Tuning Struggle Strategy in Genetic Algorithms for Scheduling in Computational Grids

Fatos Xhafa and Bernat Duran
Department of Languages and Informatic Systems
Technical University of Catalonia
Barcelona, Spain
{fatos,bduran}lsi.upc.edu
Ajith Abraham
Center of Excellence for Quantifiable Quality of Service,
Norwegian University of Science and Technology,
Trondheim, Norway
ajith.abraham@ieee.org
Keshav P. Dahal
School of Informatics, University of Bradford, Bradford BD7 1DP, UK
k.p.dahal@Bradford.ac.uk

Abstract

Job Scheduling on Computational Grids is gaining importance due to the need for efficient large-scale Grid-enabled applications. Among different optimization techniques addressed for the problem, Genetic Algorithm (GA) is a popular class of solution methods. As GAs are high level algorithms, specific algorithms can be designed by choosing the genetic operators as well as the evolutionary strategies. In this paper we focus on Struggle GAs and their tuning for the scheduling of independent jobs in computational grids. Our results showed that a careful hash implementation for computing the similarity of solutions was able to alleviate the computational burden of Struggle GA and perform better than standard similarity measures.

1. Introduction

With the emerging paradigm of Grid Computing and the development of Grid infrastructures, Grid-based applications are becoming a common approach for solving many complex problems. A key issue in this kind of applications is scheduling jobs into Grid resources efficiently, which is known to be computationally hard and much more difficult than its standard version for sequential or LAN computation environments.

Job Scheduling on Computational Grids is gaining importance due to the need for efficient large-scale Grid-enabled applications, e.g. in Optimization (Casanova et al. [1], Goux et al. [2] and Wright [3]), Collaborative/eScience Computing (Newman et al. [4], Paniagua et al. [5]) and many applications arising from concrete types of eScience Grids such as Science Grids, Access Grids, Knowledge Grids and in Data-Intensive Computing (Beynon al. [6]). Scheduling is a challenging problem in a Grid environment due its dynamic nature and the large number of resources to be managed and jobs to be scheduled. Furthermore, resources can have their own local policies (regarding access, cost etc.) to be taken into account. The problem is multi-objective in its general definition, as there are several optimization criteria to be matched, such as makespan, flowtime, and resource utilization.

Several approaches are being addressed in the literature for the problem (Abraham et al. [7], Braun et al. [8], Xhafa [9], Xhafa et al. [10, 11], Zomaya et al. [12]) aiming to obtain schedulers capable of delivering fast planning of jobs to computational resources of the grid system. In particular, Genetic Algorithms (GA) [13] have proved to be a good alternative for solving a wide variety of hard combinatorial optimization problems. GAs are a population-based approach where individuals represent possible solutions, which are successively evaluated, selected, crossovered, mutated and replaced by simulating the Darwinian evolution found in nature.

978-0-7695-3184-7/08 $25.00 © 2008 IEEE
DOI 10.1109/CISIM.2008.54

The research work on GAs has shown that a key issue in GAs is the convergence of the algorithm: a fast convergence of the population would stagnate the search to local optima whereas slower convergence would require a considerably longer time towards sub-optimal solutions. The convergence of GAs is achieved by means of selection and replacement strategies and it is, therefore, very important to carefully tune these strategies. In particular, the selective pressure directly affects the tradeoff between the exploration and exploitation of the search space. Indeed, if the population converges rapidly GA would give more priority to the exploitation and, vice-versa, when the population is kept diverse, other regions of the search space would be explored aspiring thus to find better solutions.

In this work, we focus on the importance of tuning the replacement mechanism of GA for scheduling in computational grids. The interest in investigating this aspect is motivated by the need to design efficient schedulers that will be able to deliver fast and quality planning of jobs to resources rather optimal solutions in a dynamic environment. More precisely, we study the tuning of the Struggle strategy (Grueninger [14]), which is known for its effectiveness but suffers from a high computational cost.

The rest of the paper is organized as follows. In Section 2, we briefly present the scheduling of independent jobs considered in this work. The Struggle strategy together with similarity measures are introduced in Section 3. The experimental study and some computational results are given in Section 4. We conclude in Section 5 with some remarks and indications for future work.

2. Problem definition

In this work we consider the problem formulation of job scheduling based on the Expected Time To Compute (ETC) simulation model of Braun et al. [8]. An instance of the problem consists of the following.

- A number of independent (user/application) jobs that must be scheduled. Any job has to be processed entirely in a unique resource (non-preemptive mode).
- A number of heterogeneous machines candidates to participate in the planning.
- The workload of each job (in millions of instructions).
- The computing capacity of each machine (in millions of instructions per second –*mips*).
- For any machine *m*, the time when the machine will have finished the previously assigned jobs (*ready[m]*). This parameter measures the previous workload of a machine.
- The Expected Time to Compute (*ETC*) matrix of size *number_jobs* × *number_machines*, where a position *ETC[i][j]* indicates the expected execution time of job *i* in machine *j*. This matrix is either computed from the information on workload and mips or can be explicitly provided.

We aim to minimize the *makespan* and *flowtime* and utilize the resources effectively. These parameters are defined as follows.

- *Makespan* is the finishing time of latest job:
$$\min_{schedule} \max \{F_j: j \in Jobs\} \quad (1)$$
where F_j is the finishing time of job *j*.

- *Flowtime* is the sum of finishing times of jobs:
$$\min_{schedule} \sum_{j \in Jobs} F_j \quad (2)$$

The aforementioned parameters are very important for studying and characterizing any Grid system. Makespan measures the productivity (throughput) of the grid system whereas the flowtime measures the QoS of the Grid system.

3. Struggle strategy in GAs and similarity measures

In Struggle GAs (hereafter, SGA) [14], a new generation of individuals is created by replacing only a portion of the population with the new individuals. Unlike other replacement strategies, in SGA, a new individual replaces the individual that is most similar to it only in case the new individual obtains a better fitness value than the one to be replaced. This is done in order to adaptively maintain certain diversity among the population. The aim is to preserve the optimization velocity but delaying its tendency to converge in order to reach a better convergence point.

This strategy has shown to be very effective for many problems yet, there is a critical issue here: the computational cost of this replacement strategy is very high. Indeed, in order to find which individuals should leave the population, any new individual of the intermediate population has to be compared and its similarity measured against all the individuals of the current population. Obviously, this leads to a quadratic order computational time, which could be very large, if large size populations were to be considered. In fact, this is precisely the case of scheduling independent jobs in computational grids; their large scale and scalability are critical factors since not only the number of resources and jobs submitted to the Grid system are expected to be large or very large but also they could

increase over time. It is clear that similarity measures which are not efficient could consume much of the GA running time in detriment to the proper search time.

3.1. Standard similarity measures

In order to compare the similarity between solutions, a measure of similarity or distance function has to be defined and used. Standard similarity measures include:

- *Hamming distance*: given two individuals *S1* and *S2* encoding two schedulings of *N* jobs, let *g[i]=1*, iff *S1[i] = S2[i]* and *g[0]=0*, otherwise. Similarity is then calculated as $Sim_h(S1,S2)=\sum_{(i=1..N]} g[i]/N$.
- *Euclidian distance*: This similarity is based on the Euclidean distance. Given two vector solutions *S1* and *S2*, by considering them as two points in *N*-dimensional space, the similarity is then computed as the Euclidean distance between them:

$$Sim_e(S1,S2) = \sqrt{\sum_{i=1}^{N}(S1[i]-S2[i])^2} \qquad (3)$$

- *Cosine distance*: In this case, the similarity is measured using the angle of the two vector solutions *S1* and *S2* of the *N*-dimensional space. Cosine values close to 1 would mean more similarity.

$$Sim_c(S1,S2)=\frac{\sum_{i=1}^{N} S1[i]\cdot S2[i]}{\sqrt{\sum_{i=1}^{N} S1[i]^2}\sqrt{\sum_{i=1}^{N} S2[i]^2}} \qquad (4)$$

3.2. Hash-based similarity measure

The standard similarity measures given above has linear time computational cost in number of jobs. Therefore for a population of *pop_size* the standard struggle strategies would take $O(intermediate\_pop\_size \times pop\_size) \times N$. Reducing the quadratic factor of $O(intermediate\_pop\_size \times pop\_size)$ to a linear time factor would be very desirable in this case since in each replacement step it would take a considerable time in detriment to the proper search time of the GA. In order to achieve this, we propose the use of hash techniques so that given a new individual of the intermediate population we can find in constant time the individual most similar to it.

In order to design the hash table, we have to first define the *key* to identify the individuals of the population. The key information is the basis for computing the degree of similarity of the *struggle* genetic operator: the more accurate its definition the better the performance of the operator. In fact, a poor definition of the key would simply reduce the struggle operator to a random replacement. In our definition of the key the context is crucial: the key value should resume as much as possible the genetic information encoded in an individual; hence, if two key values are similar then their respective individuals are genetically similar. The following are three possible definitions:

a) *Fitness-based key*: consists in using the fitness value, which is transformed, using a hash function, into the key value. Certainly this is a very simplistic approach by simply looking at *makespan* and *flowtime* values and clearly no genetic information is taken into account (we refer to this as 'a' key).

b) *Position-based key*: having the permutation vector of task-resource allocation, in which tasks are sorted according to the resource they are assigned to, the key is defined as the sum of number of cells a component of the vector would move to the right as indicated by its value, when the vector is read in a circular way (we refer to this as 'b' key). For instance, for the vector of 7 tasks in Figure 1 below, *key*=2+4+1+0+2+5+0=14.

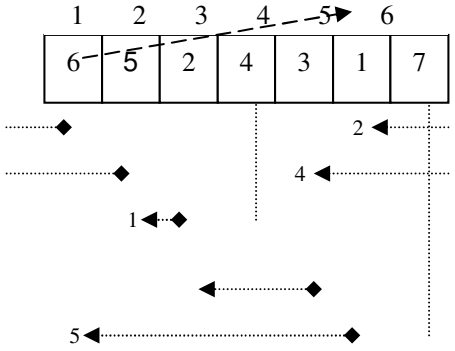

Figure 1. Example position-based key calculation.

Note that this definition uses the genetic characteristics of the solution; however, the relation task-resource is not explicitly taken into account, i.e., to which resource is assigned a task.

c) *Task-resource allocation key*: In this case both information on tasks and resources is used. The key value is now the sum of the absolute values of the subtraction of each position and its precedent in the vector of task-resource allocation (reading the vector in a circular way); we refer to this as 'c' key.

The hash function is then defined as:

$$f_{hash}(k) = \begin{cases} 0, & \text{if } k < k_{min} \\ \left\lfloor N\left(\frac{k - k_{\min}}{k_{\max} - k_{\min}}\right)\right\rfloor, & \text{if } k_{min} \le k < k_{max}, \\ N\text{-}1, & \text{if } k \ge k_{max} \end{cases} \quad (5)$$

where k_{min} and k_{max} correspond to the smallest and largest key value, respectively, in the population. The hash table has the same size as the population in order to obtain constant time access (in average). If an access fails, a few individuals in the population are randomly chosen and the most similar to the new one is considered for the replacement. Hence, the constant access is always ensured even if a failed access occurs. Therefore, the computational cost of the *struggle* operator is O(*pop_size* + *intermediate_pop_size*).

4. Experimental Results

In this Section, we present the experimental study of the proposed hash-based Struggle GA. Initially we generated a set of instances according to ETC matrix model in order to study the performance of the three key definitions and also to fine tune the rest of the parameters of Struggle GA. The best resulting configuration was then used for studying the performance of the SGA on a set of known instances from Braun et al. [8].

4.1. Performance comparison of struggle hash operators

The performance of the three struggle operators resulting from Key_a, Key_b and Key_c definitions were measured for makespan value of the schedule. For any of them, the same configuration of parameters (see Table 1 below) was used.

Table 1: Parameter configuration of Struggle GA.

nb_evolution_steps	*(max 90s)*
pop_size	*10*
intermediate_pop_size	*6 (60%)*
cross_probability	*0.9*
mutate_probability	*0.4*
start_choice	*MCT and LJFR-SJFR*
select_choice	*N tournament*
cross_choice	*Fitness Based Crossover*
Mutate_choice	*rebalance-both*

In Table 1, MCT (Minimum Completion time) and LJFR-SJFR (Longest Job to Fastest Resource – Shortest Job to Fastest Resource) are two methods used in initializing the population; rebalance-both is a mutation operator based on load balancing of resources.

We show in Figure 2, the makespan value computed by the SGA algorithm with Hamming distance measure (denoted SGA struggle quadratic) of similarity against the SGA+ a key, b key and c key implementations.

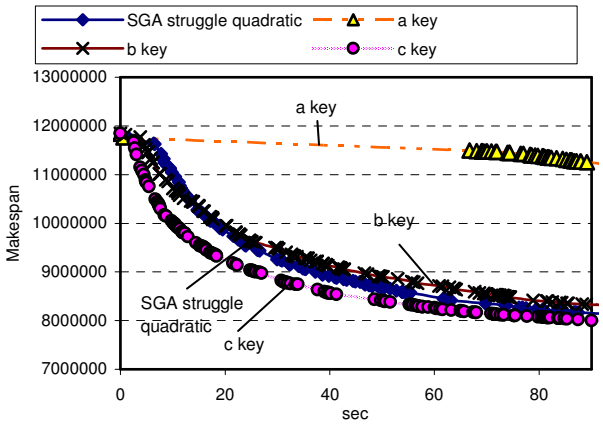

Figure 2. Comparison of makespan values (in arbitrary time units) obtained with four versions of SGA.

As evident from Figure 2, the a key implementation of SGA performs very poorly –it's not able even to mimic the behavior of SGA struggle quadratic– recall that the measure of similarity related to the a key is just the fitness. On the other hand, b key and c key behave coherently. As expected, the c key outperforms both the SGA struggle quadratic and the b key implementation. It's worth observing that the c key implementation achieves a faster reduction in makespan value –which is certainly desirable for Grid schedulers running in short periods of time.

In order to better understand the behaviour of the three hash-based SGA implementations, we monitored the similarity value computed by each of them (see Table 2 below). It can be seen that, except the a key case, the rest used the same similarity value. The best performance of c key implementation could be explained on the one hand due to its time efficiency (especially when compared with SGA struggle quadratic) and due to the fact that it achieves to introduce an new individual of better genetic information in the population.

Table 2. Similarity value of the studied Struggle GAs.

SGA version	*Similarity value (averaged)*
SGA struggle quadratic	*0,975*
a key SGA	*0,736*
b key SGA	*0,958*
c key SGA	*0,965*

Another benefit of using the hash-based struggle implementation is the scalability. Indeed, since the computational time now is scaled down to a linear order, we could increase the size of both population and that of intermediate population without affecting the search time of the GA. We observed this experimentally (hereafter, the c key SGA is used); we show this effect by considering a population of 30 individuals and intermediate population of 50% of population size and then by increasing the population size to 80 individuals. The graphical representation is shown in Figures 3 and 4.

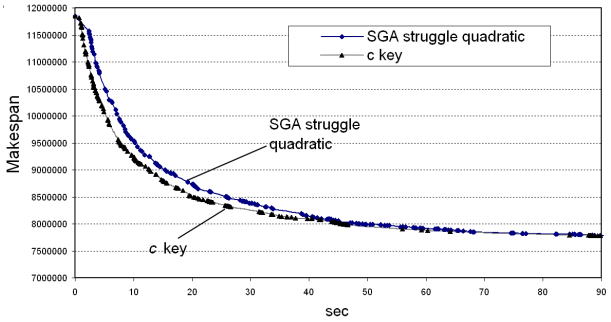

Figure 3. Makespan reduction of c key SGA vs. SGA struggle quadratic using a population of 30 individuals and intermediate population size of 50%.

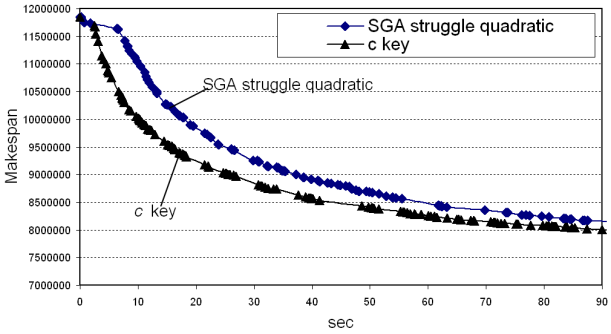

Figure 4. Makespan reduction of c key SGA vs. SGA struggle quadratic using a population of 80 individuals and intermediate population size of 50%.

Effectively, we can see from Figures 3 and 4 that, while for a population of size 30 the two SGA perform almost equally, by enlarging the population size to 80 individuals, the makespan reduction is much faster for the c key SGA than SGA struggle quadratic.

4.2. Computational results for a static benchmark

We present here some computational results for the c key SGA implementation for 12 instances from the static benchmark from Braun et al. [8]. The parameter configuration used is that of Table 1. The results given next are averaged over 10 independents runs, each of them running for at most 90 sec.; the same machine of standard configuration was used.

The computational results are shown in Table 3. The first column indicates the instance name. Here, the notation u_x_yyzz.0 means: u—uniform distribution, x—inconsistency (c—consistent, i—inconsistent and s—semi-consistent), yy—job heterogeneity (hi—high, lo—low), zz—machine heterogeneity (hi—high, lo—low); each instance consists of 512 jobs and 16 machines. Note that considered instances are representing different heterogeneous computing environments (four instances for consistent, semi-consistent and inconsistent, respectively). The second column indicates the makespan value obtained with the SGA (struggle quadratic) and the third one the makespan value obtained with SGA (c key hash implementation). Values in bold face show the best makespan of the two implementations.

Table 3: Makespan values for Braun et al. instances obtained with SGA (struggle quadratic) and SGA with c key hash implementation

Instance	SGA (struggle quadratic)	SGA (c key Hash)
u_c_hihi.0	**7722057,537**	7752689,084
u_c_hilo.0	157009,284	**156680,578**
u_c_lohi.0	**253843,0**	253926,055
u_c_lolo.0	5271,765	**5251,1462**
u_i_hihi.0	3269481,011	**3161104,915**
u_i_hilo.0	76413,844	**75598,481**
u_i_lohi.0	116981,385	**111792,174**
u_i_lolo.0	2634,997	**2620,7218**
u_s_hihi.0	4473513,333	**4433792,275**
u_s_hilo.0	98782,703	**98560,043**
u_s_lohi.0	132971,473	**130425,852**
u_s_lolo.0	3565,905	**3534,306**

As evident from Table 3, SGA with c key hash implementation outperforms SGA struggle quadratic

implementation. Moreover, the SGA with c key hash implementation shows to be robust, it performs well for three types of instances (consistent, inconsistent and semi-consistent).

5. Conclusions and future work

In this paper we have presented some results on tuning Struggle GAs the problem of scheduling of independent jobs in computational grids. This version of the scheduling requires fast reduction of makespan due to the changeability of the computational environment. In this work we have proponed hash-based implementations of the struggle genetic operator for the Gas for the problem. The resulting struggle operator showed several good properties:

- It is efficient: the quadratic computational time (in terms of the population time) is reduced to a linear time and hence the overall proper search time of the GA is increased.
- It allows to efficiently tackling large scale scheduling problems by considering larger size populations, due to the linear factor computational time.
- It achieves to introduce new individuals of better genetic information into new generation.
- It is robust: the resulting Struggle GA performed very well on almost all considered instances representing different heterogeneous computing environments.

In our future work, we plan to consider other similarity measures, now based on binary representations of task-resource allocation. Also, we would like to consider clustering-like techniques, for instance, *K-clustering,* to increase the performance of the struggle operator.

Acknowledgment

This research is partially supported by Projects ASCE TIN2005- 09198-C02-02, FP6-2004-ISO-FETPI (AEOLUS) and MEC TIN2005-25859-E and MEFOALDISI (Métodos formales y algoritmos para el diseño de sistemas) TIN2007-66523.

References

[1] Casanova, H. and Dongarra, J.: Netsolve: Network enabled solvers. IEEE Computational Science and Engineering, 5(3):57-67, 1998.

[2] Goux, J.P., Kulkarni, S., Linderoth, J. and Yoder, M.: An enabling framework for master-worker applications on the computational grid. In 9th IEEE International Sympo-sium on High Performance Distributed Computing (HPDC'00). IEEE Computer Society, 2000.

[3] Wright, S.J.: Solving optimization problems on computational grids. Optima, 65, 2001.

[4] Newman, H.B., Ellisman, M.H. and Orcutt, J.A.: Data-intensive e-science frontier research, Communications of ACM, 46(11):68-77, 2003.

[5] Paniagua, C., Xhafa, F., Caballé, S. and Daradoumis, Th.: A parallel grid-based implementation for real time processing of event log data in collaborative applications. In Parallel and Distributed Processing Techniques (PDPT2005), Las Vegas, USA, 2005.

[6] Beynon, M.D ., Sussman, A., Çatalyürek, Ü, Kure, T. and Saltz, J.: Optimization for data intensive grid applications. In Third Annual International Workshop on Active Middleware Services, pp. 97-106, 2001.

[7] Abraham, A., Buyya, R. and Nath, B. Nature's heuristics for scheduling jobs on computational grids, The 8th IEEE International Conference on Advanced Computing and Communications (ADCOM 2000) India, 45--52, 2000.

[8] Braun, H.J., Siegel, T.D., Beck, N., Bölöni, L.L., Maheswaran, M., Reuther, A.I., Robertson, J.P., Theys, M.D. and Yao, B.: A comparison of eleven static heuristics for mapping a class of independent jobs onto heterogeneous distributed computing systems. Journal of Parallel and Distributed Computing, 61(6):810--837, 2001.

[9] Xhafa, F. A Hyper-heuristic for Adaptive Scheduling in Computational Grids, International Journal on Neural and Mass-Parallel Computing and Information Systems, 17(6), 639-656, 2007.

[10] Xhafa, F., Alba, E., Dorronsoro, B. and Duran, B. Efficient Batch Job Scheduling in Grids using Cellular Memetic Algorithms, Journal of Mathematical Modelling and Algorithms, Published Online DOI: http://dx.doi.org/10.1007/s10852-008-9076-y

[11] Xhafa, F., Barolli, L. and Durresi, A. An Experimental Study On Genetic Algorithms for Resource Allocation On Grid Systems, Journal of Interconnection Networks, Volume: 8, Issue: 4 (December 2007), 427 - 443, World Sci. Pub.

[12] Zomaya, A.Y. and Teh, Y.H. Observations on using genetic algorithms for dynamic load-balancing, IEEE Transactions On Parallel and Distributed Systems, 12(9):899--911, 2001.

[13] J.H.~Holland. Adaptation in Natural and Artificial Systems, University of Michigan Press, Ann Arbor, MI, 1975.

[14] Grueninger, T.: Multimodal optimization using genetic algorithms. Technical report. Department of Mechanical Engineering, MIT, Cambridge, MA, 1997.

[15] Fogel, D.B.: Evolutionary computation: toward a new philosophy of machine intelligence. IEEE Press Piscataway, NJ, USA, 1995.

[16] Ashlock, D.: Evolutionary Computation for Modeling and Optimization. Springer Verlag, 2005.