Slot Selection Algorithms in Distributed Computing with Non-dedicated and Heterogeneous Resources

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

[1] National Research University "MPEI",
ul. Krasnokazarmennaya, 14, Moscow, 111250, Russia
{ToporkovVV,YemelyanovDM}@mpei.ru
[2] National Research University Higher School of Economics,
Moscow State Institute of Electronics and Mathematics,
Bolshoy Trekhsvyatitelsky per., 1-3/12, Moscow, 109028, Russia
atoporkova@hse.ru
[3] European Organization for Nuclear Research (CERN),
Geneva, 23, 1211, Switzerland
Alexey.Tselishchev@cern.ch

Abstract. In this work, we introduce slot selection and co-allocation algorithms for parallel jobs in distributed computing with non-dedicated and heterogeneous resources (clusters, CPU nodes equipped with multicore processors, networks etc.). A single slot is a time span that can be assigned to a task, which is a part of a job. The job launch requires a co-allocation of a specified number of slots starting synchronously. The challenge is that slots associated with different resources of distributed computational environments may have arbitrary start and finish points that do not match. Some existing algorithms assign a job to the first set of slots matching the resource request without any optimization (the first fit type), while other algorithms are based on an exhaustive search. In this paper, algorithms for effective slot selection of linear complexity on an available slots number are studied and compared with known approaches. The novelty of the proposed approach consists of allocating alternative sets of slots. It provides possibilities to optimize job scheduling.

Keywords: Distributed computing, economic scheduling, resource management, slot, job, task, batch.

1 Introduction

Economic mechanisms are used to solve problems like resource management and scheduling of jobs in a transparent and efficient way in distributed environments such as cloud computing and utility Grid [1, 2]. A resource broker model is decentralized, well-scalable and application-specific [2-4]. It has two parties: node owners and brokers representing users. The simultaneous satisfaction of various application optimization criteria submitted by independent users is not possible due to several reasons [2] and also can deteriorate such quality of service rates as total execution time of a sequence of jobs or overall resource utilization. Another model is related to virtual organizations (VO) [5-7] and metascheduling with central schedulers or a Meta-Broker [2] providing job-flow level scheduling and optimization. VOs naturally restrict the scalability, but uniform rules for allocation and consumption of resources make it possible to improve the efficiency of resource usage and to find a tradeoff between contradictory interests of different participants. In [6], we have proposed a hierarchical model of resource management system which is functioning within a VO. Resource management is implemented using a structure consisting of a metascheduler and subordinate job schedulers that interact with batch job processing systems. The significant difference between the approach proposed in [6] and well-known scheduling solutions for distributed environments such as Grids [1-5, 8, 9], e.g., gLite Workload Management System [8], where Condor is used as a scheduling module, is the fact that the scheduling strategy is formed on a basis of efficiency criteria. They allow reflecting economic principles of resource allocation by using relevant cost functions and solving a load balancing problem for heterogeneous resources. At the same time, the inner structure of the job is taken into account when the resulting schedule is formed. The metascheduler [5-7] implements the economic policy of a VO based on local resource schedules. The schedules are defined as sets of slots coming from resource managers or schedulers in the resource domains. During each scheduling cycle the sets of available slots are updated based on the information from local resource managers. Thus, during every cycle of the job batch scheduling [6] two problems have to be solved: 1) selecting an alternative set of slots (alternatives) that meet the requirements (resource, time, and cost); 2) choosing a slot combination that would be the efficient or optimal in terms of the whole job batch execution in the current cycle of scheduling. To implement this scheduling scheme, first of all, one needs to propose the algorithm for finding sets of alternative executions. An optimization technique for the second phase of this scheduling scheme was proposed in [6, 7].

The scheduling problem in Grid is NP-hard due to its combinatorial nature and many heuristic-based solutions have been proposed. In [4] heuristic algorithms for slot selection, based on user-defined utility functions, are introduced. NWIRE system [4] 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 [10] and NorduGrid [11] approaches) assign any job to the first set of slots matching the resource request conditions, while other algorithms use an exhaustive search [2, 12, 13] and some of them are based on a linear integer programming (IP) [2, 12] or mixed-integer programming (MIP) model [13]. Moab scheduler [14] implements the backfilling algorithm and during a slot window search does not take into account any additive constraints such as the minimum required storage volume or the maximum allowed total allocation cost. Moreover, backfilling does not support environments with non-dedicated resources and its execution time grows substantially with the increase of the slot numbers. Assuming that every CPU node has at least one local job scheduled, the backfilling algorithm has quadratic complexity in terms of the slot number. In our previous works [15-17], two algorithms for slot selection AMP and ALP that feature linear complexity $O(m)$, where m is the number of available time-slots, were proposed. Both algorithms perform the search of the first fitting window without any optimization. AMP (**A**lgorithm based on **M**aximal job **P**rice), performing slot selection based on the maximum slot window cost, proved the advantage over ALP (**A**lgorithm based on **L**ocal **P**rice of slots) when applied to the above mentioned scheduling scheme. However, in order to accommodate an end user's job execution requirements, there is a need for a more precise slot selection algorithm to exploit during the first stage of the proposed scheduling scheme and to consider various user demands along with the VO resource management policy.

In this paper, we propose algorithms for effective slot selection based on user defined criteria that feature linear complexity on the number of the available slots during the job batch scheduling cycle. The novelty of the proposed approaches consists of allocating a number of alternative sets of slots (alternatives). The proposed algorithms can be used for both homogeneous and heterogeneous resources. The paper is organized as follows. Section 2 introduces a general scheme for searching alternative slot sets that are effective by the specified criteria. Then four implementations are proposed and considered. Section 3 contains simulation results for comparison of proposed and known algorithms. Section 4 summarizes the paper and describes further research topics.

2 General Scheme and Slot Selection Algorithms

In this section we consider a general scheme of an **A**lgorithm searching for **E**xtreme **P**erformance (AEP) and its implementation examples.

2.1 AEP Scheme

The launch of any job requires a co-allocation of a specified number of slots, as well as in the classic backfilling variation [14]. The target is to scan a list of m available slots and to select a window W of n parallel slots with a length of the required resource reservation time. The job resource requirements are arranged into a resource request containing a resource reservation time, characteristics of computational nodes (clock speed, RAM volume, disk space, operating system etc.) and the limitation on the selected window maximum cost. The total window cost is calculated as a sum of an individual usage cost of the selected slots. According to the resource request, it is required to find a window with the following description: n concurrent time-slots providing the resource performance rate and the maximal resource price per time unit F should be reserved for a time span t_s. The length of each slot in the window is determined by the performance rate of the node on which it is allocated. Thus, in the case of heterogeneous resources, as a result one has a window with a "rough right edge" (Fig. 1). The window search is performed on the list of all available slots sorted by their start time in ascending order (this condition is necessary to examine every slot in the list and for operation of search algorithms of linear complexity [15-17]). In addition, one can define a criterion crW on which the best matching window alternative is chosen: This can be a criterion for a minimum cost, a minimum execution runtime or, for example, a minimum energy consumption. The algorithm parses a ranged list of all available slots subsequently for all the batch jobs. Higher priority jobs are processed first [6].

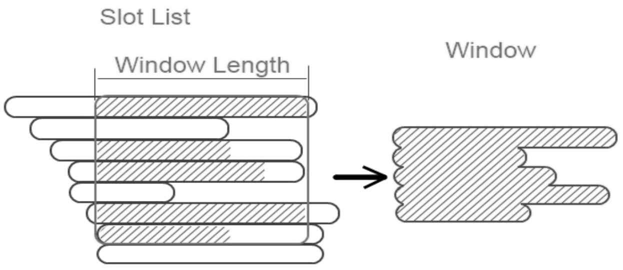


Fig. 1. Window with a "rough right edge"

Consider as an example the problem of selecting a window of size n with a total cost no more than S from the list of $m > n$ slots (in the case, when $m = n$ the selection is trivial). The maximal job budget is counted as $S = Ft_s n$. The current extended window consists of m slots $s_1, s_2, ..., s_m$. The cost of using each of the slots according to their required time length is: $c_1, c_2, ..., c_m$. Each slot has a numeric characteristic z_i in accordance to crW. The total value of these characteristics should be minimized in the resulting window.

Then the problem could be formulated as follows:

$$a_1 z_1 + a_2 z_2 + ... + a_m z_m \to \min, \; a_1 c_1 + a_2 c_2 + ... + a_m c_m \leq S,$$

$$a_1 + a_2 + ... + a_m = n, \; a_r \in \{0, 1\}, \; r = 1, ..., m.$$

Additional restrictions can be added, for example, considering the specified value of deadline. Finding the coefficients $a_1, a_2, ..., a_m$ each of which takes integer values 0 or 1 (and the total number of "1" values is equal to n), determine the window with the specified criteria crW extreme value. By combining the optimization criteria, VO administrators and users can form alternatives search strategies for every job in the batch [6, 7]. Users may be interested in their jobs total execution cost minimizing or, for example, in the earliest possible jobs finish time, and are able to affect the set of alternatives found by specifying the job distribution criteria. VO administrators in their turn are interested in finding extreme alternatives characteristics values (e.g., total cost, total execution time) to form more flexible and, possibly, more effective combination of alternatives representing a batch execution schedule. The time length of an allocated window W is defined by the execution time of the task that is using the slowest CPU node. The algorithm proposed is processing a list of all slots available during the scheduling interval ordered by a non-decreasing start time (see Fig. 1). This condition is required for a single sequential slot list scan and algorithm linear complexity on the number m of slots.

The AEP scheme for an effective window search by the specified criteria can be represented as follows:

```
/* Job - Batch job for which the search is performed ;
** windowSlots - a set (list) of slots representing the
window;*/
slotList = orderSystemSlotsByStartTime();
for(i=0; i< slotList.size; i++){
   nextSlot = slotList[i];
 if(!properHardwareAndSoftware(nextSlot))
  continue; // The slot does not meet the requirements
 windowSlotList.add(nextSlot);
 windowStart = nextSlot.startTime;
 for(j=0; j<windowSlots.size; j++){
  wSlot = windowSlots[j];
  minLength = wSlot.Resource.getTime(Job);
  if((wSlot.EndTime - windowStart) < minLength)
    windowSlots.remove(wSlot);
 }
 if(windowSlots.size >= Job.m){
   curWindow = getBestWindow(windowSlots);
   crW = getCriterion(curWindow);
   if(crW > maxCriterion){
    maxCriterion = crW;
      bestWindow = curWindow;
   }
 }
}
```

Finally, a variable `bestWindow` will contain an effective window by the given criterion `crW`.

2.2 AEP Implementation Examples

The need to choose alternative sets of slots for every batch job increases the complexity of the whole scheduling scheme [6]. With a large number of the available slots the algorithm execution time may become inadequate. Though it is possible to mention some typical optimization problems, based on the AEP scheme that can be solved with a relatively decreased complexity. These include problems of total job cost, runtime minimizing, the window formation with the minimal start/finish time.

Consider the procedure for *minimizing a window start time*. The difference with the general AEP scheme is that the first suitable window will have the earliest possible start time. Indeed, if at some step i of the algorithm (after the i-th slot is added) the suitable window can be formed, then the windows, formed at the further steps will be guaranteed to have the start time that is not earlier (according to the ordered list of available slots, only slots with non-decreasing start time will be taken into consideration). This procedure can be reduced to finding a set of the first n parallel slots the total cost of which does not exceed the budget limit S. This description coincides the AMP scheme considered in previous works [15-17]. Thus AEP is naturally an extension of AMP, and AMP is the particular case of the whole AEP scheme performing only the start time optimization. Further we will use AMP abbreviation as a reference to the window start time minimization procedure. It is easy to provide the implementation of the algorithm of finding a window with *the minimum total execution cost*. For this purpose in the AEP search scheme n slots with the minimum sum cost should be chosen. If at each step of the algorithm a window with the minimum sum cost is selected, at the end the window with the best value of the criterion crW will be guaranteed to have overall minimum total allocation cost at the given scheduling interval.

The problem to find a window with *the minimum runtime* is more complicated. Given the nature of determining a window runtime, which is equal to the length of the longest slot (allocated on the node with the least performance level), the following algorithm may be proposed:

```
orderSlotsByCost(windowSlots);
resultWindow = getSubList(0,n, windowSlots);
extendWindow = getSubList(n+1,m, windowSlots);
while(extendWindow.size > 0){
  longSlot = getLongestSlot(resultWindow);
  shortSlot = getCheapestSlot(extendWindow);
  extendWindow.remove(shortSlot);
  if((shortSlot.size < longSlot.size)&&
   (resultWindow.cost + shortSlot.cost < S)){
      resultWindow.remove(longSlot);
      resultWindow.add(shortSlot);
  }
}
```

As a result, the suitable window of the minimum time length will be formed in a variable `resultWindow.` The algorithm described consists of the consecutive attempts to substitute the longest slot in the forming window (the `resultWindow` variable) with another shorter one that will not be too expensive. In case when it is impossible to substitute the slots without violating the constraint on the maximum window allocation cost, the current `resultWindow` configuration is declared to have the minimum runtime. Implementing this algorithm of window selection at each step of the AEP scheme allows finding a suitable window with the minimum possible runtime at the given scheduling interval. An algorithm for finding a window with *the earliest finish time* has a similar structure and can be described using the runtime minimizing procedure presented above. Indeed, the expanded window has a start time `tStart` equal to the start time of the last added suitable slot. The minimum finish time for a window on this set of slots is `(tStart + minRuntime)`, where `minRuntime` is the minimum window length. The value of `minRuntime` can be calculated similar to the runtime minimizing procedure described above. Thus, by selecting a window with the earliest completion time at each step of the algorithm, the required window will be allocated at the end of the slot list.

It is worth mentioning that all proposed AEP implementations have a linear complexity $O(m)$: algorithms "move" through the list of the m available slots in the direction of non-decreasing start time without turning back or reviewing previous steps.

3 Experimental Studies of Slot Selection Algorithms

The goal of the experiment is to examine AEP implementations: to analyze alternatives search results with different efficiency criteria, to compare the results with AMP and to estimate the possibility of using in real systems considering the algorithm executions time.

3.1 Algorithms and Simulation Environment

For the proposed AEP efficiency analysis the following implementations were added to the simulation model [6, 7]: 1) *AMP* – the algorithm for searching alternatives with the earliest start time. This scheme was introduced in works [15-17] and briefly described in section 2.2; 2) *MinFinish* – the algorithm for searching alternatives with the earliest finish time. It likewise involves finding a single alternative with the earliest finish time for each batch job (the procedure is described in section 2.2); 3) *MinCost* – the algorithm for searching a single alternative with the minimum total allocation cost on the scheduling interval; 4) *MinRunTime* – this algorithm performs a search for a single alternative with the minimum execution runtime (the window's runtime is defined as a length of the longest composing slot); 5) *MinProcTime* – this algorithm performs a search for a single alternative with the minimum total node execution time (defined as a sum of the composing slots' time lengths). It is worth mentioning that this implementation is simplified and does not guarantee an optimal result and only partially matches the AEP scheme, because a random window is selected; 6) *Common Stats, AMP* (further referred to as *CSA*) – the scheme for searching multiple alternatives using *AMP*. Similar to the general searching scheme [15-17], a set of suitable alternatives, disjointed by the slots, is allocated for each job. To compare the search results with the algorithms 1-5, presented above, only alternatives with the extreme value of the given criterion will be selected, so the optimization will take place at the selection process. The criteria include the minimum start time, the minimum finish time, the minimum total execution cost, the minimum runtime and the minimum processor time used.

Since the purpose of the considered algorithms is to allocate suitable alternatives, it makes sense to make the simulation apart from the whole general scheduling scheme, described in [6]. In this case, the search will be performed for a single predefined job. Thus during every single experiment a generation of a new distributed computing environment will take place while the algorithms described will perform the alternatives search for a single base job with the resource request defined in advance. A simulation framework [6, 7] was configured in a special way in order to study and to analyze the algorithms presented. The core of the system includes several components that allow generating the initial state of the distributed environment on the given scheduling interval, a batch with user jobs and implements the developed alternative search algorithms.

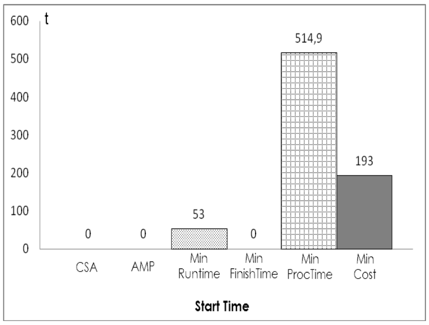
In each experiment a generation of the distributed environment that consists of 100 CPU nodes was performed. The relatively high number of the generated nodes has been chosen to allow *CSA* to find more slot alternatives. Therefore more effective alternatives could be selected for the searching results comparison based on the given criteria. The performance rate for each node was generated as a random integer variable in the interval [2; 10] with a uniform distribution. The resource usage cost was formed proportionally to their performance with an element of normally distributed deviation in order to simulate a free market pricing model [1-4]. The level of the resource initial load with the local and high priority jobs at the scheduling interval [0; 600] was generated by the hyper-geometric distribution in the range from 10% to 50% for each CPU node. Based on the generated environment the algorithms performed the search for a single initial job that required an allocation of 5 parallel slots for 150 units of time. The maximum total execution cost according to user requirements was set to 1500. This value generally will not allow using the most expensive (and usually the most efficient) CPU nodes.

3.2 Experimental Results

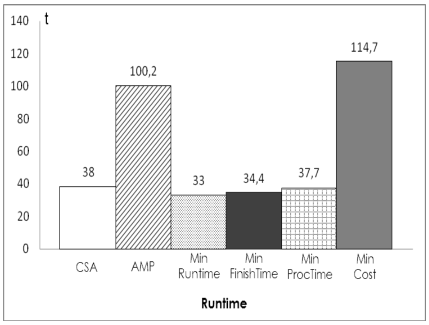
The results of the 5000 simulated scheduling cycles are presented in Fig. 2-6.

An average number of the alternatives found with *CSA* for a single job during the one scheduling cycle was 57. This value can be explained as the balance between the initial resource availability level and the user job resource requirements. Thus, the selection of the most effective windows by the given criteria was carried out among 57 suitable alternatives on the average. With the help of other algorithms only the single effective by the criterion alternative was found. Consider the average *start time* for the alternatives found (and selected) by the aforementioned algorithms (Fig. 2 (a)). *AMP*, *MinFinish* and *CSA* were able to provide the earliest job start time at the beginning of the scheduling interval ($t = 0$). The result was expected for *AMP* and *CSA* (which is essentially based on the multiple runs of the *AMP* procedure) since 100 available resource nodes provide a high probability that there will be at least 5 parallel slots starting at the beginning of the interval and can form a suitable window. The fact that the *MinFinish* algorithm was able to provide the same start time can be explained by the local tasks minimum length value, that is equal to 10. Indeed, the window start time at the moment $t = 10$ cannot provide the earliest finish time even with use of the most productive resources (for example the same resources allocated for the window with the minimal runtime). Average starting times of the alternatives found by *MinRunTime*, *MinProcTime* and *MinCost* are 53, 514.9 and 193 respectively.

The *average runtime* of the alternatives found (selected) is presented in Fig. 2 (b). The minimum execution runtime 33 was obviously provided by the *MinRunTime* algorithm. Though, schemes *MinFinish*, *MinProcTime* and *CSA* provide quite comparable values: 34.4, 37.7 and 38 time units respectively that only 4.2%, 14.2% and 15.1% longer. High result for the *MinFinish* algorithm can be explained by the "need" to complete the job as soon as possible with the minimum (and usually initial) start time. Algorithms *MinFinish* and *MinRunTime* are based on the same criterion selection procedure described in the section 2.2. However due to non-guaranteed availability of the most productive resources at the beginning of the scheduling interval, *MinRunTime* has the advantage. Relatively long runtime was provided by *AMP* and *MinCost* algorithms. For *AMP* this is explained by the selection of the first fitting (and not always effective by the given criterion) alternative, while *MinCost* tries to use relatively cheap and (usually) less productive CPU nodes.



(a)

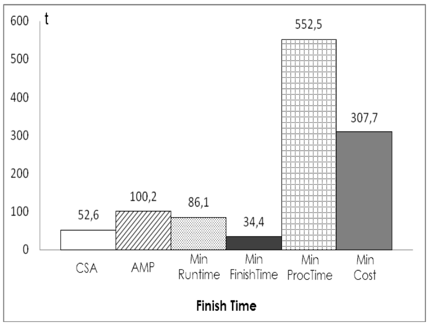


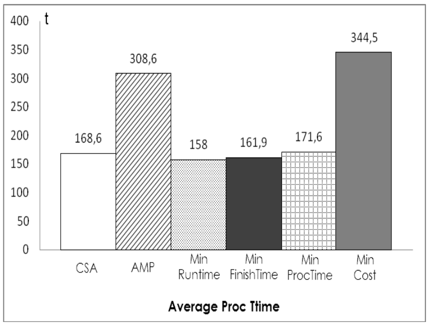
(b)

Fig. 2. Average start time (a) and runtime (b)

The minimum *average finish time* (Fig. 3 (a)) was provided by the *MinFinish* algorithm – 34.4. *CSA* has the closest resulting finish time of 52.6 that is 52.9% later. The relative closeness of these values comes from the fact that other related algorithms did not intend to minimize a finish time value and were selecting windows without taking it into account. At the same time *CSA* is picking the most effective alternative among 57 (on the average) allocated at the scheduling cycle: the optimization was carried out at the selection phase. The late average finish time 307.7 is provided by the *MinCost* scheme. This value can be explained not only with a relatively late average start time (see Fig. 2 (a)), but also with a longer (compared to other approaches) execution runtime (see Fig. 2 (b)) due to the use of less productive resource nodes. The finish time obtained by the simplified *MinProcTime* algorithm was relatively high due to the fact that a random window was selected (without any optimization) at each step of the algorithm, though the search was performed on the whole list of available slots. With such a random selection the most effective window by the processor time criterion was near the end of the scheduling interval.

The *average used processor time* (the sum time length of the used slots) for the algorithms considered is presented in Fig. 3 (b). The minimum value provided by *MinRunTime* is 158 time units. *MinFinish*, *CSA* and *MinProcTime* were able to provide comparable results: 161.9, 168.6 and 171.6 respectively. It is worth mentioning that although the simplified *MinProcTime* scheme does not provide the best value, it is only 2% less effective compared to the common *CSA* scheme, while its working time is orders of magnitude less (Tables 1, 2). The most processor time consuming alternatives were obtained by *AMP* and *MinCost* algorithms. Similarly to the execution runtime value, this can be explained by using a random first fitting window (in case of *AMP*) or by using less expensive, and hence less productive, resource nodes (in case of the *MinCost* algorithm), as nodes with a low performance level require more time to execute the job.





(a) (b)

Fig. 3. Average finish time (a) and CPU usage time (b)

Finally, let us consider the *total job execution cost* (Fig. 4).

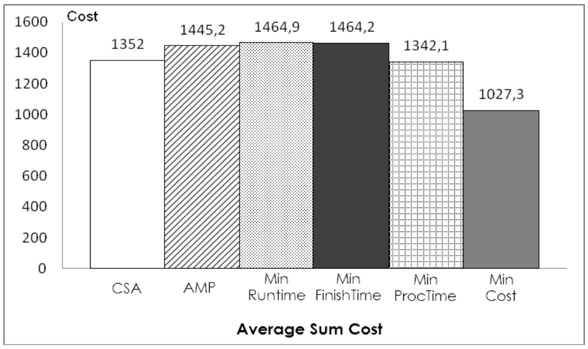


Fig. 4. Average job execution cost

The *MinCost* algorithm has a big advantage over other algorithms presented: it was able to provide the total cost of 1027.3 (note that the total cost limit was set by the user at 1500). Alternatives found with other considered algorithms have approximately the same execution cost. Thus, the cheapest alternatives found by *CSA* have the average total execution cost equal to 1352, that is 31.6% more expensive compared to the result of the *MinCost* scheme, while alternatives found by *MinRunTime* (the most expensive ones) are 42.5% more expensive.

The important factor is a complexity and an actual working time of the algorithms under consideration, especially with the assumption of the algorithm's repeated use during the first stage of the scheduling scheme [6]. In the description of the AEP general scheme it was mentioned that the algorithm has a linear complexity on the number of the available slots. However in has a quadratic complexity with a respect to the number of CPU nodes. Table 1 shows the actual algorithm execution time in milliseconds measured depending on the number of CPU nodes.

The simulation was performed on a regular PC workstation with Intel Core i3 (2 cores @ 2.93 GHz), 3GB RAM on JRE 1.6, and 1000 separate experiments were simulated for each value of the processor nodes numbers {50, 100, 200, 300, 400}. The simulation parameters and assumptions were the same as described in section 3.1, apart from the number of used CPU nodes. A row "*CSA: Alternatives Num*" represents an average number of alternatives found by *CSA* during the single experiment simulation (note that *CSA* is based on multiple runs of *AMP* algorithm). A row "*CSA per Alt*" represents an average working time for the *CSA* algorithm in recalculation for one alternative. The *CSA* scheme has the longest working time that on the average almost reaches 3 seconds when 400 nodes are available.

Table 1. Actual algorithms execution time in ms

CPU nodes number:	50	100	200	300	400
CSA:Alternatives Num	25.9	57	128.4	187.3	252
CSA per Alt	0.33	0.99	3.16	6.79	11.83
CSA	8.5	56.5	405.2	1271	2980.9
AMP	0.3	0.5	1.1	1.6	2.2
MinRunTime	3.2	12	45.5	97.2	169.2
MinFinishTime	3.2	12	45.1	96.9	169
MinProcTime	1.5	5.2	19.4	42.1	74.1
MinCost	1.7	6.3	23.6	52.3	91.5

Besides this time has a near cubic increasing trend with a respect to the nodes number. This trend can be explained by the addition of two following factors: 1) a linear increase of the alternatives number found by *CSA* at each experiment (which makes sense: the linear increase of the available nodes number leads to the proportional increase in the available node processor time; this makes it possible to find (proportionally) more alternatives); 2) a near quadratic complexity of the *AMP* algorithm with a respect to the nodes number, which is used to find single alternatives in *CSA*. Even more complication is added by the need of "cutting" a suitable windows from the list of the available slots [17]. Other considered algorithms will be able to perform a much faster search. The average working time of *MinRunTime*, *MinProcTime* and *MinCost* proves their (at most) quadratic complexity on the number of CPU nodes, and the executions times are suitable for practical use. The *AMP*'s execution time shows even near linear complexity because with a relatively large number of free available resources it was usually able to find a window at the beginning of the scheduling interval (see Fig. 1, a) without the full slot list scan.

Fig. 5 clearly presents the average working duration of the algorithms depending on the number of available CPU nodes (the values were taken from Table 1). (The *CSA* curve is not represented as its working time is incomparably longer than AEP-like algorithms.)

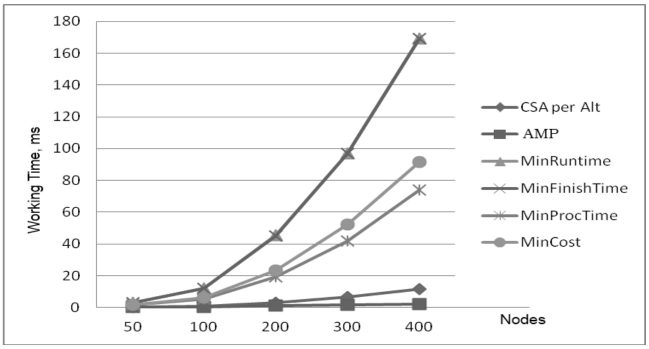


Fig. 5. Average working time duration depending on the available CPU nodes number

Table 2 contains the algorithms working time in milliseconds measured depending on the scheduling interval length.

Table 2. Algorithms working time (in ms) measured depending on the scheduling interval length

Scheduling interval length:	600	1200	1800	2400	3000	3600
Number of slots:	472.6	779.4	1092	1405.1	1718.8	2030.6
CSA:Alternatives Num	57	125.4	196.2	269.8	339.7	412.5
CSA per Alt	0.95	1.91	2.88	3.88	4.87	5.88
CSA	54.2	239.8	565.7	1045.7	1650.5	2424.4
AMP	0.5	0.82	1.1	1.44	1.79	2.14
MinRunTime	11.7	26	40.9	55.5	69.4	84.6
MinFinishTime	11.6	25.7	40.6	55.3	69	84.1
MinProcTime	5	11.1	17.4	23.5	29.5	35.8
MinCost	6.1	13.4	20.9	28.5	35.7	43.5

Overall 1000 single experiments were conducted for each value of the interval length {600, 1200, 1800, 2400, 3000, 3600} and for each considered algorithm an average working time was obtained. The experiment simulation parameters and assumptions were the same as described earlier in this section, apart from the scheduling interval length. A number of CPU nodes was set to 100. Similarly to the previous experiment, *CSA* had the longest working time (about 2.5 seconds with the scheduling interval length equal to 3600 model time units), which is mainly caused by the relatively large number of the formed execution alternatives (on the average more than 400 alternatives on the 3600 interval length). When analyzing the presented values it is easy to see that all proposed algorithms have a linear complexity with the respect to the length of the scheduling interval and, hence, to the number of the available slots (Fig. 6), and their executions times are suitable for on-line scheduling.

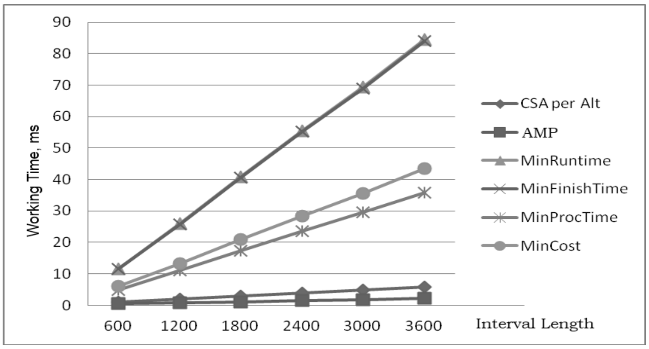


Fig. 6. Average working time duration depending on the scheduling interval length

3.3 Brief Analysis and Related Work

In this work, we propose algorithms for efficient slot selection based on user and administrators defined criteria that feature the *linear* complexity on the number of all available time-slots during the scheduling interval. Besides, in our approach the *job start time* and the *finish time* for slot search algorithms may be considered as criteria specified by users in accordance with the job total allocation cost. It makes an opportunity to perform more flexible scheduling solutions.

The co-allocation algorithm presented in [12] uses the 0-1 IP model with the goal of creating reservation plans satisfying user resource requirements. The number of variables in the proposed algorithm becomes N^3 depending on the number of computer sites N. Thus this approach may be inadequate for an on-line service in practical use. The proposed algorithms have the *quadratic complexity* with a respect to the number of CPU nodes, and not to the number of computer sites as in [12]. A linear IP-driven algorithm is proposed in [2]. It combines the capabilities of IP and genetic algorithm and allows obtaining the best metaschedule that minimises the combined cost of all independent users in a coordinated manner. In [13], the authors propose a MIP model which determines the best scheduling for all the jobs in the queue in environments composed of multiple clusters that act collaboratively. The proposed in [2, 12, 13] scheduling techniques are effective compared with other scheduling techniques under given criteria: the minimum processing cost, the overall makespan, resources utilization etc. However, complexity of the scheduling process is extremely increased by the resources heterogeneity and the co-allocation process, which distributes the tasks of parallel jobs across resource domain boundaries. The degree of complexity may be an obstacle for on-line use in large-scale distributed environments.

As a result it may be stated that each full AEP-based scheme was able to obtain the best result in accordance with the given criterion. This allows to use the proposed algorithms within the whole scheduling scheme [6] at the first stage of the batch job alternatives search. Moreover, each full AEP-based scheme was able to obtain the best result in accordance with the given criterion. Besides, a single run of the AEP-like algorithm had an advantage of 10%-50% over suitable alternatives found with

AMP with a respect to the specified criterion. A directed alternative search at the first stage of the proposed scheduling approach [6, 7] can affect the final distribution and may be favorable for the end users. According to the experimental results, on one hand, the best scheme with top results in start time, finish time, runtime and CPU usage minimization was *MinFinish*. Though in order to obtain such results the algorithm spent almost all user specified budget (1464 of 1500). On the other hand, the *MinCost* scheme was designed precisely to minimize execution expenses and provides 43% advantage over *MinFinish* (1027 of 1500), but the drawback is a more than modest results by other criteria considered. The *MinProcTime* scheme stands apart and represents a class of simplified AEP implementations with a noticeably reduced working time. And though the scheme, compared to other considered algorithms, did not provide any remarkable results, it was on the average only 2% less effective than the *CSA* scheme by the dedicated CPU usage criterion. At the same time its reduced complexity and actual working time allow to use it in a large wide scale distributed environments when other optimization search algorithms prove to be too slow.

4 Conclusions and Future Work

In this work, we address the problem of slot selection and co-allocation for parallel jobs in distributed computing with non-dedicated resources. Each of the AEP algorithms possesses a linear complexity on a total available slots number and a quadratic complexity on a CPU nodes number. The advantage of AEP-based algorithms over the general CSA scheme was shown for each of the considered criteria: start time, finish time, runtime, CPU usage time and total cost.

In our further work, we will refine resource co-allocation algorithms in order to integrate them with scalable co-scheduling strategies [6, 7].

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-316.2012.9), the Russian Foundation for Basic Research (grant no. 12-07-00042), and by the Federal Target Program “Research and scientific-pedagogical cadres of innovative Russia” (state contract no. 16.740.11.0516).

References

1. Lee, Y.C., Wang, C., Zomaya, A.Y., Zhou, B.B.: Profit-driven Scheduling for Cloud Services with Data Access Awareness. J. Parallel and Distributed Computing 72(4), 591–602 (2012)
2. Garg, S.K., Konugurthi, P., Buyya, R.: A Linear Programming-driven Genetic Algorithm for Meta-scheduling on Utility Grids. J. Parallel, Emergent and Distributed Systems 26, 493–517 (2011)

3. Buyya, R., Abramson, D., Giddy, J.: Economic Models for Resource Management and Scheduling in Grid Computing. J. Concurrency and Computation: Practice and Experience 5(14), 1507–1542 (2002)
4. 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)
5. 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 Acad. Publishers (2003)
6. Toporkov, V., Tselishchev, A., Yemelyanov, D., Bobchenkov, A.: Composite Scheduling Strategies in Distributed Computing with Non-dedicated Resources. Procedia Computer Science 9, 176–185 (2012)
7. Toporkov, V., Tselishchev, A., Yemelyanov, D., Bobchenkov, A.: Dependable Strategies for Job-flows Dispatching and Scheduling in Virtual Organizations of Distributed Computing Environments. In: Zamojski, W., Mazurkiewicz, J., Sugier, J., Walkowiak, T., Kacprzyk, J. (eds.) Complex Systems and Dependability. AISC, vol. 170, pp. 289–304. Springer, Heidelberg (2012)
8. Cecchi, M., Capannini, F., Dorigo, A., et al.: The gLite Workload Management System. J. Phys.: Conf. Ser. 219 (6), 062039 (2010)
9. Yu, J., Buyya, R., Ramamohanarao, K.: Workflow Scheduling Algorithms for Grid Computing. In: Xhafa, F., Abraham, A. (eds.) Metaheuristics for Scheduling in Distributed Computing Environments. SCI, vol. 146, pp. 173–214. Springer, Heidelberg (2008)
10. Aida, K., Casanova, H.: Scheduling Mixed-parallel Applications with Advance Reservations. In: 17th IEEE Int. Symposium on HPDC, pp. 65–74. IEEE CS Press, New York (2008)
11. Elmroth, E., Tordsson, J.: A Standards-based Grid Resource Brokering Service Supporting Advance Reservations, Coallocation and Cross-Grid Interoperability. J. Concurrency and Computation: Practice and Experience 25(18), 2298–2335 (2009)
12. 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)
13. Blanco, H., Guirado, F., Lérida, J.L., Albornoz, V.M.: MIP Model Scheduling for Multi-clusters. In: Caragiannis, I., et al. (eds.) Euro-Par Workshops 2012. LNCS, vol. 7640, pp. 196–206. Springer, Heidelberg (2013)
14. Moab Adaptive Computing Suite, `http://www.adaptivecomputing.com`
15. Toporkov, V., Toporkova, A., Bobchenkov, A., Yemelyanov, D.: Resource Selection Algorithms for Economic Scheduling in Distributed Systems. Procedia Computer Science 4, 2267–2276 (2011)
16. Toporkov, V., Yemelyanov, D., Toporkova, A., Bobchenkov, A.: Resource Co-allocation Algorithms for Job Batch Scheduling in Dependable Distributed Computing. In: Zamojski, W., Kacprzyk, J., Mazurkiewicz, J., Sugier, J., Walkowiak, T. (eds.) Dependable Computer Systems. AICS, vol. 97, pp. 243–256. Springer, Heidelberg (2011)
17. Toporkov, V., Bobchenkov, A., Toporkova, A., Tselishchev, A., Yemelyanov, D.: Slot Selection and Co-allocation for Economic Scheduling in Distributed Computing. In: Malyshkin, V. (ed.) PaCT 2011. LNCS, vol. 6873, pp. 368–383. Springer, Heidelberg (2011)