Elastic Grid Resource Provisioning with WoBinGO: A Parallel Framework for Genetic Algorithm Based Optimization

Milos Ivanovic[a,*], Visnja Simic[a], Boban Stojanovic[a], Ana Kaplarevic-Malisic[a], Branko Marovic[b]

[a] *Faculty of Science, University of Kragujevac, Radoja Domanovica 12, 34000 Kragujevac, Serbia*
[b] *School of Electrical Engineering, University of Belgrade, Bulevar kralja Aleksandra 73, 11120 Belgrade, Serbia*

Abstract

In this paper, we present the WoBinGO (**Wo**rk **Bin**der **G**enetic algorithm based **O**ptimization) framework for solving optimization problems over a Grid. It overcomes the shortcomings of earlier static pilot-job frameworks, by: (1) providing elastic resource provisioning thus avoiding unnecessary occupation of Grid resources; (2) providing friendliness towards other batching queue users thanks to adaptive allocation of jobs with limited lifetime. It hides the complexity of the underlying Grid environment, allowing the users to concentrate on the optimization problems. Theoretical analysis of possible speed-up is presented. An empirical study using an artificial problem, as well as a real-world calibration problem of a leakage model at the Visegrad power plant were performed. The obtained results show that despite WoBinGO's adaptive and frugal allocation of computing resources, it provides significant speed-up when dealing with problems that have computationally expensive evaluations. Moreover, the benchmarks were performed in order to estimate the influence of the limited job lifetime feature on the queuing time of other batching jobs, compared to a static pilot-job infrastructure.

Keywords: Grid computing, pilot-job infrastructure, Dynamic resource provisioning, Metaheuristics based optimization framework

*Corresponding address: Faculty of Science, University of Kragujevac, Radoja Domanovica 12, 34000 Kragujevac, Serbia. Tel.: +381 34 336223; fax: +381 34 335040.

Email addresses: mivanovic@kg.ac.rs (Milos Ivanovic), visnja@kg.ac.rs (Visnja Simic), bobi@kg.ac.rs (Boban Stojanovic), ana@kg.ac.rs (Ana Kaplarevic-Malisic), branko.marovic@rcub.bg.ac.rs (Branko Marovic)

1. Introduction

Grid computing [1] has emerged as an effective environment for the execution of parallel applications that require great computing power. Grid computing consists of a geographically distributed infrastructure gathering computer resources around the world in a transparent way. Users are provided access to enormous computing resources, and enabled to better meet the challenges of science and engineering. One of the most frequently encountered challenges in applied science and engineering, is optimization. Genetic algorithms (GAs) [2] have proven themselves as robust and powerful mechanisms when it comes to solving complex real-world optimization problems. GA is characterized by a large number of function evaluations. Due to the time-consuming fitness evaluation functions found in real-world problems, it may take days and months for the GA to find an acceptable solution. Speeding up the optimization process is achieved by parallelization of GA [3],[4],[5] which reduces the resolution times to reasonable levels. Grid computing environments provide the infrastructure for implementing parallel metaheuristics. There, researches face new difficulties associated with developing and deploying a Grid based application. The fact that Grid resources are distributed, heterogeneous and non-dedicated, makes writing parallel Grid-aware applications very challenging [6]. The development and execution of Grid applications requires considerable effort and expert knowledge. Understanding the basics of Grid computing and Grid middlewares is a time and energy consuming process for developers. Moreover, for each run of application on the Grid, one has to address the issues of Grid resource discovery and selection, Grid job preparation, submission, monitoring and termination which differ from one middleware to another. These differences among middlewares may limit or hinder portability between different Grid infrastructures. Aside from the complexity of the Grid infrastructures, certain limitations are also present, notably the need for users to wait, sometimes for a significant time until their requests for computing resources are processed and the lack of good support for interactive applications. The complexity involved in writing Grid-enabled applications averts researches from harnessing computational Grids by scientific applications. As Grids grow in size at an admirable rate and an increasing number of resources are put at Grid users' disposal, it is of utmost importance for the researchers to efficiently exploit computational Grids in order to solve real-world problems. In this context, tools for simplifying Grid application development, by hiding the complexity of Grid computing from the researchers, can significantly enhance Grids harnessing by scientific and engineering applications.

In this paper, we present the WoBinGO framework for solving optimization problems over heterogeneous resources, including HPC clusters and Globus-based Grids. Although it's possible to utilize diverse computing resources for solving optimization problems in parallel (multiple university clusters, Grid), having in mind the immense computing power offered by the Grid, we will restrict our discussion in this paper only to EGI (European Grid Initiative) deployment of WoBinGO. The framework was designed to meet the following

goals: (1) speeding up the optimization process by parallelization of GA over the Grid (2) relieving the researcher burden of obtaining Grid resources and dealing with various Grid middlewares; (3) enabling fast allocation of Grid jobs to avoid waiting until requests for computing resources are processed by Grid middleware; (4) providing flexible allocation of worker jobs in accordance with the dynamics of the users' requests, thus avoiding the unnecessary reservation of computing resources.

The framework is dedicated for parallel execution of single and multi-objective optimization using GA on the Grid. It uses a master-slave parallelization model and allows both: parallel evaluation of a population in GA and parallel execution of multiple instances of the parallel GA. As a novelty, this framework incorporates the Work Binder (WB) [7] which provides almost instant access to Grid resources and interactivity for client applications. Integration of WB into the framework enables the programmer to focus solely on the optimization problem without having to worry about specific details of Grid computing. Additionally, WB increases the utilization of the Grid infrastructure by offering automated elasticity in its occupancy, based on present and recent client behaviour. Furthermore, a single WB service is capable of serving multiple users with multiple GA instances, where for each instance of GA a population evaluation is also parallelized. Due to the multi-tier design, it is easily possible to replace master-slave parallelization model with hierarchical parallel GA with master-slave demes [8] or with PEGA (parallel cellular GA) [9], keeping all the other components intact. The framework also adheres to the standard Globus security mechanisms, including GSI and MyProxy.

With the master-slave parallelization model and WB, evaluation of individuals is separated from the rest of the algorithm and performed on Grid computing elements (CEs). This allows an objective function to be written in any compiled or script language, which makes our framework favourable for solving optimization problems in diverse areas of science and engineering. The framework has been developed as an effort to efficiently solve optimization problems from the field of hydrology, but can be used for any other optimization task suitable for GA treatment.

Benchmarks were carried out using EMI/UMD middleware [10] on the South-East European regional infrastructure in order to evaluate the usability and efficiency of the proposed framework. The obtained results show that the achieved speed-up is almost linear. Moreover, the benchmarks were performed in order to estimate the influence of the limited job lifetime feature on the queuing time of other batching jobs. Compared to a static pilot-job infrastructure, this waiting time was significantly reduced. Further details about pan-European Grid, aspects such as production infrastructure, the management tools and the operational services offered can be found in [11]. The process of building regional Grid infrastructure is thoroughly described in [12].

The rest of the paper is organized as follows: in Section 2, we review the related work. A description of the framework is given in Section 3. In Section 4, theoretical analysis of WoBinGO's speed-up along with the experimental results and discussion are given. A case study is presented in Section 5, followed by

concluding remarks in the last section.

2. Related Work

Grid oriented genetic algorithms (GOGAs, following the notation first introduced by [13]) have been used over the past years for solving different problems [14], [15], [16]. The research community has also proposed and implemented optimization frameworks with parallel metaheuristics included. Most of them have been using small, dedicated and homogeneous computing resources. Here, we will only discuss those that enable execution of parallel metaheuristics in Grid computing environments.

GridUFO is a service oriented optimization framework [17] that offers sharing of optimization algorithms and problems among GridUFO users and solving of optimization problems using an algorithm already registered with the framework. New algorithms and objective functions can be registered with the framework, but only C language code is acceptable. This is a huge limitation since the hydrocodes of our main interest are written in C#/.NET and Fortran. The authors report significant speed-up for the problems of a larger size, but they do not consider the time spent for scheduling the Grid job.

ParadisEO-CMW [18] is the framework for designing and deploying parallel metaheuristics on computational Grids, assembling together the ParadisEO [19] and MW [20] frameworks. Grid-enabling an application with MW involves the reimplementation of a number of virtual functions. The framework is only intended for Grids consisting of multiple Condor pools combined via flocking.

JG^{2}A [21] was created as an extension of JGA [22] to take advantage of Grid technologies, allowing population evaluation parallelization and parallelization of the GA parameter tuning experiments. JG^{2}A uses GT4 Grid middleware, but requires Condor as an underlying scheduler. This makes it inflexible because, in general, local resource managers other than Condor are used at different computing sites and these sites may be under a different administrative control, which makes it hard to enforce the deployment of Condor on all these sites.

Efficient hierarchical parallel GA framework using Grid computing (GE-HPGA) [23] hides the complexity of a Grid environment through the extended GridRPC API and a metascheduler for automatic resource discovery. It has a two level structure: at the first level, sub-populations are transferred onto remote computing clusters and subpopulation evolution is invocated using the Globus job submission protocol; at the second level, evaluation of the individuals is performed on cluster worker nodes (WNs) and may be realized using any local cluster scheduler. Empirical results using a benchmark problem and a realistic airfoil design problem show that speed-up can be attained as long as the bounds on fitness function cost, cluster size, and communication overheads are satisfied.

Some recent works utilize Hadoop computing infrastructure for the parallelization of GA [24],[25],[26],[27]. This is a novel and interesting approach but it requires dedicated physical or cloud based Hadoop cluster, which is not always available. In the mixed environments consisted of multiple university clusters and regional Grid infrastructure, our approach is a bit more flexible.

Researches have devoted great effort to facilitating the use of Grid infrastructure for scientific and engineering applications. The common approach is to overlay the existing middleware layer with a custom software which enables easier and more efficient use of Grid computational resources. Pilot-job systems have emerged as an effective solution for overlaying middleware. With pilot-based infrastructures, users' jobs are submitted to a centralized pool and pulled by pilots running on computing nodes. This late binding of user jobs and resources greatly improves the user experience, as it hides broken Grid resources and provides more accurate information about available resources. Condor has its pilot-job submission framework called GlideIn [28]. Falkon [29] is another pilot-job system. It provides a light high-throughput execution environment on top of the Globus GRAM service, and achieves remarkable scalability and work-dispatch efficiency. Other examples of pilot-job systems are Coasters [30], DIANE [31], and BigJob [32].

Much work has been done on parallelizing GAs and developing parallel metaheuristics frameworks which employ Grid computing resources. On the other hand, pilot-job systems have been developed to improve the utilization of production Grids, greatly reducing middleware overheads, by decoupling workload submission from resource assignment. However, to our knowledge, there is no framework for parallel metaheuristics that employs a pilot-job system dynamically using elastic resource provisioning. We argue that elastic resource provisioning can be as efficient as static pilot-job infrastructures used in previous frameworks, but with the key advantage of avoiding unnecessary occupation of Grid resources.

We therefore propose the parallel framework for optimization based on GA which employs the WB service. The main purpose of the WB is to quickly allocate new jobs for Grid users, and hide the complexity of the Grid infrastructure from the user. It is based on the use of pilot-jobs, but has certain advantages. Both, pilot-job infrastructures and WB start placeholder jobs asynchronously and match them with the actual work, but there are differences between these two approaches. In the pilot infrastructures, matching is done with the data describing some work to be done, while WB establishes association with a live client interacting with the job. With the WB approach, a job without a matching client can wait for some time in the pool of ready jobs, and pilot-jobs exit immediately if there is no work to be done. The communication between the client and worker can be observed, influenced, or filtered when using the WB, and the pilot-infrastructure only influences the job upon selection of a work item from the repository.

The key feature that WB provides, as a part of the WoBinGO framework, is its inherent frugality in resource consumption, provided by two distinct aspects: (1) the system adapts the number of placeholder jobs to the current workload, and (2) a defined maximum job lifetime prevents endless waiting of other users' jobs in the batching queues.

3. WoBinGO features

This section presents the key features of the WoBinGO framework, giving its advantages over other related solutions:

- Optimization using parallel GA can be performed over diverse computing resources, including Grid and HPC clusters.
- The complexity of the underlying Grid infrastructure is hidden from the user.
- Automatic adaptive allocation of jobs with limited lifetime which provides friendliness towards other batching queue users.
- A single WB service is capable of serving multiple users with multiple GA instances, sharing the same pool of ready jobs.
- Quick client-worker binding is enabled by elastic maintenance of the pool of ready jobs.
- Development of objective functions and optimization algorithms are independent, and can be performed by different developers.
- Objective function can be written in any compiled or script language supported by an underlying OS/runtime environment.
- Standard security features of GSI (Grid Security Infrastructure) and resource discovery using MDS (Monitoring and Discovery System) are employed.

The proposed framework is designed for parallel execution of single and multi-objective optimization using GA. WoBinGO employs a master-slave parallelization model, where evaluation of individuals is distributed to several slave nodes, while the master node executes the rest of the algorithm in sequential fashion. The drawback of the master-slave model is that the communication overhead can be higher than the benefits of parallel execution. However, for expensive optimization problems the master-slave model is a sufficiently good choice. Black-box fitness functions, as those found in engineering optimization problems (finite element, finite difference, meshfree methods), comprise a simulation run that takes several minutes to finish. Our framework is intended for solving these kinds of problems and, as can be seen in Section 4, it has accomplished remarkable results.

The framework's design is object oriented and includes separation among the algorithms, problem specifications and objective functions, so that different developers can independently write algorithms and objective functions. Currently, the framework incorporates simple GA for single-objective optimization and the multi-objective optimization algorithm NSGA-II [33]. The object-oriented approach of WoBinGO enables new algorithms to be integrated easily by extending the provided abstract classes. Objective function evaluation is separated

from the rest of the optimization algorithm and performed on the Grid clusters. This allows objective functions to be written in any language supported by the underlying OS or runtime environment. The authors have used objective functions written in C#, C++, Java, Fortran, Python and Matlab when solving various problems using the framework. Furthermore, all issues related to the Grid computing environment are concealed from an algorithm and the objective functions' developers. There is no need for "gridifying" a code that performs objective function evaluation, so the development time is reduced considerably. The only restrictions regarding the objective function interface are the following: (1) at the beginning, it has to read from the standard input the number showing how many parameters it will receive, and then all the parameters one by one (the term *parameter* in the context of GAs denotes one gene of the chromosome.); (2) at the end of the evaluation, it has to write to the standard output the number showing how many results it will return, and then all the results one by one (a result is an obtained objective function value of a chromosome).

A GA often requires a parameter tuning experiments in order to select the optimal values of genetic operators' parameters. Parameter values greatly determine the algorithm efficiency and whether it will find a near-optimal solution. Choosing the right parameters is a time-consuming process. WoBinGO speeds-up the parameter tuning process by allowing multiple independent optimization instances with different algorithm parameters. Furthermore, a single WB service is capable of serving multiple users with multiple GA instances, sharing the same pool of ready jobs [7].

The researchers who are using GOGAs are usually faced with manual discovery and selection of available Grid resources. Due to the large amount and dynamics of the Grid resources, this process is time-consuming and impractical. After the resources are found and selected, Grid job preparation, submission and monitoring is required. At this point, the users often have to wait a noticeable amount of time for the job submissions to be processed. By relying on the WB service, WoBinGO hides all issues related to the Grid computing environment from the users, allowing them to focus solely on the problem at hand. This is achieved by adopting the WB service.

WB is capable of utilizing Grid elastically in accordance with resource availability and the present workload. As a pilot-job based system, the WB manages the job pool with the goal of always having enough worker jobs in the pool for incoming clients, as well as minimizing stress on the Grid infrastructure. The job pool size may vary significantly, depending on the current load and dynamics of new user and job arrivals. If there are no active clients, WB is in the idle operational mode, and job pool size is maintained at a predefined minimum, leaving the Grid resources available for other users. When the number of active clients is above zero, the active mode of operation is used. In active mode two job refill strategies can be used: *regular* and *full throttle*. Regular refill strategy distributes the load fairly among CEs, keeping the number of ready jobs on each CE at the appropriate level by analysing its current status. The number of jobs that will be submitted to a CE depends on the following: (a) target pool size,

(b) current amount of ready jobs on a CE, (c) number of already submitted jobs that are expected to become ready, (d) number of busy jobs that will soon finish and become reusable. When the number of jobs that needs to be submitted is calculated, the WB service submits jobs gradually over time to compensate for the non-instant responsiveness of the CEs and to avoid batching system exposure to a sudden high load. The full throttle refill strategy is used when the number of ready jobs falls below a predefined threshold due to high clients' arrival frequency or saturation of some of the CEs. The WB service submits jobs instantly as soon as it detects that the total demand for jobs is greater than the sum of ready jobs and those that are expected to become ready in a short time. The WB service submits jobs instantly as soon as it detects that the desired total amount of jobs is greater than the actual amount of ready jobs (and the jobs that are expected to become ready in a relatively short time). In order to provide new ready jobs quickly, it picks the group of CEs with lowest response times and proportionally submits jobs to them. If the fastest CEs have reached their maximum pool sizes and there is a further need for refill, the amount of remaining refill jobs will be submitted to the rest of the CEs. The described adaptable job submission and pool management policy provide high availability of ready jobs and near-optimal usage of Grid resources.

Perhaps the most innovative feature not found in other similar frameworks is WoBinGO's friendliness towards other batching queue users, due to the worker jobs' limited lifetime. When that time is about to expire, despite there being more evaluation work to be done, the system removes the current job and submits an appropriate number of new jobs. This way, the other regular/Grid users have better chances because they are not affected by job placeholders keeping resources allocated without any actual work given.

The WoBinGO framework also takes security issues seriously into account. It employs standard Globus security features provided by GSI and MyProxy service. Each act of communication through WB has to be certified by a proper p12 certificate issued by an appropriate CA (Certification Authority).

To the best of our knowledge, no other previously developed distributed GA framework provides these characteristics in one place: efficient elastic resource provisioning, friendliness towards other batching queue users, multi-platform support, object-oriented design, clear separation between algorithms and objective functions, and inherent security.

3.1. Architecture

In this section, we present the architecture of the proposed framework.

The basic structure of the framework is illustrated in Fig. 1. The framework consists of the optimization master and the distributed evaluation system based on WB. The distributed evaluation system is composed of the evaluation pool and the WB subsystem. The master executes the main evolutionary loop and the distributed evaluation system takes care of the Grid execution of the evaluation processes. It should be noted that the evaluation pool is NOT the same entity as the pool of ready jobs created and maintained by WB itself.

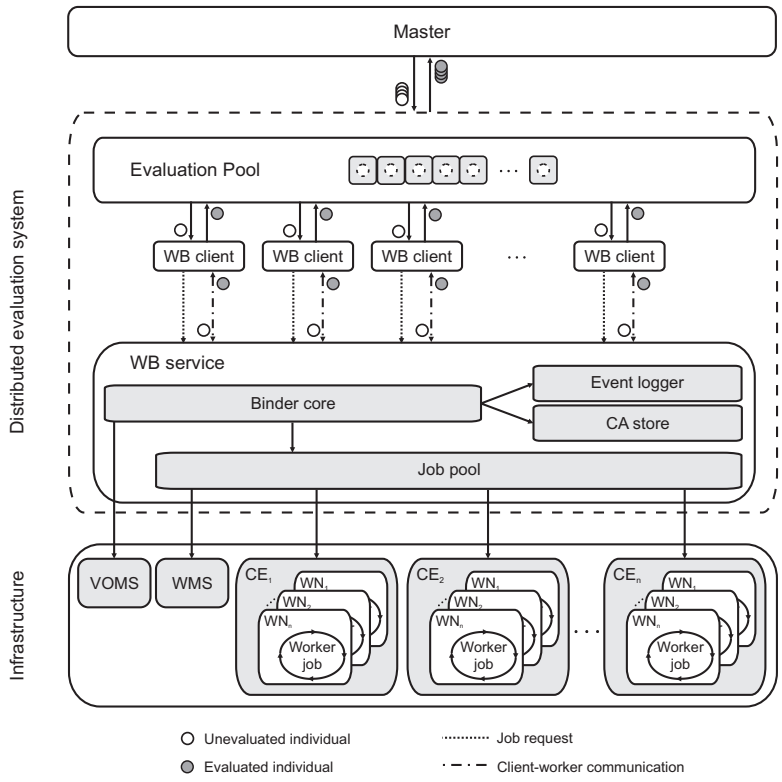


Fig. 1: WoBinGO architecture

The **master** performs the main evolutionary loop to the point when a generation has to be evaluated. At that moment, the master sends all individuals in a generation to the evaluation pool. After all individuals from a generation have been evaluated, and assigned with an objective function value, the master proceeds with the rest of the evolutionary algorithm until the stopping criteria is reached.

The **evaluation pool** acts as an intermediate layer between the master and the WB service. It provides asynchronous parallel evaluation of individuals from a generation. Each time a generation has to be evaluated, the evaluation pool receives individuals from the master and enqueues them. The evaluation pool invokes the WB client for each of the queued individuals. When an individual is evaluated, the evaluation pool receives the result back from a WB client and then assigns objective function value to the corresponding individual.

The **WB** environment consists of software components distributed in three tiers: the client, the worker, and WB service. The purpose of the WB service is to maintain the pool of ready worker jobs on the Grid and to bind them with

clients that request evaluation. It submits jobs to the Grid CEs in order to load enough worker jobs in the pool for incoming requests. The client establishes a connection to the WB service, and requests a worker. As soon as the client and worker are successfully coupled, the WB service acts as a proxy that relays traffic between them. For the purpose of distributed evaluation, the client sends an individual to a worker which computes fitness value and sends it back through WB proxy. After completing the evaluation, the worker reconnects to the WB service asking for more work within job time limits determined by WB configuration. The job lifetime cannot exceed the limit specified by the local Grid site administrator, obtained using MDS.

3.2. Workflow

In this section, we will outline the workflow of the framework. We will describe the steps necessary to perform any optimization with computationally intensive evaluations using the WoBinGO framework. Following stages depict the workflow when using framework in the case of fully automatic WB service deployment:

1. As the first step, a user has to supply a proper Grid certificate and three configuration files: a problem configuration file, an algorithm/objective function configuration file and the WB configuration file. The problem configuration file contains data regarding an optimization problem, like the number of decision variables and their ranges, and the number of objective functions. The algorithm/objective function configuration file includes the name of the algorithm that is going to be used and the location of the objective function evaluator. This file also contains data such as population size, mutation rate etc. In the WB configuration file, the URL and user credentials of the Grid UI (User Interface) is specified, along with the list of Grid CEs that are going to be used for job submission, and the maximum number of jobs allowed on each CE (denoted as p_{max}). Other WB parameters can also be specified, including various job timeouts, i.e. to reach ready state from submitted, to reach busy state from ready state, job lifetime, idle mode entrance limit etc.
2. According to the data found in the WB configuration file, the master uses templates to generate scripts and internal configuration files required by WB. The WB service will be automatically deployed on the node capable of submitting Grid jobs (UI host specified above). The master establishes connection with the UI over SFTP, and transfers two file packages and the deployment script. The first package includes WB itself, including generated scripts and configuration files. The executable for fitness evaluation and input files required for its execution (if provided) reside in the second package.
3. The master then invokes the deployment script whose first task is to create valid Globus proxy. The deployment script then stores the above mentioned fitness evaluator package to a number of Grid storage elements (SEs), creating replicas. It is worth noting that it is a common case to have

a complex fitness evaluator which runs a simulation model and requires a large amount of data. For example, in the case of hydrology problems requiring finite element computations, the evaluator package sizes ranged from 50MB up to 500MB. Sometimes, the evaluation package also has to carry a complete runtime environment in case it's not available locally (e.g. Mono framework, recent Python version, etc.). In order to minimize the time required to transfer data from SE to WNs, nearby replicas are created. To ensure that the closest replica has been transferred over to WN, the name of the nearby SE is obtained from MDS. According to [34], the replication also increases distributed application robustness.

4. Prior to the beginning of the evolutionary search, the master process starts the WB service. When the service is started, it submits jobs to CEs which are listed in the WB properties' file mentioned above. When a job arrives to a WN and transfers the evaluator package from the closest SE, the dispatcher is invoked. The dispatcher establishes the connection to the WB service which then changes the state of a corresponding job from *submitted* to *ready*. If a job and the WB service do not establish connection after a certain period of time, the job becomes *aged*. When a *ready* job couples with a client, its state is changed to *busy*. After the client session has finished, the job becomes *reusable*.
5. The master then finally begins the evolutionary process. Each time a generation has to be evaluated, the master sends individuals to the evaluation pool, where they are being queued (Fig. 2). The main thread of the algorithm remains halted until all individuals, sent to evaluation, are evaluated. For each solution to be evaluated, the evaluation pool starts the WB client in a separate thread. That way, evaluations can be processed asynchronously, without tying up the primary thread of the evaluation pool or delaying the processing of subsequent requests.
6. Upon invocation, each client tries to establish communication with the WB, and when successful, it sends its identification. Then, the WB obtains a ready worker job, which has earlier been registered as the one being ready to perform the evaluation. When the corresponding worker process is found, the client and worker become coupled. The WB service further acts as a proxy between client and worker, and the WB continues listening to the new requests. Since there is a vast number of solutions queued in the evaluation pool, the WB receives hundreds of new requests almost instantly. It automatically submits new worker jobs to the CEs in order to quickly fulfill these requests.
7. Once the client and worker have been matched by the WB, the client first sends the number of parameters needed for the evaluation, and then the parameters one by one, through the WB proxy, to the worker. The worker invokes the application that performs the evaluation with parameters received from the client. After the evaluation is done, first the number of results, and then the results themselves are sent back, from the evaluation application to the worker, and further through the WB proxy, finally reaching the client's `stdin`. The client returns the results of the evalua-

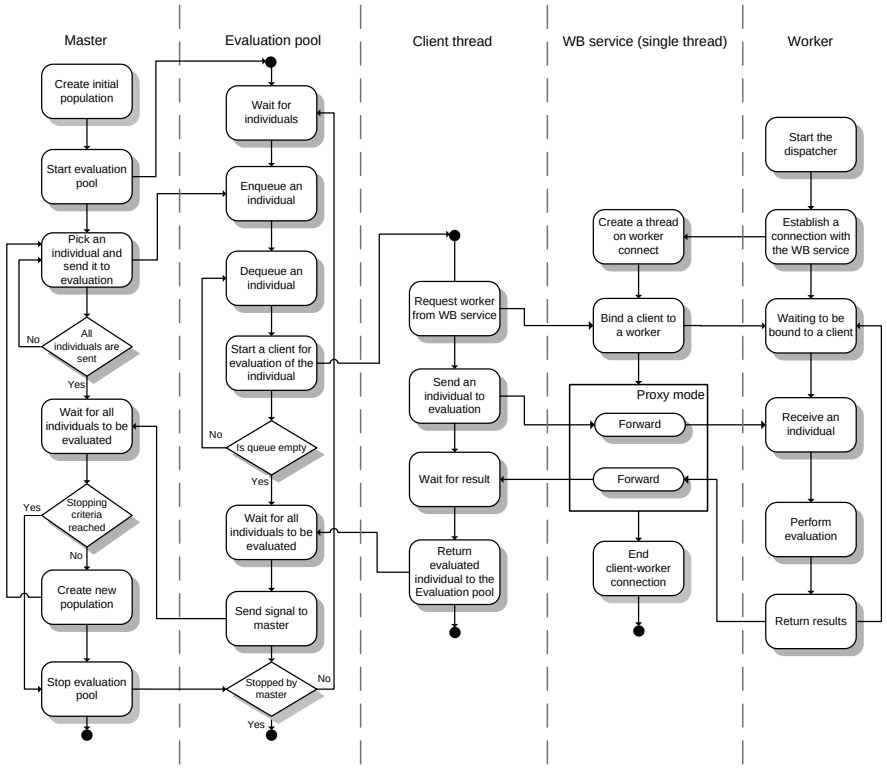


Fig. 2: Workflow of the WoBinGO framework (for the sake of brevity, deployment stages (1-4) are excluded)

tion and the evaluation pool assigns them to the corresponding individual. Once all the clients return results and there are no more individuals in the queue, the evaluation pool sends a signal to the main thread that all evaluations are over and that the master is allowed to continue with the rest of the evolution algorithm. If any client raises exception and the evaluation of the individual fails, the same individual is sent for the evaluation again.

The worker dispatcher will be restarted upon successfully performed evaluation in order to enable job reusability on the WN. Furthermore, each time the evaluation is completed, the worker dispatcher checks if the specified job time limit is exceeded in order to allow other local batching system jobs to get into the queue.

All jobs are submitted using standard Globus proxy, but the normal user proxy lifetime of 12-24 hours is usually not long enough to complete optimization when dealing with real-world problems. As a solution, the user proxy is retrieved from the MyProxy service on a regular basis.

4. Results and Discussion

With the WoBinGO framework, the goals of near-optimal usage of Grid resources and high availability of ready jobs are achieved by implementing an adaptive job submission and pool management policy. These are, however, fulfilled without undue sacrifice of Grid resources, unlike previous solutions involving static pilot-job infrastructures. The objective of this section is to determine whether such an approach affects the performance of the system and to what extent. In order to achieve this objective, we used both a theoretical and empirical approach.

4.1. Theoretical Analysis

The speed-up of the WoBinGO system can be computed using the following formula [35]:

$$S = \frac{T_{ser}}{T_{par}} \tag{1}$$

where T_{ser} denotes the time needed to evolve a population of individuals in serial and T_{par} denotes the time needed to evolve the same population in parallel. For the population of size n, the time T_{ser} is given by:

$$T_{ser} = \lambda(n) + \gamma(n) \tag{2}$$

where $\lambda(n)$ denotes the time required for the execution of the sequential part of the algorithm, and $\gamma(n)$ denotes the time required for the execution of the part of the algorithm that can be parallelized. The sequential part of the algorithm includes the genetic operators of selection, crossover and mutation performed by the master, while the evaluation of individuals is done in parallel over a number of workers. If the number of worker processors is p, then the time T_{par} is given by:

$$T_{par} = \lambda(n) + \frac{\gamma(n)}{p} + O(n, p) \tag{3}$$

where $O(n, p)$ denotes the communication overhead. With the WoBinGO framework, the communication overhead is comprised of the overhead induced by client-worker communication over WB, which is required for establishing connection and exchanging individuals and evaluation results, and the distinctive WoBinGO's overhead. In previous distributed optimization frameworks [17], [23], the number of available workers is static, which provides the possibility to group work intended for each CPU in a deterministic fashion. Unlike these frameworks, WoBinGO creates a separate client for each unevaluated individual which keeps asking the WB service for a ready worker until it gets one. This overhead is greatly reduced by using an appropriate job submission policy providing enough ready jobs in the pool; however, it still exists. As will be shown later, the overhead is maintained within a reasonable range for long enough evaluations.

According to the WB's elastic resource provisioning, the number of workers changes over time. If $\bar{p}$ is the average number of processors that perform evaluation of n individuals, and t_{eval} is the average time needed to evaluate one individual, then T_{par} can be expressed as follows:

$$T_{par} = \lambda(n) + \frac{nt_{eval}}{\bar{p}} + O(n, \bar{p}) \tag{4}$$

If the time required for solving the same optimization problem in serial is expressed accordingly as $T_{ser} = \lambda(n) + nt_{eval}$, then the speed-up is:

$$S = \frac{\lambda(n) + nt_{eval}}{\lambda(n) + \frac{nt_{eval}}{\bar{p}} + O(n, \bar{p})} \tag{5}$$

According to Amdahl's law [35], a theoretical bound on the maximum speed-up that can be achieved by the parallel algorithm is obtained by assuming $O(n, \bar{p}) = 0$ in Eq. (5):

$$S_{max} = \frac{\lambda(n) + nt_{eval}}{\lambda(n) + \frac{nt_{eval}}{\bar{p}}} \tag{6}$$

In order to achieve any speed-up, the following must hold:

$$\begin{aligned} T_{ser} &> T_{par} \\ t_{eval} &> \frac{\bar{p}}{n(\bar{p}-1)} \cdot O(n, \bar{p}) \end{aligned} \tag{7}$$

For a large number of processors, the equation (7) can be approximated as:

$$t_{eval} > \frac{1}{n} \cdot O(n, \bar{p}). \tag{8}$$

4.2. Empirical results

In this section, we present an empirical study of the WoBinGO framework. The first aim was to determine its performance in terms of infrastructure utilization, achieved speed-up and inherent overhead, while varying t_{eval}. The second aim was to obtain the framework's speed-up when t_{eval} is constant. The third aim was to estimate to which extent WoBinGO's characteristic to limit pool jobs lifetime assists other batching queue users.

Using a simple GA with real-encoded chromosomes we solved the artificial test problem. The objective function was a dummy function that does nothing except receive individuals, sleep for t_{eval} seconds, and return random fitness value to the master. Because we have aimed to determine communication overhead, it did not matter whether the workers were performing calculations or simply sleeping, it only mattered how long it took them to return their response.

The benchmark was carried out using a single Grid site, in order to put aside all internetworking effects and explore theoretical limits of the WB framework. The site AEGIS04-KG consists of 6 nodes, each equipped with 2 AMD 16-core CPUs and 96GB RAM, totalling 192 processors. The nodes are interconnected

with standard gigabit Ethernet, OS is Scientific Linux 6.4 x86_64 with a PBS Torque batching environment and Maui scheduler. As AEGIS04-KG is a part of EGI infrastructure, the jobs were not submitted by using the batching system directly, but by employing EMI/UMD services, including WMS (Workload Management System), which is not collocated with the site.

The population size during all experiments was set to 500 individuals. The simple GA was executed for 10 generations and the results reported are the average of 10 runs. The estimated average time required for the sequential part of the algorithm (denoted $\lambda(n)$ in Eq. (4)) is 200 ms.

Table 1: Empirical results obtained by running an artificial problem over WoBinGO using a simple GA (time is shown in seconds).

t_{eval}	$\bar{p}$	T_{ser}	T_{par}	T_{ser}/T_{par}	$O(n,\bar{p})$	$O(n,\bar{p})/T_{par}[\%]$
0.5	25	2.5E+03	4.13E+02	6.05	3.13E+02	76
1	49	5.0E+03	4.09E+02	12.23	3.07E+02	75
2	48	1.0E+04	4.37E+02	22.86	2.29E+02	52
5	74	2.5E+04	5.17E+02	48.4	1.78E+02	35
10	80	5.0E+04	7.40E+02	67.52	1.15E+02	16
30	91	1.5E+05	1.76E+03	85.21	1.12E+02	6
60	95	3.0E+05	3.25E+03	92.22	9.51E+01	3
120	96	6.0E+05	6.35E+03	94.55	1.17E+02	2

Table 1 shows the obtained results. The average number of processors $\bar{p}$ that have been used to evaluate individuals was determined experimentally. The WoBinGO framework was configured so that the maximum allowed number of worker jobs (p_{max}) is 100. However $\bar{p}$ varies in accordance with the time t_{eval} taken by a single fitness value calculation. With short evaluations, the workers quickly become reusable so there is rarely the need to submit new jobs. On the other hand, with longer evaluations, a significant amount of time passes until the workers become reusable. Therefore, new jobs must be submitted more often. Consequently, $\bar{p}$ has larger values for longer fitness evaluations. T_{ser} is the time needed to evolve a population of individuals on a single processor and T_{par} is the time needed to evolve the same population in parallel, using $\bar{p}$ processors. T_{ser}/T_{par} is the speed-up achieved by the WoBinGO framework, and $O(n,\bar{p})$ is the communication overhead. The value of $O(n,\bar{p})$ was determined by substituting T_{par}, $\lambda(n)$, t_{eval} and $\bar{p}$ into Eq. (4).

As can be observed from Table 1, the overheads are quite significant for smaller problems. This is due to the fact that for problems with short evaluations the frequency of client and worker requests addressed to the WB service is greater than with longer evaluations. Higher frequency implies longer waiting for the requests' fulfillment.

For the problems with the more time-consuming fitness evaluation considerable speed-ups can be achieved, as the overhead becomes almost constant for $t_{eval} > 5s$.

Therefore, the WoBinGO framework is more suitable for the problems of relatively large sizes. In order to confirm this result we further experimented with an artificial test problem where $t_{eval} = 60s$ and p_{max} was varied through

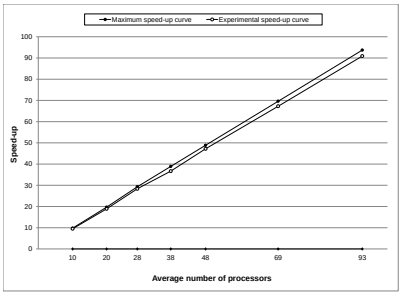


Fig. 3: Speed-up curve obtained by running an artificial problem over WoBinGO

the following values: 10, 20, 30, 40, 50, 75 and 100. Fig. 3 contains two speed-up curves: the maximum speed-up curve and the experimental speed-up curve. The maximum speed-up curve is obtained from Eq. (6) by substituting the following parameters: $n = 5000(= 500 \cdot 10)$, $t_{eval} = 60s$, $\lambda(n) = 200ms$ and $\bar{p}$, which was determined experimentally. The experimental curve is the average speed-up for 10 independent runs. As can be seen from Fig. 3, the theoretically predicted speed-up fits nicely with the observed experimental speed-up. The difference between these speed-ups comes from the communication overhead, which is excluded from Eq. (6). With an increasing number of workers, the communication overhead grows linearly, as expected. The experimentally obtained speed-up curve shows that the framework is suitable for problems with computationally expensive evaluations which are common in real-world applications. Furthermore, considering the condition (7), we have experimentally obtained that there is no point in using the WoBinGO framework for $t_{eval} < 0.1s$.

If hierarchical parallel GA with master-slave demes or PEGA are used instead of master-slave model at the top level (Fig. 1), the same speed-up results as shown in Fig. 3 will still apply, within experimental error. This statement is valid because each deme's evaluation demands will be served by a single WorkBinder instance, while GA overhead can be neglected due to relatively expensive fitness evaluation.

The final goal was to benchmark the feature that distinguishes WoBinGO framework from the previous static pilot job solutions - limited pool job lifetime. This specific feature enables other batching queue jobs to reach running state with less delay. In order to estimate that delay, the artificial test problem was used, as previously. Again, t_{eval} was set to $60s$, population size was 500 and the maximum number of processors was 150. Furthermore, the artificial load of 18 $jobs/h$ was also created, with job submission occurrence that fits the Poisson distribution. The average number of processors was 1.3 (modeled using ratio between 1-processor and 4-processor jobs of 9:1), while the duration of these artificial jobs was 44 minutes on average, respecting Gamma distribution with the shape factor of 0.344 and the scale factor of 7680 (Fig. 4). For the

sake of reproducibility, all these values were taken as an approximation of the real AEGIS04-KG load for the period of two years in production. Additionally, AEGIS04-KG scheduler was set to keep the sum of processors held by WoBinGO and the processors held by the artificial job submitter to 150. We measured how the average waiting time of a job belonging to the artificial load depends on WoBinGO jobs' lifetime. Each experimental point resulted from $12h$ run. The obtained results are shown in Fig. 5. It can be noticed that for shorter lifetimes of WoBinGO jobs, the jobs belonging to the artificial load quickly reach running state. The average waiting time increases for longer lifetimes of WoBinGO jobs, but does not exceed one hour, even with WoBinGO jobs' lifetime of two and a half hours. Considering the fact that with the static pilot job infrastructure, the jobs belonging to the artificial load would certainly have to wait in the batching queue till the end of the optimization run (12 hours), shown results are quite remarkable. The percentage ratio to the waiting time on the static pilot job infrastructure is shown on the secondary ordinate axis on the right. Even for very long WoBinGO jobs' lifetimes, the waiting time of the artificially loaded jobs is below 8 % of the time they would spend in the batching queue if a static pilot job infrastructure was used instead of WoBinGO framework.

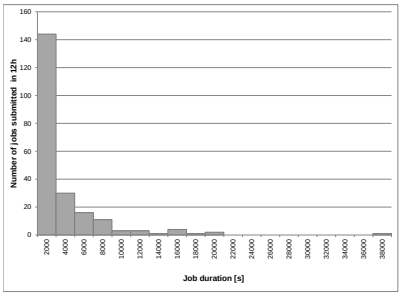


Fig. 4: Duration distribution of the jobs belonging to the artificial load

5. Case study

In this section, we will evaluate WoBinGO's performance when it comes to solving a complex real-world engineering problem. Multi-objective calibration of a leakage model at the Visegrad power plant (Republic of Srpska, Bosnia and Herzegovina) was performed and the achieved results are presented hereafter.

An initial hydraulic model of karst groundwater flow was created based on previous geological investigations. It consists of a 1D network of conduits that represent the hydraulic equivalent of the karst under the Visegrad dam. Using a finite element solver for hydraulic calculations (written in C++), simulation of the water flow through the model for the given boundary conditions (upstream and downstream elevation) was performed. In order to estimate underground

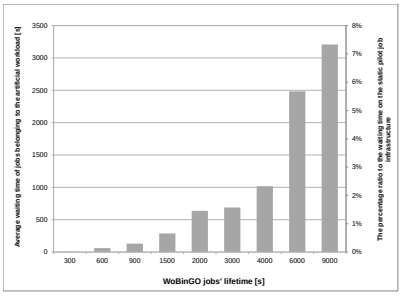


Fig. 5: The average waiting time of a job belonging to the artificial load. The percentage ratio to the waiting time on the static pilot job infrastructure is shown on secondary ordinate axis.

karst configuration and to calibrate the model, a number of measurements and experiments using various methods were carried out. A piezometric map was obtained using piezometers placed in the drill holes, while the flow velocities were measured at sinks and springs. In order to determine the main flow paths and dynamics, a number of experiments of salt and paint transport through the system were performed. Simulation of the salt and paint transport was executed using a finite difference solver for fluid mass transfer modelling (written in C#/.NET). The aim of the calibration was the determination of 1D conductor parameters, so that the described multi-model gives the best possible match to hydraulic and transport measurements and experiments.

Unknowns in the calibration process were the equivalent radii and conductivities of 1D elements. The objective functions were to minimize: (1) the root mean square error (RMSE) of the observed *vs.* calculated piezometric levels, (2) the RMSE of the observed *vs.* calculated velocities at sinks and springs, and (3) the difference between the calculated and observed time-series that represent salt and color concentrations at certain locations. Since one of the solvers used within the process of calibration was implemented in .NET, the evaluation package had to contain the complete Mono framework which was not available locally on the worker nodes.

For solving the leakage model calibration problem we used NSGA-II algorithm [33]. The following set of algorithm parameters was adopted: a population size of 500 individuals, a maximum of 500 generations, a simulated binary crossover with a probability of 0.9 and a distribution index of 20, and polynomial mutation with a distribution index of 20 and probability of $1/l$, where l is the chromosome length. All results are taken from 10 independent runs.

The experiments were performed in a Grid environment consisting of the three clusters represented by the corresponding CEs. Their characteristics are summarized in Table 2. The last column shows the average time of each computing cluster for evaluating the objective functions' values of an individual. Although the difference in computational efforts of the included clusters was

Table 2: Characteristics of clusters within research testbed

Site	**Nodes**	**CPU Type**	**RAM/Node**	**OS/LRMS**	$\mathbf{t_{eval}(s)}$
AEGIS04-KG	5	2x16 core Opt. 6276@2.3GHz	96 GB	SL6.4 x86_64 PBS Torque	68.12
AEGIS01-PHY-SCL	88	2x4 core Xeon E5345@2.33GHz	8 GB	SL6.4 x86_64 PBS Torque	66.65
AEGIS03-ELEF-LEDA	8	2x4 core Xeon E5420@2.5GHz	4 GB	SL6.4 x86_64 PBS Torque	65.77

not very significant, it was taken into account when T_{ser} was calculated. Hence, the following formula was used:

$$T_{ser} = \lambda(n) + n \sum_{i=1}^{3} \frac{\bar{p}_i}{\bar{p}} t^i_{eval} \tag{9}$$

Where: n is the population size; t^i_{eval} denotes the average time needed to evaluate the objective functions' values of an individual by the ith computing cluster; $\frac{\bar{p}_i}{\bar{p}}$ is the share of the average number of processors that are performing the evaluations on the ith cluster within the total average number of processors used by all three clusters.

Table 3 summarizes the obtained results. Due to WB's elastic behaviour, the value of T_{ser} changes in accordance to the Eq. (9). T_{par} is the experimentally obtained wall-clock time needed to evolve 500 generations of 500 individuals in parallel. Fig. 6 provides visual representation of the speed-up achieved when solving the considered real-world problem. Additionally, the wall-clock time is shown.

Table 3: Empirical results obtained by running a real-world calibration problem of a leakage model at the Visegrad power plant over WoBinGO using an NSGA-II algorithm (time is shown in seconds).

cluster1.csk.kg.ac.rs	cream.ipb.ac.rs	grid01.elfak.ni.ac.rs	$\bar{p}$	T_{ser}	T_{par}	T_{ser}/T_{par}
$\bar{p}_1$	$\bar{p}_2$	$\bar{p}_3$				
8	10	9	27	1.67E+07	7.25E+05	23.02
17	18	19	54	1.67E+07	3.35E+05	49.91
20	23	23	66	1.67E+07	2.72E+05	61.35
29	35	32	96	1.67E+07	1.86E+05	89.90
41	43	32	117	1.67E+07	1.48E+05	113.18
58	88	8	154	1.68E+07	1.11E+05	150.87
78	119	22	219	1.68E+07	7.87E+04	213.16
123	244	59	426	1.67E+07	4.18E+04	400.37

The main advantage of the WoBinGO framework is the ability to elastically allocate worker jobs on the computing Grid according to clients' requests. Infrastructure utilization can be further analysed if we consider Fig. 7. The number of processors per cluster performing the evaluation tasks through 10 generations is shown in Fig. 7a. The WoBinGO framework was configured to keep the sum of busy, ready, and submitted worker jobs at 50 per each cluster.

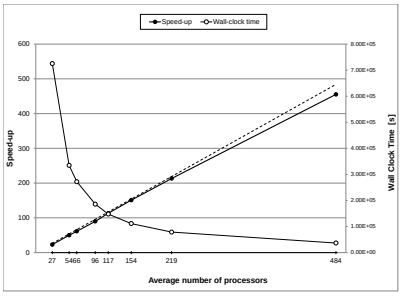
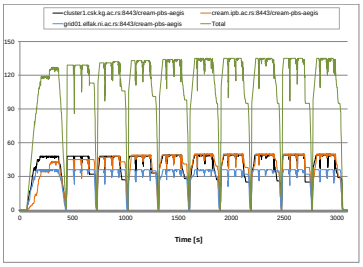
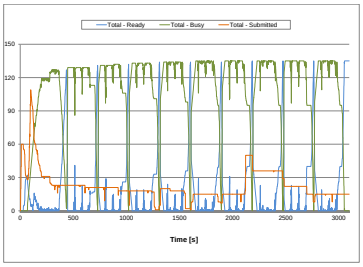


Fig. 6: The average wall-clock time and speed-up of the calibration problem of the leakage model at the Visegrad power plant over WoBinGO

The total number of ready, busy and submitted jobs for the experimental setup described above is shown in Fig. 7b.





(a) Number of busy processors per each cluster

(b) Total number of ready, busy and submitted jobs for all three clusters

Fig. 7: Computational infrastructure utilization

While there are no clients' requests, WoBinGO keeps the number of ready jobs at the predefined minimum, utilizing infrastructure resources in an efficient manner. At the moment the clients start addressing the WB service with evaluation requests, a number of new jobs are instantly submitted, soon becoming busy fulfilling clients' demands.

While evaluating a generation, the number of busy jobs stays high. At the end of the first generation, for a very short time, clients' job requests drop to zero and so does the number of busy jobs. These worker jobs immediately reach ready state, being soon employed for the evaluation of the next generation. It can also be noticed that the number of submitted jobs is non-zero throughout the entire run. This is because WoBinGO attempts to reach the preferred load on each cluster (50 workers), but some clusters do not respond due to their occupancy and/or configuration.

According to the presented case study results, it is obvious that despite WoBinGO's elastic provisioning of computing resources, it provides significant speed-up when dealing with problems that have computationally expensive evaluations.

6. Conclusion and future work

In this paper, we have presented the WoBinGO framework for solving large optimization problems over Grid or HPC clusters. The framework provides a novel, elastic approach in utilizing computing resources in accordance with the dynamics of the users' requests. Unnecessary reservation of computing resources is avoided through flexible allocation and limited lifetime worker jobs. WoBinGO conceals the complexity of the underlying computing infrastructure from the user, relieving him/her of the burden of obtaining computing resources and dealing with various middlewares.

To evaluate the efficiency of the framework, we have conducted a theoretical analysis of the maximum achievable speed-up, as well as obtained the practical conditions that have to be fulfilled in order to achieve useful speed-up. Empirical studies using a dummy problem have been performed in order to determine WoBinGO's performance in terms of infrastructure utilization, the achieved speed-up and inherent overhead. The results show that we can obtain considerable speed-ups for the problems of large sizes. What distinguishes our framework the most is the significant reduction of the waiting time of the artificially loaded jobs that compete for the computing resources with WoBinGO generated jobs. Compared to a static pilot-job infrastructure, WoBinGO enables other batching jobs to reach running state more than 10 times faster.

The case study concerning the leakage model calibration further confirms that the framework is suitable for solving optimization problems with computationally expensive evaluations which are common in real-world applications.

In the future, the authors will improve the framework in order to employ its inherent elasticity to independently manage cloud instances according to the system load.

Acknowledgements

Part of this research is supported by The Ministry of Science in Serbia, Grants III41007, OI174028, TR37013, III44010, and TR14005, and FP7 ICT-2007-2-5.3 (224297) ARTreat project.

References

[1] I. Foster, C. Kesselman, The Grid: Blueprint for a New Computing Infrastructure, Morgan Kauffman, San Francisco, 1999.

[2] D. E. Goldberg, Genetic algorithms in search, optimization, and machine learning, Addison Wesley, 1989.

[3] A. Munawar, M. Wahib, M. Munetomo, K. Akama, A survey: Genetic algorithms and the fast evolving world of parallel computing, in: 10th IEEE International Conference on High Performance Computing and Communications (HPCC '08.), 2008, pp. 897–902.

[4] E. Alba, M. Tomassini, Parallelism and evolutionary algorithms, Evolutionary Computation, IEEE Transactions on 6 (2002) 443–462.

[5] E. Alba, G. Luque, S. Nesmachnow, Parallel metaheuristics: recent advances and new trends, International Transactions in Operational Research 20 (2013) 1–48.

[6] I. Foster, Designing and building parallel programs, Addison-Wesley Reading, 1995.

[7] B. Marović, M. Potočnik, B. Čukanović, Multi-application bag of jobs for interactive and on-demand computing, Scalable Computing: Practice and Experience 10 (2001).

[8] E. Cant-Paz, D. E. Goldberg, Efficient parallel genetic algorithms: theory and practice, Computer Methods in Applied Mechanics and Engineering 186 (2000) 221–238.

[9] B. Dorronsoro, D. Arias, F. Luna, A. J. Nebro, E. Alba, A grid-based hybrid cellular genetic algorithm for very large scale instances of the CVRP, in: High Performance Computing & Simulation Conference (HPCS), 2007, pp. 759–765.

[10] B. Stephen, C. Simone, L. Elisa, L. Maarten, M. L. Patricia, M. Vincenzo, N. Christopher, S. Roberto, S. Andrea, gLite 3.2 user guide, `https://edms.cern.ch/file/722398//gLite-3-UserGuide.pdf`, 2009.

[11] T. Ferrari, L. Gaido, Resources and services of the egee production infrastructure, J. Grid Comput. 9 (2011) 119–133. URL: `http://dx.doi.org/10.1007/s10723-011-9184-1`. doi:10.1007/s10723-011-9184-1.

[12] A. Balaz, O. Prnjat, D. Vudragovic, V. Slavnic, I. Liabotis, E. I. Atanassov, B. Jakimovski, M. Savic, Development of grid e-infrastructure in south-eastern europe, CoRR abs/1112.4303 (2011).

[13] H. Imade, R. Morishita, I. Ono, N. Ono, M. Okamoto, A grid-oriented genetic algorithm for estimating genetic networks by s-systems, in: SICE 2003 Annual Conference, volume 3, 2003, pp. 2750–2755.

[14] J. Herrera, E. Huedo, R. S. Montero, I. M. Llorente, A grid-oriented genetic algorithm, in: Advances in Grid Computing-EGC 2005, Springer, 2005, pp. 315–322.

[15] H. Imade, R. Morishita, I. Ono, N. Ono, M. Okamoto, A grid-oriented genetic algorithm framework for bioinformatics, New Generation Computing 22 (2004) 177–186.

[16] A. J. Nebro, G. Luque, F. Luna, E. Alba, Dna fragment assembly using a grid-based genetic algorithm, Computers & Operations Research 35 (2008) 2776–2790.

[17] A. Munawar, M. Wahib, M. Munetomo, K. Akama, The design, usage, and performance of GridUFO: A grid based unified framework for optimization, Future Generation Computer Systems 26 (2010) 633–644.

[18] N. Melab, S. Cahon, E.-G. Talbi, Grid computing for parallel bioinspired algorithms, Journal of Parallel and Distributed Computing 66 (2006) 1052–1061.

[19] S. Cahon, N. Melab, E.-G. Talbi, Building with paradisEO reusable parallel and distributed evolutionary algorithms, Parallel Computing 30 (2004) 677–697.

[20] J.-P. Goux, S. Kulkarni, J. Linderoth, M. Yoder, An enabling framework for master-worker applications on the computational grid, in: The Ninth International Symposium on High-Performance Distributed Computing, 2000, pp. 43–50.

[21] A. Bernal, M. A. Ramirez, H. Castro, J. L. Walteros, A. L. Medaglia, JG^{2}A: A grid-enabled object-oriented framework for developing genetic algorithms, in: Proceedings of the Systems and Information Engineering Design Symposium (SIEDS'09.), 2009, pp. 67–72.

[22] A. Medaglia, E. Gutiérrez, JGA: An object-oriented framework for rapid development of genetic algorithms, in: J.-P. Rennard (Ed.), Handbook of Research on Nature Inspired Computing for Economics and Management. IDEA Publishing, Hershey, 2006, pp. 608–624.

[23] D. Lim, Y.-S. Ong, Y. Jin, B. Sendhoff, B.-S. Lee, Efficient hierarchical parallel genetic algorithms using grid computing, Future Generation Computer Systems 23 (2007) 658 – 670.

[24] G. M. Li, W. H. Zeng, J. F. Zhao, M. Liu, Master-slave parallel genetic algorithm based on mapreduce using cloud computing, Applied Mechanics and Materials 121 (2012) 4023–4027.

[25] L. Di Geronimo, F. Ferrucci, A. Murolo, F. Sarro, A parallel genetic algorithm based on hadoop mapreduce for the automatic generation of junit test suites, in: Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on, IEEE, 2012, pp. 785–793.

[26] F. Ferrucci, T. M. Kechadi, P. Salza, F. Sarro, A framework for genetic algorithms based on hadoop, arXiv preprint arXiv:1312.0086v2 (2013).

[27] A. H. Narayanan, U. Krishnakumar, M. Judy, An enhanced mapreduce framework for solving protein folding problem using a parallel genetic algorithm, in: ICT and Critical Infrastructure: Proceedings of the 48th Annual Convention of Computer Society of India-Vol I, Springer, 2014, pp. 241–250.

[28] I. Sfiligoi, glideinWMS - a generic pilot-based workload management system, in: Journal of Physics: Conference Series, volume 119, IOP Publishing, 2008.

[29] I. Raicu, Y. Zhao, C. Dumitrescu, I. Foster, M. Wilde, Falkon: a Fast and Light-weight tasK executiON framework, in: Proceedings of the 2007 ACM/IEEE conference on Supercomputing, 2007.

[30] M. Hategan, J. Wozniak, K. Maheshwari, Coasters: uniform resource provisioning and access for clouds and grids, in: Fourth IEEE International Conference on Utility and Cloud Computing (UCC), 2011, pp. 114–121.

[31] J. T. Moscicki, Diane-distributed analysis environment for grid-enabled simulation and analysis of physics data, in: IEEE Nuclear Science Symposium Conference Record, volume 3, 2003, pp. 1617–1620.

[32] A. Luckow, L. Lacinski, S. Jha, SAGA BigJob: An extensible and interoperable pilot-job abstraction for distributed applications and systems, in: 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing (CCGrid), 2010, pp. 135–144.

[33] K. Deb, A. Pratap, S. Agarwal, T. Meyarivan, A fast and elitist multiobjective genetic algorithm: NSGA-II, IEEE Transactions on Evolutionary Computation 6 (2002) 182–197.

[34] L. Guy, P. Kunszt, E. Laure, H. Stockinger, K. Stockinger, Replica management in data grids, in: Global Grid Forum, volume 5, 2002, pp. 278–280.

[35] J. Q. Michael, Parallel programming in C with MPI and OpenMP, Dubuque, IA: McGraw-Hill (2004).