# Assessing Energy Efficiency of Fault Tolerance Protocols for HPC Systems

Esteban Meneses, Osman Sarood and Laxmikant V. Kalé
Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, Illinois, 61802
{emenese2,sarood1,kale}@illinois.edu

***Abstract*—An exascale machine is expected to be delivered in the time frame 2018-2020. Such a machine will be able to tackle some of the hardest computational problems and to extend our understanding of Nature and the universe. However, to make that a reality, the HPC community has to solve a few important challenges. Resilience will become a prominent problem because an exascale machine will experience frequent failures due to the large amount of components it will encompass. Some form of fault tolerance has to be incorporated in the system to maintain the progress rate of applications as high as possible. In parallel, the system will have to be more careful about power management. There are two dimensions of power. First, in a power-limited environment, all the layers of the system have to adhere to that limitation (including the fault tolerance layer). Second, power will be relevant due to energy consumption: an exascale installation will have to pay a large energy bill. It is fundamental to increase our understanding of the energy profile of different fault tolerance schemes. This paper presents an evaluation of three different fault tolerance approaches: checkpoint/restart, message-logging and parallel recovery. Using programs from different programming models, we show parallel recovery is the most energy-efficient solution for an execution with failures. At the same time, parallel recovery is able to finish the execution faster than the other approaches. We explore the behavior of these approaches at extreme scales using an analytical model. At large scale, parallel recovery is predicted to reduce the total execution time of an application by $17\%$ and reduce the energy consumption by $13\%$ when compared to checkpoint/restart.**

## I. Introduction

As high performance computing systems grow in size and complexity, a set of new challenges emerge to keep the same level of productivity as the previous generation of machines. With exascale on the horizon, resilience and energy consumption are two of the major problems that have to be addressed [1], [2]. Resilience will become a fundamental concern due to the extremely large number of components that will form an exascale machine. Such a supercomputer will have millions of processors along with memory modules, routers and disks. With these many pieces, an exascale machine is expected to experience a failure every few minutes [2]. Power management will be the driver in the design of architectures, systems and applications for exascale. In a power-limited environment, it will be crucial to constrain all the layers of the system to meet the power budget. Furthermore, reducing power consumption by one megawatt may save around $1M/year even in a relatively inexpensive energy contract [2].

Incorporating some sort of fault tolerance technique in the system will be unavoidable. But, whatever fault tolerance strategy is used, it should be an energy efficient solution. An understanding of how costly each of these strategies is, in terms of energy consumption, is fundamental to drive the resilience research towards exascale. This paper compares three standard checkpoint-based fault tolerance methods according to their energy consumption. The first method is the traditional checkpoint/restart based on local storage that has been implemented in several libraries [3], [4]. The second strategy is a particular version of message-logging [5] that requires messages to be stored, but avoids a global rollback in case of a failure. Finally, the third approach is called parallel recovery [6] and requires the system to allow tasks to migrate after a failure. This ability potentially reduces recovery time to a small fraction of re-execution time from checkpoint.

The contributions of this paper are listed below:

- A comparative evaluation of the energy efficiency of three fault tolerance strategies. We provide results using programs in two different programming models (§IV).
- An understanding of the performance of the different strategies during recovery when failures are injected in the system (§IV).
- A model to predict how energy efficient the three different methods will be for a variety of scenarios and particularly at exascale (§V).

## II. Background

This section presents an overview of the three fault tolerant protocols that we evaluate in the rest of the paper. We assume a parallel program is composed of a collection of tasks and each task performs part of the computation and holds a piece of the data. The only way to share information among the tasks is by message passing. The parallel program runs on a machine comprising several nodes. A node may run one or more tasks depending on the decisions made by a runtime system that assigns tasks to nodes. The nodes of the system may fail according to the *fail-stop* model, where the node becomes non-functional and never comes back up. Other nodes may replace the failed one. If one node fails, the tasks running on it are lost and have to be recreated from a checkpoint. The fault tolerance techniques described below tolerate failures of one node. Figure 1 presents the composition of the protocols.

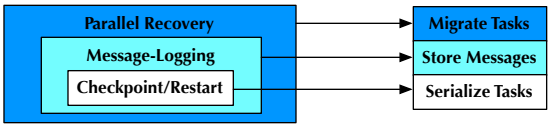



Fig. 1. Schematic view of the fault tolerance protocols.

### A. Checkpoint/Restart

The traditional approach to tolerate a failure is to have the system checkpointing its state periodically. If one node crashes, the whole system rolls back to the most recent checkpoint and restarts from there. This scheme is called *checkpoint/restart* and it is by far the most popular method in HPC to provide fault tolerance. Its underlying principle is simple, which makes it straightforward to implement. Nevertheless, it has adopted many variants, depending on the design decisions of the protocol.

One decision is to determine whether to checkpoint the whole system or just the tasks of the parallel program. In *system-level* checkpoint, the whole machine state is saved to disk. A popular library that performs this operation is BLCR [7]. On the other hand, *application-level* checkpoint assumes that the state of the tasks is enough to resume the execution of the program in case of a failure. The SCR library [3] uses this approach. One advantage of application-level checkpoint is to dramatically reduce the amount of memory to be checkpointed. However, it requires the program to identify what variables in the program should be stored and at which points of the execution. We believe this is not a major burden for the programmer of HPC applications. We assume application-level checkpoint in this paper.

Another decision in designing a checkpoint/restart protocol is whether or not to coordinate the tasks at checkpoint. An *uncoordinated* protocol allows the tasks to checkpoint independently. However, this may complicate the recovery mechanism. On the contrary, a *coordinated* protocol ensures that the tasks checkpoint a consistent global state of the system. We believe HPC applications usually contain global synchronization points where checkpoint calls can be made. We assume coordinated checkpoint in the rest of the paper.

Different devices can be used to store the checkpoints. The traditional NFS-based disk checkpoint has major drawbacks due to congestion of the file system. An approach that has been adopted is to use local storage, for instance local disks, memory or SSDs. The approach we used in this document is called double in-memory or double in-disk checkpoint [4]. As the name suggests, each node stores two copies of its checkpoint: one in its own local storage and one in the local storage of a *buddy* node. When restarting from a failure, all nodes can access its most recent checkpoint locally, with the exception of the crashed node. The replacement node can obtain a copy from its buddy node.

One fundamental question is how often to checkpoint. The answer depends on several variables of the system and the application. A good approximation for the optimum checkpoint period has been suggested elsewhere [8], [9]:

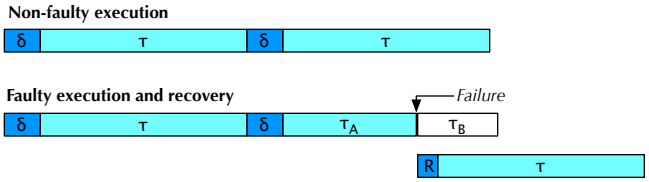



Fig. 2. Execution in non-faulty and faulty case.

$$\tau = \sqrt{2\delta(M+R)}$$

where $\tau$ is the optimum checkpoint period, $\delta$ is the time to checkpoint, $M$ is the mean-time-to-interrupt and $R$ is the restart time. Figure 2 depicts these variables and shows the two possible scenarios for execution (with and without failure). Note that $\tau_A + \tau_B = \tau$.

### B. Message Logging

An important drawback of checkpoint/restart is that it requires all the nodes to roll back in case of a failure. Imagine a system with a million nodes and having to roll back a million minus one of them just because one fails. It would be much more efficient if only the node crashing had to be rolled back. While the failed node catches up, the rest of the nodes may continue execution or may wait idle until the recovery of the crashed node has finished. This last alternative is very appealing from the energy point of view, since an idle node runs at a much lower power and consumes less energy. In figure 2 only the failed node has to re-execute during $\tau_A$.

Message-logging is an extension of checkpoint/restart that provides the ability to rollback only the node that fails. It achieves this goal by storing the messages at the sender's memory and resending those messages in case of failure. Besides logging the messages, a message-logging protocol has to ensure the recovering node reaches a consistent state with the rest of the system. In order to do that, all non-deterministic decisions have to be stored. Message reception is, in general, non-deterministic. Each non-deterministic event generates a *determinant* and those have to be stored in order to guarantee a correct recovery. Different flavors of message logging exist according to the way they handle the determinants [10].

In this paper we will use a particular message-logging protocol called *simple causal message-logging* [5]. The idea of this protocol is to store the determinants in the causal path of the messages that generate them. For instance, once a message is received at a node, the determinant generated as a consequence of that reception is piggybacked in the next outgoing message and stored at the recipient of that message. If node $X$ crashes, the nodes containing determinants from $X$ will send them to $X$ to guarantee a consistent recovery.

### C. Parallel Recovery

Message-logging only rolls back the node that fails and this presents a fundamental opportunity to speed up recovery. If the system allows *migratable tasks*, then some of the tasks that live on the recovering node may be migrated to other nodes to be

recovered in parallel. This scheme is an extension of message-logging and is called *parallel recovery* [6]. It has the potential to reduce recovery time to a fraction of the normal rework time. This speedup can potentially tolerate high failure rates, including some where the mean-time-to-interrupt is shorter than the checkpoint period. In figure 2, this means $\tau_A$ can be significantly reduced.

Since we use application-level checkpoint, tasks can only be checkpointed at synchronization points. The same rule applies for migration. The tasks that migrate as part of parallel recovery will be sent back to their original node once the application reaches the next synchronization point. This causes a transient load imbalance from the point where recovery finishes to the next checkpoint. In figure 2, it implies that $\tau_B$ will be executed under a load imbalance.

Figure 1 presents a view of the protocols. This hierarchical structure is not true for all the different flavors of checkpoint/restart and message-logging protocols, but it holds for the particular versions we evaluate in this paper. Checkpoint/restart is coordinated, application-level and uses local storage. It only requires the tasks to be serializable in some way. Our message-logging protocol extends checkpoint/restart by storing messages and determinants (but only saves one copy of the checkpoint in local storage; it does not need the second copy). Finally, parallel recovery is an extension of message-logging that requires tasks to be migratable.

## III. Experimental Setup

The three fault tolerance methods described in the previous section were implemented in the CHARM++ runtime system [11]. CHARM++ is a parallel programming language based on asynchronous method invocation between migratable objects. A parallel program in CHARM++ contains a collection of objects. The number of objects is typically independent of the number of physical nodes the program will use to run. The number of objects running per *logical* node is called its *virtualization ratio*. The runtime system assigns objects to nodes and may migrate objects during load balancing. The only communication mechanism between two objects in CHARM++ is through message passing. However, CHARM++ is based on active messages and this means there is no synchronization between the objects when sending a message. An extension to CHARM++ , called AMPI, permits any MPI program to run on the CHARM++ runtime system. In AMPI, each MPI rank is handled as a CHARM++ object and, as such, it can be migrated from one node to another.

We tested the three fault tolerance protocols on a cluster of 32 nodes (128 cores). Each node has a single socket with a four-core Intel Xeon X3430 processor chip running CentOS 5.7. The cluster nodes are connected by a 48-port gigabit ethernet switch. We use a Liebert power distribution unit installed on the rack containing the cluster to measure the machine power after at 1-second intervals on a per-node basis. We gather these readings for each experiment and integrate them over the execution time to come up with the total machine energy consumption. The results reported in Section IV are thus compiled after doing experiments on real hardware and are not from simulations.

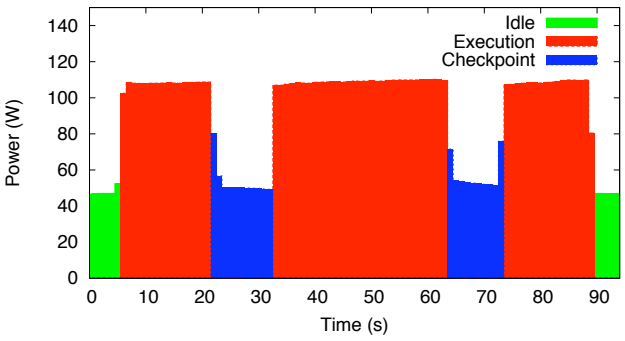



Fig. 3. Power levels during a run with checkpoint/restart.

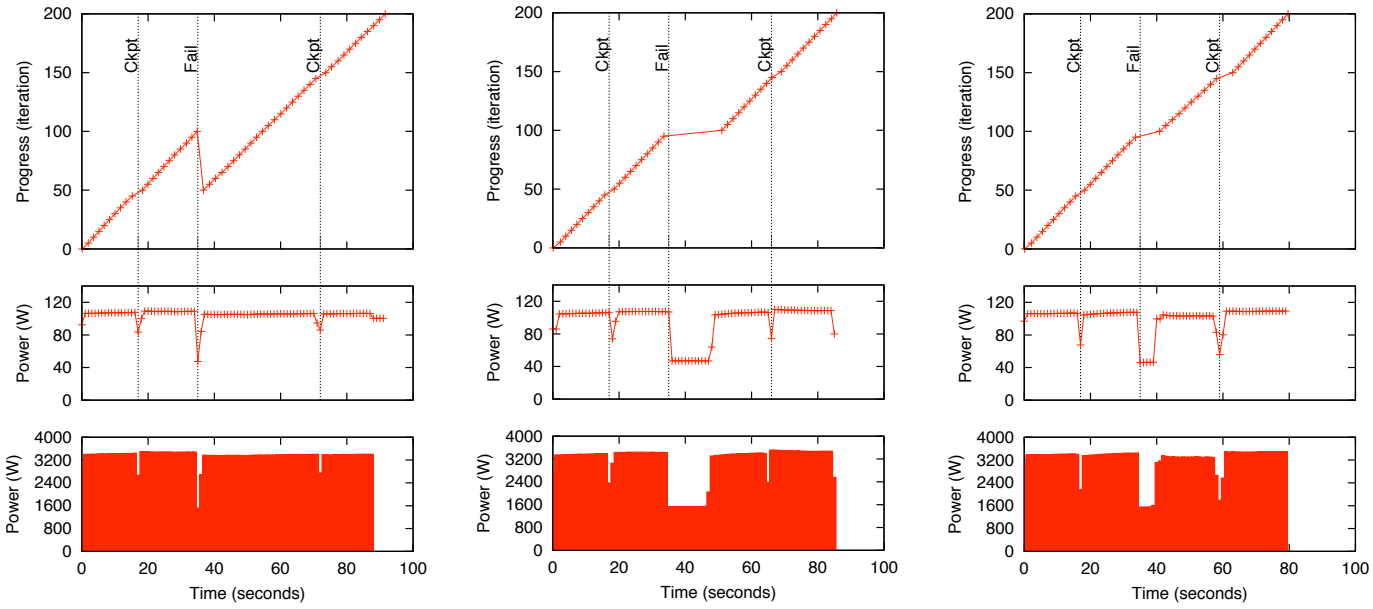
To evaluate the different fault tolerance approaches, we chose 3 applications from different programming models. The first code is a CHARM++ program called *Jacobi3D* and consists of a 7-point stencil that performs a successive over-relaxation on a three-dimensional space. This program is computation bound, but has a non-trivial communication pattern. In addition to it, we used BT and SP, from the NAS parallel benchmark suite that belong to the MPI programming model. Both these programs solve a system of nonlinear partial differential equations using different methods: BT uses a block-tridiagonal approach, whereas SP is a scalar pentadiagonal algorithm. These two benchmarks were run using AMPI.

Although we were interested in measuring the overhead of the different approaches, being able to examine the faulty case as well was fundamental for us. In order to inject failures, we killed a process corresponding to a logical node (a physical core in our testbed). The CHARM++ runtime system creates one process per each logical node. In order to simulate the failure of that node, we executed the command `kill -9 PID`, where $PID$ was the process ID.

## IV. Results

We start this section by presenting the results of running *Jacobi3D* over a $1024^3$ space. The space was divided into $64^3$ blocks to have 4096 blocks in total. Each block is represented by an object in CHARM++ . Since we were using the whole cluster (i.e. 128 logical nodes) the virtualization ratio comes down to 32. We ran the benchmark for 200 iterations with checkpoints in iteration 50 and 150.

Figure 3 presents the power draw of one node for a run of *Jacobi3D* with checkpoint/restart storing the checkpoint in the local disk. The goal of this experiment is to show the different power levels that occur throughout the execution of a program. When the node is idle, the power draw is 47 W, on average. As soon as the execution starts, the power draw rises to 106 W but drops to 51 W as the application starts the checkpoint. We intend to highlight how the power drops to almost the base power during checkpoint. In fact, the average power during checkpoint is 51 W, a value just 9% higher than the base power. This means that, in terms of energy consumption, checkpointing is cheaper compared to compute.




(a) Checkpoint/Restart (b) Message-Logging (c) Parallel Recovery

Fig. 4. Progress rate, power draw of one node and power draw of the entire machine for the different fault tolerance schemes.

In other words, if energy consumption were to be optimized, instead of execution time, the optimum checkpoint period would be much smaller. The system is better off checkpointing more frequently (spending less energy consumption) than recomputing the lost work. Similar results were observed when checkpointing to memory. However, checkpointing to memory for the particular problem size of this experiment is faster than the precision of our power meter frequency. On average, it takes little less than one second. We used a memory checkpoint in the following experiments.

We used the same scenario as the last experiment to compare the three fault tolerance approaches from section II. Since the message-logging overhead (compared to checkpoint/restart) was very low, we decided to keep the same checkpoint frequency and the same failure time. We injected a failure right in the middle of the second interval at second 35, between the two checkpoints. This occurs roughly at iteration 100.

Figure 4 presents the results of running the same program with the three fault tolerance schemes. There are three diagrams in each case. The top graph depicts the number of iterations completed in the program versus elapsed time. We call this representation a *progress diagram* and it stresses our view of a good fault tolerant technique. In HPC, what matters for most users is speedup, the ability to solve a problem faster or, in other words, to increase the progress rate of the program. The same should apply for fault tolerance. A good fault tolerance technique in HPC is one that has the ability to keep the progress rate as high as possible, even in the presence of failures. The progress diagram for checkpoint/restart exhibits the basic behavior of this approach. After a crash, there is a global rollback to the previous checkpoint. In this case, the previous checkpoint occurs at iteration 50. The program resumes execution after a short break (around 2 seconds), recovers the lost work in 17.81 seconds and eventually finishes the total execution of 200 iterations. Note that for message-logging, there is no global rollback. Instead, only the failed node is rolled back and the rest of the system waits for it to catch up. The failed node reaches the 100*th* iteration in 14.3 seconds. This time is faster than checkpoint/restart. A possible reason for it is that during recovery, the node has little interference from other nodes. All messages it requires to recover are delivered right after the failure. Finally, parallel recovery uses 8 nodes to recover and manages to accelerate the recovery to reach the 100*th* iteration in 4.14 seconds. The recovery time is much smaller due to the parallelism with which the failed node is brought back to the current iteration. In the terminology of Figure 2, this is equivalent to speed up $\tau_A$. If we remove the migration time, parallel recovery manages to recover in 2.43 seconds. The speedup is 7.3, very close to 8, the degree of parallelism available at recovery. It is important to mention that for parallel recovery the rest of the interval before the next checkpoint occurs is executed with load imbalance. Initially, each node had 32 objects. After the failure, the crashed node distributed its objects among 8 nodes (4 additional objects per node helping during recovery). This means, overloaded nodes will have $\frac{1}{8}$ additional work to perform. This means, part $\tau_B$ in Figure 2 does not perform at the same speed as the normal case. In this case, the slowdown of this imbalance is 1.14, very close to the extra $\frac{1}{8}$ work added to certain nodes.

Parallel recovery has the ability to move past a failure and recover the lost work faster. We repeated this same experiment several times and averaged the execution time. Checkpoint/restart took 90 seconds, whereas message-logging

completed the same amount of work in 86 seconds (1.04 speedup). Parallel recovery was the fastest out of the three and only took 79 seconds to complete the same work (1.14 speedup).

The second row of plots in Figure 4 show the power of one node that does not fail during execution. It is clear that during this execution, all nodes were executing at max power when using checkpoint/restart. Conversely, nodes not involved in a failure were at base power for message-logging and parallel recovery which gets reflected in the bottom row of Figure 4. The last row of Figure 4 presents the total power consumed by the cluster during the execution. It includes the aggregated power of all nodes across time. Thus, the area below the curve represents energy consumption. In total, checkpoint/restart spent 299 kJ, message-logging 255 kJ (85% of checkpoint/restart) and parallel recovery 203 kJ (68% of checkpoint/restart).

To extend the results to other programs and into a different programming model, we ran two NPB benchmarks (BT and SP) with virtualization ratio 4. In both cases we used 100 logical nodes and 400 virtual ranks to solve the class C problem. Table I summarizes some of the features of these benchmarks. In particular, they show a lower max power draw compared to *Jacobi3D*. Between the two, SP has the lower max power. Interestingly, there is almost no difference between the max power of checkpoint/restart (C) and message-logging (M). They also have a higher overhead for message-logging than *Jacobi3D*.

| | **Jacobi3D** | **NPB-BT** | **NPB-SP** |
|---|---|---|---|
| **Language** | Charm++ | MPI | MPI |
| **Problem size** | $1024^3$ | class C | class C |
| **Number of cores** | 128 | 100 | 100 |
| **Virtualization ratio** | 32 | 4 | 4 |
| **Recovery parallelism** | 8 | 4 | 4 |
| **Message-logging overhead** | 1.0% | 3.6% | 4.1% |
| **Max power (C)** | 106 | 102 | 95 |
| **Max power (M)** | 106 | 102 | 96 |

TABLE I
PROGRAM FEATURES

To improve our understanding of the different fault tolerance approaches in these benchmarks, we carried out a set of experiments. In both cases (BT and SP), we ran the program with periodic checkpoint and inserted a failure in the middle of one checkpoint period. We carefully calibrated that point for each protocol. Then, we measure the total energy consumption for each protocol in the faulty interval. In other words, we measured the energy consumption in $R + \tau$ from Figure 2. The results are shown in Figure 5 and it shows the energy consumption of message-logging and parallel recovery relative to the one of checkpoint/restart. Again, parallel recovery manages to execute through the failure with the minimum amount of energy consumption. In this case, both benchmarks show similar results, with message-logging using around 70% of checkpoint/restart and parallel recovery using close to 63%.

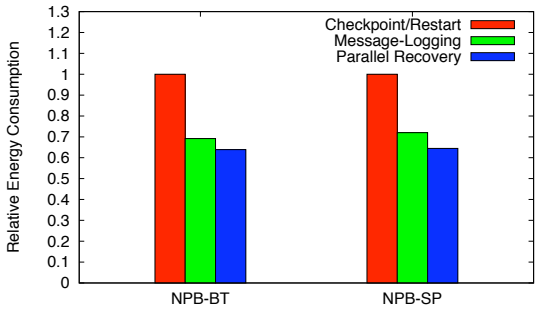



Fig. 5. Relative energy consumption during a faulty checkpoint period.

## V. ENERGY MODEL

The previous section showed the advantages of message-logging techniques to save energy consumption in faulty scenarios. Furthermore, having migratable tasks can fetch an additional benefit by means of parallel recovery. In this section, we generalize the fault tolerance framework and describe a model that incorporates the main costs of the different fault tolerance protocols described early in the paper.

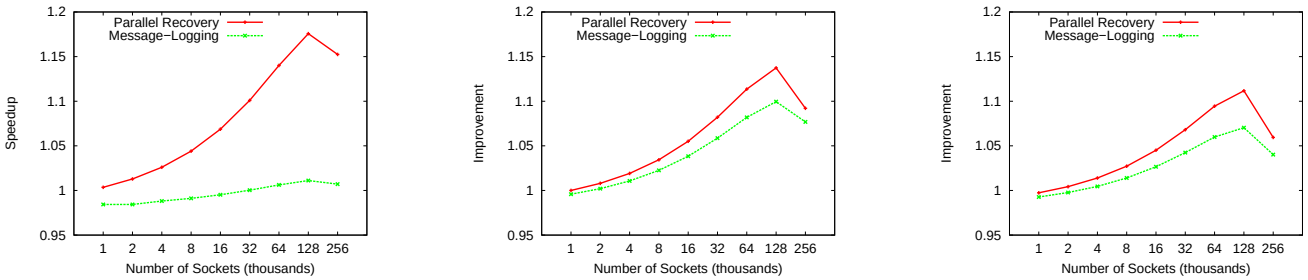
| **Parameter** | **Description** | **Value** |
|---|---|---|
| $V$ | Optimal virtualization ratio | $> 8$ |
| $W$ | Time to solution with $V$ | 25 h |
| $M$ | Mean-time-to-interrupt of the system | - |
| $S$ | Total number of sockets in the system | - |
| $\delta$ | Checkpoint time | 180 s |
| $\tau$ | Optimum checkpoint period | - |
| $R$ | Restart time | 30 s |
| $T$ | Total execution time | - |
| $E$ | Total energy consumption | - |
| $\mu$ | Message-logging slowdown | 1.02 |
| $P$ | Available parallelism during recovery | 8 |
| $\phi$ | Message-logging recovery speedup | 1.2 |
| $\sigma$ | Parallel recovery speedup | $P$ |
| $\lambda$ | Parallel recovery slowdown | $\frac{P+1}{P}$ |
| $H$ | Max power of each socket | 100 W |
| $L$ | Base power of each socket | 40 W |

TABLE II
PARAMETERS OF ENERGY MODEL

Table II presents the list of parameters for this model. The meaning of most of them should be clear to the reader. However, we will discuss a few of them. $V$ represents the number of tasks (MPI ranks or CHARM++ objects) running per logical node. $V$ determines the maximum level of parallelism during recovery, $P$. Since $P$ represents the number of logical nodes helping to recover the failed one, it must hold that $P \leq V$. We use $S$ as the system size in terms of number of sockets. This is mainly for convenience, because it is understood the reliability per socket will probably be constant across the next decade. A typical value for the MTTI (mean-time-to-interrupt) of a socket is 5 years [12]. We compute $M$ based on $S$ and the reliability per socket, $M = 5\ years/S$. For message-logging, $\mu$ is the slowdown in the execution of an application, whereas $\phi$ is the speedup during recovery. Finally, $\sigma$ is the speedup of parallel recovery for section $\tau_A$ in Figure

| Protocol | Execution Time | Energy |
|---|---|---|
| **Checkpoint/Restart** | $T = W + \left(\frac{W}{\tau} - 1\right)\delta + \frac{T}{M}\left(\frac{\tau+\delta}{2}\right) + \frac{T}{M}R$ | $E = WSH + \left(\frac{W}{\tau} - 1\right)\delta SL + \frac{T}{M}\left(\delta SL + \frac{\tau-\delta}{2}SH\right) + \frac{T}{M}RSL$ |
| **Message-Logging** | $T = W\mu + \left(\frac{W\mu}{\tau} - 1\right)\delta + \frac{T}{M}\left(\delta + \frac{\tau-\delta}{2\phi}\right) + \frac{T}{M}R$ | $E = W\mu SH + \left(\frac{W\mu}{\tau} - 1\right)\delta SL + \frac{T}{M}\left(\delta SL + \frac{\tau-\delta}{2\phi}(H + (S-1)L)\right) + \frac{T}{M}RSL$ |
| **Parallel Recovery** | $T = W\mu + \left(\frac{W\mu}{\tau} - 1\right)\delta + \frac{T}{M}\left(\delta + \frac{\tau-\delta}{2\sigma} + \frac{\tau+\delta}{2}(\lambda - 1)\right) + \frac{T}{M}R$ | $E = W\mu SH + \left(\frac{W\mu}{\tau} - 1\right)\delta SL + \frac{T}{M}\left(\delta SL + \frac{\tau-\delta}{2\sigma}(PH + (S-P)L) + \frac{\tau+\delta}{2}(\lambda - 1)SH\right) + \frac{T}{M}RSL$ |

TABLE III
EXECUTION TIME AND ENERGY FORMULAS




(a) Execution Time (b) Energy (using time-optimum $\tau$) (c) Energy (using energy-optimum $\tau$)

Fig. 6. Improvement in execution time and energy consumption.

2 and $\lambda$ is the overhead for section $\tau_B$.

Most of the parameters for this model have a determined value depending on the architecture and the application. However, in order to obtain $\tau$ we must first minimize the objective function. The objective function may be execution time or energy consumption, for instance. In principle, the users of HPC systems are interested in solving a problem faster. That is the reason to have execution time as the objective function [9]. Using this philosophy, we can define $T$ as follows [9]:

$$T = T_{Solve} + T_{Checkpoint} + T_{Recover} + T_{Restart}$$

where $T_{Solve}$ is the computation time required to solve a particular problem, $T_{Checkpoint}$ is the overhead of saving the checkpoint to local storage, $T_{Recover}$ is the total time required to recover the lost work after a failure happens, and $T_{Restart}$ the total time necessary to restart the program after a crash.

For simplicity, we will assume the failures follow an exponential distribution and occur in the middle of a checkpoint period. Although other distributions are better at modeling failures for real supercomputers, this assumption keeps the model simple to understand.

In a similar fashion, the equation for energy consumption is defined as:

$$E = E_{Solve} + E_{Checkpoint} + E_{Recover} + E_{Restart}$$

where each component of the formula above depends on the power draw and elapsed time for each part. Since energy consumption depends on execution time and execution time in turn depends on the checkpoint period $\tau$, the equation for energy consumption can have two possible solutions, depending on what value of $\tau$ is used. In one case, we may use the optimum $\tau$ to minimize time (called *time-optimum* $\tau$). In the other case, $\tau$ can be used as the optimum value to minimize energy consumption (called *energy-optimum* $\tau$). We analyze both these possibilities in the following discussion.

Table III presents the detailed formula of execution time and energy consumption for each of the fault tolerance protocols studied in this paper. To help in the clarity of such formulae, we have conveniently divided each formula into four components that match the four parts of both the execution time and energy consumption formulae above. We will only describe how we constructed the energy consumption function for parallel recovery. The other functions should be easy to interpret. The energy spent by parallel recovery to solve the problem, $E_{Solve}$ is $W\mu SH$. This is the case because all the system ($S$ sockets) will be executed at max power for $W$ units of time. Since message-logging has an overhead $\mu$, we must factor that in. For checkpoint, $E_{Checkpoint}$ is equal to the number of checkpoints ($\frac{W\mu}{\tau} - 1$) times the duration of each checkpoint, $\delta$, multiplied by the power during checkpoint that

we assume is $L$ (see Figure 3). The number of failures in the execution is given by $\frac{T}{M}$. Every failure occurs in the middle of a checkpoint period $(\tau+\delta)$. $E_{Recover}$ is equal to the number of failures times the energy consumed in recovering. Since the checkpoint has to be brought, we assume the whole system waits for $\delta$ units with power $L$. The rest of the half period of $\tau + \delta$ is accelerated by $\sigma$. During this time, $P$ sockets are working at max power $H$, while the rest execute at power $L$. Since parallel recovery adds a slowdown for the second part of the checkpoint period, we need to multiply that part by $\lambda$. The explanation for $E_{Restart}$ follows from the description above.

Using the results from our experimental section and based on the *Jacobi3D* case, we set proper values for the parameters of the model (listed in the third column of table II). We computed the execution time and energy consumption values for the different fault tolerance approaches with the formulae of table III. Figure 6 plots the relative improvement of message-logging and parallel recovery with respect to checkpoint/restart. This relative improvement is the speedup in case of execution time or the ratio between the total energy consumed in checkpoint/restart over the energy consumed of the other approach. In Figure 6(a) we see the comparison in the total execution time for a number of sockets up to 256.000. An exascale machine is expected to have at least 200.000 sockets [1]. Parallel recovery manages to reach up to 1.17 speedup with respect to checkpoint/restart, while message-logging marginally outperforms checkpoint/restart.

Figures 6(b) and 6(c) present the energy consumption dimension in both cases of $\tau$. When execution time has the priority, parallel recovery reaches a maximum of 1.13 improvement versus 1.09 of message logging. On the other hand, when energy consumption is the function to optimize, the two approaches decrease in their benefit. The reason is that in a energy-optimum $\tau$, the system will checkpoint with more frequency than in a time-optimum scenario. If the checkpoint period decreases, the advantage of message-logging and parallel recovery over checkpoint/restart decreases too. The former two approaches rely on a relatively long recovery time to decrease the energy consumption.

As any model, the one presented in this section has some sensibility to variations in certain parameters. In particular, we were interested in looking at different values of the effective parallelism during recovery $P$, that dictates the speedup for parallel recovery in segment $\tau_A$ and the slowdown $\lambda$ for segment $\tau_B$. Figure 7 shows the results of four different values of $\sigma$, ranging from 8 to 20. The higher $P$ is the more improvement in energy consumption over checkpoint/restart. Furthermore, a high value of $P$ permits scaling in the energy consumption savings to larger systems.

## VI. Analysis

There are a few important things to discuss given the experimental and the analytical model results.

First of all, we must address the power management concerns in a resilient runtime system. We demonstrated, empirically (§IV, table I), that there is no evidence to assume that

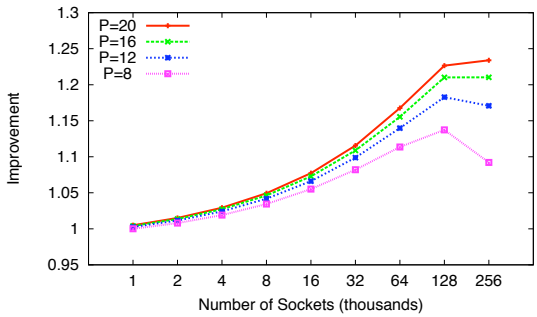



Fig. 7. Effect of changing parameter $P$ in the model.

any of the fault tolerant approaches will increase the maximum power of any of the applications studied. Even when message-logging stores messages in memory and handles determinants, that does not translate in an increase of the power draw. It does, however, impact the energy consumption as a consequence of the overhead in the execution time. In a fault-free scenario, message-logging and parallel recovery increase energy consumption directly proportional to the overhead.

Secondly, there is an impact of changing the ratio $max/base$ power. The smaller this ratio is, the more energy message-logging and parallel recovery can save. The experimental results showed that during recovery, the non-faulty nodes return to a value close to the base power (§IV Figure 4). If the base power is smaller relative to the max power, the savings during recovery can be substantial. Table IV shows a power comparison of different architectures released over the last 3 years. These results were obtained by running a 5-point stencil CHARM++ application, *Wave2D*, on a single node of each of these architectures. The table suggests the ratio base/max power is decreasing with time. The oldest machine, Intel Xeon, has the highest ratio (0.48), whereas the most recent machine, Sandy Bridge, has the lowest ratio (0.21). This trend points towards a lot of potential for energy consumption savings using message-logging and parallel recovery.

| Processor | Release Date | Max Power | Base Power | Base/Max Ratio |
|---|---|---|---|---|
| **Intel Xeon E5520** | Q1,09 | 125 | 60 | 0.48 |
| **Intel Nehalem i7 860** | Q3,09 | 151 | 52 | 0.34 |
| **Intel Sandy Bridge i7 2600** | Q1,11 | 101 | 21 | 0.21 |

TABLE IV
POWER STATISTICS FOR DIFFERENT ARCHITECTURES RUNNING *Wave2D*

Thirdly, overdecomposition (the ability of dividing a computation into small units) has an impact on how much energy consumption can be reduced. This degree of freedom empowers the runtime system. Overdecomposition is not only fundamental to the load balance of an application, but also to the available parallelism during recovery ($P$ in our model). As we saw in section V, the higher the value of $P$ the more energy saving and more scalability for parallel recovery. Programming models that encourage overdecomposition will be in a better position for accelerating recovery and achieving better utilization of the system.

## VII. Related Work

To the best of our knowledge there is only one work in the literature that deals with fault tolerance and energy consumption issues [13]. In that paper, the authors present a series of benchmarks to measure how much power draw and energy consumption is involved in doing three tasks for fault tolerance: checkpointing, coordinating tasks and logging messages. None of these tasks significantly increases the power draw of a node, but logging messages increases the total energy consumed by a program. However, in the context of four NPB applications (BT,CG,LU,SP), it seems logging messages can be comparable to coordinating tasks. They do not present results with failures of nodes, nor do they include parallel recovery and there is no model to predict the energy consumed at extreme scales.

The MPICHV project implemented several fault tolerance strategies, including checkpoint/restart and a handful of message-logging variants. A comparison between coordinated checkpoint/restart and causal message-logging [14] shows that causal message-logging is able to tolerate a higher failure rate than checkpoint/restart. However, the message-logging overhead comes close to $20\%$ in the failure-free execution. Part of the reason for such a high overhead is the use of a centralized *event logger* that handles all the determinants of the system. The results were not scaled beyond 25 cores.

## VIII. Conclusion and Future Work

This paper provided an energy efficiency comparison of three fault tolerance protocols: checkpoint/restart, message-logging and parallel recovery. We evaluated the three protocols by using a set of benchmarks from two different programming models. Using the experimental results as a guide, we built an analytical model to predict the behavior of the protocols at extreme scales.

Here is a summary of the findings in this paper:

- There is no empirical evidence to claim that message-logging will increase the *power* draw when used as a fault tolerance technique. It will, though, increase the *energy* consumption in a failure-free scenario due to the overhead it imposes in the progress rate of an application.
- The experimental results showed that parallel recovery outperforms the other two approaches in both execution time and energy consumption in a faulty execution. It does so by accelerating the recovery of a node via task migration. In that sense, parallel recovery can satisfy both users (minimum execution time) and system administrators (minimum energy consumed).
- The analytical model predicts that parallel recovery will be fundamental at extreme scales in decreasing the execution time and energy consumption of applications. The savings can be as much as $17\%$ in execution time and $13\%$ in energy consumption.

For the future, we are planning to understand how these fault tolerance protocols work with a full-fledged application. In particular, we are interested in particle-interaction simulations. We believe these programs have the ability to generate a high degree of overdecomposition and will be a good match for parallel recovery.

## Acknowledgments

This research was supported in part by the US Department of Energy under grant DOE DE-SC0001845 and by a machine allocation on the Teragrid under award ASC050039N. We thank Prof. Tarek F. Abdelzaher for giving us access to the testbed used in this paper.

## 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] J. Dongarra, P. Beckman, T. Moore, P. Aerts, G. Aloisio, D. Barkai, T. Boku, B. Chapman, X. Chi, A. Choudhary, S. Dosanjh, T. Dunning, R. Fiore, A. Geist, R. Harrison, M. Hereld, M. Heroux, K. Hotta, Y. Ishikawa, Z. Jin, F. Johnson, S. Kale, R. Kenway, D. Keyes, B. Kramer, J. Labarta, A. Lichnewsky, B. Lucas, S. Matsuoka, P. Messina, P. Michielse, B. Mohr, M. Mueller, J. Shalf, D. Skinner, M. Snir, T. Sterling, R. Stevens, F. Streitz, B. Sugar, A. V. D. Steen, J. Vetter, P. Williams, R. Wisniewski, and K. Yelick, "The international exascale software project roadmap 1."

[3] 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.

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

[5] 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.

[6] 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.

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

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

[9] 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.

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

[11] 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.

[12] 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 *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*. New York, NY, USA: ACM, 2011, pp. 44:1–44:12. [Online]. Available: http://doi.acm.org/10.1145/2063384.2063443

[13] M. Diouri, O. Gluck, L. Lefevre, and F. Cappello, "Energy considerations in checkpointing and fault tolerance protocols," in *2nd Workshop on Fault-Tolerance for HPC at Extreme Scale (FTXS 2012)*, Boston, USA, Jun. 2012.

[14] P. Lemarinier, A. Bouteiller, T. Herault, G. Krawezik, and F. Cappello, "Improved message logging versus improved coordinated checkpointing for fault tolerant MPI," *Cluster Computing, IEEE International Conference on*, vol. 0, pp. 115–124, 2004.