A Distributed and Hierarchical Strategy for Autonomic Grid-enabled Cooperative Metaheuristics with Applications

Aletéia P.F. Araújo[a], Cristina Boeres[b], Vinod E.F. Rebello[b], Celso C. Ribeiro[b]

[a]*Department of Computer Science, Universidade de Brasília, Brasília, DF 70910-900, Brazil*
[b]*Universidade Federal Fluminense, Institute of Computing, Rua Passo da Pátria 156, Niterói, RJ 24210-240, Brazil*

Abstract

The adoption of the same cluster-based programming strategies for grid applications, although requiring minimal effort from a programmer's point of view, does not always take advantage of the available computational resources to their fullest extent. This paper investigates on the impact of a distributed and hierarchical autonomic strategy on the performance of parallel metaheuristics to solve hard combinatorial optimization problems on grids. Two problems, the mirrored traveling tournament problem and the bounded diameter minimum spanning tree problem, for which high quality sequential heuristics based on the paradigms of the GRASP and ILS metaheuristics already exist, are employed as case-studies. The computational results obtained on a grid by the novel autonomic strategy show that outstanding performance improvements over the traditional master-worker parallelization approach can be achieved.

Key words: Grid computing, parallel metaheuristics, GRASP, ILS, autonomic computing.

1. Introduction and Motivation

The growing computational power requirements of large scale applications and the high costs of developing and maintaining supercomputers has fueled the drive for cheaper high performance computing environments. With the considerable increase in commodity computing and network performance, cluster computing [13] and, more recently, grid computing [25] have emerged as effective alternatives to traditional supercomputing environments for the execution of parallel applications that require significantly more computing power than communication bandwidth.

A computing cluster generally consists of a fixed number of homogeneous resources interconnected on a single administrative high speed network and dedicated to the execution of one parallel application at a time. A computational grid aims to harness sufficient computing power from a diverse set of resources typically geographically distributed across the internet, in order to execute a number of applications simultaneously. The aggregation of collections (or sites) of resources frequently means that raw compute power in larger quantities than that available at an individual cluster are now accessible to almost any researcher.

Although hard combinatorial optimization problems often require long computation times, they are still among those applications that have yet to fully embrace and benefit from grid computing. Instance sizes that can be solved by either exact algorithms or metaheuristics in reasonable time frames are limited by theoretical complexity bounds and the computing requirements of existing implementations. Harnessing computational grids in order to offer researchers the opportunity to solve realistically sized problem instances and apply these benefits to a variety of real world applications is of utmost importance.

The fact that grid resources are generally distributed, heterogeneous and non-dedicated, make writing grid-enabled parallel applications much more challenging [23]. Unlike clusters, not only are the computing power and network capacities/latencies non-uniform (due to resource heterogeneity), but these also fluctuate over time due to resource sharing with other applications belonging to either local or other grid users. Furthermore, inherent due to their size and

Email addresses: `aleteia@cic.unb.br` (Aletéia P.F. Araújo), `boeres@ic.uff.br` (Cristina Boeres), `vinod@ic.uff.br` (Vinod E.F. Rebello), `celso@ic.uff.br` (Celso C. Ribeiro)

distribution, grids are more susceptible to resource and network failures. Addressing these issues places an additional burden on programmers when adapting their cluster-based applications to run on the grid.

The main obstacle to adopting grid computing seems therefore to be the challenge of overcoming the complexity involved in writing grid-enabled parallel metaheuristics. The most efficient solution does not lie with the approach of adopting the same cluster-based programming strategies for grid applications, due to their inability to address the dynamic nature of the environment. With resources belonging to different owners, applications must also cope with different access and usage policies simultaneously.

To hide the intricacies of grid computing from the application writer, this paper describes an alternative, highly effective, low effort strategy for porting sequential metaheuristics to computational grids. In this context, the sequential version of the metaheuristic implementation is transformed into a two-layer parallel program. All issues related to the parallel execution of the metaheuristic in the grid environment are resolved with the EasyGrid Application Management System (AMS) [11, 52] implemented in an underlying *middleware layer*. This system is an application-specific middleware designed to create an autonomic application capable of utilizing the grid in accordance with resource availability [43]. It not only improves the performance and robustness of an application, but actually frees the programmer to focus solely on algorithmic aspects of the problem at hand without having to worry about specific details of a given target architecture.

The EasyGrid AMS has been enhanced with a new *application layer* that employs a hierarchy of distributed pools of elite solutions to support cooperation between parallel executions of the sequential heuristic. The hierarchical organization of the pools leads to smaller running times, while their distribution takes advantage of the grid architecture to foster diversity and facilitates effective independent local searches. This combination of EasyGrid AMS (middleware layer) with the parallelization strategy based on the hierarchy of distributed pools (application layer) not only improves the running times, but also contributes to find better quality solutions to several benchmark instances. Moreover, the complexities of grids and parallelism become transparent to the metaheuristic programmer.

This parallelization strategy can be applied to any number of metaheuristics and their hybrids. In this paper, it is validated by the results obtained for two difficult combinatorial optimization problems: the mirrored Traveling Tournament Problem (mTTP) [19, 55] and the Diameter Constrained Minimum Spanning Tree Problem (DCMSTP) problem [1, 50]. The choice of these two problems was driven by the existence of high quality algorithms for their solution that can be used not only in comparative studies, but also as the main components of the parallel implementations: a sequential heuristic and its parallel implementation using a centralized pool of elite solutions within a straightforward master-worker strategy for the mTTP, and an efficient sequential hybrid heuristic for the DCMSTP.

The remainder of the paper is organized as follows. Work related to this paper is described next. Section 2.1 reviews other projects and efforts towards the parallelization of metaheuristics on grids. Section 2.2 describes the formulation of the mirrored traveling tournament problem and summarizes the hybrid GRASP with ILS sequential heuristic of Ribeiro and Urrutia [49] and preliminary efforts towards its parallel implementation in grids [6, 7]. Section 2.3 reports on the formulation of the diameter constrained minimum spanning tree problem and gives a description of a hybrid heuristic for its solution [39]. The distributed autonomic strategy proposed for executing sequential metaheuristics on computational grids is presented in Section 3: the application layer is described in Section 3.1 and the middleware layer in 3.2. Section 4 evaluates the experimental results obtained with the proposed strategy. Concluding remarks are made in the last section.

2. Related Work

In this section, we review work related with this paper. First, we present an account of other environments and projects for the implementation of metaheuristics and exact methods for combinatorial optimization problems in grids. Next, we review the formulation and the main sequential heuristics available for the two application problems that will be used as case studies: the mirrored traveling tournament problem and the diameter constrained minimum spanning tree problem. Surveys about the parallelization of metaheuristics and applications can be found, e.g. in [4, 17, 40, 41, 54].

2.1. Grid Environments for Parallel Metaheuristics

The programming effort required to assemble grid-enabling parallel software may be hard. A variety of tools and projects under development offer middleware, functionalities, and services that facilitate the implementation of optimization algorithms in grid environments.

GridSolve [32] is a RPC based client/agent/server system that allows one to remotely access both hardware and software components. It provides an API to access and schedule grid resources in a seamless way, but is not suited for writing non-embarrassingly parallel codes, i.e., those in which the processes communicate with each other during processing [23].

The AppLeS (Application-Level Scheduling) [10] system provides a software environment for adaptively scheduling and deploying applications in grid environments. It focuses on the development of scheduling agents, in which each agent is written in a case-by-case basis. The agents use the services offered by the NWS (Network Weather Service) to monitor the varying performance of available resources, but the AppLeS agents need to know a priori the number of tasks to be scheduled.

MW [29] is another framework that allows users to parallelize scientific computations using the master-worker paradigm on a computational grid. MW is formed by a set of C++ abstract classes providing interfaces to programmers of applications and grid-infrastructure. To build a grid-enable application with MW, the application programmer must re-implement some virtual functions. Likewise, to port the MW framework to a new grid software toolkit, the grid infrastructure programmer must re-implement a number of virtual functions. The use of MW requires some knowledge about the grid platform.

The hierarchical master-worker paradigm [2, 56] was used in the implementation of grid-oriented parallel branch-and-bound algorithms for distinct combinatorial optimization problems. Services such as load balance and fault tolerance are also provided in these implementations.

Projects that provide functionalities (or services) to facilitate the implementation of grid-enabling parallel metaheuristics can be divided into two groups: those in which the application and management layers are integrated, and those where these layers are decoupled.

In the first group, GridSAT [16] is a parallel boolean satisfiability solver of non-trivial SAT problems, based on a special form of the master-worker model for grids, in which the individual workers may communicate directly with each other whenever necessary. The problem to be solved is split into subproblems, which are independently investigated for satisfiability. All management tasks are implemented exclusively in the master process: the resource manager, the worker manager, the scheduler, and the checkpoint service.

Also in the first group we find PSEPMH [35], which is a problem solving environment for combinatorial optimization based on parallel meta-heuristics to help specialists to harness heterogeneous computational resources and handle dynamic granularity control. It requires the decomposition of the problem into two sub-problems by divide-and-conquer. The compiler generates mobile agent code that automatically forms adaptive multi-granularity parallel computations at runtime by cloning itself and distributing copies along the grid environment.

In the second group, with separate application and management layers, the application programmer does not need any knowledge of the grid infrastructure. In this category, ParadisEO-CMW [15] is a framework for designing and deploying parallel metaheuristics on computational grids, assembling together the ParadisEO [14] and MW frameworks. ParadisEO (application layer) is dedicated to the reusable design of parallel hybrid metaheuristics. However, grid-enabling an application with MW (middleware) involves the reimplementation of a number of virtual functions.

In this paper, we show the benefits offered by a distributed autonomic strategy to efficiently parallelize grid-enabled metaheuristics. The distributed and hierarchical strategy in the application layer provides greater flexibility to the programmer and greater scalability. It is supported by the EasyGrid AMS running in the middleware layer, which ensures that the application may run in heterogeneous and non-dedicated platforms. This architecture increases portability, ensuring that the parallel metaheuristic may run on different grids without any changes in the application layer. Furthermore, while ParadisEO-CMW is limited to running a number of tasks that must be created in the beginning of the execution of the application, the EasyGrid AMS creates the tasks on demand during the execution.

2.2. The Mirrored Traveling Tournament Problem

Professional sport leagues involve millions of fans and significant investments in players, broadcast rights, merchandising, and advertising. Multiple agents, such as the organizers, media, players, fans, security forces and airlines, are involved and play major roles in the organization of leagues and tournaments. Teams and professional sports leagues do not want to have their return in investments in players and infrastructure wasted as a consequence of a poor scheduling of games. An annotated bibliography on fundamentals and applications of scheduling algorithms in sports appears in [36].

We consider a tournament played by n teams, where n is an even number. In a *simple round-robin* (SRR) tournament, each team plays every other exactly once in $n-1$ rounds. In a *double round-robin* (DRR) tournament, each team plays every other twice, once at home and the other away. It is assumed 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.

The *Traveling Tournament Problem* (TTP) was first established by Easton et al. [19] and consists in finding a DRR tournament schedule such that every team does not play more than three consecutive home games or more than three consecutive away games, no repeaters (i.e., two consecutive games between the same two teams) occur, and the sum of the distances traveled by the teams is minimized. 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. Benchmark challenge instances have been proposed and are available in [55]. To date, even small benchmark instances of the TTP with $n = 10$ teams cannot be solved exactly.

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 $k = 1, \ldots, n - 1$, although with reversed venues. Once again, the objective consists in minimizing 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.

GRASP (Greedy Randomized Adaptive Search Procedure) metaheuristic [47] is a multi-start heuristic, 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 [21, 22].

The ILS (Iterated Local Search) metaheuristic [38] starts from a locally optimal feasible solution. A random perturbation is applied to the current solution, which is then followed by 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.

Ribeiro and Urrutia [49] proposed the `GRILS-mTTP` heuristic for the approximate solution of the mTTP. It is not only the most effective heuristic for the mTTP, but was also able to find the best known solutions for some benchmark non-mirrored instances of the TTP. This heuristic is based on the hybridization of GRASP with ILS. Basically, the local search phase of GRASP is replaced by an ILS procedure. The pseudo-code in Figure 1 summarizes the main steps of the `GRILS-mTTP` heuristic.

Each iteration of the outer while loop in lines 1 to 11 of Figure 1 starts by a GRASP *construction phase* that builds an initial solution S in line 2. This is followed by an ILS *local search phase* that starts in line 3 by attempting to improve the current solution S. Three different types of moves are considered: team swaps, home-away swaps, and partial round swaps. Once a local optimum with respect to the team swaps is found, a quick local search using home-away swaps is performed. Next, partial round swaps are investigated, followed again by a local search using home-away swaps. This scheme is repeated until a local optimum with respect to these three neighborhoods is found and saved in S and $\underline{S}$.

The ILS phase of the iteration proceeds to the inner repeat loop in lines 4 to 10, which starts in line 5 by applying a perturbation move in the game rotation neighborhood to the current solution S, obtaining a new solution S'. A game rotation perturbation move consists in enforcing some specific game to be played at a given tournament round. The same local search strategy is then applied to S'. The solution S' resulting from local search is accepted or not as the new current solution, depending on an acceptance criterion. The best overall solution S^* and the best solution found in the current iteration of the outer loop are updated, if necessary, and a new cycle starts with the perturbation of the current solution, until a re-initialization criterion is met.

Re-initialization occurs if too many perturbations followed by local search are performed without improving the best solution in the current GRASP iteration. A new iteration of the outer loop starts if 50 consecutive deteriorating

```
Procedure GRILS-mTTP();
1.  while .NOT. stopping criterion do
2.      $S \leftarrow$ BuildGreedyRandomizedSolution();
3.      $\underline{S}, S \leftarrow$ LocalSearch($S$);
4.      repeat
5.          $S' \leftarrow$ Perturbation($S$);
6.          $S' \leftarrow$ LocalSearch($S'$);
7.          $S \leftarrow$ AcceptanceCriterion($S, S'$);
8.          $S^* \leftarrow$ UpdateGlobalBestSolution($S, S^*$);
9.          $\underline{S} \leftarrow$ UpdateIterationBestSolution($S, \underline{S}$);
10.     until ReinitializationCriterion;
11. end;
12. return $S^*$;
```

Figure 1: Pseudo-code of the GRASP with ILS heuristic for the mTTP.

solutions have been accepted since the last time $\underline{S}$ (the best solution found in the current iteration of the outer loop) was updated. The outer loop is interrupted when some stopping criterion is met.

As an attempt to speedup the sequential heuristic, a straightforward parallelization of the GRILS-mTTP heuristic was developed in [6, 7], using MPI and the conventional master-worker paradigm to exploit the benefits of cluster or grid environments. It basically consists of a single master process which coordinates multiple workers, each of them executing a slightly modified version of the sequential heuristic. The algorithm fosters cooperation between the workers by means of a centralized pool of elite solutions handled by the master, which collects and distributes elite solutions (found by the workers along their search trajectories) upon request. Whenever a worker completes an iteration, it can either request an elite solution from the pool or construct a new initial solution, with probabilities Q and $1 - Q$, respectively.

By exchanging meaningful information in a timely manner so as that the search in parallel achieves a better performance than the simple concatenation of the results of the individual methods, this parallel implementation obtained good performance and achieved reasonable speedups in clusters [7].

However, a number of issues remain to be resolved efficiently in order to harness the true potential of the grid environment and this cannot be achieved following the master-worker paradigm. The centralized master is a potential bottleneck, and worker resources remain idle due to communication latencies and synchronization costs with the master. Other management issues such as resource discovery and selection, process allocation and scheduling, and fault tolerance, add complexity to the master process and need to be addressed to optimize the execution in grid platforms.

2.3. Diameter Constrained Minimum Spanning Tree Problem

Let $G = (V, E)$ be a connected undirected graph with a set V of vertices, a set E of edges, and a cost c_{ij} associated with every edge $[i, j] \in E$. The diameter of a spanning tree of G is defined as the number of edges in the longest path linking any two nodes $i, j \in V$ in this spanning tree, with $i \neq j$. Given an integer parameter $2 \leq D \leq |V| - 1$, the *Diameter Constrained Minimum Spanning Tree Problem* (DCMSTP) seeks a least cost spanning tree of G whose diameter does not exceed D. This problem was proved to be NP-hard when $D \geq 4$ [26] and has been used as a model for applications in telecommunications [8], data compression [12], and distributed mutual exclusion in parallel computing [18, 45].

Exact algorithms for solving DCMSTP mostly rely on mixed integer linear programming formulations based on multi-commodity flows [27, 28]. Santos et al. [50] presented a model based on the lifted Miller-Tucker-Zemlin inequalities. A branch-and-cut algorithm was suggested in [30]. However, the applicability of exact methods is restricted, in practice, to instances with less than 100 nodes in complete graphs. Heuristics based on different metaheuristics have been proposed in [31, 39, 44]. The hybrid heuristic proposed by Santos et al. [39] is also built upon the hybridization of the GRASP and ILS metaheuristics and follow the same scheme presented in Figure 1.

The GRASP construction phase builds initial solutions using a randomized version of the greedy construction heuristic OTT-M2 [39]. This heuristic starts with a tree consisting of an arbitrarily chosen node that is progressively extended by nodes randomly selected from a *restricted candidate list* (RCL), until all nodes are connected. To add more diversity to the search, the heuristic alternates between two perturbations applied during the ILS local search: DCN (dislocation of the center towards a neighbor) and RSR (random substitution of the root). In a DCN perturbation, a child node of the central node of the tree is considered the new center of the tree in the even case. In the case odd, a child node of one of the extremities of the central edge becomes an extremity of the central edge. In a RSR perturbation, a node is chosen at random to be the new center of the tree in the even case, or to replace one of the extremities of the central edge, in the odd case. All children of the last central node in the odd case (respectively, from one end of the former central edge in the odd case) are inherited by the new node.

Metaheuristics may require significant computation time to find good solutions to hard optimization problems. To speedup their performance, developers are turning, though somewhat tentatively, to parallelization in clusters and grids. Easing this transition by hiding the intricacies of grid computing, the next section presents an alternative strategy to develop parallel metaheuristics.

3. Distributed Autonomic Parallelization

Although the master-worker strategy may be efficient in cluster environments, using a single master may be a potential bottleneck in the presence of a large number of processors. In a grid, the application performance will be clearly affected as the number of workers increases and by the fact that resources are geographically distributed. Also, good performance depends of an efficient mechanism to monitor and react to changes in the environment.

We propose a distributed and hierarchical strategy for the implementation of parallel metaheuristics running under the EasyGrid AMS middleware in computational grids. Management and application related functions are separated in a middleware layer and an application layer, respectively. The EasyGrid AMS middleware is able to transform existing parallel applications into autonomic and system-aware implementations, which are adaptive, fault tolerant, and capable of reacting to changes of load and resource availability of the platform. The application layer is composed of independent processes that find solutions and keep the best of them in a hierarchy of distributed pools. The integration of these two layers effectively creates a framework to transform a sequential metaheuristic into an autonomic parallel version capable of automatically adapting to run time changes within a computational grid.

3.1. Application Layer: Distributed and Hierarchical Strategy

With the middleware layer taking responsibility for managing the grid execution of the application processes, this layer focuses solely on solving the problem and is subdivided hierarchically in three levels, as seen in Figure 2. At the top level, a single *cooperation pool* (CP) management process is responsible for maintaining a global pool of elite solutions. The second level employs an *intensification pool* (IP) process at each site of the grid to maintain local pools of elite solutions for each site. At a third level, *solution provider* (SP) processes apply identical or different heuristics to build solutions for the problem at hand. The following sections describe the main functions performed at each level of the application layer.

3.1.1. Solution Providers

These user application processes contain the problem specific code (e.g. the sequential heuristic chosen to solve the given optimization problem) and are executed in parallel at the lower level of the application layer. By following a standardized communication protocol, the internal functionality of each process is independent of the strategy. Thus, the programmer can choose to employ the same or different metaheuristics in each processes. Even a mix of heuristics and exact methods can be used simultaneously. The solution providers processes are independent and asynchronous, but interact by exchanging information through the distributed and hierarchical structure of the pools created to facilitate cooperation between them, without overloading the system.

In the parallel implementation of the heuristics developed for the problems presented in Sections 2.2 and 2.3, the SP processes execute slightly modified versions of the sequential algorithms in order to use pool of elite solutions.

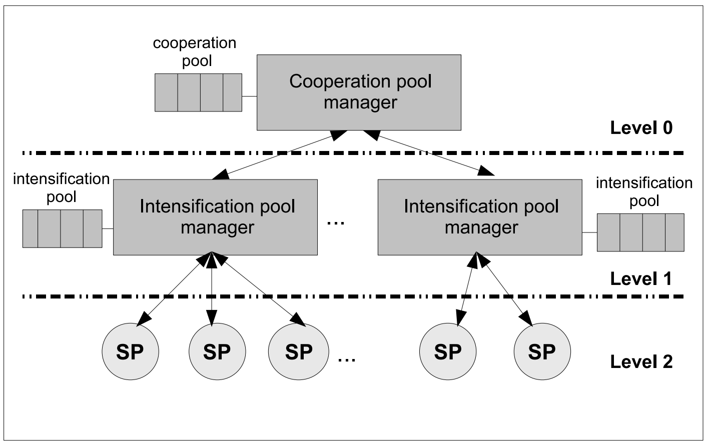

Figure 2: Application layer of the proposed strategy.

Each SP process randomly chooses if it should initiate its execution from the GRASP construction phase or from the ILS local search phase. In the first case, an SP process constructs an initial solution during the GRASP phase and then proceeds to the ILS local search phase. Otherwise, it requests an elite solution from the local intensification pool by sending a message to the respective IP manager process. If this pool is empty, the SP is obliged to initiate from the GRASP phase, otherwise it receives a solution and initiates from the ILS phase. Before finalizing, the SP process sends the best solution found to the associated local IP manager.

3.1.2. Intensification Pools

The IP manager processes play two important roles: they maintain local intensification pools of elite solutions and exchange solutions with the global cooperation pool. Each local pool has a limited number of positions that are initially empty and supports four essential operations: insertion of a new solution; selection of a pool solution from which an SP process can initiate an ILS phase; sending the best solutions to the CP manager; and refreshing the pool after its stabilization.

The first operation (insertion of a new solution) is activated every time an IP manager receives a solution from an SP process. The new solution is considered for insertion in the intensification pool only if it is different from all solutions presently stored. If the new solution was obtained by applying the ILS local search phase to an elite solution obtained from an intensification pool and it improves the original one, then the new solution replaces the original elite solution in the pool. If the new solution was obtained from a GRASP construction phase and the pool is not full, then the new solution is inserted into any free position of the pool; otherwise (i.e., if the pool is full), the new solution is compared with the worst in the pool and replaces it in case of improvement.

The second operation (selection of a pool solution) occurs whenever a SP process requests an elite solution from the intensification pool. If the pool is empty, then the IP manager indicates to the SP process to build a new solution by applying the GRASP construction phase. Otherwise, it randomly selects any elite solution from the pool. The third operation (sending the best solutions) is performed whenever an intensification pool becomes full. At this time, a number of the best solutions in the intensification pool is sent to the cooperation pool.

The fourth operation (a pool refresh) is executed whenever an intensification pool stabilizes. This situation occurs whenever a given number of consecutive solutions received from local SP processes have not been accepted for insertion in the intensification pool. In this case, the IP manager empties its intensification pool and reinitializes it with a small number of solutions (set at 50% in our implementation) that have been requested from the CP manager and randomly selected from the cooperation pool. This refresh procedure makes it possible to exchange elite solutions obtained by SP processes running in different sites of the grid.

3.1.3. Cooperation Pool Manager

Cooperation between SP processes in different sites is crucial to improve the performance of the parallel application. It is achieved through the global cooperation pool, at the top of the hierarchy. For this, the CP manager process supports three operations: insertion of new solutions received from an intensification pool; sending solutions to an intensification pool; and sending a renewal request in case of stabilization. The CP manager process is also responsible for finalizing the application.

The first operation (insertion of a new solution) follows rules very similar to those for insertion in the intensification pool. The second operation (sending solutions to an intensification pool) occurs whenever an intensification pool stabilizes. In this case, the CP manager randomly selects a fraction of its solutions and sends them to the stabilized IP. Since some of these solutions may have been supplied by different IP managers, this operation allows search diversification by providing new solutions to refresh the stabilized intensification pool. The third operation (renewal request) takes place whenever the cooperation pool stabilizes after receiving a large number (set at five times the size of the cooperation pool in our implementation) of consecutive solutions from the same intensification pool that are all worse than the solutions currently stored. In this case, the CP manager sends a renewal request to the corresponding IP manager to empty its intensification pool. The CP manager terminates when the stopping criterion is met. Termination is detected by the middleware layer, which in turn terminates the execution of the remaining application processes and IP managers.

We notice that the proposed hierarchical organization of distributed pools handles communications efficiently, since most information is locally exchanged and there is very little traffic between distinct sites.

3.2. Middleware Layer: EasyGrid AMS

Embedded into the application and permitting the latter to manage its own execution efficiently, the middleware layer is responsible for the creation of the application layer processes, message routing and delivery, load balancing and fault tolerance processes, and discovery of available resources. These management functions are carried out transparently to the application layer, in accordance with environmental conditions and application demand. The EasyGrid middleware [42, 52] is based on standard MPI implementations which support dynamic process creation (e.g. MPI/LAM) and thus requires no specialized system installation.

For reasons of portability, no modifications have been made to the standard release of MPI/LAM library and this AMS middleware does not need to be installed on grid resources. The middleware functionality is incorporated at the application level transforming the algorithm of the user into an autonomic application. Autonomic applications are adaptive, robust to resource failure, self-scheduling programs capable of reacting to changes that occur in shared, dynamic, and unstable distributed environments like computational grids. A key distinguishing characteristic of the EasyGrid AMS is the adopted execution model. Applications need to be written so that the available parallelism is maximized, independently of the number of available processors, in order to take advantage of dynamic environments. Although the traditional "one process per processor" model is effective for dedicated homogeneous clusters, an implementation using a larger number of shorter running processes can obtain better performance in a grid in spite of process overheads [51, 52]. Furthermore, for sake of efficiency, the management policies employed by the EasyGrid AMS can be tuned to the specific needs of each application.

Unlike the standard execution of an MPI program, the EasyGrid AMS does not create all SP processes at once. Their creation is carried out dynamically and orchestrated according to a distributed scheduling policy. Due to the fault tolerance limitations of MPI, the EasyGrid AMS architecture adopts a distributed three level hierarchy of management processes to control the execution of the MPI application efficiently and robustly on a computational grid. A single *global manager* (GM), at the top level, supervises the sites in the grid where the application is running (or could run). At each of these sites, a *site manager* (SM) is responsible for the allocation of the application processes to the machines available at the site. Finally, a *host manager* (HM) on each machine is responsible for the scheduling, creation and execution of the application processes associated with that host.

Initially, the GM process of the middleware layer creates an SM at each site and the CP manager (application layer). Each SM in turn creates its local IP manager and, on each resource at its site, creates a HM process with an initial number of SP processes allocated to its work queue. Note that the initial number of SP processes allocated to each resource is proportional to its processing capacity.

Executing on each grid resource, a HM is responsible for deciding, in accordance with that resource's access policy (e.g. execution may only be permitted at certain hours or when the resource is idle), when a SP may be executed. By monitoring the SP processes and resource utilization, each HM determines the appropriate number of processes that may execute concurrently, in order to maximize performance, but without allowing its host to become overloaded.

Each SM monitors the state of the site, keeping the load balanced in accordance with the computational power offered by each resource, which may be heterogeneous or may share computing cycles with other applications. The SM distributes new tasks to its resources according to their requirements. In this way, sites always have sufficient tasks and never leave resources idle unnecessarily, in accordance with a dynamic scheduler policy. Additionally, if the workload changes dynamically on the grid resources, the tasks are re-scheduled automatically by the load balancing mechanism embedded in the application.

By dynamically creating processes, distinct inter-communicators between each pair of processes are used so that faults can be easily identified and isolated. In the EasyGrid AMS, lighter SP processes than the usual one-process-per-resource in the traditional master-worker model are defined. Therefore, an SP process is reinitiated on the same processor rather than using checkpointing mechanisms in case of failure. Should the same process continue to fail, it is flagged for re-execution on an alternative resource. If re-execution is still unsuccessful, this is reported to the SM and the process is permanently removed from the list of processes to be created. The application is consequently capable of recovering if failures occur, without the need for the entire application be re-executed again.

In the mechanism of fault tolerance of the EasyGrid AMS, the only failure that the application itself cannot recover from is a failure in the GM process. In this case, the application must be re-executed. If any SM process fails, it is detected and recreated by the GM process. If a HM process fails, the SM process of its site recreates it and sends its remaning workload. If recreation is not possible, for example due to hardware failure or to a change in access policy, the workload of this HM is re-scheduled across other HMs of the same site [53].

By compiling the middleware layer together with the application layer, all features of EasyGrid AMS are automatically embedded in the application. Figure 3 shows an example of an integration between an application with five SP processes and the EasyGrid AMS managing two sites of the computational grid. This integration results in an autonomic and portable parallel application since, due to the middleware being embedded in the application, it can efficiently be executed on any grid environment with LAM/MPI installed (Globus [24] is used to provide authentication and authorization to access grid resources and to permit the file transfer of the MPI program to grid resources).

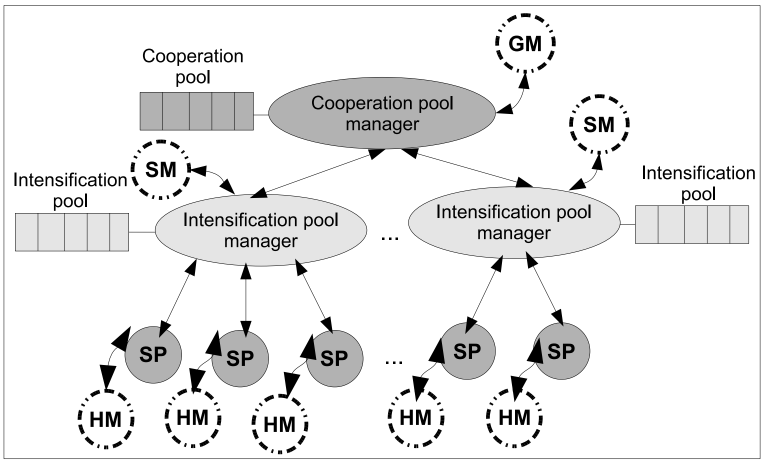

Figure 3: Integration of the application layer with the middleware layer, managing two sites of the grid.

4. Experimental Results

This section summarizes the experiments carried out to evaluate the distributed and hierarchical strategy, used in the development of parallel implementations of the GRASP with ILS heuristics for the mTTP and DCMSTP problems described in Sections 2.2 and 2.3, denoted by audi-mTTp and audi-DCMST, respectively. The audi-mTTp implementation is compared with the sequential GRASP with ILS heuristic described in Section 2.2 and with its PAR-*M*P parallel version developed in [6] and also summarized in Section2.2. audi-DCMST is compared with the sequential hybrid heuristic described in Section 2.3. The parallel implementations for both problems have been developed using C++ and version 7.0.6 of MPI/LAM [37].

The experiments used up to 60 resources from three sites of a real grid environment, known as Grid Sinergia [46] (a research oriented production class computational grid in the southeast of Brazil), located in three different cities within the state of Rio de Janeiro: (a) one cluster in Rio de Janeiro, with 22 Pentium II 400 MHz machines, denoted as S1; (b) one cluster in Niterói with 28 Pentium IV 2.6 GHz machines and three Pentium IV 3.2 GHz machines, denoted as S2 and 40km away from the first; and (c) another cluster in Petrópolis, with seven Pentium IV 3.2 GHz machines and 100km away from the first, denoted as S3. Grid Sinergia employs version 2.4 of the Globus Toolkit middleware across participating sites, which are interconnected by the Brazilian experimental high speed optical research network *Rede Giga*.

All the computation times are reported in seconds and correspond to average results over five runs with different seeds. Since the experiments could not be executed on dedicated resources, all runs took place at night when resources were typically idle and network traffic was low.

4.1. Tests with audi-mTTp

Three sets of benchmark instances have been proposed for the mTTP [19]. The first is made up of *circle instances* (circ*n* denotes an instance with *n* teams), artificially generated to represent easier instances. The second set consists of realistic instances, generated using the distances between the home cities of a subset of teams playing in the National League of the Major League Baseball in the United States (nl*n* denotes an instance with *n* teams). The third is made up of one real-life instance (br24) defined by the home cities of the 24 teams that played in the top division of the 2003 edition of the Brazilian soccer championship [49]. All instances and their best known solutions are available from [55].

The first experiment involved all circle and National League instances. Its goal was to evaluate the quality of the solutions achieved by the sequential, PAR-*M*P, and audi-mTTp implementations. Each row in Table 1 corresponds to a specific instance. For each instance, the table presents the best solution values found by the sequential implementation, by PAR-*M*P, and by audi-mTTp. The last column shows the improvement obtained by audi-mTTp with respect to the best solution value obtained by the sequential algorithm. The solutions values reported for the sequential version are the best found after five days of processing [49], while those obtained by PAR-*M*P and audi-mTTp were the best after 20 hours on ten machines from site S2.

Table 1: Solution values found by the sequential, PAR-*M*P, and audi-mTTp implementations.

Instance	Sequential	PAR-*M*P	audi-mTTp	Improvement (%)
circ8	140	140	140	-
circ10	276	272	272	1.45
circ12	456	456	456	-
circ14	714	714	**696**	2.52
circ16	1004	978	**974**	2.99
circ18	1364	1306	1306	4.25
circ20	1882	1882	1882	-
nl8	41928	41928	41928	-
nl10	63832	63832	63832	-
nl12	120655	120655	**119608**	0.87
nl14	208086	208086	**199363**	4.19
nl16	285614	279618	**279077**	2.29
br24	506433	503158	**502971**	0.68

The solution values highlighted in bold in Table 1 correspond to instances for which audi-mTTP improved upon the solutions obtained by the other implementations. audi-mTTp improved the results obtained by the sequential (resp. PAR-*M*P) implementation for eight (resp. six) out of the 13 instances tested, using essentially the same code.

These results illustrate that the proposed parallel implementation outperformed the others in terms of solution quality.

The second experiment was designed to compare the proposed distributed and hierarchical strategy with the master-worker traditional strategy, used by PAR-*M*P version. This test was performed on the same ten machines of site S2 involved in the previous experiment.

Table 2 shows for each of the 13 instances of the first and second groups, the target value used as stopping criterion for both algorithms and the average time in seconds over five runs of PAR-*M*P and audi-mTTp.

Table 2: Average times in seconds over five runs of PAR-*M*P and audi-mTTp running on ten machines.

Instance	Target	PAR-*M*P	audi-mTTp
circ8	140	0.03	1.28
circ10	276	629.43	297.54
circ12	456	0.18	2.57
circ14	714	0.47	2.53
circ16	982	3,744.72	575.66
circ18	1308	6,703.00	3,570.78
circ20	1882	26.64	27.49
nl8	41928	0.05	1.86
nl10	63832	43.18	43.19
nl12	120655	2.71	3.69
nl14	208086	2.43	4.75
nl16	280116	3,894.20	2,705.62
br24	504000	4,905.63	1,154.02

In principle, this test environment with all machines in the same cluster should be favorable to the master-worker strategy, because of its homogeneity and connexity. In fact, for the smaller instances running in less than 200 seconds, PAR-*M*P was faster than audi-mTTp. This is due to the fact that, for small instances, the time taken by audi-mTTp to activate all middleware layer processes and to create the hierarchy of pools is greater than the time needed by PAR-*M*P to reach the stopping criterion. However, the audi-mTTp implementation with only one intensification pool already outperformed PAR-*M*P for all instances that required execution times longer than 200 seconds, illustrating the efficiency of the proposed distributed and hierarchical strategy. For instance circ16, for example, the execution time of PAR-*M*P was nearly seven times longer than that of audi-mTTp.

The third experiment addressed the behavior of audi-mTTp when more than one intensification pool is used. The target values used as stopping criterion are the same depicted in Table 2. Table 3 displays, for each instance, the average times in seconds over five runs of audi-mTTp using one or two intensification pools. It also shows the reduction in percent obtained with the use of two intensification pools.

These results show that audi-mTTp benefits from the use of one additional intensification pool, since the load of the IP manager processes is divided by two and the diversity of the solutions in the intensification pools increase. The average improvement amounts to a reduction of 19% in the running times when two intensification pools are used.

The fourth experiment compares the performance of PAR-*M*P and audi-mTTp in two different platforms. Results in a LAN environment refer to experiments performed on a local cluster, formed exclusively by machines of site S2. Times on a WAN environment were obtained from runs on machines spread over two clusters, one in site S1 and the other in S2: the PAR-*M*P master process and the audi-mTTp CP manager process execute in one of the sites, while the other processes execute in the other site. This situation reflects the extreme case, in which communication between the two sites achieves its maximum. This experiment involved only the five mTTP instances with longer running times (circ10, circ16, circ18, nl16, and br24). As before, the algorithms stop whenever they find a solution with cost less than or equal to the target displayed in Table 2. All times in Table 4 are reported in seconds and have been averaged over five runs on ten machines for each instance. The execution times of PAR-*M*P and audi-mTTp (using two intensification pools) on the LAN environment were extracted from Tables 2 and 3, respectively.

The advantage of the proposed distributed and hierarchical strategy is clear even for LAN environments which favor the master-worker approach. The fourth and seventh columns in Table 4 display the deterioration in the execution

Table 3: Average times in seconds over five runs of audi-mTTp running on ten machines with one and two intensification pools.

Instance	One Pool	Two Pools	Reduction (%)
circ8	1.28	0.92	28
circ10	297.54	252.26	15
circ12	2.57	1.28	50
circ14	2.53	1.81	28
circ16	575.66	515.86	10
circ18	3,570.78	3,482.06	2
circ20	27.49	24.31	12
nl8	1.86	0.98	47
nl10	43.19	41.80	3
nl12	3.69	3.36	9
nl14	4.75	2.78	41
nl16	2,705.62	2,685.08	1
br24	1,154.02	1,129.62	2
		Average	19

Table 4: Average times in seconds over five runs of PAR-*M*P and audi-mTTp on LAN and WAN environments with ten machines.

	PAR-*M*P (seconds)			audi-mTTp (seconds)			$\frac{\text{PAR-}MP}{\text{audi-mTTp}}$	
Instance	LAN	WAN	deg.(%)	LAN	WAN	deg.(%)	LAN	WAN
circ10	629.43	747.52	18.76	252.26	257.95	2.26	2.50	2.90
circ16	3,744.72	4,537.74	21.18	515.86	521.49	1.09	7.26	8.70
circ18	6,703.00	7,532.03	12.37	3,482.06	3,516.47	0.99	1.93	2.14
nl16	3,894.20	4,500.76	15.58	2,685.08	2,723.68	1.44	1.45	1.65
br24	4,905.63	5,106.51	4.09	1,129.62	1,190.35	5.38	4.34	4.29
		Averages	14.04			2.01	3.50	3.94

time on the WAN environment with respect to the LAN environment, due to the increase in inter-site communication. They show that the deterioration in the running times is much smaller in the case of audi-mTTp, which is able to keep most communications locally even if the machines are spread over multiple sites, thanks to the use of the local intensification pools. The running times of PAR-*M*P deteriorate on average by 14.40% in the WAN environment, while audi-mTTp maintains almost the same performance with an average degradation of only 2.01%. The average speedup of audi-mTTp over PAR-*M*P is also larger for the WAN environment, showing once more that it is more appropriate to grids.

To further evaluate and compare the behavior of the two parallel implementations, their *time-to-target solution value* plots [3, 48] for the measured running times were also used. This approach is based on plots showing the empirical run time distributions. To plot the running time distribution of an algorithm, we choose a problem instance and set a target value. Next, this algorithm is executed N times and its running time to find the first solution as least as good as the target value is recorded. 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 $i = 1, \ldots, N$. Therefore, for each implementation and for each point $z_i = (t_i, p_i)$, with $i = 1, \ldots, N$, p_i denotes the probability that this implementation will find a solution at least as good as the target value in time less than or equal to t_i. In other words, run time distributions or time-to-target plots display on the ordinate axis the probability that an algorithm will find a solution at least as good as a given target value within a given running time, shown on the abscissa axis. Time-to-target plots were first used by Feo et al. [20]. Run time distributions have been advocated also by Hoos and Stützle [33, 34] as a way to characterize the running times of stochastic algorithms for combinatorial optimization.

Figure 4 displays the running time distribution (or the time-to-target plot) for instance circ10, obtained from 100 independent runs of PAR-*MP* and audi-mTTp implementations on LAN and WAN environments. The target value used as the stopping criterion was 274.

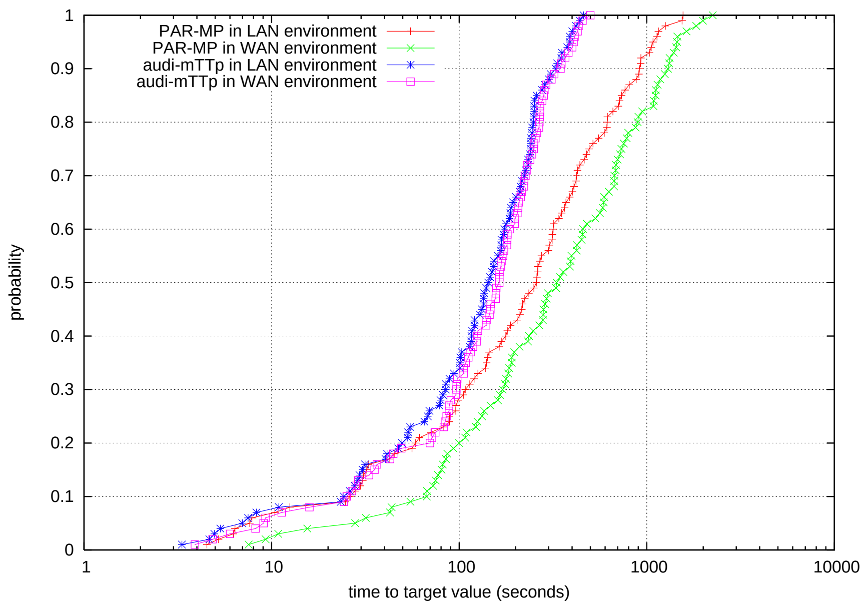

Figure 4: Empirical running time distributions of PAR-*MP* and audi-mTTp on LAN and WAN environments on ten machines.

The curves corresponding to audi-mTTp running in LAN and WAN environments in Figure 4 almost overlap, once again showing that there is almost no degradation when the LAN environment is replaced by a WAN environment.

The next experiment investigates the scalability of the audi-mTTp implementation with respect to the number of intensification pools. The experiment was performed on 60 machines of the same Grid Sinergia. Table 5 shows, for each instance, the average time in seconds over ten runs of PAR-*MP* and audi-mTTp implementations. The third column in this table shows the running times taken by PAR-*MP* to reach the stopping criterion. The next three columns show the running times of audi-mTTp when three, five, and seven intensification pools, followed by the speedups observed for audi-mTTp with respect to PAR-*MP*.

Table 5: Average times in seconds over ten runs of PAR-*MP* and audi-mTTp on 60 machines with three, five, and seven intensification pools .

		audi-mTTp			$\frac{\text{PAR-}MP}{\text{audi-mTTp}}$		
Instance	PAR-*MP*	3 pools	5 pools	7 pools	3 pools	5 pools	7 pools
circ10	72.11	36.84	31.02	32.57	1.96	2.32	2.21
circ16	980.66	345.03	319.22	321.54	2.84	3.07	3.05
circ18	5,647.42	1,822.45	1,783.69	1,801.76	3.10	3.17	3.13
nl16	2,759.23	1,975.99	1,678.51	1,708.97	1.40	1.64	1.61
br24	1,244.28	493.82	374.01	390.68	2.52	3.33	3.18
				Average	2.36	2.71	2.64

The results in this table show that for 60 machines, all execution times of audi-mTTp are shorter than for PAR-*MP*. audi-mTTp obtained the best results with five intensification pools, reducing the execution times obtained with PAR-*MP* on the average by a factor of 2.71. However, these results also show that the parallel application seems to become saturated when seven intensification pools are specified. This can also be viewed in the time-to-target plots displayed in Figure 5 for instance circ10 with the target value set at 274, in which we show the run time distributions of PAR-*MP* and audi-mTTp with three, five, seven, and nine IP managers. The best running times correspond to the leftmost curve, which is that of audi-mTTp running with five intensification pools.

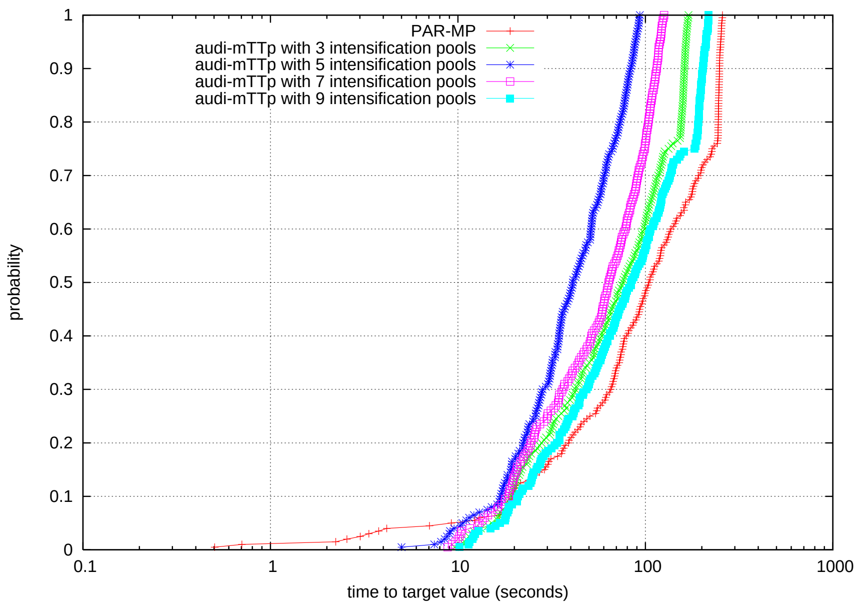

Figure 5: Empirical running time distributions of PAR-*MP* and audi-mTTp on 60 machines with three, five, seven, and nine intensification pools.

The last experiment focus on the overall quality of the solutions obtained by audi-mTTp for the National League instances. Table 6 shows, for each nl*n* instance, the best known solution value before [5], from which the results presented in this section have been extracted, followed by the best solution value found by the sequential heuristic `GRILS-mTTP` and by the audi-mTTp parallel implementation on ten machines from site S2.

Table 6: Best solutions for National League instances: results for audi-mTTp on ten machines.

Instance	Previous best	Sequential	audi-mTTp
nl16	248818	251289	249806
nl18	299903	299903	**299112**
nl20	359748	359748	359748
nl22	418086	418086	**418022**
nl24	467135	467135	**465491**
nl26	554670	554670	**548643**
nl28	618801	618801	**609788**
nl30	740458	740458	**739697**
nl32	924559	924559	**914620**

The audi-mTTp parallel implementation improved the best known solutions for seven out of the nine National League instances (results highlighted in bold), and matched the best known solution in the literature [55] for nl20. The longest running time over all National League instances was 10.2 hours (nfl24), while the shortest was 3.01 hours for nfl18.

4.2. *Tests with audi-DCMST*

Three groups of instances with different characteristics were used to evaluate audi-DCMST. Group 1 consists of 43 instances used in [45], of which 15 are sparse graphs with 20 to 60 vertices and 28 are complete graphs with 10 to 25 vertices. The diameters of the instances in this group vary from 4 to 10. Optimal solution are known for all instances in this group [39].

Group 2 is composed of 12 sparse graph instances used in [27]. Six of them have random costs, while the other six were generated in the Euclidean plane. Instances in this group have diameter values equal to 5, 7, and 9. Optimal solutions are also known for all instances in this group [39].

Group 3 is formed by 30 complete graph instances obtained from the OR-Library [9], originally proposed for the Euclidean Steiner Tree problem. These instances are formed by points in the unit square, with edge costs equal to the Euclidean distances between their extremities. We consider only the first five instances of each size (50, 70, 100, 250, 500 and 1,000 vertices), since these are the hardest instances.

The first experiment addresses the comparison of the solutions found by the sequential algorithm with those obtained by the audi-DCMST implementation running on ten machines of site S2. Both algorithms received exactly the same execution time, defined for each instance as the average running time of 500 iterations of the sequential heuristic. Algorithm audi-DCMST achieved the optimal solutions for all instances of Group 1, outperforming the sequential version that was not able to find the optimal solution for one instance with 60 vertices and 150 edges. The distributed and hierarchical implementation found the optimal solution of this instance in only 35.96 seconds. Figures 6 and 7 illustrate the solutions found by the sequential and parallel implementations, respectively.

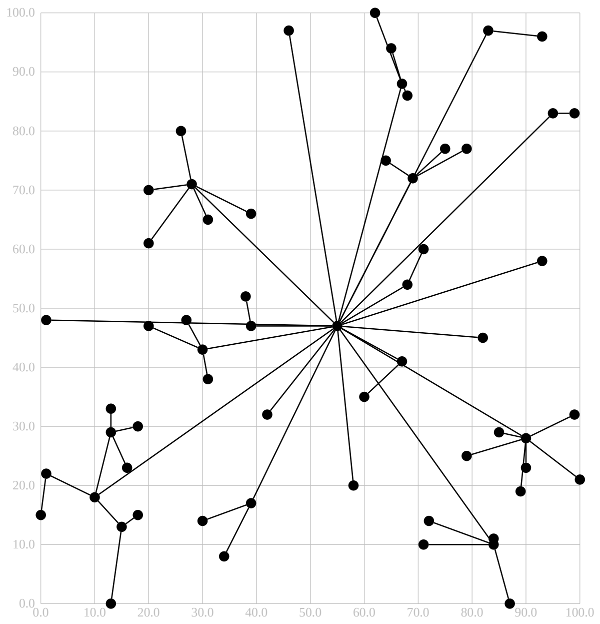

Figure 6: Solution found by the sequential heuristic, with cost equal to 983 (Group 1 instance with 60 vertices and 150 edges).

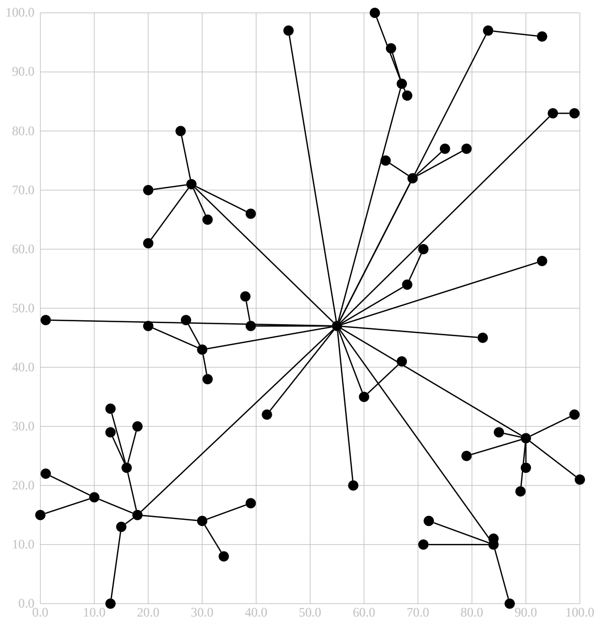

Figure 7: Optimal solution found by the audi-DCMST parallel implementation, with cost equal to 968 (same Group 1 instance with 60 vertices and 150 edges).

Results for Group 2 instances are shown in Table 7, where $|V|$ and $|E|$ denote the number of vertices and edges, respectively, while $|D|$ represents the maximum diameter. Optimal values are given in the fourth column. The time limit given to both algorithms is given in the fifth column. For each instance, the next pair of columns show the cost of the best solution and the average time in seconds to find it over five runs of the sequential heuristic. The last two columns give the cost of the best solution and the average time to find it over five runs of audi-DCMST. Each execution of the distributed heuristic runs for the same time limit as the sequential one.

The sequential heuristic did not reach the optimal solution for four out of the 12 instances in Group 2. Algorithm audi-DCMST performed better than the sequential heuristic, finding two additional optimal solutions. For the two instances that audi-DCMST was not able to find their optimal solutions, the best solutions values have not been worse by more than 0.5% of the optimal. The distributed and hierarchical implementation audi-DCMST was always faster than the sequential heuristic to find the best solution.

Table 7: Solutions found by the sequential and audi-DCMST (on ten machines) algorithms for Group 2 instances.

					Sequential		audi-DCMST	
$\|V\|$	$\|E\|$	$\|D\|$	Optimal	Time limit (s)	Best	Avg. time to best (s)	Best	Avg. time to best (s)
40	400	5	612	10	612	8.34	612	3.58
		7	527	10	527	8.68	527	4.02
		9	495	10	495	9.04	495	4.39
		5	253	10	253	8.59	253	3.88
		7	171	10	171	9.73	171	6.04
		9	154	10	154	9.17	154	5.67
60	600	5	965	50	965	39.33	965	4.80
		7	789	50	796	43.18	793	17.22
		9	738	50	741	41.77	738	18.07
		5	256	50	257	36.54	257	5.13
		7	150	50	152	48.07	150	4.60
		9	124	50	124	42.57	124	8.94

Table 8 displays the results obtained for the 30 instances in Group 3, where $|V|$ and $|E|$ denote the number of vertices and edges, respectively, while $|D|$ represents the maximum diameter. Since the optimal values of these instances are unknown, the fourth column in this table shows the currently best known solution values in the literature at the time of writing [39]. The time limit given to both algorithms is given in the fifth column. For each instance, the next pair of columns show the cost of the best solution and the average time in seconds to find it over five runs of the sequential heuristic. The last two columns give the cost of the best solution and the average time to find it over five runs of audi-DCMST. As before, each execution of the distributed heuristic runs for the same time limit as the sequential one. The distributed and hierarchical audi-DCMST implementation improved the solutions found by the sequential heuristic for all Group 3 instances with more than 100 vertices and was always faster than the sequential heuristic to find the best solution.

To further investigate the impact of the time limit given to the distributed and hierarchical implementation, we extended the time limit given to audi-DCMST in the previous experiment to 1,000 (resp. 25,000) seconds to all Group 3 instances with up to (resp. more than) 70 vertices. The first four columns of Table 9 give the same information as in the previous table. The last two columns give the cost of the best solution and the average time to find it over five runs of audi-DCMST. For six instances in Group 3, the audi-DCMST implementation was able to improve the best known results in the literature (results highlighted in bold). For all instances with more than 50 vertices, the increase in the time limit lead to better solutions.

The last experiment investigates the scalability of the distributed and hierarchical implementation audi-DCMST in a heterogeneous and non dedicated environment. Table 10 shows the results for six instances of Group 3, one for each size of the vertex set: 70, 100, 250, 500, and 1000 vertices. For each instance, this table reports the number of vertices $|V|$, the number of edges $|E|$, the value of the maximum diameter $|D|$, and the target value used as the stopping criterion. The last three columns display the average times (in seconds) over five executions of the audi-DCMST implementation running on 15, 30, and 60 machines. The algorithm scales appropriately, leading to decreasing execution times when the number of machines increase. The proposed distributed and hierarchical strategy, without a centralized processor controlling all others, is very efficient and avoids communication bottlenecks.

5. Conclusions

Computational grids aggregate significant numbers of geographically distributed resources to provide sufficient power for computationally intensive applications. However, the fact that these resources are distributed, typically heterogeneous and non-dedicated, makes writing parallel grid-enabled applications much more challenging.

This paper proposed a new grid-enabled strategy for the implementation of cooperative metaheuristics on computational grids. This strategy is based on hierarchically organized distributed pools of elite solutions and managed by the EasyGrid AMS middleware. The hierarchically distributed organization of pools keeps the majority of communications local, with relatively little traffic between sites, but without eliminating the possibility of a solution provider obtaining information generated by another one at a distinct site.

The grid-enabled distributed and hierarchical strategy was validated by applications in the solution of two challenging combinatorial optimization problems: the mirrored traveling tournament problem and the diameter-constrained minimum spanning tree, for both of which state-of-the-art sequential implementations of metaheuristics were available and at hand for performance evaluation studies.

Extensive computational experiments on benchmark instances of both problems have shown that the proposed grid-enabled approach was able to find better solutions in much smaller running times than the original sequential implementations. We also observed that grid-enabled distributed and hierarchical algorithm scales appropriately, leading to decreasing execution times when the number of machines increase. This new strategy, without a centralized process controlling all others, is very efficient and avoids communication bottlenecks. The numerical results have shown that even with state-of-the-art networks, computation with the conventional master-worker model does not efficiently utilize grid resources. Even small communication overheads can impact performance, especially as the number of processes increases.

The new strategy performs much better because it addresses three important issues. First, it avoids performance degradation caused by high communication overheads by clustering frequent communications between the pools and its solution providers on local resources. Second, it eliminates the bottleneck at a centralized cooperation pool and improves scalability by distributing this function among multiple intensification pools. Third, the computational results also highlighted the benefits of an alternative execution model for MPI programs. With the aid of an application management system, the dynamic creation of processes and the management of messages can be achieved efficiently and transparently, without the metaheuristic designer having to be preoccupied with the implementation of services such as scheduling, load balancing, and fault tolerance. The solution provider processes are executed independently in parallel. They are the only problem specific code that needs to be changed in order to use the proposed strategy to address different combinatorial optimization problems.

References

[1] N. R. Achuthan, L.Caccetta, P. A. Caccetta, and J. F. Geelen. Computational methods for the diameter restricted minimum weight spanning tree problem. *Australasian Journal of Combinatorics*, 10:51–71, 1994.

[2] K. Aida, W. Natsume, and Y. Futakata. Distributed computing with hierarchical master-worker paradigm for parallel branch and bound algorithm. In *Proceedings of the 3rd IEEE/ACM International Symposium on Cluster Computing and the Grid*, pages 156–164, Washington, 2003. IEEE Computer Society.

[3] R.M. Aiex, M.G.C. Resende, and C.C. Ribeiro. TTTPLOTS: A perl program to create time-to-target plots. *Optimization Letters*, 1:355–366, 2007.

[4] E. Alba, editor. *Parallel Metaheuristics: A new class of algorithms*. Wiley, 2005.

[5] A.P.F. Araújo. *Paralelização Autonômica de Metaheurísticas em Ambientes de Grid*. PhD thesis, Catholic University of Rio de Janeiro, Rio de Janeiro, 2008. In Portuguese.

[6] A.P.F. Araújo, M.C.S. Boeres, V.E.F. Rebello, C.C. Ribeiro, and S. Urrutia. Towards grid implementations of metaheuristics for hard combinatorial optimization problems. In *Proceedings of the 17th International Symposium on Computer Architecture and High Performance Computing*, pages 19–26, Rio de Janeiro, 2005.

[7] A.P.F. Araújo, M.C.S. Boeres, V.E.F. Rebello, C.C. Ribeiro, and S. Urrutia. Exploring grid implementations of parallel cooperative metaheuristics: A case study for the mirrored traveling tournament problem. In K.F. Doerner, M. Gendreau, P. Greistorfer, W. Gutjahr, R.F. Hartl, and M. Reimann, editors, *Metaheuristics: Progress in Complex Systems Optimization*, pages 297–322. Springer, 2007.

[8] K. Bala, K. Petropoulos, and T.E. Stern. Multicasting in a linear lightwave network. In *Proceedings of the IEEE INFOCOM'93 Conference on Computer Communications*, volume 3, pages 1350–1358, San Francisco, 1993.

[9] J.E. Beasley. Welcome to OR-Library. online reference at http://people.brunel.ac.uk/mastjjb/jeb/info.html, last visited on June 30, 2011.

[10] F. Berman, R. Wolski, H. Casanova, W. Cirne, H. Dail, M. Faerman, S. M. Figueira, J. Hayes, G. Obertelli, J. M. Schopf, G. Shao, S. Smallen, N.T. Spring, A. Su, and D. Zagorodnov. Adaptive computing on the grid using AppLeS. *IEEE Transactions on Parallel and Distributed Systems*, 14:369–382, 2003.

[11] 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.

[12] A. Bookstein and S.T. Klein. Compression of correlated bitvectors. *Information Systems*, 16:110–118, 2001.

[13] R. Buyya and C. Szyperski, editors. *Cluster Computing*. Nova Science Publishers, Commack, 2001.

[14] S. Cahon, N. Melab, and E.-G. Talbi. Paradiseo: A framework for the reusable design od parallel and distributed metaheuristics. *Journal of Heuristics*, 10:353–376, 2004.

[15] S. Cahon, N. Melab, and E.-G. Talbi. An enabling framework for parallel optimization on the computational grid. In *Proceedings of the 5th IEEE Int. Symp. on Cluster Computing and the Grid*, volume 2, pages 702–709, Washington, 2005.

[16] W. Chrabakh and R. Wolski. GridSAT: A system for solving satisfiability problems using a computational grid. *Parallel Computing*, 32:660–687, 2006.

[17] 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.

[18] N. Deo and A. Abdalla. Computing a diameter-constrained minimum spanning tree in parallel. *Lecture Notes in Computer Science*, 1767:17–31, 2000.

[19] K. Easton, G.L. Nemhauser, and M.A. Trick. The traveling tournament problem: Description and benchmarks. *Lecture Notes in Computer Science*, 2239:580–584, 2001.

[20] T.A. Feo, M.G.C. Resende, and S.H. Smith. A greedy randomized adaptive search procedure for maximum independent set. *Operations Research*, 42:860–878, 1994.

[21] P. Festa and M.G.C. Resende. An annotated bibliography of GRASP - Part I: Algorithms. *International Transactions in Operational Research*, 16:1–14, 2009.

[22] P. Festa and M.G.C. Resende. An annotated bibliography of GRASP - Part II: Applications. *International Transactions in Operational Research*, 16:131–172, 2009.

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

[24] I. Foster and C. Kesselman. Globus: A metacomputing infrastructure toolkit. *The International Journal of Supercomputer Applications and High Performance Computing*, 11:115–128, 1997.

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

[26] M.R. Garey and D.S. Johnson. *Computers and intractability: A guide to the theory of NP-completeness*. W. H. Freeman, New York, 1979.

[27] L. Gouveia and T. L. Magnanti. Network flow models for designing diameter-constrained minimum-spanning and Steiner trees. *Networks*, 41:159–173, 2003.

[28] L. Gouveia, T. L. Magnanti, and C. Requejo. A 2-path approach for odd-diameter-constrained minimum spanning and steiner trees. *Networks*, 44:254–265, 2004.

[29] J. Goux, S. Kulkarni, J. Linderoth, and M. Yoder. Master-worker: An enabling framework for applications on the computational grid. *Cluster Computing*, 4:63–70, 2001.

[30] M. Gruber and G.R. Raidl. A new 0–1 ILP approach for the bounded diameter minimum spanning tree problem. In P. Hansen, N. Mladenovic, J.A.M. Prez, B.M.Batista, and J.M. MorenoVega, editors, *The 2nd International Network Optimization Conference*, pages 178–185. ACM Press, Spa, 2005.

[31] M. Gruber and G.R. Raidl. Variable neighborhood search for the bounded diameter minimum spanning tree problem. In *18th Mini Euro Conference on Variable Neighborhood Search*, pages 1–11, Tenerife, 2005.

[32] M. Hardt, K. Seymour, J. Dongarra, M. Zapf, and N.V. Ruiter. Interactive grid-access using gridsolve and giggle. *Computing and Informatics*, 27:233–248, 2008.

[33] H.H. Hoos and T. Stützle. Evaluation of Las Vegas algorithms - Pitfalls and remedies. In *Proceedings of the 14th Conference on Uncertainty in Artificial Intelligence*, pages 238–245, 1998.

[34] H.H. Hoos and T. Stützle. On the empirical evaluation of Las Vegas algorithms - Position paper. Technical report, Computer Science Department, University of British Columbia, 1998.

[35] R. Huang, S. Tong, W. Sheng, and Z. Fan. A problem solving environment for combinatorial optimization based on parallel meta-heuristics. In *Proceedings of the 7th IEEE International Symposium on Computational Intelligence in Robotics and Automation*, pages 432–437, Jacksonville, 2007. IEEE.

[36] G. Kendall, S. Knust, C.C. Ribeiro, and S. Urrutia. Scheduling in sports: An annotated bibliography. *Computers and Operations Research*, 37:1–19, 2010.

[37] LAM/MPI parallel computing. Online document at `http://www.lam-mpi.org/`, last visited on June 30, 2011.

[38] 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.

[39] A.P. Lucena, C.C. Ribeiro, and A.C. Santos. A hybrid heuristic for the diameter constrained minimum spanning tree problem. *Journal of Global Optimization*, 46:363–381, 2010.

[40] S.L. Martins, C.C. Ribeiro, and I. Rosseti. Applications and parallel implementations of metaheuristics in network design and routing. *Lecture Notes in Computer Science*, 3285:205–213, 2004.

[41] S.L. Martins, C.C. Ribeiro, and I. Rosseti. Application of parallel metaheuristics to optimization problems in telecommunications and bioinformatics. In E.-G. Talbi, editor, *Parallel Combinatorial Optimization*, pages 301–325. Wiley, 2006.

[42] 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 *Proceedings of the 17th International Symposium on Computer Architecture and High Performance Computing*, pages 69–76, Rio de Janeiro, 2005. IEEE.

[43] M. Parashar and S. Hariri. Autonomic computing: An overview. *3566*, pages 257–269, 2005.

[44] G.R. Raidl and B.A. Julstrom. Greedy heuristics and an evolutionary algorithm for the bounded-diameter minimum spanning tree problem. In *ACM Symposium on Applied Computing*, pages 747–752, Melbourne, 2003.

[45] K. Raymond. A tree-based algorithm for distributed mutual exclusion. *ACM Transactions on Computers*, 7:61–77, 1989.

[46] V.E.F. Rebello. Grid Sinergia. online reference at http://easygrid.ic.uff.br/, last visited on June 30, 2011.

[47] 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.

[48] C.C. Ribeiro, I. Rosseti, and R. Vallejos. Exploiting run time distributions to compare sequential and parallel stochastic local search algorithms. *Journal of Global Optimization*, 2011. accepted for publication.

[49] C.C. Ribeiro and S. Urrutia. Heuristics for the mirrored traveling tournament problem. *European Journal of Operational Research*, 179:775–787, 2007.

[50] A.C. Santos, A. Lucena, and C.C. Ribeiro. Solving diameter constrained minimum spanning tree problem in dense graphs. *Lecture Notes in*

Computer Science, 3059:458–467, 2004.

[51] A.C. Sena, A.P. Nascimento, C. Boeres, and V.E.F. Rebello. Easygrid enabling of iterative tightly-coupled parallel MPI applications. In *Proceedings of IEEE International Symposium on Parallel and Distributed Processing with Applications*, pages 199–206, Los Alamitos, 2008. IEEE Computer Society.

[52] A.C. Sena, A.P. Nascimento, J. Silva, D. Vianna, C. Boeres, and V.E.F. Rebello. On the advantages of an alternative grid MPI execution model. In *Proceedings of the 7th IEEE International Symposium on Cluster Computing and the Grid*, pages 575–582, Rio de Janeiro, 2007. IEEE Computer Society.

[53] J.A. Silva and V. E. F. Rebello. Low cost self-healing in MPI applications. *Lecture Notes in Computer Science*, 4757:144–152, 2007.

[54] E.-G. Talbi, editor. *Parallel Combinatorial Optimization*. Wiley, 2006.

[55] M.A. Trick. Challenge traveling tournament instances. Online reference at http://mat.gsia.cmu.edu/TOURN/, last visited on June 30, 2011.

[56] R.V. van Nieuwpoort, T. Kielmann, and H.E. Bal. Efficient load balancing for wide-area divide-and-conquer applications. In *Proceedings of the 8th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 34–43, Snowbird, 2001.

Table 8: Solutions found by the sequential and audi-DCMST (on ten machines) algorithms for Group 3 instances.

					Sequential		audi-DCMST	
$\|V\|$	$\|E\|$	$\|D\|$	S^*	Time limit (s)	Best	Avg. time to best (s)	Best	Avg. time to best (s)
50	1225	5	7.599	10	7.599	8.34	7.599	2.10
			7.609	10	7.609	9.47	7.609	3.10
			7.240	10	7.240	8.21	7.240	1.72
			6.585	10	6.585	7.18	6.585	1.50
			7.248	10	7.248	7.65	7.248	1.31
70	2415	7	7.228	50	7.228	43.28	7.228	21.93
			7.080	50	7.080	42.34	7.080	16.27
			6.983	50	6.983	47.67	6.983	18.23
			7.499	50	7.499	48.93	7.499	32.31
			7.245	50	7.245	43.05	7.245	18.04
100	4950	10	7.759	200	7.835	195.88	7.810	104.03
			7.849	200	7.943	197.62	7.881	173.08
			7.904	200	7.976	188.60	7.930	126.89
			7.977	200	8.044	195.14	8.032	157.84
			8.164	200	8.206	193.11	8.180	131.02
250	31125	15	12.231	1,500	12.451	1,274.18	12.412	973.34
			12.016	1,500	12.307	1,237.28	12.266	847.37
			12.004	1,500	12.132	1,393.55	12.116	1,036.99
			12.462	1,500	12.700	1,432.02	12.606	1,102.66
			12.233	1,500	12.444	1,398.76	12.438	1,087.94
500	124750	20	16.778	3,000	17.212	2,887.82	16.989	2,004.01
			16.626	3,000	17.063	2,973.13	16.810	2,131.29
			16.792	3,000	17.119	2,981.76	16.953	2,200.05
			16.796	3,000	17.250	2,853.44	17.150	1,987.32
			16.421	3,000	16.941	2,799.87	16.860	1,437.81
1000	499500	25	23.434	4,500	24.660	4,367.32	24.509	2,733.40
			23.464	4,500	24.460	4,411.37	24.286	3,021.23
			23.635	4,500	24.317	4,303.19	24.134	2,457.89
			23.787	4,500	24.609	4,401.15	24.270	3,000.43
			23.837	4,500	24.268	4,413.90	24.070	3,423.11

Table 9: Solutions found by audi-DCMST (on ten machines) for Group 3 instances on longer time limits.

				audi-DCMST	
$\|V\|$	$\|E\|$	$\|D\|$	S^*	Best	Avg. time to best (s)
			7.599	7.599	2.10
			7.609	7.609	3.10
50	1225	5	7.240	7.240	1.72
			6.585	6.585	1.50
			7.248	7.248	1.31
			7.228	7.228	21.93
			7.080	7.080	16.27
70	2415	7	6.983	**6.981**	23.14
			7.499	**7.486**	305.67
			7.245	**7.238**	68.89
			7.759	**7.757**	139.11
			7.849	7.849	23,965.71
100	4950	10	7.904	7.926	504.04
			7.977	**7.973**	8,704.78
			8.164	8.176	336.92
			12.231	12.283	11,356.87
			12.016	12.123	8,978.35
250	31125	15	12.004	**11.999**	6,006.41
			12.462	12.472	11,373.14
			12.233	12.272	9,261.90
			16.778	16.899	3,004.49
			16.626	16.810	4,300.82
500	124750	20	16.792	16.907	11,572.04
			16.796	16.987	7,322.45
			16.421	16.555	12,431.17
			23.434	23.488	6,103.87
			23.464	23.575	6,117.43
1000	499500	25	23.635	23.663	5,782.18
			23.787	23.802	4,271.55
			23.837	23.887	5,003.86

Table 10: Scalability results for algorithm audi-DCMST on 15, 30, and 60 machines from Grid Sinergia.

				Time (s)		
$\|V\|$	$\|E\|$	$\|D\|$	Target	15 machines	30 machines	60 machines
70	2415	7	6.983	9.34	8.89	7.16
100	4950	10	7.981	49.23	35.11	24.08
250	31125	15	12.450	3,723.27	2,344.49	1,317.78
500	124750	20	17.063	2,627.14	2,496.45	2,006.08
1000	499500	25	24.609	3,837.79	3,401.71	2,916.25