Partnership for Advanced Computing in Europe

Introduction to HPC Programming

4. C and FORTRAN compilers; `make`, `configure`, `cmake`

Valentin Pavlov <vpavlov@rila.bg>

About these lectures

- This is the fourth of series of six introductory lectures discussing the field of High-Performance Computing;
- The intended audience of the lectures are high-school students with some programming experience (preferrably using the C programming language) having interests in scientific studies, e.g. physics, chemistry, biology, etc.
- This lecture provides an overview of the C and FORTRAN compilers and the build facilities used to construct complex software packages.
- Some of the slides provide only pointers for the lecturer (and the readers) to elaborate on since its not feasible to describe all features here!

Introduction

- 99.99% of the supercomputing software is written in 2 programming languages: C/C++[1] and FORTRAN;
- This is due to the fact that their abstraction level is ideal for this type of software: on one hand its low enough so performance-optimized code can be produces; on the other hand its high enough so scientists can express their ieas easy, without paying too much attention to low-level details.
- Recently modern dynamic languages started to make their appearence in the supercomputing field, mostly used for fast prototyping (e.g. python, lisp, etc.)

[1] C++ is an extension to C introducing Object-oriented programming paradigm in the language

Workflow process

- The process for creating a C/FORTRAN program includes:
- Figuring out the text of the program and writing it in one *or more* text files called **the source code**;
- Every source file is subjected to **pre-processing**, in which some text fragments are replaced by other text fragments;
- The result of the pre-processed source code is **compiled** and the result is stored in the so called **object code**. To each source file corresponds one object file.

Workflow process

- The mulitude of object files, produced by the previous step and which form the body of the program, are **linked** together after which the final **executable** or **library** is produced.
- The executable file is run (or in the case of batch system—submited for execution)
- There are two types of libraries: static libraries (`.a`) which are collections of functions and sub-programs that can later be included in one or more executables; and shared libraries (`.so`)

Shared libraries

- The shared libraries are basically the same as static ones—collections of functions and sub-programs, but the system (and ultimately the RAM) contains a single copy of the library.
- During linking they do not become part of the executable files; instead they are dynamically loaded into RAM when the program(s) that use it are run. This saves **a lot** of disk and memory space.
- On Blue Gene however this is not much of a benefit, since the memory of the CN contains just a single process, so RAM cannot be saved this case.

gcc

- The most important compiler in Linux land is `gcc`—GNU Compiler Collection;
- `gcc` can compile C, C++, Objective C, FORTRAN, Ada, Java and so on languages on many different platforms and also supports cross-compilation.
- Depending on the *name* of the command that is used to run it, it can do different things: when called as `gcc`, it expects its input files to be C programs; when called as `g++`, it expects C++ programs, and so on.
- It can work as a compiler or as a linker (called as `ld`, part of `binutils`);

Other compilers

- There are other important compilers for some of the supercomputers;
- The IBM XL suite of compilers (`xlc, xlf`) are important for IBM Blue Gene, since they contain optimized code generation for this machine;
- Intel C compiler `icc` is important for creating programs that utilize Intel Xeon Phi and is the only compiler now that does so;

gcc — basic options

- `gcc -E` runs just the pre-processor; the resulting text is printed on the standard output;
- `gcc -c` runs the pre-processor and compiler on the source file; produces an object file as a result;
- Example: `gcc -c hello.c` compiles hello.c and creates an object file called hello.o;
- `gcc -S` creates an output file that contains the assembly language translation of the C code;
- When none of the above options are provided, `gcc` also runs the linker: `gcc hello1.o hello2.o -o hello` links the two object files and generates the executable hello;

gcc — more options

- Pre-processor options: `-I/usr/include -DSYMBOL -USYM2`
- Linker options: `-L/usr/lib -lm -lpthread -rdynamic -static -Wl,option`
- Optimization options: `-g -O0 -O3`
- The full list contains several hundred options — use `man gcc` and `info gcc` when needed.

Some FORTRAN specifics

- Apart from the things discussed above, it also produces so called *modules* (.mod files), which act somehow as libraries;
- Some of the versions (e.g. 77) does not do pre-processing;
- Great many part of the options are the same;
- FORTRAN and C code can be linked together, at least in principle; however, expect problems with different internal naming of functions and subroutines (so called Name Mangling Scheme);

Compulers on Blue Gene

- Blue Gene (both P and Q) come with two sets of compilers; GCC and IBM XL;
- Each of the two sets have a variant that compiles for the FEN, and a variant that compiles for the CN (although CNs are the main work units, sometimes there are programs that need to be run on the FEN also);
- Compilers for the FEN use the usual commands: `gcc`, `g++`, `gfortran`, `xlc`, `xlf`;
- The compilers for the CN are: `powerpc-bgp-gcc`, etc, as well as `bgxlc`, `bgxlf`

Compulers on Blue Gene

- However, when targeting the CNs, usually the so called MPI compiler wrappers are used: `mpicc`, `mpicxx`, `mpif77`, `mpif95`, `mpixlc`, `mpixlf`, etc.
- These help reduce the hassle of dealing with various options by automatically including all the options necessary to bring in the MPI include files and libraries that are needed in order to work in distributed memory scheme.

`make`

- When writing complex software it is not feasable to keep all your sorce code in a single file; for example, the GROMACS mollecular dynamic package contains above 1.6 million lines of code. Obviously it has to be distributed into many files in order for its programmers to remain sane.
- The task of compiling them all, linking them, maintaining relationship and dependencies between them (which comes after which, when do we need to compile each file, etc.) is not trivial and a hard problem by itself;
- Luckily, there are tools to help solving it and the most commonly used one in Linux and UNIX is `make` (and more specifically – its GNU version, `gmake`);

Makefile

- It allows you to organize your source files in a kind of “project”, which can be compiled fairly easy;
- The instructions for the `make` utility are written in a file called `Makefile`
- A `Makefile` constists of *targets* — descriptions of work units;
- These descriptions have two parts — dependencies and commands to carry out the work;

Makefile

- The dependencies of a target lists other files and targets on which this one depends.
- Before issuing the commands that do the work of the target, `make` makes sure that all dependencies are fulfilled — dependend targets and files must be build beforehand.
- On the other hand, if dependent targets and files are already built and their sources not modified after that, this target need not be modified also.
- This saves **a lot** of time when re-building a project since only the newly changed parts of it have to be rebuild.

Makefile

- If all dependencies are met, `make` carries out the commands listed in order to build the target.
- Naming of the targets is arbitrary and depends on the project, but several targets are widely used:
- `all` — the default target to build if none supplied when calling `make`;
- `test`, `install`, `clean` — to perform tests on the built software, to perform installation and to clean up all produced artefacts (in order to start from the beginning);
- There are special targets like `.c.o`, which is used in all cases when an object file has to be produced from a C source file, etc.

autoconf/automake

- In a complex project the creation of Makefile itself can be a serious challange.
- Moreover, when the software package has to be made portable, so as to run on different hardware platforms, things can easily get out of hand, since each platform needs its own Makefile (or a large spaghetti one);
- autoconf and automake are utilities which can help to create complex Makefiles;
- Without digging too deep in what they do and how they do it, we will say that the result they produce is the so called `configure` script, which usually is distributed along with the software source code.

configure

- The `configure` script allows the user to choose a specific combination of options for compiling the software;
- It [usually] can automatically determine the specifics of the operating system its run on and pick up suitable values for many of the options;
- It certainly needs help sometimes, but it truly helps for creating portable software that can run on many different platforms.
- As a result of running the `configure` script, it automatically creates a Makefile suitable for the corresponding platform and having all necessary options, targets, etc. set.

Commonly used `configure` options

- `--prefix` — for chosing the location in which the compiled software to be installed;
- `--enable-XXX` — enables some XXX functionality/feature in the compiled software;
- `--with-XXX` — basically the same, but allows chosing some specifics in regards to the feature (e.g. one of several options);
- `--disable-XXX`, `--without-XXX` — the opposite of the above two options;
- Environment variables: `CC`, `CFLAGS`, `CXX`, `CXXFLAGS`, `FC`, `FFLAGS`, `LD`, `LDFLAGS`: usually written *before* the `configure` command;

`cmake` — `configure` alternative

- In the last several years another system for automatic configuration and Makefile creation became popular – `cmake`;
- It basically does the same job — autodetects the platform, sets some of the options, allows the user to set others and generates a Makefile at the end.
- It has the advantage of offering more user-friendly interface for option selection, including command line mode, interactive mode, textual dialog-based mode and GUI;
- Some software packages use `configure`, other use `cmake` and there are some that use both.