Slot Selection and Co-allocation for Economic Scheduling in Distributed Computing

Victor Toporkov[1], Alexander Bobchenkov[1], Anna Toporkova[2], Alexey Tselishchev[3], and Dmitry Yemelyanov[1]

[1] Computer Science Department, Moscow Power Engineering Institute, ul. Krasnokazarmennaya, 14, Moscow, 111250, Russia
{ToporkovVV,BobchenkovAV,YemelyanovDM}@mpei.ru
[2] Moscow State Institute of Electronics and Mathematics, Bolshoy Trekhsvyatitelsky per., 1-3/12, Moscow, 109028, Russia
annastan@mail.ru
[3] European Organization for Nuclear Research (CERN), Geneva, 23, 1211, Switzerland
Alexey.Tselishchev@cern.ch

Abstract. In this paper, we present slot selection algorithms for job batch scheduling in distributed computing with non-dedicated resources. Jobs are parallel applications and these applications are independent. Existing approaches towards resource co-allocation and job scheduling in economic models of distributed computing are based on search of time-slots in resource occupancy schedules. A launch of a parallel job requires a co-allocation of a specified number of slots. The sought time-slots must match requirements of necessary span, computational resource properties, and cost. Usually such scheduling methods consider only one suited variant of time-slot set. This paper discloses a scheduling scheme that features multi-variant search. Two algorithms of linear complexity for search of alternative variants are proposed. Having several optional resource configurations for each job makes an opportunity to perform an optimization of execution of the whole batch of jobs and to increase overall efficiency of scheduling.

Keywords: Scheduling, co-allocation, slot, resource request, job, batch, task..

1 Introduction

Economic models for resource management and scheduling are very effective in distributed computing with non-dedicated resources, including Grid [1, 2], utility computing [3], cloud computing [4], and multiagent systems [5]. There is a good overview of some approaches to forming of different deadline and budget constrained strategies of economic scheduling in [6]. In [7] heuristic algorithms for slot selection based on user defined utility functions are introduced.

While implementing economic policy, resource brokers usually optimize the performance of a specific application [1, 6, 7] in accordance with the application-level scheduling concept [8]. When establishing virtual organizations (VO), the optimization is performed for the job-flow scheduling [9, 10]. Corresponding

V. Malyshkin (Ed.): PaCT 2011, LNCS 6873, pp. 368–383, 2011.

functions are implemented by a hierarchical structure that consists of the metascheduler and subordinate resource managers or local batch-job management systems [8-10]. In a model, proposed in [2] there is an interaction between users launching their jobs, owners of computational resources, and VO administrators. The interests of the said users and owners are often contradictory. Each independent user is interested in the earliest launch of his job with the lowest costs (for example, the resource usage fee) and the owners, on the contrary, try to make the highest income from their resources. VO administrators are interested in maximizing the whole VO performance in the way that satisfies both users and owners [8].

In this work, economic mechanisms are applied for job batch scheduling in VO. It is supposed that resources are non-dedicated, that is along with global flows of external users' jobs, owner's local job flows exist inside the resource domains (clusters, computational nodes equipped with multicore processors, etc.). The metascheduler [8-10] implements the VO economic policy based on local system schedules. The local schedules are sets of slots coming from local resource managers or schedulers in the node domains. A single slot is a time span that can be assigned to a task, which is a part of a parallel job. We assume that job batch scheduling runs iteratively on periodically updated local schedules [2]. The launch of any job requires co-allocation of a specified number of slots. The challenge is that slots associated with different resources may have arbitrary start and finish points that do not coincide. In its turn, tasks of the parallel job must start synchronously. If the necessary number N of slots with attributes matching the resource request is not accumulated then the job will not be launched. This job is joined another batch, and its scheduling is postponed till the next iteration.

We propose two algorithms for slot selection that feature linear complexity $O(m)$, here m is the number of available time-slots. Existing slot search algorithms, such as backfilling [11, 12], do not support environments with heterogeneous and non-dedicated resources, and, moreover, their execution time grows substantially with increase of the number of slots. Backfilling is able to find an exact number of concurrent slots for tasks with identical resource requirements and homogeneous resources. We take a step further, so proposed algorithms deal with heterogeneous resources and jobs with different tasks.

The paper is organized as follows. Section 2 introduces a scheduling scheme. In section 3 two algorithms for search of alternative slot sets are considered. The example of slot search is presented in section 4. Simulation results for comparison of proposed algorithms are described in Section 5. Experimental results are discussed in section 6. Section 7 summarizes the paper and describes further research topics.

2 Scheduling Scheme

Let $J = \{j_1,..., j_n\}$ denote a batch consisting of n jobs. A job $j_i, i = 1,...,n$, schedule is formed as a set $\bar{s}_i$ of time slots. A job batch schedule is a set of slot sets (*a slot combination*) $\bar{s} = (\bar{s}_1,...,\bar{s}_n)$ for jobs composing this batch. The job resource requirements are arranged into a resource request containing a wall clock time t_i and characteristics of computational nodes (clock speed, RAM volume, disk space,

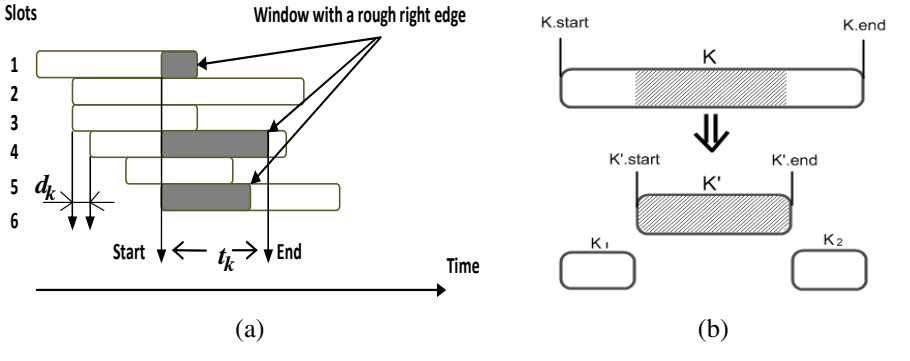
operating system etc.). The slot set $\bar{s}_i$ fits the job j_i, if it meets the requirements of number and type of resources, cost and the job wall time t_i. We suppose that for each job j_i in *the current scheduling iteration* there is at least one suitable set $\bar{s}_i$. Otherwise, the scheduling of the job is postponed to *the next iteration*. Every slot set $\bar{s}_i$ for the execution of the i-th job in the batch $J = \{j_1,..., j_n\}$ is defined with a pair of parameters, the cost $c_i(\bar{s}_i)$ and the time $t_i(\bar{s}_i) \le t_i$ for the resource usage, $c_i(\bar{s}_i)$ denotes a total cost of slots in the set $\bar{s}_i$ and $t_i(\bar{s}_i)$ denotes a time elapsed from the start till the end of the i-th job. Notice that different jobs j_{i_1}, $j_{i_2} \in J$ have different resource requirements, and $c_{i_1}(\bar{s}) \neq c_{i_2}(\bar{s})$, $t_{i_1}(\bar{s}) \neq t_{i_2}(\bar{s})$, $i_1, i_2 \in \{1,...,n\}$, even if jobs j_{i_1}, j_{i_2} are allocated to the same slot set $\bar{s}$. Here $c_{i_1}(\bar{s})$, $c_{i_2}(\bar{s})$ are functions of a cost C of slot usage per time unit.

Two problems have to be solved for job batch scheduling. First, selecting alternative slot sets for jobs of the batch that meet the requirements (resource, time, and cost). Second, choosing the slot combination $\bar{s} = (\bar{s}_1,..., \bar{s}_n)$ that would be the efficient or optimal one in terms of the whole job batch execution.

To realize the scheduling scheme described above, first of all, we need to propose the algorithm of finding a set of alternative slot sets.

Slots are arranged by start time in non-decreasing order in a list (Fig. 1 (a)). In Fig. 1 (a), d_k denotes a time offset of the slot s_k in relation to the slot s_{k-1}.

In the case of homogeneous nodes, the set $\bar{s}_i$ of slots for the job j_i is represented with a rectangle window $N \times t_i(\bar{s}_i)$. It does not mean that processes of any parallel job would finish their work simultaneously. Here a time length of the window is the time $t_i(\bar{s}_i)$ dedicated for the resource usage. In the case of nodes with varying performance, that will be a window with a rough right edge, and the resource usage time is defined by the execution time t_k of the task that is using the slowest node (see Fig. 1 (a)).



(a) (b)

Fig. 1. Slot selection for heterogeneous resources: an ordered list of available slots (a); slot subtraction (b)

The scheduling scheme works iteratively, during *the iteration* it consecutively searches for a single alternative for each job of the batch. In case of successful slot selection for the i-th job, the list of vacant slots for the $(i+1)$-th job is modified. All time spans that are involved in the i-th job alternative are excluded from the list of vacant slots (Fig. 1 (b)). The selection of slots for the $(i+1)$-th job is performed on the list modified with the method described above. Suppose, for example, that there is a slot K' among the slots belonging to the same window. Then its start time equals to the start time of the window: K′.startTime = window.startTime and its end time equals to K′.end=K′.start $+t_{k'}$, where $t_{k'}$ is the evaluation of a task runtime on the appropriate resource, on which the slot K' is allocated. Slot K′ should be subtracted from the original list of available slots. First, we need to find slot K – the slot, part of which is K′ and then cut K′ interval from K. So, in general, we need to remove slot K′ from the ordered slot list and insert two new slots K_1 and K_2. Their start, end times are defined as follows: K_1.startTime = K.startTime, K_1.endTime = K′.startTime, K_2.startTime = K′.endTime, K_2.endTime = K.endTime. Slots K_1 and K_2 have to be added to the slot list given that the list is sorted by non-decreasing start time order (see Fig. 1 (a)). Slot K_1 will have the same position in the list as slot K, since they have the same start time. If slots K_1 and K_2 have a zero time span, it is not necessary to add them to the list. After the last of the jobs is processed, the algorithm starts next search from the beginning of the batch and attempts to find other alternatives on the modified slot list. Alternatives found do not intersect in processor time, so every job could be assigned to some set of found slots without the revision of other jobs assignments. The search for alternatives ends when on the current list of slots the algorithm cannot find any suitable set of slots for any of the batch jobs. Implementation of the single alternative search algorithm becomes a serious question because characteristics of a resulting set of slots solely depend on it. Doing a search in every scheduling iteration imposes a requirement of an algorithm having complexity as low as possible. An optimization technique for choosing optimal or efficient slot combinations was proposed in [2]. It is implemented by dynamic programming methods using multiple criteria in accordance with the VO economic policy.

We consider two types of criteria in the context of our model. These are the execution cost and time measures for the job batch J using the suitable slot combination $\bar{s} = (\bar{s}_1,...,\bar{s}_n)$. The first criteria group includes the total cost of the job batch execution $C(\bar{s}) = \sum_{i=1}^{n} c_i(\bar{s}_i)$. The VO administration policy and, partially, users' interests are represented with the execution time criterion for all jobs of the batch $T(\bar{s}) = \sum_{i=1}^{n} t_i(\bar{s}_i)$. In order to forbid the monopolization of some resource usage by users, a limit B^* is put on the maximum value for a total usage cost of resources in the current scheduling iteration. We define B^* as a budget of the VO. The total slots occupancy time T^* represents owners' urge towards the balance of global (external) and local (internal) job shares. If we consider the single-criterion optimization of the

job batch execution, then every criterion $C(\bar{s})$ or $T(\bar{s})$ must be minimized with given constraints T^* or B^* for the interests of the particular party - the user, the owner and the VO administrator [2].

Let $g_i(\bar{s}_i)$ be the particular function, which determines the efficiency of the slot set $\bar{s}_i$ usage for the i-th job. In other words, $g_i(\bar{s}_i)=c_i(\bar{s}_i)$ or $g_i(\bar{s}_i)=t_i(\bar{s}_i)$. Let $f_i(Z_i)$ be the extreme value of the particular criterion using the slot combination $(\bar{s}_i,\bar{s}_{i+1},...,\bar{s}_n)$ for jobs $j_i, j_{i+1},..., j_n$, having Z_i as a total occupancy time or an usage cost. Let us define an admissible time value or a slot occupancy cost as $z_i(\bar{s}_i)$. Then $z_i(\bar{s}_i)\le Z_i \le Z^*$, where Z^* is the given limit. For example, if $z_i(\bar{s}_i)=t_i(\bar{s}_i)$, then $t_i(\bar{s}_i)\le T_i \le T^*$, where T_i is a total slots occupancy time $i, i+1,..., n$ and T^* is the constraint for values T_i, that is chosen with the consideration of balance between the global job flow (user-defined) and the local job flow (owner-defined). If, for example, $z_i(\bar{s}_i)=c_i(\bar{s}_i)$, then $c_i(\bar{s}_i)\le C_i \le B^*$, where C_i is a total cost of the resource usage for the jobs $i, i+1,..., n$, and B^* is the budget of the VO. In the scheme of backward run [2] $Z_1 = Z^*$, $z_i(\bar{s}_i)\le Z_i \le Z^*$, $Z_i = Z_{i-1} - z_{i-1}(\bar{s}_{i-1})$, having $1<i\le n$. Notice that $g_{i_1}(\bar{s})\ne g_{i_2}(\bar{s})$, $z_{i_1}(\bar{s})\ne z_{i_2}(\bar{s})$, $i_1, i_2 \in \{1,...,n\}$, even if jobs j_{i_1}, j_{i_2} are allocated to the same slot set $\bar{s}$.

The functional equation for obtaining a conditional (given $z_i(\bar{s}_i)$) extremum of $f_i(z_i(\bar{s}_i))$ for the backward run procedure can be written as follows:

$$f_i(Z_i) = \operatorname*{extr}_{s_i}\{g_i(\bar{s}_i) + f_{i+1}(Z_i - z_i(\bar{s}_i))\},\ i=1,...,n,\ f_{n+1}(Z_{n+1})\equiv 0, \tag{1}$$

where $g_i(\bar{s}_i)$ and $f_{i+1}(Z_i - z_i(\bar{s}_i))$ are cost or time functions.

For example, a limit put on the total time of slot occupancy by tasks may be expressed as:

$$T^* = \sum_{i=1}^{n} \sum_{s_i} [t_i(\bar{s}_i)/l_i], \tag{2}$$

where l_i is the number of admissible slot sets for the i-th job; $[\cdot]$ means the nearest to $t_i(\bar{s}_i)/l_i$ not greater integer.

The VO budget B^* may be obtained by formula (1) as the maximal income for resource owners with the given constraint T^* defined by (2):

$$B^* = \max_{s_i}\{c_i(\bar{s}_i) + f_{i+1}(T_i - t_i(\bar{s}_i))\}, \tag{3}$$

where $f_{i+1}(T_i - t_i(\bar{s}_i))$ is a cost function.

In the general case of the model [2], it is necessary to use a vector of criteria, for example, $<C(\bar{s}), D(\bar{s}), T(\bar{s}), I(\bar{s})>$, where $D(\bar{s}) = B^* - C(\bar{s})$, $I(\bar{s}) = T^* - T(\bar{s})$ and T^*, B^* are defined by (2), (3).

3 Slot Search Algorithms

Let us consider one of the resource requests associated with any job in the batch J. The resource request specifies N concurrent time-slots reserved for time span t with resource performance rate at least P and maximal resource price per time unit not higher, than C.

Class Slot is defined to describe a single slot:

```
public class Slot{
    public Resource cpu;     //resource on which the slot
                                          is allocated
     public int cash;        // usage cost per time unit
     public int start;       // start time
     public int end;         // end time
     public int length;      // time span
     …
   }
```

Class Window is defined to describe a single window:

```
public class Window {
     int id;                 // window id
     public int cash;        // total cost
     public int start;       // start time
     public int end;         // end time
     public int length;      // time span
     int slotsNumber;        // number of required slots
     ArrayList<Slot> slots; // window slots
           …
   }
```

Here a slot set search algorithm for a single job and resource charge per time unit is described. It is an **A**lgorithm based on **L**ocal **P**rice of slots (ALP) with *a restriction to the cost of individual slots*. Input data include available slots list, and slots being sorted by start time in ascending order (see Fig. 1(a)). The search algorithm guarantees examination of every slot of the list. If the necessary number N of slots is not accumulated, then the job scheduling is postponed until the next iteration.

1°. Sort the slots by start time in ascending order - see Fig. 1 (a).

2°. From the resulting slot list the next suited slot s_k is extracted and examined.

The slot s_k suits, if following conditions are met:

a) resource performance rate $P(s_k) \geq P$;

b) slot length (time span) is enough (depending on the actual performance of the slot's resource) $L(s_k) \geq tP(s_k)/P$ (see the condition **a)**);

c) resource charge per time unit $C(s_k) \leq C$.

If conditions **a)**, **b)**, and **c)** are met, the slot s_k is successfully added to the window list.

3°. The expiration of the slot length means that remaining slot length $L'(s_k)$, calculated like shown in **step 2**°**b**, is not enough assuming the k-th slot start is equal to the last added slot start: $L'(s_k) < (t - (T_{last} - T(s_k)))P(s_k)/P$, where $T(s_k)$ is the slot's start time, T_{last} is the last added slot's start time. Notice, in Fig. 1 (a), $d_k = T_{last} - T(s_k)$.

Slots whose time length has expired are removed from the list.

4°. Go to **step 2**°, until the window has N slots.

5°. **End** of the algorithm.

We can move only forward through the slot list. If we run out of slots before having accumulated N slots, this means a failure to find the window for a job and its scheduling is postponed by the metascheduler until the next batch scheduling iteration. Otherwise, the window becomes the alternative slot set for the job. ALP is executed for every job in the batch $J = \{j_1, ..., j_n\}$. Having succeeded in the search for window for the j_i-th job, the slot list is modified with subtraction of formed window slots (see Fig. 1 (b)). Therefore slots of the already formed slot set are not considered in processing the next job in the batch.

In the economic model [2] a user's resource request contains the maximal resource price requirement, that is a price which a user agrees to pay for resource usage. But this approach narrows the search space and restrains the algorithm from construction of a window with more expensive slots. The difference of the next proposed algorithm is that we replace maximal price C requirement by *a maximal budget of a job*. It is an **A**lgorithm based on **M**aximal job **P**rice (AMP). The maximal budget is counted as $S = CtN$, where t is a time span to reserve and N is the necessary number of slots. Then, as opposed to ALP, the search target is a window, formed by slots, whose total cost will not exceed the maximal budget S. In all other respects, AMP utilizes the same input data as ALP.

Let us denote additional variables as follows: N_S – current number of slots in the window; M_N – total cost of first N slots.

Here we describe AMP approach for a single job.

1°. Find the earliest start window, formed by N slots, using ALP excluding the condition **2**°**c** (see ALP description above).

2°. Sort window slots by their cost in ascending order.

Calculate total cost of first N slots M_N. If $M_N \le S$, go to **4**°, so the resulting window is formed by first N slots of the current window, others are returned to the source slot list. Otherwise, go to **3**°.

3°. Add the next suited slot to the list following to conditions **2**°**a** and **2**°**b** of ALP. Assign the new window start time and check expiration like in the **step 3**° of ALP.

If we have $N_S < N$, then repeat the current step. If $N_S \ge N$, then go to **step 2**°.

If we ran out of slots in the list, and $N_S < N$, then we have algorithm failure and no window is found for the job.

4°. **End** of the algorithm.

We can state three main features that distinguish the proposed algorithms. First, both algorithms consider resource performance rates. This allows forming time-slot windows with uneven right edge (we suppose that all concurrent slots for the job must start simultaneously). Second, both algorithms consider maximum price constraint which is imposed by a user. Third, both algorithms have linear complexity $O(m)$, where m is the number of available time-slots: we move only forward through the list, and never return or reconsider previous assignments.

The backfill algorithm [11, 12] has quadratic complexity $O(m^2)$, assuming that every node has at least one local job scheduled. Although backfilling supports parallel jobs and is able to find a rectangular window of concurrent slots, this can be done provided that all available computational nodes have equal performance (processor clock speed), and tasks of any job have identical resource requirements.

4 AMP Search Example

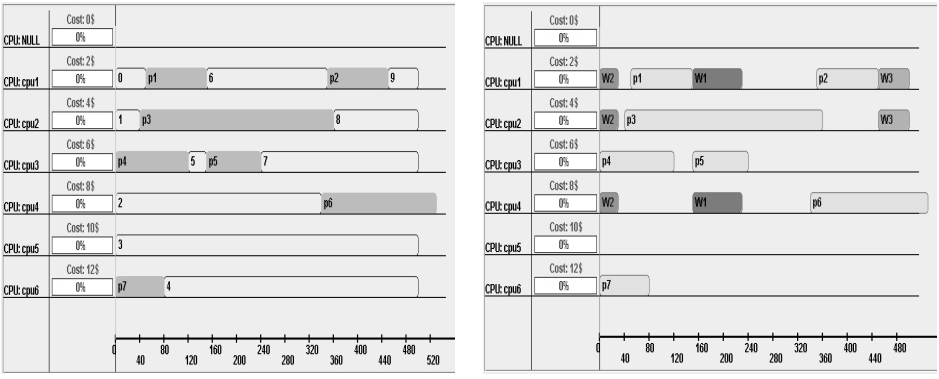
In this example for the simplicity and ease of demonstration we consider the problem with a uniform set of resources, so the windows will have a rectangular shape without the rough right edge. Let us consider the following initial state of the distributed computing environment. In this case there are six computational nodes `cpu1 - cpu6` (resource lines) (Fig. 2 (a)). Each has its own unit cost (cost of its usage per time unit). In addition there are seven local tasks `p1 - p7` already scheduled for the execution in the system under consideration. Available system slots are drawn as rectangles 0...9 - see Fig. 2 (a). Slots are sorted by non-decreasing time of start and the order number of each slot is indicated on its body. For the clarity, we consider the situation where the scheduling iteration processes the batch of only three jobs with the following resource requirements.

Job 1 requirements:
- the number of required computational nodes: 2;
- runtime: 80;
- maximum total "window" cost per time: 10.

Job 2 requirements:
- the number of required computational nodes: 3;
- runtime: 30;
- maximum total "window" cost per time: 30.

Job 3 requirements:
- the number of required computational nodes: 2;
- runtime: 50;
- maximum total "window" cost per time: 6.

According to AMP alternatives search, first of all, we should form a list of available slots and find the earliest alternative (the first suitable window) for the first job of the batch. We assume that **Job 1** has the highest priority, while **Job 3** possesses the lowest priority. The alternative found for **Job 1** (see Fig. 2 (b)) has two rectangles on `cpu1` and `cpu4` resource lines on a time span [150, 230] and named W1. The total cost per time unit of this window is 10. This is the earliest possible window satisfying



(a) (b)

Fig. 2. AMP search example: initial state of environment (a); alternatives found after the first iteration (b)

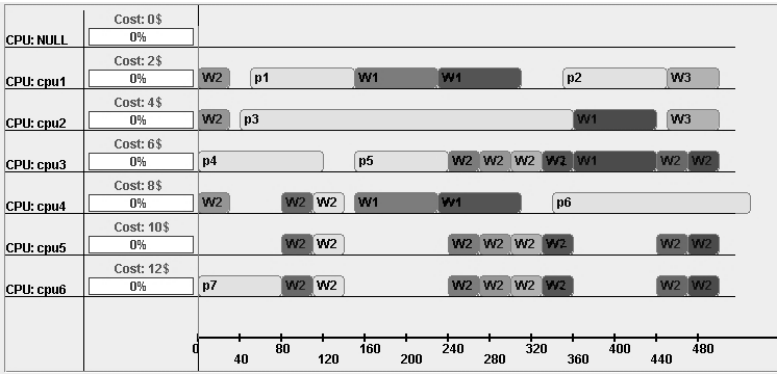


Fig. 3. The final chart of all alternatives found during AMP search

the job's resource request. Note that other possible windows with earlier start time are not fit the total cost constraint. Then we need to subtract this window from the list of available slots and find the earliest suitable set of slots for the second batch job on the modified list.

Further, a similar operation for the third job is performed (see Fig. 2 (b)). Alternative windows found for each job of the batch are named W1, W2, and W3 respectively. The earliest suitable window for the second job (taking into account alternative W1 for the first job) consists of three slots on the `cpu1`, `cpu2` and `cpu4` resource lines with a total cost of 14 per time unit. The earliest possible alternative for the third job is W3 window on a time span of [450, 500]. Further, taking into account the previously found alternatives, the algorithm performs the searching of next alternative sets of slots according to the job priority. The algorithm makes an attempt to find alternative windows for each batch job.

Figure 3 illustrates the final chart of all alternatives found during search.

Note that in ALP approach the restriction to the cost of individual slots would be equal to 10 for **Job 2** (as it has a restriction of total cost equals to 30 for a window allocated on three nodes). So, the computational resource `cpu6` with a 12 usage cost value is not considered during the alternative search with ALP algorithm. However it is clear that in the presented AMP approach eight alternatives have been found. They use the slots allocated on the `cpu6` resource line, and thus fit in the limit of the window total cost.

5 Simulation Studies

The experiment consists in comparison of job batch scheduling results using different sets of suitable slots founded with described above AMP and ALP approaches. The alternatives search is performed on the same set of available vacant system slots. The generation of an ordered list of vacant slots and a job batch is performed during the single simulated scheduling iteration. To perform a series of experiments we found it more convenient to generate the ordered list of available slots (see Fig. 1 (a)) with preassigned set of features instead of generating the whole distributed system model and obtain available slots from it.

SlotGenerator and **JobGenerator** classes are used to form the ordered slot list and the job batch during the experiment series. Here is the description of the input parameters and values used during the simulation. All job batch and slot list options are random variables that have a uniform distribution inside the identified intervals.

SlotGenerator

- number of available system slots in the ordered list varies in [120, 150];
- length of the individual slot is in [50, 300] - here we propose that the length of initial slot are varies greatly, and it will be more during the search procedure;
- computational nodes performance range is [1, 3], so that the environment is relatively homogeneous;
- the probability that the nearby slots in the list have the same start time is 0.4; this property represents that in real systems resources are often reserved and occupied in domains (clusters), so that after the release, the appropriate slots have the same start time;
- the time between neighboring slots in the list is in [0, 10], so that at each moment of time we have at least five different slots ready for utilization;
- the price of the slot is randomly selected from [0.75p, 1.25p], where p = (1.7) to the (Node Performance); here we propose that the price is a function of performance with some element of randomness.

JobGenerator

- number of jobs in the batch is in [3, 7]; the batch is not very big because we have to distribute all the jobs in order to carry out the experiment;
- number of computational nodes to find is in [1, 6];

- length (representing the complexity) of the job is in [50, 150]; this value is corresponds to the initial values of the generated slots;
- the minimum required nodes' performance is in [1, 2]; some jobs will require slots, allocated on resources with high ($P \geq 2$) performance - it is a factor of job heterogeneity.

Let us consider the task of slot allocation during the *job batch execution time minimization*: $\min_{s_i} T(\bar{s})$ with the constraint $B*$.

The number of 25000 simulated scheduling iterations was carried out. Only those experiments were taken into account when all of the batch jobs had at least one suitable alternative of execution. AMP algorithm exceeds ALP by 35% with respect to $T(\bar{s})$. An average job execution time for alternatives found with ALP was 59.85, and for alternatives found with AMP - 39.01 (Fig. 4 (a)).

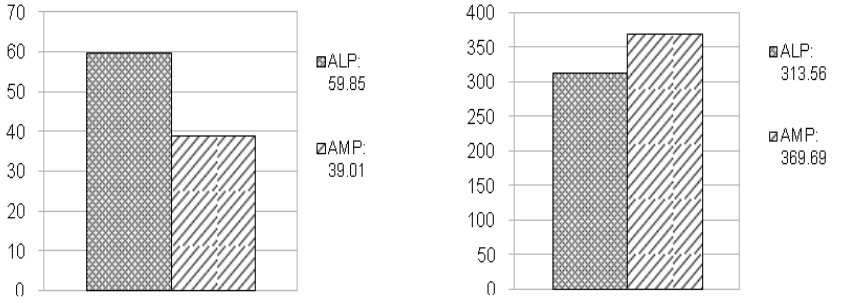


Fig. 4. Job batch execution time minimization: average job execution time (a); average job execution cost (b)

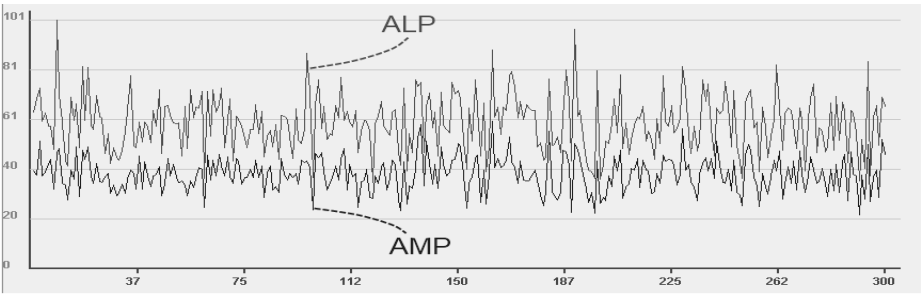


Fig. 5. Average job execution time comparison for ALP and AMP for the first 300 experiments in the job batch execution time minimization

It should be noted, that an average job execution cost for ALP method was 313.56, while using AMP algorithm the average job execution cost was 369.69, that is 15% more – see Fig. 4 (b).

Figure 5 illustrates scheduling results comparison for the first 300 experiments (the horizontal axis). It shows an observable gain of AMP method in every single experiment. The total number of alternatives found with ALP was 258079 or an average of 7.39 for a job. At the same time the modified approach (AMP) found 1160029 alternatives or an average of 34.28 for a single job. According to the results of the experiment we can conclude that the use of AMP minimizes the batch execution time though the cost of the execution increases. Relatively large number of alternatives found increases the variety of choosing the efficient slot combination [2] using the AMP algorithm.

Now let us consider the task of slot allocation during the *job batch execution cost minimization*: $\min_{s_i} C(\bar{s})$ with the constraint $T*$. The results of 8571 single experiments in which all jobs were successfully assigned to suitable slot combinations using both slot search procedures were collected.

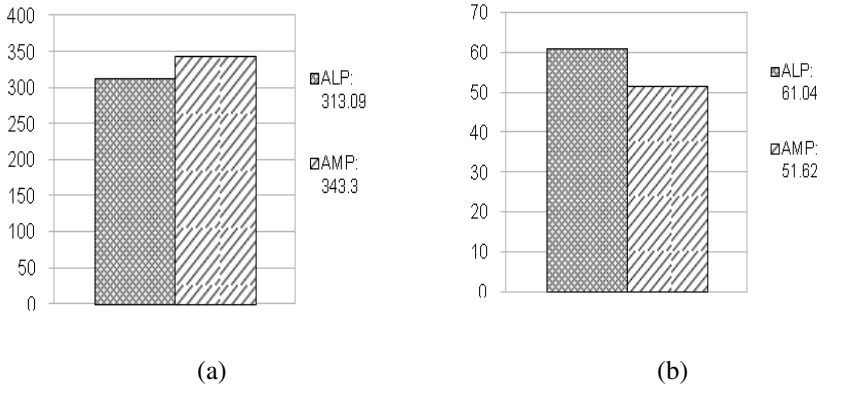


Fig. 6. Job batch execution cost minimization: average job execution cost (a); average job execution time (b)

The average job execution cost for ALP algorithm was 313.09, and for alternatives found with AMP - 343.3. It shows the advantage of only 9% for ALP approach over AMP (Fig. 6 (a)). The average job execution time for alternatives found with ALP was 61.04. Using AMP algorithm the average job execution time was 51.62, that is 15% less than using ALP (Fig. 6 (b)).

The average number of slots processed in a single experiment was 135.11. This number coincides with the average number of slots for all 25000 experiments, which indicates the absence of decisive influence of the available slots number to the number of successfully scheduled jobs.

The average number of jobs in a single scheduling iteration was 4.18. This value is smaller than average over all 25000 experiments. With a large number of jobs in the

batch ALP often was not able to find alternative sets of slots for certain jobs and an experiment was not taken into account.

The average number of alternatives found with ALP is 253855 or an average of 7.28 per job. AMP algorithm was able to found the number of 115116 alternatives or an average of 34.23 per job. Recall that in previous set of experiments these numbers were 7.39 and 34.28 alternatives respectively.

6 Experimental Results Analysis

Considering the results of the experiments it can be argued that the use of AMP approach on the stage of alternatives search gives clear advantage compared to the usage of ALP. Advantages are mostly in the large number of alternatives found and consequently in the flexibility of choosing an efficient schedule of batch execution, as well as that AMP provides the job batch execution time less than ALP.

AMP allows searching for alternatives among the relatively more expensive computational nodes with higher performance rate. Alternative sets of slots found with ALP are more homogeneous and do not differ much from each other by the values of the total execution time and cost. Therefore job batch distributions obtained by optimizations based on various criteria [2] do not differ much from each other either.

The following factors should explain the results. First, let us consider the peculiarities of calculating a slot usage total cost $C_t = CtN / P$, where C is a cost of slot usage per time unit, P is a relative performance rate of the computational node on which the slot is allocated, and t is a time span, required by the job in assumption that the job will be executed on the etalon nodes with $P = 1$. In the proposed model, generally, the higher the cost C of slot the higher the performance P of the node on which this slot is allocated. Hence, the job execution time t / P correspondingly less. So, the high slot cost per time unit is compensated by high performance of the resource, so it gets less time to perform the job and less time units to pay for. Thus, in some cases the total execution cost may remain the same even with the more "expensive" slots. The value C / P *i*s a measure of a slot price/quality ratio. By setting in the resource request the maximum cost C of an individual slot and the minimum performance rate P of a node the user specifies the minimum acceptable value of price/quality. The difference between ALP and AMP approaches lies in the fact that ALP searches for alternatives with suitable price/quality coefficient among the slots with usage cost no more than C. AMP performs the search among all the available slots (naturally, both algorithms still have the restriction on the minimum acceptable node performance). This explains why alternatives found with AMP have on the average less execution time. Second, it should be noted that during the search ALP considers available slots regardless of the entire window. The ALP window consists of slots each of which has the cost value no more than C. At the same time AMP is more flexible. If at some step a slot with cost on δ cheaper than C was added to the desired window, then AMP algorithm will consider to add slots with cost on the δ more expensive than C on the next steps. Naturally, in this case it will take

into account the total cost restriction. That explains, why the average job execution cost is more when using the AMP algorithm, it seeks to use the entire budget to find the earliest suitable alternative.

Another remark concerns the algorithms' work on the same set of slots. It can be argued that *any* window which could be found with ALP can also be found by AMP. However, there could be windows found with AMP algorithm which can't be found with a conventional ALP. It is enough to find a window that would contain at least one slot with the cost more than C.

This observation once again explains the advantage of AMP approach by a number of alternatives found. The deficiency of AMP scheme is that batch execution cost on the average always higher than the execution cost of the same batch scheduled using ALP algorithm. It is a consequence of a specificity of determining the value of a budget limit and the stage of job batch scheduling [2]. However, it is possible to reduce the job batch execution cost reducing the user budget limit for every alternative found during the search, which in this experiment was limited to $S = CtN$. This formula can be modified to $S = \rho CtN$, where ρ is a positive number less than one, e.g. 0.8. Variation of ρ allows to obtain flexible distribution schedules on different scheduling periods, depending on the time of day, resource load level, etc.

7 Conclusion and Future Work

In this paper, we address the problem of independent batch jobs scheduling in heterogeneous environment with non-dedicated resources.

The scheduling of the job batch consists of two steps. First of all, the independent sets of suitable slots (alternatives of execution) have to be found for every job of the batch. The second step is selecting the efficient combination of alternative slot sets, that is the set of slot sets for the batch. The feature of the approach is searching for a number of job alternative executions and consideration of economic policy in VO and financial user requirements on the stage of a single alternative search. For this purpose ALP and AMP approaches for slot search and co-allocation were proposed and considered. According to the experimental results it can be argued that AMP allows to find on the average more rapid alternatives and to perform jobs in a less time. But the of job batch execution using AMP is relatively higher. AMP exceeds ALP significantly during the batch execution time minimization. At the same time during the execution cost minimization the gain of ALP method is negligible. It is worth noting, that on the same set of vacant slots AMP in comparison with ALP finds several time more execution alternatives.

In our future work we will address the problem of slot selection for the whole job batch at once and not for each job consecutively. Therewith it is supposed to optimize the schedule "on the fly" and not to allocate a dedicated phase during each scheduling iteration for this optimization. We will research pricing mechanisms that will take into account supply-and-demand trends for computational resources in virtual organizations.

The necessity of guaranteed job execution at the required quality of service causes taking into account the distributed environment dynamics, namely, changes in the number of jobs for servicing, volumes of computations, possible failures of

computational nodes, etc. [13]. As a consequence, in the general case, a set of versions of scheduling, or a strategy, is required instead of a single version [13, 14]. In our further work we will refine resource co-allocation algorithms in order to integrate them with scalable co-scheduling strategies.

Acknowledgments. This work was partially supported by the Council on Grants of the President of the Russian Federation for State Support of Leading Scientific Schools (SS-7239.2010.9), the Russian Foundation for Basic Research (grant no. 09-01-00095), the Analytical Department Target Program "The higher school scientific potential development" (projects nos. 2.1.2/6718 and 2.1.2/13283), and by the Federal Target Program "Research and scientific-pedagogical cadres of innovative Russia" (State contracts nos. P2227 and 16.740.11.0038).

References

1. Garg, S.K., Buyya, R., Siegel, H.J.: Scheduling Parallel Applications on Utility Grids: Time and Cost Trade-off Management. In: 32nd Australasian Computer Science Conference (ACSC 2009), pp. 151–159 (2009)
2. Toporkov, V.V., Toporkova, A., Tselishchev, A., Yemelyanov, D., Bobchenkov, A.: Economic Models of Scheduling in Distributed Systems. In: Walkowiak, T., Mazurkiewicz, J., Sugier, J., Zamojski, W. (eds.) Monographs of System Dependability. Dependability of Networks, vol. 2, pp. 143–154. Oficyna Wydawnicza Politechnki Wroclawskiej, Wroclaw (2010)
3. Degabriele, J.P., Pym, D.: Economic Aspects of a Utility Computing Service. Technical Report HPL-2007-101, Trusted Systems Laboratory, HP Laboratories, Bristol (2007)
4. Pandey, S., Barker, A., Gupta, K.K., Buyya, R.: Minimizing Execution Costs when Using Globally Distributed Cloud Services. In: 24th IEEE International Conference on Advanced Information Networking and Applications, pp. 222–229. IEEE Press, New York (2010)
5. Bredin, J., Kotz, D., Rus, D.: Economic Markets as a Means of Open Mobile-Agent Systems. In: Mobile Agents in the Context of Competition and Cooperation (MAC3), pp. 43–49 (1999)
6. Buyya, R., Abramson, D., Giddy, J.: Economic Models for Resource Management and Scheduling in Grid Computing. J. of Concurrency and Computation: Practice and Experience 5(14), 1507–1542 (2002)
7. Ernemann, C., Hamscher, V., Yahyapour, R.: Economic Scheduling in Grid Computing. In: Feitelson, D.G., Rudolph, L., Schwiegelshohn, U. (eds.) JSSPP 2002. LNCS, vol. 2537, pp. 128–152. Springer, Heidelberg (2002)
8. 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, Dordrecht (2003)
9. Toporkov, V.: Application-Level and Job-Flow Scheduling: An Approach for Achieving Quality of Service in Distributed Computing. In: Malyshkin, V. (ed.) PaCT 2009. LNCS, vol. 5698, pp. 350–359. Springer, Heidelberg (2009)
10. Toporkov, V.V.: Job and Application-Level Scheduling in Distributed Computing. Ubiquitous Computing and Communication J. 3(4), 559–570 (2009)

11. Mu'alem, A.W., Feitelson, D.G.: Utilization, Predictability, Workloads, and User Runtime Estimates in Scheduling the IBM SP2 with Backfilling. IEEE Transactions on Parallel and Distributed Systems 6(12), 529–543 (2001)
12. Jackson, D.B., Snell, Q.O., Clement, M.J.: Core Algorithms of the Maui Scheduler. In: Feitelson, D.G., Rudolph, L. (eds.) JSSPP 2001. LNCS, vol. 2221, pp. 87–102. Springer, Heidelberg (2001)
13. Toporkov, V.V., Tselishchev, A.: Safety Scheduling Strategies in Distributed Computing. International Journal of Critical Computer-Based Systems 1/2/3 (1), 41–58 (2010)
14. Toporkov, V.V., Toporkova, A., Tselishchev, A., Yemelyanov, D.: Scalable Co-Scheduling Strategies in Distributed Computing. In: 5th ACS/IEEE Int. Conference on Computer Systems and Applications, pp. 1–8. IEEE CS Press, New York (2010)