Fault Tolerance Techniques for Scalable Computing*

Pavan Balaji, Darius Buntinas, and Dries Kimpe
Mathematics and Computer Science Division
Argonne National Laboratory
{balaji, buntinas, dkimpe}@mcs.anl.gov

Abstract

The largest systems in the world today already scale to hundreds of thousands of cores. With plans under way for exascale systems to emerge within the next decade, we will soon have systems comprising more than a million processing elements. As researchers work toward architecting these enormous systems, it is becoming increasingly clear that, at such scales, resilience to hardware faults is going to be a prominent issue that needs to be addressed. This chapter discusses techniques being used for fault tolerance on such systems, including checkpoint-restart techniques (system-level and application-level; complete, partial, and hybrid checkpoints), application-based fault-tolerance techniques, and hardware features for resilience.

1 Introduction and Trends in Large-Scale Computing Systems

The largest systems in the world already use close to a million cores. With upcoming systems expected to use tens to hundreds of millions of cores, and exascale systems going up to a billion cores, the number of hardware components these systems would comprise would be staggering. Unfortunately, the reliability of each hardware component is not improving at

*This work was supported by the Office of Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy, under Contract DE-AC02-06CH11357.

the same rate as the number of components in the system is growing. Consequently, faults are increasingly becoming common. For the largest supercomputers that will be available over the next decade, faults will become a norm rather than an exception.

Faults are common even today. Memory bit flips and network packet drops, for example, are common on the largest systems today. However, these faults are typically hidden from the user in that the hardware automatically corrects these errors by error correction techniques such as error correction codes and hardware redundancy. While convenient, unfortunately, such techniques are sometimes expensive with respect to cost as well as to performance and power usage. Consequently, researchers are looking at various approaches to alleviate this issue.

Broadly speaking, modern fault resilience techniques can be classified into three categories:

1. **Hardware Resilience:** This category includes techniques such as memory error correction techniques and network reliability that are transparently handled by the hardware unit, typically by utilizing some form of redundancy in either the data stored or the data communicated.

2. **Resilient Systems Software:** This category includes software-based resilience techniques that are handled within systems software and programming infrastructure. While this method does involve human intervention, it is usually assumed that such infrastructure is written by expert "power users" who are willing to deal with the architectural complexities with respect to fault management. This category of fault resilience is mostly transparent to end domain scientists writing computational science applications.

3. **Application-Based Resilience:** The third category involves domain scientists and other high-level domain-specific languages and libraries. This class typically deals with faults using information about the domain or application, allowing developers to make intelligent choices on how to deal with the faults.

In this chapter, we describe each of these three categories with examples of recent research. In Section 2, we describe various techniques used today for hardware fault resilience in memory, network and storage units. In Section 3, we discuss fault resilience techniques used in various system software libraries, including communication libraries, task-based models, and large data models. In Section 4, we present techniques used by application and domain-specific languages in dealing with system faults. In Section 5, we summarize these different techniques.

2 Hardware Features for Resilience

This section discusses some of the resilience techniques implemented in processor, memory, storage and network hardware. In these devices, a failure occurs when the hardware is unable to accurately store, retrieve or transmit data. Therefore most resilience techniques focus on detection and reconstruction of corrupted data.

2.1 Processor Resilience

Detecting errors in the execution of processor instructions can be accomplished by redundant execution, where a computation is performed multiple times and the results are compared. In [52], Qureshi et al. identify two classes of redundant execution: space redundant and time redundant. In space redundant execution, the computation is executed on distinct hardware components in parallel, while in time redundant execution, the computation is executed more than once on the same hardware components. The technique presented in [52] is a time redundant technique which uses the time spent waiting for cache misses to perform the redundant execution. Oh, et al. describe a space redundant technique in [47] using super-scalar processors. In this technique separate registers are used to store the results for each of the duplicated instructions. Periodically, the values in the registers are compared in order to detect errors.

2.2 Memory Resilience

A memory error can be defined as reading the logical state of one or more bits differently from how they were written. Memory errors are classified as either *soft* or *hard*. Soft errors are transient; in other words, they typically do not occur repeatedly when reading the same memory location and are caused mainly by electric or magnetic interference. Hard errors are persistent. For example, a faulty electrical contact causing a specific bit in a data word to be always set is a hard memory error. Hard errors are often caused by physical problems. Note that memory errors do not necessarily originate from the memory cell itself. For example, while the memory contents can be accurate, an error can occur on the path from the memory to the processor.

The failure rate (and trend) of memory strongly depends on the memory technology [59]. DRAM stores individual bits as a charge in a small capacitor. Because of leaking from the capacitor, DRAM requires periodic refreshing. DRAM memory cells can be implemented by using a single transistor and capacitor, making them relatively inexpensive to implement, so most of the memory found in contemporary computer systems consists of DRAM. Unfortunately, like other memory technologies, DRAM is susceptible to soft errors. For example, neutrons originating from cosmic rays can change the contents of a memory cell [24].

It is often assumed that when decreasing chip voltages in order to reduce the energy required to flip a memory bit and increasing memory densities, the per bit soft error rate will increase significantly [44, 58]. A number of studies, however, indicate that this is not the case [19, 33, 8].

The DRAM error rate, depending on the source, ranges from 10^{-10} to 10^{-17} errors per bit per hour. Schroeder and Gibson show that memory failures are the second leading cause of system downtime [56, 57] in production sites running large-scale systems.

Memory resilience is achieved by using *error detection* and *error correction* techniques. In both cases, extra information is stored along with the data. On retrieval, this extra information is used to check data consistency. In the case of an error correction code (ECC), certain errors can be corrected to recover the original data.

For error detection, the extra information is typically computed by using a hash function. One of the earliest hash functions used for detecting memory errors is the parity function. For a given word of d bits, a single bit is added so that the number of 1 bits occurring in the data word extended by the parity bit is either odd (odd parity) or even (even parity). A single parity bit will detect only those errors modifying an *odd* number of bits. Therefore, this technique can reliably detect only those failures resulting in the modification of a single data bit.

Parity checking has become rare for main memory (DRAM), where it has been replaced by error-correcting codes. However, parity checking and other error detection codes still have a place in situations where detection of the error is sufficient and correction is not needed. For example, instruction caches (typically implemented by using SRAM), often employ error detection since the cache line can simply be reloaded from main memory if an error is detected. On the Blue Gene/L and Blue Gene/P machines, both L1 and L2 caches are parity protected [63].

Since on these systems memory writes are always write-through to the L3 cache, which uses ECC for protection, error detection is sufficient in this case even for the data cache.

When an error-correcting code is used instead of a hash function, certain errors can be corrected in addition to error detection.

For protecting computer memory, hamming codes [31] are the most common. While pure hamming codes can detect up to two bit errors in a word and can correct a single-bit error, a double-bit error from a given data word and a single-bit error from another data word can result in the same bit pattern. Therefore, in order to reliably distinguish single-bit errors (which can be corrected) from double-bit errors (which cannot be corrected), an extra parity bit is added. Since the parity bit will detect whether the number of error bits was odd or even, a failed data word that fails both the ECC and the parity check indicates a single-bit error, whereas a failed ECC check but correct parity indicates an uncorrectable dual-bit error. Combining a hamming code with an extra parity bit results in a code that is referred to as single error correction, double error detection (SECDED).

Unfortunately, memory errors aren't always independent. For example, highly energetic particles might corrupt multiple adjacent cells, or a hard error might invalidate a complete memory chip. In order to reduce the risk of a single error affecting multiple bits of the same logical memory word, a number of techniques have been developed to protect against these failures. These techniques are, depending on the vendor, referred to as *chip-kill*, *chipspare*, or *extended ECC*. They work by spreading the bits (including ECC) of a logical memory word over multiple memory chips, so that each memory chip contains only a single bit of each logical word. Therefore, the failure of a complete memory chip will affect only a single bit of each word as opposed to four or eight (depending on the width of the memory chip) consecutive bits.

Another technique is to use a different ECC. Such ECC codes become relatively more space-efficient as the width of the data word increases. For example, a SECDED hamming code for correcting a single bit in a 64-bit word takes eight ECC bits. However, correcting a single bit in a 128-bit word requires only nine ECC bits. By combing data into larger words, one can use the extra space to correct more errors. With 128-bit data words and 16 ECC bits, it is possible to construct an ECC that can correct random single-error bits but up to four (consecutive) error bits.

Since ECC memory can tolerate only a limited number of bit errors and since errors are detected and corrected only when memory is accessed, it is beneficial to periodically verify all memory words in an attempt to reduce the chances of a second error occurring for the same memory word. When an error is detected, the containing memory word can be rewritten and corrected before a second error in the same word can occur. This is called *memory scrubbing* [55]. Memory scrubbing is especially important for servers, since these typically have large amounts of memory and very large uptimes, thus increasing the probability of a double error.

The use of ECC memory is almost universally adopted for supercomputers and servers. This is the case for the IBM Blue Gene/P [63] and the Cray XT5 [1]. Note that the IBM Blue Gene/L did not employ error correction or detection for its main memory. Personal

computing systems such as laptops and home computers typically do not employ ECC memory.

2.3 Network Resilience

Network fault tolerance has been a topic of continued research for many years. Several fault tolerance techniques have been proposed for networks. In this section, we discuss three techniques: reliability, data corruption, and automatic path migration.

Reliability. Most networks used on large-scale systems today provide reliable communication capabilities. Traditionally, reliability was achieved by using kernel-based protocol stacks such as TCP/IP. In the more recent past, however, networks such as InfiniBand [64] and Myrinet [18] have provided reliability capabilities directly in hardware on the network adapter. Reliability is fundamentally handled by using some form of a handshake between the sender and receiver processes, where the receiver has to acknowledge that a piece of data has been received before the sender is allowed to discard it.

Data Corruption. Most network today automatically handle data corruption that might occur during communication. Traditional TCP communication relied on a 16-bit checksum for data content validation. Such low-bit checksums, however, have proved to be prone to errors when used with high-speed networks or networks on which a lot of data content is expected to be communicated [60]. Modern networks such as InfiniBand, Myrinet, and Converged Ethernet[1] provide 32-bit cyclic-redundancy checks (CRCs) that allow the sender to hash the data content into a 32-bit segment and the receiver to verify the validity of the content by recalculating the CRC once the data is received. Some networks, such as InfiniBand, even provide dual CRC checks (both 16-bit and 32-bit) to allow for both end-to-end and per-network-hop error corrections.

One of the concerns of hardware managed data corruption detection is that they are

[1] Converged Ethernet is also sometimes referred to as Converged Enhanced Ethernet, Datacenter Ethernet, or Lossless Ethernet.

not truly end to end. Specifically, since the CRC checks are performed on the network hardware, they cannot account for errors while moving the data from the main memory to the network adapter. However, several memory connector interconnects, such as PCI Express and HyperTransport, also provide similar CRC checks to ensure data validity. Nevertheless, the data has no protection all the way from main memory of the source node to the main memory of the destination node. For example, if an error occurs after data validity is verified by the PCI Express link, but before the network calculates its CRC, such an error will go undetected. Consequently, researchers have investigated software techniques to provide truly end-to-end data reliability, for example by adding software CRC checks within the MPI library.[2]

Automatic Path Migration. Automatic path migration (APM) is a fairly recent technique for fault tolerance provided by networks such as InfiniBand. The basic idea of APM is that each connection uses a primary path but also has a passive secondary path assigned to it. If any error occurs on the primary path (e.g., a network link fails), the network hardware automatically moves the connection to the secondary fallback path. Such reliability allows only one failure instance, since only one secondary path can be specified. Further, APM protects communication only in cases where an intermediate link in the network fails. If an end-link connecting the actual client machine fails, APM will not be helpful.

A secondary concern that researchers have raised with APM is the performance implication of such migration. While migrating an existing connection to a secondary path would allow the communication to continue, it might result in the migrated communication flow interfering with other communication operations thus causing performance loss. Unfortunately, currently no techniques have been shown to work around this issue specifically, although the recently introduced adaptive routing capabilities in InfiniBand work around this problem.

[2]The MVAPICH project is an example of such an MPI implementation: http://mvapich.cse.ohio-state.edu.

2.4 Storage Resilience

Two types of storage devices can be found in modern large installation sites: electromechanical devices, which contain a spinning disks (i.e., traditional magnetic hard drives), and solid-state drives (SSD), which use a form of solid-state memory.

Spinning disks partition data into sectors. For each sector, an ECC is applied (typically a Reed-Solomon code [70]).

The most common type of solid-state drive uses flash memory internally to hold the data. There are two common types of flash, differentiated by how many bits are stored in each cell of the flash memory. In Single Level Cell (SLC) flash, a cell is in either a low or high state, encoding a single bit. In Multi Level Cell (MLC) flash, there are four possible states, making it possible to store two bits in a single cell.

For SLC devices, hamming codes are often used to detect and correct errors. A common configuration is to organize data in 512-byte blocks, resulting in 24 ECC bits. For MLC devices, however, where a failure of a single cell results in the failure of two consecutive bits, a different ECC has to be used. For these devices, Reed-Solomon codes offer a good alternative. Because of the computational complexity of the Reed-Solomon code, the Bose-Chaudhuri-Hocquenghem (BCH) algorithm is becoming more popular since it can be implemented in hardware [69].

However, while resilience techniques within each physical device can protect against small amounts of data corruption, uncorrectable errors do still occur [56, 51]. In addition, it is possible for the storage device as a whole to fail. For example, in rotating disks, mechanical failure cannot be excluded. Moreover, storage devices are commonly grouped into a larger, logical device to obtain higher capacities and higher bandwidth, increasing the probability that the combined device will suffer data loss due to the failure of one of its components.

Because of the nature of persistent storage, persistent data loss typically has a higher cost. In order to reduce the probability of persistent data loss, storage devices can be grouped into a redundant array of independent disks (RAID) [49].

A number of RAID levels, differing in how the logical device is divided and replicated among the physical devices, have become standardized. A few examples are described below.

RAID0 Data is spread over multiple disks without adding any redundancy. A single failure results in data loss.

RAID1 Data is replicated on one (or more) additional drives. Up to $n - 1$ (assuming n devices) can fail without resulting in data loss.

RAID2 Data is protected by using an ECC. For RAID2, each byte is spread over different devices, and a hamming code is applied to corresponding bits. The resulting ECC bits are stored on a dedicated device.

RAID3 and RAID4 These are like RAID2, but instead of on a bit level, RAID3 and RAID4 use byte granularity for error correction. XOR is used as error correction code. RAID3 and RAID4 differ in how the data is partitioned (block versus stripe).

RAID5 This is like RAID4, but the parity data is spread over multiple devices.

RAID6 This is like RAID5 but with two parity blocks. Therefore, RAID6 can tolerate two failed physical devices.

When a failure is detected, the failed device needs to be replaced, after which the array will regenerate the data of the failed device and store it on the new device. This is referred to as rebuilding the array. Because of the difference in increases in bandwidth and capacity for storage devices, a rebuild can take a fairly long time (hours). During this time, all RAID levels except for RAID6 are vulnerable as they offer no protection against further failures. As is the case with memory, many RAID arrays employ a form of scrubbing to detect failure before errors can accumulate.

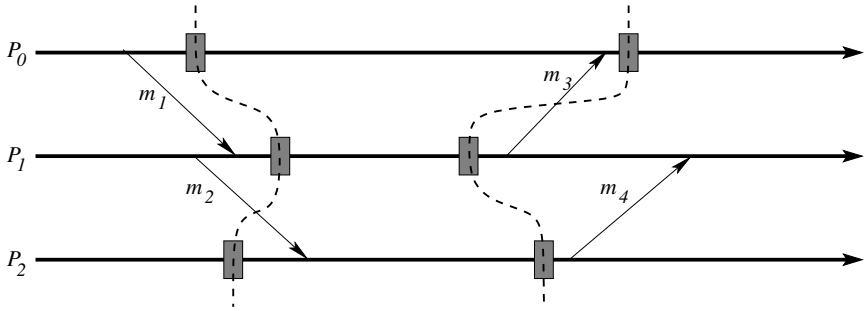

Figure 1: Consistent vs. inconsistent checkpoints

3 System Software Features for Resilience

In this section, we discuss fault resilience techniques used in various system software libraries, including communication libraries, task-based models, and large data models. We start by describing checkpointing, which is used in many programming models, then describe techniques used for specific programming models.

3.1 Checkpointing

Checkpointing is a fault-tolerance mechanism where the state of a system running an application is recorded in a global *checkpoint* so that, in the event of a fault, the system state can be rolled back to the checkpoint and allowed to continue from that point, rather restarting the application from the beginning. System-level checkpointing is popular because it provides fault-tolerance to an application without requiring the application to be modified.

A global checkpoint of a distributed application consists of a set of checkpoints of individual processes. Figure 1 shows three processes (represented by horizontal lines) and two global checkpoints (represented as dotted lines) consisting of individual checkpoints (represented as rectangles). The global checkpoint on the left is *consistent* because it captures a global state that may have occurred during the computation. Note that while the global state records message m_2 being sent but not received, this could have occurred during the computation if the message was sent and was still in transit over the network. The second global checkpoint is *inconsistent* because it captures message m_3 as being received but not

sent, which could never have occurred during the computation. Messages such as m_3 are known as *orphan* messages.

Checkpointing protocols use various methods either to find a consistent global checkpoint or to allow applications to roll back to inconsistent global checkpoints by logging messages.

3.1.1 System-Level Checkpointing

In [26], Elnozahy et al. classify checkpoint recovery protocols into *uncoordinated*, *coordinated*, *communication-induced* and *log-based* protocols.

In uncoordinated checkpoint protocols, processes independently take checkpoints without coordinating with other processes. By not requiring processes to coordinate before taking checkpoints, a process can decide to take checkpoints when the size of its state is small, thereby reducing the size of the checkpoint [68]. Also because processes are not forced to take checkpoints at the same time, checkpoints taken by different processes can be spread out over time thereby spreading out the load on the filesystem [48]. When a failure occurs, a consistent global checkpoint is found by analyzing the dependency information recorded with individual checkpoints [15]. Note, however, that because checkpoints are taken in an uncoordinated manner, orphan messages are possible and may result in checkpoints taken at some individual process that are not part of any consistent global checkpoint. In which case that process will need to roll back to a previous checkpoint. Rolling back that process can produce more orphan messages requiring other processes to roll back further. This is known as cascading rollbacks or the domino-effect [53] and can result in the application rolling back to the its initial state because no consistent global checkpoint exists.

Coordinated checkpoint protocols [20][41] do not suffer from cascading rollbacks because the protocol guarantees that *every* individual checkpoint taken is part of a consistent global checkpoint. Because of this feature, only the last global checkpoint needs to be stored. Once a global checkpoint has been committed to stable storage, the previous checkpoint can be deleted. This also eliminates the need to search for a consistent checkpoint during the restart protocol. Coordinated checkpoints can be *blocking* or *nonblocking*. In a blocking

protocol, all communication is halted, and communication channels are flushed while the checkpointing protocol executes [62]. This ensures that there are no orphan messages. In a nonblocking protocol, the application is allowed to continue communicating concurrently with the checkpointing protocol. Nonblocking protocols use markers sent either as separate messages or by piggybacking them on application messages. When a process takes a checkpoint, it sends a marker to every other process. Upon receiving a marker, the receiver takes a checkpoint if it hasn't already. If the markers are sent before any application messages or if the marker is piggybacked and therefore processed before the application message is processed, then orphan messages are avoided.

In communication-induced checkpointing [34][54][45], processes independently decide when to take a checkpoint, similar to uncoordinated checkpoints, but also take *forced* checkpoints. Processes keep track of dependency information of messages by using Lamport's *happen-before* relation. This information is piggybacked on all application messages. When a process receives a message, if, based on its dependency information and the information in the received message, it determines that processing the application message would result in an orphan message, then the process takes a checkpoint before processing the application message.

Log-based protocols [38][37][61][30] require that processes be *piecewise deterministic*, meaning that given the same input, the process will behave exactly the same every time it is executed. Furthermore, information on any nondeterministic events, such as the contents and order of incoming messages, can be recorded and used to replay the event. In *pessimistic* logging, event information is stored to stable storage immediately. While this can be expensive during failure-free execution, only the failed process needs to be rolled back, since all messages it received since its last checkpoint are recorded and can be played back. In *optimistic* logging, event information is saved to stable storage periodically, thus reducing the overhead during failure-free execution. However, the recovery protocol is complicated because the protocol needs to use dependency information from the event logs to determine which checkpoints form a consistent global state and which processes need to be

rolled back.

3.1.2 Complete vs. Incremental Checkpoints

A complete system-level checkpoint saves the entire address space of a process. One way to reduce the size of a checkpoint is to use incremental checkpointing. In incremental checkpointing unmodified portions of a process's address space are not included in the checkpoint image. In order to determine which parts of the address space have been modified, some methods use a hash over blocks of memory [2]; other approaches use a virtual memory system [35][66].

Page-based methods use two approaches. In one approach, the checkpointing system creates an interrupt handler for page faults. After a checkpoint is taken, all of the process's pages are set to read-only. When the application tries to modify a page, a page-fault is raised and the checkpointing system will mark that page as having been modified. This approach has the advantage of not requiring modification of the operating system kernel; however, it does have the overhead of a page fault the first time the process writes to a page after a checkpoint. Another approach is to patch the kernel and keep track of the dirty bit in each pages page table entry in a way that allows the checkpointing system to clear the bits on a checkpoint without interfering with the kernel. This has the benefit of not forcing page faults, but it does require kernel modification.

Incremental checkpoints are typically used with periodic complete checkpoints. The higher the ratio of incremental to complete checkpoints, the higher the restart overhead because the current state of the process must be reconstructed from the last complete checkpoint and every subsequent incremental checkpoint.

3.2 Fault Management Enhancements to Parallel Programming Models

While checkpointing has been the traditional method of providing fault tolerance and is transparent to the application, nontransparent mechanisms are becoming popular. Nontransparent mechanisms allow the application to control how faults should be handled.

Programming models must provide features that allow the application to become aware of failures and to isolate or mitigate the effects of failures. We describe various fault-tolerance techniques appropriate to different programming models.

3.2.1 Process-Driven Techniques

In [27], Fagg and Dongarra proposed modifications to the MPI-2 API to allow processes to handle process failures. They implemented the standard with their modification in FT-MPI. An important issue to address when adding fault-tolerance features to the MPI standard is how to handle communicators that contain failed processes. A communication operation will return an error if a process tries to communicate with a failed process. The process must then repair the communicator before it can proceed. FT-MPI provides four modes in which a communicator can be repaired: SHRINK, BLANK, REBUILD, and ABORT. In the SHRINK mode, the failed processes are removed from the communicator. When the communicator is repaired in this way, the size of the communicator changes and possibly the ranks of some processes. In the BLANK mode, the repaired communicator essentially contains holes where the failed processes had been, so that the size of the communicator and the ranks of the processes don't change, but sending to or receiving from a failed process results in an invalid-rank error. In the REBUILD mode, new processes are created and replace the failed processes. A special return value from MPI_Init tells a process whether it is an original process, or it has been started to replace a failed process. In the ABORT mode, the job is aborted when a process fails.

Another important issue is the behavior of collective communication operations when processes fail. In FT-MPI, collective communication operations are guaranteed to either succeed at every process or to fail at every process. In FT-MPI, information about failed processes is stored on an attribute attached to a communicator, which a process can query. It is not clear from the literature how FT-MPI supports MPI one-sided or file operations.

The MPI Forum is working on defining new semantics and API functions for MPI-3 to allow applications to handle the failure of processes. The current proposal (when this

chapter was written) is similar to the BLANK mode of FT-MPI in that the failure of a process does not change the size of a communicator or the ranks of any processes. While FT-MPI requires a process to repair a communicator as soon as a failure is detected, the MPI-3 proposal does not have this requirement. The failure of some process will not affect the ability of live processes to communicate.

Because of this approach, wildcard receives (i.e., receive operations that specify MPI_ANY_SOURCE as the sender) must be addressed differently. If a process posts a wildcard receive and some process fails, the MPI library does not know whether the user intended the wildcard receive to match a message from the failed process. If the receive was intended to match a message from the failed process, then the process might hang waiting for a message that will never come, in which case the library should raise an error for that receive and cancel it. However, if a message sent from another process can match the wildcard receive, then raising an error for that receive would not be appropriate. In the current proposal, a process must *recognize* all failed processes in a communicator before it can wait on a wildcard receive. So, if a communicator contains an unrecognized failed process, the MPI library will return an error whenever a process waits on a wildcard receive, for example, through a blocking receive or an MPI_Wait call, but the receive will not be canceled. This approach will allow an application to check whether the failed processes were the intended senders for the wildcard receive.

The proposal requires that collective communication operations not hang because of failed processes, but it does not require that the operation uniformly complete either successfully or with an error. Hence, the operation may return successfully at one process, while returning with an error at another. The MPI_Comm_validate function is provided to allow the MPI implementation to restructure the communication pattern of collective operations to bypass failed processes. This function also returns a group containing the failed processes that can be used by the process to determine whether any processes have failed since the last time the function was called. If no failures occurred since the last time the function was called, then the process can be sure that all collective operations performed

during that time succeeded everywhere. Similar validation functions are provided for MPI window objects for one-sided operations and MPI file objects to allow an application to determine whether the preceding operations completed successfully.

Process failures can be queried for communicator, window, and file objects. The query functions return MPI group objects containing the failed processes. Group objects provide a scalable abstraction for describing failed processes (compared to, e.g., an array of integers).

Another problem for exascale computing is silent data corruption (SDC). As the number of components increases, the probability of bit flips that cannot be corrected with ECC or even detected with CRC increases. SDC can result in an application returning invalid results without being detected. To address this problem, RedMPI [28] replicates processes and compares results to detect SDC. When the application sends a message, each replica sends a message to its corresponding receiver replica. In addition a hash of the message is sent to the other receiver replicas so that each receiver can verify that it received the message correctly and that if SDC occurred at the sender, it did not affect the contents of the message. Using replicas also provides tolerance to process failure. If a process fails, a replica can take over for the failed process.

3.2.2 Data-Driven Techniques

Global Arrays [46] is a parallel programming model that provides indexed array-like global access to data distributed across the machine using *put*, *get* and *accumulate* operations. In [3], Ali et al. reduce the overhead of recovering from a failure by using redundant data. The idea is to maintain two copies of the distributed array structure but distribute them differently so that both copies of a chunk of the array aren't located on the same node. In this way if a process fails, there is a copy of every chunk that was stored on that process on one of the remaining processes. The recovery process consists of starting a new process to replace the failed one, and restoring the copies of the array stored at that process. Furthermore, because the state of the array is preserved, the nonfailed processes can continue running during the recovery process. This approach significantly reduces the

recovery time compared with that of checkpointing and rollback.

3.2.3 Task-Driven Techniques

Charm++ [39] is a C++-based. object-oriented parallel programming system. In this system, work is performed by tasks, or *chares*, which can be migrated by the Charm++ runtime to other nodes for load balancing. Charm++ provides fault tolerance through checkpointing and allows the application to mark which data in the chare to include in the checkpoint image, thus reducing the amount of data to be checkpointed. There are two modes for checkpointing [40]. In the first mode, all threads collectively call a checkpointing function periodically. In this mode, if a node fails, the entire application is started from the last checkpoint. In order to reduce the overhead of restarting the entire application, checkpoints can be saved to memory or local disk as well as to the parallel filesystem. Thus, nonfailed processes can restart from local images, greatly reducing the load on the parallel filesystem.

The other checkpointing mode uses message logging so that if a process fails, only that process needs to be restarted. When a process fails, it is restarted from its last checkpoint on a new node. Then the process will replay the logged messages in the original order. When a node fails, the restarted processes need not be restarted on the same node, but can be distributed among other nodes to balance the load of the restart protocol.

CiLK [16] is a thread-based parallel programming system using C. CiLK-NOW[17] was an implementation of CiLK over a network of workstations. The CiLK-NOW implementation provided checkpointing of the entire application if critical processes failed but also was able to restart individual threads if they crashed or the nodes they were running on failed.

4 Application or Domain-Specific Fault Tolerance Techniques

While hardware and systems software techniques for transparent fault tolerance are convenient for users, such techniques often impact the overall performance, system cost, or both. Several computational science domains have been investigating techniques for application or

domain-specific models for fault tolerance that utilize information about the characteristics of the application (or the domain) to design specific algorithms that try to minimize such performance loss or system cost. These techniques, however, are not completely transparent to the domain scientists.

In this section, we discuss two forms of fault tolerance techniques. The first form is specific to numerical libraries, where researchers have investigated approaches in which characteristics of the mathematical computations can be used to achieve reliability in the case of node failures (discussed in Section 4.1). The second form is fault resilience techniques utilized directly in end applications (discussed in Section 4.2); we describe techniques used in two applications: mpiBLAST (computational biology) and Green's function Monte Carlo (nuclear physics).

4.1 Algorithmic Resilience in Math Libraries

The fundamental idea of algorithm-based fault tolerance (ABFT) is to utilize domain knowledge of the computation to deal with some errors. While the concept is generic, a large amount of work has been done for algorithmic resilience in matrix computations. For instance, Anfinson and Luk [36] and Huang and Abraham [7] showed that it is possible to encode a hash of the matrix data being computed on, such that if a process fails, data corresponding to this process can be recomputed based on this hash without having to restart the entire application. This technique is applicable to a large number of matrix operations including addition, multiplication, scalar product, LU-decomposition, and transposition.

This technique was further developed by Chen and Dongarra to tolerate fail-stop failures that occurred during the execution of high-performance computing (HPC) applications [21, 22] (discussed in Section 4.1.1). The idea of ABFT is to encode the original matrices by using real number codes to establish a checksum type of relationship between data, and then redesign algorithms to operate on the encoded matrices in order to maintain the checksum relationship during the execution. Wang et al. [67] enhanced Chen and Dongarra's work to allow for nonstop hot-replacement based fault recovery techniques (discussed in

Section 4.1.2).

4.1.1 Fail-Stop Fault Recovery

Assume there will be a single process failure. Since it's hard to locate which process will fail before the failure actually occurs, a fault-tolerant scheme should be able to recover the data on any process. In the conventional ABFT method, it is assumed that at any time during the computation the data D_i on the ith process P_i satisfies

$$D_1 + D_2 + \cdots + D_n = E, \tag{1}$$

where n is the total number of processes and E is data on the encoding process. Thus, the lost data on any failed process can be recovered from Eq. (1). Suppose P_i fails. Then the lost data D_i on P_i can be reconstructed by

$$D_i = E - (D_1 + \cdots + D_{i-1} + D_{i+1} + \cdots + D_n). \tag{2}$$

In practice, this kind of special relationship is by no means natural. However, one can design applications to maintain such a special checksum relationship throughout the computation, and this is one purpose of ABFT research.

4.1.2 Nonstop Hot-Replacement-Based Fault Recovery

For the simplicity of presentation, we assume there will be only one process failure. However, it is straightforward to extend the results here to multiple failures by using multilevel redundancy or regenerating the encoded data. Suppose that at any time during the computation, the data D_i on process P_i satisfies

$$D_1 + D_2 + \cdots + D_n = E. \tag{3}$$

If the ith process fails during the execution, we replace it with the encoding process E and continue the execution instead of stopping all the processes to recover the lost data D_i. Note that this kind of transformation can be effective only when there is an encoding relationship among the data.

From the global view, the original data is

$$D = (D_1 \cdots D_{i-1} D_i D_{i+1} \cdots D_n), \tag{4}$$

and the transformed data (after replacement) is

$$D' = (D_1 \cdots D_{i-1} E D_{i+1} \cdots D_n). \tag{5}$$

We can establish a relationship between the transformed data and the original data as

$$D' = D \times T, \tag{6}$$

and T can be represented as an $n \times n$ matrix in the form

$$T = \begin{pmatrix} 1 & & & 1 & & & \\ & \ddots & & \vdots & & & \\ & & 1 & 1 & & & \\ & & & 1 & & & \\ & & & 1 & 1 & & \\ & & & \vdots & & \ddots & \\ & & & 1 & & & 1 \end{pmatrix}, \tag{7}$$

where the elements omitted in the diagonal and the ith column are all 1 and the other elements omitted are 0. We can see that T is a nonsingular matrix. If operations on the data are linear transformations (e.g., matrix operations such as decomposition), the relationship $D' = D \times T$ will always be kept. At the end of computation, the original correct solution based on D can be recomputed through the intermediate solution based on D'. And this recomputation is actually a transformation related to T.

One can see that the encoding relationship $D' = D \times T$ cannot be maintained under all HPC applications. However, for a class of them, including matrix computations involving linear transformations (e.g., matrix decomposition, matrix-matrix multiplication, and scalar product), the encoding relationship can be maintained.

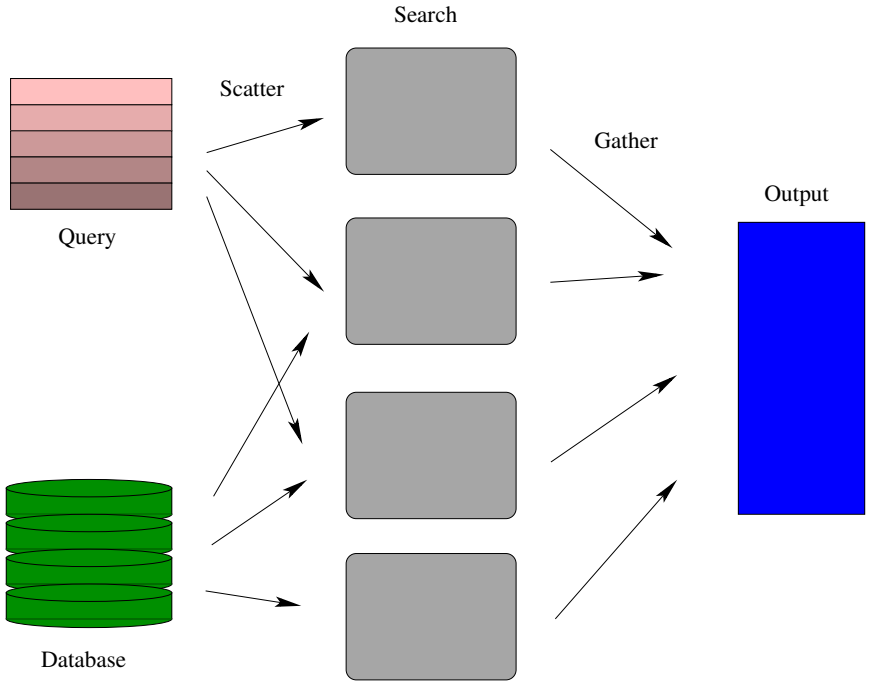

Figure 2: High-level algorithm of mpiBLAST

4.2 Application-Level Checkpointing

In this section, we present case studies for two application-specific fault tolerance techniques: sequence alignment with mpiBLAST and Green's function Monte Carlo.

4.2.1 Sequence Alignment with mpiBLAST

With the advent of rapid DNA sequencing, the amount of genetic sequence data available to researchers has increased exponentially [6]. The GenBank database, a comprehensive database that contains genetic sequence data for more than 260,000 named organisms, has exhibited exponential growth since its inception over 25 years ago [14]. This information is available for researchers to search new sequences against and infer homologous relationships between sequences or organisms. This helps in a wide range of projects, from assembling the Tree of Life [25] to pathogen detection [29] and metagenomics [32].

Unfortunately, the exponential growth of sequence databases necessitates faster search algorithms to sustain reasonable search times. The Basic Local Alignment Search Tool

(BLAST), which is the de facto standard for sequence searching, uses heuristics to prune the search space and decrease search time with an accepted loss in accuracy [4, 5]. BLAST is parallelized by mpiBLAST using several techniques, including database fragmentation, query segmentation [23], parallel input-output [42], and advanced scheduling [65]. As shown in Figure 2, mpiBLAST uses a *master-worker* model and performs a *scatter-search-gather-output* execution flow. During the scatter, the master splits the database and query into multiple pieces and distributes them among worker nodes. Each worker then searches the query segment against the database fragment that it was assigned. The results are gathered by the master, formatted, and output to the user. Depending on the size of the query and the database, such output can be large. Thus, for environments with limited I/O capabilities, such as distributed systems, the output step can cause significant overheads.

One of the characteristics of sequence alignment with mpiBLAST is that the computation and output associated with each query sequence is independent. Thus, splitting the query sequences into multiple independent executions and combining the output in a post-processing step would not affect the overall outcome of the application. This behavior is true even with the database itself. That is, for each query sequence, mpiBLAST searches for the "best matching" sequence in the database. Thus, as long as these best matching sequences are available, even deleting some of the other sequences in the database does not affect the overall outcome. In order to take advantage of such application characteristics, the ParaMEDIC framework was developed [10, 11, 13, 12, 9]. Though initially designed for optimizing the I/O requirements of mpiBLAST, the ParaMEDIC framework allows the application to work through system faults. Specifically, if a part of the computation fails because of a system fault, that part of the computation is discarded and recomputed. This approach allows the overall final output of the application to not change based on intermediate faults.

4.2.2 Green's Function Monte Carlo

The quantum Monte Carlo code developed by Steven C. Pieper and coworkers at Argonne National Laboratory [50, 43] uses the Green's function Monte Carlo (GFMC) method. The GFMC code is part of the SciDAC Universal Nuclear Energy Density Functional (UNEDF) effort to understand the physics of light nuclei. The fundamental computation involved in quantum Monte Carlo is a $3N$-dimensional integral—where N is the number of nucleons—evaluated by using Monte Carlo methods. In the first step of the calculation, variational Monte Carlo, a single integration is performed to get an approximation to the ground-state eigenvector. In the second step, GFMC uses imaginary-time propagation to refine the ground-state solution. Each step in the propagation involves a new $3N$-dimensional integral, and the entire calculation corresponds to an integral of more than 10,000 dimensions. GFMC is parallelized by using a master-worker programming model called ADLB [43].

GFMC initially relied on system-level checkpointing, as discussed in Section 3.1, for fault tolerance. However, given that GFMC is memory intensive, the amount of I/O required for each checkpointing operation was tremendous and was growing rapidly with the problem size. To address this concern, GFMC developed its own application-specific checkpointing approach that utilizes application knowledge to write only a small part of critical data to the disk, instead of the entire memory space of the application.

To explain this approach, we first describe the overall parallelism structure of GFMC. Specifically, in GFMC, the application processes are distributed into three segments:

1. **Master:** This process reads input, makes initial distribution of work, receives results (energy packets), averages them, and writes averaged results to disk and stdout.

2. **Walker nodes:** These each manage a group of GFMC configurations. They control the propagation of these configurations but the actual work is done on worker nodes by using ADLB. The walker nodes do branching, which kills or replicates configurations, and do load balancing, which redistributes configurations to other walker nodes to keep the number of configurations equal on the walkers.

3. **Worker nodes:** These accept work packages from ADLB and return results. They have no long-term data.

Most of the checkpointing in GFMC is handled by the walker nodes. The walker nodes receive all the starting configurations from the master. They then go into a loop doing propagation time steps. Each configuration is put into ADLB as a propagation work package. The walker processes loop, getting propagation answers and possibly accepting propagation work packages. If the time step is not the next branching step, the work package is put back into ADLB for another time step. Every few time steps, branching is done that can increase or decrease the total number of configurations.

The walker nodes have the current status of all the configurations. This is the only information needed to resume the calculation in case of a failure. Thus, every few time steps, the walker nodes coordinate and dump the current status of the configurations to a checkpoint file. In order to avoid failures while the checkpoint is ongoing, multiple checkpoint files are maintained.

5 Summary

With system sizes growing rapidly, faults are increasingly becoming the norm, rather than the exception. To handle such faults, researchers are working on various techniques, some of which are transparent to the end users, while others are not. But each class of fault tolerance techniques has its own pros and cons.

In this chapter, we described various fault tolerance techniques, broadly classified into three categories. The first category deals with hardware fault tolerance, which is fully transparent to the user. These are handled by hardware redundancy and other related techniques. The second category deals with resilient systems software, which is not transparent to the software stack on the machine but is handled mostly by expert users developing systems software stacks such as MPI or operating systems. Therefore, in some sense, it is still hidden from computational domain scientists developing end applications. The third

category deals with application or domain-specific fault tolerance techniques that utilize application-specific knowledge to achieve fault tolerance. This category requires changes to the applications and thus is, obviously, not transparent to the end user.

References

[1] Cray XT5 Compute Blade. `http://wwwjp.cray.com/downloads/CrayXT5Blade.pdf`. [Online; accessed 20-November-2011].

[2] S. Agarwal, R. Garg, M. S. Gupta, and J. E. Moreira. Adaptive incremental checkpointing for massively parallel systems. In *Proceedings of the 18th annual International Conference on Supercomputing*, ICS '04, pages 277–286, New York, 2004. ACM.

[3] N. Ali, S. Krishnamoorthy, N. Govind, and B. Palmer. A redundant communication approach to scalable fault tolerance in PGAS programming models. In *Parallel, Distributed and Network-Based Processing (PDP), 2011 19th Euromicro International Conference on*, pages 24–31, February 2011.

[4] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman. Basic local alignment search tool. *J. Mol. Biol.*, 215(3):403–410, October 1990.

[5] S. F. Altschul, T. L. Madden, A. A. Schaffer, J. Zhang, Z. Zhang, W. Miller, and D. J. Lipman. Gapped BLAST and PSI–BLAST: A new generation of protein database search programs. *Nucleic Acids Res.*, 25:3389–3402, 1997.

[6] S. F. Altshul, M. S. Boguski, W. Gish, and J. C. Wootton. Issues in searching molecular sequence databases. *Nat. Genet.*, 6(2):119–29, 1994.

[7] J. Anfinson and F. T. Luk. A linear algebraic model of algorithm-based fault tolerance. *IEEE Trans. Comput.*, 37:1599–1604, 1988.

[8] JL Autran, P. Roche, S. Sauze, G. Gasiot, D. Munteanu, P. Loaiza, M. Zampaolo, and J. Borel. Altitude and underground real-time SER characterization of CMOS 65 nm SRAM. *IEEE Trans. Nuclear Science*, 56(4):2258–2266, 2009.

[9] P. Balaji, W. Feng, J. Archuleta, and H. Lin. ParaMEDIC: Parallel Metadata Environment for Distributed I/O and Computing. In *Proceedings of the IEEE/ACM International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, Reno, Nevada, November 2007.

[10] P. Balaji, W. Feng, J. Archuleta, H. Lin, R. Kettimuttu, R. Thakur, and X. Ma. Semantics-based distributed I/O for mpiBLAST. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, Salt Lake City, Utah, February 2008.

[11] P. Balaji, W. Feng, and H. Lin. Semantics-based distributed I/O with the ParaMEDIC framework. In *Proceedings of the ACM/IEEE International Symposium on High Performance Distributed Computing (HPDC)*, Boston, Massachusetts, June 2008.

[12] P. Balaji, W. Feng, H. Lin, J. Archuleta, S. Matsuoka, A. Warren, J. Setubal, E. Lusk, R. Thakur, I. Foster, D. S. Katz, S. Jha, K. Shinpaugh, S. Coghlan, and D. Reed. Distributed I/O with ParaMEDIC: Experiences with a worldwide supercomputer. In *Proceedings of the International Supercomputing Conference (ISC)*, Dresden, Germany, June 2008.

[13] P. Balaji, W. Feng, H. Lin, J. Archuleta, S. Matsuoka, A. Warren, J. Setubal, E. Lusk, R. Thakur, I. Foster, D. S. Katz, S. Jha, K. Shinpaugh, S. Coghlan, and D. Reed. Global-scale distributed I/O with ParaMEDIC. *International Journal of Concurrency and Computation: Practice and Experience (CCPE)*, 22(16):2266–2281, 2010.

[14] D. A. Benson, I. Karsch-Mizrachi, D. J. Lipman, J. Ostell, and D. L. Wheeler. GenBank. *Nucleic Acids Res*, 36(Database issue), January 2008.

[15] B. Bhargava and S.-R. Lian. Independent checkpointing and concurrent rollback for recovery in distributed systems-an optimistic approach. In *Proceedings of the Seventh Symposium on Reliable Distributed Systems.*, pages 3–12, October 1988.

[16] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. CiLK: An efficient multithreaded runtime system. *SIGPLAN Not.*, 30:207–216, August 1995.

[17] R. D. Blumofe and P. A. Lisiecki. Adaptive and reliable parallel computing on networks of workstations. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 10–10, Berkeley, California, 1997. USENIX Association.

[18] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and W. K. Su. Myrinet: A gigabit-per-second local area network. *IEEE Micro '95*, 15:29–36.

[19] L. Borucki, G. Schindlbeck, and C. Slayman. Comparison of accelerated DRAM soft error rates measured at component and system level. In *Proceedings of the International Reliability Physics Symposium, 2008.*, pages 482–487. IEEE, 2008.

[20] K. M. Chandy and L. Lamport. Distributed snapshots: determining global states of distributed systems. *ACM Trans. Comput. Syst.*, 3:63–75, February 1985.

[21] Z. Chen and J. Dongarra. Algorithm-based checkpoint-free fault tolerance for parallel matrix computations on volatile resources. In *Proceedings of the 20st IEEE International Parallel and Distributed Processing Symposium*, page 76, 2006.

[22] Z. Chen and J. Dongarra. Algorithm-based fault tolerance for fail-stop failures. *IEEE Transactions on Parallel and Distributed Systems*, 19(12), December 2008.

[23] A. E. Darling, L. Carey, and W. Feng. The Design, Implementation, and Evaluation of mpiBLAST. In *ClusterWorld Conference & Expo and the 4th International Conference on Linux Cluster: The HPC Revolution*, 2003.

[24] T. J. Dell. A white paper on the benefits of Chipkill-correct ECC for PC server main memory. Technical report, IBM Microelectronics Division, November 1997.

[25] A. C. Driskell, C. Ané, J. G. Burleigh, M. M. McMahon, B. C. O'Meara, and M. J. Sanderson. Prospects for building the tree of life from large sequence databases. *Science*, 306(5699):1172–1174, November 2004.

[26] E. N. (Mootaz) Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Comput. Surv.*, 34:375–408, September 2002.

[27] G. E. Fagg and J. Dongarra. FT-MPI: Fault tolerant MPI, supporting dynamic applications in a dynamic world. In *Proceedings of the 7th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 346–353, London, UK, 2000. Springer-Verlag.

[28] D. Fiala. Detection and correction of silent data corruption for large-scale high-performance computing. In *International Symposium on Parallel and Distributed Processing Workshops and Ph.D. Forum (IPDPSW), 2011*, pages 2069–2072, May 2011.

[29] J. D. Gans, W. Feng, and M. Wolinsky. Whole genome, physics-based sequence alignment for pathogen signature design. In *12th SIAM Conference on Parallel Processing for Scientific Computing*, San Francisco, California, February 2006. (electronic version unavailable).

[30] A. Guermouche, T. Ropars, E. Brunet, M. Snir, and F. Cappello. Uncoordinated checkpointing without domino effect for send-deterministic message passing applications. In *Proceedings of the International Parallel and Distributed Processing Symposium*, 2011.

[31] R. W. Hamming. Error detecting and error correcting codes. *Bell System Technical Journal*, 29(2):147–160, 1950.

[32] S. L. Havre, B.-J. Webb-Robertson, A. Shah, C. Posse, B. Gopalan, and F. J. Brockma. Bioinformatic insights from metagenomics through visualization. *Proceedings of the Computational Systems Bioinformatics Conference*, pages 341–350, 2005.

[33] P. Hazucha and C. Svensson. Impact of CMOS technology scaling on the atmospheric neutron soft error rate. *Nuclear Science, IEEE Transactions on*, 47(6):2586–2594, 2000.

[34] J.-M. Hélary, A. Mostefaoui, and M. Raynal. Preventing useless checkpoints in distributed computations. In *Proceedings of the 16th Symposium on Reliable Distributed Systems*, SRDS '97, Washington, D.C., 1997. IEEE Computer Society.

[35] J. Heo, S. Yi, Y. Cho, J. Hong, and S. Y. Shin. Space-efficient page-level incremental checkpointing. In *Proceedings of the 2005 ACM Symposium on Applied computing*, SAC '05, pages 1558–1562, New York, 2005. ACM.

[36] K. H. Huang and J. A. Abraham. Algorithm-based fault tolerance for matrix operations. *IEEE Transactions on Computers*, C-33(6):518–528, June 1984.

[37] D. B. Johnson. *Distributed system fault tolerance using message logging and checkpointing*. PhD thesis, Rice University, Department of Computer Science, 1989.

[38] D. B. Johnson and W. Zwaenepoel. Sender-based message logging. In *Digest of Papers, FTCS-17, Seventeenth Annual International Symposium on Fault-Tolerant Computing*, pages 14–19, 1987.

[39] L. V. Kalé and S. Krishnan. CHARM++: A portable concurrent object oriented system based on C++. *SIGPLAN Not.*, 28:91–108, October 1993.

[40] L. V. Kalé and G. Zheng. Charm++ and AMPI: Adaptive runtime strategies via migratable objects. In M. Parashar, editor, *Advanced Computational Infrastructures for Parallel and Distributed Applications*, pages 265–282. Wiley-Interscience, 2009.

[41] T. H. Lai and T. H. Yang. On distributed snapshots. *Inf. Process. Lett.*, 25:153–158, May 1987.

[42] H. Lin, X. Ma, P. Chandramohan, A. Geist, and N. Samatova. Efficient data access for parallel BLAST. In *IPDPS*, 2005.

[43] E. Lusk, S. Pieper, and R. Butler. More scalability, less pain. *SciDAC Review*, 17:30–37, 2010.

[44] D. Milojicic, A. Messer, J. Shau, G. Fu, and A. Munoz. Increasing relevance of memory hardware errors: a case for recoverable programming models. In *Proceedings of the 9th workshop on ACM SIGOPS European workshop: beyond the PC: new challenges for the operating system*, pages 97–102. ACM, 2000.

[45] R. H. B. Netzer and J. Xu. Necessary and sufficient conditions for consistent global snapshots. *IEEE Trans. Parallel Distrib. Syst.*, 6:165–169, February 1995.

[46] J. Nieplocha, B. Palmer, V. Tipparaju, M. Krishnan, H. Trease, and E. Aprà. Advances, applications and performance of the global arrays shared memory programming toolkit. *Int. J. High Perform. Comput. Appl.*, 20:203–231, May 2006.

[47] N. Oh, P. P. Shirvani, and E. J. McCluskey. Error detection by duplicated instructions in super-scalar processors. *IEEE Transactions on Reliability*, 51(1):63–75, March 2002.

[48] R. A. Oldfield, S. Arunagiri, P. J. Teller, S. Seelam, M. R. Varela, R. Riesen, and P. C. Roth. Modeling the impact of checkpoints on next-generation systems. In *Proceedings of the 24th IEEE Conference on Mass Storage Systems and Technologies*, pages 30–46, Washington, D.C., 2007. IEEE Computer Society.

[49] D. A. Patterson, G. Gibson, and R. H. Katz. A case for redundant arrays of inexpensive disks (RAID). *AXM SIGMOD Record*, 17(3):109–116, June 1988.

[50] S. C. Pieper and R. B. Wiringa. Quantum Monte Carlo calculations of light nuclei. *Annual Review of Nuclear and Particle Science*, 51(1):53–90, 2001.

[51] E. Pinheiro, W. D. Weber, and L. A. Barroso. Failure trends in a large disk drive population. In *Proceedings of the 5th USENIX conference on File and Storage Technologies*, pages 2–2, 2007.

[52] Moinuddin K. Qureshi, Onur Mutlu, and Yale N. Patt. Microarchitecture-based introspection: A technique for transient-fault tolerance in microprocessors. *International Conference on Dependable Systems and Networks*, 0:434–443, 2005.

[53] B. Randell. System structure for software fault tolerance. *IEEE Trans. Software Engineering*, 1:220–232, 1975.

[54] D. L. Russell. State restoration in systems of communicating processes. *IEEE Trans. Software Engineering*, SE-6(2):183–194, March 1980.

[55] A. M. Saleh, J. J. Serrano, and J. H. Patel. Reliability of scrubbing recovery-techniques for memory systems. *IEEE Trans. Reliability*, 39(1):114–122, April 1990.

[56] B. Schroeder and G. A. Gibson. Disk failures in the real world: What does an MTTF of 1,000,000 hours mean to you? In *Proceedings of the 5th USENIX conference on File and Storage Technologies*, page 1. USENIX Association, 2007.

[57] B. Schroeder and G. A. Gibson. A large-scale study of failures in high-performance computing systems. *IEEE Trans. Dependable and Secure Computing*, 7(4):337–351, 2010.

[58] B. Schroeder, E. Pinheiro, and W. D. Weber. DRAM errors in the wild: A large-scale field study. In *Proceedings of the Eleventh International Joint Conference on Measurement and Modeling of Computer Systems*, pages 193–204. ACM, 2009.

[59] C. Slayman. Whitepaper on soft errors in modern memory technology. Technical report, OPS A La Carte, 2010.

[60] J. Stone and C. Partridge. When the CRC and TCP checksum disagree. *ACM SIGCOMM Computer Communication Review*, 30(4):309–319, October 2000.

[61] R. Strom and S. Yemini. Optimistic recovery in distributed systems. *ACM Trans. Comput. Syst.*, 3:204–226, August 1985.

[62] Y. Tamir and C. H. Séquin. Error recovery in multicomputers using global checkpoints. In *Proceedings of the International Conference on Parallel Processing*, pages 32–41, 1984.

[63] IBM Blue Gene Team. Overview of the IBM Blue Gene/P project. *IBM Journal of Research and Development*, 52(1/2):199–220, 2008.

[64] Mellanox Technologies. InfiniBand and TCP in the Data-Center.

[65] O. Thorsen, B. Smith, C. P. Sosa, K. Jiang, H. Lin, A. Peters, and W. Feng. Parallel genomic sequence-search on a massively parallel system. In *ACM International Conference on Computing Frontiers*, Ischia, Italy, May 2007.

[66] C. Wang, F. Mueller, C. Engelmann, and S. L. Scott. Hybrid full/incremental checkpoint/restart for MPI jobs in HPC environments. In *Proceedings of the International Conference on Parallel and Distributed Systems*, December 2011.

[67] R. Wang, E. Yao, P. Balaji, D. Buntinas, M. Chen, and G. Tan. Building Algorithmically Nonstop Fault Tolerant MPI Programs. In *IEEE International Conference on High Performance Computing (HiPC)*, Bangalore, India, December 2011.

[68] Y.-M. Wang. *Space reclamation for uncoordinated checkpointing in message-passing systems.* PhD thesis, University of Illinois at Urbana-Champaign, Champaign, Illinois, 1993. UMI Order No. GAX94-11816.

[69] S. W. Wei and C. H. Wei. A high-speed real-time binary BCH decoder. *IEEE Trans. Circuits and Systems for Video Technology*, 3(2):138–147, 1993.

[70] S. B. Wicker and V. K. Bhargava. *Reed-Solomon Codes and Their Applications.* Wiley-IEEE Press, 1999.