



bw|HPC – C5

bwHPC course – Tutorial: Compiling, Makefile, Parallelising

Hartmut Häfner



UNIVERSITÄT
HEIDELBERG
ZUKUNFT
SEIT 1386

Hochschule
für Technik
Stuttgart



Hochschule Esslingen
University of Applied Sciences

Universität
Konstanz



UNIVERSITÄT
MANNHEIM



Universität Stuttgart

EBERHARD KARLS
UNIVERSITÄT
TÜBINGEN



KIT
Karlsruher Institut für Technologie



ulm university universität
uulm

Funding:



Baden-Württemberg

MINISTERIUM FÜR WISSENSCHAFT, FORSCHUNG UND KUNST

www.bwhpc-c5.de

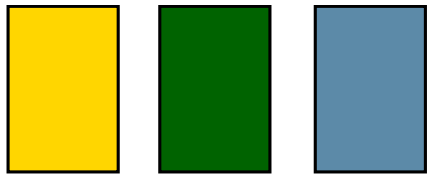
Outline

- Compiler + Numerical Libraries
 - commands
 - linking
- Makefile
 - Intro, Syntax (Explicit + Implicit Rules ...)
 -
- Parallelising
 - OpenMP
 - MPI

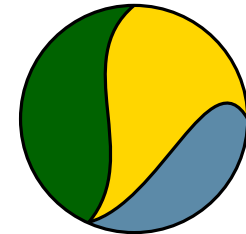
1. Compilation

Object files

source (.c)



executable (.x)

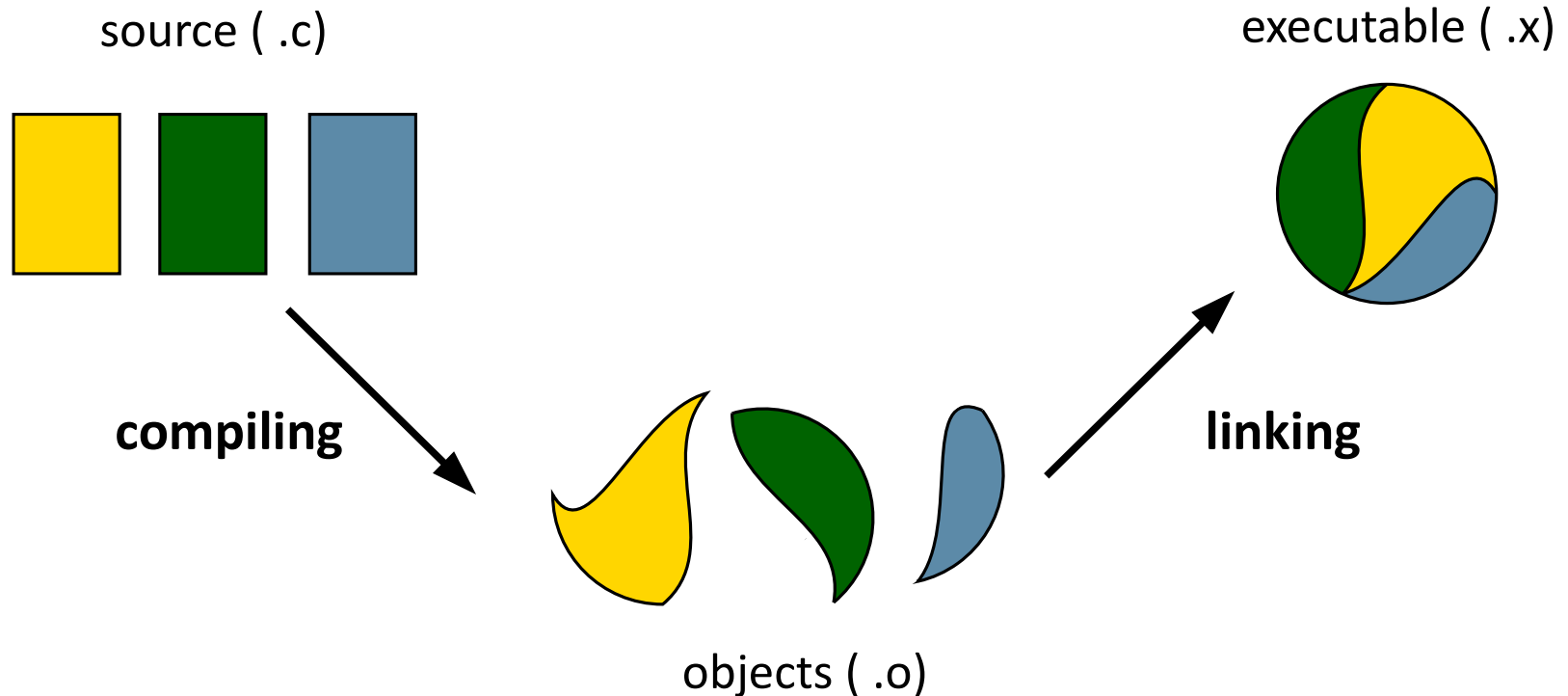


■ Example:

```
$ gcc -o exec.x src1.c src2.c src3.c
```

```
$ ./exec.x
```

Object files



```
$ gcc -c src1.c; gcc -c src2.c; gcc -c src3.c  
$ gcc -o exec.x src1.o src2.o src3.o
```

- Changes in a single file do not afford the compilation of all source code.

Include files

- Header files (.h)
 - Declaration of variables
 - Definition of static variables
 - Declaration of functions/subroutines
 - ..
- Example: include header file `/home/myincs/header.h`

- Preprocessor directive in source code:

```
#include "header.h"  
...  
src1.c
```

'#' does **not** initiate command lines but preprocessor directives in C/C++ code!

- Add header directory `-I<include_directory>`

```
$ gcc -I/home/myincs -c src1.c; gcc -c src2.c
```

```
$ gcc -o exec.x src1.o src2.o
```

```
$ ./exec.x
```

Example: Hello

Main Program

```
#include "hello.h"

int main(void){
    print_hello();

    return 0;
}

hello.c
```

Header (Declarations)

```
#ifndef _HELLO_H_
#define _HELLO_H_

int print_hello(void);

#endif

hello.h
```

Functions (Definitions)

```
#include <stdio.h>

int print_hello(void){
    printf(„hello!\n“);

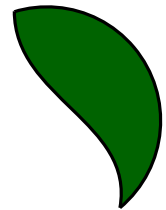
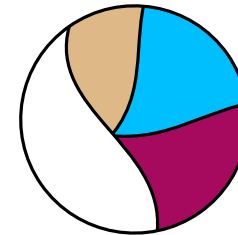
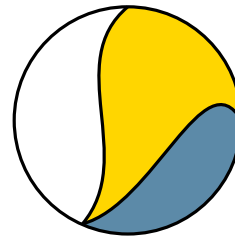
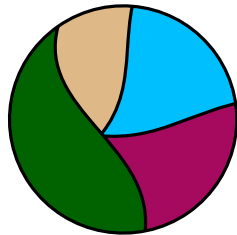
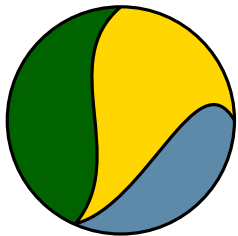
    return 0;
}

hello_fct.c
```

- Exercise: *hello*
 - Build objects *hello.o* *hello_fct.o*
 - Build executable by linking objects
 - **\$./hello**

Shared object files and Libraries

- Objects can be used by different executables.
- A **library** contains program parts (subroutines, classes, type definitions, ...) that can be used by different executables.
- **Static library**
 - Linked during building executable
- **Shared library**
 - Loaded during runtime



Module files

- Module files set/prepare following environment variables amongst others:

- `*_LIB_DIR = <library_directory>`

- `*_INC_DIR = <include_directory>`

- `LD_LIBRARY_PATH`

- Show module file setup with `$ module show <module_file>`

- Example: link NETCDF library

- Build executable:

```
$ module load compiler/intel
```

```
$ module load lib/netcdf
```

```
$ icc -I${NETCDF_INC_DIR} -c src1.c; gcc -c src2.c
```

```
$ icc -o exec.x src1.o src2.o -L${NETCDF_LIB_DIR} -lnetcdf
```

- Run executable:

```
$ module load lib/netcdf
```

```
$ ./exec.x
```



2. Makefile

Motivation

■ Interactively

- `$ gcc -o hello -I. hello.c hello_fct.c`
- Works as long as command history is active

■ Shell script

- `$./compile.sh`
- Does always recompile the whole code

■ Makefile

- `$ make`
- better organisation of code compilation
- recompiles only updated files,
make: `hello' is up to date.



Makefile

- `$ make [<target>]`
 - executes script named *Makefile* or *makefile*
 - without argument first rule in *Makefile* is executed

- Rule definition (format):

`target: prerequisites`

`<TAB>command`

Rule has to be applied, if any of these files is changed

To apply the rule, command has to be executed.

Only works with beginning tab stop!

```
hello: hello.h hello.c hello_fct.c
      gcc -o hello -I. hello.c hello_fct.c
```

Makefile.1

- Exercise: *Makefile.1*
 - define a second rule named `clean` to remove the executable

Rules - Content

■ Explicit rules

■ `hello.o:` rule to build target *hello.o*

■ Wildcards

■ `hello: *.c` *hello* depends on all files with suffix `.c` in this directory

■ Pattern rules

■ `%.o:` rule for all files with suffix `.o`

■ `%.o: %.c` `%` in prerequisites substitutes the same as `%` in the target

■ Phony Targets

■ `.PHONY: clean` target *clean* is nothing to build
`clean:`

Variables

■ Variable assignment

- = recursively expanded (referenced by reference)
- := simply expanded (referenced by value)
- = only if variable is not defined yet (no overwrite)</li- += add item to variable array

```
CC      = gcc
CFLAGS  = -I.
INC      := hello.h
OBJ      := hello.o
OBJ      += hello_fct.o
EXE      := hello

${EXE}: ${INC} ${OBJ}
        ${CC} -o ${EXE} ${CFLAGS} ${OBJ}

.PHONY: clean
clean:
        rm -f ${OBJ} ${EXE}</pre
```

Makefile.2

■ Exercise: *Makefile.2*

- „*hello.o* depends on *hello.h*“
- write an appropriate rule

Automatic Variables

- Automatic variables change from rule to rule

`$@` = target

`$<` = first item of prerequisites

`$$` = all items of prerequisites
separated by ' '

- Exercise: *Makefile.3*

- Use automatic variables
in rule to build *hello*

```
CC      ?= gcc
CFLAGS  = -I.
INC     := hello.h
OBJ     := hello.o
OBJ     += hello_fct.o
EXE     := hello

%.o: %.c ${INC}
        ${CC} -o $@ ${CFLAGS} -c $<

hello: hello.o hello_fct.o
        ${CC} -o ${EXE} ${CFLAGS} ${OBJ}

.PHONY: clean
clean:
        rm -f ${OBJ} ${EXE}
```

Makefile.3

Directives

- Conditions can be expressed by directives

- if VAR is (not) defined

```
ifdef/ifndef VAR
..
else
..
endif
```

- if A and B are (not) equal

```
ifeq/ifneq (A,B)
..
else
..
endif
```

- Example:

- Conditional assignment

```
CC ?= gcc
```

is equivalent to

```
ifndef CC
  CC = gcc
endif
```



Include

- Parts of *Makefile* can be outsourced
 - e.g. platform specific statements
- External makefile code, e.g. file *make.inc*, can be loaded in *Makefile* via
`include make.inc`

- Exercise: *hello_omp*
 - *make.inc.gnu* and *make.inc.intel* contain compiler specific makefile statements

```
CC      = gcc
CFLAGS = -I. -fopenmp

make.inc.gnu
```

- Adjust *Makefile.4*: include *make.inc* depending on `#{CC}`

```
CC      = icc
CFLAGS = -I. -openmp

make.inc.intel
```

- `$ module load compiler/gnu`
`$ make`
- `$ module load compiler/intel`
`$ make`

```
include make.inc.gnu

hello_omp:hello_omp.o
        #{CC} -o $@ #{CFLAGS} $<
```

Makefile.4

3. Parallelisation

Overview

- OpenMP is an **easy, portable** and **incremental** specification for node-level parallelisation
- Thread-based, shared memory, single-node (in contrast to MPI)
- How does it work?
 - Annotate the C/C++/FORTRAN source code with pragmas
 - The compiler transparently generates the necessary code
 - Non-parallel blocks are only executed by the main thread
 - Parallel blocks are handed to a team-of-threads and executed in parallel
- If the compiler has no support for OpenMP, or if you do not activate OpenMP, the pragmas will be ignored, the code will only run on a single core and still yield the correct result

Core syntax

- Most OpenMP pragmas apply to a „structured block“ or „parallel region“
 - A single instruction, or
 - A number of statements with a single entry point at the top and a single exit at the bottom

```
#pragma omp parallel  
{  
    // statements  
}
```

```
!$omp parallel  
    // statements  
!$omp end parallel
```

- Only statements inside a block marked with the „parallel“ clause will be executed in parallel
- It is allowed to abort the execution of the whole application within a structured block

Library functions

- Many of these will only work correctly inside of a parallel region!
- Get the number of threads in the current team: `omp_get_num_threads()`
- Get the ID of the current thread: `omp_get_thread_num()`
- Get the number of processors available: `omp_get_num_procs()`
- Get the elapsed wall clock time: `omp_get_wtime()`

Compiling

■ GCC

```
gcc -fopenmp -o openmp openmp.c
```

■ Intel

```
icc -openmp -o openmp openmp.c
```

```
# Get information about which loops were parallelized and which not  
icc -openmp -openmp-report 2 -o openmp openmp.c
```

```
# Get hints about weaknesses in the code regarding parallelisation  
icc -openmp -diag-enable sc-parallel3 -o openmp openmp.c
```

Hello World example

```
#include <stdio.h>
#include <stdlib.h>

#include <omp.h>

int main(int argc, char** argv)
{
    #pragma omp parallel
    {
        printf("Thread %i\n", omp_get_thread_num());
    }

    printf("All done!\n");

    return EXIT_SUCCESS;
}
```

Output:

```
Thread 0
Thread 3
Thread 2
Thread 1
All done!
```



Loops and reduction

```
#include <stdlib.h>
#include <stdio.h>
#include <omp.h>

static int VECTOR_LENGTH = 5;

int main (int argc, char * argv[])
{
    int i;
    double * v;
    double norm2 = 0.0, t1, tdiff;

    v = malloc (VECTOR_LENGTH * sizeof(double));

    t1 = omp_get_wtime();

    #pragma omp parallel for
    for (i=0; i < len; i++)
        v[i] = i;

    #pragma omp parallel for reduction(+:norm2)
    for(i=0; i < len; i++)
        norm2 += (v[i]*v[i]);

    tdiff = omp_get_wtime() - t1;

    printf ("norm2: %f Time:%f\n", norm2, tdiff);
    return 0;
}
```



Variable scopes

- A shared variable points to the same memory location for all threads
- A private variable points to a unique memory location for every thread

- Global variables are automatically shared
- Variables declared inside a parallel region are automatically private
- The control variable of a **do / for** construct is automatically private

- Variable handling can be controlled using the **private**, **shared**, **firstprivate** and **lastprivate** directives

Sections

```
#include <stdio.h>
#include <stdlib.h>

#include <omp.h>

int main(int argc, char** argv)
{
    #pragma omp sections
    {
        #pragma omp section
        {
            printf ("id = %d\n", omp_get_thread_num());
        }

        #pragma omp section
        {
            printf ("id = %d\n", omp_get_thread_num());
        }
    }

    return EXIT_SUCCESS;
}
```

Output 1:

```
id = 0
id = 0
```

Output 2:

```
id = 1
id = 0
```

Additional OpenMP clauses

- **critical:** Only one thread at a time executes the following region
- **atomic:** Only one thread at a time can update the following memory location
- **barrier:** The thread will wait until all other threads have reached the barrier
- **ordered:** Threads will execute the following region in serial order

- **single:** Only one of the threads executes the following region
- **master:** Only thread 0 executes the following region