Chapter 1

PARALLEL METAHEURISTICS FOR COMBINATORIAL OPTIMIZATION

Sandra Duni Ekşioğlu
Center for Applied Optimization, Department of Industrial and Systems Engineering, University of Florida, Gainesville, FL 32611 USA.
duni@cao.ise.ufl.edu

Panos M. Pardalos
Center for Applied Optimization, Department of Industrial and Systems Engineering, University of Florida, Gainesville, FL 32611 USA.
pardalos@ufl.edu

Mauricio G. C. Resende
Information Sciences Research Center, AT&T Labs Research, 180 Park Avenue, Room C241, Florham Park, NJ 07932 USA.
mgcr@research.att.com

Keywords: Parallel Local Search, Parallel GRASP, Parallel Genetic Algorithms, Parallel Simulated Annealing, Parallel Tabu Search, Variable Neighborhood Descent, Parallel Computing Environments, Heuristics, Combinatorial Optimization.

Abstract In this chapter, we review parallel metaheuristics for approximating the global optimal solution of combinatorial optimization problems. Recent developments on parallel implementation of genetic algorithms, simulated annealing, tabu search, variable neighborhood search, and greedy randomized adaptive search procedures (GRASP) are discussed.

1. INTRODUCTION

Search techniques are fundamental problem-solving methods in computer science and operations research. Search algorithms have been used to solve many classes of problems, including path-finding problems, two-player games, and constraint satisfaction problems. Classical examples of path-finding problems include many combinatorial optimization problems (e.g. integer programming) and puzzles (e.g. Rubic's cube, Eight Puzzle). Chess, backgammon, and Othello belong to the class of two player games, while a classic example of a constraint satisfaction problem is the eight-queens problem.

In this chapter, we focus on NP-hard combinatorial optimization problems. A combinatorial optimization problem P in its minimization form can be formulated as follows. Given a finite set $E = \{1, 2, \ldots, n\}$, the set of feasible solutions $F \subseteq 2^E$ of P, and an objective function $f : 2^E \rightarrow \mathbb{R}$, one wishes to find $S^* \in F$ such that $f(S^*) \leq f(S), \forall S \in F$, where $f(S)$ is the objective function defined as $f(S) = \sum_{e \in S} c(e)$, where $c(e)$ is the cost of including $e \in S$ in the solution S. For a specific optimization problem, the sets E and F, as well as the function $c(e)$ need to be defined. For combinatorial optimization problems, exact search algorithms, such as branch and bound or dynamic programming, may degenerate to complete enumeration. Because of this, exact approaches limit us to solve only moderately sized problem instances, due to the exponential increase in CPU time and memory when problem size increases. Therefore, in practice, heuristic search algorithms are necessary to find (not necessarily optimal) solutions to these problems. Heuristics explore a small portion of the solution space, where one believes good solutions can be found. Most heuristics for combinatorial optimization make use of a basic local search method called iterative improvement which can be described as in Figure 1.1 in its minimization form ([59, 74]).

The initial solution s_0 can be produced in many different ways. A solution s' is "near" another solution s if it is in the set of neighborhood solutions $N(s)$ of s. Usually, a solution is in the neighborhood of another if it can be obtained from the other solution by an elementary perturbation or move. The neighborhood is usually constructed as the set $M_N(s)$ of all elementary moves from s. Often the set $M_N(s)$ is too large for a practical algorithm, and one searches in a subset $C_N(s) \subset M_N(s)$ called the candidate set. Iterative improvement proceeds until a locally optimal solution is found. Once such a solution is found, no further improvement is possible.

One limitation of local search is getting trapped in local optima. To get around this limitation, several general purpose heuristic strategies have been developed in the last two decades. For large-scale problems, another limitation is the exponential computational complexity of iterative improvement [60]. Parallel local search algorithms are one way to cope with this problem. Parallel implementation can significantly in-

---

Layout of Iterative Improvement

Input: *A problem instance P, and a feasible solution $s_0 \in F$*

Output: *A (sub-optimal) solution $s \in F$*

1. Initialize $s = s_0$;

2. **while** (there exists $s' \in N(s)$ such that $f(s') < f(s)$) **do**
   (a) Select $s' \in C_N(s)$ such that $f(s') < f(s)$;
   (b) Let $s = s'$;

3. Output s as the (sub-optimal) solution;

Figure 1.1 Iterative Improvement

crease the size of the problems that can be solved. While there is a large body of work on search algorithms, work on parallel search algorithms is relatively sparse.

A survey of parallel local search algorithms is given by Verhoeven and Aarts [118]. They give a review of some concepts that can be used to incorporate parallelism into local search. They distinguish between single walk and multiple walk parallel local search and between asynchronous and synchronous parallelism. In a single walk algorithm only a single walk in the solution space is carried out, whereas in a multiple-walk algorithm several walks are performed simultaneously. In the class of single walk algorithms they distinguish between single step and multiple step parallel local search. The idea of single step parallelism is to evaluate neighbors simultaneously and subsequently make a single step. In an algorithm with multiple steps parallelism, several consecutive steps through the solution space are made simultaneously. In the class of multiple-walks they distinguish between algorithms that perform interacting walk and algorithms that perform multiple independent walks. Finally, both in single walk and multiple walk parallel local search, they distinguish between synchronous and asynchronous algorithms. In a synchronously parallel algorithm one or more steps of the algorithm are performed simultaneously by all processors, while synchronous parallelism requires a global clocking scheme that guarantees that communication occurs at fixed points of time. They observed that multiple-walk parallelism offers promising results to parallel tabu search and simulated annealing algorithms for a wide range of problems, and single-step parallelism can be used for most of the tabu search algorithms.

For a comprehensive overview of parallel search algorithms the reader is referred to [22, 80, 31, 32, 90, 16].

Macready et al. [71] report on some interesting investigations of the effect of local search on combinatorial optimization. Although they accept that local search methods constitute one of the most successful approaches to combinatorial optimization problems, they demonstrate that for a wide class of search techniques, increasing parallelism leads to better solution faster, but only to a certain point. At some degree of parallelism, the quality of solution abruptly degrades to that of random sampling.

In this chapter, we explore different approaches of parallel heuristic search for solving combinatorial optimization problems. We focus on issues of parallelization of genetic algorithms, simulated annealing, tabu (or taboo) search, variable neighborhood search, and GRASP (greedy randomized adaptive search procedures). These heuristic methods are often called metaheuristic since they are general purpose optimization schemes that can be adapted to address specific optimization problems. These metaheuristics have been used to approximately solve a wide spectrum of combinatorial optimization problems [98].

Portions of this chapter are based on Pardalos, Mavridou, Pitsoulis, and Resende [94].

2. PARALLEL VARIABLE NEIGHBORHOOD DESCENT

Variable neighborhood descent (VND) is a metaheuristic recently proposed by Hansen and Mladenović [45, 46, 47, 48, 49, 50, 51]. The approach is a local improvement heuristic as described in the previous section, with the difference that instead of utilizing a single neighborhood structure, several neighborhood structures are used. These extended neighborhoods allow searching for improving solutions that are "further" from the current solution, thus allowing the method to escape local optima with respect to a smaller neighborhood.

Let $N_1, N_2, \ldots, N_p$ be p neighborhood structures such that $N_k(s)$ is the set of solutions in the k-th neighborhood of s. One usually assumes that neighborhood N_{k+1} is larger than neighborhood N_k. Figure 1.2 illustrates variable neighborhood descent. The while loop (2) is repeated while $k < p$. During each iteration of the loop, a random starting

Layout of Variable Neighborhood Descent

Input: *A problem instance P, a feasible solution $s_0 \in F$, and neighborhoods $N_1, N_2, \ldots, N_p$.*
Output: *A (sub-optimal) solution $s \in F$*

1. Initialize $s = s_0$; `Improve` $=$ **true**;

2. **while** (`Improve` $==$ **true**) **do**
   (a) `Improve` $=$ **false**;
   (b) $k = 1$;
   (c) **while** $k \leq p$ **do**
      (i) Generate s' at random $N_k(s)$;

      (ii) Apply local search using N_1 and s' as the initial solution. Let s'' be the local optimum;
      (iii) **if** $f(s'') < f(s)$ **then do**
            Let $s = s''$;
            `Improve` $=$ **true**;
            break;
         else $k = k + 1$;
3. Output s as the (sub-optimal) solution;

Figure 1.2 Variable Neighborhood Descent

solution s' in the k-th neighborhood of s is generated and local search is applied using neighborhood N_1. Let s'' be the local optimal solution. If an improvement is found with local search, then $s = s''$ and the search restarts with $k = 1$. If no improvement is found in the local search, the search proceeds with $k = k + 1$.

Since VND is relatively new, it has not been investigated much from a parallelization point of view. Martins [75] outlined how parallel implementations of VND can be accomplished. In a low-level parallelization, each search of the neighborhood $N_k(s)$ could be done in parallel. $N_k(s)$ would be divided between the processors and each would return an improving neighbor in its partition. The best neighbor would be chosen as the current solution. In another strategy, $N_k(s)$ would again be partitioned and each processor would search for the best neighbor in its partition. Again, the best neighbor of the best would be chosen as the current solution. In a multi-search thread strategy, the processors would execute independent sequential VNDs and in the end, the best solution found would be output.

3. PARALLEL GENETIC ALGORITHMS

In the 1960's, biologists began to use digital computers to perform simulations of genetic systems. Although these studies were aimed at understanding natural phenomena, some were not too distant from the modern notion of a genetic algorithm (GA). Genetic algorithms, as they are used today, were first introduced by Holland [52]. Genetic algorithms try to imitate the development of new and better populations among different species during evolution, just as their early biological predecessors. Unlike most standard heuristic algorithms, GAs use information of a population of individuals (solutions) when they conduct their search for better solutions as opposed to only information from a single individual. GAs have been applied to a number of problems in combinatorial optimization. In particular, the development of parallel computers has made this an interesting approach.

A GA aims at computing sub-optimal solutions by letting a set of random solutions undergo a sequence of unary and binary transformations governed by a selection scheme biased towards high-quality solutions. Solutions to optimization problems can often be encoded to strings of finite length. GAs work on these strings [52]. The encoding is done through the structure named *chromosomes*, where each chromosome is made up of units called *genes*. The values of each gene are binary, and are sometimes called *alleles*. The problem is encoded by representing all its variables in binary form and placing them together in a single chromosome. A fitness function evaluates the quality of a solution represented by a chromosome.

There are several critical parts which strongly affect the success of genetic algorithms:

- Representation of the solutions.

- Generation of the initial population.

- Selection of the individuals in an old population that will be allowed to affect the individuals of a new population. In terms of evolution, this relates to the selection of suitable parents in the new population.

- Genetic operators, such as crossover and mutation. That is, how to recombine the genetic heritage from the parents in the previous generation.

If $P(t)$ denotes the population at time t, the GA can be described as in Figure (1.3).

Layout of Genetic Algorithm

Input: *A problem instance*

Output: *A (sub-optimal) solution*

1. $t = 0$, initialize $P(t)$, evaluate fitness of individuals in $P(t)$;

2. **while** (termination condition is not satisfied) **do**
   (a) $t = t + 1$;
   (b) Select $P(t)$, recombine $P(t)$ and evaluate $P(t)$;

3. Output the best solution among all the population as the (sub-optimal) solution;

Figure 1.3 Layout of Genetic Algorithm

$P(0)$ is usually generated at random. The evaluation of a population $P(t)$ involves computing the fitness of the individuals and checking if the current population satisfies certain termination conditions. Types of termination rules include:

- A given time limit which is exceeded.
- The population is dominated by a few individuals.
- The best objective function value of the populations is constant over a given number of generations.

Due to their inherent parallel properties, GAs have been successfully implemented on parallel computers, introducing this way a new group of GAs, *Parallel Genetic Algorithms* (PGAs). In a PGA the population is divided into subpopulations and an independent GA is performed on each of these subpopulations. Furthermore, the best individuals of a local population are transferred to the other subpopulations. Communication among the subpopulations is established to facilitate the operations of selection and recombination. There are two types of communication [86]: (1) *among all nodes*, where the best string in each subpopulation is broadcast to all the other subpopulations, and (2) *among the neighboring nodes*, i.e. only the neighboring subpopulations receive the best strings.

The most important aspects of PGAs, which result in a considerable speedup relative to sequential GAs, are the following [58]:

- *Local selection*, i.e. a selection of an individual in a neighborhood is introduced, in contrast with the selection in original GAs which is performed by considering the whole population.
- *Asynchronous behavior* which allows the evolution of different population structures at different speeds, possibly resulting in an overall improvement of the algorithm in terms of CPU time.
- *Reliability* in computation performance, i.e. the performance of one processor does not affect the performance of the other processors.

Jog et al. [58] consider two basic categories of *parallel genetic algorithms*:

1 *Coarse-grained* PGAs, where subpopulations are allocated to each processor of the parallel machine. The selection and recombination steps are performed within a subpopulation.

2 *Fine-grained* PGAs, where a single individual or a small number of individuals are allocated to each processor.

In the same reference, a review of parallel genetic algorithms applied to the *traveling salesman problem* (TSP) is presented. A PGA developed by Suh and Gucht [113], has been applied to TSPs of growing size (100-1000 cities). PGAs without selection and crossover, so called *independent strategies*, were used. These algorithms consist of running an "unlimited" number of independent sequential local searches in parallel. PGAs with *low amount of local improvement* were used and performed better in terms of quality solution than the independent strategies. In terms of computational time, the PGA showed nearly a linear-speedup for various TSPs, using up to 90 processors. The algorithms were run on a BBN Butterfly.

Another implementation of a PGA applied to the TSP can be found in [41], where an asynchronous PGA, called ASPARAGOS, is presented in detail. An application of ASPARAGOS was also presented by Muhlenbein [85] for the *quadratic assignment problem* (QAP) using a polysexual voting recombination operator. The PGA was implemented on QAPs of size 30 and 36 and for TSPs of size 40 and 50 with known solutions. The algorithm found a new optimum for the Steinberg problem (QAP of size 36). The numbers of processors used to run this problem were 16, 32, and 64. The 64 processor implementation (on a system with distributed memory) gave by far the best results in terms of computational time.

Battiti and Tecchiolli [9] presented parallelization schemes of genetic algorithms and tabu search for combinatorial optimization problems and in particular for *quadratic assignment* and *N-k problems* giving indicative experimental results.

Tanese [115] presents a parallel genetic algorithm implemented on a 64-processor NCUBE/six hypercube. Each processor runs the algorithm on each own subpopulation (coarse-grained PGA), and sends in an adjustable frequency a portion of good individuals to one of its neighboring processors. Each exchange takes place along a different dimension of the hypercube. The PGA is applied to a function optimization problem and its performance is compared with the corresponding GA, which is found to be similar. In addition, the PGA achieves comparable results with near linear speedup.

Pettey et al. [101] proposed an "island model" that restricts the communication among the subpopulations to some adjacent interconnected subpopulations [69]. The PGA was tested on four of DeJong's testbed of functions [24]. Population sizes of 50 to 800 were used, with the best answer in terms of quality solution, corresponding to size 50, which is approximately the theoretical optimal population size. The algorithm was implemented on an Intel *iPSC*, a message-based multiprocessor system with a binary *n-cube* interconnection network.

More recently, Lee and Kim [69] developed PGAs, based on the island model, for solving job scheduling problems with generally weighted earliness and tardiness penalties (GWET), satisfying specific properties, such as the *V-shaped schedule* around the due date. A binary representation scheme is used to code the job schedules into chromosomes. A GA is developed by parallelizing the population into subgroups, each of which keeps its distinct feasible schedules. The initial population is constructed so that the resulting genotype sequence satisfies the *V-shaped schedule.* Two different reproduction methods are employed, the *roulette wheel selection* and the *N-best selection method.* Also, the *crossover* and *mutation* operators are implemented in parallel. Several instances of problems with subpopulations of 30 chromosomes (jobs) and total population size of 100 to 200 where used to evaluate the efficiency of the PGA. The authors report that the roulette wheel selection scheme performs far better than the N-best selection in terms of quality solution, though the N-best selection gives good results in terms of CPU time. The mutation operator seems to improve the performance of the PGA. The chapter concludes, showing the superior performance of the parallel algorithm when compared to the sequential GA. In terms of CPU time, the parallelization of the population into several groups speeds up the convergence of the GAs as the size of the problem increases.

Parallel genetic algorithms have been applied to the graph partition problem [68], scheduling problems [19], and global optimization problems [111].

4. PARALLEL SIMULATED ANNEALING

The simulated annealing method (SA) was proposed in 1983 by Kirkpatrick et al. [62], based on the pioneering work of Metropolis et al. [79]. Since then, much research has been accomplished regarding its implementation (sequential and parallel) to solve a variety of difficult combinatorial optimization problems. Simulated annealing is based on the analogy between statistical mechanics and combinatorial optimization. The term *annealing* derives from the roughly analogous thermal process of melting at high temperatures and then lowering the temperature slowly based on an annealing schedule, until the vicinity of the solidification temperature is reached, where the system is allowed to reach the *ground state* (the lowest energy state of the system). Simulated annealing is a simple Monte Carlo approach to simulate the behavior of this system to achieve thermal equilibrium at a given temperature in a given annealing schedule. This analogy has been applied in solving combinatorial optimization problems. Given an objective function $f(x)$ over a feasible domain D, a generic simulated annealing algorithm for finding the global minimum of $f(x)$ is given in Figure 1.4.

An introduction to general concepts of parallel simulated annealing techniques can be found in Aarts and Korst [1]. Several parallel algorithms based on SA have been implemented for a variety of combinatorial optimization problems [3, 70, 63]. In the context of annealing, a parallel implementation can be presented in two forms [108]: (1) *functional parallelism*, which uses multiple processors to evaluate different phases of a single move, (2) *data parallelism*, which uses different processors or group of processors to propose and evaluate moves independently. The second form has the advantage of "easily scaling the algorithm to large ensembles of processors."

The *standard-cell approach* is a semi-custom designing method in which functional building blocks called *cells* are used to construct a part of, or an entire, VLSI chip [108]. Two basic approaches have been

---

Layout of Simulated Annealing

Input: *A problem instance*

Output: *A (sub-optimal) solution*

1. Generate initial solution at random and initialize temperature T;

2. **while** $(T > 0)$ **do**
   (a) **while** (thermal equilibrium not reached) **do**
      (i) Generate a neighbor state at random and evaluate the change in energy level ΔE;

      (ii) If $\Delta E < 0$ update current state with new state;

      (iii) If $\Delta E \geq 0$ update current state with new state with probability $e^{\frac{-\Delta E}{K_B\, T}}$;

   (b) Decrease temperature T according to annealing schedule;

3. Output the solution having the lowest energy;

Figure 1.4 Layout of Simulated Annealing

applied to the placement problem [108] – *constructive* methods and *iterative* methods. The SA technique belongs to the second group, as an approach that uses probabilistic hill climbing to avoid local minima. The `TimberWolf` program has been proposed as a version of SA implemented for the cell placement problem. It has been shown to provide enormous chip area savings compared to the already existed standard methods of cell layout. Jones and Banerjec [61] developed a parallel algorithm based on `TimberWolf`, where multiple cell moves are proposed and evaluated by using pairs of processors. The algorithm was implemented on a iPSC/1 Hypercube. Rose et al. [106] presented two parallel algorithms for the same problem. The *Heuristic Spanning* approach replaces the high temperature portion of simulated annealing, assigning cells to fixed sub-areas of the chip. The *Section Annealing* approach is used to speed-up the low temperature portion of simulated annealing. The placement here is geographically divided and the pieces are assigned to separate processors. Other parallel algorithms based on SA and implemented for the cell placement problem can be found in [15, 108, 119].

Greening [42] examines the *asynchronous parallel* SA algorithm in relation with the effects of calculation errors resulting from the parallel implementation or an approximate cost function. More analytically, the relative work analyzes the effects of *instantaneous* and *accumulated* errors. The first category contains the errors which result as the difference between the true and inaccurate costs computed at a given time. An accumulated error is the sum of a stream of instantaneous errors. The author proves a direct connection between the accumulated errors measured in previous research work, and annealing properties.

Most recently, Boissin and Lutton [10] proposed a new SA algorithm that can be efficiently implemented on a massively parallel computer. A comparison to the sequential algorithm has been presented by testing both algorithms on two classical combinatorial optimization problems: (1) the *quadratic sum assignment problem* and (2) the *minimization of an unconstrained* $0-1$ *quadratic function*. The numerical results showed that the parallel algorithm converges to high-quality suboptimal solutions. For the problem of minimization of quadratic functions with 1000 variables, the parallel implementation on a 16K Connection Machine was 3.3 times faster than the sequential algorithm implemented on a SPARC 2 workstation, for the same quality of solution.

SA general purpose software packages are provided for public use. Ingber provides an ASA-package (adaptive SA) [57] and Carter provides a straightforward, simulated annealing algorithm as a public available package [14]. A parallel simulated annealing library *parSA* can be found on [64].

5. PARALLEL TABU SEARCH

Tabu search (TS), first introduced by Glover [37, 38, 40], is a heuristic procedure to find good solutions to combinatorial optimization problems. A *tabu list* is a set of solutions determined by historical information from the last t iterations of the algorithm, where t is fixed or is a variable that depends on the state of the search, or a particular problem. At each iteration, given the current solution x and its corresponding neighborhood $N(x)$, the procedure moves to the solution in the neighborhood $N(x)$ that most improves the objective function. However, moves that lead to solutions on the tabu list are forbidden, or are tabu. If there are no improving moves, TS chooses the move which least changes the objective function value. The tabu list avoids returning to the local optimum from which the procedure has recently escaped. A basic element of tabu search is the *aspiration* criterion, which determines when a move is admissible despite being on the tabu list. One *termination* criterion for the tabu procedure is a limit in the number of consecutive moves for which no improvement occurs. A more detailed description of the main features, aspects, applications and extensions of tabu search, can be found in [39]. Several implementations of parallel algorithms based on TS have been developed for classical optimization problems, such as TSP and QAP, which will be discussed below. Given an objective function $f(x)$ over a feasible domain D, a generic tabu search for finding an approximation of the global minimum of $f(x)$ is given in Figure 1.5.

Layout of Tabu Search

Input: *A problem instance*

Output: *A (sub-optimal) solution*

1. Initialization:
   (a) Generate an initial solution x and set $x^* = x$;
   (b) Initialize the tabu list $T = \emptyset$;
   (c) Set iteration counters $k = 0$ and $l = 0$;

2. **while** $(N(x) \setminus T \neq \emptyset)$ **do**
   (a) $k = k + 1$; $l = l + 1$;
   (b) Select x as the best solution from set $N(x) \setminus T$;
   (c) If $f(x) < f(x^*)$ then update $x^* = x$ and set $l = 0$;
   (d) If $k = \overline{k}$ or if $l = \overline{l}$ go to step 3;

3. Output the best solution found x^*;

Figure 1.5 Layout of Tabu Search

Taillard [114] presents two implementations of parallel TS for the quadratic assignment problem. Computational results on instances of size up to 64 are reported. The form of TS considered is claimed to be more robust than earlier implementations, since it requires less computational effort and uses only the basic elements of TS (moves to neighboring solutions, tabu list, aspiration function). The neighborhood used is a *2-exchange neighborhood.* The tabu list is made up of pairs (i, j) of interchanged units both of which are placed at locations they had occupied within the last s iterations, where s is the size of tabu list. The size s changes its value randomly in an interval during the search. The *aspiration criterion* is introduced to allow the tabu moves to be chosen, if both interchanged units are assigned to locations they have not occupied within the t most recent iterations. In the first method, the neighborhood is divided into p parts of approximately the same size. Each part is distributed for evaluation to one of p different processors. Using a number of processors proportional to the size of the problem (precisely $p = n/10$), the complexity is reduced by a factor of n, where n is the size of the problem. Moreover, the computational results showed improvement to the quality of solution for many large problems and the best published solutions of other problems have been found. The second method performs independent searches, each of which starts with a different initial solution. The parallel algorithm was implemented on a ring of 10 transputers (T800C-G20S). The computational complexity of this algorithm is less than that of earlier TS implementations for QAP by a factor n.

Another parallel tabu search algorithm for solving the quadratic assignment problem has been developed by Chakrapani and Skorin-Kapov [17]. The algorithm includes elements of TS, such as aspiration criterion, dynamically changing tabu list sizes and long-term memory [37]. A new intensification strategy is proposed, based on the intermediate term memory, restricting the searches in a neighborhood. This results in less amount of computational effort during one iteration since the procedure does not examine the entire neighborhood. A massively parallel implementation was tested on the Connection Machine CM-2 for large QAPs of size ranging from $n = 42$ to 100, using n^2 processors. The new tabu strategy gave good results in terms of quality of solutions. For problems up to size 90, it obtained the best known solutions or close to those. For size $n = 100$, every previously best solution found, was improved upon.

Battiti and Tecchiolli [9] describe a parallelization scheme for tabu search called *reactive* tabu scheme, and apply it to the quadratic assignment problem and the N-k problem with the objective of achieving a speedup in the order of the number of processors. In the reactive tabu search, each processor executes an independent search. Furthermore, the same problems are solved by a parallel genetic algorithm in which the interaction between different search processes is strong because the generation of new candidate points depends on the consideration of many members of the population.

A tabu search for the traveling salesman problem has been proposed by Fiechter [33], and was tested on instances having 500 to 10000 vertices. A new estimation of the asymptotic normalized length of the shortest tour through points uniformly distributed in the unit square is given. Numerical results and speedups obtained by the implementation of the algorithm on a network of transputers show the efficiency of the parallel algorithm.

6. PARALLEL GRASP

A GRASP [28, 30] is an iterative process for finding approximate solutions to combinatorial optimization. Let, as before, $f(x)$ denote the objective function to be minimized over a feasible domain D. A generic layout of GRASP is given in Figure 1.6. The GRASP iterations

---

Layout of GRASP

Input: *A problem instance*

Output: *A (sub-optimal) solution*

1. Initialization: set $x^* = \infty$;
2. **while** (stopping criterion not satisfied) **do**
    (a) Construct a greedy randomized solution x;
    (b) Find local minimum $\tilde{x}$ in neighborhood $N(x)$ of x;
    (c) If $f(\tilde{x}) < f(x^*)$ then update $x^* = \tilde{x}$;
3. Output the best solution found x^*;

Figure 1.6 Layout of GRASP

terminate when some stopping criterion, such as maximum number of iterations, is satisfied. Each iteration consists of a construction phase, and a local search phase. If an improved solution is found, then the incumbent is updated. A high-level description of these two phases is given next.

In the construction phase, a feasible solution is built up, one element at a time. The choice of the next element to be added is determined by ordering all elements in a candidate list with respect to a greedy function. An element from the candidate list is randomly chosen from among the best candidates in the list, but is not necessarily the top candidate. Figure 1.7 displays a generic layout for the construction phase of a GRASP. The solution set S is initially made empty, and the

---

Layout of GRASP Construction Phase

Input: *A problem instance and pseudo random number stream*

Output: *A (sub-optimal) solution*

1. Initialization: set solution $S = \emptyset$;
2. **while** (solution construction not done) **do**
    (a) Using greedy function, make restricted candidate list (RCL);
    (b) At random, select element s from RCL;
    (c) Place s in solution, i.e. $S = S \cup \{s\}$;
    (d) Change greedy function to take into account updated S;
3. Output the solution x corresponding to set S;

Figure 1.7 Layout of GRASP Construction Phase

construction iterations are repeated until the solution is built. To do this, the restricted candidate list is setup. This candidate list contains a subset of candidates that are well ordered with respect to the greedy function. A candidate selected from the list at random is added to the solution. The greedy function is adapted to take into account the selected element.

The solutions generated in the construction are not guaranteed to be locally optimal and therefore local search is applied to produce a locally optimal solution. Figure 1.8 illustrates a generic local search procedure.

Parallel implementation of GRASP is straightforward. Two general strategies have been proposed. In search space decomposition, the search space is partitioned into several regions and GRASP is applied to each in parallel. An example of this is the GRASP for maximum independent set [29, 104] where the search space is decomposed by fixing two vertices to be in the independent set. In iteration parallelization, the GRASP iterations are partitioned and each partition is assigned to a processor. See

Layout of GRASP Local Search Phase

Input: *A problem instance, solution x, neighborhood $N(x)$*

Output: *A locally-optimal solution $\tilde{x}$*

1. **while** (x not locally optimal) **do**
   (a) Find $\tilde{x} \in N(x)$ with $f(\tilde{x}) < f(x)$;
   (b) Update $x = \tilde{x}$;
3. Output the locally optimal solution $\tilde{x}$;

Figure 1.8 Layout of GRASP Local Search Phase

[87, 94, 95, 96, 103] for examples of parallel implementations of GRASP. Some care is needed so that different random number generator seeds are assigned to the different iterations. This can be done by running the random number generator through an entire cycle, recording all N_g seeds in a `seed` array. Iteration i is started with `seed`(i). GRASP has been implemented on distributed architectures. In [96] a PVM-based implementation is described. Two MPI-based implementations are given in [6, 76]. Alvim [6] proposes a general scheme for MPI implementations. A master process manages seeds for slave processors. It passes blocks of seeds to each slave processor and awaits the slaves to indicate that they have finished processing the block and need another block. Slaves also pass back to the master the best solution found for each block of iterations.

Aiex et al. [2] study the probability distribution of solution time to a sub-optimal target value for GRASP. They conclude that the solution time to a sub-optimal target value fits a two-parameter exponential distribution, which shows that it is possible to approximately achieve linear speed-up by implementing GRASP in parallel. The same behavior is observed in a number of metaheuristics such as SA [26, 89], Iterative Improvement for the TSP [116], and Tabu search [9, 114].

7. PARALLEL COMPUTING ENVIRONMENTS

The development of efficient and user friendly software tools, together with the availability of a wide range of parallel machines with large processing capacities, high reliability, and low costs, have brought parallel processing into reality as an efficient way for implementing techniques to solve large scale optimization problems. A good choice of the programming environment is essential to the development of a parallel program.

Parallel environments are composed of programming tools, performance evaluation tools, debuggers, and optimization libraries. Parallel programs consist of a number of processes working together. The processes are executed on parallel machines, based on different memory organization methods, as shared memory or distributed memory. In shared memory machines, all processors are able to address the whole memory space. The computers communicate through operations performed by the parallel tasks on the shared memory. Each task shares a common address space, which they can asynchronously access. The advantage of this approach is that the communication can be easy and fast. However, there is a limitation to this approach, system stability is limited by the number of paths between the memory and the processors. To overcome this disadvantage, local memory is added to each processor. An alternative to the shared memory organization is the distributed memory organization. In the framework of the distributed memory organization, the memory is physically distributed among the processors. Each processor can only access its own memory, and communication between processors is performed by messages passed through the communication network. Networks of workstations are an example of this architecture. A drawback of this architecture is that the control of the memory access that is required to maintain the consistency of the data has to be done by the programmer.

The parallel programming tools used in shared or distributed memory organization, are usually sequential languages augmented by a set of special system calls. The calls provide primitive message passing, process synchronization, process creation, mutual exclusion, and other functions.

In developing parallel programs, two types of parallelism can be used: data parallelism and functional parallelism [34, 67, 82]. Data parallelism programs apply the same instruction set to different items of a data structure. Programming languages called data-parallel programming have been developed to help with data parallelism. One of these languages is HPF. Functional parallelism is based on partitioning the program into cooperative tasks. Each task executes a different set of functions/codes and the tasks run asynchronously. The application of each technique depends on the characteristics of the problem to be solved. However, functional and data parallelism are complementary techniques and sometimes can be applied together to different parts of the same problem.

Many programming tools are available to implement parallel programs, each of them being suitable for specific problem types. Examples of programming tools are:

- **PVM** (Parallel Virtual Machine) [36] is an extensively used message passing library that supports the development of parallel programs on interconnected heterogeneous machines.
- **MPI** (Message Passing Interface) [34, 83, 84, 43, 44] is a proposal for the standardization of a message passing interface for distributed memory parallel machines, to enable program portability among different parallel machines.
- **Linda** [12, 13] is a language that offers a collection of tools for process creation and communication, based on the concept of an associative shared memory.
- **HPF** (High Performance FORTRAN) [53, 65, 66, 78] extends FORTRAN 90 with parallel construct and data placement features.
- **Threads** are a general operating systems concept, not specifically related to parallel programming. However, their understanding is essential to a parallel programmer, because of the support they provide in concurrent programming. In 1995, IEEE defined a standard Application Program Interface (API) for programming with threads, known as *POSIX threads* or *Pthreads* [56].

In the process of evaluating the performance of a parallel implementation, it is important to collect suitable and reliable data. There are three basic categories of data collectors: profilers, counters, and event traces.

Profilers record the amount of time spent in different parts of a program, usually in subroutines. This information is often obtained by sampling the program counter and generating a histogram of the execution frequencies. These frequencies are used to estimate the amount of time spent in different program components. *Gprof* is a commercial tool provided by IBM to profile parallel programs developed using the IBM MPI library. *Gprof* shows execution times of the subroutines executed by each processor.

Counters record the time a specific event occurs. *Counters* are used to record global events of the program, such as the number of procedure calls, the number of messages exchanged between pairs of processors, etc. This information cannot be obtained by profilers.

Event traces record each occurrence of specific events. They can be used to investigate relationships between communications and determine sources of idle time and communication bottlenecks. Some of the public domain and commercial packages available for *event traces* are:

- *AIMS* (Automated Instrumentation and Monitoring System) [120, 121] is a public domain package that can be downloaded from *http://science.nas .nasa.gov/Software/AIMS/*.
- *Paradyn* [81] handles the interactive run-time analysis for parallel programs using message passing libraries such as PVM and MPI. *Paradyn* can be downloaded from *http://www.cs.wisc.edu/paradyn/*.
- The *Pablo* performance analysis environment [25, 105] is a public domain package that consists of several components for instrumenting and tracing parallel MPI programs. It can be downloaded from *http://vibes.cs. uiuc.edu/*.
- *VAMPIR* (Visualization and Analysis of MPI resource) [35, 88] is a commercial trace visualization tool from Pallas GmbH. Information about this tool is available at *http://www.pallas.de/pages/-vampir.htm*.
- *VT* (Visualization Tool) can be used for post-mortem trace visualization or for on-line performance monitoring of parallel programs developed with the IBM implementation of MPI. It is a commercial tool provided by IBM and can only be used on the IBM SP parallel environment.
- *XPVM* is a public domain software that provides a graphical console and monitor for PVM. It is available from *http://www.netlib.-org/pvm3/xpvm*.

A common way to debug a parallel program is by inserting instructions in the code to print relevant information. However, often the output data lacks temporal information, since the information is not necessarily printed in the same order as generated by the parallel program, due to data buffering in the processors.

PVM and LAM public domain implementation of MPI allow the programmer to launch debuggers on remote nodes. Sometimes this kind of technique may interfere with the execution of the parallel program, making it behave differently from the way it runs. Total View is a commercial debugger developed by Etnus Inc. [27] that can be used to debug MPI, PVM, HPF, and multi-threaded programs. Tuplescope [109] is a debugging tool for Linda programs.

Optimization libraries consist of a set of subroutines that can be used to solve large optimization problems. These libraries include functions that solve linear and mixed-integer programming problems, provide important features such as control variables, debugging capabilities, file reading and writing etc. CPLEX [21] and OSL provide parallel implementations of their libraries. For an extensive review of parallel computing environments refer to [77, 8].

8. CONCLUDING REMARKS

Since the 1980's, we have witnessed an explosion and availability of parallel, multiprocessing computers. At the same time, we have also seen the development of new, powerful, metaheuristics for combinatorial optimization. Since most of the interesting combinatorial optimization problems are NP-hard, it is quite natural to consider implementing algorithms on such environments. In this brief survey, we summarized recent developments in parallel implementations of several classes of new heuristics for combinatorial optimization.

References

[1] E. Aarts and J. Korst, *Simulated Annealing and Boltzmann Machines – A Stochastic Approach to Combinatorial Optimization and Neural Computing*, John Wiley and Sons, 1989.

[2] R. M. Aiex and M.G.C. Resende and C.C. Ribeiro, *Probability Distribution of Solution Time in GRASP: an Experimental Investigation*, AT&T Labs Research Technical Report: 00.7.2 (2000).

[3] I. Azencott, *Simulated Annealing: Parallelization Techniques*, Wiley (1992).

[4] S.G. Akl, D.T. Barnard and R.J. Doran, *Design, Analysis, and Implementation of a Parallel Tree Search Algorithm*, IEEE Transactions on Pattern Analysis and Machine Intelligence, Vol. PAMI-4 (1982), pp. 192–203.

[5] I. Alth, *A Parallel Game Tree Search Algorithm with a Linear Speedup*, Journal of Algorithms 15 (1993), pp. 175–198.

[6] A.C.F. Alvim. Parallelization strategies for the metaheuristic GRASP. Master's thesis, Department of Computer Science, Catholic University of Rio de Janeiro, Rio de Janeiro, RJ 22453-900 Brazil, April 1998. In Portuguese.

[7] G.Y. Ananth, V. Kumar and P. M. Pardalos, *Parallel Processing of Discrete Optimization Problems*, In *Encyclopedia of Microcomputers* Vol. 13 (1993), pp. 129–147, Marcel Dekker Inc., New York.

[8] D. Bader http://computer.org/parascope (2000).

[9] R. Battiti and G. Tecchiolli, *Parallel Biased Search for Combinatorial Optimization: Genetic Algorithms and TABU*, Microprocessors and Microsystems 16 (1992), pp. 351–367.

[10] N. Boissin and J.-L. Lutton, *A Parallel Simulated Annealing Algorithm*, Parallel Computing, 19 (1993), pp. 859–872.

[11] R.J. Brouwer, P. Banerjee, *A Parallel Simulated Annealing Algorithm for Channel Routing on a Hypercube Multiprocessor*, Proceedings of 1988 IEEE International Conference on Computer Design, pp. 4–7.

[12] N. Carriero and D. Gelernter and T. Mattson, *Linda in Context*, Communications of the ACM 32 (1989), pp. 444–458.

[13] N. Carriero and D. Gelernter and T. Mattson, *The Linda Alternative to Message-Passing Systems*, Parallel Computing, Vol. 20 (1994), pp. 458–633.

[14] E. Carter (2000), http://www.taygeta.com/annealing/simanneal.html.

[15] A. Casotto and A. Sanngiovanni-Vincentelli, *Placement of Standard Cells Using Simulated Annealing on the Connection Machine*, Proceedings ICCAD, 1987, pp. 350–352.

[16] Y. Censor and S.A. Zenios, *Parallel Optimization: Theory, Algorithms, and Applications*, Oxford University Press (1997).

[17] J. Chakrapani and J. Skorin-Kapov, *Massively Parallel Tabu Search for the Quadratic Assignment Problem*, `Annals of Operation Research`, 41 (1993), pp.327–341.

[18] R.D. Chamberlain, M.N. Edelman, M.A. Franklin, E.E. Witte, *Simulated Annealing on a Multiprocessor*, Proceedings of 1988 IEEE International Conference on Computer Design, pp. 540–544.

[19] G.A. Cleveland and S.F. Smith, *Using Genetic Algorithms to Schedule Flow Shop Releases*, Proceeding of the Third International Conference on Genetic Algorithms, (1990), Morgan Kaufmann, Los Altos, CA.

[20] J.P. Cohoon, S.U. Hegde, W.N. Martin and D. Richards, *Punctuated Equilibria: A Parallel Genetic Algorithm*, Proceedings of the Second International Conference on Genetic Algorithms and their Applications, J.J. Grefenstette (editor), July 1987, pp. 148–154.

[21] CPLEX, *Home Page of CPLEX, a division of ILOG* (2000),

[22] T.G. Crainic and M. Toulouse, *Parallel Metaheuristics*, Technical report, Centre de recherche sur les transports, Université de Montr'eal, November 1997.

[23] D. Cvijovic and J. Klinowski, *Taboo Search: An Approach to the Multiple Minima Problem*, Science, 267 (1995), pp. 664–666.

[24] K.A. De Jong, *An Analysis of the Behavior of a Class of Genetic Adaptive Systems*, Doctoral dissertation, Department of Computer and Communication Sciences, University of Michigan, 1975.

[25] L. DeRose and Y. Zhang and D.A. Reed, *SvPablo: A Multi-language Performance Analysis System*, 10-th International Conference on Computer Performance Evaluation - Modeling Techniques and Tools (1998), pp. 352–355.

[26] N. Dodd, *Slow Annealing Versus Multiple Fast Annealing Runs: An Empirical Investigation*, Parallel Computing, Vol. 16 (1990), pp. 269–272.

[27] Etnus Com., *Home page for Etnus Com.* (1999),

[28] T.A. Feo and M.G.C. Resende, *A probabilistic heuristic for a computationally difficult set covering problem*, Operations Research Letters, 8 (1989) pp. 67–71.

[29] T.A. Feo and M.G.C. Resende and S.H. Smith, *A Greedy Randomized Adaptive Search Procedure for Maximum Independent Set*, Operations Research, 42 (1994), pp. 860–878.

[30] T.A. Feo and M.G.C. Resende, *Greedy Randomized Adaptive Search Procedures*, Journal of Global Optimization, 6 (1995), pp. 109–133.

[31] A. Ferreira and P.M. Pardalos (Eds.), *Solving Combinatorial Optimization Problems in Parallel, Methods and Techniques*, Lecture Notes in Computer Science (1995), 1054.

[32] A. Ferreira and J.D.P. Rolim (Eds.), *Parallel Algorithms for Irregular Problems: State of the Art*, Kluwer Academic Publishers (1995).

[33] C.-N. Fiechter, *A Parallel Tabu Search Algorithm for Large Traveling Salesman Problems*, Discrete Applied Mathematics, 51 (1994), pp. 243–267.

[34] I. Foster, *Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering* Addison-Wesley (1995).

[35] J. Galarowicz and B. Mohr, *Analyzing Message Passing Programs on the Cray T3 with PAT and VAMPIR*, Proceedings of Fourth European CRAY-SGI MPP workshop, H. Lederer and F. Hertwick, eds. IPP-Report des MPI für Plasmaphysik, IPP R/46 (1998), pp. 29–49,

[36] A. Geist and A. Beguelin and J. Dongarra and W. Jiang and R. Manchek and V. Sunderman, *PVM: Parallel Virtual Machine - A user's guide and tutorial for networked parallel computing*, MIT Press (1994), (http://www.netlib.org/pvm3/book/pvm-book.html).

[37] F. Glover, *Tabu Search. Part I*, ORSA J. Comput., 1 (1989), pp. 190–206.

[38] F. Glover, *Tabu Search. Part II*, ORSA J. Comput., 2 (1990), pp. 4–32.

[39] F. Glover, E. Taillard and D. de Werra, *A User's Guide to Tabu Search*, `Annals of Operation Research`, 41 (1993), pp. 3–28.

[40] F. Glover and M. Laguna, *Tabu Search*, Kluwer Academic Publishers (1997).

[41] M. Gorges-Schleuter, *ASPARAGOS: A Parallel Genetic Algorithm and Population Genetics*, `Lecture Notes on Computer Science`, Vol. 565 (1989), pp. 407–518.

[42] D.R. Greening, *Asynchronous Parallel Simulated Annealing*, `Lectures in Complex Systems`, Vol. 3 (1990), pp. 497–505.

[43] W. Gropp and E. Lusk and A. Skjellum, *Using MPI: Portable Parallel Programming with the Message Passing Interface*, MIT Press (1994).

[44] W. Gropp and S. Huss-Lederman and A. Lumsdaine and E. Lusk and B. Nitzberg and W. Saphir and M. Snir, *MPI: The Complete Reference*, In The MPI Extensions, Vol. 2, MIT Press (1998).

[45] P. Hansen and B. Jaumard and N. Mladenović and A. Parreira, *Variable Neighborhood Search for Weighted Maximum Satisfiability.* In preparation (2000).

[46] P. Hansen and N. Mladenović, *Variable Neighborhood Search for the* p-*Median.* Location Science, Vol. 5, pp. 207–226, 1998.

[47] P. Hansen and N. Mladenović, *An Introduction to Variable Neighborhood Search.* In *Metaheuristics, Advances and Trends in Local Search Paradigms for Optimization*, S. Voss et al. (Eds.), pp. 433–458, 1999, Kluwer, Dordrecht.

[48] P. Hansen and N. Mladenović, *Variable Neighborhood Search: Principles and Applications.* European Journal of Operational Research (to appear).

[49] P. Hansen and N. Mladenović, *Variable Neighborhood Search.* In *Handbook of Applied Optimization*, P.M. Pardalos and M.G.C. Resende (Eds.), Oxford University Press (2001).

[50] P. Hansen and N. Mladenović and N. Labbe and M. Gendreau, *Variable Neighborhood Search for the Traveling Salesman Problem.* In preparation (2000).

[51] P. Hansen and N. Mladenović and N. Perez-Brito, *Variable Neighborhood Decomposition Search.* Journal of Heuristics (to appear).

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

[53] High Performance FORTRAN Forum, *High Performance FORTRAN Language Specification Version 2.0*,

[54] S. R. Huang and L.S. Davis, *Speedup Analysis of Centralized Parallel Heuristic Search Algorithms*, Proceedings of the International Conference on Parallel Processing, Vol. 3. Algorithms and Applications (1990), pp. 18–21.

[55] IBM, *Parallel ESSL Version 2 Release 1.2 Guide and Reference (SA22-7273)*, (1999),

[56] IEEE (Institute of Electric and Electronic Engineering), *Information Technology - Portable Operating Systems Interface (POSIX) - Part 1 - Amendment 2: Threads Extension*, (1995).

[57] L. Ingber, *ASA-package* (2000), http://www.ingber.com.

[58] P. Jog, J.Y. Suh and D. Van Gucht, *Parallel Genetic Algorithms Applied to the Traveling Salesman Problem*, SIAM Journal of Optimization, 1 (1991), pp.515–529.

[59] D.S. Johnson *Local Optimization and the Traveling Salesman Problem*, Proceedings of the Seventh Colloquium on Automata, Languages, and Programming, LNCS 447 (1990), pp. 446–461

[60] D. S. Johnson, C. H. Papadimitriou and M. Yannakakis, *How Easy is Local Search?*, Journal of Computer and System Sciences, 17:1 (1988), pp. 79-100.

[61] M. Jones and P. Banerjee, *Performance of a Parallel Algorithm for Standard Cell Placement on the Intel Hypercube*, Proceedings of the 24th Design Automation Conference, 1987, pp. 807–813.

[62] S. Kirkpatrick, C.D. Gellat Jr. and M.P. Vecchi, *Optimization by Simulated Annealing*, Science, 220 (1983), pp. 671–680.

[63] G. Kliewer and S. Tschöke, *A General Parallel Simulated Annealing Library and its Application in Airline Industry*, Proceedings of the 14th International Parallel and Distributed Processing Symposium, IPDPS (2000), pp. 55–61.

[64] G. Kliewer and K. Klohs and S. Tschöke, *Parallel Simulated Annealing Library (parSA): User manual*, Technical report, University of Paderborn (1999), http://www.upb.de/ parsa.

[65] A. Knies and M. O'Keefe and T. MacDonald, *High Performance FORTRAN: A Practical Analysis*, Scientific Programming, Vol. 3 (1994), pp. 187–199.

[66] C. Koelbel and D. Loveman and R. Schreiber and G. Steele Jr. and M. Zosel, *The High Performance FORTRAN Handbook*, MIT Press (1994).

[67] V. Kumar and A. Grama and A. Gupta and G. Karypis, *Introduction to Parallel Computing Design and Analysis of Parallel Algorithms*, Benjaming/Cummings (1994).

[68] G. von Laszewski, *Intelligent Structural Operators for K-Way Graph Partitioning Problem*, Proceeding of the Fourth International Conference on Genetic Algorithms, (1991), Morgan Kaufmann, San Mateo, CA.

[69] C.Y. Lee and S.J. Kim, *Parallel Genetic Algorithms for the Earliness-Tardiness Job Scheduling Problem with General Penalty Weights*, Computers & Ind. Eng., 28 (1995), pp. 231–243.

[70] F. Lee *Parallel Simulated Annealing on a Message-Passing Multi-Computer*, Ph.D. thesis, Utah State University (1995).

[71] W.G. Macready and A.G. Siapas and S.A. Kauffman, *Criticality and Parallelism in Combinatorial Optimization*, SCIENCE, Vol. 271 (1996), pp. 56–59.

[72] A. Mahanti and C.J. Daniels, *A SIMD Approach to Parallel Heuristic Search*, Artificial Intelligence, 10 (1993), pp. 243–282.

[73] T.A. Marsland and F. Popowich, *Parallel Game-Tree Search*, IEEE Transactions on Pattern Analysis and Machine Intelligence, Vol. PAMI-7 (1985), pp. 442–452.

[74] O. Martin and S.W. Otto and E.W. Felten, *Large-Steps Markov Chains for the Travelling Salesman Problem*, Complex Systems, Vol. 5 (1991), pp. 299–326.

[75] S.L. Martins, *Parallelization strategies for metaheuristics in distributed memory environments*, Ph.D. thesis, Department of Computer Sciences, Catholic University of Rio de Janeiro, Rio de Janeiro, Brazil, 1999.

[76] S.L. Martins and C.C. Ribeiro, *A parallel GRASP for the Steiner problem in graphs.* In A. Ferreira and J. Rolim (Editors), *Proceedings of IRREGULAR'98 – 5th International Symposium on Solving Irregularly Structured Problems in Parallel*, Lecture Notes in Computer Science, Springer-Verlag, 1457 (1998) pp. 285–297.

[77] S.L. Martins and C.C. Ribeiro and N. Rodriguez, *Parallel Computing Environments.* In *Handbook of Applied Optimization*, P.M. Pardalos and M.G.C. Resende (Editors), Oxford University Press, 2001.

[78] J. Merlin and A. Hey, *An Introduction to High Performance FORTRAN*, Scientific Programming, Vol. 4 (1995), pp. 87–113.

[79] N. Metropolis and A. Rosenbluth and M. Rosenbluth and A. Teller and E. Teller, *Equation of State Calculations by Fast Computing Machines*, Journal of Chemical Physics, 21 (1953), pp. 1087–1092.

[80] A. Migdalas and P.M. Pardalos (Eds.), *Parallel Computing in Optimization*, Kluwer Academic Publishers, 1997.

[81] B.P. Miller and M.D. Callagham and J.M. Cargille and J.K. Hollingsworth and R.B. Irvin and K.L. Karavanic and K. Kunchithapadam and T. Newhall, *The Paradyn Parallel Performance Tools*, IEEE Computer, Vol. 28 (1995), pp. 37–46.

[82] Practical Parallel Computing, AP Professional (1994).

[83] MPI Forum, *A Message-Passing Interface Standard*, The International Journal of Supercomputing Applications and High Performance Computing, Vol. 8, (1994), pp: 138–161.

[84] MPI Forum, *A Message-Passing Interface Standard*, (1995),

[85] H. Muhlenbein, *Parallel Genetic Algorithms, Population Genetics and Combinatorial Optimization*, `Lecture Notes in Computer Science`, Vol. 565 (1989), pp. 398–406.

[86] H. Muhlenbein, M. Schomisch and J. Born, *The Parallel Genetic Algorithm as Function Optimizer*, Proceedings on an International Conference on Genetic Algorithms, (1991).

[87] R.A. Murphey, P.M. Pardalos, and L.S. Pitsoulis, *A parallel GRASP for the data association multidimensional assignment problem*, In P.M. Pardalos (Editor), *Parallel processing of discrete problems*, volume 106 of *The IMA Volumes in Mathematics and Its Applications*, pages 159–180. Springer-Verlag, 1998.

[88] W. Nagel and A. Arnold and M. Weber and H. Hoppe and K. Solchenbach, *VAMPIR: Visualization and Analysis of MPI Resources*, Supercomputer, Vol. 63 (1996), pp. 69–80.

[89] L.J. Osborne and B.E. Gillett, *A Comparison of two Simulated Annealing Algorithms Applied to the Directed Steiner Problem in Networks*, ORSA J. on Computing, Vol. 3 (1991), pp. 213–225.

[90] P. M. Pardalos (Eds), *Parallel Processing of Discrete Problems*, The IMA Volumes in Mathematics and its Applications, Vol. 106 (1999).

[91] P. M. Pardalos, Y. Li and K. A. Murthy, *Computational Experience with Parallel Algorithms for Solving the Quadratic Assignment Problem*, In *Computer Science and Operations Research:*

New Developments in their Interface, O. Balci, R. Sharda, S.A. Zenios (eds.), Pergamon Press, pp. 267–278 (1992).

[92] P. M. Pardalos, and G. Guisewite, *Parallel Computing in Nonconvex Programming*, `Annals of Operations Research` 43 (1993), pp. 87–107.

[93] P. M. Pardalos, A. T. Phillips and J. B. Rosen, *Topics in Parallel Computing in Mathematical Programming*, Science Press, 1993.

[94] P. M. Pardalos, L. S. Pitsoulis, T. Mavridou, and M.G.C. Resende, *Parallel search for combinatorial optimization: Genetic algorithms, simulated annealing and GRASP*, In *Parallel Algorithms for Irregularly Structured Problems, Proceedings of the Second International Workshop –Irregular'95*, A. Ferreira and J. Rolim (Editors), *Lecture Notes in Computer Science*, Springer-Verlag, vol. 980 (1995) pp. 317–331.

[95] P. M. Pardalos, L. S. Pitsoulis and M.G.C. Resende, *A Parallel GRASP Implementation for the Quadratic Assignment Problem*, In Proceedings of *Parallel Algorithms for Irregularly Structured Problems (Irregular '94)*, Kluwer Academic Publishers (1995).

[96] P.M. Pardalos, L.S. Pitsoulis, and M.G.C. Resende, *A parallel GRASP for MAX-SAT problems*, Lecture Notes in Computer Science, 1180 (1996) 575–585.

[97] P.M. Pardalos, M.G.C. Resende, and K.G. Ramakrishnan (Editors), *Parallel Processing of Discrete Optimization Problems*, DIMACS Series Vol. 22, American Mathematical Society, (1995).

[98] P. M. Pardalos, M. G. C. Resende (Editors), *Handbook of Applied Optimization*, Oxford University Press, (2000).

[99] P.M. Pardalos and H. Wolkowicz (Editors), *Quadratic Assignment and Related Problems*, DIMACS Series Vol. 16, American Mathematical Society (1994).

[100] C. Peterson, *Parallel Distributed Approaches to Combinatorial Optimization: Benchmark Studies on Traveling Salesman Problem*, Neural Computation, Vol. 2, (1990), pp. 261–269.

[101] C.B. Pettey, M.R. Leuze and J.J. Grefenstette, *A Parallel Genetic Algorithm*, Proceedings of the Second International Con-

ference on Genetic Algorithms and their Applications, J.J. Grefenstette (editor), July 1987, pp. 155–161.

[102] C. Powler, C. Ferguson and R. E. Korf, *Parallel Heuristic Search: Two Approaches*, In *Parallel Algorithms for Machine Intelligence and Vision*, V. Kumar, P.S. Gopalakrishnan and L.N. (eds.), Springer-Verlag, pp. 42–65 (1990).

[103] M.G.C. Resende, *Computing approximate solutions of the maximum covering problem using GRASP*, J. of Heuristics, 4 (1998) pp. 161–171.

[104] M.G.C. Resende, T.A. Feo, and S.H. Smith, *Algorithm 787: Fortran subroutines for approximate solution of maximum independent set problems using GRASP*, ACM Trans. Math. Software, 24 (1998) pp. 386–394.

[105] D.A. Reed and R.A. Aydt and R.J. Noe and P.C. Roth and K.A. Shields and B. Schwartz and L.F. Tavera, *Scalable Performance Analysis: The Pablo Performance Analysis Environment*, Proceedings of the Scalable Parallel Libraries Conference, IEEE Computer Society (1993), pp. 104–113.

[106] J.S. Rose, W.M. Snelgrove and Z.G. Vranesic, *Parallel Standard Cell Placement Algorithms with Quality Equivalent to Simulated Annealing*, IEEE Transactions on Computer-Aided Design, Vol. 7 (1988), pp. 387–396.

[107] A.V. Sannier and E.D. Goodman, *Genetic Learning Procedures in Distributed Environments*, Proceedings of the Second International Conference on Genetic Algorithms and their Applications, J.J. Grefenstette (editor), July 1987, pp. 162–169.

[108] J.S. Sargent, *A Parallel Row-Based Algorithm with Error Control for Standard-Cell Placement on a Hypercube Multiprocessor*, Thesis, University of Illinois, Urbana-Illinois, 1988.

[109] Scientific Computing Associates, *Linda User's Guide and Reference Manual*, Version 3.0 (1995).

[110] B. Shirazi, M. Wang and G. Pathak, *Analysis and Evaluation of Heuristic Methods for Static Task Scheduling*, Journal of Parallel and Distributed Computing 10 (1990), pp. 222–232.

[111] P.S. de Souza, *Asynchronous Organizations for Multi-Algorithm Problems*, Ph.D. Thesis, Department of Electrical and Computer Engineering, Carnegie Mellon University, 1993.

[112] R. Shonkwiler and E.V. Vleck, *Parallel Speed-Up of Monte Carlo methods for Global Optimization*, Journal of Complexity 10 (1994), pp. 64–95.

[113] J. Suh and D. Van Gucht, *Distributed genetic Algorithms*, Tech. Report 225, Computer Science Department, Indiana University, Bloomington, IN, July 1987.

[114] E. Taillard, *Robust Taboo Search for the Quadratic Assignment Problem*, Parallel Computing, 17 (1991), pp. 443–445.

[115] R. Tanese, *Parallel Genetic Algorithm for a Hypercube*, Proceedings of the Second International Conference on Genetic Algorithms and their Applications, J.J. Grefenstette (editor), July 1987, pp. 177–183.

[116] H.M.M. Ten Eikelder and M.G.A. Verhoeven and T.W.M. Vossen and E.H.L. Aarts, *A Probabilistic Analysis of Local Search*, In I.H. Osman and J.P. Kelly (Eds.) *Metaheuristics: Theory & Applications*, pp. 605–618, Kluwer Academic Publishers (1996).

[117] N.L.J. Ulder, E.H.L. Aarts, H. -J. Bandelt, P.J.M. van Laarhoven and E. Pesch, *Genetic Local Search Algorithms for the Traveling Salesman Problem*, In *Lecture Notes in Computer Science*, Parallel Problem Solving from Nature-Proceedings of 1st Workshop, PPSN 1, Vol. 496 (1991), pp. 109–116.

[118] sc M.G.A. Verhoeven and E.H.L. Aarts *Parallel Local Search*, Journal of Heuristics, Vol. 1 (1995), pp. 43–65.

[119] C.-P. Wong and R.-D. Fiebrich, *Simulated Annealing-Based Circuit Placement on the Connection Machine System*, Proceedings of International Conference on Computer Design (ICCD '87), 1987, pp. 78–82.

[120] J. Yan and S. Sarukhai and P. Mehra, *Performance Measurement, Visualization and Modeling of Parallel and Distributed Programs Using the AIMS Toolkit*, *Software Practice and Experience*, Vol. 25 (1995), pp. 429–461.

[121] J. Yan and S. Sarukhai, *Analyzing Parallel Program Performance Using Normalized Performance Indices and Trace Transformation Techniques*, Parallel Computing, Vol. 22 (1996), pp. 1215–1237.