

Simulating Application Resilience at Exascale

Rolf Riesen¹, Kurt Ferreira², Maria Ruiz Varela³, Michela Taufer³, and Arun Rodrigues²

¹ IBM Research rolf.riesen@ie.ibm.com

² Sandia National Laboratories* {kbferre|afrodri}@sandia.gov

³ University of Delaware mruiz@udel.edu, taufer@cis.udel.edu

Abstract. The reliability mechanisms for future exascale systems will be a key aspect of their scalability and performance. With the expected jump in hardware component counts, faults will become increasingly common compared to today’s systems. Under these circumstances, the costs of current and emergent resilience methods need to be reevaluated. This includes the cost of recovery, which is often ignored in current work, and the impact of hardware features such as heterogeneous computing elements and non-volatile memory devices. We describe a simulation and modeling framework that enables the measurement of various resilience algorithms with varying application characteristics. For this framework we outline the simulator’s requirements, its application communication pattern generators, and a few of the key hardware component models.

1 Introduction

Parallel scientific applications frequently use coordinated checkpoint and restart (CCR) to recover from system failures. Failures can be anything from loss of power, human error, hardware component faults, to software bugs. For an application using CCR, all of these failures force it to abort and, at a later time, to restart from a previous checkpoint. Several studies have shown that this will not scale much beyond the machines currently in existence [4, 8, 3, 11].

For exascale systems, even if per-component reliability remains the same, the sheer number of components will lead to frequent faults. Therefore, alternative methods are needed to enable computational progress of large-scale applications.

Many alternative resilience algorithms have been proposed to replace CCR, but few have been evaluated thoroughly at large scale, with differently behaving applications, strong scrutiny of their cost – especially for recovery – and the impact on application throughput. Recovery is often assumed to be infrequent and neglected in performance studies. In exascale systems we expect failures to be common and that cascading failures during recovery might change the performance characteristics of resilience algorithms substantially.

Another aspect that is sometimes overlooked is that a given resilience algorithm may not be suitable for all types of applications. For example, CCR works

* Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy’s National Nuclear Security Administration under contract DE-AC04-94AL85000.

well for self-synchronizing applications, since they already bear the synchronization cost necessary to achieve coordination. Other applications do better without introducing additional synchronization steps.

While exascale systems will not be radically different from today’s supercomputers, there are features such as massive multicore CPUs, Solid State Disks (SSD), and non-volatile random access memory (NVRAM) that have impact on the performance of resilience algorithms. Application characteristics may also become different when they adapt to the larger scale and new programming models. Yet, self-synchronizing legacy applications need to be supported as well.

To evaluate proposed and existing resilience algorithms at scale, simulation and modeling is needed. In this paper we analyze the requirements for an evaluation framework that lets us measure the performance and overhead of various resilience algorithms with different application characteristics.

This perspective paper is meant to explore future exascale systems in terms of modeling and other relevant aspects that have to be considered when studying application recovery after failures. A goal is to generate a discussion that will help define the taxonomy of future exascale systems and the tools that will enable us to study them even before they become available.

We list the requirements we have identified in Section 2, describe our design in Section 3 and 4, and report on the status of our implementation in Section 5.

2 Requirements

The compromises and restrictions we will have to put into our simulation will prevent us from being able to make absolute and precise performance predictions. However, the goal is to make *relative* performance comparisons among resilience algorithms under various conditions. For that we need a somewhat accurate model of data movement within the system, but not the data itself nor the computations necessary to generate that data.

Before we can design an experiment, we need to get an idea of what a future exascale system might look like [2, 3]. Since we cannot simulate a complete system at scale in full fidelity, we then need to identify the aspects of a system that have a measurable impact on the performance of resilience algorithms.

2.1 Future Systems

Exascale machines are predicted to appear before the end of this decade. That is too far out to make accurate predications on what such a system will look like, but not so far that we cannot make some educated guesses. It will not be a quantum computer and most aspects of the system will be familiar to today’s users of supercomputers.

We have about five more iterations of Moore’s law ahead of us and can expect to see about 512 to 1,024 cores per socket in such a system. If the current trend continues, each core will have relatively weak performance to help with power consumption and enable the placement of that many cores onto a single die.

The current number one system on the top500 list employs 548,352 cores to achieve 8 petaflops. The number of cores per CPU, as well as their total number, will increase to reach an exaflop. These cores will be connected with each other through a Network on Chip (NoC). Most likely there will be a complex hierarchy of caches where some cores in the same “neighborhood” share lower-level caches such as an L2, and groups of cores share L3 caches, and the memories shared by these cores may not be shared coherently.

The memory hierarchy will be further complicated by some or all of main memory becoming non-volatile (NVRAM). SSD with faster access times than spinning media will also be prevalent. Some of that storage will be local, in the same rack for example, while more of it will be farther away in a dedicated storage server. Compute accelerators, such as Graphical Processing Units (GPUs), on the same motherboard or integrated into CPUs will most likely also play a role in achieving exascale performance by providing additional compute cycles and processing stream-oriented application kernels.

2.2 Simulation and Modeling

With the above assumptions, it is not possible to simulate such a system with high fidelity. There are simply too many components and not enough technological certainty for a fully detailed simulation in a reasonable amount of time. In order to make evaluation of different resiliency algorithms possible, we have to make some compromises. We can leave out some less important aspects and still arrive at results that are valid when comparing two different resiliency algorithms for a given type of application.

The first thing we will abandon is an application’s computation. Obviously, this will save a lot of simulation time by allowing us to dispense with a detailed processor model or emulation framework. Furthermore, resilience algorithms are dependent only on two aspects of the computation itself: How much data it touches and changes over time, and the duration of compute phases between data exchanges with other cores and nodes; i.e., externally visible state changes.

While we can dispense with computation, we cannot be quite so cavalier with communication. Cores on a single die will communicate with each other over the NoC and, nodes will communicate with each other over a system-wide network. The exact form of communication is less important. Some of that data will be transferred using MPI, while other data will be written directly into memory. Because these are externally visible state changing events, resilience algorithms depend on the timing of these transfers and the amount of data being moved. Performance, frequency, and location of saving and restoring state depends on data traffic. However, the actual content of these messages does not matter.

Because moving data is a large overhead and influences resilience algorithm performance, a fairly accurate simulation of data flowing through a system is necessary. The simulation needs enough resolution to detect congestion and measure its impact. The same applies to I/O. State needs to be saved into remote memory, NVRAM, and SSD devices. While access times to individual memory banks

are too fine grained to track in a simulation of this scale, access competition to these devices and transfer times do need to be tracked.

Finally, but not least, the simulation needs to provide a method to inject faults into the system. A form of notifying a resilience algorithm that a node, socket, core, or link has failed, with the corresponding data loss, is necessary. But the exact type of failure notification is not that important.

3 Design of our Simulation Infrastructure

For our exascale fault resilience simulation we have to create several components: A router that lets us configure a system wide network as well as the NoC for each socket, a storage device we can use to simulate data flow into and out of NVRAMs and SSDs, and an end-point component that generates the data traffic we need. Because of the complexity of this simulation, we also need an automatic way to generate the large configuration files.

3.1 The Structural Simulation Toolkit (SST)

We use SST, a parallel discrete event simulator developed at Sandia National Laboratories [12]. SST is a C++ framework to integrate various simulators and models and connect them via an event network. An XML file is used to configure SST at startup time. That file specifies what components are to be used and how these components are connected. Events travel along the links specified in the configuration file.

3.2 Router Model

Network routers in supercomputers are very simple and small, when compared to Ethernet routers in a data center. Supercomputer routers usually have few ports, five or seven for example to create 2-D or 3-D meshes and tori. Often they use source-based routing where the message itself contains information about which output port to use. Commonly, they are wormhole routed, and they are employed by the thousands to create the main network infrastructure of systems like Cray's XT series and IBM's Bluegene machines.

We created an SST component that models such a router at a behavioral level. A model is faster to compute and easier to write than a gate-level simulation. And, since we are feeding an approximate data stream into the network, a more accurate simulation would not provide us with more reliable results.

Figure 3.2 shows the concept of our router model. The model supports an arbitrary number of ports. When we use it as a component to create the main network of our exascale simulation, we configure it (in the SST configuration file) to have five ports: Four to create the torus topology of the main network, and a fifth port to connect to a compute node. Incoming traffic is handled in FIFO order, to preserve message ordering. Messages arrive in the form of events.

The events themselves could contain message data, but since we are not generating that data, the events only contain the number of bytes the message would contain. That message length, a configurable router latency, and bandwidth are used to calculate how long an output port will be occupied. For that duration, further messages destined for that output port, are queued.

Omitting modeling of flow control between routers reduces synchronization overhead. However, since incoming messages are queued when an output port is busy, a message traveling through two or more routers cannot move faster than the bandwidth and latency limitations – as well as other traffic in the network – allow. Messages can be delayed at the input or output port. A quick stream of short messages on an input port can be held up by a larger message using the same output port. This mechanism gives the router model a crude approximation of flow control. If a message is delayed due to a busy port, congestion statistics in the delayed event are updated.

The router model accepts, via the SST configuration file, several parameters: A hop delay specifies the minimum amount of time a message (event) is delayed when passing through a router⁴. The bandwidth parameter, together with the incoming message length, dictates how long a message occupies a port. The number of ports is also a configurable parameter. Two more parameters are used for power and thermal modeling. One is the hypothetical frequency this router runs at, and another dictates which power model to use; SST supports several [5]. The router model can use the amount of traffic and the above parameters to compute power dissipation.

Note that links configured between components, for example between two routers in the SST configuration file, also have a delay assigned to them. SST uses that when partitioning the graph of components between processes in a parallel simulation and to compute event lookahead.

We use the same router model component described so far to also create the NoC within each socket of our simulation. To keep things simple, we assume the NoC is also a torus. However, instead of using five-port routers, we allow for additional ports to connect more than one CPU core to each router in a NoC. A bit in events traveling between cores attached to the same router indicates local traffic that moves at higher bandwidth than off-CPU traffic. We assume that these cores will be communicating through a shared cache instead of making use of the NoC.

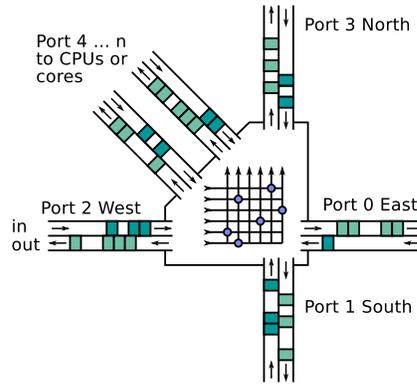


Fig. 1. The router model.

⁴ The actual delay may be much larger if there is congestion on the input or output ports.

Additionally, when used for a NoC, the router model does not use wormhole routing. This is a more realistic mode of operation when the NoC also connects to random access memory, where multiple streams of data can be overlapping and be destined for the same device.

Figure 3.2 is a diagram that shows one possible configuration of a node in our exascale simulation. The router model is used to build the NoC as well as the main network that connects the nodes in a system. Each node consists of multiple SST components described in this section.

The router model is also used as an aggregator to coordinate data traffic to a single resource. In that configuration, one port connects to the resource, and all remaining ports connect to users of the resource. We use aggregators to control access to the main network from each node. Each core can access the main network, but has to compete with any other core on that node for that resource.

This is akin to multiple cores and CPUs on a node sharing a single NIC. We also use aggregators to gate access to on-board NVRAM, which is a shared resource for the cores on that board. Each core also has access to a storage network to access a nearby SSD. We assume that access to a parallel file system will happen through the main network, as it does on most of today's machines. But, we also envision that each rack has some SSD devices for scratch storage and that nodes in the same rack have access to that storage via a separate, but local, storage network.

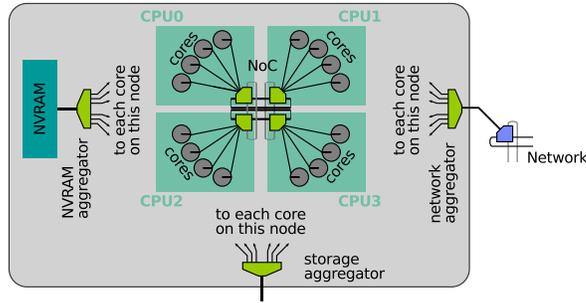


Fig. 2. Combining components for a node.

3.3 Storage Component

A requirement to evaluate resilience algorithms is to simulate access to storage. Two types of storage will be important for these algorithms. In each node we will have some amount of NVRAM that is accessible to the cores on that node. Figure 3.2 shows that we use an aggregator to coordinate access to each NVRAM. The NVRAM itself is a very simple SST component. It accepts data write requests and queues them. Based on a write speed parameter in the configuration file, this queue is processed and write acknowledgements are sent back to the requester when an item has been removed from the queue. There is a similar queue for read requests and read data events are sent back to requesters based on the read speed of the device and the number and size of pending requests.

For resilience methods that access remote memory, data has to be transferred using the main network first. Then, one of the cores on a node with direct access to the local NVRAM has to handle these remote requests.

The second type of storage we provide in our simulation is a “nearby” SSD. We assume that each rack has some amount of SSD storage that can be used for temporary data. A rack-wide, local network provides access to that storage. We use aggregators to build a two-level tree storage network for each rack.

We assume that rack SSD storage has more capacity and is more reliable than the individual local NVRAMs. A rack will have multiple, redundant power supplies, and the SSDs will be RAID devices. The NVRAM on a node is quicker to access and has less contention. But, it also has a smaller capacity and may become inaccessible if a node, or the network connection to a node, fails.

At this time we have no plans to simulate a remote parallel file server. We assume that data from the rack SSD can be trickled off to such a server in the background, if desired. Other research teams are working on full disk simulation components for SST, including an SSD device, that we will be able to integrate at a later time, if necessary.

3.4 Communication Pattern Generators and Applications

We have, in Section 2, mentioned that we cannot afford the cost of running or simulating the application processes that are the endpoints of our network infrastructure. Yet, we do need these endpoints; i.e., the processes running on each core, to transmit and receive data. For our evaluation of resilience algorithms, we need the approximate timing of these transmissions and their causal ordering. In other words, we need what we call communication pattern generators.

What we mean by a communication pattern is illustrated in Figure 3. It shows an example from the NAS parallel benchmark suite. For each rank in the computation, the diagram shows (approximately) *how many messages* each rank sends to all other ranks. Similar plots can be generated for the *amount of data* sent between ranks [10].

Generating such patterns for a variety of applications and benchmarks is not difficult. For example, a five-stencil computation using ghost (halo) cells to exchange data with neighboring ranks has the following loop structure: Send data to four neighbors, wait for data from these neighbors, perform computation, repeat. Once in a while a collective operation, such as an allreduce, is inserted to determine whether convergence of the result has been achieved.

Many similar, fundamental patterns exist that are employed by applications today. Our simulator is capable of producing all of them, as long there is no data dependency; i.e., as long as the communication pattern does not change based on the content of these messages. Currently we have communication patterns for a five-stencil ghost cell exchange, and two micro benchmarks: A ping-pong pattern that measures latency and bandwidth, and a message rate benchmark. We also have a state machine that implements a barrier operation. The only resilience algorithm implemented so far is CCR.

Additional communication patterns we are going to implement include the behavior of the NAS parallel benchmark programs FT and IS, as well as some patterns that originate from various parallel graph algorithms; e.g., [9].

3.5 Implementing Pattern Generators

SST is an event driven parallel simulator. Each component that is integrated into the SST framework needs to process events it receives and then relinquish control back to SST so that the overall simulation can proceed. A natural way of expressing and implementing the communication patterns we need to drive our simulations, are state machines. Event processing is an integral part of state machines. Therefore, that choice is simple.

What makes this choice a little bit difficult is the state explosion when we combine a communication pattern generator, a resilience algorithm, collective operations, and the handling of asynchronous I/O events and faults. What starts out with a handful of states to express a nearest neighbor data exchange becomes much more complicated when some events from a collective operation, that has already started on another rank, arrive early. Even more states are needed to process the requirements of the resilience algorithm under evaluation. The algorithm will generate I/O and completion events will arrive asynchronously. A state machine describing all these possibilities will grow very complex quickly.

In addition, we want to use the same communication pattern with different resilience algorithms and, perhaps, different implementations of collective operations. The solution we have chosen is that of a gate keeper. It is a C++ SST component from which all communication patterns inherit. Each instantiated communication pattern component specifies what other services it needs; e.g. which collective operations it will perform. The resilience algorithm is chosen through the configuration file. All of these individual, relatively simple state machines, register an event handler with the gate keeper.

In some respect, the individual state machines are all subroutines of the communication patten generator. At any given time, only one of those state machines is active. When a new event arrives at the gate keeper, it determines which state machine needs to receive that event. If that state machine is currently running, then the event is delivered right away. For currently inactive state machines, (early) events are queued. The gate keeper component provides functions for state machines to call each other and to return to a previous caller. Whenever a state machine change occurs, pending events for the newly active state machine are delivered by the gate keeper.

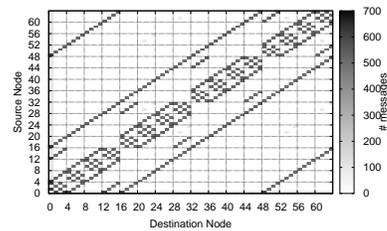


Fig. 3. Communication patterns for NAS MG, class C, 64 nodes [10].

4 Tying it all Together

In the XML configuration file for SST, every component to be used, every link between any two components, and the parameters for each component need to be specified. The file structure allows for a common shared parameter, among a set of components, to be specified only once. Nevertheless, these files, for a large simulation, are too big to be created manually. Therefore, we wrote a separate program to create configuration files specific to the subset of SST components used for our experiments.

The configuration generator takes command line arguments, for main network bandwidth for example, and inserts it into the appropriate places in the XML file. The choice of which communication pattern to use is also a command line option, while several other things are hard coded into the generator.

For example, the generator takes as command line parameters the X and Y dimension of the main network and a separate pair of parameters for the NoC on each node. But, it is currently hard coded to generate tori. This makes several things a lot simpler, including source route generation, and should suffice for our initial experiments.

The simulation design described in this paper has several limitations. Some stem from the core design. These include the inability to create communication patterns that are data dependent, computation delays between communications that vary with the results of computation, and the inaccuracies introduced by using models instead of more fine-grained simulation.

Other limitations are caused by our implementation choices and the configuration generator. These include things like the fixed topologies and the architecture of the I/O and memory subsystem. These are more easily corrected than our design choices by adapting our code to the new requirements.

5 Work in Progress

Work in progress includes the study of simple resilience algorithms for exascale systems, beyond the 256k cores we have already tested, with the support of the framework proposed in this paper. Future work includes integrating a larger set of resilient algorithms and a broader range of applications with their communication patterns in the framework.

For resilience algorithms and methods we plan to look at uncoordinated checkpoint restart with message logging, log-based rollback-recovery mechanisms [6], the RAID-like approach taken by SCR [7], and communication induced checkpointing [1].

Validation of a complex simulation tool like ours is of course made extremely difficult by the lack of existing exascale systems. Nevertheless, we plan to use micro-benchmarks to calibrate various parameters and models built into our simulation by comparing them against existing systems. Then we will run benchmarks and applications that our communication patterns are meant to mimic on existing, large-scale systems, and compare the results with our simulations.

We will be able to do this using individual multicore CPUs, large clusters, and clusters containing multicore CPUs. Viewing and comparing the actual results with our simulations from these different angles will provide us with an indication of the validity of our approach. Scaling experiments within the range of systems available to us will further assist with validation and provide us with error bars for simulations at exascale.

We are building the simulation infrastructure to evaluate resilience algorithms. However, the same infrastructure will be suitable for evaluation of many different aspects of exascale computing. We have started to investigate projects in the area of programming models and application performance on a heterogeneous network where not all components are (virtually) fully connected.

SST, including the components described in this paper, is open source and freely available.

References

1. Alvisi, L., Elnozahy, E.N., Rao, S., Husain, S.A., Mel, A.D.: An analysis of communication induced checkpointing. In: FTCS (1999)
2. Bergman, K., et al.: Exascale computing study: Technology challenges in achieving exascale systems (2008)
3. Bianchini, R., et al.: System resiliency at extreme scale (2009)
4. Elnozahy, E., Plank, J.: Checkpointing for peta-scale systems: a look into the future of practical rollback-recovery. *Dependable and Secure Computing, IEEE Transactions on* 1(2) (2004)
5. Hsieh, M., Thompson, K., Song, W., Rodrigues, A., Riesen, R.: A framework for architecture-level power, area and thermal simulation and its application to network-on-chip design exploration. In: 1st Intl. Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computing Systems (PMBS 10) (Nov 2010)
6. Maloney, A., Goscinski, A.: A survey and review of the current state of rollback-recovery for cluster systems. *Concurrency and Computation: Practice and Experience* (Apr 2009)
7. Moody, A., Bronevetsky, G., Mohror, K., Supinski, B.R.d.: Design, modeling, and evaluation of a scalable multi-level checkpointing system. In: SC (2010)
8. Oldfield, R.A., Arunagiri, S., Teller, P.J., Seelam, S., Varela, M.R., Riesen, R., Roth, P.C.: Modeling the impact of checkpoints on next-generation systems. In: 24th IEEE Conference on Mass Storage Systems and Technologies (Sep 2007)
9. Ribeiro, P., Silva, F., Lopes, L.: Efficient parallel subgraph counting using g-tries. In: *Cluster Computing* (2010)
10. Riesen, R.: Communication patterns. In: *Workshop on Communication Architecture for Clusters CAC'06* (Apr 2006)
11. Riesen, R., Ferreira, K., Stearley, J.: See applications run and throughput jump: The case for redundant computing in HPC. In: 1st Intl. Workshop on Fault-Tolerance for HPC at Extreme Scale (FTXS 2010) (Jun 2010)
12. Rodrigues, A., Cook, J., Cooper-Balis, E., Hemmert, K.S., Kersey, C., Riesen, R., Rosenfield, P., Oldfield, R., Weston, M., Barrett, B., Jacob, B.: The structural simulation toolkit. In: 1st Intl. Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computing Systems (PMBS 10) (Nov 2010)