A Message-Logging Protocol for Multicore Systems

Esteban Meneses, Xiang Ni and 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—**Although many details of an eventual Exascale machine remain unknown, we can safely make a couple of assumptions. Exascale machines will be composed of multicore nodes and will experience frequent failures. The latter means that effective resilience support is imperative to make Exascale machines usable. The former opens up opportunities for exploring new alternatives to provide resilience support. This paper examines a new fault tolerance protocol for multicore systems. The paper contains three major parts. In the first part, we start by showing evidence that a node (and not a core) is the appropriate unit of failure. When a crash hits a machine, it usually renders unusable a whole node. Rarely, the crash brings down more than one node. The second part describes a message logging protocol that tolerates the failure of whole nodes and uses an efficient shared memory scheme to minimize overhead. We present results on various clusters and scale the approach to 1024 cores with a stencil computation. The overhead is always lower than 4%. The third part performs an analysis of reliability to understand how robust the protocol is when failures affect several nodes. Using an analytical framework and the frequency of multiple-node failures, we find that our approach is able to survive more than 99% of the crashes.**

Keywords-**fault tolerance, 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 Exascale forecast a *mean time between failures* of a few hours. To make HPC applications able to scale and 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 traditional solution for fault tolerance in HPC is checkpoint/restart. Under this scheme, all the nodes running an application must checkpoint their state periodically. If one of the nodes fails, the whole set of nodes have to roll back to the most recent checkpoint and restart from there. This makes checkpoint/restart have a high recovery cost in terms of both time and energy. An alternative approach is message logging, which 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 idle spending less energy) in the meantime. Additionally, environments that permit more than one task to be run per core may recover the crashed node in parallel, thereby reducing the time to catch up with the rest of the system [3].

A multicore machine poses new challenges to the design of message logging protocols. The first issue is to determine an appropriate unit of failure. On one hand, tolerating the failure of one single core is likely to be insufficient, since a typical failure will bring down a whole node with all its corresponding cores. On the other hand, it is well known that tolerating the failure of any subset of nodes in the system will make the protocol too onerous [4]. A good balance may be to tolerate the failure of one single node, proven that multiple-node failures are rare and do not impose a serious threat to the reliability of the protocol. An additional challenge of multicore systems is to efficiently adapt a traditional message logging protocol, keeping low the overhead of the new failure unit.

The contributions of this paper can be summarized as follows: i) distributions of frequency of multiple-node failures from system logs of recent supercomputers and two functions to model those distributions, ii) the design of a message logging protocol for multicore systems that has a low overhead compared to checkpoint/restart and, iii) an analysis of the reliability of this protocol to multiple-node failures.

II. Related Work

Previous work on checkpoint/restart for multicore machines focuses on reducing the jitter at the time of checkpoint by merging multiple requests to the file system into coarser writes. Ouyang *et al* [5] have identified the benefits of write aggregation and interleaving in this scenario. Our scheme exploits shared memory too, but we implement a message logging protocol that brings new advantages over checkpoint/restart.

Most of the work on fault tolerance message logging protocols has seen a core, not a node, as the unit of failure. Some of the protocols only tolerate the failure of one single core [3], [6]. Others tolerate the failure of multiple cores but only at the cost of a high overhead [7]. More recently, protocols based on properties of HPC applications have been shown to tolerate multiple simultaneous crashes [8], but they may have a high cost on recovery. We provide evidence that a node, and not a core, should be the appropriate unit of failure.

Ropars and Morin [9] developed a distributed event logger to store the non-deterministic events in an active optimistic message logging protocol for MPI. In their work, each node holds a certain portion of the total number of determinants and there is some redundancy among the event loggers in the different nodes. As opposed to their approach, we do not have the notion of an event logger, since determinants are stored at the receivers of the messages according to a causal message logging protocol.

Another work has highlighted the failure correlation of cores on the same node [10]. The authors of that paper propose a technique to enhance message logging and deal with correlated failures. In their approach, a group of cores that have correlation according to their failure pattern checkpoint coordinately and rollback as a unit in case of failure. Their approach is more general than ours, but it does not provide any mechanism to minimize overhead by using shared memory.

III. The Right Unit of Failure

The first comprehensive study of failures in HPC systems was performed by Schroeder and Gibson [11]. They analyzed failure data from several supercomputers at Los Alamos National Laboratory. 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. Another finding in their paper 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. This particular paper was followed by other studies [12], [13].

One statistic that has eluded the analysis of system logs is the probability of a failure affecting k nodes. In other words, how likely is it 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* [13] 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 [4], [14]. Also, protocols that tolerate one single failure at a time present low overhead and good scalability [6].

In order to study how many nodes are lost due to a single failure in real-world HPC systems, we collected failure information from several available sources. The Computer Failure Data Repository (CFDR) [15] has information about failure data collected at different institutions and made public for scientific use. We extracted the failure information of several different machines from CFDR. These machines are called System i (for $i = \{12, 18, 19, 20, 21\}$) from Los Alamos National Laboratory and MPP2 from Pacific Northwest National Laboratory. We also gathered information from Tsubame supercomputer at Tokyo Institute of Technology and Mercury machine at National Center for Supercomputing Applications. All those machines have architectures with multiple cores per node and exhibit different features about memory size, number of cores per node, and so on.

Finding recent information about failures in supercomputers is not an easy task. There are several reasons for this. First, the nature of the information is very sensitive. There is little motivation for a supercomputing facility to make public their failure record, something that in a sense exposes their weaknesses. Moreover, releasing that information may impose a threat to any of the contractors that supply parts or the whole machine. Second, even if all system logs were made publicly available, parsing, interpreting and filtering failure information is a complex task. One single failure may manifest itself as a series of disconnected messages. In other cases, a failure does not really occur until certain thresholds in number of warnings or timeouts are reached. Despite these drawbacks, the community has realized the benefits of providing researchers with failure data. All the failure datasets we use in this paper were already filtered by a group of experts. They were able to determine what messages in the logs correspond to failures and what nodes are impacted at what particular time.

Thus, each dataset is basically a list of failures. Each failure typically includes: nodes affected, possible reason for the failure, date and time. We proceeded to examine each dataset in two phases. The first phase goes through the whole data set *coalescing* failures of the same node that occur within a time window smaller than Δ_C. This phase eliminates multiple instances of the same failure. Our value for Δ_C was 6 hours in accordance with a reasonable estimate of a repair time of a node [11]. The second phase traverses the reduced list obtained after the first phase and counts how many nodes failed in a time window defined by Δ_M. Our value for Δ_M was based on the time it takes for a system to detect a failure and restart. We chose a value of 1 minute (a conservative estimate according to our experimental results [16]). Thus, if k nodes fail within a time frame of Δ_M, we consider that as a failure affecting k nodes as if all of them had failed simultaneously.

In this paper, we call *multiple-node failure distribution* as the one that represents the number of nodes that simultaneously go down per failure. Figure 1 presents the multiple-node failure distribution for each of the machines we examined. 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 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 [13]. 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

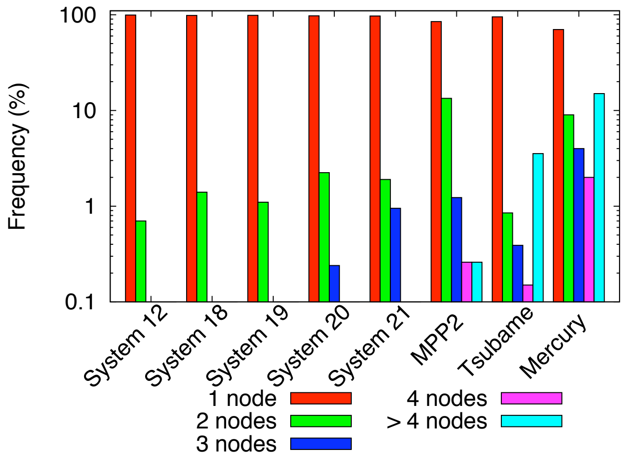

Fig. 1: Number of nodes brought down per failure.

decrease in the percentage is not as steep. Their functions correspond to a heavy-tailed distribution. The rest of the cases stay somewhere in the middle. 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. More details about how these functions fit the data can be found elsewhere [16].

IV. Message Logging Protocol

Given that most failures only make a single node to crash, we present a message logging protocol that tolerates the loss of one node per failure. In the infrequent case where several nodes go down simultaneously, the protocol may still be able to recover. However, there is no guarantee in the general case. Section V will provide an analysis of the probability of our protocol resisting multiple-node failures.

We assume that a parallel computation is decomposed into different *objects*. Each object holds a portion of the program's data and performs a fraction of the computation. The only mechanism to exchange information between the objects is through messages. The network delivers the messages in any order, there is no FIFO guarantee for each communication channel. There is a runtime system which schedules the execution of each object and monitors each message exchange. The runtime system may migrate objects from one node to another when it sees that convenient. This is a realistic model used in several applications and systems, such as Charm++ [17].

The underlying machine is thought to be a set of multicore computational nodes. Each node is comprised of various processing elements (PE), each equivalent to one core. Each node has shared memory for its own cores, but this memory cannot be shared by the objects. The runtime system is the only entity able to share the memory for communication.

We assume nodes crash according to the *fail-stop* model, in which the node ceases to work. The runtime system runs its own failure detector and as soon as one node crashes it is able to get a spare node to replace the one that just crashed. The preferred method to provide fault tolerance in HPC is checkpoint/restart [13], [18]. In a nutshell, checkpoint/restart consists in nodes frequently checkpointing their state to stable storage or the memory of another node. Nodes can coordinate their checkpoints to store them in a consistent way. If there is a failure, all nodes must roll back to the previous checkpoint and restart from there. In the double in-memory checkpoint/restart protocol, 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.

To avoid all nodes to rollback in case of a failure, we designed a message logging protocol that will make only the crashed node to rollback in case of a failure. In principle, every application message needs 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 outcome 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 $\langle senderID, receiverID, ssn, rsn\rangle$. Along with the IDs of both sender and receiver, the determinant contains the *send sequence number* (ssn) and the *receive sequence number* (rsn). Both numbers, ssn and rsn, are used during recovery. The former is required to detect duplicate messages, while the latter provides the sequence in which messages have to be processed. 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 [7].

In this paper we will use a protocol called *causal message logging* [6], [7], [14]. 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 2. 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 provide uncoordinated checkpoint. However, many applications in HPC exhibit global synchronization points and those places are ideal to trigger the checkpoint mechanism. Each node checkpoints its state in the memory of a buddy node. Unlike the double in-

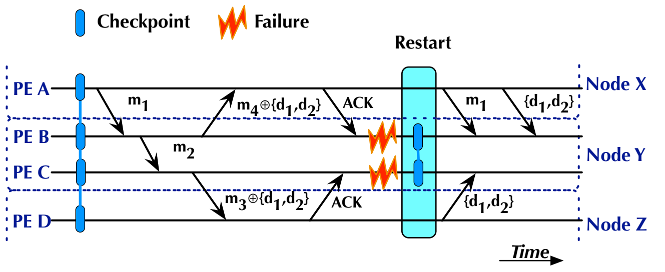

Fig. 2: Forward path and recovery in causal message logging.

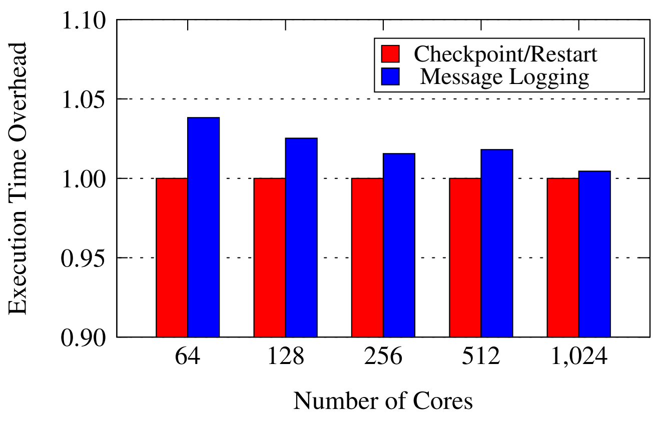

Fig. 3: Weak scaling experiment with Stencil 3D.

memory checkpoint/restart approach, each node is not required to store its state in local memory.

Every message reception generates a determinant. In figure 2 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. Note 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, a node is the minimum unit of failure and as such, all the PEs fail as soon as their containing node crashes. Figure 2 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.

There are two types of messages in our protocol. If a message is exchanged between PEs on the same node, it is called a *local* message. On the other hand, a *remote* message is exchanged between two PEs on different nodes. For instace, in figure 2, message m_2 is local and message m_3 is remote. An important thing to highlight is that local messages are not stored, since they will be lost in case of a failure. This means our protocol decreases the memory pressure due to the messages. If an important portion of the communication occurs inside the nodes, then the reduction in the memory required for the message log can be substantial. Determinants corresponding to local 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. We designed a lockless set of operations to efficiently perform these operations [16].

We implemented two fault tolerance approaches for multicore systems in the CHARM++ runtime system [17]. The first scheme corresponds to a double in-memory checkpoint/restart and the second to the message logging protocol explained above. We ran our experiments on three different clusters: *Steele* at Rosen Center for Advanced Computing (RCAC), *Ranger* at Texas Advanced Computing Center (TACC) and *Trestles* at San Diego Supercomputer Center (SDSC).

A comparison of both approaches, checkpoint/restart and message logging, was obtained by using a 7-point stencil program (Stencil 3D). 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 3 presents the results of the overhead of message logging with respect to checkpoint/restart. The overhead never goes beyond 4%, the highest value is obtained for 64 cores where the overhead is 3.82%. The rest of the data points show lower values for the overhead. This experiment was run on Ranger, which has 16-way nodes. We show performance results up to 64 nodes.

Since we conceive a node as the fundamental unit of failure, we do not log messages local to a node when using the message logging protocol. As we mentioned above, this reduces the amount of memory required to store messages. The larger the number of cores in a node, the less messages we need to save. Figure 4 presents the fraction of memory due to message log that is required in each of the three machines we used. We ran Stencil 3D program on 256 cores on each machine and counted how many bytes we store in the message log, versus the number of bytes we would store if we logged every single message. Naturally, on Steele we log more messages, since it has the smallest node size. Increasing the node size should only increase the benefits. A traditional expectation is that the reduction in memory overhead goes down linearly with the node size. However, that is not always

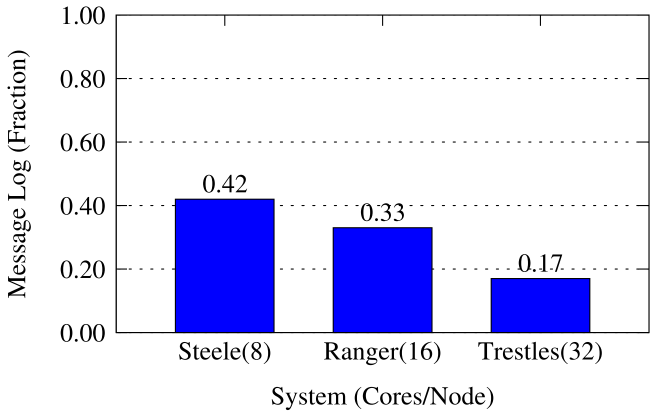

Fig. 4: Memory overhead relative to node size.

the case, since it all depends on the communication graph of the application and the assignment of objects to cores. In figure 4, we see that Ranger decreases the log size for about a fifth, relative to Steele. Trestles, however, practically halves the memory overhead compared to Ranger. We note that a smart mapping of objects to cores can make this reduction even more dramatic. In other words, if highly connected objects are mapped to the same node, not only may we get a performance boost, but also a significant reduction in message log.

V. Analysis of Reliability

The two protocols presented in section IV for multicore machines, checkpoint/restart and message logging, are designed to tolerate one single failure at a time. Multiple-node 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.

The checkpoint/restart mechanism 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. We require the buddy relationship to be bidirectional, i.e., if node X is the buddy of Y, then Y is the buddy of X. We want to answer the question of how resilient this protocol 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 buddy 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. In the end, the total number of subsets size f that will not make the whole system to collapse is $n(n-2)(n-4)...(n-2(f-1))/f!$, which gives us the following expression for the probability of surviving f concurrent failures:

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

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. In the causal message logging scheme, each determinant is replicated at least once on the memory of other node, in addition to the node where it was generated. This means, it will support with total certainty one single failure at a time. A multiple-node 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 g nodes x communicates with still alive.

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 in 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 formulae for the two protocols we get the probability of surviving a multiple-node crash based on different values of g and different values of f. Figure 5 presents the checkpoint/restart curve and four different curves for message logging with degree g equals to $\{2,4,8,16\}$ for a value of $n = 1024$. We see that all message logging curves drop more drastically than the checkpoint/restart case. That is the price of being susceptible to communication. The higher the value of g, the more rapidly the curve plummets.

As the number of nodes involved in a crash increases, the probability of survival decreases. However, the chance of having a failure with that many nodes decreases as well. Let us define *survivability* as the probability of survive *any* crash,

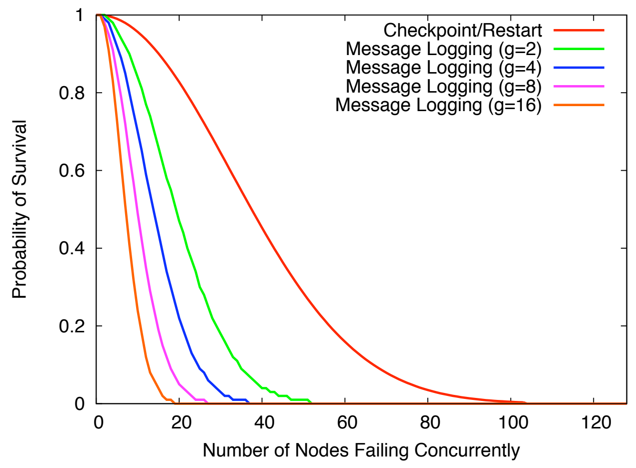

Fig. 5: Probability of surviving a multiple-node failure.

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.

We used the two functions discussed in section III with reasonable parameters [16] to model $p(i)$. Table I shows the survivability values for the different fault tolerance strategies. 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, that 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 I: Survivability

	Geometric	Zipf's
Checkpoint/Restart	0.9997	0.9992
Message Logging (g=2)	0.9988	0.9966
Message Logging (g=4)	0.9980	0.9945
Message Logging (g=8)	0.9964	0.9911
Message Logging (g=16)	0.9933	0.9854

VI. Conclusion and Future Work

This paper presented ongoing work on the design, implementation and analysis of fault tolerance strategies for multicore machines. We focused our paper on a message logging protocol that tolerates the crash of a single node and *may* survive a multiple-node failure. Our experiments showed that this protocol has low overhead. An analytical model helped us to determine the protocol has high resiliency.

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.

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] 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.

[4] 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.

[5] 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.

[6] 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.

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

[8] A. Guermouche, T. Ropars, E. Brunet, M. Snir, and F. Cappello, "Uncoordinated checkpointing without domino effect for send-deterministic mpi applications," in *IPDPS*, 2011, pp. 989–1000.

[9] T. Ropars and C. Morin, "Improving message logging protocols scalability through distributed event logging," in *Euro-Par (1)*, 2010, pp. 511–522.

[10] A. Bouteiller, T. Hérault, G. Bosilca, and J. J. Dongarra, "Correlated set coordination in fault tolerant message logging protocols," in *Euro-Par (2)*, 2011, pp. 51–64.

[11] 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.

[12] A. J. Oliner and J. Stearley, "What supercomputers say: A study of five system logs," in *DSN*, 2007, pp. 575–584.

[13] 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.

[14] 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.

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

[16] E. Meneses, X. Ni, and L. V. Kale, "Design and analysis of a message logging protocol for fault tolerant multicore systems," Parallel Programming Laboratory, Department of Computer Science, University of Illinois at Urbana-Champaign, Tech. Rep. 11-30, July 2011.

[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] 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.