Introduction to Checkpointing for Capability Computing

⊙ From the Scientific Computing Resource Center
Information Technology Laboratory
US Army Engineering Research & Development Center

⊙ Document Purpose	To provide a quick start for adding warm restart capability to applications on Garnet. To provide background information and motivation for HPC software-level fault tolerance.
Last Edit:	July 10, 2013
Referenced Documents:	"Using Advanced I/O on Garnet"

⊙ This document addresses the need for warm-restart functionality in software running on Garnet.

The *primary reasons* for adding restart functionality to parallel applications are to:

1. Provide fault tolerance in the face of node failures or other system level errors, which become more likely as jobs run longer or on more nodes.
2. Strengthen the software for future increases in parallel width of jobs.
3. Provide roll-backs points which may help in deducing sources of computational errors.

A *secondary reason* reason for warm restarts applies only to a batch scheduled shared resource. In that case, the maximum job runtime may be lower than the time needed to complete computation. Software must checkpoint its state at least once toward the end of the time allotment, then restart in a new job.

This document was originally created as part of a larger effort to transition the usage of the shared supercomputing resource Cray XE6, Garnet, toward capability class computing. That process involves changes in the management of the system and changes in the user view of this system's primary function as a scientific instrument.

Checkpoint & Restart

There follows a crash course in warm-restart for HPC applications. Section 1 contains answers to some common questions. Next, section 2 addresses users who have software with checkpointing functionality already implemented, providing guidelines for setting up intervals and controlling I/O.

Section 3 addresses users who need to assess their software to determine if it warrants updating with warm restart functionality for the Garnet environment. Finally, section 4 provides a high-level guide for users wanting to add basic coordinated checkpointing to an HPC application.

Contents

1 FAQ

⊙ **What is warm-restart?**

A : The ability of a piece of software to carry out a single computation over several executions. During execution, some amount of computational state is stored to disk. The application halts. Later, the application launches again, and restarts where it left off by reading data from disk.

⊙ **Are there other terms for warm-restart?**

A_1 : Checkpoint-restart indicates warm restart that uses checkpoint files, which are saved on some regular basis throughout execution.

A_2 : Serialization/Deserialization is a term often used in object oriented paradigms, like Java, Python, or C++. It refers to saving objects to files (serialization) and recovering them into object data structures (deserialization). Warm restart can be implemented in a pure OO paradigm by serializing then deserializing all instantiated objects.

↓ Other terms for serialize and deserialize
→ pickle and unpickle; marshal and unmarshal; deflate and inflate

⊙ **What is capability computing?**

A : A term used to describe high performance parallel computations that specifically require or benefit from hardware capabilities not found on any other platform.

⊙ **Why is warm-restart currently a software necessity on capability class systems like ORNL's Titan, the Blue Waters project, and ERDC's expanded Garnet?**

A_1 : High core count jobs or very long job runs are more likely to see hardware or software failures at runtime. Saving state for restart allows the computation to rollback after a failure – rather than start over – allowing computations to make progress.

A_2 : Shared resource scheduling algorithms cannot guarantee large jobs a reasonable time in queue if there are many long running small jobs on the system. By restarting regularly, long running small jobs allow the entire system to function more effectively

⊙ **I have never used warm-restart before. What are the major pitfalls to avoid?**

A_1 : Saving state for a restart requires significant I/O overhead, which typically increases with job size. Be careful not to checkpoint too often. Every 4-8 hours is sufficient for most jobs. Examine options for file formats and I/O functions to determine which are most efficient for your application and job size(s).

A_2 : Restart protocols should be carefully followed. Improperly restarting a computation – such as restarting on a different core count when the software does not support it – will lead to incorrect results or application crashes.

⊙ **I found warm-restart solution X available on-line or in the literature. Can we deploy it on Garnet?**

A : If this is a solution that requires alterations to Linux kernel, MPI libraries, compilers, or otherwise privileged execution, then no. Our HPC systems may include some of these solutions in the future, but there is not a mature option available to us at this time. If the solution can be compiled and run with user level privileges, then it follows the same rules as any other user software on Garnet.

⊙ **Are there other solutions to fault tolerance and scheduling that don't require checkpointing and restarting? Can we deploy these on Garnet?**

A : Yes there are, but it is highly unlikely these solutions can be deployed on Garnet. There are many solutions to fault tolerance and scheduling problems. Both are very active areas of research in high performance computing. But in computing, and especially HPC, there is no free lunch. Any solution will have some drawback or cost. Users are encouraged to talk to other HPC programmers around the country (and world), to keep up with current best practices and system limitations.

2 Instructions for users with applications that implement checkpoint/restart

Many applications, especially those developed for an HPC environment, already include warm restart functionality. The section describes how to enable the most common type of warm restart, which is interval based checkpointing.

Software that implements some other form of warm restart will still require the same basic procedure: verify correct restart, then measure overheads. If you have a non-checkpoint based system, contact your software maintainer for help in determining overheads.

2.1 Check the documentation (and possibly source code)

The LAMMPS MD Simulator, for example, has several commands for creating and reading from restart files. Software documentation should contain some information about how to enable and control the warm restart process.

Since HPC programmers are often in a hurry to keep up with their machines, documentation may not do as it should. So, it is always good to check the source code directly, when possible, if there is any doubt about what a given option does.

2.2 Enumerate configuration options

A warm restart implementation may have several different options for file formats and I/O methods. There may also be options to keep multiple checkpoints rather than always writing over the most recent checkpoint.

Looking closely at configuration options will show what your software can do, but more importantly it will provide insight into why the options exist. Typically, options exist for customizing trade-offs between I/O overheads, portability, level of checkpoint detail, ease of use, and file format conventions.

If your software uses a checkpoint-restart system, you will need to specify a checkpoint interval. Shorter intervals increase overheads and reliability. Section 2.5 explains how to choose an optimal interval based on job size. That calculation requires a measure of the checkpoint overheads, as explained in section 2.4.

2.3 Verify correct restart

Using the enumerated options, experiment with some configurations that seem right for your computations. Compare the output of computations that run all the way to completion with output from restarted computations.

Remember that a restarted computation may not have the same output as a run to completion, but still be correct. Random number generation is often a source of differences in restarted computations, when the generators are reseeded on restart. In this case, verifying correct restart requires validating that the final result makes sense based on initial inputs.

2.4 Determine overheads

Measuring overheads for warm restarting your computation can be done by simple experiment. Run the application with and without warm restart enabled and measure the difference in runtime. If using a checkpoint system, divide by the number of checkpoints made to get the individual checkpoint overhead.

With this same method, you can also get a true measure of I/O overhead by profiling the application with and without warm restart. I/O profiling is outside the scope of this document.

Checkpoint overhead metrics:

T_0	runtime without checkpointing
T_1	runtime with checkpointing
N	number of checkpoints created
C_0	checkpoint overhead
S	time to make single checkpoint

$$T_1 = T_0 + C_0$$

$$C_0 = N * S$$

2.5 Set a reasonable checkpoint interval

Checkpointing every 2 or 4 hours is common among very large ($> 2K$ nodes) computations, to account for the higher risk of losing a physical node during the computation. Smaller jobs have lower risks, and can therefore checkpoint less frequently to save space and time. Assuming your application never fails, a single checkpoint at the expected end of the run is fine...

Regular checkpoint intervals are highly recommended for jobs larger than 128 nodes.

If unsure, choose a 4 hour checkpoint interval.

The following calculations use NODE count, NOT core count.

The following example calculates the optimal interval for a job on Garnet, using Daly's first order model[1] with a mean time to failure based on job size.

Optimal checkpoint interval using Garnet node count:

S	minutes time to make single checkpoint
M	1,000,000 / node count
R	minutes restart overhead

$$\tau = \sqrt{2 * S * (W + R)}$$

For example, an application that runs on 256 nodes, takes 5 minutes to make a single checkpoint, and takes 10 minutes to restart would have an optimal interval of:

$$\tau = \sqrt{2 * 5 * (1,000,000/256 + 10)} = 197 \text{ minutes or } 3.2 \text{ hours}$$

An application that runs on 2 nodes, takes 1 minute to make a single checkpoint, and takes 1 minute to restart would have an optimal interval of:

$$\tau = \sqrt{2 * 1 * (1,000,000/2 + 1)} = 1000 \text{ minutes or } 16.6 \text{ hours}$$

2.6 Implement your target interval as best you can

Most application level checkpointing occurs at logical intervals, not wallclock intervals. For example, in a long running iterative solver, a logical interval could be to checkpoint every N iterations.

Working backward from a target wallclock frequency, you will need to determine some rough timing for logical intervals. For the solver example above, observing roughly how many long an iteration takes to complete for a particular problem set will allow configuration of the checkpoint interval accordingly.

For more on this problem of logical intervals versus wallclock intervals, see section 4.2.

3 Instructions for users with applications that DO NOT implement checkpoint/restart

3.1 Don't panic

A computational state is just a bunch of stored data. A checkpoint is just a lot of data writes. A restart is just a lot of data reads. The only complication is the pursuit of efficiency. Therefore, quickly adding some basic checkpointing may be very straightforward, if somewhat inefficient.

3.1.1 Determine whether your computations need a capability hardware solution

Before launching into a code revision, it is important to know whether such changes are even necessary. This section continues under the assumption that this software requires the hardware resources on Garnet, will be running on Garnet or other capability systems in the future, and therefore requires modification to provide warm restart functionality.

3.1.2 Determine your software development potential

Depending on your project, specific software, and team members, the level of difficulty for implementing checkpointing or other improvements will vary greatly. As above, the only real way to know if implementing changes will be worth the effort is to assess your particular needs and situation. That being said, here are a few questions to ask about the difficulty of altering your software:

↓ **Difficulty Level:**

⊙ Does your software have a support team/maintainer who is willing to implement checkpointing or at least assist with implementation?

⊙ Does your team/project contain people who have implemented checkpointing, or something similar, on another project?

⊙ Does your team/project contain people who have worked directly with the source code for some amount of time?

⊙ Is the source code version controlled?

⊙ Does the source code contain less than 100K lines?

⊙ Is the source code well structured such that you can easily read it?

⊙ Is the software build system well structured such that you can easily modify build activities and options?

→ Each answer of no to the previous questions increases the difficulty level.

→ If the answer to at least 2 of the last 6 questions is "yes", then taking an active role in the check-pointing implementation is very advisable, even if it is to be done by an external maintainer. It will speed up the process on both ends and improve your ability to use/debug the resulting code changes. The faster the code can be modernized, the sooner it catches up, the sooner more interesting problems can take the fore.

3.2 Contact the development team if one exists

When you contact the developer(s) of your software, explain your situation. Checkpoint/restart functionality is used in many different computer science contexts other than high performance computing. Even if your developers are not HPC developers, they may like the idea of code checkpointing for reliability enough to work with you to incorporate it. This will save you a great deal of time in the long run.

3.3 Contact the user community if one exists

Time to do what you aren't supposed to do for computer science homework assignments. Post your problem to mailing lists or forums. Other users may need or could really use checkpointing.

If nothing else, a forum for technical questions will be a valuable tool if you need to modify source code or build systems and run into questions about code logic or data sets.

3.4 See section 4

4 Implement Warm Restart

This section is *not* a step by step guide to adding a specific checkpointing tool or library to an application. Nor is it detailed information on checkpointing in general.

This section is a sequence of notes about how to apply general checkpointing concepts to a shared batch HPC environment.

> The process of adding warm restart to an application has two requirements:
>
> - Correctness: The warm restart must yield as valid a result as if the computation had run to completion.
> - Controlled I/O: Warm restart functionality must have predictable I/O overheads which are not excessive compared to available system resources.
>
> The second requirement is specific to using a shared resource which is sensitive to flooding I/O bandwidth.

The rest of this section describes a series of steps to take in order to meet those two requirements with coordinated checkpointing. Each subsection focuses on how *the HPC environment affects software restart requirements differently than normal computing environments*. More general information about warm restart should be obtained from external sources.

4.1 Survey Existing Tools and Methods

A quick peak at the Wikipedia entry for application checkpointing-restart suggests four attributes that describe a given checkpointing tool.

1. ***Amount of state saved:*** *In other words, how selective is the tool about what gets saved. Does it save the entire program state, or really narrow down what gets stored?*

   For massive distributed computations as used on Garnet, the amount of state saved should always be minimal and selective. If a tool tries to save, for example, the entire runtime execution stack for every MPI process in a 2K core job, the data transfer could easily be on the order of terabytes. This is too slow for reasonable checkpoint times and over-burdens the disk subsystem as well. Many checkpoint tools that work at scale were originally developed for use on HPC systems.

2. ***Automation level:*** *How much work is involved in getting this tool to work with a specific code. Does it require lots of source code modification, or just a wrapper script at compile time?*

   Unfortunately, more automation and bigger data size go together in many cases. The exception to this rule is tools targeting specific languages or libraries, which can be more intelligent in saving data. The sign of an "over-automated" tool is checkpoint file sizes that grow rapidly with job size, regardless of small or large working data set.

3. ***Portability, or Restart Consistency:*** *Can a checkpoint file from one machine be restarted on another? More relevant to our case: Can a checkpoint file for an N core job run be restarted in an M core job run, where $M \neq N$*

   Machine portability is not as big an issue for HPC checkpointing as it is for other networked/distributed computations. Instead, we have the issue of consistency in checkpoint vs. restart conditions. Restarting on different core counts can cause problems if misconfigured. In software design, complex dependencies between parallel threads may require checkpointing data in a specific order to maintain consistency. These details emerge in the process of validating that a restore operation always succeeds under a given set of conditions.

4. ***System Architecture: Does this tool operate at the OS kernel level, library level, compiler level?***

   This final attribute is very important for use on a shared HPC resource. Many existing checkpoint-restart tools require changes to OS kernel functions or running daemons. The only viable options for use on Garnet are tools in library form that be compiled and run by users from home, work, or project directories.

Given the above criteria, you can evaluate existing checkpoint-restart solutions that could be added into your applications. Issues surrounding open source licensing are outside the scope of this document. However, the Department of Defense has a very tolerant policy toward open source software, especially for scientific work. See section 5.1 for more information.

Unfortunately, prepackaged checkpoint tools may not have acceptable I/O requirements for a particular computation. Section 5.3 provides a few links to checkpointing tools that may work, but will need to be tested on an application by application basis. These tools are not installed or officially supported on Garnet in any way. They are provided as examples only. Compiling and using them is equivalent to compiling and using your own software in terms of support provided.

4.2 Locate logical checkpoint location(s)

Maybe some third party tool will work perfectly with your project, will not require modifying any source code, and will provide efficient I/O rates and file sizes. The following sections assume that there was *not* a perfect tool, and you need to manually add checkpointing, or modify an existing tool.

The subject of *independent or induced checkpointing is beyond the scope of this document.* In independent methods, checkpoint data are created incrementally on a per-process basis in response to observed communication dependencies. By contrast, the less sophisticated *coordinated checkpoint methods have all processes create globally unified checkpoints.*

This discussion assumes a programmer wanting to add a coordinated checkpointing feature to a distributed application in the following format:

```
: On all processes: For i = 1 to ...
   : Execute work for some interval N
   : Coordinate with all other processes to save checkpoint i
```

Section 2 talks about setting checkpoint intervals based on wallclock times and mean-time-to-node-failure. However, using timers for checkpointing is a complex problem. Aside from begin globally inconsistent, a timer may "go off" and initiate a checkpoint at a point in the computation where a great deal of data is being moved or worked on.

Saving and restarting when data is moving is much more difficult than saving at points in the computation where data is mostly at rest. Rather than use a timer, it is simpler to choose a logical point in the computation to add a checkpoint function.

For example, in a basic spatially decomposed parallel molecular dynamics simulation, we might have the following main loop:

```
: On all processes: For timestep = 1 to M
  : Calculate forces on particles.
  : Update particle positions based on force calculation.
  : Relocate particle data among processors based on new particle position (spa-
    tially decompose).
```

If the main loop has a global barrier at the bottom, then a simple checkpoint function can be inserted before calculate forces, followed by another barrier. Even if the main loop does *not* have a global barrier, each process could count logical iterations and save iteration tagged checkpoint files such that they could be re-combined in a consistent manner.

Clearly, not all parallel computations come down to a simple barrier loop. However, most MPI applications do fall into a generally fan-out, fan-in type of parallelism. Logical checkpoint locations occur just after the fan-in points, when a large amount of parallel work has just finished synchronizing.

Once the checkpoint function has been inserted, the application has to accept some parameter for a checkpoint interval, which will not be wallclock time. In the MD loop example, the application could take a parameter (N) for the number of iterations between checkpoints.

```
: On all processes:
: i = 1, j = 0
: For timestep = 1 to M
  : j = j + 1
  : If j == N
    : Global Barrier (first barrier coordinates checkpointing)
    : Save checkpoint i
    : Global Barrier (second barrier allows file overwrite, global hung I/O checks)
    : i = i + 1, j = 0
  : Calculate forces on particles.
  : Update particle positions based on force calculation.
  : Relocate particle data among processors based on new particle position (spa-
    tially decompose).
```

Coming back to interval selection, whoever sets up the software for a computation will test out some values for intervals to see roughly how much wall-time each iteration takes. Then they can set a reasonable wall-time between checkpoints using the parameter N.

Now, if interval testing seems too messy, a programmer *could* program an interrupt driven flag variable. Then they *could* use that flag to test at the top of every loop iteration whether to save a checkpoint. Then they *could* set up a repeating wall-clock timer on process zero, piggy-back the timer flag on the particle relocation data, and trigger a global checkpoint at a given wall-clock interval.... But why go to this trouble?

Besides, in that method, one would end up with checkpoint files that are not evenly spaced in terms of iterations, which would make rolling back a computation more logically complex. It is probably easier to compromise the wall-clock accuracy (which does not to be very accurate, really) for logical simplicity and code maintainability.

4.3 Enumerate I/O Options: Pros and Cons

This section addresses the problem of understanding and controlling I/O, which is a primary requirement when implementing checkpointing on a shared system.

If this is your first encounter with issues involving parallel I/O concepts, middleware like ADIOS, or Lustre file system striping concepts, the document "Using Advanced I/O on Garnet" will provide more detailed information.

you may read a short background and Garnet specific information in LINK TO PARALLEL IO DOCUMENT.

If you are using a third party tool for checkpointing, it may have configuration parameters to indicate different methods of I/O to choose from. Consult the documentation and source code to determine the pros and cons for all available I/O options.

This discussion will continue with a high-level discussion of parallel I/O options as they relate to checkpointing overheads. For more low-level implementation details see "Using Advanced I/O on Garnet".

Assume N processes coordinate a checkpoint between them. To store the data associated with this checkpoint, there are three typical options:

1. Each of N processes write a single file, storing the checkpoint as a set of N files.
   → Very fast at small to medium scales.
   → Easy to implement
   → Slows down for large scale computations as massive open file count increases file metadata traffic beyond filesystem tolerance.
   → May have to store and keep track of 1000s of files per checkpoint.
2. All N processes write to a shared file, storing the checkpoint as a single file.
   → Slow unless heavily optimized for specific filesystem.
   → Easy to implement naively. Hard to implement efficiently.
   → File locking for multiple writers causes serialization of file writes if done naively.
   → Convenient to store a single checkpoint file.
3. All N processes write to some set of M files, where $M < N$. M may be determined by system configuration, e.g. M = available Lustre OSTs.
   → Very fast at medium and large scales.
   → Moderate-to-hard to implement.
   → Overheads of processes intelligently sharing files can be noticeable at small scales.
   → File counts tend not to exceed an upper bound orders of magnitude lower than upper bound on process count, e.g. at most 240 files on a system with 150K cores.

Looking at the pros and cons of these methods, the clear trend is that simpler methods work fine until the number of processes gets very large, then more complex hardware-optimized methods win out. This is a common trend in HPC software.

The last two I/O options are well suited to parallel I/O libraries, which hide the messy details of file locking and sharing. Information about using parallel I/O libraries and middleware on Garnet can be found in "Using Advanced I/O on Garnet"

4.4 Identify ALL necessary stored data for computation continuation

What data are needed to restart the simple particle simulation example from section 4.2?

⊙ ***Global Data Set***

Clearly, the simulation is tracking particle positions and velocities, as well as other auxiliary particle data. The bulk of checkpointing I/O for this application simply requires writing out arrays of particle data from all processes.

⊙ ***Per Process Meta-data***

Often the main data set is easier to track and store than the meta-data that perform all of the bookkeeping for a computation. For example, does each process keep a control block of information about its particle set? This could include particle count, number of domain neighbors, etc, which are critical to accessing particle data.

⊙ ***Scoped Variable Meta-data***

The timestep value, M, is another value that needs to be reset. This might be stored in a loop variable that cannot be set inside the loop, meaning that any restart function must occur outside the loop. Even the values of i, used to track the checkpoint number, must be saved if you want checkpoints to continue in number order after restart.

⊙ ***Obscured Global Meta-data***

Simulation restarts often have the issue of random number generator reseeding, which results in different end solutions. If a deterministic result is always required, all random number generators must have their state saved and restored.

⊙ ***Every Little Datum We Forgot***

The process of correctly identifying all checkpoint data can be aided by developing psuedocode for a restart function. Thinking through the logic of restarting will help uncover critical data. The rest will show up as restart bugs during testing, so the more that get caught in design, the quicker the implementation will go.

4.5 Develop and test restart and checkpoint function(s)

By this point in the design, it should be fairly clear how a checkpoint will be structured.

If you have a specific software development plan in mind once you have completed your design, use that. If you would like a suggested development plan, here follows an outline.

↓ To implement:

1. Add stubs for checkpoint and restart functions at the identified logical places in your code. Use some static interval value for testing, e.g. checkpoint only once after 100 iterations.

2. Check basic read/write functionality.
   ↓ In the checkpoint function
   - Add I/O calls (using your library of choice) to write a set number of repetitions of some pattern. E.g. output "00110001001001111010000110110111 0011" one million times.

   ↓ In the restart function
   - Add I/O calls to read back everything from the checkpoint files, then verify that the correct pattern was read the expected number of times.
3. If restart on different core counts is required, perform previous step changing core counts between runs. In that case, also ensure that the global number of pattern repetitions does not change after restart. In other words, if each of N processes writes $\frac{1million}{N}$ copies of the pattern during checkpoint, then each of M processes should read $\frac{1million}{M}$ copies of the pattern during restart.
4. In small sets, add data items to both restart and checkpoint functions simultaneously.
   ↓ In the checkpoint function, for each data item, add two commands:
   - (enabled for test) Store static dummy value into file
   - (disabled for test) Store live dynamic value into file.

   ↓ In the restart function, for each data item, add two commands:
   - (enabled for test) Read value from file into dummy variable, then check against known static value used in checkpoint function.
   - (disabled for test) Read value from file into live variable used in computation

   To enable or disable code, commenting works but can be tedious. In languages like C with macro functions, a compile time if statement, e.g. `#ifdef`, can be used to enable or disable code sections based on compile time constants.
5. As above, for restart on different core counts, continue testing and developing using changing core counts between checkpoint and restart.
6. Once all dummy data values are stored and read correctly, toggle access to live values, and begin live testing.

↓ Tips:

⊙ Store a "magic number" inside checkpoint files that indicates software version. If the restart function changes later, old checkpoint files should be rejected, since the data format will be different.

⊙ If many checkpoint files are stored in a folder together, generate a human readable text file in the same folder that lists all files and other useful information about the checkpoint. Stumbling upon a folder full of binaries a year later can be confusing without some readable notes about the contents.

The methodology for live testing your restart function will depend on what constitutes correctness in the results of your computation. Deterministic applications can simple run with and without restart and compare final results to see if they are identical.

Nondeterministic applications, e.g. anything with non-restored random number generators, must evaluate final output from a restarted application for correctness in whatever manner suits the application. For example, simulation results after restart could be sanity checked in a visualization tool, if that is a normal part of your workflow.

4.6 Add checkpoint configuration controls to application initialization

Presumably, checkpoint intervals were set to some static value during testing. In addition to intervals, your checkpoint and restart functions may need other information, such as to which directory the application will save and restore.

After you have compiled a list of options, add them into your application initialization. Typically, these options will be additional command line flags or configuration file items.

4.7 Add checkpoint configuration controls to build system

This step is required for applications that want to provide external library support for some, but not all, checkpoint options. The primary example would be the use of some parallel I/O library that may not be installed on all systems.

For example, say an application implements I/O in two ways: 1) using the ADIOS library and 2) using simple POSIX commands as a fallback. The build system should be modified to determine if the ADIOS library is available, and to disable all ADIOS code in the application if it is not found. This will also require code to catch errors if a user initializes the application with an unsupported option.

As another example, say an application implements I/O using ADIOS, and includes the ADIOS package as part of the source distribution. The main build system should call the ADIOS build system first, then build the main code using the compiled library.

4.8 Document checkpoint configuration controls and build options

Update man pages, `INSTALL`s, `README`s, or other documents to reflect the new build and initialization options for checkpoint/restart.

5 External Links

DISCLAIMER: These links are not recommendations, only examples provided for your convenience. These links do not represent a complete set of available options. It is the responsibility of the reader to determine the best sources of information and software for their applications.

5.1 DoD Open Source Software Policy Information

Free software is work already done for you.

- ⊙ DoD Open Source Software (OSS) FAQ

  http://dodcio.defense.gov/OpenSourceSoftwareFAQ.aspx

- ⊙ Clarifying Guidance Regarding Open Source Software, Oct 16, 2009

  http://dodcio.defense.gov/Portals/0/Documents/FOSS/2009OSS.pdf

5.2 Parallel I/O Libraries

Let someone else worry about arbitrary filesystem details.

⊙ The Adaptable IO System (ADIOS)

http://www.olcf.ornl.gov/center-projects/adios/

5.3 Checkpointing Tools

Take great care here! The primary responsibility of the user in managing software checkpoints is being aware of I/O and storage. When using external tools, know their I/O requirements before deploying them at scale.

⊙ Distributed MultiThreaded CheckPointing

http://dmtcp.sourceforge.net/

OpenMPI, MPICH2, LGPL license

⊙ Scalable Checkpoint Restart Library

http://scalablecr.sourceforge.net/

⊙ Fault Tolerance Interface

http://sourceforge.net/projects/hpc-fti/

⊙ Template Reflection Library

http://trl.sourceforge.net/

C++ ONLY

References

[1] John T Daly. A higher order estimate of the optimum checkpoint interval for restart dumps. *Future Generation Computer Systems*, 22(3):303–312, 2006.