HPC tools for programming

Victor Eijkhout

2021

1 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

2 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

3 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)

4 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

5 Compilers

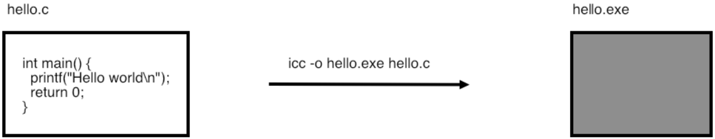
Compilers: a major CS success story.

- 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.

6 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)

7 Simple compilation



- 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
```

8 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;
}
```

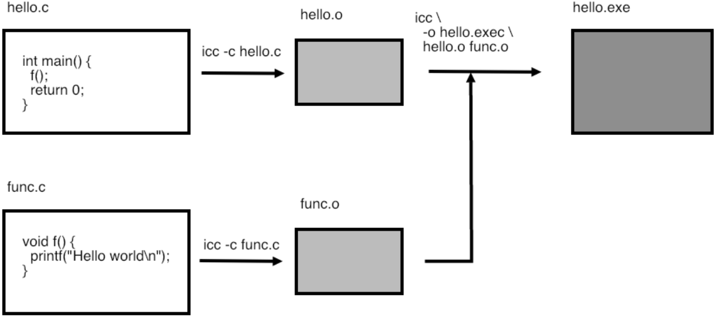
9 Exercise 1, C version

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

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

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

10 Separate compilation



- 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.

11 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;
}
```

12 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;
}
```

13 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?

14 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?

15 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 . . .

16 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
  ```

17 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

18 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.

19 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;
```

20 Example of optimization

Givens program

MISSING SNIPPET givensfunx

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
```

21 Exercise 3

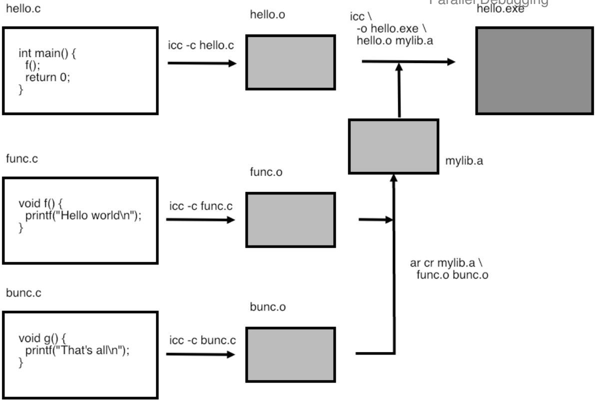
The file `rotate.c` can be speeded up by compiler transformations.
Compile this file with optimization levels `0,1,2,3`
(try both the Intel and gcc compilers)
observe run time and conjecture what transformations can explain this.

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

Write a report of your investigations.

Libraries

22 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.

23 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
```

24 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
```

25 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
```

26 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
```

27 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 directory
```

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

28 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
```

29 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
```

(Notice the bizarre combination of minuses and commas)

Profiling and debugging; optimization and programming strategies.

30 Analysis basics

- Measurements: repeated and controlled
  beware of transients, do you know where your data is?
- Document everything
- Script everything

31 Compiler options

- Defaults are a starting point
- use reporting options: `-opt-report`, `-vec-report`
  useful to check if optimization happened / could not happen
- test numerical correctness before/after optimization change
  (there are options for numerical corretness)

32 Optimization basics

- Use libraries when possible: don't reinvent the wheel
- Premature optimization is the root of all evil (Knuth)

33 Code design for performance

- Keep inner loops simple: no conditionals, function calls, casts
- Avoid small functions: try macros or inlining
- Keep in mind all the cache,TLB, SIMD stuff from before
- SIMD: Fortran array syntax helps

34 Multicore / multithread

- Use `numactl`: prevent process migration
- ‘first touch’ policy: allocate data where it will be used
- Scaling behaviour mostly influenced by bandwidth

35 Multinode performance

- Influenced by load balancing
- Use HPCtoolkit, Scalasca, TAU for plotting
- Explore ‘eager’ limit (mvapich2: environment variables)

36 Classes of programming errors

Logic errors:
functions behave differently from how you thought,
or interact in ways you didn't envision

Hard to debug

37 C

oding errors:
send without receive
forget to allocate buffer

Debuggers can help

Defensive programming

38 Defensive programming

- Keep It Simple (‘restrict expressivity’)
- Example: use collective instead of spelling it out
- easier to write / harder to get wrong
  the library and runtime are likely to be better at optimizing than you

39 Memory management

Beware of memory leaks:
keep allocation and free in same lexical scope

C++ does this automatically with RAII

40 Modular design

Design for debuggability, also easier to optimize

Separation of concerns: try to keep code aspects separate

Premature optimization is the root of all evil (Knuth)

41 MPI performance design

Be aware of latencies: bundle messages
(this may go again separation of concerns)

Consider ‘eager limit’

Process placement, reduction in number of processes

Debugging

42

Debugging is like being the detective in a crime movie where you are also the murderer. (Filipe Fortes, 2013)

What do you do when your program misbehaves?

- Insert print statements, recompile, ru again.
- Run your program in a debugger
- (also: attach a debugger, inspect a core dump)

43 Simple example: listing

tutorials/gdb/c/hello.c

```
#include <stdlib.h>
#include <stdio.h>
int main() {
  printf("hello world\n");
  return 0;
}
```

44 Simple example: running

```
%% cc -g -o hello hello.c
# regular invocation:
%% ./hello
hello world
# invocation from gdb:
%% gdb hello
GNU gdb 6.3.50-20050815 # ..... [version info]
Copyright 2004 Free Software Foundation, Inc. .... [copyright info
(gdb) run
Starting program: /home/eijkhout/tutorials/gdb/hello
Reading symbols for shared libraries +. done
hello world

Program exited normally.
(gdb) quit
%%
```

45 Source listing

```
%% cc -o hello hello.c
%% gdb hello
GNU gdb 6.3.50-20050815 # ..... version info
(gdb) list
```

Important to use the `-g` compile option!

46 Run with arguments

tutorials/gdb/c/say.c

```
#include <stdlib.h>
#include <stdio.h>
int main(int argc,char **argv) {
  int i;
  for (i=0; i<atoi(argv[1]); i++)
    printf("hello world\n");
  return 0;
}
```

```
%% gdb say
.... the usual messages ...
(gdb) run 2
Starting program: /home/eijkhout/tutorials/gdb/c/say 2
Reading symbols for shared libraries +. done
hello world
hello world
```

47 Memory problems 1

```
// square.c
  int nmax,i;
  float *squares,sum;

  fscanf(stdin,"%d",nmax);
  for (i=1; i<=nmax; i++) {
    squares[i] = 1./(i*i); sum += squares[i];
  }
  printf("Sum: %e\n",sum);

%% cc -g -o square square.c
 %% ./square
5000
Segmentation fault
```

The debugger will stop at the problem.

48 Stack trace

Displaying a stack trace	
gdb	lldb
`(gdb) where`	`(lldb) thread backtrace`

```
(gdb) backtrace
#0  0x00007fff824295ca in __svfscanf_l ()
#1  0x00007fff8244011b in fscanf ()
#2  0x0000000100000e89 in main (argc=1, argv=0x7fff5fbfc7c0) at sq
```

49 Inspecting a stack frame

Investigate a specific frame	
gdb	clang
`frame 2`	`frame select 2`

Then `print` variables and such.

50 Out-of-bounds errors

```
// up.c
  int nlocal = 100,i;
  double s, *array = (double*) malloc(nlocal*sizeof(double))
  for (i=0; i<nlocal; i++) {
    double di = (double)i;
    array[i] = 1/(di*di);
  }
  s = 0.;
  for (i=nlocal-1; i>=0; i++) {
    double di = (double)i;
    s += array[i];
  }
```

51 Out of bounds in debugger

```
Program received signal EXC_BAD_ACCESS, Could not access mem
Reason: KERN_INVALID_ADDRESS at address: 0x0000000100200000
0x0000000100000f43 in main (argc=1, argv=0x7fff5fbfe2c0) at
15          s += array[i];
(gdb) print array
$1 = (double *) 0x100104d00
(gdb) print i
$2 = 128608
```

52 Breakpoints

Set a breakpoint at a line	
gdb	lldb
`break foo.c:12`	`breakpoint set [ -f foo.c ] -l 12`

53 Stepping

Stepping through a program		
gdb	lldb	meaning
`run`		start a run
`cont`		continue from breakpoint
`next`		next statement on same level
`step`		next statement, this level or next

Memory debugging

54 Program with problems

tutorials/gdb/c/square1.c

```
#include <stdlib.h>
#include <stdio.h>
int main(int argc,char **argv) {
  int nmax,i;
  float *squares,sum;

  fscanf(stdin,"%d",&nmax);
  squares = (float*) malloc(nmax*sizeof(float));
  for (i=1; i<=nmax; i++) {
    squares[i] = 1./(i*i);
    sum += squares[i];
  }
  printf("Sum: %e\n",sum);

  return 0;
}
```

55 Valgrind output

```
%% valgrind square1
==53695== Memcheck, a memory error detector
==53695== [stuff]
10
==53695== Invalid write of size 4
==53695==    at 0x100000EB0: main (square1.c:10)
==53695==  Address 0x10027e148 is 0 bytes after a block of s
==53695==    at 0x1000101EF: malloc (vg_replace_malloc.c:236
==53695==    by 0x100000E77: main (square1.c:8)
==53695==
```

Parallel Debugging

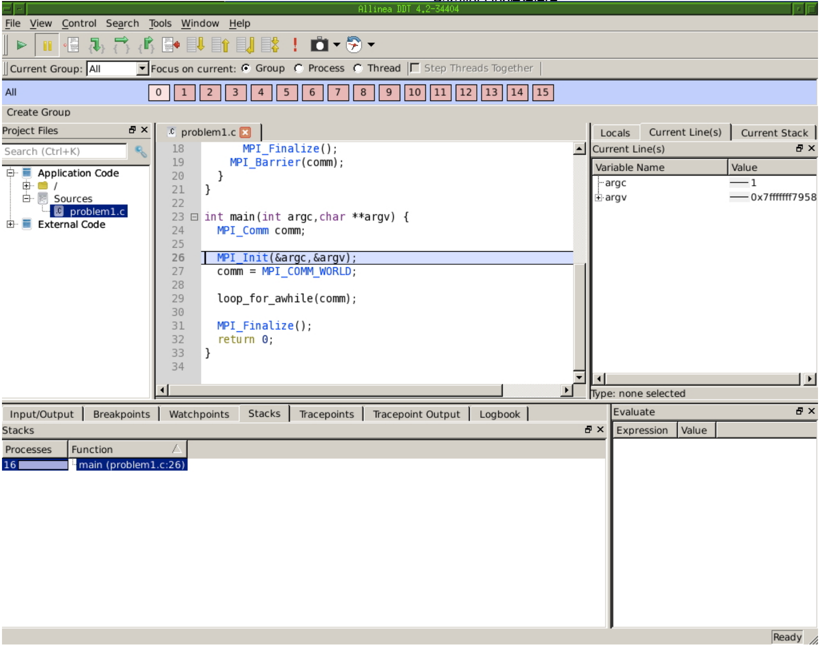
56 Debugging

I assume you know about gdb and valgrind…

- Interactive use of gdb, starting up multiple xterms
  feasible on small scale
- Use gdb to inspect dump:
  can be useful, often a program crashes hard and leaves no dump

Note: compile options `-g -O0`

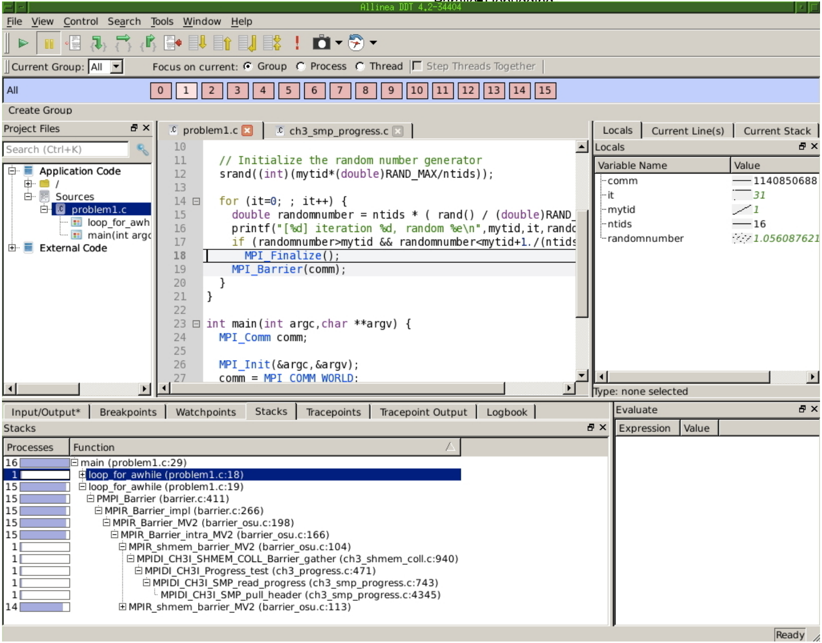
57 Parallel debuggers



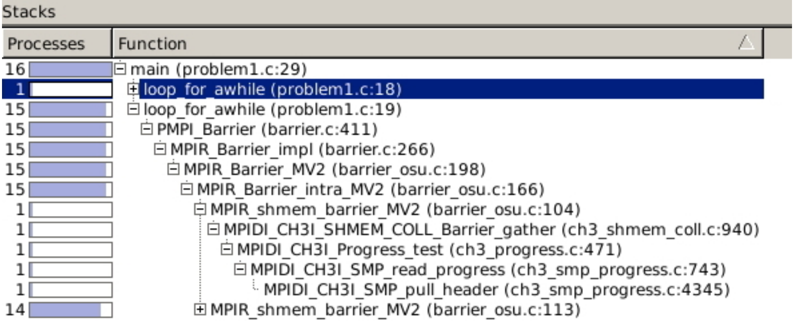
58 Buggy code

```
for (it=0; ; it++) {
  double randomnumber = ntids * ( rand() / (double)RAND_MAX
  printf("[%d] iteration %d, random %e\n",mytid,it,randomnum
  if (randomnumber>mytid && randomnumber<mytid+1./(ntids+1))
    MPI_Finalize();
  MPI_Barrier(comm);
}
```

59 Parallel inspection



60 Stack trace



61 Variable inspection