DISTRIBUTED GENETIC ALGORITHMS FOR THE FLOOR PLAN DESIGN PROBLEMS

J. P. Cohoon
S. U. Hegde
W. N. Martin
D. Richards

Computer Science Report No. TR-88-12
June 7, 1989

Distributed Genetic Algorithms for the Floorplan Design Problem†

J. P. Cohoon, S. U. Hegde, W. N. Martin, D. Richards

Department of Computer Science
University of Virginia
Charlottesville, Virginia 22903

Abstract

Floorplan design is an important stage in the VLSI design cycle. Designing a floorplan calls for arranging a given set of modules in the plane to minimize the weighted sum of area and wire length measures. This paper presents a method to solve the floorplan design problem using distributed genetic algorithms. Distributed genetic algorithms, based on the paleontological theory of punctuated equilibria, offer a conceptual modification to the traditional genetic algorithms. Experimental results on several problem instances demonstrate the efficacy of our method, and point out the advantages of using this method over other methods, such as simulated annealing. Our method has performed better than the simulated annealing approach, both in terms of the average cost of the solutions found and the best-found solution, in almost all the problem instances tried.

† Submitted to *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems.* A conference version of this paper appears in the 1988 *IEEE International Conference on Computer-Aided Design.*

1. Introduction

Floorplan design is an important stage in the VLSI design cycle. It has received considerable attention in the literature [LAPO85, OTTE82, OTTE83, OTTE84, SECH85, STOC83, WONG86, WONG88, WOO86]. Designing a floorplan calls for arranging a given set of modules in the plane to minimize the weighted sum of area and wire length measures. Therefore, floorplan design is an optimization problem.

Floorplan design problem has been tackled via simulated annealing [OTTE84, WONG86]. We present another approach based on distributed genetic algorithms. Section 2 of this paper reviews the floorplan design problem formulation. In Section 3, we briefly discuss sequential genetic algorithms, and present our distributed genetic algorithm paradigm based on the theory of punctuated equilibria. Section 4 presents the implementation details of the application of the distributed genetic algorithms to the floorplan design problem. In Section 5, we show experimental results to demonstrate the efficacy of our method. Conclusions and future directions are in Section 6. A preliminary version of this paper has appeared elsewhere [COHO88b].

2. Floorplan Design Problem

Following Wong and Liu [WONG86], the floorplan design problem is formulated as follows: there is a set M consisting of m given *modules*, named $1,2,..,m$. Throughout this paper, we restrict our attention to rectangular modules. Module i is characterized by a triple (A_i, r_i, s_i), where A_i is the area of the module, and r_i and s_i ($r_i \leq s_i$) are the lower and upper bounds, respectively on the allowed aspect ratio, i.e., the ratio of height h_i to width w_i of the module. Each module has a *fixed* or a *free* orientation. Let M_1 be the set of modules with fixed orientation, and M_2 be the set of modules with free orientation; M_1 and M_2 are disjoint, and $M = M_1 \cup M_2$. Then, for each module i, the following relationships must hold:

$$w_i h_i = A_i \tag{1}$$

$$r_i \leq \frac{h_i}{w_i} \leq s_i \text{ , if } i \in M_1 \tag{2}$$

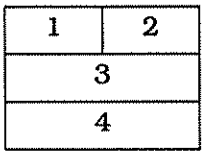
$$r_i \le \frac{h_i}{w_i} \le s_i \text{ or } r_i \le \frac{w_i}{h_i} \le s_i \text{ , if } i \in M_2. \tag{3}$$

If $r_i = s_i$ the module i is said to be *rigid;* otherwise, it is *flexible.* A *floorplan* for the given m modules consists of an enveloping rectangle R partitioned by horizontal and vertical line segments into m non-overlapping rectangular *regions;* these regions are labeled $1,2,..,m$. Region i, with width x_i and height y_i, must be large enough to accommodate module i. The aspect ratio of R is required to lie between two given numbers, p and q, $p \le q$. For each pair of modules i and j there is a cost $c_{ij} \ge 0$. The *center* of a region is the geometric center of the region. The floorplan design problem is to find a floorplan such that the objective function

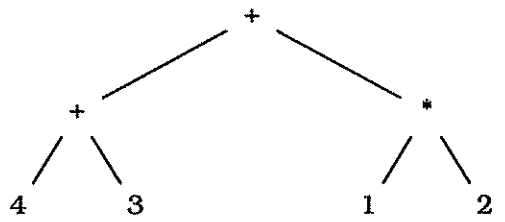
$$A + \lambda \sum_{i,j} c_{ij} d_{ij} \tag{4}$$

is minimized, where A is the area of the rectangle R, d_{ij} is the Manhattan distance between the centers of the regions i and j, $\sum_{i,j} c_{ij} d_{ij}$ is an estimate of the wire length, and λ is a user specified constant controlling the relative importance of the area and the wire length measures.

By restricting the partitions of the rectangle R into recursive subdivisions, one obtains a *slicing structure.* An example is provided in Figure 1(a). Henceforth, our attention will be restricted to floorplans which are slicing structures. Let the operations of a horizontal cut and a vertical cut be denoted by the operators + and *, respectively. Operators + and * are of different *types.* Slicing structures comprising the m given modules (also called *operands)* can be represented by *slicing trees* [OTTE82, OTTE83] or *Polish expressions* [WONG86] over the alphabet $\Sigma = \{1,2,..,m,*,+\}$. In a slicing tree operators are internal nodes and operands are leaves. There exists a one-to-one mapping from the set of slicing trees to the set of Polish expressions. The Polish expression associated with a slicing tree can be obtained by a post-order traversal [AHO74] of the slicing tree. Although there can be more than one slicing tree or Polish expression for the same slicing structure, uniqueness can be achieved by systematically partitioning R from top to bottom and from right to left at any stage of recursive subdivision. This results in slicing structures being represented as *skewed slicing trees* or *normalized Polish expressions* [WONG86]. Figures 1(b) and 1(c) both provide slicing trees and Polish expressions that correspond to slicing structure of Figure 1(a). In a skewed slicing tree, no operator of the

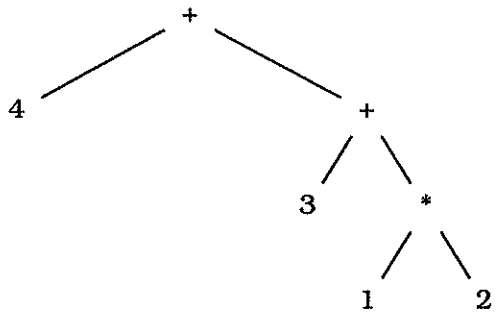


(a) Slicing structure



4 3 + 1 2 * +

(b) Skewed slicing tree and normalized Polish expression



4 3 1 2 * + +

(c) Slicing tree and Polish expression

Figure 1 — A slicing structure and its representations

same type appears as the right son of an operator. In a normalized Polish expression, operators of the same type do not occur next to one another. There also exists a one-to-one mapping from the set of skewed slicing trees to the set of normalized Polish expressions. Thus, the slicing tree and Polish expression given in Figure 1(b) are respectively skewed and normalized. while the slicing tree and Polish expression of Figure 1(c) are not. Note that a slicing tree, a Polish expression, a skewed slicing tree, and a normalized Polish expression all have m operands and $(m - 1)$ operators, resulting in a total *size* of $(2 \times m - 1)$. By convention, we number the *items*, i.e., operands or operators, in a Polish expression or a normalized Polish expression from left to right ranging from 1 to $(2 \times m - 1)$.

The set of possible dimensions, (x_i, y_i) of the region i accommodating the module i can be determined using the following information: (A_i, r_i, s_i), whether the module i has fixed or free orientation, and Equations (1)-(3). Such a set, termed the *bounding region* of module i is a subset of the Cartesian plane, and is characterized by a monotonically nonincreasing curve, called the *bounding curve* of module i. When bounding curves of the modules are piecewise linear, the bounding curve of a slicing structure can be obtained by doing simple addition operations on the "corners" identifying the piecewise linear curves [WONG86]. This gives a method of computing the area and wire length measures of a floorplan.

3. Genetic Algorithms

Sequential Genetic Algorithms: The *genetic algorithm* (GA) paradigm has been proposed to generate solutions to a wide range of problems [HOLL75, HOLL86]. In particular, several optimization problems have been investigated. These include control systems [GOLD83], function optimization [BETH81], and combinatorial problems [COHO87a, DAVI85, FOUR85, GOLD85, GREF85, SMIT85]. A more complete motivation and exposition for the material in this section can be found in our recent paper [COHO87b].

In a genetic algorithm, a *population* of solutions to the problem at hand is maintained and allowed to *evolve* through successive *generations*. A suitable encoding of each solution in the population is used to allow computation of the *fitness*, i.e., a measure of the solution's *competence*, and manipulation to form new solutions. These capabilities provide the means to create a sequence of generations. To create the next generation new solutions are formed by either merging two solutions from the current generation via a *crossover* operator or modifying an individual solution using a *mutation* operator. The solutions to be included in the next generation are then probabilistically selected according to the fitness values from the set comprising the current generation and the newly formed solutions. Typically a constant number of solutions are selected so that the maintained population is of fixed size. After an arbitrary number of generations the process is terminated and the best remaining solution (or the best ever seen) is reported.

There are many simple avenues to parallelize a sequential genetic algorithm (assuming a global shared memory), e.g., selecting and crossing-over pairs of solutions in parallel, and mutating solutions in parallel. However, such avenues result in only a simple *hardware accelerator*, and will not be suitable for local-memory, message-passing, distributed models of computation. Therefore, we have turned our attention to the theory of punctuated equilibria that provides a suitable paradigm to map genetic algorithms onto a distributed system.

Punctuated Equilibria: The theory of *punctuated equilibria* has been proposed to resolve certain paleontological dilemmas in the geological record [ELDR72, ELDR85]. Punctuated equilibria is based on two principles: allopatric speciation and stasis. *Allopatric speciation* involves the rapid evolution of new species after a small set of members of a species, *peripheral isolates*, becomes segregated into a new environment. *Stasis*, or stability, of a species is simply the notion of lack of change. It implies that after equilibria is reached in an environment there is very little drift away from the genetic composition of a species. Ideally, a species would persist until its environment changes (or the species would drift very little). Punctuated equilibria stresses that a powerful method for generating new species is to thrust an old species into a new environment, where change is beneficial and rewarded. For this reason we should expect a genetic algorithm approach based on punctuated equilibria to perform better than the typical single environment scheme.

What are the implications for the genetic algorithm approach? If the *environment* is unchanging then equilibrium should be rapidly attained. The resulting "equivalence classes" of similar solutions would correspond to "species". It is possible that solutions in the "vicinities" of minima of the objective function have not been explored. A genetic algorithm relies on the mutation and crossover operators to eventually create solutions "near" the minima. While stasis indicates that an isolated population will stabilize over time, allopatric speciation indicates that continued evolution can be obtained through the introduction of previously stabilized species into different environments. Therefore, a genetic algorithm should alternate the maintenance of populations isolated in different environments to allow the development of species with the introduction of species to new environments.

We create different environments by having the fitness measure defined relative to the current local population. In this way, exchanging sets of solutions between local populations will alter the evaluation of the members (of the local populations), and introduce new competitors thereby effecting the desired allopatric speciation. Alternate schemes for establishing different environments are possible, of course. For example, if the problem domain requires a multi-objective fitness measure, various low-order approximations to or projections of the true fitness measure could be used at different populations. This multi-objective fitness scheme and others will be investigated in our continuing research.

Genetic Algorithms with Punctuated Equilibria: Our basic model of a *genetic algorithm with punctuated equilibria* assigns a set of n solutions to each of N processors, for a total population of size $n \times N$. The set assigned to each processor is its *subpopulation.* The processors are connected by an interconnection network. In practice, we might expect a conventional topology to be used, such as a mesh or a hyper-cube. The network should have sufficient connectivity and small diameter to ensure adequate *mixing* as time progresses.

The overall structure of our approach is seen in Figure 2. There are E major iterations called *epochs.* During an epoch each processor, disjointly and in parallel, executes the genetic

---

```
initialize
for E iterations do
    parfor each processor i do
        run GA for G generations
    endfor
    parfor each processor i do
        for each neighbor j of i do
            send a set of solutions, S_ij, from i to j
        endfor
    endfor
    parfor each processor i do
        select an n element subpopulation
    endfor
endfor
```

Figure 2 — High-level description of a genetic algorithm with punctuated equilibria

---

algorithm on its subpopulation. Theoretically each processor continues until it reaches equilibrium. Since, as yet, we know of no adequate stopping criteria we have used a fixed number, G, of generations per epoch. This considerably simplifies the problem of *synchronizing* the processors, since each processor should be completed at nearly the same time. After each processor has stopped there is a phase during which each processor copies randomly selected subsets (of size $S = \lfloor S_{ij} \rfloor$) of its population to neighboring processors. Each processor now has acquired a surplus of solutions and must probabilistically select a set of n solutions to survive to be its initial subpopulation at the beginning of the next epoch.

The relationship to punctuated equilibria is the following. Each processor corresponds to a disjoint *environment* (as characterized by the mix of solutions residing in it). After G generations we expect to see the emergence of some very fit species. Then a catastrophe occurs and the environments change. This is simulated by having representatives of geographically adjacent environments regroup to form the new environments. By varying the amount of redistribution, S, we can control the amount of disruption.

The genetic algorithm code used by each processor is shown in Figure 3. The *crossover rate*, $0 \leq C \leq 1$, determines how many new offspring are produced during each generation. *Parents* are chosen probabilistically (by fitness) with replacement. The crossover itself, and

---

```
for G iterations do
    while n×C ≤ number of offspring
        created do
        select two solutions
        crossover the two solutions to create offspring
    endwhile
    add all offspring to subpopulation
    calculate fitnesses
    select a population of n elements
    generate n×M random mutations
endfor
```

Figure 3 — Genetic algorithm used within an epoch at each processor

---

other details, are discussed below. The fitnesses are recalculated, relative to the new larger population. Then, probabilistically (by fitness) without replacement, the next population is selected. Finally, in a uniformly random manner elements are mutated. The *mutation rate*, $0 \leq M \leq 1$, determines how many mutations altogether are performed during each generation.

We have already developed a system to simulate a distributed genetic algorithm, experimented with the NP-complete Optimal Linear Arrangement problem, and empirically demonstrated that our formulation is much more than a simple hardware accelerator version of sequential genetic algorithms [COHO87b].

4. Implementation Details

How should slicing structures be represented? The fact that a slicing structure can be represented by more than one Polish expression may be advantageous to a genetic algorithm because it maintains a pool of solutions. This can provide a *representational diversity* to a genetic algorithm with punctuated equilibria, specifically when a subpopulation is approaching equilibrium, in terms of the fitness measure. Therefore, we initially conducted two sets of experiments, representing slicing structures using normalized Polish expressions in one set and non-normalized (ordinary) Polish expressions in the other set. Since the results were consistently better when non-normalized Polish expressions were used, we decided to use that representation.

We have four types of crossover operators, CO_1, CO_2, CO_3 and CO_4. The first three crossover operators take two Polish expressions as parents and produce an offspring; the crossover operator CO_4 produces two offspring. The aim of these crossover operators is to bring in *building blocks* from parents into offspring [HOLL75]. Crossover operators try to identify and combine building blocks to produce "good" solutions. For the floorplan design problem, with solutions encoded as Polish expressions (equivalently, as slicing trees), building blocks are Polish subexpressions corresponding to subtrees in a slicing tree. Viewed from the point of a slicing structure (and hence a floorplan), these building blocks correspond to sub-slicing structures, i.e., an arrangement of modules belonging to a subset of the set of given

modules, *M*.

The operator CO_1 first copies the operands from a parent into the corresponding positions in the offspring. Then, it copies the operators from the second parent, by making a left to right scan, to complete the offspring. Figure 4(a) gives an illustration of CO_1. It is to be noted that CO_1 propagates groups of operands from the first parent into the offspring. The second crossover operator, CO_2 starts out by copying the operators from a parent into the corresponding positions in the offspring. Then, it completes the construction of offspring by copying the operands, by making a left to right scan, from the second parent. Figure 4(b) illustrates the operation of CO_2. By propagating the groups of operators from the first parent into the offspring, CO_2 produces an offspring having the overall nature of the slicing same as that of the first parent.

Parent 1 1 4 5 6 * + + 8 7 * 3 2 * + *
Parent 2 2 6 8 * * 7 * 5 + 4 * 1 3 + +

Offspring 1 4 5 6 * * * 8 7 + 3 2 * + +

(a) CO_1

Parent 1 1 4 5 6 * + + 8 7 * 3 2 * + *
Parent 2 2 6 8 * * 7 * 5 + 4 * 1 3 + +

Offspring 2 6 8 7 * + + 5 4 * 1 3 * + *

(b) CO_2

Parent 1 1 4 5 6 * + + 8 7 * 3 2 * + *
Parent 2 2 6 8 * * 7 * 5 + 4 * 1 3 + +

Offspring 6 5 4 1 * + + 8 7 * 3 2 * + *

(c) CO_3

Figure 4 — Illustration of the crossover operators: CO_1, CO_2, and CO_3

The third crossover operator, CO_3 first copies the operators from a parent into the corresponding positions in the offspring (as in CO_2). Then an operator is randomly selected from the first parent. The operands in the Polish subexpression associated with that operator, i.e., the operands in the subtree rooted at that operator in the slicing tree, are copied unchanged into the corresponding positions in the offspring. The Polish subexpression associated with any operator can easily be found by making a leftward scan starting from the operator, with a counter initialized to -1, incrementing the counter by one whenever an operand is encountered, decrementing the counter by one whenever an operator is encountered, till the counter becomes 1. The remaining operands required to complete the offspring are brought in from the second parent by making a left to right scan. The operation of CO_3 is illustrated in Figure 4(c), with the randomly selected operator being the rightmost + from Parent 1. The crossover operator CO_3 propagates a sub-slicing structure (with the operands in it unchanged) along with the overall nature of the slicing from the first parent into the offspring.

The crossover operator CO_4 is the most complicated of all. It attempts to exchange two equal-sized Polish subexpressions between two parents. Recall that a Polish subexpression corresponds to a slicing subtree. Let the number of operands in a subtree (of a slicing tree) rooted at an operator be defined as the *shadow number* of that operator. By definition, shadow number of an operand is 1. The operator at the root of a slicing tree has the shadow number equal to the total number of given operands, m. CO_4 starts out by computing shadow numbers of all the operators from Parents 1 and 2. The computation of shadow numbers is carried out by a stack-based algorithm. A left to right scan of a parent is made; whenever an operand is encountered its shadow number is pushed onto the stack; whenever an operator is encountered two items are removed from the stack, added together, and put back onto the stack, with the result being the shadow number of the examined operator. Note that this procedure is similar to evaluation of an arithmetic expression expressed in Polish notation. Let DSN_1 and DSN_2 be the set of distinct shadow numbers (other than 1) in Parents 1 and 2, respectively. Let $COMMON = DSN_1 \cap DSN_2$. If $COMMON = \emptyset$ the crossover fails as no subtrees

of the same size exist. Otherwise, a shadow number, $2 < k < n$ is randomly selected from *COMMON*. The restriction on k is imposed because the effect of exchanging subtrees with $k = 2$ can be easily achieved by mutation operators (described later), and the case $k = n$ results in exchanging the entire tree thus creating no new solutions. Let $OP_1(k)$ and $OP_2(k)$ be the set of operators, identified by their positions in the Polish expressions, with shadow number k in Parents 1 and 2, respectively. Let u and v be the operators selected randomly from $OP_1(k)$ and $OP_2(k)$, respectively. Let $PS(u)$ and $PS(v)$ be the Polish subexpressions associated with the operators u and v, respectively. They both have the same size, of $(2 \times k - 1)$, because the operators u and v have the same shadow numbers. The first offspring is partially created by copying $PS(v)$ into the positions of $PS(u)$; the second offspring by copying $PS(u)$ into the positions of $PS(v)$. The creation of first (second) offspring is completed by copying remaining operands and operators from Parent 1 (2) by making a left to right scan. Figure 5 illustrates the operation of CO_4.

Positions	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Parent 1	1	4	5	6	*	+	+	8	7	*	3	2	*	+	*
Parent 2	2	6	8	*	*	7	*	5	+	4	*	1	3	+	+

$DSN_1 = \{2,3,4,8\}$
$DSN_2 = \{2,3,4,5,6,8\}$
$COMMON = \{2,3,4,8\}$

$k = 4$
$OP_1(k) = \{7,14\}$
$OP_2(k) = \{7\}$

$u = 14$
$v = 7$
$PS(u) =$ 8 7 * 3 2 * +
$PS(v) =$ 2 6 8 * * 7 *

Offspring 1	1	4	5	3	*	+	+	2	6	8	*	*	7	*	*
Offspring 2	8	7	*	3	2	*	+	6	+	5	*	4	1	+	+

Figure 5 — Illustration of the crossover operator, CO_4

We believe that crossover operators CO_3 and CO_4 are more effective in manipulating building blocks compared to CO_1 and CO_2. In a series of experiments, one using the set $\{CO_1, CO_2\}$ of crossover operators, another using the set $\{CO_1, CO_2, CO_3\}$, and the third using the set $\{CO_1, CO_2, CO_3, CO_4\}$, all randomly selecting a crossover operator from the corresponding set whenever a crossover operation is to be done, the third combination (of $\{CO_1, CO_2, CO_3, CO_4\}$) performed better compared to other two combinations. This suggests the possibility of using some crossover operators more frequently than others. Currently, we are investigating that possibility. All the experiments in this study select randomly a crossover operator from the combination $\{CO_1, CO_2, CO_3, CO_4\}$ whenever a crossover operation is to be performed.

Mutations are random, diversity-increasing operators, preventing a genetic algorithm from reaching a homogeneous, single-solution state of a population. We use the *moves* specified by Wong and Liu [WONG86] as our mutation operators. They are: swapping two adjacent operands, switching a sequence of adjacent operators, and swapping an operator and a neighboring operand. Since we are not restricting Polish expressions to be normalized, we do not check for that property while mutating a solution.

The computation of area and wire length of a floorplan is carried out according to the scheme suggested by Wong and Liu [WONG86]. Since our crossover operators can produce an offspring that is drastically different from both the parents, no simple incremental method is readily available to compute the area and wire length; they are computed starting from the bounding curves of the given modules.

With any minimization problem, such as the floorplan design problem, the scores of the solutions should decrease over time. The *score* is the value of the objective function, Equation (4). Two simple fitness functions suggest themselves. First, the fitness could be inversely related to the score; this could cause excessive compression of the range of fitnesses. Second, the fitness could be a constant minus the score. The constant must be large enough to ensure all fitnesses are positive (since they are used in the selection process) and not too large (effectively causing compression). If such a constant were optimal initially, it would become a

poor choice near equilibria. For these reasons we use a time-varying "normalized" fitness.

We choose our fitness to be a function of all the scores in the current subpopulation. We have empirically found that randomly generated solutions to the floorplan design problem have scores that are "normally distributed," i.e., have a bell-shaped curve. Figure 6 shows the distribution of scores of 10,000 randomly generated solutions for a problem of size 25 to be discussed in the subsequent sections. For the data of Figure 6, 95.74% of the solutions live within two standard deviations (s.d.) of the mean, and 99.93% within four s.d.'s. (Related evidence on normal distribution of scores can be found in [COHO88a, WHIT84].) Therefore, we use

$$fitness(x) = \frac{(\mu_s - score(x)) + \alpha\sigma_s}{2\alpha\sigma_s} \tag{5}$$

where μ_s is the mean of the scores, σ_s is s.d., and α is a small constant parameter. We use clipping, i.e., setting the fitness to a very small positive value if it is negative, to ensure that fitness of a solution is positive.

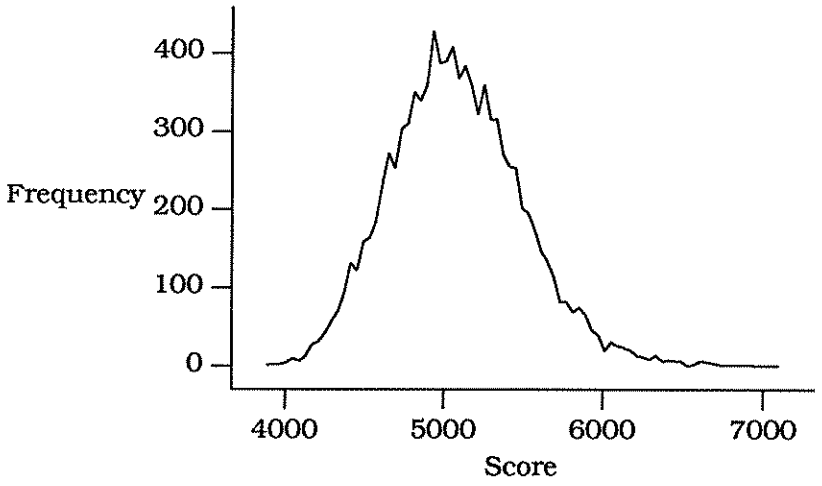


Figure 6 — Distribution of scores of 10,000 random solutions

Our current implementation, *GAPE*, is a sequential simulation of the distributed genetic algorithm with punctuated equilibria. The system has been developed in C in a UNIX environment on a VAX 11/780. We are in the process of porting our system to a hyper-cube machine recently made available to us by the University of Virginia's Institute for Parallel Computation. To make a comparative study, we also implemented a *simulated annealing* approach (*SA*) of Wong and Liu [WONG86] in C in a UNIX environment on a VAX 11/780.

5. Empirical Studies

We have performed several experiments to determine if *GAPE* is an effective approach. We have also made a comparative study of the efficacy of *GAPE* and that of *SA*. In one set of experiments, we have used the data for which an optimal floorplan is known. In another set of experiments, randomly generated data has been used. The methods used by Wong and Liu [WONG86] have been used to generate random data for the experiments using them.

For *GAPE*, the initial mix of subpopulations is created by producing n random solutions. For *SA*, the initial solution is a randomly generated normalized Polish expression. We did not use non-normalized Polish expressions in SA as it has been reported to slow down the convergence rate [WONG86].

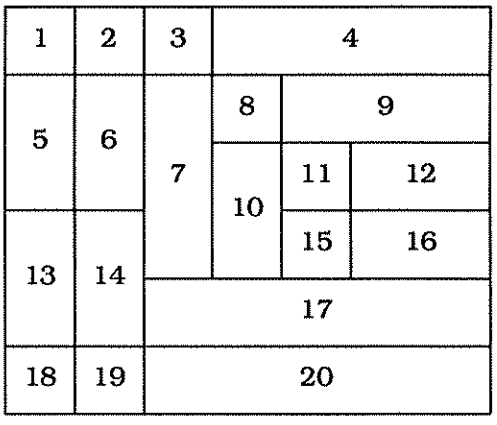


Figure 7 — Optimal floorplan for a problem instance with 20 modules

Results for the Structured Data: We have examined two problem instances, one with 16 modules and another with 20 modules. For the instance with 16 modules, all the modules are rigid (we set $r_i = s_i = 1$), have fixed orientations, and are of unit area. Therefore, the bounding curves of all the modules have a single "corner" at (1,1). The lower and upper bounds on the aspect ratio of the final chip (p and q) are 0.5 and 2.0, respectively. The optimal floorplan, with four modules in each row and column, has unit-cost wires connecting the neighboring modules. The optimal area is 16.0, and the wire length is 48.0. So, with $\lambda = 1$, the optimal floorplan has a total cost of 64.0. The second problem instance also has all rigid modules with fixed orientations, but the bounding curves of all the modules do not look alike. The lower and upper bounds on the aspect ratio of the final chip (p and q) are 0.01 and 20.0, respectively. The optimal floorplan is shown in Figure 7. The cost matrix still consists of 0/1 entries, but a module now need not be connected to all the neighboring modules (in the optimal floorplan), e.g., module 6 is connected only to module 7 although modules 2, 5, and 14 are also adjacent to it. The optimal floorplan has an area of 42.0 and wire length of 83.0, resulting in a total cost of 125.0 for $\lambda = 1$.

Distributed genetic algorithms used a mesh configuration with $N = 4$, i.e., each subpopulation being able to "communicate" during the inter-epoch transition with two other subpopulations. Since the behavior of any genetic algorithm is intricately dependent on various parameter settings, we conducted some preliminary experiments to determine the following working set of parameters: $n = 80$, $E = 16$, $G = 50$, $S = 15$, $C = 0.5$, $M = 0.3$ and $\alpha = 1$.

For the parameter settings used by *GAPE*, in all $N \times E \times G = 4 \times 16 \times 50 = 3,200$ generations are created. If we assume, in the most optimistic sense, that each crossover creates a new solution not seen before, $C \times n = 0.5 \times 80 = 40$ new solutions are seen per generation. Adding the solutions produced due to mutations (i.e., $M \times n = 0.3 \times 80 = 24$) to the above number, we come at a figure of 64 solutions per generation, resulting in $64 \times 3,200 = 204,800$ as the total number of solutions seen by the distributed genetic algorithm. (However, due to the phenomenon of stasis, not all crossovers result in new solutions, reducing the total number of distinct strings seen.)

We examined the performance of SA on the above two problem instances. To be fair in our comparisons, we allowed the *SA* to run for 175 temperature changes with 1200 moves allowed in each temperature regime. (This would result in 175×1200 = 210,000 solutions seen, and almost all of them would be distinct due to the "mutative" nature of the moves.) The annealing schedule of $T_{i+1} = \delta * T_i$, with $\delta = 0.973$ (for the instance with 16 modules) and $\delta = 0.970$ (for the instance with 20 modules) produced good solutions among the other schedules tried. The initial temperature was determined by making 1000 moves starting from a randomly generated solution, computing the average uphill cost, and setting the acceptance probability to 0.95 for that average uphill cost.

GAPE and *SA* were both run ten times on the problem instances with 16 and 20 modules, respectively. The value of λ used is 1. The average cost of solutions found by *GAPE* and *SA* for the 16 module instance are 87.6 and 103.9, respectively. The overall best solution found by *GAPE* for this instance has a cost of 72.0, and that found by SA has a cost of 94.0. For the 20 module problem instance, *GAPE* found solutions with an average cost of 183.3, while *SA* found solutions with an average cost of 183.5. The overall best solution found by *GAPE* for this instance has a cost of 150.0 while that of *SA* has a cost of 154.0. We see that *GAPE* is better than SA in terms of the best-found solution in both the cases. In terms of the average cost of the solutions found, *GAPE* is better than *SA* for the problem instance with 16 modules, and the performance of *GAPE* and *SA* on the problem instance with 20 modules is essentially the same.

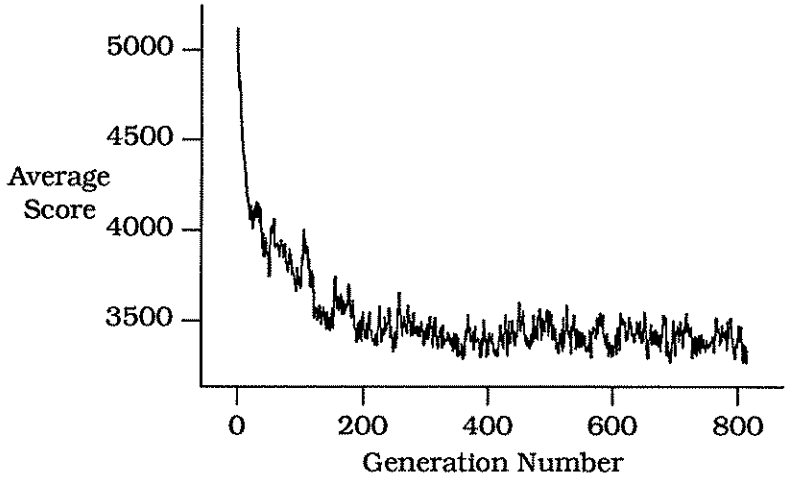
Results for the Randomly Generated Data: We have examined three problem instances, all with 25 modules. In all the instances, all the modules are flexible (we chose $r_i = s_i^{-1}$), and have free orientations. Areas of the modules are selected randomly from a uniform distribution over [1,20]. The cost matrices have random entries selected from a uniform distribution over [0,1]. In the first two instances s_i is 3 for all the modules. In the third instance s_i is randomly selected from a uniform distribution over [1,4]. In all the instances, bounding curves have been approximated by two "corners." The value of λ used is 1 throughout. The lower and upper bounds on the aspect ratio of the final chip (p and q) are 0.5 and 2.0, respectively. Since the data is random, we do not know the optimal costs.

GAPE used a hyper-cube configuration with $N = 8$, i.e., a regular cube. All the other parameters used by *GAPE* are the same as in the previous section. While making comparative studies, we allowed *SA* to run for 350 temperature changes with 1200 moves allowed in each temperature regime. We found $\delta = 0.9847$, 0.9840 and 0.9827 to produce good results in the three instances, respectively.

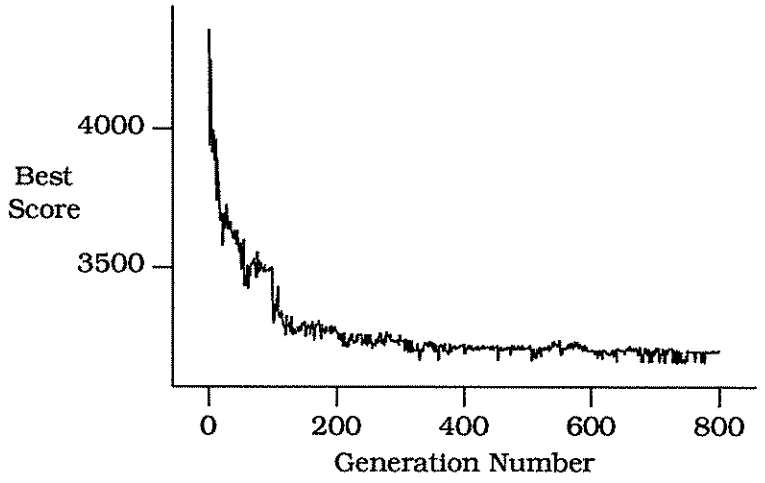
For each of the three random data instances, *GAPE* and *SA* were again respectively run ten times. The average cost of solutions found by *GAPE* and *SA* for the first random data instance is 2964.6 and 3144.3, respectively. The overall best solution found by *GAPE* for this instance is 2880.0, and it is 3083.0 for *SA*. For the second random data instance, *GAPE* and *SA* found solutions with an average cost of 3452.2 and 3652.0, respectively. *GAPE* found an overall best solution for this instance with a cost of 3371.9 while *SA* found one with a cost of 3563.2. For the third random data instance, *GAPE* and *SA* found solutions with an average cost of 3063.7 and 3186.9, respectively. The overall best solution found by *GAPE* for this instance is 2956.8 and it is 3129.6 for *SA*. We note that *GAPE* again performed consistently better than *SA*.

Figure 8 is a plot depicting the average and the best performance of one subpopulation of the hyper-cube. The average computed is the average score of the solutions forming a subpopulation after a generation has been completed. The best score is the score of the best solution produced in a generation. We note a steep fall in the first four epochs, and a local search in the later epochs. Also of interest is the fact that the curves are not monotonically decreasing. The often found increase in the average score and the best score is the result of: catastrophic mixing during the communication phase of *GAPE*, and the generation and survival of solutions with relatively "bad" scores. This is necessary to avoid getting trapped in local minima. The key parameter affecting the steepness of the curves is α. We have more to say about α in the next section.

We also ran experiments simulating *GAPE* on a 5-dimensional hyper-cube (i.e., $N = 32$). The problem instances are the three 25-module, random instances described above. The average solution costs found by *GAPE* for the three instances were respectively 2912.7, 3388.4,



(a) Performance based on Averages



(b) Performance based on the Best-generated solutions

Figure 8 — Behavior of one subpopulation of an 8-node hyper-cube

and 3037.7. The overall best solution costs found by *GAPE* were 2851.7, 3349.3, and 2911.4. The average and the best-found solutions are superior to those determined with a 3-dimensional cube (consisting of 8 nodes). The average solution cost was on average 1.5% better and the best solution cost was on average 1.1% better. This result points out the

advantage of using more subpopulations; the result is not surprising because more subpopulations imply a more diverse exploration of the solution space.

6. Conclusions and Extensions

We have shown how distributed genetic algorithms can be used to solve the floorplan design problem. We have also empirically demonstrated the efficacy of our method, and compared it with the *SA* approach. Given approximately the same number of strings to be examined, our method performed consistently better than the *SA* approach in almost all the problem instances tried, both in terms of the average cost of the solutions found and the best-found solution. Our method can be easily implemented on a local-memory, message-passing distributed computer.

GAPE is characterized by many parameters. We are currently studying the detailed dynamics of *GAPE* to understand the specific effects of these parameters. The scaling factor, α, of Equation (5) determines the importance of relative differences in the scores of members forming a subpopulation. In some of our preliminary experiments we have observed that lower values of α produce solutions (of relatively inferior quality) faster. We conjecture that α plays a role equivalent to that of temperature in simulated annealing [KIRK83]. Therefore various schedules can be tried on α. Similarly, the mutation rate can be varied over time.

The objective function in Equation (4) has two distinct components — area and wire length — to be minimized which can be tackled by two separate sets of communicating subpopulations, one set emphasizing the area term and another emphasizing the wire length term (by using different values of λ). Similarly, the area and the wire length terms can be given dynamically changing relative importance over different epochs.

7. Acknowledgements

The authors' work has been supported in part by the Office of Naval Research through grant N00014-88-K-0486 and by the Jet Propulsion Laboratory of the California Institute of Technology under Contract 957721 to the University of Virginia's Institute for Parallel Computation. The work of J. P. Cohoon has also been supported by the National Science

Foundation through grant DMC 8505354. Their support is greatly appreciated.

8. References

[AHO74] A. V. Aho, J. E. Hopcroft and J. D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, MA, 1974.

[BETH81] A. Bethke, *Genetic Algorithms as Function Optimizers*, Ph.D. Thesis, Department of Computer and Communication Sciences, University of Michigan, 1981.

[COHO87a] J. P. Cohoon and W. D. Paris, Genetic placement, *IEEE Transactions on Computer-aided Design of Integrated Circuits and Systems, cad-6*(6), November 1987, 956-964.

[COHO87b] J. P. Cohoon, S. U. Hegde, W. N. Martin and D. Richards, Punctuated equilibria: A parallel genetic algorithm, *Second International Conference on Genetic Algorithms and Their Applications*, Cambridge, MA, 1987, 148-154.

[COHO88a] J. P. Cohoon and M. T. Roberson, *Jump Starting Simulated Annealing*, Department of Computer Science, University of Virginia, 1988.

[COHO88b] J. P. Cohoon, S. U. Hegde, W. N. Martin and D. Richards, Floorplan design using distributed genetic algorithms, *IEEE International Conference on Computer-Aided Design*, Santa Clara, CA, to appear 1988.

[DAVI85] L. Davis, Job shop scheduling with genetic algorithms, *International Conference on Genetic Algorithms and Their Applications*, Pittsburgh, PA, 1985, 136-140.

[ELDR72] N. Eldredge and S. J. Gould, "Punctuated equilibria: An alternative to phyletic gradualism", in *Models of Paleobiology*, T. J. M. Schopf (editor), Freeman, Cooper and Co., 1972, 82-115.

[ELDR85] N. Eldredge, *Time Frames*, Simon and Schuster, 1985.

[FOUR85] M. P. Fourman, Compaction of symbolic layout using genetic algorithms, *International Conference on Genetic Algorithms and Their Applications*, Pittsburgh, PA, 1985, 141-150.

[GOLD83] D. E. Goldberg, *Computer-Aided Gas Pipeline Operation Using Genetic Algorithms and Learning Rules*, Ph.D. Thesis, Department of Civil Engineering, University of Michigan, 1983.

[GOLD85] D. E. Goldberg and R. Lingle, Jr., Alleles, loci, and the traveling salesperson problem, *International Conference on Genetic Algorithms and Their Applications*, Pittsburgh, PA, 1985, 154-159.

[GREF85] J. J. Grefenstette, R. Gopal, B. J. Rosmaita and D. Van Gucht, Genetic algorithms for the traveling salesperson problem, *International Conference on Genetic Algorithms and Their Applications*, Pittsburgh, PA, 1985, 160-168.

[HOLL75] J. H. Holland, *Adaptation in Natural and Artificial Systems*, University of Michigan Press, Ann Arbor, MI, 1975.

[HOLL86] J. H. Holland, K. J. Holyoak, R. E. Nisbett and P. R. Thagard, *Induction: Processes of Inference, Learning, and Discovery*, MIT Press, Cambridge, MA, 1986.

[KIRK83] S. Kirkpatrick, C. D. Gelatt and M. P. Vecchi, Optimization by simulated annealing, *Science, 220*(4598), May 13, 1983, 671-680.

[LAPO85] D. P. LaPotin and S. W. Director, Mason: A global floorplanning tool, *IEEE International Conference on Computer-Aided Design*, Santa Clara, CA, 1985, 143-145.

[OTTE82] R. H. J. M. Otten, Automatic floorplan design, *19th ACM-IEEE Design Automation Conference*, Minneapolis, MN, 1982, 261-267.

[OTTE83] R. H. J. M. Otten, Efficient floorplan optimization, *IEEE International Conference on Computer Design*, Port Chester, NY, 1983, 499-502.

[OTTE84] R. H. J. M. Otten and L. P. P. P. van Ginneken, Floorplan design using simulated annealing, *IEEE International Conference on Computer-Aided Design*, Santa Clara, CA, 1984, 96-98.

[SECH85] C. Sechen and A. Sangiovanni-Vincentelli, The Timberwolf placement and routing package, *IEEE Journal of Solid-State Circuits, SC-20*(2), 1985, 510-522.

[SMIT85] D. Smith, Bin packing with adaptive search, *International Conference on Genetic Algorithms and Their Applications*, Pittsburgh, PA, 1985, 202-206.

[STOC83] L. Stockmeyer, Optimal orientations of cells in slicing floorplan designs, *Information and Control, 59*, 1983, 91-101.

[WHIT84] S. R. White, Concepts of scale in simulated annealing, *IEEE International Conference on Computer Design: VLSI in Computer*, Port Chester, NY, 1984, 646-651.

[WONG86] D. F. Wong and C. L. Liu, A new algorithm for floorplan design, *23rd ACM-IEEE Design Automation Conference*, Las Vegas, NV, 1986, 101-107.

[WONG88] D. F. Wong, H. W. Leong and C. L. Liu, *Simulated Annealing for VLSI Design*, Kluwer Academic Publishers, Boston, MA, 1988.

[WOO86] L. S. Woo, C. K. Wong and D. T. Tang, Pioneer: A macro-based floorplanning design system, *VLSI Systems Design*, August 1986, 32-43.