PERFORMANCE OF A PARALLEL ALGORITHM FOR STANDARD CELL PLACEMENT ON THE INTEL HYPERCUBE

Mark Jones and Prithviraj Banerjee

Computer Systems Group
Coordinated Science Laboratory
University of Illinois at Urbana-Champaign

ABSTRACT

In this paper, we present a parallel simulated annealing algorithm for standard cell placement that is targeted to run on the Intel Hypercube. We present a novel tree broadcasting strategy that is used extensively in our algorithm for updating cell locations in the parallel environment. Studies on the performance of our algorithm on example industrial circuits show that it is faster and gives better final placement results than the uniprocessor simulated annealing algorithms.

1. INTRODUCTION

Given a set of standard cells of constant height and variable width, and a net list which describes the interconnections among the cells, our objective is to place the cells in a VLSI layout so as to minimize the total length of wires interconnecting the cells. The simulated annealing technique has been proposed and applied to the placement problem in a program called TimberWolf which, by applying cell displacements and exchanges randomly, avoids getting stuck at local minima and thereby achieves near-optimal placement [1,2]. A major limitation of TimberWolf is that it is extremely slow.

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 [3]. A parallel algorithm for the Traveling Salesman Problem based on simulated annealing has been reported for the Hypercube [4]. Parallel algorithms for partitioning and routing [5], macro-cell placement [6], and topological optimization of multiple level array logic [7] have been proposed by several researchers.

Two multiprocessor-based simulated annealing algorithms, called Move Decomposition and Parallel Moves, for the standard cell problem have been reported by Rutenbar and Kravitz [8,9]. Last year, Banerjee and Jones had proposed a parallel algorithm that is targeted to run on a Hypercube computer [10]. At the same time, Rose et al proposed two algorithms, called Heuristic Spanning and Section Annealing, for standard cell placement on a shared memory multiprocessor [11].

There are a number of basic differences in the previous approaches. The first approach is a shared memory algorithm that is basically simulating a serial simulated annealing environment but evaluating each individual move faster. The second algorithm is also based on a shared memory environment but 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 by Banerjee and Jones, the algorithm is based on a local memory message-passing architecture; 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, may be a subject of future research. Experimentally, it has been verified that the algorithm works. The scheme of Rose et al uses a combination of heuristic methods and simulated annealing to gain performance, and hence cannot be directly compared with the other approaches that are purely based on annealing.

A hypercube topology consists of 2^d processors that are interconnected through the topology of a cube in d dimensions. Several prototypes of such machines have been built [12,13]; two of them are now available commercially from Intel [14], and Ametek [15].

In this paper, we present an algorithm using simulated annealing on the hypercube that improves upon our earlier work presented in [10]. This enhanced algorithm reduces the communication overhead, can handle more features of the placement problem, and is more machine-specific (it is targeted to run on the Intel Hypercube). The basic idea of allowing parallel exchange and displace moves in different dimensions of the hypercube, and accepting/rejecting moves on the basis of changes in cost functions ignoring the effects of other moves remains the same. However, several new concepts were utilized that reduced the communication overhead substantially for the targeted machine.

In Section 2, we will outline the basic algorithm, 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 also present a novel tree broadcasting strategy for the hypercube that is used extensively in our algorithm for updating cell locations in the parallel environment. Section 3 describes our implementation of the algorithm on an Intel hypercube simulator. We report on the performance of our algorithm for several actual standard circuits used in industry. We show that the parallel algorithm gives about 10-20% better final placements than conventional uniprocessor simulated annealing algorithms. Finally we present some accurate estimates of the execution time for the algorithm.

2. PARALLEL ALGORITHM FOR CELL PLACEMENT

2.1. Overview of parallel algorithm

We now describe an algorithm for performing the standard cell placement using a variation of the TimberWolf [2] 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 $N >> P$.

Acknowledgment: This research was supported in part by the National Aeronautics and Space Administration under Contract NASA NAG 1-613.

```
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
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; ENDFOR; ENDWHILE
```

In the following subsections, we describe each of the steps in more detail.

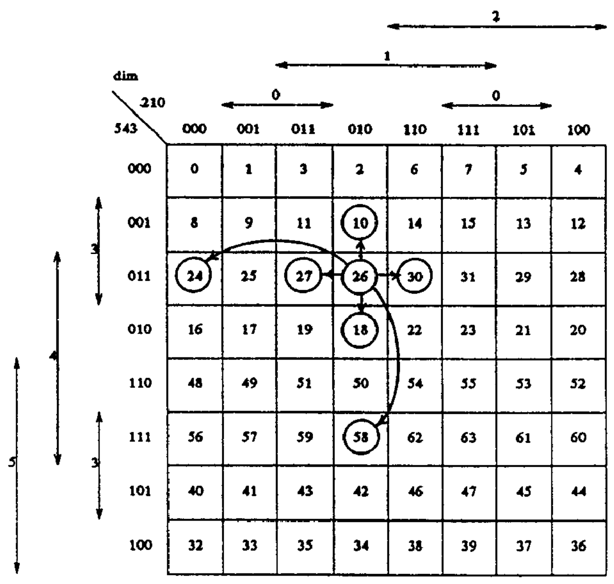
2.2. Cell Assignment to Processors

The technique for mapping a $log(P)$ dimensional hypercube onto a two-dimensional area is identical to the one described in [10]. We will briefly mention the scheme here for completeness. In a 64-processor hypercube a processor having a binary address $p_5 p_4 \cdots p_i \cdots p_0$ is connected to processor $p_5 p_4 \cdots \overline{p_i} \cdots p_0$ 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. 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. 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 Fig. 1. By choosing such a map, we guarantee that the processors that are adjacent in a pre-determined 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. This is illustrated in Fig. 1.

2.3. Distributed Data Structure

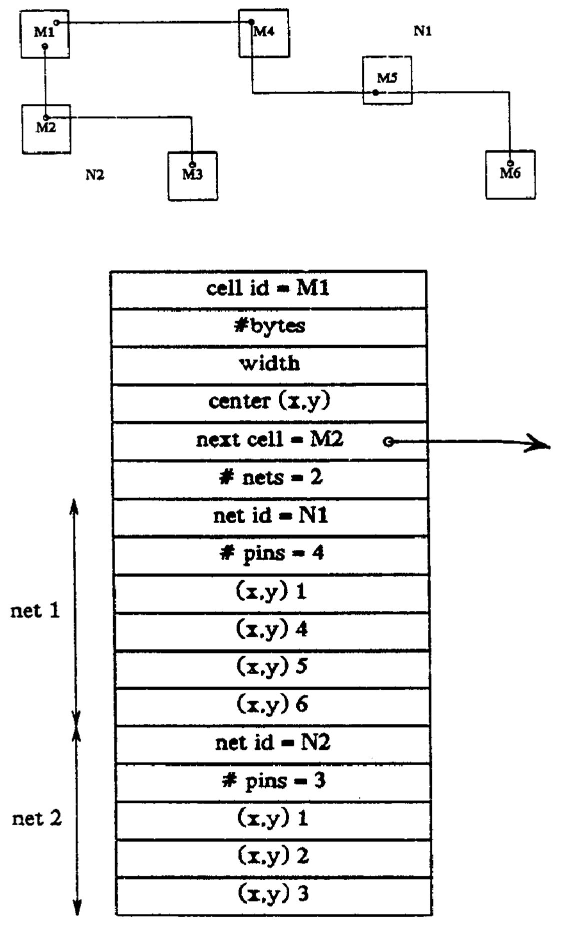
We assume that each processor contains the following information to enable the computation of the cost function in parallel among the processors in the hypercube:
(1) A list of cells currently assigned to this processor along with the following information for each cell:
(2) The width of the cell;
(3) The (x,y) coordinate location at which the center of the cell is currently placed;
(4) A list of nets to which this cell is connected;
(5) For each net listed in (4), a list of other cells, to which the net is connected, along with the (x,y) pin location(s) within these cells;
(6) 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.

The state of any particular cell is composed of the information in (2) through (5) and is packed within a continuous block of memory to allow for easy packet transfer of information between nodes. Fig. 2 shows an example of the blocked memory data structure for typical cells.



543 \ 210	000	001	011	010	110	111	101	100
000	0	1	3	2	6	7	5	4
001	8	9	11	10	14	15	13	12
011	24	25	27	26	30	31	29	28
010	16	17	19	18	22	23	21	20
110	48	49	51	50	54	55	53	52
111	56	57	59	58	62	63	61	60
101	40	41	43	42	46	47	45	44
100	32	33	35	34	38	39	37	36

Fig. 1. Area map of 64-processor Hypercube.



cell id = M1
#bytes
width
center (x,y)
next cell = M2
nets = 2
net id = N1
pins = 4
(x,y) 1
(x,y) 4
(x,y) 5
(x,y) 6
net id = N2
pins = 3
(x,y) 1
(x,y) 2
(x,y) 3

Fig. 2. Example net and corresponding memory structure.

2.4. Cost function

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 the desired row length;
(3) Area overlap between cells in the same row.

2.5. Moves

After the cells have been evenly distributed by area amongst 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 below. At each time step, $P/2$ pairs of processors participate in the evaluating $P/2$ moves.

```
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 the type of the move.

STEP 3.1. If MOVE = INTER-PROCESSOR EXCHANGE, processor p (Master) randomly selects a cell CELL(p) and sends its data structure to processor q. Meanwhile, processor q (Slave) also randomly selects a cell CELL(q) and sends its data structure 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) 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;
```

2.5.1. Discussion of moves

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 to ensure that the mastership of the pair alternates between processors in alternate iterations. The choice is not random as in [10] because it would then involve an extra 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.

Selection of move

The ratio of 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 [2]. In addition, the Master decides if the exchange or displacement move will be an intra-processor (completely within the Master) or inter-processor (between the Master and the Slave). The best results were observed to occur when the number of intra-processor moves is equal to the number of inter-processor moves.

Cost calculation

We now discuss the cost function calculation for an inter-processor exchange, i.e. STEP 4.1., which is the most complicated and of all the move types. (The other move calculations are similar). We break up the task of calculating the cost of an inter-processor exchange move into 10 sub-tasks 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 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 from location POS(p), it might remove some of the area overlaps. The term $\Delta_4(AO,CELL(p),POS(q),q)$ deals with the change in the 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 some of the area overlaps. 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).

2.6. 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:

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

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 0.8 to 0.95 and decreases to 0.1 during the final stages of the algorithm. This variation is table-driven.

In order to enhance convergence during the later stages of the algorithm, a range limiting mechanism is incorporated similar to [2]. 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. The final stopping criterion is satisfied when the temperature reaches a minimum value of 0.1.

2.7. 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 effected by the move. A very simple scheme was proposed in our earlier paper [10] which used the property of the existence of Hamiltonian circuits in the hypercube topology [16]. Each processor which had an updated cell location would inform 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 time steps for all the updated cell locations to be available at all the processors. Since each message transfer is extremely expensive (it will be shown in Section 3.2 that while a move computation takes approximately 20-30 milliseconds, a message transfer between two adjacent nodes takes about 3-4 milliseconds) we decided to abandon this simple scheme and adapt a more complicated but extremely efficient one.

In the new 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 forwards 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. Fig. 3a shows a 3-dimensional hypercube with labels on processing nodes and links. Fig. 3b shows the steps involved in broadcasting updated cell locations from processors 1, 2, and 7 which are labeled to as M1, M2, and M7 in Fig. 3b.

The entries in Fig. 3b are of the form Mi(j,k) which represents a message which originated from processor P_i during the first time step and going 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 going 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 local cell structures.

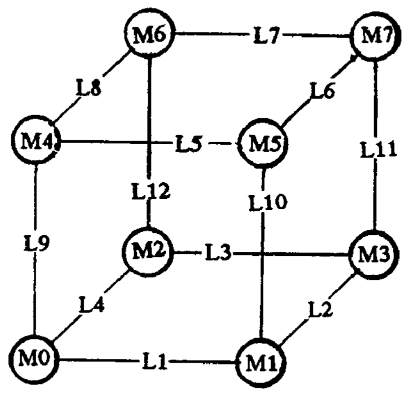


Fig. 3a. A 3-dimensional hypercube.

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)

Fig. 3b. Broadcast steps for a 3-dimensional hypercube on a message from nodes 1, 2, and 7.

3. ALGORITHM IMPLEMENTATION AND PERFORMANCE

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 workstation system under UNIX 4.2 [17]. Initial algorithm testing has only been attempted on a small scale due to excessive simulator execution times.

3.1. 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 a wide variety of standard cell circuits, some of which were randomly generated, others were obtained from industry and universities.

In Table 1, we show the results of our parallel placement algorithm on a 4-dimensional hypercube for four standard cell circuits. 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). At each temperature of the annealing process, approximately 100 new states were attempted per cell. Our parallel algorithm gives a final placement cost that is 10-20% better.

We studied the effect of the parallel simulated annealing at each temperature. We validated 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. From our studies, we observed 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 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 intra-processor displacements being attempted as governed by the implemented range limiter.

Table 1. Final placement wiring length comparison.

number cells	4-dim Hypercube	Uniprocessor (TimberWolf)	Percentage improvement
64	29248	32135	10%
183	63094	76498	21%
286	96778	115359	19%
469	159759	195066	22%

3.2. 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 our algorithm depends on two separate components: Computation and Communication. We will present estimates of both in the following sections.

Computation

To evaluate the computation cost per move (exchange and displace ment), we implemented our algorithm on a single processor of the Intel hypercube simulator. We performed 1000 random moves of both the exchange and displacement class and evaluated an average computation time. 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 workstation using a Motorola 68020 CPU which is rated to be 2.7MIPs [18]. The Intel Hypercube Nodes consist of Intel 80286 CPUs which have been reported to be 0.78MIPs[19] or 3.5 times slower than the Motorola 68020 for the types of computation performed in our algorithm. Hence, the computation time per move on the Intel Hypercube was estimated to be as shown in Table 2.

Table 2. Computation times (milliseconds) on hypercube node.

number cells	Intra Disp.	Inter Displace		Intra Exchange	Inter Exch.	
		M	S		M	S
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

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 [20]. The results are summarized in Fig. 4 which shows the delay in transfer of messages of varying size for simultaneous exchanges and unidirectional message transfers along a link . We therefore need to 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 3 shows the range of message sizes for various size standard cell circuits and corresponding communication times derived from Fig. 4.

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

Table 3. Estimation of communication costs from size of messages.

number cells	Message length (bytes)			link delay
	min	max	avg	
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. Estimate of time to complete the four types of moves in milli-seconds using Intel hypercube

number cells	Intra Displace	Inter Displace	Intra Exchange	Inter 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

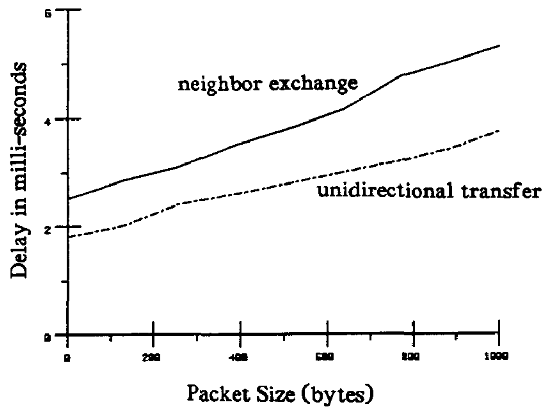


Fig. 4. Link delay for various packet sizes.

as given in Table 4. The time required to broadcast updated cell information has been shown in Section 2.7 to require only log(P) communication steps. A complete broadcast cycle for a 6-dimensional hypercube should therefore require 18.2 milli-seconds. Unfortunately, each node in the Intel Hypercube is not able to 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 6-dimensional hypercube will require 64.7 milli-seconds.

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 6-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 intra-processor 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 the other standard circuits are given in Table 5.

4. CONCLUSIONS

In this paper 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. 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. 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.

Table 5. Time to complete 32 moves in milli-seconds.

number cells	uniprocessor	6-dim. cube		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

Future research involves evaluating the algorithm on an actual Intel Hypercube. It would be interesting to explore the impact of different hypercube architectures such as the NCUBE/10, the Ametek S/14 and the Caltech MARK II on our algorithm. We are also investigating improved parallel algorithms that reduce the communication costs. Eventually, we plan to develop an integrated placement and routing package on the Hypercube.

REFERENCES

[1] C. Sechen and A. S. Vincentelli, "The TimberWolf Placement and Routing Package," *Proc. Custom Integrated Circuits Conf.*, pp. 522-527, May 1984.

[2] 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.

[3] E. H. L. Aarts, F. M. J. de Bont, E. H. A. Habers, and P. J. M. van Laarhoven, "Parallel Implementations of the Statistical Cooling Algorithm," *Integration, the VLSI Journal*, vol. 4, pp. 209-238, 1986.

[4] 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.

[5] 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.

[6] A. Casotto, F. Romeo, and A. S. Vincentelli, "A Parallel Simulated Annealing Algorithm for the Placement of Macro-Cells," *Proc. Int. Conf. on Computer-Aided Design*, Nov. 1986.

[7] S. Devadas and A. R. Newton, "Topological Optimization of Multiple Level Array Logic: On Uni and Multi-processors," *Proc. Int. Conf. Computer-Aided Design (ICCAD-86)*, pp. 38-41, Nov. 1986.

[8] S. A. Kravitz and R. A. Rutenbar, "Multiprocesssor-Based Placement by Simulated Annealing," *Proc. 23rd Design Automation Conf.*, pp. 567-573, Jun. 1986.

[9] 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.

[10] 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.

[11] J. S. Rose, D. R. Blythe, W. M. Snelgrove, and Z. G. Vranesic, "Fast, High Quality VLSI Placement on a MIMD Multiprocessor," *Proc. Int. Conf. Computer-Aided Design (ICCAD-86)*, pp. 42-45, Nov. 1986.

[12] 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.

[13] 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.

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

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

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

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

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

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

[20] D. A. Reed and D. C. Grunwald, "Benchmarking Hypercube Hardware and Software," *SIAM 2nd Conf. on Hypercube Multiprocessors (to appear)*, 1986.