Paul Stodghill
December 8, 2003

GridCPR
Request for Comments: GWD-XXX-00x-1
Obsoletes: number
Category: informational

Use Cases for Grid Checkpoint and Recovery

Status of This Memo

This memo provides information to the Grid community regarding use case scenarios for Grid Checkpointing and Recovery. It does not define any standards or technical recommendations. Distribution is unlimited. This is a DRAFT document and continues to be revised.

Abstract

This document describes a number of use cases or scenarios to be addressed by the Grid Checkpoint and Recovery Working Group (GridCPR WG). The scenarios are also used to determine a set of requirements for these standards.

Full Copyright Notice

Intellectual Property Statement

Contents

1 Introduction

One of the goals of the Grid Checkpointing and Recovery Working Group (GridCPR WG) is to,

> define a user-level API and associated layer of services that will permit checkpointed jobs to be recovered and continued on the same or on remote Grid resources. [22]

In order to understand the requirements that these API and services must meet, it is necessary to understand the situations in which they will be used.

The purpose of this document is to enumerate usage scenarios that the GridCPR WG have decided must be addressed by its specifications. These scenarios will be used to derive a set of requirements for the GridCPR API and services.

2 Fault-Tolerance

2.1 Introduction

One of the most common reasons for using checkpointing and recovery is to provide some amount of fault-tolerance or failure recovery to an application.

There are two primary customers in this scenario, the system provider and the application user. The *system provider* is the individual or institution that is providing a computational resource (e.g., workstation, cluster, supercomputer) for other parties to use. The *application user* is the party that is interested in using this resource to run an application.

Suppose that some sort of quality of service (QoS) agreement has reached between the two parties before the application user submits the application for execution. Suppose further that this agreement requires that the application user receive a refund (either complete or partial) in the event that the computational resource fails during the application execution.

2.2 Terminology

Before proceeding further, it is necessary to describe the *fault model* that we wish this scenario to address. First, we will assume that only hardware failures are addressed by this scenario. In particular, we will not attempt to detect or handle application bugs or hacker attacks.

Second, we will assume that these failures can be modeled as *fail-stop* failures [18][1]. We do not consider the more general class of Byzantine failures[2].

Third, we will assume that, once any part of the computational resource has failed, it is impossible for the application to continue execution without initiating some global form of recovery. That is, we do not consider scenarios where the application continues executing on fewer processor nodes[3].

Instead of enumerating a single or small set of scenarios, we will consider a class of fault-tolerance scenarios. The basic structure of these scenarios is as follows,

Checkpointing Periodically while the application is running, the state of the application is saved to stable storage.

Failure Processing In the event of failure, the application will be resubmitted to a computational resource for further execution.

Recovery Upon re-execution, the state of the application is restored from the stable storage, and the application resumes execution.

We will explain each of these dimensions in turn.

2.3 Checkpointing

There are a number of different techniques that can be used to save an application state to stable. For sequential applications, there are basically two approaches. In the case of *System-Level Checkpointing (SLC)*, the applications memory image is captured by some low-level mechanism (e.g., an operating system service) [25, 1, 17]. In the case of *Application-Level Checkpointing (ALC)*, the state-saving is explicitly part of the application [7]. For instance, the application source code may contain functions to save critical program variables and structures.

Regardless of whether an SLC or ALC approach is used, the state saving functionality can be manually added the application by the application programmers, or can be added automatically using either compiler-based [4], library-based [19], or operating system-based [1] approaches.

In parallel applications, it is necessary to obtain a consistent global snapshot of the each processor and the state that is shared between them (e.g., messages in flight, shared global memory). For data-parallel applications, very often, it is possible to save all of the application state with respect to a global or application point of view [21]. For other styles of parallelism (i.e., SPMD, MIMD), more complicated protocols may be required in order to ensure that a correct snapshot is saved. See [8] for a survey of these techniques.

MPICH-GF [15] is a version of MPICH that provides transparent SLC and a number of protocols for handling the state of the MPI library. The C^3 provides a layer between the application and the native MPI that provides similar functionality for ALC [3, 5].

2.4 Failure Processing

The first requirement for processing a failure is to recognize that a failure has occurred. Very often, either the application user or system provider can manually determine when a failure has occurred by monitoring the computational resource. There are a number of heartbeat mechanisms [11, 10, 6] that have been proposed for detecting fail-stop failures automatically and providing notification services.

Once a failure has been detected, the application must be resubmitted to a computation resource. For this scenario, we will assume that the application is resubmitted to the same (albiet fixed) computational resource. Issues involved with recovering on a different computation resource are discussed in Section 3.

[1]Certain non-application software failures, such as operating system failures, may appear and be treated as hardware failures using this model.

[2]Certain hardware failures, such as transient network failures or memory corruption, may not cause any processor to fail and stop. However, as long as these failures can be detected, they can handled by causing the processor to shutdown.

[3]It may be possible for the application to *resume* execution on fewer nodes.

Application resubmission can be done manually, either by the application user or system-provider, or automatically by some external services, such as a job manager.

2.5 Recovery

Once the application has started executing again after resubmission, its state must be restored to that saved during the Checkpointing phase. The details of this are dependent upon how the checkpoint was taken in the first place (e.g, SLC or ALC, sequential or parallel). However, one requirement of these approaches is that the state that was saved during the Checkpointing phase must be available during the Recovery phase.

2.6 Example systems

The checkpoint system for the Pittsburgh Supercomputer Center's Terascale Computing System [23] allows for the automated recovery of jobs following both machine failures and scheduled maintenance periods. As an added feature, this system allows that any time lost by the user process because of machine failure between the time of the failure and the time of the last checkpoint can be automatically credited back to their allocation.

Checkpointing in the European DataGrid [9] is used to provide some form of fault-tolerance to applications, which is particularly important for long-running applications, such as those in High Energy Physics. In this system, the application developer is responsible for determining the application state that must be saved and restored. The system is responsible for noticing failures and automatically resubmitting jobs for further execution.

MPICH-GF [15] supports user-transparent fault tolerance of MPI applications running within a homogeneous computing environment. MPICH-GF is provided as a library that is linked with the unmodified application code. The system provides checkpointing and message logging of the application and a job management system that monitors the application, periodically sends checkpoint signals to the application, and restart the application if a failure occurs.

3 Process Migration

There are situations where it is necessary to migrate a running application off of a computational resource. This can be because, an application is nearing the end of its batch allocation, an application with higher priority is preempting the use of the resource, or because a more appropriate resource has been found to execute the application. In any case, the application user would like to save the current state of the application in order to resume execution at some point in the future on the same or different computational resource.

In this scenario, unlike the Fault-Tolerance scenario described in Section 2, the application needs to be checkpointed only once[4].

Another difference from the Fault-Tolerance scenario is that the application may be resumed on a different machine than its initial execution. This new machine may have a different number of processing nodes and a different processor architecture. "Portable" checkpoints can be generated using over-decomposition and by using architecture independent encoding methods (e.g., XDR [24] and HDF5 [12]).

Over-decomposition is a technique whereby an application divides its work into more units than there are physical processors. That is, if an application is run on P processors, than instead of dividing its work into P units, it may divide it into cP, where c is a factor greater than 1. While this approach can introduce some amount of overhead to an application's execution, it does provide flexibility in how work units are assigned to processors. For example, if the application is restarted on Q processors, then work units can be repartitioned so that each processor is assigned cP/Q units.

The ways of implementing checkpointing and resumption for process migration are similar to those discussed in Section 2.

Condor [17] provides transparent migration of sequential processes in a homogeneous computing environment. Dome [2] is an example of a application framework that enables heterogeneous checkpointing and process migration via over-decomposition. AMPI [14] is a runtime system that enables an application to automatically migrate between machines with a different number of processing nodes. PORCH [20] is an example of a compiler system that enables an

[4]This assumes that the application has sufficient time to save its state before it is terminated. If this is not the case, then preemption can be treated as a failure as in Section 2.

application to automatically migrate between machines with different architectures. All of these systems are potential customers of the API's by the GridCPR WG.

4 Debugging

Checkpointing and recovery can be used for debugging application programs. One example of how this can be done is with *replay debugger* [26], which enable the developer to run an application apparently in reverse. This is done by periodically checkpointing the application. When the developer wishes to run the application in reverse, the application is restored to the most recent checkpoint and then allowed to run forward until it reaches the relevant breakpoint.

Another example of how checkpointing can be used in debugging can be found in systems for debugging parallel programs. In this case, a sequential version of the application is run and checkpoints are taken at certain "breakpoints" in the application. Then, a parallel version of the application is run until the computation diverges from the sequential execution. The parallel application can then be run in reverse in order to isolate where the bug occurred.

An example of a debugging system that uses checkpointing is the O'Caml debugger [16]. Checkpointing is also being added to the P2D2 debugging system [13].

5 Functional Requirements for GridCPR

A Grid Checkpoint and Recovery system will consist of a number of services that provide the functionality. These services will be enumerated and described in a future architecture document. However, in order to build portable applications and tools that use these services a number of features of these systems must be standardized.

In this section we identify the requirements that are imposed by the preceding scenarios.

Checkpoint Storage A necessary requirement of this scenario is a means to create, write, read and destroy checkpoints on stable storage. This ability must be provided in the form of an API so that the application programmer can exploit it directly (manual ALC). This API may also be used by automatic checkpointing tools (automatic ALC and SLC). This API must enable each of the various forms parallel checkpointing (data-parallel, uncoordinated, block and non-blocking coordinated).

The implementation of this API must provide certain QoS guarantees to the application user and system provider. For instance, the system provider should be able to specify that checkpoint files are stored on machines that are not part of the computation resource. The application user should be able to query the system to determine what guarantees are in place.

If an application is so designed, it must be able to resume on a machine that is different, both in terms of number of processors and architecture, than the machine on which the checkpoint files were created. This means that it must be possible to transport checkpoint files between these machines.

Non-requirement: The API need not define a particular encoding of the data in the checkpoint files. In the case of heterogeneous checkpointing, the API is not responsible for generating "portable" checkpointing files. That is the responsibility of the application or checkpointing library.

Non-requirement: The API does not have to provide for mechanisms for initiating checkpoints or for specifying how the checkpoints should be taken. The details of the checkpointing process (ALC vs. SLC, etc.) are left to the application or to other automatic tools.

Checkpoint Meta-data It must be possible to annotate checkpoint files with meta-data. For instance, it must be possible to examine stable storage and determine which set of files constitute a complete snapshot of the application. Also, certain non-blocking coordinated checkpoint protocols for parallel applications require that a set of checkpoint files first be created and then separately be marked as "complete" once the protocol terminates.

Failure Detection Some standardized API should be defined for failure detection and notification. This is not say that Failure Detection is a requirement of a GridCPR-compliant implementation. Rather, it is to the GridCPR WG should provide a standard API that developers may choose to implement.

This API should provide mechanisms for receiving notifications when failures are detected, and for allowing the application to notify the service in the event that it detects a failure. The notification system should differentiate between system detected failures and application initiated failures.

Other requirements *fixme: Should we say something about requirements for*

- *job scheduling?*
- *checkpoint transport?*

6 Security Considerations

Authentication, Authorization, and Accounting (AAA) must be provided for the application state that is saved to stable storage.

The application user must be prevented from initiating failures for their own advantage (e.g., in order to obtain refunds for failures).

7 Performance Considerations

Systems that implement the API's described above should impose as small an overhead on the application as possible. They should also endeavor to provide good scalability.

An implementation that uses the Checkpoint Storage API should only observe an overhead from the system when it calls the functions for manipulating checkpoint files. For instance, the Checkpoint Storage system should not significantly slow down an application that never takes a checkpoint.

8 Editor Information

Paul Stodghill, Department of Computer Science,
Upson Hall, Cornell University, Ithaca, NY, 14853, USA
Phone: 607-254-8838 Email: stodghil@cs.cornell.edu

Contributers

We wish to thank the following for contributing use cases and text for this document,

- Robert Hood, NASA
- Thilo Kielmann, Vrije Universiteit
- Massimo Sgaravatto, INFN Padova
- Paul Stodghill, Cornell University
- Nathan Stone, Pittsburgh Supercomputing Center
- Heon Y. Yeom, Seoul National University

Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant No. 0085969. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

References

[1] A. Barak and O. La'adan. The MOSIX multicomputer operating system for high performance cluster computing. *Journal of Future Generation Computer Systems*, 13(4-5):361–372, March 1998.

[2] Adam Beguelin, Erik Seligman, and Peter Stephan. Application level fault tolerance in heterogeneous networks of workstations. *Journal of Parallel and Distributed Computing*, 43(2):147–155, 1997.

[3] Greg Bronevetsky, Daniel Marques, Keshav Pingali, and Paul Stodghill. Automated application-level checkpointing of MPI programs. In *ACM Symposium on Principles and Practice of Parallel Programming (PPoPP 2003)*, 2002.

[4] Greg Bronevetsky, Daniel Marques, Keshav Pingali, and Paul Stodghill. C^3: A system for automating application-level checkpointing of MPI programs. In *The 16th International Workshop on Languages and Compilers for Parallel Computers (LCPC'03)*, October 2003.

[5] Greg Bronevetsky, Daniel Marques, Keshav Pingali, and Paul Stodghill. Collective operations in an application-level fault tolerant MPI system. In *International Conference on Supercomputing (ICS) 2003*, San Francisco, CA, June 23–26 2003.

[6] The DataGrid Project. Gdmp heartbeat monitor. http://project-gdmp.web.cern.ch/project-gdmp/gdmp_hb/.

[7] G. Deconinck, J. Vounckx, R. Lauwereins, and J. Peperstraete. A user-triggered checkpointing library for computation-intensive applications, 1995.

[8] M. Elnozahy, L. Alvisi, Y. M. Wang, and D. B. Johnson. A survey of rollback-recovery protocols in message passing systems. Technical Report CMU-CS-96-181, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, USA, October 1996.

[9] Alessio Gianelle, Rosario Peluso, and Massimo Sgaravatto. Datagrid: Job partitioning and checkpointing. https://edms.cern.ch/document/347730, June 3 2002.

[10] The Globus Project. The globus heartbeat monitor specification v1.0. http://www-fp.globus.org/hbm/heartbeat_spec.html.

[11] I. Gupta, T. Chandra, and G. Goldszmidt. On scalable and efficient distributed failure detectors. In *Proc. 20th Annual ACM Symp. on Principles of Distributed Computing*, pages 170–179, 2001.

[12] Hdf5 - a new generation of hdf. http://hdf.ncsa.uiuc.edu/HDF5/.

[13] Robert Hood. P2d2: A portable distributed debugger. http://www.nas.nasa.gov/Groups/Tools/Projects/P2D2/.

[14] Chao Huang, Orion Lawlor, and L. V. Kale. Adaptive MPI. In *Languages and Compilers for Parallel Computers (LCPC)*, 2003.

[15] Sangbum Kim, Namyoon Woo, Heon Y. Yeom, Taesoon Park, and Hyoungwoo Park. Design and implementation of dynamic process management for grid-enabled MPICH. In *Proceedings of the 10th European PVM/MPI Users' Group Conference*, Venice, Italy, September 2003.

[16] Xavier Leroy. Re: [Caml-list] replay debugger. http://caml.inria.fr/archives/200110/msg00033.html, October 4 2001.

[17] M. Litzkow, T. Tannenbaum, J. Basney, and M. Livny. Checkpoint and migration of UNIX processes in the Condor distributed processing system. Technical Report 1346, University of Wisconsin-Madison, 1997.

[18] Nancy Lynch. *Distributed Algorithms*. Morgan Kaufmann, San Francisco, California, first edition, 1996.

[19] James S. Plank, Micah Beck, Gerry Kingsley, and Kai Li. Libckpt: Transparent checkpointing under UNIX. Technical Report UT-CS-94-242, Dept. of Computer Science, University of Tennessee, 1994.

[20] Balkrishna Ramkumar and Volker Strumpen. Portable checkpointing for heterogenous architectures. In *Symposium on Fault-Tolerant Computing*, pages 58–67, 1997.

[21] C. Wang S. Flynn Hummel, I. Banicescu and J. Wein. Load balancing and data locality via fractiling: an experimental study. In Boleslaw K. Szymanski and Balaram Sinharoy, editors, *Languages, Compilers and Run-Time Systems for Scalable Computers*, chapter 7, pages 85–98. Kluwer Academic Publishers, Boston, MA, 1996.

[22] Derek Simmel, Thilo Kielmann, and Nathan Stone. Draft charter v.1.1, Grid Checkpoint Recovery Working Group (GridCPR). Technical report, Global Grid Forum, February 20 2003. Available at http://forge.gridforum.org/projects/gridcpr-wg/document/Charter_Version_1.1/en/1.

[23] Nathan Stone, John Kochmar, Raghurama Reddy, J. Ray Scott, Jason Sommerfield, and Chad Vizino. A checkpoint and recovery system for the pittsburgh supercomputing center terascale computing system. http://www.psc.edu/publications/tech_reports/chkpt_rcvry/checkpoint-recovery-1.0.html, December 3 2003.

[24] Sun Microsystems, Inc. RFC 1014 - XDR: External data representation standard. Published by The Internet Engineering Task Force. Available at http://www.ietf.org/rfc/rfc1014.txt.

[25] Yuval Tamir and Carlo H. Sequin. Error recovery in multicomputers using global checkpoints. In *13th International Conference on Parallel Processing*, pages 32–41, Bellaire, MI, August 1984.

[26] Andrew Tolmach. *Debugging Standard ML*. PhD thesis, Princeton University, October 1992.