A Fault-Tolerant Approach to Distributed Applications

T. Nguyen[1], J-A. Desideri[2], and L. Trifan[1]
[1]INRIA, Grenoble Rhône-Alpes, Montbonnot, Saint-Ismier, France
[2]INRIA, Sophia-Antipolis Méditerranée, Sophia-Antipolis, France

Abstract *- Distributed computing infrastructures support system and network fault-tolerance, e.g., grids and clouds. They transparently repair and prevent communication and system software errors. They also allow duplication and migration of jobs and data to prevent hardware failures. However, only limited work has been done so far on application resilience, i.e., the ability to resume normal execution after errors and abnormal executions in distributed environments. This paper addresses issues in application resilience, i.e., fault-tolerance to algorithmic errors and to resource allocation failures. It addresses solutions for error detection and management. It also overviews a platform used to deploy, execute, monitor, restart and resume distributed applications on grids and cloud infrastructures in case of unexpected behavior.*

Keywords: Resilience, fault-tolerance, distributed computing, e-Science applications, high-performance computing, workflows.

1 Introduction

This paper overviews some solutions for application errors detection and management when running on distributed infrastructures. A platform is presented relying on a workflow system interfaced with a grid infrastructure to model cloud environments. Section 2 gives some definitions of terms. Section 3 goes into details concerning a platform based on a workflow management system to support application resilience on distributed infrastructures, e.g., grids and clouds. Ttwo testcases illustrating resilience to hardware and sysem errors, and resilience to application errors respectively are described in Section 4, where an algorithm-based fault-tolerant approach is illusrated. Section 5 is a conclusion.

2 Definitions

This section provides some definitions of terms used in this paper in order to make clear some commonly used words, and ultimately avoid confusion related to the complex computer systems ecosystems [17].

The generic term *error* is used to characterize abnormal behavior, originating from hardware, operating systems and applications that do not follow prescribed protocols and algorithms. Errors can be fatal, transient and warnings, depending on their criticity level. Because sophisticated hardware and software stacks are operating on all production systems, there is a need to classify the corresponding concepts.

A *failure* is different from a process *fault*, e.g., computing a bad expression. Indeed, a system failure does not impact the correct logics of the application process at work, and should not be handled by it, but by the system error-handling software instead: “failures are non-terminal error conditions that do not affect the normal flow of the process” [11].
However, an activity can be programmed to throw a *fault* following a system failure, and the user can choose in such a case to implement a specific application behavior, e.g., a predefined number of activity retries or a termination.
Application and system software can raise *exceptions* when faults and failures occur. The exception handling software then handles the faults and failures. This is the case for the YAWL workflow management system [19][20], where so-called dedicated *exlets* can be defined by the users [21] . They are components dedicated to the management of abnormal application or system behavior. The extensive use of these exlets allows the users to modify the behavior of the applications on-line, without stopping the running processes. Further, the new behavior is stored as a new component of the application workflow, which incrementally modifies its specifications. It can therefore be modified dynamically to handle changes in the user and application requirements.
Fault-tolerance is a generic term that has long been used to name the ability of systems and applications to cope with errors. Transactional systems and real-time software for example need to be fault-tolerant [1]. Fault-tolerance is usually implemented using periodic *checkpoints* that store the current state of the applications and the corresponding data.
However, this checkpoint definition does not usually include the tasks execution states or contexts, e.g., internal loop counters, current array indices, etc. This means that interrupted tasks, whatever the causes of errors, cannot be restarted from their exact execution state immediately prior to the errors.
We assume therefore that the *recovery* procedure must restart the failed tasks from previously stored elements in the set of existing checkpoints. A consequence is that failed tasks cannot

be restarted on the fly, following for example a transient non-fatal error. They must be restarted exclusively from previously stored checkpoints.

Application *robustness* is the property of software that are able to survive consistently from data and code errors. This area is a major concern for complex numeric software that deal with data uncertainties. This is particularly the case for simulation applications [7].

Resilience is also a primary concern for the applications faced to system and hardware errors. In the following, we include both application (external) fault-tolerance and (internal) robustness in the generic term resilience [9]. This is fully compatible with the following definition of resilience: “Resilience is a measure of the ability of a computing system and its applications to continue working in the presence of system degradations and failures” [30].

In the following section, a platform for high-performance distributed computing is described (Section 3.2). Examples of application resilience are then given. They address system faults (Section 4.1), application failures (Section 4.2) and algorithm-based fault-tolerance (ABFT) (Section 4.3).

3 Application Resilience

3.1 Overview

Several proposals have emerged recently dedicated to resilience and fault management in HPC systems [14][15][16]. The main components of such sub-systems are dedicated to the management of error, ranging from early error detections to error assessment, impact characterization, healing procedures concerning infected codes and data, choice of appropriate steps backwards and effective low overhead restart procedures.

General approaches which encompass all these aspects are proposed for Linux systems, e.g., CIFTS [5]. More dedicated proposals focus on multi-level checkpointing and restart procedures to cope with memory hierarchy (RAM, SSD, HDD), hybrid CPU-GPU hardware, multi-core hardware topology and data encoding to optimize the overhead of checkpointing strategies, e.g., FTI [22]. Also, new approaches take benefit of virtualization technologies to optimize checkpointing mechanisms using virtual disks images on cloud computing infrastructures [23], and checkpoint on failure approaches [32]. The goal is to design and implement low overhead, high frequency and compact checkpointing schemes.

Two complementary aspects are considered here:

• The detection and management of failures inherent to the hardware and software systems used

• The detection and management of faults emanating from the application code itself

Both aspects are different and imply different system components to react. However, unforeseen or incorrectly handled application errors may have undesirable effects on the execution of system components. The system and hardware fault management components might then have drastic procedure to confine the errors, which can lead to the application aborting. This is the case for out of bound parameter and data values, incorrect service invocations, if not correctly taken care of in the application codes.

This raises an important issue in algorithms design. Parallelization of numeric codes on HPC platforms is today taken into account in an expanding move towards petascale and future exascale computers. But so far, only limited algorithmic approaches take into account fault-tolerance from the start.

Generic system components have been designed and tested for fault-tolerance. They include fault-tolerance backpanes [5] and fault-tolerance interfaces [22]. Both target general procedures to cope with systematic monitoring of hardware, system and applications behaviors. Performance consideration limit the design options of such systems where incremental and multi-level checkpoints become the norm, in order to alleviate the overhead incurred by checkpoints storage and CPU usage. These can indeed exceed 25% of the total wall time requirements for scientific applications [22]. Other proposals take advantage of virtual machines technologies to optimize checkpoints storage using incremental (“shadowed” and “cloned”) virtual disks images on virtual machines snapshots [23] or checkpoint on failures protocols [32].

3.2 Distributed Platform

The distributed platform is built by the connection of two components:

• workflow management system for application definition, deployment, execution and monitoring [1][2];

• middleware allowing for distributed resource reservation, and execution of the applications on a wide-area network.

This forms the basis for the cloud infrastructure.

3.2.1 Distributed workflow

The applications are defined using a workflow management system, i.e., YAWL [20]. This allows for dataflow and control flow specifications. It allows parameter definition and passing between application tasks. The tasks are defined incrementally and hierarchically. They can bear constraints that trigger appropriate code to cope with exceptions, i.e., exlets, and user-defined real-time runtime branchings. This allows for situational awareness at runtime and supports user interventions, when required. This is a powerful tool to deal with fault-tolerance and application resilience at runtime [9].

3.2.2 Middleware

The distribution of the platform is designed using an open-source middleware, i.e., Grid5000 [1]. This allows for reservation, deployment and execution of customized systems and application configurations. The Grid5000 nationwide infrastructure currently includes 12 sites in France and abroad, 19 research labs, 15 clusters, 1500 nodes, 8600 cores, connected by a 10Gb/s network. The resource reservations, deployment and execution of the applications are made through standardized calls to specific system libraries. Because the infrastructure is shared between many research labs, resource reservation and job executions, i.e., applications, are queued with specific priority considerations.

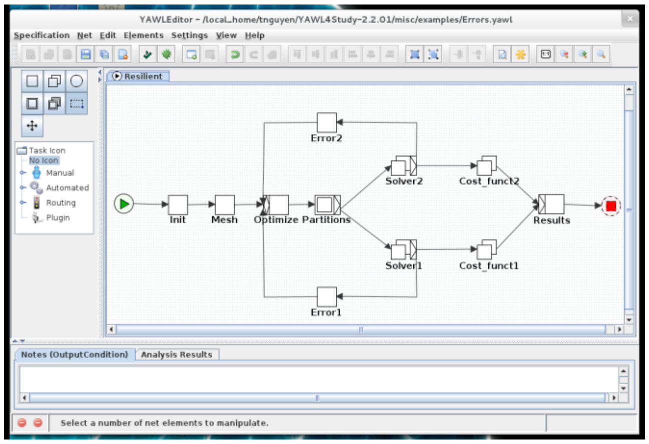

Figure 1. Distributed application workflow.

4 Experiments

Experiments are defined, run and monitored using the standard YAWL workflow system interface [6][19]. They invoke automatically or manually the tasks, as defined in the application specification interface. Tasks in turn invoke the various executable components tranparently through the middleware, using Web services [21]. They are standard in YAWL and used to invoke remote executable codes specific to each task. The codes are written in any programming language, ranging from Python to Java and C++. Remote script invocations with parameters are also possible. Parameter passing and data exchange, including files, between the executable codes are standardized in the workflow interface. Data structures are extendible user-defined templates to cope with all potential applications. As mentioned in the previous sections (Section “Workflow”, above), constraints are defined and rules trigger component tasks based on data values conditional checks at runtime. The testcases are distributed on a network of HPC clusters using the Grid5000 infrastructure. The hardware characteristics of the clusters are different. The application performance when running on various clusters are therefore different.

Two complementary testcases are described in the following sections. The first one focuses on resilience to hardware and systems failures, e.g., memory overflow (Section 4.1). The second one is focused on resilience to application faults, e.g., runtime error of a particular application component (Section 4.2). It is supported by a fault-tolerant algorithm (Section 4.3).

4.1 Resilience to hardware or system failures

We use the infrastructure to deploy the application tasks on the various clusters and take advantage of the different cluster performance characteristics to benefit from load-balancing techniques combined with error management. This approach therefore combines optimal resource allocation with the management of specific hardware and system errors, e.g., memory overflow, disk quota exceeded.

The automotive testcase presented in this section includes 17 different rear-mirror models tested for aerodynamics optimization. They are attached to a vehicle mesh of 22 million cells. A reference simulation was performed in 2 days on a 48 CPU non-distributed cluster with a total of 144 GB RAM. The result was a 2% drag reduction for the complete vehicle. The mesh will be eventually refined to include up to 35 million cells. A DES (Detached Eddy Simulation) flow simulation model is used.

The tasks include, from left to right in Figure 1:

• An initialization task for configuring the application (data files, optimization codes among which to choose…)

• A mesh generator producing the input data to the optimizer from a CAD file

• An optimizer producing the optimized data files (e.g., variable vectors)

• A partitioner that decomposes the input mesh into several sub-meshes for parallelization

• Each partition is input to a solver, several instances of which work on particular partitions

• A cost function evaluator, e.g., aerodynamic drag

• A result gathering task for output and data visualization

• Error handlers in order to process the errors raised by the solvers

The optimizer and solvers are implemented using MPI. This allows highly parallel software executions. Combined with the parallelization made possible by the various mesh partitions, and the different geometry configurations of the testcase, it follows that there are three complementary parallelization levels in this testcase, which allow to fully benefit from the HPC clusters infrastructure.

Should a system failure occur during the solver processes, an exception is raised by the tasks and they transfer the control to the corresponding error handler. This one will process the failures and trigger the appropriate actions, including:

• Migrate the solver task and data to another cluster, in case of CPU time limit or memory overflow: this is a load-balancing approach

• Retry the optimizer task with new input parameters requested from the user, if necessary (number of iterations, switch optimizer code…)

• Ignore the failure, if applicable, and resume the solver task

This approach merges two different and complementary techniques:

• Application-level error handling

• A load-balancing approach to take full benefit of the various cluster characteristics, for best resources utilization and application performance

Finally, the testbed implements the combination of a user-friendly workflow system with a grid computing infrastructure. It includes automatic load-balancing and resilience techniques. It therefore provides a powerful cloud infrastructure, compliant with the “Infrastructure as a Service” approach (IaaS).

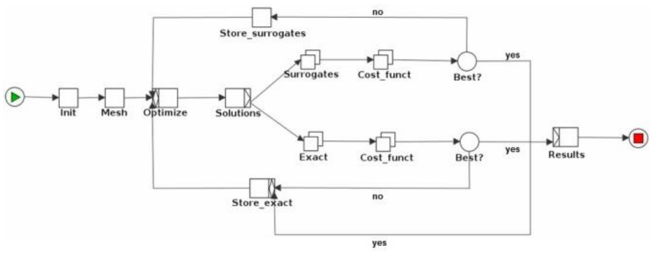

Figure 2. Distributed parallel application.

4.2 Resilience to application faults

An important issue in achieving resilient applications is to implement fault-tolerant algorithms. Programming error-aware codes is a key feature that supports runtime checks, including plausibility tests on variables values at runtime and quick tests to monitor the application behavior. Should unexpected values occur, the users can then pause the applications, analyzes the data and take appropriate actions at runtime, without aborting the applications and restarting them all over again.

It is also important that faults occurring in a particular part of the application code do not impair other running parts that behave correctly. This is fundamental to distributed and parallel applications, particularly for e-Science application running for days and weeks on petabytes of data. Thus, the correct parts can run to completion and wait for the erroneous part to restart and resume, so that the whole application can be run to satisfactory results. This is mentioned as *local recovery* in [30].

This section details an approach to design and implement a fault-tolerant optimization algorithm which is robust to runtime faults on data values, e.g., out of bounds values, infinite loops, fatal exceptions.

In contrast with the previous experiment (Section 3.3.1), where the application code was duplicated on each computing node and data migrated for effective resources utilization and robustness with respect to hardware and system failures, this new experiment is based on a fully distributed and parallel optimization application which is inherently resilient to application faults.

It is based on several parallel branches that run asynchronously and store their results in different files, providing the inherent resilience capability. This is complemented by a fault-tolerant algorithm described in the next section (Section 3.3.3).

The application is designed to optimize the geometry of an air-conditioner [24]. It uses both a genetic algorithm and a surrogate approach that run in parallel and collaborate to produce pipe geometries fitting best with two optimization objectives: minimization of the pressure loss at the output of the pipe and minimization of the flow distribution at the output (Figures 2). The complete formal definition and a detailed description of the application are given in [24].

Solutions to the optimization goals are formed by several related elements corresponding to the different optimization criteria. In case of multi-objective optimization, as is the case here with the minimization of pressure loss throughout the air-conditioner pipe and minimization of speed variations at the output of the pipe, there are two objectives. There are multiple optimal solutions than can be vizualised as Pareto fronts.

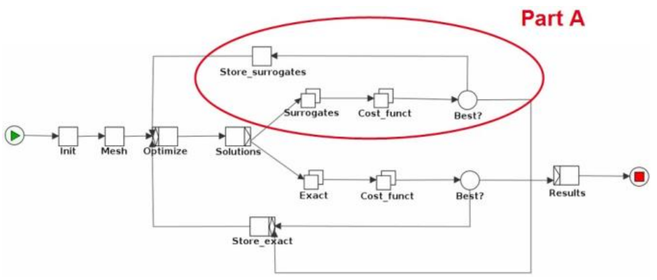

Figure 3. Fault-tolerant algorithm: Part A.

Approximate solutions using the surrogates (Figure 3: Part A) and exact solutions using the genetic algorithm (Figure 4: Part B) run asynchronously in parallel to evaluate temporary partial solutions.

When a surrogate is deemed correct, i.e., its accuracy is below-user defined thresholds with respect to the fitness criteria, it is stored in the exact file and tagged “provisional”. Future evaluations by the exact genetic algorithm will use it together with the other exact values to improve the future exact solutions (Figure 4, Part B). The genetic algorithm will eventually supersede the provisional solutions. In contrast, surrogates values are computed as long as “better” exact values are not produced.

Each part stores its results in a specific file (Figure 7: Part E and Part F). When the exact solutions satisfy predefined accuracy criteria with respect to the optimization objectives, they are stored in the final results file (Figure 7, Part G).

Each part in the application workflow implements an asynchronous parallel loop that is driven by the optimizer. Each loop runs independently of the other. Each loop is itself parallelized by multiple instances of the Surrogates (Figure 5: Part C) and Exact (Figure 6: Part D) evaluation codes that compute potential solution in parallel. Each solution is a candidate geometry for the air-conditioner pipe optimizing the fitness criteria, e.g., pressure loss and flow speed variations at the output of the pipe.

The final results file stores the combined values for the optimal solutions (Figure 7: Part G). There are multiple optimal solutions, hence the need for a specific file to store them.

4.3 Fault-tolerant algorithm

The solution to resilience for the application described in the previous section (Section 4.2) is the fault-tolerant algorithm implemented to compute the solutions to the multi-objective optimization problem. This illustrates the algorithm-based fault-tolerance approach (ABFT) used here.

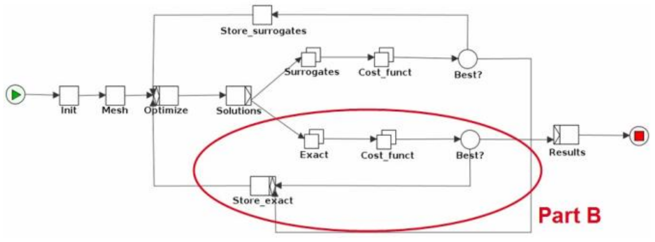

Figure 4. Fault-tolerant algorithm: Part B.

As mentioned above, the implementation of the application is distributed, parallel and asynchronous. It is distributed because the tasks are deployed on the various sites where the application codes run. It is parallel because these tasks can run multiple instances simultaneously for the computations of the surrogates and the exact solutions. It is asynchronous because the surrogates and exact solutions are computed in two distinct parallel loops and produce their results whatever the state of the other loop. This paves the way to implement an inherently fault-tolerant algorithm which is described in more details in this section.

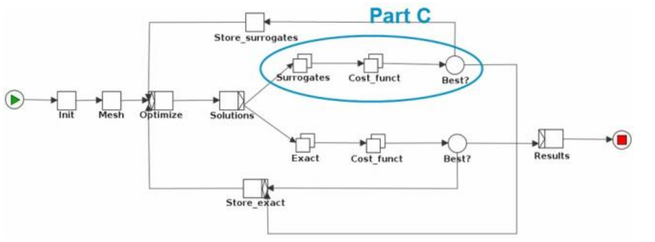

Figure 5. Fault-tolerant algorithm: surrogate branch.

There are four complementary levels of parallelism running concurrently: the surrogate (Part A) and exact (Part B) parts and inside each part, the multiple instances of approximate (Part C) and exact solutions (Part D) that are computed in parallel.

Faults in either part A and B do not stop the other part. Further, faults in particular instances of the approximate and exact solutions computations do not stop the other instances computing the other solutions in parallel.

Indeed, surrogates and exact solutions are computed in parallel using multiple instances of the task “Surrogates” and “Exact” in part C and Part D of the workflow (Figures 5 and 6). Also, the faults in a particular instance inside parts C and D do not stop the computation of the other solutions running in parallel inside those parts.

Further, the three independent files used to store the surrogates, the exact solutions and the final solutions respectively allow for the restart of whatever part has failed without impacting any other file (Figure 7). The content of the three files are indeed the checkpoints where the failed parts can restart from. This allows for effective checkpointing and restart mechanisms. Errors need not the whole applications to be restarted from scratch. They can be resumed using the most recent surrogates and exact solutions already computed.

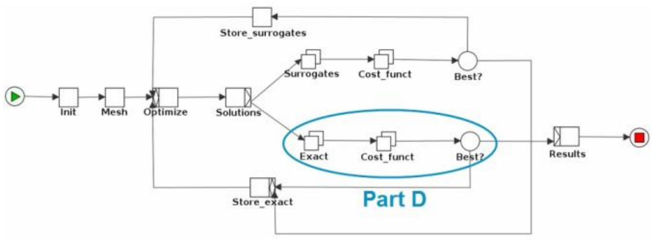

Figure 6. Fault-tolerant algorithm: exact branch.

The vulnerability of the files to faults and failures is also a critical issue for application resilience. Most file management systems provide transaction and back-up capabilities to support this. Faults and failures impacting the Part E and Part F files will have little impact since the lost data they contained is automatically recomputed by the Part C and Part D loops respectively, which take into account the current provisional and exact solutions stored. Lost data in either file after a restart will therefore be recomputed seamlessly. The overhead is therefore only the recalculation of the lost data, without the need for a specific recovery procedure.

The most critical part is the Part G file which stores the final optimal solutions. It should be duplicated on the fly to another location for best availability after errors. But the final results already stored in the Part G file are not impacted by faults and failures in either Part A, B, E and F. They need not be computed again.

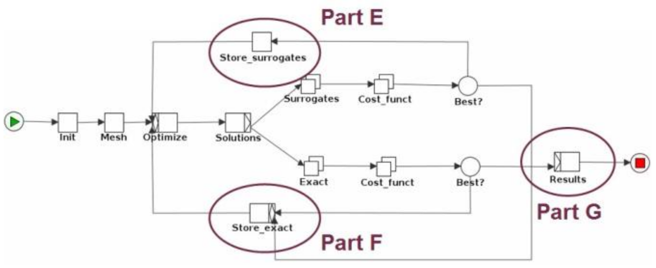

Figure 7. Fault-tolerant algorithm: result files .

5 Conclusion

High-performance computing and cloud infrastructures are today commonly used for running large-scale e-Science applications.

This has raised concerns about system fault-tolerance and application resilience. Because exascale computers are emerging and cloud computing is commonly used today, the need for supporting resilience becomes even more stringent.

New sophisticated and low-overhead functionalities are therefore required in the hardware, systems and application layers to support effectively error detection and recovery.

This paper defines concepts, details current issues and sketches solutions to support application resilience. Our approach is currently implemented and tested on simulation testcases using a distributed platform that operates a workflow management system interfaced with a grid infrastructure, providng a seamless cloud computing environment.

The platform supports functionalities for application specification, deployment, execution and monitoring. It features resilience capabilities to handle runtime errors. It implements the cloud computing "Infrastructure as a Service" paradigm using a user-friendly application workflow interface. Two example testcases implementing resilience to hardware and system failures, and also resilience to application faults using algorithm-based fault-tolerance (ABFT) are described.
Future work is still needed concerning the recovery of unforeseen errors occuring simultaneously in the applications, system and hardware layers of the platform, which raise open and challenging problems [31].

6 Acknowledgments

The authors wish to thank Piotr Breitkopf, Head of Laboratoire Roberval at the University of Compiègne (France), Alain Viari and Frederic Desprez, Directeurs de Recherche with INRIA, and Balaji Raghavan (PhD) for many fruitful discussions concerning the testcases design and deployment. This work is supported by the European Commission FP7 Cooperation Program "Transport (incl. aeronautics)", for the GRAIN Coordination and Support Action ("Greener Aeronautics International Networking"), grant ACS0-GA-2010-266184. It is also supported by the French National Research Agency ANR (Agence Nationale de la Recherche) for the OMD2 project (Optimisation Multi-Disciplines Distribuée), grant ANR-08-COSI-007, program COSINUS (Conception et Simulation).

7 References

[1] T. Nguyên, J.A Désidéri. "Resilience Issues for Application Workflows on Clouds". Proc. 8th Intl. Conf on Networking and Services (ICNS2012). pp. 375-382. Sint-Maarten (NL). March 2012.

[2] E. Deelman et Y. Gil., "Managing Large-Scale Scientific Workflows in Distributed Environments: Experiences and Challenges", Proc. of the 2nd IEEE Intl. Conf. on e-Science and the Grid. pp. 165-172. Amsterdam (NL). December 2006.

[3] H. Simon. "Future directions in High-Performance Computing 2009- 2018". Lecture given at the ParCFD 2009 Conference. Moffett Field (Ca). May 2009.

[4] Dongarra, P. Beckman et al. "The International Exascale Software Roadmap". Volume 25, Number 1, 2011, International Journal of High Performance Computer Applications, pp. 77-83. Available at: http://www.exascale.org/

[5] R. Gupta, P. Beckman et al. "CIFTS: a Coordinated Infrastructure for Fault-Tolerant Systems", Proc. 38th Intl. Conf. Parallel Processing Systems. pp. 145-156. Vienna (Au). September 2009.

[6] D. Abramson, B. Bethwaite et al. "Embedding Optimization in Computational Science Workflows", Journal of Computational Science 1 (2010). Pp 41-47. Elsevier.

[7] A.Bachmann, M. Kunde, D. Seider and A. Schreiber, "Advances in Generalization and Decoupling of Software Parts in a Scientific Simulation Workflow System", Proc. 4th Intl. Conf. Advanced Engineering Computing and Applications in Sciences (ADVCOMP2010). Pp 247-258. Florence (I). October 2010.

[8] E.C. Joseph, et al. "A Strategic Agenda for European Leadership in Supercomputing: HPC 2020", IDC Final Report of the HPC Study for the DG Information Society of the EC. July 2010. http://www.hpcuserforum.com/EU/

[9] T. Nguyên, J.A. Désidéri. "A Distributed Workflow Platform for High-Performance Simulation", Intl. Journal on Advances in Intelligent Systems. 4(3&4). pp 82-101. IARIA. 2012.

[10] E. Sindrilaru, A. Costan and V. Cristea. "Fault-Tolerance and Recovery in Grid Workflow Mangement Systems", Proc. 4th Intl. Conf. on Complex, Intelligent and Software Intensive Systems. pp. 162-173. Krakow (PL). February 2010.

[11] Apache. The Apache Foundation. http://ode.apache.org/bpel-extensions.html#

BPELExtensionsActivityFailureandRecovery

[12] P. Beckman. "Facts and Speculations on Exascale: Revolution or Evolution?", Keynote Lecture. Proc. 17th European Conf. Parallel and Distributed Computing (Euro-Par 2011). pp. 135-142. Bordeaux (F). August 2011.

[13] P. Kovatch, M. Ezell, R. Braby. "The Malthusian Catastrophe is Upon Us! Are the Largest HPC Machines Ever Up?", Proc. Resilience Workshop at 17th European Conf. Parallel and Distributed Computing (Euro-Par 2011). pp. 255-262. Bordeaux (F). August 2011.

[14] R. Riesen, K. Ferreira, M. Ruiz Varela, M. Taufer, A. Rodrigues. "Simulating Application Resilience at Exascale", Proc. Resilience Workshop at 17th European Conf. Parallel and Distributed Computing (Euro-Par 2011). pp. 417-425. Bordeaux (F). August 2011.

[15] P. Bridges, et al. "Cooperative Application/OS DRAM Fault Recovery", Proc. Resilience Workshop at 17th European Conf. Parallel and Distributed Computing (Euro-Par 2011). pp. 213-222. Bordeaux (F). August 2011.

[16] IJLP. Proc. 5th Workshop INRIA-Illinois Joint Laboratory on Petascale Computing. Grenoble (F). June 2011. http://jointlab.ncsa.illinois.edu/events/workshop5/

[17] F. Capello, et al. "Toward Exascale Resilience", Technical Report TR-JLPC-09-01. INRIA-Illinois Joint Laboratory on PetaScale Computing. Chicago (Il.). 2009. http://jointlab.ncsa.illinois.edu/

[18] Moody A., G.Bronevetsky, K. Mohror, B. de Supinski. Design, "Modeling and evaluation of a Scalable Multi-level checkpointing System", Proc. ACM/IEEE Intl. Conf. for High Performance Computing, Networking, Storage and Analysis (SC10). pp. 73-86. New Orleans (La.). Nov. 2010. http://library-ext.llnl.gov Also Tech. Report LLNL-TR-440491. July 2010.

[19] Adams M., ter Hofstede A., La Rosa M. "Open source software for workflow management: the case of YAWL", IEEE Software. 28(3): 16-19. pp. 211-219. May/June 2011.

[20] Russell N., ter Hofstede A. "Surmounting BPM challenges: the YAWL story.", Special Issue Paper on Research and Development on Flexible Process Aware Information Systems. Computer Science. 23(2): 67-79. pp. 123-132. March 2009. Springer 2009.

[21] Lachlan A., van der Aalst W., Dumas M., ter Hofstede A. "Dimensions of coupling in middleware", Concurrency and Computation: Practice and Experience. 21(18):233-2269. pp. 75-82. J. Wiley & Sons 2009.

[22] Bautista-Gomez L., et al., "FTI: high-performance Fault Tolerance Interface for hybrid systems", Proc. ACM/IEEE Intl. Conf. for High Performance Computing, Networking, Storage and Analysis (SC11), pp. 239-248, Seattle (Wa.)., November 2011.

[23] Nicolae B. and Capello F., "BlobCR: Efficient Checkpoint-Restart for HPC Applications on IaaS Clouds using Virtual Disk Image Snapshots", Proc. ACM/IEEE Intl. Conf. High Performance Computing, Networking, Storage and Analysis (SC11), pp. 145-156, Seattle (Wa.)., November 2011.

[24] Raghavan B., Breitkopf P. "Asynchronous evolutionary shape optimization based on high-quality surrogates: application to an air-conditioning duct". In: Engineering with Computers. Springer. April 2012. http://www.springerlink.com/content/bk83876427381p3w/fulltext.pdf

[25] Jeffrey K., Neidecker-Lutz B. "The Future of Cloud Computing". Expert Group Report. European Commission. Information Society & Media Directorate-General. Software & Service Architectures, Infrastructures and Engineering Unit. Jannuary 2010.

[26] Latchoumy P., Sheik Abdul Khader P. "Survey on Fault-Tolerance in Grid Computing". Intl. Journal of Computer Science & Engineering Survey. Vol. 2. No. 4. November 2011.

[27] Garg R., Singh A. K.. "Fault Tolerance in Grid Computing: State of the Art and Open Issues". Intl. Journal of Computer Science & Engineering Survey. Vol. 2. No. 1. February 2011.

[28] Heien E., et al. "Modeling and Tolerating Heterogeneous Failures in Large Parallel Systems". Proc. ACM/IEEE Intl. Conf. High Performance Computing, Networking, Storage and Analysis (SC11), pp. 45:1-45:11, Seattle (Wa.)., November 2011.

[29] Bougeret M., et al. "Checkpointing Strategies for Parallel Jobs". Proc. ACM/IEEE Intl. Conf. High Performance Computing, Networking, Storage and Analysis (SC11), pp. 33:1-11, Seattle (Wa.), November 2011.

[30] LLNL. "FastForward R&D Draft Statement of Work". Lawrence Liverrmore National Lab. US Department of Energy. Office of Science and National Nuclear Security Administration. LLNL-PROP-540791-DRAFT. March 2012.

[31] SC. LLNL-SC "The opportunities and Challenges of Exascale Computing". Summary Report of the Advanced Scientific Computing Advisory Subcommittee. US Department of Energy. Office of Science. Fall 2010.

[32] Xiao Liu, et al. "The Design of Cloud Workflow Systems". Springer Briefs in Computer Science. Springer 2012.