Compilers

Algorithms to executables

Outline

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

Compiling

What does compiling mean?

Compiling Overview

- HPC programs are usually written in a high-level, human-readable language.
  - Almost always Fortran or C (99% of all HPC applications)
  - Occasionally C++; 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

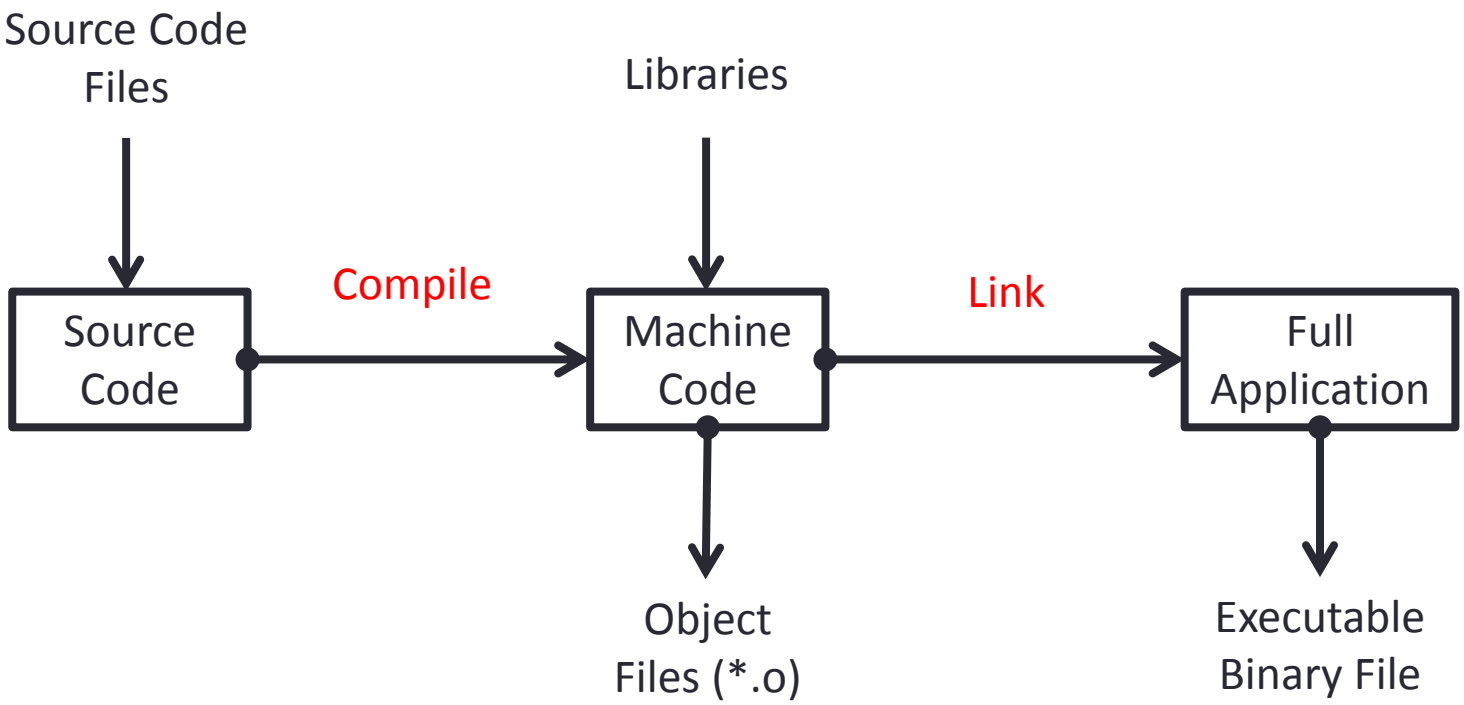
Libraries

- 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).

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

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 dynamically include when the executable is run

Compiler optimisations

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

Optimisation

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

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

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 the SIMD instructions on the core's floating point unit.
- However, they cannot produce the general, high-level parallelism required for scaling on multiple cores
  - Compilers do not have the holistic view required to produce this level of parallism
  - Data parallelism is usually easier to produce automatically than task parallelism
  - Attempts have been made to automate this but with limited success so far.

Different compilers

Why are there differences between compilers?

Standards and implementations

- Although standards exist they cannot cover all cases and contain ambiguities
- When the standard is not clear then it is up to the compiler architect to select the behaviour
  - Differences exist between compiler implementations