THE SIMULATION FACTORY: HERDING NUMERICAL SIMULATIONS

ERIK SCHNETTER

ABSTRACT. Performing large three-dimensional time-dependent simulations is a complex numerical task. Managing such simulations, often several at the same time as they execute on different supercomputers, is comparable to herding cats — supercomputers differ in their hardware configuration, available software, directory structure, queueing systems, queuing policies, and many other relevant properties.

However, these differences are only superficial, and the basic capabilities of supercomputers are very similar. We describe a set of abstractions of the tasks which are necessary to set up and successfully finish numerical simulations using the Cactus framework. These abstractions hide tedious low-level management tasks, they capture "best practices" of experienced users, and they create a log trail ensuring repeatable and well-documented scientific results. Using these abstractions, many types of potentially disastrous user errors are avoided, and different supercomputers can be used in a uniform manner.

CONTENTS

Date: 2008-10-12.

Email: mailto:schnetter@cct.lsu.edu, *Web:* http://www.cct.lsu.edu/~eschnett/.

Center for Computation & Technology, 216 Johnston Hall, Louisiana State University, Baton Rouge, LA 70803, USA, *Web:* http://www.cct.lsu.edu/.

Department of Physics & Astronomy, 202 Nicholson Hall, Louisiana State University, Baton Rouge, LA 70803, USA, *Web:* http://www.phys.lsu.edu/.

1. INTRODUCTION

Cactus: [2, 1]

Portals. Portals require a workflow abstraction, since using `qsub` et al. directly is too low-level.

State that this text is intended for numerical relativists who know Cactus and the AEI.

Motivation: workflow abstraction. middleware. best practices. initially easy to understand for users (ssh), later better technologies (certificates). seed for many other existing ideas (e.g. cleanup actions). reduce stress by avoiding mistakes. simplify things to prevent people from taking shortcuts. standardise things to make collaboration easier (e.g. common scripts for post-processing).

NR uses change code frequently.

The task of the simulation factory is to manage source trees, build executables on different machines, to submit simulations, to monitor and restart them, and to record a log.

2. CONCEPTS

The simulation factory introduces several concepts which are already implicitly present in previous workflows:

Source Tree: Conventionally, people have multiple copies of the source tree on different machines. These source trees are individually checked out or updated from a repository (e.g. CVS), or local changes are manually copied between machines. On each machine there are typically several copies containing different versions of the code, e.g. one used in production, another with a new feature being tested. The simulation factory's notion of a *source tree* is machine independent. This separates the concerns of source code version and the machine used to store them. While using multiple source

tree versions for code development is necessary, it is necessary to be able to speak of the same version as stored on different machines.

In the simulation factory workflow, a source tree version is edited on one machine (presumably locally) and then quickly replicated onto other machines. This avoids unintentional differences between versions stored on different machines, and avoids also having to set up other machines with credentials to obtain the source code from a repository.

Configuration: Source tree versions can be build with various option settings. The option settings depend on the machine on which the resulting executable should run, and also depend on a user's choice, e.g. whether the executable will be used for debugging, for profiling, or for production runs. A *configuration* captures the combination of a particular source tree version and a particular set of build options. (This is the same terminology as used in Cactus.) A source tree can contain arbitrarily many configurations.

Simulation: An executable from a configuration can be run with different parameter files. The parameters specify specify physics of a simulation as well as computational details, e.g. how many output files to produce. A *simulation* captures the combination of a (snapshot of a) configuration and a specific parameter file. As such, it describes a certain simulation completely and therefore determines uniquely its result. However, it does not specify how the simulation is to be submitted to a queuing system. A simulation thus specifies the *what*, but not the *how*.

Restart: Contemporary machines have queueing systems which require jobs to wait before they start, to be restarted multiple times due to run time limitations, and to be restarted after hardware failures. A simulation contains arbitrarily many *restarts*, where each restart either begins from scratch or from checkpoint files left by a previous restart. Each restart is essentially identical to a PBS job; it is *active* while it is queued or running, and *finished* afterwards. Only one restart can be active per simulation.

Machine database: Since all machines are different, and since each system administrator seems to enjoy improving the local installation, thereby making it different from any other machine, it is necessary to store small "recipes" for certain actions, such as e.g. commands for logging in, copying files, submitting jobs, etc. These are stored in a *machine database*, which is distributed together with the simulation factory. (The machine database is stored as Perl associative arrays in keyword/value pairs which can be easily corrected and extended.) A *user database* extends this setup, allowing per-user settings to override the "official" settings in the machine database.

Self sufficiency: Each source tree, configuration, simulation, and restart is independent of all others, and contains copies of all relevant information (option files, parameter files, executables, etc.) to make it self-sufficient. That means that e.g. updating a configuration does not affect simulations that are using this configuration — it only affects new simulations that are created later. This ensures that source trees, configurations, simulations, and restarts remain self-consistent and can continue to be used without being affected by accidental changes to other parts of the setup.

One major difference between this terminology and the implicit terminology that an HPC user would use is that there is a strict distinction between a *source tree version* and the machine which is used to store it, so that the same source tree version can be replicated on different machines. The other major difference is that the notion of a *simulation* has been introduced, where a simulation defines already, implicitly, the simulation result, but does not specify the sequence of job submissions which are required to actually calculate the result.

The machine database provides a unified user interface to machines with different setups. Although many of the differences between machines are only small, only a completely uniform interface allows automating tasks.

Based on the above concepts, the simulation factory offers commands for the most common actions, i.e., to replicate a source tree between machines, build a configuration, or manage simulations in several ways. Using these commands has several advantages over managing configurations and simulations directly:

- The simulation factory provides a uniform interface to different machines. (This is an "interface" is in the sense of shell commands, not a graphical user interface.)
- The simulation factory leaves automatically a log trail of all actions applied to a simulation, so that past actions can be reviewed in case of problems.
- The simulation factory has a certain amount of error checking built in which prevents common errors, such as e.g. submitting two jobs writing into the same output directory.
- The simulation factory enforces certain best practices which were collected by experienced users, which further reduce the chances of user errors. For example, each restart uses a different output directory, which avoids potential problems when restarting a simulation after an unclean shutdown.
- The simulation factory implicitly enforces a certain regular structure onto the output of simulations, which makes it easier to post-process simulation results automatically or semi-automatically.
- The simulation factory can make use of highly efficient command sequences which are either unknown to common users or are too complicated to be used manually, e.g. avoiding copying large files, executing compute intensive commands in parallel, or removing temporary files automatically.

The simulation factory is not meant to be a black box. Many details can go wrong when building executables or submitting jobs to a queue, and these may still require human intervention. At the same time, while the simulation factory offers convenient ways to check on the status or result of a simulation, it is still necessary to examine the simulation and output directories manually to obtain the full picture. Future versions of the simulation factory may improve this with user feedback.

It would be possible to provide a high-level graphical user interface built on the abstraction offered by the simulation factory. While it is not intended to make direct user contact with the machines unnecessary, many maintenance tasks — such as listing the current state of submitted or ongoing simulations — could be simplified even for the expert user.

3. SIMULATION FACTORY OVERVIEW

The simulation factory is distributed as a directory called `simfactory` containing a Perl script called `sim`. The `simfactory` directory should be copied into a convenient place, preferably within a Cactus source tree. Similar to Cactus, should not be installed by the system administrator. It is necessary to set up a user database (contained in the `simfactory` directory) before using the simulation factory; this is described in section 12 below.

This Perl script can be called with different subcommands, similar to the way `cvs` or `svn` is used. One particular command is

```
simfactory/sim help
```

which lists all available commands and options.

All simulation factory commands can be executed on a remote machine without manually logging in there. The feature is described in section 4.4.

4. SOURCE TREE MANAGEMENT

The simulation factory supports multiple source trees. Each source tree must be stored as a subdirectory of a certain *source base directory*. One way to manage source trees is to edit them only on one machine, e.g. a notebook or a local workstation, and to replicate the source trees to all other machines after each change. Replication, which uses `rsync`, takes typically only a few minutes.

4.1. **Multiple source trees.**

Here is the directory structure of a possible setup of several source tree versions on a notebook:

```
/home/eschnett/Calpha
/home/eschnett/Cbeta
/home/eschnett/Cvanilla
```

There are three source trees, named `Calpha`, `Cbeta`, and `Cvanilla`, respectively, in the source base directory `/home/eschnett`. Each of these three source trees is a full Cactus source tree, in this case renamed from `Cactus` to something "descriptive". Source tree names can be arbitrary, but they must be located in the same source base directory. There can be other files and directories in the source base directory; one can e.g. use one's home directory as source base directory if there is sufficient disk space available.

4.2. **Replicating source trees.**

The simulation factory replicates source trees by invoking a customised `rsync` command. The simulation factory command to replicate a source tree on a different machine is:

```
cd /home/eschnett/Calpha
simfactory/sim sync peyote
```

This replicates the source tree `Calpha` from the local machine to Peyote (a Linux cluster at the AEI). It assumes that the simulation factory is installed as the directory `simfactory` in the `Calpha` source tree.

The machine database contains the necessary options for invoking a working `rsync` command on both the local machine and on Peyote. The locations of the source base directory on both the local and remote machine are also stored in the machine database.

The simulation factory performs the following internal steps when executing this command:

(1) Determine local machine name
(2) Look up local machine in machine database
(3) Look up remote machine (`peyote`) in machine database
(4) Compose `rsync` command from both machine descriptions
(5) Output `rsync` command to screen as debugging aid
(6) Invoke `rsync` command

The simulation factory cannot replicate a source tree if the local machine is not described in the machine database.

It is convenient to store the simulation factory source code in a Cactus source tree. In this way, replicating the source tree onto a different machine automatically replicates the simulation factory as well. Replicating a source tree does not require a simulation factory to be installed on the remote machine. This avoids having to install the simulation factory on other machines, and guarantees that the user databases are identical on all machines.

Since the simulation factory accesses remote machines, it needs to be able to log in there. This is most convenient if `ssh` has been set up to allow access without having to type passwords more than once. `ssh` keys with empty passwords are dangerous. A safer alternative is a keychain, which is described in section 13.

Some machines are not accessible from the outside. In this case, the machine database needs to describe how a "trampoline machine" is used to set up an `ssh` chain. (For details see section 12.)

4.3. **Other information stored in source trees.** It is possible to store additional files in source trees. These files will also be copied to the remote machines, making this a convenient way to distribute information. For example, one can store parameter files or post-processing scripts in the source tree.

The simulation factory knows a set of additional subdirectories in the main Cactus directory which are replicated. These subdirectories include

```
AEIPhysics
simfactory
bin
carpet
parfiles
```

4.4. **Remote execution.** All simulation factory commands can be executed transparently on remote machines, i.e., without explicitly logging into the remote machine. This is done by using a `remote` prefix to the simulation factory command, as in

```
simfactory/sim remote peyote help
```

to execute the `help` subcommand on the remote machine Peyote (which in and itself is not very useful).

The machine database and the user database describe the `ssh` command and its options which are used to log into the remote machine, and the user name which is used for logging in.

If there are two remote machines located close together, both far from the local machine, then its makes sense to copy a source tree from one remote machine to another. This can be faster than copying it from a local machine. For example:

```
simfactory/sim sync peyote
simfactory/sim remote peyote sync belladonna
```

This copies first the local source tree to Peyote, and then from Peyote to Belladonna. After this, both Belladonna and Peyote contain a copy of the local source tree.

4.5. **Listing all known machines.** The simulation factory command

```
simfactory/sim list-machines
```

lists all machines in the machine database. command `list-machines` can be abbreviated as `list-mach`.

5. CONFIGURATION MANAGEMENT

Each source tree can contain several configurations. As described in section 2 above, a configuration combines a set of thorns and compiler options, defining an executable. This is exactly same concept as in Cactus; simulation factory's contribution is to store thorn lists, option lists suitable for particular machines, and to make the build process more convenient. It also stores script files (e.g. for PBS) which can be used to run a particular executable on the machine for which it was built.

5.1. **Option lists.** Coming up with a good option list for a particular machine requires experience, both with the machine's operating system and with the Cactus thorns which may require external libraries. It is customary that a power user creates such an option list, installing necessary external libraries as necessary, and that most users then use these option lists in a black box fashion. This goes nicely with simulation factory's approach of storing option lists within itself.

Option lists should be located in the simulation factory subdirectory `optionlists`. Their name should specify the machine for which it is designed. Some example option lists are

```
abe-mvapich2
eric-mvapich2
ranger-mvapich
```

If there are option lists e.g. for using different MPI implementations on the same machine, then the option lists' names should be distinct.

Option lists are templates are templates which can contain variables which are replaced when a simulation is created (see section 6.2 below). Three particular variables are `DEBUG`, `OPTIMISE`, and `PROFILE` which can be set either to `yes` or `no` when creating a configuration.

5.2. **Thorn lists.** Cactus supports building superfluous thorns into a configuration, allowing to choose at run time which thorns should be activated. This makes it possible to create standard configurations which can be used with many, very different parameter files. Since not all machines have all external libraries available (e.g. `PETSc` may not be installed), not all thorns can be built on all machines. That means that, while there is a "standard" set of thorns which are useful for a particular research direction, each machine can build only a subset of these.

Thorn lists should be located in the simulation factory subdirectory `thornlists`. Their name should (but does not have to) consist of at least two parts, identifying the purpose of the thorn list and the machine for which it is designed.

Thorn lists should be independent of the fact whether a configuration is a debug or an optimised build. Some example option lists are

```
einstein-abe
einstein-eric
einstein-ranger
```

If there are thorn lists e.g. for using different MPI implementations on the same machine, then the thorn lists' names should be distinct, since the set of supported external libraries and hence the set of thorns which can be built may be different.

5.3. **Script files.** In order to run an executable on a certain machine, it is usually necessary to submit it as job to a batch system (e.g. PBS). This requires writing a *script file* which will call the executable and perform certain additional tasks, such as creating a scratch directory or redirecting output to certain files. Even when no batch system is used, script files are useful since they control how the executable is run, ensuring they are called correctly and in a consistent manner. Script files contain commands which are tailored for a certain machine, and can also depend on the compilers or libraries that were used to build the executable.

Script files should be located in the simulation factory subdirectory `scriptfiles`. Their name should (but does not have to) identify the machine for which it is valid. Script files should be independent of the fact whether a configuration is a debug or an optimised build and should be independent of the thorn list. Some example script files are

```
abe-mvapich2
eric-mvapich2
ranger-mvapich
```

If there are script files e.g. for different MPI implementations on the same machine, then the script files' names should be distinct, since different MPI implementations typically require different commands to run an executable.

5.4. **Building configurations.** In order to build a configuration, one has to specify three items: the name of the configuration, and option list, and a thorn list. The command to build a configuration is

```
simfactory/sim remote ranger build einstein-ranger-mvapich-debug
\
  --optionlist=ranger-mvapich \
  --thornlist=einstein-ranger \
  --scriptfile=ranger-mvapich \
  --debug
```

This command creates a new configuration `einstein-ranger-mvapich-debug` (if it does not already exist), configures it with the option list `ranger-mvapich` (which has incidentally a name similar to the configuration), setting the variable `DEBUG` to `yes` in the option list, and then builds and executable using the thorn list `einstein-ranger`. The executable will later be run using the script file `ranger-mvapich`.

The simulation factory remembers the option list, thorn list, and script file that was used to create configuration. If the configuration needs to be recompiled, these options can be omitted. If either the option list or the thorn list should be changed, one or both can be given as options. The simulation factory will then

check whether they actually changed (comparing them character-wise), and if so, will re-configure or re-build the configuration.

It is possible to build multiple configurations at the same time, as in

```
simfactory/sim remote ranger build \
  einstein-ranger-mvapich-debug einstein-ranger-mvapich-optimise
```

If additional options are given in this case, they apply to all configurations.

5.5. **Other commands.** The simulation factory command

```
simfactory/sim remote ranger list-configurations
```

lists all configurations which exist for a particular source tree, together with the date at which the executable was last recompiled. The long command `list-configuration` can be abbreviated as `list-conf`. The output could look like this:

```
Configurations:
   einstein-ranger-mvapich-debug        [built 2007-08-05 23:56:13]
   einstein-ranger-mvapich-optimise     [built 2007-08-05 23:55:26]
   einstein-ranger-mvapich-profile      [built 2007-05-17  1:16:51]
```

5.6. **Directory structure.** A configuration is located in the directory

source base directory`/`*source tree name*`/configs/`*configuration name*

which is the standard location for Cactus configurations.

It contains, among many others, the following files:

```
OptionList
ScriptFile
ThornList
```

These are verbatim copies of the files given in the corresponding options when building a configuration. The file `ThornList` is already placed there by Cactus, the files `OptionList` and `ScriptFile` are added by the simulation factory.

5.7. **Thorn Formaline.** The Cactus thorn `AEIThorns/Formaline`[1] should be present in the thorn list. If it is part of a configuration, Cactus will take a snapshot of the source tree and the configuration options, and will include it into the executable. At run time, a parameter can be set to recreate the source tree which was used for a particular executable, making it impossible to lose the source code for an executable.

Formaline will also create a unique *build ID* each time the code is recompiled. This can be used to identify a configuration and source code version, which is useful to cross-reference simulation results which may have been produced by different source code versions.

Together with Formaline, the simulation factory will also generate unique *simulation IDs* and *restart IDs* to identify what restart of what simulation produced a particular result. This is discussed in section 11 below.

[1] "Formaline" is an unfortunate misspelling of the chemical "Formalin" which is used to preserve biological species.

6. SIMULATION MANAGEMENT

A simulation is essentially a combination of an executable and a parameter file. Together they specify the result of the simulation, but they do not specify which queue and how many processors to use. After a simulation has been created, it can be submitted to a queue, restarted from a checkpoint after it aborts or finishes, removed from a queue, etc. This is described in section 7 below.

In order to create a simulation, one has to specify three items: the name of the simulation, the name of the configuration which supplies the executable, and a parameter file. The command to create a simulation is

```
simfactory/sim remote ranger create qc0-reference-0000 \
  --configuration=einstein-ranger-mvapich-optimise \
  --parfile=parfiles/einstein/qc0-reference.par
```

The name of a simulation can be chosen arbitrarily. It is a good practice to name a simulation after the parameter file which is used. Since one often wants to compare similar parameter files or source code versions, it can be convenient to append a version number or similar modifier to the simulation name, as was done with the suffix `0000` in the example above.

The configuration must have been previously built, and an executable must have been created. The parameter file must exist on the remote machine, and its file name is interpreted relative to the directory containing the source tree (i.e., the main Cactus directory). In the example above, a subdirectory `parfiles` was created in the main Cactus directory which contains a set of parameter files. This means that this parameter file directory is automatically replicated on remote machine whenever the source tree is replicated.

After a simulation has been created, neither the executable nor the parameter file can be changed. (It is possible to change them manually by either editing or exchanging the corresponding files in the simulation directory.)

As described in section 2 above, a simulation is not automatically submitted to a queueing system. A separate `submit` command is required for that, as described in section 7 below.

6.1. Directory layout.

All simulations are stored as subdirectories of a *base directory* (which is different from the *source base directory*). While the source base directory is typically located on a home partition or a similar location which contains more valuable data, the base directory is often on a data disk, or scratch partition where hopefully files are not automatically deleted.

A simulation is located in the directory

base directory`/`*simulation name*

and has the following directory structure:

`exe/`
`par/`
`run/`
`SIMULATION_ID`
`LOG`
`output-`*NNNN*`[-active]/`

The directories `exe`, `par`, and `run` will contain copies of the executable, the parameter file, and the script file, respectively. Making copies ensures that the simulation is self-contained: any changes to the configuration or the original parameter file will not accidentally influence the simulation result.

The simulation factory uses a caching mechanism for the executables since making a copy of the executable can be expensive, especially if many simulations are created at the same time. The cache, which is not supposed to be modified manually, contains a copy of the executable. The executable in the `exe` directory is a hard link to the cache, which uses almost no disk space. The cache is updated automatically if necessary, in a way which does not influence existing simulations.

The parameter file and the script file are templates which can contain variables which are replaced when a simulation is started (see section 6.2 below).

A simulation can contain several *restart directories*. Active restarts have the suffix `-active`. At most one restart per simulation can be active (see section 7 below).

6.2. **Templates.** Option lists, parameter files, and script files are templates, that is, they can contain variables which are automatically replaced when a simulation is created or a job is submitted or restarted. Variables need to be written like this:

```
@VARIABLE@
```

to make it immediately clear whether something is a variable or not. Table 1 lists all variables which are defined automatically.

Arbitrary additional variables can be defined with the `--define` option using the syntax

```
simfactory/sim remote ranger create qc0-reference-0000 \
  --configuration=einstein-ranger-mvapich-optimise \
  --parfile=parfiles/einstein/qc0-reference.par \
  --define USER_VARIABLE=SomeValue
```

Variables need not be all upper case.

6.3. **Listing all simulations.** The simulation factory command

```
simfactory/sim remote peyote list-simulations
```

lists all simulations which exist on a particular machine. The long command `list-simulations` can be abbreviated as `list-sim`.

The output could look like this:

```
Simulations:
   qc0-reference-0000          [ACTIVE,   restart id "0000", job id "296"]
   qc0-reference-0016          [INACTIVE, was never active]
   qc0-reference-0017          [INACTIVE, was never active]
   qc0-reference-0018          [INACTIVE, was never active]
   sep_07.00_59a-lev5-h2.00-0001   [INACTIVE, restart id "0009"]
   sep_07.00_59a-lev5-h2.56-0001   [INACTIVE, restart id "0004"]
   sep_07.00_59a-lev5-h2.84-0001   [INACTIVE, restart id "0005"]
```

When the option `--verbose` is used, each simulation is examined in more detail, leading to output like

```
Simulations:

   qc0-reference-0000: ACTIVE (queued or running)
      active restart id: "0000"
```

Variable	Type	Example
DEBUG	string	no
OPTIMISE	string	no
PROFILE	string	no
SIMULATION_NAME	string	rnsid
SHORT_SIMULATION_NAME	string	rnsid-0002 (suitable as job name in PBS scripts)
SIMULATION_ID	string	simulation-rnsid-redshift-eschnett-2007.03.12-23.16.56-6359
RESTART_ID	string	0002
HOSTNAME	string	numrel02.cct.lsu.edu
RUNDIR	string	/home/eschnett/runs/rnsid/output-0002-active
SCRIPTFILE	string	rnsid.qsub
EXECUTABLE	string	./cactus_rnsid
PARFILE	string	rnsid.par
USER	string	eschnett
NODES	int	4
PPN	int	2
PROCS	int	8
QUEUE	string	workq
WALLTIME	string	4:00:00
WALLTIME_HH	int	4
WALLTIME_MM	int	0
WALLTIME_SS	int	0
WALLTIME_SECONDS	int	14400
WALLTIME_MINUTES	int	240
WALLTIME_HOURS	real	4.0
EXECHOST	string	ic0042

TABLE 1. Automatically defined variables. Additional variables can be defined with the option `--define`. Variables are replaced in option lists, script files, and parameter files.

```
      job id:            "296"
      job state:         running
      processors:        32
      wall time limit:   48:00:00
      disk usage:        0.1 GByte
      configuration:     einstein-peyote-mpich-optimise
      scriptfile:        ScriptFile
      parfile:           qc0-reference.par

Simulations: 1
   active:   1 (0 queued, 1 running, 0 pending)
   inactive: 0
Total disk usage: 0.1 GByte
```

7. RESTART MANAGEMENT

As described in section 2 above, the simulation factory makes a distinction between simulations and restarts. A simulation describes the source code and parameter file which is used to produce a particular result. A simulation is typically submitted to a queueing system multiple times, using checkpointing and recovery. Each such submission is called a restart. A simulation can contain many restarts, and after being created it starts out with zero restarts.

A new restart is said to be *active* while it is being processed by the queueing system. Old restarts are called *inactive*. A simulation is also either *active* or *inactive*, depending on whether it contains an active restart. These distinctions are made to determine what kinds of commands are allowed at a given time. For example, an active simulation cannot be submitted to a queueing system.

A restart undergoes the following states after being created:

```
active/queued
active/running
active/running [stopped]
active/finished
inactive
```

It starts out being queued, which means that it has been submitted to a queueing system but is not yet running. (Note that it is nevertheless already called active.) After running, or after being stopped, it ends up being finished (but still active). The restart needs to be explicitly *cleaned up* in order to become inactive. This life cycle of a restart is also described in figure 1.

7.1. Commands.

A simulation can either be *submitted*, i.e., started from scratch, or *restarted*, i.e., started from an existing checkpoint file which was written by an earlier restart. Both commands take the same arguments. When restarting, it is also possible to specify what checkpoint file to use:

```
simfactory/sim remote ranger submit qc0-reference-0000 \
  --procs=64 --walltime=24:00:00

simfactory/sim restart ranger submit qc0-reference-0000 \
  --procs=64 --walltime=24:00:00
```

The number of processors must be specified. The wall time is optional; if not specified, it is assumed to be the maximum possible wall time for the machine.

An additional option `--from-restart-id=2` can be used to specify that a simulation should be restarted from a particular checkpoint instead of the latest one.

7.2. Graceful termination.

When a running simulation is stopped by killing one of its processes or using PBS' `qdel` command, then the simulation aborts immediately. This can leave output files in inconsistent states, and it loses all progress made since the last checkpoint.

There are two mechanisms in Cactus which allow a graceful termination. The web server interface (thorn `CactusConnect/HTTPD` or thorn `AstroGrid/HTTPS`) can be used to terminate a simulation after the current iteration, guaranteeing that all output files are left in a consistent state. In addition, a termination checkpoint can be requested. This delays termination by a few minutes, but saves all current progress.

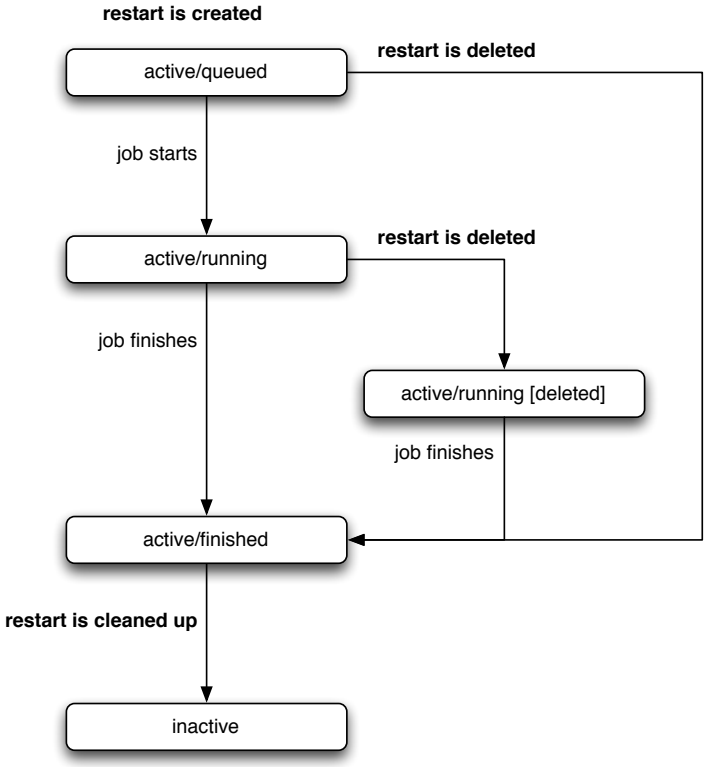

FIGURE 1. Life cycle of a restart, listing the possible restart states and their transitions. A new restart starts out as `active/queued`, and ends up as `inactive` after it has been cleaned up. The **bold** state transitions (create, stop, cleanup) have to be initiated by explicit user commands.

In a similar manner, the thorns `CCCCThorns/TriggerTerminationManual` or `AEIThorns/ManualTermination` can be used to request termination when a certain code is written to a certain file. The simulation factory knows about these thorns. If one of these thorns is active and has created an empty termination file, the simulation factory will terminate the simulation by writing the corresponding code into that file.

7.3. **Cleaning up.** After a job is finished, it needs to be cleaned up before the simulation can be resubmitted again. Ideally this cleanup should happen automatically, but this is not yet implemented, and thus has to be triggered explicitly. Typical cleanup actions are

- Remove half-written (useless) checkpoint files
- Remove temporary files
- Correct file permissions

Other actions are possible, e.g. contacting an external information service and transmitting some simulation details, or starting some automatic analysis tools.

7.4. **Restarting.** Each restart has its own output directory. This guarantees that different restarts cannot inadvertently overwrite earlier results. It also makes it possible to delete or ignore data from a bad restart.

In order to simplify parameter files, the checkpoint files from which a job restarts are made available in the job's output directory. In order to save disk space, the checkpoint files are not copied but are only hardlinked. This makes it possible to delete or archive restarts while keeping the checkpoint files which were used to start other jobs.

7.5. **Showing job output.** The command

```
simfactory/sim restart ranger show-output qc0-reference-0000
```

shows Formaline output, standard output, and standard error of the current restart of a simulation. If the restart is currently running, the output is accessed wherever the queuing system places it, if necessary.

8. MISCELLANEOUS COMMANDS

comment

9. USER SETUP

whatever is necessary before using the simulation factory

10. TUTORIAL

■ [Guide user through a sample session for the Cactus WaveToy]

Checking out Cactus source tree and simulation factory:

(1) Check out Cactus tree, as described in Cactus documentation
(2) Check out simulation factory into the Cactus source tree
(3) Create simulation directory: `mkdir $HOME/simulations`

Using the simulation factory:

```
simfactory/sim help

simfactory/sim build wavetoy-debug \
  --optionlist=ranger-mvapich --thornlist=wavetoy
  --scriptfile=ranger-mvapich --debug

simfactory/sim list-configurations

simfactory/sim create wavetoyc_rad \
  --configuration=wavetoy-debug \
  --parfile=arrangements/CactusWave/WaveToyC/par/wavetoyc_rad.par

simfactory/sim submit wavetoyc_rad --procs=4

simfactory/sim list-simulations

simfactory/sim cleanup wavetoyc_rad
```

Looking at results:

(1) `cd $HOME/simulations/wavetoyc_rad`

11. TODO

create, submit. Physics vs. job-sitting.
active, inactive. Independent of queued and running.
finished. Deleting is asynchronous. Cleanup.
simulation, job. restart.
directories for simulations. subdirectories for restarts.
hard links for checkpoint files.
subdirectories exe, par, run.
small files with human-readable information, both for simulations and restarts.
ManualTermination. TERMINATE.
copy executables. cache.
copy executables, parameter files, script files. keep everything around.
parameter files and script files are templates.
option and thorn lists, script and parameter files can be stored in simulation factory or outside.
simulation id, restart id, job id. point to formaline.

machine:
configuration:
simulation:
restart:
job:

12. MACHINE AND USER DATABASES

13. PRELIMINARIES

ssh access: public, private key. password on private key. keychain. ssh agent.
remote login. nothing on stdout for "ssh command". set up full path for remote execution. maybe source profile from bashrc, with guard.
profile:

```
export BASH_PROFILE_READ=1
```

bashrc:

```
if test -z "$BASH_PROFILE_READ"; then
    source /etc/profile
    source .bash_profile
fi
```

local .ssh/config:

```
ControlMaster           auto
ControlPath             ~/.ssh/master-%r@%h:%p
```

doesn't work.
agent forwarding:

```
ForwardAgent yes
```

do this on local machine and on trampolines. this option seems to be dangerous, at least if you do not trust root on the corresponding machine.
closing the control master closes all piggyback connections as well! make sure you don't use a simfactory connection as control master.
list various ssh/rsync errors and their causes and remedies.
mention rate limiters for ssh.

install simfactory in a Cactus directory. rsync will then copy it over. if outside a Cactus directory, the remote path will be wrong. (this could be corrected.)

local hostname. need mdb entry for laptop. `$HOME/.hostname`.

udb to overwrite mdb entries, especially user names.

ACKNOWLEDGEMENTS

Portal project for vision. AEI for hospitality.

REFERENCES

[1] Cactus Computational Toolkit home page, URL http://www.cactuscode.org/.

[2] T. Goodale, G. Allen, G. Lanfermann, J. Massó, T. Radke, E. Seidel, and J. Shalf, *The Cactus framework and toolkit: Design and applications*, Vector and Parallel Processing – VECPAR'2002, 5th International Conference, Lecture Notes in Computer Science (Berlin), Springer, 2003.