Factory:
Non-stop Batch Jobs without Checkpointing

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

Vladimir Gaiduchok
Dept. of Computer Science and Engineering
Saint Petersburg Electrotechnical University "LETI"
Saint Petersburg, Russia
gvladimiru@gmail.com

POSTER PAPER

***Abstract*—Nowadays many job schedulers rely on checkpoint mechanisms to make long-running batch jobs resilient to node failures. At large scale stopping a job and creating its image consumes considerable amount of time. The aim of this study is to propose a method that eliminates this overhead. For this purpose we decompose a problem being solved into computational micro-kernels which have strict hierarchical dependence on each other. When a kernel abruptly stops its execution due to a node failure, it is responsibility of its principal to restart computation on a healthy node. In the course of experiments we successfully applied this method to make hydrodynamics HPC application run on constantly changing number of nodes. We believe, that this technique can be generalised to other types of scientific applications as well.**

***Keywords*—job scheduling, parallel computing, cluster computing, distributed computing, fault tolerance**

I. INTRODUCTION

There are three types of node failures that may occur in computer cluster—failure of a subordinate node, failure of a master node and an unplanned electricity outage—and each type of failure is handled differently. The usual way of handling a failure of a subordinate is to periodically checkpoint each long-running parallel job, i.e. temporarily suspend it, dump its memory image to some stable storage, and resume it on healthy nodes upon a failure. To handle a failure of a master node usually means to continuously replicate its state to a backup node which takes master's role upon a failure. Similarly, to overcome an unplanned outage the state of a master node can be replicated to geographically distant backup node connected to another cluster, but the execution state of all jobs would probably be lost.

Considerable effort is being put to make dumping job's memory image to disk less costly [**?**], and approaches alternative to checkpoint-based fault tolerance are not attracting much attention in this area. Why does this happen? Usually HPC applications use message passing for communication of parallel processes and store their state in global memory space, and there is no way one can restart a failed process from its current state without writing the whole memory image to disk. Usually the total number of processes is fixed by the job scheduler, and all parallel processes restart upon a failure. There is ongoing effort to make it possible to restart only the failed process [**?**] at a cost of overloading a healthy node or maintaining a number of spare nodes. Although, it would be more practical to proceed execution of a failed application in degraded state (without failed node), the message passing library does not allow to change the number of processes at runtime, and most programmes use this number to distribute the load. So, there is no practical way to implement fault tolerance in message passing library other than restarting all parallel processes from a checkpoint or restarting failed process on a spare healthy node.

There is, however, a possibility to implement fault tolerance to continue execution of a job on lesser number of nodes than it was initially requested. In this case the load is dynamically redistributed among available nodes. Although, dynamic load balancing was implemented in a number of projects [**?**], [**?**] based on message passing library, it was not used to implement fault tolerance.

We do not deal with failure detection in this work, and conservatively assume that a node fails when the corresponding network connection prematurely closes. This allows us to concentrate on the logic of fail over, which can be possibly incorporated into any existing framework of failure detection or vice versa.

In this paper we deal with failures of subordinate and master nodes and give a hint on how an unplanned outage can be handled without loosing much of the execution state of jobs. We show how to use well-established object-oriented programming techniques to store execution state in a hierarchy of objects rather than in hard-to-manage global and local variables. Finally, we show how to implement fault tolerance on top of the message passing library with some restrictions on process restarts.

II. METHODS

A. HIERARCHY OF NODES

This work is based on the results of previous research: In [**?**] we developed an algorithm that allows to build a tree hierarchy from strictly ordered set of cluster nodes. The sole purpose of this hierarchy is to make a cluster more fault-tolerant by introducing multiple master nodes. If a master node fails, then its subordinates try to connect to another node from the same or higher level of the hierarchy. If there is no such node, one of the subordinates becomes the master.

A position of a node in a hierarchy is determined by mapping its IP address position in a network to its layer and offset in a tree hierarchy. The number of layers is controlled by a fan-out value. As nodes' IP addresses change infrequently this mapping is mostly static, and affected only by node failures. Thus with help of tree hierarchy we can precisely determine IP address of a master node without resorting to costly leader election algorithms which are commonly used for this purpose.

We use this hierarchy to perform load balancing across neighbouring cluster nodes (nodes that are adjacent in the hierarchy), i.e. if the job is launched on a subordinate node its principal node also receives a fraction of the load. This rule makes the system symmetric: Each node runs the same software and it is easy to switch from a failed master node to a backup node, it is just a matter of changing node's role. Similar design choice is applied in distributed key-value stores [**?**], [**?**] to handle failure of a master node, but we have no knowledge of job schedulers that use this to distribute the load on the cluster with multiple master nodes.

B. HIERARCHY OF COMPUTATIONAL KERNELS

Each programme that runs on top of the tree hierarchy is composed of computational kernels—objects that contain data and code to process it. To exploit parallelism a kernel may create arbitrary number of subordinate kernels which are automatically spread first across available processor cores, second across subordinate nodes in the tree hierarchy. The programme is itself a kernel (without a parent as it is executed by a user), which either solves the problem sequentially on its own or creates subordinate kernels to solve it in parallel.

Unlike main function in programmes based on message passing library, the first computational kernel is initially run only on one node, and remote nodes are used only when the local queue is overflown by kernels. This design choice allows to have arbitrary number of nodes throughout execution of a programme, and take more nodes for highly parallel parts of the code. Somewhat similar choice was made in the design of MapReduce framework [**?**], [**?**]—a user submitting a job does not specify the number of hosts to run its job on, and effective hosts are the hosts where input files are located.

From mathematical point of view kernel K can be described as a vector-valued functional which recursively maps a kernel to n-component vector of kernels:

$$K(f) : \mathbb{K} \to \mathbb{K}^n \qquad \mathbb{K}^n = \{f : \mathbb{K} \to \mathbb{K}^n\}.$$

Dummy kernel $\mathbb{O} : \mathbb{K} \to \mathbb{K}^0$, which stops recursion, is used to call the first kernel and finish execution of the programme. An argument to each kernel is interpreted using the following rules.

1) If a kernel is a new kernel, then its argument is its parent kernel.
2) If a kernel is a parent of the kernel that produced it or some other existing kernel, then the argument is the kernel that produced it.

Engine that executes kernels is implemented as a simple loop. It starts with calling the first kernel with a dummy kernel as an argument, then calls each kernel that was produced by this call and so forth. The loop finishes when a dummy kernel is returned as a result of the call.

Since kernel call may return multiple kernels they are executed in parallel. Parallel execution quickly produces a pool of kernels which permit execution in an unspecified order. Several threads concurrently retrieve kernels from the pool and may "spill" remaining kernels to neighbouring cluster nodes.

Kernels are implemented as closures—function objects containing all their arguments, a reference to parent kernel and user-supplied data. The data is either processed upon kernel call, or subordinate kernels are created to process it in parallel. When the processing is complete a parent kernel closure with its subordinate kernel as an argument is called to collect data from it.

C. HANDLING SINGLE NODE FAILURES

Basic strategy to overcome a failure of a subordinate node is to restart corresponding kernels on healthy node—a strategy employed in Erlang language to restart failed subordinate processes [**?**]. To implement this we record every kernel that is sent to remote cluster nodes, and in an event of a node failure these kernels are simply rescheduled to other subordinate nodes with no special handling from a programmer. If there are no nodes to sent kernels to, they are scheduled locally. So, in contrast to heavy-weight checkpoint/restart machinery, tree hierarchy allows automatic and transparent handling of subordinate node failures without restarting parallel processes on every node.

A possible way of handling a failure of a node where the first kernel is located is to replicate this kernel to a backup node, and make all updates to its state propagate to the backup node by means of a distributed transaction. However, this approach does not play well with asynchronous nature of computational kernels. Fortunately, the first kernel usually does not perform operations in parallel, it is rather sequentially launches execution steps one by one, so it has only one subordinate at a time. Keeping this in mind, we can simplify synchronisation of its state: we can send the first kernel along

with its subordinate to the subordinate node. When the node with the first kernel fails, its copy receives its subordinate, and no execution time is lost. When the node with its copy fails, its subordinate is rescheduled on some other node, and a whole step of computation is lost in the worst case.

Described approach works only for kernels that do not have a parent and have only one subordinate at a time, which means that they act as optimised checkpoints. The advantage is that they save results after each sequential step, when memory footprint of a programme is low, they save only relevant data, and they use memory of a subordinate node instead of stable storage.

D. HANDLING OUTAGES

Electricity outage is a serious failure, so if there is no other geographically distant cluster that can share the load, then the only choice is to hope that no important data is lost and restart every batch job after full site recovery. To reduce restart time the first kernel of each job may save its state (which is small compared to the full state of a job) to some stable storage. Such scenario complicates design of a distributed system so it was not considered in this paper.

E. IMPLEMENTATION

For efficiency reasons fault tolerance techniques described above are implemented in the C++ framework: From the authors' perspective C language is deemed low-level for distributed programmes, and Java incurs too much overhead and is not popular in HPC community. To use the framework without a job scheduler, we need to implement a daemon that maintains the state of the hierarchy of nodes and exposes API to interact with it. As of now, the framework runs in the same process as an parallel application that uses it. The framework is called Factory, it is now in proof-of-concept development stage.

III. RESULTS

Factory framework is evaluated on physical cluster (Table I) on the example of hydrodynamics HPC application which was developed in [**?**], [**?**], [**?**], [**?**]. This programme generates wavy ocean surface using ARMA model, its output is a set of files representing different parts of the realisation. From a computer scientist point of view the application consists of a series of filters, each applying to the result of the previous one. Some of the filters are parallel, so the programme is written as a sequence of big steps and some steps are made internally parallel to get better performance. In the programme only the most compute-intensive step (the surface generation) is executed in parallel across all cluster nodes, and other steps are executed in parallel across all cores of the master node.

The application was rewritten for the new version of the framework which required only slight modifications to handle failure of a node with the first kernel: The kernel was flagged so that the framework makes a replica and sends it to some subordinate node. There were no additional code changes other than modifying some parts to match the new API. So, the tree hierarchy of kernels is mostly non-intrusive model for providing fault tolerance which demands explicit marking of replicated kernels.

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

In a series of experiments we benchmarked performance of the new version of the application in the presence of different types of failures (numbers correspond to the graphs in Figure 1):

1) no failures,
2) failure of a slave node (a node where a part of wavy surface is generated),
3) failure of a master node (a node where the first kernel is run),
4) failure of a backup node (a node where a copy of the first kernel is stored).

A tree hierarchy with fan-out value of 64 was chosen to make all cluster nodes connect directly to the first one. In each run the first kernel was launched on a different node to make mapping of kernel hierarchy to the tree hierarchy optimal. A victim node was made offline after a fixed amount of time after the programme start which is equivalent approximately to $1/3$ of the total run time without failures on a single node. All relevant parameters are summarised in Table II (here "root" and "leaf" refer to a node in the tree hierarchy). The results of these runs were compared to the run without node failures (Figures 1-2).

There is considerable difference in net performance for different types of failures. Graphs 2 and 3 in Figure 1 show that performance in case of master or slave node failure is the same. In case of master node failure a backup node stores a copy of the first kernel and uses this copy when it fails to connect to the master node. In case of slave node failure, the master node redistributes the load across remaining slave nodes. In both cases execution state is not lost and no time is spent to restore it, that is why performance is the same. Graph 4 in Figure 1 shows that performance in case of a backup node failure is much lower. It happens because master node stores only the current step of the computation plus some additional fixed amount of data, whereas a backup node not only stores the copy of this information but executes this step in parallel with other subordinate nodes. So, when a backup node fails, the master node executes the whole step once again on arbitrarily chosen healthy node.

TABLE II
BENCHMARK PARAMETERS.

Experiment no.	Master node	Victim node	Time to offline, s
1	root		
2	root	leaf	10
3	leaf	leaf	10
4	leaf	root	10

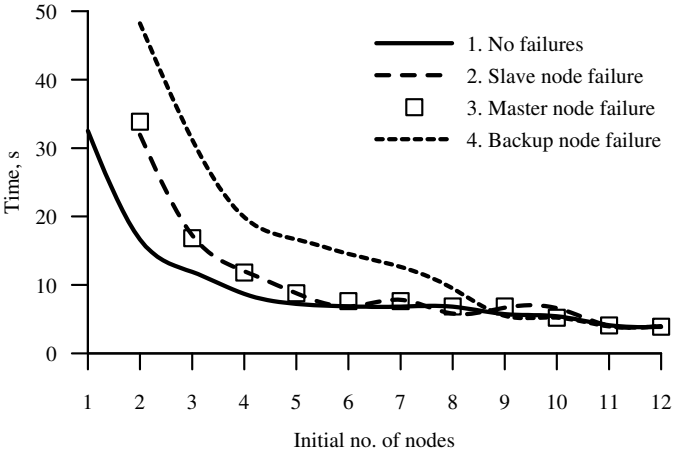

Figure 1. Performance of hydrodynamics HPC application in the presence of node failures.

Finally, to measure how much time is lost due to a failure we divide the total execution time with a failure by the total execution time without the failure but with the number of nodes minus one. The results for this calculation are obtained from the same benchmark and are presented in Figure 2. The difference in performance in case of master and slave node failures lies within 5% margin, and in case of backup node failure within 50% margin for the number of node less than 6[a]. Increase in execution time of 50% is more than $1/3$ of execution time after which a failure occurs, but backup node failure need some time to be discovered: they are detected only when subordinate kernel carrying the copy of the first kernel finishes its execution and tries to reach its parent. Instant detection requires abrupt stopping of the subordinate kernel which may be undesirable for programmes with complicated logic.

To summarise, the benchmark showed that *no matter a master or a slave node fails, the resulting performance roughly equals to the one without failures with the number of nodes minus one*, however, when a backup node fails performance penalty is much higher.

IV. DISCUSSION

The benchmark from the previous section show that it is essential for a parallel application to have multiple sequential steps to make it resilient to cluster node failures. Although,

[a]Measuring this margin for higher number of nodes does not make sense since time before failure is greater than total execution time with these numbers of nodes, and programme's execution finishes before a failure occurs.

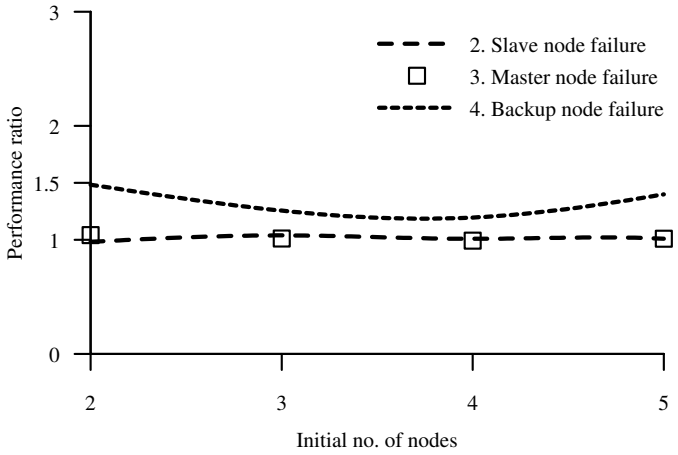

Figure 2. Slowdown of the hydrodynamics HPC application in the presence of different types of node failures compared to execution without failures but with the number of nodes minus one.

the probability of a master node failure is lower than the probability of failure of any of the slave nodes, it does not justify loosing all the data when the programme run is near completion. In general, the more sequential steps one has in an HPC application the less is performance penalty in an event of master node failure, and the more parallel parts each step has the less is performance penalty in case of a slave node failure. In other words, *the more scalable an application is the more resilient to node failures it becomes*.

In our experiments we specified manually where the programme starts its execution to make mapping of hierarchy of computational kernels to tree hierarchy of nodes optimal, however, it does not seem practical for real-world cluster. The framework may perform such tasks automatically, and distribute the load efficiently no matter whether the master node of the application is located in the root or leaf of the tree hierarchy: Allocating the same node for the first kernel of each application deteriorates fault-tolerance.

Although it may not be clear from the benchmarks, Factory does not only provide tolerance to node failures: new nodes automatically join the cluster and receive their portion of the load as soon as it is possible. This is trivial process as it does not involve restarting failed kernels or managing their state, so it is not presented in this work.

In theory, hierarchy-based fault-tolerance can be implemented on top of the message-passing library without loss of generality. Although it would be complicated to reuse free nodes instead of failed ones, as the number of nodes is often fixed in such libraries, allocating reasonably large number of nodes for the application would be enough to make it fault-tolerant. However, implementing hierarchy-based fault-tolerance "below" message-passing library does not seem beneficial, because it would require saving the state of a parallel application which equals to the total amount of memory it ccupies on each host, which would not make it more efficient than checkpoints.

The weak point of the proposed technology is the length of the period of time starting from a failure of master node up to the moment when the failure is detected, the first kernel is restored and new subordinate kernel with the parent's copy is received by a subordinate node. If during this period of time backup node fails, execution state of application is completely lost, and there is no way to recover it other than fully restarting the application. The length of the dangerous period can be minimised but the possibility of a abrupt programme stop can not be fully eliminated. This result is consistent with the scrutiny of "impossibility theory", in the framework of which it is proved the impossibility of the distributed consensus with one faulty process [**?**] and impossibility of reliable communication in the presence of node failures [**?**].

V. RELATED WORK

In [**?**] the author describes master-slave programming model suitable for dynamic load balancing. In the framework of this model multiple master nodes arranged in a ring are used to distribute the load on other nodes, the state of the master nodes is synchronised by sending work queue across the ring. Although, this model does not provide fault tolerance, from computational point of view it is similar to our tree hierarchy of nodes with an infinite maximal fan-out value. So, tree hierarchy can be seen as a generalisation of master-slave model for arbitrary number of levels.

In [**?**] the author describes popular fault tolerance approaches employed in cloud computing, with load balancing using highly-available proxy server being the most popular one. The author mentions the time to recover from the failure of a single node being several milliseconds. Although, this result was obtained in non-HPC domain, it somewhat correlates with our findings that performance of a parallel application with a slave node failure roughly equals the time without failures and without this node participating in computations.

In [**?**] the author compares various checkpoint/restart implementations used in HPC. The author mentions that one of the drawbacks of checkpoint/restart mechanism is that it is not portable, e.g. neither every checkpoint/restart implementation supports restoring network socket state, nor it is fully compatible with every operating system kernel version. Although, application level fault tolerance—the one that is provided by tree hierarchy—does not have any of these disadvantages, it cannot provide fault tolerance to existing message-passing based parallel programmes. So, there are different trade-offs for different technologies.

In [**?**] the author describes an optimised checkpointing algorithm for send-deterministic MPI applications. In a series of tests they show that the algorithm reduces the number of parallel processes that are required to restart from a checkpoint by half. It is achieved by carefully tracking causal dependencies between messages sent by every process and grouping messages by epochs—a sequential steps of programme execution. The algorithm looks promising, but still requires creating checkpoints for each process.

VI. CONCLUSION

Proposed master node fault-tolerance approach works only for kernels that do not have a parent and have only one subordinate at a time, which is act similar to how manually triggered checkpoints function. The advantage is that they

- save results after each sequential step when memory footprint of a programme is low so that they save only relevant data,
- and they use memory of a subordinate node instead of stable storage.

This allows them to be much faster than traditional checkpoints at a cost of using small amount of memory of subordinate node to store execution state of a sequential step of the programme.

Although, after a failure of backup node it takes more time to recover present execution state, it is not dangerous requiring only simple restart. At the same time a failure of master node may lead to a full programme stop, if backup node fails before master node recovery completes. One of the way to mitigate this is to make multiple copies of the first kernel and send them synchronously to different subordinate nodes. This approach requires some complicated logic to recover from master node failure, but may increase the number of nodes that may simultaneously fail. This is one of the directions of future research work.

Hierarchical dependence between computational kernels coupled with tree hierarchy of nodes simplifies implementation of application level fault-tolerance. Provided with a reasonably large amount of nodes, an application can survive failure of any node during a single programme run. So, the other direction of future work is to "daemonise" the framework to make it possible to benchmark multiple applications on the same cluster.

ACKNOWLEDGEMENTS

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) and Saint Petersburg State University (project no. 0.37.155.2014).

REFERENCES

[1] I. P. Egwutuoha, D. Levy, B. Selic, and S. Chen, "A survey of fault tolerance mechanisms and checkpoint/restart implementations for high performance computing systems," *The Journal of Supercomputing*, vol. 65, no. 3, pp. 1302–1326, 2013.

[2] H. Meyer, D. Rexachs, and E. Luque, "Radic: A faulttolerant middleware with automatic management of spare nodes*," in *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)*. The Steering Committee of The World Congress in Computer Science, Computer Engineering and Applied Computing (WorldComp), 2012, p. 1.

[3] M. Bhandarkar, L. V. Kalé, E. de Sturler, and J. Hoeflinger, "Adaptive load balancing for mpi programs," in *Computational Science-ICCS 2001*. Springer, 2001, pp. 108–117.

[4] E. L. Lusk, S. C. Pieper, R. M. Butler *et al.*, "More scalability, less pain: A simple programming model and its implementation for extreme computing," *SciDAC Review*, vol. 17, no. 1, pp. 30–37, 2010.

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

[6] J. C. Anderson, J. Lehnardt, and N. Slater, *CouchDB: The definitive guide*. O'Reilly Media, Inc., 2010.

[7] A. Lakshman and P. Malik, "Cassandra: A decentralized structured storage system," *ACM SIGOPS Operating Systems Review*, vol. 44, no. 2, pp. 35–40, 2010.

[8] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[9] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth *et al.*, "Apache hadoop yarn: Yet another resource negotiator," in *Proceedings of the 4th annual Symposium on Cloud Computing*. ACM, 2013, p. 5.

[10] J. Armstrong, "Making reliable distributed systems in the presence of software errors," Ph.D. dissertation, The Royal Institute of Technology Stockholm, Sweden, 2003.

[11] A. Degtyarev and I. Gankevich, "Evaluation of hydrodynamic pressures for autoregression model of irregular waves," in *Proceedings of 11th International Conference "Stability of Ships and Ocean Vehicles", Athens*, 2012, pp. 841–852.

[12] ——, "Wave surface generation using OpenCL, OpenMP and MPI," in *Proceedings of 8th International Conference "Computer Science & Information Technologies"*, 2011, pp. 248–251.

[13] A. Degtyarev and A. Reed, "Modelling of incident waves near the ship's hull (application of autoregressive approach in problems of simulation of rough seas)," in *Proceedings of the 12th International Ship Stability Workshop*, 2011.

[14] ——, "Synoptic and short-term modeling of ocean waves," in *Proceedings of 29th Symposium on Naval Hydrodynamics*, 2012.

[15] M. J. Fischer, N. A. Lynch, and M. S. Paterson, "Impossibility of distributed consensus with one faulty process," *Journal of the ACM (JACM)*, vol. 32, no. 2, pp. 374–382, 1985.

[16] A. Fekete, N. Lynch, Y. Mansour, and J. Spinelli, "The impossibility of implementing reliable communication in the face of crashes," *Journal of the ACM (JACM)*, vol. 40, no. 5, pp. 1087–1107, 1993.

[17] A. Bala and I. Chana, "Fault tolerance-challenges, techniques and implementation in cloud computing," *IJCSI International Journal of Computer Science Issues*, vol. 9, no. 1, pp. 1694–0814, 2012.

[18] A. Guermouche, T. Ropars, E. Brunet, M. Snir, and F. Cappello, "Uncoordinated checkpointing without domino effect for send-deterministic mpi applications," in *Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International*. IEEE, 2011, pp. 989–1000.