Bull. Nov. Comp. Center, Comp. Science, 46 (2022), 43–55

Preliminary results on fault tolerance support in LuNA system*

V. Perepelkin, V. Malkhanov, V. Zakirov

Abstract. Fault tolerance support automation is a relevant problem, because it is both demanded for large scale computations and hard to implement manually. General approaches exist, but they lack efficiency which is required in high performance computing as compared to particular approaches, which exploit peculiarities of subject domain and applied algorithm in order to reduce overhead on fault tolerance support. Usage of parallel programming systems, such as LuNA, opens possibility to automatically or semiautomatically implement fault tolerance support in constructed programs which are more efficient than general approaches due to exploitation of peculiarities of the computational model on which the system is based. The problem of automated fault tolerance support in LuNA system is considered in the work. Some preliminary results are presented, such as checkpointing technique adaptation and the problem analysis.

Introduction

Development of numerical parallel programs for high performance computers is troublesome due to the necessity to provide non-functional properties of the programs. Such properties include high efficiency, low power consumption, memory or network efficiency, dynamic load balancing etc. One of such properties is fault tolerance. Fault tolerance is the ability of a program to keep running in case of hardware faults of particular kind. In scientific computations fault tolerance is relevant for large scale supercomputers, comprising thousands and millions of cores, because the larger the supercomputer is, the higher is the hardware fault probability. Absence of fault tolerance support means that in case of hardware fault the whole computation has to be rerun from the very beginning. Thus, the longer it takes to compute, the less is the chance for computation completing in a reasonable time, not to mention the resources waste for failed executions.

Fault tolerance support is hard to implement because of different reasons. Firstly, there is no universal efficient solution to the problem. Such general approaches as periodic memory dumping imply the overhead which penalize the high performance computing, and is much higher, than some particular solution could take. Secondly, development and implementation of a particular fault tolerance tool is often not trivial and requires specific knowledge

*This work was carried out under state contract with ICMMG SB RAS 0251-2021-0005.

and skills in system parallel programming. Such skills and knowledge are not commonly possessed by supercomputer users.

One of the ways to overcome the difficulties in provision of fault tolerance support is programming automation. Many parallel programming systems and frameworks suggest the automatic provision of fault tolerance support [1–3]. Usage of high level programming languages gives the system a much deeper understanding of how the program performs, and thus the fault tolerance often can be implemented more efficiently than in the general case. For example, if there is a task-based computational model, where each task is a serializable object, then the fault tolerance can be implemented by serialization of tasks, rather than dumping the whole memory. Different programming systems and languages are based on different computational models, which provide different capabilities for automatic provision of fault tolerance.

This paper is devoted to the problem of automatic provision of fault tolerance in LuNA system. LuNA (Language for Numerical Algorithms) is a language and a system for automatic construction of numerical parallel programs for multicomputers (distributed memory computers). LuNA is an academic project of ICMMG SB RAS [4]. It is based on the theory of structured synthesis of parallel programs on the basis of computational models [5], which suggests rich possibilities in automation of non-functional properties provision. In this paper we describe how the fault tolerance can be implemented in LuNA and demonstrate some preliminary results in implementation of this kind of support.

The rest of the paper is organized as follows. In Section 1 a brief description of LuNA model is given, as well as the discussion of the basic idea of automatically supporting fault tolerance. Section 2 concerns the implementation of particular checkpointing technique. Section 3 concerns the checkpoint format. Section 5 is devoted to the problem of dynamic addition and removal of computing nodes using the means of MPI (Message Passing Interface). The paper ends with a conclusion and a related work review.

1. LuNA system and fault tolerance

In LuNA system, a program is explicitly represented as a set of triplets both in the input language and in run time. Each triplet has the form $\langle in, mod, out\rangle$, where *in* and *out* are finite sets of input and output arguments, correspondingly, and *mod* is a sequential subroutine with no side effects, capable of computing the output arguments from the input ones. Such triplet is called computational fragment (CF), and arguments are pieces of immutable data (called data fragments — DF). DFs and CFs are coarse-grained objects, rather than single numbers and operations on them. For example, a DF can be a submatrix or a mesh domain, and a CF can be

a subroutine to process such data. If a DF is an input argument of a CF and an output argument of another CF, then the second CF is information dependent on the first one.

LuNA program is a description of the set of CFs and DFs. Its execution consists in the following. At any given moment a set of CFs to be executed and a set of computed DFs are distributed over computing nodes of a multicomputer. LuNA system selects a CF and checks whether all its input DFs are computed. Then LuNA transfers the input DFs to a particular computing node and invokes corresponding subroutine. The subroutine execution produces the values of output DFs, which are then distributed to computing nodes. Multiple CFs can be executed in parallel on different computing nodes and cores.

The basic idea of the checkpointing mechanism in LuNA is straightforward. The flow of a program execution is fully defined by the sets of DFs and CFs, which exist in the system at any given moment. Thus, if all the DFs and CFs are saved into files, then they can be restored later to continue the program execution. All DFs are immutable, so data modification is not a problem. All CFs are side-effect free, therefore a CF can be executed on any node (or after the checkpoint has been restored) with exactly the same output values computed.

So, saving a checkpoint is as simple as pausing execution, waiting for currently running CFs to finish execution and then dumping all the CFs and DFs into a file. The DFs dumping is possible because all DFs in LuNA are serializable. This is necessary for the system to be able to transfer them over the network from the computing node (where a DF was produced) to the nodes where it can be stored and consumed by other CFs. Serialization of a CF is also not a problem, because a CF is actually a triplet descriptor, which LuNA is able to serialize when transferring CFs over the network.

Checkpointing in LuNA is different from checkpointing in conventional parallel programs, such as MPI programs. The main reason is because the LuNA program is not bound to any hardware resources. Any CF or DF can be stored and restored at any node with no influence on the computed values. The only exception is the execution of CFs subroutine, which is a conventional subroutine call. Therefore, CFs are not saved into a checkpoint if they are already being executed. Before saving a checkpoint, executing CFs should be either aborted or waited for completion. Absence of binding to hardware resources makes it possible, in particular, to restore a checkpoint on a different multicomputer, or on a different number of computing nodes, which is uncommon for regular checkpointing.

Another peculiarity of checkpointing in LuNA is the ability to dump only necessary part of data, not the whole memory. This is possible because all the application data are explicitly represented as DFs. Moreover, when dumping a conventional process, all the data must be put to exactly the

same addresses, because otherwise the pointers may become invalid. There are no pointers in LuNA, thus any DF can be loaded into a different memory location (or even into the memory of another computing node) with no risk of invalidating the execution. Another problem with conventional programs is open file descriptors and other local resources, which need special handling and cannot be recovered automatically. For LuNA, CF subroutines have no side effects, thus this problem is not raised at all.

In conventional programming, checkpoint saving is often made not at random time point, but at particular steps of execution. For example, when an iteration of a numerical method is performed and the intermediate data are cleaned up. This is useful to reduce the checkpoint size (and saving time) and to simplify the restoring routine. Also this allows to control the checkpointing frequency. In LuNA, more or less similar effects can be obtained automatically. In particular, since all the DFs are "visible" to the system, it can watch the total amount of data. Once the total DFs size drops to some low level, the checkpoint saving event can be triggered. The frequency of saving checkpoints can be selected by the system automatically, e.g., for a given "save time" to computing time ratio.

Another concern for considering here is the ability to resume the program execution from a checkpoint using a different run-time system. Basically, LuNA consists of a translator and a distributed run-time system. But in practice, to achieve the high performance of LuNA programs execution, multiple specialized run-time systems exist. If an application belongs to a particular class of applications (e.g., dense or sparse linear algebra application), a specialized run-time system is used to increase the execution efficiency [6,7]. Checkpointing mechanism brings here an interesting possibility to switch between the run-time systems during LuNA-program execution: a checkpoint is saved by one run-time system while it is restored by another one. This technique can be used to dynamically adapt to peculiarities of an application. However, implementation of such a portability requires a common checkpoint format, compatible with multiple run-time systems.

To implement checkpointing in LuNA, the following problems were to be concerned. Firstly, this is "pausing" and "resuming" the system required for coherent run of the program. This means all executing CFs should be completed, and all network messages should be delivered before saving a checkpoint. Secondly, the checkpoint format has to be defined. An additional requirement here is the portability across different run-time systems. Another related problem here is changing the number of nodes for operating the LuNA. This is necessary if checkpointing is used to cope with hardware faults. Therefore there should be an ability to continue execution on a reduced set of nodes, as well as to dynamically add new nodes to computation process for replacing the failed ones. The next sections of the paper concern these questions.

It is worth mentioning, that stopping the whole run-time system in a barrier manner is not the only option. Another interesting way to do this is to explicitly mark some subset of CFs and DFs as candidates to comprise a checkpoint. Multiple subsets can be marked as different checkpoint candidates. Once the system needs saving a checkpoint, it just waits for CFs and DFs from the next candidate start appearing and serializes them into a checkpoint. This method allows eliminating any barrier-like synchronization or system pausing and making a consistent checkpoint on-the-fly. And the checkpoint candidate subsets can be selected in such way that the amount of data to be saved is not too big. Marking the candidates can be performed manually (e.g. by annotating the application source code) or automatically (if a suitable algorithm is provided).

2. Stopping and resuming run-time system

Once creation of checkpoint has been triggered, the run-time system has to be paused. For that, two subsystems are to be stopped: the CF execution subsystem and the communication subsystem. If dynamic load balancing (or another system module which can affect checkpointing) is running, it must be stopped. However, this issue is out of the scope of the paper.

Stopping the CF execution subsystem. In the run-time system, CFs can exist in three states: waiting for an external event (such as receiving a DF or end of migration to other computing nodes), waiting for a thread in a thread pool to perform the next action or running the current action by a thread from the thread pool. So, the first thing to be done is preventing threads from the thread pool to accept the CFs for execution. Instead, all those CFs should be put to a buffer. A corresponding flag is set to force such behavior. The second thing is to wait until all currently running CFs will complete their execution.

Stopping the communication subsystem. It is necessary to make sure no migrating CFs, DFs or other system objects keep migrating over the network before saving the checkpoint (the whole list of objects is presented in Section 4). Once all CFs are paused, no new migration is possible. After that all the messages currently being transferred have to be delivered to the destinations. The modified Dijkstra-Scholten algorithm [8] is used for this purpose.

Resuming work. After stopping the two subsystems, the checkpoint can be saved, and then the execution can be resumed. To resume the execution, the pause flag has to be cleared and all the buffered CFs have to be re-submitted to the thread pool of the CF execution subsystem.

3. Checkpoint format

A checkpoint must store all the information necessary to resume execution from this point. The format should allow to restore execution not only for the run-time system (which saved the checkpoint), but also for other run-time systems. A particular case is when the checkpoint is restored by the same run-time system, but with different configuration (e.g., with dynamic load balancing enabled while it was disabled before the checkpoint was saved). This allows changing the configuration of the run-time system unsuitable for altering while a program is running.

Checkpoints are saved in a binary format in a structured form. The format is described below starting with primitives.

String format:

```
[length: int32]
[string: byte]...
```

Such notation is intuitive and denotes a binary object, which starts with a 4-byte integer. It encodes the length of the string, followed by a corresponding number of bytes in the string body. Note here that no terminating symbol is used in this format. Note: ellipsis means there is a sequence of such records. Here it means that multiple bytes represent the string body.

ID denotes an identifier of CF or DF. Here ID is a system record, which is a sequence of a number of integers. ID format:

```
[ids length: int32]
[element: int32]...
```

Here a number of integers in the ID is stored as a 4-byte integer, and then the corresponding number of 4-byte integers are stored, each representing an element of the ID.

Data Fragment (DF) is basically a memory region and an associated LuNA type (int, real, string or value). DF format:

```
[data size: int32]
[data: byte]...
[type: byte]
```

Computational fragment (CF) is a system record, which consists of multiple fields:

1. *next block*: an integer denoting the current step of the CF execution;
2. *args*: an array of DFs, which were defined at the CF initialization;
3. *ids*: an array of references accessible by the CF;

4. *store*: a dictionary of DFs with identifiers, which are either input or output DFs of the CF.

CF format:

```
[next block: int32]
[args length: int32]
[args: DF]...
[ids length: int32]
[ids: ID]...
[store length: int32]
[ID, DF]...
```

Note: the last sequence is the sequence of pairs of types ID and DF, representing key-value pairs of the store dictionary.

The whole checkpoint is represented as a composition of checkpoints for each node. Node Checkpoint (NC) consists of 5 elements:

1. *CFs*: array of CFs;
2. *posts*: array of DFs;
3. *requests*: array of requests (CF requests an input DF);
4. *waiters*: array of CFs waiting for responses;
5. *destroys*: DF deletion requests.

Request format:

```
[dfid: ID]
[requesters length: int32]
[cf_idx: int32, node: int32]...
```

Here *requesters* is the array of CFs, which have sent requests, *cf_idx* is the index of the CF in the CFs array of the NC, *node* is the rank of the request origin node.

Waiter format:

```
[dfid: ID]
[waiting CFs length: int32]
[cf_idx: int32]...
```

Here *dfid* is the identifier of the requested DF. It is followed by the list of CFs which are waiting for response.

NC format:

```
[CFs length: int32]
[cf: CF]...
[posts length: int32]
[id: ID, req_count: int32, df: DF]...
[requests length: int32]
[Request]...
[waiters length: int32]
[Waiter]...
[destroys length: int32]
[ID]...
```

Here *req_count* is a counter of remaining usages of DF. When the counter reaches zero the DF is deleted.

Checkpoint format:

```
[format version: int32]
[number of nodes: int32]
[node checkpoint size: int32]...
[node checkpoint: NC]...
```

CF storage problem. In LuNA run-time system, CFs have no persistent unique identifiers. All the identification was carried out using pointers to local objects, which represent CFs, which were used in callbacks and remote callbacks. Note: a callback is a pointer to a lambda function ($std :: function$) that handles an event; a remote callback is a pointer to a lambda function on another computing node. Running a remote callback is carried out as passing a message to the node and calling the lambda there. Such technique is efficient in terms of callback lookup time (which is of constant complexity), but the problem is that serialization and identification of CFs becomes not possible as is. That is why the CFs in a checkpoint are identified by their index in the CFs array (in NC). And where a CF has to be referred to, the index is used. Such approach implied some additional logic in the run-time system.

4. Dynamic nodes reallocation with MPI

Normally, parallel programs are run on a static number of computing nodes, e.g., on a computing cluster via a task queue system, such as OpenPBS [9].

However, sometimes there is a need to dynamically add more nodes to computation process, or to reduce the number of used nodes. For example, this is a must if an application consists of different computational steps requiring different amount of resources or if an application can dynamically

scale up to the currently available resources. In the context of fault tolerance, dynamic nodes reallocation is needed in two cases: 1) if there is a hardware failure on a computing node, so the node has to be excluded from the set of used nodes; 2) if a number of nodes are added to compensate the reduction of nodes due to hardware faults. No matter what is the reason, the dynamic computing nodes reallocation support is a relevant problem. Message Passing Interface (MPI) is the most widely used interface for implementation of high performance communications in scientific computations. Starting from version 2, MPI possesses some means for support such reallocation. The Section describes an attempt to use the means for dynamic nodes reallocation in LuNA (which uses MPI to perform communications).

Let us concern two simple examples demonstrating some reallocation benefits.

Example 1. Suppose there are 16 computing nodes available on a cluster with two programs running on it, each occupying 8 nodes. Assume the first program will run for about two days, and the second one — for one day only. Dynamic nodes reallocation could make use of the released 8 nodes after a day of computations for the first program, speeding up its computation rate.

Example 2. Suppose a low-priority task is running on the whole cluster. To start a new high-priority task, we could release the necessary amount of resources from the first task and use them to run the second task.

The main idea of dynamic nodes reallocation is to run another instance of the distributed run-time system in a special mode, which consists in connecting to the already running instance of the distributed run-time system. By that time the first instance has to be listening for such connections at some address.

The implementation of this feature is based on MPI-2 standard, which includes dynamic process generation and process management. The key aspect is the ability of the MPI process to participate in the creation of new MPI processes or to establish communication with MPI processes that were started separately. The MPI-2 specification describes three main interfaces through which MPI processes can dynamically establish the communication:

- MPI_Comm_spawn — allows a child processes to be spawned.
- MPI_Comm_accept / MPI_Comm_connect — allows to establish a connection between two independently running applications. These operations are blocking and collective.
- MPI_Comm_join — allows to combine two processes connected by a socket (intended only for environments that support the Berkeley sockets interface).

MPI_Comm_accept / MPI_Comm_connect was chosen as the most suitable interface for the following reasons.

Firstly, using MPI_Comm_spawn would require to initiate somehow the generation of new processes, moreover, it would also be necessary to pass a host file containing the hosts for launching the new processes. This option does not suit clusters with a task manager and this is a common case for scientific computations.

Secondly, MPI_Comm_join allows to combine only two processes when it is necessary to combine the groups of processes. Moreover, this interface supports only the Berkeley sockets.

Thirdly, MPI_Comm_accept / MPI_Comm_connect allows to re-run the required program on the necessary nodes, which then merge with the original program, which is the most convenient way.

Thus, it is possible to describe the algorithm of resource capture and release as follows:

1. Initiate the connection of two independently running programs. There are several options:
   (a) Handling UNIX/Linux process signals.
   (b) Listening to a socket in a separate thread.
   (c) Running MPI_Comm_accept in a separate thread on each process.
2. Send to all processes a tag indicating the start of resource capture or release.
3. Stop the execution system.
4. Establish a connection using MPI_Comm_accept / MPI_Comm_connect and update the main communicator, as well as the total number of processes and the ranks of each of them.
   (a) In case of resource capture, the process groups can be combined into a common communicator using MPI_Intercomm_merge.
   (b) If resources are released, MPI_Comm_split can be applied making different colors for released processes.
5. Redistribute tasks and data between processes.
6. Resume the execution system.

Stopping the system is described previously in the Section 2.

Let us consider redistribution of tasks and data between processes. Once the number of computing nodes has changed, four kind of objects are to be redistributed:

- *CFs* — a list of CFs on the node;
- *Posts* — a list of DFs being stored on the node;

- *Requests* — a list of requests of DFs by some CFs;
- *Destroys* — a list of DF deletion requests.

All four kinds of objects are distributed to computing nodes and to the node allocated for distribution are defined by a dynamic function called *locator*. This function can be considered as a function of the following form: $locator(id, size) => node$. Here *size* is the number of computing nodes. Since *size* has changed, some object's locations can alter as well. That's why each object has to be looped through and have its location recomputed. If the location has changed, the object has to migrate to the corresponding node.

Implementation of dynamic nodes reallocation using MPI faced some technical problems associated with MPI implementations (we considered MPICH and OpenMPI).

1. MPI_Comm_accept / MPI_Comm_connect does not work in OpenMPI when connecting two independently running programs.

2. MPI_Comm_accept causes all MPI functions to hang if initialized with MPI_THREAD_MULTIPLE. This problem is reproduced in MPICH. Because of this, it will not be possible to run MPI_Comm_accept in a separate background thread on each process to initiate a connection (which would be the most straightforward way to do it).

3. In the case of resource release after the communicators have been separated using the MPI_Comm_split, it is not possible to immediately stop the freed processes using MPI_Abort. According to the MPICH and OpenMPI documentation, the command MPI_Abort does not guarantee the continuation of the work of the remaining MPI processes. That is, this function terminates all processes, even those that do not relate to the transmitted communicator. It means that the real hardware fault shall cause the whole MPI application to abort. Nevertheless, we hope the MPI will evolve to the capability of sustainable running while hardware faults, because the demand of this feature tends to increase for the large-scale supercomputers.

Related work and conclusion

The software fault tolerance in distributed parallel computing has been studied for decades [10]. Many particular and general solutions have been developed. For example, a fault-tolerant MPI [11] is an attempt to implement general transparent fault tolerance under the MPI interface. Besides checkpointing, replication [12] is another common technique to support fault tolerance. Both techniques can be used together [13]. From the user's perspective, fault tolerance may take a form of rollback mechanism [14]. Many other techniques and approaches were introduced [15–23].

Although many results were introduced, no general solution is expected to be found. Particular solutions may be efficient, but hard for manual implementation or require a particular programming framework or a system. Of great importance is automated provision of fault tolerance. It is based on creating a system that predicts the kind of an application and supplies it with a suitable heuristic (or specialized) fault tolerance mechanism.

LuNA system is a system designed for analysis of running application and providing build-in automatic fault tolerance. In this paper some preliminary results are presented on automatic fault tolerance support in LuNA system. In the future research, the proposed technique has to be tested and adapted for particular application and execution conditions.

References

[1] Dean J., Ghemawat S. MapReduce: simplified data processing on large clusters // Commun. ACM. — 2008. — Vol. 51, No. 1. — P. 107–113.

[2] White T. Hadoop: the Definitive Guide. — O'Reilly, 2012.

[3] Isard M., Budiu M., Yu Y., et al. Dryad: distributed data-parallel programs from sequential building blocks // EuroSys '07: Proc. of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007. — New York, NY, USA: ACM, 2007. — P. 59–72.

[4] Malyshkin V.E., Perepelkin V.A. LuNA fragmented programming system, main functions and peculiarities of run-time subsystem // Parallel Computing Technologies. 11th International Conference, PaCT 2011, Proc. — Springer, 2011. — P. 53–61. — (LNCS; 6873).

[5] Valkovskii V., Malyshkin V. Parallel Program Synthesis on the Basis of Computational Models. — Novosibirsk: Nauka, 1988 (In Russian).

[6] Belyaev N., Perepelkin V. High-efficiency specialized support for dense linear algebra arithmetic in LuNA system // Parallel Computing Technologies. PaCT 2021. — Springer, Cham, 2021. — P. 143–150. — (LNCS; 12942).

[7] Belyaev N.A. Automatic construction of high performance parallel programs for dense linear algebra applications in LuNA system // Problems of Informatics. — 2022. — No. 3. — P. 46–60.

[8] Dijkstra E.W., Scholten C.S. Termination detection for diffusing computations // Information Processing Letters. — 1980. — Vol. 11, No. 1. — P. 1–4.

[9] OpenPBS Open Source Project web page. — https://www.openpbs.org (Accessed 29.10.2022)

[10] Egwutuoha I.P., Levy D., Selic B. et al. A survey of fault tolerance mechanisms and checkpoint/restart implementations for high performance computing systems // J. Supercomput. — 2013. — Vol. 65. — P. 1302–1326.

[11] Jung H., Shin D., Kim H., Lee H.Y. Design and Implementation of Multiple FaultTolerant MPI over Myrinet (M3). — Seattle, Washington, USA: ACM, Nov 2005. — (SC-05; 1218).

[12] Walters J., Chaudhary V. Replication-based fault tolerance for MPI applications // IEEE Transactions on Parallel and Distributed Systems. — 2009. — Vol. 20, No.7. — P. 997–1010.

[13] Chtepen M., Claeys F.H.A., Dhoedt B., et al. Adaptive task checkpointing and replication: toward efficient fault-tolerant grids // IEEE Transactions on Parallel and Distributed Systems. — 2009. — Vol. 20, No. 2. — P. 180–190.

[14] Jafar S., Krings A., Gautier T. Flexible rollback recovery in dynamic heterogeneous grid computing // IEEE Transactions On Dependable and Secure Computing. — 2009. — Vol. 6, No. 1. — P. 32–44.

[15] Yang X., Du Y., Fu P.W., Jia J. FTPA Supporting fault-tolerant parallel computing through parallel recomputing // IEEE Transactions on Parallel and Distributed Systems. — 2009. — Vol. 20, No. 10. — P. 1471–1486.

[16] Gorender S., Raynal M. An adaptive programming model for fault-tolerant distributed computing // IEEE Transactions On Dependable And Secure Computing. — 2007. — Vol. 4, No. 1. — P. 18–31.

[17] Luckow A., Schnor B. Adaptive checkpoint replication for supporting the fault tolerance of applications in the grid // Seventh IEEE International Symposium on Network Computing and Applications, 2008. — 2008. — P. 299–306.

[18] Bouteiller B., Cappello F., Herault T., et al. MPICH-V2: a fault tolerant MPI for volatile nodes based on pessimistic sender based message logging // SC '03: Proc. of the 2003 ACM/IEEE Conference on Supercomputing. — 2003. — P. 25.

[19] Leon B., Gomez-Sanchez P., Franco D., et al. Analysis of Checkpoint I/O Behavior // Computational Science – ICCS 2020. — 2020. — P. 191–205.

[20] Bajunaid N., Menasce D.A. Efficient modeling and optimizing of checkpointing in concurrent component-based software systems // J. Systems and Software. — 2018. — Vol. 139. — P. 1–13.

[21] Han L., Le Fevre V., Canon L-C., et al. A generic approach to scheduling and checkpointing work flows // Intern. J. High Performance Computing Applications. — 2019. — Vol. 33, No. 6. — P. 1255–1274.

[22] Garg R., Praveen K. A review of checkpointing based fault tolerance techniques in mobile distributed systems // Intern. J. on Computer Science and Engineering. — 2010. — Vol. 2, No. 4. — P. 1052–1063.

[23] Riesen R., Ferreira K., Da Silva D., et al. Alleviating scalability issues of checkpointing protocols // CS '12: Proc. Intern. Conf. for High Performance Computing, Networking, Storage and Analysis. — 2012. — P. 1–11.