On the use of checkpoint/recovery in RealityGrid

Stephen Pickles

January 2004.

1. Introduction

The RealityGrid project (http://www.realitygrid.org) aims to predict the realistic behaviour of matter using diverse simulation methods spanning many time and length scales and the discovery of new materials through integrated experiments. A central theme of RealityGrid is the facilitation of distributed and collaborative exploration of parameter space through computational steering and on-line, high-end visualization [1, 2, 3, 4].

A typical RealityGrid scenario involves a large-scale simulation running on a massively parallel system at on site coupled to a high-end visualization system at another site with the steering and display interfaces running at one or more remote sites. All of these components can be started and stopped independently. The simulation component periodically (or as demanded by the steering client) emits “samples” for consumption by the visualization component. An OGSI-based middle tier, implemented in OGSI::Lite [5], facilitates bootstrapping of communication between components.

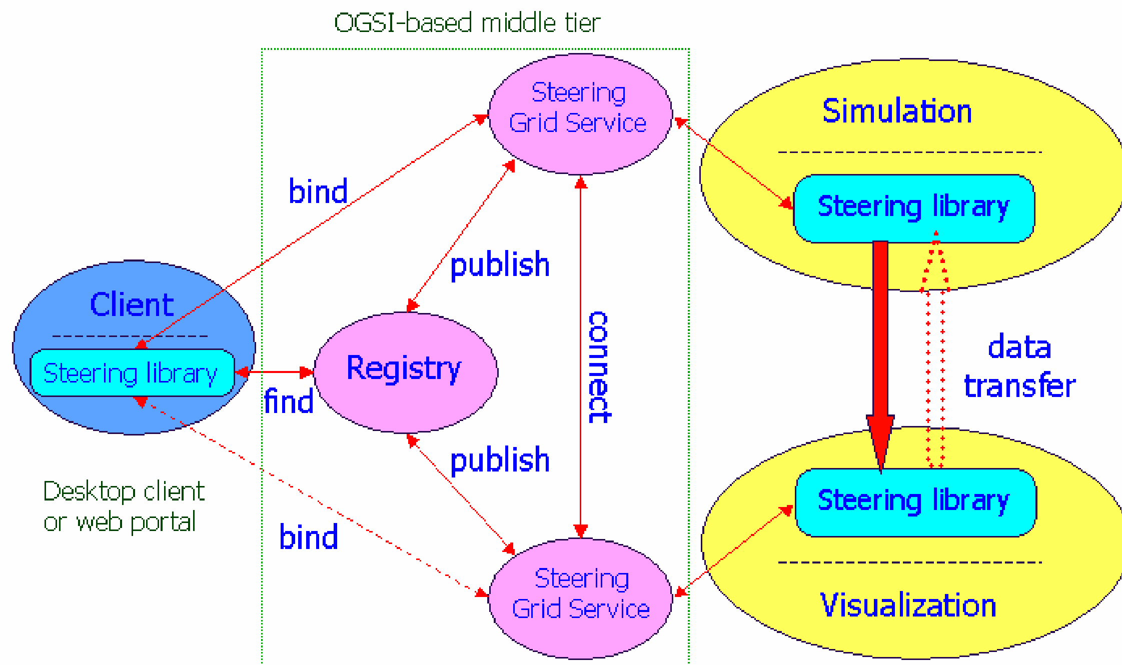

Figure 1. Computational Steering Architecture

2. Checkpoint/recovery

Checkpoint/recovery is used in RealityGrid for a number of purposes, as described below.

In all cases, the checkpoint is written by the application (*Application Level Checkpointing*), for reasons of portability and economy. We assume that the application developer knows best (a) the minimum amount of information necessary to describe completely the state of the application, and (b) at what points during execution it is practical to capture this information and write it to disc. We call these points *"safe points"*.

It is worth clarifying what we mean by the overloaded term "application". In this context, "application" means only the simulation component, which is typically a single parallel program. We do not checkpoint the visualization component, and the management of connections between components is handled "out of bands". However, there are occasions when it is convenient to speak of the distributed components (e.g. simulation, visualization, and steering client) as a single entity. In this document, we shall refer to such entities as "distributed applications".

RealityGrid is also concerned with simulations composed of smaller, independently compiled components. This finer-grained componentisation can arise naturally when different physical models are coupled together, or by the deliberate factoring of a legacy code into separate functional units. The implications of this for checkpoint/recovery are not fully understood by the author; however, he would hope that an application-level checkpoint can be achieved by checkpointing each component independently, and supplementing these independent checkpoints with information about the stateful interactions between the components.

2.1. Fault Tolerance

Large-scale simulations require both large numbers of processors and long run-times, both of which increase the probability of the simulation being affected by a system failure. The risk of losing work to such a failure can be managed by checkpointing the simulation periodically. There is a trade-off between the time required to take a checkpoint, the probability of a failure, and the cycles risked, which factors vary across platforms and problem sizes. Hence it is good practice to make the checkpoint frequency tuneable. It is also common practice to use a cycle of two (and very occasionally, more than two) checkpoints so that, if a failure occurs while a checkpoint is being taken, it is still possible to restart from the previous checkpoint in the cycle.

The use of checkpoint/recovery for this purpose is widespread.

2.2 Long computations and batch queue management

Administrators of HPC services try to ensure efficient overall usage of computational resources (in the presence of finite MTBF) at the same time as ensuring that groups of users with different workload characteristics get their fair share of the system's capacity. Usually, this leads to a compromise in which the batch queues are configured to impose a maximum duration on each job, with wall-clock limits of 12-24 hours being fairly common.

As distributed memory sizes of parallel systems continue to increase, computational scientists attempt to solve problems of ever increasing problem size. Despite continuing improvements in processor performance and wider availability of parallel systems, the time to solution of the largest problems is also increasing, due to the fact that the computational complexity of physical problems typically grows faster than the problem size.

It is therefore inevitable that many computations cannot run to completion inside the maximum job duration imposed through batch queue limits. Consequently, it is becoming increasingly commonplace for application codes to use checkpoint/recovery techniques to split a computation across several runs.

Given a self-checkpointing code, a user can submit a job, expecting it to be terminated by the system when the time limit is reached, but confident that the job can later be restarted from a checkpoint. In a slightly more sophisticated scenario, the user causes the application to run for a number of iterations which can be completed inside the allotted time, avoiding the mess of untidy termination and the wastage of cycles beyond the last checkpoint. But it is not always practical to determine beforehand how long a certain number of iterations will take. The author has worked on one code which knew the amount of wall-clock time available to it, and had enough intelligence to checkpoint and stop when insufficient time remained to complete another iteration. However, it is usually impossible to take scheduled down-time into account when deciding how many iterations to run.

A function to query the time remaining to the application could be used by sophisticated applications to determine when to checkpoint and stop. Implementations on certain platforms could use this to notify applications of imminent termination.

2.3. Process migration

RealityGrid's design philosophy is component based. Although the componentisation in the "fast track" example of figure (1) is coarse-grained (the simulation and visualization components are deployed on different systems), the RealityGrid "deep track" is investigating finer-grained componentisation in which the simulation is composed out of a number of smaller communicating components, each of which must be deployed onto (possibly remote) computational resources at run-time. RealityGrid has a significant "deep track" work package devoted to performance control. The goal of this work

package is to optimise the collective performance of the components comprising a distributed application based on performance information collected at run time. Initially, the set of resources will be assumed to be fixed during execution, and it is by redistributing components across this set of resources that the performance control system hopes to achieve performance improvement. Ultimately, however, the ambition is to adapt to utilize new resources that become available during execution.

In the performance control system, the redistribution is achieved by checkpointing each component of a distributed application, transferring its checkpoint files, and restarting it. The checkpoints must be "malleable", by which we mean that a job initially running on N processors can be restarted on M processors. The checkpoints must also permit restarting on a different architecture. One RealityGrid application (LB3D) currently has this capability; the techniques are being generalised to suit a broader class of application.

2.4. Computational steering and checkpoint trees

Computational steering is the ability to interact with, and change the behaviour of, a running application. In RealityGrid, an application is instrumented for computational steering through the RealityGrid steering library [6]. The "knobs" and "dials" of the application are exposed as operations of an OGSI-compliant "Steering Grid Service", and controlled by remote users through a graphical client tool or web-based portal (see Figure 1). A fully instrumented application supports the following operations:

- Pause/resume
- Set values of steerable parameters
- Report values of monitored (read-only) parameters
- Emit "samples" to remote systems for e.g. on-line visualization
- Consume "samples" from remote systems for e.g. resetting boundary conditions, and
- Checkpoint and windback.

"Windback" here means revert to the state captured in a previous checkpoint without stopping the application. Not all applications that support checkpoint necessarily support windback, which can sometimes be awkward to implement. However, the same effect can be achieved, at some overhead, by stopping the application and restarting it from the required checkpoint. In RealityGrid, the act of taking a checkpoint is the responsibility of the application, which is required to register each checkpoint with the library and to provide a tag to identify each one.

We see checkpoint/recovery as being a key piece of functionality for computational steering, useful in a wide variety of scenarios. Sometimes the scientist realises that an interesting transition has occurred, and wants to study the transition in more detail; this can be accomplished by winding back the simulation to an earlier checkpoint, and increasing the frequency of sample emissions for on-line visualization. Similar techniques can be employed when testing a new algorithm; often, the coarse-grain control provided by checkpoint-enhanced computational steering is a more convenient way of reaching the point where things start to go wrong than is stepping through the execution

with a parallel debugger. An even more compelling scenario arises when computational steering is used for parameter space exploration [3, 4].

A scientist may be studying a physical system which is suspected to contain a rich phase structure, but does not have sufficient resources available to embark on a brute-force exploration of its multi-dimensional parameter space. Instead, the scientist uses computational steering to begin mapping out the parameter space. Starting from some random initial condition, a simulation evolves under an initial choice of parameters until the first signs of emergent structure are seen, and a checkpoint is taken. The simulation evolves further, until the scientist recognises that the system is beginning to equilibrate, and takes another checkpoint. Suspecting that allowing the simulation to equilibrate further will not yield any new insight, the scientist now rewinds to an earlier checkpoint, chooses a different set of parameters, and observes the system's evolution in a new direction. In this way, the scientist assembles a tree of checkpoints[*] that sample different regions of the parameter space under study, while carefully husbanding his or her allocation of computer time. The scientist can always revisit a particular branch of the tree at a later time should this prove necessary. This process is illustrated in Figure 2, in which a Lattice-Boltzmann simulation is used to study the phase structure a mixture of fluids. Here the one-dimensional parameter space is explored by varying the coupling constant $\mathbf{g}_{ss}$.

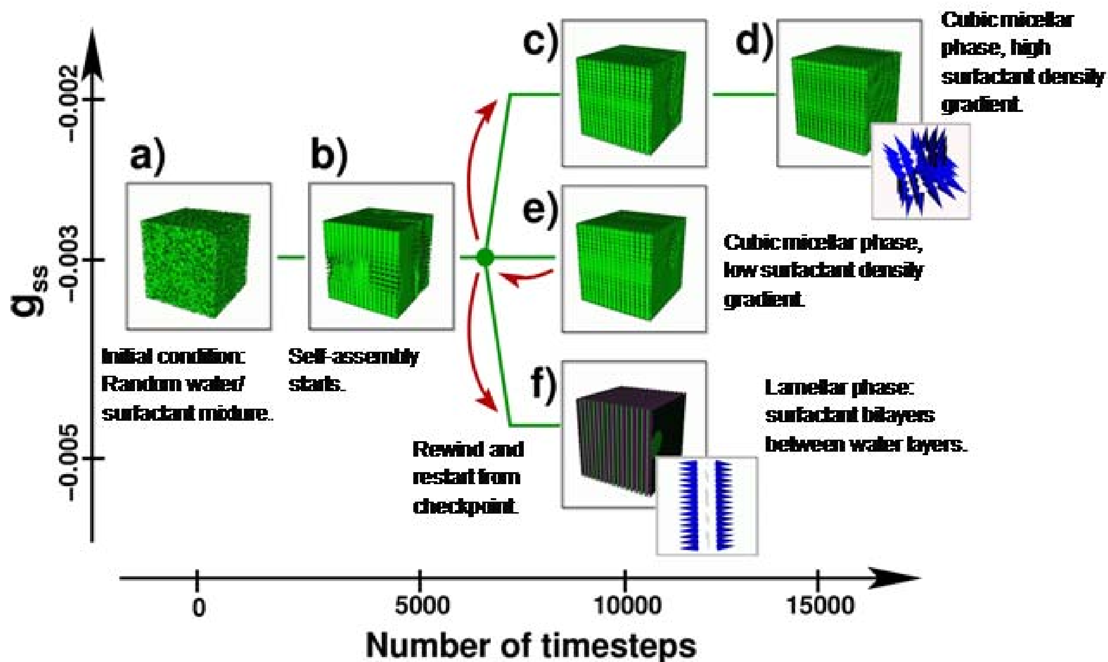

Figure 2. Parameter space exploration gives rise to a tree of checkpoints.

This exploration of parameter space can be conducted in various ways. If more than one user is involved, there are implications for access control to checkpoint data and

[*] Our use of checkpoint trees was inspired by GRASPARC [7].

metadata. If more than one computational resource is involved, then transfer of or remote access to checkpoint data and metadata is required. Unless the exploration is completed in a single steering session, then persistence of checkpoint data and metadata is also required; as checkpoints can be large, it is unreasonable to demand that checkpoint data persist indefinitely, so the ability to manage checkpoint metadata is indicated. If checkpoint files are physically copied, there arises the question of whether the location of replicas of checkpoint files should be tracked and managed. If the possibility exists that a checkpoint can be copied from one location to another while the application that created the original checkpoint is still running and liable to overwrite it — this is likely when different branches of the tree are being explored in parallel — then mechanisms are required to prevent simultaneous read and write access to the same checkpoint.

3. Considerations for a checkpoint/recovery system.

Checkpoints are used for a number of distinct purposes in RealityGrid. The same checkpoint can be used for more than one purpose, by more than one user, on more than one system, in more than one administrative domain, over an extended period of time. The possibility of simultaneous access by different runs of the application is real.

It is unreasonable to expect a generic checkpoint/recovery system to address all of the issues arising out of the interactions between different uses of checkpoint in RealityGrid. However, RealityGrid could use a simple checkpoint/recovery system of well-defined scope when building more complex systems that rely on checkpoint/recovery.

References

[1] J. M. Brooke, P. V. Coveney, J. Harting, S. Jha, S. M. Pickles, R. L. Pinning and A. R. Porter, *Computational Steering in RealityGrid*, Proceedings of the UK e-Science All Hands Meeting, September 2-4, 2003 (http://www.nesc.ac.uk/events/ahm2003/AHMCD/pdf/179.pdf)

[2] J. Chin, J. Harting, S. Jha, P. V. Coveney, A. R. Porter and S. M. Pickles, *Steering in computational science: mesoscale modelling and simulation*, Contemporary Physics 44, 417-434, 2003 (http://taylorandfrancis.metapress.com/openurl.asp?genre=article&eissn=1366-5812&volume=44&issue=5&spage=417)

[3] Stephen M. Pickles, Peter V Coveney and Bruce M Boghosian, *Transcontinental RealityGrids for Interactive Collaborative Exploration of Parameter Space (TRICEPS)*, Winner of SC'03 HPC Challenge competition in the category "Most Innovative Data-Intensive Application", (http://www.sc-conference.org/sc2003/inter_cal/inter_cal_detail.php?eventid=10701#5)

[4] Mathilde Romberg, John Brooke, Thomas Eickermann, Uwe Woessner, Bruce Boghosian, Maziar Nekovee, Peter Coveney, *Application Steering in a Collaborative Environment*, SC Global conference, November, 2003 (http://www.sc-conference.org/sc2003/inter_cal/inter_cal_detail.php?eventid=10719). A Windows

Media Stream archive of this session is available at mms://winmedia.internet2.edu/VB-on-Demand/AppSteering.asf. It is recommended that this file be viewed using Windows Media Player version 9.

[5] Mark Mc Keown, *OGSI::Lite – a Perl implementation of an OGSI-compliant Grid Services Container*. (http://www.sve.man.ac.uk/Research/AtoZ/ILCT)

[6] Stephen Pickles, Robin Pinning, Andrew Porter, Graham Riley, Rupert Ford, Ken Mayes, David Snelling, Jim Stanton, Steven Kenny, Shantenu Jha, *The RealityGrid Computational Steering API*, Version 1.0, 9 July 2003, unpublished.

[7] K. W. Brodlie, L. A. Brankin, G. A. Banecki, A. Gay, A. Poon and H. Wright. *GRASPARC: A problem solving environment integrating computation and visualization.* In G. M. Nielson and D. Bergeron, editors, *Proceedings of IEEE Visualization 93 Conference*, p. 102. IEEE Computer Society Press, 1993.