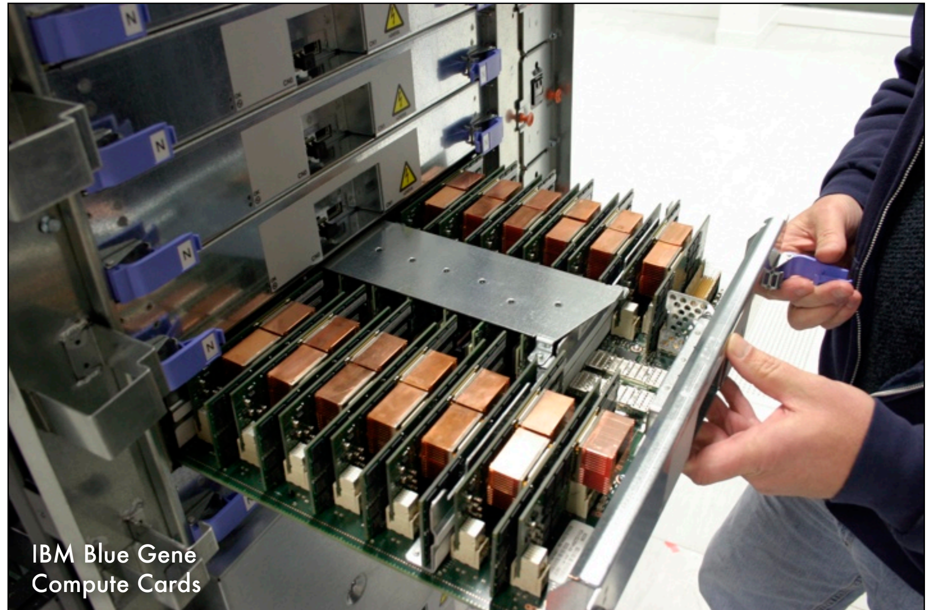
ICHEC

TECHNICAL REPORT

Alastair McKinstry
ICHEC Computational Scientist


IBM Blue Gene Compute Cards

Building & Porting :: software engineering for scientists

Computational scientists often end up managing large bodies of code. While programming is typically on the curriculum, software engineering and how to manage software projects is not. In this document we describe some software engineering methods that scientists should find useful.

Building Code

Any non-trivial software is typically built using make files or build scripts. These then ship with the program. When writing build scripts, try to stick to the mainstream, GNU conventions. That is, to build the program you should be able to just run:

```
$ ./configure
$ make
$ make install
```

./configure sets up the build system; it can take options such as –prefix to declare where the program is to be installed. ./configure –help should explain all the options. make then builds the program. make install will install the program (this may require root or administrator privileges, depending on where it is installed).

There are two good reasons for sticking to this convention: it makes it easier for people to use, as they are likely to know the tools and secondly people build toolsets around these existing methods: colorised editors and IDEs that call make, etc.

Avoid adding extra shell scripts to the build process. This makes it harder to debug and also harder to parallelize, as shown later.

The build utility make is available on most platforms, but older Unixes often come with their own proprietary version. Today, the GNU Make (often called gmake) is the most feature-full and usable; it contains many features that render additional helper shell scripts unnecessary, and it is worth getting the GNU Make manual and learning them.

building & porting

Building code	1
Testing & regression	2
Conditional code	2
Compiler problems	3

Non-Recursive Makefiles

In large projects, different sections of the code are often placed in different subdirectories for easy management. Typically, makefiles are then included in each subdirectory, called from the top-level makefile. This recursive nature loses a lot of dependency information and makes it hard to parallelize the build; to avoid this non-recursive makefiles are recommended.
see: http://www.xs4all.nl/~evbergen/nonrecursive-make.html

makefile

```
all_tests: test1 test2 test2

test1:  test1.orig test1.out
      @(diff test1.orig test1.out > /dev/null) || echo "Test1 Failed"
```

diff returns a non-zero exit code when two files differ. You can use this in make scripts:

Testing and Regression Tests

During programming you will end up testing your program on small datasets to ensure it works as expected. You should aim to develop your program so that you can easily test it, and keep copies of the tests available for future use.

You should create a tests directory within your project and keep all the tests you have used within it. Include a makefile with a top-level target make check that can check if your code is working as expected. This is useful firstly as you develop, but also for porters and other developers to check that they haven't broken anything. It also gives practical examples of how the program was supposed to work!

A good practice in this regard is to make it possible to run a code in batch mode, taking input from a file and outputting to a file. The first time round you can visually inspect the output to ensure it is correct. Next, keep a copy of the correct output, and you can check that the code has not *regressed* later by re-running the test and ensuring the output remains the same.

If you keep the output in a simple style, without putting datestamps, etc. in it, you can then compare two sets of output using the Unix diff command.

For more complex numerical output, we may need to check an array is within a small delta rather than exactly identical to an original output. Again keep the output format simple, but parse the output in python script, etc. and let it return a TRUE/FALSE if the test script, and signal an error using the exit code with sys.exit(1).

Porting to another platform

Porting is the task of taking a program that works on one platform and making it work on another. This involves compiler changes, but also potentially adding new code to make up for missing features, or using new libraries available on the new platform, etc.

Conditional Code

In porting we often have to add code that is included differently depending on the platform we are building on. A recommended approach is to place

```
#include "config.h"
```

at the top of C/C++ files, or include Fortran modules, etc. This can then include complex compile-time definitions.

When placing conditional code for different systems, name the conditional after the feature, not the platform. e.g., if building for a Cray, with a missing function srch() that is present on other platforms, then do:

```
#if !defined(HAS_DSRCH)
   ! Add dummy subroutine
   subroutine d_srch
   end
#endif
```

rather than:

```
#ifdef CRAY
   subroutine d_srch
   end
#endif
```

This makes life easier for the next porter, and makes the code easier to understand; you can understand why options are chosen, and re-using them is not an issue. If you had used 'ifdef CRAY' when including 64-bit code on a Cray machine, for example, then various 64-bit SGI and IBM systems would have 'CRAY' defined; you would also have to check the code thoroughly to be sure that the 'CRAY' conditional did not have other effects.

Additionally, remember that missing code may be added in a later release. Using this approach, all that needs to be changed for a new platform is the build system, not the code itself.

Useful Make Idioms

`make -j <n>`

Build in parallel, using up to n processors.

`gcc -M <file>`

Generate a list of dependencies from a source file, This is useful for building Make scripts. This can be done automatically. Often such generated dependencies are placed in a file 'make.dep' which is then included in the makefile using 'include make.dep'. This is less error-prone than including the dependencies manually.

Compiler problems

All (large) software projects have bugs. This includes compilers and other tools used to build your code. Typically these come in the form of optimization errors, where your code fails to compile, or produces bad output, when built with certain compiler flags. For example, building all code with 'gcc -O4 -fast' will lead to problems.

To fix this, we typically want to build some modules with different compiler flags to other modules. We can do this in a makefile as shown:

Note that:

#1 We include a special rule for dapple.c, not using the standard $(CFLAGS), etc.

#2 We log the bug, and include the bug number! We should always report bugs as we find them. If possible, include a URL to the bug report.

#3 Always comment 'strange code' to explain why it exists and why we're making this exception. Later, if the compiler bug is fixed, we can track down and remove this unnecessary code; otherwise your program will become littered with obsolete cruft as time passes.

```
# gcc bug #11234: dapple.c fails to compile
# with normal -O3 flags
dapple.o: dapple.c
	$(CC) -c -O2 -fno-fast-math dapple.c
```

If this only happens on one platform, we will only want to include this rule on that platform. To do this, we can include files in our makefile.

Here we include separate makefile fragments in subdirectories if they exist. The '-' sign before the include means that the make is not to fail if the included file does not exist: in each directory there may be a file Makefile.in.$(ARCH), which is included if it exists. The dapple.o fragment above may be in that file, for example.

```
# root source of make setup environment
MKROOT    := $(XMAKE)

# general architecture specific defaults
MKARCH    := $(MKROOT)/arch/Makefile.in.$(BUILDARCH)
include $(MKARCH)

# local setup locations and files
MKLOC     := loc

# experiment specific defaults for architecture
MKSETUP   := $(DIR)/Makefile.setup.$(BUILDARCH).$(TASK)
-include $(MKSETUP)
```