Dodging the Cost of Unavoidable Memory Copies in Message Logging Protocols

George Bosilca[1], Aurelien Bouteiller[1], Thomas Herault[2], Pierre Lemarinier[1], and Jack Dongarra[3]

[1] University of Tennessee
[2] University of Tennessee, Universite Paris-Sud, INRIA
[3] University of Tennessee, ORNL
{bosilca,bouteiller,herault,lemarinier,dongarra}@eecs.utk.edu

Abstract. With the number of computing elements spiraling to hundred of thousands in modern HPC systems, failures are common events. Few applications are nevertheless fault tolerant; most are in need for a seamless recovery framework. Among the automatic fault tolerant techniques proposed for MPI, message logging is preferable for its scalable recovery. The major challenge for message logging protocols is the performance penalty on communications during failure-free periods, mostly coming from the payload copy introduced for each message. In this paper, we investigate different approaches for logging payload and compare their impact on network performance.

1 Introduction

A general trend in High Performance Computing (HPC), observed in the last decades, is to aggregate an increasing number of computing elements [1]. This trend is likely to continue as thermic issues prevent frequency increase to progress at Moore's law rate, leaving massive parallelism, with hundred of thousands of processing units, as the only solution to feed the insatiable demand for computing power. Unfortunately, with the explosion of the number of computing elements, the hazard of failures impacting a long-living simulation becomes a major concern. Multiple solutions, integrated to middleware like MPI [2], have been proposed to allow scientific codes to survive critical failures, i.e. permanent crash of a computing node. Non-automatic fault tolerant approaches, where the middleware puts the application in charge of repairing itself, have proven to be, at the same time, very efficient in term of performance, but extremely expensive in terms of software engineering time [3]. As a consequence, only a small number of targeted applications are able to benefit from the *capability* of modern leadership computing centers; the typical workload of HPC centers suggests that most scientists still have to scale down their jobs to avoid failures, outlining the need for a more versatile approach.

Automatic fault tolerant approaches, usually based on rollback-recovery, can be grouped in two categories: coordinated or uncoordinated checkpointing mechanisms. Coordinated checkpointing relies on a synchronization of the checkpointing wave, an often blocking protocol, and a rollback of every process, even in the event of a single failure, which leads to a significant overhead at large scale [4]. Uncoordinated checkpointing let individual processes checkpoint at any time. As a benefit, checkpoint interval can be tailored on a per-node basis, and the recovery procedure effectively sandboxes the impact of failures to the faulty resources, with limited non disruptive actions from the neighboring processes.

However, the stronger resiliency of uncoordinated checkpointing comes at the price of more complexity, to solve the problems posed by *orphan* and *in-transit* messages. Historically, research on message logging have mostly focused on handling the costly orphan messages, by introducing different protocols (optimistic, pessimistic, causal). Recent works have nevertheless tremendously decreased the importance of the protocol choice [5], leading the once negligible overhead incurred by in-transit messages to now dominate. The technique considered as the most efficient today to replay in-transit messages is called sender-based message logging: the sender keeps a copy of every outgoing message. Although sender-based logging requires only a local copy, done in memory, and could theoretically be overlapped by actual communication over the network, it has appeared experimentally to remain a significant overhead.

The bandwidth overhead of the sender-based copy is now standing alone in the path of ubiquitous automatic and efficient fault tolerant software. In this article, we consider and compare multiple approaches to reduce or overlap this cost to a non-measurable overhead in the Open MPI implementation of message logging: Open MPI-V [6]. The rest of the paper is organized as follow: in section 2, we discuss the other approaches that have been taken, then we present the Open MPI architecture, and the different approaches to create copy of messages locally in section 3, that we compare on different experimental platforms in section 4, to conclude in section 5.

2 Related Works

Most of the existing works on message logging have focused on reducing the number of events to be logged: [7], the bottleneck of disk I/O was the main challenge in Message Logging, and the proposed solution consisted in reducing the generality of the targeted application to accept only behaviors that can be tolerated without logging messages. Other works [8, 9] reduced the kind of failures that can be tolerated to increase the asynchrony of the logging requirements, thus hoping to recover the I/O time with more computation. However, these approaches still require logging of messages, and the data can be passed back to the user application only when it has been copied completely.

To the best of our knowledge, no previous work has studied how the message payload should be logged by the sender, and how this level could be optimized. Many works have recently considered the more general issue of copying memory regions in multicore systems using specific hardware [10, 11], or how the memory management can play a significant role in the communication performance [12, 13]. However, the interactions between simultaneously transferring the data to the Network Interface Card and obtaining an additional copy in the application space has not been addressed.

3 Strategies for Sender-based Copies

Open MPI [14] is an open source implementation of the MPI-2 standard. It includes a generic message logging framework, called the PML V, that can be used for debugging [15] and fault tolerance [5]. One of the fault tolerant methods of the PML V is the pessimist message logging protocol. In this protocol, two mechanisms are used: event logging and sender-based message logging. The event logging mechanism defuse the threat on recovery consistency posed by orphan messages, those who carry a dependency between the non deterministic future of the recovering processes and the past of the survivors. The outcome of every non deterministic event is stored on a stable remote server; upon recovery, this list is used to force the replay to stay in a globally consistent state. In this paper, we focus our efforts on improving the second mechanism, message payload copy, thus we do not modify the event logging method. The necessity of the sender-based message logging comes from in-transit messages, *i.e.* messages sent in the past of the survivors but not yet received by the recovering processes. Because only the failed processes are restarted, messages sent in the past from the survivors can not be regenerated. The sender-based message logging approach keeps a memory copy of every outgoing message on the sender, so that any in-transit message is either regenerated (because the sender also failed and therefore is replaying the execution as well), or is readily available.

There are mostly two parameters governing the payload logging: 1) the backend storage system, and 2) the copy strategy from the user memory to the backend storage system. We have designed three backend storages: a) a file that is mapped in memory, b) heap memory as backend, allocated using memory mapping of private anonymous memory, and c) a dummy backend storage, that does not implement message logging, but provides us a mean to measure the overhead due only to the copy itself. We have also designed three copy methods: a) a pack method, that copy the message in one go into the backend space, b) a convertor method, that chops the copy of the message according to the Open MPI pipeline, and c) a thread method, that creates an independent thread responsible of doing the copies. In the following, we describe with more details these strategies.

Backend storage
Memory Mapped file. It should be noted that there is no necessity for the log to be persistent: if a process crashes, it will restart in its own history, and recreate the messages that have been logged after the last checkpoint (still, messages preceding the last checkpoint must be saved with the checkpoint image, because they are part of the state of the process). However, a file backend is natural, because the volume of message to be logged can be significant, and this should not reduce the amount of memory available for the application. Mapping the backend file into memory is the most convenient way of accessing it.

We designed this backend file as a growing storage space, on which we open a moving window using the mmap system call. When the window is too small to accept a new message (we use windows of 256 MB, unless some message exceed the size of the window), we wait that all messages are logged (depending on the copying method, described later), make the file grow if necessary, and move the window entirely to a free area of the file.

Heap Memory. If the amount of memory available on the machine is large enough to accept at the same time the application and the copy of the messages payload (up to garbage collection time), then the payload logging can be kept in memory. This second method uses anonymous private memory allocated with the mmap system call to create such a backend for our message logging system.

Dummy Storage. In order to measure independently the overhead introduced by the copy method itself, we also designed a Dummy Storage that does not really implement message logging: after a message is logged, the pointer to store the message payload is moved back to the beginning of the same memory area, reallocated if the size of the message is larger than the largest message seen until the call. When messages are sent often, the pages related to this area will most likely be present in the TLB, and for very short messages, it is even possible that the area itself remains in the CPU cache between two emissions. Though this storage cannot be considered as a backend storage for message logging, it helps us evaluate the overheads of the copy methods themselves, without considering other parameters like TLB misses and pages fault.

Copy method

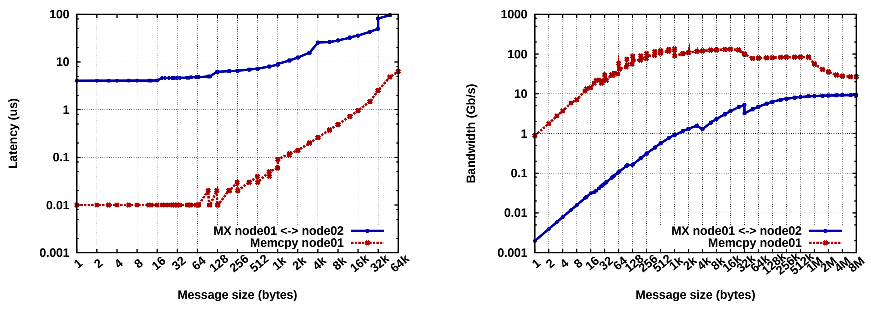
Pack. The Pack method consists in copying the payload of the message using the memcpy libc call, from the user space to the backend storage space, when the PML V intercepts the message emission for the first time. This interception can happen just after the message has been given to the network card for short messages, or just after the first bytes of the message have been given to the network card, and the network card cannot send more without blocking for longer messages.

Conv. When converting the user data to a serialized form usable by the network cards, the Open MPI data type engine can introduce a pipeline, to send multiple messages of a predetermined maximal size on the network cards, instead of sending a very large single message. Up to four messages can be given to the network card simultaneously, which will send one after the other. The data type engine tries to keep this pipeline as filled as possible, to ensure that the network card has always something to send. Using the Conv method, PML V intercepts each of these, and introduces the message payload copy at this time. This is what the Conv (short for convertor) method does: if the pipeline is enabled, each time a chunk of data is copied from the user data to the network card, the PML V copies the same amount of bytes from the user data to the backend storage. The size of the chunks in the pipeline is a parameter of this method.

Thread. The last copying method is based on a thread. A copying thread is created during the initialization. This thread waits on a queue for copies. When this queue is not empty, the thread pops the first element of the queue, and copies the whole user memory onto the backend storage, using the memcpy libc call. When a message emission is intercepted by the PML V, if the message is short, it is copied as for the Pack method. If the message is long enough and could not be sent to the network in one go, a copy request is created and pushed at the end of the request queue. When the application returns from the MPI call, it synchronizes with the copy thread, and waits that the related messages have been entirely logged before returning from the MPI call, to ensure message integrity. To ensure a fair comparison, at constant hardware resources, this thread is pinned on the same core as the MPI process that produces the message.

4 Experimental Evaluation

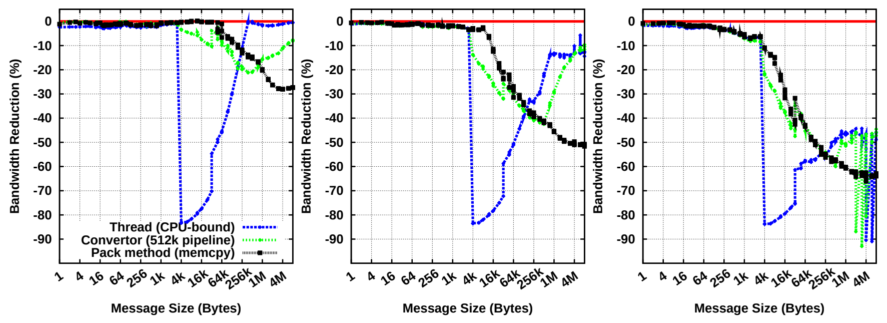
The Dancer cluster is a small 8 node cluster, each node based on a Intel Q9400 2.5Ghz quad core processor, with 4GB of memory. All nodes are connected using a dual Gigabit Ethernet links, and four feature an additional Myricom MX10G. Linux 2.6.31.2 (CAoS NSA 1.29) is deployed. The software is compiled using gcc 4.4 with -O3 flags, and uses the trunk of Open MPI (release 21423) modified to include the different logging techniques presented in Section 3. For every run, we forced Open MPI to use only the high-speed Myricom network of dancer using the MCA parameters: `-mca btl mx,self`. All latency and bandwidth measurements were obtained using the MPI version of the NetPIPE-3.7 benchmark [16]. NetPIPE evaluates latency and bandwidth by computing three times the average value on a varying number of iterations, and taking the best value of the three evaluations.



(a) Latency (b) Bandwidth

Fig. 1. Reference MPI MX NetPIPE performance between two dancer nodes compared to memcopy

Figure 1 presents the reference latency (Fig. 1(a)) and bandwidth (Fig. 1(b)) of Open MPI on the specified network, and of the memory bus of the machines used. These figures are presented here as a absolute reference of peak performance achievable without message logging. A first observation is that the high



(a) Ideal Storage Memory (b) Anonymous Memory (c) File Map

Fig. 2. NetPIPE MX bandwidth between two dancer nodes, according to the storage method

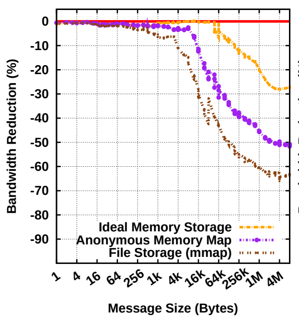
memory bandwidth and low latency compared to the High-Speed network card should enable a logging in memory with little performance impact for messages of less than 1MB. For larger messages, the bandwidth of the memory bus will become a bottleneck for the logging, and unless the time taken to transfer the message on the network can be recovered by the logging mechanism, overheads are to be expected.

A few characteristics of the underlying network and the Open MPI implementation can moreover be observed from these two figures: one can clearly see the gaps in performance for messages of 4KB (default size of the MX frame), and 32KB (change of communication protocol from eager to rendez-vous in the Open MPI library). In the rest of the paper, all other measurement will be presented relative to the bandwidth performance of the high-speed network card, to highlight the overheads due to message logging.

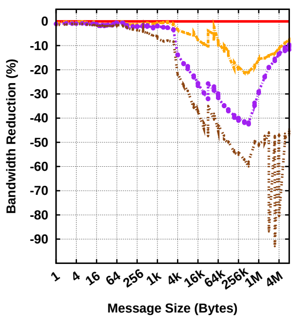
Each of the first figures grouped under Figure2 consider a specific storage medium, and compare for a given medium the overheads of the different logging methods as function of the message size.

First, we consider Figure 2(a) that uses as a storage medium the "Ideal" Storage. As described in Section 3, the Ideal storage uses a single memory area to log all the messages (thus overriding existing log with new messages). The goal of this experiment is to demonstrate the overheads due to the copy itself (and when it happens) without other effects, like page faults, etc... One can see that the logging method has no significant impact up to (and excluding) messages of 4KB. At 4KB, the Thread method suffers a huge overhead that decreases the performance by 80%, while the other methods suffer a lower overhead.

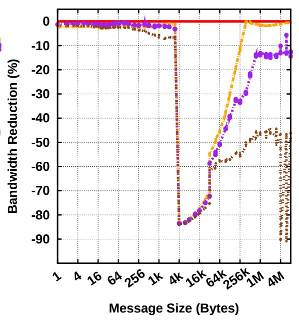
A single MX frame is of 4KB (on this platform). Thus, for messages of 4KB of payload, or more, multiple MX frames are necessary to send the message (this is true for messages of 4KB of payload too, since the message header must also be sent). When the message fits in a single frame, the logging thread can be scheduled while the message circulates on the network and is handled by the receiving peer. When the message doesn't fit in a single MX frame, the Open MPI engine requires scheduling to ensure the lowest possible latency. Since both



(a) Pack Method



(b) Convertor Method



(c) Thread Method

Fig. 3. NetPIPE MX bandwidth between two dancer nodes, according to the copy method

threads are bound on the same core, they compete for the core, and the relative performances decrease.

On one hand, when the number of frames needed for a single message is low, the MPI thread and the logging thread must alternate with a high frequency on the core (since the MPI call exits only when the message has been sent and logged). On the other hand, when the number of frames needed is high, the thread that is scheduled on the CPU can either log the whole message in one quantum, or use all available frames in the MX NIC to send as much data as possible in one go. Thus, when the number of frames increases, the relative overhead due to the logging thread decreases.

The pack method decreases almost linearly with the message size, since all copies are made sequentially after the send. Because the network is eventually saturated, the relative overhead reaches a plateau. The Convertor method uses a pipeline of 512KB. Thus, until messages are 512KB long, it behaves similarly as the Pack method. The difference is due to a slightly better cache re-use from the Pack method that send the message, then logs it, instead of first logging it during the pack operation, then sending it on the network. When the messages size is larger than the pipeline threshold, the Convertor method introduces some parallelism (although not as much as the Thread method), that is used to recover the communication time with logging time.

The other two figures (2(b) and 2(c)) demonstrate a similar behavior before 4K, although the overheads begin to be notifiable a little sooner for all methods when logging on a File. This is due to file system overheads (inodes and free blocks accounting), and memory management (TLB misses) when more pages are needed to log the messages. When the file system is effectively used (messages of 4MB and 8MB end up consuming all available buffer caches of the file system), a high variability in the relative overhead becomes observable (Figure 2(c)).

These phenomenon are more observable on the second group of three figures under Fig. 3. These figures consider each a specific logging method, and expose the impact of the medium on the overheads due to a logging method, as function of the message size. As can be seen, using a mmaped file as a storage space introduces the highest overhead, significantly higher than the overhead due to in-memory storage, even when the kernel buffers of the file system are large enough to hold this amount of data. This is due to accounting in the file system (free blocks lists, inodes status), forced synchronization of the journaling information, and a conservative policy for the copy of the data to the file system.

The difference of overhead between an anonymous memory map (in the heap of the process virtual memory), and the Ideal storage space is mainly due to TLB misses introducing additional page reclaims. This cost is unavoidable to effectively log the messages, but it is small for small messages, and amortized for very large messages. As a consequence, logging should happen in memory as long as the log can be kept small enough to fit there, and the system should resort to mmaped files only when necessary.

Figures 2(b) and 2(c) lead us to the conclusion that an hybrid approach, with different thresholds depending on the storage medium, and on the message size, should be taken: up to messages of 2KB, the method has little influence, however after this, the Pack or the convertor methods should be preferred up to messages of 128KB. For messages higher than 128KB, the use of an asynchronous thread, even if it must share the core with the application thread, is the preferred method of logging.

5 Conclusion

In this paper, we studied three techniques to log the payload of messages in a sender-based approach, in the Open MPI PML V framework that implement message logging fault-tolerance. Because the copying of the message payload must be achieved before the corresponding MPI emission is complete (either when the blocking send function exits, or when the corresponding wait operation exits), copying this payload is a critical efficiency bottleneck of any message logging approach.

One of the techniques proposed is to use an additional thread to process the copying asynchronously with the communication; a second uses the pipeline installed by the Open MPI communication engine to interlace transmissions towards the network, and copies in memory; the third simply copy the payload after it has been sent, and before the completion of the communication at the application level.

We also demonstrated that the medium used to store the payload has a significant impact on the performances of the payload logging process. We concluded that depending on the medium for storage, and the message size, different strategies should be chosen, advocating for a hybrid approach that will have to be tuned specifically for each hardware.

References

1. Meuer, Werner, H.: The top500 project: Looking back over 15 years of supercomputing experience. Informatik-Spektrum **31**(3) (2008) 203–222

2. The MPI Forum: MPI: a message passing interface. In: Supercomputing '93: Proceedings of the 1993 ACM/IEEE conference on Supercomputing, New York, NY, USA, ACM Press (1993) 878–883
3. Graham E. Fagg and Edgar Gabriel and George Bosilca and Thara Angskun and Zhizhong Chen and Jelena Pjesivac-Grbovic and Kevin London and Jack J. Dongarra: Extending the MPI specification for process fault tolerance on high performance computing systems. In: Proceedings of the International Supercomputer Conference (ICS) 2004, Primeur (2004)
4. Lemarinier, P., Bouteiller, A., Herault, T., Krawezik, G., Cappello, F.: Improved message logging versus improved coordinated checkpointing for fault tolerant MPI. In: IEEE International Conference on Cluster Computing (Cluster 2004), IEEE CS Press (2004)
5. Bouteiller, A., Ropars, T., Bosilca, G., Morin, C., Dongarra, J.: Reasons to be pessimist or optimist for failure recovery in high performance clusters. In IEEE, ed.: Proceedings of the 2009 IEEE Cluster Conference, New Orleans, Louisiana, USA (2009)
6. Bouteiller, A., Bosilca, G., Dongarra, J.: Redesigning the message logging model for high performance. In: Proceedings of the International Supercomputer Conference (ISC 2008), Dresden, Germany, Wiley (2008) to appear
7. Strom, R.E., Bacon, D.F., Yemini, S.: Volatile logging in n-fault-tolerant distributed systems. In Society, I.C., ed.: Proceedings of the Eighteenth International Symposium on Fault Tolerant Computing. (1988)
8. Strom, R.E., Yemini, S.: Optimistic recovery: an asynchronous approah to fault-tolerance in distributed systems. In: Proceedings of the 14th International Symposium on Fault-Tolerant Computing, IEEE Computer Society (1984)
9. Manivannan, D., Singhal, M.: A low-overhead recovery technique using quasi-synchronous checkpointing. Distributed Computing Systems, International Conference on **0** (1996) 100
10. Vaidyanathan, K., Chai, L., Huang, W., Panda, D.K.: Efficient asynchronous memory copy operations on multi-core systems and i/oat. In: CLUSTER '07: Proceedings of the 2007 IEEE International Conference on Cluster Computing, Washington, DC, USA, IEEE Computer Society (2007) 159–168
11. Goglin, B.: Improving message passing over ethernet with i/oat copy offload in open-mx. In: Proceedings of the 2008 IEEE International Conference on Cluster Computing, IEEE (2008) 223–231
12. Stricker, T., Gross, T.: Optimizing memory system performance for communication in parallel computers. In: ISCA '95: Proceedings of the 22nd annual international symposium on Computer architecture, New York, NY, USA, ACM (1995) 308–319
13. Geoffray, P.: Opiom: Off-processor i/o with myrinet. Future Generation Comp. Syst. **18**(4) (2002) 491–499
14. Gabriel, E., Fagg, G.E., Bosilca, G., Angskun, T., Dongarra, J.J., Squyres, J.M., Sahay, V., Kambadur, P., Barrett, B., Lumsdaine, A., Castain, R.H., Daniel, D.J., Graham, R.L., Woodall, T.S.: Open MPI: Goals, concept, and design of a next generation MPI implementation. In: Proceedings, 11th European PVM/MPI Users' Group Meeting, Budapest, Hungary (2004) 97–104
15. Bouteiller, A., Bosilca, G., Dongarra, J.: Retrospect: Deterministic replay of mpi applications for interactive distributed debugging. In: Proccedings of the 14th European PVM/MPI User's Group Meeting (EuroPVM/MPI). (2007) 297–306
16. Snell, Q.O., Mikler, A.R., Gustafson, J.L.: Netpipe: A network protocol independent performance evaluator. In: in IASTED International Conference on Intelligent Information Management and Systems. (1996)