Notes on changes to the Columbia code.

A.N. Jackson & S. Booth

Revision: 1.9 Date: 2002/06/06 13:28:25

A.N.Jackson@ed.ac.uk.

Contents

1 Version 4.0.0

The original version of the code from Columbia. Internally, this corresponds to revision 1.1.1.1 in the UKQCD QCDOC repository, dated 2001/06/15. This was given the revision tag `phys_4_0_0_open`.

2 Version 4.0.5

This version (largely notional as there was no CVS tag) referred to the basic ANSIfication of the code and the addition of the draft MPI SCU layer.

2.1 ANSIfication

A large number of small changes have been made throughout the code, to turn it into more standard C++. The code will now compile with any recent gcc (only tested under 2.8.1 & 2.95.2, but 2.95.3+ should present few serious issues). The 4.0.0 code only compiled under `gcc 2.8.1`. The compiler errors fell into the following categories:

- Attempts at using inline variables outside their scope.
- Inline declaration of variable without types.
- "assignment to `char*` or `void*` from `const char*` discards qualifiers." This does not occur on all compilers as the default type for inline character strings is compiler dependent. However, this behaviour can usually be controlled via command-line arguments to the compiler.
- No type declaration for `main`.
- Cannot pass `Float` through `(...)`. Only native types can be passed through ellipsis, see section of floating-point below.
- Could not find `sysfunc.h`, `flotar.h`, etcetera. i.e. could not find QCDSP-specific header files.
- Implicit declarations of functions and variables, mainly due to the missing header files.

2.1.1 Variable Scoping

The most common problem was the use of variables beyond their ANSI scope. For example, consider a code fragment of the form:

```
for(int i = 0; i < 10; i++ ) {
  [do something]
```

```
}
for( i = 0; i < 10; i++ ) {
  [do something else]
}
```

This is illegal under ANSI C++, as the integer i does not exist beyond the scope of the for-loop in which it was created. This was fixed by moving the declaration into the required scope:

```
int i;
for(i = 0; i < 10; i++ ) {
  [do something]
}
for( i = 0; i < 10; i++ ) {
  [do something else]
}
```

This yields the correct scoping on both old and ANSI compilers.

2.2 The MPI version of the SCU

The first MPI implementation of the SCU layer was added to the `./nga` directory. More information on this aid to portability can be found in ../mpi-scu/commspec.html.

2.3 Other Minor Issues

Apart from the ANSIisms and MPI, some other minor changes were also made. For this version, the `double64.h` header files have been moved. Identical copies of this file were found in multiple directories (under `./nga/glb_*`). These have been removed and replaced with a single `double64.h` header file in `./nga/include`.

3 Version 4.1.0 (alpha release)

The first major release from EPCC was the 4.1.0 (alpha) version, dated 2001/09/06 cvs revision tag: `phys_4_1_0_alpha_open`. This version had been successfully compiled and tested on a number of platforms (see section on testing below). As well as the 4.0.5 modifications, this version implements flexible (float/double) floating-point precision, and adds an autoconf based configuration system.

3.1 Implementing flexible floating-point precision

As indicated above (§2.1), it is not always possible to use the `rfloat` floating-point implementation on all platforms. To this end, **all** references to rfloat have been replaced with a typedef called `Float`. Also, **all** references to `float`s have been replaced with an "internal float" typedef called `IFloat`. These typedefs are in `./util/include/data_types.h`.

Setting `Float` to `rfloat` and `IFloat` to `float` recovers the original functionality.

Using `Float` = `IFloat` = `float` runs in exactly the same precision as the original, but the serial code runs approximately 20 times faster.

Using `Float` = `IFloat` = `double` makes the code to run in double precision.

This behaviour is controlled from `./config.h` using two compiler macros, e.g.

```
/* Precision of the local calculations: rfloat, float, double. */
#define LOCALCALC_TYPE double
/* Precision of other internal calcs: float, double. */
#define INTERNAL_LOCALCALC_TYPE double
```

3.1.1 Global Summations

Some global operations (essentially global sums) can be set to use double-precision separately to the rest of the code. This rests on the use of the `double64.h` header file in `./nga/include`, which defines a 64-bit floating-point type (`Double64`) for use with the global-sum calls. On QCDSP, the original behaviour (involving a specialised floating-point class) is retained. Elsewhere, the behaviour is typedef'd such that:

```
typedef GLOBALSUM_TYPE Double64;
```

Where the compiler macro `GLOBALSUM_TYPE` is defined in `./config.h`. The default is `double`.

3.2 Addition of an autoconf configuration system

There are two parallel make-structures in the codebase. The QCDSP one has been left unchanged, but the GNU one has been modified. The autoconf system revolves around a configure script, `configure.in`, which defines the platform dependencies and from which the `./configure` script is created. This latter script takes the following files:

```
./config.h.in
./Makefile.gnu.in
./Makefile.gnutests.in
./tests/regression.pl.in
```

and creates the files

```
./config.h
./Makefile.gnu
./Makefile.gnutests
./tests/regression.pl
```

from them via macro substitution. As well as controlling platform dependence, the configure script allows you to set the code precision and decide whether to compile as a serial or parallel (MPI) program. Run `./configure --help` for more information.

Note that some of the global code options set in config.h are extremely important, and so any other header files which did not directly or indirectly include config.h had to be changed to do so.

On QCDSP, this distribution should still compile straight out of the box. On other platforms, the process is slightly more involved. See ../porting/porting.html for more information.

3.3 Other Minor Changes

The seeding of the serial RNG has also been `#defined` in `config.h`, globally instead of locally. Also, a standard level of verbosity has been `#defined` in `config.h` for the test programs, instead of having a separate verbose-level in each code. Note also that a new verbosity setting has been added to the `Verbose` class so that the RNG seed(s) used by the code can be output without also having to output vast amounts of program-flow information.

3.4 Testing

Following all of these changes, the code has been tested using a slightly modified implementation of the original Qrun testing script. A perl script (tests/regression.pl) is used to generate a batch script (tests/regression.sh). Running this script will place the output of each code (both from stdio and from and .dat files) into the tests/regressions directory. This output can be checked against that provided in the following subdirectories.

```
phys/tests/regressions/v4_0_0/parallel
phys/tests/regressions/v4_0_0/serial
```

These contain the output from the original version of the code that EPCC received, prior to any of our changes but using a standardised level of verbosity.

```
phys/tests/regressions/v4_1_0/parallel
phys/tests/regressions/v4_1_0/serial
```

These contain the output from the current version of the code. The two sets of files differ only trivially, in that the Clock timings are different, the CRAM addresses have changed, and that the conditions for the outputting of the RNG printing string have been altered.

Note that all of this output is from QCDSP. The MPI version has yet to be rigorously tested.

4 Version 4.1.0 (beta release)

Date: 23rd January 2001; CVS revision tag `phys_4_1_0_beta_open`.

Since the above version, a few very minor changes have been made to the code, along with much modification of the documentation. Also, the names of the library archive files (under the GNU make-structure) have been changed to `*.a` instead of `*.lib` to ensure portability. The suggested set of CVS `$ Tag: $` tags have been added to the files added by EPCC.

5 Version 4.1.0 (beta release)

Date: 23rd January 2001; CVS revision tag `phys_4_1_0_beta_open`.

6 Version 4.1.0 (beta release)

Date: 23rd January 2001; CVS revision tag `phys_4_1_0_beta_open`.

7 Version 4.1.0 (beta release)

Date: 23rd January 2001; CVS revision tag `phys_4_1_0_beta_open`.

4.1 4.0.0 Columbia The original version of the code recieved by EPCC from Columbia.

UKQCD CVS Repository Tag: phys_4_0_0_open

4.2 4.1.0 UKQCD ANSIfication, initial MPI SCU, type flexibility (float/double), basic qcdio support includuing the qload guage configuration loader.

UKQCD CVS Repository Tag: phys_4_1_0_alpha_open

UKQCD CVS Repository Tag: phys_4_1_0_beta_open

UKQCD CVS Repository Tag: Root-of-Columbia4_1_1_test

4.3 4.1.0 Columbia Integration of new physics components

UKQCD CVS Repository Tag: Columbia4_1_1_test-branch

4.4 4.1.1 Assimilation [UKQCD+Columbia] UKQCD+Columbia 4.1.0 code merge

UKQCD CVS Repository Tag: Merged-from-Columbia4_1_1_test

4.5 4.1.2 Tested, Accepted

UKQCD CVS Repository Tag: ???

4.6 4.1.5 Assimilation [+Brookhaven] Code modifications from Brookhaven.

4.7 4.1.6 Additions

8 Pre-4.1.6: Adding a gauge-configuration saving routine

Date: 8th May 2002

There are many possible ways of implementing a routine to save the gauge configuration out of the code when running in parallel. We have to choose one that will scale acceptable given the 10,000 CPUs and 4Gbit/s (?) frontend bandwidth of QCDOC. My thoughts were:

- Each node opens its own file and dumps the local data, after which the I/O node opens all the files and remixes the data in the right order to a new set of T-ordered files. [High I/O usage, tying up one node for some time]
- Each node passes a copy of its own data all the way around the cube, x-dirn, y-dirn and then z-dirn. The I/O node opens a separate file for each x-stream (i.e. #files = local-z * local-y), and appends the data from each file as is passes through. The I/O node can then trivially concatenate these files to form the overall output file. [Quite high I/O usage, and quite a lot of communication too]
- One node requests the data it requires to form the output file in the correct order from each relevant node in turn. This can be done with no buffering at all, but will require many transations with each node. Also, there is no call for arbitrary point-to-point comms, so this will have to be written using the nearest-neighbour primitives. [High communications usage]
- Similar to the above idea, except that simultaneous comms are used to collapse the data from the x-direction (or perhaps the x and y directions) on to the nodes that hold the x=0 plane (or the x=y=0 line). This requires potentially large buffers, but allows much comms overlapping. After the collapse, the data can be collated as in method 3 above. [High memory usage]
- Collate the data onto one node in a large buffer, rearrange the data on that node and then write out the file. [Very large memory usage and ties up one PE for some time]

For the time being I will implement the simplest one, where each processor offloads its data to a different file. These files can be patched together either offline or inline.

Okay, current version writes out a file for each node, with appropriate filenames, using the given precision, transpose and byteswap. Needs testing, but difficult to test properly until the reassembly code is written.

The output file is the right size at least!

N.B. Updated the ukqcd CVS repository access instructions.

I've made the code split the data into files accross the T axis. This should allow the serial version to work perfectly, producing identical output as input. However, the test I just did failed and the files differed. On closer inspection ('od file — head'), is seems the differences were very slight - the odd byte here and there. Therefore, is seems reasonable to assume that the fact that the code was running in double precision means that some information got munged in the transformation to

and from double precision. I am recompiling the whole thing to use single precision, which should allow precise testing using this input data. 1/2 bit errors per 32-bit word.

Darn it. Got pretty much exactly the same results. I now suspect the expansion and compression from 2x3 to full 3x3 and back to 2x3 again, or possibly the transposing. The latter should not do any such thing, and the former should save out the cols/rows that got loaded in, I think. I'll check the transpose action occurs at the right spot to get the original data back out.

Quick check, switch off the transpose and try again... Damn, the same. Trying when outputting the other part of the matrix. Rom from 1 to COLORS instead of 0 to COLORS-1... Hmmm. That didn't work.

Most likely, it is the set of transformations used to reconstruct the matrix from the 3x2 form. I'll try not bothering, and this should read out the data in the same order as it is read in. Excellent! That worked. The norm-orthog-norm-orthog sequence introduces a few minor errors in the numbers.

Checking the code in, so people can at least access the saver routine in serial.

Now to run in in parallel... Some compilation issues, forgot to export CC=mpCC, reconfigure etc. Now trying a complete clean and build on lomond front-end. That worked. Ran in parallel and produced umpteen files with different chunks of the matrix in them. Now all I have to do is remix the data correctly.

30/05/2002

Change of plan. Moving over to a more sensible system where each x,y,z=0 processor opens each TXX output file, and the data is pumped down from the TXX x,y,z cube to the I/O node. The method is borrowed from ideas/code from SPB. Every node loops over all the data, the I/O nodes load the data, all relvent nodes participate in the pumping of data to the destination nodes, and then the destination nodes unpack the data. Much the same code is used to perform the reverse pack-pump-save routine.

This is in place in essence, but the pumping routine appears to be wrong and prism is showing me that there are messages left unheeded and recieved sent without a matching message too.

31/05/2002

Finally working okay, creating a reversible process where data is loaded in a saved out again while being identical (note that the reconstruction of the missing row is commented out to ensure bitwise-identical results). Now stripping out the paranoid (per pump) sync commands and retesting. Seems to be okay, and is a darn sight faster, tens of minutes down to minutes on the lomond front-end (not too surprising really). Yup. Looks fine.

Now I'll check this in and let Craig know its there, but in a beta state as the actual data on each node has not been fully tested yet. Also, I should add some more comments, at least to the head of each subroutine.

4/06/2002

Created a 'maketestdata.c' program and added it to the cvs repository in tests/qcdio/. This creates a set of dummy data files filled with simple incremental data. I'm currently running this through the code to check the data on each node has been loaded correctly.

5/06/2002

The test program for qcdio now creates its own test data, loads it in, checks it has been loaded correctly and then saves it out. The only gap is that it does not check that the resulting files are bit-identical to the input files, but the test.bindiff.sh script will do that. This routine is now considered complete and ready for use by others.

However, the routine to write out the g/c .par file is not complete, and the code in qcdio.C is under-commented. These issues will remain outstanding for some time as I have to work on other projects.