



Toward Exascale Resilience

Franck Cappello^{**}, Al Geist[°], Bill Gropp*, Sanjay Kale*, Bill Kramer*, Marc Snir*,
^{**}INRIA, ^{*}UIUC, [°]ORNL

Abstract

Over the past few years resilience has become a major issue for HPC systems, in particular in the perspective of large Petascale systems and future Exascale ones. These systems will typically gather from half a million to several millions of CPU cores running up to a billion of threads. From the current knowledge and observations of existing large systems, it is anticipated that Exascale systems will experience various kind of faults many times per day. It is also anticipated that the current approach for resilience, which relies on automatic or application level checkpoint-restart, will not work because the time for checkpointing and restarting will exceed the mean time to failure of a full system. This set of projections leaves the community of fault tolerance for HPC systems with a difficult challenge: finding new approaches, possibly radically disruptive, to run applications until their normal termination despite the essentially unstable nature of Exascale systems. Yet, the community has only five to six years to solve the problem. This white paper synthesizes the motivations, observations and research issues considered as determinant of several complementary experts of HPC in applications, programming model, distributed system and system management.

*Technical Report of the Illinois-INRIA Joint Laboratory on PetaScale Computing
TR-JLPC-09-01*



Toward Exascale Resilience

Franck Cappello^{**}, Al Geist[°], Bill Gropp^{*}, Sanjay Kale^{*}, Bill Kramer^{*}, Marc Snir^{*},
INRIA^{**}, *UIUC, °ORNL

Abstract

Over the past few years resilience has become a major issue for HPC systems, in particular in the perspective of large Petascale systems and future Exascale ones. These systems will typically gather from half a million to several millions of CPU cores running up to a billion of threads. From the current knowledge and observations of existing large systems, it is anticipated that Exascale systems will experience various kind of faults many times per day. It is also anticipated that the current approach for resilience, which relies on automatic or application level checkpoint-restart, will not work because the time for checkpointing and restarting will exceed the mean time to failure of a full system. This set of projections leaves the community of fault tolerance for HPC systems with a difficult challenge: finding new approaches, possibly radically disruptive, to run applications until their normal termination despite the essentially unstable nature of Exascale systems. Yet, the community has only five to six years to solve the problem. This white paper synthesizes the motivations, observations and research issues considered as determinant of several complementary experts of HPC in applications, programming model, distributed system and system management.

1. From Fault Tolerance to Resilience

Essentially, users of HPC systems want to be able to submit long-running jobs and have them run to completion in a timely manner. This demand is even more stringent for users of top-level supercomputers because these systems are acquired to run jobs that cannot complete in a timely manner on smaller systems. However several obstacles make this demand difficult to achieve even for today's supercomputers. Because of their scale and complexity, current supercomputers have frequent failures and can run for only a few days before some part of the system requires rebooting. While techniques for fault tolerance and continuous and tightly-coupled operation exist and are used in some specialized systems, these techniques have not been scaled to the level required for supercomputing, and are extremely expensive. The cheaper alternative of maintaining a safe state on stable disk storage does not work well for large, tightly coupled applications and results in the loss of significant compute work whenever a failure occurs.

The current response to faults in existing systems consists in restarting the execution of the application and the pieces of its software environment that have been affected by faults. To avoid restarting from the beginning, users may checkpoint the execution of their applications periodically and restart them from a safe checkpoint after faults have occurred. Note that in some situations, several pieces of the software environment have to be restarted as well. However, checkpointing and restarting has a cost: it takes time and energy. Some projections estimate that, with the current technique, the time to checkpoint and restart may exceed the mean time to interrupt of top supercomputers before 2015. This not only means that a computation will do little progress; it also means that fault-handling protocols have to handle multiple errors -- current solutions are often designed to handle single errors. Moreover, the current approach for fault tolerance is to apply the same technique (checkpoint-restart) to all types of faults (permanent node crash, detected transient errors, network errors, file system failures) and for the whole duration of the execution. However, not



all faults require the general and expensive checkpoint-restart approach. As an example, detected transient hardware faults (soft errors) may be managed in a more efficient way. Some recent work [GCG07] on BlueGene/L suggests that simple dedicated approaches (in terms of time and energy), could solve specific, well understood faults (errors in the L1 cache in the case of BlueGene/L).

If we observe the situation more closely, we see that none of the higher layers of the software stack have been specifically designed to cope with faults. Only a few software components, such as some MPI libraries, have been partially retrofitted to tolerate some faults. Moreover, there is no communication and coordination between software layers and software components within every layer for fault detection and management. An example of this lack is the MPI environment itself: even if some MPI libraries have been adapted to tolerate failures, their associated runtime environment is not fault tolerant, and requires a restart from scratch at every fault. Another example is the lack of coordination regarding fault detection and management between an application and the libraries used by the application. As a consequence, even if applications themselves were designed to resist to faults, most parts of their software environment would not let the execution survive the faults.

Since Exascale supercomputers, which are expected by 2018, will exhibit much more complexity and many more faults than today's supercomputers, one can gauge the challenge that the community in HPC is facing: it is not only adapting or optimizing well known and proven techniques but it is also making the full software stack fault tolerant and/or fault aware and ensuring that fault detection and management is consistent across the whole software stack. The problem of building reliable systems out of unreliable components did preoccupy the first generation of computing system designers -- see, e.g. [VN56], as first generation computers were very failure prone. The succeeding generations of system designers did succeed in building fairly reliable systems; application visible faults became rare events so that all faults could be treated in the same way. The inherent instability of Exascale systems, the envisioned diversity of their faults and the limitations of generic "one fits all" fault tolerance approaches will force the community to reconsider the fault tolerance problem as a whole, develop resilient software and provide a set of dedicated solutions in addition to (or instead of) a general purpose one.

2. A rapid tour of existing technologies and open issues at PetaScale

All current production quality technologies for fault tolerance in HPC systems and applications are designed for fail stop errors (node, OS or process crash, network disconnections, etc.) and rely on the classic checkpoint restart approach (automatic or application level) Most of the solutions focus on applications using MPI, albeit there are some exceptions like for Charm++. Almost all are storing the application state on remote storage, generally a parallel file system, through I/O nodes. Two recent exceptions are the Charm++ in memory distributed checkpoint scheme and the Scalable Checkpoint / Restart (SCR) library used at LLNL that tolerates a single node failure following the diskless checkpointing approach.

Existing technologies for checkpointing include BLCR (at the OS level) [BLCR], the application level checkpointing environment and SCR library [SCR] developed at Livermore, C3 (Cornell Checkpoint pre-Compiler) [BMP03] and virtual machine checkpointing with XEN. Other software environments have been used in the past but are no longer used in large centers: Libckpt (OS level improved by programmer annotations) [Libckpt] and the Condor stand alone checkpoint library (OS level) [CSCL]. The current most popular environment (BLCR) does not provide incremental checkpointing (at the time we are writing this article) and does not provide annotations that would allow the programmer a fine control of the checkpoint process. In addition to these non-commercial technologies, some vendors like IBM and SUN have developed their own sophisticated checkpoint/restart technologies.



At the parallel computer level, fault tolerant environments mostly concern MPI. Several environments have been designed or adapted to make MPI applications fault tolerant: FT-MPI (UTK) [FT-MPI], Open MPI (Consortium) [OpenMPI], MPICH-V (INRIA) [MPICH-V], LAM-MPI (Indiana) [LAM/MPI], MPVAPICH (Ohio) [MVAPICH], LA-MPI (Los Alamos), AMPI (UIUC). These environments differ in the type of faults they are managing, their design, fault tolerance approach and the fault tolerance protocol they are using. Other environments like Charm++ provide fault tolerance features. On the other hand, none of the PGAS environments are fault tolerant. Addressing this lack can be considered as an important priority.

Since, in the past, most of the development efforts on fault tolerance have been addressing fail stop errors with classic checkpoint restart, many issues remain open concerning the management of other types of faults and other fault tolerance approaches that have not been studied enough in-depth. As a matter of fact, many fundamental parameters of fault tolerance for HPC systems and parallel applications are not well understood. In the following paragraphs we give five examples of problems requiring more research even at PetaScale.

A first example is related to RAS analysis. Several analyses have been made concerning the root cause behind failures [OS07] [SG07] [Lu05]. The three most recent publications in this domain reach contradictory conclusions concerning the respective responsibility of software and hardware. This is not really surprising since these three studies analyze three different sets of systems in three different locations. This limited comprehension of root causes makes fault effect avoidance (the capability to avoid the effects of faults) difficult. Without a good understanding of root causes, it seems illusory to design and validate fault prediction mechanisms. Without good fault prediction systems, research on proactive actions is almost useless. In addition, even if at some point, we are capable of predicting errors accurately, we still have to find: 1) acceptable solutions to handle false negatives, and 2) how to handle predicted software errors (process or virtual machine migration is not a response for software errors).

Another example is “Silent Errors”. Silent errors are simply faults that never get detected, or get detected long after they generated erroneous results. They can be transient as in the case of hardware soft errors. Transient flipping of bits happens continuously in the memory of the largest systems in the world, but ECC memory automatically detects and corrects these faults. Silent hardware errors arise when any part of the memory is not protected by ECC, in data paths, processor registers or units that are not protected, or when multiple memory faults cancel each other out preventing the detection of the faults. Silent errors are not limited to transient effects; one can have permanent undetected hardware faults that are silent. Often they are only discovered when an application running on this hardware gives a clearly wrong answer, fails to complete, or completes much more slowly than usual. By then it may be too late for the application to recover. Silent errors are not limited to hardware faults. There have been several cases where software or firmware code has had bugs in it that only manifest in rare cases, for example, router-chip software that changes the bits in one message out of every billion. The key characteristic of silent errors is that they are undetected; therefore, there is no opportunity for an application to adapt or recover from the fault at the moment when it hits the system. If the rate of silent errors is too high, then a user must worry that the results of his simulation are incorrect. Designing mechanisms to tolerate silent errors depend on a better comprehension of these errors. Very few results are available about the quantitative evaluation of their likelihood at large scale during execution.

While large parallel computers often have hardware mechanisms for fault detection in specific subsystems (e.g., the interconnection network), they do not usually provide an integrated approach that ensures fault tolerance at the system level. The problem is compounded by the use of commodity components designed for cost sensitive applications that may tolerate high failure rates and frequent silent errors (e.g., gaming). Modern data centers (such as at Google) that are designed to be resilient,



in order to provide continuous service; such a design is facilitated by the nature of the workloads they support (embarrassingly parallel, or very coarse grain). Parallel computers (whatever their size) by the nature of their workloads (fine grain parallelism) makes such a design harder. However, there are many opportunities of hardware supports with SSDs devices, specific networks, Transactional Memory, Hardware diagnostics, fault detectors, etc. There is a need to understand what kind of hardware could be added and how to integrate it in the systems with reasonable cost and energy consumption.

We already mentioned the lack of coordination between software layers with regards to errors and fault management. Currently, when a software layer or component detects a fault it does not inform the other parts of the software running on the system in a consistent manner. As a consequence, fault-handling actions taken by this software component are hidden to the rest of the system. Imagine that two software components detect faults with the same root cause; they may take contradictory actions because of this lack of coordination. Likewise, if a software layer detects an erroneous condition, it often has insufficient or incorrect information to determine the best response. In an ideal word, if a software component detects a potential error, then the information should propagate to other components that may be affected by the error or that control resources that may be responsible for the error. Some software components may also contribute to the decision making process, deciding on a plan for recovery, reconfiguration or adaptation. All software components should then conform to this decision and take corresponding actions. There is already an attempt to develop a coordination system for software called CIFT [CIFT]. However few components are available and there is certainly a need to put more efforts on that topic.

One suggested way to cope with silent errors and continuous stream of faults is to design fault oblivious algorithms. These algorithms should demonstrate resilience and correctness in the face of faults [EG05]. Very little is known today about how to create such algorithms except in the simplest cases that are nearly embarrassingly parallel or for problems that have simple checkers [BK95]. An example of questions that should be answered is the following: to what extent are existing numerical methods such as asynchronous iterative methods and chaotic relaxation approaches resistant to information losses due to communication corruption, temporary disconnection, transient hardware failures or damaged software presenting a Byzantine behavior? Addressing this challenge does not rely only on application developers; the system software also needs to cope with a continuous stream of faults and being in a constant state reconfiguration of the system.

As discussed in the previous section, the fault tolerance approaches used in production are supposed to be generic, managing all fault cases in the same global way whatever are the faults, the application and the execution phase within the application. Adaptive data protection may help to provide a more efficient approach by allowing specific responses according to the fault context. For example, not all data need to be saved at every checkpoint. This observation is the root of the “Memory exclusion” mechanism proposed by Plank [PYK99]. Advanced techniques such as “compiler assisted memory exclusion” help programmers to identify explicitly, in the code of the application, the data to be removed from the checkpoint. Another observation is that the current procedure for checkpoint/restart which essentially relies on copying useful memory content and storing it on a stable storage (and vice versa for restart) may not be the most efficient/rapid way of saving and reconstructing the data. Programmers are in the best position to know how to save and restore quickly the critical data of their application but they cannot use such adaptive data protection other than by doing it manually. More generally, there is a need to investigate the notion of “correctness models”. The programmer may provide in the program annotations about ways to protect or check key data, computations or communications. The programmer may provide assertions that help improve anomaly detection. The system may use such annotations and assertions adaptively. This is similar to adaptive resource management in a run-time system: The programmer annotates the program to provide information on its properties; the run-time uses this information, and



information on the state of the system, to allocate resource. One can think "fault-tolerance" budget that is managed by a "fault-tolerance manager", based in information on the computation provided by the user, and information on the system, collected from various sensors.

3. Issues in Exascale systems

There is a broad consensus in the community about the fact that Exascale systems will be hit by errors/faults much more frequently than Petascale systems. There are two main reasons behind this belief: (1) an Exascale system will be composed of many more components than PetaScale systems and (2) the mean time to failure (MTTF) of each of these components will not improve enough to compensate for (1).

Current projections of Exascale systems are considering that a single supercomputer of this category will feature millions of CPU cores and may have to run up to a billion of threads. If we look to the past ten years, the performance increase of the best supercomputers in the Top500 resulted from an increase in CPU clock frequency, an increase in the number of transistors per chip and an increase in the number of sockets in a machine. Clock frequency has flattened in the last few years, so that the increase in the number of sockets can be expected to accelerate. As a consequence, the number of components in Exascale system will be much higher than the one of PetaScale systems (100,000 is the order of magnitude of the number of sockets we may see in Exascale systems). Moreover, the reliability of individual components is not likely to improve significantly in the near future. The lifetimes of consumer products provide no incentive for manufacturers to change the existing reliability levels of components, which is typically a few years. Indeed, the reliability of the components in HPC systems has not improved and may have degraded in the last 10 years. HPC vendors have compensated for this by adding more hardware redundancy and error checking in their systems.

Even if there is not yet a consensus on this aspect, there is a suspicion that software errors will dominate in Exascale system. The rationale behind this belief is that (1) the software stack running on every node of a parallel computer is already very complex (current estimations of the number of code lines in such software is several millions) and (2) this software stack has not been designed or tested with high availability and resilience in mind. As a consequence most of the software parts: (a) are not restartable or replaceable without impacting the other software parts, (b) do not integrate enough fault-error detectors, and (c) have not been tested, validated or formally verified at the (much more expensive) level used for critical software.

The community has translated these projections and suspicions into the following statement: faults/errors/failures will not be rare events anymore and should be considered as normal events. In other words Exascale systems will need to resist a continuous stream of faults/errors/failures.

The community, based on its past experiences and the observations of the current largest systems, envisions the following major issues:

- 1) Some faults will not be detected (silent errors). Both hardware and software silent errors are likely to happen.
- 2) Detected but uncorrectable transient errors may represent a large fraction of errors. The assumption is that with the increase of the integration level, the phenomena causing transient errors will have a much wider impact on the affected components, despite the redundancy mechanisms added by the manufacturers.
- 3) Correctable errors will increase the hardware jitter due to the background error recovery activities (e.g., memory scrubbing & error handling) that will become significant.
- 4) Long running jobs may be hit by hardware & software faults (of multiple types) several times before completion.

- 5) Current designs and practices of global, synchronized Checkpoint-Restart (on remote file system) will not work anymore.

The community concurs that research for more reliability and robustness is critical at every layer between the hardware and the end-user. Considering the existing technologies, the state of the art in research and the forecasted faults/errors characteristics of Exascale systems, new resilience paradigms are required.

4. First set of recommendation: improving/facilitating exchanges and sharing in the community

The resilience topic has several characteristics that distinguish it from the other research topics related to Exascale systems with regards to the involved community.

First the community is very large and involves: A) computers scientists whose main field is fault tolerance and resilience, B) computer scientists whose main field is in other domains but with some experience or specific knowledge on fault tolerance and resilience (people working on programming models, file systems, libraries, etc.), C) application developers, D) researchers and developers in numerical algorithms E) system managers and experts in HPC centers, F) users of large scale long running applications and G) software and hardware vendors.

Second, the community is spread over the world but not organized at the international level. Skills of top-level research teams are mostly complementary; however there are some overlaps like in the domains of fault tolerant MPI or event log analysis. An important observation is that not any single team contains all the required skills to address the critical resilience issues listed in the following paragraphs.

All the participants of the HPC domain involved in resilience come with their own culture and perception of faults, errors, failures, their origins and consequences. The research methodology is not well established and shared across the HPC community. Simple examples are: 1) the lack of common metrics and benchmarks to stress fault tolerance or resilience approaches, 2) the lack of standardization and common structure for event logging and analysis, 3) the lack of experimental environment.

As one can imagine, event logs are very important sources of information to understand the root cause behind failures. However, there is no existing standard on how system managers enter events in the logs that have not been reported by the automatic detection systems and automated systems do not follow standard formats. Moreover, it happens that the human entered information is not consistent with the one generated by the automatic detection systems. This raises significant difficulties when event logs have to be filtered, analyzed and mined to determine root causes.

Another remarkable element, which is a good example of problems raised when the coordination is weak, is the lack of common definition on some critical issues. Take as example the definition of “soft errors”. This term is becoming widely used to express a kind of faults that is expected to be prevalent (or at least very frequent) in Exascale systems. The definition of this term varies. In some communities “soft errors” means transient physical faults, while in the general definition, published in one reference paper on dependability “Errors produced by intermittent faults are usually termed soft errors” and “elusive development faults and transient physical faults leads to both classes being grouped together as intermittent faults.” [ALR04]. The last definition clearly mentions software as a potential source of “soft errors” which is not the case for the former definition. “Examples are memory bloating and leaking, unterminated threads, unreleased file-locks, data corruption, storage space fragmentation, and accumulation of round-off errors” [ALR04]. Obviously, sharing common definitions on key issues is essential as a very first step of community coordination.



The notion of intermittent or transient errors itself is not well established in the HPC domain. Obviously transient or intermittent errors may be managed in different ways than permanent errors. What are the existing criteria and procedures to qualify an error in HPC systems as intermittent or transient? Who is responsible for making this distinction: the system manager or some algorithms analyzing the event logs? The answers to these questions are still open in our community and far from being shared.

As a consequence, there is a need for more exchanges and sharing between the members of the resilience community at the international scale.

A first element to address this situation would be to establish a conference on Resilience in HPC. As a matter of fact, there is no conference in HPC dedicated to the resilience topic and vice versa. One can question the need for such a dedicated conference, since fault tolerance and resilience are well-identified research domains having their own top-level general-purpose conferences. However, fault-tolerance and resilience in general are wide research domains and some of their application domains (distributed systems, sensor networks, databases, HPC, etc.) have commonalities in research topics, several research questions are not overlapping. For examples, 1) self-stabilizing algorithms are key mechanisms of sensors networks but have not been used so far in HPC environments, 2) replication is the major approach for fault tolerance in all applications domains except HPC, 3) diskless checkpointing or application based fault tolerance are considered as interesting mechanisms only in HPC but are disregarded by other domains. Moreover we consider that the spectrum of questions related to resilience in HPC systems is large enough to motivate the organization of an annual even gathering the community. The format and organization of such events remains to be defined.

Another important element is to improve the information and methodology sharing across the community. The fact that event logs of the most recent large-scale supercomputer facilities are not easily accessible by a large research community is a major impediment to progress in the domain of fault, error and failure analysis. Event logs anonymization has been considered as a possible solution to this problem but remains to be investigated and applied. In terms of methodology, there is no consensus in the community on “what, how and where” to measure and test fault tolerance/resilience techniques for HPC. Several metrics have been defined recently for characterizing a full system, such as “survivability” and “performability” [NA03] but there are not widely used. Other metrics could be of interest such as “reaction time” for the different mechanisms used in a fault-tolerant/resilient system and “resistance factors” to express how well a system resists high fault frequencies or to high numbers of simultaneous faults.

5. Research issues considered as important

The community has identified four main research topics considered as determinant to ensure the correct termination of parallel executions on Exascale systems:

- 1) fault detection, propagation and understanding
- 2) fault recovery
- 3) fault-oblivious algorithms
- 4) Stress-testing proposed fault-tolerance solutions



5.1 Fault detection and understanding

Fault detection and understanding covers three sub-problems: improving fault detection and layer coordination, understanding faults and silent errors and improving situational awareness.

5.1.1 Fault detection and layer coordination

There are only few ways to deal with undetected faults (silent errors): 1) ignore them, assuming that they will stay marginal and that they will not degrade the result quality to an unacceptable level, 2) develop fault oblivious algorithms that can resist by construction to a high level of undetected errors, 3) use some form of redundancy and on-line checking or 4) store, transfer and compute on an encoded version of the data (like with the ABFT technique). The first is unacceptable as undetected faults are too frequent. The second will require years if not decades of efforts to transform existing algorithms and methods to resilient versions resisting to a large variety of fault and errors. The third is feared to be too expensive in terms of overhead. The fourth is currently mostly limited to linear algebra operations. Thus, considering the risk of not managing undetected errors and the cost for their management, the community considers that more software and hardware detectors should be integrated in HPC systems to reduce the occurrence of undetected errors.

Not only should more errors be detected, but a higher error detection resolution should be used to determine initial/likely software component failure, for both system wide and non-system wide software caused failures.

As discussed in previous sections, there is a need to coordinate fault-error detection, propagation and management across the software layers. To progress toward this objective, there is an immediate need to improve the propagation of error conditions through software layers and hardware components. At NERSC, about a year was spent trying to get correct errors encodings coming up through the software layers for a recent HPC system [KR08]. Efforts for correct propagation were required in almost every software layer.

5.1.2 Understanding faults and silent errors

Detecting errors/faults is a necessary objective to trigger fault tolerance mechanisms and let them implement corrections. Of equal importance are the finding of the root causes and the analysis of the causalities chain that provoked the detected errors. Finding root causes helps applying specific corrections in order to avoid future occurrences of the error. Since we envision that software will be responsible for a large fraction of the errors, there is a need for a better analysis of root cause behaviors across software errors. For example, we need to verify if the informal root cause analysis in some bug reports that indicate a large portion of software errors are caused by software that is in error recovery or exception handling is itself correct. Analyzing the causality chain helps understanding how to stop the “contamination” of a root error by the introduction of confinement mechanisms. Several researches have been conducted on this topic. However log analysis is still a very complex issue for several reasons: lack of standardization, lack of consistence in the log files and lack of research in log analysis engines.

The Petascale System Integration Workshop [KR07], organized a couple of years ago already identified as important the format standardization for all logs. Another important objective would be a standard for error encoding. Obviously, standardization is fundamental to efficient log analysis of log files coming from different sources and also to limit the development effort and concentrate most of it on the analysis engine and not on translators. The consistent integration between human entered event logging and automated information is regarded as another significant issue. In analyzing the error logs for 5 years at NERSC - it took about a person year to make all the operational and repair logs consistent with automated log (<http://pdsi.nersc.gov>). Whatever the standardization level and the consistency of the logs are, the challenge is to create the analysis engines. Finding root causes requires browsing huge amounts of

information and using filters detecting redundancies. Several recent papers have presented analysis results for different supercomputers. Since their conclusions are significantly diverging, it is important to understand the sources of this variation. Clearly, there is a need for more efforts and collaborations in this domain. The data mining community could probably help in this research.

Despite the existing error detection system, some errors are not detected and reported. These silent errors are not well understood in their source, consequences and quantification. Silent errors are suspected and observed by their effects for example when an application gives results considered as too diverging from the expected ones, fails to complete without detected errors or completes much more slowly than usual. Designing mechanisms to tolerate these errors depends on a better comprehension of the errors especially when they come from the hardware. However, we are not aware of any quantitative evaluation of their likelihood at large scale during the application executions. The question here is: do we really need to develop specific mechanisms for these errors if they only affect a very small fraction of the executions? To give an answer to this question there is a need to determine if existing error occurrence models are enough to estimate quantitatively the likelihood of these errors for Exascale systems. If the models are not accurate enough, we will need to use or create measurement platforms.

Understanding the sensitivity of HPC usage scenarios, applications and algorithms to faults, errors and failures is as important as understanding the errors and their root causes (faults)¹. In the HPC context the impact of faults and errors could be a significant slowdown of the application, a crash or an incorrect result. If the fault tolerance mechanism reacts to faults and errors without considering the context of the execution, the result could be undesirable. For example, it was observed that some systems spend so much time in error recovery that performance becomes unpredictable. As a matter of fact, the impact of a failure in HPC applications varies and how to recover from errors depends on the context. Some HPC users launch large sets of small or medium scale simulations following a parameter sweep like method. If one of the simulations fails, the user can decide whether or not it is worthwhile to re-launch it. In this scenario, the consequence of a failure is less catastrophic than in a scenario where a single large simulation crashes or gives an incorrect result after tens hours of execution. Some applications written following the master-worker paradigm do not need, in theory, to be fully restarted if an error or a fault happens in one worker node: only the failed process(es) have to be re-launched. Some algorithms based on chaotic relaxation seem to present good resilience qualities and can tolerate a certain amount of information loss, though these algorithms may not perform as well as less resilient ones. In a distributed system, the self-stabilizing algorithms can follow a forward recovery approach that can even tolerate Byzantine attacks. Such algorithms do not need an “external” correction of the errors to return eventually to a correct state. Thus, the response to a failure or an error depends on the context and the specific sensitivity to faults of the usage scenarios, applications and algorithms. However, even if this dependency to the context seems clear, there is no existing study evaluating this sensitivity and the minimal response for a large variety of scenarios, applications and algorithms.

In particular, little is known on the sensitivity of the algorithms to silent errors. A study published in SC 2004 could be a good starting point [CR04]. Using software fault injection in a non-protected hardware (memory without ECC, no protocol checksum on communications, etc.), the authors have simulated single bit memory errors, register file upsets and MPI message payload corruption and measured the consequences for a suite of MPI applications. These experiments showed that most applications are very sensitive to even single errors. The errors were often undetected (thus silent), leading to incorrect outputs. This study needs to be continued and expanded because it considered only three applications at small scale (less than 200 processes) and only silent hardware faults. Going further in this direction, there is a need to understand the fundamental origin of the algorithm sensitivity to silent errors.

¹ By definition a failure is the impact of an error itself caused by a fault.

5.1.3 Situational awareness

In some cases, a major error is reported too late to the system manager for corrective actions despite the fact that several minor errors have occurred and could have been reported. The problem comes from the automatic systems that silently correct some errors. An example is disk controllers that are very good at correcting errors and rebuilding data. However, disk controllers rarely inform the system manager about error detection and repair. In some situations, the repetition of minor errors eventually leads to major defects that controllers cannot fix. The system manager is then informed but too late to take corrective action, such as replacing the failing device.

Situational awareness² supposes excellent filtering systems that warn the system manager only about relevant issues, but provides sufficient information in a timely manner for the system manager to take appropriate action. Situation awareness is not a new problem for domains like air traffic control, power plant operations, emergency response, military command and control and medicine. This is a major concern in these domains and many efforts were devoted to studying what information is needed by human managers so, when automated systems fail, humans can step in and make effective judgments. There is a need to investigate more this area and probably learning from lessons in the mentioned domains using situational awareness.

5.2 Fault Recovery

5.2.1 Non-masking approaches

The principle of non masking approaches is to report faults and errors to the applications or the users and let them: a) decide on which strategy to adopt according to the faults and errors and have the runtime environment or the OS manage the recovery process (FT-MPI) or b) actually organize most of the recovery process with minimal or no intervention of the runtime environment or the OS (Application level checkpoint-restart and ABFT).

The non-masking approach was the original one for most applications and users. Most large scale, long lasting applications have some dedicated portion of the code devoted to checkpoint and restart. Checkpoint-Restart in these codes has several objectives: it can be used to debug the application, to pause its execution to release the computing resources to other applications and for fault tolerance. In such applications, the code is not directly managing all aspects of fault tolerance. In particular there is generally no mechanism for fault detection. When the environment (the batch system for example) reports a failure (generally the crash of the application), the user is informed and has the option to resubmit the application asking to restart its execution from the latest valid checkpoint. Saving the state of the application is the responsibility of the programmer who has: 1) to decide what data to save on a stable storage (local disk or remote file system), 2) to make sure that states of all participating processes are consistent before saving the data and 3) to rearrange the application code to make the application restartable from the saved data. An application level checkpoint restart library can help the application programmer.

Application level checkpoint-restart has several known limitations that have not been really quantified by a thorough analysis. First, since the programmer puts the checkpoint code directly into the application, there is no easy way to externally control the checkpoint interval. Indeed, without external control, the checkpoint interval depends directly on the performance of the processor and not on their estimated reliability. There is a way to alleviate this issue, under some conditions, by intercepting the checkpoint

² Situation Awareness (SA) is a term used by the Human Factors community to indicate, *"the perception of elements in the environment within a volume of time and space, the comprehension of their meaning, and the projection of their status in the near future,"* [EN95]. Another definition is knowing what is happening and what has happened so a person can take the correct action.



requests made by the application and have a control environment deciding whether or not to checkpoint. A second problem is the size of the checkpoint generated by this approach. It is suspected that the programmers save more data than needed because they cannot always easily track the modifications of the data structures between consecutive checkpoints. However, too few quantitative results are available on the overhead caused by a non-optimal checkpoint interval and on the size of the data saved during the application that is not required for restart. Furthermore, many third party software packages do not have checkpoint restart options. An in-depth study of this approach would help to understand its real limits.

One promising non-masking approach is Algorithm Based Fault tolerance [HA84]. It is classified as a non masking approach because the programmers are required to adapt their algorithms to integrate ABFT. The ABFT approach consists of computing on data encoded with some level of redundancy. If an error or a fault occurs during the computation and if the encoded results has enough redundancy, it remains possible to reconstruct the missing part of the result. ABFT was proposed for systolic arrays. It is based on a mathematical property of the algorithm: the decoded results correspond to the algorithm applied on the non-encoded input parameters. In other words the coded result stays consistent with the encoded input parameters thanks to the consistence preservation property of the algorithm. ABFT has several limitations in practice: 1) it works for linear algebra and FFT but, to our knowledge, the question of its applicability to other domains stays open. 2) ABFT was designed for off-line detection and correction. Since execution in Exascale systems are supposed to be hit by several faults before their termination, there is a need for on-line error detection and correction. On-line ABFT [Ch08] consists in stopping the execution of the algorithm at some point before the termination, detecting and potentially correcting errors on the data sets. A recent paper demonstrates that the consistence conservation property is not respected during the computation for several matrix product algorithms. As a consequence these algorithms cannot be used for on-line ABFT. 3) To detect errors on the data sets, ABFT needs global reduction operations. Error correction requires solving a system of linear equations. Reduction and linear system solving involving billions of threads is probably not feasible in an efficient manner. 4) Although some recent work demonstrates that the numerical stability is not harmed in some very specific cases, the general understanding on the impact of ABFT on the numerical stability remains to be investigated. Thus to understand the applicability of ABFT to future Exascale computers and find solutions to its current limitations requires more analysis and research.

5.2.2 Masking approaches with rollback-recovery

With the masking approach the faults-errors and failures of the hardware and software are masked to the application (and thus its programmers) and end-users. Masking approaches concern three main fault tolerance techniques: Rollback Recovery, Proactive Action and Replication.

As explained previously, most of the work in fault tolerance for HPC is based on Rollback Recovery. Many research projects have been conducted and results published in the past year about automatic checkpoint-restart on remote storage and diskless checkpointing. Despite the fact that these domains have received a lot of attention, several important issues at Exascale remain open.

All checkpoint-restart techniques need to address two main problems: 1) find a scalable fault tolerant protocol ensuring a consistent cut of the parallel execution in-order to built correct restartable checkpoint images and 2) reduce the computational and/or I/O cost of checkpointing.

Concerning fault tolerant protocols, the progress in the past 10 years is substantial [EAW02] [BGD08]. The two classes of protocols (coordinated checkpointing [CL85] and message logging) were the subjects of strong optimization research. As a consequence both classes of protocols are very efficient even on high speed networks and the differences in performance and overhead between them have become less and less significant. However we still do not know how well these protocols work for one million processes.



Moreover, in some sense, the optimal protocol for extreme scale has not been found: such protocol should, in theory, avoid global checkpoint coordination, message logging and domino effect. A solution could come from a third class of protocols called “communication induced checkpoint” (CIC). However, previous researches have demonstrated that this kind of protocol could force many more checkpoints than the other classes. Since the cost of checkpointing is supposed to be very high, all previous research has considered that the fewer checkpoints are taken during the execution the better. This raises the question of the checkpoint cost in Exascale systems. If the checkpoint cost could be reduced significantly (for example by redesigning the architecture and using dedicated hardware) then CIC protocols may provide good trade-off.

If fault tolerance protocols are well understood for MPI environments, there were few studies in the context of PGAS languages and distributed objects at large scale. What kind of fault tolerant protocols are preferable for these environments? Are there some fundamental differences that preclude the use of some protocols in these environments or make them inefficient?

Independently of the programming and runtime environment, there is a lack of understanding of the application of fault tolerant protocols in parallel applications. Since the beginning of the research in fault tolerance in parallel applications, the assumption is that parallel programs essentially behave like distributed systems and as a consequence, their state should be saved using the same mechanisms. This initial assumption is questionable since parallel applications present some essential properties that could be the opportunity for designing or specializing fault tolerance principles specifically for them.

The cost of checkpointing is the second main problem to address for checkpoint-restart. It has been observed that a large fraction of the checkpointing cost is due to the remote storage of the checkpoint image. In the current “balanced parallel architecture” followed by most of the HPC centers, checkpoint images are stored on a parallel file system through I/O nodes. The bandwidth of the I/O nodes is limited and according to some measurements and evaluation, the time to store a checkpoint could be as high as 30 minutes. A technique to avoid this bottleneck is to store the checkpoint image on computing nodes. Some redundancy mechanisms are used to ensure that the checkpoints will stay available even if several computing nodes fail. The extreme approach in this domain is diskless checkpointing [PLP98] [Lu05] [ZSK04] that essentially stores the checkpoint images in the memory of the computing nodes, still with some form of redundancy to ensure the checkpoint availability in case of failures. Diskless checkpointing has been the object of many research studies. A version of diskless checkpointing supporting a single node failure is implemented in SCR [SCR] and used since 2007 at LLNL. It has some important drawbacks like the doubling of the memory occupation and some need for coordination (when computing the checksums). These two drawbacks could be considered as unacceptable for Exascale systems where the memory will be a very expensive resource and where coordination should be avoided as much as possible. However, new technologies like SSD devices seem to offer a new performance and stability trade-off compared to RAM and disks. This is seen by the community as an opportunity to revisit diskless checkpointing studies in the light of these new technologies.

Besides avoiding the bottleneck of remote file systems, another way to reduce the checkpoint time is to reduce the checkpoint size. This topic also has been the object of many researches from various angles: OS level incremental checkpointing, programmer directed memory exclusion, compiler detected dead variables, etc. The two key points exploited by these techniques are: 1) not all data have to be saved to construct a correct checkpoint and 2) data that need to be saved may not be completely updated between 2 consecutive checkpoints. The researched optimality is the following: between 2 subsequent checkpoints, save only the updated data that will be read in the future (as parameters of next operations inside the application or as results of the execution). This supposes an accurate determination of the past updates



(from the previous checkpoint) and the future reads. As a matter of fact none of the existing systems matches this objective. As a consequence, there is certainly a need for further research in this domain.

In addition to the increasing the performance in checkpoint-restart, there is a need to ensure the correctness of the data saved and restored for the checkpoint (whether it's a system wide or an application based checkpoint). This issue, which is mostly ignored currently, is expected to be significant at Exascale. End-to-end data correctness/integrity is not a novel topic in storage systems. Several standards already exist (e.g ANSI T10-DIF). However and especially on the case of checkpoint-restart, every component (HW and SW) from registers to hard disk need to conform to have complete confidence. More progresses and researches are needed in this domain.

5.2.3 Other masking approaches (proactive actions and replication)

The principle of proactive action is to avoid recovery from faults, errors and failures by predicting them (from notification from sensors, error rates, heuristics, etc.) and proactively replace the suspected components (in theory hardware and software ones) by other correctly working components providing the same function. Two main issues have to be addressed: 1) faults, errors and failure prediction and 2) replacement of suspected components by correct ones. Several researches have been conducted on this domain and results were published in prediction algorithms [Sah06][Yaw07][Sah02] and proactive process or virtual machine migration [Cha08][Sat06][Sco09][CMK05].

One limitation of the prediction algorithm studies is the number of RAS files considered for the prediction algorithms. Most of the published studies considered two log files: one from a cluster of 350 nodes (small compared to our perspective of Exascale systems) and the other one from the first 100 days of a BlueGene/L machine. These two traces are not representative even of current Petascale systems and it is obviously very important to design prediction algorithms based on the most recent traces and a large number of traces. Furthermore, systems currently log only a small subset of the information that can be captured and would be relevant for error prediction. The sheer volume of such data may require on the fly analysis, rather than full capture. We consider efforts in this area as a priority if research on proactive actions is to be continued.

Research on proactive actions has been conducted despite the immaturity of prediction algorithms. This research is useful to understand how the software could use predictions to avoid recovering from faults, errors and failures. The results are quite encouraging, evaluating the overhead of proactive migration to few percents of the execution time, while significantly reducing the occurrence of fatal errors. A first limitation of these studies is that they only consider hardware fault-error-failure prediction. In such context the migration of a process or a virtual machine from a suspected node to a safe one makes sense. However, none of the studies investigates the case of predicted software faults-errors-failures. This lack is significant since one can estimate from available the RAS analysis results that software is responsible for about 50% of the faults-errors-failures. Avoiding recovery for predicted software faults-errors-failures means being able either to confine their effects or to track the affected software parts and to be able to replace the suspected software parts. A frequent misconception is that software errors are systemic and strongly correlated, so that replacing software components would mean having a different code implementing each software component. In fact, many software errors are intermittent and manifest themselves only in specific, rare execution states (specific memory allocation, specific schedule, etc.); many such errors can be avoided by tinkering with the execution state [QTSZ05]. Research on the handling of software errors is still at early stages and has not been explored in the context of HPC applications and systems.

More generally there is a need to better understand the benefit of proactive actions. If these are limited to hardware, they would still require some recovery system to tolerate the large amount of unpredicted



failures. Even if proactive-actions are extended to software faults-errors-failures, there would still remain a fraction of false negatives (actual faults-errors-failures that will not be predicted), forcing the use of recovery mechanisms. There may be systemic differences between predictable errors and aleatory errors that cannot be predicted reliably. Categorizing errors according to their predictability could lead to much improved error handling. We consider developing the understanding of predictable errors and proactive actions as a main priority.

Replication is another masking approach for fault tolerance. It is the most frequently used resilience technique in all large scale distributed systems: sensor networks, Grid, Cloud, P2P, Desktop Grid, etc. Replication has two main costs: 1) the need to at least double the number of replicated components in the system and 2) the overhead to guarantee the replication consistency and detect errors. Both costs seem unacceptable in the context of HPC systems (except for some very limited cases): CPUs, memory and network are very expensive and high energy consuming components; the speed of communication and asynchrony are essential for performance so that matching replicates is hard; and much state is not externally visible, so that errors can be detected long after their occurrence. Thus classic replication may not be feasible for HPC unless a dramatic change in the hardware design of HPC machines emerges. Such breakthrough should not be disregarded since the past has demonstrated that hardware developers can develop radically new approaches, based on new objectives (like a drastic reduction of the power consumption) and surprise the community. Besides classic replication, there is probably an opportunity to investigate partial replication of only the most critical and fault-errors-failure prone hardware components. However such an approach would require a deep understanding of the faults-errors-failures sources in very large systems, which is by itself still an open question.

5.2.4 Specialized approaches

The difficulty of designing or improving a “one size fits all” fault tolerance approach for Exascale systems motivates the investigation of dedicated fault tolerance and recovery mechanisms targeting specific failure modes: a kind of divide and conquer approach to fault tolerance. The expected benefit of this approach is that one would trigger the minimal sufficient response when a failure happens; the ultimate goal being to be able to confine the recovery only to the affected subsystems and avoid as much as possible a global rollback of the execution.

Let’s take the example of the transient errors in the L1 cache. Many processors have a write-through L1 and an inclusive L2, so that all L1 values are also present in the L2 cache. As a result, it is sufficient to detect errors in the L1 cache; errors can be corrected by invalidating the L1 cache so that values are reloaded from the L2. Vendors take advantage of this by using a cheaper and faster parity check in the L1, rather than a more expensive ECC.

Such specialized approaches are already used in HPC machines: For example, the network interface often has retransmission mechanisms that are used when transmission errors or uncertain states (e.g. timeouts) are detected; one can trade off the overhead of error correction in the network against the overhead of maintaining a copy of data sent until reception is acknowledged. However, the mechanisms in place cannot correct all errors cases and “a posteriori” dedicated approaches could be a solution for certain cases.

One problem with the dedicated approach is that it is machine dependent. Another problem is that error detection may occur long after the error occurred; the error is not contained, and it can affect a large portion of the system before it is detected, so that recovery becomes global. If we take the example of a detected but uncorrected transient error in the main memory, affecting a communication receive buffer, then the recovery process would typically involve the sender of the message. Such dependency leads to



some coordination between several processes and has a much higher cost than a local corrective technique.

Dedicated techniques rely on the capability to confine faults. The capability to quickly detect an error and to limit its contamination is essential.

As discussed earlier, situational awareness is very important to intervene before minor problems treated silently become or lead to very critical ones. Thus even if dedicated techniques are applied, there is still a need to inform other components about the detected events and how they are being treated. Moreover if the event is perceived by several detectors and can be corrected in different ways, coordination will be required.

Specialized approaches are undoubtedly appealing for their very low recovery cost but further investigation is required to clearly understand their limits.

5.2.5 Hardware support

The four main mechanisms used in rollback recovery are for: 1) detecting failures, 2) storing and restoring the state of a parallel execution, 3) ensuring that the state of the parallel execution is always consistent, despite failures and 4) computing an accurate difference between 2 successive states. Note that other fault-tolerance approaches need several or all of these mechanisms, in addition to others.

In the current supercomputers very little hardware has been designed specifically to help implement fault tolerance. For example, when checkpointing on a remote file system, the process checkpoint images are transmitted by the communication network used by applications during their execution and stored on disks used for storing users files. No specific hardware is available to implement fault tolerant protocols. For example, message logging is done in the main memory of the node. Another essential mechanism for fault tolerance (incremental checkpointing) currently relies on OS mechanisms.

However, because the speed of detection, reaction and recovery will be critical in effectively supporting fault tolerance in Exascale systems, there is a need to investigate the cost, benefits and limitations of using specific hardware for fault-tolerance. There are several opportunities for the four mechanisms used in rollback recovery. SSD devices have demonstrated superior performance compared to disks and could be good alternative for checkpoint image storage and retrieval. A specific network can help to circulate information about the detected faults-errors-failures and their current treatment. A specific reduction network can also be used to speed up the computation of checksums in diskless checkpointing. Another example is the hardware support for transactional memory that can be reused to implement efficient incremental checkpointing [TNT06].

5.2.6 Software support

Software is suspected to be responsible for a significant fraction, if not the large majority, of the failures in Exascale systems. The software run on large-scale parallel computers is already very complex but has not been developed with the same rigor of complex hardware (like current microprocessors) with regards to its verification. There is a need to use better development methods to make code more robust (robust journaling capability). There is also a need to quicker detect and better confine software errors in order to limit their contamination and make the error correction more local (avoiding global rollback).

5.3 Fault oblivious algorithms

Having the applications and their algorithms indifferent to fault-errors-failure would be an ultimate goal of the resilience community. As mentioned before this could only be a long-term effort since not only the



application and its algorithms but also all software involved in the execution affected by the fault-error-failure should also be fault tolerant (or resilient).

Very little is known in this domain of auto-repairable algorithms, even if some algorithms for distributed systems, like self-stabilization algorithms [Dij74], can recover themselves from faults. Self-stabilization is a property of an algorithm to eventually recover from a fault using a forward recovery strategy: instead of being externally stopped and rolled-back to a previous correct state, the algorithm continues its execution, impaired by the fault, potentially not respecting its given specifications (giving wrong results), until the algorithm corrects itself the effects of the faults without external influence. Self-stabilization has several limits in the context of HPC applications: 1) self-stabilized algorithms have been studied uniquely in the context of basic distributed system operations (leader election, counting, etc.) and never approached for HPC, 2) the stabilization phase duration is unknown and 3) any fault occurring during the stabilization phase essentially restarts the stabilization. Thus it is not clear that this approach makes any sense for the numerical algorithms themselves. However, many mechanisms in runtime environments that support parallel computations use distributed system operations that could benefit from this family of algorithms.

It is suspected that iterative methods have some fundamental properties allowing them to be good starting points for the exploration of fault oblivious algorithms [GE07] [Li02]. Asynchronous iterative methods tolerate delays in the communications of data between processes. Research must be conducted to understand how these methods resist information loss and to what extent they still converge despite a certain level of faults. According to applied mathematicians, these methods have never been directly studied in this context. We see this research as a multidisciplinary one mixing research in mathematics and computer science.

5.4 Stress Testing

In addition to progress in application, system and hardware, there is a need for experimental environments that are able to stress and compare different fault tolerance approaches and techniques in a scientific way. Large-scale testbeds are essential in the observation and understanding of complex phenomena. Software environments capable of reproducing usage and fault scenarios are also needed to test and debug new resilience concepts at large scale, before putting them in production.

6. Summary

This paper argues that resiliency is a critical challenge that cannot be ignored for Exascale systems. Incremental improvements of current methods will not be sufficient to meet the failure challenges present in Exascale systems, which are anticipated to experience various kinds of faults many times per day. This paper lays out several categories of errors, including silent and transient, that will have to be dealt with at the Exscale. It also identifies a range of error sources, including software issues. The paper goes on to identify four major research topics that need to be explored to enable Exascale. Hence, this white paper synthesizes the motivations, observations and research issues considered as determinant of several complimentary experts of HPC in applications, programming model, distributed system and system

References

- [ALR04] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr, "Basic Concepts and Taxonomy of Dependable and Secure Computing," IEEE Transactions on Dependable and Secure Computing, vol. 1, pp. 11-33, 2004.



- [BGD08] A. Bouteiller, G. Bosilca, and J. Dongarra, « Redesigning the message logging model for high performance », in ISC 2008, International Supercomputing Conference, Dresden, Germany, June 17-20, 2008.
- [BK95] Blum, M. and Kannan, S. 1995. Designing programs that check their work. J. ACM 42, 1 (Jan. 1995), 269-291.
- [BLCR] <http://ftg.lbl.gov/CheckpointRestart/CheckpointRestart.shtml>
- [BMP03] G. Bronevetsky, D. Marques, K. Pingali, and P. Stodghill. « C3: A System for Automating Application-level Checkpointing of MPI Programs », in Proceedings of the 16th International Workshop on Languages and Compilers for Parallel Computing, October 2003 (LCPC 2003).
- [Ch08] Z. Chen, « Extending algorithm-based fault tolerance to tolerate fail-stop failures in high performance distributed environments », in proceedings of IEEE Parallel and Distributed Processing Symposium, Page(s):1 – 8, April 2008
- [Cha08] C. Wang, F. Mueller, C. Engelmann, S. L. Scott, “Proactive Process-Level Live Migration in HPC Environments”, Proceeding of Supercomputing 2008, Tampa.
- [CIFT] <http://www.mcs.anl.gov/research/cifts/index.php>
- [CL85] K.M. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. ACM Transactions on Computer Systems, Vol. 3, No. 1, pp 63-75, 1985
- [CMK05] Sayantan Chakravorty, Celso Mendes and L. V. Kale, "Proactive Fault Tolerance in Large Systems", HPCRI Workshop in conjunction with HPCA 2005, 2005.
- [CR04] Charn-ga Lu, Daniel A. Reed, “Assessing Fault Sensitivity in MPI Applications”, in Proceedings of the 2004 ACM/IEEE conference on Supercomputing, 2004
- [CSCL] http://www.cs.wisc.edu/condor/manual/v6.8/4_2Condor_s_Checkpoint.html
- [Dij74] E.W. Dijkstra, “Self-stabilizing systems in spite of distributed control”. Communication of the ACM, vol 17, num 11, 1974
- [EAW02] E. N. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson, « A survey of rollback-recovery protocols in message-passing systems », ACM Computing Surveys, 34(3):375–408, September 2002.
- [EG05] C. Engelman, A. Geist, “Super-Scalable Algorithms for Computing on 100,000 Processors”, in Proceedings of the International Conference on Computational Science, May 2005.
- [FT-MPI] <http://icl.cs.utk.edu/ftmpi/>
- [GCG07] J. N. Glosli, D. F. Richards, K. J. Caspersen, R. E. Rudd, J. A. Gunnels, F. H. Streitz “Extending stability beyond CPU millennium: a micron-scale atomistic simulation of Kelvin-Helmholtz instability, A Micron-Scale Atomistic Simulation of Kelvin-Helmholtz Instability”, in Proceedings of the 2007 ACM/IEEE conference on Supercomputing, Reno, 2007.
- [GE07] A. Geist, C. Engelmann, « Development of Naturally Fault Tolerant Algorithms for Computing on 100,000 Processors », <http://www.csm.ornl.gov/~geist/Lyon2002-geist.pdf>
- [HA84] K. Huang and J. Abraham, “Algorithm-Based Fault Tolerance for Matrix Operations”, IEEE Trans. on Computers, Vol. C-33, No. 6, pp. 518-528, June 1984
- [KR07] William Kramer, et al. "Report of the Workshop on Petascale Systems Integration for Large Scale Facilities." LBNL Technical Report Number 63538, Lawrence Berkeley National Laboratory, 2007, <http://repositories.cdlib.org/lbnl/LBNL-63538/>.
- [KR08] W. Kramer, « PERCU: A Holistic Method for Evaluating High Performance Computing Systems », A Dissertation, University of California Berkeley, October 2008.
- [LAM/MPI] <http://www.lam-mpi.org/>
- [Li02] Gui-Rong Liu, “Mesh Free Methods: Moving Beyond the Finite Element Method”, CRC Press, 2002, ISBN 0849312388
- [Libckpt] <http://www.cs.utk.edu/~plank/plank/www/libckpt.html>
- [Lu05] C. D. Lu, « Scalable Diskless Checkpointing for Large Parallel Systems », Ph.D. dissertation, University of Illinois at Urbana-Champaign, 2005



- [MPICH-V] G. Bosilca, A. Bouteiller, F. Cappello, S. Djilali, G. Fédak, C. Germain, T. Héroult, P. Lemarinier, « MPICH-V: Toward a scalable fault tolerant MPI for volatile nodes », In Proceedings of SuperComputing 2002. IEEE, Nov., 2002 : <http://mpich-v.lri.fr/>
- [MVAPICH] <http://mvapich.cse.ohio-state.edu/overview/mvapich/>
- [NA03] K. Nagaraja, X. Li, R. Bianchini, R. Martin, and T. D. Nguyen. "Using fault Injection and Modeling to Evaluate the Performability of Cluster Based Services." *Proceedings fo teh 4th USENIX Symposium on Internet Technologies and Systems*. Seattle, WA, March 2003.
- [OpenMPI] <http://www.open-mpi.org/>
- [OS07] A. Oliner, and J. Stearley, « What Supercomputers Say: A Study of Five System Logs », in Proceedings of the International Conference on Dependable Systems and Networks (DSN). 2007.
- [PLP98] J. Plank, Li, K., M. Puening, « Diskless checkpointing », IEEE Transaction on Parallel Distributed Systems, 9(10):972–986, 1998.
- [PYK99] J. S. Plank, Y. Chen, Kai Li, M. Beck, G. Kingsley. "Memory Exclusion: Optimizing the Performance of Checkpointing Systems". *Journal Software: Practrice and Experience*, John Wiley & Sons, 29(2): 125-142, 1999
- [QTSZ05] F. Qin, J. Tucek, J. Sundaresan and Y. Zhou. Rx: Treating bugs as allergies---a safe method to survive software failure. In proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP'05), October 2005
- [Sah02] R. K. Sahoo, M. Bae, R. Vilalta, J. Moreira, S. Ma, M. Gupta, "Providing Persistent and Consistent Resources through Event Log Analysis and Predictions for Large-scale Computing Systems", in Proceedings of IEEE/ACM Supercomputing 2002.
- [Sah06] Y. Liang Y. Zhang Morris Jette, A. Sivasubramaniam Ramendra Sahoo, "BlueGene/L Failure Analysis and Prediction Models", in Proceedings of IEEE DSN 2006.
- [Sat06] S. Chakravorty, C. L. Mendes, L. V. Kale, Proactive Fault Tolerance in MPI Applications via Task Migration, In Proceedings of HIPC 2006, LNCS volume 4297, page 485
- [Sco09] S. Scott, C. Engelmann, G. Vallee, T. Naughton, A. Tikotekar, G. Ostrouchov, C. Leangsuksun, N. Naksinehaboon, R. Nassar, M. Paun, F. Mueller, C. Wang, A. Nagarajan, J. Varma ,A Tunable Holistic Resiliency Approach for High-Performance Computing Systems. Poster at the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP) 2009, Raleigh, NC, USA.
- [SCR] <http://scalablecr.sourceforge.net/>
- [SG07] B. Schroeder, G. Gibson, « Understanding Failures in Petascale Computers » in SciDAC 2007: Journal of Physics: Conference Series 78 (2007) 012022.
- [TNT06] R. Teodorescu, J. Nakano, J. Torrellas: SWICH: A Prototype for Efficient Cache-Level Checkpointing and Rollback. IEEE Micro 26(5): 28-40 (2006)
- [VN56] J. Von Neuman, Probabilistic logics and the synthesis of reliable organisms from unreliable components, in "Automata studies," edited by C. E. Shannon and J. McCarthy, Princeton University Press, 1956, pp. 43-98.
- [ZSK04] G. Zheng and L. Shi and L. V. Kale, "FTC-Charm++: An In-Memory Checkpoint-Based Fault Tolerant Runtime for Charm++ and MPI", 2004 IEEE International Conference on Cluster Computing, San Diego, CA, September, 2004. pp. 93-103.