HPC tools for programming

Victor Eijkhout

2022

Justification

High Performance Computing requires, beyond simple use of a programming language, a number programming tools. These tutorials will introduce you to some of the more important ones.

Intro to file types

File types

Text files	
Source	Program text that you write
Header	also written by you, but not really program text.
Binary files	
Object file	The compiled result of a single source file
Library	Multiple object files bundled together
Executable	Binary file that can be invoked as a command
Data files	Written and read by a program

Text files

- Source files and headers
- You write them: *make sure you master an editor*
- The computer has no idea what these mean.
- They get compiled into programs.

(Also ‘just text’ files: READMEs and such)

Binary files

- Programs. (Also: object and library files.)
- Produced by a compiler.
- Unreadable by you; executable by the computer.

Also binary data files; usually specific to a program.
(Why don’t programs write out their data in readable form?)

Compilation

Compilers

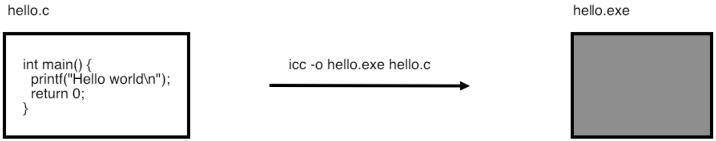
Compilers: a major CS success story.

- The first Fortran compiler (Backus, IBM, 1954): multiple man-years.
- These days: semester project for graduate students.
  Many tools available (`lex`, `yacc`, `clang-tidy`)
  Standard textbooks (‘Dragon book’)
- Compilers are very clever!
  You can be a little more clever in assembly – maybe
  but compiled languages are $10\times$ more productive.

Compilation vs interpreted

- Interpreted languages: lines of code are compiled 'just-in-time'.
  Very flexible, sometimes very slow.
- Compiled languages: code is compiled to machine language:
  less flexible, very fast execution.
- Virtual machine: languages get compiled to an intermediate language
  (Pascal, Python, Java)
  pro: portable; con: does not play nice with other languages.
- Scientific computing languages:
  - Fortran: pretty elegant, great at array manipulation
    Note: Fortran20003 is modern; F77 and F90 are not so great.
  - C: low level, allows great control, tricky to use
  - C++: allows much control, more protection, more tools
    (kinda sucks at arrays)

Simple compilation



```
int main() {
  printf("Hello world\n");
  return 0;
}
```

- From source straight to program.
- Use this only for short programs.

```
%% gcc hello.c
%% ./a.out
hello world
```

```
%% gcc -o helloprog hello.c
%% ./helloprog
hello world
```

Exercise 1, C++ version

Create a file with these contents, and make sure you can compile it:

```
#include <iostream>
using std::cout;

int main() {
  cout << "hello world\n";
  return 0;
}
```

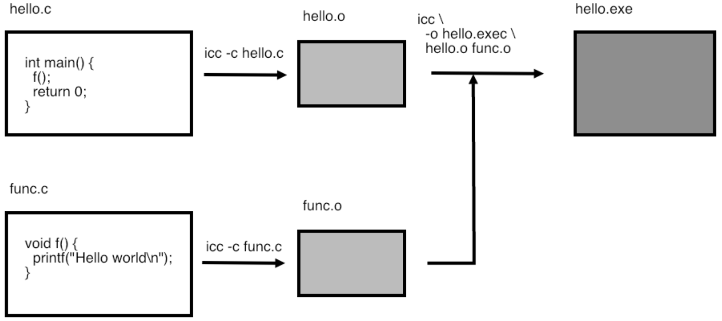
Exercise 1, C version

Create a file with these contents, and make sure you can compile it:

```c
#include <stdlib.h>
#include <stdio.h>

int main() {
  printf("hello world\n");
  return 0;
}
```

Separate compilation



```
int main() {
  f();
  return 0;
}
```

```
void f() {
  printf("Hello world\n");
}
```

- Large programs best broken into small files,
- … and compiled separately (can you guess why?)
- Then 'linked' into a program; linker is usually the same as the compiler.

Exercise 2, C++ version

Make the following files:

Main program: `fooprog.cxx`

```
#include <iostream>
using std::cout;
#include <string>
using std::string;

extern void bar(string);

int main() {
  bar("hello world\n");
  return 0;
}
```

Subprogram: `foosub.cxx`

```
#include <iostream>
using std::cout;
#include <string>
using std::string;

void bar( string s ) {
  cout << s;
}
```

Exercise 2, C version

Make the following files:

Main program: `fooprog.c`

```
#include <stdlib.h>
#include <stdio.h>

extern void bar(char*);

int main() {
  bar("hello world\n");
  return 0;
}
```

Subprogram: `foosub.c`

```
#include <stdlib.h>
#include <stdio.h>

void bar(char *s) {
  printf("%s",s);
  return;
}
```

Exercise 2 continued, C++ version

- Compile in one:

  ```
  icpc -o program fooprog.cxx foosub.cxx
  ```

- Compile in steps:

  ```
  icpc -c fooprog.cxx
  icpc -c foosub.cxx
  icpc -o program fooprog.o foosub.o
  ```

  What files are being produced each time?

Can you write a shell script to automate this?

Exercise 2 continued, C version

- Compile in one:

  ```
  icc -o program fooprog.c foosub.c
  ```

- Compile in steps:

  ```
  icc -c fooprog.c
  icc -c foosub.c
  icc -o program fooprog.o foosub.o
  ```

  What files are being produced each time?

Can you write a shell script to automate this?

Header files

- `extern` is not the best way of dealing with ‘external references’
- Instead, make a header file `foo.h` that only contains

```
void bar(char*);
```

- Include it in both source files:

```
#include "foo.h"
```

- Do the separate compilation calls again.

Now is a good time to learn about makefiles . . .

Compiler options 101

- You have just seen two compiler options.
- Commandlines look like

  ```
  command [ options ] [ argument ]
  ```

  where square brackets mean: ‘optional’
- Some options have an argument

  ```
  icc -o myprogram mysource.c
  ```
- Some options do not.

  ```
  icc -g -o myprogram mysource.c
  ```
- Question: does `-c` have an argument? How can you find out?

  ```
  icc -g -c mysource.c
  ```

Object files

- Object files are unreable. (Try it. How do you normally view files? Which tool sort of works?)
- But you can get some information about them.

```
#include <stdlib.h>
#include <stdio.h>
void bar(char *s) {
  printf("%s",s);
}
```

```
[c:264] nm foosub.o
0000000000000000 T _bar
U _printf
```

Where `T`: stuff defined in this file
`U`: stuff used in this file

Compiler options 102

- Optimization level: `-O0`, `-O1`, `-O2`, `-O3`
  ('I compiled my program with oh-two')
  Higher levels usually give faster code. Level 3 can be unsafe. (Why?)
- `-g` is needed to run your code in a debugger. Always include this.
- The ultimate source is the 'man page' for your compiler.

Compiler optimizations

Common subexpression elimination:

```
x1 = pow(5.2,3.4) * 1;
x2 = pow(5.2,3.4) * 2;
```

becomes

```
t = pow(5.2,3.4);
x1 = t * 1;
x2 = t * 2;
```

Loop invariants lifting

```
for (int i=0; i<1000; i++)
  s += 4*atan(1.0) / i;
```

becomes

```
t = 4*atan(1.0);
for (int i=0; i<1000; i++)
  s += t / i;
```

Example of optimization

Givens program

```
// rotate.cxx
void rotate(double& x,double& y,double alpha) {
  double x0 = x, y0 = y;
  x = cos(alpha) * x0 - sin(alpha) * y0;
  y = sin(alpha) * x0 + cos(alpha) * y0;
  return;
}
```

Run with optimization level 0,1,2,3 we get:

```
Done after 8.649492e-02
Done after 2.650118e-02
Done after 5.869865e-04
Done after 6.787777e-04
```

Exercise 3

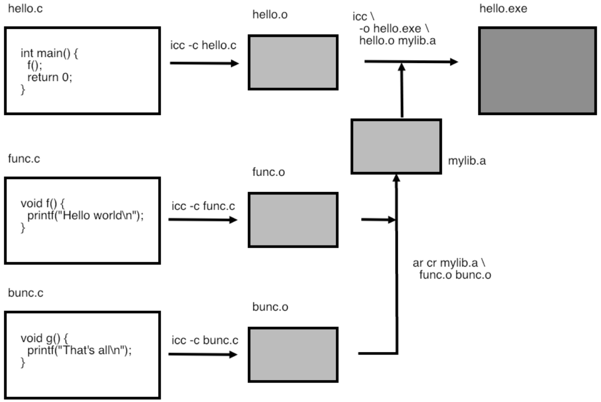
The file `rotate.cxx` (or `rotate.c`) can be speeded up by compiler transformations.
Compile this file with optimization levels `0,1,2,3`
(try both the Intel and gcc compilers)
observe run time and conjecture what transformations can explain this.

Apply these transformations by hand and see if they indeed lead to improvements.

Write a report of your investigations.

Libraries

Libraries



- Sometimes you have many object files: convenient to bundle them
- Easier to link to
- Easy to distribute as a product.
- Software library: collection of object files that can be linked to a main program.

Static / non-shared libraries

- Static libraries are created with `ar`
- Inspect them with `nm`
- Link as object file:

  ```
  icc -o myprogram main.o ../lib/libfoo.a
  ```
- Or:

  ```
  icc -o myprogram main.o -L../lib -lfoo.a
  ```

```
mkdir ../lib
ar cr ../lib/libfoo.a foosub.o
```

```
%% nm ../lib/libfoo.a

../lib/libfoo.a(foosub.o):
00000000 T _bar
         U _printf
```

Static library example

Use `ar` to add object files to `.a` file.

```
icc -g -O2 -std=c99 -c foosub.c
for o in foosub.o ; do \
  ar cr libs/libfoo.a ${o} ; \
done
icc  -o staticprogram fooprog.o -Llibs -lfoo
-rwx------ 1 eijkhout G-25072 38192 Sep 23 18:15 staticprogram
./staticprogram
hello world
```

Dynamic/shared libraries

Created with the compiler,
`-shared` flag.

```
icc -O2 -std=c99 -fPIC -c foosub.c
icc -o libs/libfoo.so -shared foosub.o
icc -o dynamicprogram fooprog.o -Llibs -lfoo
```

Executable size

Static libraries are baked into the executable
shared libraries are linked at runtime.

```
# Making static library
icc  -o staticprogram fooprog.o -Llibs -lfoo
-rwx------ 1 eijkhout G-25072 28232 Sep 23 14:25 staticprogram
# Using dynamic library
icc -o dynamicprogram fooprog.o -Llibs -lfoo
-rwx------ 1 eijkhout G-25072 28160 Sep 23 14:25 dynamicprogram
```

Needs something more

Program can not immediately be run.
Use `ldd` to see what libraries it needs:

```
./dynamicprogram: error while loading shared libraries:
      libfoo.so: cannot open shared object file: No such file or director

ldd dynamicprogram | grep libfoo
        libfoo.so => not found
```

The ell-dee library path

Libraries are found by updating the `LD_LIBRARY_PATH`:

```
export LD_LIBRARY_PATH=${LD_LIBRARY_PATH}:./libs
ldd dynamicprogram | grep libfoo
        libfoo.so => ./libs/libfoo.so (0x00002ad6604c1000)
./libs dynamicprogram
hello world
```

The rpath

You can also bake the path into the program:

```
icc -O2 -std=c99 -fPIC -c foosub.c
icc -o libs/libfoo.so -shared foosub.o
icc -o rpathprogram fooprog.o \
    -Wl,-rpath,./libs -Llibs -lfoo
-rwx------ 1 eijkhout G-25072 28160 Sep 23 13:41 rpathprogram
./rpathprogram
hello world
```

Note

- the bizarre combination of minuses and commas
- you may also come across the `rpath=` syntax, but that's a GNU extension.