HPC/Exascale Centre of Excellence in Personalised Medicine

Centre of Excellence for Computational Biomolecular Research

PATC course

Introduction to HPC for Life Scientists

31 January - 02 February 2022

BioExcel and the PerMedCoE projects have received funding from the European Union's Horizon 2020 research and innovation programme under the grant agreements 823830 & 675728 and 951773, respectively.

Horizon 2020 European Union Funding for Research & Innovation

Per Med CoE
HPC/Exascale Centre of Excellence in Personalised Medicine

bioexcel
Centre of Excellence for Computational Biomolecular Research

Compilers

Holly Judge

PATC course: Introduction to HPC for Life Scientists, 31 January - 02 February 2022

Partners

Funding

Horizon 2020 European Union Funding for Research & Innovation

Compilers

Algorithms to executables

Partners

Funding

Horizon 2020
European Union Funding
for Research & Innovation

Reusing this material

Outline

- What does compiling mean?
- Anatomy of a compiler
- Compiler “optimisations”
- Can the compiler parallelise my code?
- Why are there differences in compilers?

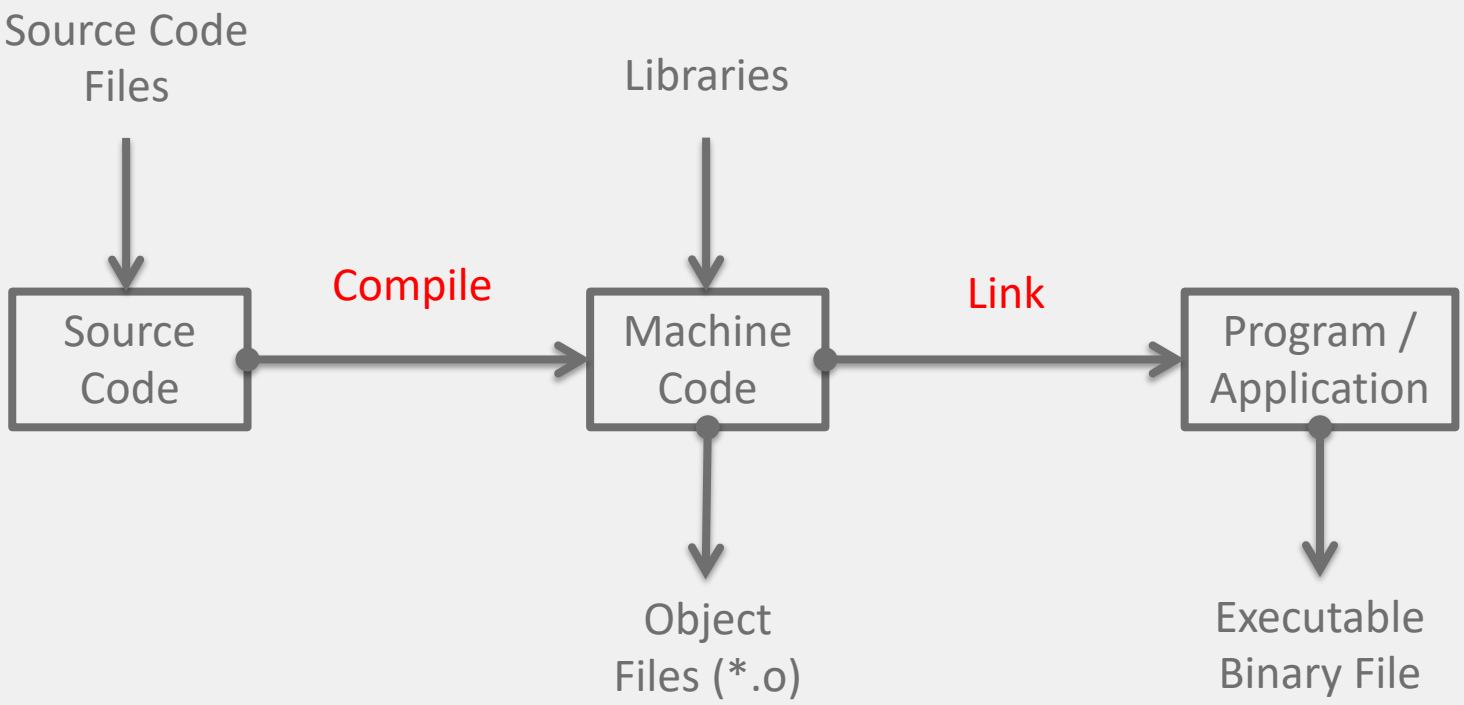
Compiling Overview

- HPC programs are usually written in a high-level, human-readable language.
  - Almost always Fortran, C, or C++ ("99%" of all HPC applications)
  - Rarely something else
- Processors execute machine code (via instruction sets)
- Compilers convert high-level *source code* into machine code.
  - Also incorporate functionality from external *libraries*
  - Usually try to *optimise* the code produced so that it runs as fast as possible on the processors

Anatomy of a compiler

How does it actually work?

Compiler Flow



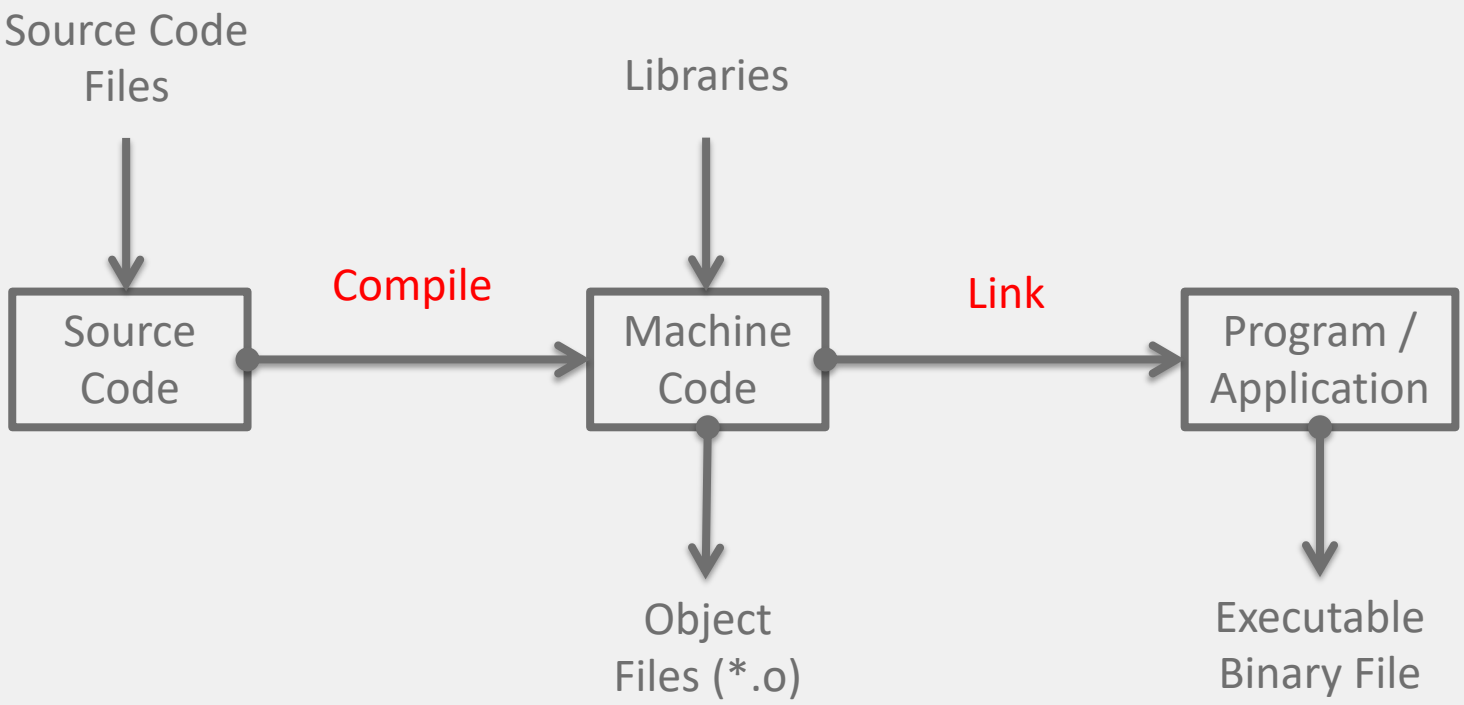
Compile Stage

- Transforms high level source to machine code
  - Produces *object* files – usually one object file per source file
- Actually consists of a number of sub-stages
  - Details are beyond this course
- *Optimisations* are performed at this stage
  - More on optimisations later

Libraries

- Libraries provide functionality that is common across multiple programs
  - Low level – e.g. filesystem access. Usually not interesting to users
  - Optimised numerical operations – e.g. linear algebra, Fourier transformations
  - Communications and parallelism – e.g. Message Passing Interface (MPI), OpenMP
- The compiler combines the code in these libraries with the code generated from the user's program to produce the final executable.
  - Linking at *run time* is also possible – known as dynamic linking (or shared libraries).

Compiler Flow



Link Stage

- Object files are combined (*linked)* to produce the actual application
  - Application is an *executable binary* file
- Any library code required by the application is also linked at this stage
- Two forms of linking:
  - *Static* – All code is combined into a single executable file
  - *Dynamic* – Code from libraries is not combined into executable file, instead this code is called and executed dynamically when the executable is run

What does this look like in practice?

- When building an application you may see object files being created in the build process, following by linking to make the executable:

```
gcc -c main.c
gcc -c input.c
gcc -c fftw-wrapper.c
gcc -o program main.o input.o fftw-wrapper.o
```

- If a required library cannot be found you may get errors at the link stage:

```
cc -o program main.o input.o fftw-wrapper.o
/usr/bin/ld: fftw-wrapper.o: in function `use_fftw':
fftw-wrapper.c:(.text+0x15): undefined reference to `fftw_malloc'
/usr/bin/ld: fftw-wrapper.c:(.text+0x2a): undefined reference to `fftw_malloc'
/usr/bin/ld: fftw-wrapper.c:(.text+0x4b): undefined reference to
`fftw_plan_dft_1d'
/usr/bin/ld: fftw-wrapper.c:(.text+0x5b): undefined reference to `fftw_execute'
collect2: error: ld returned 1 exit status
```

Compiler optimisations

What do they do? When should/shouldn't I use them?

Optimisation

- Compiler will try to alter code so it runs more quickly
  - This can be done at a number of levels (high-level, assembly code, machine code) and can include the reordering of operations
- Note: although these are called optimisations, this is a misnomer
  - Resulting code is never optimal
  - Usually a predetermined sequence of transformations that is known to produce performance gains for some codes.
  - Seldom any iterative process
  - Seldom any attempt to quantify effect of any transformations

Common optimisation strategies

- Loop index reordering (to match memory layout)
- Loop unrolling
- Use of fast mathematical operators
- Function inlining (avoiding a function call)
- Operation reordering to allow for cache reuse

Compiler flags

- Compiler flags pass options to the compiler such as
  - Language, warning and debugging options
  - Performance options
  - Level of optimisation

Option	Comment
-fopenmp	Compile with OpenMP
-O0	No optimisations (for debugging)
-O1	More optimisation
-O2	Recommended optimisation
-O3	More optimization (can have an impact on correctness)

When to use optimisation

- Simple answer: always
- You should always use the performance gains given by optimisation
- If you are debugging then you usually switch optimisation off to ensure that the statements are being executed in the order you specified
- If you suspect that compiler optimisations are causing a problem you can turn them off gradually
  - All good compilers allow the specification of a range of optimisation levels so you can turn it off gradually

Compilers and parallelisation

Can compilers parallelise my code?

Compiler parallelisation

- Compilers can produce parallel (or vector) instructions
  - Makes use of “SIMD” (Single Instruction, Multiple Data) instructions available on processor cores’ floating point units.
- However, they cannot (yet) produce the general, high-level parallelism required for scaling on multiple cores or nodes
  - Compilers do not have the holistic view required to produce this level of parallelism
  - Data parallelism is usually easier to produce automatically than task parallelism
  - Attempts have been made but with limited success so far.

Different compilers

Why are there differences between compilers?

Standards and implementations

- Different compilers have different standards and implementations
- Different optimization strategies
  - Some compilers are open source (GNU), others commercial (Intel) and can take advantage of detailed knowledge about hardware behavior
- Different ideas about code correctness
  - A program that compiles with gcc may not compile with Intel due to these differences
  - Newer versions of compilers may be more strict on correctness which means older code will not compile with these newer compilers
    - Sometimes flags can convert errors to warnings at compile time

Summary

- Compilers are hugely important to HPC application performance
- Correct usage can provide significant performance benefits
  - With some caveats
- It is important to be aware of the differences between compilers and whether your code requires a specific compiler
- Available compilers and recommended flags for a machine can usually be found in the documentation
- Consult the documentation for the code you are trying to compile