Compiling scientific codes: Basic concepts and tools

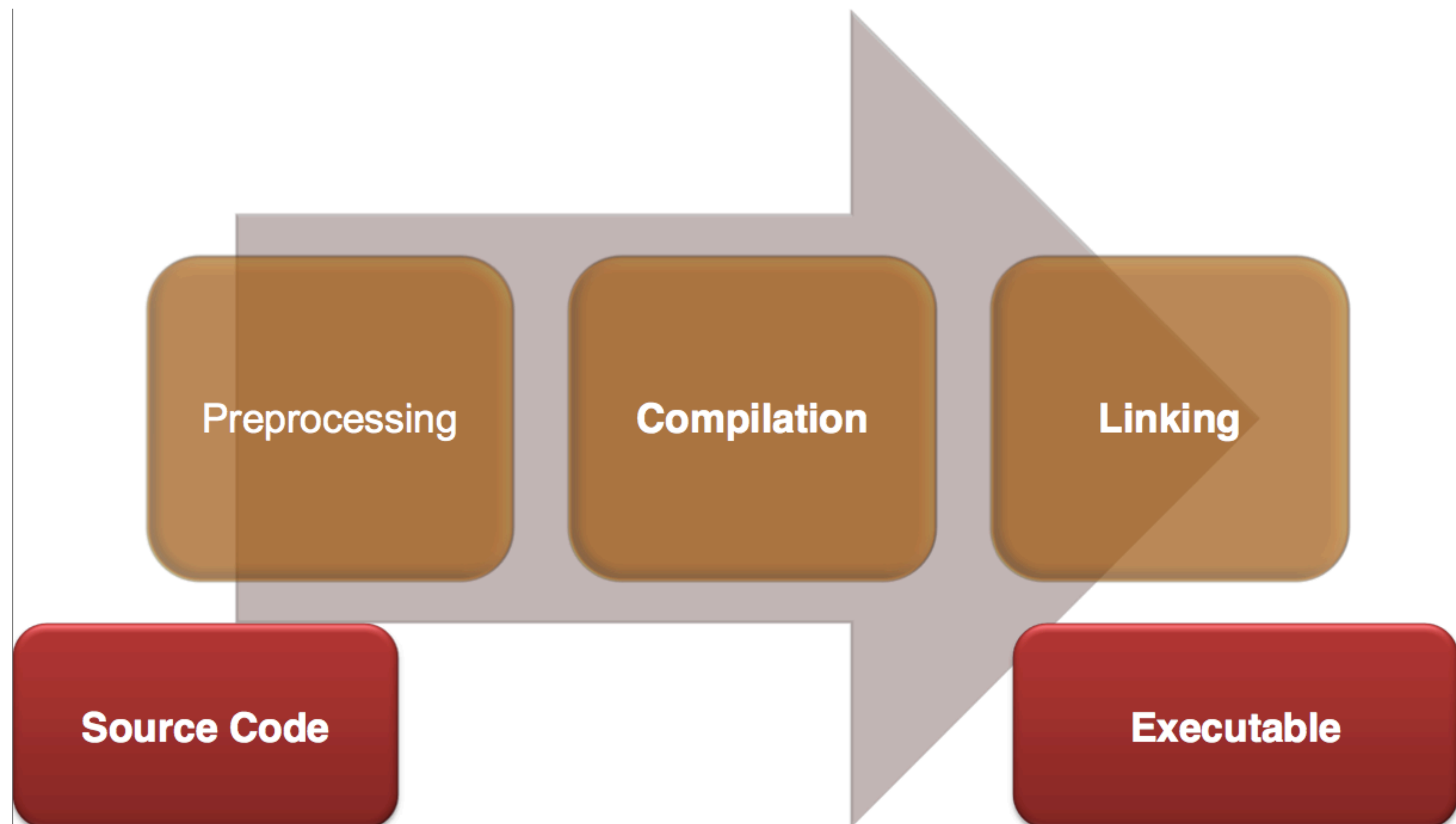
Antun Balaž
Scientific Computing Laboratory
Institute of Physics Belgrade
http://www.scl.rs/

21 Feb 2012

Overview

- Compiling
  - The preprocess / compile / linking process
    - Individual steps in detail
  - Preprocessing in C and Fortran
  - The C-preprocessor, typical directives
  - Compilers and Vendors
  - Compiler Flags
- Makefiles
  - Concept
  - Syntax
  - Rules, pattern rules, special rules
  - Conventions

How it works



Example of the process

- Consider the minimal C program, hello.c:

```
#include <stdio.h>
int main (int argc, char **argv) {
    printf(“hello world\n”);
    return 0;
}
```

- What happens if we do:

```
cc –o hello hello.c
```

Scientific Computing Laboratory

Scientific Computing Laboratory

21 Feb 2012

Step 1: Preprocessing

- Handles all line in source code with „#" directives
  - File inclusion
  - Conditional compilation
  - Macro expansion
- In our simple example:
  - #include <stdio.h>

  This translates to „insert file /usr/include/stdio.h" into my source code.
- If you would like to see the pre-processed source:
  - cc –E hello.c –o hello.pp.c

Preprocessing, what is it good for?

- Selective compilation:

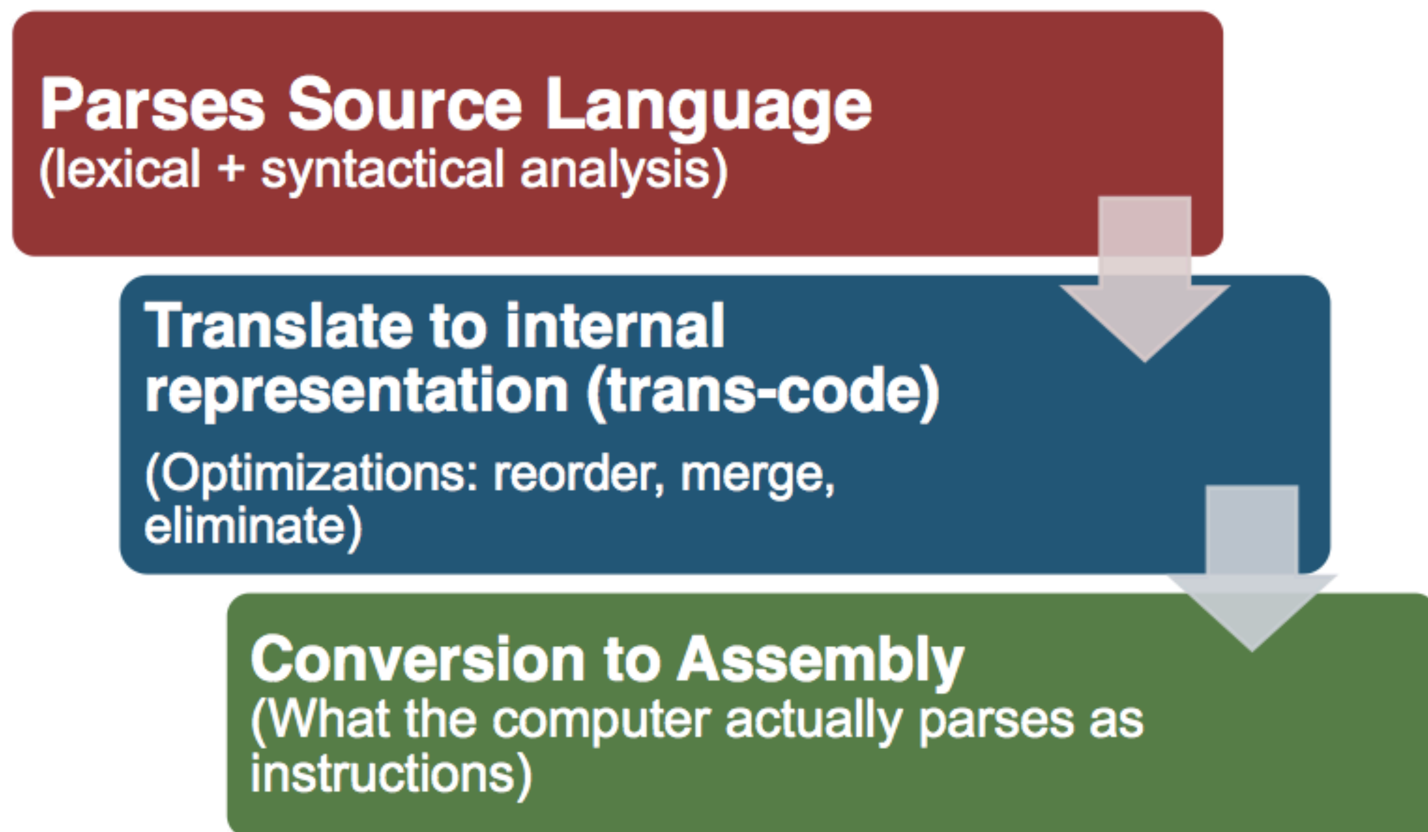
```
#include <stdio.h>
int main (int argc, char **argv) {

#ifdef LINUX
      printf("Hello world from Linux!\n");
#else
      printf("Hello world from something else!\n");
#endif

      return 0;
}
```

- cc –DLINUX hello.c –o hello
  - prints “Hello world from Linux!”

Step 2: Compilation



SCIENTIFIC COMPUTING LABORATORY

21 Feb 2012

Assembly language

- cc –S hello.c
  produces hello.s

```
.LC0:
        .string "hello world\n"
        .text
.globl main
        .type   main,@function
main:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $8, %esp
        andl    $-16, %esp
        movl    $0, %eax
        subl    %eax, %esp
        subl    $12, %esp
        pushl   $.LC0
        call    printf
        addl    $16, %esp
        movl    $0, %eax
        leave
        ret
```

Assembler

- Assembler (as) translates assembly to binary
- Creates the so-called object files
  - cc –c hello.c

    produces hello.o

Step 3: Linking

- The Linker (ld) puts it all together
- It adds startup code and library code to binary for creation of final executable.
  - ld –o hello hello.o
  - ./hello

    prints “hello world”

Adding libraries

- Libraries are a powerful tool to give programmers access to optimized or highly used functions
- libmath example
  exp(double) is a function provided by libmath

```
#include <math.h>
#include <stdio.h>
int main(int argc, char **argv) {
    printf("exp(2.0)=%f\n", exp(2.0));
    return 0;
}
```

- To compile: cc –o hello hello.c -lm

Practical Compiling Issues

- Preprocessing in C and Fortran
  - C/C++ mandatory
  - Optional in Fortran
    - Often implicit via file name: name.F, name.F90, name.FOR
- Can set define variables on the command line
  - -DDEF_ARR=200
  - Use capital letters to signal a define

C Pre-processor directives

- #define MYVAL 100
- #undefMYVAL
- #if defined(MYVAL) && defined(__LINUX)
- #elif(MYVAL < 200)
- #else
- #endif
- #include “myfile.h”
- #include <mysysfile.h>

SCIENTIFIC COMPUTING LABORATORY

Compilers

- GNU Compiler Collection
- Intel Compilers
- Portland Group Compilers
- IBM XL Compilers

GNU Compilers Collection

- GNU Compiler Collection or GCC is standard compiler on most modern Unix-like computer operating systems
- Includes front ends for compiling:
  - C
  - C++
  - Objective-C
  - Fortran
  - Java
  - Ada
  - Go

GNU Compilers Collection

- Ported to a wide variety of processor architectures
- Widely deployed as a tool in commercial, proprietary and closed source software development environments
- Performs well on a variety of native and cross targets
- Available for most embedded platforms
- http://gcc.gnu.org/

Intel Compilers

- Intel's suite of compilers with front ends for C, C++, and Fortran:
  - C, C++ - icc, icpc
  - FORTRAN – ifort
- Compilers are distributed through different package suites:
  - Intel Parallel Studio, Intel Parallel Studio XE, the Intel C++ Composer package, the Intel C++ Composer XE package, the Intel Composer XE package and the Intel Cluster Studio
- Compilers available for GNU/Linux, Mac OS and Microsoft Windows systems

Intel Compilers

- Intel tunes its compilers to optimize for its hardware platforms therefore producing highly optimized code for Intel processors
- Different optimization features and multithreading capabilities
  - Automatic vectorizer that can generate SSE, SSE2, SSE3, SSSE3, SSE4 and AVX SIMD instructions,
  - Supports both OpenMP 3.1 and automatic parallelization for symmetric multiprocessing
- http://software.intel.com/en-us/articles/intel-compilers/

PGI Compilers

- The Portland Group, Inc or PGI
- Set of commercially available Fortran, C and C++ compilers for high-performance computing system
- Incorporate:
  - global optimization
  - vectorization
  - software pipelining
  - shared-memory parallelization
  - capabilities targeting both Intel and AMD processors

PGI Compilers

- PGI supports the following high-level languages:
  - Fortran 77, Fortran 95, Fortran 2003
  - High Performance Fortran (HPF)
  - ANSI C99
  - ANSI/ISO C++
- Recently PGI has been involved in the expansion of the use of GPGPUs for high-performance computing, developing CUDA Fortran with NVIDIA Corporation and PGI Accelerator Fortran and C compilers which use programming directives
- http://www.pgroup.com/

IBM XL Compilers

- Proprietary suite of compilers from IBM developed for its platforms:
  - POWER
  - Cell CPUs
  - Blue Gene systems
- Well established in HPC
- Support for C, C++ and FORTRAN
- Producing highly optimizing code for IBM CPUs
- Access to highly-tuned libraries, optimization utilities and development utilities
- Recommended to use with IBM hardware

Using compilers

- GCC compiler toolchain is, by default, available on the users path:
  - gcc, g++
  - gfortran
  - f77, g77
- To use Intel, PGI, IBM or other compilers, some scripts usually have to be sourced (or added to your bash profile)

Important flags (1)

- -o exe_file : names the executable exe_file
- -c : generates the correspondent object file. Does not create an executable.
- -g : compiles in a debugging mode
- -Idir_name : specifies the path where include files are located
- -Ldir_name : specifies the path where libraries are located
- -l<lib_name> : asks to link against the lib<libname>
- -pg: profiling with gprof (needed at the compiling time)
- -Wall : show all reasonable warnings
- -static-intel : link Intel provided libraries statically (icc)

Important flags (2)

- Optimizations:
  - -O0, -O1, -O2, -O3: optimization levels
  - Intel specific:
    - -ip, -ipo: inter-procedural optimizations (mono and multi files)
    - -fast: default high optimization level (-O3 -ipo -static)
    - -opt_report: generates a report which describes the optimization in stderr (-O3 required)
    - -x<code> generate code to run exclusively on processors (SSE4.1, SSE4.2, AVX…)
- Compile with OpenMP:
  - gcc: -fopenmp
  - icc: -openmp

Scientific Computing Laboratory

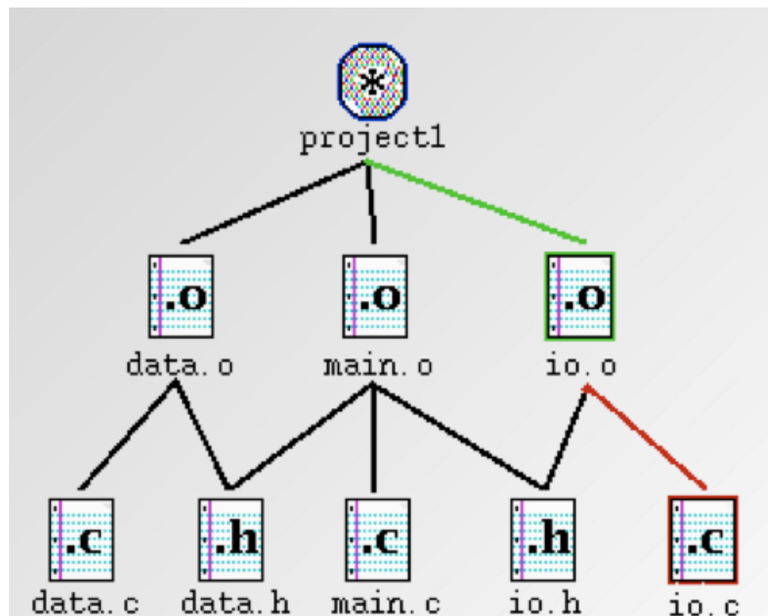
21 Feb 2012

How to find flags for my code?

- Each of these compilers have hundreds of flags
- You will very likely not need to know but a few to do your work and get decently optimized code.
- First place to look is documentation
- The compiler man page should tell you every flag and what it does.

Makefiles

- Concept
  - Simplify building large code projects
  - Speed up re-compile on small changes
  - Consistent build command: make
  - Platform specific configuration via Variable definitions



Makefiles syntax

- Rules

```
target:      prerequisites command
        ^ this must be a 'Tab' (|<- ->|)
```

- Variables:

```
NAME= VALUE1 VALUE2 value3
```

- Comments:

```
# this is a comment
```

- Special keywords:

```
include linux.mk
```

SCIENTIFIC COMPUTING LABORATORY

Rules examples

```
#first target is default:
all:  hello sqrt

hello:   hello.c
         cc -o hello hello.c

sqrt:    sqrt.o
         f77 -o sqrt sqrt.o

sqrt.o:  sqrt.f
         f77 -o sqrt.o -c sqrt.f
```

Variables examples

```
#uncomment as needed
CC= gcc
#CC= icc -i-static
LD=$(CC)
CFLAGS= -O2

hello:    hello.o
          $(LD) -o hello hello.o

hello.o:  hello.c
          $(CC) -c $(CFLAGS) hello.c
```

SCIENTIFIC COMPUTING LABORATORY

Automatic variables

- $@ - the target of the rule
- $< - the first dependency
- $^ - all of the dependencies
- $? - all of the dependencies that are newer than target
- $* - the stem of a pattern matching rule

```
CC= gcc
CFLAGS= -O2
howdy:  hello.o yall.o
        $(CC) -o $@ $^

hello.o:  hello.c
        $(CC) -c $(CFLAGS) $<
yall.o: yall.c $(CC)-c $(CFLAGS) $<
```

Pattern rules

```
OBJECTS=hello1.o hello2.o

howdy:  $(OBJECTS)
        $(CC) -o $@ $^

hello1.o:    hello1.c
hello2.o:    hello2.c

.c.o:
        $(CC) -o $@ -c $(CFLAGS) $<
```

Special targets

```
.SUFFIXES:
.SUFFIXES: .o .c

.PHONY: clean install

.c.o:
        $(CPP) $(CPPFLAGS) $< -o $*.c
        $(CC) -o $@ -c $(FFLAGS) $*.c

clean:
        rm -f *.f *.o
```

Calling make

- Override variables:

make CC=icc CFLAGS='-O2 -unroll’

- Dry run (don't execute):

make -n

- Don't stop at errors (dangerous):

make -i

- Parallel make (requires careful design)

make -j8

- Alternative Makefile

make -f make.pgi

Conventions

- Always set the SHELL variable:

SHELL=/bin/sh

(make creates shell scripts from rules).

- GNU make has many features, that other make programs don't have and vice versa
- Use only a minimal set of Unix commands: cp, ls, ln, rm, mv, mkdir, touch, echo,...
- Implement some standard 'phony' targets: all, clean, install