Design and Analysis of a Message Logging Protocol for Fault Tolerant Multicore Systems

Esteban Meneses, Xiang Ni, Laxmikant V. Kalé
Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, IL 61801, USA
E-mail: {emenese2, xiangni2, kale}@illinois.edu

***Abstract*—A look at Exascale reveals a future with multicore supercomputers that will inexorably experience frequent failures. Providing scalable and efficient fault tolerance support is one of the major concerns to pave the road for the next generation of machines. Checkpoint/restart remains as the standard *de facto* approach to provide fault tolerance in supercomputers. However, its high recovery cost has brought the attention of the community to an alternative mechanism, message logging. In this paper we present the design of a message logging protocol that targets multicore machines based on two fundamental assumptions. First, a multicore node is the minimum unit of failure and very frequently only one node goes down per failure. Second, the shared memory is a key resource to bring down the overhead of message logging. This paper also presents an analysis of failure data from recent supercomputers that show that most of the time a failure involves one single computational node. We offer two different distributions to model the data. Using those distributions we build a model for the survivability of the message logging protocol to multiple concurrent failures. We demonstrate our technique has a low overhead. The results of an experiment with a stencil program show the execution time penalty is below 5% when the program scales up to 1024 cores. Moreover, even when the protocol was designed to tolerate one single failure at a time, it provides a high probability of survival to a failure involving any number of nodes. Using real-world data from recent supercomputers we demonstrate the chances of survive any failure are higher than 99%.**

***Index Terms*—fault tolerance, causal message logging, multicore systems.**

I. Introduction

One of the major concerns for the future of high performance computing (HPC) is fault tolerance [1], [2]. The most optimistic predictions about failures in Exascale forecast a mean time between failures of a few hours. To pave the road for HPC applications to scale and be able to make progress in spite of frequent failures, it is necessary to design new fault tolerance techniques that consider the peculiarities of modern supercomputing environments (hardware, systems and applications) and optimize for the common case.

The typical solution for fault tolerance in HPC is checkpoint/restart. Under that scheme, all the nodes running the application must checkpoint their state periodically. Several flavors of checkpoint/restart exist. For instance, checkpoint can be triggered by the system, middleware or by the user. On the other hand, a node's checkpoint can be stored in stable storage or in the memory of other node. In this paper we assume the checkpoint is initiated by the user and checkpoints are stored in main memory.

The drawback of checkpoint/restart is its high cost in recovery, either in terms of time or energy. After a crash, all nodes have to roll back and restart the execution. An alternative approach is message logging, that requires messages to be stored and additional information to be computed after each message is received. The benefit of message logging is that a crash on one node only requires that node to roll back, while the rest of nodes resend the messages to the recovering node and keep making progress (or sitting idle and spending less energy) in the meantime.

A multicore machine offers new opportunities to the design of message logging protocols. There are two major properties about multicore we take advantage from in this paper. First, a node is typically a single unit of failure. After a crash of a node, all its composing *processing elements* (PEs) are lost as it is the memory of the node. The message logging protocol can use that fact to tolerate the failure of a subset of PEs. Second, a node has a memory that can be shared by all its PEs. In this way, data structures common to the whole set of PEs of the same node can reside in shared memory and can be used to store the additional information message logging protocols require.

The contributions of this paper can be summarized as the following: i) a collection of distributions for failures from real-world supercomputer's data and two models for the distribution of multiple concurrent failures, ii) a design of a message logging protocol for multicore arquitectures that has a low overhead compared to checkpoint/restart and, iii) an analysis of the survivability of this protocol to multiple concurrent failures.

The paper is organized in the following way. Section II shows the results of analyzing failure datasets of five different supercomputers and getting the distribution for multiple concurrent failures. We show two different functions to model those distributions. An adaptation to multicore of a well-know message logging protocol is presented in Section III. Results from an evaluation of the protocol appear in Section IV. Section V contains an analysis of the adapted protocol when multiple concurrent failures occur. Finally, Section VI concludes the paper.

II. Failures in HPC Systems

The first comprehensive study of failures in High Performance Computing (HPC) systems was performed by Schroeder and Gibson [3]. They analyzed failure data from several supercomputers at Los Alamos National Laboratory. The methodology followed by them was to find a good model to fit different variables. For instance, they found Poisson and exponential distributions to be poor fits for number of failures per node and time between failures, respectively. On the contrary, Weibull and lognormal distributions were shown to be a good model for time between failures and repair time, correspondingly. Their analyses helped to understand better the behavior of failures in large-scale computing systems. One of their findings was that failure rates do not grow significantly faster than linearly with system size. Based on that fact, we can assume failure frequency will scale as supercomputers grow within the same architecture type.

One statistic Schroeder and Gibson did not analyze was the probability of a failure affecting k nodes. In other words, how likely it is that k nodes go down as part of the same failure in the system. As noted by other authors, failures rarely affect more than one node. For instance, Moody *et al* [4] mention that 85% of the failures disable at most one compute node on the clusters at Lawrence Livermore National Laboratory where The Scalable Checkpoint/Restart (SCR) library is run. Understanding this distribution is important, since it directly dictates what design decisions should be made when laying out the fault tolerance protocol. It is well known that tolerating the concurrent failure of any k nodes in a system has a high impact on the performance of message logging protocols [5], [6]. On the contrary, we have found that protocols that tolerate one single failure at a time present low overhead and good scalability [7].

In this paper, we call *multiple concurrent failure distribution* the one that represents the number of nodes that simultaneously go down per failure. We collected failure information from several available sources and analyzed the information to generate the multiple concurrent failure distribution for 5 different machines. The Computer Failure Data Repository (CFDR) [8] has information about failure data collected at different institutions and made public for scientific use. We extracted the failure information of 3 different machines from CFDR. These machines are called System 7, System 8 from Los Alamos National Laboratory and MPP2 from Pacific Northwest National Laboratory. We also gather information from Tsubame supercomputer at Tokyo Institute of Technology and Mercury machine at National Center for Supercomputing Applications. All those machines have multicore architecture with different features about memory size, number of cores per node and so on.

Figure 1 presents the multiple concurrent failure distribution for each machine. We only present the percentage of failures that involve 1, 2, 3, 4 or more than 4 nodes. There are two important things to notice. First of all, the distribution is very skewed, to the point that the y axis had to be logarithmic

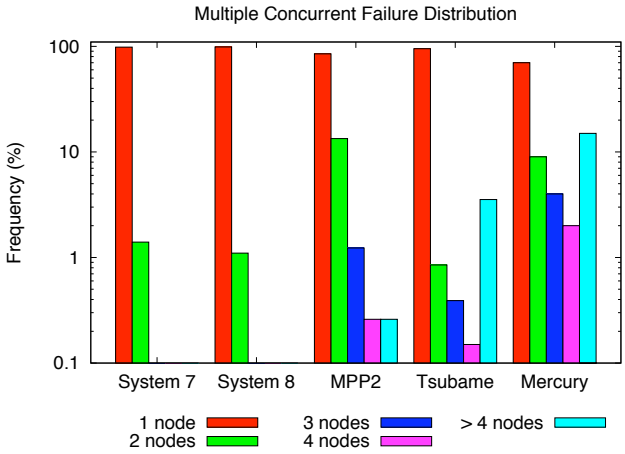


Fig. 1: Distribution of failures according to the number of crashed nodes per failure.

in order to appreciate the percentage of the less common cases. This means a very high percentage of the time a failure involves only one node. This is consistent with the findings of other authors [4]. Second, we see two different types of cases according to the shape of the curve. In the first group (consisting of the first 3 machines) we can appreciate that the percentage of cases reduces exponentially from 1 node to more than 4. The last two machines show a slightly different story, where the decreasing in the percentage is not exponential, granted they have a seemingly heavy-tailed distribution.

We proceeded to find good-fit probability distributions to model the two different types of curves in the data collected: exponential decay and heavy tail. For the former case we chose the geometric distribution, whereas for the latter we selected Zipf's.

A Bernoulli trial is an experiment with only two possible outcomes. As a random variable, the outcome has probability p of being a *success* and probability $(1-p)$ of being a *failure*. The geometric distribution models the probability of having x failures before getting the first success. More mathematically,

$$f(x) = (1-p)^{(x-1)}p$$

where p is the only parameter of the distribution. The geometric distribution can be thought as the discrete counterpart of the exponential distribution. That means, it decays quickly as x grows and for a large x the probability of x tends to zero.

On the other hand, to model distributions with a heavy tail, we recurred to Zipf's distribution. Notoriously by being applied to information retrieval to model the frequency of words in a text, Zipf's distribution is described by the following formula,

$$f(x) = \frac{\frac{1}{x^s}}{\sum_{i=1}^{n} \frac{1}{i^s}}$$

which has two parameters. First, s is a parameter controlling how skewed the distribution is. If s equals 1, then the denominator of the fraction is x multiplied by the generalized

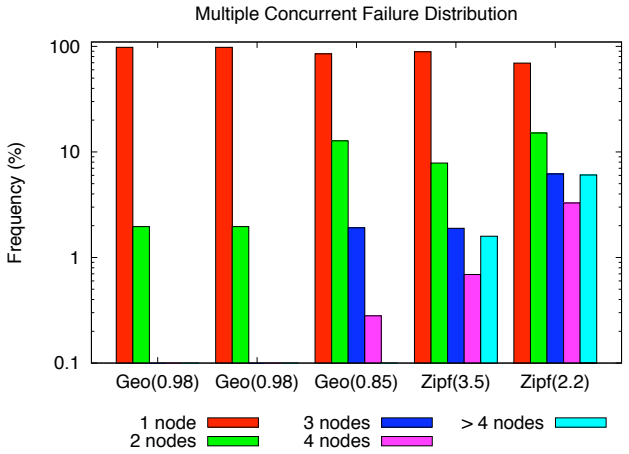


Fig. 2: Different models for multiple concurrent failure distribution.

harmonic number. Second, n is the maximum value of x, being 1 the minimum. A property of Zipf's distribution is that it does not decay exponentially and provides a long tail. This means, the accumulated probability of values larger than x is never negligible, no matter how big x is.

Using these two functions we chose proper parameters to model the distributions in figure 1. The results are shown in figure 2 and the parameters are shown in parenthesis. For the heavy-tail cases we used a value of n equals to 1024. The first two distributions were modeled by a geometric function and fit very closely the original data. The third distribution is also geometric and has a good fit, but the tail was not completely captured by the geometric function. The fourth case was the more challenging, since it has a surprisingly long tail. In order to account for the long tail, we had to reduce value s and reduce how skewed the curve was. Still the tail does not account exactly for the original value, but is not negligible in the model. We would not have been able to do this by using a geometric distribution. Finally, the last distribution is almost matched perfectly by a Zipf's distribution.

III. Fault Tolerance Mechanisms

An important design decision when devising a fault tolerance technique is whether to tolerate one single node failure or the simultaneous failure of any subset of nodes. The previous section gives us a fundamental clue on this regard. A high fraction of the failures in HPC systems brings down only one single node. We will present two major mechanisms for fault tolerance that survive the crash of one single failure at a time. Section V will provide an analysis of the probability of these two mechanisms to resist multiple concurrent failures.

A. Checkpoint/Restart

The preferred method to provide fault tolerance in HPC is checkpoint/restart [4], [9]. In a nutshell, checkpoint/restart consists in nodes frequently checkpointing their state to stable storage or the memory of another node. The optimum time to checkpoint has been analyzed elsewhere [10]. Nodes can coordinate their checkpoints to store them in a consistent way. As an option, each node may checkpoint at its own pace, but the system must take care of the state of the channel at the time of checkpoint. If there is a failure, all nodes must roll back to the previous checkpoint and restart from there.

Previous work on checkpoint/restart for multicore machines has focused on reducing the jitter at the time of checkpoint by merging multiple requests to the file system into coarser writes. Ouyang *et al* have identified the benefits of write aggregation and interleaving in this scenario [11].

We designed and implemented a checkpoint/restart strategy for multicore machines to work as a comparison point for the message logging protocol. There are a few design decisions we made in our approach. First, we target applications with a moderate memory footprint, such that checkpoint can be stored in main memory. Second, the application checkpoints in a synchronized fashion. This means, all the nodes in the application reach a point where they start a global collective operation to checkpoint. Third, we assume we always have spare nodes to substitute those other nodes that crash.

Each node at checkpoint will store its state in two places: its own memory and the memory of its *buddy* node. If a node fails, its buddy will provide the required state to restart. More details on a similar protocol that targets machines with one core per node are presented elsewhere [12]. It should be clear that our protocol tolerates one failure at a time. If both a node and its buddy are lost due to a failure, then restart is compromised. However, there are multiple concurrent failures that our protocol can still tolerate. A deeper analysis is presented in section V.

B. Message Logging

The main drawback of checkpoint/restart is its cost of recovering from a failure. Since all nodes have to rollback after one of them crashes, there is a high impact on the energy and execution time necessary to tolerate a failure. Message logging techniques have been explored as a promising alternative to checkpoint/restart [13], [14].

The main advantage of message logging comes from the fact that a crash in one node does not mean all the rest have to roll back. Instead, the surviving nodes can keep making progress in the application or at least remain idle and consuming less energy while the crashed node recovers.

Of course, that advantage does not come for free. Message logging requires every application message to be stored. For instance, if the sender node X stores all the messages it sends, then after a failure of one of its recipient nodes Y, X is able to re-send to Y all the messages it sent before the crash. That way, X does not roll back and Y is able to recover.

In addition to storing messages, message logging protocols must also store extra information about the messages. A *determinant* is the result of any non-deterministic decision made by a node. For example, a message reception is in general non-deterministic. After receiving a message, a node will generate a determinant consisting of four components

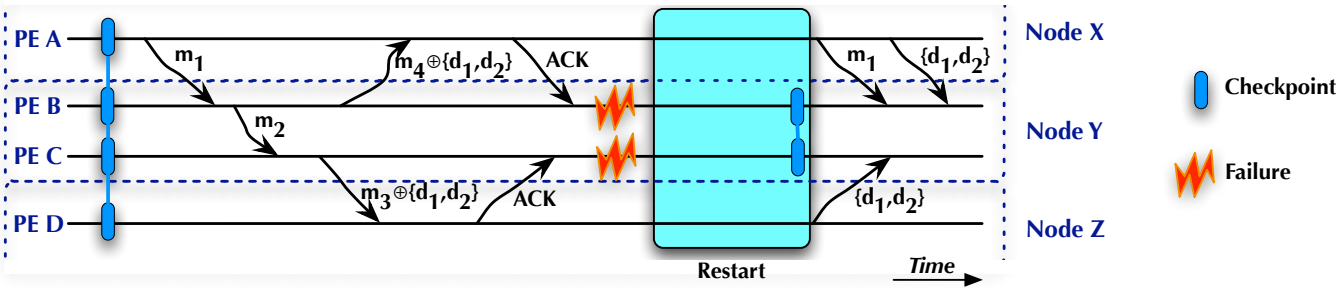


Fig. 3: Forward path and recovery in causal message logging.

$\langle senderID, receiverID, ssn, rsn \rangle$. Along with the IDs of both sender and receiver, the determinant contain the *send sequence number* (ssn) and the *receive sequence number* (rsn). A determinant is necessary to provide a consistent recovery, discarding repeated messages and ordering the reception of messages resent. Depending on how determinants are manipulated, several flavors of message logging are possible [15].

In this paper we will use a protocol called *causal message logging* [5], [7], [15]. The main intuition behind this protocol is that determinants will be stored in the causality path that starts at their creation. In other words, a determinant d produced at PE A will be stored somewhere else only if there is a message leaving A that occurs after d has been generated at A. If no message connects determinant d with any other determinant, then it is fine to lose d in a failure, since it did not have any causal effect on the system.

The way causal message logging works is depicted in figure 3. There are two parts presented in the figure. The failure-free scenario or *forward path* is shown on the left extreme. The recovery after a failure is shown on the right. The execution in the figure starts with a checkpoint that we assume is globally coordinated. This assumption can be easily removed as one of the major advantages of message logging is to provided uncoordinated checkpoint. However, many applications in HPC exhibit global synchronization points and those places are ideal to kick off the checkpoint mechanism. Each node checkpoints its state in the memory of a buddy node, but this time a node does not store its state in its own memory. As opposed to checkpoint/restart, that extra copy of a node's state does not bring any advantage to the protocol.

Every message reception generates a determinant. In figure 3 message m_1 is sent by PE A and received by PE B. As soon as the message is received, B generates determinant d_1. Once a determinant is generated, it will be piggybacked on every outgoing message until it is safely stored in other node. Notice that message m_2 does not piggyback any determinant, since it is a *local* message, i.e., within the same node. However, at C, the reception of m_2 generates determinant d_2. The next outgoing message, m_3 in this case, piggybacks both determinants. We denote the piggyback operation by the $\oplus$ symbol. Notice that message m_4 also piggybacks both determinants, since at the time of sending m_4 out of node Y, the acknowledgments for the determinants have not been received.

In our causal protocol the node is the minimum unit of failure and as such, all the PEs failed as soon as part of the crash of their containing node. Figure 3 presents the failure of node Y. Once the system finds substitute for Y, all the PEs are recreated on that node from their last checkpoints. All the other nodes resend the messages they sent from the last checkpoint and they also send any determinants they have stored from node Y. With messages and their respective determinants, node Y is able to recover from the crash.

An important thing to highlight is that local messages are not stored, since they will be lost in case of a failure. Determinants corresponding to those messages are nevertheless generated and treated as any other determinant. To leverage the potential of multicore machines, we devised a method to manage the determinants generated at a particular node. All the PEs share a common structure for storing determinants. This structure behaves like a queue, where different PEs will insert a new determinant, copy several of them to piggyback and acknowledge the safe storage of other determinants. Since all PEs will simultaneously access this structure, we must have a way to synchronize the access to the structure for the three different operations. A straightforward way is to use a single lock and define three critical regions protected by that lock. This will inevitably result in a high lock contention for nodes with several PEs and potentially many threads running per PE. We ran a simple experiment to have an idea of how much time lock contention could stand for. We used 32 nodes with 8 PEs each and 4 threads running per PE. Table I shows the timing results for lock contention on the three different operations: adding a determinant, piggybacking and acknowledging them. In general we can see the high cost of having a simple synchronization mechanism. The added extra latency for every message could be measure in dozens of microseconds.

TABLE I: Lock Contention (microseconds)

Add	Piggyback	Acknowledge
41	60	79

However, a coarse synchronization for the structure holding

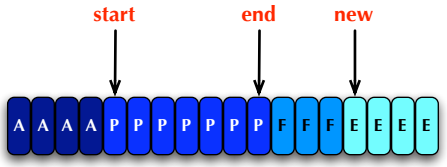


Fig. 4: Determinants.

the determinants is clearly an overkill. Having a fine-grained synchronization mechanism can easily get rid of the lock and avoid the high cost of lock contention. We decided to use a smarter way to access the determinants in shared memory. Figure 4 presents a simplified view of the queue storing determinants. There are four possible states for an entry in that queue. A determinant may have been acknowledged (A), or it has to be piggybacked (P), or is currently being filled up (F), or it may be empty (E). Three different indexes regulate access to the queue. The *start* index indicates the start of the segment of determinants that have to be piggybacked in the next outgoing messages. If an acknowledgment is received, that index has to be moved forward. The other limit for the piggybacked determinants is signaled by *end*. Finally, every time a thread needs to enter a new determinant, it is given the entry pointed by *new*. Once the thread finishes writing all the information on that entry, the *end* index has to be moved accordingly.

Figure 5 presents the pseudo-algorithm for the different operations on the queue of determinants Q. We are able to remove all locking operations to access the queue and instead use atomic operations over the indexes of the queue. Notice that when a new determinant is to be added to the queue, a new position t is obtained, but this position has to be committed in order, to keep consistency in the queue.

Add $\langle d, Q \rangle$:
- $t \leftarrow$ `AtomicFetchAndIncrement`(new)
- Fill entry t with d
- **while** $t \neq (end + 1)$ **do**
  - NO OP
- **end while**
- `AtomicIncrement`(end)

Piggyback $\langle Q \rangle$:
- $last \leftarrow end$
- $first \leftarrow start$
- **if** $last \geq first$ **then**
  - `Piggyback`$(first, last)$
- **end if**

Ack $\langle index, Q \rangle$:
- **if** $index \geq start$ **then**
  - `AtomicSet`$(start, index + 1)$
- **end if**

Fig. 5: Lockless shared access to determinant queue.

Our protocol ensures the recovery from a single node failure. A multiple concurrent failure may compromise the recovery. Section V analyzes the likelihood of our approach to survive a multiple concurrent failure.

IV. Experiments

We implemented both approaches explained in the previous section, checkpoint/restart and causal message logging, in the CHARM++ runtime system [16]. As a programming language, CHARM++ allows the programmer to split the computation and data of an application into objects. Usually, the application is over-decomposed and the system has more objects than physical PEs. As a runtime system, CHARM++ offers a substrate that implements features like object migration and fault tolerance.

A. Experimental Setup

We ran our experiments on two different clusters. The first is Steele supercomputer at Rosen Center for Advanced Computing (RCAC). Steele is a 60-Teraflop machine and consists of 893 nodes. Each node is a 64-bit, 8-core Dell system with either 16 or 32 GB in RAM. The nodes are connected through 1 Gigabit Ethernet. For the experiments we considered each core on a node to be a PE. The second is Ranger supercomputer at Texas Advanced Computing Center (TACC). Ranger is a 579.4-Teraflop machine that comprises 3,936 nodes for a total of 62,976 processing cores. Each node has 16 cores and 32 GB of main memory. An Ethernet network connects the different nodes.

We chose various applications to run the experiments. First we use a 3D stencil computation that decomposes a tri-dimensional space into blocks. Each block is modeled thorough a CHARM++ object and has 6 neighbors. In every iteration, an object exchanges its halo cells with all its neighbors and performs a relaxation computation over the block of cells it is responsible for. The execution proceeds for a number of iterations. The other program is called FT Test and represents a configurable benchmark that provides several parameters to customize: communication topology, message size, computation grain-size and a few others.

B. Results

We started the experiments by measuring two important common features of checkpoint/restart and message logging. Both schemes use a similar checkpoint and restart mechanism. Our interests was to see how scalable these two features were.

For the checkpoint part we used FT Test and measured how much time it takes to perform an in-memory checkpoint. We changed both the data size and the number of cores on Steele. Figure 6 presents the results for the different cases when the data size is changed from 64 MB to 1024 MB. In every case we used one single object per PE. There are two things to notice in this case. All the curves look flat as the number of cores should not affect the checkpoint time in a significant way. Also, time to checkpoint grows linearly with data size, ensuring that the dominant cost in checkpointing is data sending and not any

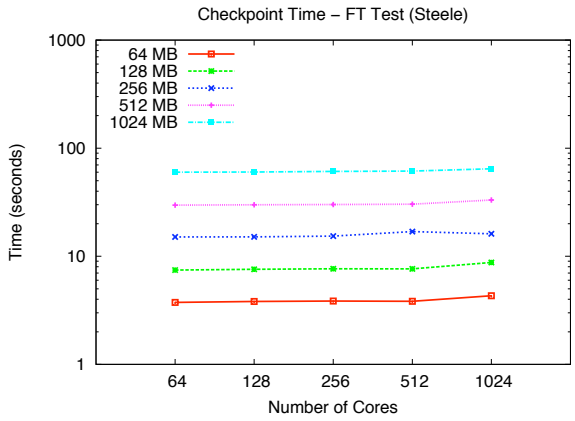


Fig. 6: Checkpoint time as a function of data size.

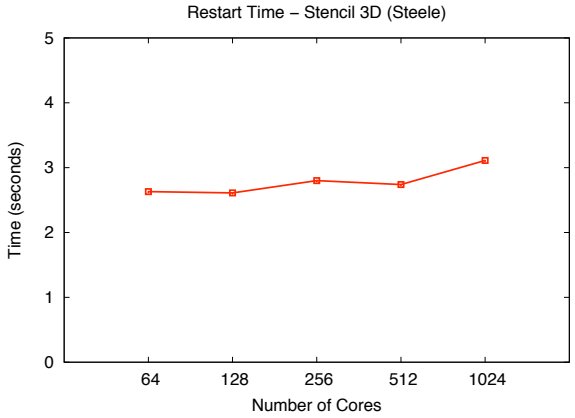


Fig. 7: Restart time as a function of number of cores.

other manipulation. It is important to mention that at 1024 MB per object, we were effectively exhausting the physical memory of nodes on Steele.

Next, we studied the behavior of restart time. Recalling from figure 3, the restart time is the time span between a failure and the point where the application is ready to run again. It includes relaunching the crashed node in one spare node and obtaining the checkpoint from the buddy. Using Stencil 3D we measured the restart time on Steele and changed the number of cores. We see in figure 7 the results of restarting after a crash. The penalty of increasing the number of nodes or cores is very small and the dominant factor seem to be the transmission of data across the network.

A comparison of both approaches, checkpoint/restart and causal message logging, was obtained by using Stencil 3D program. Using a weak scaling approach, where each core has four objects (each having a block of size $128x128x128$) we ran the program from 64 cores up to 1024 cores. Figure 8 presents the results of the overhead of message logging with respect to checkpoint/restart. The overhead never goes beyond 7%, the highest value is obtained for 64 cores where the overhead is 6.45%. The rest of the data points show lower values for the overhead. This experiment was run on Ranger that has 16-way

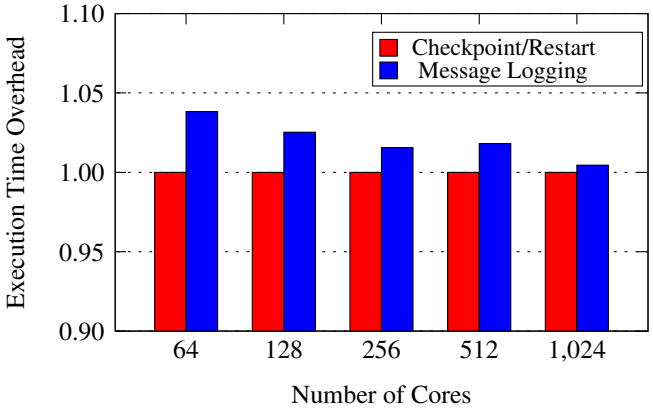


Fig. 8: Weak scaling experiment with Stencil 3D.

nodes and the test scaled from 4 nodes to 64.

We decided to inspect on the bandwidth overhead imposed by the determinants. Using the same configuration as the previous experiment and running on 256 cores on Ranger, we measured how many determinants were sent in all the messages. Table II summarizes our findings. The table shows the statistics per PE (core) throughout the whole execution. By observing the number of determinants generated by a PE and piggybacked by the same PE, we can compute the average number of times a determinant is piggybacked. In our case this number is 6.34. This means, that a particular determinant is replicated more than 6 times. Even more, the number of determinants piggybacked per message goes beyond 15.

TABLE II: Determinants and Messages

Determinants Generated	2456
Determinants Piggybacked	15584
Messages	1830
Determinants per Message	15.8

V. Analysis of Survivability

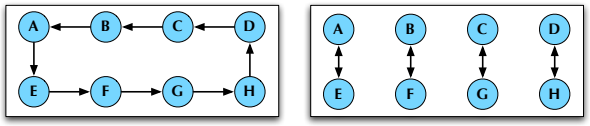
The protocols presented in section III for multicore machines, checkpoint/restart and message logging, are designed to tolerate one single failure at a time. Multiple concurrent failures *may* be tolerated, though. This section presents an analysis of how likely it is for those protocols to survive a crash involving multiple nodes.

A. Checkpoint/Restart

Recalling from section III, checkpoint/restart bases its fault tolerance strategy on frequent checkpoints. The checkpoints can be stored either in disk or in a remote memory. In our protocol we prefer the latter, since it provides a faster checkpoint time, although may become a problem for memory hungry applications. In-memory checkpoint/restart requires each node to have a *buddy* node where it will checkpoint. Besides the checkpoint stored in its buddy, each node will

store a checkpoint in its own memory. That way, the crash of the buddy will not affect the recovery of a particular node.

The assignment of buddies to nodes is a mapping or a bijection among the nodes. There are, however, different mappings with very different features. For instance, figure 9 presents two possible mappings, called the *ring map* and *pair map*. Even when both seem simple enough, pair map is much more resilient to multiple concurrent failures than ring map. Remember that in order to survive a multiple crash we need not to lose a node and its buddy (because that will make impossible to restart a node). Now, imagine a double concurrent crash in the scenario of figure 9 where we have 8 nodes. Out of the 28 possible cases, ring map survives 20, whereas pair map survives 24.



(a) Ring Map (b) Pair Map

Fig. 9: Different mapping schemes for buddy assignment.

So, if we have to choose between the two previous mappings, we should go for pair map. We will assume we are using pair mapping in the rest of the paper. The question is how resilient this mapping is when we have n nodes and f nodes fail concurrently. Since we have a total of n nodes and a subset of size f nodes fail, the total possible number of such subsets is $\binom{n}{f}$. We need to compute, how many out of those subsets are not catastrophic, given the pair mapping.

In order to survive a multiple crash of f nodes, we need to compute how many subsets of f nodes do not take down a node and its buddy. Now, if we need to choose f nodes with such property, then the first node of the f has n options, the second has $n-2$ (since we do not want to include the buddy of the first), the third has $n-4$, and so on. So, at the end we have this total number of subsets size f that will not make the whole system to collapse:

$$\frac{n(n-2)(n-4)...(n-2(f-1))}{f!}$$

which gives us the following expression for the probability of surviving f concurrent failures:

$$\frac{n(n-2)(n-4)...(n-2(f-1))}{n(n-1)(n-2)...(n-f+1)} = \frac{\prod_{i=0}^{f-1}(n-2i)}{\prod_{i=0}^{f-1}(n-i)}$$

As an illustration, figure 10 shows the curve for the previous equation as we plot the value for each f in the range 0 to 128. In this particular case, we chose n equals to 1024. The curves drops smoothly, showing a value higher than 80% for up to 16 nodes crashing concurrently.

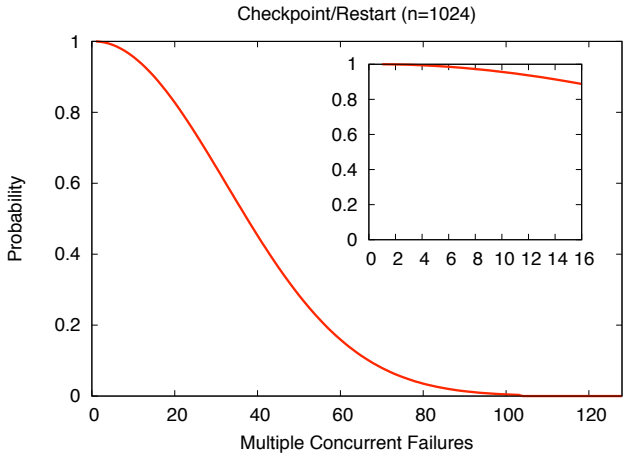


Fig. 10: Probability of surviving a multiple concurrent failure in the checkpoint/restart model.

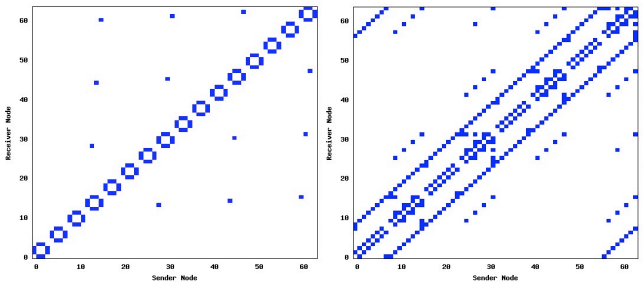
B. Message Logging

The checkpoint/restart model is oblivious of communication. In other words, having more nodes connected via messages does not affect the resilience of checkpoint/restart. That metric, however, makes a big impact on message logging. Before getting into the theoretical formulation of multiple concurrent failure for message logging, we will address an important point regarding communication topology on parallel programs.

Many parallel computing applications result from composing different modules that apply some structured pattern of communication. Programmers may use one of Berkeley's thirteen dwarfs [17] as building blocks for their applications. For instance, a halo exchange communication pattern is used to implement a stencil computation, whereas a wavefront computation pattern is useful in many parallel dynamic programming algorithms.

Figure 11(a) shows a communication topology for the NAS-CG (class D) benckmark with 512 ranks on 64 nodes. The x-axis presents the sender and the y-axis the receiver. As we can appreciate the communication pattern is very regular, to the point where most of the communication is symmetric. Each node contacts on average little more than 2 other nodes. The same can be said about NAS-MG (class D) benchmark. Figure 11(b) presents the communication graph for this program. Although communication is more spread throughout the system, on average every node contacts little more than 4 other nodes.

In the causal message logging scheme, each determinant is only replicated one single time on the memory of other node, additionally to the one where it was generated. This means, it will support with total certainty one single failure at a time. A multiple concurrent failure can be tolerated as long as there is a copy of the determinant alive after the crash. More specifically, if node x communicates with g other nodes, and there is a multiple failure where x is involved (i.e., x crashes), then the only hope to tolerate such a failure is to have the other



(a) NPB-CG (b) NPB-MG

Fig. 11: Communication Topology.

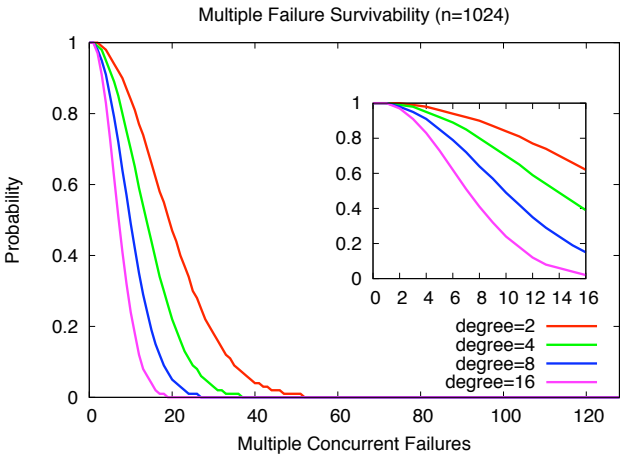


Fig. 12: Probability of surviving a multiple concurrent failure in the message logging model.

g nodes x communicates with still alive.

The question we are trying to solve is how likely a system using the causal message logging protocol is able to survive a multiple concurrent failure. Let's formalize a few concepts. The system has n nodes where each node communicates with g others. For simplicity, we will assume each node chooses randomly the other nodes it exchanges information with. Let us assume a multiple failure involves f nodes from the system. The set of crashed nodes is denoted by F.

To compute the probability to survive the failure of set F, we need to compute how likely it is for F to *not* intersect the f different communication subgroups of elements in f. In other words, there cannot be 2 or more nodes in F that communicate with each other. It is a combinatorics problem. Let's pick one element x in F and compute how likely it is that the rest of F does not intersect the subset of g elements x communicates with (denoted by G). Since the system has n elements, the number of subsets of g elements that x may contact is:

$$\binom{n-1}{g}$$

Now, the number of subsets of size g that do *not* intersect F are given by:

$$\binom{n-f}{g}$$

With this two quantities we are ready to compute the probability of the set G not intersecting F. Moreover, the probability of set F not intersecting any of the communication subsets of its members and, by definition, the probability of tolerating a multiple concurrent failure of f nodes is:

$$\left[\frac{\binom{n-f}{g}}{\binom{n-1}{g}}\right]^f$$

Using the previous formula, we can plot the probability of surviving a multiple crash based on different values of g and different values of f. Figure 12 presents four different curves for values of degree g $(2, 4, 8, 16)$ for a value of $n = 1024$. Running horizontally in the figure, we have f, the number of concurrent failures. Given that the formula of survivability is dominated by f in the exponent, the decay in the probability is exponential. For example, if g is 8, then probability of surviving 2 concurrent failures is 0.97, whereas probability of surviving 4 and 8 concurrent failures is 0.83 and 0.41, respectively. In general we see that all message logging curves drop more drastically than the checkpoint/restart case. That is the price of being susceptible to communication.

C. Survivability

If we were to compute how likely it is that we survive any crash, we should sum over all the cases and multiply each case by its corresponding probability. Let us define *survivability* as the probability of survive *any* crash, regardless of how many nodes are involved in the failure. The formula is given by:

$$\mathcal{S} = \sum_{i=1}^{n} s(i)p(i)$$

where $s(i)$ represents the probability of surviving a crash that involves i nodes and $p(i)$ is the probability of a random failure involving i nodes.

Using the definition of $\mathcal{S}$, we can compute the survivability of the different approaches of section III. In order to model the probability of failure we used the two functions discussed in section II. Since other authors found evidence that single node failure probability is higher than 85% [4], we chose the parameters of the distributions accordingly. For the geometric distribution we set p value to be 0.85 and for the Zipf's distribution we used s value equals to 3.2. Table III shows the survivability values for the different fault tolerance strategies. The number in parenthesis for the message logging approach stands for the degree g. Although the survival of checkpoint/restart is better than message logging, which has a curve that drops exponentially as the number of concurrent failures increases, the difference does not translate into a big difference for survivability. The reason comes from the fact that functions to model the probability of multiple concurrent failures are very skewed, making negligible the contribution of larger values of f.

TABLE III: Survivability

	Geometric	Zipf's
Checkpoint/Restart	0.9997	0.9992
Message Logging (2)	0.9991	0.9974
Message Logging (4)	0.9983	0.9953
Message Logging (8)	0.9967	0.9919
Message Logging (16)	0.9936	0.9862

VI. Conclusion and Future Work

The paper presented the design, implementation and analysis of fault tolerance strategies for multicore machines. We focused our paper in the design of a message logging protocol that tolerates the crash of a single node and *may* survive the failure of several nodes simultaneously.

We analyzed several failure datasets to generate the distribution of failures according to the number of nodes that crash as part of a failure. We discovered such distributions are highly skewed and can be classified into two families. The first family presents distributions with exponential drop that can be modeled by a geometric distribution, whereas the second family shows a longer tail that is fitted by a Zipf's distribution. Using this knowledge, we designed a message logging protocol for multicore machines, where a shared data structure keeps track of the determinants generated in the node. We devised a fine-grained synchronization mechanism to access the shared data structure. Our experiments showed a small overhead of the message logging implementation over the standard checkpoint/restart mechanism. An analysis of the message logging protocol for multiple concurrent crashes showed that we can tolerate the crash of several nodes with high probability, granted that number is single digit. We built a model for multiple concurrent failures and determined the probability of tolerating any failure is higher than 99% for most of the cases.

For future work, we are planning to analyze the cases where multiple concurrent failures are correlated. Since there are architectural constraints that may cause several nodes to fail in tandem, we may design message logging protocols to tolerate those multiple correlated failures.

Acknowledgments

This work partially supported by the US Department of Energy under grant DOE DE-SC0001845 and by a machine allocation on the Teragrid under award ASC050039N. We would like to thank Ana Gainaru and Leonardo Bautista-Gomez for providing us failure data of Mercury and Tsubame, respectively. Discussions with Chao Mei and Aaron Becker at the Parallel Programming Laboratory proved to be helpful for the ideas about lockless access to a shared structure.

References

[1] P. Kogge, K. Bergman, S. Borkar, D. Campbell, W. Carlson, W. Dally, M. Denneau, P. Franzon, W. Harrod, J. Hiller, S. Karp, S. Keckler, D. Klein, R. Lucas, M. Richards, A. Scarpelli, S. Scott, A. Snavely, T. Sterling, R. S. Williams, and K. Yelick, "Exascale computing study: Technology challenges in achieving exascale systems," 2008.

[2] F. Cappello, "Fault tolerance in petascale/ exascale systems: Current knowledge, challenges and research opportunities," *IJHPCA*, vol. 23, no. 3, pp. 212–226, 2009.

[3] B. Schroeder and G. Gibson, "A large scale study of failures in high-performance-computing systems," in *International Symposium on Dependable Systems and Networks (DSN)*, 2006.

[4] A. Moody, G. Bronevetsky, K. Mohror, and B. R. de Supinski, "Design, modeling, and evaluation of a scalable multi-level checkpointing system," in *SC*, 2010, pp. 1–11.

[5] 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.

[6] 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.

[7] E. Meneses, G. Bronevetsky, and L. V. Kale, "Evaluation of simple causal message logging for large-scale fault tolerant hpc systems," in *16th IEEE Workshop on Dependable Parallel, Distributed and Network-Centric Systems in 25th IEEE International Parallel and Distributed Processing Symposium (IPDPS 2011).*, May 2011.

[8] CFDR. (2011, May) Computer failure data repository. [Online]. Available: http://cfdr.usenix.org/

[9] E. N. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson, "A survey of rollback-recovery protocols in message-passing systems," *ACM Comput. Surv.*, vol. 34, no. 3, pp. 375–408, 2002.

[10] J. T. Daly, "A higher order estimate of the optimum checkpoint interval for restart dumps," *Future Generation Comp. Syst.*, vol. 22, no. 3, pp. 303–312, 2006.

[11] X. Ouyang, K. Gopalakrishnan, T. Gangadharappa, and D. K. Panda, "Fast checkpointing by write aggregation with dynamic buffer and interleaving on multicore architecture," in *HiPC*, 2009, pp. 99–108.

[12] G. Zheng, L. Shi, and L. V. Kalé, "FTC-Charm++: An In-Memory Checkpoint-Based Fault Tolerant Runtime for Charm++ and MPI," in *2004 IEEE International Conference on Cluster Computing*, San Diego, CA, September 2004, pp. 93–103.

[13] 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.

[14] A. Bouteiller, P. Lemarinier, G. Krawezik, and F. Cappello, "Coordinated checkpoint versus message log for fault tolerant MPI," *Cluster Computing, IEEE International Conference on*, vol. 0, p. 242, 2003.

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

[16] 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.

[17] K. Asanovic, R. Bodík, J. Demmel, T. Keaveny, K. Keutzer, J. Kubiatowicz, N. Morgan, D. A. Patterson, K. Sen, J. Wawrzynek, D. Wessel, and K. A. Yelick, "A view of the parallel computing landscape," *Commun. ACM*, vol. 52, no. 10, pp. 56–67, 2009.