

A tool for detecting transient faults in execution of parallel scientific applications on multicore clusters

Diego Montezanti¹, Enzo Rucci¹, Dolores Rexachs²,
Emilio Luque², Marcelo Naiouf¹ and Armando De Giusti^{1,3},

¹ III LIDI, Facultad de Informática, Universidad Nacional de La Plata
Calle 50 y 120, 1900 La Plata (Buenos Aires), Argentina
{dmontezanti, erucci, mnaiouf, degiusti}@lidi.info.unlp.edu.ar

² Computer Architecture and Operating Systems Department, Universitat
Autònoma de Barcelona

Campus UAB, Edifici Q, 08193 Bellaterra (Barcelona), Spain
{dolores.rexachs, emilio.luque}@uab.es

³ Consejo Nacional de Investigaciones Científicas y Tecnológicas

Abstract. Transient faults are becoming a critical concern among current trends of design of general-purpose multiprocessors. Because of their capability to corrupt programs outputs, their impact gains importance when considering long duration, parallel scientific applications, due to the high cost of re-launching execution from the beginning in case of incorrect results. This paper introduces SMCV tool which improves reliability for high-performance systems. SMCV replicates application processes and validates the contents of the messages to be sent, preventing the propagation of errors to other processes and restricting detection latency and notification. To assess its utility, the overhead of SMCV tool is evaluated with three computationally-intensive, representative parallel scientific applications. The obtained results demonstrate the efficiency of SMCV tool to detect transient faults occurrences.

Keywords: Transient fault, parallel scientific application, soft error detection tool, message content validation.

1 Introduction

The increase in the integration scale, in order to improve computing performance of current processors, as well as the growing size of the computer systems (towards upcoming exascale), are factors that make reliability an important issue. Particularly, transient faults, also named soft errors, are becoming a critical concern because of their capability to affect program correctness [1].

A transient fault is caused by interference from the environment, such as electromagnetic radiation, overheating or input power variations. It can alter signal transfers, register values, or some other processor component, temporarily inverting one or several bits of the affected hardware element [2]. Although short-lived transient faults do not cause permanent physical damage to the processor, depending on the moment or specific location of the occurrence, they may corrupt computations, resulting in either control flow faults or data faults that may propagate and cause

incorrect program execution [3][4]. Soft errors have led to costly failures in high-end systems in recent years [5][6].

The increasing number of transistors per chip involves lower voltage thresholds and higher internal operating temperatures. As a consequence, the vulnerability of the entire chip to transient faults (i.e. the soft error rate) is expected to increase significantly [7][8]. As soft errors can cause serious reliability problems, all general purpose microprocessors (especially those that form part of high availability systems) should employ fault-tolerance techniques to ensure right operation.

The impact of transient faults becomes more significant in the context of High Performance Computing (HPC). Since the year 2000, error reports due to transient faults in large computers or server groups have become more frequent [5][6]. Moreover the impact of the faults becomes more relevant in the case of long-duration applications, given the high cost of re-launching execution from the beginning. These factors justify the need for a set of strategies to improve the reliability of high-performance computation systems.

Historically, transient faults have been a design concern in critical environments, such as flight systems or high-availability servers. To face them, additional hardware is introduced, varying from watchdog co-processors to redundant hardware threads [9][10][11][12][13][14]. Storage devices, memories, caches have efficient built-in mechanisms such as Error Correcting Codes (ECC's) or parity bits, capable of detecting or even correcting this type of faults [4]. In practice, these techniques are costly or impossible to apply to processor elements [3] and they result inefficient in general purpose computers, due mainly to the high cost of designing, developing and verifying redundant custom hardware [1]. In this context, the faults that affect processor registers are a concern. In addition, as architectural trends point toward multicore designs, there is substantial interest in adapting such parallel hardware resources for transient fault tolerance.

To provide protection with lower (or zero) hardware costs, software-only approaches have been proposed [3][4]. Despite having some limitations (they have to execute additional instructions and are unable to examine microarchitectural state), software-only techniques have shown promise, in the sense that they can significantly improve reliability with reasonable performance overhead [15][16][17]. This characteristic makes software-redundancy-based strategies to be the most appropriate for general purpose computational systems.

Most software-duplication based techniques are designed for serial programs. From this standpoint, a parallel application can be viewed as a set of sequential processes that have to be protected from the consequences of transient faults adopting the software-based techniques.

MPI [18] is currently the de facto standard that defines an API for a message-passing parallel programming model. MPI is designed for achieving portable high-performance communication in parallel applications. However, while the current parallel computing systems are improving their robustness, the MPI specification does not fully exploit the capabilities of the current architectures [19][20].

Because the addition of reliability features in communication increases processing and resource overheads, MPI offers limited fault-handling facilities. Despite the fact that MPI processes may fail because of any external fault (e.g. processor, network or power failures), detection of such faults is not defined in the standard.

According to such scenario, in recent past SMCV methodology has been proposed [21], which is a software-only approach specifically designed for the detection of transient faults in message-passing parallel scientific applications that execute on multicore clusters.

In order to facilitate the usability of SMCV methodology, this paper presents SMCV tool, which is a library of modified MPI functions and data types with extended functionality for fault detection by comparison upon sending, message contents duplication upon reception, and concurrency control between replicas. SMCV tool has the goal of helping programmers and users of parallel scientific applications to achieve reliability in their executions, obtaining correct final results or, at less, reporting the silent fault occurrence and avoiding its consequences by leading to a safe-stop state. This avoids the unnecessary and costly wait until execution finishes, allowing application re-launching after a restricted delay due to latency of detection. This is an important feature, owing to the long duration executions of such applications.

To estimate the impact of SMCV tool on performance of message-passing parallel scientific applications, and in order to evaluate the convenience of its utilization, a set of experiments was made, using three benchmark parallel applications: matrix multiplication [22]; solution to Laplace's equation [23]; and DNA sequence alignment [24]. With these experiments, the performance of the tool was evaluated for various problem sizes using different number of processes, obtaining 93.7 maximum and 24.3 average percent overhead in absence of faults. As explained further on, at least two executions of the original application and final results comparison are needed to determine if a transient fault has occurred when no fault tolerance strategy is employed by the system. Accordingly, these results demonstrate the efficiency of SMCV tool.

The rest of this paper is organized as follows: Section 2 discusses related works. Section 3 reviews the theoretical context of transient faults. Section 4 and Section 5 describes SMCV methodology and SMCV tool respectively. In Section 6, the experimental work carried out is described, whereas Section 7 presents and analyzes the obtained results. Finally, Section 8 presents the conclusions and future lines of work in relation to this paper.

2 Related Works

Redundancy techniques can be broadly classified into two kinds: hardware-based and software-based. There have been various implementations of software-only, hardware-only, and hybrid techniques for transient fault mitigation [3][4].

All hardware-based approaches require the addition of some new hardware logic to meet redundancy requirements. Several researchers have also made use of multiplicity of hardware blocks readily available on multithreaded/multicore architectures to implement computation redundancy [10][11][12][14].

Fault tolerance based on software replication is a well-populated field with decades of history. Their main advantage is that they do not require any additional hardware. Among the purely software solutions, PLR [1] is a process replication-based one.

Other software-only techniques for transient fault detection are the compiler-based ones. At compile time, they insert redundant computations [16], control flow assertions [15] or both [4].

As regards to hybrid strategies, in [3], the authors propose a fault-tolerant typed assembly language, in an attempt to exploit the benefits of both hardware and software-based systems for fault tolerance.

All the previously mentioned proposals are designed for sequential applications. SMCV is specific for message-passing parallel scientific applications.

There are some approaches that extend MPI to implement process replicas on MPI applications for hard faults. MPI/FT [20] is an MPI-based middleware that provides additional services for detection and recovery of failed MPI processes. FT-MPI [19] specifies the semantics of a fault tolerant version of MPI and implements that specification. Whereas the two mentioned strategies provide support for failures that make a process to terminate, SMCV provides a mechanism for detecting transient faults in MPI applications improving at the same time system availability. No proposals for transient fault detection in parallel scientific applications based on message validation were found while researching for this work.

3 Background on soft errors

As aforementioned, transient faults affect system hardware elements, but their effects are observed on the program execution (assuming deterministic programs). According to these effects, they can be classified into the following categories:

- Latent Error (LE): also called benign fault, is a fault that corrupts data that are not used by the application so, despite the fault effectively happening, it does not propagate to affect the correctness of the execution and has no impact on the results.
- Detected Unrecoverable Error (DUE): is a detected error that has no possibility of recovery. DUEs are a consequence of faults that cause abnormal conditions that are detectable on some intermediate software layer level (e.g. Operating System, communication library). Normally, they cause the abrupt stop of the application.
- Time-Out Error (TO): due to fault, the program does not terminate within a given amount of time.
- Silent Data Corruption (SDC): is the alteration of data during the execution of a program that does not cause any abnormal condition and goes undetected by system software. Its effects are silently propagated to corrupt final results. This is the worst case, because the application appears to execute correctly but silently produce incorrect output [1].

4 SMCV Methodology Overview

SMCV is a detection strategy based on validating the contents of the messages to be sent in deterministic parallel scientific applications. In particular, SMCV intercepts

faults that produce TOs and SDCs. Under this approach, each application process is duplicated and the process and its replica run concurrently, which requires a synchronization mechanism. When a communication is to be performed (point-to-point or collective), the process temporarily stops execution and waits for its replica to reach the same point. Once there, all fields from the message to be sent are compared to validate that the contents of both threads are the same. Only if this proves true, one of the threads sends the message, ensuring that no corrupt data are propagated to other process. The recipient(s) of the messages stop upon reception and remain on hold. Once received, it copies the contents of the message to its replica and both processes continue with their computation. Finally, when application execution finishes, the obtained results are checked to detect faults that may have occurred after communications ended, (i.e. the serial part of the application).

Figure 1 shows SMCV detection outline whereas Figure 2 shows the SMCV behavior in presence of transient faults. More details about SMCV methodology can be found in [21].

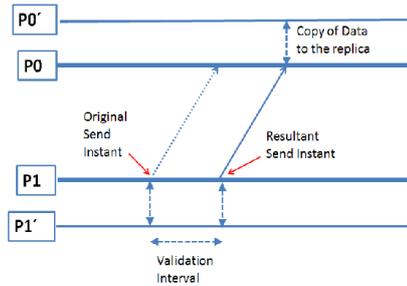


Fig. 1. SMCV detection outline.

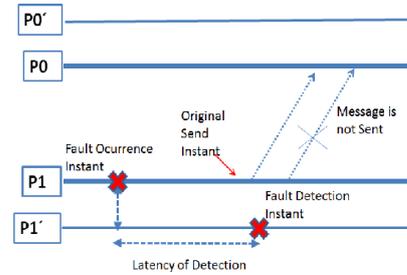


Fig. 2. SMCV behavior in presence of transient faults.

5 SMCV Tool

5.1 Description

To implement SMCV methodology, SMCV tool was developed. It consists of a library of modified MPI functions and data types that can be used in MPI applications developed using C language. SMCV library redefines MPI functions and data types with only on syntactic change (the MPI prefix is replaced with SMCV). In turn, it adds two new functions: `SMCV_Call` and `SMCV_Validate`. For threads replication and synchronization, Pthreads functions are used. MPI functions redefinition is necessary to provide transient fault detection in a transparent way to applications code and their programmers. This implies application source code modification and recompiling.

5.2 Basic Functions

MPI standard defines six basic functions [25]. The SMCV library core consists of the six redefined MPI basic functions and two other. These eight functions are enough to develop a wide range of parallel applications that are able to detect transient faults. SMCV basic functions are described below:

SMCV_Init. Initiate a SMCV environment.

SMCV_Finalize. Terminate a SMCV environment.

SMCV_Comm_size. Determine number of processes.

SMCV_Comm_rank. Determine process identifier.

SMCV_Call. Create a new thread that executes the code to be validated.

SMCV_Send. Synchronize the process and its replica. The second to reach the synchronization point compares all the fields of the message to be sent (byte to byte). If all fields match, the first thread sends the message. Once sent, both threads continue with their execution. If any field differs, a safe-stop is produced because a SDC has occurred. Moreover, there is a (configurable) time for the second thread to reach the synchronization point, in order to be able to intercept TOs.

SMCV_Recv. Synchronize the process and its replica. The first to reach the synchronization point receives the message and remains on hold. When the second thread arrives, it copies the contents of the message received. After that, both threads continue with their execution. Like `SMCV_Send`, there is a (configurable) time for the second thread to reach the synchronization point, in order to be able to intercept TOs.

SMCV_Validate. Synchronize the process and its replica. The second to reach the synchronization point compares both threads' final result (byte to byte). If the final results match, the threads continue with their execution. Otherwise, a safe-stop is produced because a SDC has occurred. Like `SMCV_Send`, there is a (configurable) time for the second thread to reach the synchronization point, in order to be able to intercept TOs.

5.3 Usage

The next steps must be followed to incorporate SMCV features in MPI application code:

1. Replace MPI header with SMCV header.
2. Encapsulate the code to be validated (data and instructions) in a `void *` function.
3. Make a call to `SMCV_Call` function passing the previously defined function to it as an argument.
4. Replace MPI prefix with SMCV in all MPI functions and data types.
5. Make a call to `SMCV_Validate` in order to validate the application final result.

Figure 3 shows an example of how to adapt an MPI application in order to incorporate SMCV features.

<pre> #include <mpi.h> int main (int argc, char **argv) { MPI_Init(); /* Process data, instructions and MPI functions */ MPI_Finalize(); return 0; } </pre>	<pre> #include <smcv.h> int main (int argc, char **argv) { SMCV_Init(); SMCV_Call(&smcv_process) SMCV_Finalize(); return 0; } void * smcv_process () { /* Thread data, instructions and SMCV functions */ SMCV_Validate(); } </pre>
---	---

Fig. 3. Example of how to adapt a MPI application in order to incorporate SMCV features. Left: MPI application source code. Right: SMCV-adapted MPI application source code.

6 Experimental Work

6.1 Architecture Used

Experimental work was carried out on a cluster of Blade multicores with four blades. Each blade has two quad core Intel Xeon e5405 2.0GHz processors with 6Mb L2 cache (shared between pairs of cores) and 10 Gb RAM memory (shared between both processors). The operating system is GNU/Linux Debian 6.0.7 (64 bits, kernel version 2.6.32) and the MPI library used is OpenMPI (version 1.6.4).

6.2 Benchmark Applications Used

Three benchmark parallel applications were selected: matrix multiplication [22]; solution to Laplace's equation [23]; and DNA sequence alignment [24]. These benchmark applications were selected because of three main reasons: first, they are well-known, representative scientific applications; second, they are computationally intensive; and third, they have three different communication patterns: Master-Worker, Single-Program-Multiple-Data (SPMD) and Pipeline, respectively.

Tests were carried out using MPI and SMCV versions of the three selected benchmark applications. The steps described in Subsection 5.3 were followed to incorporate SMCV features to original applications' codes. Finally, because SMCV was especially designed to be used in context of HPC applications, the `-O` optimization level was used at compile time.

6.3 Tests Carried Out

Benchmark applications were tested using different number of processes: $P=\{4, 8, 16\}$. Various problem sizes were used for each application: $N=\{2048, 4096, 8192,$

16384} for matrix multiplication; $N=\{4096, 8192, 16384\}$ for solution to Laplace's equation and $N=\{65536, 131072, 262144, 524288\}$ for DNA sequence alignment. At most four processes were mapped by node, which means that in original applications execution only four cores of each node were used. In the case of SMCV applications, all the cores of each node were used (the replicas execute on available cores). Each experiment was run five times and the results were averaged to improve stability.

7 Results

To assess the incidence of SMCV tool over the applications performance when escalating the problem and/or the architecture, the *Overhead* metric is analyzed. The overhead is a consequence of the processes duplication, the synchronization with the replicas, the comparison and duplication of the messages contents and the final validation of the results. In addition, the processes duplication increases contention for system resources. Equation 1 indicates how to calculate this metric, where APP_ET is the original application execution time and $SMCV_APP_ET$ is the SMCV-adapted application execution time.

$$Overhead = \frac{(SMCV_APP_ET - APP_ET)}{APP_ET} \times 100 . \quad (1)$$

Figures 4, 5 and 6 shows the overheads obtained with SMCV applications (matrix multiplication, solution to Laplace's equation and DNA sequence alignment, respectively) for various problem sizes using different number of processes.

The charts show that the three benchmark applications present similar behaviors. As it can be observed, overhead decreases as the problem size grows. This is due to, with larger problem sizes, applications spend more time computing than communicating and, consequently, the time required to synchronize threads and to duplicate and validate message contents reduces (in the case of matrix multiplication, data duplication produces disk-swapping when $N=16384$ and $P=\{8,16\}$ and, as a consequence, overhead reduction does not remain). On the other hand, the number of messages to be sent increases as the number of processes grows. This leads to an overhead increase because time required to synchronize threads and to compare and duplicate message contents enlarges.

As mentioned above, overhead behaviors are similar, but the same does not occur with overhead values. Matrix multiplication is the application with largest overhead values. This is due to the sizes of the messages that processes send (matrix sizes go from 16MB to 1GB according to N), aggravated by the fact that they use collective communication operations for it. Unlike OpenMPI, SMCV library does not optimize this kind of communication operations [26]. Last, the final result of this application is a matrix and the time required to validate it is not insignificant.

Overhead values of the solution to Laplace's equation are lower than the corresponding ones to matrix multiplication. Even though processes repeatedly interchange messages (which increases the number of synchronizations), the time required to validate them reduces because of the smaller message size (they go from 16KB to 64KB depending on N). Another influence factor is that the final result is a single number and, in consequence, the time necessary to validate it is negligible.

DNA sequence alignment presents overhead values even lower than the corresponding ones to the solution to Laplace's equation. All the processes receive and send messages repeatedly (except the first and the last of the pipeline). Because of these messages are of fixed size and very small (136B), the time required to validate them is not significant. Like the previous case, final result validation does not demand considerable time.

In this set of experiments, SMCV tool provides fault detection with 93.7 maximum and 24.3 average percent overhead. This represents an advantage with respect to the original execution, which has to be repeated (and final results have to be compared) to ensure a correct output if a SDC does not occur. Moreover, if a SDC occurs, a third re-execution (and a new comparison) is required to pick the outputs of the runs that form a majority as the correct ones.

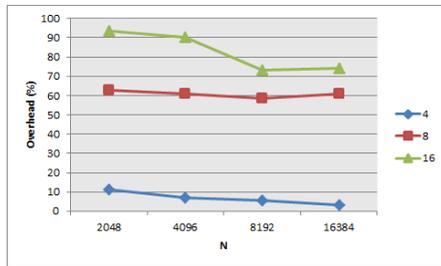


Fig. 4. Overheads obtained for SMCV-matrix multiplication for various problem sizes using different number of processes.

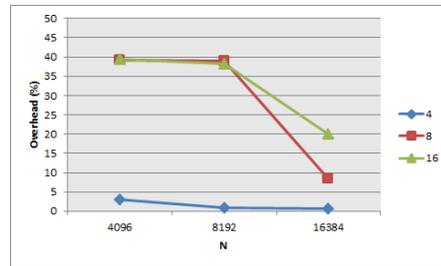


Fig. 5. Overheads obtained for SMCV-solution to Laplace's equation for various problem sizes using different number of processes.

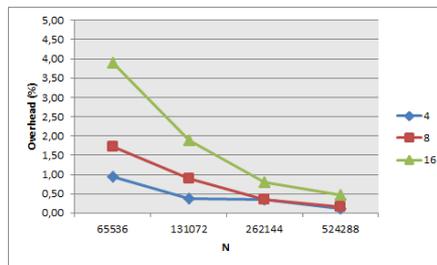


Fig. 6. Overheads obtained for SMCV-DNA sequence alignment for various problem sizes using different number of processes.

8 Conclusions and Future Work

Transient faults are becoming more frequent in large computers and their impact is higher in the case of long duration applications. In this paper, SMCV tool is presented to help programmers and users of scientific parallel applications to achieve reliability in their executions, obtaining correct final results or, at less, reporting the silent fault

occurrence within a limited time lapse and leading to a safe-stop state. Experimental results show that, when running three different benchmark parallel applications on a multicore cluster for various problem sizes and using different number of processes, SMCV tool provides fault detection with 93.7 maximum and 24.3 average percent overhead. These results demonstrate the tool's efficiency to provide transient fault detection in message-passing parallel scientific applications.

Future lines of work focus on four aspects:

- Extending current SMCV library implementation to give full support to MPI applications (at the moment it only supports blocking communication functions and some collective communication routines).
- Optimizing collective communications implementation to take benefit of MPI features, in order to minimize overheads.
- Automating the procedure to adapt the original application source code to use SMCV tool.
- Emulating non-deterministic functions, to extend SMCV methodology for giving support to transient fault detection in non-deterministic MPI scientific applications.

References

1. Shye, A., Blomstedt, J., Moseley, T., Reddi, V. J., Connors, D. A.: PLR: A software approach to transient fault tolerance for multicore architectures; *IEEE Transactions on Dependable and Secure Computing*. 6(2), pp. 135--148 (2009)
2. Wang, N. J., Quek, J., Rafacz, T. M., Patel, S. J.: Characterizing the Effects of Transient Faults on a High-Performance Processor Pipeline. In: *Proceedings of the International Conference on Dependable Systems and Networks*, pp. 61--70. IEEE Press, Florence (2004)
3. Perry, F., Mackey, L., Reis G. A., Ligatti, J., August, D. I., Walker, D.: Fault-tolerant typed assembly language. In: *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, pp. 42--53. ACM Press, San Diego (2007)
4. Reis, G. A., Chang, J., Vachharajani, N., Rangan, R., August, D. I.: SWIFT: Software Implemented Fault Tolerance. In: *Proceedings of the International Symposium on Code generation and optimization*, pp. 243--254. IEEE Press, Washington DC (2005)
5. Baumann, R. C.: Soft errors in commercial semiconductor technology: Overview and scaling trends. In: *IEEE 2002 Reliability Physics Tutorial Notes, Reliability Fundamentals*, pp. 121 01.1--121 01.14.
6. Michalak, S. E., Harris, K. W., Hengartner, N. W., Takala, B. E., Wender, S. A.: Predicting the number of fatal soft errors in Los Alamos National Laboratory's ASC Q computer; *IEEE Transactions on Device and Materials Reliability*. 5(3), pp. 329--335 (2005)
7. Gramacho, J., Rexachs del Rosario, D., Luque, E.: A Methodology to Calculate a Program's Robustness against Transient Faults. In: *Proceedings of the International 2011 Conference on Parallel and Distributed Processing Techniques and Applications*, pp. 645--651. WorldComp Press, Las Vegas (2011)
8. Mukherjee, S.; Weaver, C.; Emer, J.; Reinhardt, S., Austin, T.: A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor.

- In: Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture, pp. 29--40. IEEE Press, San Diego (2003)
9. Mahmood, A., McCluskey, E. J.: Concurrent error detection using watchdog processors-a survey. *IEEE Transactions on Computers*. 37(2), pp. 160--174 (1988)
 10. Reinhardt, S. K., Mukherjee S. S.: Transient Fault Detection via Simultaneous Multithreading. In: Proceedings of the 27th annual International Symposium on Computer Architecture, pp. 25--36. IEEE Press, Vancouver (2000)
 11. Kontz M., Reinhardt S. K., Mukherjee S. S.: Detailed Design and Evaluation of Redundant Multithreading Alternatives. In: Proceedings of the 29th Annual International Symposium on Computer Architecture, pp. 99--110. IEEE Press, Anchorage (2002)
 12. Vijaykumar T. N., Pomeranz, I. Cheng, K.: Transient-Fault Recovery using Simultaneous Multithreading. In: Proceedings of the 29th Annual International Symposium on Computer Architecture, pp. 87--98. IEEE Press, Anchorage (2002)
 13. Gomaa M., Scarbrough C., Vijaykumar T. N., Pomeranz, I.: Transient-Fault Recovery for chip Multiprocessors. In: Proceedings of the 30th Annual International Symposium on Computer Architecture, pp. 98--109. IEEE Press, San Diego (2003)
 14. Rotenberg E.: AR-SMT: A Microarchitectural Approach to Fault Tolerance in Microprocessors. In: Proceedings of the 29th Annual International Symposium on Fault-Tolerant Computing, pp. 84--91. IEEE Press, Wisconsin (1999)
 15. Oh, N., Shirvani, P. P., McCluskey, E. J.: Control-flow checking by software signatures. *IEEE Transactions on Reliability*, 51(1), pp. 111--122 (2002)
 16. Oh, N., Shirvani, P. P., McCluskey, E. J.: Error detection by duplicated instructions in super-scalar processors; *IEEE Transactions on Reliability*. 51(1), pp. 63--75 (2002)
 17. Reis, G. A., Chang, J., August, D. I.: Automatic instruction level software-only recovery methods; *IEEE Micro Top Picks*. 27 (1), pp. 36--47 (2007)
 18. Message Passing Interface Forum, <http://www.mpi-forum.org/>
 19. Fagg, G.E., Gabriel, E., Chen, Z., Angskun, T., Bosilca, G., Pjjesivac-Grbovic, J., Dongarra, J.J.: Process Fault-Tolerance: Semantics, Design and Applications for High Performance Computing; *International Journal of High Performance Applications*. 19(4), pp. 465--478 (2005)
 20. Batchu, R., Dandass, Y., Skjellum, A., Beddhu, M.: MPI/FT: A Model-Based Approach to Low-Overhead Fault Tolerant Message-Passing Middleware; *Cluster Computing*. 7 (4), pp. 303--315 (2004)
 21. Montezanti, D., Frati, F.E., Rexachs, D., Luque, E., Naiouf, M.R., De Giusti, A.: SMCV: a Methodology for Detecting Transient Faults in Multicore Clusters.; *CLEI Electron. J*. 15(3), pp. 1--11 (2012)
 22. Leibovich, F., Gallo, S., De Giusti, A., De Giusti, L., Chichizola, F., Naiouf, M.: Comparación de paradigmas de programación paralela en cluster de multicores: pasaje de mensajes e híbrido. In: *Anales del XVII Congreso Argentino de Ciencias de la Computación*. pp. 241--250. Editorial RedUNCI, La Plata (2011)
 23. Andrews, G.: *Foundations of Multithreaded, Parallel, and Distributed Programming*. Addison Wesley Longman, EEUU (2000).
 24. Rucci, E., Chichizola, F., Naiouf, M., De Giusti, A.: Parallel Pipelines for DNA Sequence Alignment on Cluster of Multicores. A comparison of communication models.; *Journal of Communication and Computer*. 9(12), pp. 516--522 (2012)
 25. Dongarra, J., Foster, I., Fox, G., Gropp, W., Kennedy, K., Torczon, L., White, A.: *The Sourcebook of Parallel Computing*. Morgan Kauffman, EE.UU. (2003)
 26. Graham, R., Shipman, G.: MPI Support for Multi-core Architectures: Optimized Shared Memory Collectives. In: Proceedings of the 15th European PVM/MPIUsers' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface. pp. 130--140. Springer-Verlag Berlin (2008)