Micro-Scheduling for Dependable Resources Allocation

Victor Toporkov and Dmitry Yemelyanov

Abstract In this work, we introduce a general approach for slot selection and co-allocation algorithms for parallel jobs in distributed computing with non-dedicated and heterogeneous resources. Parallel job scheduling provides many opportunities for the resources allocation and usage efficiency optimization. Firstly, there are many options to select the appropriate set of resources based on primary target criteria in a knapsack-like problem. The secondary optimization, or , is possible when selecting over a variety of suitable resources providing the same primary target criteria values. Micro-scheduling step usually relies on the resources meta-features, secondary parameters and their actual utilization. Such two-level optimization may be used to obtain heuristic solutions for many scheduling problems. In this paper we present micro-scheduling applications for the dependable and coordinated resources co-allocation, resources usage efficiency optimization, preference-based and fair scheduling implementations.

Keywords Distributed computing · Grid · Dependability · Micro-scheduling · Coordinated scheduling · Resource management · Slot · Job · Allocation · Optimization · Preferences

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

V. Toporkov · D. Yemelyanov (✉)
National Research University "MPEI", Krasnokazarmennaya, 14, Moscow 111250, Russia
e-mail: YemelyanovDM@mpei.ru

V. Toporkov
e-mail: ToporkovVV@mpei.ru

G. Bocewicz et al. (eds.), *Performance Evaluation Models for Distributed Service Networks*, Studies in Systems, Decision and Control 343,
https://doi.org/10.1007/978-3-030-67063-4_5

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].

HPCS organization and support bring certain economical expenses: purchase and installation of machinery equipment, power supplies, user support, maintenance works, security, etc. Thus, HPCS users and service providers usually 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 coordinate contradictory preferences of computing system participants and stakeholders [2–5].

There are different approaches for a job-flow scheduling problem in distributed computing environments. Application level scheduling [3] is based on the available resources utilization and, as a rule, does not imply any global resource sharing or allocation policy. Job flow scheduling in VOs [6–9] suppose uniform rules of resource sharing and consumption, in particular based on economic models [2–5]. This approach allows improving the job-flow level scheduling and resource distribution efficiency. VO policy may offer optimized scheduling to satisfy both users' and VO global preferences. The VO scheduling problems may be formulated as follows: to optimize users' criteria or utility function for selected jobs [2, 10], to keep resource overall load balance [11, 12], to have job run in strict order or maintain job priorities [13, 14], to optimize overall scheduling performance by some custom criteria [15, 16], etc.

Computing system services support interfaces between users and providers of computing resources and data storages, for instance, in datacenters. Personal preferences of VO stakeholders are usually contradictive. Users are interested in total expenses minimization while obtaining the best service conditions: low response times, high hardware specifications, 24/7/365 service, etc. Service providers and administrators, on the contrary, are interested in profits maximization based on resources load efficiency, energy consumption, and system management costs. The challenges of system management can lead to inefficient resources usage in some commercial and corporate cloud systems.

Thus, VO policies in general should respect all members to function properly and the most important aspect of rules suggested by VO is their fairness. A number of works understand fairness as it is defined in the theory of cooperative games [10], such as fair job flow distribution [12], fair user jobs prioritization [14], fair prices mechanisms [5]. In many studies VO stakeholders' preferences are usually ensured only partially: either owners are competing for jobs optimizing users' criteria [3], or the main purpose is the efficient resources utilization not considering users' preferences [13]. Sometimes multi-agent economic models are established [3, 5]. Usually they do not allow optimizing the whole job flow processing.

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 present general algorithm for an optimal or near-optimal heterogeneous resources selection by a given criterion with the restriction to a total cost. Further this algorithm serves as a basis for the two-level optimization (or a micro-scheduling) approach and some practical implementations for a dependable resources allocation problem.

The rest of the paper is organized as follows. Section 2 presents related works for the resources usage optimization when scheduling single parallel jobs and whole job-flows. Section 3 introduces a general scheme for searching slot sets efficient by the specified criterion. Then several implementations are proposed and considered. Sections 4–7 present heuristic micro-scheduling algorithms and applications for different HPSC scheduling problems. Section 8 summarizes the paper and describes further research topics.

2 Related Works

2.1 Resources Selection Algorithms and Approaches

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 [17] and NorduGrid [18] approaches) assign any job to the first set of slots matching the resource request conditions, while other algorithms use an exhaustive search [15, 19, 20] and some of them are based on a linear integer programming (IP) [15] or mixed-integer programming (MIP) model [19]. Moab/maui scheduler [13] implements *backfilling* algorithm and during the 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 [4, 5] 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 [5] 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

[21], 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 [22] 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 [23] performs job's response and finish time minimization and doesn't take into account constraint on a total allocation budget. [24] 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 [16] 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.

2.2 Job-Flow Scheduling and Backfilling

Backfilling [25–27] is a widely used procedure for a job *queue* scheduling in high-performance computing. The base algorithm relies on jobs runtime estimates and makes use of advanced reservations tools. This mechanism prevents starvation of jobs requiring large number of computing nodes and reduces resources idle time. The main idea behind these improvements is implemented by placing smaller jobs from the back of the queue to the any currently idle slots even ahead of their 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 job. 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 [25–27]. The look-ahead optimizing scheduler [26] 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.

Thus, the jobs priorities represent an important factor for the integral job-flow scheduling efficiency. General priority compliance contributes to a fair scheduling model and may support even more complex and high-level priority functions. On the other hand, it is possible to adjust order of the jobs scheduling and execution to achieve more efficient resources usage scenarios. We consider the problem of the resources usage optimization without affecting the initial jobs scheduling order (or in some cases with a controlled amount of changes [28]).

3 General Resource Co-allocation Algorithm

3.1 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.

Additionally we introduce criterion f as 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 Window Search Procedure

For a general window search procedure for the problem statement presented in Sect. 3.1, we combined core ideas and solutions from algorithm AEP [16] and sys-

tems [23, 24]. 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 functions 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 (as any other n slots from S_W at the current step) 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.

The same logic applies for a number of other important criteria, including window start time, runtime and a total cost minimization.

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:

$$Z = x_1 z_1 + x_2 z_2 + \cdots + x_m z_m, \tag{1}$$

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, which imposes obvious limitations on the practical methods to solve it. However, we used 0-1 knapsack problem as a base for our implementation. Indeed, 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:

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

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

where $f_i\left(C_j, n_k\right)$ 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 $f_i\left(C_j, 0\right) = 0$, meaning $Z = 0$ when we have no items in the knapsack. Then we perform forward propagation and calculate $f_1\left(C_j, n_k\right)$ values for all C_j and n_k based on the first item and the initialized values. Then $f_2\left(C_j, n_k\right)$ is calculated taking into account second item and $f_1\left(C_j, n_k\right)$ and so on. So after the forward propagation procedure (2) is finished the maximum value $Z_{max} = f_m\left(C, n\right)$. 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)).

4 Dependable Job Placement

4.1 Job Placement Problem

As a first practical implementation for a general optimization scheme SSA (Slots Subset Allocation) we study a window *placement problem*. Figure 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).

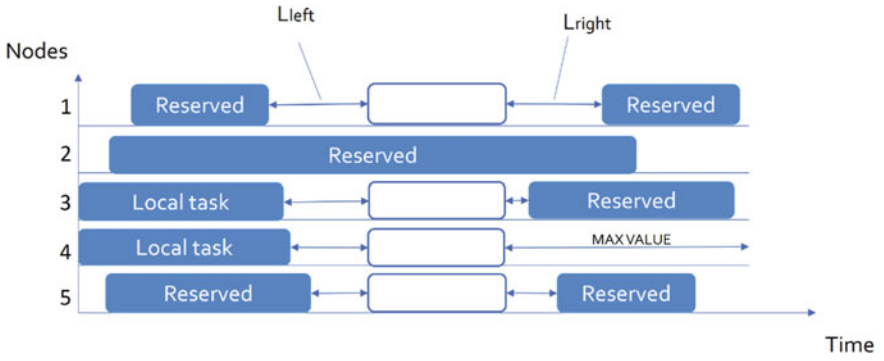

Fig. 1 Dependable window co-allocation metrics

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.

4.2 Simulation Environment Setup

An experiment was prepared as follows using a custom distributed environment simulator [2, 16, 28]. 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 [4]) 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

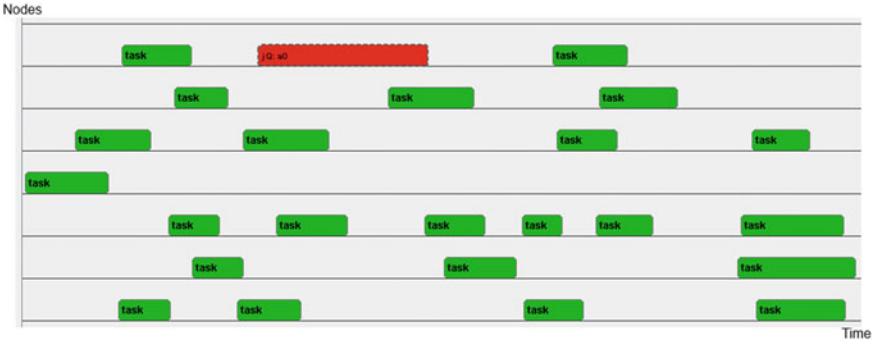

Fig. 2 Initial resources utilization example

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). The simulation has been performed in Java runtime environment on Intel Core i3 based workstation with 16Gb RAM and SSD storage.

4.3 Analysis of Dependable Resources Allocation

For the simulation study we introduce the following algorithms.

- *FirstFit* performs a square window allocation in accordance with a general scheme described in Sect. 3.2. Returns first suitable and affordable window found. In fact, performs window start time minimization and represents algorithm from [23, 24].
- *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 criteria value. In this way *MultipleBest* is similar to [7] approach.
- *Dependable (DEP)* and *DEP Lite* perform $L_{\min\ \Sigma}$ maximization, i.e. maximize the distance to the nearest running or reserved tasks. *DEP* implements a general square window search procedure with an optimal slots subset allocation (2). *DEP 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. Thus, *DEP Lite* has much less computational complexity compared to *DEP* but doesn't guarantee an accurate solution [16]

- *Coordinated (COORD)* and *COORD Lite* minimize $L_{\max\,\Sigma}$: average distance to the farthest neighboring tasks. *COORD* and *COORD Lite* represent *DEP* and *DEP Lite*, but with a different target criterion of $L_{\max\,\Sigma} \rightarrow min$.

So, by setting $L_{\min\,\Sigma}$ and $L_{\max\,\Sigma}$ as target optimization criteria we performed simulation of 2000 independent scheduling cycles. The results 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. 3a. The resulting *DEP* $L_{\min\,\Sigma}$ distance value is 4.3 times longer compared to *FirstFit* and almost 1.5 longer compared to *MultipleBest*.

Similarly, *COORD* provided minimum values for the considered criteria: 9 and 52 time units. Example allocation is presented on Fig. 3b 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.

However due to a higher computational complexity it took *DEP* and *COORD* almost 1.7 s 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.

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*

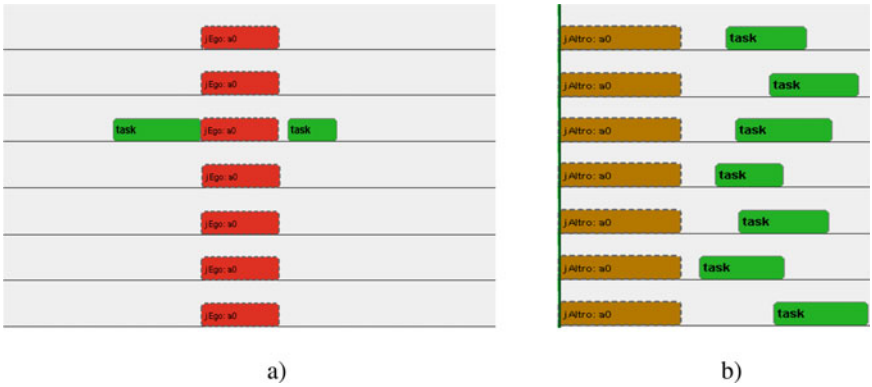

Fig. 3 Simulation examples for dependable (**a**) and coordinated (**b**) resources allocation for the same job

5 Micro-Scheduling and Coordinated Resources Allocation

5.1 Micro-Scheduling in Resources Allocation

Another important aspect for the overall resources allocation *efficiency* is the resources selection and *micro-scheduling* in regard to the anticipated resources utilization schedule.

Figure 4 shows the same Gantt chart from Fig. 1 of 4 slots co-allocation in a computing environment with resources pre-utilized with local and high-priority jobs. Slots 1–4 represent candidates for a job scheduling and execution. If the job requires only three nodes there are four different options to allocate the resources providing the same finish time. Job execution finish time corresponds to traditional queue scheduling criteria in backfilling-like algorithms. Thus, the process of a secondary optimization when selecting a particular subset of resources providing the same primary criteria value we call micro-scheduling.

L_{left} or L_{right} criteria alone can't improve the whole job-flow scheduling solution according to the conventional makespan or average finish time criteria. So, as an alternative a special set of breaking a tie rules is proposed in [27] to choose between slots providing the same earliest job start or finish time.

These rules for Picking Earliest Slot for a Task (PEST) procedure may be summarized as following.

1. Minimize number of idle slots left after the window allocation: 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.

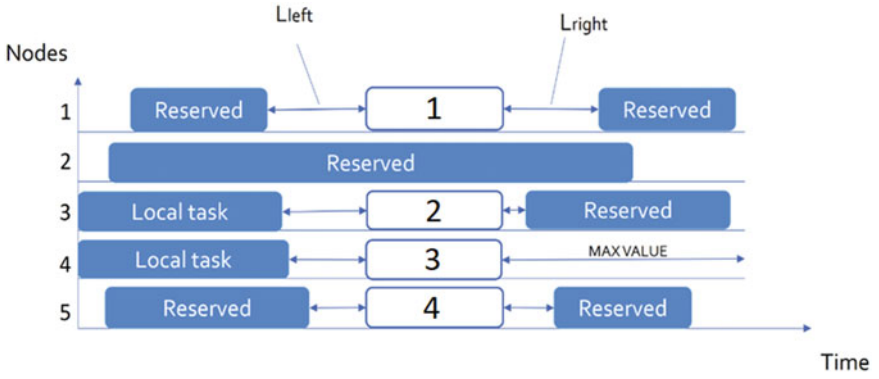

Fig. 4 Coordinated window co-allocation and placement metrics

With similar intentions we proposed the following Coordinated Placement (CoP) [29] heuristic rules.

1. Prioritize slots allocated on nodes with lower performance. The main idea is to leave higher performance slot vacant for the subsequent jobs.
2. Prioritize slots with relatively small distances to the neighbor tasks: $L_{left\ i} \ll T$ or $L_{right\ i} \gg T$.
3. Penalize slots leaving significant, but insufficient to execute a full job distances $L_{left\ i}$ or $L_{right\ i}$.
4. On the other hand, equally prioritize slots leaving sufficient compared to the job's runtime distances $L_{left\ i}$ or $L_{right\ i}$.

5.2 *Coordinated Placement Algorithms*

The main idea behind CoP is to minimize overall resources fragmentation by allocating slots to jobs with fairly matching runtime demands. Based on these heuristic rules we implemented the following scheduling algorithms and criteria for SSA-based resources allocation.

1. Firstly we consider two conservative backfilling variations. BFs successively implements *start time* minimization for each job during the resources selection step. As SSA performs criterion maximization, BFs criterion for i-th slot has the following form: $z_i = -s_i.startTime$. By analogy BFf implements a more solid strategy of a *finish time* minimization which is different from BFs in computing environments with heterogeneous resources. BFf criterion for SSA algorithm is the following: $z_i = -s_i.finishTime$.
2. PEST-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 $(L_{right\ i} == 0) : z_i = z_i + \delta_1$; PEST rule 1
(d) if $(L_{left\ i} == 0) : z_i = z_i + \delta_1$; PEST rule 1
(e) $z_i = z_i - \alpha_2 * L_{right\ i}$; PEST 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 $(L_{right\ i} < \varepsilon_1 * T) : z_i = z_i + \delta_1$; CoP rule 1
(d) if $(L_{left\ i} < \varepsilon_1 * T) : z_i = z_i + \delta_1$; CoP rule 1
(e) if $(L_{right\ i} > \varepsilon_2 * T$ AND $L_{right\ i} < \varepsilon_3 * T)$: $z_i = z_i - \delta_1$; CoP rule 2
(f) if $(L_{left\ i} > \varepsilon_2 * T$ AND $L_{left\ i} < \varepsilon_3 * T)$: $z_i = z_i - \delta_1$; CoP rule 2
(g) if $(L_{right\ i} > T)$: $z_i = z_i + \delta_3$; CoP rule 3
(h) if $(L_{left\ i} > T)$: $z_i = z_i + \delta_3$; 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 PEST and CoP contain multiple constant values defining rules behavior, namely $\alpha_1, \alpha_2, \delta_1, \delta_2, \varepsilon_1, \varepsilon_2, \varepsilon_3$. 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 criterion 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 determines 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 PEST and CoP implementations. Based on heuristic considerations and some preliminary experiment results the values we used during the present experiment are presented in Table 2.

Because of heuristic nature of considered algorithms and their speculative parametrization (see Table 2) hereinafter by PEST [27] we will mean PEST-like approach customly implemented as an alternative to CoP.

Table 2 PEST 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*

5.3 Simulation Results

The results of 2000 independent simulation experiments are presented in Tables 3 and 4. 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 3 contains average finish time in simulation time units (t.u.) provided by algorithms BFs, BFf, BFshort, PEST and CoP for different number of jobs pre-accumulated in the queue.

As it can be seen, with a relatively small number N_Q of jobs in the queue, both CoP and PEST provide noticeable advantage by nearly 1% over a strong BFf variation, while CoP even surpasses BFshort results. At the same time, the less successful

BFs approach provides almost 6% later average completion time, thus, highlighting difference between a good (BFf) and a regular (BFs) possible scheduling solutions. So BFshort, CoP and PEST advantage should be evaluated against this 6% interval. Similar conclusion follows from the average makespan (i.e. the latest finish time of the queue jobs) values presented in Table 4.

However with increasing the jobs number CoP advantage over BFf decreases and tends to zero when $N_Q = 200$. This trend for PEST 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 a general case all the algorithms' parameters α_i, δ_i and ε_i (more details we provided

Table 3 Simulation results: average job finish time, t.u

Jobs number N_Q	BFs	BFf	$BFshort$	$PAST$	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

Table 4 Simulation results: average job-flow makespan, t.u

Jobs number N_Q	BFs	BFf	$BFshort$	$PAST$	CoP
50	807.8	683	675	678	673.3
100	1407	1278	1264	1272	1264
150	2003	1863	1842	1857	1844
200	2622	2474	2449	2476	2455

in Table 2) should be refined to correspond to the actual computing environment utilization level.

6 Advanced Simulation and Hindsight Job-Flow Scheduling

6.1 Hindsight Scheduling

An important feature of CoP and similar breaking a tie approaches [27, 29] is that they do not affect the primary scheduling criterion and do not change a base scheduling procedure.

Depending on the computing environment setup and job queue size the advantage of CoP over the baseline backfilling procedure reaches 1–2% by average jobs finish time and makespan criteria. Although these relative advantage values do not look very impressive, an important result is that they were obtained almost for free: CoP represent the same backfilling procedure, but with a more efficient resources usage.

Figure 5 presents a distribution of a relative difference (%) between average jobs finish times provided by CoP and BF:

$$100\% * (BF.avFinishTime - CoP.avFinishTime)/CoP.avFinishTime. \tag{3}$$

The distribution was obtained from 250 simulations with $N = 50$ jobs in the queue. Positive values represent scenarios when earlier job-flow finish time was provided by CoP, while negative values – scenarios when a better solution is provided by *BF*.

As it can be observed, CoP generally provides better scheduling outcomes and resources usage compared to the baseline backfilling. In a number of experiments CoP advantage over BF reaches 10–15% with the maximum of 29% earlier finish time. On the other hand, sometimes CoP provide much later job-flow finish times: up to 12% behind BF (see Fig. 5). Thus, CoP average advantage of nearly 1% includes many outcomes with serious finish time delays.

Considering that CoP and BF represent the same scheduling procedure it is possible to implement a joint approach by choosing the best outcome precalculated by CoP, BF or a family of similar algorithms. In this case we will always use the most successful scenario in terms of the resources efficiency. Such joint (or a *hindsight*) approach may be used when it's possible to consider job queue execution on some scheduling interval or horizon.

For this purpose we consider *a family* of backfilling-based algorithms with different breaking a tie rules: *Random*, *Greedy* and *CoP*. After the scheduling step is over the best solution obtained by these algorithms is chosen as a Hindsight result. Except for CoP, this family is chosen almost arbitrary in order to evaluate how each rule will contribute to the final solution.

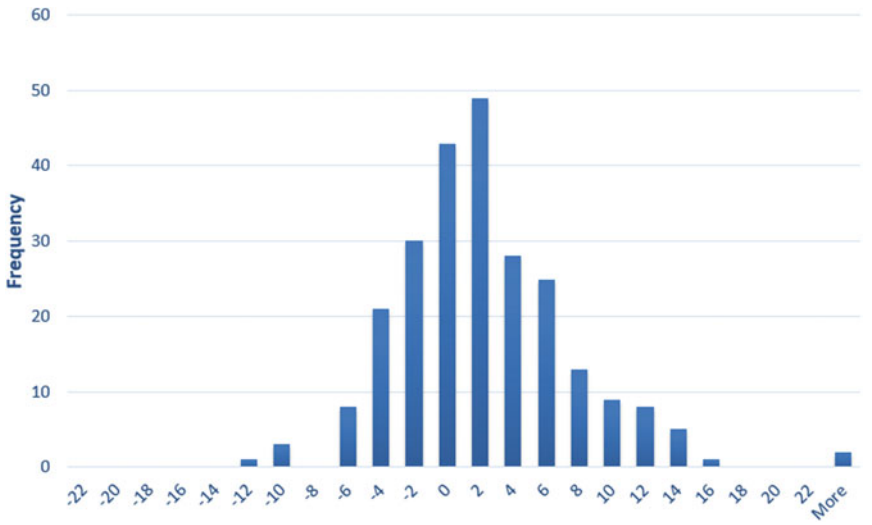

Fig. 5 Average CoP and BF job finish time difference distribution, $N = 50$

6.2 Simulation Setup and Analysis

Based on heuristic rules described in Sect. 6.1 we implemented the following scheduling algorithms and strategies for SSA-based resources allocation.

1. Firstly, we consider conservative backfilling *BFf* procedure. For a finish time *minimization*, BFf criterion for i-th considered slot has the following form: $z_i = -s_i.finishTime$. As there are no secondary criteria, BF generally selects a random subset of slots providing the earliest finish time for a job execution.
2. *Rand* algorithm uses SSA algorithm for a fully random resources selection over a conservative backfilling: $z_i = -s_i.finishTime + r_i$. Here r_i is a small random value uniformly distributed on interval [0; 01] representing the secondary criterion.
3. *Greedy* backfilling-based algorithm performs resources selection based on the following greedy metric of the resources *profitability*: $\frac{p_i}{c_i}$. Thus, the resulting SSA criterion function is: $z_i = -s_i.finishTime + \alpha\frac{p_i}{c_i}$. Here α defines the weight of secondary criteria and is supposed to be much less compared to a primary finish time criterion in order to break a tie between otherwise even slots. In the current study we used $\alpha = 0.1$ value.
4. *CoP* resources allocation algorithm for backfilling is implemented in accordance with rules and priorities described in Sect. 5. More details were provided in [29].

The experiment setup included a job-flow of $N = 50$ jobs in a domain consisting of 42 heterogeneous computing nodes. The jobs were accumulated at the start of the simulation and no new jobs were submitted during the queue execution. Such

Table 5 Breaking a tie heuristics scheduling results

Characteristic	BF	CoP	Rand	Greedy	Hindsight
Number of experiments	500	500	500	500	500
Average makespan	677	670	683	680	662
Average finish time	254	250	257	257	246
Earliest finish number	114	238	62	86	500
Earliest finish number, %	22.8	47.6	12.4	17.2	100%
Algorithm working time, s	0.01	52.6	54.4	53.2	160.2

problem statement allows us to statically evaluate algorithms' efficiency and simulate high resources load.

Table 5 contains simulation results obtained from 500 scheduling experiments for a family of breaking a tie heuristics contributing to the Hindsight solution.

We consider the following global job-flow execution criteria: a makespan (finish time of the latest job) and an average jobs' finish time.

Without taking into account the Hindsight solution, the best results were provided by CoP: nearly 1% advantage over BF and 2% over both Rand and Greedy algorithms.

Hindsight approach reduces makespan and average finish time even more: 1% advantage over CoP, 2% over BF and 3% over Rand and Greedy.

Although these relative advantage values do not look very impressive, an important result is that they were obtained almost for free: CoP or Hindsight represent *the same baseline backfilling procedure*, but with a more efficient resources usage.

CoP made the largest contribution to the Hindsight solution: in 238 experiments (47.6%) CoP provided the earliest job-flow finish time. Baseline BF contributed to Hindsight in 114 experiments (22.8%). Greedy provided the earliest finish time in 86 experiments (17.2%), Rand – in 62 experiments (12.4%).

In this way, CoP ruleset actually implements heuristics which allow better resources allocation for parallel jobs compared to other considered approaches. At the same time even algorithm with a random tie breaking procedure outperformed BF, Greedy and CoP in 17.2% of experiments. Thus, by combining a larger number of random algorithms in a single family may result in comparable or even better Hindsight solution.

However, the major limiting factor for the Hindsight approach is SSA's actual working time. Baseline BF with a single criterion implements a simple procedure with almost a linear computational complexity over a number of available resources $O(|R|)$. Consequently, its working time is relatively short: only 10 ms for a whole

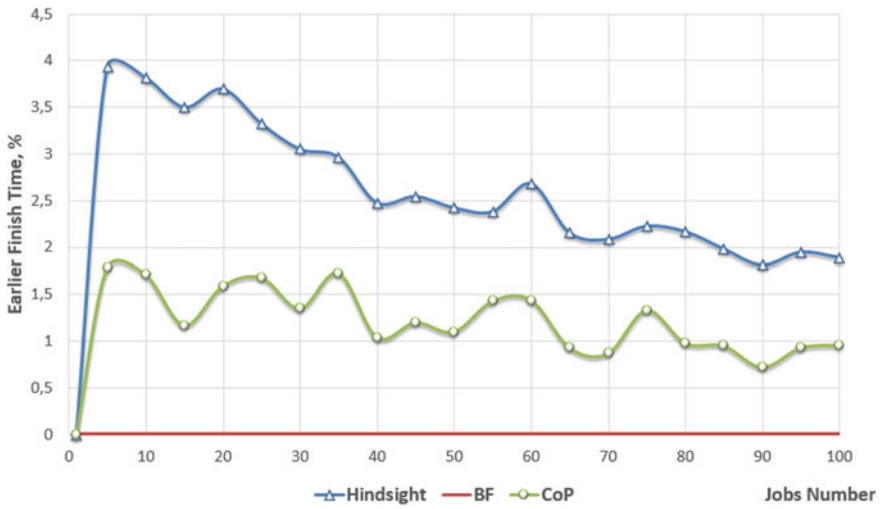

Fig. 6 Average job finish time difference between CoP, Hindsight and BF, %

50 jobs queue scheduling. SSA computational complexity is $O(|R| * n * C)$ and it required almost a minute to perform the same job-flow scheduling in each experiment. Hindsight approach requires all completion of all the component algorithms. Thus, in our experiment setup Hindsight algorithm was executed for almost 3 min to obtain the resulting scheduling solution.

Figure 6 shows relative finish time difference for Hindsight, CoP and BF in the same experiment set. CoP provides the same 1–2% earlier finish times than BF, while *Hindsight* is able to provide a more solid 2–4% advantage.

7 Preference-Based Resources Coordination

7.1 Preference-Based Criteria Design

The same micro-scheduling heuristic may be used for a preference-based resources allocation. Introducing fair scheduling in VO requires mechanisms to influence scheduling results for VO stakeholders according to their private, group or common integral preferences. Individual users may have special requirements for the allocated resources, for example, total cost minimization or performance maximization. From the other hand, VO policies usually assume optimization of a joint resources usage according to accepted efficiency criteria. One straightforward example is a maximization of the resources load.

The proposed Slots Subset Algorithm (SSA) performs window search optimization by a general 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 window W. These criterion values z_i may represent different slot characteristics: time, cost, power, hardware and software features, etc.

In order to support both private and integral job-flow scheduling criteria we consider the following target criterion function in SSA for a single slot i:

$$z_i^* = z_i^I + \alpha z_i^U. \tag{4}$$

Here z_i^I and z_i^U represent criteria for integral and private jobs execution optimization correspondingly. z_i^I usually represents the same function for every job in the queue, while z_i^U reflects user requirements for a particular job optimization. $\alpha \in [0; +\infty]$ coefficient determines relative importance between private and integral optimization criteria.

By using SSA with z_i^* criterion and different α values it is possible to achieve a balance between private and integral job-flow scheduling preferences and policies.

For the integral job-flow scheduling criterion we used jobs finish time minimization ($z_i^I = -s_i.finishTime$) as a conventional metric for the overall resources load maximization.

7.2 Preference-Based Scheduling Algorithms and Analysis

For the SSA preference-based resources allocation efficiency study we implemented the following scheduling algorithms.

1. Firstly, we consider two conservative backfilling variations. *BFs* successively implements start time minimization for each job during the resources selection step. So, BFs criterion for slot i has the following form: $z_i = -s_i.startTime$. *BFf* implements finish time minimization: $z_i = -s_i.finishTime$. Both *BFs* and *BFf* algorithms represent extreme preference optimization scenario with $\alpha = 0$.
2. Secondly, we implement a preference-based conservative backfilling (BP) with SSA criterion of the following form: $z_i^* = -s_i.finishTime + \alpha z_i^U$ (4), where z_i^U depends on a private user criterion uniformly distributed between resources performance maximization ($z_i^U = s_i.nodePerformance$) and overall execution cost minimization ($z_i^U = -s_i.usageCost$). So in average half of jobs in the queue should be executed with performance maximization, while another half are interested in the total cost minimization.
   Considered α values covered different importance configurations of private and integral optimization criteria: $\alpha \in [0.01; 0.1; 1; 10; 100; 1000]$.
3. As a special extreme scheduling scenario with $\alpha \to \infty$ we implemented pure conservative backfilling with SSA criterion $z_i^* = z_i^U$, i.e. without any global parameters optimization.

The results of 1000 scheduling simulation scenarios are presented in Figs. 7, 8, 9 and 10. Each simulation experiment includes computing environment and job queue

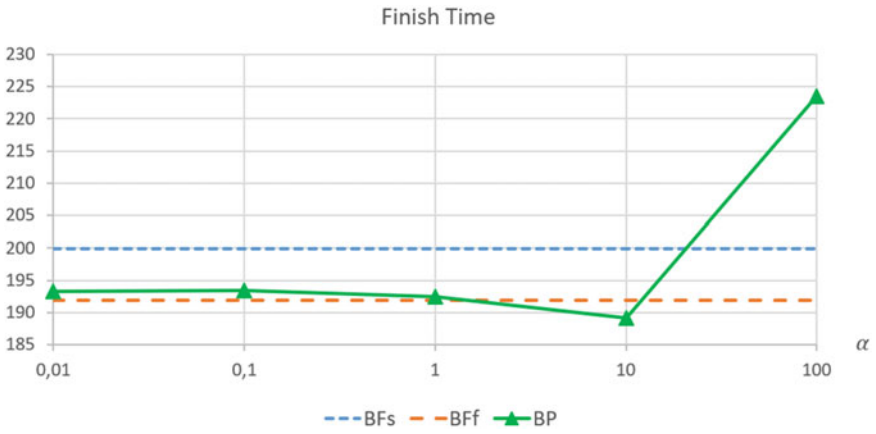

Fig. 7 Simulation results: average jobs finish time

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.

Figure 7 shows average jobs finish time for BFs, BFf and BP depending on α values on a logarithmic scale. BFs and BFf plots are represented by horizontal lines as the algorithms are independent of α.

As expected BFf provides 5% earlier jobs finish times compared to BFs. BFf with a job finish time minimization considers both job start time and runtime. In computing environments with heterogeneous resources job runtime may vary and depends on the selected resources performance. Thus, BFf implements more accurate strategy for the resources load optimization and a job-flow scheduling efficiency.

Similar results may be observed on Fig. 8 presenting average job queue execution makespan. This time the advantage of BFf by the makespan criterion exceeds 10%.

Interestingly, with $\alpha = 10$ BP provides even earlier average jobs finish time compared to BFf. In such configuration finish time minimization remains an important factor, while private performance and cost optimization lead to a more efficient resources sharing. At the same time BFf increases advantage by makespan criterion (Fig. 8) as some jobs in BP require more specific resources combinations generally available later in time.

Figures 9 and 10 show scheduling results for considered private criteria: average job execution cost and allocated resources performance. BPc and BPp in Figs. 9 and 10 represent BP scheduling results for jobs subsets with cost and performance private optimization correspondingly. Dashed lines show limits for BP, BPc and BPp, obtained in a pure private optimization scenario ($\alpha \to \infty$) without the integral finish time minimization.

The figures show that even with relatively small α values BP implements considerable resource share between BPc and BPp jobs according to the private preferences. The difference reaches 7% in cost and 5% in performance for $\alpha = 0.01$.

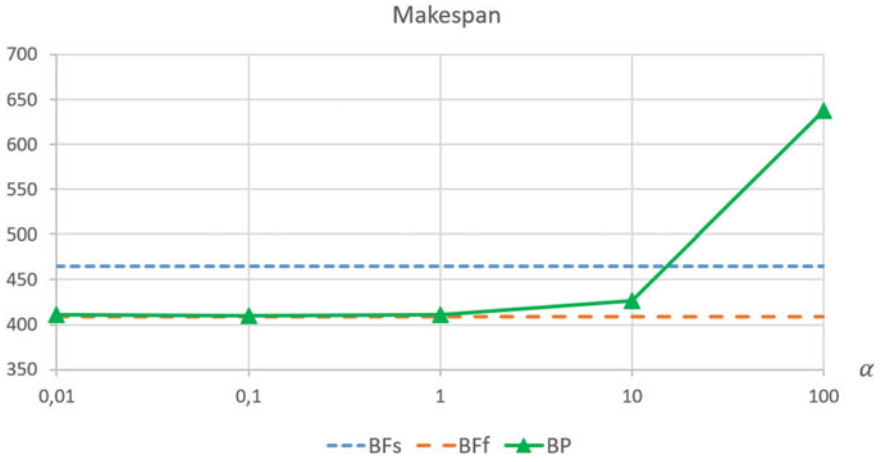

Fig. 8 Simulation results: average jobs queue execution makespan

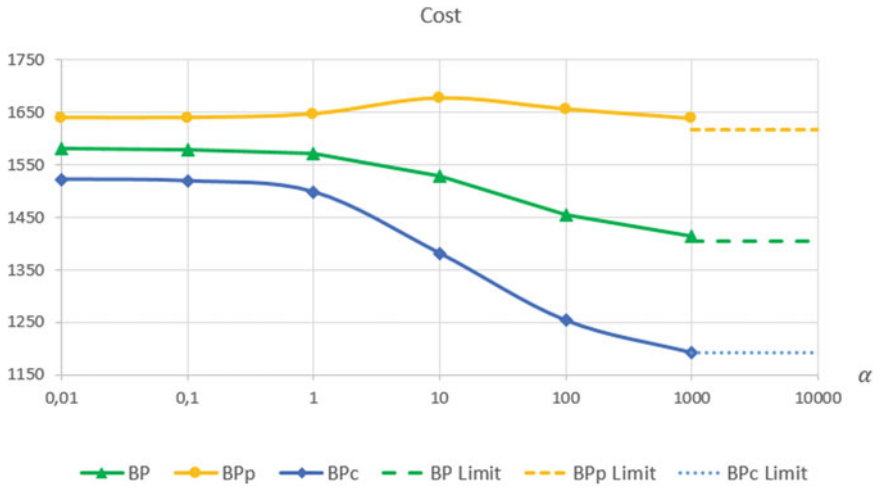

Fig. 9 Simulation results: average jobs execution cost

More noticeable separation up to 30–40% is observed with $\alpha > 1$. With higher importance of the private criteria, BP selects more specific resources and increasingly diverges from the backfilling finish time procedure and corresponding jobs execution order. The values obtained by BP with $\alpha = 100$ are close to the practical limits provided by the pure private criteria optimizations.

We may conclude from Figs. 7, 8, 9 and 10 that by changing a mutual importance of private and integral scheduling criteria it is possible to find a trade-off solution. Even the smallest α values are able to provide a considerable resources distribution according to VO users private preferences. At the same time BP with $\alpha < 10$ main-

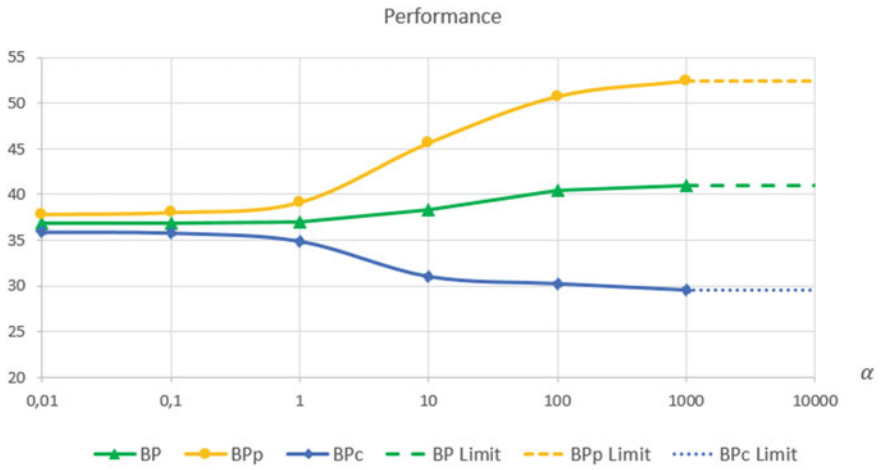

Fig. 10 Simulation results: average performance of the allocated resources

tains adequate resources utilization efficiency comparable with BFf and provides even more efficient preference-based resource share.

8 Conclusion and Future Work

In this work, we address the problems of a dependable and efficient resources co-allocation for parallel jobs execution in heterogeneous computing environments using the micro-scheduling technique. For this purpose a general window allocation algorithm was proposed along with four practical micro-scheduling implementations including dependable, coordinated and preference-based scheduling optimization. Coordinated micro-scheduling approach performs secondary optimization based on a baseline main scheduling procedure. A family of micro-scheduling algorithms may be used for a joint hindsight solution to prepare different job-flow execution strategies.

Sections 4–7 discuss different scheduling problems and provide corresponding simulation results and analysis. Dependable resources allocation may be used for an efficient placement of the execution windows against unreliable and highly utilized resources. Coordinated placement algorithm may improve backfilling scheduling results by selecting resources with a special set of heuristic meta-rules. Hindsight solution may be formed based on a family of different micro-scheduling algorithms to precalculate their scheduling outcomes and to choose the most appropriate strategy for the whole job-flow execution. Composite target optimization criteria are able to follow multiple optimization preferences, thus providing fair resources share between single HPCS users and administrators.

The main drawback for the whole micro-scheduling approach is a relatively high computational complexity of the core general resources allocation algorithms SSA. In our further work, we will refine a general resource co-allocation scheme in order to decrease its computational complexity.

Acknowledgements 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. Dimitriadou, S.K., Karatza, H.D.: Job Scheduling in a distributed system using backfilling with inaccurate runtime computations. In: Proceedings 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 Intell. Humaniz. Comput. **6**(6), 733–740 (2015)
3. Buyya, R., Abramson, D., Giddy, J.: Economic models for resource management and scheduling in grid computing. J. Concurr. Comput. Pract. Exp. **5**(14), 1507–1542 (2002)
4. Calheiros, R.N., Ranjan, R., Beloglazov, A., De Rose, C.A.F., Buyya, R.: CloudSim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms. J. Softw. Pract. Exp. **41**(5.1), 23–50 (2011)
5. Samimi, P., Teimouri, Y., Mukhtar M.: A combinatorial double auction resource allocation model in cloud computing. J. Inf. Sci. **357**(C), 201–216 (2016)
6. Foster, I., Kesselman C., Tuecke S.: The anatomy of the grid: enabling scalable virtual organizations. Int. J. High Perform. Comput. Appl. **15**(5.4), 200–222 (2001)
7. 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)
8. Rodero, I., Villegas, D., Bobroff, N., Liu, Y., Fong, L., Sadjadi, S.M.: Enabling interoperability among grid meta-schedulers. J. Grid Comput. **11**(5.2), 311–336 (2013)
9. Carroll, T., Grosu, D.: Formation of virtual organizations in grids: a game-theoretic approach. Econ. Models Algorithms Distrib. Syst. **22**(14), 63–81 (2009)
10. Rzadca, K., Trystram, D., Wierzbicki, A.: Fair game-theoretic resource management in dedicated grids. In: IEEE International Symposium on Cluster Computing and the Grid (CCGRID 2007), pp. 343–350. Rio De Janeiro, Brazil, IEEE Computer Society (2007)
11. Vasile, M., Pop, F., Tutueanu, R., Cristea, V., Kolodziej, J.: Resource-aware hybrid scheduling algorithm in heterogeneous distributed computing. J. Future Gener. Comput. Syst. **51**, 61–71 (2015)
12. Penmatsa, S., Chronopoulos, A.T.: Cost Minimization in Utility Computing Systems. Concurrency and Computation: Practice and Experience, Wiley, vol. 16(5.1), pp. 287–307 (2014)
13. 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)
14. Mutz, A., Wolski, R., Brevik, J.: Eliciting honest value information in a batch-queue environment. In: 8th IEEE/ACM International Conference on Grid Computing, pp. 291–297. New York, USA (2007)
15. Takefusa, A., Nakada, H., Kudoh, T., Tanaka, Y.: An advance reservation-based co-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)
16. Toporkov, V., Toporkova, A., Tselishchev, A., Yemelyanov, D.: Slot selection algorithms in distributed computing. J. Supercomput. **69**(5.1), 53–60 (2014)
17. Aida, K., Casanova, H.: Scheduling mixed-parallel applications with advance reservations. In: 17th IEEE International Symposium on HPDC, pp. 65–74. IEEE CS Press, New York (2008)
18. Elmroth, E., Tordsson J.: A Standards-based grid resource brokering servicesupporting advance reservations, co-allocation and cross-grid interoperability. J. Concurr. Comput. Pract. Exp. **25**(18), 2298–2335 (2009)
19. 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)
20. Garg, S.K., Konugurthi, P., Buyya, R.: A Linear programming-driven genetic algorithm for meta-scheduling on utility grids. Int. J. Parallel Emergent Distrib. Syst. **26**, 493–517 (2011)
21. Toporkov, V., Toporkova, A., Bobchenkov, A., Yemelyanov, D.: Resource selection algorithms for economic scheduling in distributed systems. In: Proceedings International Conference on Computational Science, ICCS 2011, June 1–3, 2011, Singapore, Procedia Computer Science. Elsevier, vol. 4. pp. 2267–2276 (2011)
22. Kovalenko, V.N., Koryagin, D.A.: The grid: analysis of basic principles and ways of application. J. Program. Comput. Softw. **35**(5.1), 18–34 (2009)
23. Makhlouf, S., Yagoubi, B.: Resources co-allocation strategies in grid computing. In: CIIA, vol. 825, CEUR Workshop Proceedings (2011)
24. 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)
25. 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)
26. Shmueli, E., Feitelson, D.G.: Backfilling with lookahead to optimize the packing of parallel jobs. J. Parallel and Distrib. Comput. **65**(9), 1090–1107 (2005)
27. 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)
28. Toporkov, V., Yemelyanov, D., Toporkova, A.: Preference based and fair resources selection in grid VOs. In: Malyshkin, V. (ed.) PaCT 2019, LNCS 11657, Springer Nature Switzerland AG, pp. 80–92 (2019)
29. Toporkov, V., Yemelyanov, D.: Coordinated resources allocation for dependable scheduling in distributed computing. In: Zamojski, W., et al. (eds.) DepCoS-RELCOMEX 2019, AISC 987, Springer Nature Switzerland AG, pp. 515–524 (2020)