A Distributed Architecture Supporting Heuristic and Metaheuristic Optimization Methods

Celso M. da Costa[1], Fernando L. Dotti[1], Eder N. Mathias[1], Felipe M. Müller[2]
[1] Programa de Pós-Graduação em Ciência da Computação, Pontifícia Universidade Católica do Rio Grande do Sul
Av. Ipiranga, 6681, Prédio 16, CEP 90619-900, Porto Alegre, RS, Brazil
celso, fldotti, mathias {@inf.pucrs.br}
[2] Programa de Pós-Graduação em Engenharia de Produção, Universidade Federal de Santa Maria
Av. Roraima, Prédio 7, CEP 97105-900, Santa Maria, RS, Brazil
{felipe@inf.ufsm.br}

Abstract—
This paper presents a distributed software architecture that allows the cooperation among research institutions in the field of Operations Research. It has as main aims to share existing algorithms for optimization problems, to allow the easy testing of these algorithms with existing instances, and to share computational power among the cooperating institutions. This is achieved respecting the autonomy and heterogeneity of the cooperating institutions. The five functional blocks that build the architecture are discussed here and also a study case of a Parallel Memetic Algorithm to solve the Traveling Salesman Problem running on this environment is analysed.

Keywords— **Optimization Algorithms, Memetic Algorithm, Master-Slave, Distributed Environment.**

I. Introduction

Due to the growing competition in various sectors of the economy, there is a well defined trend towards the optimization of current processes in order to achieve lower costs, better efficiency, and higher flexibility. Optimization techniques are becoming almost mandatory and widely used. Optimization problems are no rare complex, involving lots of variables, and having a combinatorial structure. Therefore, the Operations Research scientific community is investing efforts in this area. Better and new algorithms for solving optimization problems are under investigation in various research centers. Very often, these algorithms have similar structure, could reuse parts of the solutions, as well as use same input data sets to be tested. Metaheuristic methods combine various solutions in a higher level one, whereby it is important to easily combine executions of existing algorithms.

In order to foster the research work in this area (Operations Research – Optimization Algorithms), this paper presents a distributed environment to support the cooperation among researchers from different institutions.

Due to the intrinsic characteristics of distribution, autonomy and heterogeneity of cooperating parts (research centers, universities, enterprises, etc.), this support environment is organized in the form of a federation of cooperating optimization centers. Each optimization center is capable of supporting the life-cycle of optimization algorithms, and may work stand-alone or within a federation. In the context of a federation, optimization centers share processing power and cooperate to better solve the requests issued by users of that federation. A center supports also a base of input data sets to test optimization algorithms that can be extended with contributions from the users.

The execution of a Memetic Algorithm, which involves various existing heuristics, may have processes spread in various nodes of various optimizations centers of a federation.

This text is organized as follows. In section II we give a brief overview of optimization techniques. In section III a distributed architecture to support the cooperative environment is showed, as well as some related work. In section IV the user interfaces for utilization through the Internet and all phases of a request process to solve an optimization problem are showed. In section V we present a Parallel Memetic Algorithm to solve the Traveling Salesman Problem which runs in the architecture. Finally, in the section VI conclusions and future works developments are outlined.

II. Optimization Techniques

Optimization methods[1] are broadly classified in exact and heuristic. As polynomial exact methods[2] to solve hard combinatorial optimization problems probably does not exists, research efforts to find good solutions in reasonable computational times are important. With this objective, new heuristic methods have been developed. Newell et. al. [NEW58] defined heuristic as "a process that solves a given problem without any guarantees about the quality of the solution found". Heuristics have thus great importance to solve real problems with considerable dimensions in reasonable computational times. Nevertheless, the quality of solutions found using classic heuristics is not satisfactory to all problem classes. Osman [OSM91] divides the heuristics in some

[1]The terms "optimization method" and "optimization algorithm" are used as equivalent in this text.

[2]Methods that solve a problem in a time witch is polinomially bounded by the instance size (the term "instance" is used in the field of optimization to mean the input data set for an optimization algorithm).

groups. We focus on constructive and improvement heuristics and also metaheuristics.

Constructive heuristics create a solution by adding individual components (nodes, arcs, variables, etc.) step by step, until a solution to the problem is generated. This solution can be feasible or not [OSM91]. A feasible solution has a good value in relation the objective function. The objective function is the target of an optimization algorithm.

The Improvement heuristics (local search methods) have as input data a feasible solution. In each phase of these heuristics, components are excluded and inserted in the solution. This mechanism generates a better solution or, in the worst case, a solution with equal quality to the input one.

Metaheuristics are methods that conduct a local search operator to explore the good characteristics and new hopeful regions of solutions [BUR00]. According to the local search operator, the metaheuristics can be divided in two classes:

- the first class is comprised by heuristic methods that explore only one element at the neighborhood (values in the solutions space) in each iteration. Examples are: Simulated Annealing [KIR83], Tabu-Search [GLO93] [FIE94], GRASP (Greedy Randomized Adaptive Procedure) [FEO94], Neural Networks [POT93] and Simulated Jumping [AMI99];
- the second class is comprised by heuristic methods that use a population (a set) of solutions. These heuristics are known as Population Algorithms. Examples are: Genetic Algorithms [GOL89], Memetic Algorithms [MOS99], Scatter Search [GLO77] and Ant Colony Systems – ACO [STÜ99].

III. The Architecture

When developing new optimization techniques is thus very important to publish and reuse within the scientific community; to test various methods against same input data sets; very often it is necessary to share computational resources in order to allow CPU intensive tests; and when working with metaheuristics it is also desirable to be able to combine existing methods (e.g., improvement) or operators (e.g. crossover, mutation) in new methods.

A. Related Work

Concerning related work, three research projects can be mentioned: *NetSolve* [CAS95], *NEOS* [CZY97] and *MetaNEOS* [MET98]. NetSolve was developed at Tennessee University together with the Oak Ridge National Laboratory. NEOS (acronym for Network-Enabled Optimization System Server) and MetaNEOS were developed by Argonne National Laboratory. Below, architectural and functional aspects of these projects are briefly commented.

NetSolve builds a distributed environment where users can submit requests to remote servers supporting numerical libraries. Communication modules called agents are responsible for receiving requests from the user and submitting them to the appropriate server. This environment eases the access to those libraries supporting their location, download and installation. After Casanova [CAS95] these are time consuming tasks for the academic community.

NetSolve agents support communication among servers and are responsible to find out the better server for a given requisition through a load balancing policy. The architecture developed for NetSolve allows for servers or agents to be dynamically started and killed without compromising the integrity of the system. Fault tolerance aspects were also considered in the architecture.

NEOS is an environment developed to allow the resolution of optimization problems using the Internet. It works following a standardized structure for input data and has a list of registered optimization problem solvers. Through the Internet, the user chooses the problem solver, informs the input data set, and issues a request. At the end of the execution, the users receives back the results and run time statistics. To each registered problem solver a manager is assigned which is responsible for the computational resources needed by the server and for answering possible questions coming from users. To cope with resource allocation management (e.g. load balancing) in a distributed environment, NEOS uses the CONDOR environment [LIT88].

The MetaNEOS project is a proposal to add functionality to the already existing environment supported by NEOS. It aims at solving optimization requests in a metacomputing environment. A metacomputing environment is an abstraction to represent the computational resources existing in an infrastructure of loosely coupled processor nodes (e.g. the Internet). The user transparently uses the computational power of the environment without being aware of distribution aspects.

To build this metacomputing environment, a key module is the MetaNEOS resource broker. This module is responsible for the global resource allocation. It manages schedulers defined by MetaNEOS, interacts with schedulers local to the nodes non exclusively dedicated to MetaNEOS processing, as well as deal with heterogeneity aspects in the environment. MetaNEOS plans to support local resource allocation using techniques defined in CONDOR and GLOBUS [GLO97].

B. Our Architecture

The main objective of the architecture here described is to offer an environment to support the cooperation between persons and institutions researching in the area of combinatorial optimization. The following features are identified:

- **Distribution**: the scenario is inherently distributed, as various institutions physically dispersed may want to cooperate;

- **Autonomy and Cooperation**: due to the autonomy of the institutions, a hierarchical approach is regretted, and a cooperative one is followed. Each institution is able to provide a support infrastructure configured according to the local decisions, called an *Optimization Center*. Optimization Centers may cooperate with each other to solve optimization problems issued by their users. In this case a group of Optimization Centers will build a *Federation*;
- **Heterogeneity**: as a consequence of autonomy, the heterogeneity of computational infrastructures must be taken into consideration.

The architecture here described considers these features. Each Optimization Center has five kind of modules: Center Manager, Center Scheduler, Node Manager, Legacy – Executor, and User Interfaces. Figure 1 depicts the whole architecture.

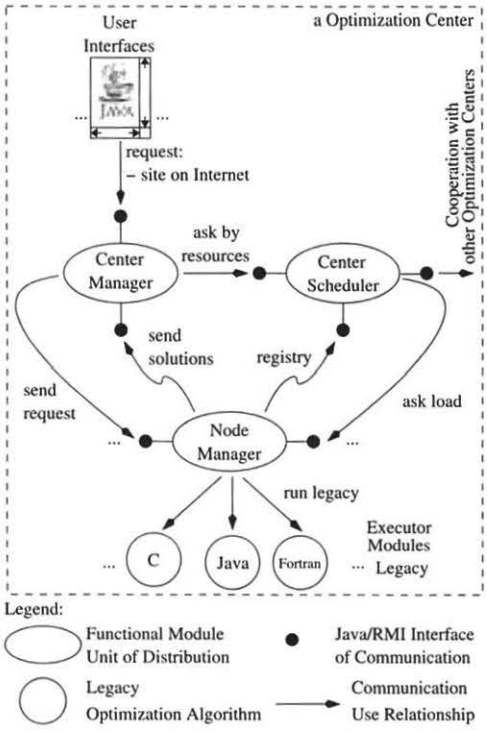

Fig. 1. Distributed architecture to solve optimization problems.

The functional modules may have various computational interfaces. Each module is a unit of distribution, i.e. the various modules can be allocated to different nodes in a computer network. The computational interfaces are defined following an object oriented methodology, as well as the information passed between the modules. Each module of the architecture is discussed next:

- **Executor Modules**: represent implementations of available optimization algorithms. These implementations may be legacy modules for different computational platforms (hardware and operating system) which are integrated in the architecture; or may also be developed having the functionality of the architecture in mind, as the case study in section V: a Parallel Memetic Algorithm to solve the Traveling Salesman Problem;
- **Node Manager Module**: The Node Manager manages a physical node which participates in an Optimization Center. It controls the execution of Executor Modules registered with that node. Upon requisition for an optimization algorithm, it triggers the appropriate Executor Module. The Node Manager has three tables: one with instances, another of Executor Modules that it can execute and another with undergoing requests. During initialization, it registers with the Center Scheduler.
- **Center Manager Module**: The Center Manager initializes a Center Scheduler of the Optimization Center, receives requests from User Interface and sends them to a Node Manager supporting the specified optimization algorithm. This Node Manager is chosen by the Center Scheduler. When the executor finishes it works, it sends the solution (answer to the request) to the Center Manager, which delivers it to the appropriate user.
- **Center Scheduler Module**: A Center Scheduler controls the load balancing of an Optimization Center. It also has a table with all Optimization Centers that it cooperates with, and a table with Node Managers working in that center. When a request is submitted to a Center Manager, it asks the associated Center Scheduler where (in which node) to execute that request. The Center Scheduler further asks the local Node Managers and the federating Center Schedulers.
- **User Interface Module**: The User Interface is the module responsible for interacting with human users of an Optimization Center. While the interface communicates with a center, the service perceived by the user is relative to the whole federation. The User Interfaces communicate with the Center Manager Module in order to load the offers (methods and instances) of the federation, as well as to request the execution of an optimization algorithm. As optimization algorithms may have long execution times, the interface offers the possibility for the user to receive the results in an asynchronous manner, via e-mail. With this, the user can launch a request and disconnect from the Center Manager. The user may also prefer a synchronous behavior, by which the interface is blocked until the reception of the results of the preceding request.

C. *Initializing Process*

The Figure 2 shows the initializing process of n Optimization Centers. In the following we discuss about each phase of this process.

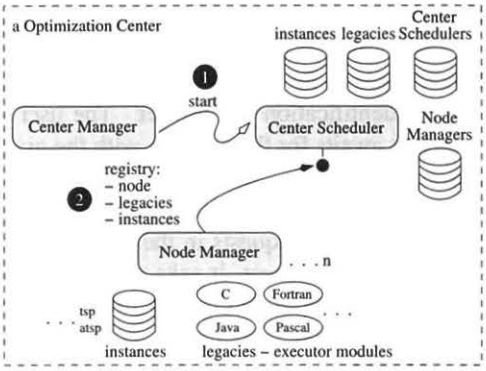

Fig. 2. Initializing process.

- **Phase 1**: The Center Manager is initialized using a name (for example, `CenterManager::PUCRS`) in a machine with an Internet address. It then starts a Center Scheduler (for example, with name `CenterScheduler::PUCRS`) in the same machine. The Center Scheduler has a table (in hard disk) listing all other Center Schedulers that it cooperates with. The job of the Center Manager is to receive requests from users, find out a node to run the request using the Center Scheduler, and send the request to the node chosen by the Center Scheduler;
- **Phase 2**: The n Node Manager can be initialized in any node using a name (for example, `NodeManager::01`) in a machine with Internet address. When a Node Manager starts, it registers itself with a Center Scheduler informing its instances and executor modules.

The result of this initialization process is an Optimization Center with one Center Manager, one Center Scheduler and n Node Managers. The Center Scheduler knows all the instances, executors and Node Managers that take part in this Optimization Center, it also knows other Center Schedulers in other Optimization Centers, if the case.

IV. Request Process

The environment supports two user interfaces: one allows users to request heuristic methods to be processed and another one requests metaheuristic methods. Currently we have a parallel and a sequential version of a Memetic Algorithm for the Traveling Salesman Problem running as metaheuristics in the environment.

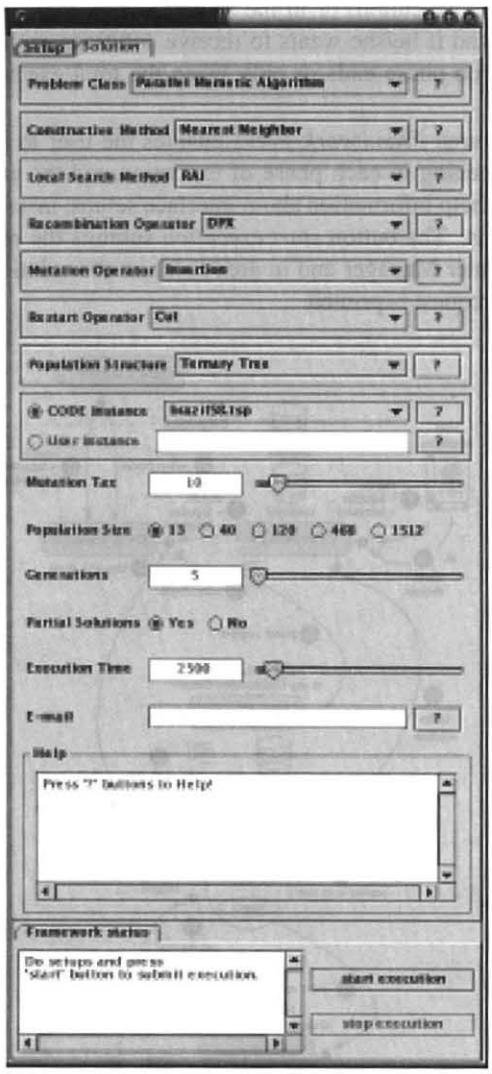

Fig. 3. User Interface to Memetic Algorithms.

Figure 3 shows the user interface used to submit a Parallel or Sequential Memetic Algorithm request to the environment. This interface is a Java applet and runs with any Internet browser. When the user starts it, the applet communicates with the Center Manager from which the applet has been downloaded and a search process is triggered to find instances and methods available in the whole federation (cooperating centers). In this case, for example, Instances, Constructive and Improvement methods for the Traveling Salesman Problem.

Functions available to the user allow him/her to, for instance: select the maximum execution time, select the number of times that the selected method will be applied in the selected instance, set the reception of solutions via e-mail

(off-line execution) or in the same window (on-line execution), and if he/she wants to receive solutions each time an execution phase ends or only when the total execution finishes.

The area *Framework status* enables the user to see what is happening in each phase of execution and the area *Help* shows help information about interface setups, instances and methods. The button *start execution* submits the request to the Center Manager and in area *Solution* the solution found to the request is printed.

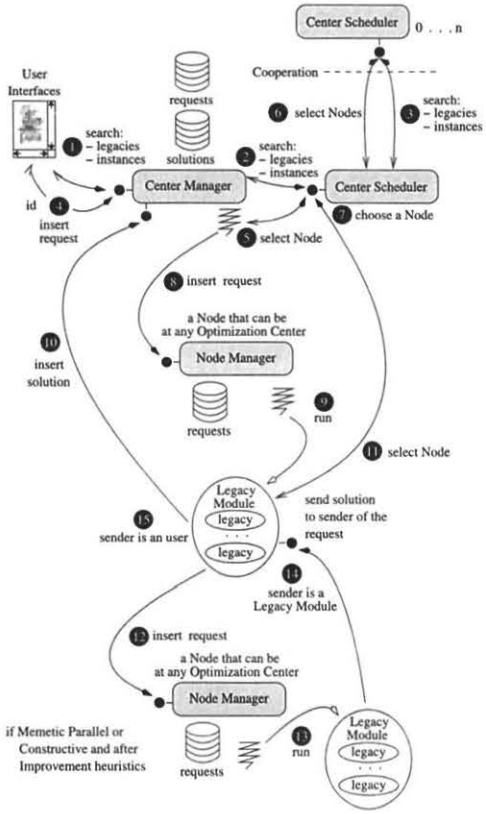

Fig. 4. Request process.

Figure 4 shows the request process, comprising: the initialization the user interface, the creation and submission of the request, the execution of an executor module, and finally the solution returned to the user. In the following we discuss about each phase of this process:

- **Phase 1**: The user interface for heuristics or metaheuristics is initialized. The "search process" is started, it communicates with the Center Manager in an Optimization Center to find all instances and methods available in the distributed environment;
- **Phase 2 and 3**: The Center Manager asks to the Center Scheduler about the instances and methods available in this Optimization Center. The Center Scheduler cooperates (creating a new "search process") with other Optimization Centers to find the instances and methods available in that centers. The Center Scheduler returns the results of the "search process" to the Center Manager that returns to the user interface;
- **Phase 4**: The user configures a request using the interface and submits it to the Center Manager. The Center Manager inserts the request in a request table and returns an identification to the user. The user creates a thread that awaits for the solution with the given identification;
- **Phase 5**: In the Center Manager a thread is responsible for launching new requests in the request table. When it detects a new request, it asks the associated Center Scheduler ("select Node process") for the address of a Node Manager that supports the required method (Executor Module);
- **Phase 6**: The Center Scheduler cooperates with other Center Scheduler to find a Node Manager to handler the request;
- **Phase 7**: After the search, the Center Scheduler selects only one node using the schedulers polices and returns its Internet address to the Center Manager thread;
- **Phase 8**: The Center Scheduler inserts the user request in requests table of chosen Node Manager;
- **Phase 9**: A thread in the Node Manager is awakened to treat the new request in the table. It then starts the Executor Module that handles the request. An Executor Module may control more than one method;
- **Phase 10**: If the request is to execute a simple constructive method, at the end of execution the Executor Module sends the solution to the Center Manager that issued the request. The Center Manager analyses the solution and sends it to the user by e-mail or creates a new entry in the solutions table with `<identification, solution>` and awakens the user thread waiting for that solution (see Phase 4);
- **Phase 11**: If the request is to execute a constructive method followed by an improvement of the achieved solution, for example, the Executor Module executes the constructive method and asks ("select Node process") the Center Scheduler for a Node Manager that can handle the improvement request. Phases 12 to 14 are executed.
- **Phase 12**: The Executor Module sends the request to the chosen Node Manager;
- **Phase 13**: The Node Manager detects the request in its table of requests and triggers the Executor Module that handles the request;
- **Phase 14 and 15**: At the end of execution of the Executor Module, it sends the solution to the previous Ex-

ecutor Module that issued the request (see Phase 11) or sends the solution to the Center Manager that issued the initial request (see Phase 10).

A. Scheduler Algorithm

Currently, the scheduling of tasks (requests in Center Manager) is done following a simple algorithm based on a single circular list of Node Managers.

When the Center Scheduler receives request for Node Manager selection ("select Node process"), it asks other Center Schedulers (that it cooperates with) for a list of all Node Managers that can handle the request.

In this implementation, the resource management algorithm has no knowledge about the state of the machines where the Node Managers are running, for example, the amount of general processes of the operating system, the amount of optimization processes, or the current memory usage in the node.

V. Study Case - Traveling Salesman Problem

The Traveling Salesman Problem, or TSP for short, is one of the most classic optimization problems and it has the following definition[BUR00]: given a finite number of "cities" along with the cost of travel between each pair of them, find the cheapest way of visiting all the cities and returning to the starting point. The travel costs are symmetric in the sense that traveling from city x to city y costs just as much as traveling from y to x, if this does not occur, the travel costs are asymmetric; the "way of visiting all the cities" is simply the order in which the cities are visited. To put it differently, the data consist of integer weights assigned to the edges of a finite complete graph; the objective is to find a Hamiltonian cycle (that is, a cycle passing through all the vertices) with the minimum total weight. In this context, Hamiltonian cycles are commonly called *tours*.

A. Memetic Algorithms

Memetic Algorithms (MAs) are used to identify a broad class of metaheuristics (i.e. general purpose methods aimed to guide an underlying heuristic) which constitutes one of the most successful approaches for combinatorial optimization. They are search algorithms based on mechanics of natural selection and natural genetics. They combine survival of the fittest among string structures with a structured yet randomized information exchange to form a search algorithm with some of innovative flair of human search. In every generation, a new set of artificial individuals is created using bits and pieces of the fittest old individuals; an occasional new part is tried for good measure [MOS99].

MAs use a collection (or population) of solutions, from which, using selective breeding and recombination strategies, better and better solutions can be produced. Simple genetic operators such as *crossover* and *mutation* are used to construct new solutions from pieces of old ones, in such way that for many problems the population steadily improves. Crossover is a simple idea of how to create a new solution (*offspring*) produced by combining the pieces of the original parents, in the TSP case, the parents are two distinct tours. Mutation is the other most commonly used operator, which provides the opportunity to reach parts of the search space which perhaps cannot be reached by crossover alone. It is suggested that at least one "parent" should always be chosen on the basis of its *fitness* – in terms of combinatorial problems, this implies some monotonic function of the objective function value [RAY89]. A key feature, present in most MA implementation, is the use of a population-based search which intends to use all available knowledge about the problem. This knowledge is incorporated in form of heuristics, approximation algorithms, local search techniques, specialized recombination operators, truncated exact methods, and many other ways.

B. Parallel Memetic Algorithm to solve TSP - PMA

The basic idea behind most of parallel problems is to divide a task in small tasks and solve them at the same time using multiple processors (nodes). This technique is known as *divide-and-conquer* and can be applied in many ways in population algorithms. The follow classification is based on a proposition by Cantú-Paz [CAN98] to the genetic algorithms. There are three main types of Parallel Memetic Algorithms (PMA) in the literature: PMA with global single population using master-slave approach, PMA with a single population spatially structured and PMA with multiple populations.

In a master-slave PMA there is a single population (just as in a sequential MA), but the fitness evaluation is distributed among several processors.

The master-slave paradigm does not assume the existence a structured computational architecture. It can be implemented on environments having shared or distributed memory. On an environment with shared memory, the population can be laid in shared memory area where each processor can read the individuals reserved to it and write the result of the evaluation without conflicts.

On a distributed memory environment, the population can be laid in a specified processor. The *master* process is in charge of sending requests for population initialization, evaluation of individuals and other operations to the *slave* process. The slave processes are on other nodes of the distributed environment.

PMA with a single population spatially structured are suited for massively parallel computers. Selection and matching (crossover) are restricted among all the individu-

als. The ideal case is to have only one individual for every processing element available.

PMA with multiple populations (or multiple-demes) are more sophisticated, as they consist on several sub-populations witch exchange individuals occasionally. This exchange of individuals is called migration. Multiple-demes PMA are known with different names. For example. they are known as "distributed" PPA because they are usually implemented on distributed memory MIMD computers.

Algorithm 1 Generic algorithm to PMA.

```
Process Manager ( ) {
   // Parallel initialization of the population
   For i = 0 to InitialPopLen do {
      SelectNode (node);
      Create Process (node,
              Executor (GENINDIVIDUAL, null));
   }
   // Parallel local search for each individual
   For i = 0 to InitialPopLen do {
      ind=Wait_individual ( );
      SelectNode (node);
      Create Process (node,
                      Executor (LOCALSEARCH, ind));
   }
   Repeat {
      // Launch parallel recombinations and mutations
      SelectNode ( node );
      Create Process (node,
                      Recombinations
      (Number_Recombinations));
      // master processes and wait for their termination
      SelectNode (node);
      // before next iteration
      Create Process (node,
                      Mutations(Number_Mutations));
      WaitAll ( );
      Pop = SelectPop (Pop);
      If Pop has converged then pop = RestartPop ( );
   Until Termination_Condition = True;
   }
}
Process Recombinations (Number_Recombinations) {
   For i = 0 until Number_Recombinations) {
      (ind1,ind2) = SelectToMerge ();
      ind3 = Recombine (ind1,ind2);
      SelectNode (node);
      Create Process (node,
                      Executor(LOCALSEARCH, ind3));
    }
}
Process Executor (TYPEOFEXECUTION, individual) {
   If TYPEOFEXECUTION == GENINDIVIDUAL
   Then ind = GeraIndividual ( );
   Else ind = LocalSearch (individual);
   Evaluate (ind);
   AddInPopulation (ind);
}
Process Mutations (Number_Mutations) {
   For i = 0 until Number_Mutations) {
      ind1 = SelectToMutate ( );
      ind2 = Mutate (ind1);
      SelectNode (node);
      Create Process (node,
                      Executor(LOCALSEARCH, ind2));
   }
}
```

We propose a Parallel Memetic Algorithm to solve the TSP problem. It has a global single population using master-slave approach. This approach was chosen because the architecture showed in section III follows this paradigm, what makes it simple to map the PMA to the existing architecture.

A pseudo code for the PMA is showed in Algorithm 1. The process manager controls the execution of the algorithm until its termination. During the initialization of the population, a parallel process is created to generate each individual. The individuals generated are inserted in a population, which is global and stored in the node where the process manager is executing. For each new individual inserted in the population, the process manager creates a new parallel process to perform the local search with that individual. The recombination and mutations processes work as masters. After the selection of individuals of the population (for mutation or recombination), a parallel process is created to perform the local search algorithm with each individual.

Therefore, n parallel processes are created for each of recombination and mutations processes, where n represents the number of iterations. Fitness evaluation and insertion in the population are performed by the executor process. Another important aspect to mention is the use of a global load balancing algorithm to achieve an homogeneous load distribution on the environment.

B.1 Mapping of the PMA under the architecture

The PMA is an Executor Module called `ParalelMemeticAlgorithm`. When an execution request arrived to the Node Manager that has this module, its execution is started (Phases 8 and 9 of Figure 4). The fist phase of a PMA is to create the population. Suppose that the population size is of 13 solutions (individuals), the constructive method to create each individual need to be executed thirteen times. When the PMA wants to create an individual, it asks the Center Scheduler to select a node to execute it (Phases 11, 6 and 7 of Figure 4). Then it sends a new request to the chosen Node Manager to execute the constructive algorithm and awaits for the solution (Phases 12, 13 and 14 of Figure 4). When a solution (individual) is available, a new processes for Local Search is created. This has the same behavior as in the constructive phase, i.e. the Executor Module asks to Center Scheduler for a Node Manager to execute the local search method.

At least a crossover phase is applied on the population. As the crossover and mutation operators are not CPU-bound, it would be not cost effective to execute them in parallel. Crossover and mutation are then executed in the same node of the Executor Module `ParallelMemeticAlgorithm`.

Each phase of the PMA produces partial solutions, they are sent to the Center Manager that issued the request and the user can get it. When the execution is finished, it sends the last solution (Phase 10 of Figure 4) and exits.

VI. Conclusions and Future Works

This paper showed an architecture to support the execution of classical heuristics and metaheuristics. A Parallel Memetic Algorithm was showed to execute over an environment that supports the cooperation among two universities (PUCRS and UFSM). The system is operational and can be found on *http://code.ladd.pucrs.br* or *http://glover.ce.ufsm.br/~mathias*. The algorithm presented here is being implemented using Java/RMI as programming language and communication mechanism.

We are about to weight the cost of the different phases of the distributed execution in order to better analyse the performance. Following to this, a scheduling policys for the existly environment will be choosen, and its impact on the performance of the system measured.

We ared now undergoing a performance evaluation phase where we compare sequential and parallel Memetic Algorithms.

References

[AMI99] AMIN, S. Simulated Jumping. **Operations Research**, EUA, p. 23–38, 1999.

[BUR00] BURION, Luciana Salete. Algoritmo Memético para o Problema do Caixeiro Viajante Assimétrico como parte de um Framework para Algoritmos Evolutivos. **Master Thesis**, Universidade Estadual de Campinas, Faculdade de Engenharia Elétrica e de Computação, São Paulo, Brazil, 2000.

[CAN98] CANTÚ-PAZ, E. A Survey of Parallel Genetic Algorithms. **Calculateurs Paralleles, Reseaux et Systems Reparties**, Paris, v. 10, n. 2, p. 141–171, 1998.

[CAS95] CASANOVA, H.; DONGARRA, J. NetSolve: A Network server for solving computacional science problems. **Technical Report CS-95-313**, University of Tennessee, Knoxville, Tennessee, 1995.

[CZY97] CZYZYK, J.; MESNIER, M. P.; MORE, J. J. The Network-Enabled Optimization System (NEOS) Server, **Preprint MCS-P615-1096**, Argonne National Laboratory, Argonne, Illinois, 1997.

[FEO94] FEO, T.; REZENDE. A greed randomized adaptative search procedure for maximum independent set. **Operations Research**, EUA, ed. 42, p. 860—879, 1994.

[FIE94] FIECHTER, C. N. A Parallel Tabu Search Algorithm for Large Traveling Salesman Problems. **Discrete Applied Mathematics**, EUA, p. 243–267, 1994.

[GLO97] GLOBUS RESOURCE MANAGER SPECIFICATION. Globus working note, available at **http://www.globus.org**, 1997.

[GLO77] GLOVER, F. Heuristics for Integer Programming Using Surrogate Constraints. **Decision Sciences**, EUA, ed. 8, p. 156–166, 1977.

[GLO93] GLOVER, F.; LAGUNA, M. Tabu Search. **Modern Heuristic Techniques**, Blackwell Scientific Publications, Oxford, EUA, p. 70–150, 1993.

[GOL89] GOLDBERG, D. E. Genetic Algorithms in Search, Optimization, and Machine Learning. **Addison-Wesley**, EUA, 1989.

[KIR83] KIRKPATRICK, S.; GELATT, C. D.; VECCHI, M. P. Optimization by Simulated Annealing. **Science**, EUA, ed. 220, p. 671–679, 1983.

[LIT88] LITZKOW, M. J.; LIVNY, M.; MUTKA, M. W. Condor - A hunter for idle workstations. In **Proceedings of the 8th International Conference on Distributed Computing Systems**, Whashington, District of Columbia, IEEE Computer Society Press, p. 108–111, 1988

[MET98] METANEOS: METACOMPUTING ENVIRONMENTS FOR OPTIMIZATION, propose available at **http://www.mcs.anl.gov/metaneos**, 1998.

[MOS99] MOSCATO, Pablo. Memetic Algorithms: A Short Introduction. **New Ideas in Optimization**, McGraw-Hill, Washington, EUA, cap. 2, 1999.

[NEW58] NEWELL, A.; SHAW, J. C.; SIMON, H. A. Empirical explorations with the logic theory machine. **Western Joint Computer Conference**, EUA, p. 218–239, 1958.

[OMG01] OMG-Document. Common Object Request Broker Architecture, **http://www.omg.org**, 2001.

[OSM91] OSMAN, Ibrahim. Heuristics for Combinatorial Optimization Problems: Developments and New Directions. In **Proceedings of the first seminar on Information Technology and Applications**, Marfield Conference Center, EUA, Sep. 1991.

[POT93] POTVIN, J. Y. The Traveling Salesman Problem: A Neural Network Perspective. **ORSA Journal on Computing**, EUA, ed. 5, p. 338–348, 1993.

[RAY89] RAYWARD-SMITH, V. J.; OSMAN, I. H.; REEVES, C. R.; SMITH, G. D. Genetic Algorithms in Search, optimization and Machine Learning. **Addison-Wesley**, EUA, 1989.

[STÜ99] STÜTZLE, T.; DORIGO, M. ACO Algorithms for the Traveling Salesman Problem. **Evolutionary Algorithms in Engineering and Computer Science: Recent Advances in Genetic Algorithms, Evolution Strategies, Evolutionary Programming, Genetic Programming and Industrial Applications**, EUA, 1999.

[VIN97] VINOSKI, S. CORBA: Integrating Diverse Applications Within Distributed Heterogeneous Environments, **IEEE Communications Magazine**, 1997.

[VIN98] VINOSKI, S. New Features for CORBA 3.0, **Communications of the ACM**, vol 41, no 10, October 1998.