Stefano Cozzini

CRS – OGS Udine – 26 Novembre 2019

Programming languages and compilers for scientific computing

Aims&Goals

- AIM: examine how to go from a piece of source code to a running application
- Discuss:
  - which programming languages are available (briefly)
  - what is a compiler and how it works
  - different stages involved in the process of compiling
  - how a program actually runs
  - A bunch of useful commands/options when dealing with code
- Motivation: the interaction between application and the system is often poorly understood but a greater knowledge can be helpful to efficient software development

Programming languages

- choice of language for a given task is often a thorny issue.
- For us: a programming language is only a tool for writing scientific code.
- The computer language that you use will hopefully be the one that best facilitates this task.
- Many differences between high level language which may be viewed as advantages or disadvantages depending on the task you are trying to solve.

Interpreted languages

- An interpreter is a program which itself executes other programs.

  - JavaScript, Python and awk

- Advantages:
  - it can be quicker to run the code under the interpreter than compile and run it with a compiler.
  - code is easier to debug (interpreter will analyze each statement in the code each time it is executed)
- Where to use interpreted languages:
  - for small applications prototyping and testing of code when an edit-interpret-debug cycle can often be much quicker than an edit- compile-run-debug cycle.
- NOT a good idea to perform numerically intensive calculations using interpreted languages.

Python: “batteries included”

- Suitable for:
  - Text processing
  - Data pre/post processing
  - small/large tasks
- Built-in comprehensive library of functions (“ the python way”)
- Easy to write maintainable code

Python for scientific computing

- Easy to prototype
- Easy to manage complex workflow
- A plethora of scientific libraries
  - numpy/scipy/matplotlib/scikit-learn
- Easy to run intensive computational jobs if right highly optimized libraries below
- Easy to write ugly and unefficient code

compiled languages

- Needed to perform numerically intensive calculations
- Run Time >> Compiling/debugging time
- Examples:
  - Fortran
  - C
  - C++
  - ~~Java~~
- Others ?

Fortran

- Still the main language within the scientific community
- It is likely to hold this position for a long time.
- Why?
  - Some issues within C that make the language inherently more difficult to compile and produce good optimization (mainly dynamical d-referencing of pointers).
  - Tons of libraries written in Fortran
  - Tons of computational codes written in F77
  - laziness of users

C for scientific computing?

- C works well in many domains: graphics, I/O, O.S. world
- C does not natively have library and tools for scientific computing
- Limits (?) in numerical computation:
  - F90 array notation is missing
  - Tons of numerical software is written in F77/90

C++

- C++ was initially an extension to C to incorporate full object-oriented(OO)programming techniques.
- C can be regarded as a subset of C++, so a C++ compiler will (hopefully) be able to compile a C code to achieve the same performance.
- The main design aim of C++ is to design and build large applications using OO techniques.
- However performance can drop if you use OO without care..

Why Compilers?

- Compiler
  - A program that translates from one language to another
  - It should create an efficient version of the target language
  - Your best friend !
- In the beginning, there was machine language
  - Ugly – writing code, debugging
  - Then came textual assembly – still used on some devices..
  - High-level languages – Fortran, Pascal, C, C++
  - Machine structures became too complex and software management too difficult to continue with low-level languages

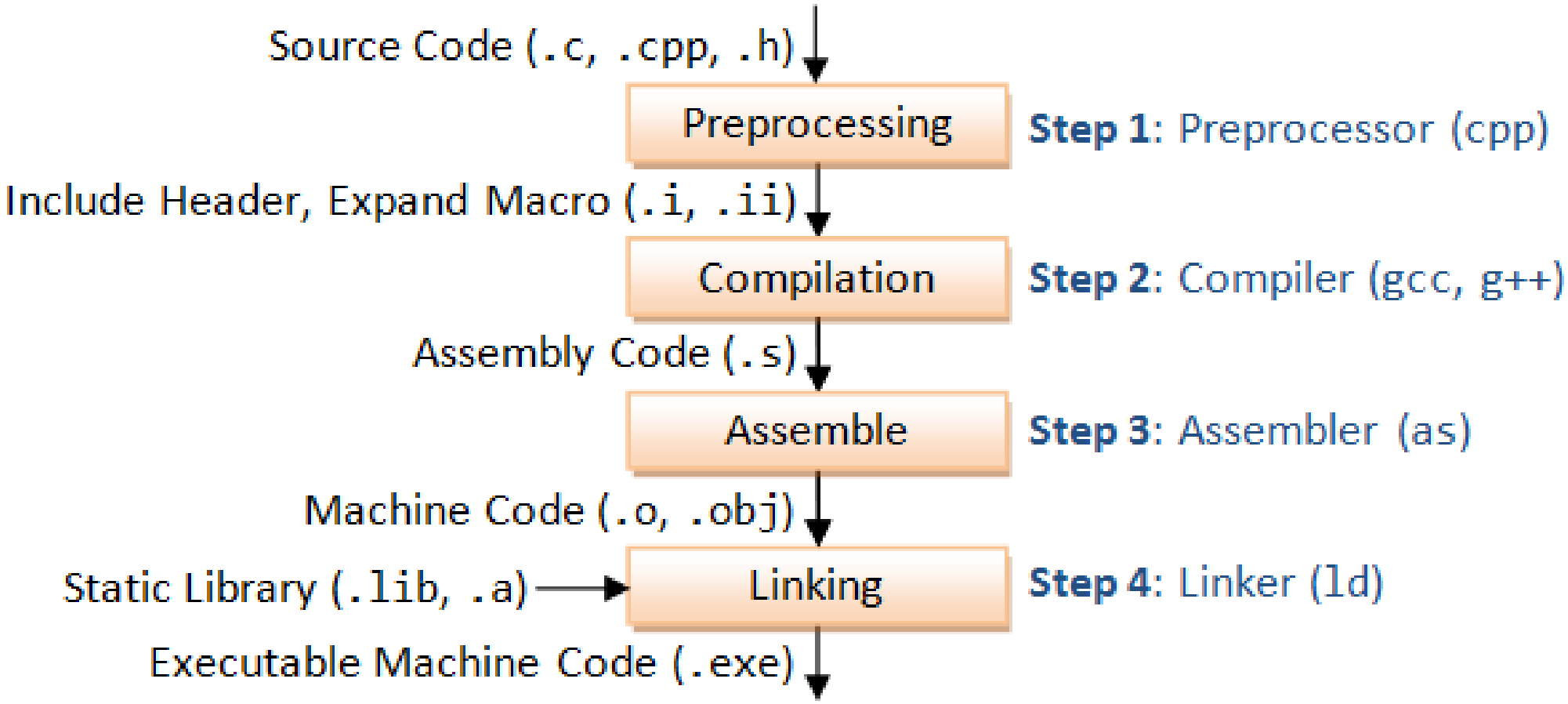
Compilers for Linux

- Free/Open Source:
  - GNU http://www.gnu.org/ (Fortran 77, C, C++, …)
- Commercial:
  - PGI (Fortran 77, Fortran 90, C, C++)
  - Intel (Fortran 77/95, C/C++)
  - And many other...
- Almost all allow you a 30 day evaluation license

Knowing your compiler

- Calling a compiler you invoke a driver program that hides the different compilation stages.
- You can use compiler flags to show all the in-between stages and/or output intermediate results from any of these stages.
- These flags tends to be compiler dependent

Compilation steps..



A word of caution...

- The terminology here might be slightly confusing as we are using the term compiler in two ways
  - the compilation process is what will take source code and produce executable machine code.
  - In the diagram above we use "compiler" to encompass just a step of the procedure:
  - actually this is composed by following four stages:
    - 1. lexical analysis;
    - 2. syntax and
    - 3.semantic analysis;
    - 4. Intermediate Level Code (ILC) generation.

Using a preprocessor

- C files are automatically pre-processed before they are passed to the front end of the compiler.
- You can output preprocessed files (*.i suffix) using -P -E
- To preprocess Fortran files use *.F (fixed) or *.F90 (free) extensions for the source files.
- Preprocessing can be done explicitly by cpp, or fpp or using embedded preprocessor that comes with the F90 compiler itself.

```
cc -E demo.c
```

Using a preprocessor (2)

- Typically preprocessors perform mainly text based manipulations.
- Preprocessor commands (known as "pragmas" in C) always begin with #, (#include, #define, #ifdef)
- It is possible to use the same preprocessor for Fortran programs and the # must be located in the first character position.
- These commands instruct the preprocessor to:
  - include external (header) files conditionally
  - enable source code compilation
  - perform textual substitution (expansion on Macro/embedding constant)

conditional compilation (an example)

- to comment out sections of code, possibly machine specific:

```
#ifdef X86_64
 /* 64 bit architecture specific code */
 #endif
#ifdef DEBUG
 /* enable extra printout and/or extra checks/control
#endif
```

- The code will be compiled: at compile time by adding #define DEBUG at the top of the relevant source file or in a generic header file
- from the command line:

```
cc -DDEBUG -i demo.c
```

Preprocessing: final considerations

- Great tools: very useful (portability)
- However: for the sake of clarity, do not overuse #ifdef.
- TIP: If you find that you have a lot of different options it might be better to separate the source code into separate files and then use **make** to perform the required compilation.

Compiler stages

- The front end stages:
  - several front end (each for any language)
  - each of one produce the ILC (portable)
- the back-end stage (as)
  - produce machine-specific assembler code then to re-locatable object code.
- the linker stage: (ld)
  - link together all the pieces to produce the executable

The front-end: actions

- first: The parser, syntax and semantic analysis removes unnecessary white spaces and any remaining comments.
- second: The source code is split into "tokens"
- third: syntax checker makes sure that each is a valid construct (The majority of the errors that are detectable by the compiler are caught here)
- fourth: The code generator produces the ILC output.
- fifth: the The code optimiser attempts to optimise the ILC

The assembler

- the assembler creates object files from assembly language source files (as)
- The assembler code is then converted to relocatable object code by the assembler. (A relocatable object file can be loaded starting at any location in memory)
- It is then added to all the addresses in the object file, so the object file could be loaded into any location in memory by the Unix operating system.)

```
cc -c -g demo.c
 nm demo.o
```

Compilation phase compiler flags..

- Try this:

```
cc -S demo.c
less demo.s
```

- check man pages to see which flags are available and what they do:
  - enforcing strict syntax checking
  - looking for un-used variables etc.
  - Looking for optimization flags

Optimization

- How to make the code go faster
- Classical optimizations
  - Dead code elimination – remove useless code
  - Common subexpression elimination – recomputing the same thing multiple times
- Machine independent (classical)
  - Useful for almost all architectures
- Machine dependent
  - Depends on processor architecture
  - Memory system, branches, dependences

Optimization (2)

- Compilers do many different transformations to produce fast code
- Some of them could be controlled by flags on the command line
- This is not always straightforward (complex inter-dependencies)
- To increase performance play with flags.. (learn a lot about that next lecture...)
- Check out results are still correct.

Symbol Table

- produced by compiler and used by the linker to find the information required to build the whole code.
- Like a dictionary that records each identifier or keyword found:
  - the type (variable, array, procedure, . . .)
  - the data type (integer,real, . . .)
  - the run-time address pointer to access more information (like the bounds of an array, . . .
- The symbol table is very important for debugging.
- Debuggers use a more complete symbol table with every variable listed and references to the source lines where they are modified.
- This is generally produced with the -g flag.

the linker

- All the different object files are finally glued together by the linker.
- to know about it : man ld (very system specific)
- Some of Actions it performs:
  - identifies the main routine as the initial entry point when execution begins.
  - resolves subroutine and identifies function calls by putting in the correct addresses
  - If there are any unresolved symbols it will then try to link in any external libraries which have been specified and default ones from the system.
- The result is an executable file:

a(ssembler).out(put)

Linker (2)

- Error messages are printed if there are remaining unrsolved symbols.

```
 cozzini@elcid 1_Compiled_languages]$ cc demo.c
/tmp/ccy4sMiS.o: In function `main':
demo.c:(.text+0x19): undefined reference to `sin'
collect2: ld returned 1 exit status
```

- Solution: add the maths library at the end of the compilation process:

```
cozzini@elcid 1_Compiled_languages]$ cc demo.c -lm
```

Where are the libs?

- Standard places are searched if you use standard libraries [/usr/lib /usr/local/lib/ ..]
- check out the /etc/ld.so.conf, LD_LIBRARY_PATH
  - define where to search for shared libraries
- Otherwise: explicitly specify the path to the library and the name of that library: -L/Path_to_library -lmpi
- this refers to a library file called:
  - /Path_to_library/libmpi.s (shared) o (bject) --> (dynamic library)
  - /Path_to_library/libmpi.a(rchive) [see ar command] --> (static library)

Static vs Dynamic libraries...

- Dynamic libraries the external reference will not be resolved until the code starts running and even then the linking will not take effect until an actual call is made to the routine requiring that library.
- Static libraries the actual code to execute the external routine will be physically copied into your executable. This is the way that linking used to always be done.
- There are advantages/disadvantages to both methods

Static libraries

Pro:

- Code is more portable/relocatable
- It should be more efficient in term of performance

Issues:

- Symbols are resolved “from left to right”, so circular dependencies require to list libraries multiple times or use a special linker flag
- Executable are larger (very minor issue)
- Not always available by default (check your system please)

NOTE: When linking only the name of the symbol is checked, not whether its argument list matches

Shared Libraries

- Pro:
  - Executable is smaller
  - Standard and easy way to go
- Cons:
  - Not always easy to find out the right combination (see next slide)
  - That could be a penalty in performance

Please name your executable!

- By default all compiler will produce an executable named a.out.
- You can use a -o flag (standard for all the compiler) to generate something more meaningful ..

```
azorka~ 13>gfortran -o my_code.x  my_code.f90
```

How to choose a compiler for scientific computing?

- Efficiency
  - Does it produce efficient code?
  - Does it produce correct code?
  - Is it able to exploit the hardware?
- Interoperability
  - Does it operate with other tools/compiler/languages?
- Utilities / Tools
  - Does it have a Debugger/ Profiler / other utilities?
- Diagnostic Capabilities
  - Is it able to detected errors/bugs in programs?
- Documentation/ support /training/cost

Gnu compiler collection

- The Cross-Platform compiler package
- Supports many OS/CPU combinations
- Already bundled with Linux distributions
- Support for C/C++/F90 good
- OpenMP Support
- Debugger, several GUI frontends
- Profiler, GUI frontends
- Many additional, supporting tools available