Using Imperialist Competition Algorithm for Independent Task Scheduling in Grid Computing

Article *in* Journal of Intelligent and Fuzzy Systems · August 2014

DOI: 10.3233/IFS-130988

4 authors:

Zahra Pooranian
Sapienza University of Rome
21 PUBLICATIONS **193** CITATIONS
SEE PROFILE

Mohammad Shojafar
Consorzio Nazionale Interuniversitario per le …
68 PUBLICATIONS **578** CITATIONS
SEE PROFILE

Bahman Javadi
Western Sydney University
62 PUBLICATIONS **832** CITATIONS
SEE PROFILE

Ajith Abraham
Machine Intelligence Research Labs (MIR Lab…
1,347 PUBLICATIONS **18,239** CITATIONS
SEE PROFILE

Journal of Intelligent & Fuzzy Systems xx (20xx) x–xx
DOI:10.3233/IFS-130988
IOS Press

Using imperialist competition algorithm for independent task scheduling in grid computing

Zahra Pooranian[a], Mohammad Shojafar[b,*], Bahman Javadi[c] and Ajith Abraham[d]
[a]*Department of Computer Engineering, Islamic Azad University, Andimeshk, Iran*
[b]*Department of Information Engineering, Electronics (DIET), Sapienza University of Rome, Via Eudossiana, Rome, Italy*
[c]*School of Computing, Engineering and Mathematics, University of Western Sydney, Sydney, Australia*
[d]*Machine Intelligence Research Labs (MIR Labs), WA, USA*

Abstract. A grid computing environment provides a type of distributed computation that is unique because it is not centrally managed and it has the capability to connect heterogeneous resources. A grid system provides location-independent access to the resources and services of geographically distributed machines. An essential ingredient for supporting location-independent computations is the ability to discover resources that have been requested by the users. Because the number of grid users can increase and the grid environment is continuously changing, a scheduler that can discover decentralized resources is needed. Grid resource scheduling is considered to be a complicated, NP-hard problem because of the distribution of resources, the changing conditions of resources, and the unreliability of infrastructure communication. Various artificial intelligence algorithms have been proposed for scheduling tasks in a computational grid. This paper uses the imperialist competition algorithm (ICA) to address the problem of independent task scheduling in a grid environment, with the aim of reducing the makespan. Experimental results compare ICA with other algorithms and illustrate that ICA finds a shorter makespan relative to the others. Moreover, it converges quickly, finding its optimum solution in less time than the other algorithms.

Keywords: Grid computing, scheduling, artificial intelligence algorithm, imperialist competition algorithm (ICA), independent task scheduling

1. Introduction

Application of a new technology, or scientific evolution, requires scientific proof and practical implementation. Because it is time-consuming to implement practical research and mistakes can arise because of in attention to problems in theoretical subjects, there is a need to use simulations in some contexts instead of real implementations. The computations that are needed to simulate and study all aspects of scientific research projects require a significant amount of computational power that a single computer would take too much time to provide. Superscalar computers, vector processors, and pipeline processing were proposed to address this problem. Although they provide more computational power and greater speed than a single computer, technological limitations related to speed and the high cost of their design and manufacture make them available only to users with no financial limitations. Grid computations, which are based on distributed systems, were proposed to solve such problems. Grid systems have

*Corresponding author. Mohammad Shojafar, Department of Information Engineering, Electronics (DIET), Sapienza University of Rome, Via Eudossiana 18, 00184 Rome, Italy. E-mail: shojafar@diet.uniroma1.it.

been proposed as a solution overcoming the limitations of hardware availability and computer locations, so that unused computers and their computational power can be exploited [35]. Grid computing systems are well-known for solving complicated large-scale problems in science, engineering, and finance [19], and have provided a wide range of heterogeneous and distributed resources for data-intensive computations [22]. In recent years, grid computing has been the subject of much research and has been used in commercial environments [9].

A resource management system (RMS) is the most important component of grid computing; it has a significant role in controlling and supervising the usage of resources. The most important function of an RMS is to schedule incoming tasks, assigning them to available compatible resources [38]. However, the heterogeneous and dynamic state of resources in grid systems poses difficulties, particularly when combined with complex task scheduling. Deterministic algorithms don't have the necessary efficiency to solve these scheduling problems, so a considerable amount of research has been devoted to using heuristic algorithms such as genetic algorithms (GAs) [21], simulated annealing (SA) [17], particle swarm optimization (PSO) [15], ant colony optimization (ACO) [36], Queen-Bee Algorithm [26], tabu search (TS) [27], and various combinations of these [8, 12, 13, 28, 39] to produce better results in reasonable time. The heuristic ICA [6], proposed in 2007, was inspired by the sociopolitical evolution of imperial phenomena and has been used for solving many optimization problems in continuous space.

This paper proposes a discrete version of ICA for solving the independent task scheduling problem in grid computing systems. The present paper converts ICA from a continuous state algorithm to a discrete state algorithm by changing the assimilation stage. The resulting algorithm is compared with SA and other heuristic algorithms and is shown to produce better results than these. This algorithm simultaneously considers makespan and completion time by using appropriate weights in the mean total cost function.

The rest of the paper is organized as follows. Section 2 presents related artificial intelligent methods that have been used in grid scheduling. Section 3 defines the independent task scheduling problem as a system model. Section 4 describes the ICA. Section 5 presents our proposed method with discrete ICA, and Section 6 compares it with related methods using several combinations of parameters. Finally, Section 7 summarizes and concludes the paper.

2. Related work

A large number of heuristic algorithms have been proposed for grid scheduling. Most of them try to minimize the maximum completion time of tasks, or *makespan*. Each task has its own deadline, and we try to decrease the makespan in order to prevent tasks from failing to execute because of their deadlines. That is, decreasing the makespan results in the ability to execute more tasks in the network. It also helps to provide efficient resource allocation and energy utilization.

The hierarchic genetic strategy (HGS) algorithm was proposed in [20] for scheduling independent tasks in a grid system and is implemented in dynamic grid environments in batch mode. This algorithm simultaneously considers optimization of flow time and makespan. The authors generate root nodes based on two other algorithms: longest job to fastest resource and shortest job to fastest resource (LJFR-SJFR) [3] and minimum completion time (MCT) [37], and they generate the rest of the population stochastically. In LJFR-SJFR, initially, the highest workload tasks are assigned to machines that are available. Then the remaining unassigned tasks are assigned to the fastest available machines. In MCT [37], tasks are assigned to machines that will yield the earliest completion time.

The rotary chaotic particle swarm optimization (RCPSO) algorithm was proposed in [34] for solving the workflow scheduling problem in grid systems. This algorithm considers reliability as one of the quality of service (QOS) parameters, in addition to time and cost. It also incorporates a new rotary discrete rule (RD-rule) method that helps the PSO algorithm optimize workflow scheduling in a discrete space. PSO simulates a swarm algorithm that orchestrates the movement of groups of fish and birds. Like other evolutionary algorithms, PSO stochastically initializes its population. The population particles are collections of unknown parameters whose optimal value must be determined. Each particle thus provides a possible problem solution. The parameters can be defined in *realcoding* based on the problem conditions. PSO searches the solution environment in a way that tries to move particles toward the best positions they have found, in hope of arriving at a better processing time. Eventually, all particles will converge to a single optimal point. RCPSO incorporates cancellation of the history velocity, double perturbations, and detecting the exact time and dimension of the double perturbations in order to prevent premature convergence and enhance the algorithm's performance

[34]. The drawback of RCPSO is that it is unable to optimize control of the chaotic system.

The balanced ant colony optimization (BACO) algorithm was proposed in [11] for task scheduling in a grid environment. Ant colony algorithms imitate the behavior of ants, which have an outstanding ability to cooperate with one another for finding the best food resources and how to reach them. The BACO algorithm changes ACO to balance the entire system load and minimize the makespan for a set of tasks. The local and global functions of the pheromone have been changed to perform the load balancing.

Fuzzy PSO was proposed in [23] to schedule tasks with the position and velocity representations for particles based on fuzzy matrices instead of the real vectors used in conventional PSO. This approach dynamically provides an optimal schedule for task completion with minimal makespan while allocating resources for utilization.

Use of the TS algorithm was proposed in [14] for solving the flowshop scheduling problem. Flowshop scheduling is defined as assigning sequences of jobs to a group of machines in the same order. TS attempts to diminish the makespan. Here the solution neighborhood presents various combinations of exchange mechanisms as successful search results for decreasing makespan.

The GA-SA algorithm, a combined evolutionary algorithm, was proposed in [39] for solving the independent task scheduling problem in grid systems. The main purpose of this algorithm is to find a solution that minimizes the total completion time. Since GAs search the problem space globally and are weak in local searches, the combination of a GA with the local SA search algorithm tries to remedy this weakness, thus exploiting the advantages of the two algorithms.

An algorithm that combines a GA and TS for scheduling independent tasks in computational grids was proposed in [8]. The GA is implemented as the main algorithm and the TS procedure is called to improve population numbers.

A PSO algorithm that simultaneously minimizes makespan and flowtime was proposed in [16]. There are two parameters for the fitness function, with one coefficient. The first issue in applying PSO for optimization is that we want to have a mapping between problem solutions and particles. In this paper, an $m \times n$ matrix is used, where m is the number of nodes (resources) and n is the number of tasks. The matrix elements are set to 0 or 1. Let X_k be the position vector for the k-th particle. If $X_k(i, j) = 1$, this means that the j-th task executes on the i-th resource. Also, a star neighborhood topology [18] is used for defining nbest, the best positions of all particles in all previous steps (we could call this the *best global solution*). A combination of PSO and thermal simulation, HPSO, was proposed in [13], with SA used to avoid falling into local optimums.

To date, the ICA has not been used for solving the independent task scheduling problem in grid systems. It is able to optimize similar and even higher than other mentioned optimization algorithms in different problems. In addition, it has adequate speed in finding optimum solution (i.e., less execution time). The ICA, which is discussed in detail in Section 4, is based on stochastic populations that imitate human sociopolitical evolution. In this algorithm, several imperialists together with their colonies try to find an optimal state of the empire for a given optimization problem. ICA has been used to solve various optimization problems, including the design of proportional integral derivative controllers (PID controllers) [5], transmission expansion planning [2], feature selection [24], and solving the traveling sales man problem [25].

ICA has also recently been used to solve the flowshop scheduling problem. In [7], ICA is used for flowshop scheduling by considering the minimization of earliness and using quadratic tardiness penalties. In [4], ICA is used for solving the flexible flowshop scheduling problem with the goal of minimizing maximum completion time by limiting waiting time. This paper considers a flexible flowshop scheduling problem called *flowshop with multiple processors*. The algorithm consists of multiple stages with each stage containing parallel machines, and each job on each machine includes a series of operations where each operation has a specific setup time and processing time. In [32], the hybrid imperialist competitive algorithm (HICA) and stochastic imperialist competitive algorithm (SICA) are proposed for no-wait two-stage flexible manufacturing in flowshops. The HICA algorithm assigns the selected task to the machine with the earliest available time, while in SICA, the assignment is random. Finally, in [33], ICA is used for bi-criteria assembly flowshop scheduling.

3. System model

The effectiveness of a proposed algorithm for the independent task scheduling problem studied in this paper depends on minimizing makespan. Task scheduling in a grid includes n tasks $T = T_1, T_2 \ldots, T_n$

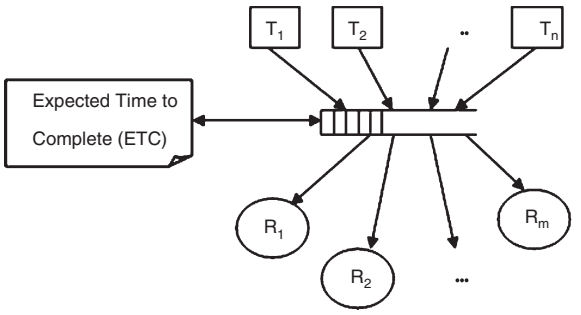

Fig. 1. View of the scheduling problem.

that should be mapped by the broker to m available resources $R = R_1, R_2, \ldots, R_m$ in a way that minimizes makespan. This problem is illustrated in Fig. 1. In this figure, the completion time for each task T_i on each resource R_j is calculated dynamically in each step and is entered in a matrix ETC at index $[i, j]$. This matrix is available to the broker. $Ready[j]$ is the time that resource R_j needs to complete its previous task. The maximum of the sums of these times is the makespan of a solution, as defined in equation (1). It is assumed that every task is executing on a resource and is not mapped to any other machine until its execution is stopped.

$$Makespan = \max \sum_{i=1}^{n} (ready[j] + ETC[i, j]) \quad J = 1, \ldots, m \tag{1}$$

The running time of each task for each resource must be calculated for the purpose of scheduling. If the processing speed of resource R_j is PS_j, then the processing time for task S_i can be calculated by equation (2):

$$T_{ij} = C_i / PS_j \tag{2}$$

where T_{ij} is the processing time for task S_i by resource R_j and C_i is the computational complexity of the task S_i [1]. The values obtained from Equation (2) are stored in the ETC matrix.

4. The ICA

In this section, we describe ICA in detail. Imperialism is the policy of expanding the power and influence of a country outside its known territory. A country can control another country by direct legislation or through indirect methods such as controlling raw materials and goods. Imperialism has been a pervasive phenomenon in shaping world history. Imperialism in its first stages is political–military influence exercised in other countries in order to use their political, human, and mineral resources. Sometimes imperialism is practiced to prevent any influence of competitor imperialist countries. Imperialist countries engage in intensive competitions to colonize the colonies of other imperialist countries. This competition in turn results in the improvement and expansion of imperialist countries from the political, military, and economic points of view, because countries have to expand in order to make competition possible. Initially, imperialists only want to increase their power by using the human and mineral resources of their colonies, and it is not important whether their colonies improve or not. But later, because of increased international relations and growth of populations, imperialists need some kind of public support in order to continue their influence. To this end, imperialists begin to develop and improve their colonies. Thus, colonies see improvements in their economic, human, and social areas because these are necessary for the imperialists.

The ICA proposed by Atashpaz [6] was inspired by the mathematical modeling of imperialist competitions. Figure 2 shows a flowchart for the algorithm. Like other evolutionary algorithms, this algorithm begins with a number of initial random populations, each of which is called a *country*. Countries are possible solutions to the problem at hand and are equivalent to chromosomes in GA and particles in PSO. A number of the best elements of the population are selected as imperialists. The remainder of the population is considered to be comprised of colonies.

Figure 3 shows the initial empire formation. By applying an assimilation policy in the direction of various optimization axes, imperialists gain the favor of their colonies. The total power of each empire is modeled as the sum of the imperialist power and a percentage of the mean power of its colonies. After the initial formation of empires, imperialistic competition starts among them. Any empire that has no success in the imperialistic competition with nothing to add to its power is eliminated from the competition. So the survival of an empire depends on its power to assimilate competitors' colonies. As a result, the power of greater empires is gradually increased in imperialistic competitions and weaker empires will be eliminated. Empires have to make improvements in their colonies in order to increase their power. For this reason, colonies will eventually become like empires from the point of view of power, and we will see a kind of convergence. The stopping condition of the algorithm is having a single empire in the world.

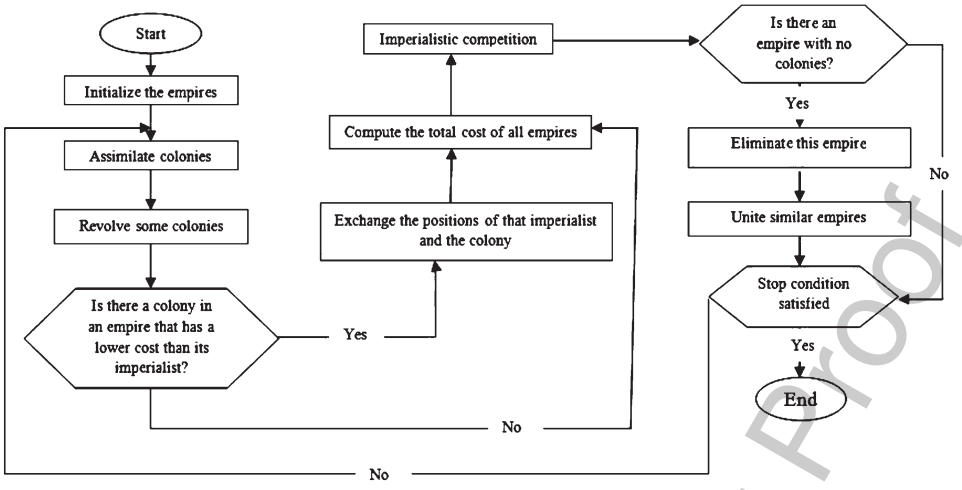

Fig. 2. ICA flowchart.

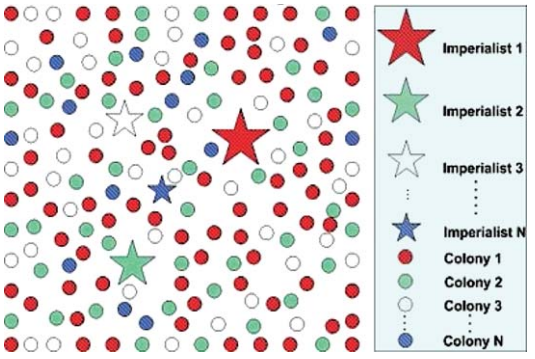

Fig. 3. Formation of the empire.

T_1	T_2	T_3	T_4	T_5	T_6	T_7	T_8	T_9
R_3	R_3	R_2	R_1	R_2	R_3	R_1	R_1	R_3

Fig. 4. An example solution for a problem with nine tasks running on three resources.

5. Discrete ICA for independent task scheduling

The independent task scheduling problem includes n tasks and m machines, where each task must be processed by each of the machines in a way that minimizes makespan. The original version of ICA is defined with real values, for contiguous spaces. But the independent task scheduling problem is discrete. Therefore, a discrete version of ICA [31] is required. In this version, ICA is transformed from contiguous states to discrete states by changing the assimilation stage and the way the solution is represented.

5.1. Generation of initial population

The initial population of countries in the ICA is generated randomly. Each country is a $1 \times n$ array, where n is the number of input tasks. The values in each country are resource numbers. Figure 4 shows an example solution to the independent task scheduling problem. In this example, nine tasks are considered for execution, and tasks T_1, T_2, T_6, and T_9 are run on resource R_3:

The cost of each country is calculated with the fitness cost function in equation (3).

$$fitness\,(country) = makespan\,(country) \quad (3)$$

According to the cost function, the lower a country's makespan is, the more appropriate is the solution it represents for solving the scheduling problem.

At the outset, a number $N_{country}$ of countries are produced, and a number N_{imp} of the best members of this population (countries with the least cost function values) are selected as imperialist countries. The remaining N_{col} countries are colonies, each of which belongs to an empire. The colonies are divided between the imperialists proportionally to the imperialists' power. To do this, the normalized cost C_i of each imperialist i is computed based on the cost of all imperialists, through equation (4):

$$C_i = max_j(c_j) - c_i \quad (4)$$

where c_i is the cost of imperialist i and $max_j\ (c_j)$ is the maximum cost of the imperialists. Imperialists with

the highest costs (weaker imperialists) have lower normalized costs. Using the normalized costs, the relative normalized power P_i of each imperialist is calculated using equation (5); these values are used to proportionally divide the colonies among the imperialists.

$$p_i = \left| \frac{c_i}{\sum_{j=1}^{N_{imp}} c_j} \right| \tag{5}$$

The number $N \cdot C_i$ of initial colonials of an imperialist i is then defined as in equation (6):

$$N \cdot C_i = round(P_i \times N_{col}) \tag{6}$$

where round is a function that yields the closest integer to a decimal number. The initial number of colonies for each imperialist is randomly selected. Given the initial state of the imperialists, the imperialistic competition begins. The evolution process continues until the stopping condition is satisfied. It is obvious that in the division of colonies, more powerful imperialists will have more colonies.

5.2. Colonies moving toward (Assimilation)

Historically, the assimilation policy was developed with the purpose of moving the culture and social structure of colonies toward the culture of the central government. Depending on how a country is represented for solving an optimization problem, the central government can apply an assimilation policy to try to make its colonies similar to it in various ways. This part of the colonization process in an optimization algorithm models colonies moving toward the imperialist country's culture. Specifically, an operation is applied to make part of the colonies' structures the same as the imperialist's structure. This operation is shown in Fig. 5 and is implemented as follows:

1. First, some cells (approximately 40%)[2] of the Imperialist array are randomly selected (cells 1, 4, 8, and 9 in the figure).
2. The selected cells are directly copied into the New-Colony array at the same indexes.
3. The remaining cells of the New-Colony array are copied from the Colony array at the same indexes (cells 2, 3, 5, 6, and 7 in the figure).

	T_1	T_2	T_3	T_4	T_5	T_6	T_7	T_8	T_9
Imperialist	R_3	R_3	R_2	R_1	R_2	R_3	R_1	R_1	R_3
Colony	R_1	R_2	R_3	R_3	R_2	R_1	R_2	R_2	R_2
New-Colony	R_3	R_2	R_3	R_1	R_2	R_1	R_2	R_1	R_3

Fig. 5. Modification that moves colonies toward imperialist.

5.3. Revolution operation

To perform this operation, two cells are first randomly selected in the colony and their values are exchanged. This stage (random exchange or revolution operation) is repeated based on a percentage of the total number of tasks; this percentage is indicated by the %Revolution parameter. If the new colony is better than the old colony, it replaces the old colony; otherwise, this procedure is repeated. This operation is illustrated in Fig. 6.

5.4. Position exchanges of colony and imperialist

While moving toward the imperialist country, some colonies may reach a superior status compared to the imperialist (with a cost function value two points less than the imperialist's cost function value). In this case, the imperialist and the colony exchange their positions, and the algorithm continues with the new imperialist country applying the assimilation policy to its newly acquired colonies. The colony and imperialist's position exchange is shown in Fig. 7. In this figure, the best imperial colony that has a lower cost than the imperialist

	T_1	T_2	T_3	T_4	T_5	T_6	T_7	T_8	T_9
Colony	R_1	R_2	R_3	R_3	R_2	R_1	R_2	R_2	R_2
New-Colony	R_2	R_2	R_3	R_3	R_2	R_1	R_2	R_1	R_2

Fig. 6. Example of one revolution operation for resolving task scheduling.

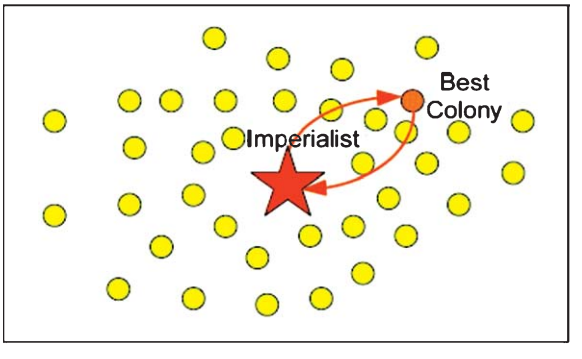

Fig. 7. Exchanging the position of a colony and the imperialist.

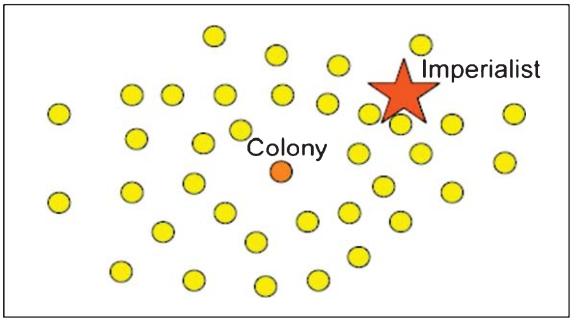

Fig. 8. The entire empire after position exchange.

is shaded. Fig. 8 shows the entire empire after the exchange.

5.5. Total power calculation in an empire

The power of an empire is the power of the imperialist country plus some percentage of the power of all its colonies. Thus, the total cost $T \cdot C_i$ of the i-th empire is defined as in equation (7):

$$T \cdot C_i = cost(imperialist_i) + \zeta\, mean(cost(colonies\ of\ empire_i)) \quad (7)$$

where ζ is a positive real number between 0 and 1. Using a small value for ζ leads to the total cost of an empire being equal to cost of its central government (imperialist country), while increasing ζ results in increasing the effect of colonies' costs on the empire's total cost.

5.6. Imperialistic competition

As noted earlier, each empire that fails to increase its power will be eliminated in imperialistic competitions. This elimination occurs gradually. Over time, powerless empires lose their colonies (usually one colony at a time), and more powerful empires take possession of these colonies and increase their own power. In each iteration of the algorithm, one or more of the most powerless colonies of an empire are selected, and a competition for possession of these colonies takes place among all empires. Possession of these colonies won't necessarily go to the most powerful empire, but more powerful empires have greater chances of taking possession. To model the competition among empires for possession of these colonies, each empire's normalized total cost $N \cdot T \cdot C_i$ is first calculated according to equation (8), based on the empire's total cost $T \cdot C_i$ and the maximum empire cost $max_j(T \cdot C_j)$:

$$N \cdot T \cdot C_i = \max_j(T \cdot C_j) - T \cdot C_i \quad (8)$$

Empires with lower total costs will have higher normalized total costs. Each empire's probability P_{pi} of taking possession (which is proportional to the empire's power) under the competition for colonies is then calculated through equation (9):

$$P_{pi} = \left| \frac{N.T.C_i}{\sum_{j=1}^{N_{imp}} N.T.C_j} \right| \quad (9)$$

Given the probability of possession for each empire, a mechanism such as the roulette wheel in GA is needed so that in the competition for colonies, the probability of a colony being assigned to an empire is proportional to the empire's power. But because the roulette wheel has a high computational cost, a less costly method proposed by Atashpaz [6] is used here. Because colonies are randomly divided among empires, a $1 \times N_{imp}$ vector P containing the empires' probabilities of possession is first defined as in equation (10):

$$P = [P_{p1}, P_{p2}, \ldots, P_{pNimp}] \quad (10)$$

Then a vector R of the same size as P is filled with random numbers with a uniform distribution in the range [0, 1], as in equation (11):

$$R = [r_1, r_2, \ldots, r_{Nimp}] \quad \text{where } r_1, r_2, \ldots, r_{Nimp} \sim U(0, 1) \quad (11)$$

A vector D is then formed as in equation (12):

$$D = P - R = [D_1, D_2, \ldots, D_{Nimp}] = [P_{p1} - r_1, P_{p2} - r_2, \ldots, P_{pNimp} - r_{Nimp}] \quad (12)$$

The colonies are given to the empire corresponding to the index in vector D that contains the greatest value.

5.7. Eliminating powerless empires

As already mentioned, powerless empires are gradually eliminated in the imperialistic competitions, and more powerful empires are given possession of their colonies. In the present algorithm, an empire is eliminated when it loses all of its colonies and it then becomes an object of competition among the remaining empires.

5.8. Termination condition

The convergence condition proposed in this paper is that the total number of iterations has completed or all but one of the empires has fallen. In either case, the imperialistic competition ends.

6. Performance evaluation

In this section, we stress that experimental structures are possible under the following circumstances.

6.1. Experimental setup

The two metrics used to evaluate the scheduling approaches are the makespan constraint and execution time. The former indicates whether the schedule produced by the scheduling approach achieves the required time, while the latter indicates how long it takes to schedule the independent tasks on the test bed. Our QOS results are therefore evaluated based on these metrics. Here we simulate four algorithms (SA, GA, GSA[29], and GGA[30]) in addition to ICA for various states.

We have used a sample configuration named *GR* [10] for our tested tasks. The features of resources are alike so that we could meaningfully compare the various scheduling algorithms. Each resource has one processor. Table 1 shows this configuration for ten resources.

Here, only ten resources are considered. But we used different resource configurations based on GR. The performance measurements used are mean makespan and average runtime for each algorithm. All of the algorithms were implemented in a Java environment on a 2.66 dual core CPU with a 4-GB RAM. First, it was important to determine the values of some required parameters for this evaluation. The best parameter values, determined by several tests, are listed in Table 2.

Table 1
GR Resource configuration

Resource ID	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10
Processing Speed (MIPS)	8	4	7	6	10	3	9	5	7	11

Table 2
Tuned values for the algorithms

Algorithm	Parameter	Value
ICA	ζ	0.1
	P-Revolution	0.3
GA	P-Crossover	0.85
	P-Mutation	0.02

We have made several assumptions for the grid task and resource modeling, as follows. We considered tasks that are independent and produced stochastically. Each task has a certain length, measured in terms of a million instructions (MI); these lengths vary and are uniformly distributed (a range (Min . . . Max) is selected for task/job lengths and then a stochastic number is selected in this so that task lengths are uniformly distributed). For the most homogenous state, the range of task lengths is (100000 . . . 110000) instructions. In this paper, we considered task lengths as multiples of 1000MI for convenience. Therefore, we express this range as (100 . . . 110).

Since the most important aspect of scheduling algorithms is execution time on the resources, the, network delay for handling files is not considered. Also, the tasks we consider are completely computational; at the outset, they receive their required values from an input file and at the end, they place the final results in an output file. Thus, regardless of their interactions, only their computational aspects are discussed.

We assume a similar structure for all resources in the experiments, so that we can compare the schedulers for similar parameters. All resources have a single processor. Each resource has its own processing power (execution rate in terms of a million instructions per second, MIPS). With this basis, we have simulated our proposed method and all the other algorithms and have compared the results. These results are in some cases close to one another, so we have used the student's *t*-test for sample pairs, which shows with high confidence that the differences between the measurements in the results are greater than a statistical error. We used a standard queue (FIFO data structure) for each resource. Whenever a request arrives; it is placed in the queue of an appropriate resource based on its QOS. We consider the deadline for each task, and whenever a request (task) is assigned to a resource, it uses the resource non-preemptively until it completes its execution.

6.2. Experimental results

The results of a simulation of the proposed method on a Netbean 6.8×86 simulators are compared to the results of other methods. We have used similar data structures for all algorithms and we had no cache misses for the algorithms. The difference between the algorithms is only found in the processing they perform to find an optimal way to execute tasks on the resources. Tables 3 and 4 show the makespan and execution time results. The values are the arithmetic average of three

Table 3
Makespan evaluation

Iter.	(pop, task, resource)	SA	GA	GSA	GGA	ICA
100	(50,50,10)	124.32	81.81	80.99	82.42	80.97
	(100,50,20)	97.63	50.94	48.5	49.73	47.46
	(200,50,30)	69.97	35.05	34.33	36.45	34.05
	(50,100,10)	293.22	162.16	158.67	164.88	156.58
	(100,100,20)	155.53	90.71	87.24	90.75	86.6
	(200,100,30)	115.97	66.72	V.L.	68.22	62.51
	(50.200,10)	583.55	329.36	316.66	317.78	313.6
	(100,200,20)	325.97	177.34	166.54	175.55	169.55
	(200,200,30)	269.33	122.11	V.L.[1]	121.51	117.8
200	(50,50,10)	115.34	82.28	82.95	81.91	81.96
	(100,50,20)	83.95	45.75	45.14	47.08	44.36
	(200,50,30)	67.39	34.94	32	35.71	33.45
	(50,100,10)	285.8	158.33	156.03	159.48	158.1
	(100,100,20)	181.83	88.46	86.19	85.13	85.3
	(200,100,30)	130.39	V.L.	V.L.	62.93	64.7
	(50.200,10)	656.32	311.97	314.03	320.34	310.35
	(100,200,20)	275.75	168.17	162.69	170.17	172.94
	(200,200,30)	246.5	V.L.	V.L.	V.L.	121.3
300	(50,50,10)	111.91	81.18	81.37	81.27	80.76
	(100,50,20)	73.75	45.76	45.09	45.75	44.56
	(200,50,30)	69.78	34.44	V.L.	34.16	34.18
	(50,100,10)	234.72	157.98	157.31	158.56	157.25
	(100,100,20)	169.86	86.08	83.14	84.43	86.56
	(200,100,30)	128.37	60.28	V.L	59.23	62.12
	(50.200,10)	566.16	313.21	310.19	313.5	309
	(100,200,20)	307.25	166.98	166.28	163.38	167.16
	(200,200,30)	256.22	V.L.	V.L.	V.L.	123.92

[1]Very Large (with respect to the grid environment, time played a crucial role). Here the makespan values have risen enormously and are not suitable for the grid environment. Therefore, we consider them as Very Large, abbreviated as V.L. in the table.

recursive executions of each method using the aforementioned parameters.

Table 3 shows the makespan for the SA, GA, GSA, GGA, and ICA. We can separate the tasks into three groups: (1) soft tasks, of which there are at most 50; (2) medium tasks, of which there are approximately 100; and (3) heavy tasks, of which there are more than 200. We tested with exactly 50, 100, and 200 tasks. We used three sizes for the population: 50, 100, and 200; we tested all methods for 100, 200, and 300 iterations; and we considered 10, 20, and 30 resources.

Table 4 shows the execution times for the SA, GA, GSA, GGA, and ICA methods.

Figure 9 demonstrates the various algorithms' makespan improvements for 100, 200, and 300 iterations. As shown, ICA has better results for most of the states, and the makespans it finds are less than those of the other algorithms under these conditions. For example, with a population of 200, 50 independent tasks, 30 resources, and 100 iterations (200, 50, 30; iterations 100), our proposed method's makespan is approximately 34.05 units, whereas SA's makespan is almost 69.97 units and GSA's is approximately 34.33 units, the best makespan other than ICA's. Hence, our proposed method achieved better results than the others.

Figure 10 illustrates the state (50, 200, 10) makespans for 100, 200, and 300 iterations of the algorithms. We tested each of the various states and iterations several times and normalized the results, to easily show the differences. We omitted the SA algorithm from this figure because its makespan differed considerably from the other methods for the different numbers of iterations. ICA and GSA are the only two methods whose makespans decrease in a continuously decreasing slope. The decreasing slope is $\frac{310.19-316.66}{3-1} = -3.235$ for GSA and $\frac{309-313.6}{3-1} = -2.3$ for ICA.

Figure 11 depicts the time consumed by these algorithms for this state. ICA's execution time is much less than GSA's and is close to 0. ICA took approximately 1 second to execute for the various numbers of iterations. On the other hand, the other algorithms took more than 1 minute to find the best solution for scheduling 200 tasks on 10 separate resources for a population of 50. In this case, GSA was the worst algorithm, because its runtime increases rapidly as the number of iterations increases; hence, this is not a good algorithm for scheduling.

Table 4
Execution time evaluation

Iter.	(pop, task, resource)	SA	GA	GSA	GGA	ICA
100	(50,50,10)	0	45.66	89.66	32.33	2
	(100,50,20)	0.33	181.66	193	127	4.33
	(200,50,30)	0	895.33	2240	553.33	10
	(50,100,10)	0.33	90.33	157.33	58	2.66
	(100,100,20)	0	391.66	340	253.66	6.66
	(200,100,30)	0	1696.33	3330	1040.33	15.66
	(50.200,10)	0.33	158	266.33	114.66	4.33
	(100,200,20)	0.33	751	1402	508.33	12
	(200,200,30)	0	3138.33	V.L	2051.66	25.33
200	(50,50,10)	0	81.66	123.66	62.33	2.66
	(100,50,20)	0	471.66	538.33	278	4.33
	(200,50,30)	0	1538.66	2166	1115.66	13
	(50,100,10)	0.33	150.66	224	114.33	2.33
	(100,100,20)	0.33	721.66	1021	519.33	9.33
	(200,100,30)	0.33	V.L	V.L	2100	16.66
	(50.200,10)	0.33	298.33	431.66	223	4.66
	(100,200,20)	0.33	1394.66	1987	1011.66	11.66
	(200,200,30)	0.33	V.L.	V.L.	V.L.	24
300	(50,50,10)	0.33	149.66	185	93.33	1.66
	(100,50,20)	0.33	713	811	408.66	5
	(200,50,30)	0.33	2395.33	V.L	1726.33	11
	(50,100,10)	0.33	251.66	345.33	174	3.66
	(100,100,20)	0.33	1117.66	1527.33	174	8.33
	(200,100,30)	0.33	4619	V.L	3217.33	16
	(50.200,10)	0.66	449.66	662	325.66	4.66
	(100,200,20)	0.33	2296	2988	1525.33	18.33
	(200,200,30)	0.66	V.L.	V.L.	V.L.	27.33

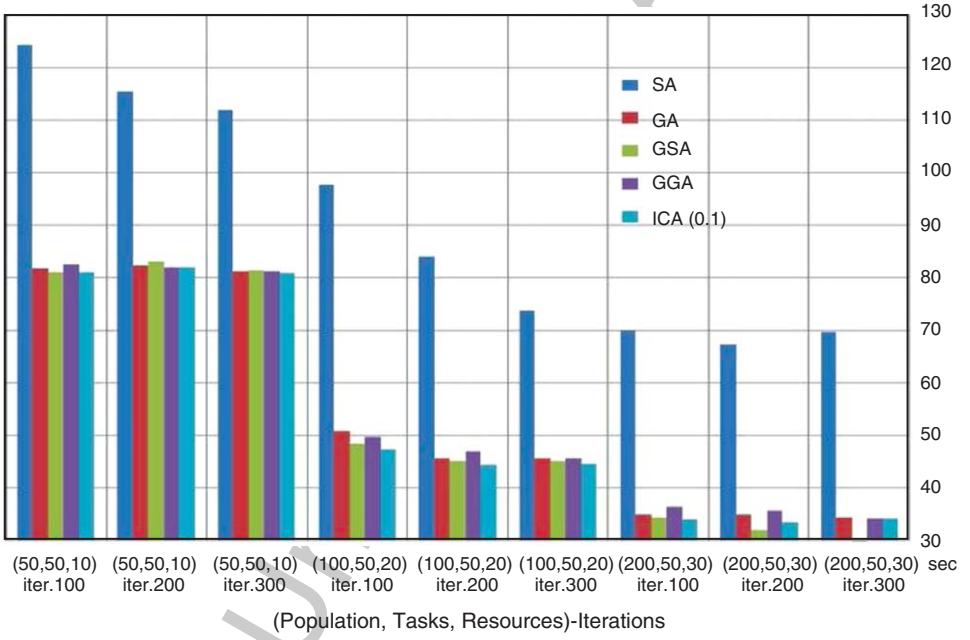

Fig. 9. Makespans for the different algorithms.

As shown in Table 5, as the population and resources increased, the makespan found by all of the algorithms except GSA decreased. For all populations, ICA is the best solution for scheduling because of its shorter makespan compared to the others. We did not consider SA while varying the population because SA has only

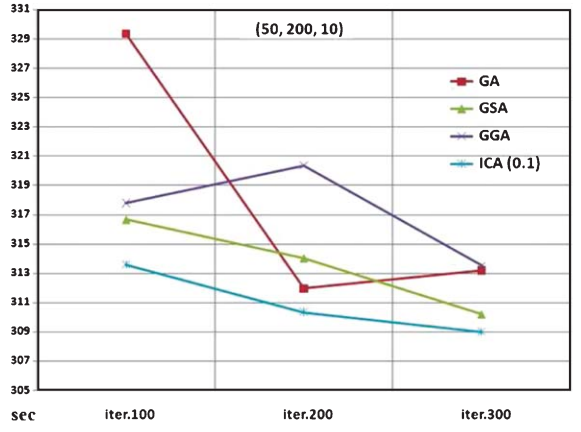

Fig. 10. Makespans for the state (50, 200, 10) for 100, 200, and 300 iterations.

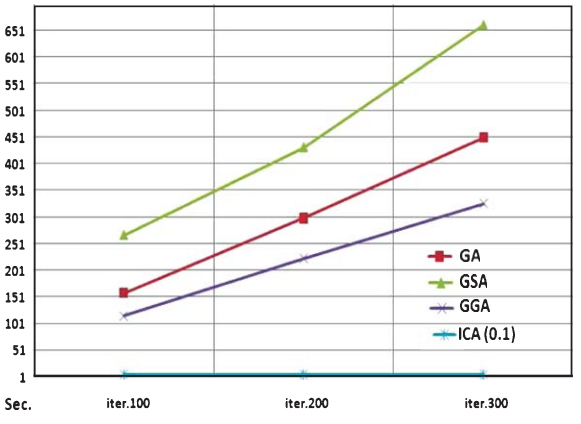

Fig. 11. Runtimes for the state (50, 200, 10) for 100, 200, and 300 iterations.

Table 5
Makespans for different populations and resources

No. of iterations	(Pop., Task, Res.)	GA	GSA	GGA	ICA
300	(50,50,10)	81.18	81.37	81.27	80.76
300	(100,50,20)	45.76	45.09	45.75	44.56
300	(200,50,30)	34.44	Very large	34.16	34.18

one population, so that population changes do not affect the makespan it finds. Figure 12 shows the runtimes for different populations with 300 iterations for 50 tasks.

Figure 12 shows that ICA takes at most approximately 11 seconds to execute for a population of 200, while GSA, GA, and GGA take more than 1300 seconds (i.e., more than 20 minutes) to find the best solution for scheduling 50 independent tasks with 300 iterations. Hence, these methods are less suitable for time-independent requests in grid computing. Also, the

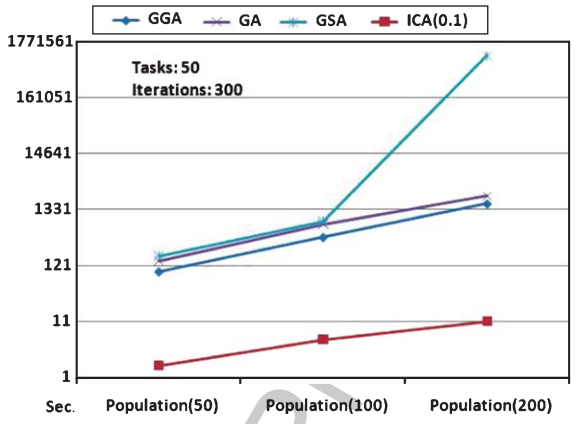

Fig. 12. Runtimes for 50 independent tasks with 300 iterations.

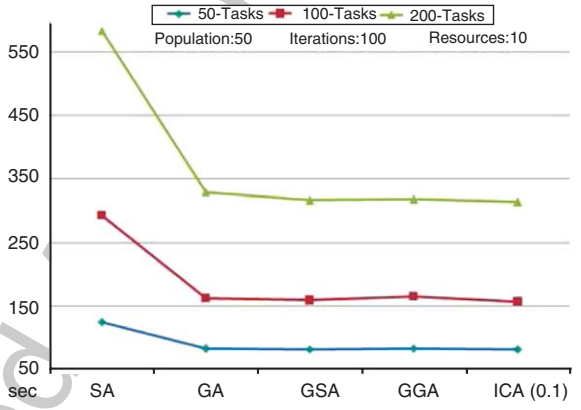

Fig. 13. Makespans for task variations with 100 iterations and 10 resources with a population of 50.

increase in execution time when the population reaches four times its original size of 50 is approximately 11 times for ICA, more than 10000 times for GSA, more than 18 times for GGA, and more than 16 times for GA. Thus, ICA is the best method in terms of execution time for problems where population sizes increase significantly.

Figure 13 shows the makespans for various tasks for a population of 50, with 100 iterations, and 10 resources for all of the algorithms. ICA finds a shorter makespan for all three types of tasks (thin, medium, and heavy) than the other methods do. When the number of tasks increases from 50 to 200, the makespan found by SA increases rapidly from just less than 150 units (124.32 units) to more than 550 units (583.55 units). Hence, SA is not appropriate for heavy tasks. All of the algorithms except SA are good for medium tasks and thin tasks because they have approximately the same makespan.

Table 6
Algorithm runtimes for 100 iterations, population of 50, and 10 resources (in seconds)

Tasks	SA	GA	GSA	GGA	ICA
50	0	45.66	89.66	32.33	2
100	0.33	90.33	157.33	58	2.66
200	0.33	158	266.33	114.66	4.33

Table 6 shows that SA is faster than the other algorithms, but considering that its makespan is the worst (Fig. 13), we should select the second fastest algorithm. This is the ICA. When the number of tasks increases from 50 to 200, the runtime for ICA increases by 2.33, whereas for GSA it increases by 176.67 seconds, for GA it increases by 112.36 seconds and for GGA it increases by 82.33 seconds. Thus, ICA is the fastest of these algorithms.

7. Conclusion

Grid computing involves a collection of resources that are geographically distributed and that share applications, data, and computational resources with one another. Computational grids have been proposed as a new computational model for solving large, complicated problems in various domains. Since resources are distributed in computational grids, they have very dynamic communication lines and computational beds. The conditions of the computational resources are highly variable, and communication lines have delays and are unreliable. Thus, resource management in computational grids should have the necessarily ability to adapt with the dynamic states of the environment. Task scheduling in computational grids should distribute the workload among available resources in the most efficient way. Various models, methods, and algorithms have been proposed for task scheduling in grid environments.

This paper proposes the use of ICA, an evolutionary optimization algorithm based on modeling the colonization process, for solving the independent task scheduling problem for grid systems. The performance of ICA was compared with SA, GA, GSA and GGA algorithms that have also been proposed for grid scheduling, and the simulations show that ICA finds a shorter makespan than the other algorithms and also has faster computations for finding the optimal solution. On the other hand, ICA compares to classical optimization methods has less convergence speed because it escapes from the local optimum points and leads to converge slower than classical optimizations. As a result, it will be good we will join both these methods together to resolve both sides' problems in the future.

References

[1] M.A. Azgomi and R. Eetezari-Maleki, Task scheduling modeling and reliability evaluation of grid services using colored Petri nets, *Future Generation Computer Systems* **26**(8) (2010), 1141–1150.

[2] E.A. Duki, H.A. Mansoorkhani, A. Soroudi and M. Ehsan, A discrete imperialist competition algorithm for transmission explanation planning, *25th International Power System Conference* (2011), 1–10.

[3] A. Abraham, R. Buyya and B. Nath, Nature's heuristics for scheduling jobs on computational grids, *Proceeding of the 8th IEEE International Conference on Advanced Computing and Communications*, India (2000), 1–8.

[4] S.F. Attar, M. Mohammadi and R. Tavakkoli-Moghaddam, A novel imperialist competitive algorithm to solve flexible flow shop scheduling problem in order to minimize maximum completion time, *International Journal of Computer Applications* **28**(10) (2011), 27–32.

[5] E. Atashpaz-Gargari and C. Lucas, Designing an optimal PID controller using imperialist competitive algorithm, *First Joint Congress on Fuzzy and Intelligent Systems* (2007), 1–6.

[6] E. Atashpaz-Gargari and C. Lucas, Imperialist competitive algorithm: An algorithm for optimization inspired by imperialist competitive, *IEEE Congress on Evolutionary computation*, Singapore (2007), 4661–4667.

[7] J. Behnamian and M. Zandieh, A discrete colonial competitive algorithm for hybrid flowshop scheduling to minimize earliness and quadratic tardiness penalties, *Expert Systems with Applications* **38**(12) (2011), 14490–14498.

[8] S. Benedict and V. Vasudevan, Improving scheduling of scientific workflows using Tabu search for computational grids, *Information Technology Journal* **7**(1) (2008), 91–97.

[9] B.T.B. Khoo and B. Veeravalli, Pro-active failure handling mechanisms for scheduling in grid computing environments, *Journal of Parallel Distributed Computing* **70** (2010), 189–200.

[10] R. Buyya, The World-Wide Grid (WWG) (2002). http://www.buyya.com/ecogrid/wwg/

[11] R. Chang, J. Chang and P. Lin, An ant algorithm for balanced job scheduling in grids, *Future Generation Computer Systems* **25**(1) (2009), 20–27.

[12] R. Chen, D. Shiau and S. Lo, Combined discrete particle swarm optimization and simulated annealing for grid computing scheduling problem, *Lecture Notes in Computer Science*, Springer **57** (2009), 242–251.

[13] M. Cruz-Chavez, A. Rodriguez-Leon, E. Avila-Melgar, F. Juarez-Perez, M. Cruz-Rosales and R. Rivera-Lopez, Genetic-annealing algorithm in grid environment for scheduling problems, *Security-Enriched Urban Computing and Smart Grid Communications in Computer and Information Science*, Springer **78** (2010), 1–9.

[14] B. Eksioglu, S.D. Eksioglu and P. Jain, A tabu search algorithm for the flowshop scheduling problem with changing neighborhoods, *Computers & Industrial Engineering* **54**(1) (2008), 1–11.

[15] S. Garcia-Galan, R.P. Prado and J.E.M. Exposito, Fuzzy scheduling with swarm intelligence-based knowledge

acquisition for grid computing, *Engineering Applications of Artificial Intelligence* **25**(2) (2012), 359–375.

[16] H. Izakian, B. TorkLadani, K. Zamanifar and A. Abraham, A novel particle swarm optimization approach for grid job scheduling, *Communications in Computer and Information* **31** (2009), 100–109.

[17] A. Kazem, A.M. Rahmani and H.H. Aghdam, A modified simulated annealing algorithm for static scheduling in grid computing, *Proceedings of the 8th International Conference on Computer Science and Information Technology* (2008), 623–627.

[18] J. Kennedy and R. Mendes, Neighborhood topologies in fully informed and best-of-neighborhood particle swarms, *IEEE Transactions on Systems, Man, and Cybernetics* **36**(4) (2006), 515–519.

[19] J. Kołodziej and F. Xhafa, Meeting security and user behavior requirements in grid scheduling, *Simulation Modeling Practice and Theory* **19** (2011), 213–226.

[20] J. Kołodziej and F. Xhafa, Enhancing the genetic-based scheduling in computational grids by a structured hierarchical population, *Future Generation Computer Systems*, Elsevier **27**(8) (2011), 1035–1046.

[21] J. Kołodziej and F. Xhafa, Integration of task abortion and security requirements in GA-based meta-heuristics for independent batch grid scheduling, *Computers & Mathematics with Applications* **63**(2) (2012), 350–364.

[22] S.K. Garg, R. Buyya and H.J. Siegel, Time and cost trade-off management for scheduling parallel applications on utility grids, *Future Generation Computer Systems* **26**(8) (2010), 1344–1355.

[23] H. Liu, A. Abraham and A. Hassanien, Scheduling jobs on computational grids using a fuzzy particle swarm optimization algorithm, *Future Generation Computer Systems* **26**(8) (2010), 1336–1343.

[24] S.J.M. Rad, F.A. Tab and K. Mollazade, Application of imperialist competitive algorithm for feature selection: A case study on bulk rice classification, *International Journal of Computer Applications* **40**(16) (2012), 41–48.

[25] K. Nemati, S.M. Shamsuddin and M.S. Kamarposhti, Using imperial competitive algorithm for solving traveling salesman problem and comparing the efficiency of the proposed algorithm with methods in use, *Australian Journal of Basic and Applied Sciences* **5**(10) (2011), 540–543.

[26] Z. Pooranian, M. Shojafar, J.H. Abawajy and A. Abraham, An efficient meta-heuristic algorithm for grid computing, *Journal of Combinatorial Optimization (JOCO)* (2013), doi:10.1007/s10878-013-9644-6.

[27] Z. Pooranian, A. Harounabadi, M. Shojafar and J. Mirabedini, Hybrid PSO for independent task scheduling in grid computing to decrease makespan, *International Conference on Future Information Technology IPCSIT* **13**, Singapore (2011), 435–439.

[28] Z. Pooranian, A. Harounabadi, M. Shojafar and N. Hedayat, New hybrid algorithm for task scheduling in grid computing to decrease missed task, world academy of science, *Engineering and Technology* **79** (2011), 924–928.

[29] Z. Pooranian, M. Shojafar and B. Javadi, Independent task scheduling in grid computing based on queen-bee algorithm, *IAES International Journal of Artificial Intelligence (IJ-AI)* **1**(4) (2012), 171–181.

[30] Z. Pooranian, M. Shojafar, R. Tavoli, M. Singhal and A. Abraham, A hybrid meta-heuristic algorithm for job scheduling on computational grids, *Informatica* **37** (2013), 501–505.

[31] A.S. Mamaghani and M.R. Meybodi, An application of imperialist competitive algorithm to solve the quadratic assignment problem, *International Conference Internet Technology and Secured Transactions (ICITST)* (2011), 562–565.

[32] R. Shafaei, N. Moradinasab and M. Rabiee, Efficient meta heuristic algorithms to minimize mean flow time in no-wait two stage flow shops with parallel and identical machines, *International Journal of Management Science and Engineering Management* (2011), 421–430.

[33] E. Shokrollahpur, M. Zandieh and B. Dorri, A novel imperialist competitive algorithm for bi-criteria scheduling of the assembly flowshop problem, *International Journal of Production Research* **49**(11) (2011), 3087–3103.

[34] Q. Tao, H. Chang, Y. Yi, C.H. Gu and W. Li, A rotary chaotic PSO algorithm for trustworthy scheduling of a grid workflow, *Computers & Operations Research* **38**(5) (2011), 824–836.

[35] L.Y. Tseng, Y.H. Chin and S.C. Wang, The anatomy study of high performance task scheduling algorithm for grid computing system, *Computer Standards & Interfaces*, Elsevier **31**(4) (2009), 713–722.

[36] L. Wei, X. Zhang, Y. Li and Yu Li, An improved ant algorithm for grid task scheduling strategy, *Physics Procedia*, Elsevier **24** (2012), 1974–1981.

[37] C. Weng and X. Lu, Heuristic scheduling for bag-of-tasks applications in combination with QoS in the computational grid, *Future Generation Computer Systems* **21**(2) (2005), 271–280.

[38] F. Xhafa and A. Abraham, Computational models and heuristic methods for Grid scheduling problems, *Future Generation Computer Systems* **70**(3) (2010), 608–621.

[39] F. Xhafa, J. Gonzalez, K. Dahal and A. Abraham, A GA(TS) hybrid algorithm for scheduling in computational grids, *Hybrid Artificial Intelligence Systems Lecture Notes in Computer Science*, Springer **5572** (2009), 285–292.