RC23944 (W0604-126) April 25, 2006
Computer Science

IBM Research Report

On-line Evolutionary Resource Matching for Job Scheduling in Heterogeneous Grid Environments

Vijay K. Naik
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598

Pawel Garbacki
Delft University of Technology
P.O. Box 5031
2600 GA Delft, The Netherlands

Krishna Kummamuru
IBM India Research Lab
EGL Business Park
Bangalore 560071, India

Yong Zhao
Computer Science Department
University of Chicago
Chicago, IL 60637

Research Division
Almaden - Austin - Beijing - Haifa - India - T. J. Watson - Tokyo - Zurich

LIMITED DISTRIBUTION NOTICE: This report has been submitted for publication outside of IBM and will probably be copyrighted if accepted for publication. It has been issued as a Research Report for early dissemination of its contents. In view of the transfer of copyright to the outside publisher, its distribution outside of IBM prior to publication should be limited to peer communications and specific requests. After outside publication, requests should be filled only by reprints or legally obtained copies of the article (e.g., payment of royalties). Copies may be requested from IBM T. J. Watson Research Center , P. O. Box 218, Yorktown Heights, NY 10598 USA (email: reports@us.ibm.com). Some reports are available on the internet at http://domino.watson.ibm.com/library/CyberDig.nsf/home .

On-line Evolutionary Resource Matching for Job Scheduling in Heterogeneous Grid Environments

Vijay K. Naik
IBM T. J. Watson Research Center
P. O. Box 218
Yorktown Heights, NY 10598
vkn@us.ibm.com

Paweł Garbacki
Delft University of Technology
P. O. Box 5031
2600 GA Delft, The Netherlands
p.garbacki@ewi.tudelft.nl

Krishna Kummamuru
IBM India Research Lab
EGL Business Park
Bangalore 560071, India
kkummamu@in.ibm.com

Yong Zhao
Computer Science Dept
University of Chicago
Chicago, IL 60637
yongzh@cs.uchicago.edu

Abstract

In this paper, we describe a resource matcher (RM) developed for the on-line resource matching in heterogeneous grid environments. RM is based on the principles of Evolutionary Algorithms (EA) and supports dynamic resource sharing, job priorities and preferences, job dependencies on multiple resource types, and resource specific and site-wide policies. We describe the evolutionary algorithm and the models used for representing the resource requirements, preferences, and policies. We evaluate three different methods for bootstrapping RM. We then describe a Evolutionary Matcher (EM) Service – a Web Service based architecture, design and implementation of RM for on-line resource matching. Preliminary performance results indicate that the EM service is efficient in speed and accuracy and can keep up with high job arrival rates – an important criterion for on-line resource matching systems. The service oriented architecture makes the EM service scalable and extensible and can be integrated with already existing grid services in a straightforward manner.

1. Introduction

We consider the resource matching problem in grid environments where resources are heterogeneous, not always available, belong to multiple administrative domains, jobs require multiple types of resources, and system managers often set conflicting policies such as continuous resource sharing, load balancing, high resource utilization and/or throughput. In such an environment, the conditions for resource matching evolve dynamically and are subject to resource specific and system-wide policies. A resource matcher must be able to handle all such requirements.

The problem of matching independent jobs to heterogeneous resources is known to be NP-complete [5]. This means a resource matcher must use a heuristic-based approach. For dynamic grid environments, such an approach must be adaptive and efficient. To be adaptive, the resource matcher must be able to adjust to the dynamic changes in the system and produce reasonable matchings even under high variability. An efficient resource matching algorithm should be able to match jobs with resources in an on-line manner, i. e. it should be fast enough to keep up with the job arrival rate. In this paper, we present a fast and efficient on-line resource matching algorithm that belongs to the class of Evolutionary Algorithms (EA) [1].

The contributions of this paper are as follows. We describe an approach for modeling resource requirements for jobs, job preferences and simultaneous applications of different policies. We express the resource matching problem as an optimization problem and solve it using an evolutionary algorithm. We discuss three alternatives for bootstrapping the EA based matcher. With our approach, we can handle a wide class of job resource requirements, preferences and affinities, job priorities, resource capacity constraints, resource usage and sharing policies. Another contribution of this work is the design of a service based architecture and a prototype implementation of the resource matcher suitable for large-scale grid environments.

The rest of the paper is organized as follows. In Sec-

tion 2, we describe the resource matching problem and introduce our terminology and a model for representing resource characteristics and job requirements. In Section 3, we discuss our EA based resource matching algorithm. In Section 4, we describe the Web service based architecture for the Evolutionary Matcher. Some preliminary performance results are presented in Section 5. Section 6 discusses the related work and we conclude the paper in Section 7.

2. Resource Matching in Grid Environments

In this section, we describe the problem of resource matching in a heterogeneous grid environment and introduce the terminology used in the rest of the paper.

2.1. Model of a Grid Environment

A Grid environment consists of a set of *resources*. Each resource is an instance of a *resource type*. Examples of resource types are servers, file systems, network subsystems, middleware components, databases, applications, etc. Each resource type has one or more *static attributes* and zero or more *dynamic attributes*. For example, a resource type "server" may have the following static attributes: host name, CPU architecture, CPU speed, number of CPUs, number of network adapters, speed of each network adapter, IP address, OS name, an so on. Examples of dynamic attributes of a resource type "server" are: availability status (e.g., up or down), current CPU load, memory usage, available disk space, etc.

Grid resources provide the infrastructure for executing *jobs*. Each job describes its resource requirements by providing a set of *resource dependencies* on one or more types of resources. Each dependency places *attribute constraints* on the attribute values of the resources of a specific type. For example, a job may depend on a resource of type "server" with CPU speed of at least 600 MHz. In addition, a job may define optional *temporal* and *location constraints*. An example of a temporal constraint is a specific job completion time deadline. The location constraint, on the other hand, is specified by a job requesting a set of resources that are collocated; i.e., on the same machine, within the same subnet, or at the same site. In addition to the constraints, a job may specify *preferences*. Preferences provide selection criteria when multiple resource sets satisfy dependencies associated with a job. A job may specify its preferences either by providing a method of ordering qualifying resources or by simply identifying specific resource instances by attribute value or by name. Finally, each job defines the expected *usage values* (consumptions) for the dynamic attributes of the dependent resources. The constraints resulting from resource dependencies, temporal and location restrictions, preferences, and attribute usage values together constitute the requirements of a job. To prioritize jobs, system administrators may assign *job priorities* to the in-coming jobs. Job priorities are relative and may depend on the current workload.

2.2. Policies

Just as jobs have preferences for resources, resources may have preferences for the class of jobs that can use those resources. In some cases, a resource may allow only certain class of jobs to run during certain times of the day. These and other resource usage directives are specified as *resource specific policies*.

The overall use of resources by jobs and arbitration among jobs competing for resources are governed by system-wide policies or goals set by site administrators. Some examples of such policies are: (i) find resource matches for as many jobs as possible (*maximize throughput*), (ii) match higher priority jobs first and then match lower priority jobs (*maximize prioritized throughput*), (iii) balance the workload evenly across the resources (*load balance*), (iv) minimize the number of resources matched, and (v) match high priority jobs with high performance resources.

The system wide policies guide the selection of resources for matching with jobs. In the resource matching problem, these policies are modeled as *objective functions* or *fitness values*. In some cases, system administrators may want to set complex goals as a combination of one or more simpler goals listed above.

2.3. Matching Jobs with Resources

The resource matching problem can be described as finding a match between the jobs and the resources taking into account job requirements and available resource capacities, while optimizing one or more objective functions. In the following, we first clarify the notion of "job requirement" by introducing some additional terminology.

We define the *Qualifying Resource Collection (QRC)* for a job as a minimal (in the sense of inclusion) set of resources that need to be assigned to the job to satisfy its requirements. For each resource in the QRC it is specified how much of its capacity and in which time interval is required by the job. For a given job, there may exist many QRCs, and they collectively represent all possible assignments of the resources to the job. Figure 1 shows the *problem graph*. It represents the job dependencies on QRCs and the mappings from QRCs to resources as a directed acyclic graph.

A solution to the resource matching problem can be represented by a subgraph of the problem graph. Figure 2 shows a *solution graph* for the problem graph in Figure 1.

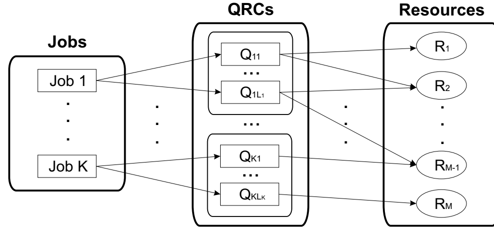

Figure 1. A directed graph representing job dependencies on QRCs and mapping from QRCs to resources.

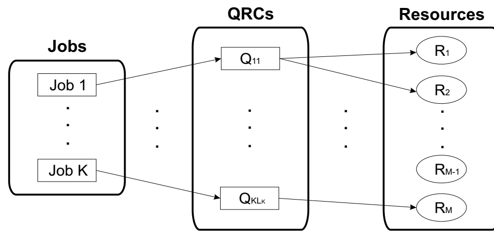

Figure 2. An example of a Solution graph.

An edge between a job and a QRC indicates that the requirements and consumptions of the job are met by the resources in the corresponding QRC. Since a single resource may be shared by multiple jobs, it may occur in many QRCs defined for different jobs. Consequently assigning a certain QRC to a job may render some QRCs infeasible for other jobs because of the resource capacity constraints. An edge in the problem graph guarantees the compatibility of the job requirements with the resource capacities, where as an edge in the solution graph additionally ensures that job consumptions do not exceed resource capacities. The process of finding a feasible/valid solution is to identify the edges of a solution graph.

In this paper, we further extend the requirements according to the resource matching algorithm by constraining its execution time. Regardless of the complexity of the job requirement specifications and the system size measured in the number of available resources, it should be possible to match jobs with resources in an on-line manner. In other words, the algorithms used for resource matching should be fast enough to keep up with job arrival rate. Obviously, the higher the system complexity and the job arrival rate, the poorer the quality of the matchings. The matching algorithm should be able to dynamically adjust to the current conditions producing reasonable matchings even under high system load.

3. Evolutionary Resource Matching

In this section, we describe a resource matcher that we have developed using an evolutionary algorithm. Evolutionary Algorithms (EAs) are optimization methods inspired by the nature [1, 3]. EAs perform a focused random search by simulating an evolution of a *population of solutions*. EAs iteratively simulate evolution of a constant size population of solutions. During each iteration, mutation, recombination, and selection operators are applied to the current population of solutions. The mutation operator randomly perturbs a selected solution. The recombination operator produces new solutions. The selection operator prevents the population size from growing. Evolutionary algorithms are shown to converge under certain conditions to a global optimum [12, 13].

3.1. Overview of Evolutionary Matcher

We apply the evolutionary approach to the problem of resource matching in heterogeneous grid environments. The pseudocode of our algorithm is shown in Figure 3. The resource matcher maintains a constant-size population of solutions. Each solution represents an assignment of resources to jobs taking into account job, resource, and policy specific constraints. As shown in Figure 3, an EA is an iterative procedure, where a number of solutions are initialized before the iterations begin. We have developed a few alternative initialization methods, which we describe in Section 3.2. The iterations continue until some optimization criteria is met. In each iteration, a new set of solutions, referred to as *offspring*, is created by *mutating* the existing solutions. The offspring solutions are added to the population as candidate solutions for evolving the next generation. After the mutation step, a subset of solutions are *selected* according to their *fitness values*. The fitness value represents the quality of a particular solution with respect to a selected objective function. Our evolutionary matcher supports any (combination) of the objective functions introduced in Section 2. The cycle of mutation and selection is repeated until the termination criteria is met. In the following, we explain the key steps in more detail.

3.2. Initialization

In [4], the author has discussed the benefits of initializing the solution populations using domain specific knowledge. Following this approach, we propose three alternative schemes to compute the initial solutions.

Random initializer. As in a typical evolutionary algorithm, the solutions in the initial population are generated randomly and independently of one another. The initializer

Input:
A set of resources with their attributes and capacities
A set of jobs with their dependence on resources
A set of policies

Output: A solution graph

Evolutionary Matcher:
Initialize the population with P solutions
`do` {
  Mutate each solution with a probability
    to generate an offspring solution
  Perform *selection* from original and mutated solutions
    based on their fitness values to obtain P solutions
} `while` (Termination condition is not met)
`return` the best solution found so far.

Figure 3. Evolutionary Matcher

randomly matches jobs with available QRCs without violating the capacity constraints.

Greedy initializer. Some problem instances may have very simple solutions. Evolutionary algorithms, with all their sophistication and generality are cost-ineffective in case of such problems. A simple greedy algorithm, which follows the problem solving meta-heuristic of making a locally optimal choice, may find a solution of the same quality as the evolutionary algorithm, but at a much lower cost [8]. To overcome this shortcoming, we include a solution from a greedy algorithm in the initial solution populations. The remaining solutions are still computed with the random algorithm. If the solution found by the greedy approach is optimal, it will quickly dominate other solutions in the initial population, which will, in turn, seriously decrease the number of iterations of the evolutionary algorithm. We use the first-fit strategy as the greedy heuristic. Our algorithm considers jobs in non-increasing priority order and tries to match the highest priority unassigned job with a QRC. Any job that cannot be matched within the capacity constraints is omitted from the solution.

Backtracking initializer. The natural extension of the population initialization with a greedy algorithm is to use more complex heuristics. Striking a balance between the initializer cost and the range of solution space it opens up for exploration, we consider an initializer based on limited-depth backtracking. This algorithm matches jobs, one-at-a-time and in the order of their priorities, with the available resources. When a job cannot be matched, the algorithm backtracks to the most recent successful job assignment, and tries an alternative of assignment for that job. When all alternatives are exhausted, it tries the alternative assignments for the next previous successful job assignment. If there are no more choice points, the matching omits the current job and moves to the next one. Because the worst case execution cost of the backtracking algorithm is exponential, we put a bound on the number of backtracking steps performed in the process of matching each job.

3.3. Mutation

The mutation operation is used to create a new solution from an existing solution. First, a small fraction of job-QRC matchings in an existing solution are selected according to a random distribution biased towards jobs with lower priorities. For the selected matchings, the job-QRC assignment is removed with a predefined *mutation probability*. This releases some of the resource capacities creating opportunities for other job-resource assignments. Next, new job-resource matchings in the solution are determined as follows: (i) an unmatched job is selected randomly with a probability biased towards jobs with higher priorities, (ii) the selected job is randomly assigned to one of its QRCs that will not violate resource capacity constraints using a probability distribution biased by job specified resource preferences, and (iii) the capacities of the resources in the selected QRC are reduced to reflect the assignment. The above steps are repeated until no more job-resource matchings are possible. This creates a single new solution offspring. At the end of the mutation step, a constant multiple of existing solutions are created and added to the population.

Note that we use a domain-specific mutation operator and a domain-specific representation of solutions. Note also that a solution is always in the feasible region and the mutation operator is able to explore the entire feasible region.

3.4. Selection

In the selection phase, the over-sized population of solutions is reduced back to the initial number of solutions. Each solution in the population is assigned a reproduction probability depending on its fitness value. Using these probabilities, a fraction of solutions from the pool of the old and offspring solutions are selected to form the solution population for the next generation. Additionally, the selection algorithm guarantees that a constant fraction of the fittest solutions in the over-sized population will be retained in the new population.

3.5. Termination condition

The on-line aspect of the resource matching problem, introduces a constraint on the maximal execution time for the evolutionary algorithm. The incremental and evolutionary nature of the EA approach allows one to stop the execution at the end of any iteration and still get an approximation to the optimal solution. Naturally, the longer the algorithm runs, the better the quality of the final solution. In

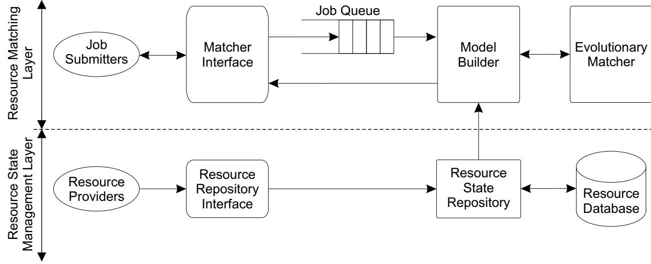

Figure 4. Architecture of the on-line resource matching system.

our system, any of the following three conditions terminates the execution of the evolutionary matcher: (i) the execution time of the matcher exceeds a specified limit, (ii) the number of iterations exceeds a predefined maximal value, (iii) the best found solution has remained unchanged for n iterations, where n is a configurable parameter. When the resource matcher terminates, the best solution found so far is output as the solution to the resource matching problem.

4. Architecture of On-line Evolutionary Matcher

We have designed and implemented an on-line resource matching system, referred to as the *Evolutionary Matcher* (EM) Service, based on the evolutionary matching algorithm described above. Inputs to EM are: the resource requirements for a batch of jobs, job priorities and preferences, current resource states, and resource and site specific policies. EM models the resource matching problem as an optimization problem and solves it for the specified input. Figure 4 shows the architecture of EM. The components of EM can be divided into two functional layers: the Resource State Management Layer and the Resource Matching Layer.
Resource State Management (RSM) Layer. This layer keeps track of the current state of each resource available for job execution. Various *resource providers* specify the static resource attributes and the current values of dynamic attributes. An example of a resource provider is a site administrator, who may specify for a resource instance its resource type, static attribute values, and sharing policies for that resource. The available capacities of the dynamic attributes are updated periodically to reflect the current resource usage. Usually the update is performed by a resource usage monitor, which also acts as a resource provider.

The resource attribute information is input to the RSM Layer as XML documents by making a Web service call to the *Resource Repository Interface* (RRI). (See Figure 4.) RRI is a Web service for translating the resource specific information from the XML documents to the internal data format of the *Resource State Repository* (RSR). RSR maintains the state and other resource specific information for all resources in a *Resource Database*. In addition, RSR performs job requirement-specific intelligent query parsing and query optimizations by caching information.
Resource Matching (RM) Layer. The components in this layer perform the actual matching of jobs with resources. *Job submitters* submit jobs asynchronously by invoking *Matcher Interface*. The resource requirements, consumptions and preferences of the submitted jobs are described in XML. The Matcher Interface is a Web service that translates the job descriptions to the internal EM data structures and, after the matching is finished, performs a reverse operation of creating and sending an XML document describing the matched resources to the job submitters.

Job Queue caches all arriving jobs if the resource matcher is busy processing jobs that arrived earlier. *Model Builder* consults RSR to obtain the current state of all resources relevant to a batch of jobs waiting in the Job Queue and constructs a model which is used by the EA-based *Resource Matcher*.

5. Performance Evaluation

We now describe a prototype implementation of EM and present the performance results obtained with this implementation.

5.1. Experimental Setup

We have implemented the evolutionary resource matching algorithm described in Section 3 and encapsulated it in a Web service using the architecture described in Section 4. All EM components are implementation is in Java and are developed for deployment on IBM WebSphere Application Server with the Resource Database implemented on top of IBM DB2 database server. The performance results presented here were obtained with the Application Server running on a Windows XP, Pentium M 1.8GHz CPU, 1GB RAM machine and the DB2 server running on a Windows 2000, dual Xeon 2.6GHz CPU, 3GB RAM server.

Our implementation provides both a browser-based and a command line interface for defining resource characteristics and job requirements. For testing and performance studies, we have implemented a random resource and job generator, which is capable of generating a range of heterogeneous resources and jobs with different requirements. This generator takes as input an XML template file specifying a set of rules that are followed in generating the resource and job instances. For the experiments reported in the following, we use a template for defining four resource types each with 3 to 7 attributes. The actual attribute values are

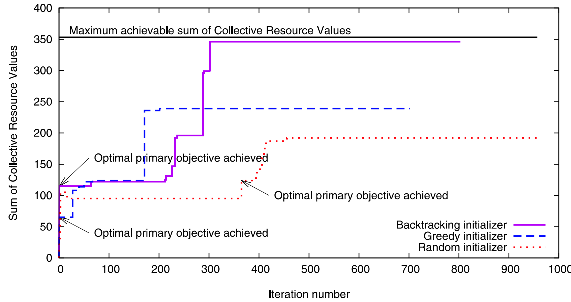

Figure 5. Effect of the initializers on the performance of the evolutionary resource matcher. The results presented are for a system with 20 jobs and 20 resources.

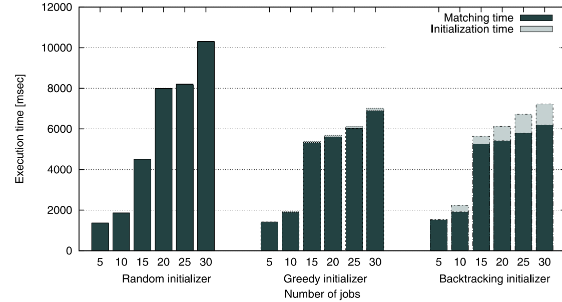

Figure 6. Effect of the initializers on the total execution time of the evolutionary algorithm as a function of the numbers of jobs in a batch. The results are for a system with 20 resources.

selected randomly from a predefined range or a probability distribution specified in the template. Similarly, the job description template allows specification of job priorities as a range or as a probability distribution. For the experiments reported here, job priorities were selected randomly from a uniform distribution over a range 1 to 20. Each job had dependencies on four resource types with consumption requirements and preferences specified on up to four resource attributes.

For the experiments, we set the solution population size in the EA implementation to 20. The optimizations were carried out using a multi-objective function. The primary objective was to maximize the sum of the priorities of the matched jobs and the secondary objective was to maximize the sum of the *Collective Resource Values (CRV)* of all the matchings in a solution. The Collective Resource Value of the matchings to a job is the product of the *Resource Preference Values (RPV)* for the assigned resources as specified by the job. Note that the a resource may have different Preference Values for different jobs. The primary objective has higher priority over the secondary objective meaning that when comparing two solutions the value of the secondary objective is taken into account only if the values of the primary objective are equal for these solutions. In all the experiments, the EA terminated if the best solution did not change in 500 consecutive iterations.

5.2. Experimental Results

In the first set of experiments, we compare the quality of the matchings when the EA is initialized with random, greedy, and backtracking algorithms (see Section 3.2). We made ten optimization runs on a system with 20 resources and 20 jobs. The average of the ten runs are shown in Figure 5, where we plot the sum of the CRVs of the matchings in the best solution found as a function of number of iterations. The initialization cost is included in the first iteration. For each run, we used a different random seed to initialize the EA but we kept the same set of resources and jobs, which were generated randomly as described in Section 5.1. Only the matcher initialized with the backtracking algorithm achieves a close-to-optimum value for the secondary objective. However, all three variants of the matcher find solutions that maximize the primary objective. Due to the space limitations, we do show the sum of the job priorities (primary objective) in the best solution as a function of the number of iterations. Instead, in Figure 5 we indicate the iteration number when the matcher achieved the maximal value for the primary objective function. Note that the optimal value for the primary objective function was achieved in the initialization phase itself when the greedy and the backtracking initializers were used.

In the second set of experiments, we measure the execution time of the EM as a function of the number of jobs in a batch. The execution times for different variants of the EM, again averaged over 10 runs, are shown in Figure 6. The execution time is the sum of the initialization time and the matching time. The random initialization has clearly the lowest overhead, which is close to zero even for larger batch sizes. The backtracking algorithm based initializer has the most overhead. Nevertheless, the results show that even with the overhead in the initialization phase, the total execution time for the matcher is significantly lower when greedy or backtracking initializers are used. Thus, with non-random initializers, the EA produces better quality matchings in lesser time. Clearly, proper solution initialization has a significant impact on the EM performance. We also measured the time overhead incurred in making the Web service call to the Matcher Interface and the time overhead of fetching the resource state information from the RSR while building the model. Both overheads were con-

stant for the range of jobs considered and for the size of the resource repository.

6. Related Work

In our previous work [11] we have addressed the problem of on-line resource matching using a linear programming based approach. In that work, the optimization ability of the linear programming matcher is limited to a single objective function at a time.

A certain class of evolutionary algorithms, the genetic algorithms [3], have been applied in the past to job shop scheduling [16] and resource allocation in highly available distributed systems [7], and other NP-complete problems [6]. They have also been explored in multi-objective optimization and their convergence has been analyzed [15, 2, 13]. However, none of these approaches were optimized for on-line resource matching in heterogeneous grid environments.

A combination of a genetic algorithm and a greedy optimizer has been studied in [8]. A design and evaluation of a genetic grid super-scheduler that assigns jobs to resources based on multiple criteria has been presented in [10]. The execution time of the super-scheduler, measured in tens of minutes for 32 jobs, excludes it, however, as an on-line service. The existing on-line job schedulers based on genetic algorithms, e.g., [14], are domain-specific and do not support the general class of jobs considered in this work.

The concept of combining features from evolutionary algorithms and simulated annealing has been described and evaluated in [9].

7. Conclusions

Efficient and accurate resource matching is crucial in grid environments. We have presented an Evolutionary Algorithm based matcher for matching heterogeneous grid resources with heterogeneous jobs taking into account the resource usage policies, the job requirements, and job preferences. The EM described here is capable of finding solutions that meet multi-objective function criteria. We introduce several alternative initialization methods and prove experimentally that appropriate initialization leads to significant improvements in the quality and matching performance. We also describe the design and implementation of a web service based on-line resource matching system. The performance results obtained with the EM service indicate that it is able to produce good quality resource matching results even at job arrival rates in excess of 200 jobs per minute.

References

[1] D. B. Fogel. *Evolutionary Computation: Toward a New Philosophy of Machine Intelligence*. IEEE Press, Piscataway, NJ, 1995.

[2] C. M. Fonseca and P. J. Fleming. An overview of evolutionary algorithms in multiobjective optimization. *Evolutionary Computation*, 1(3):1–16, 1995.

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

[4] J. J. Grefenstette. *Genetic Algorithms and Simulated Annealing*, chapter Incorporating problem specifi c knowledge in genetic algorithms. Morgan Kaufmann, 1987.

[5] O. H. Ibarra and C. E. Kim. Heuristic algorithms for scheduling independent tasks on nonidentical processors. *J. ACM*, 24(2):280–289, 1977.

[6] K. A. D. Jong and W. M. Spears. Using genetic algorithms to solve np-complete problems. In J. D. Schaffer, editor, *Proc. of the 3rd Int'l Conf. on Genetic Algorithms*, pages 124–132, San Mateo, CA, 1989. Morgan Kaufmann.

[7] K. Krishna and V. K. Naik. Application of evolutionary algorithms in controlling semiautonomous mission-critical distributed systems. In *Proc. of the 5th Joint Conf. on Information Sciences*, pages 1015–1018, 2000.

[8] W. B. Langdon. Scheduling planned maintenance of the national grid. In *Selected Papers from AISB Workshop on Evolutionary Computing*, pages 132–153, London, UK, 1995. Springer-Verlag.

[9] D. Levi. Hereboy: a fast evolutionary algorithm. In *In Proc. of the Second NASA/DoD Workshop on Evolvable Hardware*, pages 17–24, 2000.

[10] V. D. Martino. Sub optimal scheduling in a grid using genetic algorithms. In *Proc. of the 17th IEEE IPDPS'03*, 2003.

[11] V. K. Naik, C. Liu, L. Yang, and J. Wagner. Online resource matching in a heterogeneous grid environment. In *In Proc. of the 6th IEEE CCGrid'05*, Cardiff, UK, 2005.

[12] G. Rudolph. Convergence analysis of canonical genetic algorithms. *IEEE Trans. Neural Networks*, 5(1):96–101, 1994.

[13] G. Rudolph. On a multiobjective evolutionary algorithm and its convergence to the pareto set. In *Proc. of the IEEE Int'l Conf. on Evolutionary Computation(ICEC'98)*, pages 511–516, Piscataway NJ, 1998. IEEE Press.

[14] S. Song, Y.-K. Kwok, and K. Hwang. Security-driven heuristics and a fast genetic algorithm for trusted grid job scheduling. In *Proc. of the 19th IEEE IPDPS'05*, 2005.

[15] N. Srinivas and K. Deb. Multiobjective optimization using nondominated sorting in genetic algorithms. *Evolutionary Computation*, 2(3):221–248, 1994.

[16] F. Zhang, Y. F. Zhang, and A. Y. C. Nee. Using genetic algorithms in process planning for job shop machining. *IEEE Trans. Evolutionary Computation*, 1(4):278–289, 1997.