# Compilers/Linkers/Libraries

Johan Hellsvik

# Example code

The lesson and the code examples are edited on

https://github.com/PDC-support/comp-link-lib

Obtain the code by cloning the git repository

```
git clone https://github.com/PDC-support/comp-link-lib.git
```

# Contents

- Concepts: compilers, linkers, libraries
- The Gnu Compiler Collection (GCC)
- Building a Fortran library and a Fortran program with
  - static linking
  - dynamic dynamic linking
- Building and running code on large clusters
- Building a C program with dynamical libraries
  - with linking to a Cray Parallel Environment (PE) provided library
  - to a user/staff installed optional library

# Concepts: compilers, linkers, libraries

- Compilers: A compiler parses the source code and generates object files and executable files. Under the hood a compiler invokes assemblers and linkers.
- Linkers: A tool to combine two or more or more object files into executable files. Linking can be static or dynamic.
- Libraries: Collection of object files for the purpose of re-use of functionality for different programs. Libraries can be static or dynamic.

# The GCC compiler suite

- The GNU Compiler Collection includes front ends for C, C++, Fortran and a few other languages.

Reference: https://gcc.gnu.org/

# GCC intermediate steps

**The four steps of compilation with GCC**

- Preprocessing: Inserts contents of header files. Removes comments
- Compilation: Converts the preprocessed code to assembly code. The completion is a complex process that proceed in multiple steps.
- Assembly: Converts assembly code to machine code object files
- Linking: Connects object files and libraries to executable code

# Hello world in C and Fortran

## C code: hello_world.c

```
#include <stdio.h>
int main() {
  printf("Hello world\n");
  return 0;
}
```

## Fortran code: hello_world.f90

```
program hello_world
  write(*,'(a)') 'Hello world'
end program hello_world
```

## Build and run C code

```
gcc hello_world.c -o hello_world_c.x
./hello_world_c.x
```

## Build and run Fortran code

```
gfortran hello_world.f90 -o hello_world_fortran.x
./hello_world_fortran.x
```

# Building a Fortran program with multiple source code files

In this example we are working with a small Fortran program with the three source code files. The program is calculating the cross product of two vectors by calling a subroutine.

## parameters.f90

A source code file that defines custom kinds of real numbers

```
module parameters
   implicit none
   integer, parameter :: snglprec = selected_real_kind(6, 37)   !< define precision for single reals
   integer, parameter :: dblprec = selected_real_kind(15, 307)  !< define precision for double reals
   integer, parameter :: qdprec = selected_real_kind(33, 4931)  !< define precision for quad reals
end module parameters
```

## crossproduct.f90

The subroutine in which the cross product is calculated

```
!cross product of two 3-vectors a and b
subroutine crossproduct(a,b,c)
  use parameters
  implicit none
  real(dblprec), dimension(3), intent(in) :: a  !< Left factor
  real(dblprec), dimension(3), intent(in) :: b  !< Right factor
  real(dblprec), dimension(3), intent(out) :: c  !< The cross product of a and b
  c(1) = a(2) * b(3) - a(3) * b(2)
  c(2) = a(3) * b(1) - a(1) * b(3)
  c(3) = a(1) * b(2) - a(2) * b(1)
  return
end subroutine crossproduct
```

## mathdemo.f90

The main program. Here two vectors are defined and assigned to values, and a call made to the subroutine crossproduct.

```
program mathdemo
  use parameters
  implicit none
  real(dblprec), dimension(3) :: x,y,z
  x = (/7, 6, 3/)
  y=  (/8, 9, 5/)
  call crossproduct(x,y,z)
  write(*,10001) 'The cross product of'
  write(*,10002)  'x=', x, 'and'
  write(*,10002)  'y=', y, 'is'
  write(*,10002)  'z=', z
10001 format (a,f12.6,a)
10002 format (a,3f12.6)
end program mathdemo
```

# Compiling and linking via object files

```
gfortran -c parameters.f90
gfortran parameters.o crossproduct.f90 mathdemo.f90 -o mathdemo.x
```

```
gfortran parameters.f90 crossproduct.f90 mathdemo.f90 -o mathdemo.x
```

# Building with static linking to library

Compile the source code files parameters.f90 and crossproduct.f90.

Create a library with the command `ar`

```
gfortran -c parameters.f90 crossproduct.f90
ar r libcrossproduct.a parameters.o
```

View contents of the library

```
ar t libcrossproduct.a parameters.o
parameters.o
crossproduct.o
```

Build program mathdemo with static linking to the library

```
gfortran mathdemo.f90 libcrossproduct.a -o mathdemo.x
```

# Building with dynamic linking to library

Compile the source code files parameters.f90 and crossproduct.f90. Specify position independent code with `-fpic`. Create a library with the flag `-shared`

```
gfortran -fpic -c parameters.f90 crossproduct.f90
gfortran -shared -o libcrossproduct.so parameters.o crossproduct.o
```

Compile mathdemo.f90

```
gfortran -c mathdemo.f90
```

Link mathdemo.o to libcrossproduct.so

```
gfortran mathdemo.o -o mathdemo.x  -L`pwd` -lcrossproduct
```

# Building and running code on large clusters

- With a large pool of users comes large number of programs and libraries. This needs to be managed to avoid interference
- A common solution is to use so called module systems to selectively activate/deactive the programs and libraries that are of relevance.
- Programs and libraries are typically built and installed with regular user privilegies. Use root rights only when really needed.
- Updates of libraries need to be made with caution as they potentially can break the functionality of large number of programs

# Cray programming environment (CPE)

The CPE is used on the Dardel HPE Cray EX supercomputer

Reference page: Compilers and libraries

The Cray Programming Environment (CPE) provides consistent interface to multiple compilers and libraries.

- In practice, we recommend
  - `ml cpeCray/23.12`
  - `ml cpeGNU/23.12`
  - `ml cpeAOCC/23.12`
- The `cpeCray`, `cpeGNU` and `cpeAOCC` modules are available after `ml PDC/23.12`
- No need to `module swap` or `module unload`

# Compiler wrappers

- Compiler wrappers for different programming languages
  - `cc` : C compiler wrapper
  - `CC` : C++ compiler wrapper
  - `ftn` : Fortran compiler wrapper
- The wrappers choose the required compiler version and target architecture optinons.
- Automatically link to MPI library and math libraries
  - MPI library: `cray-mpich`
  - Math libraries: `cray-libsci` and `cray-fftw`

# Compile a simple MPI code

- `hello_world_mpi.f90`

```
program hello_world_mpi
include "mpif.h"
integer myrank,mysize,ierr
call MPI_Init(ierr)
call MPI_Comm_rank(MPI_COMM_WORLD,myrank,ierr)
call MPI_Comm_size(MPI_COMM_WORLD,mysize,ierr)
write(*,*) "Processor ",myrank," of ",mysize,": Hello World!"
call MPI_Finalize(ierr)
end program
```

```
ftn hello_world_mpi.f90 -o hello_world_mpi.x
```

# What flags does the `ftn` wrapper activate?

- Use the flag `-craype-verbose`

```
ftn -craype-verbose hello_world_mpi.f90 -o hello_world_mpi.x
```

# Test run the hellow world MPI code

- Run on eight cores in the `shared` partition

```
user@uan01:> salloc -n 8 -t 10 -p shared -A <name-of-allocation>
user@uan01:> srun -n 8 ./hello_world_mpi.x
 Processor             4  of             8 : Hello World!
 Processor             6  of             8 : Hello World!
 Processor             7  of             8 : Hello World!
 Processor             0  of             8 : Hello World!
 Processor             1  of             8 : Hello World!
 Processor             2  of             8 : Hello World!
 Processor             3  of             8 : Hello World!
 Processor             5  of             8 : Hello World!
```

# Compile a simple linear algebra code

Link to the code

Use cray-libsci

```
ml PDC/23.12 cpeGNU/23.12
```

```
cc dgemm_test.c -o dgemm_test_craylibsci.x
```

# Compile a simple linear algebra code

Use openblas

```
ml PDC/23.12 openblas/0.3.28-cpeGNU-23.12
```

```
cc dgemm_test.c -o dgemm_test_openblas.x -I$EBROOTOPENBLAS/include -L$EBROOTOPENBLAS/lib -lopenblas
```

where the environment variable `EBROOTOPENBLAS` had been set when loading the OpenBLAS module. Its module file includes a statement

```
setenv("EBROOTOPENBLAS","/pdc/software/23.12/eb/software/openblas/0.3.28-cpeGNU-23.12")
```

which corresponds to an export statement

```
export EBROOTOPENBLAS=/pdc/software/23.12/eb/software/openblas/0.3.28-cpeGNU-23.12
```

# Check which shared libraries are needed in runtime

```
ldd dgemm_test_craylibsci.x
```

```
ldd dgemm_test_openblas.x
```

# Check which shared libraries are needed in runtime

```
ldd dgemm_test_craylibsci.x

...
libsci_cray.so.6 => /opt/cray/pe/lib64/libsci_cray.so.6
...
```

```
ldd dgemm_test_openblas.x

...
libopenblas.so.0 => /pdc/software/23.12/eb/software/openblas/0.3.28-cpeGNU-23.12/lib/libopenblas.so.0
...
```

# Test run the dgemm_test code

- Run on a single core in the `shared` partition

```
salloc -n 1 -t 10 -p shared -A <name-of-allocation>
srun -n 1 ./dgemm_test_craylibsci.x
srun -n 1 ./dgemm_test_openblas.x
exit
```

- Expected output:

```
    2.700     4.500     6.300     8.100     9.900    11.700    13.500
    4.500     8.100    11.700    15.300    18.900    22.500    26.100
    6.300    11.700    17.100    22.500    27.900    33.300    38.700
```

# Exercise: Compile and run `fftw_test` code

```
ml cray-fftw/3.3.10.6

wget https://people.math.sc.edu/Burkardt/c_src/fftw/fftw_test.c

cc --version
cc fftw_test.c -o fftw_test.x

ldd fftw_test.x

salloc -n 1 -t 10 -p shared -A <name-of-allocation>
srun -n 1 ./fftw_test.x
```

# Compilation of large program

- Examples at https://www.pdc.kth.se/software
- See for instance instructions for building the density functional theory program VASP https://www.pdc.kth.se/software/software/VASP/cpe23.12/6.4.3-vanilla/index_building.html

# Environment variables for manual installation of software

- Environment variables for compilers

```
export CC=cc
export CXX=CC
export FC=ftn
export F77=ftn
```

- Environment variables for compiler flags
  - add `-I`, `-L`, `-l`, etc. to Makefile
- Environment variables at runtime
  - prepend to `PATH`, `LD_LIBRARY_PATH`, etc.

# What happens when loading a module

```
ml show elpa/2023.05.001-cpeGNU-23.12
```

```
whatis("Description: ELPA - Eigenvalue SoLvers for Petaflop-Applications")
prepend_path("CMAKE_PREFIX_PATH","/pdc/software/23.12/eb/software/elpa/2023.05.001-cpeGNU-23.12")
prepend_path("LD_LIBRARY_PATH","/pdc/software/23.12/eb/software/elpa/2023.05.001-cpeGNU-23.12/lib")
prepend_path("LIBRARY_PATH","/pdc/software/23.12/eb/software/elpa/2023.05.001-cpeGNU-23.12/lib")
prepend_path("PATH","/pdc/software/23.12/eb/software/elpa/2023.05.001-cpeGNU-23.12/bin")
prepend_path("PKG_CONFIG_PATH","/pdc/software/23.12/eb/software/elpa/2023.05.001-cpeGNU-23.12/lib/pkgconfig")
...
```

# When running your own code

- Load correct programming environment (e.g. `cpeGNU`)
- Load correct dependencies (e.g. `openblas` if your code depends on it)
- Properly prepend to environment variables (e.g. `PATH`, `LD_LIBRARY_PATH`)
- Choose correct SLURM settings

# References

- Fortran-lang.org Fortran quickstart tutorial
- https://opensource.com/article/22/5/dynamic-linking-modular-libraries-linux
- https://opensource.com/article/22/6/static-linking-linux
- https://opensource.com/article/20/6/linux-libraries
- Compiling and running code on CPU nodes, in PDC intro presentation