the present cell placement is determined and output.

WEIGHTED: Causes cost calculations regarding wiring to be based on formula

$$1/2 \text{ perimeter bounding box} \times MIN[1, sqrt(\text{number pins in net} - 2)]$$

instead of just 1/2 the bounding box.

CENTER: Specifies that the input file will have cells with only a single pin in each cell centered in the middle of the cell.

AUTOTEMP: Inclusion of this definition causes the algorithm to complete an initial step during which a few representative cell movements are attempted in order to find an optimal initial temperature that causes 95% of moves with increased cost to be accepted in the initial algorithm iteration. Normally, the initial temperature will be set to 4,000,000.

Input File

Input of system parameters and logic-cell-specifications is via a file named "data." The first six integer values in this file have the following meanings to the program:

1. Number of attempted moves per cell at each iteration/temperature of system.
2. Standard height of each logic cell.
3. Bytes of memory required to hold all cell-specification structures.
4. Desired length of every row of cells in circuit.
5. Number of rows of cells in circuit.
6. Desired character prefix for output file.

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

1. Unique global cell identification number (consecutively numbered from 0)
2. Cell width
3. Total number of nets cell is a member of
4. For each net specified in 3.
    a) Unique global net identification number
    b) Total number of pins in net specified in a.
    c) For each of the pins specified in b.
        i) Identification number of cell in which pin is located

        IF (preprocessor definition CENTER not defined)

            ii) X and Y location of pin relative to center of cell

Algorithm Output

The majority of the initial and final placement statistics will appear on screen or may be routed to a user-specified file via the UNIX routing commands. At the start of program execution, the cost of the initial random placement of cells will be given along with a graphical representation of the relative positions of the cells within each row. At each iteration of the algorithm, various system parameters will be outputted to inform the user of the algorithm's progress. At completion of the algorithm, the final placement cost and another graphical circuit representation will be given.

An output file named *n*.place will also be created in addition to the on screen output. The prefix *n* in this file name is the character specified by the user in the input file. This results file will be used to display the status of the algorithm at each iteration. At the completion of the algorithm, this file will contain a series of records which can be used via the UNIX command *pic* | *roff* *–me* to create an exact representation of the final placement.

REFERENCES

[1] O. Wing, "Automated gate-matrix layout," *Proc. IEEE International Symposium on Circuits and Systems*, pp. 681-685, 1982.

[2] A. D. Lopez and H. S. Law, "A dense gate-matrix layout style for MOS LSI," *IEEE Jour. Solid State Circuits*, vol. SC-15, 4, pp. 736-740, Aug. 1980.

[3] M. R. Gorey and D. S. Johnson, in *Computers and Intractability: A Guide to the Theory of NP-Completeness*. Freeman, 1979.

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

[5] M. Hanan and J. M. Kurtzberg, "Placement Techniques," in *Design Automation of Digital Systems: Theory and Techniques*, ed., M. A. Breuer. Prentice-Hall, pp. 213-282, 1972.

[6] M. Hanan and P. K. Wolff, "Survey of placement techniques," *Jour. Decision and Fault Tolerant Comput.*, pp. 28-61, Oct. 1976.

[7] M. A. Breuer and A. D. Friedman, "A survey of state of the art automation," *Computer*, vol. 1, Oct. 1981.

[8] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, "Optimization by simulated annealing," *Science*, vol. 220, pp. 671-680, May 1983.

[9] C. Sechen and A. S. Vincentelli, "The TimberWolf placement and routing package," *Proc. Custom Integrated Circuits Conf.*, pp. 522-527, May 1984.

[10] C. Sechen and A. S. Vincentelli, "TimberWolf3.2: a new standard cell placement and global routing package," *Proc. 23rd Design Automation Conf.*, pp. 432-439, Jun. 1986.

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

[12] E. Felten, S. Karlin, and S. W. Otto, "The traveling salesman problem on a hypercubic, MIMD computer," *Proc. 1985 Parallel Processing Conf.*, pp. 6-10, Aug. 1985.

[13] M. J. Chung and K. K. Rao, "Parallel simulated annealing for partitioning and routing," *Proc. IEEE Int. Conf. on Computer Design (ICCD-86)*, pp. 238-242, Oct. 1986.

[14] S. A. Kravitz and R. A. Rutenbar, "Multiprocesssor-based placement by simulated annealing," *Proc. 23rd Design Automation Conf.*, pp. 567-573, Jun. 1986.

[15] R. A. Rutenbar and S. A. Kravitz, "Layout by annealing in a parallel environment," *Proc. IEEE Int. Conf. on Computer Design (ICCD-86)*, pp. 434-437, Oct. 1986.

[16] D. J. Faber, "The distributed computer system," *Comp. Con. Digest*, pp. 31-34, 1973.

[17] K. E. Batcher, "STARAN parallel processor system hardware," *Proc. AFIPS-NCC*, vol. 43, pp. 405-410.

[18] S. H. Fuller and S. Habison, "The C.mmp multiprocessor," in *Technical Report*, Pittsberg, PA., 1978.

[19] M. Jordon, "A special purpose architecture for finite element analysis," *Proc. of International Conf. on Parallel Processing*, pp. 263-266, Aug. 1978.

[20] J. O. Tuazon, "Homogeneous array of microcomputers," *Proc. of IEEE Conf. on Circuits, Networks, and Signals*, pp. 748-752, Oct. 1980.

[21] E. P. DeBenedictis, "Multiprocessor programming with distributed variables," in *Proc. SIAM 1st Conf. on Hypercube Multiprocessors*, Knoxville, TN., Aug. 1985.

[22] G. C. Fox and A. Kolawa, "Implementation of the high performance crystalline operating system on Intel iPSC hypercube," in *Proc. SIAM 1st Conf. on Hypercube Multiprocessors*, Knoxville, TN., Aug. 1985.

[23] N. Carriero and D. Gelernter, "Linda on hypercube multicomputers," in *Proc. SIAM 1st Conf. on Hypercube Multiprocessors*, Knoxville, TN., Aug. 1985.

[24] R. L. Page and L. S. Barasch, "Parallel computation, functional programming, and Fortran 8x," in *Proc. SIAM 1st Conf. on Hypercube Multiprocessors*, Knoxville, TN., Aug. 1985.

[25] D. Jefferson and B. Beckman, "Virtual time and time warp on the JPL hypercube," in *Proc. SIAM 1st Conf. on Hypercube Multiprocessors*, Knoxville, TN., Aug. 1985.

[26] J. Barhen and E. C. Halbert, "ROSES: an efficient scheduler for precedence-constrained tasks on concurrent multiprocessors," in *Proc. SIAM 1st Conf. on Hypercube Multiprocessors*, Knoxville, TN., Aug. 1985.

[27] D. W. Krumme, K. N. Venkataraman, and G. Cybenko, "Hypercube embedding is NP-complete," in *Proc. SIAM 1st Conf. on Hypercube Multiprocessors*, Knoxville, TN., Aug. 1985.

[28] H. Sullivan and T. R. Brashkow, "A large scale homogeneous machine I & II," *Proc. 4th Annual Symp. on Comput. Architecture*, pp. 105-124, 1977.

[29] M. C. Pease, "The indirect binary n-cube microprocessor array," in *IEEE Tran. Comput.*, pp. 458-473, May 1977.

[30] N. M. Allakhverdiyev and S. S. Sarafaliyeva, "Choice of multiprocessor system configuration for digital signal processing," in *SR Report - Cybernetics, Computers and Automation Technology*, Foreign Broadcast Information Service, July 7, 1983.

[31] D. J. Evans, in *Parallel Processing Systems*. London, England: Cambridge University Press, 1982.

[32] R. W. Hockney and C. R. Jesshope, in *Parallel Computers*. Adam Hilger Ltd., pp. pp. 10, 42, 321, 323-324, 1981.

[33] C. L. Seitz, "The cosmic cube," in *Comm. ACM*. pp. 22-33, Jan. 1985.

[34] Intel Scientific Computers, "iPSC: The First Family of Concurrent Supercomputers," 1985, product announcement.

[35] *Ametek System 14 User's Guide C Edition Version 2.0*. Ametek Computer Research Div., May 1986.

[36] J. F. Palmer, "A VLSI parallel supercomputer," in *Proc. SIAM 1st Conf. on Hypercube Multiprocessors*, Knoxville, TN., Aug. 1985.

[37] Intel Corporation, "Introduction to the iAPX 286," 210308-001, Feb. 1982.

[38] D. A. Reed and D. C. Grunwald, "Benchmarking hypercube hardware and software," *SIAM 2nd Conf. on Hypercube Multiprocessors (to appear)*, Aug. 1986.

[39] J. Tuazon, J. Peterson, M. Pniel, and D. Leberman, "Caltech/JPL Mark II hybercube concurrent processor," *Proc. 1985 Parallel Processing Conference*, pp. 666-673, Aug. 1985.

[40] J. C. Peterson, J. Tuazon, D. Lieberman, and M. Pniel, "The Mark III hypercube-ensemble concurrent computer," *Proc. 1985 Parallel Processing Conference*, pp. 71-73, Aug. 1985.

[41] G. C. Fox, "Annual Report 1983-1984 and Recent Documentations," in *Caltech Concurrent Computation Project*, Jet Propulsion Laboratory, Pasadena CA., Aug. 30, 1984.

[42] J. P. Hong, R. D. Tomlinson, N. Patel, and L. H. Pollard, "A hypercube project and a simulator for a hypercube of computers," in *Proc. SIAM 1st Conf. on Hypercube Multiprocessors*, Knoxville, TN., Aug. 1985.

[43] Intel Corporation, "Hypercube Simulator Version 2.1," 310175-002, Jun. 1986.

[44] N. Metropolis, A. Rosenbluth, M. Rosenbluth, A. Teller, and E. Teller, "Equations of state calculations by fast computing machines," *Jour. Chem. Physics*, vol. 21, pp. 1087-1091, 1953.

[45] F. Romeo and A. Sangiovanni-Vincentelli, "Probabilistic hill climbing algorithms: properties and applications," in *Reprint, Dept. Elec. Eng.*, University of California, Berkeley, 1984.

[46] S. Nahar, S. Sahni, and E. Shragowitz, "Experiments with simulated annealing," *Proc. 22nd Design Automation Conference*, pp. 748-752, Jun. 1985.

[47] J. W. Greene and K. J. Supowit, "Simulated annealing without rejected moves," *Proc. IEEE Int. Conf. on Computer Design (ICCD-84)*, pp. 658-663, Oct. 1984.

[48] S. B. Gelfand and S. K. Mitter, "Analysis of simulated annealing for optimization," in *Proc. 24th Conf. on Decision and Control*, Fort Lauderdale, FL., Dec. 1985.

[49] D. Mitra, F. Romeo, and A. Sangiovanni-Vinceltelli, "Convergence and finite-time behavior of simulated annealing," in *Proc. 24th Conf. on Decision and Control*, Fort Lauderdale, FL., Dec. 1985.

[50] B. Hajek, "A tutorial survey of theory and applications of simulated daAnnealing," in *Proc. 24th Conf. on Decision and Control*, Fort Lauderdale, FL., Dec. 1985.

[51] S. Gemon and D. Gemon, "Stochastic relaxation, gibbs distribution, and the Bayesian restoration of images," *IEEE Trans. on Pattern Analysis and Machine Intelligence*, vol. 6, pp. 721-741, 1984.

[52] B. Hajek, "Cooling schedules for optimal annealing," in *Reprint, Dept. Elec. Eng., Coordinated Science Laboratory*, Chaimpaign-Urbana, 1985.

[53] B. Gidas, "Non-stationary markov chains and convergence of the annealing algorithm," *Jour. of Stat. Physics*, vol. 39, pp. 73-131, 1985.

[54] P. Banerjee and M. Jones, "A parallel simulated annealing for standard cell placement on a hypercube computer," *Proc. IEEE Int. Conf. Computer-Aided Design (ICCAD-86)*, Nov. 1986.

[55] N. Deo, in *Graph Theory with Applications to Engineering and Computer Science*. Englewoods Cliffs, N.J.: Prentice-Hall, Inc., 1974 .

[56] D. MacGregot, D. Mothersole, and B. Moyer, "The Motorola MC68020," *IEEE Micro*, pp. 101-118, Aug. 1984.

[57] M. Jones and P. Banerjee, "Performance of a parallel algorithm for standard cell placement on the Intel hypercube," *submitted 24th Design Automation Conf.*, June 1987.

REPORT DOCUMENTATION PAGE

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-87-2209 (CSG-60)	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 Avenue Urbana, IL 61801	7b. ADDRESS (City, State, and ZIP Code) NASA Langley Research Center Hampton VA 23655
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) NASA Langley Research Center Hampton, VA 23655	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO.

11. TITLE (Include Security Classification)

A Parallel Simulated Annealing Algorithm for Standard Cell Placement on a Hypercube Computer

12. PERSONAL AUTHOR(S)
Jones Mark Howard

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Technical	FROM _____ TO _____	January 87	85

16. SUPPLEMENTARY NOTATION

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	Module Placement, Uniprocessor Algorithm, Parallel Simulated Annealing Algorithm, Standard Cell Placement, Hypercube Computer

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

SEE BACK

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

ABSTRACT

A parallel processing algorithm for standard cell placement suitable for execution on a hypercube computer is presented. In the past there have been proposed several parallel algorithms for performing module placement that are suitable for execution on a two-dimensional array of processors. These algorithms had several limitations; namely, they got stuck at local minima, were susceptible to oscillation, could not handle variable size modules (standard cells), and allowed only nearest neighbor exchanges. Recently, simulated annealing, a general purpose method of multivariate optimization, has been applied to solve the standard cell placement problem on conventional uniprocessor computers. These algorithms do not get stuck at local minima and can handle modules of various sizes, but take an enormous amount of time to execute. In this thesis, a parallel version of the simulated annealing algorithm is presented which is targeted to run on a hypercube computer. A strategy for mapping the cells in a two-dimensional area of a chip onto processors in an n-dimensional hypercube is proposed such that both small and large distance moves can be applied. Two types of moves are allowed: cell exchanges and cell displacements. The computation of the cost function in parallel among all the processors in the hypercube is described along with a distributed data structure that needs to be stored in the hypercube to support parallel cost evaluation. A novel tree broadcasting strategy is used extensively in the algorithm for updating cell locations in the parallel environment. Studies on the performance of the algorithm on example industrial circuits show that it is faster and gives better final placement results than the uniprocessor simulated annealing algorithms. An improved uniprocessor algorithm is proposed which is based on the improved results obtained from parallelization of the simulated annealing algorithm. This enhanced algorithm, through the use of nonuniformly distributed moves and slightly outdated placement data, is found to be less likely to get stuck at local minima, and is found to converge to a better final placement for a variety of industry standard circuits.