Parallel Fault Tolerant Algorithms for Parabolic Problems

Hatem Ltaief, Marc Garbey, Edgar Gabriel

Department of Computer Science, University of Houston
4800 Calhoun Road, Houston, TX 77204, USA
{ltaief, garbey, gabriel}@cs.uh.edu

Abstract. With increasing number of processors available on nowadays high performance computing systems, the mean time between failure of these machines is decreasing. The ability of hardware and software components to handle process failures is therefore getting increasingly important. The objective of this paper is to present a fault tolerant approach for the implicit forward time integration of parabolic problems using explicit formulas. This technique allows the application to recover from process failures and to reconstruct the lost data of the failed process(es) avoiding the roll-back operation required in most checkpoint-restart schemes. The benchmark used to highlight the new algorithms is the two dimensional heat equation solved with a first order implicit Euler scheme.

1 Introduction

Today's high performance computing (HPC) systems offer to scientists and engineers powerful resources for scientific simulations. At the same time, the reliability of the system becomes a paramount key: systems with tens of thousands of processors face inherently a larger number of hardware and software failures, since the mean time between a failure is related to the number of processors and network interface cards (NICs). This is not necessarily a problem for short running application utilizing a small/medium number of processors, since re-running the application in case a failure occurs does not waste a large amount of resources. However, for long running simulations requiring many processors, aborting the entire simulation just because one processor has crashed is often not an option, either because of the significant amount of resources being involved in each run or because the application is critical within certain areas.

Nowadays, a single failing node or processor on a large HPC system does not imply, that the entire machine has to go down. Typically, the parallel application utilizing this node has to abort, all other applications on the machine are not affected by the hardware failure. The reason that the parallel application, which utilized the failed processor, has to abort is mainly because the most widespread parallel programming paradigm MPI [1], is not capable of handling process failures. Several approaches how to overcome this problem have been proposed, most of them relying on some forms of checkpoint-restart [2, 3]. While these solutions require few modifications of the application source code, checkpoint-restart has inherent performance and scalability limitations. Another approach suggest by Fagg et. all [4] defines extensions to the MPI specification giving the user the possibility to recognize, handle and recover from process failures. This approach does not have built-in performance problems, requires however certain changes in the source code, since it is the responsibility of the application to recover the data of the failed processes.

In the last couple of years, several solutions have been proposed how to extend numerical applications to handle process failures on the application level. Geist et al. suggest a new class of so-called naturally fault tolerant algorithms [6] based on mesh-less methods and chaotic relaxation. In-memory checkpointing techniques [7] avoid expensive disk I/O operations by storing regular checkpoints in the main memory of neighbor/spare processes. In case an error occurs, the data of the failed processes can be reconstructed by using these data items. However, the application has to roll-back to the last consistent distributed checkpoint, loosing all the subsequent work and adding a significant overhead for applications running on thousands of processors due to coordinated checkpoints. Further, while it is fairly easy to recover numerically from a failure with a relaxation scheme applied to an elliptic problem, the problem is far more difficult with the time integration of a parabolic problem. As a matter of fact the integration back in time is a very ill-posed problem. Further time integration of unsteady problem may run for very long time and are more subject to process failures.

In this paper, we concentrate on the heat equation problem that is a representative test case of the main difficulty and present a new explicit recovery technique which avoid the roll-back operation and is numerically efficient.

The paper is organized as follows: section 2 defines our test-system and describes two different fault tolerant algorithms. Section 3 discusses implementation issues with respect to the communication and checkpointing scheme applied in our algorithms. Section 4 presents some results for the recovery operation. Finally, section 5 summarizes the results of this paper and presents the ongoing work in this area.

2 Description of the Fault Tolerant Algorithms

The work presented in this paper is based on the Fault Tolerant MPI (FT-MPI) framework developed at the University of Tennessee. FT-MPI extends the MPI specification by giving applications the possibility to discover process failures. Furthermore, several options how to recover from a process failure are specified: the application can either continue execution without the failed processes (COMM_MODE_BLANK) or replace them (COMM_MODE_REBUILD). The current implementation of the specification is based on the HARNESS framework [5]. HARNESS provides an adaptive, reconfigurable runtime environment, which is the basis for the services required by FT-MPI. While FT-MPI is capable of surviving the simultaneous failing of $n-1$ processes in an n processes job, it remains up to the application developer to recover the user data, since FT-MPI does not perform any (transparent) checkpointing of user-level data items.

2.1 Definition of the Problem

The model problem used throughout the paper is the two dimensional heat equation as given by

$$\frac{\partial u}{\partial t} = \Delta u + F(x,y,t),\ (x,y,t) \in \Omega \times (0,T),\ u_{|\partial\Omega} = g(x,y),\ u(x,y,0) = u_o(x,y). \tag{1}$$

We suppose that the time integration is done by a first order implicit Euler scheme, $\frac{U^{n+1}-U^n}{dt} = \Delta U^{n+1} + F(x,y,t^{n+1})$, and that Ω is partitioned into N subdomains $\Omega_j,\ j = 1..N$.

For the sake of simplicity, the explanations in the paper will be restricted to the one dimensional heat equation problem $\Omega = (0,1)$, discretized on a regular cartesian grid, which leads to

$$\frac{U_j^{n+1} - U_j^n}{dt} = \frac{U_{j+1}^{n+1} - 2U_j^{n+1} + U_{j-1}^{n+1}}{h^2} + F_j^{n+1}. \tag{2}$$

Furthermore, we assume that $dt \sim h$. In case process j fails, the most recent values for U_j are not available for continuing the computations, assuming that the runtime environment can survive process failures. On each *up and running* process the last computed solution is still available. The goal of the approach presented in the paper is therefore to design an algorithm which reconstructs the solution of the failed process(es) efficiently based on the checkpointed data.

The general fault tolerant approach is based on periodic checkpoints of the local data, e.g. to persistent disks or spare processes. Furthermore, processes are not coordinated for the checkpointing procedure for performance reasons, e.g. each process might save its local data at different time steps. As soon as a process failure occurs, the runtime environment will report it through a specific error code to the application. The application initiates the necessary operations to recover first the MPI environment and replace the failed process(es). In a second step, the application has to ensure, that the data on the replacement processes is consistent with the other processes. For this, the last checkpoint of the failed processes has to be retrieved. However, since the checkpoint of each process might have been taken at a different time step, this data does not yet provide a consistent state across all processes. Therefore, we discuss two mathematical methods based on time integration for constructing a consistent state from the available, inconsistent checkpoints. This difficulty is characteristic of a time dependent problem with no easy reversibility in time.

Figure 1 gives an example for the status of different process(es) after re-spawning a failed process and retrieving the last available checkpoint for this process. The thick lines represent the available data from which the recovery procedure will start. The circle lines correspond to the lost solution which we are trying to retrieve mathematically. The dashed lines are the boundary interfaces between subdomains. In the following, we will review two numerical methods to reconstruct a uniform approximation of U^M at a consistent time step M on the entire domain Ω.

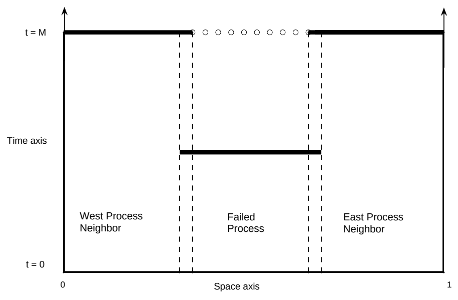

Fig. 1. Available data on main memory processors before starting the reconstruction algorithms

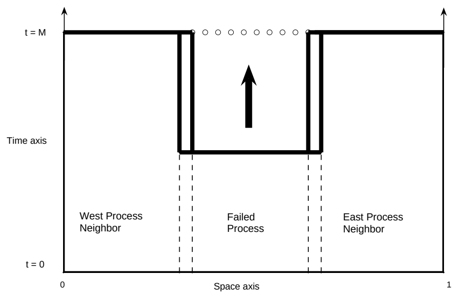

Fig. 2. Reconstruction procedure in one dimension using forward time integration.

2.2 The Forward Implicit Reconstruction

For the first approach, the application process j has to store every K time steps its current solution $U_j^{n(j)}$. Additionally, the artificial boundary conditions $I_j^m = \Omega_j \cap \Omega_{j+1}$ have to be stored for all time steps $m < M$ since the last checkpoint. The solution U_j^M can then be reconstructed with the forward time integration (2). Figure 2 demonstrates how the recovery works. The vertical thick lines represent the boundary data that need to be stored, and the intervals with circles are the unknowns of the reconstruction process.

The major advantage of this method is that it is using the same algorithm as in the standard domain decomposition method. The only difference is, that it is restricted to some specific subsets of the domain. Thus, the identical solution U_j^M as if the process had no failures can be reconstructed. The major disadvantage of this approach is the increased communication volume and frequency. While checkpointing the current solution $U_j^{n(j)}$ is done every K time steps, the boundary values have to be saved each time step for being able to correctly reconstruct the solution of the failed process(es).

2.3 The Backward Explicit Reconstruction

This method does not require the storage of the boundary conditions of each subdomain at each time step, but it allows to retrieve the interface data by computation instead. For this, the method requires the solution for each subdomain j at two different time steps, $n(j)$ and M, with $M - n(j) = K > 0$. The solution at time step M is already available on each *running* process after the failure occurred. The solution at time step $n(j)$ corresponds to the last solution on subdomain j saved to the spare memory before the failure happened. This is

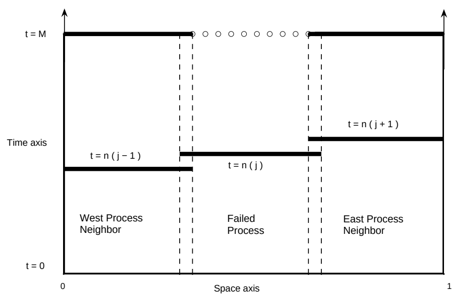

Fig. 3. Available data on main memory processors before starting the reconstruction algorithms.

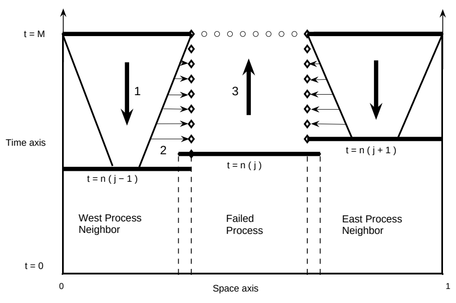

Fig. 4. Reconstruction procedure in one dimension using explicit backward time stepping.

the starting point for the *local* reconstruction procedure. In this approach, only the replacement of the crashed process and its neighbors are involved (figure 3) in the reconstruction. The reconstruction process is split in two explicit numerical schemes. The first one comes from the Forward Implicit scheme (2) which provides an explicit formula when going backward in time:

$$U_j^n \;=\; U_j^{n+1} \;-\; dt\;\frac{U_{j+1}^{n+1} - 2U_j^{n+1} + U_{j-1}^{n+1}}{h^2} \;-\; F_j^{n+1}. \tag{3}$$

The existence of the solution is granted by the forward integration in time. Two difficulties arise: first, the numerical procedure is instable, and second, one is restricted to the cone of dependence (Step 1 in figure 4). We have in Fourier modes $\hat{U}_k^n \;=\; \delta_k\ \hat{U}_k^{n+1}$, with $\delta_k \;\sim\; -\frac{2}{h}(\cos(k\ 2\ \pi\ h) - 1),\ \ |k| \;\leq\; \frac{N}{2}$. The expected error is at most in the order $\frac{\nu}{h^L}$ where ν is the machine precision and L is the number of time steps which we can compute backwards. Therefore, the backward time integration is still accurate up to time step L with $\frac{\nu}{h^L} \sim h^2 \Longleftrightarrow L \sim \frac{\log \nu}{\log h} - 2$. Then, the precision may deteriorate rapidly in time. Thus, to stabilize the scheme, one can use an hyperbolic regularization such as on the telegraph equation. Further details regarding this result can be found in [9, 10].

To construct the solution outside the cone of dependencies and therefore to determine the solution at the subdomain interface, we used a standard procedure in inverse heat problem, the so-called space marching method [8] (Step 2 in figure 4). This method is second order but may require a regularization procedure of the solution obtained inside the cone using the product of convolution $\rho_\delta\ *$

$u(x\ ,\ t)$, where $\rho_\delta = \frac{1}{\delta\sqrt{\pi}}\exp(-\frac{t^2}{\delta^2})$. The space marching scheme is given by:

$$\frac{U_{j+1}^n\ -\ 2\,U_j^n\ +\ U_{j-1}^n}{h^2} = \frac{U_j^{n+1} - U_j^{n-1}}{2\ dt} + F_j^n. \qquad (4)$$

Equation 4 is unconditionally stable, given that $\delta \geq \sqrt{\frac{2\ dt}{\pi}}$. The neighbors of the failed processes apply these two methods successively. At the end of the procedure, these processes are able to provide to the replacement of the crashed process the artificial boundary conditions. Then, the respawned process can rebuild its lost data using the forward time integration as shown in section 2.2 (Step 3 in figure 4). The backward explicit time integration is well known to be an ill-posed problem and works only for few time steps. Indeed, for example if we assume $\nu = 10^{-12}$ and $h = 0.05$, the solution computed may blow up for $L > 7$. This would be equivalent to set at most the frequency of backup time step to $K = 9$. We refer to [9] for more details on the accuracy of our numerical scheme. Let us mention also that neither the backward explicit scheme nor the space marching scheme are limited to Cartesian grids.

In the following, we would like to compare the communication and checkpointing overhead imposed by the two methods described up to now.

2.4 Performance Comparison of the Checkpointing Operations

The two methods described in the sections 2.2 and 2.3 have different requirements with respect to what data has to be checkpointed by each process. While the backward explicit scheme requires only the storage of the domain of each process every K time steps, the forward implicit scheme requires additionally saving the boundary values in every time step.

For the performance comparison between both methods, two different codes have been evaluated, one based on a two dimensional domain decomposition and a code using a three dimensional domain decomposition. For the two-dimensional tests (1), we tested three different problem sizes per processor ($50*50$, $100*100$ and $200*200$) on four different processor configurations (9, 16, 25 and 36 processes). The processes are arranged in a regular two dimensional mesh. Each column of the processor-mesh has a separate checkpoint processor assigned to it. The data each checkpoint process receives is stored in their memory, avoiding therefore expensive disk I/O operations. More details to the checkpointing scheme are given in the section 3.

The cluster used for the 2-D testcase consisted of 154 Intel Itanium 2 processors with 4 GB of main memory and a Gigabit Ethernet interconnect. The results for the configuration described above are displayed in figures (5-8). The abscissa gives the checkpointing frequency in number of time steps between each checkpoint, while the ordinate shows the overhead compared to the same code and problem size without any checkpointing. As expected, the overhead is decreasing with increasing distance between two subsequent checkpoints. Furthermore, saving the boundary conditions each time step adds only a negligible overhead, especially for the largest test case with 36 processes and $200 * 200$ problem size per process. However, for higher dimension problem, saving at each time step the artificial boundary conditions slows the code execution down significantly. While interface conditions are one dimension lower than the solution itself, the additional message passing is interrupting the application work and data flow. Figure 9 and 10 shows the checkpointing time cost on the 3D version of the model problem defined in (1) for a small $(50 * 50 * 50)$ and a larger problem size $(102 * 102 * 102)$. For the large problem, saving the boundary conditions with a backup frequency of ten time steps slows the application down dramatically on an Intel EM64T cluster with a Gigabit Ethernet network. As shown in 10, the overhead compared to the method not requiring to store the boundary condition can double in the worst case. Figure (5-6-7-8) for the two dimensional problem and figure (9-10) for the three dimensional problem make us confident, that saving the local solution each 9 time steps brings a small overhead (between 5% and 15%) on the overall execution time. Moreover, such numerical methods are very cheap in term of computation and very fast. Therefore, the focus of the following sections is on the implementation aspects of the backward explicit scheme.

3 Implementation Details

As described in section 2.4, the checkpointing infrastructure utilized in the 2-D test case is implemented by using two groups of processes: a solver group composed by processes which will only solve the problem itself and a spare group of processes whose main function is to store the data from solver processes using local asynchronous checkpointing and non-blocking communications.

The communication between the solver processes and the checkpointing processes is handled by an inter-communicator. Since it does not make sense to have as many checkpointing processes as solver processes, the number of spare processes is equal to the number of solver processes in the x-direction. Thus, while the solver processes are arranged in the 2-D cartesian topology, the checkpoint processes are forming a 1-D cartesian topology.

Figure 11 gives a geometrical representation of the two groups with a local numbering. This figure shows furthermore, how the local checkpointing is applied for a configuration of 16 solver processes and 4 spare processes. Each spare process is in charge of storing the data of a single subgroup, depicted by the ellipses in figure 11. To further improve the performance, asynchronous checkpointing has been used. Thus, each spare process stores the solution of only few solver processes of its subgroup at each time step. This approach further reduces the load on the network. As an example, suppose the backup time step is set to 10. The solver processes $j = [0 - 1 - 2 - 3]$ with $floor(j/4) = 0$ will send to the checkpoint process $[0 - 1 - 2 - 3]$ respectively their solution at the 1^{st} time step and then at the 11^{th}, at the 21^{st} time step and so one. The solver processes $j = [8 - 9 - 10 - 11]$ with $floor(j/4) = 2$ will send it to the spare process $[0 - 1 - 2 - 3]$ respectively at the 3^{rd} time step and then at the 13^{th}, at the 23^{rd} time step etc... Before starting the numerical reconstruction procedure

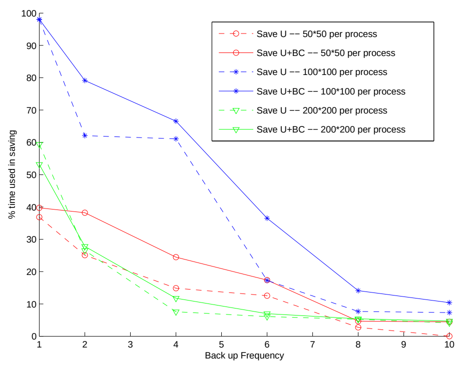

Fig. 5. Asynchronous checkpointing overhead with 9 processes.

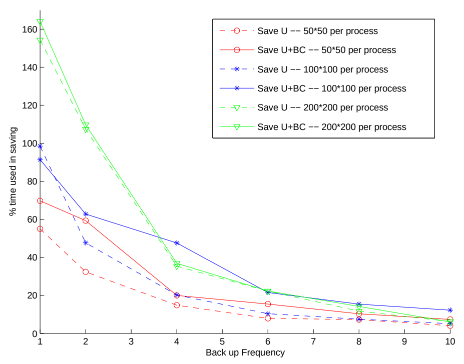

Fig. 6. Asynchronous checkpointing overhead with 16 processes.

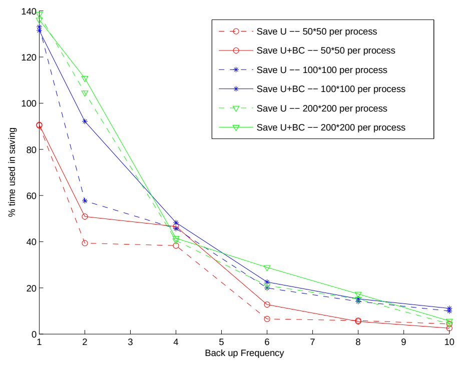

Fig. 7. Asynchronous checkpointing overhead with 25 processes.

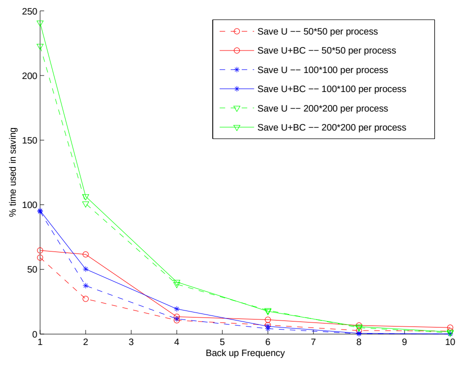

Fig. 8. Asynchronous checkpointing overhead with 36 processes.

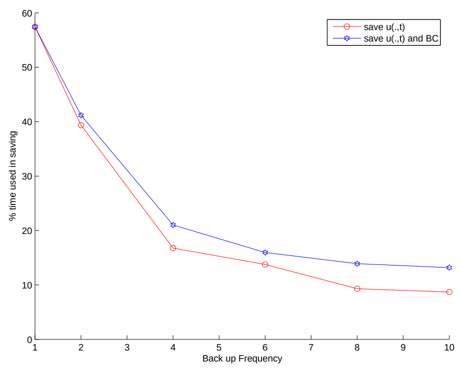

Fig. 9. Asynchronous checkpointing overhead for the small 3-D test case.

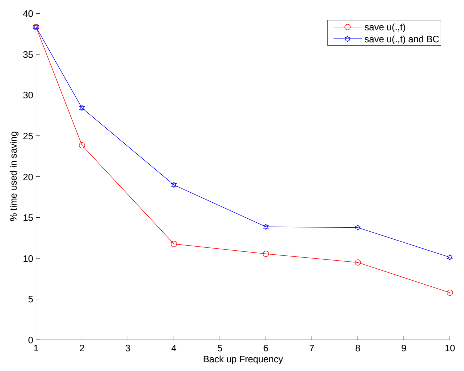

Fig. 10. Asynchronous checkpointing overhead for the large 3-D test case.

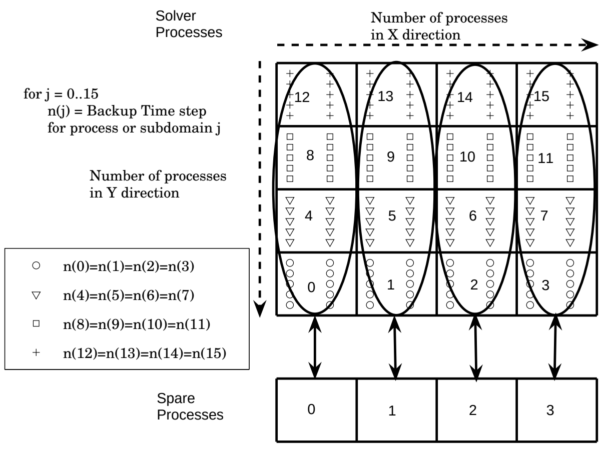
Solver Processes
Number of processes in X direction
for j = 0..15
n(j) = Backup Time step for process or subdomain j
Number of processes in Y direction
○ n(0)=n(1)=n(2)=n(3)
▽ n(4)=n(5)=n(6)=n(7)
□ n(8)=n(9)=n(10)=n(11)
+ n(12)=n(13)=n(14)=n(15)
12
13
14
15
8
9
10
11
4
5
6
7
0
1
2
3
Spare Processes
0
1
2
3

Fig. 11. Scheme of the local checkpointing

discussed in section 2.3, the spare process(es) will send the last local solution(s) of the failed process(es) to their replacement process(es). Additionally, the last checkpoint of all neighbors of the failed process(es) will be distributed to them, since this data is required in the algorithm presented previously as well. From that on, the three step *local* reconstruction procedure can start and only the crashed process(es) and its neighbors will be involved.

4 Results

In the following, we would like to present the costs of a recovery operations in case a process failure occurs. Using the 2-D testcase described in section 2.4 we simulated a process failure and measured the execution time required for respawning the failed process and to reconstruct the data of this process using the backward explicit scheme. Two testcases using 9 and 16 solver processes have been analyzed. The recovery time for both cases was in the order of 2% of the overall execution time of the same simulation for the same number of time steps. For the 9 processor case, the average recovery time of the application was 0.16 seconds for the small problem size and 0.2 seconds for the largest one. The recovery time for the 16 processes test cases was in the same range, the recovery operation after a process failure took up to 0.34 seconds. These results show, that while the recovery time is increasing with the number processes used, its overall effect is still negligible.

5 Summary

This paper discusses two approaches on how to handle process failures for parabolic problems. Based on distributed and uncoordinated checkpointing, the numerical methods presented here can reconstruct a consistent state in the parallel application, despite of storing checkpoints of various processes at different time steps. The first method, the forward implicit scheme, requires for the reconstruction procedure the boundary variables of each time step to be stored along with the current solution. The second method, the backward explicit scheme, only requires checkpointing the solution of each process every K time steps. Performance results comparing both methods with respect to the checkpointing overhead have been presented. We presented the results for recovery time of a 2-D heat equation. Currently ongoing work is focusing on the implementation of these explicit methods for a 3D Reaction-Convection-Diffusion code simulating the Air Quality Model [11].

References

1. MPI Forum : *MPI: A Message-Passing Interface Standard.* Document for a Standard Message-Passing Interface, University of Tennessee, 1993, 1994.
2. Sriram Sankaran, Jeffrey M. Squyres, Brian Barrett, Andrew Lumsdaine, Jason Duell, Paul Hargrove, and Eric Roman, The LAM/MPI Checkpoint/Restart Framework: System-Initiated Checkpointing, in *International Journal of High Performance Computing Applications*, 2004.
3. G. Bosilca, A. Bouteiller, F. Cappello, S. Djilali, G. Fedak, C. Germain, T. Herault, P. Lemarinier, O. Lodygensky, F. Magniette, V. Neri, and A. Selikhov, MPICH-V: Toward a Scalable Fault Tolerant MPI for Volatile Nodes, in SC'2002 Conference CD, IEEE/ACM SIGARCH, Baltimore, MD, 2002.
4. Graham E. Fagg, Edgar Gabriel, Zizhong Chen, Thara Angskun, George Bosilca, Jelena Pjesivac-Grbovic, and Jack J. Dongarra, 'Process Fault-Tolerance: Semantics, Design and Applications for High Performance Computing', in 'International Journal of High Performance Computing Applications', Volume 19, No. 4, pp. 465-477, Sage Publications 2005.
5. Beck, Dongarra, Fagg, Geist, Gray, Kohl, Migliardi, K. Moore, T. Moore, Papadopoulous, Scott, and Sunderam, 'HARNESS: a next generation distributed virtual machine', Future Generation Computer Systems, 15, 1999.
6. C. Engelmann and G. A. Geist. "Super-Scalable Algorithms for Computing on 100,000 Processors". Proceedings of International Conference on Computational Science (ICCS) 2005, Atlanta, GA, USA, May 2005.
7. Zizhong Chen, Graham E. Fagg, Edgar Gabriel, Julien Langou, Thara Angskun, George Bosilca, Jack J. Dongarra, 'Fault Tolerant High Performance Computing by a coding approach', Proceedings of the 2005 ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '05), Chicago, IL, June 15-17, 2005, ACM Press.
8. D.A. Murio, The Mollification Method and the Numerical Solution of Ill-posed Problems, Wiley, New York (1993).
9. M. Garbey, H. Ltaief, *Fault Tolerant Domain Decomposition for Parabolic Problems*, Domain Decomposition 16, New York University, January 2005. To appear.
10. W. Eckhaus and M. Garbey, Asymptotic analysis on large time scales for singular perturbation problems of hyperbolic type, SIAM J. Math. Anal. Vol 21, No 4, pp867-883 (1990).
11. Dupros, M.Garbey and W.E.Fitzgibbon, A Filtering technique for System of Reaction Diffusion equations, Int. J. for Numerical Methods in Fluids, in press 2006.