Team-Based Message Logging: Preliminary Results

Conference Paper · January 2010

DOI: 10.1109/CCGRID.2010.110 · Source: DBLP

3 authors, including:

Esteban Meneses
Costa Rican Institute of Technology (ITCR)
37 PUBLICATIONS **190** CITATIONS
SEE PROFILE

Laxmikant V. Kalé
University of Illinois, Urbana-Champaign
203 PUBLICATIONS **2,790** CITATIONS
SEE PROFILE

Team-based Message Logging: Preliminary Results

Esteban Meneses, Celso L. Mendes and Laxmikant V. Kalé
Department of Computer Science
University of Illinois at Urbana-Champaign, Urbana, Illinois, USA
{emenese2, cmendes, kale}@illinois.edu

Abstract

Fault tolerance will be a fundamental imperative in the next decade as machines containing hundreds of thousands of cores will be installed at various locations. In this context, the traditional checkpoint/restart model does not seem to be a suitable option, since it makes all the processors roll back to their latest checkpoint in case of a single failure in one of the processors. In-memory message logging is an alternative that avoids this global restoration process and instead replays the messages to the failed processor. However, there is a large memory overhead associated with message logging because each message must be logged so it can be played back if a failure occurs. In this paper, we introduce a technique to alleviate the demand of memory in message logging by grouping processors into teams. *These teams act as a failure unit: if one team member fails, all the other members in that team roll back to their latest checkpoint and start the recovery process. This eliminates the need to log message contents within teams. The savings in memory produced by this approach depend on the characteristics of the application, the number of messages sent per computation unit and size of those messages. We present promising results for multiple benchmarks. As an example, the NPB-CG code running class D on 512 cores manages to reduce the memory overhead of message logging by 62%.*

1. Introduction

With the advent of clusters that comprise hundreds of thousands of cores and with the exascale era on the horizon, where that number will reach millions, one thing becomes obvious: we will need to deal with failures. At that scale, designing applications, middleware and operating systems that tolerate the loss or misbehavior of one component is a fundamental requirement.

The traditional way to cope with processor failure in the High Performance Computing (HPC) community has been to use checkpoint/restart. The state of the application is saved with a certain frequency (which typically depends on the mean time between failures of the system) and, in case of a failure, the application rolls back to the previous checkpoint and resumes from there. Although this approach is commonly used, it has a serious disadvantage: all the processes have to roll back even when only one out of the million cores failed.

Another approach, known as message logging, reduces this cost by saving more than the state of the application. Message logging also stores every message sent during the execution. Traditional formulations saved the message data in stable storage, but in our optimized implementation the data is saved in main memory and periodically checkpointed to disk. In the case of a failure, only the process that failed is rolled back. All other processes in the system may send their logged messages to the failed process and continue executing. This approach presents various benefits. First, processors that do not fail can continue their execution until they need data from a failing processor. Second, a processor that requires data from a failed processor may use less energy as it waits for the data to arrive. Third, because of our virtualized infrastructure in Charm++ [8], the activity in the failing processor can be restarted in parallel on the remaining processors.

The main drawback of message logging is its high overhead in terms of memory usage. Since the messages have to be stored in main memory to avoid an I/O operation for every message and the corresponding increase in latency, the message log increases the memory pressure. Sharing the same memory between the user data and the message log might disturb the data locality of the algorithm employed by the application. Besides, as the system memory is exhausted more quickly, it forces a more frequent checkpointing.

Another problem of message logging is the potentially expensive protocols needed to guarantee that the recovery is correct, leaving no inconsistent processes. One way to meet this requirement is to implement a protocol where every message is tagged with a number specifying its reception order, via a handshake with the receiver. Enforcing these

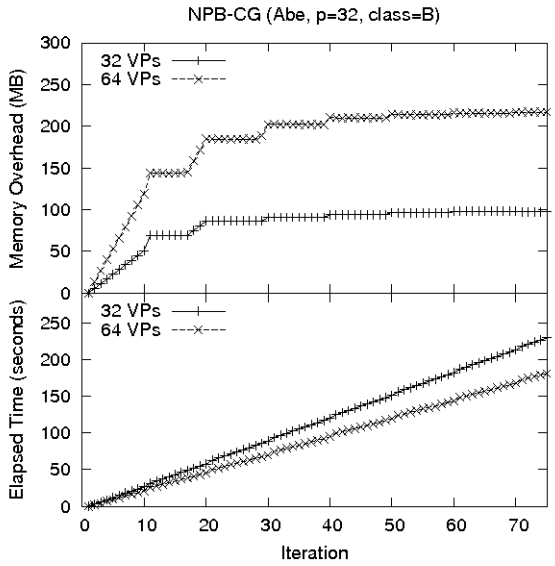

Figure 1. Memory overhead of higher virtualization ratio.

ticket numbers increases the latency in the communication, demanding new strategies to reduce the impact on application performance. For instance, in AMPI [6], processor virtualization is used to alleviate this problem. In Figure 1 we see the behavior of the NPB-CG class B benchmark running on 32 processors on an x86-based cluster. The horizontal axis presents the iterations executed by the program, while the vertical axis at the top is the memory overhead and at the bottom is the elapsed time. We used the AMPI version of these benchmarks. In AMPI, every MPI process is implemented as a lightweight user-level thread, allowing the creating of a number of virtual processors (VP) different from the number of physical processors. Using only 32 virtual processors (VP = 32) the program executes in 232.31 seconds. If we use 64 virtual processors (VP = 64) running on the same 32 physical cores, we manage to improve the execution time to 183.64 seconds. However, this change also causes an increase in memory overhead, from 97.40 MB to 216.92 MB. As a result, increasing the virtualization ratio in NPB-CG decreases execution time, but increases the memory pressure.

We propose a new method to decrease the memory footprint of message logging by grouping the processors into *teams* that will act as a unit when logging messages and during recovery. The main idea is that only messages that go across teams are stored in the log, while all the messages internal to the team are only ticketed to get their reception order, but are never logged, i.e. only their metadata is stored. Also, if one of the processors in the team fails, all the processors in the team roll back and start the recovery. Thus, this scheme represents a significant improvement over the original message logging approach: while there is a slightly higher roll back overhead in case of a failure, there is also a concrete memory usage reduction in the forward path.

This paper starts by presenting a review of message logging strategies in Section 2. In Section 3, we introduce the fundamentals of team-based message logging and describe what data structures are required to provide a correct recovery. The results of several experiments are offered in Section 4. Finally, conclusions and future developments are left for the final section.

2. Background

Message logging has been a very well known fault tolerance strategy since several decades ago. In this section we survey the main contributions to this area.

The seminal paper by Strom and Yemini [10] presented a recovery protocol for distributed systems based on message logging and dependency tracking. Their protocols avoid the undesirable *domino effect* (which causes multiple processors to roll back after the failure of another processor) while asynchronously allowing computation, communication, checkpointing and committing. They also introduced the Piecewise Deterministic (PWD) assumption, which states that all non-deterministic events can be recorded in a log (along with the data for those events) and this is enough to recreate these events in the same order during recovery. In this way, the failed processor can be brought up to date to a consistent state.

Johnson and Zwaenepoel [7] introduced the concept of sender-based message logging, where every sent message is logged in *volatile* memory on the sender's machine. Their paper describes the data structures required to secure the correctness of the protocols. All these structures revolve around two different tickets associated with every message: the send sequence number and the receive sequence number. These two numbers conform the metadata associated with a particular message. They presented two basic protocols and discussed the issues about *orphan* processes. We say a process becomes orphan if during recovery its state is not consistent with the rest of the processes anymore.

Manetho [5] was the implementation of these ideas in a system that was able to reduce the protocol overhead (in the case of a failure-free execution), providing at the same time limited rollback and fast output commit. One key component of this system was the *antecedence graph*, which was in charge of recording the happens-before relationship between certain events in the computation.

Alvisi and Marzullo [1] described the three big families of message logging algorithms: pessimistic, optimistic and causal. A message logging algorithm is said to be pes-

simistic if a particular message is sent whenever there is certainty that the message and all previously delivered messages have been logged. This restriction increases the latency to send messages, since sending a message becomes more cumbersome. On the other hand, an optimistic algorithm does not wait until the message is logged to send it, creating the possibility of having orphan processes during recovery. This type of algorithm offers better performance in the fault-free case but may have to cascade back during recovery, potentially leading to unbound rollbacks and requiring multiple past checkpoints to be stored. The third group is the causal family, which does not create orphans during recovery and does not need the process to wait to send a message. However, its implementation is much more complicated.

Several implementations of message logging protocols over Message Passing Interface (MPI) library have been developed by Cappello *et al.* [2, 3, 9]. The well known MPICH-V included the three different protocols (pessimistic, optimistic and causal) into the same library and it was one of the first fault tolerant distributions of MPI.

3. Team-based Approach

In this section we describe a new method to reduce the memory consumption of message logging schemes called *team-based* message logging.

3.1. Basic Method

In Charm++, Chakravorty and Kalé [4] implemented a pessimistic sender-based message logging scheme. Charm++ is an object oriented parallel programming language based on C++ classes. It provides a methodology in which the programmer decomposes the data and computation in the program without worrying about the number of physical processors on which the program will run. The runtime system is in charge of distributing those objects among the processors. The message logging protocol is also implemented at the runtime level. Each object stores all the messages it sends and keeps track of metadata using two structures. The `SequenceTable` stores the number of messages the object has sent to every other object in the system. To guarantee the execution of the messages in order, every message has to have a ticket number, assigned by the receiver. So, every object γ also keeps a `TicketTable` that logs which ticket is associated with a combination of sender and sequence number for each message sent to γ.

Figure 2 shows the process to produce the metadata before sending a message. In this case, an object living on processor A has to send a message to another object on processor B. Before the actual send, a ticket is requested from A and generated by B. Once the ticket has been received

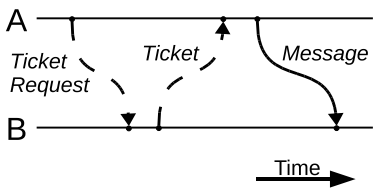

Figure 2. Pessimistic sender-based message logging.

by A, the object on A is able to send the message. This increase in the latency of communication is the price one pays for correctness in case of a failure. During recovery, the messages are processed in the same order as they were ticketed.

3.2. Reducing Memory Overhead

The intuition of the proposed method is very simple: group the processors into *teams* and avoid logging intra-team messages. Only messages across teams will be logged. However, if there is a failure in one processor of a particular team, all the members of that team have to roll back to their previous checkpoint. In this way, we trade a reduction in the message log size for an increase in work at recovery time.

As straightforward as it seems, there are many complexities to consider in the implementation of this method. Before sending a message from one object to another, we have to ask for a ticket number, regardless of the team affinity of the two parts involved. This ensures that messages are replayed in the same sequence during recovery. Metadata is always stored, even for messages within a team, but the message itself might not be stored depending on what teams the sender and receiver belong to. Along with all the data structures described above, we included an extra data structure to keep all the information regarding intra-team communication. We named this structure the `TeamTable`.

The `TeamTable` stores metadata of messages to members of the same group. It has the same architecture as the `TicketTable`, i.e., it associates a combination of object and sequence number with a ticket number. Moreover, the `TeamTable` is also sender-based. This means that the information stored in this data structure corresponds to the intra-team messages *sent* by the object. For instance, if object α in processor A sends a message to object β in processor B, and both A and B belong to the same team, then object α will store the tuple $[< \beta, S >, T]$ in the `TeamTable`, where S and T are the associated sequence and ticket number for that particular message.

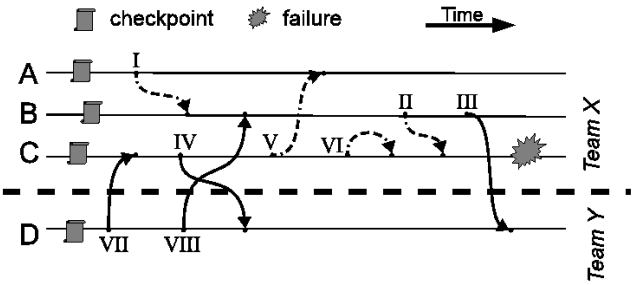

Figure 3. The team-based message logging approach.

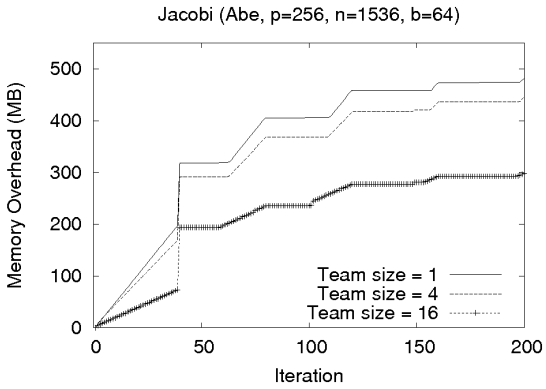

Figure 4. Memory overhead in Jacobi with different team sizes.

Saving these tuples is necessary to be able to make a correct recovery. Since the only source of non-determinism in Charm++ is the message reception order, we need to guarantee that after a crash, the resent messages are processed in the same order as before. Figure 3 depicts the different cases that occur in practice. In the figure we see two teams, X and Y. Processors A, B and C belong to team X, while processor D belongs to team Y. The messages represented by solid arrows will be logged, since they cross the team boundary. All other messages depicted by dotted arrows are not logged because they are exchanged by objects on processors of the same team. Now, suppose the processors checkpoint and then a few messages are sent before processor C crashes. All types of messages, from *I* to *VIII* need to be replayed. Let us consider them case by case.

Since A, B and C are all part of the same team, after C crashes A and B return to their previous checkpoint. Assuming spare processors, C is also restarted from its last checkpoint. Given that C crashed, all its team information is lost. The good news is all the other members of the team roll back with it but maintain the `TeamTable` intact. This means that metadata about messages II and V is available at processors B and A, respectively. Message I is regenerated during the recovery and its metadata is kept in both A and B. A special case appears with message VI, which is a local message. All local messages save their metadata in another processor, known as the *buddy*. Thus, every processor has a buddy processor that stores its checkpoint and metadata related to local messages.

On the other hand, messages across groups are sent again. Message VII is stored in the log of processor D, as is message $VIII$. Messages III and IV are regenerated during recovery and the `TicketTable` in processor D contains all the metadata to make sure these messages are ignored once they are received by D during recovery.

We keep the team size constant throughout the execution of the application. Given p processors, we divide them into teams of t processors each. Thus, the team size in the team-based approach offers a tradeoff between the checkpoint-restart mechanism and the traditional message-logging. For instance, if t is equal to p we are in front of the checkpoint-restart approach, whereas if t equals 1 then we are dealing with the traditional message logging scheme.

Although we implemented the team-based approach using a pessimistic sender-based message logging, here we claim it can be adapted to any other message logging scheme. The intuition is to make the recovery unit not a single processor, but a set of processors. Even if the protocol uses an optimistic or causal model, teams are suitable to decrease the memory overhead.

4. Experimental Results

We start this section by presenting the different benchmarks used in the experiments and describing the machine where all the runs were made. Following that, we show several results obtained with the new approach.

4.1. Benchmarks

We used the Abe cluster at NCSA (National Center for Supercomputer Applications) to run all our experiments. Abe consists of 1200 blades Dell PowerEdge 1955, each blade having 8 cores for a total of 9600 cores. Every blade (or node) is comprised of two Intel 64 (Clovertown) 2.33 GHz dual socket quad core chips. A total of 8 GB is available per node (1 GB per core). Although Abe has both Infiniband and Ethernet interconnects, we only used Ethernet: the available software layers supporting Infiniband are not convenient for fault-tolerance experiments, because

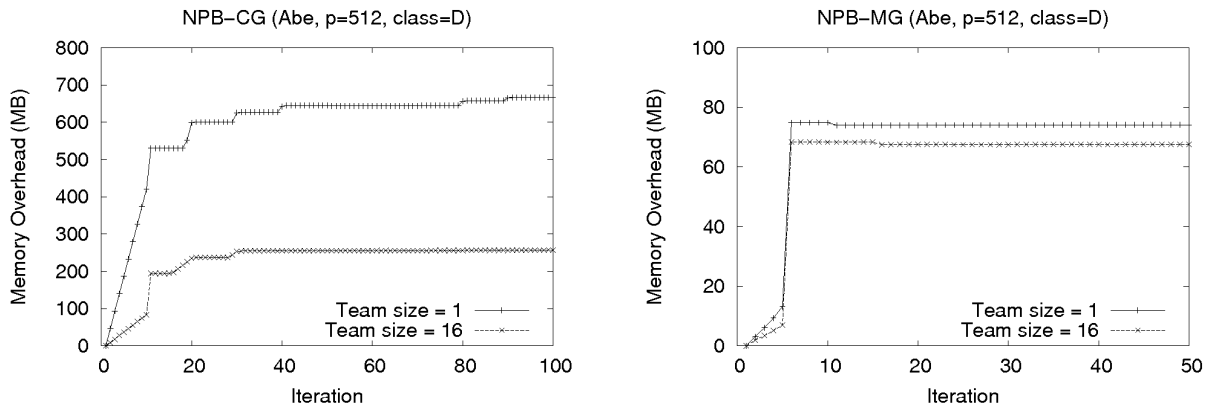

Figure 5. Memory overhead in NPB-CG and NPB-MG with two team sizes.

they automatically kill the entire job when one of the processes goes down.

Three different applications were selected as benchmarks for this project. The first is a Jacobi relaxation program, which is a stencil computation. It is written in Charm++ and uses a three dimensional space. In every iteration, each cell will update its value according to the values of the six neighboring cells. The number of iterations is specified by the user. We use a division of the matrix into blocks and assign blocks to different processors. The two parameters for this benchmark are n, the size of one side of the matrix, and b, the size of the block.

Thanks to the ability of Charm++ to execute MPI programs using the Adaptive-MPI (AMPI) extension [6], we were able to test some of the NPB benchmarks. Two different programs were tested for the experiments, CG and MG. The Conjugate Gradient (CG) benchmark estimates the largest eigenvalue of a sparse symmetric matrix using an iterative method. On the other hand, Multi Grid (MG) approximates the solution of a discrete three dimensional Poisson equation. The class D problem was tested for both codes.

4.2. Results

Using the Jacobi benchmark, we obtained the results in Figure 4. The test was conducted on Abe with $p = 256$ processors. The configuration for the Jacobi problem was $n = 1536$ and $b = 64$, which means the array of objects had dimensions $(24, 24, 24)$. The plot shows three different team sizes, ranging from 1 to 16. We see that there is a benefit in using teams, but that improvement is very similar in the cases of team sizes 2, 4 and 8. The memory overhead goes from 485 MB to 451 MB, which is a reduction of 8%. However, if we use a team size of 16 then the memory footprint reduces even further to 298 MB, meaning a reduction of 39%. The reason for this sudden change in the memory consumption is that the default mapping in Charm++ uses a particular distribution of the blocks into processors that is not aware of the communication network and does not cluster together, necessarily, objects that exchange a lot of messages. In this case, after crossing the threshold of 16 processors, the benefits become more evident, since the nearest neighbor communication pattern of Jacobi and the object distribution makes a higher fraction of the messages to be transferred in the same team. Studying the effects of different object assignments is part of our ongoing work.

Figure 5 presents at the left the results with the NPB-CG benchmark class D. In this case, 512 cores were reserved on Abe to run 512 different MPI processes. In other words, the virtualization ratio is 1. Two different team sizes were used in this case. Going from a team size of 1 to 16 makes a huge difference for this benchmark. The memory overhead of message logging plummets from 666 MB to 256 MB, for a savings of 62%. Part of the big reduction comes from the fact that NPB-CG is known to be communication bound and then it makes a lot of sense to use the team-based approach since it will save a significant amount of memory.

The NPB-MG benchmark was also run and produced a significantly different results than NPB-CG. The right hand side of Figure 5 corresponds to the NPB-MG memory overhead. As NPB-MG is computation bound, the total reduction of the memory overhead came down from 77 MB to 67 MB, for a reduction of just 13%. This is not a substantial reduction, and it illustrates that the utility of the team approach depends on the characteristics of the benchmark. Particularly, NPB-MG does not allow for a big reduction, given its communication pattern.

Finally, the last experiment was designed to measure the recovery time after a crash. Using the Jacobi application

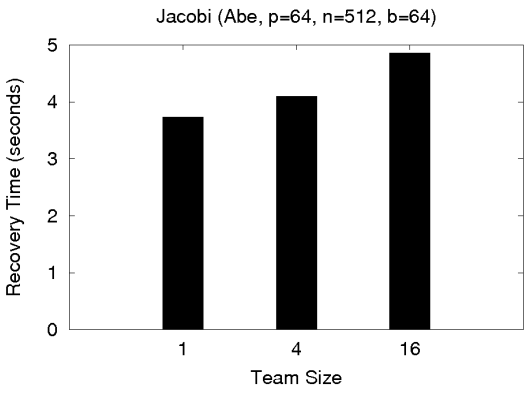

Figure 6. Recovery time in Jacobi for different team sizes.

with 64 processors, we inserted a processor failure after the first checkpoint and measure how much time was required to recover the failed processor. Figure 6 presents the recovery time for three different group sizes. When using no teams at all, the program recovers in 3.73 seconds. When using teams of 4 processors, the recovery is slower with 4.10 seconds and growing the team size to 16 increases the recovery time to 4.86 seconds. These numbers illustrate the tradeoff in the team-based approach: bigger teams decreases memory overhead but increases recovery time.

5. Conclusions and Future Work

We presented a modification to the traditional message logging scheme that is able to reduce the memory pressure. The tradeoff is a lower memory overhead for an increased recovery time. Since the results are promising, we see several opportunities to further improve this proposal.

First of all, we would like to group the processors into teams using a more clever strategy. We could use the first iterations of the application to profile the communication pattern and create the teams accordingly. Objects exchanging a lot of messages can be assigned to processors in the same group. However, there is also a tradeoff between this intelligent strategy and load balancing. We want to minimize the number of messages logged, but we also want to keep the load balanced to avoid hurting the performance of the application.

Another focus of our future work will be the improvement of performance for the recovery phase. Integrating the parallel restart strategy [4] will more likely accomplish this goal.

Besides reducing the memory footprint of message logging, we are also interested in reducing the increased latency due to the pessimistic model. Using a causal protocol would bring a faster transmission time. We will combine the two approaches to make a more efficient protocol that can minimize the latency overhead of message logging.

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. The idea of partitioning the set of processors to reduce the memory overhead in message logging came first from a suggestion of Greg Bronevetsky.

References

[1] L. Alvisi and K. Marzullo. Message logging: pessimistic, optimistic, and causal. *Distributed Computing Systems, International Conference on*, 0:0229, 1995.

[2] G. Bosilca, A. Bouteiller, F. Cappello, S. Djilali, G. Fedak, C. Germain, T. Herault, P. Lemarinier, O. Lodygensky, F. Magniette, V. Neri, and A. Selikhov. Toward a scalable fault tolerant MPI for volatile nodes. In *Proceedings of SC 2002*. IEEE, 2002.

[3] A. Bouteiller, P. Lemarinier, G. Krawezik, and F. Cappello. Coordinated checkpoint versus message log for fault tolerant mpi. *Cluster Computing, IEEE International Conference on*, 0:242, 2003.

[4] 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.

[5] E. N. Elnozahy and W. Zwaenepoel. Manetho: Transparent roll back-recovery with low overhead, limited rollback, and fast output commit. *IEEE Trans. Comput.*, 41(5):526–531, 1992.

[6] C. Huang, G. Zheng, S. Kumar, and L. V. Kalé. Performance evaluation of adaptive MPI. In *Proceedings of ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming 2006*, March 2006.

[7] D. B. Johnson and W. Zwaenepoel. Sender-based message logging. In *In Digest of Papers: 17 Annual International Symposium on Fault-Tolerant Computing*, pages 14–19. IEEE Computer Society, 1987.

[8] L. Kalé. The Chare Kernel parallel programming language and system. In *Proceedings of the International Conference on Parallel Processing*, volume II, pages 17–25, Aug. 1990.

[9] 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*, 0:115–124, 2004.

[10] R. Strom and S. Yemini. Optimistic recovery in distributed systems. *ACM Trans. Comput. Syst.*, 3(3):204–226, 1985.