Dependable and Coordinated Resources Allocation Algorithms for Distributed Computing

Victor Toporkov (✉) and Dmitry Yemelyanov

National Research University "MPEI", Moscow, Russia
{ToporkovVV, YemelyanovDM}@mpei.ru

Abstract. In this work, we introduce slot selection and co-allocation algorithms for parallel jobs in distributed computing with non-dedicated and heterogeneous resources. A single slot is a time span that can be assigned to a task, which is a part of a parallel job. The job launch requires a co-allocation of a specified number of slots starting and finishing synchronously. Some existing resource co-allocation algorithms assign a job to the first set of slots matching the resource request without any optimization (the first fit type), while other algorithms are based on an exhaustive search. In this paper, algorithms for efficient, dependable and coordinated slot selection are studied and compared with known approaches. The novelty of the proposed approach is in a general algorithm efficiently selecting a set of slots according to the specified criterion.

Keywords: Distributed Computing · Grid · Dependability · Coordinated Scheduling · Resource Management · Slot · Job · Allocation · Optimization

1 Introduction

Modern high-performance distributed computing systems (HPCS), including Grid, cloud and hybrid infrastructures provide access to large amounts of resources [1, 2]. These resources are typically required to execute parallel jobs submitted by HPCS users and include computing nodes, data storages, network channels, software, etc. These resources are usually partly utilized or reserved by high-priority jobs and jobs coming from the resource owners. Thus, the available resources are represented with a set of time intervals (slots) during which the individual computational nodes are capable to execute parts of independent users' parallel jobs. These slots generally have different start and finish times and vary in performance level. The presence of a set of heterogeneous slots impedes the problem of resources allocation necessary to execute the job flow from HPCS users. Resource fragmentation also results in a decrease of the total computing environment utilization level [1, 2].

There are different approaches for a job-flow scheduling problem in distributed computing environments. Multi-agent application level scheduling [3] actually performs individual jobs execution optimization and, as a rule, does not imply any global resource sharing or allocation policy. Such approach with an unrestricted competition

for the available computing resources may result in an inefficient and unbalanced resources usage and hence poor overall job-flow execution efficiency.

Job flow scheduling in virtual organizations (VO) [4, 5] suggests uniform rules of resource sharing and consumption, in particular based on economic models. This approach allows improving the job-flow level scheduling and resource distribution efficiency. VO formation and performance largely depends on mutually beneficial collaboration between all the related stakeholders. However, users' preferences and owners' and administrators' preferences may conflict with each other. Users are likely to be interested in the fastest possible running time for their jobs with least possible costs whereas VO preferences are usually tuned for available resources load balancing or node owners' profit boosting. Thus, VO policies in general should respect all members and the most important aspect of rules suggested by VO is their fairness. At the same time VO scheduling policies usually limit individual jobs optimization opportunities, may violate queue order and possess disadvantages common for centralized scheduling structures [6]. In order to implement any of the described job-flow scheduling schemes and policies, first, one needs an algorithm for selecting sets of simultaneously available slots required for each job execution. Further, we shall call such set of simultaneously available slots with the same start and finish times as execution *window*.

In this paper, we study algorithms for optimal or near-optimal heterogeneous resources selection by a given criterion with the restriction to a total cost. Additionally we consider practical implementations for a dependable resources allocation problem.

The rest of the paper is organized as follows. Section 2 presents related works. Section 3 introduces a general scheme for searching slot sets efficient by the specified criterion. Then several implementations are proposed and considered. Section 4 contains simulation results for comparison of proposed and known algorithms. Section 5 summarizes the paper and describes further research topics.

2 Related Works

The scheduling problem in Grid is *NP*-hard due to its combinatorial nature and many heuristic-based solutions have been proposed. In [7] heuristic algorithms for slot selection, based on user-defined utility functions, are introduced. NWIRE system [7] performs a slot window allocation based on the user defined efficiency criterion under the maximum total execution cost constraint. However, the optimization occurs only on the stage of the best found offer selection. First fit slot selection algorithms (backtrack [8] and NorduGrid [9] approaches) assign any job to the first set of slots matching the resource request conditions, while other algorithms use an exhaustive search [10-12] and some of them are based on a linear integer programming (IP) [10] or mixed-integer programming (MIP) model [11]. Moab scheduler [13] implements the backfilling algorithm and during a slot window search does not take into account any additive constraints such as the minimum required storage volume or the maximum allowed total allocation cost.

Modern distributed and cloud computing simulators GridSim and CloudSim [14, 15] provide tools for jobs execution and co-allocation of simultaneously available computing resources. Base simulator distributions perform First Fit allocation algorithms without any specific optimization. CloudAuction extension [15] of CloudSim implements a double auction to distribute datacenters' resources between a job flow with a fair allocation policy. All these algorithms consider price constraints on individual nodes and not on a total window allocation cost. However, as we showed in [16], algorithms with a total cost constraint are able to perform the search among a wider set of resources and increase the overall scheduling efficiency.

GrAS [17] is a Grid job-flow management system built over Maui scheduler [13]. The resources co-allocation algorithm retrieves a set of simultaneously available slots with the same start and finish times even in heterogeneous environments. However, the algorithm stops after finding the first suitable window and, thus, doesn't perform any optimization except for window start time minimization.

Algorithm [18] performs job's response and finish time minimization and doesn't take into account constraint on a total allocation budget. [19] performs window search on a list of slots sorted by their start time, implements algorithms for window shifting and finish time minimization, doesn't support other optimization criteria and the overall job execution cost constraint.

AEP algorithm [20] performs window search with constraint on a total resources allocation cost, implements optimization according to a number of criteria, but doesn't support a general case optimization. Besides AEP doesn't guarantee same finish time for the window slots in heterogeneous environments and, thus, has limited practical applicability.

Main contribution of this paper is a window co-allocation algorithm performing resources selection according to the user requirements and restrictions. The novelty of the proposed approach consists in implementing a dynamic programming scheme in order to optimize heterogeneous resources selection according to the scheduling policy.

3 Resource Selection Algorithm

3.1 General Problem Statement

We consider a set R of heterogeneous computing nodes with different performance p_i and price c_i characteristics. Each node has a local utilization schedule known in advance for a considered scheduling horizon time L. A node may be turned off or on by the provider, transferred to a maintenance state, reserved to perform computational jobs. Thus, it's convenient to represent all available resources as a set of slots. Each slot corresponds to one computing node on which it's allocated and may be characterized by its performance and price.

In order to execute a parallel job one needs to allocate the specified number of simultaneously idle nodes ensuring user requirements from the resource request. The resource request specifies number n of nodes required simultaneously, their minimum applicable performance p, job's computational volume V and a maximum available

resources allocation budget C. The required window length is defined based on a slot with the minimum performance. For example, if a window consists of slots with performances $p \in \{p_i, p_j\}$ and $p_i < p_j$, then we need to allocate all the slots for a time $T = \frac{V}{p_i}$. In this way V really defines a computational volume for each single job subtask. Common start and finish times ensure the possibility of inter-node communications during the whole job execution. The total cost of a window allocation is then calculated as $C_W = \sum_{i=1}^{n} T * c_i$.

These parameters constitute a formal generalization for resource requests common among distributed computing systems and simulators. The overall problem statement lacks specific features of internal nodes processing as well as network communication aspects that are important for a job execution performance in data-intensive HPC systems. However, the problem remains independent from specific HPCS configurations and, thus, the problem solutions may be evaluated against general case criteria. For this purpose we introduce criterion f representing a user preference for the particular job execution during the scheduling horizon L. f can take a form of any additive function and as an example, one may want to allocate suitable resources with the maximum possible total data storage available before the specified deadline.

3.2 General Window Search Procedure

For a general window search procedure for the problem statement presented in Section 3.1, we combined core ideas and solutions from algorithm AEP [20] and systems [17, 19]. Both related algorithms perform window search procedure based on a list of slots retrieved from a heterogeneous computing environment.

Following is the general square window search algorithm. It allocates a set of n simultaneously available slots with performance $p_i > p$, for a time, required to compute V instructions on each node, with a restriction C on a total allocation cost and performs optimization according to criterion f. It takes a list of available slots ordered by their non-decreasing start time as input.

1. Initializing variables for the best criterion value and corresponding best window: $f_{max} = 0, w_{max} = \{\}$.
2. From the slots available we select different groups by node performance p_i. For example, group P_k contains resources allocated on nodes with performance $p_i \geq P_k$. Thus, one slot may be included in several groups.
3. Next is a cycle for all retrieved groups P_i starting from the max performance P_{max}. All the sub-items represent a cycle body.
   a. The resources reservation time required to compute V instructions on a node with performance P_i is $T_i = \frac{V}{p_i}$.
   b. Initializing variable for a window candidates list $S_W = \{\}$.
   c. Next is a cycle for all slots s_i in group P_i starting from the slot with the minimum start time. The slots of group P_i should be ordered by their non-decreasing start time. All the sub-items represent a cycle body.

(1) If slot s_i doesn't satisfy user requirements (hardware, software, etc.) then continue to the next slot (3c).
(2) If slot length $l(s_i) < T_i$ then continue to the next slot (3c).
(3) Set the new window start time $W_i.start = s_i.start$.
(4) Add slot s_i to the current window slot list S_W
(5) Next a cycle to check all slots s_j inside S_W
    i. If there are no slots in S_W with performance $P(s_j) = P_i$ then continue to the next slot (3c), as current slots combination in S_W was already considered for previous group P_{i-1}.
    ii. If $W_i.start + T_i > s_j.end$ then remove slot s_j from S_W as it can't consist in a window with the new start time $W_i.start$.
(6) If S_W size is greater or equal to n, then allocate from S_W a window W_i (a subset of n slots with start time $W_i.start$ and length T_i) with a maximum criterion value f_i and a total cost $C_i < C$. If $f_i > f_{max}$ then reassign $f_{max} = f_i$ and $W_{max} = W_i$.

4. End of algorithm. At the output variable W_{max} contains the resulting window with the maximum criterion value f_{max}.

3.3 Optimal Slot Subset Allocation

Let us discuss in more details the procedure which allocates an optimal (according to a criterion f) subset of n slots out of S_W list (algorithm step 3c(6)). For some particular criterion function f a straightforward subset allocation solution may be offered. For example for a window finish time minimization it is reasonable to return at step 3c(6) the first n cheapest slots of S_W provided that they satisfy the restriction on the total cost. These n slots will provide $W_i.finish = W_i.start + T_i$, so we need to set $f_i = -(W_i.start + T_i)$ to *minimize* the finish time at the end of the algorithm.

However in a general case we should consider a subset allocation problem with some additive criterion: $Z = \sum_{i=1}^{n} c_z(s_i)$, where $c_z(s_i) = z_i$ is a target optimization characteristic value provided by a single slot s_i of W_i. In this way we can state the following problem of an optimal n - size window subset allocation out of m slots stored in S_W:

$$Z = x_1 z_1 + x_2 z_2 + \cdots + x_m z_m, \quad (1)$$

with the following restrictions:

$$x_1 c_1 + x_2 c_2 + \cdots + x_m c_m \le C,$$
$$x_1 + x_2 + \cdots + x_m = n,$$
$$x_i \in \{0,1\}, i = 1..m,$$

where z_i is a target characteristic value provided by slot s_i, c_i is total cost required to allocate slot s_i for a time T_i, x_i - is a decision variable determining whether to allocate slot s_i ($x_i = 1$) or not ($x_i = 0$) for the current window.

This problem relates to the class of integer linear programming problems and we used 0-1 knapsack problem as a base for our implementation. The classical 0-1 knapsack problem with a total weight C and items-slots with weights c_i and values z_i have

the same formal model (1) except for extra restriction on the number of items required: $x_1 + x_2 + \cdots + x_m = n$. To take this into account we implemented the following dynamic programming recurrent scheme:

$$f_i(C_j, n_k) = \max\{f_{i-1}(C_j, n_k), f_{i-1}(C_j - c_i, n_k - 1) + z_i\}, \quad (2)$$

$$i = 1,..,m,\ C_j = 1,..,C, n_k = 1,..,n,$$

where $f_i(C_j, n_k)$ defines the maximum Z criterion value for n_k-size window allocated out of first i slots from S_W for a budget C_j. After the forward induction procedure (2) is finished the maximum value $Z_{max} = f_m(C, n)$. x_i values are then obtained by a backward induction procedure.

For the actual implementation we initialized $f_i(C_j, 0) = 0$, meaning $Z = 0$ when we have no items in the knapsack. Then we perform forward propagation and calculate $f_1(C_j, n_k)$ values for all C_j and n_k based on the first item and the initialized values. Then $f_2(C_j, n_k)$ is calculated taking into account second item and $f_1(C_j, n_k)$ and so on. So after the forward propagation procedure (2) is finished the maximum value $Z_{max} = f_m(C, n)$. Corresponding values for variables x_i are then obtained by a backward propagation procedure.

An estimated computational complexity of the presented recurrent scheme is $O(m * n * C)$, which is n times harder compared to the original knapsack problem ($O(m * C)$). On the one hand, in practical job resources allocation cases this overhead doesn't look very large as we may assume that $n << m$ and $n << C$. On the other hand, this subset allocation procedure (2) may be called multiple times during the general square window search algorithm (step 3c(6)).

3.4 Dependable and Coordinated Resources Allocation

As a practical implementation for a general optimization scheme we propose to study a resources allocation placement problem. Fig. 1 shows Gantt chart of 4 slots co-allocation (hollow rectangles) in a computing environment with resources pre-utilized with local and high-priority tasks (filled rectangles).

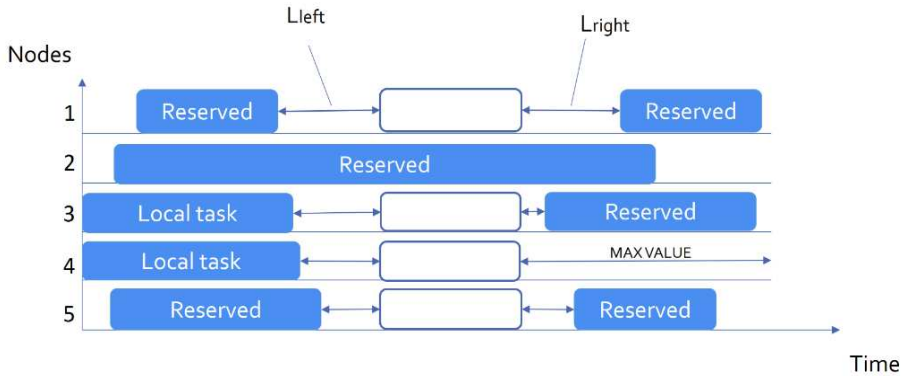

Fig. 1. Dependable window co-allocation metrics.

As can be seen from Fig. 1, even using the same computing nodes (1, 3, 4, 5 on Fig. 1) there are usually multiple window placement options with respect to the slots start time. The window placement generally may affect such job execution properties as cost, finish time, computing energy efficiency, etc. Besides, slots *proximity* to neighboring tasks reserved on the same computing nodes may affect a probability of the job execution delay or failure. For example, a slot reserved too close to the previous task on the same node may be delayed or cancelled by an unexpected delay of the latter. Thus, dependable resources allocation may require reserving resources with some reasonable distance to the neighboring tasks.

As presented in Fig. 1, for each window slot we can estimate times to the previous task finish time: L_{left} and to the next task start time: L_{right}. Using these values the following criterion for the window allocation represents average time distance to the nearest neighboring tasks: $L_{\min\Sigma} = \frac{1}{n}\sum_{i=1}^{n} min(L_{left\,i}, L_{right\,i})$, where n is a total number of slots in the window. So when implementing a dependable job scheduling policy we are interested in maximizing $L_{\min\Sigma}$ value.

On the other hand such *selfish* and individual job-centric resources allocation policy may result in an additional resources fragmentation and, hence, inefficient resources usage. Indeed, when $L_{\min\Sigma}$ is maximized the jobs will try to start at the maximum distance from each other, eventually leaving truncated slots between them. Thus, the subsequent jobs may be delayed in the queue due to insufficient remaining resources.

For a coordinated job-flow scheduling and resources load balancing we propose the following window allocation criterion representing average time distance to the farthest neighboring tasks: $L_{\max\Sigma} = \frac{1}{n}\sum_{i=1}^{n} max(L_{left\,i}, L_{right\,i})$, where n is a total number of slots in the window. By minimizing $L_{\max\Sigma}$ our motivation is to find a set of available resources best suited for the particular job configuration and duration. This *coordinated* approach opposes selfish resources allocation and is more relevant for a virtual organization job-flow scheduling procedure.

4 Simulation Study

4.1 Simulation Environment Setup

An experiment was prepared as follows using a custom distributed environment simulator [2, 16, 20]. For our purpose, it implements a heterogeneous resource domain model: nodes have different usage costs and performance levels. A space-shared resources allocation policy simulates a local queuing system (like in GridSim or CloudSim [14]) and, thus, each node can process only one task at any given simulation time. The execution cost of each task depends on its execution time, which is proportional to the dedicated node's performance level. The execution of a single job requires parallel execution of all its tasks.

During the experiment series we performed a window search operation for a job requesting $n = 7$ nodes with performance level $p_i \geq 1$, computational volume $V = 800$ and a maximum budget allowed is $C = 644$. During each experiment a

new instance for the computing environment was automatically generated with the following properties. The resource pool includes 100 heterogeneous computational nodes. Each node performance level is given as a uniformly distributed random value in the interval [2, 10]. So the required window length may vary from 400 to 80 time units. The scheduling interval length is 1200 time quanta which is enough to run the job on nodes with the minimum performance. However, we introduce the initial resource load with advanced reservations and local jobs to complicate conditions for the search operation. This additional load is distributed hyper-geometrically and results in up to 30% utilization for each node (Fig. 2).

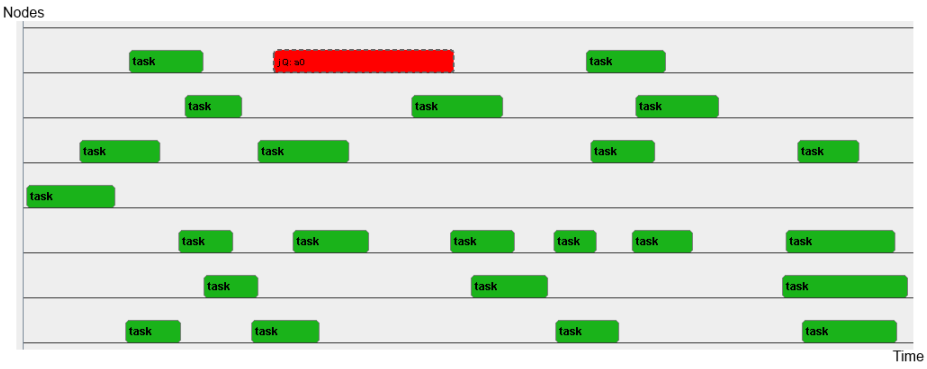

Fig. 2. Initial resources utilization example.

Additionally an independent value $q_i \in [0; 10]$ is randomly generated for each computing node i to compare algorithms against $Q = \sum_{i=1}^{n} q_i$ window allocation criterion.

4.2 General Algorithms Comparison

Firstly we intend to study the proposed resources allocation algorithm against an abstract general-case criterion Q. For this purpose we implemented the following window search algorithms based on the general window search procedure introduced in Section 3.2.

- *FirstFit* performs a square window allocation in accordance with a general scheme described in Section 3.2. Returns first suitable and affordable window found. In fact, performs window start time minimization and represents algorithm from [17, 19].
- *MultipleBest* algorithm searches for multiple non-intersecting alternative windows using *FirstFit* algorithm. When all possible window allocations are retrieved the algorithm searches among them for alternatives with the maximum Q value. In this way *MultipleBest* is similar to [7] approach.
- *MaxQ* implements a general square window search procedure with an optimal slots subset allocation (2) to return a window with maximum total Q value.

- *MaxQ Lite* follows the general square window search procedure but doesn't implement slots subset allocation (2) procedure. Instead at step 3c(6) it returns the first n cheapest slots of S_W. The total Q value of these n slots is returned as a target criterion, which is then maximized during the search procedure. Thus, *MaxQ Lite* has much less computational complexity compared to *MaxQ* but doesn't guarantee an accurate solution [20].

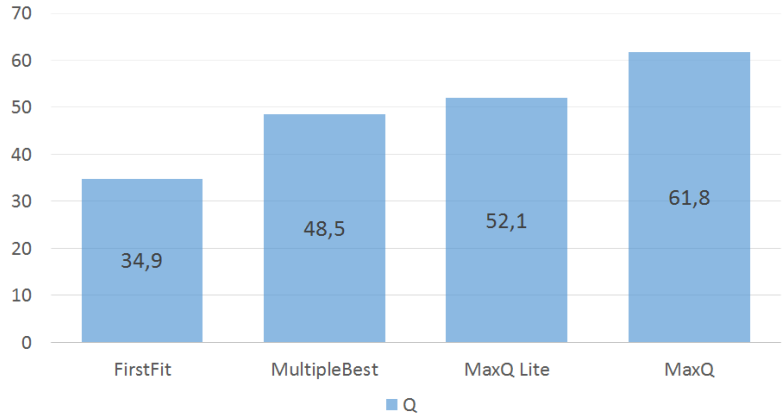

Fig. 3. Simulation results: average window Q value.

Fig. 3 shows average $Q = \sum_{i=1}^{n} q_i$ value obtained during the simulation. Parameter q_i was generated randomly on a [0; 10] interval and is independent from other node's characteristics. Thus, for a single window of 7 slots we have the following practical limits specific for our experiment: $Q \in [0; 70]$.

As can be seen from Fig. 3, *MaxQ* is indeed provided the maximum average criterion value $Q = 61.8$, which is quite close to the practical maximum, especially compared to other algorithms. The advantage over *MultipleBest* and *MaxQ Lite* is almost 20%. *MaxQ Lite* implements a simple heuristic but still is able to provide a better solution compared to the best of 50 different alternative executions retrieved by *MultipleBest*. *First Fit* provided average Q value exactly in the middle of [0; 70] which is 44% less compared to *MaxQ*.

4.3 Dependable Resources Allocation

For the window placement problem along with *FirstFit* and *MultipleBest* we introduce two pairs of algorithms based on *MaxQ* and *MaxQ Lite* approaches.

- *Dependable (DEP)* and *DEP Lite* perform $L_{\min \Sigma}$ maximization, i.e. maximize the distance to the nearest running or reserved tasks.
- *Coordinated (COORD)* and *COORD Lite* minimize $L_{\max \Sigma}$: average distance to the farthest neighboring tasks.

So, by setting $L_{\min \Sigma}$ and $L_{\max \Sigma}$ as target optimization criteria we performed scheduling simulation with the same settings described in Section 4.1. The results of 2000 independent scheduling cycles are compiled in Table 1.

As expected *DEP* provided maximum average distances to the adjacent tasks: 369 and 480 time units, which is comparable to the job's execution duration. An example of such allocation from a single simulation experiment is presented on Fig. 4 (a). The resulting *DEP* $L_{\min \Sigma}$ distance value is 4.3 times longer compared to *FirstFit* and almost 1.5 longer compared to *MultipleBest*.

Table 1. Window placement simulation results

Algorithm	Distance to the nearest task $L_{\min \Sigma}$	Distance to the farthest task $L_{\max \Sigma}$	Average Operational Time, ms
Multiple Best	253	*159*	*103*
First Fit	85	*342*	***4.2***
DEP	**369**	***480***	*1695*
DEP Lite	275	*440*	*4.5*
COORD	**9**	***52***	*1694*
COORD Lite	31	*148*	*4.5*

Similarly, *COORD* provided minimum values for the considered criteria: 9 and 52 time units. Example allocation is presented on Fig. 4 (b) where left edge represents the scheduling interval start time. As can be seen from the figure the allocated slots are highly coincident with the job's configuration and duration. Here the resulting average distance to the farthest task is three times smaller compared to *MultipleBest* and 9 times smaller when compared with *DEP* solution.

(a) (b)

Fig. 4. Simulation examples for dependable (a) and coordinated (b) resources allocation for the same job.

However due to a higher computational complexity it took *DEP* and *COORD* almost 1.7 seconds to find the 7-slots allocation over 100 available computing nodes,

which is 17 times longer compared to *Multiple Best*. At the same time simplified *Lite* implementations provided better scheduling results compared to *Multiple Best* for even less operational time: 4.5ms. *FirstFit* doesn't perform any target criteria optimization and, thus, provides average $L_{\min \Sigma}$ and $L_{\max \Sigma}$ distances with the same operational time as *Lite* algorithms.

MultipleBest in Table 1 has average distance to the farthest task smaller than to the nearest task because different alternatives were selected to match the criteria: $L_{\min \Sigma}$ maximization and $L_{\max \Sigma}$ minimization. Totally almost 50 different resource allocation alternatives were retrieved and considered by *MultipleBest* during each experiment.

5 Conclusion and Future Work

In this work, we address the problems of dependable and coordinated slot selection and co-allocation for parallel jobs in distributed computing with non-dedicated resources. For this purpose a general window allocation algorithm was proposed along with two practical implementations: for dependable and coordinated resources allocation policies.

A simulation study was carried out to prove the algorithm's optimization efficiency according to the target criteria. As a result, the advantage of the proposed general scheme over traditional scheduling algorithms reaches 20% against an abstract general case criterion and more than 50% when we consider window placement problem.

As a drawback, the general case algorithm has a relatively high computational complexity, especially compared to First Fit approach. In our further work, we will refine a general resource co-allocation scheme in order to decrease its computational complexity.

Acknowledgments. This work was partially supported by the Council on Grants of the President of the Russian Federation for State Support of Young Scientists (YPhD-2297.2017.9), RFBR (grants 18-07-00456 and 18-07-00534) and by the Ministry on Education and Science of the Russian Federation (project no. 2.9606.2017/8.9).

References

1. Dimitriadou, S.K., Karatza, H.D.: Job Scheduling in a Distributed System Using Backfilling with Inaccurate Runtime Computations. In: Proc. 2010 International Conference on Complex, Intelligent and Software Intensive Systems, pp. 329-336 (2010)
2. Toporkov, V., Toporkova, A., Tselishchev, A., Yemelyanov, D., Potekhin P.: Heuristic Strategies for Preference-based Scheduling in Virtual Organizations of Utility Grids. J. Ambient Intelligence and Humanized Computing 6 (6), 733–740 (2015)
3. Buyya, R., Abramson, D., Giddy, J.: Economic Models for Resource Management and Scheduling in Grid Computing. J. of Concurrency and Computation: Practice and Experience 5 (14), 1507-1542 (2002)

4. Foster, I., Kesselman C., Tuecke S.: The Anatomy of the Grid: Enabling Scalable Virtual Organizations. Int. J. of High Performance Computing Applications 15 (3), pp. 200-222 (2001)
5. Carroll, T., Grosu, D.: Formation of Virtual Organizations in Grids: A Game-Theoretic Approach. Economic Models and Algorithms for Distributed Systems 22 (14), 63-81 (2009)
6. Yang, R., Xu, J.: Computing at massive scale: Scalability and dependability challenges. Service-Oriented System Engineering (SOSE), 2016 IEEE Symposium on, pp. 386-397 (2016)
7. Ernemann, C., Hamscher, V., Yahyapour, R.: Economic Scheduling in Grid Computing. In: Feitelson, D.G., Rudolph, L., Schwiegelshohn, U. (eds.) JSSPP 2002. LNCS, vol. 2537, pp. 128-152. Springer, Heidelberg (2002)
8. Aida, K., Casanova, H.: Scheduling Mixed-parallel Applications with Advance Reservations. In: 17th IEEE Int. Symposium on HPDC, pp. 65-74. IEEE CS Press, New York (2008)
9. Elmroth, E., Tordsson J.: A Standards-based Grid Resource Brokering ServiceSupporting Advance Reservations, Co-allocation and Cross-Grid Interoperability. J. of Concurrency and Computation: Practice and Experience 25 (18), 2298-2335 (2009)
10. Takefusa, A., Nakada, H., Kudoh, T., Tanaka, Y.: An Advance Reservation-basedCo-allocation Algorithm for Distributed Computers and Network Bandwidth on QoS-guaranteed Grids. In: Frachtenberg E., Schwiegelshohn U. (eds.) JSSPP 2010. LNCS, vol. 6253, pp. 16-34. Springer, Heidelberg (2010)
11. Blanco, H., Guirado, F., Lrida, J.L., Albornoz, V.M.: MIP Model Scheduling for Multi-clusters. In: Euro-Par 2012. LNCS, vol. 7640, pp. 196-206. Springer, Heidelberg (2013)
12. Garg, S.K., Konugurthi, P., Buyya, R.: A Linear Programming-driven Genetic Algorithm for Meta-scheduling on Utility Grids. Int. J. of Parallel, Emergent and Distributed Systems 26, 493-517 (2011)
13. Moab Adaptive Computing, http://www.adaptivecomputing.com, last accessed 2018/04/12
14. Calheiros, R. N., Ranjan, R., Beloglazov, A., De Rose, C. A. F., and Buyya,R.: CloudSim: A Toolkit for Modeling and Simulation of Cloud Computing Environments and Evaluation of Resource Provisioning Algorithms. J. Software: Practice and Experience 41 (1), 23-50 (2011)
15. Samimi, P., Teimouri, Y., Mukhtar M.: A combinatorial double auction resource allocation model in cloud computing. J. Information Sciences 357 (C), 201-216 (2016)
16. Toporkov, V., Toporkova, A., Bobchenkov, A., Yemelyanov, D.: Resource Selection Algorithms for Economic Scheduling in Distributed Systems. In: Proc. International Conference on Computational Science, ICCS 2011, June 1-3, 2011, Singapore, Procedia Computer Science. Elsevier, vol. 4. pp. 2267-2276 (2011)
17. Kovalenko, V.N., Koryagin, D.A.: The grid: Analysis of basic principles and ways of application. J. Programming and Computer Software 35(1), 18-34 (2009)
18. Makhlouf, S., Yagoubi, B.: Resources Co-allocation Strategies in Grid Computing. In: CIIA, vol. 825, CEUR Workshop Proceedings, 2011.
19. Netto, M. A. S., Buyya, R.: A Flexible Resource Co-Allocation Model based on Advance Reservations with Rescheduling Support. In: Technical Report, GRIDSTR-2007-17, Grid Computing and Distributed Systems Laboratory, The University of Melbourne, Australia, October 9, 2007.
20. Toporkov, V., Toporkova, A., Tselishchev, A., Yemelyanov, D.: Slot Selection Algorithms in Distributed Computing. Journal of Supercomputing 69 (1), 53-60 (2014)