Building an Algorithmic Skeleton for Block Data Processing on Enterprise Desktop Grids[*]

✉ Sergei Vostokin [0000-0001-8106-6893] and Irina Bobyleva [0000-0001-5660-3503]

Samara National Research University,
Samara, Russia

✉easts@mail.ru, ikazakova90@gmail.com

Abstract. The paper presents a method for building an algorithmic skeleton and automation of the pairwise processing of block data on enterprise desktop grid computing environment. The automation is based on the principle of the round-robin (each with each) tournament. The semantics of calculations and decomposition of the algorithmic skeleton into sequential subprograms using the model of actors is given. A graphical notation explaining the relationship between the elements of the algorithmic skeleton is introduced. The applicability of the method was studied on block sorting of a large data set.

Keywords: Actor model·Enterprise desktop grid·Round-robin tournament·Block sort·Algorithmic skeleton

1 Introduction

The desktop computer grids have long and successful history of usage in the field of scientific computing, both on the Internet and on the scale of enterprises. The main reason why the desktop grids are in wide use is the radical reduction of the computational cost. The grids built on desktop computers (broadly, on any personal computing systems: smartphones, tablets, laptops, etc.) makes it possible to use a large amount of devices on the Internet for your computations with a consent of the devices owners. In many cases, that is more convenient and cheaper than using cluster or supercomputer. The use of temporarily idle (for example at night hours) equipment of the enterprise is another alternative solution for cheaper computations.

While hardware costs are reduced, programming costs begin to play an important role in the overall cost of grid computing. The costs remain low, while desktop grids are used to implement simple search or brute force strategies with massive parallelism. This is due to the fact that such problems are native to desktop grids. The problems are easy to program with the desktop grid APIs. Also there are ready to use algorithmic skeletons [1] of the MAP type (when an operation is applied to all elements of data set

[*] Supported by the Ministry of Education and Science of the Russian Federation in the framework of the State Assignments program (#9.1616.2017/4.6).

in parallel) in which the general management of calculations is already implemented [2]. The researcher only needs to define algorithms for the tasks to be solved on the desktop computer grid.

However, as communication and computing equipment improves, the use of desktop grid systems for solving computational problems with more complex control (compared to massively parallel control) becomes relevant. For example, these may be complex data analysis tasks. An enterprise can accumulate data during the day and process it on idle computers at night.

In the research we propose a method and apply it to build an algorithmic skeleton for automation of calculations on desktop grid systems. The skeleton is designed to simplify the programming of pairwise processing of data blocks. This processing is similar to the round-robin sport tournament, where teams play with each other. This type of processing can be used for sorting, constructing frequency distributions, and solving similar problems.

The article has the following structure. Firstly, based on the model of actors and a special graphical notation, generalized description of the parallel computing semantics in the desktop grid is created. The description follows the principles of top-bottom decomposition of algorithms for the grid systems and allows one to define data types and sequential procedures for a specific parallel algorithm. Secondly, we specify the description of the data types and procedures by defining the well known algorithmic skeleton called *bag of tasks*. In turn, we present a new skeleton called *asynchronous round-robin tournament*. The skeleton is built on the basis of the bag of tasks skeleton. Finally, the applicability of the asynchronous round-robin tournament skeleton is studied experimentally. Using a cluster model of enterprise's desktop grid environment, we implement and test block sorting application for large data arrays and make a conclusion about possible speedup of data processing in real enterprise grid systems.

2 Related Work

The most common systems for distributing calculations across the desktop grid are Condor [3], BOINC [4], XtremWeb [5], OurGrid [6]. These systems can be implemented at various scales, ranging from office or laboratory to all the world [7]. The characteristics and classification of desktop computer grids are given in [8, 9].

The desktop computer grids can be used not only in the interests of science, but also in the interests of various enterprises. Not all grid systems can really be effectively applied within the local network of an enterprise, since some of them can solve only a narrow class of problems and require a high-speed and uninterrupted connection between elements of the system, which is not always possible. But the solution of this issue and systems capable of working in the network of an enterprise are considered in [10]. Task scheduling techniques can also be used to minimize server load and optimize the desktop grid efficiency [11].

The model presented in the paper is based on the actor model — a model of parallel computation proposed by Carl Hewitt in 1973. Since its inception, it has been actively explored and applied to solve various problems. In the article [12], the authors described in detail all the basic properties of the model of actors, as well as the entire history of its changes over the past time. We also use the bag of tasks model of computations. The authors set forth in detail the bag of tasks model and presented the results of a number of experiments on its application in [13].

In experimental research, we considered a cluster as a model for the desktop grid. The use of a computing cluster as a desktop grid was also studied in [14, 15].

In the article we provide a point of view on actor model and its application in the context of building algorithmic skeletons [1] for the desktop grids controlled by Everest platform [16] which we consider relevant.

3 The Actor Model of Algorithmic Skeletons for Desktop Grid Applications

From a general point of view, the desktop grid calculations are organized according to the master-workers scheme. The master process coordinates the work of connected worker processes. The master and workers processes form communication graph with a star topology. Let's build an actor model of the master-workers scheme. For this, we use a special graphical notation for the actor's semantics of execution.

The *master* class is shown in Fig. 1a. The master process simulates the work of the controlling process or the orchestrator of the desktop grid.

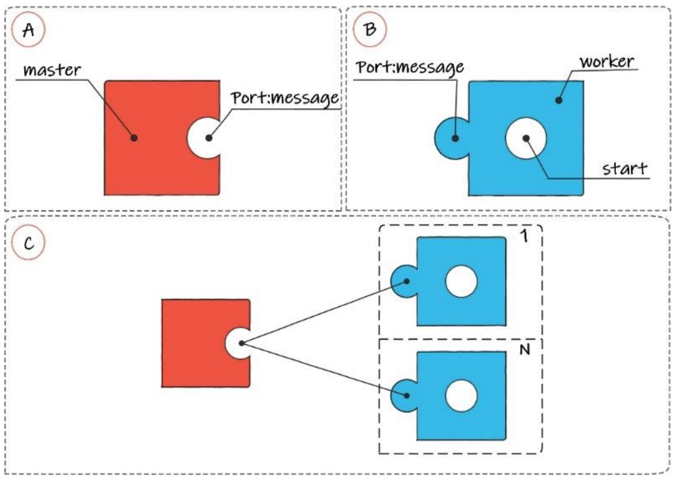

Fig. 1. The master and the worker: a) the master; b) the worker; c) the interaction of master and workers.

Fig. 1a means that objects of type *master* can receive messages of type *message* in the *port*. In C ++ programming language, this can be encoded as follows: *struct message{..}; struct master{void port_handler(message&m){..}..};*.

The class of *worker* processes is shown in Fig. 1a. This class simulates the work of a desktop computer connected to the grid.

The notation in Fig. 1b is interpreted according to the following description. The worker process can receive messages of type *message* on port named *port*. Additionally, an instance of the *message* type is associated with each *worker* instance. The circle inside the worker process symbol means a handler with the name *worker::start*. This is a system message handler. The message arrives at the beginning of the calculation. In C++, this can be encoded as follows: *struct worker{ void port_handler(message&m){..}; message port; void start(){..}..};*.

The behavior of message objects and the order of calls to message handlers **_handler* is based on the actor's semantics of execution. The message object can be in two states: (a) in the state of interconnection with some actor object; (b) in the state of delivery to port of some actor object. At the time of delivery, the message handler **_handler* is activated. The actor may have a special message handler named *start*. The *start* handler is activated at the beginning of the calculations.

There are rules for accessing actor variables and messages from message handlers. Access to variables of the actor object for which the handler is called is allowed. Access to variables of the message object being in the state of interconnection with the actor to which the handler is called is also allowed.

Two primitive operations are available for managing messages in the context of the ** _handler* and *start* handlers: *access*() and *send*(). The access operation is used to check the availability of the message. Checking the availability of a message means asking the runtime system whether the message is in a state of interconnection with the actor object or not. The *m.send*() operation is used to send a message *m* to a port of some actor. The *m.send*() call is allowed if you have an access to the message *m* (the *access*(*m*) call returns *true*). After the execution of *m.send*() the access is lost until the completion of the current handler. When the handler is activated, the message *m* transmitted as a handler *void *_handler(message&m)* parameter is available (the *access*(*m*) call returns *true*). The runtime ensures that the message *m* is eventually delivered after the execution of *m.send*() operation, but no assumptions are made about the message delivery sequence.

A message object is used for communication between a pair of actor objects. For this purpose the link between two actor's ports are established. The links are shown on Fig. 1c. In the model of calculations on the desktop grid, one master actor is associated with N instances of the worker actors. In C++, links are encoded as follows: *struct master{void port(message&){..}..}; struct worker{message port;..}; master a_master; worker a_worker; a_master.port(a_worker.port);*.

At the initial moment, the message is available in the actor where the message was declared. In Fig. 1c, these are actors of *worker* type. The message is then used to request the port of another associated actor. In Fig. 1c, this is a *master* actor. After processing, the same message object is used for the response. The interaction can be repeated many times. Thus, the client-server interaction between actors is implemented.

Our goal is to develop an algorithmic skeleton, which is a specialization of the model in Fig. 1c. Therefore, we will further need to define the following types: *message*, *master*, and *worker*. Also we will need to define a message handlers *master::port_handler*, *worker::port_handler*, and *worker::start* associated with the listed types. The definition is given in the next two sections.

4 Specification of Bag-of-Tasks Skeleton

Let's further clarify the computation model for the desktop grid system. To solve an applied problem, the user defines the following parts of the code. He defines the state of the master process – *struct bag{..}*; the state of a task – *struct task{..}*; the function that tests for the presence of a task – *bool test(bag&)*; the function that gets a new task – *void get(task&,bag&){..}*; the function that processes a task – *void proc(task&){..}*; the function that puts the results of calculations in the state of the master process – *void put(task&,bag&)*. The interaction of the listed functions can be described in the form of sequential C++ code as follows: *bag b; task t; while(test(b)){ get(t,b); proc(t); put(t,b); }*.

The algorithmic skeleton that takes the listed types and functions as parameters is usually called the bag of tasks. Having a previously defined general model of computations on the desktop grid, now we can define the bag of tasks skeleton.

The bag will be a part of the master actor state*: struct master{bag b;..}*; the task will be a part of the message state: *struct message{task t;..}*. When processing a message, it is necessary to determine whether the result of the previous calculation is delivered in it. To do this, we introduce the flag: *struct message{bool is_first;..}*. When sending messages from a worker to the master at the beginning of calculations, the flag of the first message is set to *true*: *void worker::start(){port.is_first=true; port.send();}*. When a response is received from the master, the task transferred to the worker is processed: *void worker::port_handler(message&m){proc(m.t); m.send();}*. The result of the processing is sent in a reply message. This is where the worker description is complete.

To describe the behavior of the master process, it is required to keep a list of pointers to messages from workers waiting for the task*: struct master{list<message*> wait;..}*. The message processing method in the master process is performed in 3 consecutive steps: *void master::port_handler(message&m){Step_1;Step_2;Step_3;}*.

Step 1. We put the results of processing the message (*m*) into the state (*b*) of the bag, and the pointer to the message into the wait queue: *if(m.is_first) m.is_first=false; else put(m.t,b); wait.push_back(&m);*. Notice that the first message from the worker does not contain the result of task calculation and is not processed in the *put*().

Step 2. We issue tasks from the bag for processing by the waiting workers: *while(!wait.empty() && test(b)){message*m=wait.back(); wait.pop_back(); get(m->t, b); m->send();}*.

Step 3. We check the completion of calculations: *if(wait.size()==N) stop();*. Calculations are completed if all workers are waiting for tasks. *N* is the number of worker processes in Fig. 1c.

Thus, we defined the semantics of the algorithmic skeleton called BOT (bag of tasks) as a higher order function on types and ordinary functions: *BOT<struct bag; struct task; bool test(bag&); void get(task&,bag&); void proc(task&); void put(task&,bag&)>*.

5 Specification of Asynchronous Round-Robin Tournament Skeleton

Having the definition of the bag of task skeleton, we can define more specialized skeleton for pairwise processing of data blocks. We call this skeleton the asynchronous round-robin tournament: *ART<void prepare(int team); void play(int team_i, int team_j)>*. The functionality of the skeleton can be represented as the organization of a sports circular tournament, in which M teams participate. Each team plays with each other team. In this case, the total number of games played is $M(M-1)/2$. You cannot simultaneously assign a team to play with more than one rival team. Additional restrictions with the required properties of the tournament may be imposed. However, we expect that these restrictions do not change in time depending on the results of already played games.

For example, a plan of a tournament in the form of a sequence of games can be written as a program in C++:

```
for (int i = 0; i < M; i++) prepare(i);                                (1)
for (int i = 1; i < M; i++) for (int j = 0; j < i; j++) play(j, i);
```

For an asynchronous parallel tournament plan, a directed acyclic graph (DAG) is required. This graph determines for every game after which games this game is played. It can be shown that it is enough to track only two games preceding the *play*(*i*,*j*) of the game: namely a game in which ith team played, and a game in which jth team played. Then the rules of an asynchronous tournament in the form of an oriented acyclic dependency graph of its tasks can be encoded with the two matrices of size $M \times M$ (*int I1[M][M]; int I2[M][M];*) as shown in Fig. 2.

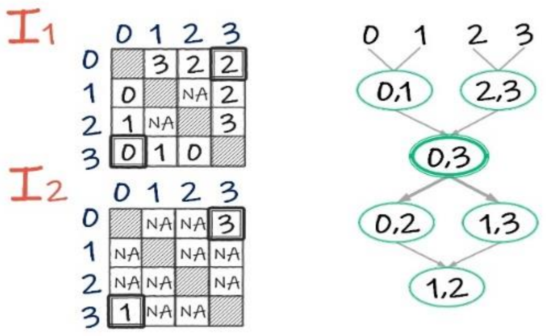

Fig. 2. Coding the task dependencies of an asynchronous round-robin tournament.

Game plays for which the *play*(*i*,*j*) is the previous one can be defined as *play*(*I1[j][i]*, *I1[i][j]*) and *play* (*I2[j][i]*, *I2[i][j]*), where *I1* and *I2* are matrices of size $M \times M$. If the *play*(*i*, *j*) of a game (*i*, *j*) does not have dependent games or has only one such game, then the unused elements of the matrices *I1* and *I2* contain special value: *const int NA = -1;* .

Using the matrices *I1* and *I2*, the auxiliary matrix *int S[M][2]* is calculated. This matrix encodes the *play*(*S[i][0], S[i][1]*), immediately following the preparation *prepare*(*i*) of the ith command.

Let's consider the variables needed to track the current state of the tournament. The *int D[M][M]* matrix stores the number of unplayed games *D[i][j]* preceding the *play*(*i, j*) directly. The *list<pair<int,int>> games*; contains games (play operations) or preparations to the games (prepare operations) that are not yet assigned to run. To distinguish the preparation from the play in the list of games, a special value *NA* is recorded in the second element of the *pair <int, int>*.

The above description of the state and methods of BOT skeleton from Section 4 allow us to determine the state and methods of the ART skeleton (asynchronous round-robin tournament) as follows.

The bag state is supplemented by the following data fields: *struct bag{int D[M][M]; list<pair<int,int>> games; int I1[M][M]; int I2[M][M]; int S[M][2];}*. Matrices *I1*, *I2*, and *S* are filled at the beginning and do not change during the calculations. The matrix *S* is trivially calculated from the matrices *I1* and *I2*. The list of games is initially filled according to the C++ code: *for(int i = 0; i < M; i++) {pair<int,int> p(i, NA); games.push_back(p);}*. The task state is defined as *struct task{int i; int j;}*. Data fields of the structure correspond to *play*(*i, j*) for $j \neq NA$ and *prepare*(*i*) for $j = NA$.

The code for checking whether a task is present in the current state of the bag is *bool test(bag&b){return !b.games.empty();}*.There are tasks for the processing if the games list is not empty.

The task retrieval code is *void get(task&t,bag&b){pair<int,int>p=b.games.back(); b.games.pop_back(); t.i=p.first;t.j=p.second;}*. It means taking the last element of the list of games as a task.

The task processing code is *void proc(task&t){if(t.j==NA) prepare(i); else play(t.i,t.j);}*. It means performing a user-defined prepare or play procedure.

The code for calculating the next tasks at completion of the task *t* implements the following idea of adding tasks to the games list. When a prepare task has completed, it affects the launch of one play task. When a play task has completed, it can affect the launch of one or two play tasks. The tasks that can be performed at the completion of a current task are found by the matrices *I1*, *I2*, *S*. The value of the counter *D* decreases for the tasks that can be potentially performed. If the counter has reached zero, then the task stop waiting for the completion of the previous tasks and is planned by adding on the games list.

The procedure of getting the matrices *I1* and *I2* for a tournament with given properties is not discussed in detail here. We only note that in the experimental study described below we used the following procedure. First, a sequence of tournament games was formed, for example, using the algorithm (1). Then this sequence was parallelized. At the end, the resulting dependency graph of the tournament games was encoded with matrices *I1* and *I2*.

Thus, we defined the semantics of the algorithmic skeleton named asynchronous round-robin tournament as the higher order function: *ART<void prepare(int i); void play(int i, int j)>*.

6 Experimental Study of Asynchronous Round-Robin Tournament Skeleton

The asynchronous round-robin tournament skeleton was used to build the orchestrator program that controls task submission to the enterprise desktop grids. The prepare and play parameters of the skeleton are algorithms for processing tasks on the desktop grid system. Testing was performed on a typical problem of sorting a large data set. The data set was divided into several files. The *prepare*(*i*) algorithm implemented the sorting of a single file identified by the block number *i* in the data set. As a result of the *play*(*i, j*) algorithm, the numbers in the pre-sorted files *i* and *j* were ordered so that the concatenation of files *i* and *j* formed an ordered sequence of numbers. The overall processing result was a sorted set of *M* files with identifiers 0, 1, 2, .., *M*-1. Sequential concatenation of 0, 1, 2, .., *M*-1 files formed an ordered sequence of numbers at the end of processing. At the beginning of processing the files were filled with random numbers.

In the experiments we studied: the rate of issuing prepare/play tasks in the implementation of the orchestrator according to the scheme in Fig. 1c; the sorting speedup when prepare/play tasks are calculated in distributed computing environment; the setting of the orchestrator to work in the Everest platform. For all experiments, the orchestrator was implemented in C++ language using Visual Studio 2015 compiler for the desktop system and GCC 4.1.2 compiler for the cluster system. We used x86_64 optimal performance compilation mode.

The rate of task submission and the correctness of the algorithm that implements the ART skeleton was tested on Intel(R) Core(TM) i3 - 3220T CPU @ 2.80GHz computer with 4 GB of RAM on board. We used stubs of the prepare/play algorithms and 4 worker processes (*N*=4 in Fig. 1c). Blocks containing only one integer number were processed in the prepare/play stubs. The data set was stored in RAM. The number of data elements varied from 100 to 1000 in increments of 100. The maximum number of tasks in the experiment was 1000 prepare tasks and $1000 \times (1000 - 1)/2$ play tasks, that was 500,500 tasks in total. Two types of tournaments were tested: the TRIV sort tournament constructed by parallelizing the algorithm (1); the OPTIM sort tournament with a smaller DAG diameter compared to the TRIV sort. The test results are shown in Table 1.

The experiment showed that sequential actor implementation with message exchange is suitable for controlling distributed processing in the desktop grid even in the case of a large number of short tasks. One task requires two messages (see Fig. 1c). It results in sending 1,001,000 messages between actors, when the number of blocks is equal to 1000. But the processing time still remains less when 0.5 second on a processor with relatively low performance. Also it can be seen that different tournaments (differing in matrices *I1* and *I2*) are almost the same in terms of execution time.

In the second series of experiments, the possibility of obtaining a speedup when processing a large set of data on the enterprise desktop grid was investigated. As a model of enterprise desktop grid environment, we used Sergey Korolev cluster system. Sergey Korolev cluster is installed at the Samara University. The cluster configuration is presented on the *hpc.ssau.ru* website. In the tested MPI program, the control process (rank

= 0) served as the orchestrator, the remaining worker processes (rank > 0) processed tasks from the orchestrator. The files being sorted were available in all MPI-processes using the IBM GPFS distributed file system.

Table 1. Total submission time of all tournament tasks depending on the number of blocks in the data set.

Number of blocks	OPTIM sort, s	TRIV sort, s
100	0.00260576	0.00328716
200	0.0131043	0.013965
300	0.0362816	0.0354059
400	0.0487192	0.0705439
500	0.0654783	0.0765974
600	0.142984	0.122198
700	0.215872	0.185085
800	0.199767	0.319882
900	0.277575	0.260136
1000	0.390505	0.420731

The sorting was applied to a data set of size from 10 to 100 files in increments of 10. Each file in the set contained 189,000,000 four-byte integers (~ 720MB). The test results are shown in Table 2. For a given number of files, only the best result is shown among the results with different distribution of worker processes on the nodes of the cluster (in Table 2 *nnode* is the number of nodes, *ppn* is the number of simultaneously running tasks on one node). The number of nodes ranged from 2 to 19, while the number of processes per node was fixed to 2. This cluster configuration corresponds to a small enterprise desktop grid. The speedup was estimated based on the average execution time of prepare task (55.7468 seconds) and play task (6.70886 seconds) when sorting a set of 20 files in the configuration *nnode* = 1, *ppn* = 1. Thus, the sequential execution time in seconds for processing M files was estimated by the formula $55.7468 \times M + 6.70886 \times M \times (M - 1)/2$.

Experiments show that it is possible to achieve significant speedup while sorting sets of 30 files. The best result was ~5 times speedup and ~1 hour reduction of computing time. As the number of files increases, the network was overloaded and the speedup dropped. However, we sorted 100 files with the speedup of 3.23. Due to the longer sorting time, the absolute reduction in time reached ~7 hours, which is a significant result.

We conducted a simulation experiment to assess how fast sorting could be performed with our ART skeleton, if we have no loss of performance due to the network overload. In the experiment the time of *prepare* task was also taken to be 55.7468 seconds, and the time of *play* task was taken to be 6.70886 seconds of model time. Simulations were computed with required number of workers (for example, 50 workers for 100 files) for the OPTIM sort tournament. The speedup was evaluated similarly to the experiments presented in Table 2. The results of the simulation experiment are shown in Table 3.

Table 2. The dependence of the sorting time on the number of files.

Number of files	Nnode	ppn	Sorting time, s	Speedup
10	2	2	288.434	2.97
20	4	2	457.772	5.22
30	7	2	911.613	5.03
40	7	2	1679.97	4.44
50	13	2	2974.02	3.70
60	11	2	3962.42	3.84
70	13	2	5605.15	3.58
80	15	2	7368.07	3.48
90	17	2	9548.26	3.33
100	19	2	12000.3	3.23

Table 3. The dependence of the sorting time on the number of files (simulation experiment).

Number of files	Sorting time, s	Speedup
10	149.671	5.74171
20	250.304	9.54688
30	350.937	13.0814
40	451.57	16.5263
50	552.202	19.9305
60	652.835	23.3129
70	760.177	26.4467
80	854.101	30.043
90	961.443	33.1649
100	1055.37	36.7489

From Table 3 it can be seen that for a small data sets up to 30 files, the actual speedup (in Table 2) is only 2 .. 2.5 times less than theoretical speedup (in Table 3), but for larger data sets the difference reaches 10 times or even more. This suggests that there is a room for further optimization. For example, one can use caching and/or direct file transfer between worker processes.

We also set up a testbed for testing the orchestrator application in the desktop grid running the Everest platform. The tuning and testing of its functionality was carried out similarly to [17], except that the ART skeleton with the OPTIM sort tournament was used in the orchestrator.

7 Discussion

Let's discuss some particularities of our approach. The article presents a sorting method that resembles bubble sorting (see the algorithm (1)), except that it uses blocks and is parallel. This similarity leads to some loss of performance, but makes sorting applicable to desktop grids.

Both the TRIV and OPTIM sorting algorithms correspond to the ART skeleton. They differ only in DAG diameters. The OPTIM sorting has less diameter, thus potentially more parallel. The advantage of TRIV sorting over OPTIM sorting is that it takes up less memory. The TRIV sorting has linear memory complexity by block number (see [17] for details).

The difference between our work and [18] is that the AllPairs skeleton implies task independence and is performance oriented. For example, the ART skeleton prohibits simultaneous submission of (1,2) and (2,3) tasks, (1,2) and (1,3) tasks, etc. The problem of performance optimization, solved in AllPairs, is also of interest to our skeleton, but is not covered in this article.

The scope of application of our method includes the scope of the AllPairs skeleton. In addition, the method can be used to solve problems in which the processing of a pair implies write access to the data area associated with the element of the pair. For example, we plan to use Everest platform to determine the frequency of words in Twitter on an enterprise desktop grid during periods of it inactivity.

The MapReduce [19] skeleton is beyond the scope of our work because it is based on peer-to-peer communication between compute nodes. On the contrary, desktop grid systems, namely systems built on the Everest platform, support the star topology.

There are a number of grid systems that use static DAGs of tasks, for example DAGman meta scheduler in the CondorHPC (https://research.cs.wisc.edu/htcondor/). Using dynamic tasks helps us to manage larger graphs potentially with millions of tasks. This feature was demonstrated in the experiments and shown in Table 1.

8 Conclusions

A method for the development of algorithmic skeletons for automating computations in enterprise desktop grids based on a variant of the actor model of calculations is proposed. The method was successfully applied to the development of the skeleton called asynchronous round-robin tournament. The practical use of the asynchronous round-robin tournament skeleton to build the orchestrator of a grid system was demonstrated in solving the block sorting problem.

References

1. González-Vélez, H., & Leyton, M.: A survey of algorithmic skeleton frameworks: high-level structured parallel programming enablers. Software: Practice and Experience Vol. 40, Issue 12, pp. 1135-1160. John Wiley & Sons, Inc. New York, NY, USA (2010)
2. Volkov S., Sukhoroslov O.: Running Parameter Sweep Applications on Everest Cloud Platform. Computer Research and Modeling, Vol. 7, No. 3, pp. 601-606. Institute of Computer Science, Izhevsk, Russia (2015)
3. Schlinker, B., Mysore, R.N.: Condor: Better topologies through declarative design. In: Schlinker, B., Mysore, R.N., (eds.) SIGCOMM 2015 - Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication, pp. 449-463. Association for Computing Machinery, Inc, London (2015)

4. Anderson, D.P.: BOINC: A system for public-resource computing and storage. Proceedings - IEEE/ACM International Workshop on Grid Computing, pp. 4-10. IEEE, Pittsburgh, PA, USA (2004)
5. Fedak, G., Germain, C., Neri, V., Cappello, F. XtremWeb: A generic global computing system. Proceedings - 1st IEEE/ACM International Symposium on Cluster Computing and the Grid, CCGrid 2001, No. 923246, pp. 582-587. IEEE, Brisbane, Queensland, Australia, Australia (2001)
6. Andrade, N., Cirne, W., Brasileiro, F., Roisenberg, P. OurGrid: An approach to easily assemble grids with equitable resource sharing. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), No. 2862, pp. 61-86. Springer, Berlin, Heidelberg (2003)
7. Afanasyev A.P., Lovas R.: Increasing the computing power of distributed systems with the help of grid systems from personal computers. . In: Afanasyev A.P., Lovas R., (eds.) Proceedings of the conference "Parallel Computational Technologies (PCT'2011)", pp. 6-14. Publishing Centre NRU, Chelyabinsk (2011)
8. Cérin, C., Fedak, G.: Desktop grid computing. CRC Press, Paris (2012)
9. Choi, S., Kim, H.: Characterizing and classifying desktop grid. In: Choi, S., Kim, H., (eds.) Proceedings - Seventh IEEE International Symposium on Cluster Computing and the Grid, CCGrid. No. 4215446, pp. 743-748. IEEE, Rio De Janeiro, Brazil (2007)
10. Ivashko E.: Enterprise Desktop Grids/ CEUR Workshop Proceedings, 1502, pp. 16-21. CEUR-WS, Dubna (2015)
11. Mazalov, V.V., Nikitina, N.N., Ivashko, E.E.: Task scheduling in a desktop grid to minimize the server load. International Conference on Parallel Computing Technologies, pp. 273-278 (2015). doi: 10.1007/978-3-319-21909-7_27
12. De Koster, J., Van Cutsem, T., De Meuter, W.: 43 years of actors: A taxonomy of actor models and their key properties. AGERE 2016 - Proceedings of the 6th International Workshop on Programming Based on Actors, Agents, and Decentralized Control, co-located with SPLASH, pp. 31-40. Association for Computing Machinery, Inc, New York (2016)
13. Senger, Hermes & da Silva, Fabrício.: Bounds on the Scalability of Bag-of-Tasks Applications Running on Master-Slave Platforms. Parallel Processing Letters. vol. 22, No. 2. World Scientific, USA (2012)
14. Farkas, Z., Kacsuk, P., Balaton, Z., Gombás, G.: Interoperability of BOINC and EGEE. Future Generation Computer Systems, vol. 26, is. 8, pp. 1092-1103 (2010)
15. Afanasiev, A.P., Bychkov, I.V., Zaikin, O.S.: Concept of a multitask grid system with a flexible allocation of idle computational resources of supercomputers. In: Afanasiev, A.P., Bychkov, I.V., Zaikin, O.S., (eds.) Journal of Computer and Systems Sciences International, vol. 56, is.4, pp. 701-707 (2017). doi: 10.1134/S1064230717040025
16. Sukhoroslov, O., Volkov, S., Afanasiev, A. A.: Web-Based Platform for Publication and Distributed Execution of Computing Applications. 14th International Symposium on Parallel and Distributed Computing (ISPDC), pp. 175-184, IEEE (2015)
17. Vostokin, S.V., Sukhoroslov, O.V., Bobyleva, I.V., Popov, S.N.: Implementing computations with dynamic task dependencies in the desktop grid environment using Everest and Templet Web. CEUR Workshop Proceedings, Volume 2267, pp. 271-275. CEUR-WS, Dubna (2018)
18. Moretti, C., Bulosan, J., Thain, D., Flynn, P. J.: All-pairs: An abstraction for data-intensive cloud computing. 2008 IEEE International Symposium on Parallel and Distributed Processing, pp. 1-11, IEEE (2008)
19. Dean, J., Ghemawat, S.: MapReduce: simplified data processing on large clusters. 51(1), pp.107-113, Communications of the ACM (2008)