Addressing the Last Roadblock for Message Logging in HPC: Alleviating the Memory Requirement Using Dedicated Resources

Tatiana Martsinkevich[1](✉), Thomas Ropars[2], and Franck Cappello[3]

[1] Inria, University of Paris Sud, Orsay, France
tatiana.mar@inria.fr
[2] Inria, Bordeaux, France
thomas.ropars@inria.fr
[3] Argonne National Laboratory Argonne, Lemont, IL, USA
cappello@mcs.anl.gov

Abstract. Currently used global application checkpoint-restart will not be a suitable solution for HPC applications running on large scale as, given the predicted fault rates, it will impose a high load on the I/O subsystem and lead to inefficient resource usage. Combining application checkpointing with message logging is appealing as it allows restarting only the processes that actually failed. One major issue with message logging protocols is the high amount of memory required to store logs. In this work we propose to use additional dedicated resources to save the part of the logs that would not fit in the memory of a compute node. We show that, combined with a cluster-based hierarchical logging technique, only few dedicated nodes would be required to accommodate the memory requirement of message logging protocols. We additionally show that the proposed technique achieves a reasonable performance overhead.

Keywords: High-performance computing · Fault tolerance · Message logging · Hierarchical message-logging protocols · Dedicated resources

1 Introduction

As the scale and performance of newly built supercomputers grow, we are getting closer to being able to tackle extremely large problems. On the other hand, the mean time between failures (MTBF) of such systems is expected to decrease to several hours [5], making it more challenging for high-performance computing (HPC) applications to progress with computations in the presence of failures.

Today, many message-passing applications use checkpoint-restart techniques for fault tolerance. This approach may become unacceptable at large scale, however, because the decreased MTBF will require that applications take checkpoints more frequently, thus spending less time doing useful work. Some efforts have been made to lower the cost of checkpointing at scale [7,10,13]. However, the technique may still be impractical from the point of view of power consumption and time to restart the whole application every time a failure occurs.

S. Hunold et al. (Eds.): Euro-Par 2015 Workshops, LNCS 9523, pp. 644–655, 2015.
DOI: 10.1007/978-3-319-27308-2_52

Log-based fault tolerance protocols offer an alternative solution that has a better failure containment: In a message-logging protocol only the failed process has to roll back and restart in the best case. The possibility of avoiding the whole application restart makes such protocols look appealing for large-scale executions. Because processes store the message payload in their memory, however, message logging can be memory demanding, especially for communication-intensive applications [14,16].

A basic way to limit the log size is to use periodic checkpointing followed by a garbage collection of logs. Therefore, message logging usually goes hand in hand with checkpointing. However, high checkpointing frequency would be required to avoid memory exhaustion in applications with high log rate. For instance, assuming a per-process log growth rate of 4 MB/second[1] and assuming that a process has 2 GB of memory available but can use no more than 10 % of that memory for message logging, a checkpoint would have to be taken approximately every 50 seconds.

Hierarchical message-logging protocols have proved efficient in reducing log size at an expense of weaker failure containment: These techniques apply a message-logging protocol not to individual processes but to clusters of processes. Hence, the whole cluster has to restart upon a failure. Although clustering helps reduce the average log growth rate per process, hierarchical protocols do not fully solve the log size problem because some processes may still experience a rapid growth of their logs [16].

In this work we propose to use additional dedicated resources, or logger nodes, that cooperate with the compute nodes by storing part of their message logs in the memory. They can be regarded as a memory extension for compute nodes. The purpose is to avoid being bound by memory constraints on the node when choosing the checkpointing period for an application.

The contributions of this paper can be summarized as follows:

– We provide a basic algorithm for dumping message logs to logger nodes. The algorithm uses proactive dumping in order to overlap application computations and flushing of logs.
– We evaluate the impact of different factors on the overhead of log dumping. We show that using less than 10 % of additional resources can be enough to achieve a reasonable overhead of log dumping for some applications.
– We demonstrate that using hierarchical logging protocols together with logger nodes can be the ultimate solution to the memory limitation problem in logging protocols. We use an example approximating a real-life application execution to show that one can find a balanced configuration with process clustering, logger nodes, and a memory quota that keeps the execution overhead below 10 %.

This work is structured as follows. Section 2 gives background information on message logging. Section 3 describes our algorithm for log dumping. We start

[1] Note that some applications have a much higher log growth rate; see [14,16].

Sect. 4 by giving some details on the implementation of the algorithm and then we present evaluation results. Section 5 summarizes our results and discusses future work.

2 Background and Related Work

In this paper we consider MPI applications executed on an asynchronous distributed system where nodes fail by crashing. The set of processes is fixed (no dynamic process creation) and multiple concurrent failures can occur. In this context, Sect. 2.1 provides an overview of log-based fault tolerance protocols. Section 2.2 describes related work on techniques to reduce log size.

2.1 Message-Logging Protocols

Message-logging protocols save a copy of the payload of all the messages delivered by a process, as well as the corresponding delivery events (determinants), to be able to replay them in the same order upon a failure. Assuming that message delivery events are the only source of nondeterminism during the execution, this approach is enough to ensure that the process state can be restored correctly after a failure.

Implementing efficient message logging requires solving two problems: saving the message payload efficiently and saving the determinants efficiently. In both cases, data has to be stored in such a way that it can be retrieved after a failure. A simple solution to the first problem is sender-based message logging [11]. A process P_0 sending a message to a process P_1 saves a copy of the payload in its memory. If P_1 fails, P_0 is able to resend the message. If P_0 fails as well, the copy of the message is lost but it will be generated again during the re-execution of P_0. Research has shown that sender-based message logging can be implemented with almost no performance penalty on communication [16].

On the other hand, logging the determinants efficiently is more difficult because they must include the delivery order and, hence, must be logged by the receiver. Therefore, the determinants must be stored reliably in order to avoid losing them if the process fails. Techniques for determinant logging fall into three categories: optimistic, pessimistic, and causal [1]. However, they all suffer from performance issues at scale [3,17], one exception being the combination of optimistic logging with coordinated checkpointing [14].

In the context of MPI applications, a few solutions have been proposed to reduce the number of determinants to be logged, or even completely eliminate the need to log them. For example, a blocking named reception in MPI is a deterministic event and, hence, does not need to be logged [2]. The partial determinism of most MPI HPC applications can also be leveraged to fully avoid logging determinants [6,16].

Given these optimizations, the main problem that remains to be solved is related to the size of logs in message logging protocols.

2.2 Reducing the Log Size

Using checkpointing does not solve the problem of large message logs completely, because some applications have such high log rates that they would require unacceptably frequent checkpointing.

Ferreira et al. [9] studied the possibility of reducing the log size by compressing it. They showed that, while one can achieve positive results for some workloads, the method is not universally good. Additionally, the question of the impact of this technique on application performance remains open.

Hierarchical protocols that use process clustering are another approach proposed for reducing the memory footprint of message logging [4,12,16]. The amount of logged data decreases naturally because message logging is applied only to intercluster messages. Hierarchical protocols work well in many cases thanks to the low connectivity degree of the applications' communication graph: For many applications, the percentage of processes to restart in the event of a failure can be kept below 15 % while logging less than 15 % of all messages [15].

Even with hierarchical protocols, some processes might have to log a lot of data. This is the case, for instance, if a process is *at the border* of a cluster and communicates mainly with processes from other clusters. We observed this phenomenon in our tests (see Sect. 4.6) even with some optimized clustering strategy [15]. Hence, the problem of large memory footprint of message logging, along with the memory limitation of compute nodes, still remains relevant.

3 Dedicated Logger Nodes

To alleviate the problem of limited memory available to message-logging protocols, we propose to use additional dedicated resources, or logger nodes. A logger node does not participate in the computation but is used only for storing in its memory the portion of message logs that does not fit in the memory of a computation process.

Compute processes track how much memory is being used by the logging protocol during execution. Once this value reaches a predefined limit, the process has to free some memory by flushing a part of its log to a logger node.

In practice, we start flushing logs proactively to avoid the situation where the process stops progressing because it has no memory left for message logging and has to free some memory first. Specifically, we use nonblocking MPI send routines to be able to overlap log dumping with computations and thus reduce the performance penalty as much as possible.

Algorithm 1 presents a pseudo code for dumping message logs. The user specifies a memory quota M for the message-logging protocol, and we set a $DUMP\_THRESHOLD$ for the proactive dump smaller than M. Every time the process logs a message, it checks whether it has reached this threshold (lines 7–8). If yes, it tries to free some memory.

The dumping process consists of two phases. First, the process sends a memory allocation request to a logger node. If the logger accepts the request, the

Algorithm 1. Log dumping algorithm

```
 1: procedure $log(msg)$
 2:     while true do
 3:         if $copy(msg)$ then     ▷ copy() returns true if $msg$ was copied to local logs
 4:             break
 5:         else
 6:             $dump\_log(sizeof(msg))$
 7:     if $log\_sz > DUMP\_TRESHOLD$ then
 8:         $proactive\_dump\_log()$
 9:
10: procedure $request\_logger\_mem(mem\_size)$
11:     for each $lgr$ in $loggers$ do
12:         $send(lgr, mem\_size)$
13:         $resp \leftarrow receive(lgr)$
14:         if $resp = 1$ then return $lgr$
15:
16: procedure $dump\_log(mem\_size)$
17:     if $dump\_req \neq NULL$ then
18:         $wait\_complete(dump\_req)$
19:     $logger \leftarrow request\_logger\_mem(mem\_size)$
20:     $send(logger, log)$                         ▷ $log$: the portion of log to dump
21:
22: procedure $proactive\_dump\_log()$
23:     if $dump\_req \neq NULL$ then
24:         if $!test\_complete(dump\_req)$ then
25:             return
26:     $logger \leftarrow request\_logger\_mem(dump\_size)$
27:     $dump\_req \leftarrow isend(logger, log)$          ▷ $log$: the portion of log to dump
```

process posts an asynchronous send request for the log portion it wants to free (line 26–27); otherwise, it tries another logger node. The value of *dump_size* is implementation-dependant. The process checks whether the request has completed when *proactive_dump*() is called again. No more logs are sent to loggers if the previous dump is not finished yet (lines 23–25) or if the memory usage by the protocol has gone below the $DUMP\_THRESHOLD$.

If proactive dumping of logs does not suffice and the process reaches the limit M, the process first tries to complete any pending dump request and then uses a blocking communication call to flush more memory before continuing with logging (lines 2–6).

The failure of a logger node is harmless as long as no application process, that would need messages stored by that logger for replaying, fails. If this happens, the processes that lost a portion of their logs will have to restart from the last checkpoint to generate these messages again. Careful strategies for assigning logger nodes to processes would have to be designed to limit the extend of rollbacks in unfortunate scenarios. Discussing such strategies is outside the scope of this paper. In our experiments, each process simply contacts first the logger with $id = myrank \textbf{ mod } nloggers$.

4 Evaluation

In this section we first describe our implementation of logger nodes. We then estimate the runtime overhead of log dumping with different numbers of logger nodes and different memory limits. We also demonstrate the benefits of combining hierarchical logging techniques with logger nodes.

4.1 Implementation

To evaluate the cost of dumping message logs to dedicated nodes we implemented a basic sender-based message-logging protocol in the PMPI profiling layer. This protocol is loosely based on our previous work [16]: We rely on the channel deterministic property of many HPC applications to avoid logging of determinants. Thus, we log only message payloads and any information necessary to be able to replay sending of messages. However, we point out that the proposed dumping algorithm could be applied to any sender-based message-logging protocol.

To facilitate memory management for the logging protocol, we allocate and free memory in blocks of fixed size. When dumping to loggers, a process sends one whole memory block at a time. After some fine-tuning tests, we chose the memory block size to be 32 KB in our evaluation tests. Logger processes are started together with the user program, and they wait for incoming requests to store message logs. A logger may accept or decline the request depending on whether it has enough memory left. If a logger declines the request, the compute process tries the next logger in round-robin fashion.

During the execution of the application, we need to decide what portion of log should be flushed during a proactive dump (*dump_size*): if the portion is too large it may impact the application performance; if it is too small it may increase the risk of falling back to blocking mode for flushes. Hence, we think that *dump_size* should be computed dynamically during execution, based on the current log growth rate of the process. In our experiments, we simply use an application-specific coefficient α to compute $dump\_size = \alpha \times lograte$.

4.2 Experimental Setup

To evaluate the overhead of dumping message logs to logger nodes we conducted a set of experiments on the Graphene cluster of the Grid5000 testbed. Graphene is a 144-node cluster where each node has one 2.53 GHz Intel Xeon X3440 CPU. Each CPU has four cores and a total of 16 GB of memory per node. Nodes are connected by the high-performance InfiniBand-20G network. We use the OpenMPI-1.4.5 library.

Four applications were used in the tests: a molecular dynamics simulator LAMMPS; a numerical model CM1, used to study atmospheric phenomena;

Table 1. Application input parameters

LAMMPS	Lennard-Jones liquid benchmark, 6912000 particles, 150 steps
GTC	micell=100, mecell=100, npartdom=1, 8 steps
CM1	$1536 \times 1536 \times 40$ grid points, timax=70
MILC	nx=64, ny=64, nz=64, nt=16

and two benchmarks from the NERSC-8 benchmark suite—a 3D gyrokinetic toroidal code GTC and the MILC lattice quantum chromodynamics code. In all the experiments, each MPI process is placed on one core of a node, four MPI processes per node in total. Each logger node runs four logger processes, one process binded to each core.

For the settings of the logging protocol, α was set based on some fine-tuning tests to 0.1 for MILC, 0.2 for LAMMPS, and 1.0 for GTC and CM1. The memory threshold after which processes start proactive dumping was set to 5 MB before the memory limit used in the test (e.g., if the memory limit is 20 MB, the threshold is set to 15 MB).

We took an average runtime over four executions in all tests. We then compared the overhead[2] of an execution with message logging with the native execution time without any logging.

4.3 Dumping Overhead with Different Number of Logger Nodes

We first examine the application behavior when dumping message logs. The applications were run with 64 processes on 16 nodes; input parameters are presented in Table 1.

As we want to study the impact of logger nodes contention on performance, we set the memory limit for the logging protocol to a small value in order to force processes to start dumping logs almost immediately after they start. Each process logs all the outgoing messages. Table 2 gives information about the applications' log rate, the log size per process, and the total amount of data dumped to the loggers. To prevent memory exhaustion on loggers, we used a circular buffer of fixed size (10 MB) for incoming logs in this experiment.

Figure 1 (line labeled "*no_clu*") presents the overhead of the four applications for a number of logger nodes varying from 1 to 16. MILC has the highest overhead with 11.70 % in the test with one logger node. Such overhead is expected given the high log rate of this application and the total amount of dumped logs. The higher the log rate, the more frequently an application has to free memory for logging new messages. CM1 shows the best behavior, with only 1.76 % overhead in the test with one logger node. LAMMPS and GTC fall in between.

[2] The overhead computation only takes into account the execution time. Logger nodes also require using additional resources.

Table 2. Application logging statistics

	LAMMPS	GTC	CM1	MILC
Log growth rate (MB/S)	5.20	2.35	1.75	13.51
Average logged per process (MB)	300	227	232	738
Totally dumped to loggers (GB)	18.5	14.1	14.1	45.8

4.4 Combining Hierarchical Protocols with Logger Nodes

Using process clustering effectively reduces the number of messages that need to be logged, at a cost of increasing the number of processes that will have to roll back and restart should a failure occur. Combined with logger nodes, clustering can be used for finding a better trade-off between the number of logger nodes and the failure containment. We used a clustering tool [15] to generate clusters that minimize the number of messages logged. Note that in each configuration, all clusters have the same size.

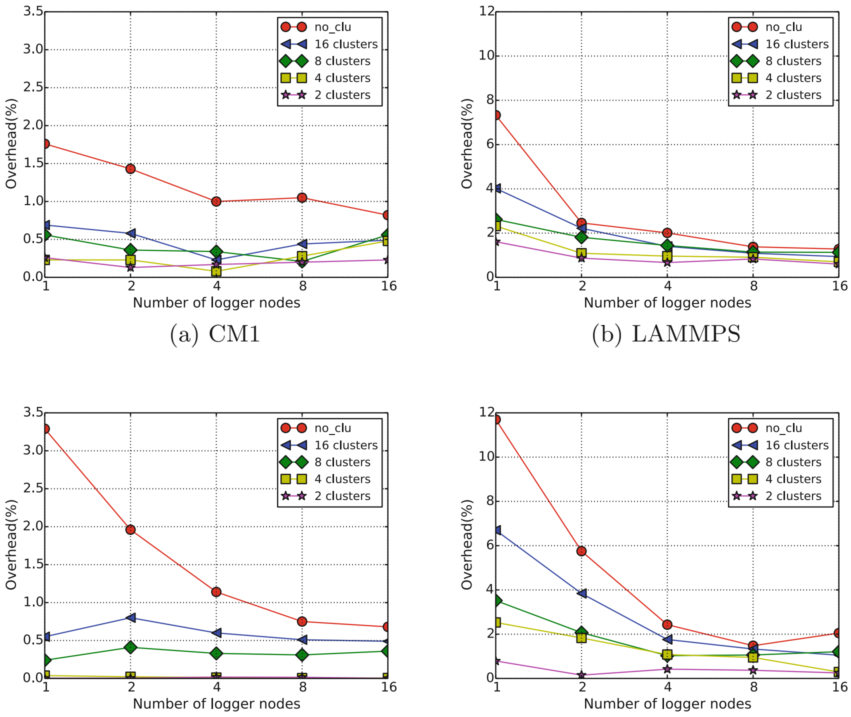
Figure 1 shows how clustering 64 processes can significantly reduce the runtime overhead of dumping message logs. For example, in MILC, the overhead of the execution with no clustering (processes log all outgoing messages) can be reduced from 11.70 % to 6.70 % either by dividing the processes into 16 clusters or by increasing the number of logger nodes to two (5.75 % overhead). For LAMMPS, the configuration with one logger node and 8 clusters gives the same runtime overhead as the one with no clustering but two logger nodes. In GTC and CM1 any configuration with clustering diminishes overhead to less than 1 %.

4.5 Dumping Overhead with Different Memory Limits

Next, we investigate how varying the memory limit for logging influences the execution. In practice, the more memory the protocol has at its disposal, the longer an application can run without having to dump logs and, consequently, the smaller will be the overhead. To quantify this, we took two applications with the highest overhead—MILC and LAMMPS—and performed several runs varying the portion of the total node memory yielded to the logging protocol from 5 to 25 % (25 % of node memory counts as 4 GB, hence 1 GB for each MPI process).

Since, upon a node crash, all the MPI processes on this node would have to restart, we did not log the messages sent between the processes residing on the same physical node; instead, we considered one node as one process cluster. We also increased the number of steps for each application to make the total log size surpass the 25 % of node memory threshold and to force some log dumping in all the tests. The tests were run with 64 processes (16 nodes) and one logger node.

As seen in Fig. 2, increasing the memory limit can help further reducing the runtime overhead, but the performance gain is not always very significant.



(a) CM1 (b) LAMMPS

(c) GTC (d) MILC

Fig. 1. Dumping overheads with different numbers of clusters and logger nodes

4.6 Use Case

Finally, we take a look at a use case that approximates a real-life application execution. We chose GTC and LAMMPS as applications that have moderate log rates and, at the same time, are relatively easy to scale. We ran them on 512 processes (128 nodes) and increased the number of steps so that they executed for 15–20 min—an optimal checkpointing period [18] for these applications considering a checkpoint cost between 30 and 60 seconds and a system with MTBF of 4 hours [8].

To find a suitable configuration of process clustering, memory quota for message logging, and number of logger nodes, we perform the following steps:

1. We run an application for several steps to record its communication pattern. We assume that the pattern stays approximately the same throughout the execution and that the log rate does not change significantly.
2. Based on the communication pattern, we get the clustering with the best balance between the log size and cluster size using a clustering tool [15], and obtain the corresponding log size.

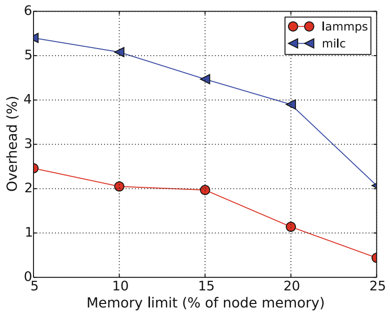


Fig. 2. Execution overhead with different memory limitations

3. Based on the predicted log size, we choose different memory quotas for the logging protocol and compute how much data in total would be flushed to loggers.
4. We choose a quota with which we would not have to use more than 10 % of additional nodes as loggers to accommodate all the flushed logs.

The resulting best configuration for LAMMPS is four clusters (128 processes per cluster), 20 % of node memory for logging protocol (3.2 GB per node), and four logger nodes to accommodate 61 GB of logs predicted to be dumped. An alternative configuration would be to limit the memory to 10 %, but in this case 16 logger nodes would be necessary. Therefore, a trade-off must be made between the memory limit and the number of logger nodes. In our case, we wanted to keep the number of loggers below 10 %, so we chose the first configuration.

The resulting configuration for GTC is four clusters, 5 % of node memory (820 MB), and three logger nodes to accommodate 47 GB of logs. Only 5 % of node memory is enough because, thanks to clustering, most of the processes do not need to log anything and only the processes at cluster border have large logs. Such clustering resulted from the specific communication pattern of GTC.

To determine the overhead, we ran tests with these configurations. The overhead is approximately 5.5 % for GTC and is 0.2 % for LAMMPS. The higher overhead for GTC is most likely due to the imbalance of log sizes between processes. As mentioned before, only the processes at cluster border had something to log, and they could have caused some desynchronization with other processes when flushing logs.

5 Conclusion

Log-based fault tolerance techniques are gaining more attention as an attractive fault tolerance solution at large scale, because such techniques allow limiting the number of processes that have to roll back and restart upon a failure. This approach is more efficient from the point of view of resource and energy usage.

The main unresolved issue of such protocols is their high memory requirement. In this work we addressed this issue and proposed using additional dedicated resources for storing part of message logs. In our tests we showed that dumping logs has a reasonable execution overhead and does not necessarily require many additional resources.

We also showed that logger nodes can be used effectively as an aid to process clustering, in order to store all the logs that exceed the available memory quota for message-logging protocol.

Future work includes evaluating the performance of recovery with logger nodes, and more generally, comparing the efficiency of the proposed approach with other state-of-the-art fault tolerant techniques.

Acknowledgements. Experiments presented in this paper were carried out using the Grid'5000 experimental testbed, being developed under the INRIA ALADDIN development action with support from CNRS, RENATER and several Universities as well as other funding bodies (see https://www.grid5000.fr).

This material is based upon work supported by the U.S. Department of Energy, Office of Science, under contract DE-AC02-06CH11357.

The submitted manuscript has been created by UChicago Argonne, LLC, Operator of Argonne National Laboratory ("Argonne"). Argonne, a U.S. Department of Energy Office of Science laboratory, is operated under Contract No. DE-AC02-06CH11357. The U.S. Government retains for itself, and others acting on its behalf, a paid-up nonexclusive, irrevocable worldwide license in said article to reproduce, prepare derivative works, distribute copies to the public, and perform publicly and display publicly, by or on behalf of the Government.

References

1. Alvisi, L., Marzullo, K.: Message logging: pessimistic, optimistic, causal, and optimal. IEEE Trans. Softw. Eng. **24**(2), 149–159 (1998)
2. Bouteiller, A., Bosilca, G., Dongarra, J.: Redesigning the message logging model for high performance. Concurrency Comput. Pract. Experience **22**, 2196–2211 (2010)
3. Bouteiller, A., Collin, B., Herault, T., Lemarinier, P., Cappello, F.: Impact of event logger on causal message logging protocols for fault tolerant MPI. In: Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS 2005), vol. 1, p. 97, April 2005
4. Bouteiller, A., Herault, T., Bosilca, G., Dongarra, J.J.: Correlated set coordination in fault tolerant message logging protocols. In: Jeannot, E., Namyst, R., Roman, J. (eds.) Euro-Par 2011, Part II. LNCS, vol. 6853, pp. 51–64. Springer, Heidelberg (2011)
5. Cappello, F., Geist, A., Gropp, B., Kale, L., Kramer, B., Snir, M.: Toward exascale resilience: 2014 update. Supercomput. Front. Innovations **1**(1), 1–28 (2014)
6. Cappello, F., Guermouche, A., Snir, M.: On communication determinism in parallel HPC applications. In: 19th International Conference on Computer Communications and Networks (ICCCN 2010) (2010)

7. Cores, I., Rodriguez, G., Martin, M., González, P.: Reducing application-level checkpoint file sizes: towards scalable fault tolerance solutions. In: 2012 IEEE 10th International Symposium on Parallel and Distributed Processing with Applications, pp. 371–378, July 2012
8. Di Martino, C., Kalbarczyk, Z., Iyer, R., Baccanico, F., Fullop, J., Kramer, W.: Lessons learned from the analysis of system failures at petascale: the case of Blue Waters. In: 2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), pp. 610–621, June 2014
9. Ferreira, K.B., Riesen, R., Arnold, D., Ibtesham, D., Brightwell, R.: The viability of using compression to decrease message log sizes. In: Caragiannis, I., Alexander, M., Badia, R.M., Cannataro, M., Costan, A., Danelutto, M., Desprez, F., Krammer, B., Sahuquillo, J., Scott, S.L., Weidendorfer, J. (eds.) Euro-Par Workshops 2012. LNCS, vol. 7640, pp. 484–493. Springer, Heidelberg (2013)
10. Jin, H., Ke, T., Chen, Y., Sun, X.H.: Checkpointing orchestration: toward a scalable hpc fault-tolerant environment. In: Proceedings of the 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, CCGRID 2012, pp. 276–283 (2012)
11. Johnson, D.B., Zwaenepoel, W.: Sender-based message logging. In: Digest of Papers: The 17th Annual International Symposium on Fault-Tolerant Computing, pp. 14–19 (1987)
12. Meneses, E., Mendes, C.L., Kalé, L.V.: Team-based message logging: preliminary results. In: Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing, CCGRID 2010, pp. 697–702 (2010)
13. Moody, A., Bronevetsky, G., Mohror, K., de Supinski, B.R.: Design, modeling, and evaluation of a scalable multi-level checkpointing system. In: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2010, pp. 1–11 (2010)
14. Riesen, R., Ferreira, K., Da Silva, D., Lemarinier, P., Arnold, D., Bridges, P.G.: Alleviating scalability issues of checkpointing protocols. In: IEEE/ACM SuperComputing 2012, SC 2012 (2012)
15. Ropars, T., Guermouche, A., Uçar, B., Meneses, E., Kalé, L.V., Cappello, F.: On the use of cluster-based partial message logging to improve fault tolerance for MPI HPC applications. In: Proceedings of the 17th international conference on Parallel processing, Euro-Par 2011, pp. 567–578 (2011)
16. Ropars, T., Martsinkevich, T., Guermouche, A., Schiper, A., Cappello, F.: SPBC: Leveraging the characteristics of MPI HPC applications for scalable checkpointing. In: IEEE/ACM SuperComputing 2013 (SC13) (2013)
17. Ropars, T., Morin, C.: Active optimistic and distributed message logging for message-passing applications. Concurrency Comput. Pract. Experience **23**(17), 2167–2178 (2011)
18. Young, J.W.: A first order approximation to the optimum checkpoint interval. Commun. ACM **17**(9), 530–531 (1974)