# Wolverines: Standard Cell Placement on a Network of Workstations

S. Mohan and Pinaki Mazumder, *Member, IEEE*

***Abstract*—Today's typical computer-aided design environment consists of a number of workstations connected together by a high-speed local area network. Although many CAD systems make use of the network to share files or design databases, few, if any, CAD programs make use of this distributed computing resource to parallelize and speed up their work. This paper presents a placement program that makes use of this distributed computing environment to achieve linear speedup without sacrificing the quality of the results obtained by the serial version of the program. The placement program is based on the genetic algorithm, which is a heuristic search method inspired by biological evolution models. The parallel implementation has other desirable features such as the ability to operate in a heterogeneous network environment and dynamic and static load balancing. This paper describes the implementation of the placement program and detailed experimental studies of the behavior of the algorithm with various parameter settings, network capabilities, and communication patterns.**

## I. Introduction

The typical CAD environment consists of a number of workstations connected together by a high-speed network that allows a team of designers to access the design data base. Most individual CAD problems are typically solved on a single workstation. Compute-intensive jobs such as circuit simulation or maze routing are sometimes executed on a remote machine, which is more powerful than the regular workstations—this remote machine may be a supercomputer or a special-purpose computer designed to execute efficiently a single algorithm such as routing or simulation. However, the loosely coupled parallel computing environment provided by the network is not used effectively by CAD algorithms. This paper explores the use of a new class of algorithms, called *genetic algorithms*, which can make effective use of this "freely available" resource to achieve speedup by spreading the computational effort over all available processors in the network. The particular CAD problem chosen here is standard cell placement, which is a well-studied problem with many successful parallel solutions, all of which require extensive communication between the processors. The main contribution of this paper is in showing how the new problem-solving methodology can be applied to this problem to obtain an efficient parallel algorithm that runs on a network of workstations, unlike previous parallel algorithms that required special parallel machines with shared memory or dedicated interconnection networks.

Standard cell placement is a crucial step in the layout of VLSI circuits that has a major impact on the final speed and cost of the chip. This problem has been studied for several years, and the best solutions have been obtained by iterative improvement algorithms such as simulated annealing [16], simulated evolution [11], and stochastic evolution [15], which have some mechanism for escaping from local minima. Although these algorithms produce good solutions, they are typically characterized by long run times, prompting many researchers to search for parallel implementations on many different types of parallel machines.

Parallel simulated annealing has been implemented with good speedup on many different types of parallel machines, such as shared memory machines, message passing machines with local memory [10], and massively parallel machines such as the connection machine [5]. A study by Kravitz and Rutenbar [12] showed that simulated annealing on a shared memory multiprocessor could be accelerated in two ways, by performing several moves in parallel and by performing the subtasks for each move in parallel. The amount of parallelism within a move is limited, and good speedup can be achieved only by performing several moves in parallel. However, if moves are performed in parallel, the system can get into an inconsistent state unless the processors are synchronized after every set of noninteracting moves is calculated in parallel. Parallel moves are effective when the number of accepted moves is low, in the low-temperature regime of annealing; this fact has been utilized by Rose *et al.* [14] to achieve excellent speedup by using a fast min-cut approach to avoid the slow high-temperature annealing part. In the low-temperature phase, the chip area is partitioned and assigned to different processors such that each processor moves cells in a particular area and, whenever a move is accepted, broadcasts the result to all processors. There is a tradeoff between the communication costs and the need to broadcast information after each accepted move; typically several moves are accepted before a broadcast, which leads to some errors. Although error estimates for standard cell placement algorithms are not

Manuscript received June 24, 1991; revised December 28, 1992. This paper was recommended by Editor M. Marek-Sadowska. This work was supported in part by National Science Foundation Grant MIPS 8808978, URI Program Grant DAAL 03-92-G-0109, and a Digital Equipment Corporation Faculty Development Grant.

The authors are with the Department of Electrical Engineering and Computer Science, University of Michigan, Ann Arbor, MI 48109.

IEEE Log Number 9207793.

0278-0070/93$03.00 © 1993 IEEE

available, error control for the related problem of macrocell placement has been studied by various authors [1], [9], and the efficiency of the multiprocessor is reduced to less than half when the error is held within 1%. Since communication costs for simulated annealing are high even on a tightly coupled system, it is not desirable to speed up simulated annealing on a distributed system where communication time is of the order of several milliseconds.

Whereas special-purpose parallel machines required for accelerating simulated annealing or some other CAD algorithm may be available in a few CAD environments, the typical CAD environment consists of a few workstations connected together by a local area network such as ethernet. The workstations typically can support several user tasks, one of which can run in the foreground while the others run in the background. When the workstation is used for a text editing or graphics editing program, it may use its CPU only a fraction of the time and spend most of its cycles waiting for user input. The placement program described in this paper makes use of the idle cycles on all the networked machines to speed up the placement process. The communication cost is low because the communication is handled by the operating system through the network interface, and the CPU does not have to wait for the communication to complete. The cumulative error due to this mode of operation has been found to be negligible in all test cases. The program is robust and dynamically adapts to run only on the lightly loaded machines.

The genetic algorithm for standard cell placement that forms the basis for the distributed algorithm is described in the next section. The distributed algorithm is then described, followed by a discussion of the placement and speedup results obtained. This is followed by a discussion of some of the parameter settings used in running the algorithm and the performance of the algorithm in dynamically changing, heterogeneous computing environments.

## II. Standard Cell Placement Using the Genetic Algorithm

### A. Standard Cell Placement

Given a logic circuit described in terms of primitive cells and their interconnections, standard cell placement is the problem of assigning these cells of (almost) uniform height and varying widths to locations on the circuit layout such that the overall area of the layout consisting of the cells and their interconnect wiring is minimized. Usually, in addition to the layout area, several other circuit parameters, such as wiring delays, are taken into account by the placement algorithm. Standard cell placement may be modeled as a search process in the space of legal placements for a given circuit. A legal placement is a function mapping the set of cells to a set of locations $(x, y)$ such that no two cells overlap and the set of assigned $y$ coordinates has small cardinality; that is, the cells are assigned to a small number of rows. The cost function for a legal placement is the resultant chip area or netlength or the chip operating speed or some combination of the preceding. However, many standard cell placement algorithms do not generate strictly legal placements in the search process, allowing some degree of cell overlap. They use area and netlength as objective functions and include penalties for overlap of cells and for row length mismatch.

Well-known placement algorithms such as simulated annealing start with a random or loose initial placement followed by iterative improvement in which a cell is moved to a new spot or swapped with another cell in each "move." The best solutions to date have been obtained by algorithms that try to find a sequence of moves that decrease the cost functions without requiring the sequence to be strictly decreasing. This means that these algorithms can escape from local minima of the cost function by accepting subsequences of moves that increase rather than decrease the cost function. A good survey of various placement techniques following this paradigm can be found in [18].

### B. Genetic Placement Algorithm

The genetic algorithm is an entirely different approach to the placement problem. Whereas other algorithms iteratively improve a placement by moving a single cell or swapping two cells, the genetic algorithm works with a set of initial placements called the initial *population*. It attempts to combine the good features of two different placements to form a new placement. These good features, referred to as *schema*, are the relative placements of subsets of cells. Each placement of $n$ cells, also called an *individual*, is an instance of $2^n - 1$ schema corresponding to the nonempty subsets of the set of $n$ cells in the placement. The genetic algorithm repeatedly performs the reproduce-evaluate cycle as follows. In each iteration of this cycle, known as a *generation*, some of the individuals in the current population are selected probabilistically and new copies of these individuals are created. This process is known as *reproduction*, and the individuals are selected for reproduction based on their *fitness* relative to the rest of the population, with the more fit individuals getting a higher probability of reproduction. The fitness is related to the cost function, and, in the case of a function minimization problem, is typically the inverse of the cost function. The new individuals produced by the reproduction process are then subjected to the genetic operations of crossover, inversion, and mutation. *Crossover* is the most important of the operators, and it selects some schema from one *parent* individual and some schema from another parent to form a *child* individual which thus inherits schema from both parents. The crossover operation also creates many new schema that did not exist in the parents; it is the most efficient search mechanism in this algorithm. This is explained in more detail below. *Inversion* is an operator that changes the representation of the individual without actually changing the in-

dividual so that the child is more likely to inherit certain schema from one parent; an example of how this works is shown subsequently, after certain other terms are defined. *Mutation* is the analog of a pairwise interchange in conventional placement algorithms. It helps to pull the algorithm out of local minima and prevents the crossover and reproduction mechanism from replacing all the individuals in the population with multiple copies of a single fit individual. After the genetic operators are applied, the new individuals are evaluated on the basis of the cost function, and a new population is created by probabilistically selecting individuals from the old population and from among the newly created individuals, according to their relative fitness. This completes the reproduce-evaluate cycle for one generation or iteration of the genetic algorithm. The basic ideas of the genetic algorithm were inspired by the process of biological evolution, where crossover is the basis of sexual reproduction and inheritance of characteristics, while mutation induces random changes in the individual. These ideas were developed several years ago [8] and are now being applied successfully in various practical problems [6].

The single fundamental theorem of genetic algorithms is the *schema theorem*, which states that the number of instances of a schema in a given population increases from one generation to the next in proportion to its relative fitness compared with the other schema in the present population. Convergence proofs of the genetic algorithm are difficult, since many different schema are created and destroyed in a single crossover operation and the fitness of any given schema relative to the rest of the population changes every generation as the population changes. For example, the genetic algorithm is modeled as a Markov process in [4]. Each state in the Markov chain corresponds to a particular population of individuals. It is shown that when the population size and the genetic parameters (crossover and mutation rates) are held constant, a time-homogeneous Markov chain is obtained with a final stationary distribution of the probabilities of the states. However, this does not guarantee that the globally optimum solution would ever be found.

Another problem with the genetic algorithms is the choice of optimal values for the population size, the crossover rate, the mutation rate, the inversion rate, and other parameters. This problem is currently being investigated by many different researchers, but theoretical results obtained so far are for extremely restricted classes of problems, and their application to practical problems is being investigated [7]. Under certain restrictive convergence criteria, the optimal population size has been shown [7] to be 3. However, under less restrictive convergence criteria, the optimal population size is related to the length $n$ of the genetic representation of the individual as $2^n$. For a string of length 50 the optimum population size is close to 2000; for a string of length 60, it is more than 8000. The string representation of an individual in a placement problem with $n$ cells is $O(n \log n)$ bits long, rendering the theoretical results meaningless when $n$ is usually in the thousands. In the work reported here, the optimal population sizes were determined empirically, as were the other genetic parameters. Despite the lack of a good theoretical basis for the genetic algorithm, initial experiments with the genetic algorithm for standard cell placement [17] showed that this algorithm could perform as well as the then best known simulated annealing placement program, Timberwolf-3.3 [16]. At the same time, the process of migration that occurs in nature was incorporated into a parallel version of the genetic algorithm as a new genetic operator [2]. The main motivation for this present work was the recognition that this new genetic operator could be applied to design a placement algorithm with minimal communication requirements to run on a network of workstations without sacrificing the result quality obtained by the sequential or uniprocessor genetic algorithm. Hence, the rest of this paper focuses on the speedup and placement quality of the multiprocessor algorithm as compared with the single-processor genetic algorithm. The simulated annealing program that was compared with the sequential version of the genetic algorithm has subsequently been improved vastly by incorporating the results of several theoretical studies, adding heuristics and fine-tuning the algorithm to the placement problem [13]. It is expected that future work in the theory of genetic algorithms will lead to improvements in the speed and placement quality of the serial genetic algorithm. However, the parallelization studies presented here would still allow this algorithm to be accelerated by running it on a network of workstations.

### C. Basic Placement Algorithm

The basic serial algorithm for placement using just the crossover, inversion, and mutation operators is presented here. A placement configuration is represented by a string of records representing cells. Each record contains the following information.

- Cell id
- Cell position $x$
- Cell position $y$

The position of the cell record in the string does not always determine the physical location of the cell in the layout. However, at certain points in the algorithm, the cell position is recalculated based on the ordering of the cell records in the string, as follows. Starting from the first row, list the cells in order from left to right; at the end of the row go on to the next higher numbered row and list cells in reverse order, and continue until all cells are listed, changing directions after each row. Given a string of records, cells can be assigned positions in rows, by reversing the above process. The first cell in the string is assigned to the leftmost position in the first two $start_x$,

the next cell is assigned to a location $start_x + sizeof(first_cell)$ and so on, until the row is filled, and then the row number is incremented and the process goes on until all cell coordinates are assigned. There is no cell overlap, and row length variation is kept within $\pm sizeof(largest_cell)$. The area does not change with different placement configurations since the number of rows is fixed and the spacing between rows is assumed constant. Hence, the only variation is in the netlength, and the cost function is based purely on netlength.

Three genetic operators are used. They are *crossover*, *mutation*, and *inversion*. *Crossover* operates on two individuals (*parents*) at a time. It generates a new individual (*offspring*) by combining schemata from both parents. *Mutation* produces random changes in a single individual. *Inversion* is an operation that changes in effective length of a schema without altering the fitness of the individual to increase the survival probability of longer schema. In effect, it changes the representation of an individual but does not affect the underlying placement.

Crossover in its simplest form involves exchanging substrings between two individuals. However, this simple form of crossover cannot be applied here, as it can create strings that have no physical representation or logical meaning. For example, a simple crossover between two strings ABCD and BDCA could produce two new strings ABCA and BDCD. Here, each string has two repeated characters and one missing character (from the alphabet). If each character corresponds to a cell in the placement problem then a valid string should have one and only one instance of each character. Hence, more complex crossover operators that preserve the correctness of the placement are needed. A modified crossover operator known as *cycle crossover* is used herein.

Fig. 1(a) shows how cycle crossover is performed. Start with the cell in location 1 of the first parent and copy it to location 1 of the offspring. Then consider the cell in location 1 of the second parent. This cannot be inherited by the child from this parent as the position is already occupied. Hence, copy this character into the child at the corresponding location in parent 1. Now consider the character at this position in the second parent. Obviously, this cannot be inherited by the child, so let this character be inherited from parent 1. Continue this process until the new character to be considered is one that is already in the child, that is, when the cycle is complete. Then choose the first possible character in the second parent and repeat the whole process above, reading parent 2 for parent 1 and vice versa. The numbers shown below the child slots in Fig. 1(a) show the order in which the child slots get filled.

If the child inherits $x$ positions from parent 1, it inherits $n - x$ positions from parent 2. Suppose the two parents differ in $x_1$ of the $x$ positions inherited from parent 1 and in $x_2$ of the $n - x$ positions inherited from parent 2. Then the child is an instance of $(2^{x_1} - 1)(2^{x_2} - 1)(2^{n - x_1 - x_2} - 1)$ schema which do not occur in either of the parents. The child is also an instance of $2^x - 1$ schema inherited from parent 1 and $2^{n-x} - 1$ schema inherited from parent 2. Hence, each crossover affects many different schema simultaneously leading to an intrinsic parallelism in the search process of the genetic algorithm.

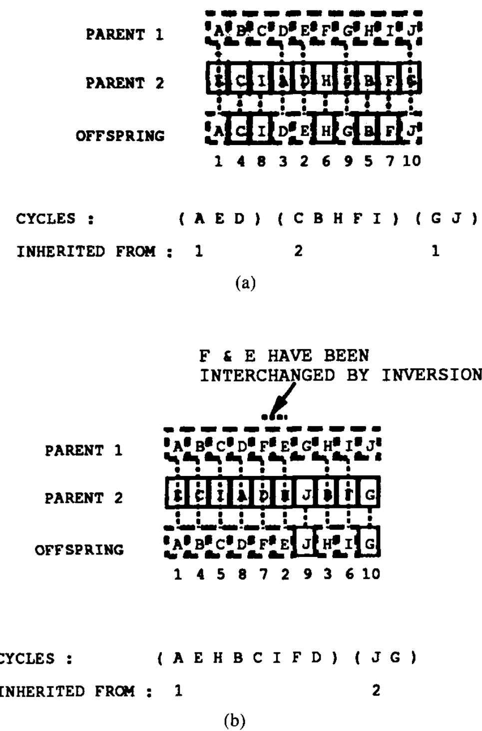



Fig. 1. Cycle crossover: (a) normal; (b) after inversion.

The inversion operator changes the positions of cell records in the string representing a placement but does not change the actual physical locations of the cell in the layout. This operator takes a string and two randomly chosen points in the string and reverses the substring between the inversion points. For example, given a string AB.CDE.F, with the randomly chosen inversion point shown as dots, the result of the inversion operation is the string ABEDCF. However, the cell position in the records is not updated at this stage, so the new string still represents the same placement. Thus an inversion operation may interchange the positions of cells E and F in parent 1 (the inverted substring is EF) [see Fig. 1(b)] but this parent 1 still represents the same placement as the parent 1 in Fig. 1(a), since the cell position records are not changed in the inversion process. However, the crossover operator now produces a totally different child [Fig. 1(b)] and a much longer schema is transferred to the child from parent 1.

The serial genetic algorithm for standard cell placement is shown in Fig. 2. An initial population of randomly generated placement configurations is the starting point of the

```
Read net list;

Create Initial Population of P
   Placement Configurations;

Evaluate all Placement Configurations
                  in the Population;

For g Generations (iterations) do

begin

  Select c_r * P configurations for reproduction

   and make copies of all of them;

  Perform Inversion operation on selected
              Placement Configurations;

  Perform Crossover on selected Placeement
   Configurations to generate new
   Placement Configurations (Offspring);

  Perform Mutation operation on selected
     Placement Configurations;

  Evaluate new Placement COnfigurations;

  Find Population average fitness;

  Select  P Configurations  (Individuals)  for
   next generation,  with fitter individuals
   getting a higher probability of selection;

end;

Write out best Placement.
```

Fig. 2. Serial placement algorithm.

algorithm. Once the population size $p$ is known (generally specified by the user, since there is no "best choice" for every case), $p$ random permutation strings over the set of integers from 1 to $n$, where $n$ is the number of cells in the circuit, are generated. These permutation strings are used to create strings of cell records with the cell-id corresponding to the particular element in the permutation string. The cell position records are then filled in as described earlier. The netlength corresponding to each placement configuration is computed, and the *fitness* value for each individual is recorded as $1/netlength$. This completes the process of creating the initial population.

The first step in the iterative loop of the serial algorithm is the selection of individuals for reproduction. This process is carried out as follows. First, the mean fitness value for the entire population is calculated as $f_{mean}$. Then the crossover rate parameter is used to determine the number of offspring or new individuals to be generated. If the crossover rate is $c_r$ and the population size is $p$, the number of offspring is $p_{off} = p * c_r$, where $c_r$ is a fraction in the interval (0, 1). Next, individuals are chosen for reproduction in a probabilistic manner, so that the expected number of offspring of an individual $I$ with fitness $f_I$ is $(f_I/f_{mean})p$. Hence, the fitness of the individual relative to the rest of the population determines the individual's chances of reproduction; the higher the relative fitness the better the chances and vice versa. Once the number of offspring for each individual is determined, the offspring are created by making copies of the string representations of the parent individuals. The offspring are then subjected to the genetic operations of mutation, inversion, and crossover, in that order. These operations have been described earlier. Each individual among the offspring is mutated with a probability that is specified by the mutation rate parameter. Then the inversion operation is performed on each individual with a probability specified by the inversion rate parameter. Finally, the individuals in the offspring population are paired off at random and crossed over. The cell positions are updated at this stage, and the fitness of all the new individuals is computed. Then $p$ individuals for the next generation are selected from this set of $p + p_{off}$ individuals. This selection is done in the same manner as before, with the probability of selection being dependent on the relative fitness of the individual.

The individual with the highest fitness in the final population is the solution point or the best placement. The fitness of an individual is a function of the netlength associated with the placement: the higher the netlength, the lower the fitness. The initial population is chosen randomly. Hence repeated runs of the algorithm could produce different solutions. Selection of individuals for the next generation is based on the fitness criterion: the higher the fitness of an individual relative to the population average fitness, the higher its probability of survival into the next generation. The number of offspring to be generated and the fraction of the population to be mutated are decided by the values assigned to the genetic parameters *crossover rate* and *mutation rate*, respectively. Optimal values for these parameters were obtained by using metagenetic optimization [17]. Typical values of various parameters are $p = 48$ (population size), $c_r = 0.33$ (crossover rate), $\mu_r = 0.05$ (mutation rate), $i_r = 0.15$ (inversion rate), $g = 30\,000$.

The termination condition for the genetic algorithm is typically the identification of convergence. The population is said to have converged when all the individuals have the same genetic representation, that is, the entire population consists of multiple copies of the same individual. This condition was rarely observed in our experiments, except when the population size was reduced to the barest minimum for effective crossover (3). Hence the primary termination conditions were determined empirically, based on the rate of improvement in fitness.

### D. Analysis of Serial Algorithm

Let the run time of GASP, the serial genetic algorithm, be represented by $\tau_{serial}$. This time is a function of the population size $p$, the number of generations $g$, the problem size $n$ (number of cells and nets), and the values assigned to the genetic parameters. Since the genetic parameters are the same for both the serial algorithm and the parallel versions to be discussed later, they are not shown explicitly in the following equations.

$$\tau_{serial}(n, p, g) = t_{init}(n, p) + g * (t_{rep}(n, p) + t_{eval}(n, p)) + t_{out}(n) \quad (2.1)$$

$$t_{init}(n, p) = t_{read}(n) + t_{setup}(n, p) \quad (2.2)$$

$t_{read}$ is the time required to perform the first step in the algorithm, reading in the circuit, and is a function of the

circuit size. $t_{setup}$ is the time required to create the initial population and is a function of both the circuit size and the population size. $t_{rep}$ is the time required to perform all the genetic operations (for one generation) and create a set of new individuals. $t_{eval}$ is the time required to evaluate the cost function for each of the new individuals and to select the survivors into the next generation. It is known that the largest single time factor in (2.1) is $t_{eval}$, and $g * t_{eval}$ is about 75% of $\tau_{serial}$. $t_{out}$ is the time required to write the best results out on to the disk, and is a function of circuit size. The typical breakup of the time spent by the serial algorithm on various tasks is evaluation 77%, reproduction/crossover 22%, and initialization and input-output about 1%.

Hence the benefits of parallelizing the reproduction and evaluation functions are obvious. Evaluation can easily be parallelized by evaluating each new individual on a separate processor or by evaluating different aspects of the cost functions on different processors. On a distributed memory system this would entail the overhead of passing $\Theta(p * n)$ data over the communication medium in each generation. The reproduction/crossover phase begins with the selection of individuals, which is an inherently sequential process that reduces the achievable speedup. The actual crossover operation can be parallelized in an obvious way but the communication cost remains.

The rest of this paper is devoted to presenting a simple, yet effective parallelization technique that avoids the communication costs associated with trivial parallelization schemes and achieves maximum speedup while producing solutions of comparable quality to the serial algorithm.

## III. Distributed Placement Algorithm

The distributed placement procedure runs a basic genetic algorithm on each processor in the network and introduces a new genetic operator, *migration*, which transfers placement information from one processor to another across the network. Migration transfers genetic material from one environment to another, thereby introducing new genetic information and modifying the new environment. If the migrants are fitter than the existing individuals in the new environment they get a higher probability of reproduction and hence their genetic material is incorporated into the local population. When the population is very small it tends to converge after a few generations, in the sense that all the individuals come to resemble one another. Migration prevents this premature convergence or inbreeding by introducing new genetic material [2]. Hence, the genetic algorithm may be modified by splitting the large population over different processors and using the migration mechanism to prevent premature convergence.

Although the idea of a distributed floor-planning algorithm using the migration mechanism was suggested in [3], that work did not study the speedup since the floor-planner was presented as a distinct algorithm in itself, rather than as a parallelization of a given algorithm. Furthermore, the floor-planner had a synchronization requirement, whereby a processor had to wait for migrants from some other processor. This requirement increases the worst case effective communication time to the actual physical time required to transmit data from one machine to another across a shared network. This time can be large and can reduce the speedup by a significant factor. Our work avoids the synchronization problem by delegating the communication work to the operating system/communication processor, and by also not waiting on a *read* operation. Results presented here show that the result quality of the parallel algorithm is not degraded significantly by not waiting on *read* operations, whereas the speedup achieved is almost linear in the number of processors used because the communication time is now vastly reduced. Furthermore, a detailed experimental study of the migration mechanism is done. The statistical variations inherent in the algorithm are thoroughly studied. It is shown that the distributed network environment can actually be used to speed up a successful placement algorithm to a significant extent.

The basic algorithm, which runs on each processor in the network, is shown in Fig. 3. The genetic operators—crossover, invert, and mutate—are the same as those used in the serial algorithm [17]. The new feature here is the migration mechanism, which combines the algorithms running on different processors into a single distributed genetic algorithm. The migration mechanism transfers some individuals (placement configurations) to other processors, once every few generations. The *epoch length* is defined as the number of generations between two successive migrations. The effectiveness of the migration mechanisms depends on several factors, such as the number of individuals transferred in each migration, the selection of the individuals for migration, and the epoch length. Too much migration can force the populations on different processors to become identical, making the parallel algorithm inefficient, whereas too little migration may not effectively combine the subpopulations. This has been observed experimentally, as detailed in the next section.

The processing time $\tau_K$ with $K$ processors on the network may be written as

$$\begin{aligned}\tau_K = {} & t_i(K) + t_{init}(n, p/K) \\ & + n_e * (e * (t_{rep}(n, p/K) + t_{eval}(n, p/K)) \\ & + n_{migr} * t_{comm}(n, K)) + t_{out}(n, K) \qquad (3.1)\end{aligned}$$

where

$K$ = Number of processors/workstations
$p$ = Total population size
$n_e$ = Number of epochs
$e$ = Epoch length
$t_{rep}$ = Time to perform reproduction/crossover operations
$t_{eval}$ = Time to evaluate population fitness

```
read net list;

Create Initial Population of p
Placement Configurations;

Evaluate all Configurations (Individuals);

for n_e epochs do {
   for e Generations (iterations) do {
    Select c_r * P configurations for
    reproduction and make copies of them;

    Perform Inversion operation on
            selected Individuals;

    Perform  Crossover  operation on
     selected Configurations to generate
     new Configurations (Offspring);

    Perform  Mutation  operation  on
     selected Offspring;

    Evaluate Offspring and find Population
      average fitness;
   }

   Select and write n_migr Individuals on to network;
   Read 0-n_migr Individuals from network

   Select p Individuals for next generation
    with fitter Individuals getting a
    higher probability of selection;
}
write out best Placement Configuration.
```

Fig. 3. Parallel algorithm.

$n_{migr}$ = Number of individuals transferred at each migration
$t_{comm}$ = Time to transfer one individual to the communication subsystem
$t_i$ = Time to start up the program on $K$ processors
$t_{init}$ = Time required to read the netlist and create and evaluate the population
$t_{eval} >> t_{comm}$
$n_e * e = g$ = Number of generations

It may be seen that there is an extra time factor $t_{comm}$ not seen in (2.1), and the reproduction and evaluation times are now functions of a smaller population assigned to each processor. As already observed in the case of the serial algorithm, the initialization and output times are negligible and may be safely ignored. The reproduction and evaluation times are both linear functions of the population size $p$ and, hence, scale linearly when the population is divided among the available processors. Hence, speedup of the parallel algorithm depends on the total communication cost, which, in turn, depends on the migration rate, the epoch length, $e$, and the number of migrants $n_{migr}$. The minimum amount of information that has to be transferred in the migration process, contains for each cell in the placement configuration, its location ($x$, $y$) and a serial number used by the inversion operator. In addition, the cost of each configuration is also transmitted as the extra communication time required to transfer one number is small, compared with the time required for cost function evaluation. Hence, the communication time is a function of the problem size $n$, the number of migrants $n_{migr}$, and the epoch length $e$. In a typical workstation, the actual communication is handled by a separate ethernet subsystem, and the CPU continues processing after downloading the data to the communication subsystem for a *write*. The *read* process takes place when the CPU finds data waiting at the communication subsystem and reads it; if there is no data waiting to be read, the CPU does not wait for the data to become available and continues with the next step in the algorithm. Hence, the actual communication time for the CPU is reduced to the time required to transfer data to and from the communications subsystem, and measurements have shown this to be negligible. However, the communication pattern does load the network, and the tradeoff is between network congestion and closer coupling bewteen subpopulations on different processors. It has been empirically observed that there is an optimum level of coupling between subpopulations, and the network congestion factor at this communication rate is low (see next section).

If the total time for communication and assimilation of the migrants $n_{migr} * t_{comm}$ is small compared with the total time for performing all other actions in one epoch, the speedup is close to ideal. With increasing $K$, the speedup increases but the efficiency goes down and the network traffic increases. The total network traffic $T$ due to $K$ processors is $C_2 * K * n/\tau_e$, where $C_2$ is some constant, $n$ is the number of cells in the circuit to be placed, and $\tau_e$ is the time for one epoch which is given by $\tau_e = e * (C_1/K * (t_{rep}(n, p) + t_{eval}(n, p)))$. In other words, for a given circuit and population size, $\tau_e = C_3/K$; $T = C_4 * K^2$. Here $C_1$ is the linear scaing factor for the reproduction and evaluation ties considered as functions of the population size. Observations indicate that $C_1$ is close to 1 (see Section IV). $C_3$ is derived from $C_1$ for a fixed population size and circuit. $C_2$ is related to the amount of information transmitted during each migration. Hence the traffic increases as $K^2$ and using empirical values for the constants in the above equations and assuming a network capacity of $1.25 \times 10^6$ bytes/s (10 Mb/s ethernet) speedup of close to 25 can be achieved with up to 25 processors with a 50% load on the network. Speedup of 8 can be achieved with eight processors and less than 10% loading of the network. With a large number of processors, the initial loading on the network is high, but after a few epochs the loading tends to stabilize to its average value.

The major results of this analysis follow:

- If the communication time is small, the speedup is close to ideal.
- The implementation of the communication mechanism ensures that the communication time is always small, and close to ideal speedup is guaranteed.
- The speedup can be increased by increasing the number of processors, but this results in increased network loading, and up to 25 processors may be used without unduly loading the network.
- The network loading factor depends on the genetic parameters as well as on the problem size.

It may be noted that the speedup is specified in terms of the time required to perform a fixed number of genetic

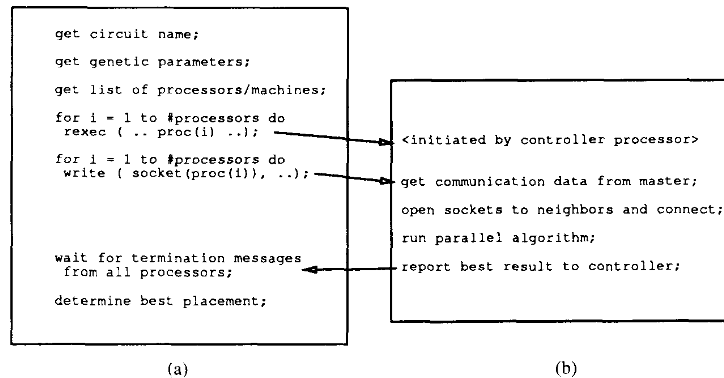



Fig. 4. Implementation of Distributed Algorithm: (a) master controller; (b) processor.

operations and fitness evaluations on constant total population. The migration mechanism is assumed to ensure the result quality, and empirical observation of this is presented in the next section. An alternative measure of speedup that considers the time taken to achieve a given fitness value is also measured in the next section.

### A. Implementation Issues

The distributed placement algorithm has been implemented on a network of workstations (Sun systems). To facilitate the evaluation of various topologies, the program has been organized as follows: there is one master program that provides the user interface and also manages all other processes (processors). The master program uses the "rexec" facility to create processes on different systems. Processors communicate by using the socket mechanism. The master processor determines the communication pattern. In the first version of the program, in order to explore different communication patterns, the master program also acts as the routing controller and routes all messages between any two processors. Hence, in this version, actual physical communication is between the master processor and the other processors only. However, with a fixed routing pattern, there need be no communication bottleneck, and the steps executed by the master and slave processors are shown in Fig. 4.

The master controller obtains from the user, the circuit name and the processors or machines to use. It then uses the *rexec* function to start the slave programs on all the specified machines. Once this is done, the controller sends each processor some information about the other processors with which it should communicate and then goes to sleep. Each processor executes the parallel algorithm of Fig. 3 and sends a message to the master controller before terminating. When all the processors have terminated, the master controller identifies the best overall solution. The data structures used in the genetic algorithm running on each processor are the same as those used in the serial algorithm and described in Section II.

Both the master controller and the slave programs are written in C, and make use of the *rexec* and *socket* functions provided by the UNIX system. Typically the master controller runs on the same machine as one of the slave programs since there is not much overlap between the two. All experimental observations have been made with the machines running their normal daily loads in addition to the distributed placement algorithm.

## IV. Experimental Results

The following major issues in the behavior of the distributed algorithm were studied experimentally.

- *Result Quality:* The placement results of the serial algorithm were compared with the multiprocessor placements to determine if the implementation of the migration mechanism was effective. Multiple runs with different circuits showed that the result quality was the same.
- *Statistical Variations in Result Quality:* The serial and parallel versions of the placement algorithm were run many times, and the mean and standard deviations of the final results were computed. It was found that the migration mechanism managed to preserve the essential characteristics of the serial algorithm results while providing good speedup.
- *Speedup:* Speedup was measured in two ways. The first speedup was based on the time required to examine a fixed number of configurations (3.1). By keeping the communication costs low, this speedup was measured to be close to ideal. At the same time, the result quality was also monitored to ensure that the speedup was not at the cost of result quality. The second speedup calculation was based on the statistical observations, showing that for any desired final result achievable using the serial algorithm, the parallel version could provide linear speedup (close to $K$ for $K$ processors).
- *Migration:* The effect of migration rate variations and the interplay between migration and the crossover mechanism were studied. Good parameter settings for migration and crossover were identified.
- *Communication Patterns:* Whereas the bus-based network provides full connectivity, allowing any

machine to communicate with any other machine, the effects of the actual communication pattern on the result quality and speedup were studied. It was seen that as the number of processors was increased, an efficient communication pattern was required to make the migration mechanism effective.

- *Coping with Network Heterogeneity:* The effects of wide differences in machine capabilities and loads on the performance of the placement algorithm was studied. It was seen that the algorithm was robust. Static and dynamic load balancing schemes were implemented.

All experiments were performed on the set of standard cell circuits containing from 100 to 5814 cells, including the following eight circuits used in benchmarking the serial genetic algorithm [17]: *cktA* 100 cells, *cktB* 183 cells, *cktC* 469 cells, *cktD* 752 cells, *cktE* 800 cells, *cktF* 2357 cells, *cktG* 2907 cells, *cktH* 5814 cells.

### A. Convergence and Result Quality

Fig. 5 plots the best netlength obtained against the generation number, for the genetic algorithm running on one, two, four, and eight processors. These placement results are for the circuit *cktA*. Similar results were obtained for the other circuits. It was seen that in all the plots, as in Fig. 5, there was a region of rapid improvement followed by a region of more gradual improvement. The total population and the total number of configurations examined were kept constant in all these experiments. Thus for "$K = 1$," the uniprocessor case, the population size was 48, while for "$K = 8$," the eight-processor case, the subpopulation on each processor was six, making the total population 48. The other genetic parameters, such as crossover rate, mutation rate, inversion rate, and epoch length, were the same on all processors. The curves for the uni- and multiprocessor experiments all lie close together for all the circuits. To get a more accurate comparison of the results, the difference in the best netlength obtained in the uniprocessor and multiprocessor experiments in each generation has been plotted.

Fig. 6 plots $\Delta netlength = (netlength_1 - netlength_K) * 100/netlength_1$ as a function of generation number for $K = 2, 4, 8$. It can be seen that $\Delta netlength$ is initially negative. This means that, in the initial stages, the uniprocessor algorithm does slightly better than the multiprocessor algorithm. However, $\Delta netlength$ becomes positive later on for all three cases in this experiment. It was observed in some other experiments that the $\Delta netlength$ varied somewhat from $+5\%$ to $-5\%$. To test the hypothesis that this variation was due to the randomness inherent in the algorithm and not due to the migration mechanism itself, the serial algorithm was run repeatedly (1000 times), and the results obtained were compared with the results from repeated runs of multiprocessor algorithm running on eight processors. Fig. 8 shows the scaled cost function on the $X$ axis, and the fraction of the runs in which that cost function was the result is shown on the $Y$ axis. It can

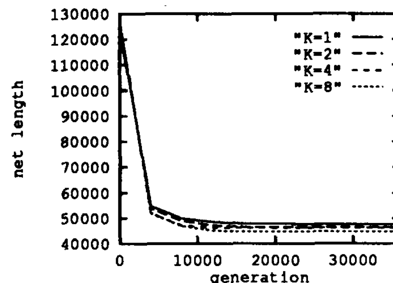



Fig. 5. *cktA* convergence on multiple processors.

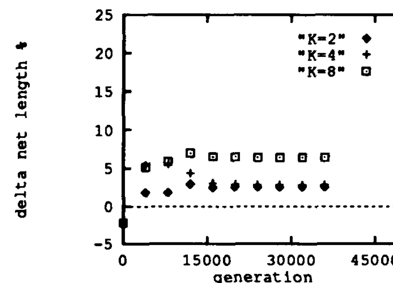



Fig. 6. *cktA* result quality versus generation.

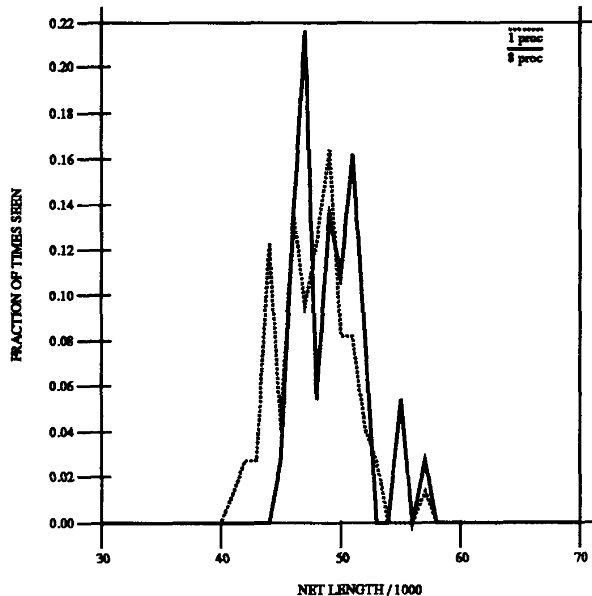



Fig. 7. Variation of final netlength in repeated experiments.

be seen that the one- and eight-processor results are similar. The means and standard deviations of the distributions were 49.708 and 2.712 for the eight-processor case and 48.107 and 2.956 for the one-processor case. When this factor is taken into account, it can be said that the parallel algorithm produces results of the same quality as the serial algorithm (to within 5%). Fig. 7 shows the final netlength difference in another set of runs on 1 to 16 pro-

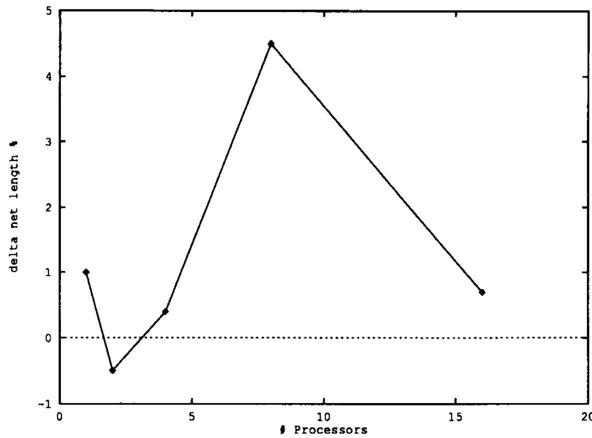



Fig. 8. *cktA* result quality versus number of processors.

TABLE I
SPEEDUP AND RESULT QUALITY

| | Num. Proc. | 2 | 4 | 8 |
|---|---|---|---|---|
| cktB | Netlength | 116 492 | 119 832 | 105 880 |
| | Delta | 0.76 | −2.08 | 9.8 |
| | Time (s) | 1417 | 736 | 360 |
| | Speedup | 1.9 | 3.67 | 7.51 |
| cktD | Netlength | 156 471 | 153 528 | 165 088 |
| | Delta | 2.04 | 3.88 | −3.34 |
| | Time (s) | 6224 | 3248 | 1744 |
| | Speedup | 1.88 | 3.61 | 6.72 |
| cktF | Netlength | 112 233 | 116 566 | 118 505 |
| | Delta | −.37 | −4.24 | −5.97 |
| | Time (s) | 18 835 | 9813 | 5303 |
| | Speedup | 1.94 | 3.73 | 6.91 |

cessors. Table I shows the consolidated result quality and speedup results for three representative circuits. It was observed that the runs of the multiprocessor algorithm produced final results within 5% of the uniprocessor result with extremely high probability.

The effect of population size on result quality was observed to determine optimum population sizes. Fig. 9 shows the final netlength as a function of population size for the sequential algorithm and for the parallel version. All the data points shown in the figure were obtained with the same set of genetic parameters except the population size $p$ and the number of generations $g$. To keep the total number of configurations examined by the algorithm the same in all cases, the product $p * g$ was kept constant as the population size was varied. The best result with the sequential algorithm was obtained with a population size of 96. This population was then split across, two, four, and eight processors in repeated runs of the parallel algorithm. It may be observed that $K = 2$ the parallel algorithm performs somewhat better than the seqential algorithm in terms of result quality while in other cases it is somewhat worse. In all cases the parallel algorithm produces results within ±5% of the best sequential result, as observed earlier.

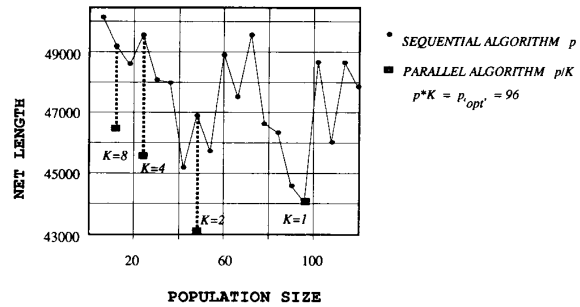



Fig. 9. Effect of population size on result quality.

## B. Speedup

The first set of speedup measurements was based on measurements of the run time of the parallel algorithm for a fixed number of generations, $\tau_K$ (3.1). The speedup for a $K$ processor experiment is computed as $\tau_K/\tau_1$. Fig. 10 shows the speedup obtained for *cktC*. Two curves are plotted; the curve labeled "total" is the speedup computed by dividing the total run time of the serial algorithm by the total run time of the parallel algorithm, whereas the curve labeled "region1" is the speedup obtained in the initial stages of the algorithm where the improvement in netlength is fastest. While the curve labeled "total" is smooth and shows almost linear speedup, the curve labeled "region1" is less smooth, due to the nonuniform convergence of the algorithm in the initial stages. This nonuniform convergence is due to the effect of the population size and the choice of the initial random population. Similar results were obtained for other circuits. Fig. 11 shows the total speedup for three representative circuits *cktA*, *cktE*, and *cktG* in another form. The consolidated speedup and result quality figures for three other circuits are shown in Table I. It can be seen that the final speedup in all cases was close to ideal, and the final result quality was not significantly different either. The speedup was close to ideal because, among the various time factors in (3.1), the communication time $t_{comm}$ was extremely low, the initialization and setup times were negligible, and the reproduction and evaluation times $t_{rep}(n, p)$ and $t_{eval}(n, p)$ scaled properly with the population size, as expected.

Additional studies on the speedup of the parallel algorithm were done to rule out scattering effects (due to the random number generator) that might lead one to erroneous conclusions about the speedup. The algorithm was run repeatedly on one, two, four, and eight processors, and the best netlength obtained in each epoch was noted. Then a single best-fit curve was obtained for each of the four cases (Fig. 12). It was found that in all cases the curves had the form $a + b/t^{0.5}$. The $a$ values for all the curves were almost the same, indicating that the result quality would be the same, given sufficient run time. The $b$ values decreased as the number of processors increased, reflecting the faster convergence and the speedup obtained by using multiple processors. The observed $b$ values show that the time required to reach a certain given

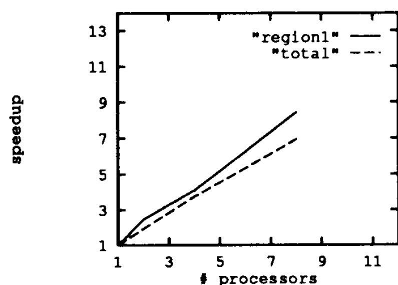



Fig. 10. Speedup of *cktC*.

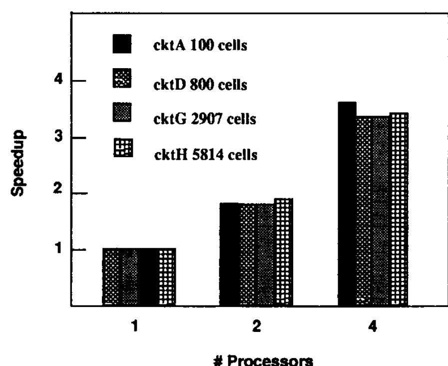



Fig. 11. Speedup on multiple processors for three circuits.

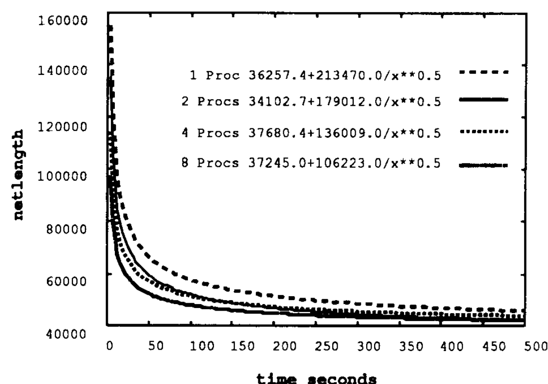



Fig. 12. Empirical convergence curves.

value of the cost function can be reduced by a linear factor by adding more processors. Let the required value of the cost function be $y_0$. Then the equations obtained by curve-fitting predict the time required to obtain this value to be $(b/y_0 - a))^2$. Let $b_1$ be the $y$ value for the uniprocessor case and $b_K$ be the $b$ value for the $K$ processor case. Assuming the $a$ values are equal, the speedup obtained using $K$ processors is $(b_1/b_K)^2$. It is observed that this factor is a linear function of $K$. This second speedup result shows that if an adaptive termination condition were to be used, stopping the algorithm once the algorithm achieved some preset solution quality, the parallelization scheme presented here would still achieve excellent speedup. For example, if the placement program were modified for gate array placement, with a cost function that emphasizes routability rather than minimum wire length, the algorithm may be modified to terminate once the solution is "routable" instead of examining a predetermined number of placement configurations.

Hence, the parallel algorithm attains linear speedup in two different senses. In the first case, the parallel algorithm examines the same number of configurations as the serial algorithm, and the speedup factor is close to $K$ as the communication, synchronization, and initialization costs are kept extremely small. At the same time, the result quality is not significantly different from the uniprocessor case. In the second case, the same result is demanded in both the uniprocessor and the $K$ processor runs. The time required to do this decreases linearly with the number of processors used, showing a linear speedup.

### C. Migration and Crossover

Fig. 13 plots the best netlength obtained on each of the sixteen processors as a function of the generation number. It can be seen that the curves intersect quite often, indicating that the improvement in netlength occurs in bursts, with intervening periods of stagnation. Whenever a particular processor is stagnant for a long time, a new individual introduced into the subpopulation by migration provides the impetus for a new burst of improvement.

The efficacy of the migration mechanism was explored in the next set of experiments. Fig. 14 shows the convergence curves for different values of epoch length. The number of migrants was kept constant, while the epoch length was varied. The best netlength obtained, taking into account all the subpopulations, was plotted against the generation number for different epoch lengths ranging from 10 to infinity.

It can be seen that there is a big difference in the convergence rates and the final solution for different epoch lengths. Very short epoch lengths (10 and 20) and very long epoch lengths (90 and infinity) led to poor results. Long epoch lengths imply less transfer of information between subpopulations. In the extreme case of infinite epoch length, there is no migration at all and the parallel genetic algorithm running on $K$ processors with a total population of $p$ is equivalent to a sequential genetic algorithm run $K$ times with a population of $p/K$. Hence the solution obtained in the case of long epoch lengths corresponds to the best solution obtained with multiple runs of the serial algorithm using a much smaller population. Conversely, when the epoch length is very small the subpopulations tend to converge. This means the effective population size is still $p/K$, but the final solution is worse than the case of no migration at all because all the subpopulations are identical, making this run of the $K$ processor algorithm equivalent to a single run of the uniprocessor algorithm with a population of $p/K$ rather than $K$ runs as in the case of no migration. It was observed that the $K$ processor algorithm performed best for some intermediate value of epoch length, as seen in Fig. 14.

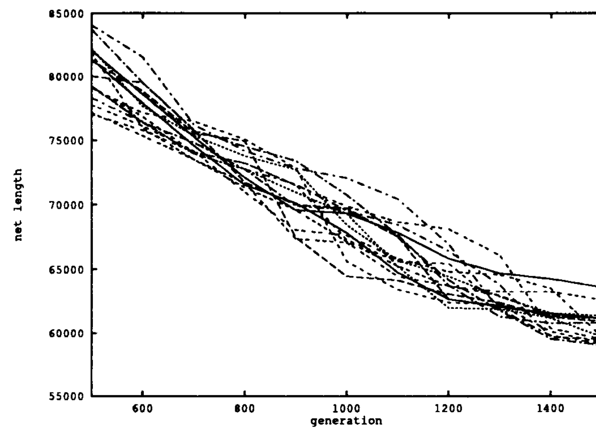



Fig. 13. Convergence curves for 16 processors.

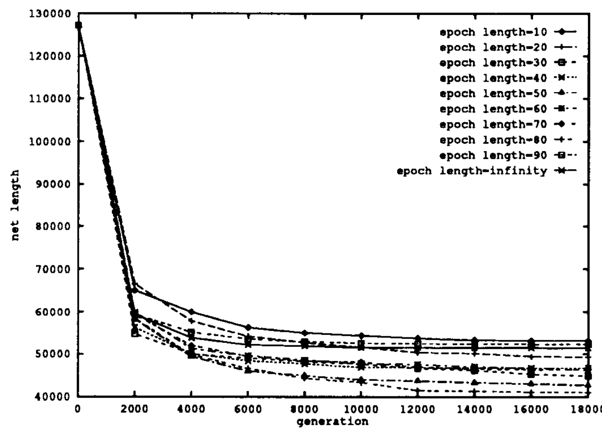



Fig. 14. Migration rate variation.

The migration rate is a function of the epoch length and the number of migrants transferred in each epoch. Increasing the number of migrants while keeping the epoch length and crossover rate constant corresponds to the transfer of larger fractions of the subpopulation from one processor to another. This causes the subpopulations to become almost identical after a few generations, just as in the case of extremely short epoch lengths. Conversely, decreasing the number of migrants while keeping the epoch length and crossover rate fixed, or increasing the crossover rate while keeping the number of migrants and the epoch length fixed, reduces the beneficial influence of migration. If the epoch is too short, the subpopulations do not have enough time to generate new, highly fit individuals through crossover before the next migration. Hence a single highly fit individual quickly gets replicated in every subpopulation, and the net effect is of $K$ almost identical subpopulations evolving in parallel. This represents a huge wastage of search effort. Similar effects would be seen if the number of migrants increased to a large fraction of the population size. Conversely, if the crossover rate is high, good solutions (schema) are not fully exploited because the population size is fixed and new, good solutions are generated at a high rate (due to the high crossover rate), and the survival probability of any one of the good solutions is decreased. Thus a higher crossover rate increases the exploration part of the genetic algorithm while decreasing the exploitation part which is actually the chief strength of the algorithm as compared with stochastic hill-climbing techniques such as simulated annealing. This means a higher crossover rate reduces the effectiveness of the exploitation part of the algorithm, and the migration mechanism, which is supposed to aid in the exploitation of good schema, becomes irrelevant.

The crossover rate was doubled, keeping the migration rate constant, and the convergence curves for different epoch lengths were plotted. The effect of epoch length variation was seen to be minimal, and the best final solution was not as good as that obtained with a lower crossover rate. With the observation that the crossover rate could influence the effect of the migration rate (epoch length), the next experiment was an attempt to obtain the best crossover rate for a given migration rate. The epoch length and the number of migrants per epoch were kept constant while the crossover rate was varied. The results are shown in Table II, which shows the final netlength for various crossover rates indicated by the number of offspring per generation; the higher the crossover rate, the more offspring per generation (see Subsection II-C). Large crossover rates nullify the effect of migration leading to poor solutions. Hence, the best results are obtained by keeping the crossover rate and the number of migrants low and finding an optimal epoch length to obtain the best solution.

### D. Communication Time

The total communication time is a function of the epoch length, as stated earlier. Fig. 15 plots the percentage of time spent on communication as a function of epoch length. The data were obtained from several runs of the distributed algorithm on circuit *cktA*. The actual communication and run times for several runs of the parallel algorithm on 8 processors are shown in Table III. It can be seen that the communication time is less than 5% of the total run time when the epoch length is greater than 50. This observation, along with the results of the previous section, shows that the communication time is not a major factor for typical runs of the distributed genetic algorithm. It may be noted that the communication time here is just the time required to communicate with the network subsystem in the workstation; the actual communication time across the network does not directly affect the speedup or the solution quality.

### E. Effect of Communication Pattern on Placement Quality

This section discusses the effect of different communication patterns on the distributed genetic algorithm. The communication pattern establishes a virtual network to-

TABLE II
CROSSOVER RATE VARIATION

| Offspring per Generation | Final Netlength |
|---|---|
| 10 | 52 611 |
| 8 | 46 437 |
| 6 | 47 166 |
| 4 | 46 301 |
| 2 | 46 225 |

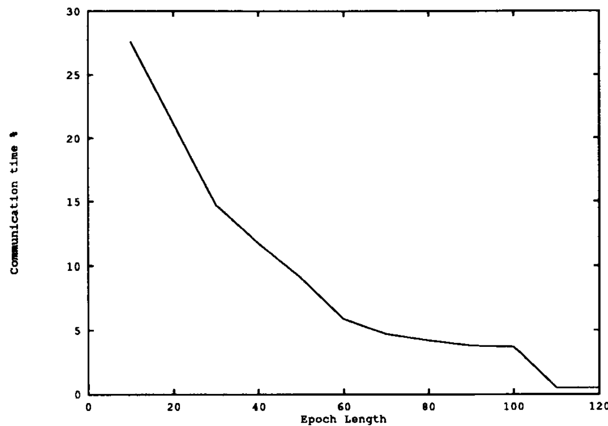



Fig. 15. Communication time.

TABLE III
COMMUNICATION AND RUN TIMES

| Epoch Length | Total Run Time (s) | Communication Time (s) |
|---|---|---|
| 30 | 68 | 10 |
| 40 | 68 | 8 |
| 50 | 66 | 6 |
| 60 | 67 | 4 |
| 70 | 63 | 3 |

pology on top of the ethernet bus structure. The algorithm succeeds in generating a good solution if

- The subpopulations do not converge prematurely.
- The different subpopulations converge in the sense that the distance between two populations is constantly reduced.

The first condition is satisfied by the migration mechanism. The second condition can be satisfied if there is *sufficient* communication or migration. The value of $n_{migr}$ should be chosen carefully so as to avoid premature convergence and ensure proper transfer of genetic material. Hence the communication pattern and virtual network diameter are important factors in ensuring placement quality. For a given population size, the time to converge depends on the time to propagate an individual across the network to all the subpopulations. This time is clearly related to the diameter $D$ of the network. The minimum number of epochs needed to propagate information to all the subpopulations is $D$, where $D$ is the diameter of the network.

Five different newtworks of eight processors were simulated using the simulation mechanism built into the program (as described earlier). The migration mechanism of the parallel algorithm has two components: "sending" individuals to other processors and "getting" individuals from other processors. "Send" always succeeds, but a "get" operation can fail if the processor that has to supply the data is not ready. When a "get" operation fails in one epoch and succeeds in the next, the data obtained in the next "get" operation correspond to the best solutions of the previous epoch. This is the same as increasing the time to propagate solutions across the network by one epoch length, and is hence most critical in a ring network. In practice, at least a few, but not all, "get" operations can be expected to fail, unless the processors are either perfectly synchronized or have totally unbalanced loads. Hence two kinds of ring networks have been simulated: one in which every get operation succeeds, called "ring+," and one in which every "get" operation is delayed by 1 epoch, called "ring−." Hypercube and square mesh topologies were simulated, as well as a totally connected network in which each processor sends and gets data to and from randomly chosen processors in each epoch. In the hypercube connection, processors communicate along one dimension in each epoch. For example, processor (100) (100 represents the coordinates of the processor in three-space where each coordinate value can be either 0 or 1) in a three-dimensional hypercube communicates with processors (000), (110), and (101), respectively, in successive epochs. In a square mesh, on the other hand, a processor at location $(x, y)$ communicates with processors at locations $(x-1, y)$, $(x, y-1)$, $(x+1, y)$, and $(x, y+1)$ in successive epochs (processors at the boundary of the mesh may thus not have any communication in certain epochs). Hence the largest number of epochs required to transmit information from any one processor in the network to another processor in the network is just the diameter $D$ of the network, $\log K$ for an $N$-dimensional hypercube ($K = 2^N$), and $K^{0.5}$ for a square mesh with $K$ processors.

With the epoch length kept small, ring+ was better than ring− as expected, and all five solutions were comparable, with the mesh connection producing the best results (see Fig. 16). With a longer epoch length, convergence was delayed on all networks, but the results were all still roughly equal, with the hypercube producing the best results (see Table IV). This suggests that, for small problem sizes, when network loading is not an important factor, the actual communication pattern is not important. However, for large problem sizes, with significant network loading at short epoch lengths, the epochs have to be long to avoid network congestion problems, and an efficient communication pattern is essential for proper convergence.

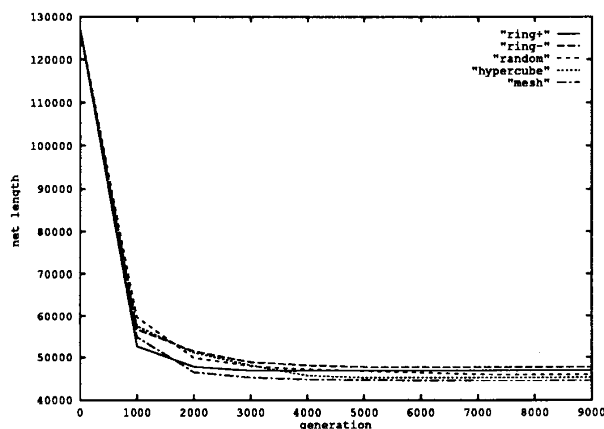



Fig. 16. Effect of network topology on convergence.

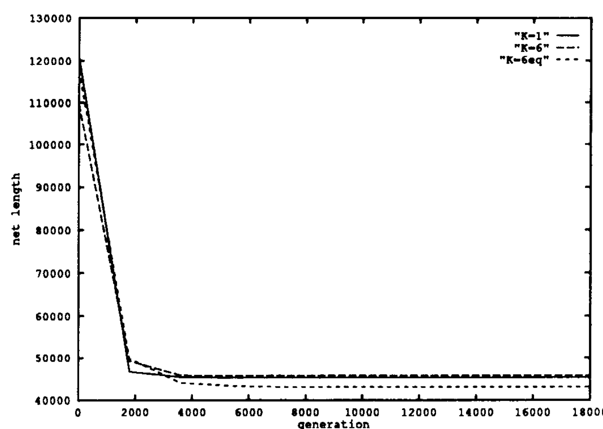



Fig. 17. Equal and unequal division of load.

TABLE IV
EFFECT OF NETWORK TOPOLOGY

| Topology | Final Network |
|---|---|
| Random | 49 536 |
| Ring+ | 47 999 |
| Ring− | 47 768 |
| Mesh | 46 524 |
| Hypercube | 46 127 |

### F. Load Balancing

The next experiment tested the robustness of the algorithm in a distributed environment where processors of varying capabilities could be present. The population was divided unequally across six processors in such a way that the expected run time of each processor was the same (to ensure maximum efficiency). The result obtained was compared to the uniprocessor result and the result obtained by splitting the population evenly across six similar processors. These results are shown in Fig. 17, where "$K = 1$" corresponds to the uniprocessor case, "$K = 6$" corresponds to six unequal processors, and "$K = 6$eq" corresponds to six equal processors. The total population in all three cases was 108, and the subpopulations in the case of the unequal processors were 48, 24, 12, 12, 6, and 6. It can be seen that the curves for $K = 1$ and $K = 6$ are almost coincident, showing that the parallel algorithm produces acceptably good results even if the population is unevenly divided across the processors. Hence static load balancing is feasible.

With the knowledge that static load balancing is feasible, the next step was to implement dynamic load balancing. Two different schemes for dynamic load balancing were considered. In the first scheme, the number of processors used remains constant, but the population on each processor varies with time. In the second scheme, the master controller has a list of $n$ processors from which it selects the $K$ least loaded machines. The load on each machine is checked periodically. When the load on any machine exceeds a specified level, the controller terminates the process running on that machine and transfers the population to the least loaded machine on its list.

The first scheme is not practicable because of the granularity of the genetic operators with respect to the population sizes used. Small changes in the population size do not change the resources consumed by the program by the same factor; for example, the resources consumed may double when the population is doubled, but an increase in the population by a factor of 1.5 may not change the load by a factor of 1.5. Hence the second scheme was adopted and found to work well in practice.

### G. Fault Tolerance

A degree of fault tolerance has been built into the distributed algorithm. Whenever a processor *dies*, any attempt to communicate with it results in an error message from the operating system. The program can detect these errors and stop communicating with the failed processor. The total population is thus reduced and the result quality is also affected to some extent as it is a function of the population size. This defect can be corrected by assigning a more active role to the master controller in dynamic load balancing and error detection and correction.

## V. CONCLUSIONS

This paper described a new distributed placement algorithm for standard cells. The algorithm runs concurrently on a network of workstations to achieve speedup linear in the number of processors used. The low communication overhead for this algorithm, as compared with the synchronization and communication overhead for conventional placement algorithms, makes this speedup possible. The parallel algorithm preserves the result quality of the serial version while achieving this speedup. Analysis of several hundred runs of the parallel algorithm with different values of $K$ (the number of processors) has shown that, if a particular solution can be obtained in time $\tau_1$ using a single processor, the same solution can be obtained in time $\tau_1/K * C$ by running the algorithm on $K$

processors, where $C$ is a constant whose value is close to 1.

The parallel algorithm has been observed to be robust in the sense that it can run in a heterogeneous computing environment where different processors on the network have different speed ratings and load factors. The speedup and result quality are maintained even in this uneven environment, which is a true representative of practical distributed computing networks, by means of static and dynamic load balancing.

Whereas the typical workstation network is just a bus with many different machines connected to the bus, the communication pattern between the machines may be random, with a machine sending migrants to a randomly selected machine in each epoch, or deterministic, with a virtual ring or hypercube network topology. The migration mechanism is not very sensitive to this virtual topology when the number of processors is low, but when the number of processors is increased, a regular virtual network topology with low diameter, such as the hypercube topology, helps to improve the performance of the parallel algorithm. An important feature of the current implementation of the placement algorithm is that it allows the communication pattern to be chosen to obtain the best result quality.

Finally, this work can be readily applied to a large class of other CAD problems and also to a host of large optimization problems in other domains, where adaptive search capabilities of the genetic algorithm have been found to yield high-quality solutions.

## REFERENCES


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

[2] J. P. Cohoon et al., "Punctuated equilibria: a parallel genetic algorithm," in *Proc. 2nd Int. Conf. Genetic Algorithms*, J. J. Grefenstette, ed. NJ: Lawrence Erlbaum Associates, 1987, pp. 148–154.

[3] J. P. Cohoon, S. U. Hegde, W. N. Martin, and D. S. Richards, "Distributed genetic algorithms for the floor plan design problem," *IEEE Trans. Computer-Aided Design*, vol. CAD-10, no. 4, pp. 483–492, Apr. 1991.

[4] T. E. Davis and J. C. Principe, "A simulated annealing like convergence theory for the simple genetic algorithm," in *Proc. 4th Int. Conf. Genetic Algorithms*, 1991, pp. 174–181.

[5] M. D. Durand, "Parallel simulated annealing accuracy vs. speed in placement," *IEEE Design and Test of Computers*, pp. 8–34, June 1989.

[6] D. E. Goldberg, *Genetic Algorithms in Search Optimization and Machine Learning.* Reading, MA: Addison-Wesley, 1989.

[7] D. E. Goldberg, "Sizing populations for serial and parallel genetic algorithms," in *Proc. 3rd Int. Conf. Genetic Algorithms*, 1989, pp. 70–79.

[8] J. H. Holland, *Adaptation in Natural and Artificial Systems.* Ann Arbor, MI: University of Michigan Press, 1975.

[9] R. Jayaraman and R. Rutenbar, "Floorplanning by annealing on a hypercube multiprocessor," in *Proc. IEEE Int. Conf. Computer-Aided Design*, 1987, pp. 346–349.

[10] M. Jones and P. Banerjee, "Performance of a parallel algorithm for standard cell placement on the Intel Hypercube," in *Proc. 24th ACM/IEEE Design Automation Conf.*, 1987, pp.807–813.

[11] R-M. Kling and P. Banerjee, "ESP: A new standard cell placement package using simulated evolution," in *Proc. 24th ACM/IEEE Design Automation Conf.*, 1987, pp. 60–66.

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

[13] J. Lam and J-M. Delsome, "Performance of a new annealing schedule," in *Proc. 25th ACM/IEEE Design Automation Conf.*, 1988, pp. 306–311.

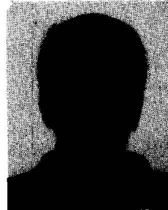
[14] J. Rose, W. Snelgrove, and Z. Vranesic, "Parallel standard cell placement algorithms with quality equivalent to simulated annealing," *IEEE Trans. Computer-Aided Design*, vol. CAD-7, no. 3, pp. 387–396, Mar. 1988.

[15] Y. G. Saab and V. B. Rao, "Combinatorial optimization by stochastic evolution," *IEEE Trans. Computer-Aided Design*, vol. CAD-10, no. 4, pp. 525–535, Apr. 1991.

[16] C. Sechen, *VLSI Placement and Global Routing Using Simulated Annealing.* Kluwer, 1988.

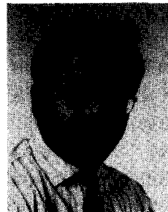
[17] K. Shahookar and P. Mazumder, "A genetic approach to standard cell placement using meta-genetic parameter optimization," *IEEE Trans. Computer-Aided Design*, vol. CAD-9, no. 5, pp. 500–511, May 1990.

[18] K. Shahookar and P. Mazumder, "VLSI cell placement techniques," *ACM Computing Surveys*, vol. 23, no. 2, pp. 143–220, June 1991.


**S. Mohan** received the B.Tech. degree in electrical engineering from the Indian Institute of Technology, Madras, India, in 1985, the M.S. degree in computer science and engineering from the University of Michigan, Ann Arbor, in 1991, and is currently in the Ph.D. program at the University of Michigan, Ann Arbor.

From 1985 to 1989 he was with the CAD Group of Indian Telephone Industries Ltd., Bangalore, India, working on physical design algorithms. His research interests include distributed CAD and the design of ultrafast circuits.


**Pinaki Mazumder** (S'84–M'87) received the B.S.E.E. degree from the Indian Institute of Science in 1976 the M.Sc. degree in computer science from the University of Alberta, Canada, in 1985, and the Ph.D. degree in electrical and computer engineering from the University of Illinois at Urbana-Champaign in 1987.

Presently, he is working as an Associate Professor at the Department of Electrical Engineering and Computer Science of the University of Michigan at Ann Arbor. Prior to this, he worked two years as a research assistant at the Coordinated Science Laboratory, University of Illinois, and over six years in India at Bharat Electronics Ltd. (a collaborator of RCA), where he developed several types of analog and digital integrated circuits for consumer electronics products. During the summers of 1985 and 1986, he worked as a member of the technical staff in the Naperville branch of AT&T Bell Laboratories. His research interests include VLSI testing, physical design automation, ultrafast digital circuit design, and neural hardware. He was a Guest Editor of the *IEEE Design and Test Magazine's* special issue on multimegabit memory testing published in March 1993.

Dr. Mazumder is a recipient of Digital's Incentives for Excellence Award, National Science Foundation Research Initiation Award, and Bell Northern Research Laboratory Faculty Award. He is a member of IEEE, Sigma Xi, Phi Kappa Phi, and ACM SIGDA.