ISSN 0361-7688, Programming and Computer Software, 2016, Vol. 42, No. 1, pp. 17–26. © Pleiades Publishing, Ltd., 2016.
Original Russian Text © L.B. Sokolinsky, A.V. Shamakina, 2016, published in Programmirovanie, 2016, Vol. 42, No. 1.

Methods of Resource Management in Problem-Oriented Computing Environment

L. B. Sokolinsky and A. V. Shamakina

South Ural State University, pr. im. V.I. Lenina 76, Chelyabinsk, 454080 Russia
e-mail: Leonid.Sokolinsky@susu.ru, shamakinaav@susu.ru

Received April 22, 2015

Abstract—One of the important classes of computational problems is problem-oriented workflow applications executed in distributed computing environment. A problem-oriented workflow application can be represented by a directed graph whose vertices are tasks and arcs are data flows. For a problem-oriented workflow application, we can get a priori estimates of the task execution time and the amount of data to be transferred between the tasks. A distributed computing environment designed for the execution of such tasks in a certain subject domain is called problem-oriented environment. To efficiently use resources of the distributed computing environment, special scheduling algorithms are applied. Nowadays, a great number of such algorithms have been proposed. Some of them (like the DSC algorithm) take into account specific features of problem-oriented workflow applications. Others (like Min–Min algorithm) take into account many-core structure of nodes of the computational network. However, none of them takes into account both factors. In this paper, a mathematical model of problem-oriented computing environment is constructed, and a new problem-oriented scheduling (POS) algorithm is proposed. The POS algorithm takes into account both specifics of the problem-oriented jobs and multi-core structure of the computing system nodes. Results of computational experiments comparing the POS algorithm with other known scheduling algorithms are presented.

DOI: 10.1134/S0361768816010084

1. INTRODUCTION

Development of distributed computation technologies in the late 1990s made it possible to combine heterogeneous resources distributed over the world. It became possible to solve large-scale scientific, engineering, and commercial problems using geographically distributed resources belonging to different owners. Studies in this field resulted in the emergence of the concepts of grid computing [1–4] and, later, cloud computing [5, 6]. To take advantage of potential capabilities of using distributed computing resources, efficient scheduling algorithms to manage the resources are needed.

The basic task of a distributed computation technology is to ensure access to globally distributed resources by means of special tools. The difficulty of the resource management is associated with the fact that different computers may be used to launch and execute the task and to access the data. Global distributed computer networks are formed from autonomous resources the configuration of which varies dynamically. Moreover, distributed resources may belong to different administrative domains, which requires coordination of various administration policies. Another important problem is heterogeneity of the resources. Earlier works on resource management in distributed computing environments [7–10] focusing on resource heterogeneity resulted in creation of standard resource management protocols and mechanisms of description of requirements on resource specification. However, practice showed that effective scheduling methods and algorithms for homogeneous isolated multiprocessor systems are badly adapted to distributed heterogeneous systems [11]. Resource management in heterogeneous distributed computing environments requires new models of computation and resource management. Currently, a promising direction of research is associated with the use of distributed computation technologies for solving resource-intensive scientific problems in various subject domains, such as medicine, engineering design, nanotechnologies, climate forecasting, and the like. Computational problems in such subject domains, in many cases, have flow structure and can be described by means of workflow models [12], in the framework of which a job is represented as a directed acyclic graph whose vertices are tasks and arcs are data flows transmitted from one task to another. Note that the set of tasks the jobs consist of is finite and predefined. Problem-oriented specificity of workflows in such complex applications consists in that, in the majority of cases, certain characteristics (such as task execution time on one processor core, scalability limits, and the amount of generated data) of the tasks can be estimated before running the job. The use of such information in a par-

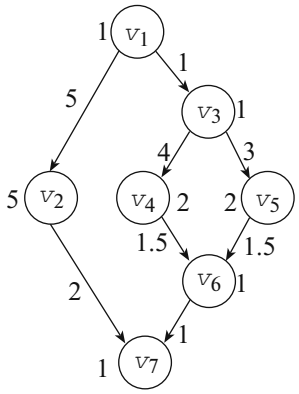

Fig. 1. Directed acyclic graph.

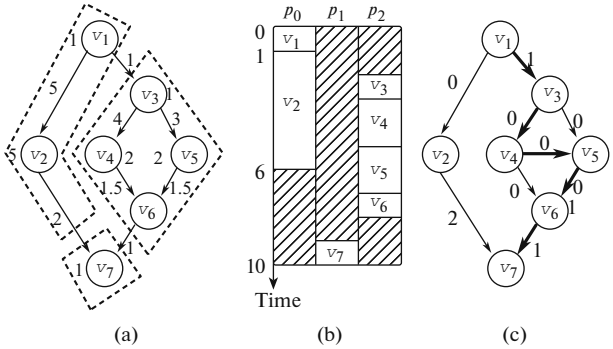

Fig. 2. (a) Clustered graph and its critical path; (b) Gantt chart; (c) scheduled graph and its dominant sequence.

ticular problem-oriented domain can essentially improve efficiency of the resource management methods. Two basic classes of algorithms designed for scheduling applications with workflow structure are known [13]. These are clustering algorithms and list algorithms. Examples of the clustering algorithms are, for instance, the Kim–Brown algorithm [14] and the DSC algorithm [15]. Algorithms from this class use information about the problem-oriented specificity of tasks composing the computational job; however, they permit execution of a task on a single processor core of the multiprocessor system. One of the most popular list algorithms is the Min-min algorithm [16]. The basic disadvantage of the list algorithms is that they do not analyze the entire graph of jobs. Then, it follows that there is a need in the development of the resource management methods and algorithms for problem-oriented distributed computing environments that take into account specificity of subject domains and scalability of individual tasks in a job and make use the possibility of execution of one task on several processor cores.

The paper is organized as follows. Section 2 surveys known resource scheduling algorithms in problem-oriented computing environments. In Section 3, a mathematical model of problem-oriented distributed computing environment is constructed. Section 4 describes a new problem-oriented scheduling (POS) algorithm. Results of computational experiments on comparison of the POS algorithm with other known algorithms are presented in Section 5.

2. SCHEDULING IN PROBLEM-ORIENTED ENVIRONMENTS

In this section, several known scheduling algorithms for job execution in problem-oriented computing environments are surveyed. A computational job in such environments is represented as a directed acyclic graph, an example of which is shown in Fig. 1. The vertices of the graph are interrelated computational tasks, and the arcs represent data flows between the individual tasks. The set of classes of the tasks from which jobs are constructed is finite and predefined. For each task, the time of task execution on one processor core is specified. The amount of data transmitted is given by the arc weight.

One of the most well-known scheduling algorithms for problem-oriented computing environments is the dominant sequence clustering (DSC) algorithm [17]. The basic idea of the DSC algorithm consists in partitioning the set of graph vertices into nonintersecting subsets (clusters). An example of partitioning a job graph into three clusters, which are shown by the dashed lines, is presented in Fig. 2a. Tasks from one cluster are executed sequentially on one processor core in a certain order determined by the algorithm. For the clustered graph, job execution schedule is constructed, which specifies for each task the number of the processor core on which it is executed and the time moment to start it. An example of such a schedule is depicted in Fig. 2b as a Gantt chart. Here, tasks v_1 and v_2 are executed on the processor core p_0; v_7, on p_1; and v_3–v_6, on p_2.

In the clustered graph, communication weights of arcs connecting vertices of one cluster are set equal to zero. As a result, we obtain a scheduled graph. An example of such a graph is shown in Fig. 2c. The DSC algorithm takes into account the order the tasks in the clusters are executed by creating additional pseudoarcs in the scheduled graph. In Fig. 2c, such an additional pseudoarc is the arc $e = (v_4, v_5)$. For the scheduled graph, the DSC algorithm constructs the dominant sequence, which, in essence, is the critical path with regard to the pseudoarcs. In Fig. 2c, the dominant sequence is depicted by the bold arrows.

At the initial moment of the DSC algorithm operation, each vertex is placed into a separate cluster, and all arcs of the job graph are marked as "unconsidered." After an arc has been considered on whether its weight can be set equal to zero, the arc is denoted as "considered," and the vertex from which the arc originates is marked as "scheduled." The scheduled and unsched-

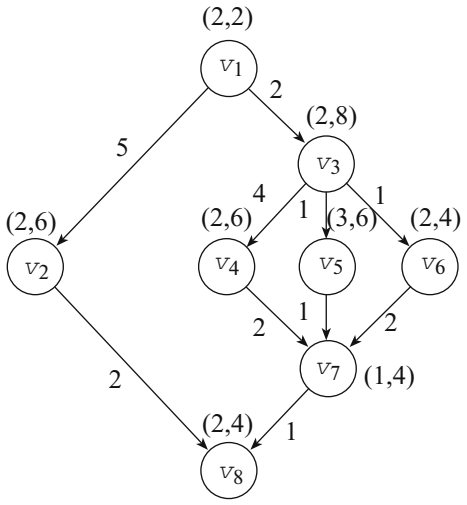

Fig. 3. Job graph.

uled vertices form sets SN and USN, respectively. A vertex is said to be "free" if all vertices preceding it are scheduled. On the first step of the DSC algorithm, out of the arcs belonging to the dominant sequence of the job graph, the first unconsidered arc is selected. On the second step, the arc is set to be zero, and its vertices are combined in one cluster if the parallel time does not increase. The order the tasks in a cluster are executed is determined by the greatest value $bot\_level(n_x, i)$, which is calculated as the sum of all communication costs of the arcs and computational costs of the vertices between the vertex n_x and the lower vertex of the graph in the dominant sequence. The DSC algorithm terminates when all arcs have been considered. The basic disadvantage of the DSC algorithm is associated with its limited applicability to computing systems with many-core processors, since the DSC algorithm schedules execution of each task on only one processor core.

Another known scheduling algorithm for problem-oriented environments is that of linear clustering by Kim and Brown [14], which is also known as the KB/L algorithm. The KB/L algorithm is designed for the job graph clustering under the assumption of an unbounded number of computing nodes. Initially, all arcs of the graph are marked as "unconsidered." On the first step, the KB/L algorithm seeks the critical path of the job graph, which includes only "unconsidered" arcs, by means of the weight cost function (1). All vertices of the critical path found are combined into one cluster, with the communication costs of the arcs being set zero. On the second step, the arcs that are incidental to the vertices of the critical path are marked as "considered." These steps are repeated until all arcs are considered. In [14], Kim uses the cost function

$$Cost\,function = w_1 * \sum \tau_i + (1 - w_1) * \\ * \left(w_2 * \sum c_{ij} + (1 - w_2) = \sum c_{ij}^{adj}\right) \quad (1)$$

to determine length of the critical path of the job graph. In (1), w_1 and w_2 are normalizing multipliers, $\sum \tau_i$ is the sum of computational costs of all vertices of the critical path, and c_{ij}^{adj} is the communication cost of the arcs between a vertex of the critical path all incidental vertices not belonging to the critical path. The objective function of the KB/L algorithm to be minimized is the graph parallel time. The KB/L algorithm does not take into account the fact that the computing nodes are many-core ones.

One more known clustering algorithm is Sarkar's algorithm [18], which can be described as follows. All arcs of the job graph are sorted out in a descending order of their communication costs. The arcs are set equal to zero starting from the arc with the greatest communication cost. An arc communication cost is set zero only if the parallel time does not increase on the next step. The Sarkar's algorithm terminates when all arcs of the job graph are considered. The objective function of the algorithm to be minimized is also the graph parallel time. Note that this algorithm also does not take into account the fact that the computing nodes are many-core ones.

A popular list algorithm is Min–Min [16], which iteratively performs the following operations. On every step, for each task, the early completion time (ECT) for all available resources and the minimum estimated completion time (MCT) are calculated. The task with the least MCT metrics gets all resources required for its completion first. The scheduling process terminates when all tasks are scheduled. This algorithm is usually used for scheduling jobs that consist of many independent tasks with large modules and intensive calculations. The basic disadvantage of the Min–Min algorithm is that it does not analyze the entire job graph.

3. COMPUTING ENVIRONMENT MODEL

In this section, we construct a mathematical model of a problem-oriented distributed computing environment.

A *job graph* is a labeled weighted directed acyclic graph $G = \langle V, \boldsymbol{E}, init, fin, \delta, \gamma \rangle$, where V is a set of vertices corresponding to the tasks, E is a set of arcs corresponding to the data flows, $init : E \to V$ is a function defining the *initial vertex* of an arc, and $fin : E \to V$ is a function defining the *final vertex* of the arc. Weight $\delta(e)$ of the arc e determines the amount of data to be transmitted through the arc e from the task associated with the vertex $init(e)$ to the task associated with the vertex $fin(e)$. The label

$$\gamma(v) = (m_v, t_v) \quad (2)$$

determines the maximum number of processor cores m_ν on which task ν has speed-up close to *linear* and time t_ν of execution of task ν on one core. This model assumes that *computational cost* $\chi(\nu, j_\nu)$ of task ν on j_ν processor cores is determined by the formula

$$\chi(\nu, j_\nu) = \begin{cases} t_\nu / j_\nu, & \text{if} \quad 1 \le j_\nu \le m_\nu; \\ t_\nu / m_\nu, & \text{if} \quad m_\nu \le j_\nu; \end{cases} \tag{3}$$

In other words, if the number of processor cores varies in the range from 1 through m_ν, the decrease in the computation time is directly proportional to the increase in the number of cores; further increase of the number of cores from m_ν to $+\infty$ does not result in any speed-up.

Figure 3 shows an example of a job graph containing eight vertices. Each vertex is marked by a label of the form (m_ν, t_ν). Each arc of the graph has a weight $\delta(e)$, which is the amount of data transmitted through the arc.

A *computing node* P is an ordered set of processor cores $\{c_0, ..., c_{d-1}\}$.

A *computing system* is an ordered set of computing nodes $\mathfrak{P} = \{P_0, ..., P_{k-1}\}$. In practice, it may be a distributed computing system including several computing clusters, each of which is a separate node of this system.

Clustering is a one-to-one mapping $\omega : V \to \mathfrak{P}$ of the set of vertices V of a job graph G onto the set of computing nodes $\mathfrak{P}$.

Let a computing system $\mathfrak{P} = \{P_0, ..., P_{k-1}\}$ consisting of k nodes be given. A *cluster* W_i is a subset of all vertices mapped onto computing node $P_i \in \mathfrak{P}$:

$$W_i = \{\nu \in V | \omega(\nu) = P_i \in \mathfrak{P}\}, \tag{4}$$

where

$$W_i \cap W_j = \emptyset \quad \text{for} \quad i \ne j, \tag{5}$$

$$V = \bigcup_{i=0}^{k-1} W_i. \tag{6}$$

Let a job graph $G = \langle V, \boldsymbol{E}, init, fin, \delta, \gamma \rangle$ be given for which a clustering function $\omega(\nu)$ is defined. Then, the graph is said to be *clustered* and is denoted as $G = \langle V, \boldsymbol{E}, init, fin, \delta, \gamma, \omega \rangle$.

In the framework of the model, we assume that the transmission time of any amount of data between vertices belonging to one cluster is equal to zero and that between vertices belonging to different clusters is proportional to the amount of data transmitted with the coefficient 1. Accordingly, we may define *communication cost* (time) $\sigma : E \to \mathbb{Z}_{\geqslant 0}$ of data transmission through an arc $e \in E$ as

$$\sigma(e) = \begin{cases} 0, & \text{for} \quad \omega(init(e)) = \omega(fin(e)); \\ \delta(e), & \text{otherwise}. \end{cases} \tag{7}$$

Let a clustered graph $G = \langle V, \boldsymbol{E}, init, fin, \delta, \gamma, \omega \rangle$ be given. A *schedule* of G is a mapping $\xi : V \to \mathbb{Z}_{\geqslant 0} \times \mathbb{N}$ that, to an arbitrary vertex $v \in V$, assigns a pair

$$\xi(\nu) = (\tau_\nu, j_\nu), \tag{8}$$

where t_ν determines the time to run task ν and j_ν is the number of processor cores allocated for the task ν associated with this vertex. Let s_ν denote the termination time of task ν. We have

$$s_v = \tau_v + \chi(v, j_v), \tag{9}$$

where χ is the time complexity function defined in (3). A schedule is said to be *correct* if it satisfies the conditions

$$\forall e \in E (\tau_{fin(e)} \ge \tau_{init(e)} + \chi(init(e), j_{init(e)}) + \sigma(e)); \tag{10}$$

$$\forall \nu \in V \quad (j_\nu \le m_\nu); \tag{11}$$

$$\forall t \in N \quad \left(\forall i \in [0, \ldots k-1] \left(\sum_{\nu \in W \,\&\, \tau_\nu < t \le s_\nu} j_\nu \le |P_i| \right) \right). \tag{12}$$

Condition (10) means that, for any two adjacent vertices $\nu_1 = init(e)$ and $\nu_2 = fin(e)$, the launch time of ν_2 cannot be less than the sum of the following quantities: launch time of ν_1, execution time of ν_1, and the communication cost of arc e. Condition (11) means that the number of cores allocated for task ν_1 does not exceed linear scalability bounds specified by the marking γ in the context of formula (2). Condition (12) implies that, at any time t, the number of processor cores allocated for the tasks on the node with number i cannot exceed the total number of cores on this node. In what follows, any schedule is assumed correct unless otherwise specified.

A clustered graph with a specified schedule is called *scheduled* and denoted as $G = \langle V, \boldsymbol{E}, init, fin, \delta, \gamma, \omega, \xi \rangle$.

A *tier-parallel form* (TPF) [19] is a partition of the set of vertices V of a directed acyclic graph $G = \langle V, \boldsymbol{E}, init, fin \rangle$ into numbered subsets (tiers) L_i ($i = 1, ..., r$) satisfying the following conditions:

$$\left.\begin{aligned} & V = \bigcup_{i=1}^{r} L_i; \\ & \forall i \ne j \in \{1, \ldots, r\} (L_i \cap L_j) = \emptyset; \\ & \forall (v_1, v_2) \in E (\forall i \ne j \in \{1, \ldots, r\} \\ & (v_1 \in L_i \,\&\, v_2 \in L_j \Rightarrow i < j)). \end{aligned}\right\} \tag{13}$$

The last condition means that, if there is an arc from a vertex v_1 to a vertex v_2, then the vertex v_2 belongs to a tier whose number is greater than that of the tier where the vertex v_1 is located. The number of vertices in a tier L_i is called its *width*. The number of tiers in an TPF and the maximum width of its tiers are called *height* and *width* of the TPF. An TPF is said to be *canonical*

Table 1. Parameters of job graphs from the MCO class

Parameter	Semantics	Value
m_v	Task scalability	10
t_v	Task execution time on one core	100
δ	Amount of data transmitted through the arc	50

Table 2. Parameters of job graphs from the MCO class

Parameter	Semantics	Value
l	Job graph height	10
m_v	Task scalability	10
t_v	Task execution time on one core	100
δ	Amount of data transmitted through the arc	50

[19] if all *entry* vertices (the vertices that have no entry arcs) belong to the tier with number one and the maximum length of paths terminating at a vertex belonging to the kth tier is equal to $k - 1$.

Let $G = \langle V, \boldsymbol{E}, init, fin, \delta, \gamma, \omega, \xi\rangle$ be a scheduled graph and $y = (e_1, e_2, ..., e_n)$ be a simple path in it. The *cost of a path y* is calculated as

$$u(y) = \chi(fin(e_n), j_{fin(e_n)}) + \sum_{i=1}^{n} X(init(e_i), j_{init(e_i)}) + \\ + \max(\sigma(e_i), \tau_{fin(e_i)} - s_{init(e_i)})), \quad (14)$$

where ξ is the computational cost of the vertex given by formula (3); σ is the communication cost of the arc given by formula (5); j_v and τ_v are given by (6); and s_v is determined by formula (7).

Let Y be a set of all simple paths in a scheduled graph $G = \langle V, \boldsymbol{E}, init, fin, \delta, \gamma, \omega, \xi\rangle$. A simple path $\overline{y} \in Y$ is called *critical path* if

$$u(\overline{y}) = \max_{y \in Y} u(y); \quad (15)$$

i.e., the critical path has the maximum cost.

Proposition. Any critical path in a scheduled graph $G = \langle V, \boldsymbol{E}, init, fin, \delta, \gamma, \omega, \xi\rangle$ begins with an entry vertex and ends at an exit one (a vertex having no outgoing arcs).

Proof. We prove the proposition by contradiction. Suppose that there exists a critical path $\overline{y} = (e_1, e_2, ..., e_n) \in Y$ that begins with a vertex that is not an entry one. Then, there exists an arc e_h such that $fin(e_h) = init(e_h)$. Since G is an acyclic graph, it follows that $\tilde{y} = (e_1, e_2, ..., e_n) \in Y$. By virtue of (2), (3), and (14), we have $u(\overline{y}) < u(\tilde{y})$, which contradicts (15). Similarly, we arrive at a contradiction assuming that there exists a critical path ending at a vertex that is not an exit one. The proposition is proved.

4. ALGORITHM POS

In this section, we describe the POS (problem-oriented scheduling) algorithm designed for scheduling resources in distributed problem-oriented computing environments. A distinctive feature of the POS algorithm is that it takes into account information about specific subject domain when scheduling resources. In the framework of the model described in Section 3, this information is presented in labels of vertex-tasks specifying execution time of the task on one core and its scalability and in arc weights specifying amounts of data to be transmitted. The POS algorithm is designed for use in distributed computing systems with many-core processors.

To simplify the description and understanding of the algorithm, we will use three-level structure of the algorithm procedures. A first-level procedure is the main one. A step of the first-level procedure can be described as a second-level procedure. Such a step is highlighted by the semibold type. A similar approach can be used to describe second-level procedures.

4.1. Main Procedure

Consider a computing system in the form of an ordered set of computing nodes $\mathfrak{P} = \{P_0, ..., P_{k-1}\}$. Let a job graph $G = \langle V, \boldsymbol{E}, init, fin, \delta, \gamma\rangle$ be given. Suppose that the following conditions hold:

$$|V| \le |\mathfrak{P}|, \quad (16)$$

$$\forall v \in V (\forall P \in \mathfrak{P}(m_v \le |P|)), \quad (17)$$

where m_v is a linear scalability threshold given by a labeling function γ. Let us partition graph G into a canonical TPF with tiers L_i ($i = 1, ..., r$) and number vertices $V = (v_1, ..., v_q)$ of G such that the following property holds:

$$\forall i, j \in \{1, \ldots, q\} \\ ((v_i \in L_a \ \& \ v_j \in L_b \ \& \ a < b) \Rightarrow i < j); \quad (18)$$

i.e., vertices with greater numbers are located on lower tiers.

The most general form of the main procedure is as follows:

Step 1. **Construct an initial configuration** G_0.

Step 2. $i := 0$.

Step 3. **Construct configuration** G_i.

Step 4. If there are unconsidered arcs, $i := i + 1$ and go to Step 3.

Step 5. **Pack configuration** G_{i+1}.

Step 6. Stop.

The procedure operation consists in construction of a sequence of configurations. When turning to a

current configuration, at least one arc of the graph is marked as a considered one. Since the number of arcs is finite, the procedure stops on some iteration. The last constructed configuration G_{i+1} is selected to be the resulting configuration.

4.2. Procedure Constructing an Initial Configuration

Step 1.1. An initial clustering function ω_0 is specified as follows: $\forall i \in \{1, ..., q\}(\omega_0(v_i) = P_{i+1})$. That is, each vertex is mapped onto a separate computing node, and, accordingly, each cluster includes only one vertex.

Step 1.2. An initial schedule $\xi_0(v) = (t_v, j_v)$ is specified by determining launch time t_v iteratively by the TPF tiers:

$$\left.\begin{aligned} &\forall v \in L_1(\tau_v := 0);\\ &\forall v \in L_{i>1}\left(\tau_v := \max_{v'' \in L_i;\, v' \in L_{j<i}} (\lambda(v', v''))\right). \end{aligned}\right\}$$

Here,

$$\lambda(v', v'') = \begin{cases} s_{v'}, & \text{if} \quad (v', v'') \notin E;\\ s_{v'} + \sigma((v', v'')), & \text{if} \quad (v', v'') \in E, \end{cases}$$

where $s_{v'}$ is calculated by formula (9). The number of cores j_v allocated to vertex v is determined as $\forall v \in (j_v = m_v)$.

Step 1.3. $G_0 = \langle V, \boldsymbol{E}, init, fin, \delta, \gamma, \omega_0, \xi_0\rangle$.

Step 1.4. End of the procedure.

4.3. Procedure Constructing Configuration G_{i+1}

We define a *subcritical path* as a path having maximum cost among all paths containing at least one unconsidered arc. The procedure constructing configuration G_{i+1} is as follows:

Step 3.1. Find a subcritical path in $\tilde{y}_i = (e_1, ..., e_n)$.

Step 3.2. Find the first unconsidered arc e_j ($1 \leqslant j \leqslant n$) in $\tilde{y}_i$ and mark it as a considered one.

Step 3.3. If $i = 0$, then mark vertex $init(e_j)$ as a fixed one.

Step 3.4. If vertices $init(e_j)$ and $fin(e_j)$ are fixed, go to step 3.14.

Step 3.5. If vertex $fin(e_j)$ is not fixed, then $v'' := fin(e_j)$, $v' := init(e_j)$.

Step 3.6. If vertex $init(e_j)$ is not fixed, then $v'' := init(e_j)$, $v' := fin(e_j)$.

Step 3.7. Construct clustering function ω_{i+1} that differs from function ω_i by only one value: $\omega_{i+1}(v'') := \omega_i(v')$.

Step 3.8. **Construct schedule** ξ_{i+1}.

Step 3.9. $G_{i+1} = \langle V, \boldsymbol{E}, init, fin, \delta, \gamma, \omega_{i+1}, \xi_{i+1}\rangle$.

Step 3.10. Find a critical path $\overline{y}_i$ in G_i.

Step 3.11. Find a critical path $\overline{y_{i+1}}$ in G_{i+1}.

Step 3.12. If $u(\overline{y}_{i+1}) \leqslant u(\overline{y}_i)$, go to Step 3.16.

Step 3.13. $G_{i+1} := G_i$.

Step 3.14. If there are unconsidered arcs in $\tilde{y}_i$, go to Step 3.2.

Step 3.15. If there are unconsidered arcs in G_i, go to Step 3.1.

Step 3.16. End of the procedure.

4.4. Procedure Constructing Schedule ξ_{i+1}

Let us introduce the following notation: $T(x)$ is the number of the tier the vertex x belongs to; $W_{\omega_i(x)} = \{v | v \in C, \omega_i(v) = \omega_i(x)\}$ is the cluster the vertex x belongs to. The procedure constructing schedule ξ_{i+1} includes the following steps:

Step 3.8.1. $R := W_{\omega_i(v')} \cap L_{T(v'')}$.

Step 3.8.2. If $R = \emptyset$ or $\sum_{v \in R} j_v \leq |P_{\omega_i(v')}|$, go to Step 3.8.7.

Step 3.8.3. For $h = q, ..., T(fin(e_j) + 1)$, perform $L_{h+1} := L_h$.

Step 3.8.4. $L_{T(v'')+1} := \{v''\}$; $L_{T(v'')} := L_{T(v'')} \backslash \{v''\}$.

Step 3.8.5. $q := q + 1$.

Step 3.8.6. Construct a new schedule ξ_{i+1} by calculating launch times t_v for all vertices $v \in V$.

Step 3.8.7. Mark vertex v'' as a fixed one.

Step 3.8.8. End of the procedure.

4.5. Procedure Packing Configuration G_{i+1}

The goal of the packing procedure is to minimize the number of the computing nodes involved. This procedure is applied to the clusters that contain only one vertex. This constraint is used because, if there are two adjacent vertices in a cluster, then transition of one of them to another cluster can increase the total execution time of the job. The procedure for packing configuration G_{i+1} is as follows:

Step 5.1. $\mathfrak{M} := \emptyset$.

Step 5.2. For all $v' \in V$, do the loop

Step 5.2.1. $W = \{v | v \in V, \omega_{i+1}(v) = \omega_{i+1}(v')$.

Step 5.2.2. If $W \in \mathfrak{M}$, go to the next iteration of the loop.

Step 5.2.3. $\mathfrak{M} := \mathfrak{M} \cup \{W\}$.

Step 5.4. $\mathfrak{E} := \{W \in \mathfrak{M} \| W| = 1\}$; $\mathfrak{V} := \{W \in \mathfrak{M} \| W| > 1\}$.

Step 5.5. For all $W' \in \mathfrak{E}$, do the loop

Step 5.5.1. For $l = 1, ..., r$, do the loop

Step 5.5.1.1. If $W' \cap L_l = \emptyset$, go to the next iteration of the loop.

Step 5.5.1.2. For all $W' \in \mathfrak{V}$, do the loop

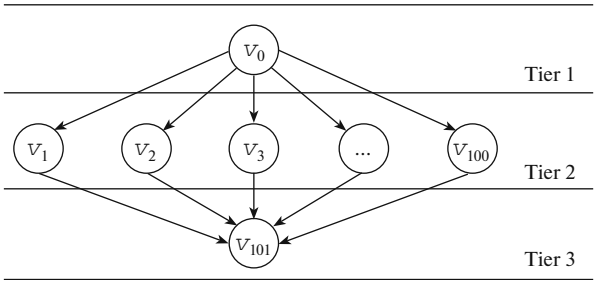

Fig. 4. An example of a job graph from the MCO class.

Step 5.5.1.2.1. If $W'' \cap L_l = \emptyset$, go to the next iteration of the loop.

Step 5.5.1.2.2. Take $v' \in W'$.

Step 5.5.1.2.3. Take $v'' \in W''$.

Step 5.5.1.2.4. If $j_{v'} + \sum_{v \in W'' \cap L_l} j_v > \omega_{i+1}(v'')$, go to the next iteration of the loop.

Step 5.5.1.2.5. $i := i + 1$.

Step 5.5.1.2.6. Construct clustering function ω_{i+1} that differs from function ω_i by only one value: $\omega_{i+1}(v') = \omega_{i+1}(v'')$.

Step 5.5.1.2.7. $W'' := W'' \cup W'$.

Step 5.5.1.2.8. Go to Step 5.5.3.

Step 5.5.1.3. End of loop.

Step 5.5.2. End of loop.

Step 5.5.3. Go to the next iteration of the loop.

Step 5.6. End of loop.

Step 5.7. End of the procedure

Note that Step 5.2 organizes the loop that constructs the set of all clusters $\mathfrak{M}$ that constitute the job. On Step 5.4, the set $\mathfrak{E}$ of *unitary clusters* (clusters containing only one vertex) and the set $\mathfrak{B}$ of *multiclusters* (clusters containing two or more vertices) are calculated. Step 5.5 organizes the loop over all unitary clusters. For every unitary cluster, we find a tier of the parallel form to which the given unitary cluster belongs. In the framework of this tier, we try to combine the unitary cluster with some multicluster. This is possible if the multicluster uses not all processor cores and the number of free cores is sufficient in order to perform the unitary cluster being joined.

5. STUDY OF THE POS ALGORITHM

The POS algorithm was studied on the following two classes of jobs: multicriteria optimization (MCO) and random jobs (RJ). Further, we consider both classes.

5.1. The MCO Class

The *MCO class* includes computational jobs in the field of multicriteria optimization, which constitute a large portion of load of modern supercomputers and distributed computing systems. The TPF of job graphs from the MCO class consists of three tiers. The first and third tiers of the TPF contain one vertex each. The second tier contains w vertices. The number w is specified upon graph generation. The vertex from the first tier is connected by arcs with all vertices of the second tier. To each task–vertex, two numbers—maximum scalability m_v of the task and the task execution time t_v on one processor—are assigned. Numbers m_v and t_v are constant for all tasks. Similarly, to each arc, the amount of data transmitted δ is assigned, which is the same for all arcs. Figure 4 shows an example of a job graph from the MCO class for the width $w = 100$.

Table 3. Parameters of job graphs

Parameter	Semantics	Value
m_v	Task scalability	20
t_v	Task execution time on one core	40
δ_{M1}	Average arc weight for the group M1	40
δ_{M2}	Average arc weight for the group M2	10
δ_{M3}	Average arc weight for the group M3	400
d	The number of cores on computing node	100

5.2. The RJ Class

The *RJ class* includes random jobs with different numbers of vertices and arcs. A qualitative characteristic of a job graph in the RJ class is the ratio T/Δ, where T is an average time of job execution on one core and Δ is an average arc weight. In terms of this parameter, the following three important groups can be identified in the RJ class [17]:

1. M1. *Balanced* job graphs with T/Δ. For these jobs, time required for data transmission is comparable with the computation time.

2. M2. *Coarse-grained* job graphs with T/Δ. In these jobs, the major part of time is spent on calculation.

3. M3. *Fine-grained* job graphs with T/Δ. In these jobs, the major part of time is spent on data transmission, while calculations take insignificant time.

5.3. Results of Experiments

In this section, we present results of computational experiments on comparison of the POS algorithm with other known algorithms.

In the first series of the experiments, we studied density of schedules generated by the POS algorithm. Under the density, we mean here the quantity that is inversely proportional to the number of the computing

Table 4. Comparison of algorithms POS, DSC, and Min–Min for group M1

2*no.	2*l	2*w	2*$\|V\|$	2*$\|E\|$	The number of computing nodes involved			Ratio of job execution times	
					POS	DSC	Min–Min	DSC/POS	Min–Min/POS
1	5	10–20	51	126	7	21	4	3.78	8.86
2	10	10–20	117	296	4	36	4	3.67	9.53
3	10	20–30	211	505	16	68	6	3.77	8.36
4	20	5–10	137	252	2	46	2	10.19	17.74
Average value								5.35	11.13

Table 5. Comparison of algorithms POS, DSC, and Min–Min for group M2

2*no.	2*l	2*w	2*$\|V\|$	2*$\|E\|$	The number of computing nodes involved			Ratio of job execution times	
					POS	DSC	Min–Min	DSC/POS	Min–Min/POS
1	5	10–20	49	113	3	25	4	6.67	5.67
2	10	10–20	130	390	12	42	4	4.79	2.51
3	10	20–30	206	464	17	73	6	4.05	3.47
4	20	5–10	141	290	13	41	2	4.79	1.47
Average value								5.08	3.28

Table 6. Comparison of algorithms POS, DSC, and Min–Min for group M3

2*no.	2*l	2*w	2*$\|V\|$	2*$\|E\|$	The number of computing nodes involved			Ratio of job execution times	
					POS	DSC	Min–Min	DSC/POS	Min–Min/POS
1	5	10–20	59	190	4	21	4	4.50	13.15
2	10	10–20	123	378	3	34	4	4.89	42.46
3	10	20–30	201	453	9	61	6	2.06	10.78
4	20	5–10	133	287	2	29	2	3.13	7.88
Average value								3.64	18.57

nodes used for the job execution. First, we studied schedule density for tasks from the MCO class. Experiments were carried out for three values of the job graph width w: 100, 200, and 300. For all vertices and arcs of the graphs, identical weight values and labeling were used, which are shown in Table 1.

Results of the experiments are shown in Fig. 5a in the form of dependence of the number of the computing nodes used on the number of the processor cores in one computing node. The plots demonstrate that, when the number of cores in a computing node increases, the schedule density for MCO jobs increases and tends to its maximum value equal to 1 in all considered cases. It should be noted that the schedule density grows considerably when the job graph width increases.

Then, we studied schedule density for tasks from the RJ class. Experiments were carried out for three values of the job graph width w: 30, 50, and 70. For all vertices and arcs of the graphs, identical weight values and labeling were used, which are shown in Table 2.

Results of the experiments are shown in Fig. 5b. The plots demonstrate that, when the number of cores in a computing node increases, the schedule density for RJ jobs also increases and tends to its maximum value equal to 1 in all considered cases. It should be noted that the schedule density grows considerably upon increase of the job graph width.

5.4. Comparison of POS with Other Algorithms

In the second series of the experiments, we studied efficiency of the POS algorithms compared to the DSC [17] and Min—Min [16] scheduling algorithms. To this end, we prepared three groups of job graphs—M1, M2, and M3—with the parameters shown in Table 3.

In each group, graph height l and weight w were varied. Results of the experiments are presented in Tables 4–6. We compared the POS algorithm with the DSC and Min—Min algorithms by calculating the numbers of the computing nodes used and the ratios of

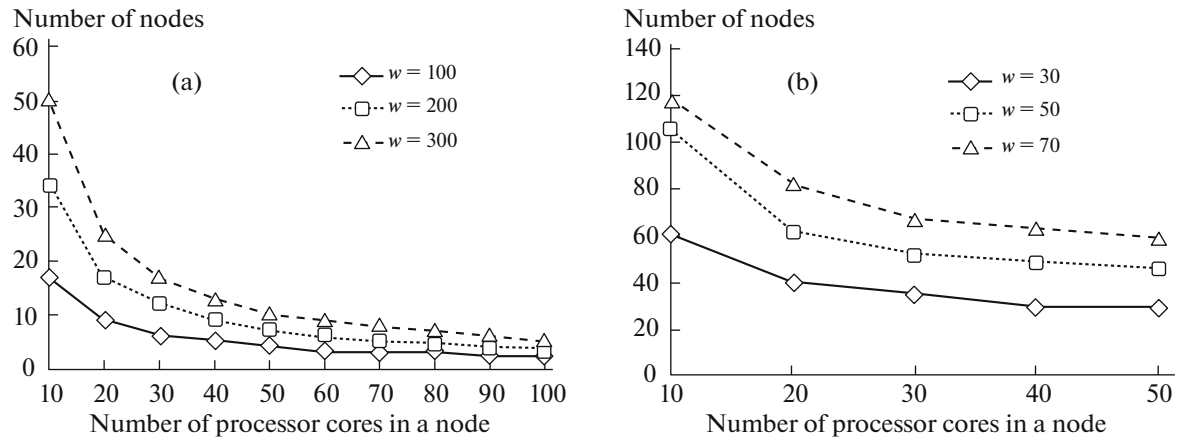

Fig. 5. Dependence of the number of the computing nodes involved on the number of the processor cores in a computing cluster for (a) the MCO class and (b) the RJ class.

the job execution times by the DSC and Min–Min algorithms to the job execution time by the POS algorithm.

The results of the experiments show that, for various jobs from the MCO and RJ classes, the POS algorithm generates schedules that are more efficient than those generated by the DSC and Min–Min algorithms. This is achieved owing to the facts that the POS algorithm analyzes all data dependencies like the DSC algorithm and ensures possibility of distribution of tasks over processor cores like the Min–Min algorithm.

6. CONCLUSIONS

In the paper, we discussed resource management in problem-oriented distributed computing environments. A new model of the computing environment and POS (problem-oriented scheduling) algorithm for scheduling resources for the workflow applications are suggested. The algorithm allows one to schedule execution of one task on several processor cores with regard to constraints on scalability of the task. Results of computational experiments on studying adequacy and efficiency of the proposed algorithm for work in problem-oriented distributed computing environments are presented.

ACKNOWLEDGMENTS

This work was supported by the Russian Foundation for Basic Research, project no. 15-29-07959.

REFERENCES

1. Buyya, R., Abramson, D., *et al.*, Economic models for resource management and scheduling in grid computing, *J. Concurrency Comput.: Practice Experience*, 2002, vol. 14, nos. 13–15, pp. 1507–1542.
2. Foster, I. and Kesselman, C., *The Grid. Blueprint for a New Computing Infrastructure*, San Francisco: Morgan Kaufman, 1999.
3. Foster, I., Roy, A., and Sander, V., A quality of service architecture that combines resource reservation and application adaptation, *Proc. of the 8th Int. Workshop on Quality of Service*, Pittsburgh, 2000, pp. 181–188.
4. Foster, I., Kesselman, C., and Tuecke, S., The anatomy of the grid: Enabling scalable virtual organizations, *Int. J. Supercomput. Appl. High Perform. Comput.*, 2001, vol. 15, no. 3, pp. 200–222.
5. Jennings, R., *Cloud Computing with the Windows Azure Platform*, Indianapolis: Wiley, 2009.
6. Marshall, P., Keahey, K., and Freeman, T., Improving utilization of infrastructure clouds, *Proc. of the IEEE/ACM Int. Symp. on Cluster, Cloud and Grid Computing (CCGrid 2011)*, Newport Beach, CA, 2011, pp. 205–214.
7. Berman, F., Wolski, R., *et al.*, Application-level scheduling on distributed heterogeneous networks, *Proc. of the ACM/IEEE Conf. on Supercomputing*, Pittsburgh, 1996, paper no. 39.
8. Iverson, M. and Ozguner, F., Dynamic, competitive scheduling of multiple DAGs in a distributed heterogeneous environment, *Proc. of Seventh Heterogeneous Computing Workshop*, Orlando, 1998, pp. 70–78.
9. Khokhar, A.A., Prasanna, V.K., *et al.*, Heterogeneous computing: Challenges and opportunities, *IEEE Comput.*, 1993, vol. 26, no. 6, pp. 18–27.
10. Maheswaran, M., Ali, S., *et al.*, Dynamic matching and scheduling of a class of independent tasks onto heterogeneous computing systems, *J. Parallel Distrib. Comput.*, 1999, vol. 59, no. 2, pp. 107–131.
11. Zhu, Y. and Ni, L.M., A Survey on grid scheduling systems, *Technical Report no. SJTU_CS_TR_200309001, Dept. of Computer Science and Engineering, Shanghai Jiao Tong Univ.*, 2013.
12. Belhajjame, K., Vargas-Solar, G., and Collet, C., A flexible workflow model for process-oriented applications, *Proc. of the Second Int. Conf. on Web Informa-*

tion Systems Engineering (WISE'01), Kyoto, 2001, vol. 1, no. 1, pp. 72–80.

13. Dong, F. and Akl, S.G., Scheduling algorithms for grid computing: State of the art and open problems, *Technical Report no. 2006-504, Queen's University, Canada*, 2006.
14. Kim, S.J., A general approach to multiprocessor scheduling, *Report TR-88-04, Dept. of Computer Science, University of Texas at Austin*, 1988.
15. Yang, T. and Gerasoulis, A., DSC: Scheduling parallel tasks on an unbounded number of processors, *IEEE Trans. Parallel Distrib. Syst.*, 1994, vol. 5, no. 9, pp. 951–967.
16. Yu, J., Buyya, R., and Ramamohanarao, K., Workflow scheduling algorithms for grid computing, in *Metaheuristics for Scheduling in Distributed Computing Environments*, ser. *Studies in Computational Intelligence*, Xhafa, F. and Abraham, A., Eds., Berlin: Springer, 2008, vol. 146, pp. 173–214.
17. Gerasoulis, A. and Yang, T., A comparison of clustering heuristics for scheduling directed acyclic graphs on multiprocessors, *J. Parallel Distrib. Comput.*, 1992, vol. 16, no. 4, pp. 276–291.
18. Sarkar, V., *Partitioning and Scheduling Parallel Programs for Execution on Multiprocessors*, Cambridge, MA: MIT Press, 1989.
19. Voevodin, V.V. and Voevodin, Vl.V., *Parallel'nye vychisleniya* (Parallel Computations), St.-Petersburg: BKhV-Peterburg, 2002.

Translated by A. Pesterev