# Solving Quadratic Assignment Problems With Parallel Genetic Algorithms

Jerri Hines
John T. Thorpe *
Department of Computer Science

Kenneth B. Winiecki, Jr. †
Department of Electrical Engineering
Clemson University
Clemson, South Carolina 29634

Frederick C. Harris, Jr.
Department of Computer Science
University of Nevada
Reno, Nevada 89557
fredh@cs.unr.edu

**Abstract**

Parallel processing has been valuable for improving the performance of many algorithms and is attractive for solving intractable problems. Traditionally, exhaustive search techniques have been used to find solutions to NP-complete problems; however, parallelization of exhaustive search algorithms can provide only linear speedup, which is typically of little use since problem complexity increases exponentially with problem size. Genetic algorithms can help to provide satisfactory results to such problems. This paper presents a genetic algorithm that uses parallel processing to solve the quadratic assignment problem.

**keywords:** Quadratic Assignment Problem, Parallel Genetic Algorithms

## 1 Introduction

Parallel processing has proven to be valuable for increasing the performance of various algorithms. Intractable problems traditionally solved by exhaustive search techniques seem to resist the speedup typically produced by parallelization. Algorithms for these problems, when run in parallel, typically give a speedup only directly proportional to the number of processors working in concert on the problem. This paper presents an alternative to traditional exhaustive search methods using a variation on genetic algorithms [1, 3].

*Current Address: Common Development Team, AT&T Global Information Solutions, Greenville, SC 29615, John.Thorpe@ClemsonSC.ATTGIS.COM

†Current Address: Loral Aerosys, NASA Goddard Space Flight Center Greenbelt, MD 20771 kwiniec@vlsi9.gsfc.nasa.gov

In the early 1970's John Holland at the University of Michigan developed a heuristic search technique he termed a *genetic algorithm* [3]. Because they are heuristic techniques, genetic algorithms are not guaranteed to find optimal solutions to complex systems, but rather to find "satisfactory" ones. A satisfactory result is defined by the problem and by how close to the optimal solution the user deems acceptable. It is hoped that using genetic algorithms in parallel can produce acceptable results in a "reasonable" amount of time, as good as or better than exhaustive search techniques can in that same time.

NP-complete problems are a class of decision problems that are considered intractable; i.e., solutions to these problems probably will not be found with a polynomial time algorithm. Although it has not been proven whether or not NP-complete problems are truly intractable, it appears that a major breakthrough is necessary to solve them in polynomial time. The quadratic assignment problem has been classified as an NP-complete problem that is a transformation of the HAMILTONIAN CIRCUIT problem [2]. For a more complete overview of NP-completeness and for a list of known NP-completeness problems see [2]. For more work on NP-Complete problems see Johnson's series of *NP-Completeness Columns* [5].

The purpose of this paper is to demonstrate the effectiveness of using genetic algorithms along with parallel processing to obtain good solutions for large or intractable problems. The problem chosen to demonstrate this effectiveness is the quadratic assignment problem. Simply stated this problem involves the placement of $m$ production plants at $n$ sites so as to minimize transportation costs between them.

In Section 2 the backgrounds of the quadratic as-

signment problem and genetic algorithms are presented. Section 3 explains the heuristic used to solve the problem. Results and conclusions are presented in Section 4, and future work is presented in Section 5.

## 2 Background

### Quadratic Assignment Problem

The problem chosen is known as the Quadratic Assignment Problem (QAP). Consider the problem of placing each of $m$ plants at one of $n$ possible sites such that the total cost of transporting materials from one site to another is minimized. Each plant must be placed at some site, and at most one plant may be placed at a single site. Let $d_{i,j}$ be the number of items to be transported from site $i$ to site $j$, let $c_{i,j}$ be the cost of transporting a single item from site $i$ to site $j$, and let

$$A = (\pi(1), \pi(2), \ldots, \pi(m))$$

be a mapping of plants to sites where $1 \leq \pi(i) \leq n$ is the site at which plant $i$ is located. The objective is to find a mapping $A$ such that the cost

$$F(A) = \sum_{i=1}^{m} \sum_{j=1}^{m} (d_{i,j} c_{\pi(i),\pi(j)})$$

is minimized.

The following example is taken from Horowitz and Sahni [4]. Assume two plants ($m = 2$) and three possible sites ($n = 3$). Also assume:

$$\begin{pmatrix} d_{1,1} & d_{1,2} \\ d_{2,1} & d_{2,2} \end{pmatrix} = \begin{pmatrix} 0 & 4 \\ 10 & 0 \end{pmatrix}$$

and

$$\begin{pmatrix} c_{1,1} & c_{1,2} & c_{1,3} \\ c_{2,1} & c_{2,2} & c_{2,3} \\ c_{3,1} & c_{3,2} & c_{3,3} \end{pmatrix} = \begin{pmatrix} 0 & 9 & 3 \\ 5 & 0 & 10 \\ 2 & 6 & 0 \end{pmatrix}$$

Sample placement of the plants at sites and their corresponding costs are shown in the table below. The third row represents the optimal solution.

| $\pi(1)$ | $\pi(2)$ | $F$ |
|---|---|---|
| 1 | 2 | $9 \times 4 + 5 \times 10 = 86$ |
| 3 | 1 | $2 \times 4 + 3 \times 10 = 38$ |
| 1 | 3 | $3 \times 4 + 2 \times 10 = 32$ |

### Genetic Algorithms

Genetic algorithms were first developed by John Holland at the University of Michigan in the early 1970's. Holland was interested in how an algorithm could simulate natural selection. The goals of Holland's research included explaining the adaptive processes of natural systems and then designing artificial systems software which would retain the important mechanisms of natural selection [3]. Thus, the power of genetic algorithms should lie in their robustness, or ability to adapt, just as in natural systems.

Genetic algorithms are heuristic search algorithms. Thus, the goal of a genetic algorithm is not necessarily to find the optimal solution to a complex system, but to produce a "satisfactory" one. Random choice is used to guide a genetic algorithm as it searches. As a genetic algorithm iterates, better solutions may be discovered.

The structure of a genetic algorithm is based on natural selection. First, an initial population of feasible solutions is randomly generated. The initial population consists of "chromosomes," encoded representations of solutions. "Selection" takes place between members of the population, and a "child" is formed from a combination of the "parent" chromosomes. For each new child an evaluation function is used to determine the "fitness" of that child. Whether or not the child becomes a member of the population depends on its fitness value. Each new child chromosome is compared against the worst member of the population, and the better one is kept in the population. By producing new generations in this manner, the population improves and the best member of the final population is the solution returned by the algorithm.

A chromosome is traditionally a binary string. According to Goldberg [3], genetic algorithms should be blind to the application; that is, the genetic algorithm should have no information as to what the bit string represents. Davis [1], on the other hand, suggests that close inspection of the encoding can give clues as to what makes a good solution "good" and what makes a bad solution "bad" and can improve the quality of the search by including this information in the genetic algorithm.

Since they are based on genetics, major elements of genetic algorithms include selection, recombination and mutation. Selection is the process of choosing parent chromosomes which will be recombined to form the next generation. The choice of parent chromosomes can be completely random but is usually biased in some manner so that better chromosomes are more likely to be used as parents. Two popular biased ran-

dom number generators used with genetic algorithms are linear bias and roulette wheel.

Recombination is the process of taking the parent chromosomes and forming a child chromosome. Simple methods for recombination include one-point crossover and two-point crossover. In these methods a random number is generated to correspond with one (or two) positions in the encoded parent solutions, and then the portions of the parent chromosomes around these positions are switched to form a child chromosome. The one- and two-point crossover methods of recombination require no knowledge as to what makes a solution good or bad.

Mutation is necessary in a genetic algorithm to prevent the some potentially useful genetic material from being ignored and can guide the search in new directions. Mutation occurs in a genetic algorithm at the time of recombination. Mutation can take place by randomly generating a child chromosome or by randomly changing part of the encoding of a child chromosome. As in nature, mutation generally occurs only a small portion of the time.

Another feature of natural selection which is simulated in a genetic algorithm is survival of the fittest, achieved through the use of an evaluation function. When a child chromosome is evaluated for fitness, its survival, in essence, is being determined. If the child's fitness is not good enough for it to be inserted into the population, then the child does not "survive". The evaluation function used depends upon the particular problem to which the genetic algorithm is being applied.

A genetic algorithm stops when it reaches convergence or when it has run for a predetermined number of iterations. Some definitions of convergence could involve having all identical solutions in the population, having all fitness values equal in the population, or having all fitness values within a certain range of each other. When convergence has occurred or when the predetermined number of iterations has been reached, the best solution is returned from the final population.

## 3 Method

Why use genetic algorithms implemented in parallel to find solutions to the quadratic assignment problem? An initial attempt might be to use an exhaustive search method to place plants at various cities and then evaluate the resulting configuration. For large problem sizes (10 or more plants), this approach is obviously time consuming and is not guaranteed to produce a satisfactory result within a reasonable amount of time.

An alternative to the exhaustive search is a heuristic such as a genetic algorithm. The premise of the genetic algorithm is that by generating a large number of solutions, or population, and continually recombining the solutions, a satisfactory result will eventually be produced through "survival of the fittest" as "better" results replace "worse" results. A genetic algorithm working on a single processor may produce good results, but if multiple versions of the algorithm were to operate in parallel in some *cooperative fashion*, it is likely that the concurrent version would produce even better results than the single processor implementation.

The algorithm used for the quadratic assignment problem is basically a genetic algorithm with some modifications that enhance its use on a parallel processing system. For this experiment four nodes of a parallel processing machine (in this case an iPSC/2) are allocated to run the genetic algorithms. Each node generates its own initial population and begins executing the genetic algorithm. Each iteration of the genetic algorithm produces six children to evaluate and possibly insert into the node's population. A "mutant" is generated periodically and inserted into the population. Mutation is a technique to help prevent stagnation of the population. Once the genetic algorithm meets one of its convergence criteria (time limit, number of iterations, difference in the cost between the best and the worst solutions), the algorithm halts and broadcasts its results to the host program. If all nodes have converged to the same cost, the host stops and reports the results.

As previously noted, the algorithm used does not follow the traditional approach described by Goldberg [3] in which the *entire* population is replaced at each iteration. Instead, a combination of Davis' and Goldberg's approaches was used in which each genetic algorithm uses a Davis-like approach to insert a few new members into the population [1].

There is no strategy involved in creating initial populations. The plants are randomly placed at the sites and the cost of each configuration is determined. After the initial population is generated, the genetic algorithm selects three sets of parent "chromosomes" to recombine. Two set are chosen via a simple linear bias, and the other is chosen from a normal distribution of the best ten percent (10%) of the population. The reason for this selection method is that using the "best" parent for every generation was found to cause the population to converge to local "best" solutions rather

than generating better solutions. In other words, the population tended to be dominated by variations on the best parent.

Genetic algorithms were originally designed for use on single-processor machines. To take advantage of parallel processing, a variation on traditional genetic algorithms is in order. By allowing some sort of "cross-pollination" of chromosomes between genetic algorithms operating in parallel, information can be shared, and it is hoped that the interaction will improve performance of the genetic algorithms. This modification is logically consistent and exists in "real-world" genetics.

A pollination rate $P$ is set as a parameter to the program, and each genetic algorithm sends a solution to the host program once every $P$ iterations. The host then chooses the best solution and broadcasts it back to all the nodes to use in their recombination. This method of cross-pollination has provided exceptionally better results then simply running four genetic algorithms independently, an observation made by the authors in preliminary work. One additional note: when cross-pollination occurs too frequently, the populations tend to converge very quickly, and rarely do they produce a satisfactory result.

The recombination technique used in this work is based on uniform order-based crossover presented by Davis [1]. According to this recombination, a chromosome is a bit string that represents the contribution of the parents to a child. A one in the bit string indicates that the vertex corresponding to that bit index will contribute to the construction of child 1, and a zero indicates that the vertex will contribute to child 2. This bit string is generated randomly for each generation with parent 1 receiving the bit string and parent 2 receiving its complement. Hence, child 1 is composed of vertices marked with ones and child 2 is made of vertices marked with zeroes. This type of chromosome recombination is illustrated in Figure 1. This bit string is then used to determine the six children (two from each pairing of the three parents). These children are then evaluated by the cost function (our fitness criteria) and inserted into the population if the value returned is better than the worst member.

In prior versions of this program, convergence was based upon three criteria: a 15 minute time limit, a maximum number of generations, and the difference between the best and the worst solutions in the population. The major change that was made to the parallel version of this algorithm (beside cross-pollination) was the addition of a "second chance." Based upon the performance of previous versions we decided that

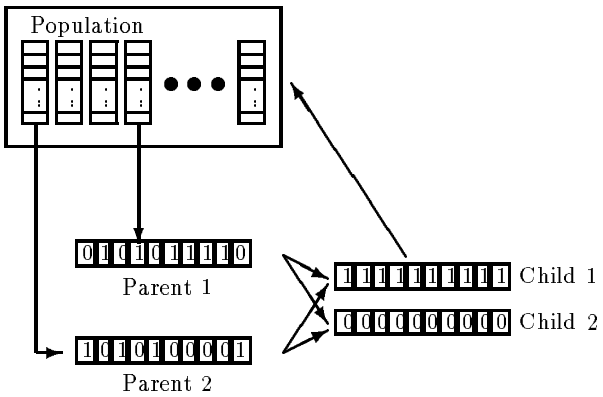


Figure 1: Recombination of Chromosomes

better results could be produced if we took the top ten members of a converged population and used them as members of a completely new population. This new population was then allowed to converge under the same criteria as the original population. This second-chance scheme is then repeated until one of the original convergence criteria is met.

## 4 Results and Conclusions

The results shown in the following tables represent the first time the convergent cost was produced. The actual running time was, on average, about 10% longer than when the best result was first produced. The best result produced by the algorithm is indicated by a <= in the right margin. Three different problem sizes were tried: 15, 20 and 30 points. The results for the two larger problems are presented in Table 1 and Table 2 respectively.

The genetic algorithm implemented to address the quadratic assignment problem produced results that were on average good, but not perfect. This finding is consistent with the processes that genetic algorithms attempt to model, i.e. "success begets success" but not necessarily perfection. Therefore, the implementation itself is a success.

The implementation exploited the parallelism offered by the MIMD distributed-memory message-passing machine (iPSC/2 Hypercube) in the forms of cross-pollination, second-chance, and concurrent effort (as previously described). It produced consistently better results than the earlier non-parallel implementation. Also, the value of cross-pollination in achieving better results was not conclusive (too much was bad, but what amount is too little?).

| Pop'n Size | Mut. Rate | Linear Bias | Poll. Rate | Sol'n Value | Num. Iter. | Time (in sec.) |
|---|---|---|---|---|---|---|
| 100 | 13 | 2.65 | 19 | 2656 | 425 | 17 |
| 200 | 13 | 2.65 | 19 | 2640 | 841 | 45 |
| 200 | 13 | 2.65 | 33 | 2598 | 838 | 36 |
| 200 | 13 | 3.10 | 19 | 2636 | 862 | 36 |
| 200 | 21 | 2.65 | 19 | 2674 | 857 | 49 |
| 300 | 11 | 3.00 | 33 | 2632 | 1337 | 79 |
| 350 | 13 | 3.75 | 33 | 2640 | 1616 | 80 |
| 375 | 13 | 3.75 | 33 | 2654 | 1438 | 59 |
| 400 | 13 | 2.65 | 19 | 2644 | 2004 | 104 |
| 400 | 11 | 3.00 | 33 | 2606 | 1964 | 113 |
| 400 | 11 | 2.00 | 33 | 2612 | 2630 | 130 |
| 400 | 11 | 4.00 | 33 | 2666 | 1296 | 81 |
| 400 | 9 | 3.00 | 33 | 2612 | 1789 | 104 |
| 400 | 13 | 3.00 | 33 | **2586** | 1967 | 116 |
| 400 | 15 | 3.00 | 33 | 2604 | 2068 | 112 |
| 400 | 13 | 2.50 | 33 | 2664 | 1845 | 75 |
| 400 | 13 | 3.75 | 33 | 2602 | 1905 | 125 |
| 400 | 13 | 3.85 | 33 | 2622 | 1604 | 72 |
| 400 | 13 | 4.00 | 33 | 2632 | 2617 | 164 |

<=

Table 1: Problem Size 20

| Pop'n Size | Mut. Rate | Linear Bias | Poll. Rate | Sol'n Value | Num. Iter. | Time (in sec.) |
|---|---|---|---|---|---|---|
| 200 | 15 | 2.65 | 19 | 6548 | 1336 | 78 |
| 200 | 10 | 2.65 | 19 | 6500 | 1099 | 63 |
| 200 | 10 | 3.00 | 19 | 6434 | 1053 | 57 |
| 300 | 15 | 2.65 | 19 | 6376 | 2568 | 177 |
| 400 | 10 | 2.65 | 19 | 6284 | 3402 | 232 |
| 400 | 10 | 2.65 | 25 | 6258 | 3120 | 174 |
| 400 | 5 | 2.65 | 25 | 6282 | 2777 | 255 |
| 400 | 5 | 2.65 | 22 | 6358 | 2693 | 184 |
| 400 | 5 | 2.65 | 28 | 6210 | 2657 | 160 |
| 400 | 5 | 2.65 | 30 | **6202** | 2402 | 170 |
| 400 | 10 | 2.65 | 28 | 6338 | 2791 | 179 |
| 400 | 7 | 2.65 | 25 | 6250 | 3601 | 295 |
| 400 | 7 | 2.65 | 28 | 6346 | 3613 | 227 |
| 400 | 15 | 2.50 | 21 | 6340 | 2560 | 189 |
| 400 | 13 | 3.10 | 15 | 6360 | 1822 | 125 |
| 400 | 13 | 3.75 | 33 | 6268 | 2836 | 188 |
| 400 | 7 | 3.75 | 33 | 6418 | 1953 | 156 |
| 400 | 7 | 3.75 | 25 | 6370 | 2428 | 149 |
| 400 | 13 | 3.00 | 33 | 6358 | 3293 | 218 |

<=

Table 2: Problem Size 30

The marriage of parallel processing and genetic algorithms seems to be reasonably effective for obtaining results for intractable problems. The method presented is somewhat crude; however, with refinement it appears to have the potential of becoming an even more valuable problem-solving heuristic. As there was no comparison between the presented algorithm and an equivalent exhaustive search technique, few, if any, conclusions can be made about the superiority of one method over the other. The performance of a traditional exhaustive search for an optimal solution is known to have a speedup linearly proportional to the number of processors working in concert on the problem. Genetic algorithms do not guarantee optimal solution and so do not provide speedup (in the technical sense of the word); however, they can provide satisfactory solutions in a short period of time, so the results generated by this implementation indicate that the genetic algorithm can be a valuable search tool.

## 5 Future Work

As with any project, extensions and modifications can be made to increase utility and/or performance, yet a further, and perhaps more significant, experiment would be to investigate and make use of the general characteristics of "good" solutions. For instance, the results produced by the program might have distinct characteristics which could be exploited in generating the initial population(s).

Research into the representation of the solution should also be performed. A genetic algorithm is only as good as its solution representation, and it is not clear that the one chosen for this implementation carried as much fitness information as feasible. On the other hand, a representation with more fitness information may be unwieldy in some way. It is thus likely that a compromise should be investigated, and it is possible that the one used herein is the best one available.

## References


[1] L. Davis, editor. *Handbook of Genetic Algorithms.* Van Nostrand Reinhold, New York, 1991.

[2] M.R. Garey and D.S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W.H. Freeman and Company, New York, 1982.

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

[4] L.S. Horowitz and F. Sahni. *Fundamentals of Data Structures in Pascal.* Computer Science Press, 1987.

[5] D.S. Johnson. The NP-Completeness column: an ongoing guide. *Journal of Algorithms*, 3(1):89–99, March 1982.