

Common Runtime Environment Definition

Efstratios Efstathiadis - BNL
Don Holmgren - FNAL
Bálint Joó - JLAB
James Simone - FNAL
Amitoj Singh - FNAL
Chip Watson - JLAB

June 14, 2006

1 Introduction

1.1 Design and Implementation Philosophy

The primary goal of the CRE is to facilitate the usage of the QCDOC at BNL and clusters at JLab and FNAL by physicists, as opposed to developers. The building and running of application codes should be as portable as possible. To compile and link applications, scientists should have access to build systems which can be used with minimal modification at the three sites. To run applications, users should have access to batch commands and run script examples which can also be used with minimal modification. The CRE should provide the definitions, environment variables, utilities, file system layouts, naming conventions, and tool chains that allow developers to implement and test such scripts and build systems. The CRE should be sufficiently well documented so that interested physicists can understand and modify scripts and makefiles. The CRE should also provide the tools to find, access, and transport (archive, restore, move) the data products necessary for production.

1.2 Document Scope

This document defines the Common Runtime Environment to be deployed across the USQCD Metafacility. The document defines a model of the parallel systems of the metafacility which will need to be mapped onto the concrete systems at each site. The mapping can vary from site to site. The abstract model defines the following features

- A model filesystem
- A model of job execution

In order to accommodate implementation on the diverse computing resources the model defines an abstract view of the filesystems. These can be mapped onto real filesystems through environment variables. In order to facilitate running of jobs uniformly we define several scripts which can wrap the actual running commands. Finally we define a baseline of libraries that are needed to build the applications.

In order to run on a particular system the mapping between abstract filesystems and libraries needs to be made concrete. This can be done by way of a setup script that needs to be called or sourced, which will set up the configuration via environment variables and symbolic links.

1.3 What we do not define

There are a large number of ways to compile up SciDAC application code suites such as `chroma`, `CPS`, `FermiQCD` and `MILC`, even for a particular parallel architecture such as the gigabit Ethernet mesh, or the `QCDOC`. Therefore we do not mandate the installation of applications. Our baseline will be comprised of support libraries to allow the compilation of these packages to be carried out in a uniform and straightforward manner.

Likewise some matters are matters of site policy, and it is inappropriate to mandate them, however we can indicate our intent and leave it up to sites whether to and how to implement them.

1.4 Scope of Implementation

There is nothing to stop a particular site from installing more packages or to provide more features than what we define here. We merely provide a minimum so that

- Users can define batch scripts that can be reused between systems with minimal modifications
- Users can define scripts to compile code on a system and expect the code to compile on other systems with minimal modification.¹

1.5 Relation to USQCD SciDAC Common Runtime Environment

Our definition here is a subset of the SciDAC Runtime Environment. We do not implement that in full because large portions of it remain in development.

1.6 Portability

Our target deployment consists of the three USQCD Metafacility sites: at BNL, FNAL and JLab. We have designed a system that may be easy to port to other platform, at least the part that is governed through environment variables. Some issues are more difficult to make portable to non-target systems: eg installed software, compiler toolchains etc, which may well be under local system staff control.

1.7 Document Structure

This document is structured as follows. In section 2 we give an overview of the high level model of our systems and discuss how this can map onto concrete systems within the metafacility. In section 3 we outline concrete details of the scripts and the environment variables and how the environment may be tested. In section 4 we make some recommendations to implementers about some policy aspects that we have envisaged and recommend, but do not mandate and cannot enforce. In 5 we give some recommendations to users as to what they can or cannot expect from the system during use. We summarize in section 6

¹Different builds on different systems may need different modules so building cannot be completely standardized

2 Model Computer

We envisage our model of the computational system to have the following distinct components.

- Standardized filesystem defined through environment variables
- Interactive node (IN) for application compilation with a baseline set of libraries needed for compilation
- A notional Batch Script Execution Node (BSEN) on which batch scripts will execute.
- Support for file management on this system through scripts
- A notion of Compute Nodes (CN) on which parallel jobs run.
- The notion of a Master I/O (MIN) for singlefile reading and writing.
- Support for launching jobs uniformly on the CN through scripts

2.1 Model Filesystem

In the case of filesystems we will specify the names of the filesystems through UNIX shell environment variables for now to indicate the notions that they can be mapped to a site's local setup.

- A *home directory* (\$HOME) for the users, mounted on at least the IN and the BSEN
- A *large filesystem* (\$QCACHE) for storing computation results over a long time. This node potentially needs to be mounted only on the IN. The implementation will split this into two separate environment variables \$QCACHE_PROJECT and \$QCACHE_USER, one pointing to project specific storage, and one pointing to user specific storage. However, for the remainder of this document we will simply refer to \$QCACHE to refer to both these storage areas collectively.
- A *filesystem for staging data for a parallel job* (\$QDATA). This node potentially needs to be mounted to the IN, BSEN and the MIN.
- A filesystem or filesystems that is/are accessible to individual I/O capable compute nodes (\$QSCRATCH). This/These filesystems need to be mounted only on the CN.

The setup above is fairly general and can be mapped onto clusters at the Jefferson lab and Fermilab, where the BSEN is expected to be one of the nodes allocated by the scheduler to be the *head node* of the parallel job. At the Jefferson lab we have the following filesystem mappings:

\$HOME - /home/username

\$QCACHE_PROJECT - /cache/<project name>

\$QCACHE_USER - /cache/users/<user name>

\$QDATA - /scratch on the MIN.

\$QSCRATCH - /scratch

IN - qcdi02, qcdi03

BSEN - Nominated nodes of the 3G and 4G GigE clusters, potentially any infiniband node.

CN - The compute nodes of the 3G, 4G and 6N clusters.

MIN - The head node of the PBS job, same as BSEN.

In addition all compute nodes are I/O capable and \$HOME is mounted to every node.

On a QCDOC we may have the following mappings

\$HOME - /home/username

\$QCACHE_PROJECT - /cache/<project name>

\$QCACHE_USER - /cache/<username>

\$QDATA - /host/<username>

\$QSCRATCH - /pfs/<username>

IN - qcdocfe

BSEN - qcdocfe

CN - QCDOC Compute Nodes

MIN - Node 0 of a partition

In addition every node is I/O capable and mounts \$QDATA and \$QSCRATCH.

2.2 File Management

Since \$QDATA and \$QSCRATCH may well be different filesystems we envisage the following operations will become necessary in time

qcp - Copy between \$QCACHE and \$QDATA, \$QCACHE and \$QSCRATCH and \$QDATA and \$QSCRATCH

qsplit - Split a single file into several files, one for each distinct \$QSCRATCH system (every I/O capable node). We restrict functionality of this component to QIO compliant files. This tool is only be concerned with the splitting of the file, rather than the distribution of the various node specific files to their machine specific locations. It is expected that this utility will be implemented through QIO.

qunsplit - The inverse of qsplit - we assume that the individual files written by a job have been collected into one place, and then the qsplit program will re-assemble them into a single file. It is expected that this will be implemented through QIO.

qbcast - Broadcast the copy of a file to all the \$QSCRATCH filesystems to the same path in that filesystem.

qscatter - Take a file that has been split with qsplit or otherwise and scatter it to the \$QSCRATCH filesystem on the various nodes.

qcollect - Collect files from a particular place within the \$QSCRATCH directories into one local directory so they can qunsplit

These file management operations especially the qsplit, qunsplit, qscatter and qgather involve some subtleties. Which we should mention up front as they influence choices in our definitions later in the next section.

Layout subtleties: - We need to know which file needs to go to which node. This information needs to be set up possibly outside the application (ie if pre-staging). On some architectures this is not really an issue since a partition's physical node layout is fixed and known, to the tune of being able to identify up-front which physical node will have which QMP ID. This is the case on the Gig-E machine partitions at the Jlab and the QCDOC after booting and axis remap. On the other hand, MPI based targets can present difficulties as one would need to set up a map from MPI IDs to physical nodes, and then also to ensure the mapping of MPI IDs to QMP IDs is predictable. This aspect requires *further investigation*.

Multi Node/Core subtleties: - On a multi-node/multi-core system it is entirely possible (likely/certain) that more than one QMP process will have shared access to the same `$QSCRATCH` filesystem. Hence QMP process specific files, *need to be annotated* with the *QMP process ID*. QIO does this automatically and so we restrict *qsplit* and *qunsplit* to QIO files. Likewise we will restrict *qgather* and *qscatter* of files that follow the QIO filename conventions.

In addition to the above command which are to be used within batch script we require that the interactive nodes be accessible via SSH. We will give concrete details of this later.

2.3 Launching Parallel Jobs - Resource Allocation

There are two major parts to running jobs. The first is *resource allocation* by which a set of Compute Nodes is acquired. On clusters and commodity parallel computers this is typically done by submitting jobs to a *queuing system* of some sort. On custom hardware typically a fixed amount of resource may be allocated permanently to projects which then have that resource *on demand*. To complicate the situation posed by these two opposing models, not all queuing systems are compatible. For example *PBS* and *LSF* use commands such as `qsub` and `qdel` whereas the *LoadLeveler* system from IBM prefers `lsubmit` and `llcancel` to do the same jobs. While Grid projects are attempting to alleviate this problem by coming up with queuing interfaces that unify these systems, this aspect of operation is not going to be standardized in this definition. It is a moving target and may be standardized in the future. Rather in this document we will attempt to define commonality through the batch scripts that are run. This way we factor out the different startup sequences.

2.4 Launching Parallel Jobs – Batch Script Environment

In addition to having access to the environment variables above, and to the file management commands the following features are needed to execute the parallel job

`qsetGeometry` - This command will provide a suitable mapping from the processor nodes to a virtual processor topology required by the job. This geometry must be propagated to QMP. An additional role for this command is to set up layout specifications to enable the operation of `qsplit`, `qunsplit`, `qbcast`, `qscatter` and `qgather`.

`qcdrun` - This command is expected to execute the actual parallel executable itself and pass any command line arguments. It is responsible for communicating the previously set geometry argument to QMP if that is not already taken care of.

One may question the rationale of the two stage procedure above. It is driven by the fact that not all machines support a fully switchable topology, but do support more flexibility than the hardwired machine setup. On the QCDOC for example the `qsetGeometry` can map onto a QOS call to `qpartition_remap` to try and setup the required geometry. Alternatively, since we assume that the batch script may already be fully set up, it could simply check that

the required geometry is conforming with the currently set axis map. In this case `qcdrun` is a straight map to the QOS `qrun` function. A similar situation may occur on other fixed grid machines.

On the clusters and typically in a batch queue of an MPI based system the job will have been allocated a set of nodes or MPI processes, and we are free to map these as we wish. In this case `qsetGeometry` can set an environment variable that may be used to specify `--qmp-geom` flags to QMP. Alternatively and perhaps most usefully, it can set up a process to processor mapping file, allowing more efficient use of the communication resource for example on the Infiniband architecture. `qcdrun` can then unify the call to one of the myriad *mpirun* variants.

In addition to the above the batch system requires access to the following general and scripting tools:

`perl` - the PERL scripting language

`python` - yet another common scripting language

although we note that these scripts do not need to be able to manipulate the back end but simply may be used within a context of an external shell to control job workflow.

2.5 Interactive System, Compiling, Libraries and Tools

We expect the interactive system to be a standard UNIX like system with at least the `bash` and `tcsh` standard shells. We expect the system to have compilers for both front end and back end (cross) compilation. We also require some basic development tools such as the `cvs` version control system, GNU `make` as well as the GNU `autoconf` and `automake` tools. A detailed list of these will be given later.

In order to support building the SciDAC codes we require several external libraries to be present, in a form prepared for linking into codes for the target system (in other words cross built if necessary). Each external library will be characterized by the following four environment variables (where XXX is used to denote the name of a generic library)

`XXX_HOME` - the installation prefix for the library (eg `/usr/local/lxxx`).

`XXX_CFLAGS` - the flags required by the compiler to compile with this library: (typically listing include paths (`-I`) and CPP macro definitions (`-D`))

`XXX_LDFLAGS` - the flags required to be passed to the linker for example to set up a link library path (`-L`).

`XXX_LIBS` - the list of external libraries needed for linking.

Where several versions of the library exist, these variables always point to a production version.

2.6 Where are the SciDAC codes?

It is our belief that the combinatorial space for the variety of configurations of SciDAC software is too large to support in a standardized environment. Further a specialized site like BNL supporting the QCDOC may not wish to have an optimized installation of the `chroma` software should they lack the manpower to keep up to date with bugfixes new features etc, and would prefer to concentrate on their local codes with which they are doing large scale production (in their case, the `cps`). Likewise, it would be burdensome for another site to maintain a version of the `cps` for clusters where it may never be used. However, we do require that all the necessary tools and external libraries be present to make building codes straightforward.

On the other hand, several SciDAC libraries are useful and are generally needed by the application layer. Into this class we consider QLA, QDP-C and QIO and qdp++ so while we do not mandate installation of these libraries we do strongly encourage their installation.

2.7 Setting Up the Environment

The environment is expected to be set up by sourcing a setup script both in batch and interactive mode. This needs to be in a standard location so that it can always be found. The setup script takes one parameter for now, and that is the architecture of the system. Supported architectures currently are: `gigE`, `ib`, `qcdoc`. In the future we may extend this list (candidates for inclusion are `bgl`, `xt3`, `power` once we determine whether we can access facilities with these architectures and whether the run time environment is portable there.)

While environment variables are straightforward to set, standard program names may be less so. Here the solution may be to manipulate the user's system path (`$PATH`) and or to use symbolic links.

Additionally, we will provide a separate setup script specific to the setting up of a particular compiler toolchain version. More details follow in the next section.

3 Concrete Definitions

We now list concrete definitions and how we expect to be able to test them.

3.1 Setup Script

Item: Environment Variable `$CRE_HOME`

Definition: This environment variable points to the root of the Common Runtime environment installation. It needs to be set either by login scripts or by the user. Rationale: it should allow us to use the CRE on a system in user space.

Testing Strategy: Check that `$CRE_HOME` is not empty. Check that `$CRE_HOME/bin` exists and contains `setup.sh` and `setup.csh`. Check that `$CRE_HOME/etc` exists and contains the file `supported_architectures`

Item: `setup.sh` and `setup.csh`

Definition: These are Bash and TCSH startup files respectively. Their job is to place the standardized commands to the front of the users `$PATH`, to set the filesystem related environment variables and to set the library related environment variables for the standard set of libraries. Arguments are: `gigE`, `qcdoc` and `ib`. A site may support only a subset of these encoded in the `supported_architectures` file

Testing Strategy: After setting `$CRE_HOME`, source the script with a particular argument. After this test for the existence and non-emptiness of all defined environment variables, and that all the defined programs are on the `$PATH`. This test should work from both interactive mode and from within a script ran on the BSEN node.

Item: `supported_architectures` file

Definition: Located in `$CRE_HOME/etc` this file lists the supported architectures at the given sites. Each line should contain a new architecture. These architectures are the valid arguments to the setup scripts at this site.

Testing Strategy: Test for the existence of the file. Test that each line is one of the valid targets for `setup.[c]sh` referring to the previous definition. A stronger test: Execute the `setup.[c]sh` script with each line of this file as its argument from both interactive and batch modes. All tests must succeed

3.2 Filesystem

Item: Environment Variable `$QCACHE`

Definition: This environment variable points to a large disk that is accessible from the IN. We define the actual implementation of this as two environment variables `$QCACHE_PROJECT` and `$QCACHE_USER` for project and user specific directories separately. These directories are solely for data use. .

Testing Strategy:

Interactive Nodes: Check that one can `cd` to the `$QCACHE` directories (both `$QCACHE_PROJECT` and `$QCACHE_USER`). For each project directory check that it is group owned by the corresponding project group and has suitable permissions for group read,write,execute and if possible has a sticky tag to enforce group writeability of subdirectories. For each user directory check that it is owned by the appropriate user and has appropriate permissions

Batch Script: From a batch job check that the environment variables exist. Check that a suitable `qcp` command invocation can copy files from the BSEN to the area, and then copy it back into an area directly accessible from the batch script

Item: Environment Variable `$QDATA`

Definition: This points to the location of the staging area which is shared between the IN and any potential BSEN as well as at least one nominated master I/O compute node. It is expected to have no further defined substructure and can be written to directly by the user. Must exist both on the INs, the BSEN and master I/O nodes of the parallel job

Testing Strategy:

Interactive Nodes: Check that `$QDATA` exists, is a directory and has owner RWX permissions and that the owner is the current user.

Batch Nodes: Check that `$QDATA` exists, is a directory and has owner RWX permissions and that the owner is the current user

Parallel Running: Specified master I/O nodes should be able to open and write files to this filesystem.

Item: Environment Variable `$QSCRATCH`

Definition: Each node is expected to be able to write to its own high performance filesystem *scratch* area. The path to this area should be the same on all nodes from the point of view of the parallel program. This path is encoded in the batch script in the `$QSCRATCH` variable. *On a multicore or multinode system there may be contention for filenames as the cores/nodes may share the same scratch filesystem. In this case it is a user responsibility to suitably annotate filenames with a QMP process ID (QID). QIO does this automatically*

Testing Strategy: A parallel program is given `$QSCRATCH` as an input argument. Each QMP process writes a file `myfile.QID` into this directory. The file contains the unique QID of the process. The process reads back its file and asserts that it read the correct QID

Item: `getProcessScratchPath` script/program

Definition:

Invocation: `getProcessScratchPath processID`

Arguments:

`QID` : The QMP process ID (QID) whose scratch path is sought

Discussion: This script should be used to ascertain the URL of the scratch path of a given process with `process ID` so that the batch script can place files there. This directory is accessible to the parallel job through the value of `$QSCRATCH` that has been passed to it.

Pre Conditions: the geometry has been set with `qsetGeometry` in order to be able to determine the mapping between processes and their nodes

Post Conditions: None

Testing Strategy: The test strategy here is somewhat elaborate, and involves both the batch script and the parallel job. The batch script has to write a file to `$QDATA`. It then should get a random number from the allowed process IDs (which we will call `RPROC`.) It should ascertain the scratch path of `RPROC` using `getProcessScratchPath`, and use `qcp` to move the file from `$QDATA` to this URL. A parallel job can then be started with `RPROC` and `$QSCRATCH` as arguments. In this job, process with ID `RPROC` tries to read the file from the supplied `$QSCRATCH` directory, and assert its contents are as expected

Item: Environment Variable `$HOME`

Definition: This is the users home directory. The home directory has path `/home/username` where `username` is identical to the users log in ID

Testing Strategy: Check that `cd` with no arguments places you in `$HOME`. Check ownership and access permissions.

3.3 File Management

Item: `qcp` program/script

Definition:

Invocation: `qcp source dest`

Arguments:

`source`: The source file URL or local file path

`dest`: The destination file URL or local file path

Discussion: This can be used within a batch job to copy files between filesystems that are not mounted on a given node. The copying must proceed without the requirement for a password. As such the copy is restricted to local systems.

Pre Conditions: None

Post Conditions: None

Testing Strategy: Execute a `qcp` command to copy from `$QDATA` to `$QCACHE`. Then execute the reverse transfer except move the file to a different name from original. Verify the file integrity.

Item: `qsplrit` program/script

Definition:

Invocation: `qsplrit singlefile dest`

Arguments:

`singlefile` : The singlefile QIO file to split.

`dest` : The destination directory in which the split files will be placed.

Discussion: Split a QIO file into pieces. The split files remain in the `dest` directory and have `singlefile` as their QIO Prefix

Pre Conditions: `qsetGeometry` has to have been called in order to set up the map of QIDs to nodes and `$QSCRATCHdirectories`

Post Conditions: none

Testing Strategy: Split a QIO file. Invert the process with `qunsplrit`. Check that the file has maintained integrity. Try to split a file before the geometry has been defined. It should result in an error

Item: `qunsplrit` program/script

Definition:

Invocation: `qunsplrit source singlefile`

Arguments:

`source` : The directory where the split files are

`singlefile` : The file prefix of the split files, also the name of the resulting singlefile file.

Discussion: Inverse operation of `qsplrit`. It is expected that the files in the `source` directory have `singlefile` as their QIO prefix.

Pre Conditions: `qsetGeometry` has to have been called in order to set up the map of process IDs to nodes and `$QSCRATCHdirectories`

Post Conditions: none

Testing Strategy: A parallel job can write a MULTIFILE or PARTFILE QIO file, which can be collected on exit of this program by the `qcollect` command and then unsplit with `qunsplrit`. It can then be renamed to a different name and resplit using `qsplrit`. Once can assert that the original pieces and the newly split pieces contain the same data.

Item: qbcast script/program

Definition:

Invocation: qbcast file destdir

Arguments:

file: The name of the file to broadcast

destdir: The path under \$QSCRATCH where the copies should be placed

Discussion: Copies a given file to the destdir within \$QSCRATCH for every process

Pre Conditions: qsetGeometry has to have been called to create the mapping from processes to nodes and \$QSCRATCH directories. The broadcast files will be found by the parallel programs in \$QSCRATCH/destdir/file

Post Conditions: None

Testing Strategy: Create a file in \$QDATA containing the number 1. Broadcast it with qbcast to some location. Launch a parallel job which reads the integer from each file. The global sum of this integer should be the number of parallel processes

Item: qscatter script/program

Definition:

Invocation: qscatter srcdir destdir file

Arguments:

srcdir: The name of source directory where the files to be scattered are.

destdir: The path under \$QSCRATCH where the files are to be scattered

file: The file prefix of the file to be scattered.

Discussion: Copies a given file from the srcdir into \$QSCRATCH/path for every QMP process. The filenames of the files to be scattered must be in the style of QIO (ie: file suitably postfixed by the QID).

Pre Conditions: qsetGeometry has to have been called to create the mapping from processes to nodes and \$QSCRATCH directories.

Post Conditions: None

Testing Strategy: Take a single node QIO program and write a sufficiently large file. Use qsplit to split it for a parallel layout. Scatter the files with qscatter. Gather the files with qgather to a different directory. Reassemble with qunsplit and verify the integrity of the resulting file. This is an end to end test.

Item: qgather script/program

Definition:

Invocation: qgather srcdir destdir file

Arguments:

srcdir: The path under \$QSCRATCH where the files to be collected from.

destdir: The path of the directory where the collected files should be collected to.

file: The name of the file to be collected (QIO file prefix).

Discussion: Copies a given file from the `srcdir` within `$QSCRATCH` for every process, into a local directory called `dstdir`. The filenames of the files to be gathered must be in the style of QIO (file suitably postfixed by the QID).

Pre Conditions: `qsetGeometry` has to have been called to create the mapping from processes to nodes and `$QSCRATCH` directories. The files will be placed in the directory: `file/` in the working directory where the command is invoked.

Post Conditions: None

Testing Strategy: Each process in a parallel program should write a QIO file into `$QSCRATCH` and write its QID into it. After the program has terminated the files should be collected with `qgather`. The gathered files should then be `rescattered`. One should verify that the re-scattered files contain the correct QID number in them.

3.4 Launching Parallel Jobs - Batch Script Environment

Item: `qsetGeometry` program/script

Definition:

Invocation: `qsetGeometry LX LY LZ LT LS`

Arguments:

LX : Length of processor grid in the X direction

LY : Length of processor grid in the Y direction

LZ : Length of processor grid in the Z direction

LT : Length of processor grid in the T direction

LS : Length of processor grid in the S direction

Discussion: Set up QMP with the desired topology mapping if possible, and then create a mapping from QMP process IDs to `$QSCRATCH` URLs. Otherwise throw an error. In addition this job may perform switch connectivity optimisation. Two scenarios exist

- Fixed layout machines where QMP ID assignment is predictable (QCDOC after booting and axis remap and Toroidal Cluster Panels). In this case the job is straightforward. The script merely needs to check that what is requested is compliant with what is available and throw an error if this is not so. (This is similar in style to the QMP topology setup calls).
- Volatile layout machines (MPI based). In this case the job is difficult, as the mapping of QIDs to process nodes is not immediately straightforward and may need to be configured with system/MPI specific parameter files.

Pre Conditions: Machine should be placed in a state where the mapping of QIDs to nodes/processes becomes predictable.

Post Conditions: A QID map file is created containing a mapping from every process to its `$QSCRATCH` directory. A node file suitable for the relevant run command is created (eg `mpi.conf`) if appropriate. On the QCDOC after the execution of this step the user should not need to call `qpartition_remap`

Testing Strategy: Call `qsetGeometry` to set a particular geometry. Run a parallel program with `qcdrun` on the geometry. The program should open a file in the `$QSCRATCH` area of each node and write its process ID into the file called `$QSCRATCH/myfile.procID`. After the job terminates, the script should go through the mapping file produced by `qsetGeometry` and recover all the output files and verify their contents.

Item: qcdrun program script:

Definition:

Invocation: qcdrun executable arguments

Arguments:

executable: The executable to run

arguments: Command line arguments to the executable

Discussion: The script/executable should launch the parallel executable. It is expected to be called after `qsetGeometry` has set up the geometry so it should need no further arguments (internally it can use any files `qsetGeometry` may have produced.)

Pre Conditions: `qsetGeometry` has to have had been called

Post Conditions: None

Testing Strategy: Launch a parallel job that writes a single file into `$QDATA` containing the string Hello World.

Item: perl programming language

Definition: Perl v5 should be available from the batch environment

Testing Strategy: Execute a Perl script and test its output

Item: python programming language

Definition: Python v2.4 or better should be available to the batch environment

Testing Strategy: Execute a Python Script

3.5 Interactive System, Compiling Libraries and Tools

In this section we forgo the testing strategies, as most of the tools involve installing fairly standard packages. A common set of packages which may be run for interactive purposes and should be available at every site is listed in table 1. Additionally packages that have special relevance on Intel based hardware are listed in table 2 while QCDOC specific packages are listed in table 3. Specifically required Python modules are listed in table 4.

With regard to libraries we distinguish 2 kinds: 3rd party external libraries and SciDAC libraries. Recall that we mentioned earlier that each library will have 4 associated environment variables: `XXX_HOME`, `XXX_CFLAGS`, `XXX_LDFLAGS` and `XXX_LIBS`. In tables 5 and 6 we list the third party, and SciDAC libraries we require including their `XXX` prefixes.

The cut of SciDAC libraries has been made to be able to straightforwardly compile up other user codes (eg MILC, CPS, Chroma) and to suitably shorten the compilation chain. We distinguish between libraries that we'd like to see versus strictly required (based on earlier rationale). We only consider production (Parallel/Cross compiled) version of SciDAC packages. Host versions where appropriate may of course be provided. We recommend an environment variable name scheme similar to the one below prefixed by the string `HOST_`.

Finally in this section we make a few extra definitions:

Package	Version
bash	2.0 or higher
tcsh	6.11 or higher
GNU make	3.80 or higher
cvs	1.11 or higher
autoconf	2.59 or higher
automake	1.9.3
GNU m4	1.4.1 or higher
perl	5.x series
python	2.4.x series
java	1.4.2 or higher
apache maven	2.0.4 or higher
xmgrace	any version
emacs	a recent version
gnuplot	4.0
axis (plotting package)	???
GNU version of UNIX plot	???
subversion (svn)	1.3.x

Figure 1: Native packages to install on INs

Package	Version
gcc/g++ toolchain	3.2.x
gcc/g++ toolchain	3.4.x
gcc/g++ toolchain	4.1.x
valgrind	2.2 or higher
boost C++ libraries	?
gdb	5.0 or higher

Figure 2: Additional packages to install on Intel Systems

Package	Version
powerpc-gnu-elf-gcc/g++ cross compiler toolchain	3.4.x
XIC and xlc (native - front end) toolchain	v6 or higher
RiscWATCH	appropriate version

Figure 3: Additional packages to install on QCDOC Systems

Python Module
scipy
numarray
Numeric
Gnuplot

Figure 4: Python Modules that Need to be Installed

XXX for XXX_HOME etc	Library	version
GMP	GNU Multiple Precision Library (libgmp)	4.1.4
LIBXML2	LibXML2 XML Parser (libxml2)	2.6.6 ²
BAGEL	Bagel Assembler generator	1.4.0

Figure 5: Third Party Libraries that Need to be Installed in possibly cross compiled mode

XXX for XXX_HOME etc	Library	Required
QMP	SciDac QMP	Yes
QLA	SciDAC QLA-C library	No
QIO	SciDAC QIO library	No
QDPC	SciDAC QDP-C library	No
QDPXX_SINGLE	SciDAC QDP C++ library (single prec)	No
QDPXX_DOUBLE	SciDAC QDP C++ library (double prec)	No

Figure 6: SciDAC Libraries

Item: \$QCC Environment variable

Definition: Path to the C (cross) compiler

Testing Strategy:

Item: \$QCXX Environment variable

Definition: Path to the C++ (cross) compiler

Testing Strategy:

Item: \$QAS Environment variable

Definition: Path to the (cross) assembler

Testing Strategy:

Item: \$QLD Environment variable

Definition: Path to the (cross) linker

Testing Strategy:

Item: \$QAR Environment variable

Definition: Path to the (cross) archiver

Testing Strategy:

Item: \$QRANLIB Environment variable

Definition: Path to the (cross) version of ranlib

Testing Strategy:

Item: qsetToolchain Script

Definition:

Invocation: qsetToolchain toolchain version

Arguments:

toolchain Name of the back end toolchain. Allowed values are gcc, xlc and icc the latter is the Intel Compiler.

version A version appropriate for the toolchain eg: 3.4.1 for gcc 9.0 for icc.

Discussion: Sets up the environment variables for the toolchain.

Pre Conditions: None

Post Conditions: None

This command can be used to set up a particular back end development toolchain. If this command is not invoked, a gcc version 3.4 toolchain should be set up

Testing Strategy: This command is not straightforward to test uniformly. For gcc targets executing gcc --version and qcxx --version should display the compiler version, which can be checked against the version asked for.

Item: \$CRE_HOME/etc/toolchains file

Definition: Lists all available toolchains. The format is as follows. Each line contains a toolchain as shown in figure 7. Each pair of toolchain and version must be a valid pair of parameters to qsetToolchain

Testing Strategy: For each line in the toolchains file, execute qsetToolchain. Inspect the properties of the resulting gcc etc files.

4 Advice to implementing sites

In this section we draw together those things that we feel are not reasonably within our control because they are a local administrative matter at the sites themselves, or because we don't know of a straightforward way to automatically test them.

```
# $CRE_HOME/etc/toolchains file example
gcc,3.2.1
gcc,3.4.1
xlc,6
```

Figure 7: Example `$CRE_HOME/etc/toolchains` file showing two gcc versions and one xlc version

4.1 Filesystem

We had the following vision about the implementation of the model filesystem

- The `$HOME/username` - ie: homedirectory space - should be backed up frequently and should have a small quota. This is because the homedirectories may be mounted on the nodes and we would like to discourage its use as a staging space - there is `$QDATA` for that. Of course on some system the homedirectory space may play the role of `$QDATA` in which case this point is moot.
- `$QCACHE` need not be backed up, and may act as a front end for tertiary storage (eg a data grid, SAN, or tape system). We have not expected to mount this filesystem to the BSEN or the compute nodes, because
 - we do not expect NFS to scale to the number of nodes in a parallel computer of $O(100)$ - $O(10000)$ nodes under simultaneous access by all the nodes. Therefore we do not wish to destabilize either the compute nodes, or the large filesystem.
 - if the area is used as a staging disk for tape or data grid it is possible that file access times can be long (on the order of several minutes in the case of tape). We do not wish this latency to impact running jobs - in other words, we do not wish jobs to have to wait for access to these files which may waste their time. Hence the intended policy of pre-staging to `$QDATA`, in which case the disk may still be not too high performance, but at least has low latency.

Having said all that, mechanisms exist for example in PBS to mount the `$QCACHE` filesystem on demand to just the BSEN of a job through the prologue scripts and it could be unmounted through the epilogue script - thus reducing the number of mounts to just the BSENs (or $O(10)$ mounts at a site) to which we can safely expect NFS to scale, and the problems of disk latency may not appear in the case of say a SAN - in which case the limiting factor may be network bandwidth.

Regarding other attributes of this filesystem, it should endeavor to always keep at least 100Gbytes of disk space so that output of jobs can always be copied to it. In the case where the facility has a tape storage attached to it auto-migration may also be desirable. We expect large quotas to be in effect in `$QCACHE`, possibly in line with the declarations for space made at the time of SciDAC proposals.

- We recommend that the `$QSCRATCH` filesystems be high performance from the point of view of being low latency and high bandwidth, since otherwise staging temporary files may be too time consuming. We currently envision that the aggregate size of `$QSCRATCH` over all the compute nodes is modest – at least one terabyte per teraflops sustained. The `$QSCRATCH` area is expected to be transient/at risk, although how this is enforced is up to the local site. It can be cleaned automatically by prologue and epilogue scripts under PBS or in the case of the QCDOC they may be automatically cleaned on machine allocation or de-allocation.

4.2 Interactive Systems

We make the following suggestions for the interactive nodes

- We envisage that the systems can be accessed via *ssh*. Security constraints may require that access be achieved through more than two hops (eg hop to a login machine followed by a hop to the actual IN).
- This may require the installation of Kerberised version of *ssh* at some sites. This currently is most easily achieved by having a standalone machine on which the *ssh* tools have been replaced by the kerberised *ssh* tools available from Fermilab (and in Scientific Linux).
- While interactive nodes may need multiple hops, for file transfers to and from the sites single hop transfers are very highly desirable. In order for this to work tho, sites may need to open some ports in their firewall, otherwise transfer would have to proceed through the same number of hops as the logins. However, the intermediary login host may not have sufficient space to stage the multi-hop transfers. Transfers can proceed with *scp* for now, tunneled through some pre-agreed ports, possibly kerberised to talk to Fermilab. We expect in the future some kind of grid transfer tool such as *SRM* which may transfer files under the hood with *GridFTP* or *HTTP* protocols. However, even for this, issues of firewalls will need to be faced.

4.3 Batch Systems

We make the following suggestions for the batch system (BSEN).

- We expect `$HOME` and `$QDATA` to be mounted to the BSEN. `$QCACHE` need not be mounted
- We envision providing at least the `bash` and `tcsh` shells, however we are aware of the constraint on the QCDOC that currently the machine is operated solely through `tcsh`. While the `qdaemon` architecture makes it possible to control the operating system through sockets from say Perl or Python or Bash this would require a large development effort. So for this definition we make special allowance for the QCDOC. There the batch script is exclusively a modified *tcsh* called *qsh*.

We do make the note tho that *tcsh* has a maximum command line limit. This has impact on cluster systems running MPI. On the Jlab infiniband system running *mpirun* with *tcsh* as the login shell failed when scaled to run beyond a certain amount of nodes, as the *mpirun* command constructed an underlying command invocation which was too long.

- We expect the copy tool on the BSEN to not require a password

4.4 Miscellany: Large Files

As we increase lattice sizes we may hit the limit of 32bit files (generally 2GB). This can cause trouble. We strongly recommend the usage of filesystems that are not limited by this problem however this in itself is not necessarily sufficient to solve the problem. The following issues remain:

- Support for large files is not always standard across systems. Different systems require different compile flags to write large files.
- While the underlying filesystem may support large files, tools may not. In particular we need to make sure *qcp* and *scp* does.
- On the QCDOC unless this has changed recently, the NFS client on the nodes could not cope with large files at all. This can be worked around by the users.

5 Advice to Users

In this section we list some advice to users to enable them to use the system more effectively.

5.1 Filesystem

- Files may need to be explicitly copied from \$QCACHE to \$QDATA. The `qcp` command should eventually make this transparent
- \$HOME should be backed up, but \$QDATA and \$QCACHE may not be. Please check documentation and policy of the local site.
- Migration to tape from \$QCACHE is site dependent. Refer to local documentation.
- Use QIO which is most likely to be supported by the `qscatter`, `qgather`, `qsplit` and `qunsplit`. *if QIO is not used, `qunsplit` and `qsplit` may not work. In addition at least the files suffixes must be in QIO conventions for `qgather` and `qscatter` to work correctly*
- Files over 2Gb still cause difficulty on 32bit systems. This is a problem that is especially keenly felt when dealing with propagators or eigenvalue/eigenvector calculations which are generally larger than just the gauge field. In particular writing many eigenvectors into one file can most easily overrun the 2Gb limit even for modest lattice sizes. Even if the filesystems themselves can handle large files you may still experience problems because.
 - The way C/C++ code handles large file support is not standard across all systems. Different compiler/linker flags may be needed on different architectures. Please refer to your local documentation or machine expert.
 - On the QCDQC the nodes cannot support large files at all. In this case the only recourse may be to
 - * Use the MULTIFILE mode of QIO and write lots of smaller files. These then need to be collected from the front end. Likewise for input data, files may need to be exploded into files that are less than 2GB in size and pre staged.
 - * Prefer writing individual fermions in separate files

5.2 Interactive System

- Interactive systems should look like normal UNIX systems. We have defined the minimum amount of library support here so you may need to compile your own SciDAC library. We expect to have documentation on how to do this. Some sites may provide pre compiled versions of various application level codes such as the CPS or Chroma, but this is not mandated.
- Be aware that some machines may need cross compilation, in other words the compiled programs do not run on the machines on which you compile them. We provide the wrappers for the compilation toolchain eg: `gcc`, `gccx` `gas`, `qld`, `qar` and `qranlib` but nothing beats actually knowing what you are doing and working with the tools directly. Particular nasties are the cross versions of `ar` and `ranlib` which can cause strange error messages when say a cross compiler tries to use an archive that has been created with the native archiver etc.
- Be aware that you may need multiple hops to log into the interactive nodes
- Be aware that until single hop file transfers are implemented you may need multi hop file transfers
- If Kerberos is involved you may need to do your transfer from a special kerberised host, Refer to local site documentation.

5.3 Batch System Execution Node (BSEN)

We make the following general remarks about the Batch System Execution Node:

- On the QCDOC you must use the *qssh* to control your machine/partition
- Under PBS you have several useful environment variables available to you, in particular the `PBS_NODEFILE` lists the nodes your job is using, typically one per process (in multi-core system host names are repeated). You can use this to count the number of nodes. Also typically the BSEN node is the first of this list.
- Be aware `$QSCRATCH` is only in existence to be passed to a parallel program. It may be completely useless to use in any other way in the script, since the scripts view of the various scratch directories may be vastly different from the running job.
- In multi core systems, several nodes can write to the same `$QSCRATCH` directory, causing potential filename collisions. In a multi core system, the whole `$QSCRATCH` technique makes sense only if the files themselves are annotated by a *core ID* or *process ID*. QIO does this automatically. If you don't use QIO you may have to do it yourself. However, we only support QIO for conversion between single file and multifile files.
- Be aware that `$QSCRATCH` is potentially volatile, and may be wiped after your job completes, or you deallocate your machine. Your job scripts should preserve any files you should need onto `$QDATA` using the `qunsplit` and `qcollect` commands before the script exits or the partition is deallocated.

5.4 Parallel Jobs

- Your homedirectory may not be visible from compute nodes. Only the directory pointed to by `$QSCRATCH` is guaranteed to be visible to every node. Special I/O nodes (which may be all the nodes in some instances) can also see `$QDATA`.
- The Unix Streams *stdout* and *stderr* may be live from all nodes. Using them excessively may generate a lot of unwanted data. Some systems provide special commands to output solely through one node and this is preferred (eg: in QDP++ one should use `QDPPIO::cout` rather than `std::cout`)
- On the QCDOC only Node 0 is connected directly to the terminal and other nodes streams may need to be subsequently retrieved with the `qnodes_print` command when execution has finished. Nevertheless, the memory area to which these messages are written is limited and may wrap around - obscuring previous messages
- The *stdin* input stream is not supported for parallel jobs. Instead one should use parameter files or command line arguments.

6 Summary

In this document we have outlined our definition of the Common Runtime Environment. We have tried to keep it minimalist in terms of features to aid implementation and portability while maximizing its usefulness. We believe it to be straightforwardly implementable across the current metafacility sites.

In summary the main features of the environment are:

- A standardized model of a filesystem: `$HOME`, `$QCACHE`, `$QDATA`, `$QSCRATCH`

- A standardized targeted toolchain defined through environment variables: \$QCC, \$QCXX, \$QAS, \$QLD, \$QAR, \$QRANLIB
- A standard set of file manipulation tools from within a batch script: qcp, qsplint, qunsplint, qbcast, qcollect
- A standard set of scripts to run jobs: qsetGeometry, qcdrun
- A set of setup scripts and capability files:
 - Setup the environment for an architecture: setup.sh and setup.csh
 - Setup for a particular version of the toolchain:qsetToolchain
- A series of standard libraries, and an environment variable scheme to keep track of them.

6.1 Potential Future Developments

We envisage that in the future the following improvements may be made but which require research and development

- Data Grid tools may be deployed on the INs and the BSEs.
- Introspection of available resources (systems, compilers etc) may be placed on a more grid like footing through Grid Information Services.
- Job submission may be standardized through usage of Grid Jobmanagers – although it is likely that the custom system of the QCDOC may need to be excluded from this activity
- Port the environment to other systems if feasible