CPS: Overall Code Structure.

A.N. Jackson & Bálint Joó

Revision: 1.3 Date: 2002/07/23 09:36:26

Contents

1 Overall aims

The CPS code structure and build system aims to:

- Use standard GNU filenames, extensions, directory structures, and compile using standard UNIX/GNU tools.
- Reflect the structure of the object-oriented code in the structure of the source directory tree.
- Allow sources, libraries and binaries for different platforms to coexist.
- Build all libraries relevant to a given platform, so that the specific implementation can be chosen at link time.
- Maintain Compatibility with the current QCDSP hardware and development environment.

2 The Directory Structure

2.1 The top-level of the tree

All files are held in a directory called *cps/*, for the Columbia Physics System. The directory structure at this uppermost level reflects standard GNU practice for organising software distributions.

2.1.1 Files

- *cps/Makefile*
- *cps/configure*
- *cps/README*
- *cps/INSTALL*
- *cps/...etc...*

The autoconf-based configure script, the overall makefile and other top-level files are examined further in section 3 below.

2.1.2 Directories

- *cps/src/* - Directory containing the source tree.
- *cps/include/* - Directory containing the source header files.
- *cps/lib/* - Directory holding the binary library files for a given platform.
- *cps/docs/* - The documentation directory.
- *cps/test/* - The test suite (see section 4).

We consider each directory in turn.

2.2 The Source Tree: *cps/src/...*

There are three design criteria for the organisation of the source tree:

- The top-level of the source tree should reflect logically separable classes of code.
- The deeper directory structure should accurately reflect the object-oriented structure of the code.
- The final leaves of the directory-tree should effectively separate code for different platforms, so that the source files for a particular platform can be easily identified.

2.2.1 The top-level of the source tree

The source directory in this version of the code corresponds to a combination of the source files from *phys/[util,alg,nga,task,mem]*. The original layout was driven by the target platform, whereas the new layout moves the platform-dependencies further down the tree. However on Consultation with Bob, we decided to keep the original *util, alg, nga, task, mem* layout to separate cleanly the functions in these directories. The functions outlined by Bob are:

util – These are physics related utilities, to do with lattices, Dirac Operators, Vectors, linear algebra and the like.

alg – The classes herein correspond to algorithms to perform measurements, such as quark propagators, $\bar{\psi}\psi$ etc.

task – The original intention here was to have combinations of *alg* classes to perform more complicated tasks, such as a Hybrid Monte Calro, with inline measurements and so forth, including running scripts. However, the original development “ran out of energy” and so this directory is somewhat disorganised.

nga – This directory contains routines to perform various kinds of communications (NGA stands for Node Gate Array – the communications hardware of the QCDSP). It is envisaged that we rename this to something like *comms*.

mem – This directory contained code to deal with loading data in and out of internal memory of the DSP chips on QCDSP. It is highly likely that future machines, may need similar routines, for example the QCDOC will have some EDRAM that may need to be accessed through specialised routines.

Bearing in mind our consultation by video conference on Aug 20, 2002, the suggested code structure at this level is as follows:

- *cps/src/alg/* - Directory containing the Alg class.
- *cps/src/comms/* - The comms routines.
- *cps/src/comms/gsum/* - The global sum routines.
- *cps/src/io/* - I/O routines.

- *cps/src/io/qcdio* – The currend QCDIO library
- *cps/src/mem* - Routines for managing internal memories
- *cps/src/task/* - Directory containing the tasks.
- *cps/src/util/DiracOp/* - Directory containing the DiracOp class.
- *cps/src/util/Error/* - Directory containing the Error class.
- *cps/src/util/GlobalJobParameter/* - Directory containing the GlobalJobParameter class.
- *cps/src/util/lapack/* - The lapack linear algebra routines.
- *cps/src/util/Lattice/* - Directory containing the Lattice class.
- *cps/src/util/Mom/* - Directory containing the Mom class.
- *cps/src/util/pmalloc/* - The pmalloc routines.
- *cps/src/util/Random/* - Directory containing the Random class.
- *cps/src/util/RComplex/* - Directory containing the RComplex class.
- *cps/src/util/RFloat/* - Directory containing the RFloat class.
- *cps/src/util/smalloc/* - The smalloc routines.
- *cps/src/util/Vector/* - Directory containing the Vector class.
- *cps/src/util/Verbose/* - Directory containing the Verbose class.

Here we have added an *io* directory which is expected to contain functions for loading and saving configurations, propagators and the like.

In “leaf” directories, upper-case names indicate a corresponding class exists (i.e. the class DiracOp is defined in DiracOp.C in *cps/src/DiracOp*). Lowercase directory names indicate that this directory contains utility functions that are not wrapped in C++ classes.

It has been agreed in the videoconference that leaf directories and library names should follow the names of the classes (ie libDiracOp.a, on UNIX and DiracOp.olb on QCDSP). After some testing it appears that the Compiler/Linker/Archiver toolchains can deal with this naming convention on QCDSP (Tartan), Solaris (cc/CC), AIX (xlc) and Linux (gcc). Hence this naming scheme will be adopted in the new code structure.

There exist currently some functions that are not wrapped in C++ classes, in the short term it is acceptable to put these in a *cps/src/notCPP* directory, but in the long term such a directory ought to be eliminated. Insofar as the pure C / Assembler functions are related to a C++ class, they should reside in the same directory as that class. separate *cps/src/misc/* directory?

Query.1: Should pmalloc and smalloc be combined into one “memory” directory, including all from *phys/mem/*.

2.2.2 The deeper directory structure

This should reflect the OO structure. So, if we look at the Doxygen documentation
(see http://www.epcc.ed.ac.uk/~ukqcd/cps/)
for a given class, e.g. the Dirac Operators:
http://www.epcc.ed.ac.uk/~ukqcd/cps/doxygen/html/class_DiracOp.html

This suggests the following structure:

- *cps/src/DiracOp/*
- *cps/src/DiracOp/DiracOpStagTypes/*
- *cps/src/DiracOp/DiracOpStagTypes/DiracOpStag/*
- *cps/src/DiracOp/DiracOpWilsonTypes/*
- *cps/src/DiracOp/DiracOpWilsonTypes/DiracOpClover/*
- *cps/src/DiracOp/DiracOpWilsonTypes/DiracOpDwf/*
- *cps/src/DiracOp/DiracOpWilsonTypes/DiracOpWilson/*

2.2.3 Separating code for different platforms

File common to all platforms are held in the object's directory. e.g. The pure-C++ file that defines the abstract base class `DiracOp` is held in *cps/src/DiracOp/DiracOp.C*. Any platform dependent files are stored in further subdirectories:

- *cps/src/DiracOp/* - Source common to all platforms.
- *cps/src/DiracOp/noarch/* - Portable versions of usually platform-dependent code.
- *cps/src/DiracOp/qcdsp/* - QCDSP-only files.
- *cps/src/DiracOp/qcdoc/* - QCDOC-only files.
- *cps/src/DiracOp/alpha/* - Alpha-chipset-only files.
- *cps/src/DiracOp/x86linux/* - (x86 chip-series, Linux OS)-only files.

In some cases, for a given platform, multiple implementations of the same functionality are present. For example, there are a large number of QCDSP-specific global sum routines and Dirac operation routines that fulfill the same function. Each of these will be held in a separate directory within the *.../qcdsp/* directory and each will be compiled into a separate library using the name of the directory as the library name. e.g.

phys/util/dirac_op/d_op_wilson_opt_lcl_nos/

becomes

cps/src/DiracOp/DiracOpWilsonTypes/DiracOpWilson/qcdsp/d_op_wilson_opt_lcl_nos/

and (on QCDSP only) will result in a library file called *cps/lib/libd_op_wilson_opt_lcl_nos.a*,

sitting alongside the more general

cps/lib/libDiracOp.a.

There is a large amount of code replication in the global sum and Dirac operator directories. It is not expected that such a large number of classes / libraries will exist in general. The current strategy is to put all the sources and libraries into a machine specific *cps/src/DiracOp/DiracOpWilsonTypes/DiracOpWilson/qcdsp* directory and build them all as is done currently. A second pass may then try and eliminate the replication of codes.

2.3 The Include Files: *cps/include/...*

The current code structure has separated include files to reside relatively close to the source files:

- *phys/util/include* – Include files for the sources in *util*
- *phys/alg/include* – Include files for the sources in *alg*
- *phys/mem/include* – Include files for the sources in *mem*
- *phys/nga/include* – Include files for the sources in *nga*

It is intended to reorganise these, into one single include directory. To prevent undue flattaning of the current structure and mixing of names we envisage keeping the current distinctions within this include directory as in:

- *cps/include/util* – Include files for the sources in *util*
- *cps/include/alg* – Include files for the sources in *alg*
- *cps/include/mem* – Include files for the sources in *mem*
- *cps/include/comms* – Include files for the sources in *comms* – the new name for the *nga* directory.

This has the advantage of maintaining current distinctions. A user can then set his "include path" to *cps/include*, and include files with directives such as #include <util/lattice.h> and so forth. Alternatively, several include paths can be set (for the GNU compiler: *-Icps/include/util* -*Icps/include/alg/mem* -*Icps/include/comms*) in which case one can include headers in sources with directives such as #include<lattice.h>.

Query.2: Which of the above two are preferred?

2.4 The Library Files: *cps/lib/...*

All the library files, one for each of the main directories in *cps/src/* plus ones for platform-dependent variants of a given library (as described at the end of section 2.2.3).

2.5 The Documentation: *cps/docs/...*

This is essentially the same as in the original EPCC-Columbia distribution, with HTML as the primary format, and including the Doxygen-ized version of the code.

2.6 The Test Suite: *cps/test/...*

At first, this will be a copy of the original CPS test suite, or at least the parts of that suite that work on all platforms.

Eventually a more general test suite must be defined and implemented, taking advantage of the new hypercubic RNG implemented at Columbia.

3 The Build System

The build strategy is simple. We use a `configure;make;make install` scheme, where the install can be made to separate binaries for different platforms. For any given target platform, we make every library that is relevant to that platform. The only compilation option that cannot be separated out into a separate library is the decision whether to use float or double precision floating-point arithmetic. Broadly:

```
%./configure --target=[TARGET]  \
             --prefix=/cps/[TARGET]/  \
             --enable-double-precision=[yes|no]
%make
%make install
```

The parallel/serial and other platform-specific options are chosen at the link stage, the tests do this and thus require a separate configure+make system.

`make install` will make and populate a directory called (generically) instdir, by creating the following structure

- *instdir/*
  - *arch/*
    - *include/* copy of include directory.
    - *lib/* all libs for target.
    - *doc/* documents.
      - *doxygen/* doxygened docs, for source used in build.
    - *bin/* executables created by build.
    - *src/* a copy of the sources used in this build.

instdir would be specified as usual by the –prefix option to configure. Note that arch/ may be superfluous, as may bin/ the bin is suggested by the facts that there would always be some stock executables that would be built (e.g. the test ones)

3.1 Makefile Rules

It is envisaged that there would be some file called Makefile.rules say, that encapsulates the rules for building the particular application. This would live in the top-level source directory. The rules could contain:

- Invocations for the compiler, archiver, assembler etc.
- Various compiler and assembler flags (include paths, preprocessing options such as -DHAVE_CONFIG).
- Definitions of suffixes and rules for building object files, e.g. from C++ and from assembler files.

Makefile rules could be either pre-written for specific systems, e.g. Makefile.rules.alpha Makefile.rules.sparc or could have their values filled in by autoconf, or perhaps be generated through some combination. (e.g.: Autoconf fills in path information and optimisation flags, whereas the rest are pre written) (Makefile.rules.alpha.in...)

In subdirectories, one would use recursive files, just like the current QCDSP build system (using the more portable GNU make). The leaf makefiles could all include the top-level rules file. The leaf makefiles would presumably work on some wildcard mechanism (i.e. compile all .C and .S files in the current directory). Library names would be specified by base names of current directory.

Working out how to traverse the right set of directories for a particular architecture, is a problem to be solved by us. Our goals in this direction are that the build system be as simple as possible, for future maintenance.

As a final statement, we need to describe why building the executables is harder than to build the library. We don't know what mode of operation is expected as there are so many options:

- Should we run with no safety (nos)?
- Should we run with dirac_opt_stag or dirac_opt_stag_rdm?
- Should we run a parallel build with a simulated SCU?
- Should we run with QMP?
- ...

One way around this would be to have independent makefiles for the executables (for the most common targets) that could include the rules from the first level configuration – this punts the issue of what options we need to pass to autoconf, and may fit well into the current testing framework.

The list of builds is relatively easy (already exists on QCDSP – and doesn't really exist in a crystallised form for other targets). One possibility is to have a test/qcdsp test/alpha etc etc directories where we could have the different architectural tests (e.g. tests that exercise hardware like SCUs and so worth) another possibility is to have a single directory of test cases with makefiles for all architectures in them (the top-level rules would be used for building, but the library lists would need to be hardwired depending on the test in question.) I lean towards the first option myself. We should discuss with Chris Miller how this fits into his testing environment. Similar setup could be used to build stock applications (hmc, etc etc)

An alternative scheme for building the executables has been suggested by Craig McNeile, which would involve having some generic GUI interface, through which users can choose the particular library versions they desire. This has the advantage, that the choice of libraries for a particular architecture can be "institutionalised". Whether such a GUI is to be a C++ program, Tcl/Tk script, Jave applet or a web form is yet open to debate.

It is not in the remit of this particular reorganisation to decide on, or implement this strategy. Our current brief is to make sure that the libraries can be built and installed.

3.2 External Libraries

Somehow we need to tell the compiler about where these live (e.g. –with-mpi=/usr/mpich, or –with-mpi-include=/usr/mpich/include – so that the right include files can be found – for the libraries only the include files matter. However for the executables obviously the libraries are needed too.

What would be classed as an external library?

- *MPI* (for sure)
- *GM* (on clusters say)
- *QMP* (default no – could be based on MPI or SCU or implemented directly...)
- *SCU* (default no – too ingrained in the code)

4 The Test Suite

This will require a complex configure script to allow the user to choose between the different possible libraries available for a given platform. However, for a general platform, the core option is whether to run in serial or in parallel.