PROCEEDINGS OF THE 1992 ACM/SIGAPP SYMPOSIUM ON APPLIED COMPUTING (SAC'92), Kansas City Convention Center March 1-3, 1992.

Solving Combinatorial Optimization Problems Using Parallel Simulated Annealing and Parallel Genetic Algorithms*

Pooja P. Mutalik
Leslie R. Knight
Joe L. Blanton
Roger L. Wainwright

Department of Mathematical and Computer Sciences
The University of Tulsa

ABSTRACT

There are many combinatorial optimization problems for which there exists no direct or efficient method of solution. Simulated annealing (SA) and genetic algorithms (GA) are two promising techniques for solving large optimization problems. The authors have developed a parallel simulated annealing algorithm and a parallel genetic algorithm for a hypercube multiprocessor system. To compare the performance of these algorithms, we investigated two representative combinatorial optimization problems, the Traveling Salesman Problem (TSP) and the one-dimensional Package Placement Problem (PPP). The parallel genetic algorithm performed consistently better than the parallel simulated annealing algorithm in all of the cases tested. In addition, we tested five crossover functions on the sequential genetic algorithm for the Package Placement Problem and determined in every case the edge recombination crossover function was superior. There are some significant differences between genetic algorithms and simulated annealing that may account for the superior performance of the parallel genetic algorithm for these types of problems. We found it fairly easy to fine tune the parameters that drive a parallel GA for near optimal performance (population size, migration rate, and migration interval) compared to the parameters that drive a parallel simulated annealing algorithm. Furthermore, our parallel genetic algorithm is more mature than our newly developed parallel simulated annealing algorithm. Several future enhancements to the parallel simulated annealing algorithm are presented.

* Research partially supported by OCAST Grant AR0-038 and Sun Microsystems, Inc.

INTRODUCTION

In the fields of Operations Research and Artificial Intelligence, there are many combinatorial optimization problems for which there exists no direct or efficient method of solution. Researchers have recently become interested in solving large combinatorial optimization problems using large numbers of processors. These algorithms offer optimal or near optimal solutions to many important optimization problems [14, 15, 18]. On a sequential machine for large n, timings from $O(n^3)$ time to exponential time are commonly required. Since this is too expensive, less time consuming algorithms have been developed such as greedy algorithms. These less expensive algorithms rapidly find a "good" solution, but not an optimal solution or even near optimal solution. With the advent of parallel processors, new opportunities open up for effective solutions of combinatorial problems that were not available only a few years ago.

Finding a solution to many combinatorial optimization problems requires an organized search through the problem space. An unguided search is extremely inefficient since many of these problems are NP-complete. Genetic algorithms and simulated annealing are promising techniques for solving large optimization problems. In this paper we look at two representative combinatorial optimization problems, the Traveling Salesman Problem (TSP) and the one-dimensional Package Placement Problem (PPP). Both problems were implemented and solved using a parallel simulated annealing algorithm and a parallel genetic algorithm developed by the authors. The rest of the paper is presented as follows. In Section 2 we review the Traveling Salesman Problem and the Package Placement Problem. In Section 3 the fundamentals of simulated annealing are reviewed. In Section 4 a parallel simulated annealing algorithm is described. In Section 5 the fundamentals of genetic algorithms are reviewed and in Section 6 a parallel implementation is described. Results and conclusions are presented in Section 7. Future research issues are given in Section 8.

Package Placement and Traveling Salesman Problems

In the Package Placement Problem (sometimes called the Optimal Linear Arrangement Problem), we are given n packages and a network of interconnections between the packages. The problem can be stated as a undirected graph problem where the packages are the vertices and the edges are labeled as weights between a pair of packages. The weight $w(a,b)$ of edge (a,b) represents the number of "wires" connected between packages *a* and *b*. The problem is to linearly arrange the packages, p_1, p_2, ..., p_n, such that the sum of $|i-j|\, w(p_i,p_j)$ over all pairs *i* and *j* is minimized [1]. This problem has a number of applications. The packages could be boards, chips, cells, etc. and the interconnections are the number of wires to be connected between each of the packages. This concept can easily be extended to two dimensions, where the packages have a height and width in a two dimensional plane such as a circuit board.

Given a set of n points in a plane corresponding to the location of n cities, find the minimum distance closed path that visits each city exactly once. This is called the Traveling Salesman Problem. The Traveling Salesman Problem belongs to a class of minimization problems for which the objective function has many local minima. The objective function is simply the total length of the tour. The traveling salesman route can be thought of as a circular arrangement of n cities, or as a permutation of a list of n cities. Solving this problem requires O(n!) computation time since the number of possible tours for n cities is (n-1)!. For even a modest size n it would be impossible to check all possible permutations of n cities in a reasonable amount of time even with current parallel hardware. In this paper it is assumed each city is directly connected to every other city by Euclidian distance. That is, distances satisfy the triangle inequality, which means that the direct route between any two cities is never more than an indirect route between two cities. This assumption helps in the design of approximation algorithms for this problem.

We chose The Traveling Salesman Problem and the Package Placement Problem because they are representative problems of a wide variety of combinatorial optimization problems where the solution space is all permutations of n objects. Other combinatorial optimization problems that fall into this category include the Bin Packing Problem, Job Scheduling problems, Stock cutting, vehicle routing and transportation scheduling problems, etc. Developing efficient parallel simulated annealing and parallel genetic algorithms to solve these two problems will have direct applications for solving a host of other practical combinatorial optimization problems.

SIMULATED ANNEALING

Simulated annealing is a stochastic computational technique for finding near optimal solutions to large optimization problems. The method of simulated annealing is an analogy with thermodynamics, specifically in the manner that metals cool (anneal), or in the way liquids freeze and crystallize. If liquid metal cools too rapidly, atoms do not have time to line themselves up to form a pure crystal. Instead a rather high energy state is reached. However, if the metal is cooled slowly atoms have time to line themselves up to form a minimum energy system [19]. Metropolis [21] in 1953 was the first to incorporate these concepts into numerical calculations. Kirkpatrick [16] in 1983 was the first to use this concept to solve combinatorial optimization problems.

In the Package Placement Problem, simulated annealing starts with an arbitrary initial linear arrangement. The objective function, which is the sum of the length of the wires needed to connect all of the packages, is analogous to the current energy state of the system. A convergence condition of simulated annealing is that any state can be reached from any other state. A simple perturbation function, package exchange, is used to move from one state to another. That is, two packages exchange positions in the linear arrangement. Changes from one state to another (ie., a new placement from a previous placement) that result in a reduced objective function are always accepted. This is a analogous to slow cooling. However, changes that increase the objective function are only accepted with probability $p(d,T) = \exp(-d/T)$, where d is the change in the objective function from one arrangement to another, and T is the "temperature" of the system. The temperature parameter controls the annealing process. By occasionally allowing the energy of the system to rise slightly before cooling again is analogous to occasionally allowing for a worse package placement. This may allow the system to avoid falling into a local minima where it cannot get out.

The annealing schedule, controlled by the parameter T, requires some experimentation. Generally T is normalized and begins at one. At each step T is decreased by some amount, 0.01 for example, until a minimum state is encountered. In our system the user can input the initial temperature and the change in temperature at each step. Typically the system freezes at a certain temperature. At each temperature one iterates some number of times producing and accepting new package arrangements. This may range from one to several thousand iterations at a given temperature. Nahar *et al.* [23] give an excellent overview of simulated annealing.

PARALLEL SIMULATED ANNEALING

The parallel simulated annealing and parallel genetic algorithms have been developed on a hypercube. The algorithms can easily be adapted to shared memory multiprocessor systems or other distributed memory systems. An *n*-dimensional hypercube consists of 2^n processors interconnected as an *n*-dimensional binary cube. Each processor is a node of the binary cube with its own local memory and CPU. Each processor is directed connected to *n* other adjacent (neighboring) processors. Communication is a hypercube is accomplished by message passing. It is assumed the reader is generally familiar with the hypercube multiprocessor system.

Our algorithm for implementing the Package Placement Problem using simulated annealing has three phases. We use a ring topology hypercube arrangement. In general terms, phase 1 is concerned with the distribution of the packages among the processors and the interaction among the local packages within each processor. In phase 2 some interconnection scheme is used to move or redistribute the packages among the processors who have packages great distances apart. In phase three packages are exchanged (rotated) among adjacent processors in the ring.

Initially the packages are distributed evenly as possible among the ring of processors. The arrangement of the packages taken as a whole around the ring of processors constitutes the linear package arrangement. In phase 1 each of the local packages within each processor interact with each other using the package exchange perturbation function. That is, two packages in each processor are selected and exchange position. We tried two different ways for selecting the packages: randomly and adjacent packages. After extensive study we determined selecting randomly a pair of adjacent packages and exchanging them was superior to selecting two packages at random for exchange. The objective function is calculated to determine if this new arrangement is accepted. Global communication in a hypercube is very expensive and should be avoided, if possible. Notice each processor can determine if exchanging two of its packages results in a better arrangement independent of having to know how other processors are rearranging their packages. This is a vital part of the algorithm since global communication during this phase is not necessary. The following theorem addresses this issue.

Theorem:
Local exchange between two adjacent packages p_i and p_{i+1}, $1 <= i <= n$, is independent of the order of packages to their left and to their right. It is assumed packages to the left of p_i and to the right of p_{i+1} do not switch sides.

Proof:
Given two adjacent packages, p_i and p_{i+1}, an exchange is made if the following condition holds [1]:

$$\mathrm{lval}(p_i) - \mathrm{rval}(p_i) + \mathrm{rval}(p_{i+1}) - \mathrm{lval}(p_{i+1}) + 2\,w(p_i,p_j) < 0 \qquad (1)$$

where $\mathrm{lval}(p_i)$ is defined as the sum of weights between p_i and packages to the left of p_i, and $\mathrm{rval}(p_i)$ is defined as the sum of weights between p_i and packages to the right of p_i.

That is, $\mathrm{lval}(p_i) = \mathit{Sum}\ w(p_k, p_i)$, $k = 1,i-1$ and $\mathrm{rval}(p_i) = \mathit{Sum}\ w(p_k, p_i)$, $k = i+1,n$.

Notice, the distance between the packages is not considered in the formula. The condition (1) above can also be restated as: exchange p_i, p_j if the following expression is negative

$$\underset{k<i}{\mathit{Sum}}\ w(p_k, p_i) - \underset{k>i}{\mathit{Sum}}\ w(p_i, p_k) + \underset{k>i+1}{\mathit{Sum}}\ w(p_{i+1}, p_k)$$

$$- \underset{k<i+1}{\mathrm{Sum}}\ w(p_k, p_{i+1}) + 2\,w(p_i, p_{i+1})$$

Consider any two packages, p_x and p_y, to the left of p_i. Assume p_x is left of p_y. Now suppose p_x and p_y exchange positions and p_y is now left of p_x. This has no effect on the above criteria (1) for exchanging p_i and p_{i+1}. Similarly exchanging the position of any two packages to the right of p_{i+1} does not effect the criteria for exchanging p_i and p_{i+1}, as long as the two packages remain to the right of p_{i+1}, which proves the theorem.

After some number of local package exchanges a "hyperswap" (phase 2) step takes place. Here packages are exchanged with adjacent nodes in the hypercube along each of the dimensions of the hypercube. Thus in a d-dimensional hypercube a given node will cycle through all of its d adjacent neighbors in d successive phase 2 steps. This is traditionally called "hyperswaps". During phase 3, synchronization of the packages is accomplished by each processor shifting one half of its packages in a clock-wise direction to its neighboring node in the ring topology. This is called a half-spin. It is also during this phase that some global communication is performed among all of the processors before cycling back to phase 1 again.

Phase one of the algorithm performs local minimization whereas phase two enables two packages to be brought together which might be a great distance apart in the linear arrangement. These two phases, even implemented collectively, do not eliminate completely the local minima problem. Therefore, the half-spin was introduced to give rise to new combinations that will enhance the performance of local package exchanging and hyperswapping. A half-spin results in each node retaining 50% of its packages and receiving 50% new packages. Note this algorithm could work without phase 2, but the other phases are absolutely essential.

Our algorithm obtains a good mix of the packages. The simulated annealing process surrounds the three phases of the algorithm. The parallel PPP SA algorithm is described below:

```
1. Compute an initial package arrangement, p.
2. Choose the initial temperature parameter, T
3. Loop
4.   Loop
5.     Loop
         Compute new state, p' using package exchange.
         p = f(p',p)
         Until equilibrium
6.     Loop
         Compute new state, p' using hyperswapping
         p = f(p',p)
         Until equilibrium
7.     Shift clockwise half of the packages to the neighboring
       processor.
     Until equilibrium
8. Decrease T
   Until crystallization

function f(p',p)
1. If p' is a better package arrangement than p, then return(p')
2. d = objective function (p') - objective function (p)
3. return(p') with probability minimum(1,exp(-d/T))
   otherwise return(p)
4. end function
```

We have previously implemented the Traveling Salesman Problem using simulated annealing. This work is described in detail in [19]. Briefly this algorithm combines the strong points of three recent implementations [2, 3, 9] with some new features. Results show our algorithm to be vastly superior to these three algorithms. The TSP solution from our algorithm represented a 44%, 56%, and 52% improvement over Felton's algorithm [9], modified Allwright's algorithm [2], and the Braschi-like algorithm [3], respectively, for a 320 city problem. An improvement of 39%, 30%, and 55% occurred respectively, in the case of a 1024 city problem. We will be comparing the simulated annealing TSP results against the genetic algorithm implementations of the same 320 and 1024 city problems. Braschi [3] gives an excellent discussion of simulated annealing applied to the Traveling Salesman Problem.

GENETIC ALGORITHMS

The genetic algorithm designed by Holland [13] is a robust search and optimization technique based on the principles of natural genetics and survival of the fittest. Genetic algorithms use the laws of genetics and natural selection to guide a non-deterministic search. In genetics, a set of chromosomes contain all of the genes which form the "blueprint" for a species. In a genetic algorithm, a chromosome is a string which encodes a possible solution for a combinatorial problem. Instead of deoxyribonucleic acid (DNA) and protein, the genes are made up of bits, integers, reals, etc. The genetic algorithm has the ability to create an initial population of feasible solutions, and then recombine them in a way to guide its search to only the most promising areas of the state space.

Genetic Algorithms are applicable to a wide variety of problems. In particular, genetic algorithms are a promising new approach to global optimization problems. GAs have been used successfully for a wide range of applications areas including (1) scheduling: (facility, production, job, and transportation), (2) design: (circuit board layout, communication network design, keyboard layout, parametric design in aircraft), (3) control: (missile evasion, gas pipeline control, pole balancing), (4) machine meaning: (designing neural networks, classifier systems, learning rules), (5) robotics: (trajectory planning, path planning), (6) combinatorial optimization: (TSP, bin packing, set covering, graph bisection, routing, package placement), (7) signal processing such as filter design, and (8) image processing such as pattern recognition.

Genetic algorithms search through the solution space by emulating biological selection and reproduction. Each new generation is created by a biased reproduction, the more "fit" members of the population have a better chance of reproduction. The parameters of the model to be optimized are encoded into a finite length string, usually a string of bits. Each parameter is represented by a portion of the string. The string is called a chromosome, and each bit is called a gene. Each string is given a measure of "fitness" by the fitness function, sometimes called the evaluation or objective function. The fitness of a chromosome determines its ability to survive and reproduce offspring. The "least fit" or weakest chromosomes of the population are displaced by more fit chromosomes. Genetic algorithms are blind without the fitness function. The fitness function drives the population toward better solutions and is the most important part of the algorithm. The fitness function is what distinguishes one problem from another [17].

Genetic algorithms use probabilistic rules to evolve a population from one generation to the next. The transition rules going from one generation to the next are called genetic recombination operators. These include Reproduction (of the more "fit" chromosomes), Crossover, where portions of two chromosomes are exchanged in some manner, and Mutation. Crossover combines the "fittest" chromosomes and passes superior genes to the next generation thus providing new points in the solution space. Mutation is performed infrequently. A new individual (point in the solution space) is created by altering some of the bits of an individual. Mutation ensures the entire state space will eventually be searched (given enough time), and can lead the population out of a local minima. Genetic algorithms retain information from one generation to the next. This information is used to prune the search space and generate plausible solutions within the specified constraints [17]. Genetic algorithm packages for a single processor have been available for only a few years. GENITOR [28] and GENESIS [12] are the two such packages. Both of these packages have been installed at our university and have been used for developing sequential genetic algorithms. Goldberg and others provide an excellent in depth study of genetic algorithms [6, 7, 10, 24].

PARALLEL GENETIC ALGORITHMS

Several researchers have investigated distributed genetic algorithms on various architectures [5, 11, 17, 20, 25, 26, 27, 30]. These investigations have shown that parallel genetic algorithms work very well. Furthermore, parallel genetic algorithms have proved very successful in solving some NP-complete problems [4, 8, 22, 29].

HYPERGEN is a distributed genetic algorithm for a hypercube developed at our university [17]. HYPERGEN distributes the initial population evenly among the processors. Each processor (island) executes a sequential GA on its subpopulation performing crossover and mutation. HYPERGEN uses a one-at-a-time reproduction scheme, similar to GENITOR, where only a few members of the population are removed at each iteration. That is, the reproduction cycle consists of selecting two "fit" chromosomes from the population pool, performing crossover and perhaps mutation to yield a single offspring. The parents remain in the population pool. The offspring is placed into the population pool according to its fitness function, and the weakest chromosome is removed. The population pool size remains constant. After a prescribed number of reproductions, (called the *migration interval*) the "fittest" chromosomes in each processor are exchanged among other processors introducing new genetic material into each island. The amount of genetic material to exchange is called the *migration rate*. This process continues until the entire population stabilizes.

HYPERGEN is a modular collection of routines for generating the initial population, evaluation function, selection (based on a bias function), reproduction, mutation, migration interval, migration rate and summary statistics. The sequential GA on each processor and the periodic migration of genetic material between processors is performed automatically for the user.

In the Package Placement Problem for *n* packages a chromosome is simply a permutation of the numbers 1..*n* representing the number of the packages. The parallel genetic algorithm begins by generating a random set of chromosomes representing the initial population and distributing them evenly among all of the processors. Each processor evolves their own subpopulation by evaluating each chromosome, selection fit chromosomes for crossover, perform occasional mutation, and in general mature the population. After an appropriate number of generations chromosomes are exchanged among the processors in a hyperswap fashion as described earlier. Only the "fittest" chromosomes migrate to new populations. The Traveling Salesman Problem is implemented in a similar fashion. The chromosomes are represented in the same way, as a permutation of *n* integers. The only change between PPP and TSP is that a different evaluation function is used to determine the fitness of each chromosome. Of course, each problem has its own set of unique parameters that allows the genetic algorithm to perform better.

RESULTS AND CONCLUSIONS

Table I shows the results of our experiments for the Package Placement Problem using 32, 64, and 128 packages. We tested these problems using our parallel simulated annealing algorithm, parallel genetic algorithm (HYPERGEN), and a sequential genetic algorithm developed using the GENITOR package. For comparison purposes the optimum and worst package arrangements are included. The parallel genetic algorithm performed consistently better than the parallel simulated annealing algorithm and the sequential genetic algorithm except in the 128 case. In this case GENITOR took over 10 hours to complete, while HYPERGEN took only 2 hours and on a slower processor. We are convinced, given more time and fine tuning of the parameters, that HYPERGEN would obtain the same or better results than GENITOR. The sequential genetic algorithm given enough CPU time out performed the parallel simulated annealing algorithm in the package placement examples.

Table II shows the results of our experiments for the Traveling Salesman Problem for the parallel simulated annealing, parallel genetic algorithm, HYPERGEN, and a sequential genetic algorithm using GENITOR. Each algorithm was tested using datasets of 320 and 1024 cities. The cities were randomly placed in a square of length 500 on a side. As in the package placement problems, the parallel genetic algorithm performed consistently better than the parallel simulated annealing algorithm and the sequential genetic algorithm. Furthermore, the sequential genetic algorithm given enough CPU time out performed the parallel simulated annealing algorithm. Recall the parallel simulated algorithm used here [19] was considerably superior to other recently developed parallel simulated annealing algorithms.

All parallel problems were run on an iPSC hypercube using 32 processors. In the parallel simulated annealing algorithm for both the PPP and TSP the initial temperature was set to 1.0. The temperature decreased by 0.01 at each step until a final value of 0.01. Equilibrium is achieved when no changes occur during an iteration or 10 iterations, whichever comes first. The parallel genetic algorithm implementation for both the TSP and PPP problems over all datasets used a population set of 100 per node, with a migration interval of 5%, and a migration rate of 10%. That is, each processor performs 5 (5% of 100) reproductions before exchanging 10 (10% of 100) chromosomes with another processor. The parallel and sequential genetic algorithms both used the edge recombination crossover operator [29] for the Traveling Salesman Problem.

In the sequential GA (GENITOR) Package Placement Problem we tested several crossover functions: an edge recombination crossover adapted for the Package Placement Problem, PMX, position, cycle, and order2. These functions are described in [30]. We tested each of these crossover functions on the 32, 64, and 128 package problems. Results are shown in Table III. In every case the edge recombination crossover function yielded better results than any of the others. For the 32, 64, and 128 package problems the best results were 5712, 44,704, and 355,072, respectively. These are the values reported in Table I for GENITOR.

In Table III(a) for 32 packages, the sequential GA using the edge recombination crossover operator converged to the optimum solution after 32,000 trials. The results using the other crossover operators after 32,000 trials are given for comparison. In addition, we allowed the other crossover functions to continue to 100,000 trials to see how much they were able to improve. Notice even after 100,000 trials none of the other operators were able to match the edge recombination crossover operator after only 32,000 trials.

Similarly in Table III(b) for 64 packages, the edge recombination crossover operator converged to the optimum solution after 166,000 trials. We allowed the other crossover operators to continue to 600,000 trials and noted their results. Again even after 600,000 trials none of the other crossover operators could match the edge recombination crossover operator after only 166,000 trials. The same pattern is repeated again in Table III(c) using 128 packages. The edge recombination crossover operator was the best performer. Recall a trial is not a generation. During one trial two fit parents produce one child. All three chromosomes are placed into the population pool displacing the weakest chromosome.

Notice in all cases the CPU times do not differ very much from one crossover function to another. The edge recombination function has some initial one time expense to set up a table of values. This is the reason for the poor CPU performance of edge recombination operator compared to the others in Table III(a) for 32,000 trials. However, this one time expense paid for itself after 100,000 trials where the edge recombination CPU time was very competitive.

There are some significant differences between genetic algorithms and simulated annealing that may account for the superior performance of the parallel genetic algorithm for these types of problems. First, simulated annealing begins with one feasible solution to manipulate over and over, while the genetic algorithm has a population size in the hundreds or thousands. The temperature schedule is very sensitive and difficult to get just right for optimal performance in simulated annealing. In a distributed SA, since one feasible solution is distributed over several processors, it is very important which perturbation function to use so that each processor can improve their portion of the solution with out undoing optimization work performed by other processors. In other words, a series of local optimizations may not always lead to a global optimization for some perturbation functions. Parallel genetic algorithms, on the other hand, partition the population into separate islands for independent development, then exchange "fit" genetic material. This lends itself naturally to distributed processing. Furthermore, we have found it fairly easy to fine tune the parameters that drive a parallel genetic algorithm for near optimal performance, that is, population size, migration rate, and migration interval. We found the genetic algorithm parameters were easier to fine tune for good performance compared to the parameters that drive a parallel simulated annealing algorithm.

FUTURE RESEARCH ISSUES

The parallel genetic algorithm out performed the parallel simulated annealing algorithm for the problems tested. This does not mean we should give up on simulated annealing. HYPERGEN, while continuing to change, is still a more mature package compared to our newly developed parallel simulated annealing algorithm. Our implementation of the parallel simulated annealing algorithm is still being enhanced, and we intend to continue to develop additional options to the package. These options include allowing for other perturbation functions such as Single Move, where a single randomly selected item is moved to another location, and Cycle of Three, where three randomly selected items are exchanged in a three way cycle, etc. These perturbation functions as well as others are described in detail in [23]. In addition we are implementing what is called the sequence heuristic [23] which means we accept a new solution p' with $h(p') >= h(p)$ if the last *k* perturbations on p failed to generate a p' with $h(p') < h(p)$, where h is the objective function. That is, if we failed to find a better solution in *k* tries, we will accept the next solution, whatever it is. The parameter *k* is adjusted with temperature, and is a user supplied parameter. In addition, the current hyperswap is quite simple, and we intend to develop a more effective version.

ACKNOWLEDGMENTS

This research has been partially supported by OCAST Grant AR0-038. The authors also wish to acknowledge the support of Sun Microsystems, Inc.

REFERENCES

[1] A.V. Aho, J.E. Hopcroft, and J.D. Ullman, Data Structures and Algorithms, Addison-Wesley, 1987.

[2] J. Allwright and D. Carpenter, A Distributed implementation of Simulated Annealing for the Traveling Salesman Problem, *Parallel Computing* 10 (1989) 335-338.

[3] B. Braschi, Solving the Traveling Salesman Problem Using The Simulated Annealing on a Hypercube, *Proceedings of the Fourth Conference on Hypercubes, Concurrent Computers, and Applications*, March, 1989.

[4] D.E. Brown, C.L. Huntley and A.R. Spillane, "A Parallel Genetic Heuristic for the Quadratic Assignment Problem", *Proceedings of the Third International Conference on Genetic Algorithms,* Morgan Kaufmann, 1989.

[5] J. Cohoon, S. Hegde, W. Martin, and D. Richards, "Punctuated equilibria: a Parallel Genetic Algorithm, *Proceedings of the Second International Conference on Genetic Algorithms,* Lawrence Erlbaum, 1987.

[6] L. Davis, ed., *Handbook of Genetic Algorithms*, Van Nostrand Reinhold, 1991.

[7] L. Davis, ed., *Genetic Algorithms and Simulated Annealing,* Morgan Kaufmann Publisher, 1987.

[8] K.A. De Jong and W.M. Spears, "Using Genetic Algorithms to Solve NP- Complete Problems", *Proceedings of the Third International Conference on Genetic Algorithms,* June, 1989, pp. 124-132.

[9] E. Felton, S. Karlin and S. Otto, The Traveling Salesman Problem on a Hypercubic, MIMD Computer, *Proceedings of the 1985 International Conference on Parallel Processing*, August, 1985.

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

[11] M. Gorges-Scheuter "ASPARAGOS: A Asynchronous Parallel Genetic Optimization Strategy", *Proceedings of the Third International Conference on Genetic Algorithms,* Morgan Kaufmann, 1989.

[12] J. Grefenstette, GENESIS, Navy Center for Applied Research in Artificial Intelligence, Navy research Lab., Wash. D.C. 20375-5000.

[13] J.H. Holland "Adaptation in Natural and Artificial Systems", Ann Arbor: The University of Michigan Press, 1975.

[14] E. Horowitz and S. Sahni, "Fundamentals of Computer Algorithms", Computer Science Press, 1984.

[15] T.C Hu, "Combinatorial Algorithms", Addision-Wesley, 1982.

[16] S. Kirkpatrick, C.D. Gelatt Jr. and M.P. Vecchi, Optimization by Simulated Annealing, *Science*, May, 1983, vol. 220, pp 671-680.

[17] L. Knight and R. Wainwright, "HYPERGEN: A Distributed Genetic Algorithm on a Hypercube", submitted.

[18] L. Kronsjo, "Computational Complexity of Sequential and Parallel Algorithms", John Wiley, 1985.

[19] D.R. Mallampati, P.P. Mutalik and R.L. Wainwright, "A Parallel Multi-Stage Implementation of Simulated Annealing for the Traveling Salesman Problem", *Proceedings of the Sixth Distributed Memory Computing Conference,* April 28 - May 2, 1991.

[20] B. Manderick and P. Spiessens, "Fine-Grained Parallel Genetic Algorithms", *Proceedings of the Third International Conference on Genetic Algorithms*, Morgan Kaufmann, 1989.

[21] N. Metropolis, A.W. Rosenbluth, M.N. Rosenbluth, A.H. Teller, E. Teller, Equation of State Calculation by Fast Computing Machines *J. Chem. Phys.* vol. 21, 1953, p. 1087.

[22] H. Muehlenbein, "Parallel Genetic Algorithms, Population Genetics and Combinatorial Optimization", *Proceedings of the Third International Conference on Genetic Algorithms*, Morgan Kaufmann, 1989.

[23] S. Nahar, S.S. Sahni, and E. Shragowitz, "Simulated Annealing and Computational Optimization", *International Journal of Computer Aided VLSI Design,* vol. 1, pp. 1-23, 1989.

[24] G. Rawling, ed., *Foundations of Genetic Algorithms,* Morgan Kaufmann Publishers, 1991.

[25] P. Spiessens and B. Manderick, "A Massively Parallel Genetic Algorithm Implementation and First Analysis", *Proceedings of the Fourth International Conference on Genetic Algorithms*, Morgan Kaufmann, 1991.

[26] T. Starkweather, D. Whitley, and K. Mathias, "Optimization Using Distributed Genetic Algorithms," in Parallel Problem Solving from Nature, ed. H. Schwefel and R. Maenner, Springer Verlag, Berlin, Germany, 1991.

[27] R. Tanese, "Distributed Genetic Algorithms, *Proceedings of the Third International Conference on Genetic Algorithms,* ed. J.D. Schaffer, Morgan Kaufmann, 1989, pp. 434-439.

[28] D. Whitney and J. Kauth, GENITOR: A Different Genetic Algorithm, *Proceedings of the Rocky Mountain Conference on Artificial Intelligence,* Denver, Co., 1988, pp. 118-130.

[29] D. Whitney, T. Starkweather, and D. Fuquat, "Scheduling Problems and Traveling Salesman: The Genetic Edge Recombination Operator", *Proceedings of the Third International Conference on Genetic Algorithms,* June, 1989.

[30] D. Whitley and T. Starkweather, "GENITOR II: A Distributed Genetic Algorithm, *Journal of Experimental and Theoretical Artificial Intelligence,* 2(1990) 189-214.

Algorithm	Number of Packages		
	32	64	128
Parallel SA	5884	50,462	397,940
Parallel GA	5712	44,704	375,556
GENITOR	5712	44,704	355,072
The Optimum	5712	44,704	353,600
The Worst	10912	87,360	699,008

Table I: Results of the Package Placement Problem

Algorithm	Number of Cities	
	320	1024
Parallel SA	29,339	91,323
Parallel GA	15,308	51,784
GENITOR	18,676	88,436

Table II: Results of the Traveling Salesman Problem

Crossover Algorithm	Number of Trials			
	32,000		100,000	
	Solution	CPU Sec.	Solution	CPU Sec.
Edge	5712	190	5712	494
PMX	5854	163	5854	531
Position	6060	145	5954	502
Cycle	6168	172	6004	517
Order2	6340	167	6122	490

Table III(a): Sequential GA for the 32 Package Placement Problem Using Different Crossover Functions

Crossover Algorithm	Number of Trials			
	166,000		600,000	
	Solution	CPU Sec.	Solution	CPU Sec.
Edge	44,704	2,308	44,704	7,796
PMX	47,147	2,414	45,868	8,309
Position	47,420	2,006	45,908	7,083
Cycle	48,428	1,954	46,468	6,968
Order2	48,524	2,236	46,844	7,756

Table III(b): Sequential GA for the 64 Package Placement Problem Using Several Crossover Functions

Crossover Algorithm	Number of Trials			
	600,000		800,000	
	Solution	CPU Sec.	Solution	CPU Sec.
Edge	355,072	32,419	355,072	37,827
PMX	367,550	33,139	366,732	39,887
Position	376,962	25,400	366,930	29,040
Cycle	370,942	28,159	366,808	30,247
Order2	380,526	25,280	365,484	29,032

Table III(c): Sequential GA for the 128 Package Placement Problem Using Several Crossover Functions