

Compilers

Algorithms to executables

Partners



Funding



Reusing this material



This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License.

http://creativecommons.org/licenses/by-nc-sa/4.0/deed.en_US

This means you are free to copy and redistribute the material and adapt and build on the material under the following terms: You must give appropriate credit, provide a link to the license and indicate if changes were made. If you adapt or build on the material you must distribute your work under the same license as the original.

Note that this presentation contains images owned by others. Please seek their permission before reusing these images.

Outline

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

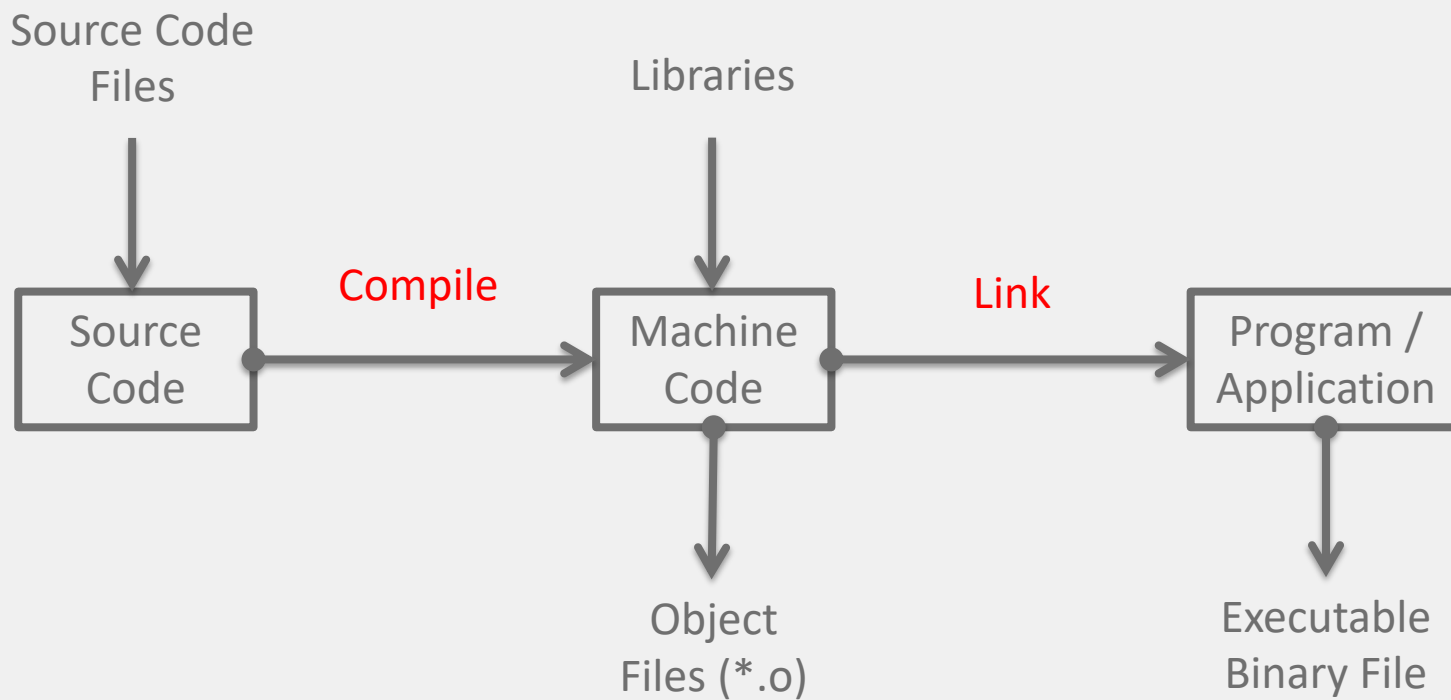
Compiling Overview

- 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



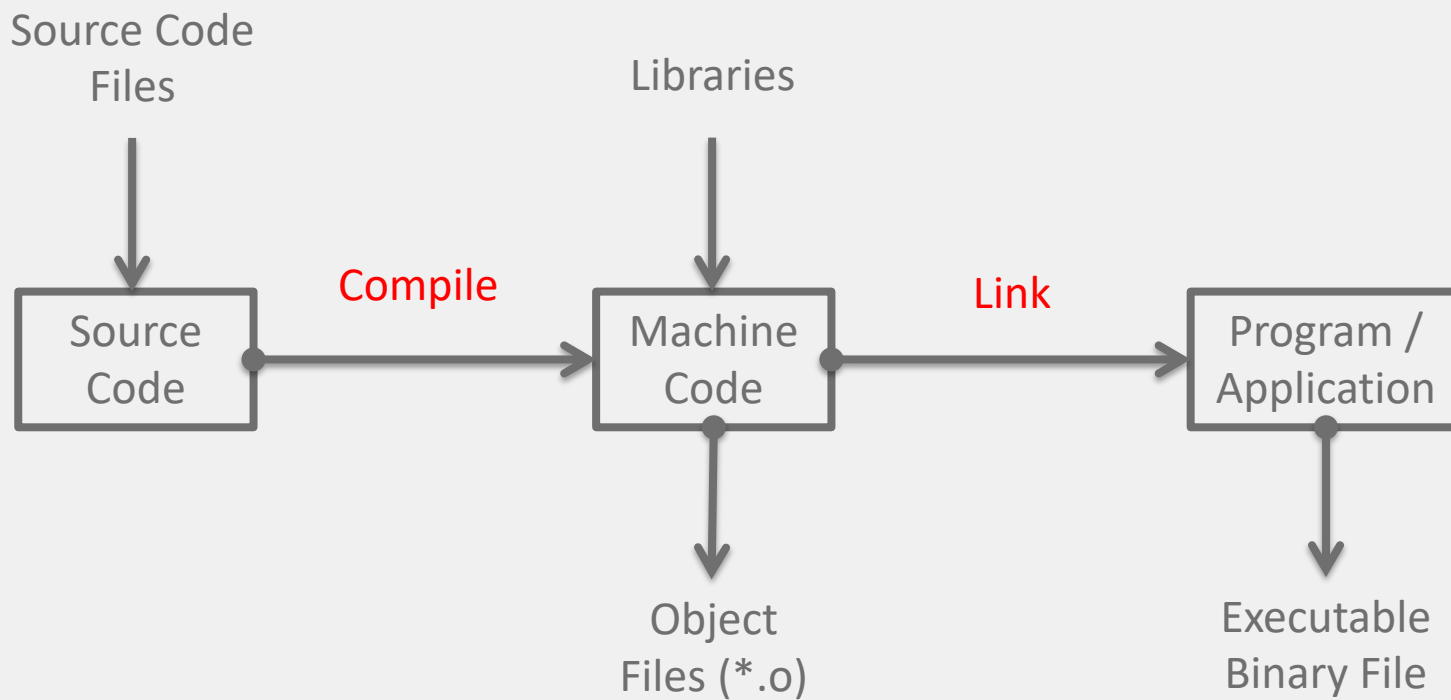
Compile Stage

- 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

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 involved
 - Seldom any attempt to quantify effect of any transformations

Common optimisation strategies

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

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

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

- 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

- Compilers are hugely important to HPC
- 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