Heuristic Rules for Coordinated Resources Allocation and Optimization in Distributed Computing

Victor Toporkov[0000−0002−1484−2255]
and Dmitry Yemelyanov[0000−0002−9359−8245]

National Research University "Moscow Power Engineering Institute", Moscow, Russia
ToporkovVV@mpei.ru, YemelyanovDM@mpei.ru

Abstract. In this paper, we consider heuristic rules for resources utilization optimization in distributed computing environments. Existing modern job-flow execution mechanics impose many restrictions for the resources allocation procedures. Grid, cloud and hybrid computing services operate in heterogeneous and usually geographically distributed computing environments. Emerging virtual organizations and incorporated economic models allow users and resource owners to compete for suitable allocations based on market principles and fair scheduling policies. Subject to these features a set of heuristic rules for coordinated compact scheduling are proposed to select resources depending on how they fit a particular job execution and requirements. Dedicated simulation experiment studies integral job flow characteristics optimization when these rules are applied to conservative backfilling scheduling procedure.

Keywords: Distributed computing · Resource allocation · Scheduling · Slot · Backfilling · Economic model · Optimization.

1 Introduction and Related Works

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.

There are two important classes of users' parallel jobs. Bags of tasks (BoT) represent parallel applications incorporating a large number of independent or weakly connected tasks. Typical examples of BoT are parameter sweeps, Monte Carlo simulations or exhaustive search. Workflows consist of multiple tasks with control or data dependencies. Such applications may be presented as directed graphs and represent complex computational or data processing problems in many domains of science [2–4].

Most BoT and workflow applications require some assurances of quality of services (QoS) from the computing system. In order to ensure QoS requirements and constraints, a coordinated allocation of suitable resources should be performed [5–7]. Most QoS requirements are based on either time or cost constraints such as total job execution cost, deadline, response time, etc. [8–11].

Some of the most important efficiency indicators of a distributed computational environment include both system resources utilization level and users' jobs time and cost execution criteria [4, 8, 9, 12]. In distributed environments with non-dedicated resources, such as utility Grids, the computational nodes are usually partly utilized and reserved in advance by jobs of higher prority [10]. Thus, the resources available for use are represented with a set of slots - time intervals 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 a performance difference. The presence of a set of slots impedes the problem of coordinated selection of the resources necessary to execute the job-flow from computational environment users. Resource fragmentation also results in a decrease of the total computing environment utilization level [12, 13].

High-performance distributed computing systems organization and support bring certain economical expenses: purchase and installation of machinery equipment, power supplies, user support, etc. As a rule, HPCS users and service providers interact in economic terms and the resources are provided for a certain payment. In such conditions, resource management and job scheduling based on the economic models is considered as an efficient way to take into account contradictory preferences of computing participants [3, 14–16].

A metascheduler or a metabroker is considered as an intermediate link between the users, local resource management and job batch processing systems [8, 17]. It defines uniform rules of a resource sharing and consumption to improve the overall scheduling efficiency [12, 13, 16].

The main contribution of this paper is a set of heuristic rules for a coordinated resources allocation for parallel jobs execution. The algorithm takes into account the system slots configuration as well as individual jobs features: size, runtime, cost, etc. When used in HPCS metaschedulers during the resources allocation step, it may improve overall system utilization level by matching jobs with resources and providing better jobs placement.

The rest of the paper is organized as follows. Section 2 presents resources allocation problem in relation to job-flow scheduling algorithms and backfilling (Subsection 2.2). Different approaches for a coordinated resources allocation are described and proposed in Section 3. Section 4 contains algorithms implementation details along with simulation results and analysis. Finally, Section 5 summarizes the paper and describes further research topics.

2 Resources Allocation for Job-flow Scheduling

2.1 Computing Model

In order to cover a wide range of computing systems we consider the following model for a heterogeneous resource domain.

Constituent computing nodes of a domain have different usage costs and performance levels. A space-shared resources allocation policy simulates a local queuing system (like in CloudSim [14, 15] or SimGrid [4, 18]) and, thus, each node can process only one task at any given time. Economic scheduling model [14, 15] assumes that users and resource owners operate with some currency to coordinate resources allocation transactions. This model allows to regulate interaction between different organizations and to settle on fair equilibrium prices for resources usage.

Thus we consider a set R of heterogeneous computing nodes with different performance p_i and price c_i characteristics.

A node may be turned off or on by the provider, transferred to a maintenance state, reserved to perform computational jobs. Thus each node has a local utilization schedule known in advance for a considered scheduling horizon time L.

The execution cost of a single task depends on the allocated node's price and execution time, which is proportional to the node's performance level. 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. These parameters constitute a formal generalization for resource requests common among distributed computing systems and simulators [12, 14, 18].

In heterogeneous environment 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 internode communications during the whole job execution. The total cost of a window allocation is then calculated as $C = \sum_{i=1}^{n} T * c_i$.

2.2 Job-flow Scheduling and Backfilling

The simplest way to schedule a job-flow execution is to use the First-Come-First-Served (FCFS) policy. However this approach is inefficient in terms of resources utilization and Backfilling [19] was proposed to improve system utilization.

Backfilling procedure makes use of advanced resources reservations which is an important mechanism preventing starvation of jobs requiring large number of computing nodes. Resources reservations in FCFS may create idle slots in the nodes' local schedules thus decreasing system performance. So the main idea behind backfilling is to backfill jobs into those idle slots to improve the overall system utilization. And the backfilling procedure implements this by placing smaller jobs from the back of the queue to these idle slots ahead of the priority order.

There are two common variations to backfilling - conservative and aggressive (EASY). Conservative Backfilling enforces jobs' priority fairness by making sure that jobs submitted later can't delay the start of jobs arrived earlier. EASY Backfilling aggressively backfills jobs as long as they do not delay the start of the single currently reserved jobs. Conservative Backfilling considers jobs in the order of their arrival and either immediately starts a job or makes an appropriate reservation upon the arrival. The jobs priority in the queue may be additionally modified in order to improve system-wide job-flow execution efficiency metrics. Under default FCFS policy the jobs are arranged by their arrival time. Other priority reordering-based policies like Shortest job First or eXpansion Factor may be used to improve overall resources utilization level [9, 10, 13].

Multiple Queues backfilling separates jobs into different queues based on metadata, such as jobs resource requirements: small, medium, large, etc. The idea behind this metaheuristic is that earlier arriving jobs and smaller-sized jobs should have higher execution priority. The number of queues and the strategy for dividing tasks among them can be set by the system administrators. Sometimes different queues may be assigned to a dedicated resource domain segments and function independently. In a single domain the metaheuristic cycles through the different queues in a round-robin fashion and may consider more jobs from the queues with smaller-sized tasks [13].

The look-ahead optimizing scheduler [10] implements dynamic programming scheme to examine all the jobs in the queue in order to maximize the current system utilization. So, instead of scanning queue for single jobs suitable for the backfilling, look-ahead scheduler attempts to find a combination of jobs that together will maximize the resources utilization.

2.3 Resources Selection Algorithms

Backfilling as well as many other job-flow scheduling algorithms in fact describe a general procedure determining high level policies for jobs prioritization and advanced resources reservations. However, the resources selection and allocation step remains sidelined since its more system specific nature. Consequently resource selection algorithms specifications usually either too hardware specific or lack certain restrictions or model features in order to cover a broader class of computing systems.

On the other hand, applying different resources allocation policies based on system or user preferences may affect scheduling results not only for individual jobs but for a whole job-flow.

In [6, 7] we presented a Slot Subset Allocation (SSA) dynamic programming scheme for resources selection in heterogeneous computing environments based on economic principles. In a general case system nodes may be shared and reserved in advance by different users and organizations (including resource owners). So it's convenient to represent all available resources as a set of time-slots (see Section 2.1). Each slot corresponds to one computing node on which it is allocated. SSA algorithm takes these time slots as input and performs resources selection for a specified job in accordance with the computing model and constraints described in Section 2.1. The resulting window satisfies user

QoS requirements from the resource request and may be reserved for the job execution.

Additionally SSA may perform window search optimization by a general additive criterion $Z = \sum_{i=1}^{n} z(s_i)$, where $z(s_i) = z_i$ is a target optimization characteristic value provided by a single slot s_i of window W. For this purpose SSA implements the following dynamic programming recurrent scheme to allocate n–size window with a maximum total cost C from m simultaneously available slots:

$$f_i(C_j, n_k) = \max\{f_{i-1}(C_j, n_k), f_{i-1}(C_j - c_i, n_k - 1) + z_i\}, \quad (1)$$
$$k = 1, \ldots, n, i = 1, \ldots, m, j = 1, \ldots, C,$$

where $f_i(C_j, n_k)$ defines the maximum Z criterion value for n_k - size window allocated out of first i available slots for a budget C_j. After the forward induction procedure (1) is finished, the maximum criterion value can be found as $Z_{\max} = f_m(C, n)$. Corresponding resources are then obtained by a backward induction procedure.

These criterion values z_i may represent different slot characteristics: time, cost, power, hardware and software features, etc. Thus SSA-based resources allocation is proved to be a flexible tool for a preference-based job-slow execution [6].

3 Coordinated Resources Allocation Heuristics

3.1 Dependable Job Placement Problem

One important aspect for a resources allocation efficiency is the resources placement in regard to an actual slots configuration. So as a practical implementation for a general z_i parameter maximization 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 jobs (filled rectangles).

As can be seen from Fig. 1, even using the same computing nodes (1,3,4,5) there are usually multiple window placement options with respect to the slots start time. The slots' actual placement generally may affect such job execution properties as cost, finish time, computing energy efficiency, etc. Besides that, slots proximity to neighboring tasks reserved on the same computing nodes may affect the efficiency of the resources utilization. For example, reserving a slot close to a neighboring task may increase resources load by minimizing the corresponding node's idle time. On the other hand, leaving larger idle distances to the occupied or reserved slots sometimes may prove practical for the subsequent queue jobs scheduling.

For a quantitative placement criterion 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} (Fig. 1). Using these values we consider the following criteria for the whole window allocation optimization:

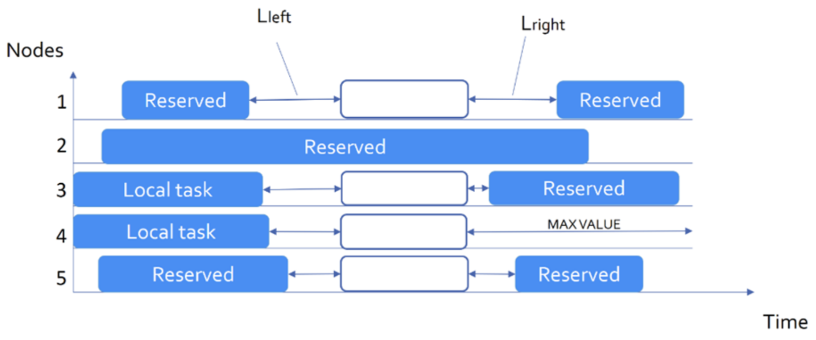

Fig. 1: Dependable window co-allocation metrics

- $L_{\Sigma} = \frac{1}{n}\sum_{i=1}^{n}(L_{lefti} + L_{righti})$ represents average time distance between window and the neighboring tasks reserved on the same nodes;
- $L_{\min\Sigma} = \frac{1}{n}\sum_{i=1}^{n}\min(L_{lefti}, L_{righti})$ displays average time distance to the nearest neighboring tasks.

Based on L_{Σ} or $L_{\min\Sigma}$ criteria idle space minimization or maximization strategies can be implemented using SSA algorithm.

3.2 Job Placement Heuristics

However these L_{Σ} or $L_{\min\Sigma}$ criteria alone can't improve the whole job-flow scheduling solution according to the conventional makespan or average finish time criteria. Preliminary experiments showed 30%-100% longer makespan for $L_{\min\Sigma} \rightarrow \min$ resources allocation strategy compared to a traditional backfilling procedure [9, 13] with jobs finish time minimization.

The reason for this result is that finish time minimization criterion in some degree incorporates and combines L_{left} minimization for early start time with a suitable (by performance) resources types selection. Consequently a greedy application of a finish time criterion in backfilling procedure provides efficient overall job-flow scheduling solution. But still there are some additional more complex heuristics which may improve scheduling results when combined with a finish time criterion.

For example in [13] a special set of *breaking a tie* rules is proposed to choose between slots providing the same earliest job start time. These rules for Picking Earliest Slot for a Task (PAST) procedure may be summarized as following.

1. Minimize number of idle slots left after the window allocation; i.e. slots adjacent (succeeding or preceding) to already reserved slots have higher priority.
2. Maximize length of idle slots left after the window allocation; so the algorithm tends to left longer slots for the subsequent jobs in the queue.

With similar intentions we propose the following Coordinated Placement (CoP) heuristic rules slightly different from PAST [13].

0. Prioritize slots allocated on nodes with lower performance. The main idea is that when deciding between two slots providing the same window finish time it makes sense to leave higher performance slot vacant for the subsequent jobs. This breaking a tie principle is applicable for heterogeneous resources environments and do not consider slots placement configuration. However, during the preliminary simulations this heuristic alone was able to noticeably improve scheduling results so we will use it as an addition to the placement rules.
1. Prioritize slots with relatively small distances to the neighbor tasks: $L_{lefti} << T$ or $L_{righti} << T$. The general idea is similar to the first rule in PAST, but CoP don't expect perfect match and defines threshold values for a satisfactory window fit.
2. Penalize slots leaving significant, but insufficient to execute a full job distances L_{lefti} or L_{righti}. For example when $\frac{T}{3} > L_{righti} > \frac{T}{5}$, the resulting slot may be to short to execute any of the subsequent jobs and will remain idle, thus, reducing the resources overall utilization.
3. On the other hand equally prioritize slots leaving sufficient compared to the job's runtime distances L_{lefti} or L_{righti}. For example with $L_{lefti} > T$.

So the main idea behind CoP is to fill in the gaps in the resources reservation schedule by providing quite an accurate resources allocation matching jobs runtime. Unlike PAST, CoP do not expect perfect matches but makes realistic heuristic decisions to minimize general resources fragmentation.

A simple resources allocation example on Fig. 2 demonstrates differences between these approaches. Fig. 2 represents a computing environment segment consisting of eight nodes with some resources already allocated or reserved for six jobs A - F. Each job is represented as a filled rectangle (or a set of rectangles) spanning in both resource and time axes.

Consider a scenario when a next job requires three simultaneously available slots and the earliest finish time is achievable by using slots 1,2,3 or 4 from Fig. 2. In this case backfilling without any heuristic *breaking a tie* rules will choose slots 1,2 and 3 just according to a simple slots order. PAST would choose slots 1,3 and 4, minimizing resources fragmentation and leaving longer slot for the subsequent jobs. CoP would allocate slots 2,3 and 4 as they provide better fit for the job, while slot 1 if allocated will leave two short slots likely unprofitable for future use.

4 Simulation Study

4.1 Implementation and Simulation Details

Based on heuristic rules described in Section 3.2 we implemented the following scheduling algorithms and criteria for SSA-based resources allocation.

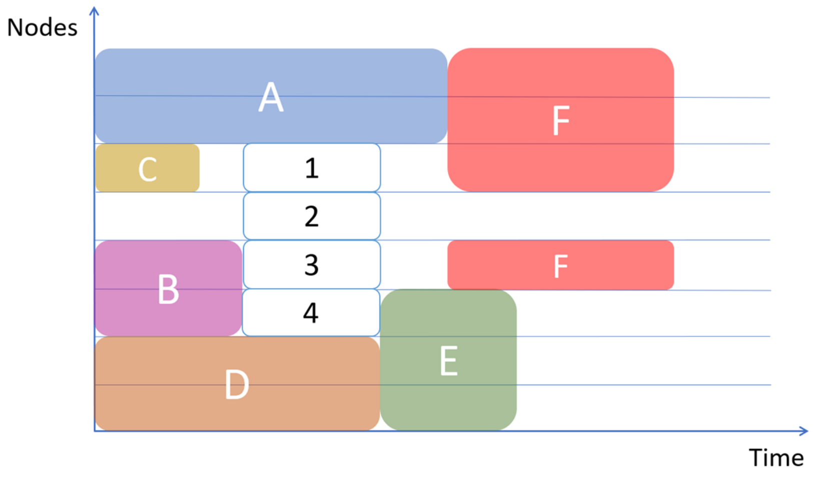

Fig. 2: Job placement heuristic rules example

1. Firstly we consider two conservative backfilling variations. *BFstart* successively implements start time minimization for each job during the resources selection step. As SSA performs criterion maximization, *BFstart* criterion for i-th slot has the following form: $z_i = -s_i.startTime$
   By analogy *BFfinish* implements a more solid strategy of a finish time minimization which is different from *BFstart* in computing environments with heterogeneous resources. *BFfinish* criterion for SSA algorithm is the following: $z_i = -s_i.finishTime$
2. PAST-like backfilling approach has a more complex criterion function which may be described with the following set of rules:
   (a) $z_i = -s_i.finishTime$; finish time is the main criterion value
   (b) $z_i = z_i - \alpha_1 * s_i.nodePerformance$; node performance amendment
   (c) $if(L_{righti} == 0) : z_i = z_i + \delta_1$; PAST rule 1
   (d) $if(L_{lefti} == 0) : z_i = z_i + \delta_1$; PAST rule 1
   (e) $z_i = z_i - \alpha_2 * L_{righti}$; PAST rule 2
3. CoP resources allocation algorithm for backfilling may be represented with the following criterion calculation:
   (a) $z_i = -s_i.finishTime$; finish time is the main criterion value
   (b) $z_i = z_i - \alpha_1 * s_i.nodePerformance$; node performance amendment
   (c) $if(L_{righti} < \epsilon_1 * T) : z_i = z_i + \delta_1$; CoP rule 1
   (d) $if(L_{lefti} < \epsilon_1 * T) : z_i = z_i + \delta_1$; CoP rule 1
   (e) $if(L_{righti} > \epsilon_2 * T \& L_{righti} < \epsilon_3 * T) : z_i = z_i - \delta_1$; CoP rule 2
   (f) $if(L_{lefti} > \epsilon_2 * T \& L_{lefti} < \epsilon_3 * T) : z_i = z_i - \delta_1$; CoP rule 2
   (g) $if(L_{righti} > T) : z_i = z_i + \delta_2$; CoP rule 3
   (h) $if(L_{lefti} > T) : z_i = z_i + \delta_2$; CoP rule 3

4. Finally as an additional reference solution we simulate another abstract backfilling variation *BFshort* which is able to reduce each job runtime for 1% during the resources allocation step. In this way each job will benefit not only from its own earlier completion time, but from earlier completion of all the preceding jobs.

The criteria for PAST and CoP contain multiple constant values defining rules behavior, namely $\alpha_1, \alpha_2, \delta_1, \delta_2, \epsilon_1, \epsilon_2, \epsilon_3$. ϵ_i coefficients define threshold values for a satisfactory job fit in CoP approach. α_i and δ_i define each rule's effect on the criteria and are supposed to be much less compared to z_i in order to break a tie between otherwise suitable slots. However their mutual relationship implicitly determine rules' priority which can greatly affect allocation results. Therefore there are a great number of possible α_i, δ_i and ϵ_i values combinations providing different PAST and CoP implementations. Based on heuristic considerations and some preliminary experiment results the values we used during the present experiment are presented in Table 1.

Table 1: PAST and CoP parameters values

Constant	α_1	α_2	δ_1	δ_2	ϵ_1	ϵ_2	ϵ_3
Value	0.1	0.0001	1	0.1	0.03	0.2	0.35

Because of heuristic nature of considered algorithms and their speculative parametrization (see Table 1) hereinafter by PAST [13] we will mean PAST-like approach customly implemented as an alternative to CoP.

4.2 Simulation Results

The experiment was prepared as follows using a custom distributed environment simulator [12, 16]. 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 CloudSim or SimGrid [14, 18]) 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. More details regarding the simulation computing model were provided in Section 2.1.

Besides that, the simulator implements a graphical interface representing Gantt diagram for the resulting job-flow scheduling outcome.

During each simulation experiment a new instance for the computing environment segment consisting of 32 heterogeneous nodes was automatically generated. Each node performance level is given as a uniformly distributed random value in the interval [2, 16]. This configuration provides a sufficient resources diversity level while the difference between the highest and the lowest resource performance levels will not exceed one order.

In this environment we considered job queue with 50, 100, 150 and 200 jobs accumulated at the start of the simulation. The jobs are arranged in queues by priority and no new jobs are submitted during the queue execution. Such scheduling problem statement allows to statically evaluate algorithms' efficiency in conditions with different resources utilization level. The jobs were generated with the following resources request requirements: number of simultaneously required nodes is uniformly distributed in interval $n \in [1; 8]$, computational volume $V \in [60; 1200]$ also contribute to a wide diversity in user jobs.

The results of 2000 independent simulation experiments are presented in Tables 2-3. Each simulation experiment includes computing environment and job queue generation, followed by a scheduling simulation independently performed using considered algorithms. The main scheduling results are then collected and contribute to the average values over all experiments.

Table 2 contain average finish time provided by algorithms *BFstart*, *BFfinish*, *BFshort*, PAST and CoP for different number of jobs pre-accumulated in the queue.

Table 2: Simulation results: average job finish time

Jobs N_Q	*BFstart*	*BFfinish*	*BFshort*	PAST	CoP
50	318,8	302,1	298,8	300,1	298
100	579,2	555	549,2	556,1	550,7
150	836,8	805,6	796,8	809	800,6
200	1112	1072,7	1060,3	1083,3	1072,2

As it can be seen, with a relatively small number N_Q of jobs in the queue, both CoP and PAST provide noticeable advantage by nearly 1% over a strong *BFfinish* variation and CoP even surpasses *BFshort* results. At the same time less successful *BFstart* approach provides almost 6% later average completion time highlighting difference between a good (*BFfinish*) and a regular (*BFstart*) possible scheduling solutions. So *BFshort*, CoP and PAST advantage should be evaluated against this 6% interval.

However with increasing the jobs number CoP advantage over *BFfinish* decreases and tends to zero when $N_Q = 200$. This trend may be observed on Fig. 3 presenting relative finish time advantage over *BFfinish* for all considered algorithms. *BFshort* graph is represented as an almost straight line 1% above reference *BFfinish* solution, which is expected by design.

CoP graph starts above *BFshort* and gradually decreases to the *BFfinish* 0% line. However as PAST average performance decreases contemporaneously with CoP, latter maintains 0.5%-1% advantage over PAST for all considered simulation experiments.

The performance decrease trend for PAST and CoP heuristics may be explained by increasing accuracy requirements for jobs placement caused with increasing N_Q number. Indeed, when considering for some intermediate job resource selection the more jobs are waiting in the queue the higher the probability that some future job will have a better fit for current resource during the backfilling procedure. In order to adapt to higher resources utilization levels, threshold parameters ϵ_i may be changed to encourage even better job placement fits during the resources allocation. In a general case all the algorithms' parameters $\alpha_i, \delta_i, \epsilon_i$ (more details we provided in Section 3.2) should be refined to correspond to the actual computing environment utilization level.

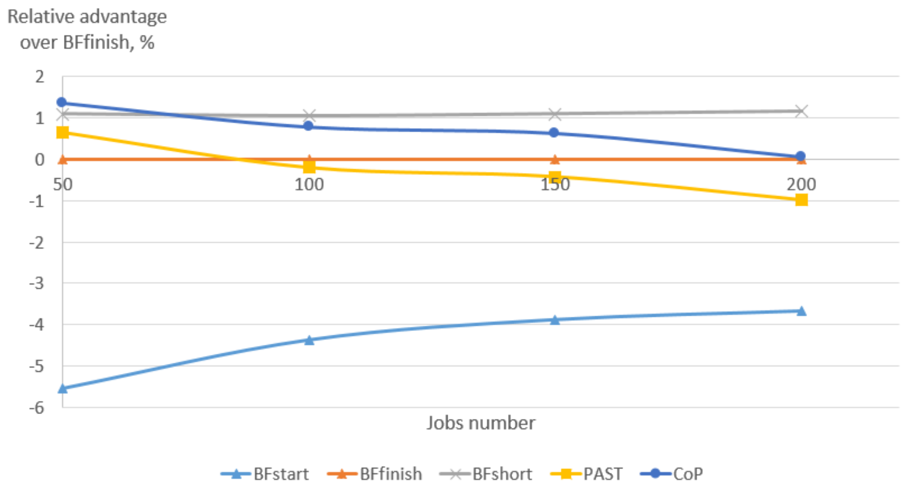

Fig. 3: Simulation results: relative advantage over BFfinish by jobs finish time criterion depending on the jobs queue size

Table 3: Simulation results: average job finish time for jobs distributed over half of a makespan interval

Jobs N_Q	*BFstart*	*BFfinish*	*BFshort*	PAST	CoP
50	381,7	375,4	371,8	371,8	369,5
100	672,5	662,6	656,9	657,9	653,4
150	942,4	922,6	915,8	921	914,9
200	1208,2	1184,2	1173,2	1184,1	1173,8

However if we distribute jobs arrival time over some interval we may derive similar results as average number of jobs waiting in the queue for backfilling will be less. In the following experiment we perfromed job queue scheduling with the same settings except that jobs had random arrival times in the range up to half of the makespan obtained during the first experiment. Corresponding average job finish times from another 2000 simulations is presented in Table 3.

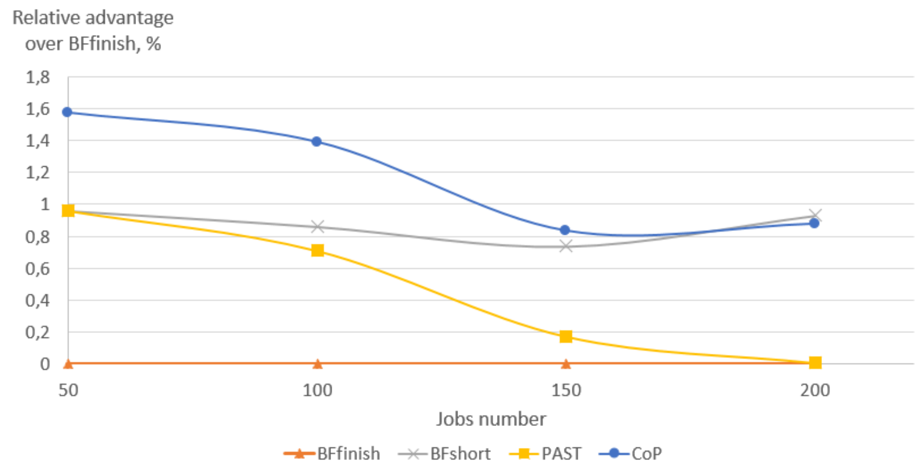

Fig. 4: Simulation results: relative advantage over *BFfinish* by jobs finish time criterion in scenario with jobs distributed over half of a makespan interval

Based on data from Table 3 Fig.4 shows relative advantage over *BFfinish* against finish time criterion for *BFshort*, PAST and CoP aprroaches in scenario with jobs' arrival times dynamically distributed over a period of time. The trend is different from the static scenario (Fig. 3) as CoP maintains 1% advantage even for 200 jobs in the queue.

One important property of the proposed heuristic approach is that it generally preserves integral job-flow execution parameters of the base scheduling algorithm: jobs' priority and processing order, average execution cost. For example, maximum average difference in jobs-flow execution cost between CoP and *BFfinish* over all 4000 simulations reaches 0.25%.

5 Conclusion

In this work, we address the problem of a coordinated resources allocation for parallel jobs scheduling optimization in heterogeneous computing environments. Modern job-flow scheduling algorithms optimize integral job-flow scheduling characteristics mainly by determining jobs prioritization and execution order, leaving resources selection step aside as too system specific. Based on a *Slots*

Subset Allocation resources selection algorithm, we propose and implement a set of heuristic job placement rules for jobs' Coordinated Placement (CoP). The main idea behind CoP approach is to fill in the gaps in the resources utilization schedule by allocating resources tailored to particular jobs runtimes.

Simulation study shows overall job-flow scheduling efficiency improvement just by using CoP rules during the resources allocation step in conservative backfilling. The advantage over a basic resources allocation strategy reaches 1.5% against average job-flow finish time criteria. At the same time CoP preserves job-flow execution parameters, jobs priorities and processing order.

In our further work, we will refine job placement heuristics to include and balance user preferences with global scheduling criteria during the resources allocation step.

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-2979.2019.9), 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. Lee, Y.C., Wang, C., Zomaya, A.Y., Zhou, B.B.:Profit-driven Scheduling for Cloud Services with Data Access Awareness. J. of Parallel and Distributed Computing, 72(4), 591–602 (2012)
2. Bharathi, S., Chervenak, A.L., Deelman, E., Mehta, G., Su, M., Vahi, K.: Characterization of scientific workflows. In: 2008 Third Workshop on Workflows in Support of Large-Scale Science, pp. 1–10 (2008)
3. Rodriguez, M.A., Buyya, R.: Scheduling dynamic workloads in multi-tenant scientific workflow as a service platforms. Future Generation Computer Systems, 79(P2), 739–750 (2018)
4. Nazarenko, A., Sukhoroslov, O.: An experimental study of workflow scheduling algorithms for heterogeneous systems. In: V. Malyshkin (ed.) Parallel Computing Technologies, pp. 327–341. Springer International Publishing (2017)
5. 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)
6. Toporkov, V., Yemelyanov, D.: Dependable slot selection algorithms for distributed computing. Advances in Intelligent Systems and Computing. Vol. 761, pp. 482-491. Springer Verlag (2019)
7. Toporkov, V., Yemelyanov, D.: Optimization of Resources Selection for Jobs Scheduling in Heterogeneous Distributed Computing Environments. Lecture Notes in Computer Science, 10861 LNCS, Springer Verlag, pp. 574–583 (2018)
8. Kurowski, K., Nabrzyski, J., Oleksiak, A., Weglarz, J.: Multicriteria Aspects of Grid Resource Management. In: Nabrzyski, J., Schopf, J.M., Weglarz J. (eds.) Grid resource management. State of the art and future trends, pp. 271-293. Kluwer Academic Publishers (2003)

9. Srinivasan, S., Kettimuthu, R., Subramani, V., Sadayappan, P.: Characterization of Backfilling Strategies for Parallel Job Scheduling. In: Proceedings of the International Conference on Parallel Processing, ICPP'02 Workshops, pp. 514–519 (2002)
10. Shmueli, E., Feitelson, D.G.: Backfilling with lookahead to optimize the packing of parallel jobs. Journal of Parallel and Distributed Computing, 65(9), 1090–1107 (2005)
11. Menasc'e, D.A., Casalicchio, E.: A Framework for Resource Allocation in Grid Computing. In: The 12th Annual International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems (MASCOTS 2004), pp. 259-267. Volendam, The Netherlands. (2004)
12. 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)
13. Khemka, B., Machovec, D., Blandin, C., Siegel, H.J., Hariri, S., Louri, A., Tunc, C., Fargo, F., Maciejewski, A.A.: Resource Management in Heterogeneous Parallel Computing Environments with Soft and Hard Deadlines. In: Proceedings of 11th Metaheuristics International Conference (MIC'15) (2015)
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., Yemelyanov, D., Toporkova, A.: Fair Scheduling in Grid VOs with Anticipation Heuristic. In R. Wyrzykowski et al. (Eds.): PPAM'17, pp. 145–155. LNCS 10778, Springer International Publishing AG (2018)
17. Rodero, I., Villegas, D., Bobroff, N., Liu, Y., Fong, L., Sadjadi, S.: Enabling interoperability among grid meta-schedulers. Journal of Grid Computing, 11(2), 311–336 (2013)
18. Casanova H., Giersch A., Legrand A., Quinson M., Suter F.: Versatile, scalable, and accurate simulation of distributed applications and platforms. Journal of Parallel and Distributed Computing, 74(10), 2899–2917 (2014)
19. Jackson, D., Snell, Q., Clement, M.. Core algorithms of the Maui scheduler. In: Revised Papers from the 7th International Workshop on Job Scheduling Strategies for Parallel Processing, JSSPP '01, pp. 87-102 (2001)