Evaluation of Simple Causal Message Logging for Large-Scale Fault Tolerant HPC Systems

Esteban Meneses
Department of Computer Science
University of Illinois
Urbana, Illinois, USA
emenese2@illinois.edu

Greg Bronevetsky
Center for Applied Scientific Computing
Lawrence Livermore National Laboratory
Livermore, California, USA
greg@bronevetsky.com

Laxmikant V. Kalé
Department of Computer Science
University of Illinois
Urbana, Illinois, USA
kale@illinois.edu

***Abstract*—The era of petascale computing brought machines with hundreds of thousands of processors. The next generation of exascale supercomputers will make available clusters with millions of processors. In those machines, mean time between failures will range from a few minutes to few tens of minutes, making the crash of a processor the common case, instead of a rarity. Parallel applications running on those large machines will need to simultaneously survive crashes and maintain high productivity. To achieve that, fault tolerance techniques will have to go beyond checkpoint/restart, which requires all processors to roll back in case of a failure. Incorporating some form of message logging will provide a framework where only a subset of processors are rolled back after a crash. In this paper, we discuss why a simple causal message logging protocol seems a promising alternative to provide fault tolerance in large supercomputers. As opposed to pessimistic message logging, it has low latency overhead, especially in collective communication operations. Besides, it saves messages when more than one thread is running per processor. Finally, we demonstrate that a simple causal message logging protocol has a faster recovery and a low performance penalty when compared to checkpoint/restart. Running NAS Parallel Benchmarks (CG, MG, BT and DT) on 1024 processors, simple causal message logging has a latency overhead below 5%.**

***Keywords*-causal message logging; pessimistic message logging; migratable objects; parallel applications.**

I. Introduction

The growing size and complexity of supercomputers is clearly illustrated by the last few releases of Top500 list [1]. What was considered a big machine few years before, is today one medium-sized supercomputer. Unfortunately, the rise in size has not been accompanied by an increase in the reliability of the components. The net result is that as they grow in size, supercomputers become less reliable. The era of petascale computing brought machines with hundreds of thousands of processors. The next generation of exascale supercomputers will make available clusters with millions of processors [2], [3]. In those machines mean time between failures will range from a few minutes to few tens of minutes [2]–[4], making the crash of a processor the common case, instead of a rarity.

In an environment where failures are unavoidable, large-scale parallel applications will need to simultaneously survive crashes and maintain high productivity. To achieve this goal, particularly if the failure rate is high, fault tolerance techniques will be required to go beyond the traditional checkpoint/restart approach. Message logging protocols present a promising alternative to checkpoint/restart because they do not require a global rollback. Instead, only the crashed processor is brought back to the previous checkpoint, while the other processors may keep making progress or wait for the recovering processor in a low-power state. Alternatively, recovery can be accelerated by having a parallel restart mechanism [5].

Although message logging mechanisms have been evaluated in other contexts, we believe that literature is lacking a comparison of two particular versions of sender-based message logging for high scale HPC systems. The first version of message logging is the traditional pessimistic approach, where communication is stalled until meta-data about messages is safely stored. The advantage of this method is its simplicity of implementation, including the garbage collection for old messages and the recovery process. On the other hand, its main drawback is an increased latency for communication. Collective operations, common in parallel appications, only exacerbate this problem. The second approach is called causal message logging. In this method, communication is not delayed but meta-data is piggybacked on top of application messages. Therefore, we think it is a more promising method for message logging in highly scalable HPC applications. The paper aims to evaluate a simple causal message logging protocol, trying to highlight its strenghts and weaknesses for the different types of applications.

The contributions of this paper are i) an analysis of various scenarios that make pessimistic message logging compromise the performance to keep a consistent execution, ii) a performance comparison of pessimistic and causal approaches for message logging with different applications, and iii) a performance evaluation of the simple causal message logging protocol for applications that scale up to 1024 processors.

This paper is organized in the following sections. We start by presenting the relevant related work in Section II. We describe the pessimistic approach for message logging in Section III. This is typically the preferred implementation of message logging, since it provides a simple abstraction and has low cost during recovery. This protocol, however, can perform poorly under some circumstances. Section IV describes various scenarios where the pessimistic approach

has a high overhead. Section V presents the causal message logging method. We discuss why we believe a simple version of this protocol is a promising alternative to the pessimistic approach. To support that claim, Section VI shows the experimental results about performance of the two message logging techniques. We evaluate several applications, trying to understand what cases make the superiority of causal message logging more dramatic. Finally, we leave conclusions and future work for Section VII.

II. Related Work

This section provides a background on the different contributions about message logging on parallel applications.

A seminal paper in message logging is by Strom and Yemini [6]. They describe a system that permits communication, computation and checkpoint to proceed asynchronously, thus introducing the concept of message logging and causality tracking. Their approach is called optimistic recovery and it is based on receiver-side message logging plus a component to detect the causal relationships between the recovery units. They introduced the concept of *orphan* as a process that is not consistent with the rest of the system after a failure. Also, they pioneered the piecewise deterministic (PWD) assumption, which states that logging all non-deterministic decisions is enough to provide a correct recovery.

Sender-based message logging was introduced by Johson and Zwaenepoel [7]. In their paper they describe a system that logs messages using a pessimistic approach. Every message is piggybacked with a unique *send sequence number* (SSN). The goal of SSN is to discard duplicate messages during recovery. Before sending a message, the sender will ask the receiver for a *receive sequence number* (RSN) that will form a total order of message reception at the destination. In other words, each process will deliver the received messages in accordance to the RSN assigned to each of them.

Manetho [8] was the first system to use causal message logging. They assume an application execution is composed by a sequence of piecewise deterministic state intervals. The main structure to keep track of in Manetho is called the *antecedence graph*. This is a directed acyclic graph that contains the causal relationships between the deliveries of all the messages in the application. Application messages carry sufficient information to reconstruct the antecedence graph on recovery. In case of a process failure, the other processes can use their portions of the antecedence graph to help the crashed process to reconstruct the execution.

Alvisi and Marzullo [9] present a classification of the different message logging schemes into three families: pessimistic, optimistic, and causal. The first approach requires stalling the message transmission until the sender receives the RSN. In this way, the message is logged along with what is called the *determinant*, i.e., the tuple $\langle senderID, receiverID, SSN, RSN \rangle$. The optimistic approach does not delay message transmission, but incurs in the risk of producing orphan processes in case of a failure. Causal message logging is a compromise where message transmission is not delayed, but orphans are avoided. However, it relies on piggybacking additional information on application messages. Depending on the causal message logging protocol, this overhead can be significant.

The first implementation of a causal protocol for high performance computing (as far as the authors know) is due to Bouteiller *et al* [10] in MPICH-V library. They built three different causal protocols: Manetho, LogOn and Vcausal. The first two use a dependence graph whereas the last only propagates determinant information. The main point of their paper is to argue that a centralized entity (called Event Logger) provides good performance for any of the protocols. However, the scalability of the centralized Event Logger is questionable.

An interesting study regarding the tradeoff of piggybacked information in causal message logging is by Bhatia *et al* [11]. In their paper, they compare different strategies according to the amount of piggybacked information. The whole goal is to survive the crash of f concurrent failures by having every determinant replicated $f + 1$ times. Then, additionally to piggyback a determinant, some extra information can be included. For instance, every determinant can carry the number of replicas that have been made along its route to the destination (similar to a time-to-live value). The main conclusion of the paper is that including too much information about the determinants to prevent them to be over-replicated can be intolerable.

Prior work on evaluating the performance of causal message logging has been limited in various ways such as focusing on systems with few nodes or omitting an evaluation of causal message logging. Elnozahy and Zwaenepoel [8] evaluated the Manetho causal message logging protocol, comparing it to checkpointing and pessimistic logging on a 16-processor cluster with 10Mb/s Ethernet. They showed that in their setup coordinated checkpointing outperforms message logging and argue that they should be combined to support interactions with the outside world. Rao et al [12] evaluated the cost of recovery in optimistic, pessimistic and causal message logging protocols on an 8-processor system with 100Mb/s Ethernet, showing that for single failures pessimistic and causal logging perform best and for multiple failures the best performance is achieved by receiver-based protocols that also have a higher failure-free overhead. Lemarinier et al [13] compare the performance of causal and pessimistic logging to Chandy-Lamport coordinated checkpointing [14] on a 25-processor cluster with upto 700Mb/s networks. They found that the two logging protocols have similar performance and predict that message logging will be more efficient than checkpointing if the mean time between failures is shorter than 9 hours. Finally, Boutellier et al [15] compare the overhead of pessimistic and optimistic logging in the context of the OpenMPI implementation of MPI on upto 256 processes on 1Gb/s and 10Gb/s networks. They find that when communication is logged at the level of entire MPI messages there is little performance difference between the two protocols, with less than 10% overhead on the NAS Parallel Benchmarks [16].

The following sections describe our implementation of pessimistic and causal message logging for the Charm++ runtime

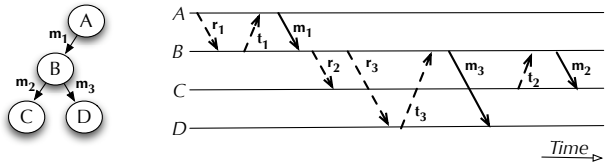

Fig. 1. Pessimistic Message Logging.

system [17]. Charm++ is a model for parallel computing that is based on over-decomposing the computation and data into objects. A smart runtime system is in charge of distributing and migrating these objects in accordance with the performance characteristics of the underlying machine. In Charm++ objects are naturally migratable and this feature makes checkpoint and fault tolerance in general easier to incorporate into the system. Furthermore, an MPI program can be run on top of the Charm++ infrastructure thanks to Adaptive MPI (AMPI) framework [18]. Then, we can provide fault tolerance to an MPI application by using the Charm++ runtime system.

III. Pessimistic Message Logging

This section describes the common alternative when implementing a message logging scheme for fault tolerance in HPC [19]. Pessimistic message logging provides strong guarantees in terms of correctness after a failure and it is in general easier to understand than other methods. We have implemented a version of this protocol that we called *simple* pessimistic message logging, since it only guarantees successful recovery after the crash of one processor. The concurrent failure of multiple processors is not always tolerated by the scheme.

A. Forward Path Protocol

The essential property of pessimistic message logging is that no message is delivered until both the message and its determinant are safely stored. There are several alternatives for saving messages and determinants: stable storage, a reliable component or the memory of other process.

Our protocol saves each application message and its determinant in the sender. This implies that a process, before sending a message, has to ask the receiver for a *ticket* (the reception sequence number) to compose the determinant for that message. The determinant and the message are stored in the memory of the sender and at that point the message can be actually sent to the receiver. Messages at the receiver are processed according to their assigned ticket number and on recovery ticket numbers can be used to recreate the reception order of all messages.

Figure 1 shows an example of how the protocol works. Let us assume processes $\{A, B, C, D\}$ form part of a system and they are organized as a broadcast tree rooted at process A, as shown on the left part of the figure. Before A actually sends message m_1 to B it has to ask B for the receiption ticket that m_1 will receive at B. We can see how this happens on the right part of the figure. The request for the ticket of m_1 is represented by r_1 and its reply is denoted by t_1. Once this exchange is done, message m_1 can be sent. The same protocol applies for messages m_2 and m_3. However, notice that this protocol runs asynchronously with the rest of the execution. In other words, B sends r_2 and then r_3 without waiting for the ticket of the first request to arrive. Ticket t_3 can arrive before t_2 and thus message m_3 is sent before m_2.

This protocol works with either coordinated or uncoordinated checkpointing. For either case, every processor A has a *buddy* processor assigned to it. The buddy processor of A will be in charge of storing the checkpoint of A in its own memory. In case A needs the checkpoint after a failure, its buddy will have to provide the checkpoint. Although saving checkpoints in memory increases the memory pressure, it significantly speeds up checkpoint storage and recovery from checkpoints.

B. Garbage Collection and Recovery

Every process logs all messages sent to any other process, which can become a significant storage overhead. However, because the application takes periodic checkpoints there is always a bound on how far it can roll back. Any messages that precede this bound do not need to be stored and can be discarded. For instance, process A sends message m to B. B processes m and then it checkpoints. At that point, the copy of message m held by A is unnecessary. If B crashes, the data of m will not be useful in recovering the state of B since it was received before B's last checkpoint. Thus, to garbage collect the message logs in every process, we follow a simple procedure. After every checkpoint, a process will send a message to all its senders. This message contains the highest ticket number that is part of the checkpoint. Upon reception, the senders will remove from the message log all messages with a smaller or equal ticket number. The garbage collection runs concurrently with the execution of the application. Clearly, this protocol is only guaranteed to tolerate a single concurrent failure. For instance, if A sends message m to B and both crash before B's next checkpoint, then both the message and the meta-data are lost.

Whenever a failure is detected, a failure message is sent to every process. Each process then resends all messages sent to the failed process before it failed. Thus, when it is recovering, the failed process will receive the replayed messages and process them in their original order using their ticket numbers. This ensures it will reconstruct the exact same state it had before the crash. Also, a restarting processor may send *spurious* messages during recovery, i.e., messages that were sent before and have been processed by the receivers. Those messages are easily detected since their ticket numbers will be lower than than the last processed ticket number, and will be discarded.

IV. Limitations of Pessimistic Message Logging

While pessimistic message logging is relatively simple, there are various scenarios where it performs poorly. Specifically, pessimistic message logging suffers from 3 main drawbacks. First, it introduces latency overhead that may affect the performance of collective operations. Second, in practice

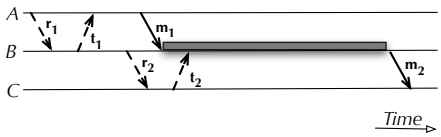

Fig. 2. Delay in Pessimistic Message Logging.

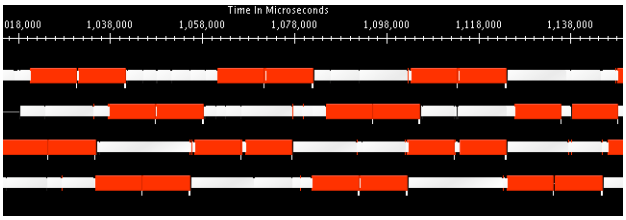

Fig. 3. Alternated Execution.

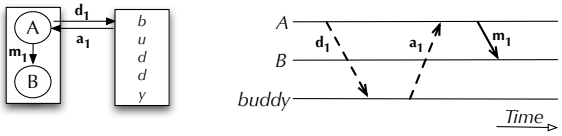

Fig. 4. Overhead of local messages.

the multi-step handshake necessary to send a message will be even slower due to interference from the application. Finally, when multiple processes are running on the same processor information about their messages needs to be communicated to other processors.

A. Increased Latency

One of the big limitations of pessimistic message logging is the extra latency it adds to every message transmission. This overhead corresponds to a message round-trip. This problem is most significant for collective operations. Since collectives are usually implemented through a structure (spanning tree, hypercube, mesh), the additional latency impacts each level. For instance, in figure 1 the link between A and B adds some extra latency, as well as the links between B and C and B and D. At the end, the total extra latency added by the protocol has to be multiplied by the levels of the tree or by the diameter of a more general structure.

The extra delay introduced by any message sent in pessimistic message logging may or not may be a problem for an application. For molecular dynamics applications, which rely on strong scaling of time-steps and small messages, the overhead will be intolerable. However, more latency-tolerant applications may still perform well with this protocol. Regardless, we cannot expect to have a competitive protocol for those sections in the application where collective operations are heavily used.

B. Interference

Another potential problem with the pessimistic approach is that for some applications it may break the natural synchronization among the processes. To illustrate this case we will refer to figure 2. Remember from figure 1 when B sends request r_2, it already has message m_2 ready to be sent, but it has to wait for t_2 to arrive. In the meantime, it may receive another message that triggers a function at B (represented in figure 2 by a narrow rectangle). While the function is executing, t_2 may arrive at B but it will be unable to send message m_2 until the function completes. This situation further delays processing of message m_2 at C.

We experienced exactly this problem when running the pessimistic message logging protocol in Charm++ with a 7-point stencil application. Since in Charm++ a function triggered by a message is usually non preemptive, the function must finish execution until the scheduler can process messages. Most single-threaded MPI implementations are implemented in this way. Figure 3 shows a visualization produced by a performance analysis tool for the 7-point stencil program. We can see 4 horizontal bars representing the execution of four different processors. White spaces stand for idle time, where processors wait until the reception of the next message. The type of pattern of the figure corresponds to an *alternated* execution where a set of processors execute a function, then other subset of processors execute the function and they rarely overlap in computation. We detected that the scenario of figure 2 was causing this performance problem.

One solution to the previous problem is to decrease the grainsize of the computation. Systems like Charm++ permit overdecomposing the computation into objects, potentially having several objects per processor. An implementation of the pessimistic protocol has shown some benefits [5]. However, having several objects executing in the same process can aggravate other problems. For instance, a finer decomposition will send more messages for the same problem size, increasing the memory overhead of message logging [20].

C. Local Communication

By virtualizing a processor we may have multiple objects living on the same processor and continuously exchanging messages. In this case, we have to take extra precautions in pessimistic message logging since the minimum unit of failure is a processor. Let us call a message *local* when it is exchanged between objects or processes executing on the same processor. We will refer a message as *remote* when it is sent to another processor. To provide consistent recovery we must log the determinant of every *local* message in another processor, presumably the *buddy* of the processor where the objects reside. Thus, we must incur additional inter-process communication for every local message. Figure 4 illustrates this case. If A and B are processes or objects living on the same processor and A wants to send a message to B, it must first send the determinant of m_1 to the buddy processor and wait for the acknowledgment.

V. Causal Message Logging

To overcome the three main limitations explained in the previous sections, we consider the causal approach for imple-

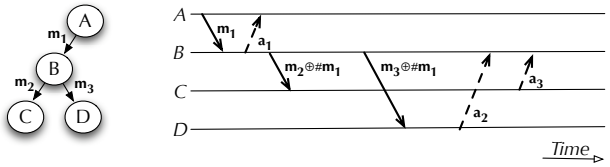

Fig. 5. Causal Message Logging.

menting message logging. This method provides lower latency for communication but incurs in bandwidth overhead. Causal message logging requires piggybacking determinants on top of application messages. This section describes this protocol and the major properties that make it a promising option for scalable fault tolerance. We have implemented a *simple* variant that only guarantees recovery from a concurrent single failure.

A. Forward Path Protocol

The intuition behind causal message logging is that the determinant of an event is only needed by processes that depend on that event. For instance, suppose a process A performs several non-deterministic actions, sends messages to processes B and C and then fails. When A restarts, processes B and C are the ones most interested in A executing its non-deterministic events exactly as before because their state depends on the outcomes of these events. In contrast, if A performed any additional non-deterministic actions that were not communicated to any other process, they can be discarded since no functioning process depends on them. As such, causal logging protocols piggyback determinants of events on outgoing messages so that they are stored by their receivers.

Figure 5 summarizes our simple variant of the protocol. It presents the same scenario as figure 1, where A is sending a broadcast to all other processes through a spanning tree. In this protocol, A does not need to delay in sending message m_1 to B. However, B has to acknowledge all message receptions so that if the message had any determinants, A would know when they were safely stored by B. Upon reception of message m_1, B generates a determinant for m_1, that we will denote by $\#m_1$. This determinant has to be safely stored somewhere else other than B's memory. It will be piggybacked in the next message sent by B. Notice that message m_2 piggybacks the determinant $\#m_1$ and it will piggyback that determinant on every outgoing message until it receives the acknowledgement from some other process that confirms that the determinant has been stored. Specifically, $\#m_1$ is piggybacked onto m_3 before B receives the acknowledgement from D. We must emphasize that B only needs one confirmation to assume a determinant is safely stored. This is due to the fact that we are dealing with the *simple* version of the protocol. Multiple acknowledgments about the same determinant enable the protocol to survive multiple concurrent failures.

B. Garbage Collection and Recovery

Similarly to pessimistic message logging, each processor has a *buddy* that will store the checkpoint and from where we will retrieve the last saved state of a failed processor. In the causal message protocol it is more complex to remove old messages from the message log than the pessimistic case. This is because the message reception ticket for message m is stored on the one or more recipients of messages causally after m. As such, after processor A checkpoints, it sends all the determinants to the senders, for all them to match the list and delete the old messages from their message logs. In the next section we will discuss a possible optimization that will remove the burden of garbage collection in causal message logging.

To recover from the crash of process A, the protocol requires the data and message tickets of messages sent to A. Thus on recovery, all processors that sent messages to A will replay those messages and processors that store determinants regarding A will send the determinants back to A. Once A receives messages and determinants, it will match them up to deliver messages in the same order as before the crash. Analogous to the pessimistic model, repeated messages are discarded. To detect a repeated message we need a different strategy than in the pessimistic model. Every process keeps track of the messages it has received from any other source. If a message matches a previously received combination of sender ID and message ID, it must be a repeated message.

VI. Results

This section describes the major findings in our implementation of the simple causal message logging protocol. We evaluate several different applications to understand the behavior of the protocol under different circumstances.

A. Experimental Setup

We used the Charm++ runtime system to implement the pessimistic and causal message logging and compare their performance. In all the cases we used the standard checkpoint/restart technique as a baseline. In Charm++, checkpoint/restart is implemented in memory. In this way, every processor checkpoints in the memory of two processors: itself and a *buddy*. This mechanism has a higher memory consumption for the checkpoint, but it uses little communication to transfer checkpoints and once the first checkpoint is taken, memory does not grow as part of the fault tolerance protocol. In all the applications, we use synchronization points to checkpoint. This method allows message logging to avoid a complex garbage collection protocol, since at after every checkpoint, all data structures can be cleaned up.

The machine used in these experiments is called Abe and it is installed at the National Center for Supercomputer Applications (NCSA). Abe is an Intel 64 cluster with a peak speed of 89.47 TFLOPS. It contains 1200 dual-socket nodes with quad-core processors, for a total of 9600 processors. Each node has a total 8 GB of memory or 1GB per processor. Although Abe has Infiniband network connection, we based all our experiments on the Ethernet layer. We injected failures by killing the Charm++ process running on a given core and

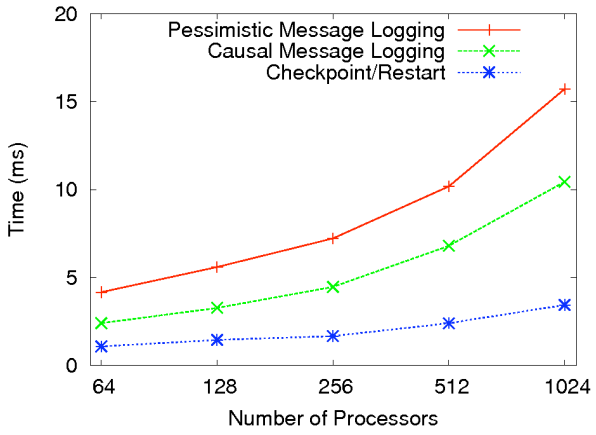

Fig. 6. Performance comparison in collective benchmark.

that way all the work on that core. The runtime system will detect the failure and react accordingly.

B. Experiments

We will first present a comparison between pessimistic and causal message logging, trying to address the main issues in the performance evaluation. Secondly, we will examine more closely the behavior of causal message logging and the way it works under different circumstances.

We start by comparing the two message logging protocols with checkpoint/restart on a benchmark that performs collective communication and no computation. This collective benchmark repeatedly performs a broadcast of a single integer from one processor, immediately followed by a reduction of the integer. In our implementation these two collective operations are implemented in different fashion and we where trying to exercise both of them. This program executes the same step (broadcast followed by reduction) 100 times and reports the average after removing the first iterations to avoid outliers.

Figure 6 shows the results of the three methods and their performance for this program, running on different processor counts, from 64 to 1024. We can see causal message logging does a better job than pessimistic, but still has some considerable overhead. This is due to the piggyback of determinants. Since the message for the collectives only carries a payload of one integer, the determinant piggybacked is larger than the payload itself. We are dealing here with the extreme case for causal message logging, where no computation takes place, and the performance penalty is severe. The causal slowdown goes from a factor of 2.21 at 64 processors to 3.03 at 1024 processors.

Next, we use a more realistic scenario to compare the protocols. This time we focus on the performance of the NAS Parallel Benchmarks. In particular we chose CG, MG, BT and DT to make this comparison since they represent four different parallel dwarfs or parallel programming kernels. Using 256 processors, we present the execution time in figure 7, normalized to each benchmark with checkpoint/restart.

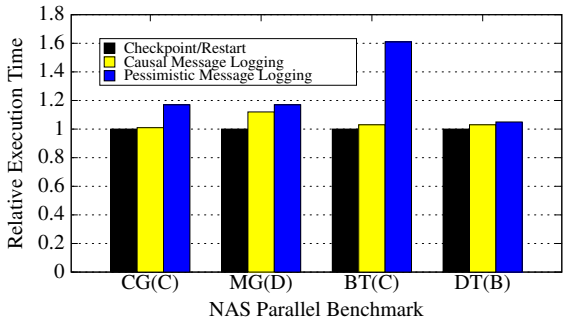

Fig. 7. NPB Benchmarks Relative Performance Results (256 processors).

Causal message logging performs better than pessimistic in all four cases. For the CG benchmark causal's overhead is 1% whereas pessimistic's is 17%. MG offers a more comparable scenario for overheads of 12% and 17% for causal and pessimistic, respectively. BT simply breaks down pessimistic for an overhead of 61%, not comparable to the 3% of causal. Finally, DT has a small difference between the approaches, for causal having 3% overhead and pessimistic 5%.

To better understand the results of figure 7, we show in tables I and II various properties of the different benchmarks. Table I offers a view into the bandwidth consumption of these programs. It presents a list of measurements per processor. The first two lines display the total number of messages sent by one processor and the total number of messages sent per second, respectively. In this regard, it is clear that CG and BT are heavy in communication, whereas MG seems to be more computation bound. DT has very little communication. Table I continues by showing the average size of each message. Benchmarks CG, MG and BT are comparable, while DT sends large messages. The next lines of table I present the total number of determinants generated by processor and the total number of determinants piggybacked. The last row has the total number of determinants piggybacked per message. We see that, among the benchmarks that show considerable communication, MG has the highest value in this respect and that hurts the performance of causal message logging, since it is the variable that directly increases its penalty.

TABLE I
BANDWIDTH CONSUMPTION

Benchmark	CG	MG	BT	DT
Messages	25890	8122	19340	7
Messages/s	260.67	35.45	213.14	0.33
Message Size (KB)	28.49	34.43	23.78	4608.59
Determinants	26036	8518	19382	12
Piggybacked	57045	23292	41787	29
Determinants/msg	2.20	2.86	2.16	4.16

On the other hand, table II shows the memory overhead of causal message logging. For each benchmark we can see the total size of checkpoint. DT has a minute checkpoint, while both CG and BT have relatively small checkpoint size. MG has a much larger checkpoint size. The second row presents the

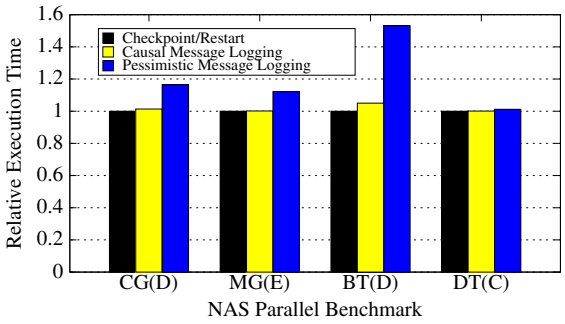

Fig. 8. NPB Benchmarks Performance Results (1024 processors).

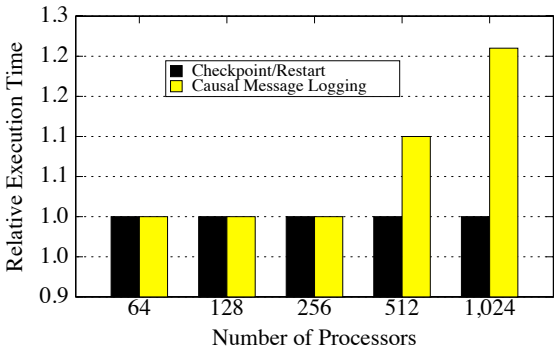

Fig. 9. Strong scaling 7-point stencil.

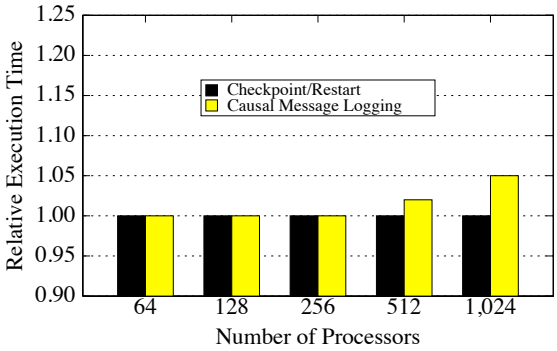

Fig. 10. Weak scaling 7-point stencil.

total size of the message log. In this case, CG and BT have a larger message log since both are communication bound. The final two rows of table II refer to the amount of memory needed to store determinants. The *local* determinants refer to those determinants created by the processor. They must be stored at that processor, since they may be piggybacked again in case of a failure. The *remote* determinants refer to those determinants coming from other processors and that also need to be stored in case of a failure. We must highlight that determinant size is measured in KB, meanwhile message log size is measured in MB. Two orders of magnitude separate the two sets. Thus, determinants will rarely create memory pressure in causal message logging.

TABLE II
MEMORY CONSUMPTION

Benchmark	CG	MG	BT	DT
Checkpoint (MB)	13.63	121.25	14.43	1.08
Message Log (MB)	98.45	49.48	91.83	35.41
Local Determinants (KB)	250.12	109.13	278.59	0.49
Remote Determinants (KB)	548.75	297.90	600.53	1.64

To finish the comparison of the two message logging protocols, we offer in figure 8 the numbers for a 1024 processors run with the same set of NAS Parallel Benchmarks. Although pessimistic message logging has an overhead as high as 53% on BT, causal message logging never exceeds a 5% overhead.

We measured the scaling properties of the simple causal message logging protocol by using a 7-point stencil program that computes the values of a 3-dimensional grid. The program is written in Charm++ and has two main parameters, n the size of one side of the cubic space (in other words, the total space has dimensions nxnxn) and b the size of the block. The program divides the total space into blocks of dimensions bxbxb and creates as many objects as blocks in the program. The runtime system divides evenly the objects into the processors. Each object communicates with its 6 nearest objects in the simulation space. The way the program progresses is by computing one iteration and then performing a global reduction to check for convergence. For simplicity, we are only reporting the time taken in the first 200 iterations.

Figure 9 presents the results of the strong scaling test with processor numbers ranging from 64 to 1024. In all those cases we are solving a space of 1024x1024x1024 points with different block sizes. There is no noticeable difference in the first three data points. However, at 512 processors the overhead of causal message logging is 10%. At this point, communication starts to take over the program, since computation is diluted into the extra processors we have available. Thus, the benchmark slowly converges to the scenario in figure 6 where communication dominates the execution time. This is more clear at 1024 processors where the overhead of causal reaches 21%. Then, although causal does not stall the communication, collectives still remain a challenge, since the determinant piggybacking will exceed the cost of executing the collective itself.

On the other hand, figure 10 presents the results for a weak scaling test. The number of processor goes from 64 to 1024. This time, however, each processor will always do the same amount of work. Each processor will solve a space of 256x256x256. At 64 processors both figures 9 and 10 coincide. Beyond that point the behavior is very similar in the two figures. The major difference is that overhead at 512 processors is 2%, while at 1024 processors is 5%. We see the impact of weak scaling on causal message logging. As long as computation does not diminish as communication cost increases, we can manage to have a low overhead.

Finally, figure 11 presents the progress of the 7-point stencil application in case of a failure. We define progress as the total number of iterations completed. We compare the two approaches: checkpoint/restart and causal message logging.

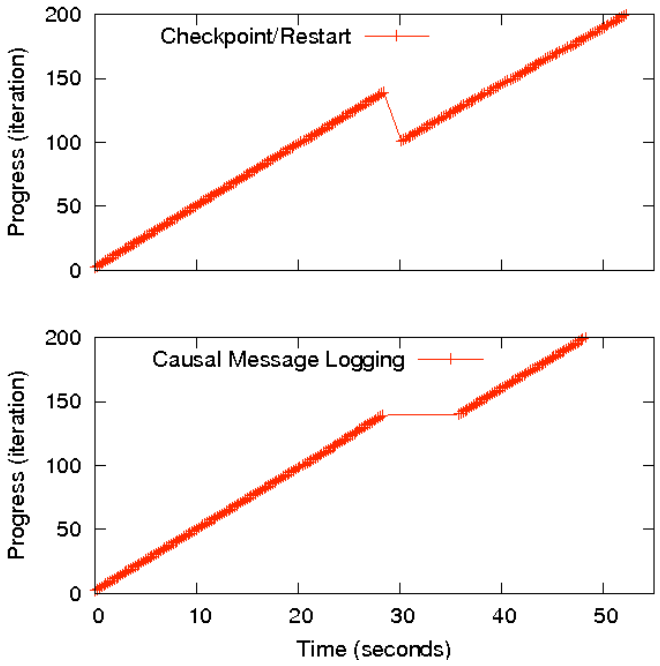

Fig. 11. Effect of a failure on 7-point stencil.

In both cases, we introduced a failure at iteration 140, out of the 200 total iterations simulated. There is one single checkpoint at iteration 100. The top curve shows the way checkpoint/restart deals with the failure. Since all processors roll back to the previous checkpoint, all processes roll back to the last checkpoint and re-execute all computation and communication. The bottom curve shows the behavior of causal message logging, where only one processor rolls back, repeating its own computation and communication, while the other processors operate at low power. Ultimately, causal message logging is faster than checkpoint restart because it omits global reductions during restart and is more power-efficient because only one processor re-executes lost work.

VII. Conclusions and Future Work

We analyzed the performance of simple causal message logging and compared the performance with pessimistic message logging technique. Our evaluation identifies multiple performance problems of pessimistic message logging and shows that causal logging has better performance and scalability for all the programs we ran in our experiments. There are however, challenges for causal message logging. Specifically, it imposes a higher latency on communication, which can be a problem for strong scaling and collective operations, and it requires a modest amount of additional memory to store determinants.

For the future we plan to extend our causal message logging to the multicore world, where a node will be the minimum unit of failure. We also plan to have a strategy for correlated multiple concurrent failures. Since failures of different system components correlate according to the machine's topology, we will exploit this fact to design efficient protocols that tolerate the multiple failures with a very high probability.

Acknowledgments

This research was supported in part by the US Department of Energy under grant DOE DE-SC0001845 and by a machine allocation on the Teragrid under award ASC050039N. This work was partially performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344.

References

[1] "Top500 supercomputing sites," http://top500.org.
[2] E. Peter Kogge, "Exascale software study: Software challenges in extreme scale systems," Tech. Rep., 2009.
[3] ——, "Exascale computing study: Technology challenges in achieving exascale systems," Tech. Rep., 2009.
[4] B. Schroeder and G. A. Gibson, "Understanding failures in petascale computers," *Journal of Physics: Conference Series: SciDAC*, vol. 78, 2007.
[5] S. Chakravorty and L. V. Kale, "A fault tolerance protocol with fast fault recovery," in *Proceedings of the 21st IEEE International Parallel and Distributed Processing Symposium*. IEEE Press, 2007.
[6] R. Strom and S. Yemini, "Optimistic recovery in distributed systems," *ACM Trans. Comput. Syst.*, vol. 3, no. 3, pp. 204–226, 1985.
[7] D. B. Johnson and W. Zwaenepoel, "Sender-based message logging," in *In Digest of Papers: 17 Annual International Symposium on Fault-Tolerant Computing*. IEEE Computer Society, 1987, pp. 14–19.
[8] E. N. Elnozahy and W. Zwaenepoel, "Manetho: Transparent roll back-recovery with low overhead, limited rollback, and fast output commit," *IEEE Trans. Comput.*, vol. 41, no. 5, pp. 526–531, 1992.
[9] L. Alvisi and K. Marzullo, "Message logging: pessimistic, optimistic, and causal," *Distributed Computing Systems, International Conference on*, vol. 0, p. 0229, 1995.
[10] A. Bouteiller, B. Collin, T. Herault, P. Lemarinier, and F. Cappello, "Impact of event logger on causal message logging protocols for fault tolerant mpi," in *IPDPS'05*, 2005, p. 97.
[11] K. Bhatia, K. Marzullo, and L. Alvisi, "The relative overhead of piggybacking in causal message logging protocols," in *SRDS '98: Proceedings of the The 17th IEEE Symposium on Reliable Distributed Systems*. Washington, DC, USA: IEEE Computer Society, 1998, p. 348.
[12] S. Rao, L. Alvisi, and H. M. Vin, "The cost of recovery in message logging protocols," *IEEE Trans. Knowl. Data Eng.*, vol. 12, no. 2, pp. 160–173, 2000.
[13] P. Lemarinier, A. Bouteiller, T. Herault, G. Krawezik, and F. Cappello, "Improved message logging versus improved coordinated checkpointing for fault tolerant mpi," *Cluster Computing, IEEE International Conference on*, vol. 0, pp. 115–124, 2004.
[14] K. M. Chandy and L. Lamport, "Distributed snapshots : Determining global states of distributed systems," *ACM Transactions on Computer Systems*, Feb. 1985.
[15] A. Bouteiller, T. Ropars, G. Bosilca, C. Morin, and J. Dongarra, "Reasons for a pessimistic or optimistic message logging protocol in mpi uncoordinated failure, recovery," in *CLUSTER*, 2009, pp. 1–9.
[16] D. B. E. B. J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, S. Fineberg, P. Frederickson, T. Lasinski, R. Schreiber, H. Simon, V. Venkatakrishnan, and S. Weeratunga, "The NAS parallel benchmarks," NASA Ames Research Center, Tech. Rep. RNR-04-077, 1994.
[17] L. Kalé and S. Krishnan, "CHARM++: A Portable Concurrent Object Oriented System Based on C++," in *Proceedings of OOPSLA'93*, A. Paepcke, Ed. ACM Press, September 1993, pp. 91–108.
[18] C. Huang, O. Lawlor, and L. V. Kalé, "Adaptive MPI," in *Proceedings of the 16th International Workshop on Languages and Compilers for Parallel Computing (LCPC 2003), LNCS 2958*, College Station, Texas, October 2003, pp. 306–322.
[19] A. Bouteiller, F. Cappello, T. Hérault, G. Krawezik, P. Lemarinier, and F. Magniette, "MPICH-V2: A fault tolerant MPI for volatile nodes based on the pessimistic sender based message logging programming via processor virtualization," in *Proceedings of SC'03*, November 2003.
[20] C. L. M. Esteban Meneses and L. V. Kale, "Team-based message logging: Preliminary results," in *3rd Workshop on Resiliency in High Performance Computing (Resilience) in Clusters, Clouds, and Grids (CCGRID 2010).*, May 2010.