J Sched (2015) 18:345–357
DOI 10.1007/s10951-014-0382-0

Scheduling parallel batch jobs in grids with evolutionary metaheuristics

Piotr Switalski · Franciszek Seredynski

Received: 7 December 2013 / Accepted: 25 April 2014 / Published online: 20 May 2014

Abstract In this paper we propose an efficient offline job scheduling algorithm working in a grid environment that is based on a relatively new evolutionary metaheuristic called generalized extremal optimization (GEO). We compare our experimental results with those obtained using a very popular evolutionary metaheuristic, the genetic algorithm (GA). The scheduling algorithm implies two-stage scheduling. In the first stage, the algorithm allocates jobs to suitable machines of a grid; GEO/GA is used for this purpose. In the second stage, jobs are independently scheduled on each machine; this is performed with a variant of a list scheduling algorithm. Both GEO and GA belong to the class of evolutionary algorithms, but GEO is much simpler and requires the tuning of only one parameter, whereas GA requires the tuning of several parameters. The results of the experimental study show that GEO, despite its simplicity, outperforms the GA in a whole range of scheduling instances and is much easier to use.

Keywords Grid computing · Two-stage scheduling · Generalized extremal optimization · Genetic algorithm · Parallel job

P. Switalski (✉)
Computer Science Department, Siedlce University of Natural Sciences and Humanities, 3 Maja 54, 08-110 Siedlce, Poland
e-mail: piotr.switalski@ii.uph.edu.pl

F. Seredynski
Department of Mathematics and Natural Sciences, Cardinal Stefan Wyszynski University in Warsaw, Woycickiego 1/3, 01-938 Warsaw, Poland
e-mail: fseredynski@gmail.com

1 Introduction

Computational grids have recently become a very popular environment for providing high-performance computing for computationally intensive applications. The concept of grids is being studied by researchers in many fields, including high-performance computing (Foster and Kesselman 1998), networking (Smith et al. 2004), distributed systems (Casanova 2002), and web services (Parastatidis et al. 2005). One of the most important issues, from both practical and theoretical points of view, related to grids is job scheduling and load balancing. It derives from theoretical studies and practical observations that scheduling problems are computationally very complex, which is formally described as a problem belonging to the class of NP-hard problems (Garey and Johnson 1979). In practical terms, this means that different metaheuristics can be used to deliver not exact but approximate solutions in a reasonable time, and these solutions can be used in practice related to grid operation. The search for effective metaheuristics for scheduling problems in grid environments is currently an important issue in computer science.

Computational grids are systems that work in real time, and scheduling decisions should also be made in real time; this raises the issue of online scheduling, which is also a complex problem. The performance of online grid-scheduling algorithms highly depends on their adaptivity and flexibility in changing environments (Flling and Lepping 2012). In practice, the problem of online scheduling is often simplified by converting it into a series of offline problems (Flling and Lepping 2012). Therefore, in this paper we will focus on searching for effective metaheuristics to solve offline scheduling problems in grids. It is assumed that all jobs and grid properties are known in advance. We will consider a nonpreemptive offline scheduling of parallel batch jobs in a computational grid.

An efficient and flexible grid management (scheduling) system is required to fulfill users' job computational requests. Many studies propose either a distributed management system (Ernemann and Yahyapour 2003) or centralized scheduling (Ernemann et al. 2002). There are also combinations of distributed and centralized management (Vazquez-Poletti et al. 2007) that can be characterized as a type of hierarchical multilayer resource management system (Schwiegelshohn et al. 2008). In this approach the first (higher) layer is often controlled by a global grid scheduler. In this layer, jobs are scheduled on the machines in the grid. In the second layer (lower), a local management system exists that schedules assigned jobs on a local machine. In Flling et al. (2010) the authors addressed a similar grid-scheduling problem with a decentralized two-level approach. They constructed workload exchange policies for decentralized computational grids using an evolutionary fuzzy system. The users in the grid were assumed to submit their jobs to the middleware layer of their local site, which in turn decided on the delegation and execution either on the local system or on remote sites in situation-dependent scenarios.

The idea of grid computing has forced the development of new algorithms for a large number of heterogeneous resources. The execution of a user's application must satisfy both job execution constraints and system policies. The scheduling algorithms applied to traditional multiprocessor systems are often inadequate for grid systems. Many algorithms (see Ghafoor and Yang 1993; Hall et al. 2007) have been adapted to grid computing. However, many open problems remain in this area, including an architecture for multilayered system that would be suitable from the point of view of scheduling processes.

Metaheuristic approaches (Talbi 2009; Izakian et al. 2009) are widely applied today to solve NP-hard problems, in particular scheduling problems. In Aggarwal et al. (2005) GA was used to allocate jobs in a grid system when the makespan and flow time were minimized. In another paper (Kim and Weissman 2004), the authors proposed a GA-based approach to decomposing a job into multiple subtasks while simultaneously considering communication and computation. In YarKhan and Dongarra (2002), simulated annealing (SA) algorithm was used to tasks scheduling in grid systems. The authors used SA as the scheduling mechanism for a ScaLAPACK LU solver on a grid. In our paper we will use a relatively new metaheuristic called generalized extremal optimization (GEO) (Sousa et al. 2004) to solve a scheduling problem and compare its efficiency with that obtained using the most popular metaheuristic – a GA-based scheduling approach. Both GEO and GA belong to the same class of evolutionary algorithms, but they are inspired by different evolutionary processes. GA is inspired by mechanisms of simulated Darwinian evolution, which assumes that evolution is a permament process of changing and developing species. GEO is inspired by observations from paleontology and assumes that changes in species happen rarely at some moments in time. GEO is much simpler and requires the tuning of only one parameter, while GA requires the tuning of several parameters. Our earlier study (Switalski and Seredynski 2012) suggests that GEO-based schedulers can compete with GA-based schedulers working in a grid environment. In this paper we compare in detail both evolutionary approaches and show that a GEO-based scheduler is more effective in the sense of the quality of solutions and less computationally expensive while tuning parameters than a GA-based approach.

The paper is organized as follows. In the next section we define both the grid model and the scheduling problem. Section 3 presents concepts of the GEO and GA algorithms and their application to the scheduling problem. In Sect. 4 we present a local scheduling algorithm. Section 5 contains the results of test runs of the GEO algorithm to tune its parameters. Next, in Sect. 6 we analyze the experimental results comparing use of GEO and GA-based scheduling algorithms. The last section presents conclusions.

2 Grid model and scheduling

2.1 Grid system and parallel batch jobs

The grid model and offline scheduling problem are defined as follows (Tchernykh et al. 2010). We assume a heterogenous system consisting of a set of m parallel machines $M_1, M_2, \ldots, M_m$. Each machine M_i has m_i identical processors, also called the size of machine M_i. Figure 1a shows an example of a set of parallel machines in a grid system. One can see that machine M_1 consists of four identical processors, machine M_2 three processors, and machine M_m two processors.

In a grid system, a set of n jobs $J_1, J_2, \ldots, J_n$ waits for scheduling. A job J_j is described by a triple $(r_j, \text{size}_j, t_j)$. We consider nonpreemptive offline scheduling; therefore, the release time r_j is equal to zero. All jobs are known and available before the scheduling process. The size_j refers to the processor requirements. It specifies a number of processors required to run job J_j within the assigned machine. We can define this as a *degree of parallelism* or a *number of threads*. All threads of the job must run at the same time and on the same machine. The t_j is defined as the execution time of job J_j. Figure 1b shows an example of the job. As is evident, the job has three threads and requires three processors to execute them.

The $w_j = t_j * \text{size}_j$ denotes the *work* of job J_j. A machine executes a job of size_j when size_j processors are allocated to it during time t_j. We assume that job J_j needs to be allocated only within the assigned machine M_i. In other words, job

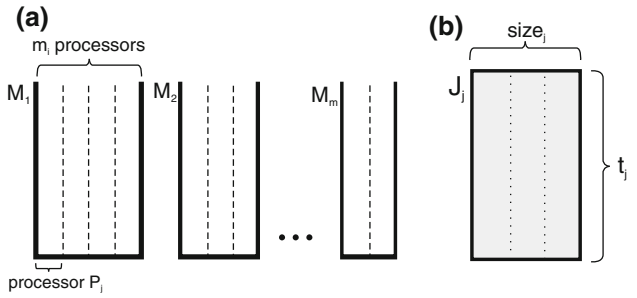

Fig. 1 Parallel machines in grid system (**a**) and parallel batch job (**b**)

threads cannot be assigned to and executed on two or more machines simultaneously. A job cannot be reallocated to a different machine. The machine must be capable of allocating a job to its processors, so $\text{size}_j \leq m_i$ must be satisfied.

A central problem of scheduling algorithms is the creation of a schedule such that the latest finish time of a job is minimized. Given processing times for n jobs $J_1, J_2, \ldots, J_n$ and their processor requirements size_j, we need to find an assignment of the jobs to m parallel machines $M_1, M_2, \ldots, M_m$ such that the completion time, also called the *makespan*, is minimized. Let us denote by S a schedule. By S_i we denote the schedule on machine M_i. The completion time of the jobs on machine M_i in schedule S_i is denoted by $C_i(S_i)$. We consider a minimization of the time $C_i(S_i)$ on each machine M_i across the system. The makespan is defined as

$$\text{Cmax} = \max_i(C_i(S_i)). \tag{1}$$

2.2 Scheduling in grid

The scheduling is performed using a two-stage algorithm. An important part of the scheduling process is an appropriate allocation of jobs on the machines. We consider a heterogenous system with a different number of processors on the machines. We require an algorithm that globally distributes the jobs throughout the system. Following distribution of the jobs, local schedulers assign threads' jobs to the processors. The jobs should be allocated in such a way that the makespan Cmax is minimized.

3 Global scheduling

3.1 Generalized extremal optimization and job allocation in grid

To solve the problem of allocating jobs among the machines of a grid system, we propose a relatively new evolutionary algorithm called GEO, presented by Sousa et al. (2004). The idea of this algorithm is based on the Bak–Sneppen model (Bak 1996; Bak and Sneppen 1993). It assumes that for each species of an ecosystem, some value, called a ranking, is

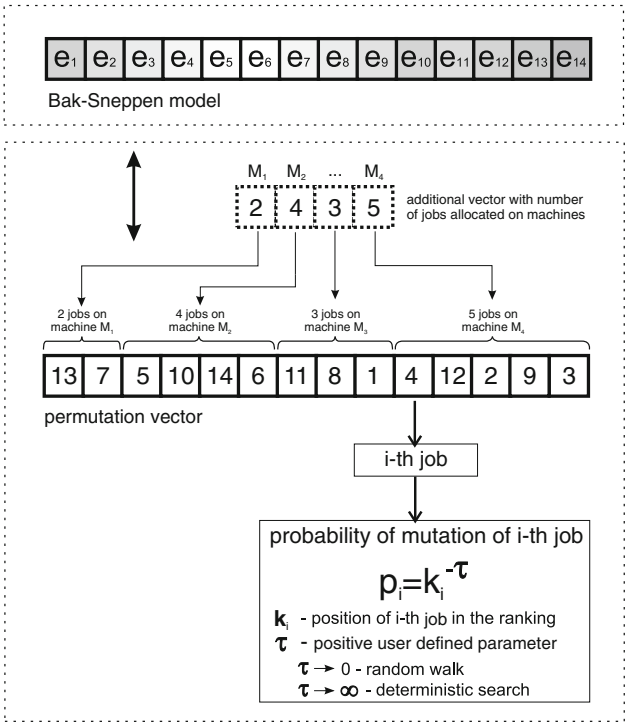

Fig. 2 Population of species in Bak–Sneppen model and its correspondence in GEO-based scheduling algorithm

assigned. Evolution in this model is driven by a process in which the weakest species in the population, together with its nearest neighbors, is always forced to mutate. The dynamics of this extremal process shows the characteristics of the self-organized criticality (SOC) (Bak et al. 1987), such as punctuated equilibrium, which are also observed in natural ecosystems. Punctuated equilibrium is a theory known in evolutionary biology. It states that in evolution, there are periods of stability punctuated by changes in an environment that force relatively rapid adaptation by generating *avalanches*, large catastrophic events that affect the entire system. The probability distribution of these avalanches is described by a power law in the form

$$p_i = k_i^{-\tau}, \tag{2}$$

where p_i is the probability of mutation of the ith species in an ecosystem consisting of n species, k is the position of the ith species in the ranking established between species (see below), and τ is a positive parameter. If $\tau \to 0$, then the algorithm performs a random search, but when $\tau \to \infty$, the algorithm performs a deterministic search. Bak and Sneppen Bak and Sneppen (1993) developed a simplified model of an ecosystem in which n species $e_1, e_2, \ldots, e_n$ are placed side by side on a line (upper part of Fig. 2). The lower part of Fig. 2 presents the idea of the GEO algorithm for grid scheduling based on the Bak and Sneppen model.

From the point of view of the scheduling algorithm, the GEO algorithm operates on a population of species represented by a string of jobs (called a *permutation vector*) allocated to machines in the grid system.

The lower part of Fig. 2 shows a set of 14 jobs allocated to 4 machines. The jobs are distributed among the machines. Indexes of jobs to allocated the machines are stored in an additional vector. One can see that jobs 13 and 7 are allocated to machine M_1, jobs 5, 10, 14, and 6 to machine M_2, and so forth. The permutation vector is a subject of the optimization process. The length of the vector is equal to the total number of jobs in the grid system.

3.2 GEO-based scheduling algorithm

The GEO algorithm was originally used to solve function optimization problems, where a population of species was represented by a sequence of bits. In the context of the scheduling problem, a population is represented by a permutation vector (Fig. 2). In the scheduling algorithm a job is forced to mutate with a probability proportional to the makespan, which corresponds to moving a single job to another machine. Moving a single job to another machine and the related change of the job position in the permutation vector results in changing the makespan. It also indicates the level of adaptability of each job in the permutation vector corresponding to a current solution of the scheduling problem. The quality of the solution (value of the makespan) can be higher or lower if a job is mutated (moved). After performing a single change caused by a mutation of the job and calculating the corresponding change in the makespan, we can create a ranking of the jobs according to changes in the makespan. In GEO scheduling, the ranking is created by sorting jobs in descending order (the higher the change in the makespan, the higher the ranking). From this moment on, the probability of mutation p_i of each ith job placed in the ranking can be calculated according to Eq. 2.

Figure 3 explains the scheme of the mutation in the GEO algorithm. The X-axis shows the jobs sorted based on their ranking. Let us assume that jobs are sorted according to $\Delta F_i = \mathrm{Cmax}_i - R$ (Y-axis on left in Fig. 3), where Cmax_i is the value of a makespan when the ith job is mutated, which means a change in the location of a job (see below). R is a positive constant value. The jobs are ordered from worst-adapted (maximal value of makespan) to best-adapted (minimal value of makespan), and the probability of mutation p_i is calculated. The probability indicates the chances of mutation of each job. One can see that the worst-adapted jobs have a significantly higher probability of mutation than the best-adapted jobs.

After calculating the ranking, job i from the permutation vector is selected with a uniform probability. The job is mutated with the corresponding probability of mutation

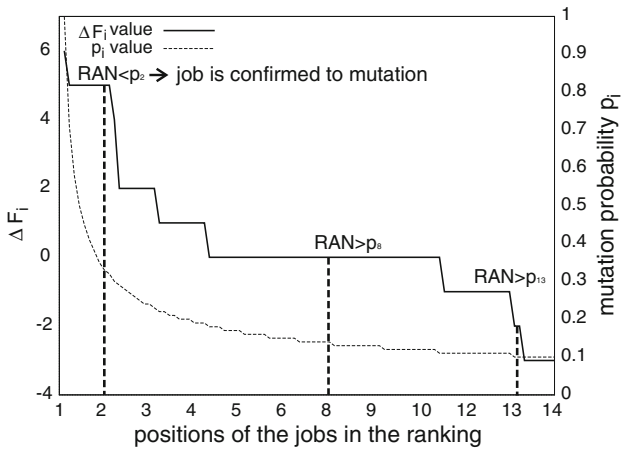

Fig. 3 Mutation scheme in GEO algorithm

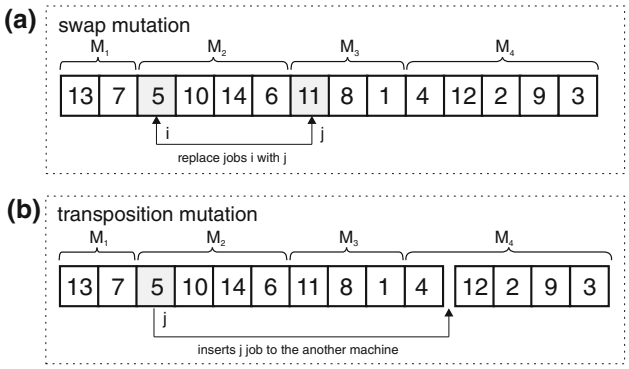

Fig. 4 Proposed mutation operators for GEO algorithm: swap (**a**), transposition (**b**)

p_i. To mutate the selected job, we generate a random number RAN with a uniform distribution in the range [0,1]. If RAN $< p_i$, then the job is accepted for mutation. Otherwise, the algorithm chooses with a uniform probability another job from the permutation vector and attempts to mutate it. In Fig. 3, one can see that the algorithm attempted to mutate sequentially the jobs at positions $k = 8$,$k = 13$, and $k = 2$. The first two jobs were not mutated because the probability of mutation was smaller than RAN. Finally, job 2 was mutated because p_2 was greater than RAN.

Figure 4 shows two mutation operators proposed for use in the GEO-based scheduling algorithm. In the first operator, called *swap mutation* (Fig. 4a), two jobs are selected randomly (indexed as i and j). Afterwards, jobs i and j are reversed.

The next type of proposed operator is called *transposition* (Fig. 4b). First, a job j is selected. The chosen job is moved to a machine having the shortest total time (machine M_4 in Fig. 4b). This type of mutation is oriented toward improving the load balancing in the grid system. The jobs and machines

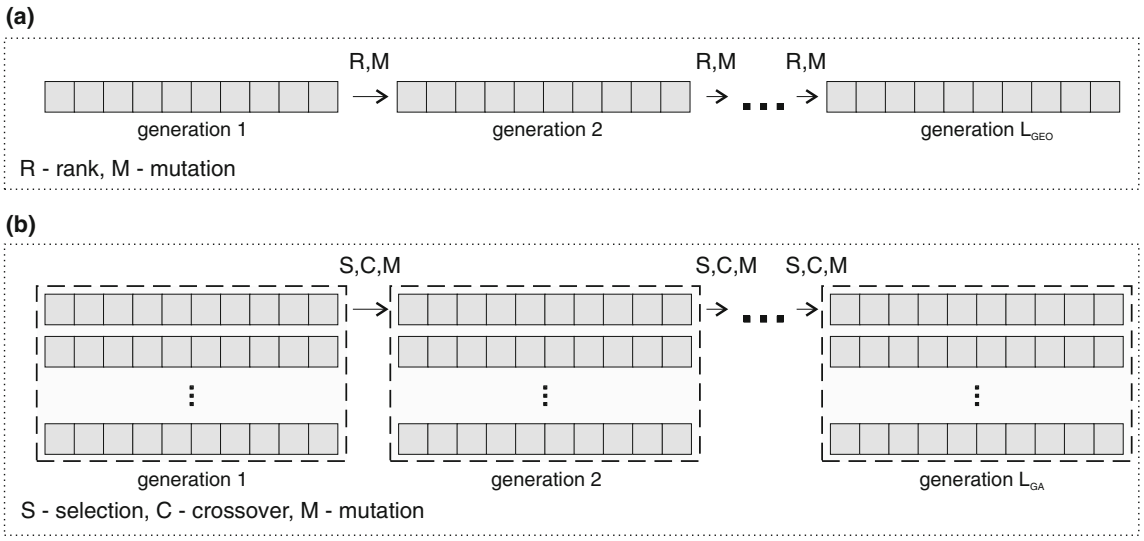

Fig. 5 Evolutionary engines for searching solutions in GEO algorithm (**a**) and GA (**b**)

considered by the mutation operator must satisfy size $\leq m$; otherwise the mutation must be repeated on another job.

Following mutation of the job, the makespan is calculated; then the best makespan found so far is saved. Algorithm 1 presents the GEO-based scheduling algorithm used in this paper.

Algorithm 1: GEO-based scheduling algorithm

1. Initialize randomly a permutation vector of length L that encodes n jobs for a given instance of the scheduling problem.
2. For the current configuration K of jobs (permutation vector), calculate the value Cmax corresponding to the makespan (Eq. 1) and set $K_{\text{best}} = K$ and $\text{Cmax}_{\text{best}} = \text{Cmax}$.
3. For each job i do:
   (a) mutate a job and calculate the makespan value Cmax_i for the string configuration K_i;
   (b) set the job fitness ΔF_i as $(\text{Cmax}_i - R)$, where R is a positive constant; the job fitness indicates the relative gain (or loss) that is a result of the job mutation;
   (c) return the job to its previous state.
4. Rank n jobs according to the makespan values, from $k = 1$ for the least-adapted job (at the top of the rankings) to $k = L$ for the best-adapted job (with the lowest ranking). For scheduling problems higher values of ΔF_i are at the top of the rankings. If two or more jobs have the same makespan, rank them in random order, but follow the general ranking rule.
5. Choose a job i with a uniform probability and mutate it according to the probability distribution $p_i = k^{-\tau}$, where τ is an adjustable parameter. This process is continued until some job is mutated.
6. Set $K = K_i$ and $\text{Cmax} = \text{Cmax}_i$.
7. If $\text{Cmax}_i < \text{Cmax}_{\text{best}}$, then set $\text{Cmax}_{\text{best}} = \text{Cmax}_i$ and $K_{\text{best}} = K_i$.
8. Repeat steps 3–8 until a given stopping criterion is reached.
9. Return K_{best} and $\text{Cmax}_{\text{best}}$.

3.3 GA-based scheduling algorithm

A GA is a search technique used to find an approximate solution in function and combinatorial optimization problems. It is a particular class of evolutionary algorithms (EAs) that uses mechanisms inspired by natural (Darwinian) evolution. The algorithm operates on a population of chromosomes that code the potential solutions. Chromosomes are usually strings of bits. In the context of our problem, strings are permutations of jobs similar to the GEO algorithm's permutation vector. Figure 5 shows the differences between the GEO and GA algorithms. As contrasted with the GEO algorithm, which operates on one string, a GA operates on a set of strings (individuals). Furthermore, both algorithms have different operators. In the GEO algorithm, the ranking is created and then the mutation operator is used. The GA algorithm uses three genetic operators: selection, crossover, and mutation.

The makespan Cmax is computed for each individual (chromosome). Algorithm 2 presents a GA-based scheduling algorithm. Note that, in contrast to the GEO algorithm, the GA has three parameters: population size, probability of crossover, and probability of mutation. These parameters

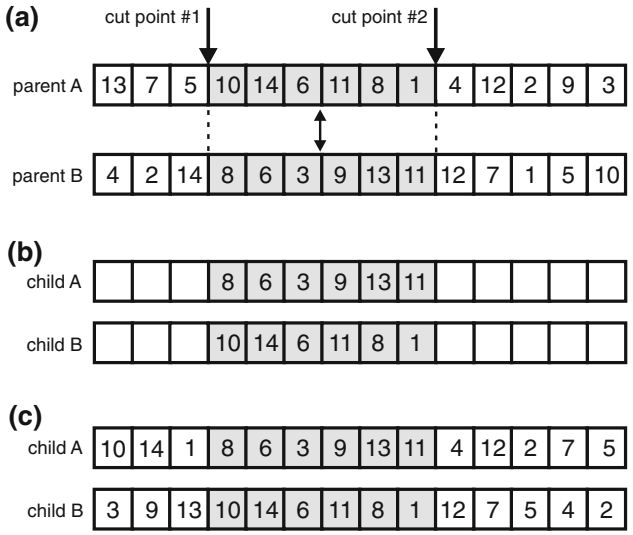

Fig. 6 Schema of *ordered crossover* (OX) operator: parent chromosomes (**a**), child chromosomes after exchange of substrings (**b**), filled child chromosomes (**c**)

are usually problem dependent and require tuning, which requires effort and computational time.

Algorithm 2: GA-based scheduling algorithm

1. Create an initial population of individuals (permutation vectors).
2. Calculate the makespan of each individual in the population.
3. Repeat.
   (a) Use roulette wheel operator to select permutation vectors for reproduction.
   (b) Apply genetic operators – crossover and mutation – to generate new solutions (permutation vectors).
   (c) Calculate the makespan for new permutation vectors.
   (d) Replace the population with new permutation vectors.
4. Repeat until stop condition is satisfied.

In the GA, specific crossover and mutation operators are used. For mutation we propose a transposition mutation (Fig. 4d). This forces jobs to migrate among machines.

As crossover operator we consider one that preserves the correct permutation of jobs. We use the *ordered crossover* (OX) operator proposed by Davis (1985). Offspring are created by choosing a subsequence of the jobs from one parent and preserving the relative order of the jobs from the other parent.

Figure 6 presents the OX operator. First, the two cut points for the parent chromosomes are randomly selected. To create an offspring, the string between two cut points in parent A is copied to child B and from parent B to child A. In Fig. 6a, the substring (10, 14, 6, 11, 8, 1) from parent A was copied to child B, and the substring (8, 6, 3, 9, 13, 11) in parent B was copied to child A. Then, the remaining positions are filled by considering the sequence of the jobs in the parent, starting after cut point 2. Let us consider the parent A chromosome. The next job in the chromosome after cut point 2 is job 4. This job does not exist in the copied substring in child A, so it is copied to the child chromosome at the considered position. The next one is job 12. This job is copied at the next position in the child chromosome. Also, the next job, job 2, is copied in the same way. At the next position is job 9. However, this job is already included in the substring in the child chromosome and is skipped, and we continue on to the next job, job 3, so we also skip this job. When the end of the chromosome is reached, the procedure is continued at position 1 of the parent chromosome. The job at position 1 is job 13. The job exists in the substring, so we need to skip it. The next four jobs (7, 5, 10, 14) are sequentially copied to a child chromosome (they do not exist in the substring). After that, jobs 6, 11, and 8 are omitted (they exist in the substring). The last job, job 1, is copied to the child chromosome. As a result, child A chromosome is completely filled. The child B chromosome is filled in the same way considering the parent B chromosome.

4 Local scheduling

Following distribution of jobs among machines, the local scheduling algorithm allocates the jobs within a particular machine. The applied algorithm is a variant of the list scheduling algorithm (Coffman 1976) – a relatively simple but effective heuristic used to solve scheduling problems. The main idea of this algorithm is to arrange jobs according to some priority list. Let us assume that size will be considered a priority. The jobs with the highest degree of parallelization have higher priority; therefore, they will be scheduled first. In our algorithm, the priority list of jobs is constructed according to the model presented in Sect. 2.1. Jobs are assigned at the earliest possible time on the available processors where the constraints (number of required processors and time) are preserved.

Figure 7 gives an outline of the local scheduling algorithm. It is assumed that a subset (1, 2, 3, 4, 5, 6) of six jobs was assigned to a machine (Fig. 7a, left). In this example, the jobs are sorted according to degree of parallelization size from largest to smallest values of this parameter (Fig. 7a, right). The jobs were ordered as follows. Jobs J_4 and J_2 contain four threads, so they are inserted into the list first. Next, jobs J_5 and J_6, with two threads, are considered by the list algorithm. Finally, jobs J_1 and J_3, containing one thread, are inserted into the list.

Figure 7b–g presents the sequence of schedules of jobs one a machine according to the job priority list. Job J_4, which has

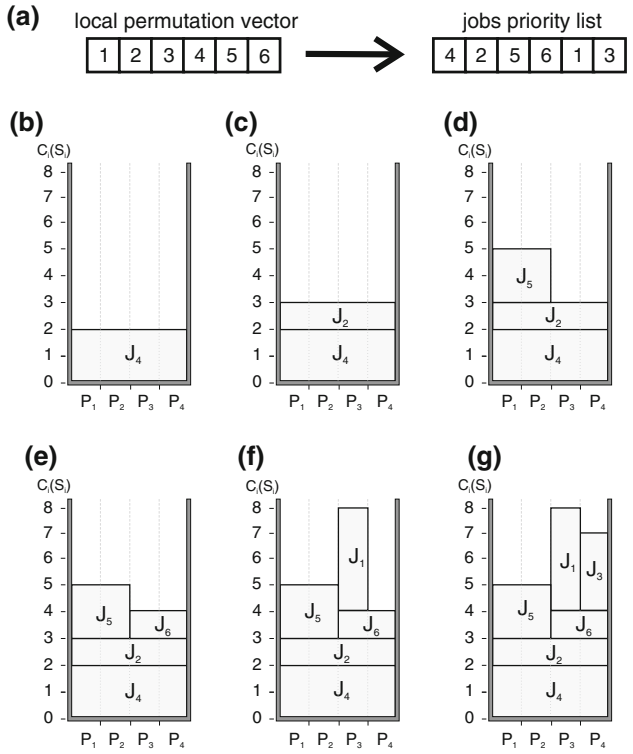

Fig. 7 Outline of local scheduling algorithm: transformation of a local permutation vector into a priority list (**a**), sequence of allocation of jobs on machine according to job priority list (**b**–**g**)

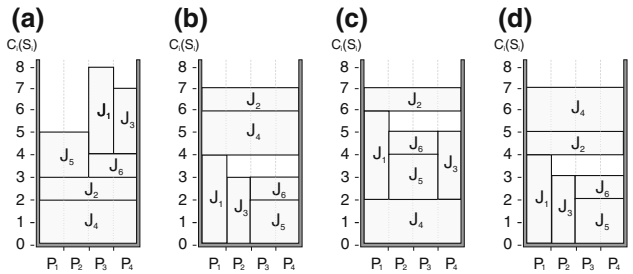

Fig. 8 Examples of schedules for variants of job sorting: degree of parallelization (**a**), job execution time (**b**), work involved in job (**c**), job order (**d**)

the highest priority, will be allocated first (Fig. 7b). Then job J_2 is scheduled (Fig. 7c). The remaining jobs are assigned to processors based on the previously created priority list until the algorithm has scheduled all the jobs.

A local scheduling algorithm is used to allocate the jobs for each machine separately. After that, the algorithm calculates the makespan Cmax for the grid system. The makespan can be different for the sorting variant. In this paper we assumed four variants of job sorting: degree of parallelization size_j, execution time of job t_j, work involved in job w_j, job order. The last variant preserves the original job order obtained using the GEO algorithm. Figure 8 shows examples of schedules for the aforementioned variants.

Table 1 Probabilities of job mutation for different values of τ parameter and position of job in rankings

Job position	Value of τ parameter		
	0.5	4.0	8.0
1	1.0000	1.0000	1.0000
2	0.7071	0.0625	0.0039
3	0.5774	0.0123	0.0002
4	0.5000	0.0039	0.0000
5	0.4472	0.0016	0.0000
6	0.4082	0.0008	0.0000
7	0.3780	0.0004	0.0000
8	0.3536	0.0002	0.0000
9	0.3333	0.0002	0.0000
10	0.3162	0.0001	0.0000
11	0.3015	0.0001	0.0000
12	0.2887	0.0000	0.0000
13	0.2774	0.0000	0.0000
14	0.2673	0.0000	0.0000

5 Tuning of GEO algorithm

5.1 Effect of τ parameter

In Sect. 3.1, the probability distribution p_i (Eq. 2) of avalanches in punctuated equilibria is presented. The value of this probability depends significantly on the position of a mutated job in the rankings and on the τ parameter. This parameter controls the size of a subset of the jobs in a permutation vector that could be mutated. Let us consider a string of, e.g., 14 jobs arranged by ranking, as shown in the first column of Table 1. Let us also consider values of τ in a range of 0.5 to 8. Columns 2–4 in Table 1 show the probabilities of mutation of the string for the given ranking for different values of τ.

One can see that the probability p_i of job mutation at the top of the rankings is notably higher than the probability of mutation for jobs at the bottom of the rankings. For small values of τ ($\tau = 0.5$) any job selected from the rankings will have a relatively high chance of being mutated. This probability decreases as the value of τ increases. For example (Table 1), if $\tau = 8.0$, then only the first job from the rankings will be forced to mutate with a probability of 1. Note that a high value of mutation related to low values of τ corresponds to a random search, while a very high value of τ corresponds to a deterministic search. Thus, the issue is finding the right value of τ that must be determined for the search process. The first set of experiments is oriented toward defining the right value of τ.

In the first experiment a set consisting of 100 jobs is used. The average execution time of each job is set to 3, and

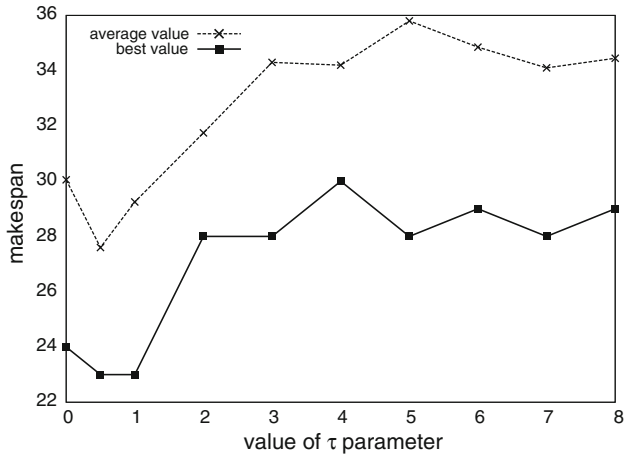

Fig. 9 Effect of τ parameter on makespan: scheduling 100 jobs in 8-machine environment

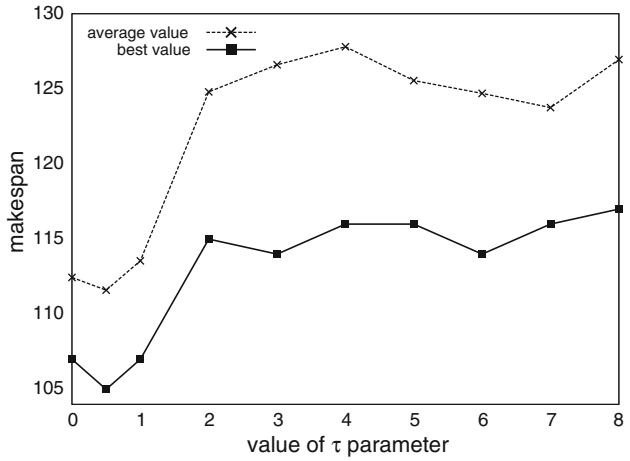

Fig. 10 Effect of τ parameter on makespan: scheduling 500 jobs in 8-machine environment

the average required number of processors is set to 2. The jobs will be scheduled on eight machines of a grid system. Machines contain four to eight processors. Figure 9 presents the results averaged on 20 runs of the algorithm for values of τ from 0 to 8. What value of the τ parameter is the best from the point of view of the makespan? One can see that the best value of the makespan occurs at $\tau = 0.5 \div 1$. However, looking at the average value of the makespan (Fig. 9) we conclude that $\tau = 0.5$ is the best value for this experiment. For values of $\tau > 1$ the algorithm mainly finds solutions with higher makespan values.

In the next experiment we use a set consisting of 500 jobs. Figure 10 presents the results averaged on 20 runs of the algorithm. One can see that the best value of the makespan is at $\tau = 0.5 \div 1$. For values of $\tau \leq 1$ the algorithm found noticeably better makespan values than at $\tau > 1$. Let us compare the convergence of the algorithm for 100- and 500-job sets at various values of τ. On the basis of the two conducted experiments, we conclude that the optimal value of τ is 0.5, which does not depend on the size of the job set. Figure 11 shows the run of the experiment for the optimal value $\tau = 0.5$. This value of τ makes the search process well oriented in finding of the solution. For comparison we present a run of the experiment for $\tau = 10$ (Fig. 12).

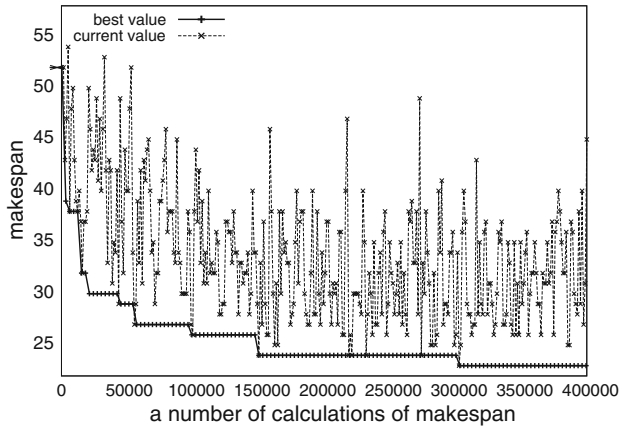

Fig. 11 Typical run of GEO algorithm for $\tau = 0.5$ and 100-job set

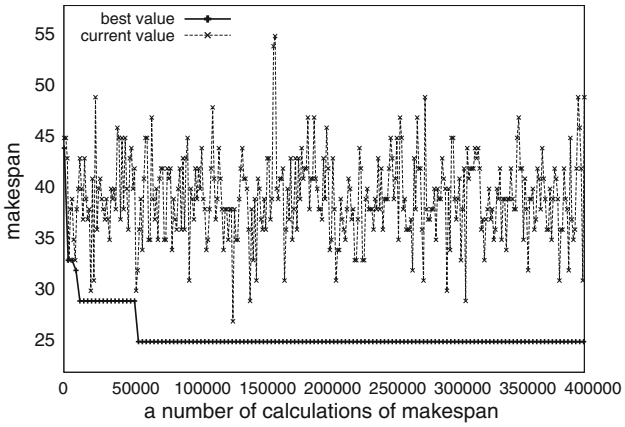

Fig. 12 Typical run of GEO algorithm for $\tau = 10$ and 100-job set

One can see that the process of searching for solutions is relatively slow for $\tau = 10$. This shows that the correct adjustment of τ has a significant effect on search process.

5.2 Migration of jobs

In Sect. 3.2, we presented two types of mutation that can be used in the GEO algorithm. Typically, during the GEO algorithm's search process, only one type of mutation is used. The alternate usage of more than one type of mutation might prove useful from the point of view of the optimization process.

Let us assume that two mutation types can be used in a single run of the GEO algorithm, and changing the type of mutation is controlled by the *migration probability* p_{m}. This means that mutation type 1 will be applied with probability p_{m} and mutation type 2 will be applied with probability 1-p_{m}.

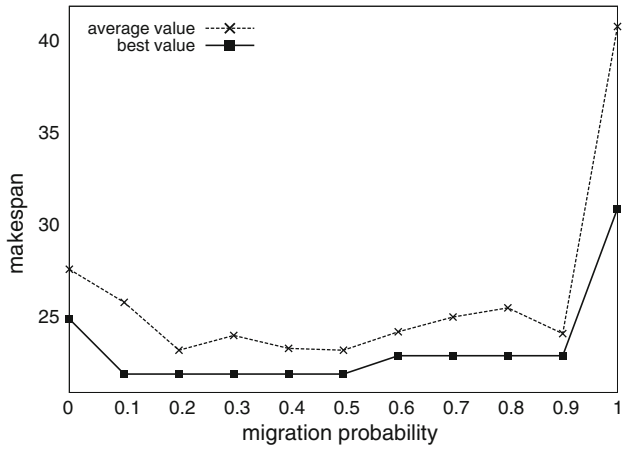

Fig. 13 Effect of migration probability on makespan: scheduling 100 jobs in 8-machine environment

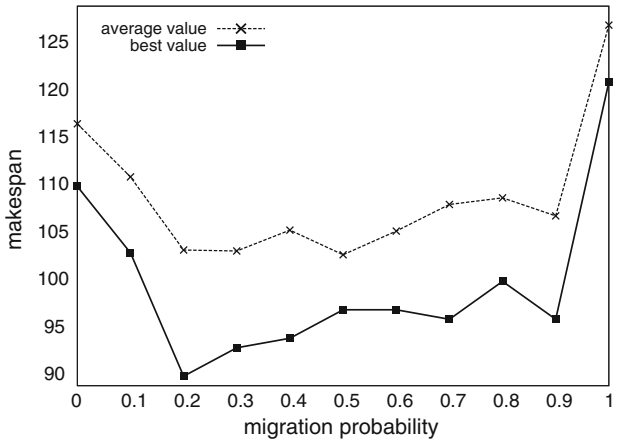

Fig. 14 Effect of migration probability on makespan: scheduling 500 jobs in 8-machine environment

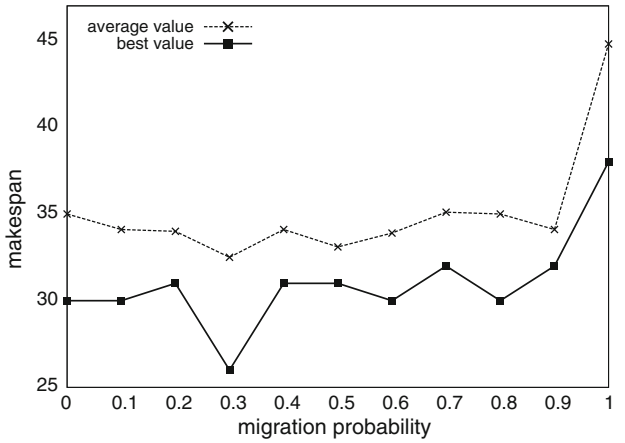

Fig. 15 Effect of migration probability on makespan: scheduling 500 jobs in 32-machine environment

In the experiment, we use two mutation types: swap mutation (Fig. 4a) and transposition mutation (Fig. 4b). Both the swap and transposition mutation operators move jobs to another machine, but the transposition mutation operator moves jobs using local optimization (Sect. 3.2). The question is what value of the migration probability p_m is optimal. In the following experiments, we will study this issue.

In the first experiment we use the set consisting of 100 jobs. The jobs will be scheduled on eight machines. Figure 13 presents the average results on ten runs for values of migration probability p_m of 0 to 1.

One can see that the algorithm has found a relatively good solution for a wide range of the migration probability p_m. As the optimal value of the migration probability we can assume $p_m = 0.1 \div 0.5$. However, as p_m increases in value, the computational costs of the algorithm also increase. The transposition mutation requires additional calculations of the makespan because we must find the machine with the lowest time. Thus, it is reasonable to use small values of p_m. In this experiment the optimal value of p_m is 0.2.

In the next experiment we increase the number of jobs and see how the makespan value depends on the migration probability. Figure 14 shows the averaged results on ten runs for values of migration probability p_m from 0 to 1. The optimal value of p_m is more evident. For $p_m = 0.2$ the algorithm definitely finds the best solutions.

In the next experiment we will find the optimal value of the migration probability for the 500-job set scheduled on 32 machines and see how the migration probability is correlated with the number of machines. Figure 15 shows the results of the experiment.

One can see that in this case the probability needs to be slightly increased to 0.3. Only for this value did the algorithm find the best makespan.

In Figs. 16, 17, and 18 we present typical runs for $p_m = 0$, $p_m = 0.2$, and $p_m = 0.95$, respectively. For $p_m = 0$ (Fig. 16) the algorithm cannot find the optimal makespan values. In this case the algorithm uses only a swap mutation. This operator is insufficient to obtain a good makespan because the jobs cannot be moved to another machine. The algorithm can only change the permutation of the jobs within the machines.

For the optimal value of migration probability $p_m = 0.2$ (Fig. 17) the algorithm is able to find very good makespan values. In the majority of cases, the jobs are swapped, and occasionally the algorithm moves the jobs between machines.

When the algorithm uses the transposition mutation very frequently (high value of $p_m = 0.95$) it cannot find a satisfactory solution (Fig. 18). If we look at the current value of the run, we realize that the process of searching for the makespan is slow. There are long periods when the algorithm does not change the best value of the makespan.

In this part we have shown the use of two types of mutation in the GEO algorithm. The optimal value of migration

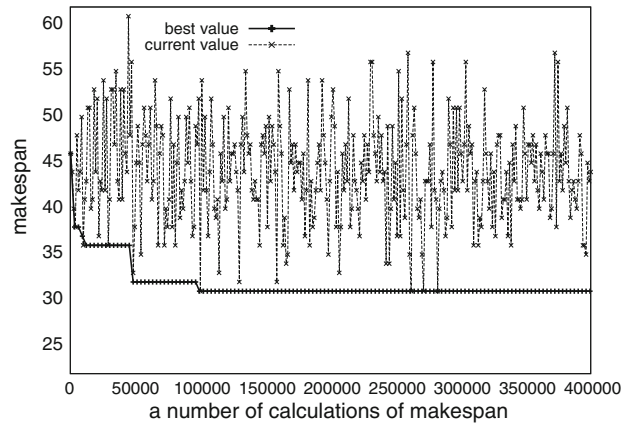

Fig. 16 Typical run of GEO algorithm for migration probability $p_m = 0.0$ (100-job set, 8-machine environment)

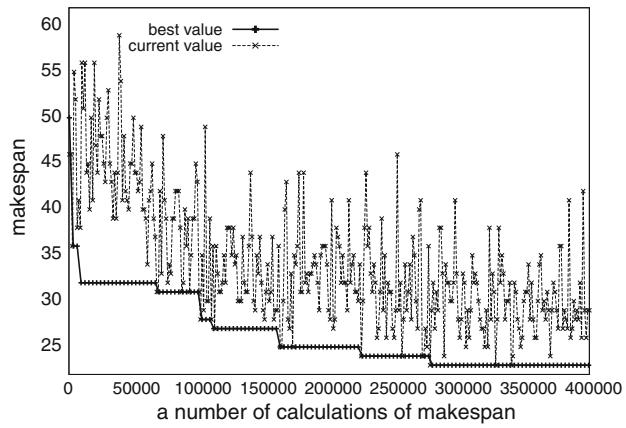

Fig. 17 Typical run of GEO algorithm for migration probability $p_m = 0.2$ (100-job set, 8-machine environment)

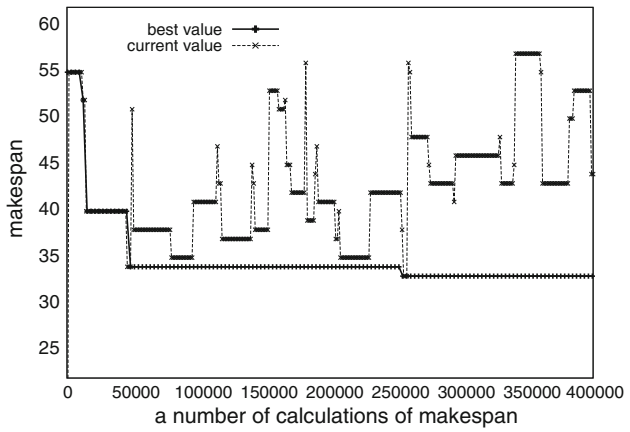

Fig. 18 Typical run of GEO algorithm for migration probability $p_m = 0.95$ (100-job set, 8-machine environment)

probability p_m were found. The algorithm is sensitive to the type of mutation. This issue has a significant effect on outcomes.

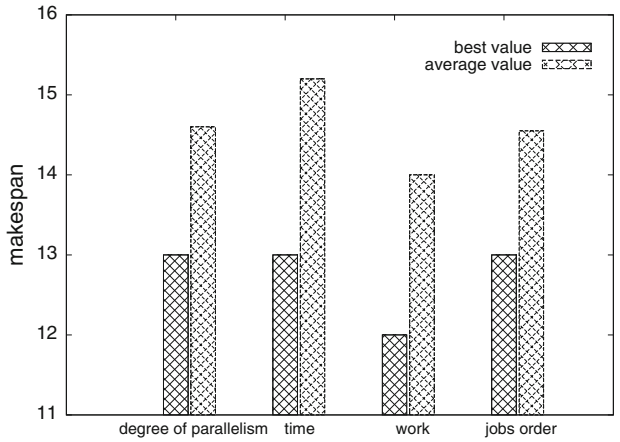

Fig. 19 Makespan values for different variants of job sorting

5.3 Local scheduling variants

Local scheduling is the second stage of the scheduling process. In Sect. 4 we presented four variants of job sorting. In the following experiment, we use these sorting variants for local scheduling. The experiment is conducted for the 100-job, 8-machine set. In Fig. 19 we see that the values of the makespan are different for each variant of job sorting. The best values were achieved for one of the variants: work involved in job w. Similar results were obtained for sets involving larger numbers of jobs and machines. For the next experiments we use this variant of the local scheduling algorithm.

6 Experimental results

6.1 Experimental settings

In this section we show the performance of the GEO-based scheduling algorithm for the scheduling problem considered in this paper. We also compare the GEO algorithm with the GA-based approach to scheduling. For experiments we use some randomly generated sets of jobs and machines. The jobs contain various numbers of threads and have various execution times. We assumed three sets of jobs. The sets are denoted by

$$xxx\_jobs\_y\_z,$$

where xxx is the number of jobs in the set, y is the average execution time of each job in a range from *1* to *2y*, z is the average required number of processors (threads) in a range of *1* to *2z*. The jobs were scheduled in environments of 4, 8, 16, 32, and 48 machines. The machine sets were denoted by

$$v\_machines\_w,$$

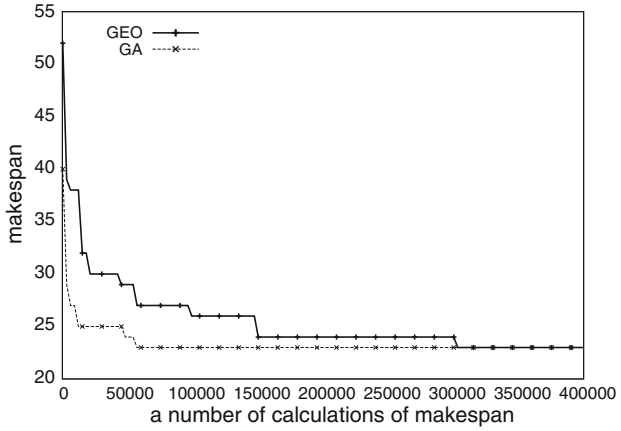

Fig. 20 Typical run of GEO algorithm and GA: experiment with 100-job set

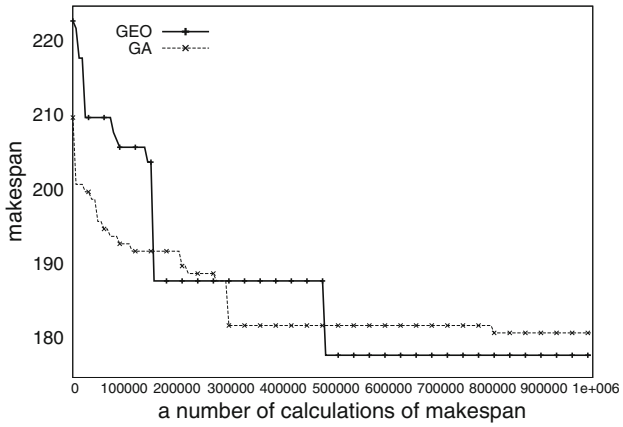

Fig. 21 Typical run of GEO algorithm and GA: experiment with 500-job set

where v is the number of machines in the set, w is the average number in a range of *1* to *2w* of processors in each machine.

Before the experiments we set the optimal parameters for both algorithms. The GA was previously in advance; however, this part of the work is not presented in the paper. We focused on the results obtained by these algorithms. For the GEO algorithm the parameters are as follows:

- $\tau = \mathbf{0.5}$;
- Migration probability of $p_{\mathrm{m}} = \mathbf{0.2}$ for environments of 4, 8, and 16 machines, $p_{\mathrm{m}} = \mathbf{0.3}$ for a 32-machine environment,
- Types of mutation used: swap, transposition.

In the GA we use the following parameters:

- Mutation probability of **0.1**;
- Crossover probability of **0.9**;
- Population size: **100** for small job sets (100 jobs); **200** for sets of 500 and 1,000 jobs.

Because the GEO algorithm and the GA have clearly different optimization mechanisms, we need to establish fair rules governing their evaluation. The calculation of the makespan is the main source of the time complexity of the presented algorithms, and the number of evaluations of the makespan in both algorithms may be different. To be able to make a comparison of both algorithms, we allowed them to be run in such a way that the number of evaluations of the makespan for both algorithms was the same.

Figures 20 and 21 show typical runs of both the GEO and GA-based scheduling algorithms for the 100- and 500-job sets, respectively, scheduled on 8 machines. One can see (Fig. 20) that for a relatively small set of jobs (100 jobs), the typical runs of both algorithms are similar. In the case of 500 jobs, the GEO-based scheduling algorithm works more persistently (Fig. 21). Avalanches occurring from time to time lead to a remarkable decrease in the makespan, which results in the discovery of a better quality solution.

6.2 Results

In this section we will present the results of the conducted experiments using different instances of the scheduling problem and with the application of the GEO algorithm and the GA. Tables 2, 3, and 4 present the results, averaged on the basis of 30 runs. We use three scenarios for the experiments:

- Machine sets: *4-48_machines_16* and job sets: *100-500_jobs_3_4*,
- Machine sets: *4-48_machines_16* and job sets: *100-500_jobs_6_4*,
- Machine sets: *4-48_machines_8* and job sets: *100-500_jobs_9_2*.

We start with the small instances of the problem. In Tables 2, 3, and 4 we present the minimal time (makespan) obtained by the algorithms, the average makespan (italics), and, in parentheses, the standard deviation.

One can see that for the 100-job set the results (minimal makespan) are similar for both algorithms, but the GEO algorithm slightly outperforms the GA. In addition, the average and standard deviation are smaller for the GEO algorithm. Only for the experiment on 48 machines is the GA better than the GEO algorithm. However, both algorithms found the same minimal makespan.

For the experiments involving the use of 200 jobs, the results differ for both algorithms. The GEO algorithm is significantly better than the GA, especially on four machines. Again, both algorithms are similar with 48 machines.

The last instances use 500 jobs. They are the most difficult cases because the length of the string representing a

Table 2 Comparison of makespan obtained by GEO algorithm and GA for machine sets *4–48_machines_16* and jobs sets *100–500_jobs_3_4*

Job set	4 machines		8 machines		16 machines		32 machines		48 machines	
	GEO	GA	GEO	GA	GEO	GA	GEO	GA	GEO	GA
100 jobs	25	26	12	13	8	8	6	5	5	5
	25.40	*26.13*	*12.90*	*14.43*	*8.10*	*9.10*	*7.27*	*6.37*	*6.30*	*5.67*
	(0.50)	(0.35)	(0.31)	(0.97)	(0.31)	(0.55)	(0.87)	(0.61)	(0.95)	(0.48)
200 jobs	49	52	23	29	13	16	11	10	9	9
	49.50	*55.60*	*24.03*	*33.17*	*17.13*	*18.27*	*16.43*	*11.67*	*16.20*	*9.83*
	(0.51)	(2.04)	(1.50)	(2.17)	(2.93)	(1.28)	(2.13)	(0.71)	(3.67)	(0.59)
500 jobs	116	137	60	80	39	42	25	24	18	19
	119.27	*147.00*	*79.67*	*88.07*	*48.67*	*46.73*	*32.73*	*26.37*	*24.87*	*21.03*
	(6.61)	(5.97)	(10.37)	(3.29)	(8.50)	(2.07)	(3.96)	(1.10)	(7.09)	(0.93)

Italics: average makespan; parentheses: standard deviation

Table 3 Comparison of makespan obtained by GEO algorithm and GA for machine sets *4-48_machines_16* and jobs sets *100–500_jobs_6_4*

Job set	4 machines		8 machines		16 machines		32 machines		48 machines	
	GEO	GA	GEO	GA	GEO	GA	GEO	GA	GEO	GA
100 jobs	50	51	24	25	14	16	11	11	11	11
	50.57	*52.60*	*25.73*	*29.47*	*17.03*	*18.30*	*15.80*	*12.73*	*16.67*	*11.50*
	(0.57)	(1.77)	(0.69)	(2.19)	(1.16)	(0.84)	(2.87)	(0.91)	(3.67)	(0.78)
200 jobs	92	96	43	53	27	31	20	20	18	17
	93.30	*103.40*	*46.50*	*59.87*	*38.40*	*34.37*	*27.90*	*22.23*	*26.93*	*19.30*
	(0.75)	(3.76)	(3.50)	(3.22)	(8.81)	(1.73)	(6.05)	(1.04)	(5.19)	(1.02)
500 jobs	224	284	143	155	77	86	45	48	36	39
	259.19	*299.96*	*176.93*	*180.79*	*101.10*	*98.07*	*62.40*	*55.20*	*48.70*	*43.77*
	(22.70)	(9.13)	(19.33)	(8.52)	(14.12)	(5.30)	(10.54)	(2.68)	(8.39)	(2.01)

Italics: average makespan; parentheses: standard deviation

Table 4 Comparison of makespan obtained by GEO algorithm and GA for machines sets *4-48_machines_8* and job sets *100-500_jobs_9_2*

Job set	4 machines		8 machines		16 machines		32 machines		48 machines	
	GEO	GA	GEO	GA	GEO	GA	GEO	GA	GEO	GA
100 jobs	76	80	41	44	25	28	17	17	17	17
	77.10	*82.33*	*42.53*	*49.53*	*27.17*	*29.57*	*22.43*	*17.73*	*25.37*	*18.83*
	(0.71)	(2.15)	(1.41)	(2.99)	(1.23)	(0.90)	(3.96)	(0.83)	(6.71)	(1.23)
200 jobs	130	139	65	85	38	46	22	26	25	27
	130.83	*147.37*	*69.33*	*92.60*	*50.97*	*49.27*	*30.93*	*28.07*	*32.10*	*29.37*
	(0.59)	(4.57)	(3.87)	(4.53)	(10.41)	(1.98)	(5.46)	(1.23)	(5.82)	(1.35)
500 jobs	374	402	221	233	122	124	71	66	68	65
	375.93	*428.97*	*254.90*	*246.97*	*171.67*	*142.47*	*90.70*	*72.53*	*97.00*	*71.77*
	(2.02)	(15.02)	(18.92)	(9.67)	(24.54)	(5.88)	(13.90)	(3.23)	(17.23)	(3.07)

Italics: average makespan; parentheses: standard deviation

solution in the GEO algorithm dramatically increases. For these instances the GEO algorithm outperforms the GA in all instances. The average and standard deviation were smaller in comparison with the GEO algorithm. However, the GEO algorithm finds more high-quality schedules.

7 Conclusions

In this paper we have proposed a two-stage grid scheduling algorithm for a grid environment, where we used a relatively new metaheuristic called a GEO algorithm. We compared the obtained simulation results with those obtained with the use of the GA. We showed that the GEO-based scheduling algorithm outperformed the GA-based scheduling algorithm in terms of the makespan in a wide range of scheduling instances.

The application of the GEO algorithm to scheduling problems has confirmed that this algorithm is useful for such problems. Its simplicity is one of its advantages. The performance of the algorithm depends, in fact, in a basic version, on only one parameter – the value τ – or, in a version with more complex mutations, on two parameters – the value τ and the migration probability. These values were established experimentally. The results of the experiments show that, despite the GEO algorithm's simplicity, the algorithm can find good-quality schedules.

References

Aggarwal, M., Kent, R. D., & Ngom, A. (2005). Genetic Algorithm Based Scheduler for Computational Grids. In *Proc. of the 19th Annual International Symposium on High Performance Computing Systems and Application (HPCS'05)* (pp. 209–215).

Bak, P. (1996). *How nature works*. New York: Copernicus, Springer.

Bak, P., & Sneppen, K. (1993). Punctuated equilibrium and criticality in a simple model of evolution. *Physical Review Letters*, *71*, 4083–4086.

Bak, P., Tang, C., & Wiesenfeld, K. (1987). Self-organized criticality. *Physical Review Letters*, *59*, 381–384.

Casanova, H. (2002). Distributed computing research issues in grid computing. *ACM SIGACT News*, *33*(3), 50–70.

Coffman, E. G. (1976). *Computer and job-shop scheduling theory*. New York: Wiley.

Davis, L. (1985). Job-shop Scheduling with Genetic Algorithms. *Proceedings of an International Conference on Genetic Algorithms and Their Applications* (pp. 136–140).

Ernemann, C., & Yahyapour, R. (2003). Applying Economic Scheduling Methods to Grid Environments. In *Grid Resource Management - State of the Art and Future Trends* (pp. 491–506). Kluwer Academic Publishers.

Ernemann, C., Hamscher, V., Schwiegelshohn, U., Streit, A., & Yahyapour, R. (2002). On Advantages of Grid Computing for Parallel Job Scheduling. *Proceedings of 2nd IEEE International Symposium on Cluster Computing and the Grid* (pp. 39–46).

Foster, J., & Kesselman, C. (1998). *Computational Grids. Chapter 2 of The Grid: Blueprint for a Future Computing Infrastructure*. Ian Foster and Carl Kesselman, Morgan Kaufman.

Flling, A., Grimme, C., Lepping, J., & Papaspyrou, A. (2010). Robust load delegation in service grid environments. *IEEE Transactions on Parallel and Distributed Systems*, *21*(9), 1304–1316.

Flling, A., & Lepping, J. (2012). Knowledge discovery for scheduling in computational grids. *Wiley Interdisciplinary Reviews*, *2*(4), 287–297.

Garey, M. R., & Johnson, D. S. (1979). *Computers and intractability: A guide to the theory of NP-completeness*. Basingstoke: Macmillan Higher Education.

Ghafoor, A., & Yang, J. (1993). A distributed heterogeneous supercomputing management system. *Computer*, *26*(6), 78–86.

Hall, R., Rosenberg, A. L., & Venkataramani, A. (2007). A Comparison of Dag-Scheduling Strategies for Internet-Based Computing. *IEEE International Parallel and Distributed Processing Symposium* (pp. 1–9).

Izakian, H., Abraham, A., & Snasel, V. (2009). Metaheuristic based scheduling meta-tasks in distributed heterogeneous computing system. *Sensors*, *9*, 5339–5350.

Kim, S., & Weissman, J. B. (2004). A Genetic Algorithm Based Approach for Scheduling Decomposable Data Grid Applications. In *Proc. of the 2004 International Conference on Parallel Processing (ICPP'04)* (pp. 406–413).

Parastatidis, S., Watson, P., & Webber, J. (2005). *Grid Computing Using Web Services*. Technical Report, University of Newcastle upon Tyne, School of Computing Science.

Schwiegelshohn, U., Tchernykh, A., & Yahyapour, R. (2008). Online Scheduling in *Grids. 22nd IEEE International Parallel and Distributed Processing Symposium (IPDPS 2008)* (pp. 1–10).

Smith, M., Friese, T., & Freisleben, B. (2004). Towards a Service-Oriented Ad Hoc Grid. *Third International Symposium on Parallel and Distributed Computing/Third International Workshop on Algorithms, Models and Tools for Parallel Computing on Heterogeneous Networks* (pp. 201–208).

Sousa, F. L., Ramos, F. M., Galski, R. L., & Muraoka, I. (2004). Generalized Extremal Optimization: A New Meta-heuristic Inspired by a Model of Natural Evolution. *Recent Developments in Biologically Inspired Computing* (pp. 41–60).

Switalski, P., & Seredynski, F. (2012). A grid scheduling based on generalized extremal optimization for parallel job model, parallel processing and applied mathematics. *Lecture Notes in Computer Science*, *7204*, 41–50.

Talbi, E.-G. (2009). *Metaheuristics: From design to implementation*. Hoboken: Wiley.

Tchernykh, A., Schwiegelshohn, U., Yahyapour, R., & Kuzjurin, N. (2010). On-line hierarchical job scheduling on grids with admissible allocation. *Journal of Scheduling*, *13*(N5), 545–552.

Vazquez-Poletti, J. L., Huedo, E., Montero, R. S., & Llorente, I. M. (2007). A comparison between two grid scheduling philosophies: EGEE WMS and grid way. *Journal Multiagent and Grid Systems*, *3*(N4), 429–440.

YarKhan, A., & Dongarra, J. (2002). *Experiments with scheduling using simulated annealing in a grid environment. Grid computing GRID 2002*. Berlin: Springer.