Subordination: Providing resilience to simultaneous failure of multiple cluster nodes

(WORK-IN-PROGRESS)

Ivan Gankevich Yuri Tipikin Vladimir Korkhov
Dept. of Computer Modeling and Multiprocessor Systems
Saint Petersburg State University
Saint-Petersburg, Russia
Email: i.gankevich@spbu.ru, y.tipikin@spbu.ru, v.korkhov@spbu.ru

Abstract—In this paper we describe a new framework for creating distributed programmes which are resilient to cluster node failures. Our main goal is to create a simple and reliable model, that ensures continuous execution of parallel programmes without creation of checkpoints, memory dumps and other I/O intensive activities. To achieve this we introduce multi-layered system architecture, each layer of which consists of unified entities organised into hierarchies, and then show how this system handles different node failure scenarios. We benchmark our system on the example of real-world HPC application on both physical and virtual clusters. The results of the experiments show that our approach has low overhead and scales to a large number of cluster nodes.

I. Introduction

In large scale cluster environments node failures are common. In general this does not lead to global cluster malfunction, but it has huge impact on job running on faulty resources. Classical MPI programmes fail, if any one of cluster nodes on which the programme is running fails. Today existing solutions mainly focus on making application checkpoints, but with increasing size of supercomputers and HPC clusters this approach becomes less efficient. Our approach to make cluster computations reliable and efficient is to use special framework focused on structuring parallel programme in strict hierarchy of parallel and sequential parts. Using different fault tolerant scenarios based on hierarchy interactions, this framework provides continuous execution of a parallel programme in case of hardware errors or electricity outages.

The aim of the research reported here is to investigate how continuous execution of parallel programmes in the presence of node failures can be provided on the level of software framework. This framework replaces both MPI library and batch job scheduler by introducing the notion of a kernel — a unit of work which can be copied between cluster nodes and re-executed any number of times — if it is required to provide resilience to node failures. In this paper we present an algorithm that guarantees continuous execution of a parallel programme upon failure of all nodes except one. This algorithm is based on the one developed in previous papers [1], [2], where only one node failure at a time is guaranteed to not interrupt programme execution.

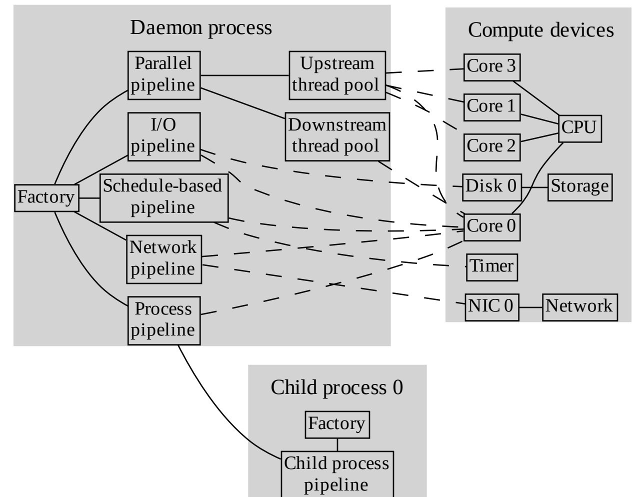

Fig. 1. Mapping of parent and child process pipelines to compute devices. Solid lines denote aggregation, dashed lines denote mapping between logical and physical entities.

In this paper failure detection methods are not studied, and node failure is assumed if the corresponding network connection abruptly closes. Node failure handling, provided by our algorithm, is transparent for a programmer: there is no need explicitly specify which kernels should be copied to other cluster nodes. However, its implementation cannot be used to provide fault tolerance to existing parallel programmes based on MPI or other libraries: the purpose of software framework developed here is to seamlessly provide fault tolerance for new parallel applications. If a failure is detected by some external programme, then removing this node from the cluster is as simple as killing the daemon process which is integral part of the framework.

II. System architecture

Our model of computer system has layered architecture (fig. 1):

a) Physical layer: Consists of nodes and direct/routed physical network links. On this layer full

network connectivity, i.e. an ability to send packet from one cluster node to any other, is assumed.

b) Daemon layer: Consists of daemon processes residing on cluster nodes and hierarchical (master/slave) logical links between them. Master and slave roles are dynamically assigned to daemon processes, i.e. any physical cluster node may become a master or a slave. Dynamic reassignment uses leader election algorithm that does not require periodic broadcasting of messages, and the role is derived from node's IP address. Detailed explanation of the algorithm is provided in [1]. Its strengths are scalability to a large number of nodes and low overhead, which are essential for large-scale high-performance computations, and its weakness is in artificial dependence of node's position in the hierarchy on its IP address, which may not desirable in virtual environments, where nodes' IP addresses may change without a notice.

The only purpose of daemon hierarchy is to provide load balancing and automatically reconfigurable logical tree hierarchy of cluster nodes. This hierarchy is used to distribute the load from the current node to its neighbours by simply iterating over all directly connected daemons. Upon reconfiguration due to node failure or due to new node joining the cluster, daemons exchange messages telling each other how many daemons are "behind" the corresponding link in the hierarchy. This information is used to distribute the load evenly, even if a parallel programme is launched on a slave node. In addition, this topology reduces the number of simultaneous connections, thus preventing network overload.

Load balancing is implemented as follows. When daemon A tries to become a subordinate of daemon B, it sends a message to a corresponding IP address telling how many daemons are already connected to it (including itself). If there are no connections, then it counts itself only. After all links between daemons in the cluster are established, every daemon has enough information to tell, how many nodes exist behind each link. If the link is between a slave and a master, and the slave wants to know, how many nodes are behind the master, then it simply subtracts the total number of nodes behind all of its slave nodes from the total number of nodes behind the master to get the correct amount. To distribute kernels across nodes we use simple round-robin algorithm, i.e. iterate over all links of the current daemon (including the link to its master) taking into account how many nodes are behind each link: the pointer advances to a next link, only when enough number of kernels are sent through the current link. That way even if an application is launched on a slave node in the bottom of the hierarchy, the kernels will be distributed evenly across all cluster nodes. A kernel can not be sent through the link, from which it was received.

The advantage of this approach is that it can be extended to include sophisticated logic into load distribution policy. Any metric, that is required to implement such policy, can be sent from the directly linked daemon during the link initialisation. As of now, only the number of nodes behind the link is sent. The disadvantage of the approach is that an update of the metric happens only when a change in the hierarchy occurs: if a metric changes periodically, then periodically sending update messages is also required for implementing the policy, and too frequent updates may consume considerable amount of network bandwidth. The other disadvantage is that when reconfiguration of the hierarchy occurs due to a node failure or a new node joining the cluster, the kernels that are already executed on the nodes are not taken into account in the load distribution, so frequent updates to the hierarchy may cause uneven load distribution (which, however, balances over time). Uneven load distribution does not cause node overload, since there is a kernel pool on each node that queues the kernels prior to execution.

Dynamic master/slave role distribution coupled with kernel distribution makes overall system architecture homogeneous within single cluster. On every node the same daemon is run, and no configuration is needed to make a hierarchy of daemons — it happens automatically.

c) Kernel layer: Consists of kernels and hierarchical (parent/child) logical links between them. The only purpose of kernel hierarchy is to provide fail over for kernels.

The framework provides classes and methods to simplify development of distributed applications and middleware. The focus is to make distributed application resilient to failures, i.e. make it fault tolerant and highly available, and do it transparently to a programmer. All classes are divided into two layers: the lower layer consists of classes for single node applications, and the upper layer consists of classes for applications that run on an arbitrary number of nodes. There are two kinds of tightly coupled entities in the framework — kernels and pipelines — which are used together to compose a programme.

Kernels implement control flow logic in theirs act and react methods and store the state of the current control flow branch. Domain-specific logic and state are implemented by a programmer. In act method some function is either sequentially computed or decomposed into subtasks (represented by another set of kernels) which are subsequently sent to a pipeline. In react method subordinate kernels that returned from the pipeline are processed by their parent. Calls to act and react methods are asynchronous and are made within threads spawned by a pipeline. For each kernel act is called only once, and for multiple kernels the calls are done in parallel to each other, whereas react method is called once for each subordinate kernel, and all the calls are made in the same thread to prevent race conditions (for different parent kernels different threads may be used).

Pipelines implement asynchronous calls to act and react, and try to make as many parallel calls as possible considering concurrency of the platform (no. of cores per node and no. of nodes in a cluster). A pipeline consists of a kernel pool, which contains all the subordinate kernels

sent by their parents, and a thread pool that processes kernels in accordance with rules outlined in the previous paragraph. A separate pipeline exists for each compute device: there are pipelines for parallel processing, schedule-based processing (periodic and delayed tasks), and a proxy pipeline for processing of kernels on other cluster nodes (see fig. 1).

In principle, kernels and pipelines machinery reflect the one of procedures and call stacks, with the advantage that kernel methods are called asynchronously and in parallel to each other. The stack, which ordinarily stores local variables, is modelled by fields of a kernel. The sequence of processor instructions before nested procedure calls is modelled by act method, and sequence of processor instructions after the calls is modelled by react method. The procedure calls themselves are modelled by constructing and sending subordinate kernels to the pipeline. Two methods are necessary because calls are asynchronous and one must wait before subordinate kernels complete their work. Pipelines allow circumventing active wait, and call correct kernel methods by analysing their internal state.

III. Resilience to multiple node failures

To disambiguate hierarchical links between daemon processes and kernels and to simplify the discussion, we will use the following naming conventions throughout the text. If the link is between two daemon processes, the relationship is master-slave. If the link is between two kernels, then the relationship is principal-subordinate (or parent-child). Two hierarchies are orthogonal to each other in a sense that no kernel may have a link to a daemon, and vice versa. Since daemon hierarchy is used to distribute the load on the cluster, kernel hierarchy is mapped onto it, and this mapping can be arbitrary. It is common to have principal kernel on a slave node with its subordinate kernels distributed evenly between all cluster nodes (including the node where the principal is located). Both hierarchies can be arbitrarily deep, but “shallow” ones are preferred for highly parallel programmes, as there are less number of hops when kernels are distributed between cluster nodes. Since there is one-to-one correspondence between daemons and cluster nodes, they are used interchangeably in the paper.

In our system a node is considered failed if the corresponding network connection is abruptly closed. Developing more sophisticated failure detection techniques is out of scope of this paper. For the purpose of studying recovery procedures upon node failure this simple approach is sufficient.

Consequently, any kernel which resided on the failed node is considered failed, and failure recovery procedure is triggered. Depending on the position of the kernel in kernel hierarchy recovery is carried out on the node where parent or one of the subordinate kernels resides. Recovery procedure for failed subordinate kernel is re-execution of this kernel on a healthy node, which is triggered automatically by the node where its parent kernel is located. If the subordinate communicates with other subordinates of the same parent kernel and one of them fails, all kernels as well as their parent are considered failed, and a copy of the parent is re-executed on a healthy node. If parent kernel fails, then its copy, which is sent along with every subordinate on other cluster nodes, is re-executed on the node where the first survived subordinate kernel resides. Kernel failure is detected only for kernels that are sent from one node to another (local kernels are not considered). A healthy node does not need to be a new one, any already loaded node will do: recovery does not overload the node, because each node has its own pool of kernels in which they wait before being executed by a pipeline.

When a kernel is sent to other node, its copy is saved in the outbound buffer (a list of kernels, that were sent to a particular node), from which it is removed only when the kernel returns to its parent. If the corresponding connection closes, all kernels from this buffer are retrieved and distributed across available nodes including the current node. The fail over algorithm is straightforward for a subordinate, but for a principal it is more involved. Whereas a subordinate is implicitly copied to another node as a consequence of load distribution, a principal is left on the one node. In order to implement resilience to a principal failure, one needs to copy it along with each of its subordinates to other nodes, and provide a rule to determine from which copy the principal is restored upon the node failure. The following paragraphs discuss this algorithm and the rule in detail.

A. Failure scenarios

The main purpose of the system is to provide continuous execution of kernels in the presence of node failures. There are three types of such failures.

1) Simultaneous failure of at most one node.
2) Simultaneous failure of more than one node but less than total number of nodes.
3) Simultaneous failure of all nodes (electricity outage).

For the sake of simplicity, it is assumed that parallel programme runs on all cluster nodes. Our system provide resilience to node failures for the first and the second scenario.

By dividing kernels into principals and subordinates we create recovery points. Each principal is, mainly, a control unit, with a goal. To achieve it, principal divides the task into parts and creates a subordinate to compute each of them. The principal copies itself to each subordinate in the order of their creation, and includes in each subordinate a list of all node IP addresses to which previously created subordinates were sent (a list of neighbours). When a connection from master node to slave node closes either as a result of a node failure, or as a consequence of the daemon hierarchy change, all kernels which reside on the corresponding cluster node are considered failed,

and recovery process is triggered in accordance with the following scenarios.

Scenario 1 & 2: With respect to kernel hierarchy, there are three possible variants of this failure: when a principal fails, when a subordinate fails (and both of them may or may not reside on the same cluster node) and when any combination of a principal and its subordinates fail.

When a subordinate fails, its copy is simply restored from the outbound buffer on the node where its principal is located. When the corresponding network connection closes all kernels from the buffer are automatically distributed across available nodes, or executed locally if there are no network connections.

When a principal fails every subordinate has its copy, but we need to restore it only once and only on one node to correctly continue programme execution. To ensure that the principal is restored only once, each subordinate tries to find the first surviving node from the IP address list of neighbours. If such node is online, the search stops and the subordinate is deleted. If the node is not found, the subordinate restores the principal from the copy on the current node and deletes itself. This algorithm is executed on every node, to which a copy of the principal was sent, and the guarantee that only one copy of the principal is restored is provided the implied hierarchy of IP addresses: every subordinate of the principal has the list of nodes to which only previously created subordinates were sent, and no communication originating from previously created subordinate to the newer subordinate is possible (only the other way round). Subordinate deletion is necessary, because the whole computational step, modelled by the principal, is re-executed from the initial state, and there is no simple and reliable way of taking into account partial results which were produced so far by the subordinates. Simultaneous failure of a combination of a principal and a number of its subordinates is handled the same way.

Deep kernel hierarchies: In deep kernel hierarchy a kernel may act as a subordinate and as a principal at the same time. Thus, we need to copy not only direct principal of each subordinate kernel, but also all principals higher in the hierarchy recursively. So, the additional variant is a generalisation of the two previous ones for deep kernel hierarchies.

Handling principal failure in a deep kernel hierarchy may involve a lot of overhead, because its failure is detected only when a subordinate finishes its execution. So, for sufficiently large number of subordinates, there can be a situation in which some of them finished their execution and triggered principal recovery, whereas other continue their execution in parallel to the newly created subordinates from the recovered principal. This behaviour may not be a desired one for programmes with sophisticated logic, which interact with external databases, as this may lead to deadlocks or information corruption in the corresponding database. For batch processing jobs this means, that writing to files by multiple subordinates is not reliable, and to avoid data loss programme logic should be changed so that only one (principal) kernel writes to a file, whereas subordinates only process separate dataset parts.

Scenario 3: Both failure scenarios are handled at runtime: the system will not stop execution of a programme, if some of its kernels are placed on failed node, unless all nodes on which the programme runs, fail simultaneously. This scenario is commonly occur as a result of electricity outage, and the main difference of this scenario is kernel log usage. Kernel log is stored on reliable storage and contains kernel initial states, recorded at a beginning of their execution, and each “update” to this state, recorded after a subordinate returns to its principal (a call to react). Each daemon maintains its own kernel log file, which is replicated on the selected number of nodes to provide resilience. Replication is configured externally by means of a parallel file system, RAID array or any other suitable technology.

When a daemon starts, recovery from the failure of all cluster nodes is handled as the follow.

- First, the system waits until a defined timeout elapses before staring recovery process to ensure as many nodes as possible are bootstrapped.
- Next, the system builds a sequential unified log from all log files for every programme that was run on the cluster when the failure occurred.
- After the unified log is ready, the system detects latest states of all alive kernels and re-executes them.

Recovery from a failure of all nodes is the most inefficient, because it involves the use of persistent storage and there is no reliable way to ensure that all cluster nodes have been bootstrapped. If some nodes were not bootstrapped properly, missing kernels are considered failed in accordance with the first and the second scenarios. This may lead to re-execution of considerable portion of parallel programme, especially when multiple principal kernels in the same hierarchy branch have failed. If a node fails in the middle of recovery process, the whole process is restarted from the beginning.

IV. Evaluation

Proposed node failure handling approach was evaluated on the example of real-world application [3]. The application generates ocean wavy surface in parallel with specified frequency-directional spectrum. There are two sequential steps in the programme. The first step is to compute model coefficients by solving system of linear algebraic equations. The system is solved in parallel on all cores of the principal node. The second step is to generate wavy surface, parts of which are generated in parallel on all cluster nodes including the principal one. All generated parts are written in parallel to individual files. So, from computational point of view the programme is embarrassingly parallel with little synchronisation between concurrent processes; the

TABLE I
Test platform configuration.

CPU	Intel Xeon E5440, 2.83GHz
RAM	4Gb
HDD	ST3250310NS, 7200rpm
No. of nodes	12
No. of CPU cores per node	8
Interconnect	1Gbit Ethernet

corresponding kernel hierarchy has one principal and N subordinates.

All experiments were run on physical computer cluster consisting of 12 nodes. Wavy ocean surface parts were written to Network File System (NFS) which is located on a dedicated server. The input data was read from NFS mounted on each node, as it is small in size and does not cause big overhead. Platform configuration is presented in Table I. A dry run of each experiment — a run in which all expensive computations (wavy surface generation and coefficient computation) were disabled, but memory allocations and communication between processes were retained — was performed on the virtual cluster.

The first failure scenario (see Section III-A) was evaluated in the following experiment. At the beginning of the second sequential application step all parallel application processes except one were shutdown with a small delay to give principal kernel time to distribute its subordinates between cluster nodes. The experiment was repeated 12 times with a different surviving process each time. For each run total application running time was measured. In this experiment the principal kernel was executed on the first node, and subordinate kernels are evenly distributed across all nodes including the first one. The result of the experiment is the overhead of recovery from a failure of a specific kernel in the hierarchy, which should be different for principal and subordinate kernel.

In the second experiment we benchmarked overhead of the multiple node failure handling code by instrumenting it with calls to time measuring routines. For this experiment all logging and output was disabled to exclude its time from the measurements. This test was repeated for different number of cluster nodes. The purpose of the experiment is to measure precisely the overhead of multiple node failure handling code and to investigate how failure handling overhead affects scalability of the application to a large number of nodes.

V. Results

The first experiment showed that in terms of performance there are three possible outcomes when all nodes except one fail (fig. 2). The first case is failure of all kernels except the principal and its first subordinate. There is no communication with other nodes to find the survivor and no recomputation of the current sequential step of the application, so it takes the least time to recover from the failure. The second case is failure of

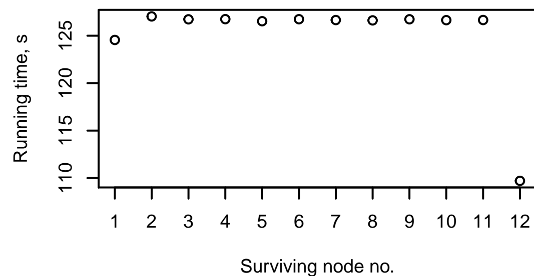

Fig. 2. Application running time in the presence of a failure of all physical cluster nodes except one for different surviving cluster nodes.

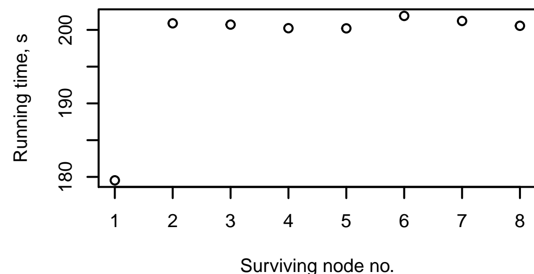

Fig. 3. Application running time in the presence of a failure of all virtual cluster nodes except one for different surviving cluster nodes.

all kernels except any subordinate kernel other than the first one. Here the survivor tries to communicate with all subordinates that were created before the survivor, so the overhead of recovery is larger. The third case is failure of all kernels except the last subordinate. Here performance is different only in the test environment, because this is the node to which standard output and error streams from each parallel process are copied over the network. So, the overhead is smaller, because there is no communication over the network for streaming the output. The same effect does not occur on virtual cluster (fig. 3). To summarise, performance degradation is larger when principal kernel fails, because the survivor needs to recover initial principal state from the backup and start the current sequential application step again on the surviving node; performance degradation is smaller when subordinate kernel fails, because there is no state to recover, and only failed kernel is executed on one of the remaining nodes.

The second experiment showed that overhead of multiple node failure handling code increases linearly with the number of nodes (fig. 4), however, its absolute value is small compared to the programme run time. Linear

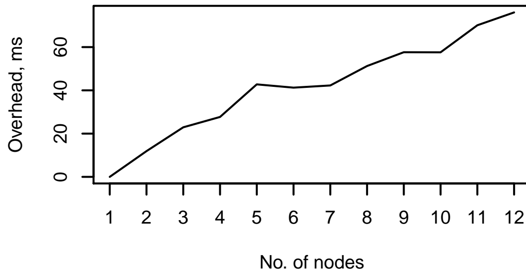

Fig. 4. Overhead of failure handling code for different number of physical cluster nodes.

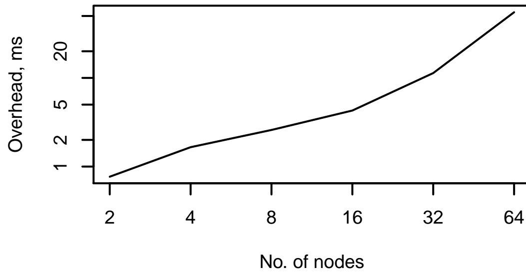

Fig. 5. Overhead of failure handling code for different number of virtual cluster nodes.

increase in overhead is attributed to the fact that for each subordinate kernel linear search algorithms are used when sending or receiving it from other node to maintain an array of its neighbours. When subordinate kernel is sent to remote node, all of its previously created neighbours IP addresses are added to the neighbours array without duplication, and the kernel itself is appended to the global internal map which stores principal kernels and theirs subordinates; when subordinate kernel returns from the remote node, it is removed from the array of its principal subordinates (retrieved from the internal map), which also requires linear search. So, the second experiment showed that for real-world programme overhead of multiple node failure handling is small.

VI. Discussion

Linear complexity in multiple node failure handling code can be avoided by replacing arrays with sets or maps, but the total overhead is small, so we deemed this optimisation unnecessary complication of the source code. Moreover, in real-world scenario it is probably impractical to copy principal kernel state to each subordinate node, and minimal number of copies may be configured in the programme instead. In this case using maps and sets over arrays may incur more overhead as they require certain amount of elements to make searching for an element more efficient than in arrays [4], [5]. There is no such thing as minimal number of object copies that ensures fault-tolerance in HPC, but for parallel file systems there is a number of replicas. This number is typically set to 2 or 3 depending on the particular site. We believe that there is no need to set number of object copies more than that, as it allows to tolerate simultaneous failure of 2 and 3 nodes respectively: it should be more than enough to tolerate node failures which are common at large scale [6]. So, using arrays with linear search complexity is more efficient than maps and sets, because the number of elements in them is small, and linear search takes less time than fixed time hash-based lookup.

Transmitting IP addresses of previous nodes is an optimisation over mapping to only linear hierarchies, that is hierarchies where only one subordinate is allowed at any given time point. For a hierarchy consisting of a principal kernel with multiple subordinates there is unique mapping that transforms it to linear hierarchy: the principal creates and sends to the pipeline only the first subordinate, after that the first subordinate creates and sends the second subordinate to the pipeline and so on. This approach is inefficient because creation of subordinates is sequential and each subordinate is created after sending the previous one to a cluster node. Moreover, each subordinate carries a copy of its parent to be able to proceed programme execution when the parent fails. Instead of transforming initial hierarchy to a linear one, one can copy IP addresses of all previously created subordinates along with the next subordinate to the cluster node. The number of copies may be adjusted in the programme or a configuration file. When principal kernel fails each subordinate determines alive subordinate kernel starting from the first address in the list. If such kernel is not found, execution proceeds on the current node. The sequence of IP addresses in the list implicitly forms linear hierarchy, which makes this optimisation equivalent to the transformation.

There are essentially two scenarios of failures. Failure of more than one node at a time and electricity outage. In the first scenario failure is handled by sending a list of previous IP addresses to the subsequent kernels in the batch. Then if subordinate node and its master fail simultaneously, the surviving subordinate nodes scan all of the IP addresses they received until they find alive node and the parent is revived on this node.

We believe that having kernel state and their inter-dependencies is enough to mitigate any combination of node failures: given that at least one node survives, all programmes continue their execution in possibly degraded state. However it requires recursively duplicating principals and sending them along with the subordinates. Only electricity outage requires writing data to disk, other failures can be mitigated by duplicating kernels in memory.

The framework has not been compared to other similar approaches, because to the best of our knowledge there is no library/framework that provides resilience to simultaneous failure of more than one node (including master node), and comparison to checkpoint/restart approach would be unfair, as we do not stop all parallel processes of an application and dump RAM image to stable storage, but only copy kernels into memory of another node. This approach is far more efficient than checkpoint/restart as no data is written to disk, and only a small fraction of the whole memory occupied by the application is copied to the other node.

VII. Related work

The feature that distinguishes our research with respect to some others, is the use of hierarchy as the only possible way of defining dependencies between objects, into which a programme is decomposed. The main advantage of hierarchy is trivial handling of object failures.

In [7] the authors describe codelet model for exascale machines. This model breaks a programme into small bits of functionality, called codelets, and dependencies between them. The programme dataflow represents directed graph, which is called well-behaved if forward progress of the programme is guaranteed. In contrast to our model, in codelet model hierarchical dependencies are not enforced, and resilience to failures is provided by object migration and relies on hardware fault detection mechanisms. Furthermore, execution of kernel hierarchies in our model resembles stack-based execution of ordinary programmes: the programme finishes only when all subordinate kernels of the main kernel finish. So, there is no need to define well-behaved graph to guarantee programme termination.

In [8] the authors describe migratable objects model for parallel programmes. In the framework of this model a programme is decomposed into objects that may communicate with each other by sending messages, and can be migrated to any cluster node if desired. The authors propose several possibilities, how this model may enhance fault-tolerance techniques for Charm++/AMPI programmes: proactive fault detection, checkpoint/restart and message logging. In contrast to our model, migratable objects do not compose a hierarchy, but may exchange messages with any object address of which is known to the sender. A spanning tree of nodes is used to orchestrate collective operations between objects. This tree is similar to tree hierarchy of nodes, which is used in our work to distribute kernels between available cluster nodes, but we use this hierarchy for any operations that require distribution of work, rather than collective ones. Our model does not use techniques described in this paper to provide fault-tolerance: upon a failure we re-execute subordinate kernels and copy principal kernels to be able to re-execute them as well. Our approach blends checkpoint/restart and message logging: each kernel which is sent to other cluster node is saved (logged) in the outbound buffer of the sender, and removed from the buffer upon return. Since subordinate kernels are allowed to communicate only with their principals (all other communication may happen only when physical location of the kernel is known, if the communication fails, then the kernel also fails to trigger recovery by the principal), a collection of all logs on each cluster nodes constitutes the current state of programme execution, which is used to restart failed kernels on the surviving nodes.

To summarise, the feature that distinguishes our model with respect to models proposed for improving parallel programme fault-tolerance is the use of kernel hierarchy — an abstraction which defines strict total order on a set of kernels (their execution order) and, consequently, defines for each kernel a principal kernel, responsibility of which is to re-execute failed subordinate kernels upon a failure.

With respect to various high-availability cluster projects [9]–[11] our approach has the following advantages. First, it scales with the large number of nodes, as only point-to-point communication between slave and master node is used instead of broadcast messages (which has been shown in the previous work [1]), hence, the use of several switches and routers is possible within single cluster. Second, our approach does not require the use of standby servers to provide high availability of a master node: we provide fault tolerance on kernel layer instead. As the computation progresses, kernels copy themselves on nodes that are logically connected to the current one, and these can be any nodes from the cluster. Finally, high-availability cluster projects do not deal with parallel programme failures, they aim to provide high-availability for services running on master node (NFS, SMB, DHCP, etc.), whereas our approach is specifically targeted at providing continuous execution of parallel applications.

VIII. Conclusion

In the paper we propose a system architecture consisting of two tree hierarchies of entities, mapped on each other, that simplifies provision of resilience to failures for parallel programmes. The resilience is solely provided by the use of hierarchical dependencies between entities, and is independent on each layer of the system. To optimise handling failure of multiple cluster nodes, we use the hierarchy implied by the order of creation of subordinate entities. The hierarchical approach to fault tolerance is efficient, scales to a large number of cluster nodes, and requires slow I/O operations only for the most disastrous scenario — simultaneous failure of all cluster nodes.

The future work is to standardise application programming interface of the system and investigate load-balancing techniques, which are optimal for a programme composed of many computational kernels.

Acknowledgement

The research was carried out using computational resources of Resource Centre “Computational Centre of Saint Petersburg State University” (T-EDGE96

HPC-0011828-001) within frameworks of grants of Russian Foundation for Basic Research (projects no. 16-07-01111, 16-07-00886, 16-07-01113).

References

[1] I. Gankevich, Y. Tipikin, and V. Gaiduchok, “Subordination: Cluster management without distributed consensus,” in High Performance Computing & Simulation (HPCS), 2015 International Conference on. IEEE, 2015, pp. 639–642.

[2] I. Gankevich, Y. Tipikin, V. Korkhov, and V. Gaiduchok, “Factory: Non-stop batch jobs without checkpointing,” in High Performance Computing & Simulation (HPCS), 2016 International Conference on. IEEE, 2016, pp. 979–984.

[3] I. Gankevich and Y. Tipikin, “Factory: A framework for distributed computing,” https://igankevich.github.io/factory/index.html.

[4] A. Alexandrescu, Modern C++ design: generic programming and design patterns applied. Addison-Wesley, 2001.

[5] B. Stroustrup, “Software development for infrastructure,” IEEE Computer, vol. 45, no. 1, pp. 47–58, 2012.

[6] B. Schroeder and G. A. Gibson, “Understanding failures in petascale computers,” in Journal of Physics: Conference Series, vol. 78, no. 1. IOP Publishing, 2007, pp. 12–22.

[7] S. Zuckerman, J. Suetterlein, R. Knauerhase, and G. R. Gao, “Using a codelet program execution model for exascale machines: position paper,” in Proceedings of the 1st International Workshop on Adaptive Self-Tuning Computing Systems for the Exaflop Era. ACM, 2011, pp. 64–69.

[8] E. Meneses, X. Ni, G. Zheng, C. L. Mendes, and L. V. Kale, “Using migratable objects to enhance fault tolerance schemes in supercomputers,” IEEE transactions on parallel and distributed systems, vol. 26, no. 7, pp. 2061–2074, 2015.

[9] A. Robertson, “Linux-HA heartbeat system design.” in Proc. of 4th Annual Linux Showcase & Conference. Atlanta, Georgia: USENIX, 2000, pp. 305–316. [Online]. Available: http://static.usenix.org/publications/library/proceedings/als00/2000papers/papers/full_papers/robertson/robertson_html/

[10] I. Haddad, C. Leangsuksun, and S. L. Scott, “HA-OSCAR: the birth of highly available OSCAR,” Linux Journal, vol. 2003, no. 115, p. 1, 2003.

[11] C. B. Leangsuksun, L. Shen, T. Liu, and S. L. Scott, “Achieving high availability and performance computing with an ha-oscar cluster,” Future Generation Computer Systems, vol. 21, no. 4, pp. 597–606, 2005.