Using Migratable Objects to Enhance Fault Tolerance Schemes in Supercomputers

Esteban Meneses, Xiang Ni, Gengbin Zheng, Celso L. Mendes, and Laxmikant V. Kalé, *Fellow, IEEE*

Abstract—Supercomputers have seen an exponential increase in their size in the last two decades. Such a high growth rate is expected to take us to exascale in the timeframe 2018-2022. But, to bring a productive exascale environment about, it is necessary to focus on several key challenges. One of those challenges is fault tolerance. Machines at extreme scale will experience frequent failures and will require the system to avoid or overcome those failures. Various techniques have recently been developed to tolerate failures. The impact of these techniques and their scalability can be substantially enhanced by a parallel programming model called migratable objects. In this paper, we demonstrate how the migratable-objects model facilitates and improves several fault tolerance approaches. Our experimental results on thousands of cores suggest fault tolerance schemes based on migratable objects have low performance overhead and high scalability. Additionally, we present a performance model that predicts a significant benefit of using migratable objects to provide fault tolerance at extreme scale.

Index Terms—migratable objects, fault tolerance, resilience, checkpoint/restart, message logging.

1 Introduction

Nature provides us with a vast set of examples of resilient systems. From short-term reconstruction of epidermal cells after a scratch to long-term adaptation of species to new environments, there is a built-in capacity in the biosphere to overcome failure. Unlike biological systems, the majority of applications in high performance computing (HPC) do not have an inherent ability to recover from failures. Most of the time, parallel programs are written optimistically, assuming that there will be no failures during execution.

Although failures were rare on the supercomputers of the past, which contained fewer components, that is not the case for some of the current machines and it will most likely not be true for future supercomputers. Figure 1 shows in two parts why fault tolerance is becoming a main concern as we approach exascale. The first part, plot 1(a), shows a historical view of the largest systems in the Top 500 list [1] according to the number of sockets. The growth of these systems has been exponential, from machines with a few thousand sockets in 1994 to a machine with more than a hundred thousand sockets in 2007. Although the number of cores per socket continues to increase, an exascale machine (expected to be delivered by 2018-2022) will contain more than 200,000 sockets [2]. The flip side of such an impressive rate of growth is an increased probability of component failure. The second part, plot 1(b), presents the expected mean-time-between-failures (MTBF) for a machine with that many sockets and the reliability per socket ranging from 5 to 80 years. When the number of sockets reaches the expected size of an exascale machine, the MTBF of the machine drops to a value that can be measured in minutes. That estimation is not pessimistic. Most predictions for failures at exascale foresee MTBF values of several minutes [2], [3]. In addition, several trends in architecture design (smaller feature size, near-threshold voltage) may even increase the rate of errors.

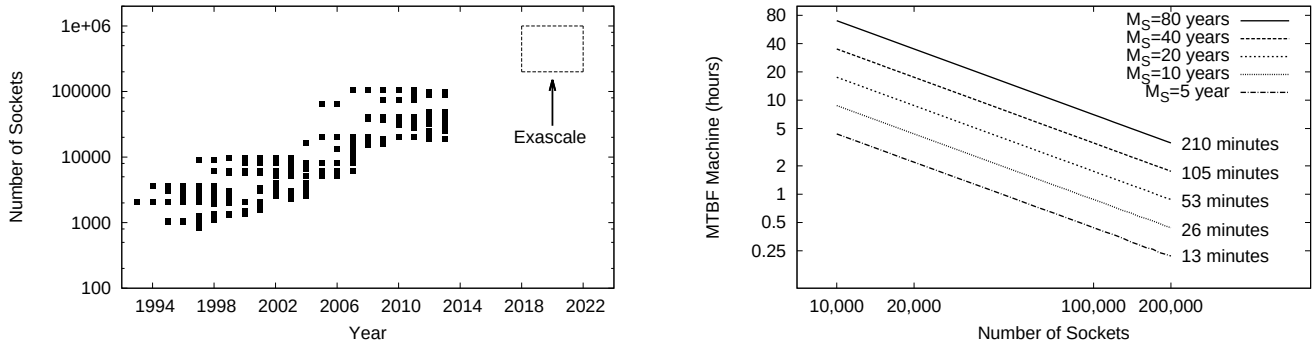
Aware of this situation, the HPC community has recently focused on developing a diverse range of fault tolerance techniques. There are protocols that *avoid* failures by relying on failure predictors and taking actions before the failure actually happens. Other protocols *overcome* failures by ensuring the system can recover from crashes of certain components.

Several of the fault tolerance strategies in the HPC literature can be enhanced by leveraging a model of parallel programming called *migratable objects*. In this model, an application is divided into small data and computation units. A smart runtime system is then responsible for assigning these units to nodes for execution. Additionally, the runtime system is able to migrate the units around to speed up the computation. The ability of over-decomposing a computation and migrating the units is what makes this paradigm a lever for different fault tolerance strategies. In this paper, we describe how this model has improved several fault tolerance strategies. We demonstrate the potential of these ideas with an implementation of the different protocols. We ran several parallel programs on real systems with thousands of cores. With the help of an analytical model, we project how these techniques will perform at extreme scale.

The contributions of this paper can be summarized as follows:

- A comprehensive and detailed description of various fault tolerance methods that have been enhanced using the migratable-objects model (§3, §4, §5).
- A comparative evaluation of these methods on thousands

- *E. Meneses is with the Center for Simulation and Modeling at the University of Pittsburgh, Pittsburgh, PA, 15260. E-mail: emeneses@pitt.edu*
- *X. Ni and L.V. Kalé are with the Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, IL, 61801. E-mail: {xiangni2,kale}@illinois.edu*
- *G. Zheng and C.L. Mendes are with the National Center for Supercomputing Applications, Urbana, IL, 61801. E-mail: {gzheng,cmendes}@illinois.edu*



(a) Historical trend in the number of sockets per machine.

(b) MTBF of a machine for various MTBF values per socket (M_S).

Fig. 1. Size and mean-time-between-failures (MTBF) of large scale machines.

of cores with an application set that includes programs from multiple fields and written in two different programming languages (§6).

- A performance model to predict the benefit of different approaches to fault tolerance support enhanced with migratable objects at extreme scale (§7).

Our vision of an enhanced fault tolerance scheme is based on our philosophy of what a resilient supercomputing system should be. The driving force in the HPC community is to build more powerful systems to solve a problem quickly or to solve a larger problem. In any case, the notion of *speedup* is fundamental. A bigger system accelerates the discovery of interesting scientific facts. Our vision of an ideal resilient runtime system is one that keeps the same execution speed despite the failures in the system. That means, a good fault tolerance mechanism keeps the *progress rate* of an application as high as possible.

2 Migratable Objects

Our model for parallel programming assumes a machine is composed of a collection of *processing entities* (PEs) connected through a network that does not guarantee in-order delivery of messages. The set of PEs is dynamic, i.e., it may grow or shrink depending on what nodes become available or are declared inaccessible. A failure, for instance, may render a PE inaccessible. The memory in each PE is private and the only way to share information is via message passing. In modern supercomputers, a multicore node would be the equivalent of a PE.

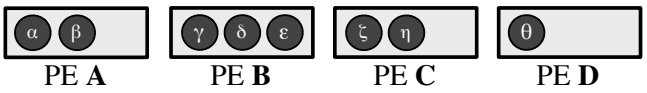
In the migratable-objects model, the programmer is expected to decompose a parallel program into a large number of objects. Each of these objects holds a portion of the data and performs part of the computation. These objects do not share memory, but may interact with other objects via messages. This mechanism provides a *message-driven execution* of the application. Each object is a reactive agent, responding to the messages it receives by potentially sending more messages. The number of objects in an application is independent of the number of PEs in the machine. This way, the programmer does not need to be aware of what size the system is. Instead, his responsibility is to *over-decompose* the application into many objects and coordinate the work among them. The average number of objects per PE is referred as *virtualization ratio*. A balance must be struck in defining the virtualization ratio. A higher value increases concurrency, but also the communication and synchronization overhead.

A runtime system is in charge of assigning the objects onto the set of PEs. This assignment may optimize for the peculiarities of the application, such as making highly connected objects reside on the same PE. Moreover, the mapping of objects to PEs is dynamic (i.e. it can be changed) during the execution of an application. The runtime system may shuffle objects around if it sees a potential performance benefit. For example, a load imbalance in the execution may result in objects being migrated to even the load across the set of PEs. The ability to move objects from one place to another requires each object to be *migratable*. This means that each object knows how to serialize its state to be shipped somewhere else in the system.

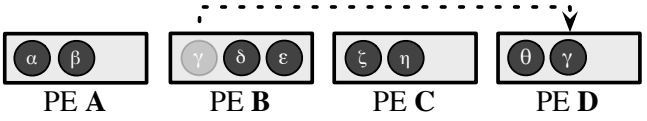
Figure 2 shows graphically the two properties previously explained: over-decomposition and migratability. In this case, a system contains 4 PEs (labeled from A to D) and 8 objects (named from α to θ). On Figure 2(a), we represent over-decomposition by showing each PE containing multiple objects. The distribution of objects into PEs does not need to be uniform, necessarily. Figure 2(b) presents the ability of the runtime system to migrate object γ from PE B to PE D. These two properties permit several novel capabilities in fault tolerance strategies. In the following sections, we will describe the way in which each of these strategies is improved.

CHARM++ [4] is an implementation of this model. In CHARM++, the programmer decomposes an application into a set of C++ objects, called *chares*. Each chare exposes a list of methods other chares can call. The set of chares share information via asynchronous method invocation. CHARM++ also provides an adaptive runtime system that handles object placement, load balancing, fault tolerance and many other tasks associated with the execution of an application. An extension to CHARM++, called AMPI [5], provides the set of abstractions in the migratable-objects model for MPI programs. In AMPI, an MPI application is run as a CHARM++ application, where each MPI rank is seen as a CHARM++ chare. AMPI allows MPI applications have over-decomposition and migration.

The migratable-objects model can be extended to include the fault tolerance dimension. We assume the underlying



(a) Over-decomposition: the program is composed by many objects spread throughout the set of PEs.



(b) Migratability: objects can be moved from one PE to another by the runtime system.

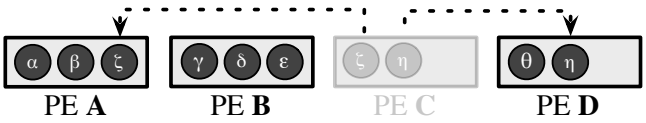
Fig. 2. Two fundamental properties of the migratable-objects model.

machine is not reliable and experiences failures with a certain frequency. Each failure knocks out one PE in what is known as the *fail-stop* model. A failed PE stops working and becomes nonfunctional for the rest of the execution. This means, a failed PE does not send messages out and all in-transit messages are lost. All the objects residing in the failed PE are lost. Mechanisms for recovering those objects are explained in the following sections. Spare PEs may replace the failed PEs. If the system has *spare* PEs available, then the replacement PE takes over the work of the failed PE. If no spare PEs are available, then the set of PEs shrinks by one PE. The runtime system adapts to this scenario and continues the execution with one less PE.

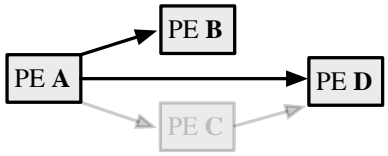
3 Proactive Fault Tolerance

The first fault tolerance mechanism we will discuss is a proactive approach that *avoids* failures by migrating the objects from PEs that are predicted to fail soon. This mechanism assumes that there is an agent in the system that predicts failures. Although failure prediction is a hard problem in HPC, there are many situations where measurements from different sensors can point to an impending failure [6], [7], [8]. If that is the case, the runtime system can receive a signal and proactively move away all the objects from the PE that is expected to fail. Other types of failures may not be predictable, but there is nothing that prevents combining a proactive approach with a *reactive* method (such as those in Sections 4 and 5).

It is easy to see how the migratable-objects model makes a proactive approach for fault tolerance more effective. All that is needed for the *evacuation* of a PE is already available in the model. The objects can be naturally migrated from their current PE to other safer locations and the system should be flexible enough to cope with the update of data structures for a correct execution. Figure 3 shows two basic functionalities that a proactive fault tolerance approach should have: migration of tasks and reconstruction of data structures. In Figure 3(a), the evacuation of PE C is illustrated. After the system receives the alarm of an impending fault in PE C, object ζ is moved to PE A and object η is moved to PE D. These migrations are naturally implemented in the migratable-objects model. Moreover, having multiple objects in one PE is not a problem, given the over-decomposition property of the model.



(a) Evacuation of a PE. All objects living on that PE are migrated upon the reception of an impending-fault signal.



(b) Spanning tree reconstruction. As soon as a PE is removed from the system, the spanning tree for collective communication operations is rearranged.

Fig. 3. Operations in proactive fault tolerance.

Figure 3(b) presents the modification of a spanning tree for collective communication operations. Initially, PE A is the root of the spanning tree with children PE B and PE C. PE D is a child of PE C. Once PE C is evacuated, the runtime system should reconstruct this spanning tree by making the necessary adjustments. The re-arrangement of the spanning tree only affects the parent PE A and children PE D of the warned PE C. The warned PE will first send the tree modification message to its parent and children. After receiving the message, parent and children store the changes but do not make them to the current tree until all outstanding collective communication operations are finished.

The major challenge of proactive fault tolerance is to keep the communication mechanisms effective when a migration occurs. The evacuation of a PE happens asynchronously with the execution of the program. The application does not stop to wait until a PE is evacuated. The runtime system must ensure that point-to-point communication works correctly during migration of objects. Scalable approaches for a correct message delivery in the face of asynchronous migration of objects can be found elsewhere [9]. Herein, we will describe what problems may arise and what data structures should be updated.

As pointed out in Section 2, the mapping of migratable objects to PEs changes dynamically. The system assigns each object to a *home* PE, which always knows where the object is currently on. This data structure is called the *object-to-home* mapping. However, an object may not necessarily reside on the home PE, but on a different *host* PE. For instance, imagine an object η whose home PE is B, but whose host PE is C. If a message from PE A targets η and PE A does not know where η resides, it will send the message to B, the home PE of η. PE B knows that η lives on PE C, so it will forward the message to PE C. Additionally, PE B will send a control message to PE A to update its routing table. The next time PE A sends a message to η, it will send it directly to PE C.

As we mentioned earlier, a proactive fault tolerance approach can be combined with other fault tolerance strategies and even with a load balancing framework. The migratable-objects model has the capability to rearrange the objects among the PEs to improve performance and speed up the application. It is natural to assume that a load imbalance

will arise as a result of an evacuation. So, an equally natural decision is to run a load balancer after the evacuation.

An implementation of the ideas described in this section is available in the CHARM++ runtime system [10]. The utility of this approach for MPI applications has been demonstrated [11]. The implementation of AMPI allows the runtime system to migrate AMPI threads even when messages are in flight. This means a thread may have multiple outstanding MPI requests when it is migrated. If a thread migrates from PE C to PE D, the queue of requests is also packed on PE C and sent to PE D. On PE D, the queue is unpacked and the AMPI thread restarts waiting on the queued requests. However, almost all the outstanding send and receive requests are associated with a user-allocated buffer where the received data should be placed. Packing and moving the buffers would cause the buffers have different addresses on the destination PE. One way to solve this problem is by using the *isomalloc* technique proposed in PM^2 system [12]. This technique reserves a unique range of virtual address space for each thread. That way, there is no threat of memory violations after migration.

4 CHECKPOINT/RESTART

In *reactive* fault tolerance, the objective of the protocols is to *overcome* a failure by providing a recovery mechanism after one component fails. We assume the system has a failure-detection mechanism. Once the failure is detected, the runtime system starts a recovery protocol that will bring the application back on track with the execution. We will assume that recovery is *automatic*. That means, the user does not need to be aware that a failure has happened, the runtime system will take care of the failure without the intervention of the user.

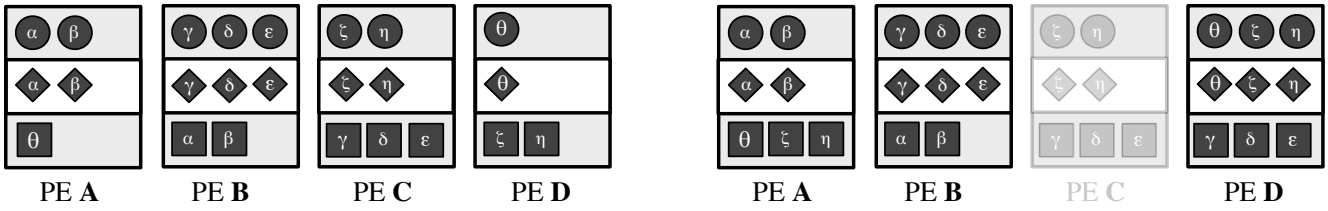
Checkpoint/restart is easily the most popular technique in HPC to provide fault tolerance. It is simple and effective enough in situations where failures are relatively rare. The fundamental principle of checkpoint/restart is to periodically save the state of the whole system. If a component crashes, the system rolls back to the most recent checkpoint and restarts execution from there. Although the workings of checkpoint/restart are straightforward, it adopts many variants.

The checkpoint of a system can be *coordinated* if the different components agree on when to store their state. The Chandy-Lamport algorithm for global snapshots [13] creates a collection of node checkpoints plus in-flight messages that constitute a global checkpoint. An *uncoordinated* protocol, on the other hand, allows nodes to checkpoint at their own discretion. However, collecting all node checkpoints does not make a consistent global checkpoint. If messages are not stored, then this scheme may suffer *cascading rollback*, a pathological situation where the rollback of one node may require the entire system to rollback several checkpoints. Coordinated checkpoint can be *blocking* or *non-blocking*, depending on whether the application has to stop execution during checkpoint, or the checkpoint process executes along with the application. A comparison between the two approaches can be found elsewhere [14]. A compromise between those two types of checkpointing, called *semi-blocking* checkpoint [15] requires the application to reach a synchronization point, after which the checkpoint runs asynchronously with the application.

When it comes to generate a global checkpoint, the amount of data stored can differ according to what is included in the checkpoint [16]. In a *system-level* checkpoint, the whole state of the machine (including the complete address space of processes, CPU registers, file descriptors) is to be saved. This mechanism makes the application oblivious of the checkpoint. BLCR [17] is a library that implements this abstraction. Conversely, *application-level* checkpoint makes the application an active participant of the checkpoint process. The user must write a checkpoint function and decide what to store in a checkpoint. That way, the amount of data to checkpoint may be dramatically reduced. Additionally, the knowledge of the programmer is used to insert the checkpoint calls at appropriate places. SCR [18] is a library that implements this method. The migratable-objects model encourages *runtime-based* checkpoint [19], where the runtime system provides an interface for the programmer to write the checkpoint methods. The runtime system may also participate more actively in deciding when to trigger a checkpoint. The migratable-objects model can work with transparent checkpoints. Checkpoint transparency and migratability are two orthogonal dimensions.

A global checkpoint of the system, obtained with any of the mechanisms above, used to be stored only in the file system. This was a natural place to store the checkpoint, since the file system would survive the crash of a node. However, the file system bandwidth can not easily cope with the increasing size of the supercomputers and the data size that needs to be checkpointed. The file system quickly becomes a bottleneck during checkpoint. Various alternatives have been explored to solve this problem. One popular choice is to store the checkpoint in local storage (either main memory, disk or solid-state drive). One such protocol is called *double in-memory* checkpoint/restart [19]. In this method, a PE stores copies of its checkpoint in its own memory and in the memory of a *buddy* PE. Figure 4(a) illustrates the memory footprint of this approach. PEs A through D contain several objects each. The system uses a cyclic buddy assignment, where PE B is the buddy of A, C is the buddy of B, and so on. At worst, this mechanism triples the memory requirements of the application, but it is able to checkpoint rapidly, scale to large systems and it is applicable for a wide range of HPC applications [20]. Extensions of this basic protocol drastically reduce the memory footprint [21].

Double in-memory checkpoint/restart tolerates a failure by using spare PEs to substitute for the failed ones. For example, in Figure 4(a), if PE C fails, a replacement PE will receive the checkpoint from PE D and the system can continue execution. However, the migratable object model empowers this scheme in several ways. First of all, depicted in Figure 4(b) is the scenario where spare PEs are not available. In such circumstances, there is no replacement for the failed PE C. The adaptive runtime system solves this situation by distributing the objects of C into the rest of the system. For this particular case, all the objects that were on C are moved to D. The buddy assignment is updated and the checkpoint placement corresponds to this new assignment. Other data structures have to be adjusted as well, such as the spanning trees for collectives. For a more detailed discussion on how to update



(a) Double in-memory checkpoint/restart. The checkpoint of an object is stored in two places: the local memory of its host PE and the remote memory of the buddy PE.

(b) Recovery from failure in the migratable-objects model. After PE C fails, PE D takes over C's objects. PE A gets more remote checkpoints.

Fig. 4. Memory footprint and failure recovery in double in-memory checkpoint/restart. Circles represent objects in application, rhombi are local checkpoints and squares are remote checkpoints.

those structures, we refer the reader to Section 3.

The second way in which migratable objects improves this approach is by offering a load balancing framework in the case of no spare PEs. Once a failure hits the system and PEs are lost as a result, the system can even the burden if a PE ends up with a much higher load than the average. Finally, migratable objects provides the right environment for serialization methods to be written in a simple way. The runtime system naturally handles migration of the objects, because that is an intrinsic characteristic of the model.

An implementation of double in-memory checkpoint and other asynchronous checkpoint methods can be found in the CHARM++ system [10]. These protocols also work for MPI applications through the AMPI extension. Specific versions of these protocols are also available for a version of the runtime system specific to systems with multicore nodes [22].

5 MESSAGE LOGGING

Although checkpoint/restart is a very popular alternative in HPC to provide fault tolerance, it embodies a fundamental disadvantage. It requires a *global* rollback: all PEs have to roll back to the latest checkpoint in case of a failure. That downside becomes critical in an extreme-scale system; millions of PEs would have to roll back if one of them fails, resulting in a massive waste of time and energy.

Message logging is a technique that avoids global rollback by saving the messages an application sends and only rolls back the failed PE. It then requires only a *local* rollback and saves energy by having the rest of the system idle or making progress on their own [23]. It may save time too, because messages have no delay or contention during recovery. Additionally, it allows the checkpoint to be either coordinated or uncoordinated. In case of a failure of PE A, all other PEs that have stored messages to A will re-send those messages upon PE A's failure. To catch up with the rest of the system, PE A will sort the re-sent messages and process them. To provide a correct recovery, message logging requires storing information about non-deterministic events. Message reception is, in general, non-deterministic. Thus, every time a non-deterministic event occurs, a *determinant* is generated. A determinant will contain all information required to ensure recovery reaches a consistent global state. This mechanism is based on the *piece-wise deterministic assumption* (PWD) [24], which states that logging determinants is enough to guarantee a consistent recovery. For example, a determinant could be formed by the tuple $\langle sender, receiver, ssn, rsn \rangle$. Both $sender$ and $receiver$ represent objects. The *send sequence number* (ssn) is a unique identifier for each message, assigned by the sender. The receiver will generate a *receive sequence number* (rsn) upon reception of the message. The rsn totally orders the reception of the message and provides a strict sequence in which messages have to be processed during recovery. There are several message-logging protocols [25] that differ in the way they handle determinants. *Causal* message-logging makes the determinants travel with the messages that causally depend on them. More specifically, determinants are piggybacked on application messages until they are safely stored. A specific protocol in this family, called *simple causal* message-logging [26] has demonstrated scalability and low overhead. Strategies to decrease the memory overhead of the message log can be found elsewhere [27].

Figure 5 illustrates how message logging works. Using the same scenario as in Figure 2(a), we see a portion of the execution of an application. Every message is stored at the sender PE. Each message reception generates a determinant and that determinant has to be stored on at least one PE, aside from the one that generated it. For instance, message m_1 generates determinant d_1 at PE C. The next message leaving C, m_2, carries d_1. Eventually, d_1 gets stored in PE D and the acknowledgement message is sent from D to C. Upon the reception of the ACK message, PE C stops piggybacking that determinant. After m_3 is received, its determinant, d_3, has to be stored somewhere else. Consequently, message m_4 piggybacks d_3. Figure 5 presents the failure of PE C and the loss of all objects on that PE. The checkpoint buddy of C, PE D, provides the latest checkpoint of objects ζ and η to the replacement of PE C, named PE C'. We assume a pool of spare PEs for this protocol. During restart, PE C' receives the determinants stored in other PEs. These determinants will guarantee that subsequent messages are processed in the same order as before the crash. Once all determinants have been collected and it has been verified there are no missing determinants, PE C' resumes execution by processing the messages re-sent from all other PEs. Once PE C' starts processing the messages, the PWD assumption ensures PE C' will send the same messages PE C sent before the crash. This means, messages m_2 and m_4 will be sent

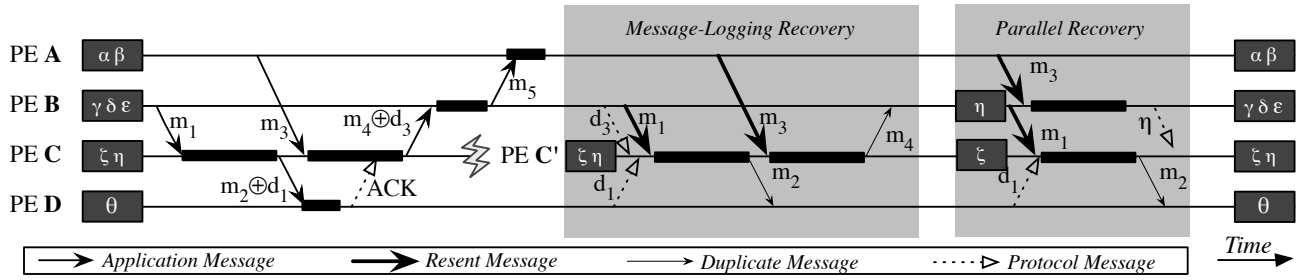


Fig. 5. How message logging and parallel recovery work.

again. The receiver of those messages will detect they are duplicate messages and will avoid processing them. Detecting duplicate messages is straightforward since the send sequence number uniquely identifies each message. A duplicate message is simply a message with an ssn that has been processed before. There are different types of messages in Figure 5. The first type are the regular *application messages*. Second, the *resent messages* are those resent as part of the recovery process by PEs that did not crash. The third type are the *duplicate messages* that are messages sent during recovery from the PE that crashed. Finally, *protocol messages* are the additional messages required to provide a consistent fault tolerant scheme. This category includes checkpoint messages, determinant messages and acknowledgements.

5.1 Parallel Recovery

The migratable-objects model provides a fundamental advantage for message logging. A key observation, concerning many HPC applications, is that most codes are tightly coupled and if one PE fails, the rest of the system will remain idle until the failed PE catches up with the execution of the application. Instead of waiting idle, surviving PEs can help accelerate recovery by receiving objects from the failed PE and perform what we call *parallel recovery* [28]. Objects living on a failed PE are distributed among other PEs for a speedup in recovery. Figure 5 also shows the parallel recovery in the same base scenario. The difference appears when object η does not return to PE C' but gets distributed to PE B. Therefore, objects ζ and η are effectively recovered in parallel on PEs C and B, respectively. The distribution of objects for recovery will create some messages to change their source or destination. Note message m_3 was originally resent to object η and has to be redirected to PE B. The same is true for determinants. In this example, determinant d_3 must be forwarded to PE B. Message m_4 now comes from PE B instead of PE C because it is sent by object η.

The distribution of objects to achieve parallel recovery creates a transient load imbalance. Imagine the distribution of objects in Figure 5 provides an even load among the PEs. After PE C crashes, object η is migrated to PE B to be recovered in parallel with object ζ on PE C'. Once the recovery is finished, object η does not return immediately to PE C', but waits until the next checkpoint. During this time period, between the completion of recovery and the next checkpoint, the system suffers a load imbalance. More precisely, PE C' will be underloaded, whereas PE B will be overloaded.

Parallel recovery empowers message logging by increasing the progress rate during recovery. If failures are common, parallel recovery is able to recover faster and make progress even in the case where the MTBF is smaller than the checkpoint period. Parallel recovery is one of the signature features of fault tolerant CHARM++. It has been implemented and tested with several different message-logging protocols. A comparison of the implementation of those protocols in CHARM++, and which represent a better opportunity for parallel recovery, can be found elsewhere [26].

6 EXPERIMENTAL EVALUATION

6.1 Setup

We ran our experiments on *Ranger*, *Lonestar*, and *Stampede* supercomputers at Texas Advanced Computing Center (TACC). Ranger is a 579-teraFLOPS machine with a total of 62,976 compute cores. Each node contains a 16-way SMP processor and 32 GB of memory. Lonestar is 300-teraFLOPS computer with 22,656 cores. Each node has a 12-way SMP processor and 24 GB of memory. Stampede is a 10-petaFLOPS machine with more than 96,000 cores divided into 6,400 nodes. Each node contains 32GB of memory. All supercomputers feature a fat-tree network topology on an Infiniband interconnect.

We chose a set of CHARM++ and AMPI programs to evaluate the different fault tolerance techniques. The first CHARM++ program is *Wave2D*, which runs a finite difference method to compute pressure information on a two-dimensional grid. *Jacobi3D* is a 7-point stencil that computes the transmission of heat on a three-dimensional space. The last CHARM++ code is *LeanMD*, a mini-application that emulates the communication pattern in NAMD [29]. It computes the interaction forces between particles in a three-dimensional space and this computation is based on the Lennard-Jones potential. We included various MPI programs in our evaluation. We adapted the NAS Parallel Benchmarks suite (NPB) to AMPI with migratable MPI threads. The NPB is a collection of linear algebra numerical methods [30]. We focused our experiments on four benchmarks from NPB: block-tridiagonal (BT), conjugate gradient (CG), multi-grid (MG) and scalar pentadiagonal (SP). Finally, we also ran *Sweep3D*, a mini-application that solves a neutron transport problem. Sweep3D uses discrete ordinates in a three-dimensional space. Table 1 summarizes the most important characteristics of the applications we used in the experiments.

Program	Language	Domain	Problem Size	Virt. Ratio
Wave2D	Charm++	Physics	32768^2	4
Jacobi3D	Charm++	Physics	2×2048^3	8
LeanMD	Charm++	Molecular Dynamics	256 K particles	28
NPB	MPI	Linear Algebra	class D,E	1,4
Sweep3D	MPI	Physics	250^3	4

TABLE 1
Main features of the programs used in experiments.

All the fault tolerance strategies discussed in this paper were implemented in the CHARM++ runtime system. For proactive fault tolerance, the preventive evacuation mechanism moves away the objects from a particular PE. Although we did not use a failure predictor, we provide an interface to plug a failure prediction module into the runtime system. To evaluate our evacuation framework, we used a mechanism to *inject* a warning into the system. This warning informs the system about the impending crash of a particular PE. The double in-memory checkpoint/restart mechanism serializes all the objects in the system every time the checkpoint call is made. These function calls have to be introduced into the code by the programmer. Identifying those synchronization points is fundamental to guarantee a consistent recovery. The same applies to the implementation of message logging. To simulate a crash, the failure injection mechanism allows the user to specify any number of crashes and the wall time at which each failure will occur. The runtime system will simulate a crash by making a particular PE unresponsive. Then, the failure detection mechanism (implemented through a pair-wise heartbeat) will raise a flag and the restart process will begin. This failure detection mechanism is scalable since it has a constant overhead for each PE. For all the experiments on reactive fault tolerance we assumed there were replacement PEs in the system.

6.2 Proactive Fault Tolerance

One of the most important features of an effective proactive fault tolerance approach is to provide a quick response mechanism. We investigated how rapidly a PE is evacuated by running Jacobi3D with 2,048 cores on Stampede. Table 2 presents the evacuation time when the data size per core ranges from 16 to 512 MB. The total time to migrate away all the objects on a PE can be measured in milliseconds and linearly depends on the data size. There are two datasets in the table, representing two important events in evacuation. The *local confirmation* stands for the moment when the failing PE has released all the objects. The *remote* confirmation represents the time when the failing PE has received a confirmation from all the destinations of the objects. A remote confirmation is expected to increase the local confirmation by a roundtrip through the network and the processing of the objects, which can be seen as the constant difference between the two datasets. The real evacuation time lays somewhere between the two, and it is constant regardless of the system size.

Data Size (MB)	16	32	64	128	256	512
Local ACK (ms)	11	26	50	105	206	428
Remote ACK (ms)	530	542	582	647	761	1054

TABLE 2
Fast evacuation time with different data sizes.

The ideal complement to a fast evacuation mechanism is a load balancing framework. Once a PE is evacuated, the additional objects assigned to the receiving PEs may cause load imbalance. To even out the load in the collection of PEs, a load balancer looks for a redistribution of the objects to decrease the load excess on any PE. Figure 6 shows the interaction of evacuation and load balancing in NPB-BT multi-zone with 256 cores on Ranger. This benchmark has an initial load imbalance that is later solved by calling a load balancer right before iteration 20. The effect of the load balancer is dramatic. The average iteration time is drastically reduced, providing a speedup of 2.65. Then, at iteration 70 the system receives a warning of an impending failure and evacuates a PE. That creates a load imbalance in the system, which increases the iteration time by 22%. Finally, that loss in performance is solved by applying the load balancer once again and bringing down the iteration time to a level similar to the one before the evacuation (little over 1% overhead).

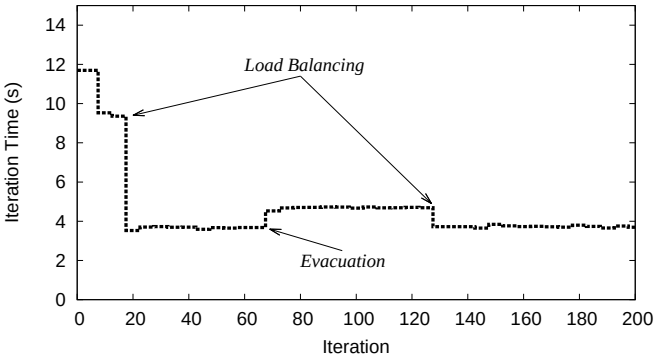


Fig. 6. Effect of evacuation and load balancing on performance.

6.3 Checkpoint/Restart

To illustrate the full potential of the migratable-objects model in reducing the checkpoint overhead to a small level, we show the results of checkpoint and restart with two different types of applications. For weak-scaling, we use Jacobi3D and for strong-scaling LeanMD. LeanMD computes the forces between particles in an iterative fashion. By placing the checkpoint calls at synchronization points between iterations, we manage to checkpoint only the fundamental data: the position of particles. All other intermediate data structures are not stored as part of the checkpoint. In doing that, the size of a checkpoint drastically decreases. Similarly, Jacobi3D only stores the necessary data structures in the checkpoint. All other temporary data structures are not included. The top part of Figure 7 presents the checkpoint time for the two applications on Stampede. These results show the time to

checkpoint is fast (measured in milliseconds) and that the checkpoint framework scales well. The bottom part of Figure 7 shows the restart time, that includes notifying all the system about the crash, synchronizing the rollback of all PEs and retrieving the checkpoint of the failed PE. The total restart time is constant for the weak-scaling experiment. The strong-scaling case shows that the restart time is initially dominated by the checkpoint transmission, but later the synchronization cost prevails.

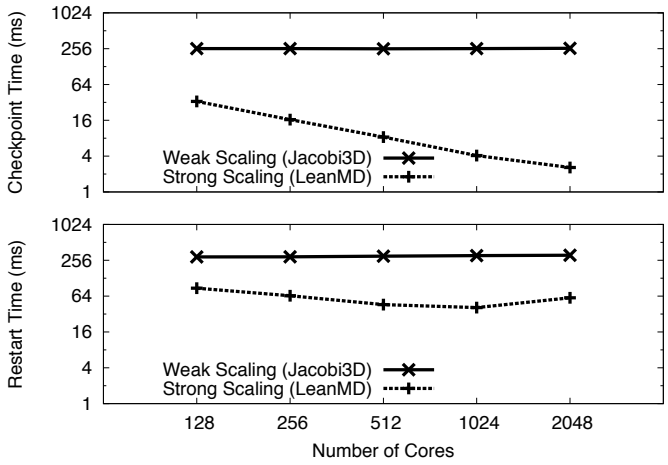


Fig. 7. Fast checkpoint and restart time.

6.4 Message Logging

Message logging can be seen as an improvement of checkpoint/restart that prevents it from rolling back all PEs after a crash. If only the crashed PE is required to roll back and restart, important energy savings can be obtained [23]. However, message logging needs some metadata to be managed. In particular, determinants must be generated, piggybacked and properly handled to guarantee a consistent recovery. That imposes some overhead. Figure 8 shows the execution-time overhead in different applications with 1,024 cores on Ranger. The range of message-logging overhead varies from 0.8% to 10.0%. There are many variables that determine how high the overhead of message logging will be. Extremely relevant are the communication characteristics and the computation/communication ratio of the application. An application that sends many messages with high frequency will require many determinants to be generated and processed. This is the case of NPB-SP and Sweep3D. Conversely, if the application features a high amount of computation, the communication overhead can be hidden to a certain degree. This is the situation with Wave2D, LeanMD and NPB-MG. Also, if the application is communication bound, the additional burden of determinants will impact the performance more drastically. NPB-CG is a good example of this scenario. Jacobi3D and NPB-BT have a relatively dense communication graph that increases message-logging overhead.

We measured how message logging scales in both weak-scaling and strong-scaling settings on Ranger. Jacobi3D was used to run a weak-scaling test, whereas LeanMD served as the test code for a strong-scaling experiment. Several runs were

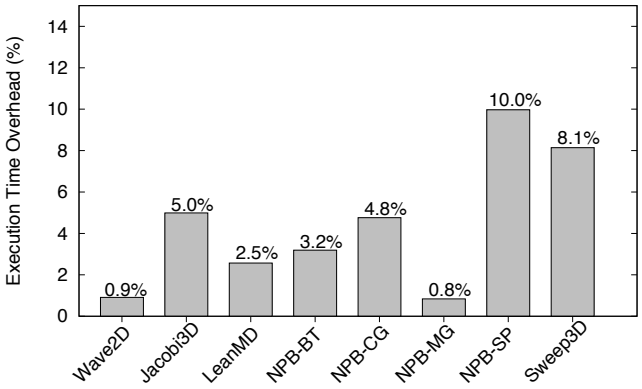


Fig. 8. Execution-time overhead of message logging.

executed and average time is reported. Figure 9 shows the results for the weak-scaling experiment. The overhead is approximately constant throughout the whole spectrum and it is close to 5%. The strong-scaling experiment of Figure 9 shows an interesting story. We ran two different molecular systems to test the effect of a larger problem size. The $256K$-particle case provides evidence of how a communication-bound scenario affects message logging. When the computation/communication ratio is small, the overhead of message logging cannot be hidden by the computation in the application. A different situation occurs when a larger problem size is used. The $1M$-particle case shows how message logging scales better and has an overhead around 3%.

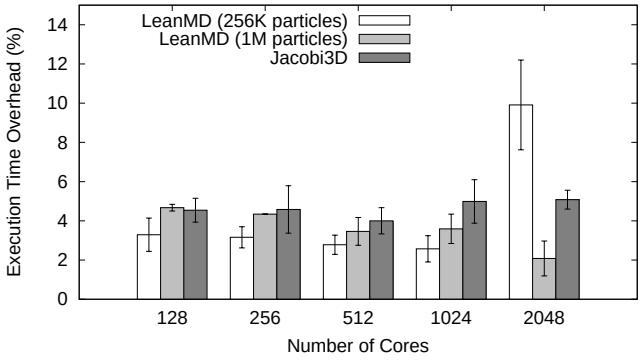


Fig. 9. Strong and weak scaling of message logging.

Migratable objects enhance message logging by allowing parallel recovery. If objects from the failed PE are distributed across other PEs to be recovered in parallel, then recovery time can be reduced to a fraction. Additionally, migratable objects allow overlapping the latency of the determinant-acknowledgment cycle with computation for other objects. Figure 10 presents a *progress diagram*, which shows the progress of an application (using an application-specific metric) versus time. In an iterative application, the number of completed iterations can be used as a progress value. A progress diagram is a useful visualization device, because it allows the viewer to focus on what really matters in a fault-tolerance strategy for HPC: progress rate. Not only is

it important for a fault tolerance method to have low overhead (a small increment in the slope of the curve in a progress diagram), but also to provide a fast recovery. We ran Jacobi3D with 256 cores on Ranger and compared the progress rate of checkpoint/restart and message logging. Figure 10 shows the two approaches and the recovery time (shaded region below the curve) when a failure is introduced at second 50 of the execution. The run executes a total of 200 iterations and checkpoints at iterations 40 and 160. Even when message logging incurs an overhead of 5%, it manages to recover the work lost in a failure much faster than traditional checkpoint/restart. Whereas checkpoint/restart takes more than 30 seconds to recover, parallel recovery manages to bring down that time to less than 5 seconds. In this particular test, 8 PEs helped in the recovery of a failed PE.

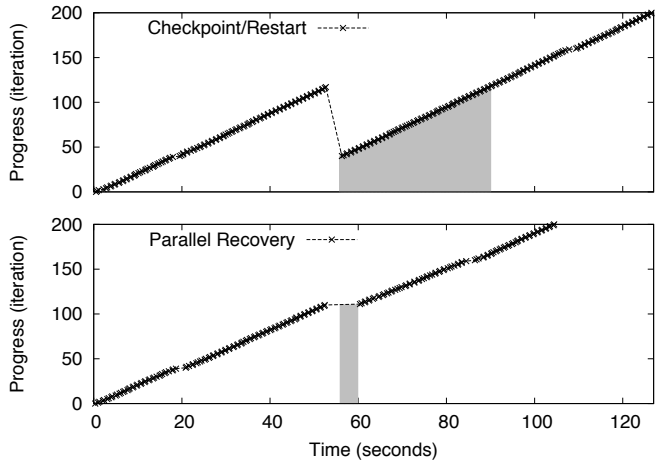


Fig. 10. Progress rate diagram and recovery.

The more PEs are available to help in recovery, the faster the failed PE can catch up with the rest of the system. Table 3 shows this effect. Using the same scenario as Figure 10, we changed the number of PEs helping to recover, from 1 (no parallel recovery) to 8. It is important to mention the superlinear effect in Table 3. Message logging alone can recover faster than normal execution. The reason is that for this particular test, during recovery only one PE is executing. It receives the messages it needs to keep making progress all at once. Recovery then becomes computation bound, even if the application is otherwise not computation-intensive. Network latency becomes roughly zero and there is no synchronization delay of any kind.

P	1	2	4	8
σ	2.53	3.65	4.99	7.26

TABLE 3
Recovery speedup (σ) as parallelism level (P) increases.

7 Performance Model

To predict the performance of the different fault tolerance approaches presented in the previous sections, we developed a model to estimate the total execution time for each approach under different circumstances. Table 4 presents a list of the parameters of the model along with a short description for each. A parallel application requires W time units to finish execution in a particular system that has a mean-time-between-failures of M. Proactive fault tolerance requires 3 parameters (κ, π and ρ). Reactive fault tolerance requires 3 parameters for checkpoint (δ, τ and R) and 5 more for message logging with parallel recovery (μ, P, σ, λ, and κ).

Parameter	Description
W	Time to solution in a fault-free scenario
M	Mean-time-to-interrupt of the system
T	Total execution time
δ	Checkpoint time
τ	Optimum checkpoint period
R	Restart time
κ	Evacuation/migration cost
π	Precision of failure predictor
ρ	Recall of failure predictor
μ	Message-logging slowdown
P	Available parallelism during recovery
σ	Parallel recovery speedup
λ	Parallel recovery slowdown

TABLE 4
Parameters of the performance model.

The main goal of the performance model is to predict the total execution time for an application that runs on a faulty system. The model requires all parameters in Table 4 as inputs, with the exception of two: the checkpoint period and the total execution time. The checkpoint period τ will be set by the model. Part of the goals of this model is to find the value of the checkpoint period that minimizes the total execution time. The output of the model is T, the total execution time. A fundamental insight provided by this model is to predict the potential advantage of different methods under different circumstances. Figure 11 presents the execution assumptions in the model. The system periodically performs a global coordinated checkpoint with duration δ. Then, it executes for τ time units. This pattern is repeated until a failure disrupts it. The figure shows PE C failing and being replaced by a spare PE. The failure occurs t time units after the last checkpoint. The system must recover those t time units of execution. In checkpoint/restart all PEs must rollback to the previous checkpoint. In the case of message logging, only the crashed PE rolls back. If parallel recovery is used, only the crashed PE rolls back, but other PEs help during recovery.

An important guideline in the model is that checkpoint/restart is assumed as the basic fault tolerance infrastructure. This base will be augmented with the different approaches we discuss in this paper. We first present the model for a checkpoint/restart mechanism that also has a failure predictor and can proactively migrate objects before the PE they reside in crashes. After that, we present a model for checkpoint/restart enhanced with parallel recovery. Finally, we describe a comprehensive model that includes both proactive evacuation and parallel recovery.

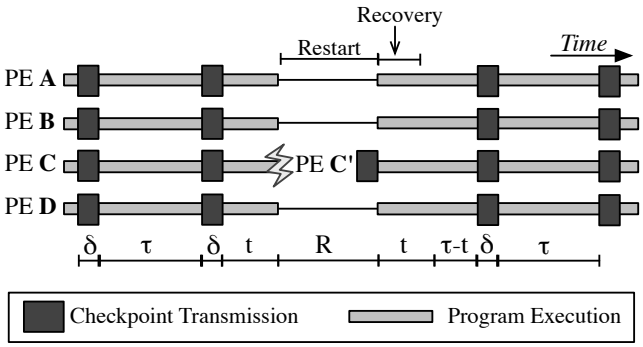


Fig. 11. Execution framework in the performance model.

7.1 Checkpoint/Restart

The frequency of checkpoints has an important impact on the performance of checkpoint/restart. If the system checkpoints too often, the overhead of dumping the state of the system may be high enough to make it impractical. Conversely, if the system checkpoints too seldom, chances are that a failure will make the system re-execute a huge portion of the already completed work. Clearly, a sweet spot must be found to optimize the use of checkpoint/restart.

An early analytical model to determine the optimum checkpoint period was developed by Young [31]. This model was later extended by Daly [32] to provide a higher-order estimate. The parameters of the model appear in the first section of Table 4. Essentially, this basic model defines the components of the total execution time T:

$$T = T_{Solve} + T_{Checkpoint} + T_{Restart} + T_{Recover} \quad (1)$$

where T_{Solve} stands for the computation time to solve the problem, $T_{Checkpoint}$ is the overhead of dumping the state of the system, $T_{Restart}$ is the time spent on setting up the system to resume execution after a failure (which includes the time to obtain the latest checkpoint), and $T_{Recover}$ represents the time to recover the lost work. Using the input parameters, the basic model transforms Equation 1 into the following:

$$T = W + \left(\tfrac{W}{\tau} - 1\right)\delta + \tfrac{T}{M}R + \tfrac{T}{M}\left(\tfrac{\tau+\delta}{2}\right) \quad (2)$$

It is possible to analytically compute an expression for the optimum checkpoint period τ using Equation 2. However, a simple expression for τ that is usually applied as a rule-of-thumb is [32]: $\tau = \sqrt{2\delta M} - \delta$.

7.2 Checkpoint/Restart with Evacuation

We can extend the performance model above to include proactive evacuation of PEs. This new model assumes an imperfect failure predictor. A failure successfully predicted will be proactively avoided by evacuation. However, there may be failures that will not be predicted and then the system will reactively rollback and resume execution. Also, there might be false alarms if the predictor incorrectly raises a warning flag about a failure that never occurs. Thus, the accuracy of the failure predictor will impact the whole performance of the fault tolerance framework. To capture the efficiency of the failure predictor and better model the performance of a proactive approach, we will introduce three new parameters. Firstly, κ will stand for the evacuation cost. Regardless of the final result on a failure prediction, this will always be the cost of evacuating one PE. Secondly, we will define two traditional terms in information retrieval, *precision* and *recall*, to measure the accuracy of the predictor [7]. Precision and recall will be denoted by π and ρ, respectively. The formulas for both of them are presented below:

$$\pi = \frac{\mathcal{T}_P}{\mathcal{T}_P + \mathcal{F}_P} \qquad \rho = \frac{\mathcal{T}_P}{\mathcal{T}_P + \mathcal{F}_N}$$

where symbols $\mathcal{T}_P, \mathcal{F}_P$ and $\mathcal{F}_N$ represent true positives, false positives and false negatives, respectively. These quantities are commonly used in statistical hypothesis testing. The subscripts P and N represent the prediction of the failure predictor, whereas $\mathcal{T}$ and $\mathcal{F}$ stand for the correctness in the prediction. For example, $\mathcal{T}_P$ is the number of correctly predicted failures (they were positively detected as failures and the prediction was true) and $\mathcal{F}_N$ is the number of missed failures.

To accommodate the new parameters for proactive fault tolerance in the basic checkpoint/restart model, we extend Equation 1 to include evacuation time:

$$T = T_{Solve} + T_{Checkpoint} + T_{Evacuate} + T_{Restart} + T_{Recover} \quad (3)$$

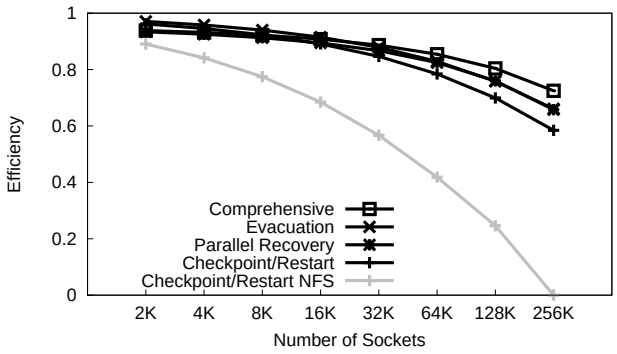
To factor in the possibility of the failure predictor to be wrong, we include parameters κ, π and ρ:

$$T = W + \left(\tfrac{W}{\tau} - 1\right)\delta + \tfrac{T}{M}\tfrac{\rho}{\pi}\kappa + \tfrac{T}{M}(1-\rho)R + \tfrac{T}{M}(1-\rho)\left(\tfrac{\tau+\delta}{2}\right) \quad (4)$$

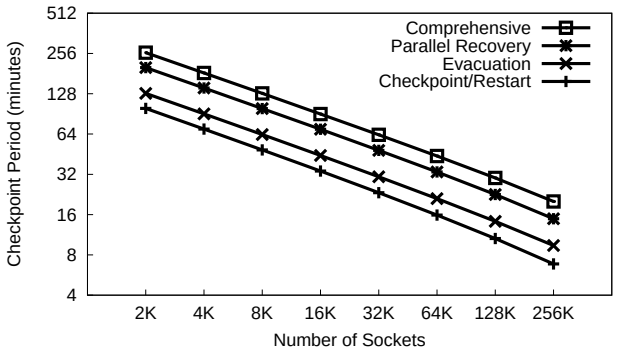
where the fraction $\frac{\rho}{\pi}$ represents the ratio of number of warnings with respect to the number of failures. If we multiply this quantity times $\frac{T}{M}$ (the expected number of failures in an execution), we will end up with the total number of warnings raised by the failure predictor (including false positives).

7.3 Checkpoint/Restart with Parallel Recovery

The checkpoint/restart model of Equation 2 can be extended to use message logging and parallel recovery. Table 4 contains the list of necessary parameters to incorporate parallel recovery. First of all, μ stands for the performance overhead introduced by the message-logging protocol (message copying, but especially determinant handling). To allow a recovery in parallel, we introduce P, that stands for the maximum parallelism that can be achieved during recovery. This number depends on the virtualization ratio of the application. In general, we should not expect P to be higher than virtualization ratio. Parameter σ captures the actual speedup achieved by recovering in parallel [28]. This speedup applies only to the first t time units after the crash (see Figure 11). Once the crashed PE catches up with the rest of the system, all PEs resume execution until the next checkpoint is reached. The last $\tau - t$ time units of the checkpoint period are executed with a load imbalance, since some PEs have received additional objects. Parameter λ accounts for that slowdown. The following checkpoint will return the migrated objects to the original PE and solve the



(a) Scaling of efficiency. The traditional checkpoint/restart scheme will not reach exascale. Other alternatives, empowered by the migratable-objects model, present a viable solution.



(b) Scaling of optimum checkpoint period. Higher failure rate forces the system to checkpoint more often. Parallel recovery and failure avoidance increase the checkpoint period significantly.

Fig. 12. Efficiency and checkpoint period of fault tolerance schemes at different scales.

load imbalance (κ). The extended equation to include parallel recovery considerations is the following:

$$T = W\mu + \left(\frac{W\mu}{\tau} - 1\right)\delta + \frac{T}{M}(R+\kappa) + \frac{T}{M}\left(\frac{\tau}{\tau+\delta}\left(\frac{\tau}{2\sigma} + \frac{\tau}{2}(\lambda-1)\right) + \frac{\delta}{\tau+\delta}\left(\frac{\tau}{\sigma} + \frac{\delta}{2}\right)\right) \quad (5)$$

7.4 Comprehensive Approach

Finally, since both proactive evacuation and parallel recovery are not mutually exclusive, we can merge the two in one single approach. We call this strategy comprehensive fault tolerance. Combining equations 4 and 5, we obtain:

$$T = W\mu + \left(\frac{W\mu}{\tau} - 1\right)\delta + \frac{T}{M}\frac{\rho}{\pi}\kappa + \frac{T}{M}(1-\rho)(R+\kappa) + \frac{T}{M}(1-\rho)\left(\frac{\tau}{\tau+\delta}\left(\frac{\tau}{2\sigma} + \frac{\tau}{2}(\lambda-1)\right) + \frac{\delta}{\tau+\delta}\left(\frac{\tau}{\sigma} + \frac{\delta}{2}\right)\right) \quad (6)$$

7.5 Large-Scale Projections

The major goal of the performance model is to provide a prediction for large-scale executions. We use the analytical framework developed in this section to estimate the total *efficiency* of a system. In this context, efficiency is defined as the ratio of useful work over total execution time. In other words, efficiency is $\frac{W}{T}$. To obtain good estimates for the parameters in the model, we examined the relevant literature [7], [33], [34] and used a projection from the values obtained in Section 6. Table 5 summarizes the baseline values we used to obtain the projections in this section. The value for M (MTBF of the system) depends linearly on the number of sockets. We use a MTBF per socket (M_S) equal to 10 years. Then, we assume the time between failures follows an exponential distribution. Additionally, we assume failures are independent. Thus, the total MTBF of the system is an exponentially distributed random variable. The value for M can be computed by dividing M_S among the total number of sockets in the system.

Equation 2 can be used to model several kinds of checkpoint/restart protocols. For instance, a higher value of δ will represent a shared file system checkpoint scheme, whereas a smaller value of δ will stand for a double in-memory checkpoint mechanism. We estimate $\delta = 2$ minutes will be a feasible value for double in-memory checkpoint. However, if the checkpoints were to be stored in a network file system (NFS), then that value would increase significantly. To model checkpoint/restart on NFS we use a value of $\delta = 20$ minutes.

Parameter	W	M_S	δ	R	π	ρ
Value	24h	10 years	120s	30s	0.7	0.4
Parameter	μ	P	σ	λ	κ	
Value	1.05	8	8	$\frac{P+1}{P}$	$\frac{\delta}{P}$	

TABLE 5
Baseline values of parameters in the model.

Figure 12(a) presents the value of efficiency obtained at different system sizes. We scale the system to 256K sockets (the expected number of sockets at exascale is at least 200K). The higher the socket count, the higher the failure rate. Then, it is natural to see all curves dropping as the socket count increases. The traditional NFS-based checkpoint/restart will not reach exascale. With an efficiency of 0 at exascale, NFS-based checkpoint restart will not make progress, as all the time will be spent rolling back and redoing the work lost in failures. Using in-memory checkpoint/restart will improve efficiency and reach exascale with 58% efficiency. Having either proactive evacuation or parallel recovery will fetch an additional 7% increase in efficiency, for a combined improvement of 14%.

The benefit of dodging failures with proactive evacuation comes from the fact that it is much faster to migrate objects from a failing PE than to recover from that failure. In the case of parallel recovery, failures are recovered faster. In either case, checkpoint period is longer than in the checkpoint/restart case. Figure 12(b) presents the checkpoint period (τ in the model) for each technique. Even when the benefit in efficiency of both proactive evacuation and parallel recovery is about the same, the size of the checkpoint period is longer in parallel recovery.

The different parameters of the model have different impacts on efficiency. A complete sensitivity analysis is out of the scope of this paper [27], but it is insightful to understand how the values of precision and recall affect efficiency. The heatmap in Figure 13 shows that increasing both recall and precision improve efficiency. The model predicts that recall has a higher impact on efficiency (but it is harder to improve

in real life). From a starting point of $(0.5, 0.5)$, an increase of precision to $(0.9, 0.5)$ only increases efficiency by 1%, whereas the same increase in recall to $(0.5, 0.9)$ gains 14% in efficiency. If precision is really small (lower than 0.1), there is a negligible benefit in increasing recall. Therefore, precision must have an acceptable value (greater than 0.3).

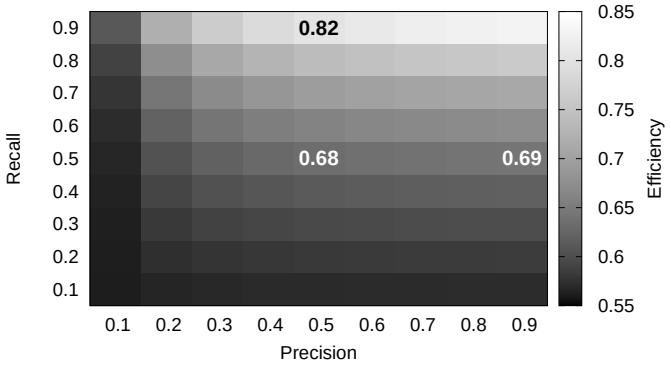


Fig. 13. Effect of precision and recall on efficiency.

The level of available parallelism also has an impact on efficiency. Table 6 shows how different values of P in the model affect efficiency. Clearly, more parallelism is better, but there are diminishing returns as P increases beyond 32.

P	2	4	8	16	32	64
Efficiency	0.55	0.65	0.73	0.78	0.81	0.83

TABLE 6
Effect of parallelism in recovery.

7.6 Model Validation

The performance model was validated against the fault tolerance strategies presented in this paper. The experiments used 256 cores on Lonestar and emulated a run ($W = 3600$ s, $\delta = 10$ s) with exponentially distributed random failures ($M = 120$ s). Table 7 shows the close match between model and experiment in the total execution time.

	Estimate (s)	**Run (s)**	**Error(%)**
Proactive F.T. (§3)	3639	3647	0.2
Checkpoint/R. (§4)	5792	6005	3.68
M. Logging (§5)	4694	4843	3.18

TABLE 7
Model estimates and experimental results.

8 Related Work

CoCheck [35] is a tool that explored the parallel between task checkpointing and task migration in the MPI model. It provided disk checkpoint and dynamic migration of MPI ranks. As opposed to evacuation in the migratable-objects model, task migration in CoCheck was synchronous with the execution of the system, i.e., before an MPI rank could be migrated, CoCheck had to make sure all communicating ranks would hold back messages until the rank was at its new location. Similar tools have recently used process-level live migration in MPI applications [36], [37], [38]. Those tools combine health monitoring of nodes with live migration to provide proactive fault tolerance. The migration of an MPI rank is called *live* because it occurs concurrently with the execution of the application. Therefore, the set of dirty pages of the migrating rank have to be sent before the migration process is complete. The system enforces a consistent state [13] in which all in-flight messages are stored and all processes freeze until the migrating rank completes the migration.

A comparative study about checkpoint versus migration [39] introduced an analytical model to contrast the two. That study predicted that migration is more beneficial in the short term and will eventually tie with checkpoint as larger systems become available. However, the model assumes a perfect failure predictor. In this paper, we removed that restriction by having different values of precision and recall in our performance model of Section 7. A different model that incorporated both proactive migration and checkpoint is FT-Pro [40]. It uses a stochastic model to adaptively schedule checkpoints and migrate processors when impending failures are detected.

Replication may be a viable alternative to tolerate failures at exascale [41]. If every rank in an MPI application gets replicated, the system recovers very fast. Should a node go down, the replicas of the lost ranks may replace them and execution continues with almost no cost in restart and recovery. Each rank and its replica must be synchronized for this approach to be correct. That continuous synchronization mechanism has an impact on the performance of the application. Moreover, replication will always have an efficiency cost higher than 50% of the system. In the minimal replication case (duplication), at least half of the resources are dedicated to run redundant ranks. Replication could only be effective if failures are very frequent and the synchronization cost can be kept low. Replication can be used in the migratable-objects model. The smart runtime system could even achieve partial replication and decide which objects get replicated. The replication ratio could be adjusted as the execution develops.

Hierarchical schemes in checkpoint/restart provide different levels of checkpoints during the execution of the application. The SCR library [18] implements three types of checkpoints. Each level has different reliability features and different costs. An associated stochastic model computes the optimum checkpoint interval for each level. Similarly, the FTI library [42] provides a multilevel checkpoint scheme. FTI uses Reed-Solomon encoding to tolerate failures that may include multiple nodes. It also uses dedicated threads to perform the checkpoint and different storage devices to store the checkpoints. Hierarchical schemes are completely compatible with migratable objects.

Reducing the performance overhead of message logging has been tackled by exploring deterministic communication patterns in applications [43]. For instance, a *send deterministic* application will always send the same sequence of messages regardless of the order of reception of previous non-causally related messages. This property has been used to develop faster message-logging protocols. These protocols can be used in the migratable-objects model with parallel recovery.

9 Conclusions and Recommendations

The major contributions of the migratable-objects model to enhancing fault tolerance techniques are:

- Having migratable objects makes a *proactive* approach work smoothly during the execution of the application. Evacuating a PE is no different than migrating all the objects living on it. All the necessary features to provide such ability are already contemplated in the model.
- Writing checkpoint methods for *reactive* protocols is easier with the migratable-objects model. Checkpointing an object can be seen as a migration of the object to storage. Moreover, the interface exported by the runtime system and the expertise of the programmer can significantly reduce the size of the checkpoint.
- Since the migratable-objects model does not directly depend on the number of physical PEs used for computation, the system can adapt to the loss of a PE and continue execution on a shrunk system.
- To accelerate recovery with message logging, objects in the failed PE can be migrated to other locations and be recovered in parallel. This distribution of objects organically attaches to the computational model.

In spite of its breadth, the collection of fault tolerance techniques presented in this paper is not meant to be exhaustive. There may still be many other fault tolerance techniques that can be enhanced by the migratable-objects model. Fundamentally, this model provides a flexible mechanism to dynamically shift computations from one part of the system to another, enabling the implementation of a large variety of methods to handle faults during execution on a large system.

The future of HPC will bring larger and faster machines at the cost of higher failure rates. We make the following recommendations to provide resilience at extreme scale:

- Automatic restart is imperative. An important fraction of the total turnaround time of a job is the wait time in a queue of the scheduler. Having the system detecting and restarting the job as nodes crash will save precious time that is spent in that queue. We believe it is essential to make a coordinated effort in the HPC community to bring about an interface to run through failures without relaunching (especially manually) the job after a crash.
- A smart implementation of the checkpoint functionality is fundamental in bringing down the cost of checkpointing and scaling rollback-recovery mechanisms further. Migratable objects are a good means to provide such functionality. Our thinking is that fast checkpoint mechanisms must be made available to HPC applications.
- We envision future systems using failure predictors that raise few false alarms and capture most of the failures. Fast migration mechanisms, such as migratable-objects, empower accurate failure prediction.
- The ability to shorten recovery time is indispensable for the success of rollback-recovery mechanisms. Parallel recovery uses an over-decomposed application to get high degrees of parallelism during recovery. We recommend the exploration of over-decomposition to improve the efficiency of a system in the future.

Acknowledgments

This research was supported in part by the US Department of Energy under grant DOE DE-SC0001845 and by a machine allocation on XSEDE under award ASC050039N.

References

[1] "Top 500 supercomputer sites," http://top500.org, 2012.

[2] 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.

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

[4] L. Kalé and S. Krishnan, "Charm++ : A portable concurrent object oriented system based on C++," in *Proceedings of the Conference on Object Oriented Programmi ng Systems, Languages and Applications*, September 1993.

[5] C. Huang, O. Lawlor, and L. V. Kalé, "Adaptive MPI," in *Proceedings of the 16th International Workshop on Languages and Compilers for Parallel Computing (LCPC 2003), LNCS 2958*, College Station, Texas, October 2003, pp. 306–322.

[6] S. Fu and C.-Z. Xu, "Exploring event correlation for failure prediction in coalitions of clusters," in *Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, ser. SC '07. New York, NY, USA: ACM, 2007, pp. 41:1–41:12.

[7] A. Gainaru, F. Cappello, M. Snir, and W. Kramer, "Fault prediction under the microscope: A closer look into hpc systems," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '12. Los Alamitos, CA, USA: IEEE Computer Society Press, 2012, pp. 77:1–77:11.

[8] Z. Lan, J. Gu, Z. Zheng, R. Thakur, and S. Coghlan, "A study of dynamic meta-learning for failure prediction in large-scale systems," *J. Parallel Distrib. Comput.*, vol. 70, no. 6, pp. 630–643, Jun. 2010.

[9] O. S. Lawlor and L. V. Kalé, "Supporting dynamic parallel object arrays," *Concurrency and Computation: Practice and Experience*, vol. 15, pp. 371–393, 2003.

[10] Sayantan Chakravorty, Celso Mendes and L. V. Kale, "Proactive fault tolerance in large systems," in *HPCRI Workshop in conjunction with HPCA 2005*, 2005.

[11] S. Chakravorty, C. L. Mendes, and L. V. Kalé, "Proactive fault tolerance in mpi applications via task migration." in *HiPC*, ser. Lecture Notes in Computer Science, vol. 4297. Springer, 2006, pp. 485–496.

[12] G. Antoniu, L. Bouge, and R. Namyst, "An efficient and transparent thread migration scheme in the PM^2 runtime system," in *Proc. 3rd Workshop on Runtime Systems for Parallel Programming (RTSPP) San Juan, Puerto Rico. Lecture Notes in Computer Science 1586*. Springer-Verlag, April 1999, pp. 496–510.

[13] K. M. Chandy and L. Lamport, "Distributed snapshots : Determining global states of distributed systems," *ACM Transactions on Computer Systems*, Feb. 1985.

[14] D. Buntinas, C. Coti, T. Hérault, P. Lemarinier, L. Pilard, A. Rezmerita, E. Rodriguez, and F. Cappello, "Blocking vs. non-blocking coordinated checkpointing for large-scale fault tolerant mpi protocols," *Future Generation Comp. Syst.*, vol. 24, no. 1, pp. 73–84, 2008.

[15] X. Ni, E. Meneses, and L. V. Kalé, "Hiding checkpoint overhead in HPC applications with a semi-blocking algorithm," in *IEEE Cluster 12*, Beijing, China, September 2012.

[16] M. Schulz, "Checkpointing," in *Encyclopedia of Parallel Computing*, 2011, pp. 264–273.

[17] P. H. Hargrove and J. C. Duell, "Berkeley lab checkpoint/restart (BLCR) for linux clusters," in *SciDAC*, 2006.

[18] 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.

[19] 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 Cluster*, San Diego, CA, September 2004, pp. 93–103.

[20] G. Zheng, X. Ni, and L. V. Kale, "A Scalable Double In-memory Checkpoint and Restart Scheme towards Exascale," in *Proceedings of the 2nd Workshop on Fault-Tolerance for HPC at Extreme Scale (FTXS)*, Boston, USA, June 2012.

[21] J. S. Plank, K. Li, and M. A. Puening, "Diskless checkpointing," *IEEE Trans. Parallel Distrib. Syst.*, vol. 9, no. 10, pp. 972–986, Oct. 1998.

[22] E. Meneses, X. Ni, and L. V. Kale, "A Message-Logging Protocol for Multicore Systems," in *Proceedings of the 2nd Workshop on Fault-Tolerance for HPC at Extreme Scale (FTXS)*, Boston, USA, June 2012.

[23] E. Meneses, O. Sarood, and L. V. Kale, "Assessing Energy Efficiency of Fault Tolerance Protocols for HPC Systems," in *Proceedings of the IEEE 24th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, New York, USA, October 2012.

[24] R. Strom and S. Yemini, "Optimistic recovery in distributed systems," *ACM Trans. Comput. Syst.*, vol. 3, no. 3, pp. 204–226, 1985.

[25] L. Alvisi and K. Marzullo, "Message logging: pessimistic, optimistic, and causal," *International Conference on Distributed Computing Systems*, pp. 229–236, 1995.

[26] 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.

[27] E. Meneses, "Scalable message-logging techniques for effective fault tolerance in HPC applications," Ph.D. dissertation, Dept. of Computer Science, University of Illinois, 2013.

[28] 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.

[29] J. C. Phillips, G. Zheng, S. Kumar, and L. V. Kalé, "NAMD: Biomolecular simulation on thousands of processors," in *Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, Baltimore, MD, September 2002, pp. 1–18.

[30] D. Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, S. Fineberg, P. Frederickson, T. Lasinski, R. Schreiber, H. Simon, V. Venkatakrishnan, and S. Weeratunga, "The NAS parallel benchmarks," NASA Ames Research Center, Tech. Rep. RNR-04-077, 1994.

[31] J. W. Young, "A first order approximation to the optimal checkpoint interval," *Commun. ACM*, vol. 17, no. 9, pp. 530–531, 1974.

[32] 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.

[33] 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.

[34] Z. Lan, J. Gu, Z. Zheng, R. Thakur, and S. Coghlan, "A study of dynamic meta-learning for failure prediction in large-scale systems," *J. Parallel Distrib. Comput.*, vol. 70, no. 6, pp. 630–643, Jun. 2010.

[35] G. Stellner, "CoCheck: Checkpointing and Process Migration for MPI," in *Proceedings of the 10th International Parallel Processing Symposium (IPPS '96)*, Honolulu, Hawaii, 1996.

[36] C. Wang, F. Mueller, C. Engelmann, and S. L. Scott, "Proactive process-level live migration in HPC environments," in *SC*, 2008, p. 43.

[37] X. Ouyang, S. Marcarelli, R. Rajachandrasekar, and D. K. Panda, "Rdma-based job migration framework for mpi over infiniband," in *Proceedings of the 2010 IEEE International Conference on Cluster Computing*, ser. CLUSTER '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 116–125. [Online]. Available: http://dx.doi.org/10.1109/CLUSTER.2010.20

[38] K. Z. Ibrahim, S. Hofmeyr, C. Iancu, and E. Roman, "Optimized pre-copy live migration for memory intensive applications," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '11. New York, NY, USA: ACM, 2011, pp. 40:1–40:11. [Online]. Available: http://doi.acm.org/10.1145/2063384.2063437

[39] F. Cappello, H. Casanova, and Y. Robert, "Preventive migration vs. preventive checkpointing for extreme scale supercomputers," *Parallel Processing Letters*, vol. 21, no. 2, pp. 111–132, 2011.

[40] Z. Lan and Y. Li, "Adaptive fault management of parallel applications for high-performance computing," *IEEE Trans. Computers*, vol. 57, no. 12, pp. 1647–1660, 2008.

[41] K. Ferreira, J. Stearley, J. H. Laros, III, R. Oldfield, K. Pedretti, R. Brightwell, R. Riesen, P. G. Bridges, and D. Arnold, "Evaluating the viability of process replication reliability for exascale systems," in *Supercomputing*. New York, NY, USA: ACM, 2011, pp. 44:1–44:12.

[42] L. Bautista-Gomez, D. Komatitsch, N. Maruyama, S. Tsuboi, F. Cappello, and S. Matsuoka, "FTI: High performance fault tolerance interface for hybrid systems," in *Supercomputing*, Nov. 2011, pp. 1 –12.

[43] F. Cappello, A. Guermouche, and M. Snir, "On communication determinism in parallel HPC applications," in *ICCCN*, 2010, pp. 1–8.

Esteban Meneses is a research assistant professor at the University of Pittsburgh. His research interests include fault tolerance and load balancing for large-scale systems. He works on energy-efficient low-overhead techniques for fault tolerance based on the principle of local recovery. He has developed message-logging protocols that exploit application characteristics to reduce the total memory footprint of the message log. He holds a Ph.D. in Computer Science from the University of Illinois at Urbana-Champaign. He received a bachelor's and master's degree in Computer Science from the Costa Rica Institute of Technology. Esteban is a recipient of a Fulbright-LASPAU scholarship.

Xiang Ni is a doctoral candidate working with Prof. Laxmikant V. Kalé in the Parallel Programming Laboratory of the University of Illinois at Urbana-Champaign. She has a broad interest in parallel and distributed computing with a focus on fault tolerance. She works on implementing cost-effective proactive and reactive strategies, using log analysis for accurate prediction of faults and enabling advanced actions for dealing with faults. She has a master's degree in Computer Science from the University of Illinois at Urbana-Champaign. She received her bachelor's degree in Computer Science at Beihang University in Beijing, China.

Gengbin Zheng is currently a staff member of the National Center for Supercomputing Applications at Illinois. His research interests include parallel runtime support with dynamic load balancing for highly adaptive parallel applications, simulation-based performance prediction for large parallel machines and fault tolerance. He received his Ph.D. degree in the Computer Science Department at the University of Illinois at Urbana-Champaign in 2005. He received the BS (1995) and MS (1998) degrees in Computer Science from the Peking University, Beijing, China. A paper co-authored by him on scaling the molecular dynamics program NAMD was one of the winners of the Gordon Bell award in SC2002.

Celso L. Mendes is currently a senior staff member of the National Center for Supercomputing Applications at Illinois, where he has been participating in the Blue Waters deployment project. He was a senior technologist at the National Institute for Space Research (INPE), in Brazil. As a member of the Illinois' Pablo group, led by Dan Reed, he worked on performance analysis tools and techniques for parallel systems. He worked at the Parallel Computing Laboratory, also at Illinois, on applications of adaptive runtime systems for large parallel machines. He received a Ph.D. in Computer Science from the University of Illinois at Urbana-Champaign and received both an Engineer and a Master of Engineering degrees from the Aeronautics Technology Institute (ITA), in Brazil.

Laxmikant V. Kalé is a Professor at the University of Illinois at Urbana-Champaign. He has been working on various aspects of parallel computing, with a focus on enhancing performance and productivity via adaptive runtime systems, and with the belief that only interdisciplinary research involving multiple CSE and other applications can bring back well-honed abstractions into computer science that will have a long-term impact on the state-of-art. His collaborations include the widely used Gordon-Bell award winning biomolecular simulation program NAMD, and other collaborations on computational cosmology, quantum chemistry, rocket simulation, and other unstructured mesh applications. He received a Ph. D. in computer science from the State University of New York, Stony Brook. He received the B. Tech. degree in electronics engineering from Benares Hindu University, Varanasi, India, and the M. E. degree in computer science from the Indian Institute of Science in Bangalore, India. He is an IEEE Fellow and a recipient of the Sidney Fernbach Award.