Chapter 1

EXPLORING GRID IMPLEMENTATIONS OF PARALLEL COOPERATIVE METAHEURISTICS

A Case Study for the Mirrored Traveling Tournament Problem

Aletéia P.F. Araújo[1,3], Cristina Boeres[2], Vinod E.F. Rebello[2], Celso C. Ribeiro[1,2], and Sebastián Urrutia[1]

[1]*Department of Computer Science, Catholic University of Rio de Janeiro, Rio de Janeiro, Brazil*
{aleteia, celso, useba}@inf.puc-rio.br

[2]*Instituto de Computação, Universidade Federal Fluminense, Niterói, Brazil*
{boeres, vinod, celso}@ic.uff.br

[3]*Department of Computer Science, Catholic University of Brasília, Brasília, Brazil*

Abstract Metaheuristics are general high-level procedures that coordinate simple heuristics and rules to find good approximate solutions to computationally difficult combinatorial optimization problems. Parallel implementations of metaheuristics appear quite naturally as an effective approach to speedup the search for approximate solutions. Besides the accelerations obtained, parallelization also allows solving larger problems or finding better solutions. We present in this work four slightly differing strategies for the parallelization of an extended GRASP with ILS heuristic for the mirrored traveling tournament problem, with the objective of harnessing the benefits of grid computing. Computational experiments on a dedicated cluster illustrate the effectiveness and the scalability of the proposed strategies. In particular, we show that the parallel strategy implementing cooperation through a pool of elite solutions scales better than the others and is able to find solutions that cannot be reached by the others. Computational grids are distributed high latency environments which offer significantly more computing power than traditional clusters. The best parallel strategy was also implemented and tested using a true grid platform. We report original results from pioneer computational experiments on a shared computational grid formed by 82 machines distributed over four clusters in three cities, illustrating the potential of the application of computational grids in the fields of metaheuristics and combinatorial optimization.

Keywords: Parallel metaheuristics, grid computing, computational grids, traveling tournament problem, GRASP, iterated local search

1. Introduction

The organization and management of sporting events and championships is a worldwide multi-billion dollar industry. Schedules with minimum traveling times and offering similar costs and conditions to all teams taking part in a competition are of major interest to teams, leagues, sponsors, fans, and the media. In the case of the Brazilian national soccer championship, a single trip from Porto Alegre to Belém takes almost a full day's journey, with numerous connections due to the absence of direct flights, to cover a distance of approximately 4,000 kilometers. The total distance traveled becomes a key issue to be minimized, so as to reduce costs and to give the players more time to train and time off along the season that lasts for approximately eight months.

Several authors in different contexts (see e.g. [4, 5, 8, 20, 27, 35–37, 40, 41]) have tackled the problem of tournament scheduling for a variety of leagues and sports including soccer, basketball, hockey, baseball, rugby and cricket, using different techniques such as integer programming, tabu search, genetic algorithms, simulated annealing, and constraint programming.

The Traveling Tournament Problem is an inter-mural championship timetabling problem that abstracts certain characteristics of scheduling problems in sports [11]. It combines tight feasibility constraints with a difficult objective function to be optimized. The objective is to minimize the total distance traveled by the teams, subject to the constraint that no team can play more than three consecutive games at home or away. Since the total distance traveled is a major issue for every team taking part in the tournament, solving a traveling tournament problem may be a starting point for the solution of real timetabling applications in sports, in general.

Metaheuristics are general high-level procedures that coordinate simple heuristics and rules to find good approximate (often optimal) solutions to computationally difficult combinatorial optimization problems. Among them, we find simulated annealing, tabu search, Greedy Randomized Adaptive Search Procedure (GRASP), genetic algorithms, scatter search, Variable Neighborhood Search, ant colonies, and others. They are based on distinct paradigms and offer different mechanisms to escape from locally optimal solutions. Metaheuristics are among the most effective strategies for solving hard combinatorial optimization problems. The customization (or instantiation) of a metaheuristic to a given problem yields a heuristic for that problem.

Recent years have witnessed huge advances in computer technology and communication networks. Cung et al. [9] noted that parallel implementations of metaheuristics not only appear as quite natural alternatives to speed up the search for good approximate solutions, but also facilitate solving larger problems and finding improved solutions, with respect to their sequential counterparts, due to the partitioning of the search space and to the increased possibilities for search intensification and diversification. As a consequence, parallelism can improve the effectiveness and robustness of metaheuristic-based algorithms. The latter are less dependent on sophisticated parameter tuning and their success is not limited to a few or small classes of problems.

The growing computational power requirements of large scale applications and the high costs of developing and maintaining supercomputers has fuelled the drive for cheaper high performance computing environments. With the considerable increase in commodity computers and network performance, cluster computing and, more recently, grid computing [15, 16] have emerged as a real alternatives to traditional supercomputing environments for executing parallel applications that require significant amounts of computing power.

A computing cluster generally consists of a fixed number of homogeneous resources, interconnected on a single administrative network, which together execute one parallel application at a time. Grids in some sense are just the opposite, aiming to harness sufficient computing power from a diverse pool of resources, available on the internet, to execute a number of applications simultaneously. Grids aggregate geographically distributed collections (or sites) of resources which typically belong to different owners and thus are shared between multiple users. Each of these sites could consist of one or more uni-processor machines, a symmetric multiprocessor cluster, a distributed memory multicomputer system, or a massively parallel supercomputer. Clearly, the physical nature of the resources and the computing power available are both heterogeneous. Unlike local area network environments, grids are more susceptible to resource and network failures. Additionally, since the resources and network are being shared, the computational power available and communication costs fluctuate. These issues require careful consideration when developing grid enabled applications. The fact that these resources are distributed, heterogeneous and non-dedicated, make writing parallel grid-aware applications much more challenging [14]. While in theory optimization problems should easily benefit from grid computing, in practice appropriate design, careful tuning and thorough reevaluation of parallel implementations are necessary. Most of all, this requires a thorough understanding of how metaheuristics behave in such environments.

This work aims to investigate the practical benefits that large scale parallel processing can bring to metaheuristics for combinatorial optimization problems. In particular, this paper describes four simple but efficient strategies for the parallelization in grid environments to improve the extended GRASP with ILS heuristic for the mirrored traveling tournament problem proposed in [34]. The sequential strategy substitutes the local search phase of a GRASP heuristic by an ILS procedure, obtaining high-quality solutions that are among the best known in the literature for benchmark instances of this problem [38].

The remainder of the paper is organized as follows. The following section reviews the formulation of the mirrored traveling tournament problem. Section 3 summarizes the extended GRASP with ILS sequential heuristic. In Section 4, some important issues concerning the parallel implementation of metaheuristics are reviewed. Section 5 describes the four parallel implementations for the mirrored traveling tournament problem. Section 6 presents and compares experimental results obtained with the proposed strategies. Results on a computational grid employing 82 resources from sites in three different cities are reported in Section 7. Concluding remarks are made in the last section.

2. The mirrored traveling tournament problem

We consider a tournament played by n teams, where n is an even umber. In a *simple round-robin* (SRR) tournament, each team plays every other exactly once in $n-1$ prescheduled rounds. In a *double round-robin* (DRR) tournament, each team plays every other twice, once at home and once away. A *mirrored double round-robin* (MDRR) tournament is a simple round-robin tournament in the first $n-1$ rounds, followed by the same tournament with reversed venues in the last $n-1$ rounds. We assume that each team in the tournament has a stadium in its home city and that the distances between the home cities are known. Each team is located at its home city at the beginning of the tournament, to where it returns at the end after playing the last away game. Whenever a team plays two consecutive away games, it goes directly from the city of the first opponent to the other, without returning to its own home city.

The Traveling Tournament Problem (TTP) was first established by Easton et al. [11]. Given n teams and the distances between their home cities, the TTP consists in finding a DRR tournament such that every team does not play more than three consecutive home or away games, no repeaters (i.e., two consecutive games between the same two teams at different venues) occur, and the sum of the distances traveled by the teams is minimized. Benchmark instances are available in [38]. To date, even small benchmark instances of the TTP with $n = 10$ teams cannot be solved exactly. The largest instance for which the optimal solution is known ($n = 8$ teams) took four days of processing time using twenty processors in parallel [10]. We also refer to this problem as the non-mirrored TTP, for which both mirrored and non-mirrored solutions are feasible.

The mirrored Traveling Tournament Problem (mTTP) has an additional constraint: the games played in round k are exactly the same played in round $k + (n - 1)$ for $k = 1, \dots, n - 1$, with reversed venues. Repeaters do not occur in mirrored schedules. Mirrored tournaments are a common tournament structure in Latin America.

The TTP has raised significant interest in recent years especially after challenge instances were proposed in [38]. Easton, Nemhauser and Trick [11] applied an Integer Linear Programming approach to the TTP. They modified their three-phase approach [27] previously used to schedule the Atlantic Coast Conference (ACC) basketball league, generating new high-quality solutions. Some of the results were later improved upon by Benoist et al. through the combination of Lagrangian relaxation and constraint programming [6]. Anagnostopoulos et al. [4] proposed a simulated annealing algorithm for the TTP to explore a large neighborhood with complex moves. The algorithm included a strategic oscillation technique and applied re-heat to both balance the exploration of feasible and infeasible regions and to escape from local minima at very low temperatures. Lim et al. [22] used a two-stage approach consisting of simulated annealing and hill-climbing techniques were able to improve even further some of the results obtained by [4]. Recently, Rasmussen and Trick [28] also considered Benders decomposition approaches to the TTP. Gaspero and Schaerf [17] explored a composite-neighborhood tabu search approach. It is important to point out that these works provide solutions for the non-mirrored Traveling Tournament Problem, while our work focus on the mirrored version. As far as we know, there is no parallel metaheuristic solutions for the mTTP (nor TTP), which is the focus of the work proposed here.

3. Extended GRASP with ILS heuristic

The GRASP (Greedy Randomized Adaptive Search Procedure) metaheuristic [29] is a multi-start or iterative process, in which each iteration consists of two phases: construction and local search. The construction phase builds a feasible solution, whose neighborhood is investigated during the local search phase until a local minimum is found. The best overall solution is kept as the result.

The construction and local search phases are problem-dependent and should be customized for each problem. GRASP has experienced continued development and has been applied in a wide range of areas [13]. Resende and Ribeiro [29, 30] described successful implementation techniques and parameter tuning strategies, as well as enhancements, extensions, and hybridizations of the original algorithms.

The ILS (Iterated Local Search) metaheuristic [23] starts from a locally optimal feasible solution. A random perturbation is applied to the current solution, which is then followed by a local search. If the local optimum obtained after these steps satisfies some acceptance criterion, then it is accepted as the new current solution, otherwise the latter does not change. The best solution is, if necessary, updated and the above steps are repeated until some stopping criterion is met.

A hybridization of the GRASP and ILS metaheuristics into an effective hybrid heuristic for the mTTP was proposed in [34]. Basically, the authors substituted the local search phase of GRASP by an ILS procedure. The pseudo-code in Algorithm 1.1 summarizes the main steps of the `GRILS-mTTP` heuristic for finding approximate solutions for the mirrored traveling tournament problem.

The outer **while** loop in Algorithm 1.1 executes a GRASP *construction phase* followed by an ILS *local search phase*, until a stopping criterion is met. Typically, the algorithm continues executing until a solution is found with a cost that is as good as or better than a given *target* value, or until a given period of time has elapsed.

During the GRASP phase of each iteration, an initial solution S is constructed to which a local search algorithm is then applied, returning a new current solution S. This solution is also used to initialize the best solution $\underline{S}$ in the current iteration.

The ILS phase of the iteration is the inner **repeat** loop which applies a perturbation to the current solution S obtaining a new solution S'. A local search algorithm is applied to S', where four neighborhood

Procedure `GRILS-mTTP`();
while `.NOT.`*StoppingCriterion* **do**
    $S \leftarrow$ `BuildGreedyRandomizedSolution`();
    $\underline{S}, S \leftarrow$ `LocalSearch`(S);
    repeat
        $S' \leftarrow$ `Perturbation`(S);
        $S' \leftarrow$ `LocalSearch`(S');
        $S \leftarrow$ `AcceptanceCriterion`(S, S');
        $S^* \leftarrow$ `UpdateGlobalBestSolution`(S, S^*);
        $\underline{S} \leftarrow$ `UpdateIterationBestSolution`($S, \underline{S}$);
    until *ReinitializationCriterion*;
end;
return S^*;

Figure 1.1. Pseudo-code of the GRASP with ILS heuristic for the mTTP.

structures are used. The first three are simple exchanges in which TS (team swap), HAS (home-away swap) and PRS (partial round swap) neighborhoods are explored by local searches. The GR (game rotation) ejection chain neighborhood, explored only as a diversification move, is performed less frequently by the heuristic as a perturbation [34].

A first-improving strategy similar to the VND (Variable Neighborhood Descent) procedure [19] was used to implement the local search algorithm. Once a local optimum with respect to the TS neighborhood is found, a quick local search using the HAS neighborhood is performed. Next, the PRS neighborhood is investigated, followed again by a local search using the HAS neighborhood. This scheme is repeated until a local optimum with respect to these three neighborhoods is found.

In this context, the new solution S' is accepted or not as the new current solution, depending on an acceptance criterion. The best overall solution S^* and the best solution in the current GRASP iteration are updated, if necessary, and a new cycle starts with the perturbation of the current solution, until a re-initialization criterion is met.

A new GRASP iteration starts if 50 consecutive deteriorating moves to neighbor solutions have been accepted since the last time $\underline{S}$ (the best solution found in this GRASP iteration) was updated. Re-initialization occurs if too many perturbations followed by local search are performed without improving the best solution in the current GRASP iteration. It is important to notice that a GRASP iteration is not interrupted if the current solution S is still being improved.

The parallelization of this algorithm does not only aim to reduce the total running time, but also to improve its effectiveness and robustness. The use of several processors concurrently to explore different search trajectories, as described in Section 5, may lead to a more thorough investigation of the neighborhoods.

4. Parallel implementation of metaheuristics

Programming paradigms commonly used to develop low communication parallel programs on distributed clusters include the *master-slave* (also often referred to as task farming) and the *client-server* models [14]. These approaches are especially attractive, since they can generally be applied to take advantage of all available resources in a grid environment.

Cung et al. [9] reviewed some major issues on parallel implementations of metaheuristics, such as the types of parallelism as well as appropriate parallel programming models and parallelization strategies for this class of heuristics. With respect to parallelization strategies [9, 39], two main approaches are used: single-walk and multiple-walk. Each iteration of a metaheuristic generally starts with the construction of an initial solution, followed by a search to improve the solution. New neighboring solutions are evaluated by making a series of minor alterations to a given solution. The sequence of solutions evaluated is known as a *walk* or *trajectory*. In the case of a single-walk parallelization, one unique search trajectory is traversed in the solution space and the search for the best neighbor at each iteration is performed in parallel. The neighborhood search is performed faster in parallel, but the search trajectory is the same as the one followed in the corresponding sequential implementation. On the other hand, a multiple-walk parallelization strategy is characterized by the investigation in parallel of multiple trajectories, each of them performed by a different processor. A search "thread" is a process running in each processor traversing a walk in the solution space. These processes can be either independent (where no information is exchanged among processes) or cooperative (the information collected along a trajectory is disseminated and used by other processes to improve or to speed up the search).

Cooperative strategies are the most general and promising, but often incur in additional costs in terms of communication and storage. However, if cooperation is well explored and implemented, it can globally lead to better solutions in smaller computation times even if each individual iteration may take longer, see e.g. [32].

Developing and tuning efficient parallel implementations of metaheuristics require a thorough programming effort and keen implementation skills. In the context of grid computing, where communication is at a premium, one of the most difficult aspects to be determined is the nature of the information to be shared, in order to improve the search without taking too much additional memory or time to be collected, as well as the frequency at which this information is exchanged. The information shared by the search threads can be implemented either as global variables stored in a shared memory, or as a pool in the local memory of a dedicated central processor. In the case of the latter, information is exchanged with the other processors via message passing.

5. Parallel strategies for the extended GRASP with ILS heuristic

This section presents four simple, but efficient, strategies for the parallelization of the best known algorithm (the hybrid heuristic `GRILS-mTTP` [34] summarized in Section 3) for solving the mTTP. Besides obtaining speedups in execution times, improvements in solution quality are also sought. All four versions are based on the master-worker programming paradigm and adopt a multiple-walk search strategy. This work aims to investigate how degrees of cooperation and increased diversity (in terms of number of trajectories investigated and the amount of information being shared) affect the `GRILS-mTTP` heuristic.

Initially, the master process generates and distributes distinct seeds to be used by the pseudo-random number generator of each worker process. As the number of workers increases, this will foster greater diversity. In order to reduce the chance that processes search the same neighborhood (i.e., evaluate the same solutions), each process uses a different sequence of pseudo-random numbers. The Mersenne Twister random number generator of Matsumoto and Nishimura [24] was chosen based on the recommendation in [33].

5.1 Parallel strategy with independent processes

This version, denoted by *PAR-I*, is representative of executing the sequential algorithm simultaneously on multiple machines independently of each other (e.g. as a parameter sweep application).

After receiving their seeds, each worker starts a cycle in which it generates a new solution during a GRASP construction phase and then executes an ILS local search phase until the re-initialization criterion is met. This cycle is repeated until a solution with a cost equal to or better than a given target value (used as a stopping criterion) is found. Although no communication takes place between the independent searches, once the stopping criterion has been met, a controller process (master) receives and records the solution found and responds by broadcasting a halt message to each worker to terminate their execution.

5.2 Parallel strategy with one-off cooperation

This version, *PAR-O*, is identical to the previous one, with the exception of the first iteration of the main loop. After each worker executes the GRASP phase, the best initial solution encountered by each of them is sent to the master, which in turn selects and broadcasts back to all the workers the best overall solution. Therefore, all workers will execute the ILS local search phase of the first iteration using the same initial solution. The following iterations are executed independently. This is called one-off cooperation because this exchange only occurs during the first iteration.

5.3 Parallel strategy with one elite solution

One of the possible shortcomings of the previous versions is the lack of continuous cooperation between the workers during their execution, i.e., each worker process does not learn from searches carried out in parallel (or from solutions found) in previous iterations by other workers. In the earlier strategies, the current best solution is not available to all workers. Information gathered from good solutions should be used to implement more effective strategies [12, 31]. Typically, in these history-based parallel cooperative strategies, the master manages the exchange of information collected along the trajectories investigated by each worker.

In this version, PAR-1P, the master keeps the best (or elite) solution received from any worker. Each time the best solution is improved, the master broadcasts the solution's cost to all workers to avoid unnecessary communication from them. The intuition is to use this information not only to converge faster to a target solution, but also to find better solutions than the independent search strategies.

In PAR-1P, there is no one-off cooperation during the first iteration. Instead, each time a worker completes the ILS local search phase, it will compare the cost of the solution found with that of the best solution held by the master. If it is lower, the worker sends its solution to the master, otherwise the solution is discarded. After this, two outcomes are possible. Either, the worker requests the best solution held by the master to repeat the ILS local search phase with this solution, or the worker continues with the next iteration (i.e. re-initialization causes a new initial solution during the GRASP construction phase to be created and proceeds with the next steps of the sequential heuristic) as in the previous versions.

The probability of each outcome is denoted by Q and $1 - Q$, respectively. Q was fixed at 10% in all experiments reported in Section 6. In this way, workers indirectly exchange elite solutions found along their search trajectories. This parallel cooperative strategy promotes a more thorough search of the space around good solutions, characteristic of single-walk parallelization approaches.

5.4 Parallel strategy with a pool of elite solutions

In this cooperative strategy, PAR-MP, the master is dedicated to managing a centralized pool of elite solutions (and their costs), including collecting and distributing them upon request. As in the previous version, workers start their searches from different initial solutions and can exchange and share elite solutions found along their search trajectories.

In PAR-MP, the master will update the elite solution pool with a newly received solution according to given criteria which are based on the quality of the solutions already in the pool (as described below). When a worker completes an iteration, it can either request an elite solution from the pool or construct a new initial solution randomly, again with probabilities of Q and $1 - Q$, respectively. Q was fixed at 10% in all experiments reported in Section 6.

The goal to be achieved by cooperation through the pool of elite solutions is to exchange meaningful information in a timely manner, so as that the parallel search finds better solutions than the simple concatenation of the results of the individual methods. Developments in metaheuristics have proved to be particularly successful when their basic concepts are combined with cooperative methods. These hybrid cooperative approaches maintain a reference set of high quality solutions which are repeatedly used during the search to guarantee a fruitful balance between diversification and intensification [18].

Pool management A very important aspect of this strategy is the management of the pool of elite solutions. However, maintaining such a pool is not trivial. The main issue consists in finding a balance between the attempt to collect a number of high quality solutions, which often share similar properties, and the need to guarantee a certain degree of diversity in the pool. Empirically, previous research (see e.g. [12])

observed that history-based heuristics are less likely to be successful if the recorded solutions are very similar.

The pool consists of a limited number M of positions, which are initialized with null solutions. The pool manager supports two essential operations: the insertion of a new solution into its appropriate position in the pool and the selection of a solution from the pool from which a worker will initiate a new search.

To guarantee the diversity within the pool, the insertion of a new solution depends on the state of the pool and on how the solution was generated. When the new candidate solution has been derived from an elite solution in the pool, the cost of the new solution must be better than the cost of the elite solution from which it was generated. If true, the new solution will obligatorily take the place of that elite solution. On the other hand, if the solution was derived from a solution produced by the GRASP construction phase, the solution can be inserted directly into any vacant position. In the case where the pool is full, the solution is inserted only if it is as good as the worst elite solution already in the pool (thus replacing the latter).

When a worker process requests an elite solution from the master, a solution is selected at random from the pool and sent back to worker process.

6. Experimental results

The four parallel algorithms PAR-I, PAR-O, PAR-1P, and PAR-MP, described in Section 5, were implemented using C++ and version 7.0.6 of the LAM grid-enabled implementation [21] of the message passing interface standard MPI [25]. For evaluation purposes, the experiments reported in this section were executed in isolation on a dedicated cluster (of 1.7 GHz Pentium 4 processors, each of which with 256 Mbytes of RAM memory, interconnected by a Fast Ethernet network) to avoid external influences in performance. Each processor has a local copy of the executable code and the problem data.

Two sets of benchmark instances have been proposed for the traveling tournament problem[11]. The first is made up of *circle instances*, artificially generated to represent easier instances. The name circ$n$ is used to denote a circle instance with $4 \leq n \leq 20$ teams. Each circle instance is built from a graph, generated as follows. Nodes are placed at equal unit distances along a circumference and labeled $0, 1, \ldots, n-1$. There are edges only between nodes i and $i + 1 \bmod n$, for $i = 0, 1, \ldots, n-1$. In the corresponding circle instance, the distance between the home cities of teams i and j (with $i > j$) is given by the length of the shortest path between them in the graph and is equal to the smaller of $i - j$ and $j - i + n$. The second set are realistic instances generated using the distances between the home cities of a subset of teams playing in the National League of the MLB (Major League Baseball) in the United States. These national league instances are denoted by nl$n$, with $4 \leq n \leq 16$. We did not consider the smaller instances with $n = 4$ and $n = 6$, for which optimal solutions have already been found. Furthermore, an additional real-life instance has been created by Ribeiro and Urrutia [34], named br24. This instance is made up of the home cities of the 24 teams playing in the first division of the 2003 edition of the Brazilian soccer championship. All instances and their best known solutions are available from [38].

The experiments aim to investigate how parallel computing can be used to harness cooperation and diversity, improving solution quality and convergence when executing the `GRILS-mTTP` heuristic in distributed computing environments. The parameter M was set to P, where P is the number of worker processes used in the parallel execution. The probability, Q, of choosing a solution from the pool was fixed at 10%.

Table 1.1 displays, for each instance, the cost of the best known solution at the time of writing obtained by the sequential implementation of the `GRILS-mTTP` heuristic after five days of processing time [38]. These are compared with the cost of the best solutions found during the following experiments by the four parallel implementations of the `GRILS-mTTP` heuristic. The execution time required varies with the number of processors used and these details are described in the following experiments. Notice that PAR-I, PAR-O, and PAR-1P found the same cost solutions for each of the benchmark instances, while the PAR-MP implementation was able to improve the best solution found by the three others in the case of three instances. The last column gives the relative improvement obtained by PAR-MP over the cost of the best known sequential solution.

In the experiments reported next, the costs of the best solutions found by the sequential heuristic, as reported in Table 1.1, are referred to as the **easy** targets. The costs of the solutions obtained by the PAR-I, PAR-O, and PAR-1P implementations are referred to as the **medium** targets, for the instances for which the best solution obtained by these versions improved the easy targets (instances circ10, circ16, circ18, nl16, and br24). The PAR-MP implementation further improved the best known solutions for three of these instances, and these best solution costs are referred to as the **hard** targets (for instances circ16, circ18, and nl16).

The scalability of the parallel strategies was evaluated to study the benefits of searching an increasing number of multiple trajectories. Ex-

Table 1.1. Solutions found by the sequential and parallel implementations.

Instance	Sequential	PAR-I/O/1P	PAR-MP	Improvement (%)
circ8	140	140	140	-
circ10	276	272	272	1.45
circ12	456	456	456	-
circ14	714	714	714	-
circ16	1004	984	978	2.59
circ18	1364	1308	1306	4.25
circ20	1882	1882	1882	-
nl8	41928	41928	41928	-
nl10	63832	63832	63832	-
nl12	120655	120655	120655	-
nl14	208086	208086	208086	-
nl16	285614	280174	279618	2.09
br24	506433	503158	503158	0.65

ecuting with more processes offers a greater diversity due to the use of multiple distinct initial seeds and solutions. Table 1.2 (resp. Table 1.3) shows the average execution times over five runs with different seeds for the sequential and the PAR-I and PAR-O (resp. PAR-1P and PAR-MP) parallel versions. These tables present the time in seconds required to find a solution whose cost is at least as good as the corresponding easy target, using one processor for the sequential implementation and eight, 16, and 24 processors for each parallel version.

The parallel versions converged faster than the sequential one for all instances. As the number of processors used increases, all parallel versions were able to find the corresponding easy targets faster, as reflected by the speedups presented in Tables 1.4 and 1.5. The speedups of the PAR-MP parallel version were greater than those of the other implementations. For example, the average speedups of the four algorithms on 24 processors were 12.51 for PAR-I, 12.62 for PAR-O, 13.19 for PAR-1P and 13.41 for PAR-MP.

The following experiment considers the computation times taken by the parallel versions to find solutions at least as good as the medium targets (exclusively for the instances for which the latter are smaller than the corresponding easy targets), addressing the benefits of exchanging information between the workers instead of letting them execute independently. The average processing times in seconds, based on five executions, on 24 processors are reported in Table 1.6. Results show that PAR-MP presents the smallest computation time in most cases. Note that this version makes use of a cooperative strategy based on a pool of

Table 1.2. Average computation times in seconds to find the easy targets on eight, 16, and 24 processors (PAR-I and PAR-O parallel versions).

	Sequential		PAR-I			PAR-O	
Instance	$P = 1$	$P = 8$	$P = 16$	$P = 24$	$P = 8$	$P = 16$	$P = 24$
circ8	0.87	0.27	0.15	0.14	0.26	0.15	0.14
circ10	197.64	26.32	18.32	15.98	26.31	18.32	15.97
circ12	4.48	1.33	1.11	0.55	1.32	1.11	0.54
circ14	3.46	0.87	0.82	0.73	0.86	0.73	0.71
circ16	413.73	56.54	33.93	24.78	56.58	33.97	24.77
circ18	175.14	43.19	30.06	13.62	43.24	29.99	13.62
circ20	800.94	177.28	82.99	43.47	176.67	83.05	43.47
nl8	0.79	0.23	0.08	0.07	0.22	0.08	0.06
nl10	453.54	113.22	39.03	19.52	113.22	39.03	19.51
nl12	22.13	4.23	1.83	1.34	4.23	1.83	1.33
nl14	33.23	5.32	4.80	4.34	5.32	4.80	4.35
nl16	1433.05	474.12	243.14	73.62	474.26	243.11	73.58
br24	156.32	40.87	33.83	29.08	40.54	33.70	28.97

Table 1.3. Average computation times in seconds to find the easy targets on eight, 16, and 24 processors (PAR-1P and PAR-MP parallel versions).

	Sequential		PAR-1P			PAR-MP	
Instance	$P = 1$	$P = 8$	$P = 16$	$P = 24$	$P = 8$	$P = 16$	$P = 24$
circ8	0.87	0.19	0.16	0.12	0.19	0.15	0.11
circ10	197.64	26.31	17.04	15.97	26.31	17.04	15.96
circ12	4.48	1.33	0.39	0.35	1.33	0.38	0.34
circ14	3.46	0.65	0.53	0.52	0.65	0.53	0.51
circ16	413.73	55.30	33.93	24.45	55.31	33.94	24.50
circ18	175.14	24.73	21.96	13.60	24.70	23.96	13.62
circ20	800.94	106.34	70.50	43.47	106.30	70.47	43.46
nl8	0.79	0.17	0.10	0.08	0.16	0.08	0.07
nl10	453.54	113.21	39.01	19.52	125.43	39.05	19.51
nl12	22.13	3.91	1.61	1.33	3.91	1.64	1.33
nl14	33.23	5.85	4.80	4.35	5.84	4.80	4.34
nl16	1433.05	323.33	236.36	73.59	322.27	223.61	73.54
br24	156.32	33.99	25.29	20.23	30.77	23.71	16.55

M elite solutions ($M = 24$). Although PAR-1P also shares information, it only records one elite solution. Therefore, the degree of diversity is smaller than in PAR-MP, possibly leading the workers to search the same region and, consequently, taking longer to converge to the target.

Results in Table 1.1 have shown that the PAR-MP implementation found better solutions than those obtained by the three other parallel

Table 1.4. Speedups on eight, 16, and 24 processors (PAR-I and PAR-O parallel versions).

	PAR-I			PAR-O		
Instance	$P=8$	$P=16$	$P=24$	$P=8$	$P=16$	$P=24$
circ8	3.23	5.69	5.87	3.28	5.70	6.21
circ10	7.50	10.79	12.37	7.51	10.79	12.38
circ12	3.37	4.02	8.21	3.39	4.03	8.21
circ14	3.99	4.21	4.74	4.00	4.72	4.74
circ16	7.32	12.20	16.70	7.31	12.18	16.71
circ18	4.06	5.83	12.87	4.05	5.84	12.86
circ20	4.52	9.65	18.42	4.53	9.64	18.43
nl8	3.51	9.36	11.15	3.61	9.38	12.34
nl10	4.00	11.62	23.24	4.01	11.62	23.25
nl12	5.23	12.09	16.56	5.23	12.12	16.63
nl14	6.25	6.93	7.65	6.24	6.93	7.64
nl16	3.02	5.89	19.47	3.02	5.89	19.48
br24	3.83	4.62	5.38	3.86	4.64	5.40
average	4.60	7.91	12.51	4.62	7.96	12.62

Table 1.5. Speedups on eight, 16, and 24 processors (PAR-1P and PAR-MP parallel versions).

	PAR-1P			PAR-MP		
Instance	$P=8$	$P=16$	$P=24$	$P=8$	$P=16$	$P=24$
circ8	4.42	5.40	7.23	4.57	5.65	7.51
circ10	7.51	11.60	12.37	7.51	11.60	12.39
circ12	3.36	11.23	12.67	3.35	11.58	13.07
circ14	5.30	6.50	6.69	5.29	6.54	6.78
circ16	7.48	12.19	16.92	7.48	12.19	16.89
circ18	7.08	7.98	12.87	7.09	7.31	12.86
circ20	7.53	11.36	18.42	7.53	11.37	18.43
nl8	4.80	7.24	9.58	4.99	9.90	9.95
nl10	4.01	11.63	23.24	3.62	11.62	23.25
nl12	5.66	13.75	16.59	5.67	13.50	16.64
nl14	5.68	6.92	7.64	5.68	6.92	7.66
nl16	4.43	6.06	19.47	4.45	6.40	19.49
br24	4.60	6.18	7.72	5.08	6.59	9.45
average	5.53	9.08	13.19	5.56	9.32	13.41

implementations for three instances. Table 1.7 summarizes the results obtained by PAR-MP and gives the average overall computation times (based on five executions), in seconds, required to find the new solutions using 24 processors and the relative improvement with respect to the best

Table 1.6. Computation times in seconds to find solutions at least as good as the medium targets.

Instance	Target	PAR-I	PAR-O	PAR-1P	PAR-MP
circ10	272	5,768.81	4,650.31	7,209.82	3,725.68
circ16	984	5,366.12	735.73	3,881.09	959.24
circ18	1308	9,323.88	8,565.20	13,972.76	10,620.86
nl16	280174	12,207.18	11,488.44	7,058.52	3,171.40
br24	503158	4,322.45	4,268.41	5,046.40	2,220.69

solutions found by the other parallel implementations (i.e. the medium targets). We notice that the solution obtained by PAR-MP for instance circ18 is also the best known solution for the corresponding instance of the non-mirrored version of the TTP. Nevertheless, PAR-MP still requires just under six hours on average to find the solution.

Table 1.7. New best solutions obtained by the parallel version PAR-MP.

Instance	New best solution cost	Improvement (%)	Time (s)
circ16	978	0.61	4,690.83
circ18	1306	0.15	20,883.81
nl16	279618	0.20	14,586.73

The following experiment addresses the robustness of the parallel implementations from another point of view. Compared to the times needed by version PAR-MP to find the hard targets, we investigate whether PAR-I, PAR-O, and PAR-1P can also manage the same feat. Given the time taken by PAR-MP to find the best known solutions reported in the Table 1.7, the other parallel versions PAR-I, PAR-O, and PAR-1P were allowed to run for approximately twice this time, again using 24 processors. The values of the best solutions found by each version for each instance are presented in Table 1.8. These results show that the other parallel implementations were not able to find solutions as good as those obtained by PAR-MP, even if significantly more processing time is given, illustrating the effectiveness of the cooperation scheme implemented in the latter.

We used *time-to-target solution value* plots [1, 2] for the measured computation times to further evaluate and compare the behavior of the four parallel versions running on different numbers of processors. This approach is based on plots showing empirical distributions of the random variable *time-to-target solution value*. To plot the empirical distribution,

Table 1.8. Solutions found by PAR-I, PAR-O, and PAR-1P when executed for twice the time taken by PAR-MP.

Instance	Time (s)	Target	PAR-I	PAR-O	PAR-1P
circ16	10,000	978	984	984	984
circ18	40,000	1306	1308	1308	1308
nl16	30,000	279618	280174	280174	280174

we first fix a given problem instance and a target solution value. Next, each algorithm is executed N times, recording the running time to find the first solution as least as good as the target value. For each algorithm, we associated with the i-th sorted running time t_i a probability $p_i = (i - \frac{1}{2})/N$ and plot the points $z_i = (t_i, p_i)$, for $i = 1, \ldots, N$.

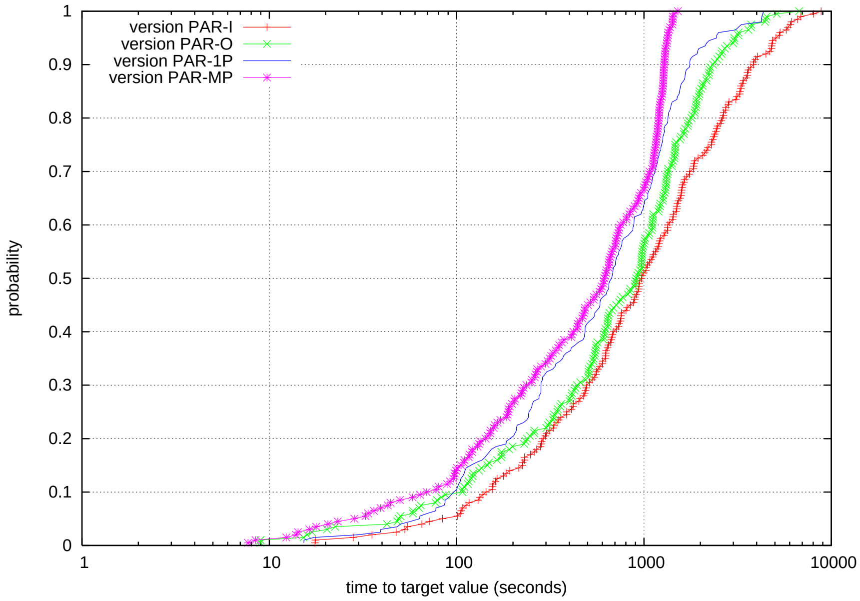

Figure 1.2. Empirical distributions of the random variable time-to-target solution value for the parallel versions PAR-I, PAR-O, PAR-1P, and PAR-MP using 8 processors on instance nl16.

Figure 1.2 displays the empirical distributions of the *time-to-target solution value* for the four parallel versions associated with the instance nl16 and the target cost of 284000 (a value between the easy and medium target), obtained from $N = 200$ independent runs of each version on $P = 8$ processors. Version PAR-MP behaves better than the other versions, finding the target value in less than 1,400 seconds with probability

95% compared to 2,445 seconds with the same probability for the next fastest amongst the others (PAR-1P). PAR-MP took at most approximately 1,500 seconds in the slowest run, while PAR-1P took more than 4,300 seconds in the worst run. The behavior depicted in this plot is common to different instances and target values. The plot of the empirical distribution associated with the PAR-MP strategy is clearly to the left of those of the other versions, illustrating that the former is more robust since it is able to find with higher probabilities the same solutions found by the others in the same computation time.

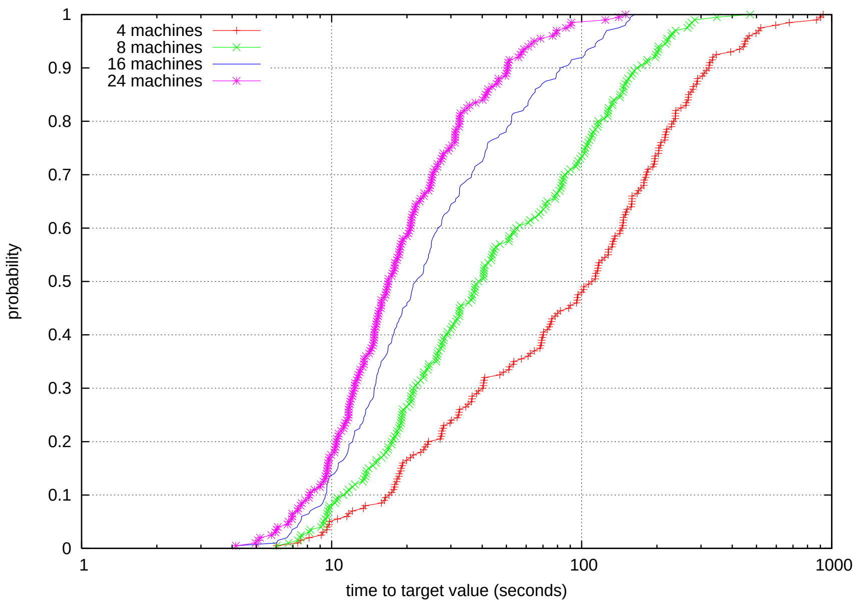

Figure 1.3. Empirical distributions of the random variable time-to-target solution value for the parallel version PAR-MP using 4, 8, 16, and 24 processors for the instance br24.

The plot in Figure 1.3 further illustrates the scalability of the parallel version PAR-MP, as already shown in Tables 1.2 and 1.3. This plot depicts the empirical distributions for four, eight, 16, and 24 processors, obtained from $N = 200$ runs on instance br24 using the easy target.

7. Grid implementation and experiments

Numerous scientists and engineers, from diverse scientific and technological fields, are showing interest in exploiting the potential of grid computing. This section briefly highlights a number of observations with respect to executing metaheuristics on a computational grid. For performance evaluation purposes, the experiments presented earlier in Section 6 were necessarily obtained with exclusive access to a dedicated cluster of processors, since grids are inherently shared computing environments. This sharing, together with the fact that grid resources are heterogeneous, means that the computing power available from resources is neither identical nor constant.

As computational environments scale to hundreds of individual computational resources, failures are more likely to occur especially during the execution of long-running applications. Users naturally want their programs to adapt to both faults and changes in available performance in order to continue executing efficiently. While the current implementations of MPI are suitable for use in the static environments like computing clusters, in practice they are not robust enough for computational grids. For example, grid enabled implementations of MPI do not provide support for dynamic rescheduling of processes. Furthermore, a single process failure will cause the whole application to abort.

In an effort to hide the intricacies of grid environments, grid engineers have been developing *grid middleware* (an intermediate layer of software) to provide tools which insulate users from the underlying complexities, and management systems, that automatically and efficiently adapt grid-enabled applications to the dynamically changing characteristics of the grid.

The EasyGrid AMS middleware [7] provides for a robust and efficient execution of programs in grid environments. Parallel MPI applications are transformed *automatically* into *system-aware* versions by incorporating grid middleware into the user's application without modification to the latter. These system-aware applications or *Smart G-Apps* are adaptive, robust to resource failure (fault tolerant), self-scheduling programs capable of reacting to changes which occur in shared, dynamic, unstable distributed environments like computational grids.

This approach aims to relieve programmers and users of the task of enabling (existing) applications to execute efficiently in grid environments. Turning aspects related to the grid transparent to the programmer avoids the need to develop one version of the application for a local cluster computing platform and another for the grid. The methodology is based on application-centric middleware which provides services (e.g. static and dynamic scheduling, and integrated fault tolerance strategies) specifically tuned to the needs of each individual application [26].

Given that the PAR-MP implementation produced the best results for the mTTP when compared to the other parallel strategies, a grid enabled version of PAR-MP based on the EasyGrid AMS was evaluated in a real grid environment. *Grid Sinergia*computational grid is an initiative to create and operate a research oriented, production level computational grid across three states (Rio de Janeiro, São Paulo, and Espírito Santo) in the south east of Brazil. The objectives include providing researchers with a realistic and practical environment for distributed computing research and offering system administrators practical experience in management and operation of grid computing environments. Grid Sinergia currently employs the Globus Toolkit middleware [3] across the participating sites which are interconnected by the Brazilian National Research Network's experimental high speed (10Gbit) optical network *Rede Giga*.

An initial experiment was carried out employing 82 resources from the following three sites of Grid Sinergia, located in three different cities within the state of Rio de Janeiro: (a) two clusters in the city of Rio de Janeiro, one with 30 Linux PCs (Pentium II 400 MHz) and the other with 24 Linux PCs (Pentium IV 1.7GHz), each of them connected by a fast ethernet network; (b) in Niterói (a distance of 40 Km from cluster (a)), a cluster of 26 Linux PCs (Pentium IV 2.6 GHz) interconnected via Gigabit switches; (c) and in Petrópolis (approximate 100 Km from both clusters (a) and (b)) two Linux PCs (Pentium IV 3.2 GHz).

Table 1.9 displays the average processing times (measured over five runs) in seconds required to find a solution whose cost is at least as good as the corresponding medium target, on the shared resources of the computational grid during the day (i.e. normal working hours) and during the night (after working hours, the resources tend to be utilized less). We notice that, although the resources were being shared with other users, the practical benefits obtained with the computational grid were outstanding. For example, in the case of instance circ18, for the medium target, an almost four fold improvement was achieved by the grid with respect to the dedicated 24-processor cluster.

Table 1.9. Average processing times of the parallel cooperative strategy PAR-MP when executed in 82 resources from three sites of Grid Sinergia.

	Time (seconds)		
Instance	Dedicated cluster (24 CPUs)	Shared grid (82 CPUs, working hours)	Shared grid (82 CPUs, after hours)
circ10	3,725.68	1,077.38	333.48
circ16	959.24	747.94	513.63
circ18	10,620.86	2,376.76	2,313.73
data16	3,171.40	1,044.31	908.95

8. Concluding remarks

Metaheuristics have found their way into the standard toolkit of combinatorial optimization methods. Parallel implementations of metaheuristics can be applied to hard combinatorial optimization problems, often allowing reductions in computation times. Although independent strategies can already obtain good computational results, parallelizations based on cooperative search strategies lead to more robust implementations.

The computational results reported in this paper show that the sequential heuristic for the mirrored traveling tournament problem benefits from low communication parallel implementations, which are capable of finding better solutions with respect to their sequential counterpart. In particular, the use of a pool of elite solutions offers a diversity of high quality solutions from which workers can restart their searches for better solutions. The pool also provides a mean to implement cooperation and to achieve faster convergence.

Consistent speedups were obtained on experiments performed on a dedicated cluster with up to 24 processors. The cooperative implementation PAR-MP obtained average results systematically better than the others. In particular, it was able to improve the hard targets to several instances and to disclose previously unknown solutions.

This parallel strategy was also implemented and tested using a true grid platform. We reported original results from pioneer computational experiments on a shared computational grid formed by 82 machines distributed over four clusters in three cities, illustrating the potential of the application of computational grids in the fields of metaheuristics and combinatorial optimization.

Given the above favorable results and the enormous potential of computational grids, a broader investigation is underway, exploring the use of a significantly larger number of processors and investigating new programming challenges.

References

[1] R.M. Aiex, M.G.C. Resende, and C.C. Ribeiro. Probability distribution of solution time in GRASP: An experimental investigation. *J. of Heuristics*, 8:343–373, 2002.

[2] R.M. Aiex, M.G.C. Resende, and C.C. Ribeiro. TTTPLOTS: A perl program to create time-to-target plots, 2005, submitted for publication.

[3] Globus Alliance. Globus. online reference at http://www-unix.globus.org/toolkit/, last visited on May 04, 2005.

[4] A. Anagnostopoulos, L. Michel, P. Van Hentenryck, and Y. Vergados. A simulated annealing approach to the traveling tournament problem. In *Proceedings of CPAIOR'03*, 2003.

[5] B.C. Ball and D.B. Webster. Optimal schedules for even-numbered team athletic conferences. *AIIE Transactions*, 9:161–169, 1997.

[6] T. Benoist, F. Laburthe, and B. Rottembourg. Lagrange relaxation and constraint programming collaborative schemes for traveling tournament problems. In *Integration of AI and OR Techniques in Constraint Programming*, pages 15–26, Kent, 2001.

[7] C. Boeres and V.E.F. Rebello. EasyGrid: Towards a framework for the automatic grid enabling of legacy MPI applications. *Concurrency and Computation Practice and Experience*, 17:425–432, 2004.

[8] D. Costa. An evolutionary tabu search algorithm and the NHL scheduling problem. *INFOR*, 33:161–178, 1995.

[9] V.-D. Cung, S.L. Martins, C.C. Ribeiro, and C. Roucairol. Strategies for the parallel implementation of metaheuristics. In C.C. Ribeiro and P. Hansen, editors, *Essays and Surveys in Metaheuristics*, pages 263–308. Kluwer, 2002.

[10] K. Easton, G. Nemhauser, and M. Trick. Solving the travelling tournament problem: a combined integer programming and constraint programming approach. In E. Burke and P. Causmaecker, editors, *Selected Papers from the 4th International Conference on the Practice and Theory of Automated Timetabling*, volume 2740 of *Lecture Notes in Computer Science*, pages 100–109. Springer-Verlag, 2003.

[11] K. Easton, G.L. Nemhauser, and M.A. Trick. The traveling tournament problem: Description and benchmarks. In T. Walsh, editor, *Principles and Practice of Constraint Programming*, volume 2239 of *Lecture Notes in Computer Science*, pages 580–589. Springer-Verlag, 2001.

[12] E.R. Fernandes and C.C. Ribeiro. Using an adaptive memory strategy to improve a multistart heuristic for sequencing by hybridization. In S.E. Nikoletseas, editor, *4th International Workshop on Experimental and Efficient Algorithms*, volume 3503 of *Lecture Notes in Computer Science*, pages 4–15. Springer-Verlag, 2005.

[13] P. Festa and M.G.C. Resende. GRASP: An annotated bibliography. In C.C. Ribeiro and P. Hansen, editors, *Essays and Surveys in Metaheuristics*, pages 325–367. Kluwer, 2002.

[14] I. Foster. *Designing and Building Parallel Programs.* Addison-Wesley, 1995.

[15] I. Foster and C. Kesselman, editors. *The GRID: Blueprint for a New Computing Infrastructure.* 2nd edition. Morgan Kaufmann, 2004.

[16] I. Foster, C. Kesselman, and S. Tuecke. The anatomy of the grid: Enabling scalable virtual organizations. *International Journal of Supercomputer Applications*, 15:200–222, 2001.

[17] L. Gaspero and A. Schaerf. A composite-neighborhood tabu search approach to the traveling tournament problem. *Journal of Heuristics*, to appear.

[18] P. Greistorfer and Stefan Voss. Controlled pool maintenance for metaheuristics. In C. Rego and B. Alidaee, editors, *Metaheuristic Optimization via Memory and Evolution: Tabu Search and Scatter Search*, pages 387–421. Springer, 2004.

[19] P. Hansen and N. Mladenovic. Developments of variable neighborhood search. In C.C. Ribeiro and P. Hansen, editors, *Essays and Surveys in Metaheuristics*, pages 415–439. Kluwer, 2002.

[20] M. Henz. Scheduling a major college basketball conference revisited. *Operations Research*, 49:163–168, 2001.

[21] LAM/MPI parallel computing. Online document at `http://www.lam-mpi.org/`, last visited on July 25, 2005.

[22] A. Lim, B. Rodrigues, and X. Zhang. A simulated annealing and hill-climbing algorithm for the traveling tournament problem. *European Journal of Operations Research*, 2005 (to appear).

[23] H.R. Lourenço, O. Martins, and T. Stutzle. Iterated local search. In F. Glover and G. Kochenberger, editors, *Handbook of Metaheuristics*, pages 321–353. Kluwer, 2002.

[24] M. Matsumoto and T. Nishimura. Mersenne twister: A 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Transactions on Modeling and Computer Simulation*, 8:3–30, 1998.

[25] Message Passing Forum. MPI: A message passing interface. Technical report, University of Tennessee, 1995.

[26] A.P. Nascimento, A.C. Sena, J.A. da Silva, D.Q.C. Vianna, C. Boeres, and V.E.F. Rebello. Managing the execution of large scale MPI applications on computational grids. In C. Amorim, G. Silva, V. Rebello, and J. Dongarra, editors, *Proceedings of the 17th International Symposium on Computer Architecture and High Performance Computing*, pages 69–76, Rio de Janeiro, 2005. IEEE Computer Society Press.

[27] G.L. Nemhauser and M.A. Trick. Scheduling a major college basketball conference. *Operations Research*, 46:1–8, 1998.

[28] R. Rasmussen and M. Trick. A Benders approach to the constrained minimum break problem. *European Journal of Operational Research*, to appear.

[29] M.G.C. Resende and C.C. Ribeiro. Greedy randomized adaptive search procedures. In F. Glover and G. Kochenberger, editors, *Handbook of Metaheuristics*, pages 219–249. Kluwer, 2003.

[30] M.G.C. Resende and C.C. Ribeiro. GRASP with path-relinking: Recent advances and applications. In T. Ibaraki, K. Nonobe, and M. Yagiura, editors, *Metaheuristics: Progress as Real Problem Solvers*, pages 29–63. Kluwer, 2005.

[31] C.C. Ribeiro and I. Rosseti. A parallel GRASP for the 2-path network design problem. In B. Monien and R. Feldman, editors, *Parallel Processing: 8th International Euro-Par Conference*, volume 2400 of *Lecture Notes in Computer Science*, pages 922–926. Springer-Verlag, 2002.

[32] C.C. Ribeiro and I. Rosseti. Efficient parallel cooperative implementations of GRASP heuristics, 2005, submitted for publication.

[33] C.C. Ribeiro, R.C. Souza, and C.E.C. Vieira. A comparative computational study of random number generators. *Pacific Journal of Optimization*, 1:565–578, 2005.

[34] C.C. Ribeiro and S. Urrutia. Heuristics for the mirrored traveling tournament problem. *European Journal of Operational Research*, to appear.

[35] R.A. Russell and J.M. Leung. Devising a cost-effective schedule for a baseball league. *Operations Research*, 42:614–625, 1994.

[36] J.A.M. Schreuder. Combinatorial aspects of construction of competition Dutch professional football leagues. *Discrete Applied Mathematics*, 35:301–312, 1992.

[37] J.M. Thompson. Kicking timetabling problems into touch. *OR Insight*, 12:7–15, 1999.

[38] M.A. Trick. Challenge traveling tournament instances. Online document at `http://mat.gsia.cmu.edu/TOURN/`, last visited on May 29, 2005.

[39] M.G.A. Verhoeven and E.H.L. Aarts. Parallel local search. *Journal of Heuristics*, 1:43–65, 1995.

[40] M.B. Wright. Scheduling English cricket umpires. *Journal of the Operational Research Society*, 42:447–452, 1991.

[41] J.T. Yang, H.D. Huang, and J.T. Horng. Devising a cost effective basketball scheduling by evolutionary algorithms. In *Proceedings of the 2002 Congress on Evolutionary Computation*, pages 1660–1665, Honolulu, 2002.