Part III

Compiling, Linking, and Makefiles

Compilation

- You (usually) write your code in human-readable form
- Compilation turns this into machine-executable code
- There are various steps taken by the compiler to do this.
- Here I cover only C(++) and Fortran as done by `gcc`.
- Other compilers should have very similar options.

Basic compilation instructions

- The most simple example:

  ```
  gcc myProgram.C -o myProgram
  ```

  compiles a single file `myProgram.C` into an executable `myProgram`.
- `gcc` detects, based on file-extension (`.C` here) the language it should assume (`C++` here).
- Alternatives are: `.f`, `.f90`, `.F`, `.c`, `.cxx`, `.cpp`, and more to compile C, C++, Fortran, etc.
- You can force `gcc` to assume a particular language when compiling, ignoring the file extension, but it's best to stick with convention.

Single file compilation

- The compiler goes through many stages in compiling even a single file:
- Preprocessor - processes `#include`, `#define`, etc. to generate a single file (C/C++ only)
- Parser - turns code into an internal representation of the program
- Optimizer - Analyzes the parsed code and identifies and applies possible optimizations
- Compilation - Turns the parsed code into assembly language
- Assembly - Turns the assembly language into executable machine code

Examining different phases

- Preprocessor: Passing `-E` will emit preprocessed source-code
- Parser: Output not available. This representation is for compiler developers only.
- Optimizer: Try the option `-fopt-info-all` to see a detailed optimization report
- Compilation - Passing `-S` will emit assembly language
- Assembly - Default is to output compiled machine-code (use `-c`)

Linking

- If you have inter-dependent functions in separate files (for readability) then we need an extra stage.
- For each source file, pass `-c` to the compiler to generate an object file `.o`.
- For example:

```
gcc -c myFile.C -o myFile.o
gcc -c myFile2.C -o myFile2.o
```

- This creates two separate files `myFile.o` and `myFile2.o` which are not complete programs and cannot be executed.

Linking continued

- To make an executable, we must link multiple object files together:

```
gcc myFile.o myFile2.o -o myExecutable
```

- This analyzes the object files listed, determines which functions are defined in each of them, and which functions are called by each of them.
- Any references from one object file to another object file's function(s) are replaced with appropriate machine code calls.
- If any references are not found, the linker will crash with `Undefined reference` and should tell you which file called the function whose definition was not found.
- If this happens, you need to include the file in which the function *is* defined in the linking command.

Libraries

- As well as your own functions, you may wish to call functions from other libraries (LAPACK, BLAS, etc.)
- For these, shared-object files are usually present on the computer.
- Look in `/usr/lib/x86_64-linux-gnu/` for many examples, e.g. `/usr/lib/x86_64-linux-gnu/libblas.so`
- On my system (Ubuntu 20.04), this is eventually symlinked to `/usr/lib/x86_64-linux-gnu/openblas-pthread/libblas.so.3`
- To link to this library, add the option `-lblas` when linking.
- Other Linux OS will have libraries in different places.
- E.g. CSD3 has `/usr/lib64/libblas.so.3.8.0`.

Libraries continued

- The linker will look for `libblas.so` in all available library directories, mainly `/usr/lib`, and any others you specify.
- For the full list, run `gcc -print-search-dirs`
- If you have multiple versions of a library installed, or if they are installed in non-standard places, you need to tell the linker where to find the library.
- Pass `-L/opt/path/to/blas/lib` where this defines where the file `libblas.so` can be found.

In summary:

```
gcc myFile.o -L /opt/blas-intel/lib -lblas -o myProgram
```

will link to the version of BLAS in `/opt/blas-intel/lib`.

Run-time libraries

- Even though you have specified at linking where the library is, you must still specify its location at run-time.
- Linux will look in all directories in the `LD_LIBRARY_PATH` environment variable for the correct libraries.
- If a library is not found, Linux will give:
  `EulerSolver_1D: error while loading shared libraries: libblas.so: cannot open shared object file: No such file or directory`
- Usually, you set up your `LD_LIBRARY_PATH` in your `~/.bashrc` file using appropriate `export LD_LIBRARY_PATH=/opt/blas-intel/lib:$LD_LIBRARY_PATH` commands
- The `~/.bashrc` file is executed whenever a terminal is opened (or `ssh` session started).

Optimization

- The compiler can usually make very good attempts at optimizing your code, but it is not a mind reader.
- It is best to make your code as simple as possible, so that the compiler can find the optimizations it can make.
- Pass `-O<n>` where $n = 0, 1, 2, 3$ for different levels of optimization.
- Further optimization options are available, see `http://gcc.gnu.org` for details.

Machine specific optimization

- Much of the improvement in processors in the past 20 years has been due to extra instruction types.
- For example, if appropriate, a compiler can generate instructions to operate on 4 floats at once (SSE instructions).
- If you know that your code will only be running on processors with these instructions, you can pass `-msse`, `-msse3`, `-mavx`, etc. options to allow the compiler to generate these instructions.
- This means that your code will only run on processors with these instructions, though (watch out for `SIGILL`).
- This may or may not produce significant speed-up, depending on your code.
- Some codes cannot easily be vectorized and so may not benefit from these instructions.

Optimization techniques

- Over the years, many heuristic optimization techniques have been developed.
- These will usually help performance, although some may hinder it.
- Not all optimizations may apply to your problem; the compiler tries to determine which are appropriate.

Inlining

- When calling a function, code must be generated to store the current function location on the stack, allocate space for local variables, and move the execution point to the new function.
- If the function is called often enough, and is very simple, this generates a lot of overhead.
- The compiler can copy the function contents to the place from which it is called, and avoid the overhead.
- Downside: this may increase the program size slightly.
- In general, this can only be done if the function calling and the function called are in the same source-file (or one is in a header file)
- Compilers (`gcc`, `Intel`) do now have inter-procedural-optimization (IPO) to inline functions between separate object files, but this can be very expensive at link-time.

Constant propagation

- If an expression is reused within many calculations, the compiler *may* be able to compute it once and use the result multiple times.
- This can be applied to integer expressions easily, but usually not to floating point ones.
- Integer expressions may enter into loop-ranges, array offsets/indexes, etc. and may lead to other optimizations being possible.
- For floating-point, it's usually better to do this by hand:

```
const double sin2X = sin(x) * sin(x);
const double result = 10 + 3*sin2X + 8*sin2X*sin2X;
```

- The example above could probably not be done by the compiler since `sin` must set the global `errno` flag if `x` is infinity.

More constant propagation

- Constant propagation is more important for memory offsets:

```
const int a = 10;
double b[100];
for(int i=0 ; i < 10 ; i++){
  b[i*a] = i;
}
```

- Here the compiler knows the value of `a` and could therefore pre-compute `i*a` and determine memory offsets at compile-time.
- (Whether it does so depends on the compiler itself.)

Loop unrolling

- The loop

```
for(int i=0 ; i < 5 ; i++){
  a[i] = 2*b[i];
}
```

  has its number of iterations known at compile-time.

- The compiler can avoid creating and testing `i` by turning the loop body into 5 separate instructions.
- The compiler may then be able to do further optimizations on the remaining instructions.

Partial loop unrolling

- Even for loops where the number of iterations is unknown, the compiler may be able to do partial unrolling:

```
for(int i=0 ; i < N ; i++){
  a[i] = 2*b[i];
}
```

becomes

```
for(int i=0 ; i < N ; i+=4){
  a[i] = 2*b[i];
  a[i+1] = 2*b[i+1];
  a[i+2] = 2*b[i+2];
  a[i+3] = 2*b[i+3];
}
```

(with allowances for N not being divisible by 4).

- This reduces the number of increment-and-test instructions for `i`.

Vectorizing

- If there are no data dependencies between separate loop iterations, then the compiler may be able to use vectorized instructions:

```
float a[4], b[4], c[4];
for(int i=0 ; i < 4 ; i++){
  a[i] = b[i] + c[i];
}
```

- This can be done using a single SSE instruction: `addps` Add Packed Scalar.

How to confuse a compiler

- Compilers can be confused by non-simple constructs:

```
bool doneOneIter=false;
for(int i=0 ; !doneOneIter || i != 0 ; i=(i+7) % 20){
  a[i] = 2*b[i];
  doneOneIter = true;
}
```

is equivalent to the same loop from 0 to 19 (inclusive) but the compiler can't tell that (probably).

Aliasing

- Specific to C/C++ is the issue of aliasing:

```
void f(int *a, int *b, int *c){
  for(int i=0 ; i < 4 ; i++){
    a[i] = b[i] + c[i];
  }
}
```

- This cannot (probably) be optimized into an SSE instruction by the compiler because it does not know whether `a`, `b`, and `c` are distinct arrays.
- We could have called the function as

```
f(&a[0], &a[2], &a[3]);
```

  which is valid C, but means that the loop iterations are no longer independent, and vectorizing it would be incorrect.

Strength reduction

- One technique that a compiler cannot usually apply is that of strength reduction.
- This means replacing one arithmetic expression by a mathematically equivalent, but less expensive, one.
- For example:

```
a = pow(x, 2) - 1 == x*x - 1 == (x+1)*(x-1)
```

- The compiler cannot do this itself because the result may be numerically different (finite precision).
- *You* may need to worry about floating-point accuracy.
- The compiler can do some strength reduction:

```
a = c / 4;
a = c >> 2; // right-shift by 2 bits
```

are equivalent for integers, and the second may be less expensive.

Cross-compilation

- It is possible to compile code on one machine architecture for a completely different one.
- For example, to compile 32-bit code on a 64-bit machine, or AMD code on an Intel machine
- Or, if you really want, to compile a Windows-compatible executable on a Linux machine.
- This is for specialists only. If you really need it, consult the `gcc` manual.

Makefiles

You will have noticed that compiling and linking large programs takes many lines of shell script:

```
gcc -c main.c -o main.o
gcc -c ODEs.c -o ODE.o
gcc -c output.c -o output.o
gcc main.o ODE.o output.o -o ODEsolver -lm
```

You could put all these into a shell-script that you run every time you change a single file.
But, wouldn't it be nice if you could only recompile the files that needed it?

Make

- The `make` utility was invented in the 1970s as part of Unix.
- The most widely used implementation of this is `GNUmake` which is available on most Linux systems.
- It is capable of scheduling the commands required to generate a file, based on its declared dependencies, and whether these have changed more recently than the file.
- It uses a text-based `Makefile`, which lays out the rules for which files have which dependencies, and what commands need to be executed.

Makefile

- When `make` is executed, it looks for files in the current directory called `makefile`, `Makefile`, or `GNUMakefile`.
- If one of these is found, it is processed by `make`.
- A typical usage might be:

```
make clean
make euler_2D
```

Rules

A rule in `make` takes the form

```
target1 target2 : dependency1 dependency2
        command1
        command2
```

Note that each `command`n line begins with a TAB. This is important, and a common source of errors as spaces look like a TAB in many editors.

If spaces are used in place of a TAB, either instructions will not be carried out as you expect, or you may get the message

```
Makefile:3: *** missing separator (did you mean TAB instead of
    8 spaces?). Stop.
```

which is at least reasonably helpful.

Rule example

An example of a rule to make an object file would be:

```
euler_2D.o : euler_2D.c euler_2D.h
             gcc -c euler_2D.c -o euler_2D.o -O3
```

which will trigger if either `euler_2D.c` or `euler_2D.h` is changed.
A rule to make an executable would be

```
euler_2D : euler_2D.o
           gcc euler_2D.o -o euler_2D -lm -lhdf
```

So, if you change either of the source-files, and run `make euler_2D`, then the executable will be recompiled.

Automatic variables

In the context of a rule, some variables are automatically defined:

- $@ - The filename of the target
- $< - The filename of the first prerequisite
- $^- The filenames of all prerequisites
- $? - The filenames of all prerequisites newer than the target
- $* - The stem of the target filename

```
euler_2D.o : euler_2D.c euler_2D.h
             gcc -c $< -o $@ -O3
```

is equivalent to the first rule on the previous slide.

General rules

For a project with many files to compile, specifying each file's rule separately would become tedious and error-prone.
Instead we could use

```
%.o : %.c
            gcc -c $< -o $@ -O3
```

which will make all object files depend on their corresponding source file and compile accordingly.
Extra effort is required to make object files depend on all their correct header files.
The rule to make the final executable:

```
euler_2D : euler_2D.o RK2.o settings.o output.o
         gcc $^ -lm -o $@
```

and if these were the only two rules, `Make` would compile and link `euler_2D` correctly.

Variables

Within `Makefiles`, variables can be set and read:

```
CXX=g++
CXXFLAGS=-O3 -Wall -Wextra -g
%.o : %.c
            $(CXX) -c $< -o $@ $(CXXFLAGS)

CXXLIBS=-lm
euler_2D : euler_2D.o RK2.o settings.o output.o
          gcc $^ $(CXXLIBS) -o $@
```

Any variable names could be used, but these are standard ones.

Silent commands

- You often want to only output the actual command-output, not the command itself.
- For example, if you have turned on all compiler warnings, you want to see only those (or preferably their absence!), and not the long list of compiler options.
- To do this, prefix a command by @:

```
%.o: %.C
        @echo "Compiling $<"
        @$(CXX) $(CFLAGS) $(CPPFLAGS) $(CXXFLAGS)\
                $(CXXINCLUDES) -c $< -o $@
```

  would only output "Compiling Output.C" if there were no compiler errors or warnings.
- The \ is a line continuation character.

PHONY targets

- Sometimes you want special targets such as `all` or `clean`:

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

- However, if a file `clean` happens to exist, then this will not work.
- We can use:

```
.PHONY: clean
```

  to mark `clean` as a special target that will always be executed.

Parallel make

- Since multiple files can be compiled at once, we may wish to use our multi-core processor.
- `make -j<n>` will launch at most n command rules at once.
- If command rules can be executed independently (according to the dependency-graph), then they will be.
- Try using n to be the same as the number of CPU-cores.
- Actual speed-up may depend on input/output performance of your disk/file-system, which may now be the bottle-neck.

More information

- `Make` has substantially more features than I have described here.
- It permits macros, loops, and functions, and is Turing complete.
- For full documentation, go to `http://www.gnu.org/software/make/manual/`