Compilers

Algorithms to executables

Reusing this material

Outline

- What does compiling mean?
  - Where do libraries come in?
- Anatomy of a compiler
- Compiler “optimisations”
- Can the compiler parallelise my code?
- Why are there differences in compilers?
  - On ARCHER we have the Cray, Intel and GNU compilers

Compiling

What does compiling mean?

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

Libraries

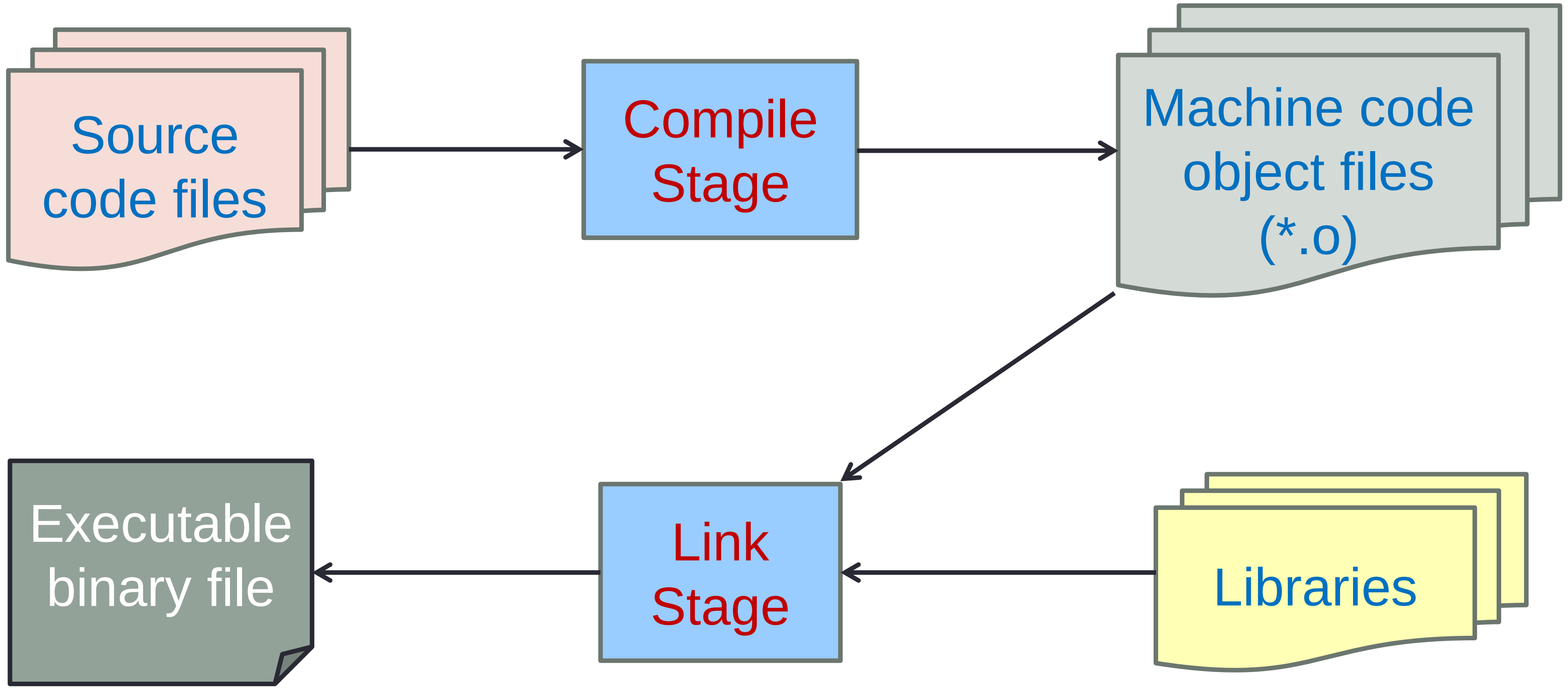
- Libraries provide functionality that is common across multiple programs
  - Low level – e.g. filesystem access. Usually not interesting to users
  - Optimised numerical operations – e.g. linear algebra, Fourier transformations
  - Communications and parallelism – e.g. Message Passing Interface (MPI), 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).

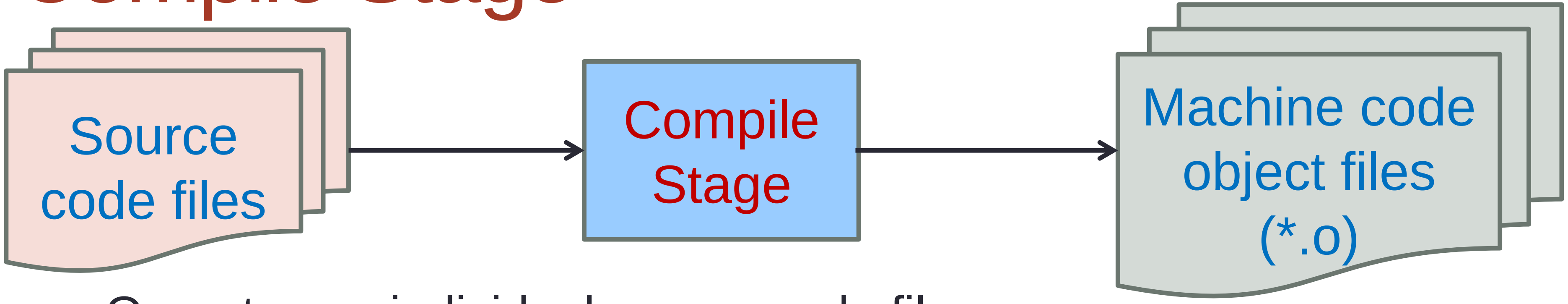
Anatomy of a compiler

How does it actually work?

Compiler Flow



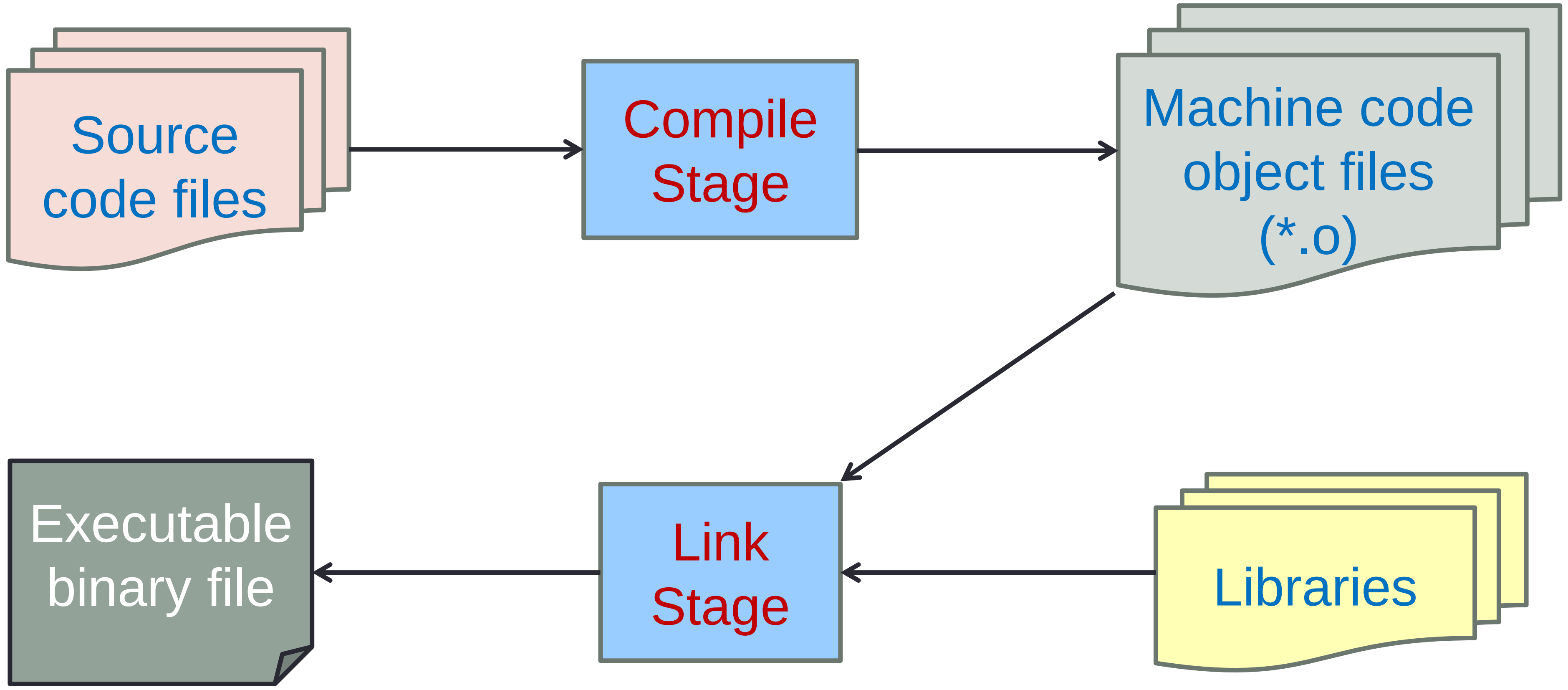
Compile Stage



- Operates on individual source code files
- Transforms high level source to machine code
  - Produces *object* files – usually one object file per source file

- Error and warning checking performed
- *Optimisations* are performed
  - More on optimisations later
- Actually consists of a number of sub-stages
  - Details are beyond this course

Compiler Flow

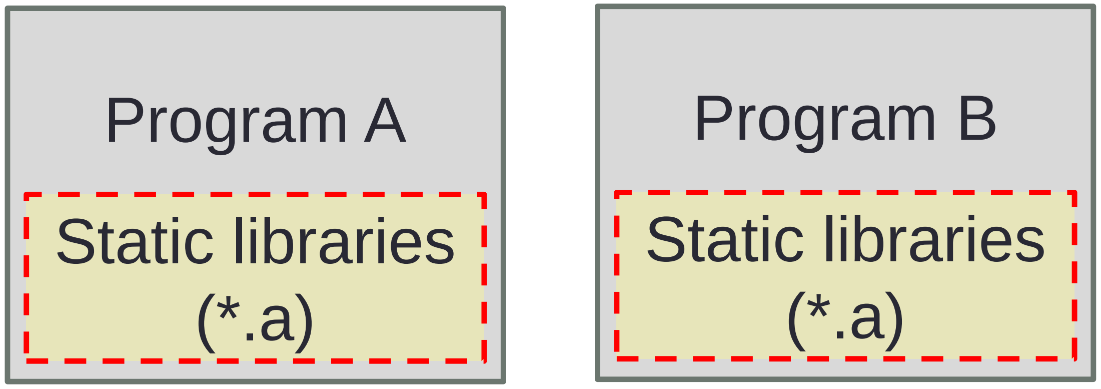


Link Stage

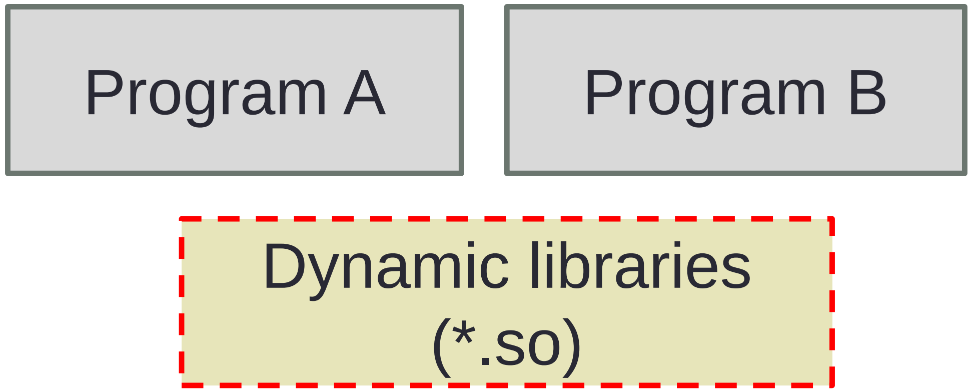
- Object files are combined (*linked)* to produce the actual application
  - Application is an *executable binary* file

- 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

Illustration of library linking



Static linking at compile time, executable contains the libraries



Dynamic linking at runtime, no libraries contained in the executable and these are loaded in when the program runs

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
  - 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 or make more effective use of the cache
- Loop unrolling
  - Reduces the number of (or avoids) termination checks & jumps
- Use of fast mathematical operators
  - Non IEEE compliant mathematical operations can speed up arithmetic
  - But can no longer be sure the answer is reproducible or correct (as disables correctness checking.)
- Function in-lining
  - 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

- Compilers commonly combine optimisations into different levels
  - O0, O1, O2, O3 ← where 0 is no optimisation and 3 the most extreme
  - Other optimisations (such as Os for executable size.)

A warning on optimisation

- Some optimisations can change the order of calculations
  - Which means that your code might produce slightly different results with or without that optimisation enabled.
  - When enabling new optimisations it is always worth ensuring that the code still produces “correct” results

- 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
  - An easy initial test is to reduce the optimisation level, i.e. to go from O3 to O2

Cray, Intel and GNU compiler flags

Feature	Cray	Intel	GNU
Listing	-ra (fnt) -hlist=a (cc/CC)	-opt-report3	-fdump-tree-all
Free format (ftn)	-f free	-free	-ffree-form
Vectorization	By default at -O1 and above	By default at -O2 and above	By default at -O3 or using -ftree-vectorize
Inter-Procedural Optimization	-hwp	-ipo	-flto (note: link-time optimization)
Floating-point optimizations	-hfpN, N=0...4	-fp-model [fast|fast=2|precise|except|strict]	-f[no-]fast-math or -funsafe-math-optimizations
Suggested Optimization	(default)	-O2 -xAVX	-O2 -mavx -ftree-vectorize -ffast-math -funroll-loops
Aggressive Optimization	-O3 -hfp3	-fast	-Ofast -mavx -funroll-loops
OpenMP recognition	(default)	-fopenmp	-fopenmp
Variables size (ftn)	-s real64 -s integer64	-real-size 64 -integer-size 64	-freal-4-real-8 -finteger-4-integer-8
Debugging	-g	-g	-g

Compilers and parallelisation

Can compilers parallelise my code?

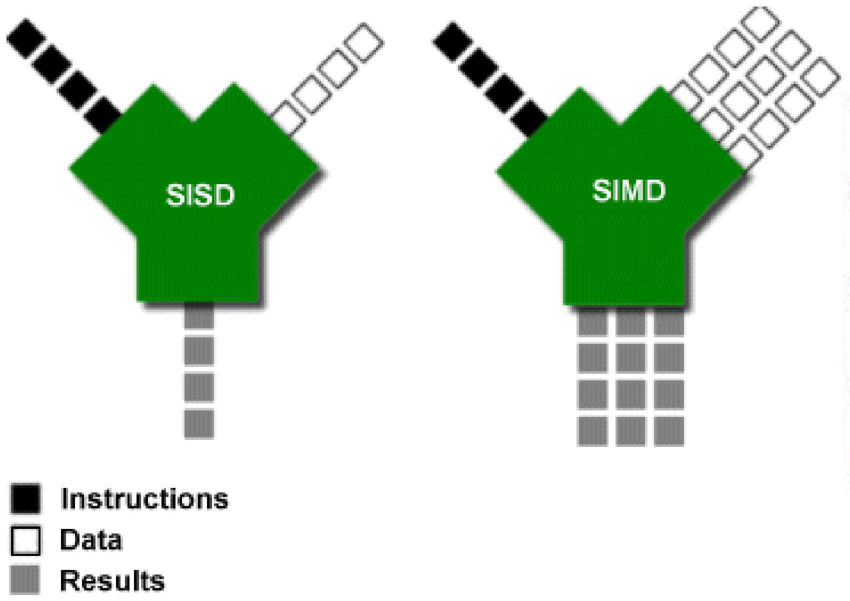
Compiler parallelisation

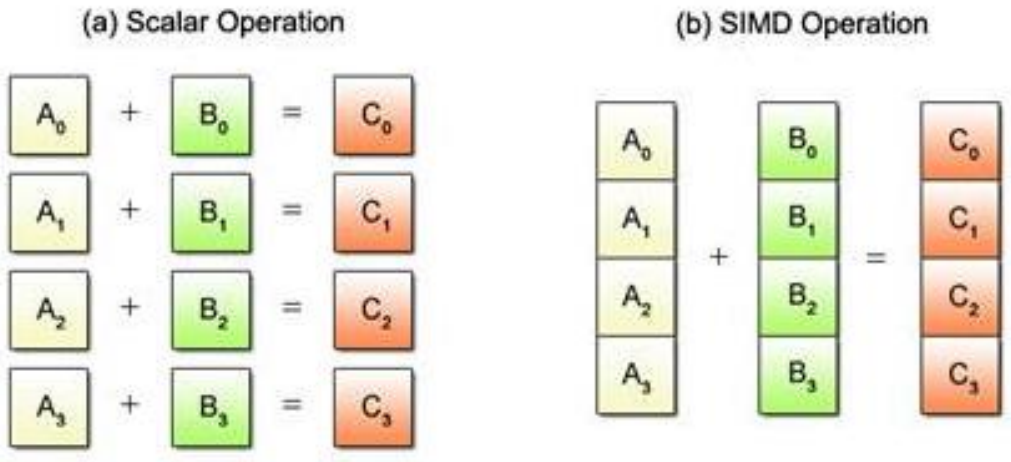
- They cannot (yet) produce the general, high-level parallelism required for scaling on multiple cores or nodes
  - Compilers do not have the holistic view required to produce this level of parallism
  - Data parallelism is usually easier to produce automatically than task parallelism
  - Attempts have been made but with limited success so far.

- However, compilers often make a good job of automatically parallelising floating point operations at the CPU instruction level

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.





Different compilers

Why are there differences between compilers?

Standards and implementations

- Compilers implement the behaviour specified in agreed standards for languages
  - Multiple standards exist and change over time
  - Standards cannot cover all cases and can contain ambiguities
  - Some details are left unspecified

- Wherever the standard is not clear it is up to the compiler architects to select the behaviour
  - Leads to differences between compiler implementations
  - Facilitates or hinders different optimisation possibilities
- Some compilers are open source (GNU), others commercial (Intel) and can take advantage of detailed knowledge about hardware behaviour

Summary

Summary

- The compiler is a hugely important part of the HPC workflow
- 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