Enhancing Fault-Tolerance of Large-Scale MPI Scientific Applications

G. Rodríguez, P. González, M.J. Martín, and J. Touriño

Computer Architecture Group, Dep. Electronics and Systems
University of A Coruña, Spain
{grodriguez,pglez,mariam,juan}@udc.es

Abstract. The running times of large-scale computational science and engineering parallel applications, executed on clusters or Grid platforms, are usually longer than the mean-time-between-failures (MTBF). Therefore, hardware failures must be tolerated to ensure that not all computation done is lost on machine failures. Checkpointing and rollback recovery are very useful techniques to implement fault-tolerant applications. Although extensive research has been carried out in this field, there are few available tools to help parallel programmers to enhance their applications with fault tolerance support. This work presents an experience to endow with fault tolerance two large MPI scientific applications: an air quality simulation model and a crack growth analysis. A fault tolerant solution has been implemented by means of a checkpointing and recovery tool, the CPPC framework. Detailed experimental results are presented to show the practical usefulness and low overhead of this checkpointing approach.

Keywords: Fault tolerance, checkpointing, parallel applications, MPI.

1 Introduction

Checkpointing has become a widely used technique to provide fault tolerance by periodically saving the computation state to stable storage, so that this state can be restored in case of execution failure.

One of the most remarkable properties of general checkpointing techniques is *granularity*. Checkpointing can be performed from two different granularity levels: *data segment level* and *variable level*. On data segment level the entire application state is saved (data segment, stack segment and execution context), recovering it when necessary. Most of fault-tolerance tools present in the bibliography [1,2,3,4,5] perform data segment level checkpointing. This approach presents a general advantage: its transparency from the user's point of view, since the application is seen as a black box. However, saving the application state entirely leads to lack of portability, as a number of non-portable structures will be saved along with application data (as application stack or heap).

A variable level approach saves only restart-relevant state to stable storage. Many fault tolerant solutions implement variable level checkpointing by manually determining the data to be saved, and inserting code to save that data on disk and to restart the computation after failure. The code becomes as portable as the original application and, provided that checkpoints are saved in a portable format, the application can be restarted on different platforms. Unfortunately, this method requires a data-flow analysis, which can be a tedious and error-prone task to be performed by the user. Thus, a recent approach [6], developed by the authors, tries to automatize a variable level checkpointing of message-passing parallel applications by means of a checkpointing library and a compiler that instruments MPI code.

The purpose of this work is to develop fault tolerant solutions for two different computationally intensive MPI codes, an air quality model [7] and a crack growth analysis [8]. A variable level checkpointing approach is followed, implemented through the use of our checkpointing and recovery tool, *CPPC*.

The structure of this paper is as follows. Section 2 introduces the problem of endowing parallel applications with fault tolerance, and gives an overview of the CPPC tool and how it solves the major issues. Section 3 describes the applications used for the tests. Experimental results about the use of the CPPC tool are presented in Section 4. Finally, Section 5 concludes the paper.

2 Checkpointing and Recovery of Parallel Applications: The CPPC Tool

There are several issues to be solved in implementing checkpointing solutions for parallel applications, such as consistency, portability, memory requirements, or transparency. CPPC is a checkpointing infrastructure that implements scalable, efficient and portable checkpointing mechanisms. This section details various aspects of CPPC's design associated with major issues.

2.1 Global Consistency

Consistency is a key issue when dealing with the checkpoint of a parallel program using the message-passing paradigm. The state of a parallel application is defined as the set of all its processes states. There are two situations that require actions to be performed in order to achieve a correct restart: existence of *in-transit* messages (sent but not received), and existence of *ghost* messages (received but not sent) in the set of processes states stored.

Checkpoint consistency has been well-studied in the last decade [9]. Approaches to the consistent recovery can be categorized into different protocols: uncoordinated, coordinated and communication-induced checkpointing; and message logging.

In uncoordinated checkpoint protocols the checkpoint of each process is executed independently of the other processes, leading to the so called *domino* effect (process may be forced to rollback up to the beginning of the execution). Thus, these protocols are not used in practice. In coordinated checkpoint protocols, all processes coordinate their checkpoints so that the global system state composed of the set of all process checkpoints is coherent. Communication-induced checkpoint tries to take advantage of uncoordinated and coordinated checkpoint techniques. Based on the uncoordinated approach, it detects risk of inconsistent state, and forces processes to checkpoint. While this approach seems to be very interesting theoretically, in practice it turns out to be quite inefficient.

Message logging saves messages with checkpoint files in order to replay them for the recovery. The main disadvantage of log-based recovery is its high storage overhead.

CPPC achieves global consistency by using spatial coordination, rather than temporal coordination. Checkpoints are thus taken at the same relative code points by all the processes (assuming SPMD codes). To avoid problems caused by messages between processes, checkpoint directives must be inserted at points where it is guaranteed that there are no in-transit, nor ghost messages. These points are called *safe points*. For an automatic identification of safe points, a static analysis of interprocess message flow is needed. This automatization is currently under development.

2.2 Portability

The availability of the application to be executed across multiple platforms plays an important role in current trends towards new computing infrastructures, such as heterogeneous clusters and Grid systems.

A state file is said to be portable if it can be used to restart the computation on an architecture (or OS) different from that where the file was generated on. This means that state files should not contain hard machine-dependent state, which should be recovered at restart time using special protocols.

The solution used in CPPC is to recover non-portable state by means of the re-execution of the code responsible for creating such opaque state in the original execution. Hence, the new code will be just as portable as the original code was. Moreover, in CPPC the effective data writing will be performed by a selected writing plugin implementation, using its own format. This enables the restart on different architectures, as long as a portable dumping format is used for program variables. Currently, a writing plugin based on HDF5 is provided. HDF5 [10] is a general purpose library and file format for storing scientific data in a portable way. The CPPC HDF5 plugin allows the generated checkpoint files to be used across multiple platforms. CPPC-generated HDF5 files are much like binary files, except that all data are tagged to make conversions possible when restarting on different platforms.

2.3 Memory Requirements

The solution of large large scientific problems may need the use of massive computational resources, both in terms of CPU effort and memory requirements. Thus, many scientific applications are developed to be run on a large number of processors. The checkpointing of this kind of applications would lead to a great amount of stored state, the cost being so high as to become impractical.

CPPC reduces the amount of data to be saved by including in its compiler a live variable analysis in order to identify those variable values that are only needed upon restart. Besides, the HDF5 library can accommodate data in a variety of ways, including a compressed format based on the ZLib library [11]. This, or other compression algorithms, can be included in a writing plugin without recompiling the CPPC library. A multithreaded dumping option [12] is also provided by the CPPC tool to improve performance when working with large datasets. A new thread handles checkpoint file creation while the application continues normal execution.

2.4 Transparency

This property is measured in terms of user effort to insert checkpoint support into the application. On the one hand, data segment level approaches are completely transparent to programmers, as they do not need much information about the applications being treated. On the other hand, variable level strategies have to get some metadata about the application in order to operate correctly, and they usually get it from the programmer.

The CPPC tool appears to the user as a compiler tool and a runtime library which help achieve the goal of inserting fault tolerance into a parallel application in an almost transparent way. The library provides checkpoint-support routines, and the compiler tool seeks to automatize the use of the library. The user must insert only one compiler directive into the original application (the `cppc checkpoint` pragma) to mark points in the code where the relevant state will be dumped to stable storage in a checkpoint file. The compiler performs a source-to-source transformation, automatically identifying both the variables to be dumped to the checkpoint file and the non-portable code to be re-executed upon restart; and it also inserts the necessary calls to functions of the CPPC library, as well as flow control code needed to recover the non-portable state.

3 The Applications

In this section, two large-scale scientific applications are described: an air quality model and a crack growth simulation. Both applications were found to be good candidates for using the CPPC tool. Originally, none of them provided fault-tolerance. However, being long running critical applications, both would benefit from this feature.

The STEM-II Model. Due to the increasing sources of air pollutants, the development of tools to control and prevent the pollutants' accumulation has become a high priority. Coal-fired electrical power plants constitute one of the most significant sources of air pollutants, thus its study is a key issue in pollution control specifications. The STEM-II model [13] is used to know in advance how the meteorological conditions, obtained from a meteorological prediction model, would affect the emissions of pollutants by the power plant of As Pontes (A Coruña, Spain) in order to fulfill EU regulations.

Air quality models can be mathematically described as time-dependent, 3D partial differential equations. The underlying equation used is the atmospheric-diffusion equation. The numerical solution of this equation consists of the integration of a system of coupled non-linear ordinary differential equations. STEM-II solves this system using a finite element method (FEM).

The sequential program consists mainly of four nested loops, a temporal loop (`loop_t`) and a loop for each dimension of the simulated space (`loop_x`, `loop_y` and `loop_z`). The main modules of the code are: horizontal transport, vertical transport and chemical reactions, and I/O module. The model requires as input data the initial pollutant concentrations, topological data, emissions from the power plant and meteorological data. The initial pollutant concentrations and topological data are read only once, at the beginning of the simulation. The meteorological data and the emissions from the power plant are time-dependent and must be read each 60 iterations, that is, each new hour of simulation. The output consists of spatially and temporally gaseous and aqueous concentrations of each modeled specie, reaction rates, in and out fluxes, amount deposited and ionic concentrations of hydrometeor particles. As this model is computationally intensive, it has been parallelized using MPI [7].

Crack Growth Analysis Using Dual BEM (DBEM). Cracks are present in all structures, usually as a result of localised damage in service, and may grow by processes such as fatigue, stress-corrosion or creep. The growth of the crack leads to a decrease in the structural strength. Thus, fracture occurs, leading to the failure of the structure.

The Boundary Element Method (BEM) has been acknowledged as an alternative to FEM in fracture mechanic analysis. BEM reduces the dimensionality of the problem under analysis through the discretization of the boundary domain only.

Despite the reduction of dimensionality using BEMs instead of FEMs, the crack growth analysis leads to a large number of discretized equations that grow at every step when the crack growth is evaluated. Analysis of real structural integrity problems may need the use of large computational resources, both in terms of CPU and memory requirements.

The boundary element code to assemble the linear equations is essentially a triple-nested DO loop. The external loop is over the collocation nodes, the middle loop is over the boundary elements, and the internal loop is over the Gauss points. Coarse grain parallelization can be achieved by distributing collocation nodes among processors [8].

Although assembling the linear equations is a key task in the simulation process, the bottleneck of the crack growth analysis is the solution of the resultant dense linear system. The traditional method for the solution of a dense linear system would be the application of the Gauss elimination method. However, as the problem size increases the use of iterative methods is demanded. This application uses the GMRES iterative method, regarded as the most robust of the Krylov subspace iterative methods.

Table 1. Applications' summary

				running on 4 nodes	
Tested application	Programming Language	Number of files	Lines of Code	Memory requirements	Disk quota
STEM	F77	149	9609	180MB	560MB
DBEM	F77	45	13164	370MB	170MB

4 Experimental Results

In this section, the results of applying the CPPC tool to the large-scale applications described in the previous section are presented. Results include checkpointing overhead, restart overhead, portability and checkpoint file size. Tests were performed on a cluster of Intel Xeon 1.8 Ghz nodes, 1GB RAM, connected through an SCI network.

Table 1 summarizes the two tested applications: the air quality simulation model (from now on referred to as STEM) and the crack growth simulation (DBEM).

CPPC treats the applications as black boxes, and automates the insertion of checkpoint-support routines provided by the CPPC library, identifying the variables to be dumped and the non-portable code to be re-executed upon restart, and inserting flow control code. The `cppc checkpoint` is the only directive not yet automated, and thus the programmer must find a safe point in the original code for the checkpointing file dumping. Safe points can be easily found in both codes, since they follow the SPMD paradigm. This point has been found at the end of the outer loop (`loop_t`) in the STEM code. In these experiments it executes 1440 iterations of the outer loop, which corresponds to 24 hours of real-time simulation. In the DBEM code, the checkpoint directive has been placed at the beginning of the main loop in the GMRES solver. In these experiments DBEM performs a crack growth simulation on a mesh of 496 collocation nodes, which involves the solution of a dense linear system of 1494 equations.

Figure 1 shows the execution times for both applications. Results are shown for the original execution, execution with CPPC checkpointing instrumentation, and two executions including different checkpoint frequencies. CPPC instrumentation includes calls to CPPC library routines, such as CPPC initialization or variable registration routines, and flow control code. As can be seen in the figure, the overhead introduced by the CPPC instrumentation remains under 5% for both applications.

The overhead of a single checkpoint file dumping depends on the amount of data to be stored and the format used for the data storage. Results shown in Figure 1 were obtained using HDF5 format. Early tests were carried out with one checkpoint file dumping each 60 iterations (labeled as "1/60" in the figure). Then, more tests were performed increasing the checkpoint frequency up to one checkpoint each ten iterations (labeled as "1/10"). Increasing the checkpoint frequency did not noticeably vary the total execution time, since

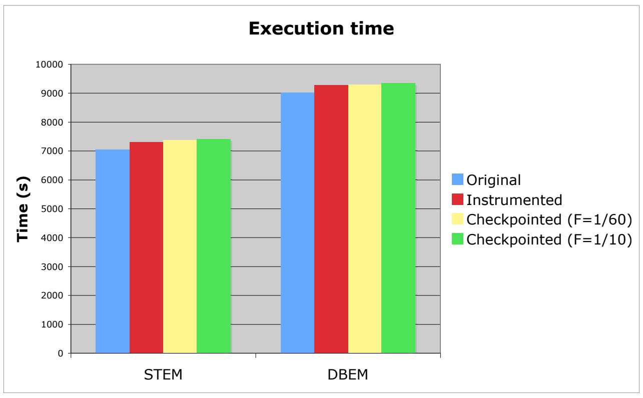

Fig. 1. Execution times in failure-free tests

once the instrumentation overhead is introduced, the multithreaded technique hides the overhead of the data dumping step.

These results have been obtained assuming no failures during the execution. In other case, the restart time should be also considered in the total execution time. Restart overhead is less important than checkpointing overhead. The application is expected to be restarted only in case of failure and, in long running applications, it will be always better than to re-execute the application from the beginning. Results for restart execution times can be seen in Figure 2. The total restart time is divided in two sections: overhead due to the checkpoint file read and overhead due to state recovery. Results labeled as “native” correspond to those obtained when restarting an application from checkpoint files generated in the same platform. In order to perform also a portability test, these applications were executed on an HP Superdome located at the Galician Supercomputing Center (Intel Itanium 2 nodes at 1.5Ghz, 3GB RAM, connected through Infiniband) with its proprietary Fortran compiler and MPI implementation. Checkpoint files created in this platform were used to restart the applications on the SCI cluster, thus allowing the comparison of restart times using both native and imported files (native and cross-platform results, respectively, in Figure 2). Reading time increases if data transformations are needed, since they will take place at application restart. Results have shown that the overhead introduced is low enough to be negligible, even in the cross-platform case.

As pointed out in Section 2, when dealing with large-scale applications, checkpointing could lead to a great amount of state stored. Hence, techniques to reduce the checkpoint file size are of capital importance. Table 2 compares CPPC generated file sizes to those obtained using a segment level approach. As can be seen, CPPC achieves very important size reductions by performing a live variable analysis (the number of live variables registered by CPPC are shown in the table). Table 2 also shows chekpoint file generation time (dumping time) when using the CPPC tool. Results of dumping time with and without the multithreading

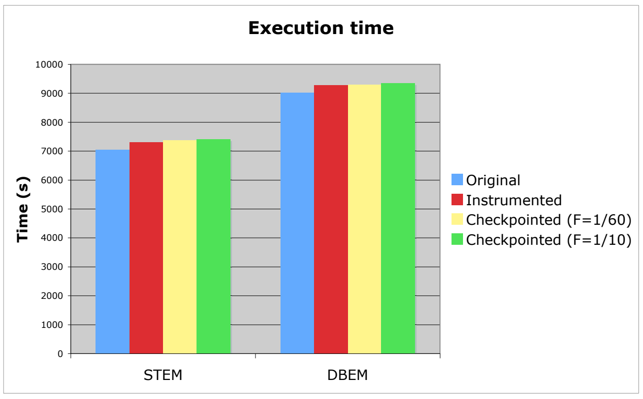

Fig. 2. Restart overhead

Table 2. Checkpoint file generation results

Tested application	Segment level ckpt-file size	CPPC ckpt-file size	CPPC registered variables	CPPC dumping time (s) absolute	CPPC dumping time (s) multithread
STEM	187 MB	121 MB	156	0.42	0.18
DBEM	290 MB	145 MB	178	0.91	0.52

option demonstrate that the checkpoint file generation has a minimal influence on the performance of long running applications.

5 Conclusions

Currently, there are several solutions available that deal with checkpointing of parallel applications. However, most of them implement data segment level approaches, which present serious drawbacks for real scientific applications, such as memory requirements or portability. Thus, development of new tools to provide variable level solutions with a high level of transparency from the user's point of view becomes a great challenge.

In this paper a variable level checkpointing tool, CPPC, has been tested with two large-scale scientific applications. CPPC resolves major issues in implementing scalable, efficient and portable checkpointing by using a variable level, non-coordinated, non-logging, portable approach. Experimental results have demonstrated the efficacy of this approach, in terms of execution times, checkpointing overhead, memory requirements, portability and usability.

CPPC version 0.5 can be downloaded at `http://cppc.des.udc.es`.

Acknowledgments. This work has been supported by the Ministry of Education and Science of Spain (ref: TIN-2004-07797-C02 and FPU grant AP-2004-2685), Galician Government (ref: PGIDIT04TIC105004PR) and CYTED Program (ref: 506PI0293). We gratefully thank CESGA (Galician Supercomputing Center) for providing access to the HP Superdome computer.

References

1. Bosilca, G., Bouteiller, A., Cappello, F., Djilali, S., Fedak, G., Germain, C., Herault, T., Lemarinier, P., Lodygensky, O., Magniette, F., Neri, V., Selikhov, A.: MPICH-V: Toward a scalable fault tolerant MPI for volatile nodes. In: Proceedings of the 2002 ACM/IEEE Supercomputing Conference, pp. 1–18 (2002)
2. Louca, S., Neophytou, N., Lachanas, A., Evripidou, P.: MPI-FT: Portable fault tolerance scheme for MPI. Parallel Processing Letters 10(4), 371–382 (2000)
3. Agbaria, A., Friedman, R.: Starfish: Fault-tolerant dynamic MPI programs on clusters of workstations. In: 8th IEEE International Symposium on High Performance Distributed Computing, pp. 167–176 (1999)
4. Rao, S., Alvisi, L., Vin, H.: Egida: An extensible toolkit for low-overhead fault tolerance. In: 29th International Symposium on Fault-Tolerant Computing (FTCS-29), pp. 48–55 (1999)
5. Bronevetsky, G., Marques, D., Pingali, K., Stodghill, P.: Automated application-level checkpointing of MPI programs. In: ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming (PPOPP), pp. 84–94 (2003)
6. Rodríguez, G., Martín, M., González, P., Touriño, J.: Controller/precompiler for portable checkpointing. IEICE Transactions on Information and Systems E89-D(2), 408–417 (2006)
7. Martín, M., Singh, D., Mouriño, J., Rivera, F., Doallo, R., Bruguera, J.: High performance air pollution modeling for a power plant environment. Parallel Computing 29(11-12), 1763–1790 (2003)
8. González, P., Cabaleiro, J.C., Pena, T.F., Rivera, F.F.: Dual BEM for crack growth analysis in distributed-memory multiprocessors. Advances in Engineering Software 31(12), 921–927 (2000)
9. Elnozahy, E., Alvisi, L., Wang, Y., Johnson, D.: A survey of rollback-recovery protocols in message-passing systems. ACM Computing Surveys 34(3), 375–408 (2002)
10. National Center for Supercomputing Applications: HDF5: File Format Specification [last accessed May 2007] `http://hdf.ncsa.uiuc.edu/HDF5`
11. Gailly, J., Adler, M.: Zlib home page [last accessed May 2007] `http://www.zlib.net`
12. Li, K., Naughton, J.F., Plank, J.S.: Low-latency concurrent checkpointing for parallel programs. IEEE Transactions on Parallel and Distributed Systems 5(8), 874–879 (1994)
13. Carmichael, G., Peters, L., Saylor, R.: The STEM-II regional scale acid deposition and photochemical oxidant model - I. An overview of model development and applications. Atmospheric Environment 25A(10), 2077–2105 (1991)