# Tutorials for High Performance Scientific Computing

The Art of HPC, volume 4

Victor Eijkhout

2022, last formatted March 28, 2024
Book and slides download: https://tinyurl.com/vle394course
Public repository: https://bitbucket.org/VictorEijkhout/scientific-computing-public

## Preface

The field of high performance scientific computing requires, in addition to a broad of scientific knowledge and 'computing folkore', a number of practical skills. Call it the 'carpentry' aspect of the craft of scientific computing.

As a companion to the book 'Introduction to High Performance Scientific Computing', which covers background knowledge, here is then a set of tutorials on those practical skills that are important to becoming a successful high performance practitioner.

The tutorials should be done while sitting at a computer. Given the practice of scientific computing, they have a clear Unix bias.

**Public draft** This book is open for comments. What is missing or incomplete or unclear? Is material presented in the wrong sequence? Kindly mail me with any comments you may have.

You may have found this book in any of a number of places; the authoritative download location is `https://theartofhpc.com/` That page also links to `lulu.com` where you can get a nicely printed copy.

Victor Eijkhout `eijkhout@tacc.utexas.edu`
Research Scientist
Texas Advanced Computing Center
The University of Texas at Austin

# Contents

| | | | | Exercises | |
|---|---|---|---|---|---|
| lesson | Topic | Book | Slides | in-class | homework |
| 1 | Unix | 1 | unix | | 1.42 |
| 2 | Git | 5 | | | |
| 3 | Programming | 2 | programming | 2.3 | 2.4 |
| 4 | Libraries | 2 | programming | | |
| 5 | Debugging | 11 | | | root code |
| 6 | LaTeX | 15 | | | 15.13 |
| 7 | Make | 3 | | | 3.1, 3.2 |

Table 1: Timetable for the carpentry section of an HPC course.

A good part of being an effective practitioner of High Performance Scientific Computing is what can be called 'HPC Carpentry': a number of skills that are not scientific in nature, but that are still indispensable to getting your work done.

The vast majority of scientific programming is done on the Unix platform so we start out with a tutorial on Unix in chapter 1, followed by an explanation of the how your code is handled by compilers and linkers and such in chapter 2.

Next you will learn about some tools that will increase your productivity and effectiveness:

- The *Make* utility is used for managing the building of projects; chapter 3.
- Source control systems store your code in such a way that you can undo changes, or maintain multiple versions; in chapter 5 you will see the *subversion* software.
- Storing and exchanging scientific data becomes an important matter once your program starts to produce results; in chapter 7 you will learn the use of *HDF5*.
- Visual output of program data is important, but too wide a topic to discuss here in great detail; chapter 9 teaches you the basics of the *gnuplot* package, which is suitable for simple data plotting.

We also consider the activity of program development itself: chapter 10 considers how to code to prevent errors, and chapter 11 teaches you to debug code with *gdb*. Chapter 13 contains some information on how to write a program that uses more than one programming language.

Finally, chapter 15 teaches you about the LaTeX document system, so that you can report on your work in beautifully typeset articles.

Many of the tutorials are very hands-on. Do them while sitting at a computer!

Table 1 gives a proposed lesson outline for the carpentry section of a course. The article by Wilson [24] is a good read on the thinking behind this 'HPC carpentry'.

# Chapter 1

# Unix intro

Unix is an *Operating System (OS)*, that is, a layer of software between the user or a user program and the hardware. It takes care of files and screen output, and it makes sure that many processes can exist side by side on one system. However, it is not immediately visible to the user.

Most of this tutorial will work on any Unix-like platform, however, there is not just one Unix:

- Traditionally there are a few major flavors of Unix: ATT or *System V*, and *BSD*. Apple has Darwin which is close to BSD; IBM and HP have their own versions of Unix, and Linux is yet another variant. These days many Unix versions adhere to the *POSIX* standard. The differences between these are deep down and if you are taking this tutorial you probably won't see them for quite a while.
- Within Linux there are various *Linux distribution*s such as *Red Hat* or *Ubuntu*. These mainly differ in the organization of system files and again you probably need not worry about them.
- The issue of command shells will be discussed below. This actually forms the most visible difference between different computers 'running Unix'.

## 1.1 Shells

Most of the time that you use Unix, you are typing commands which are executed by an interpreter called the *shell*. The shell makes the actual OS calls. There are a few possible Unix shells available

- Most of this tutorial is focused on the `sh` or `bash` shell.
- For a variety of reasons (see for instance section 3.5), `bash`-like shells are to be preferred over the `csh` or `tcsh` shell. These latter ones will not be covered in this tutorial.
- Recent versions of the *Apple Mac OS* have the `zsh` as default. While this shell has many things in common with `bash`, we will point out differences explicitly.

## 1.2 Files and such

**Purpose.** In this section you will learn about the Unix file system, which consists of *directories* that store *files*. You will learn about *executable* files and commands for displaying data files.

### 1.2.1 Looking at files

**Purpose.** In this section you will learn commands for displaying file contents.

| Commands learned in this section | |
|---|---|
| `ls` | list files or directories |
| `touch` | create new/empty file or update existing file |
| `cat > filename` | enter text into file |
| `cp` | copy files |
| `mv` | rename files |
| `rm` | remove files |
| `file` | report the type of file |
| `cat filename` | display file |
| `head,tail` | display part of a file |
| `less,more` | incrementally display a file |

#### *1.2.1.1 `ls`*

Without any argument, the *`ls`* command gives you a listing of files that are in your present location.

**Exercise** 1.1. Type *`ls`*. Does anything show up?

*Intended outcome.* If there are files in your directory, they will be listed; if there are none, no output will be given. This is standard Unix behavior: no output does not mean that something went wrong, it only means that there is nothing to report.

**Exercise** 1.2. If the *`ls`* command shows that there are files, do `ls name` on one of those. By using an option, for instance `ls -s name` you can get more information about `name`.

*Things to watch out for.* If you mistype a name, or specify a name of a non-existing file, you'll get an error message.

The *`ls`* command can give you all sorts of information. In addition to the above `ls -s` for the size, there is `ls -l` for the 'long' listing. It shows (things we will get to later such as) ownership and permissions, as well as the size and creation date.

**Remark 1** *There are several dates associated with a file, corresponding to changes in content, changes in permissions, and access of any sort. The `stat` command gives all of them.*

#### *1.2.1.2 `cat`*

The *`cat`* command (short for 'concatenate') is often used to display files, but it can also be used to create some simple content.

**Exercise** 1.3. Type `cat > newfilename` (where you can pick any filename) and type some text. Conclude with `Control-d` on a line by itself: press the `Control` key and hold it while you press the `d` key. Now use *`cat`* to view the contents of that file: `cat newfilename`.

*Intended outcome.* In the first use of `cat`, text was appended from the terminal to a file; in the second the file was cat'ed to the terminal output. You should see on your screen precisely what you typed into the file.

*Things to watch out for.* Be sure to type `Control-d` as the first thing on the last line of input. If you really get stuck, `Control-c` will usually get you out. Try this: start creating a file with `cat > filename` and hit `Control-c` in the middle of a line. What are the contents of your file?

**Remark 2** *Instead of* `Control-d` *you will often see the notation* `^D`*. The capital letter is for historic reasons: you use the control key and the lowercase letter.*

#### *1.2.1.3 man*

The primary (though not always the most easily understood) source for unix commands is the *man* command, for 'manual'. The descriptions available this way are referred to as the *manual page*s.

**Exercise** 1.4. Read the man page of the `ls` command: `man ls`. Find out the size and the time / date of the last change to some files, for instance the file you just created.

*Intended outcome.* Did you find the `ls -s` and `ls -l` options? The first one lists the size of each file, usually in kilobytes, the other gives all sorts of information about a file, including things you will learn about later.

The *man* command puts you in a mode where you can view long text documents. This viewer is common on Unix systems (it is available as the *more* or *less* system command), so memorize the following ways of navigating: Use the space bar to go forward and the `u` key to go back up. Use `g` to go to the beginning fo the text, and `G` for the end. Use `q` to exit the viewer. If you really get stuck, `Control-c` will get you out.

**Remark 3** *If you already know what command you're looking for, you can use man to get online information about it. If you forget the name of a command,* `man -k keyword` *can help you find it.*

#### *1.2.1.4 touch*

The *touch* command creates an empty file, or updates the timestamp of a file if it already exists. Use `ls -l` to confirm this behavior.

#### *1.2.1.5 cp, mv, rm, ln*

The *cp* can be used for copying a file (or directories, see below): `cp file1 file2` makes a copy of `file1` and names it `file2`.

**Exercise** 1.5. Use `cp file1 file2` to copy a file. Confirm that the two files have the same contents. If you change the original, does anything happen to the copy?

*Intended outcome.* You should see that the copy does not change if the original changes or is deleted.

*Things to watch out for.* If `file2` already exists, you will get an error message.

A file can be renamed with *mv*, for 'move'.

**Exercise** 1.6. Rename a file. What happens if the target name already exists?

Files are deleted with `rm`. This command is dangerous: there is no undo. For this reason you can do `rm -i` (for 'interactive') which asks your confirmation for every file. See section 1.2.4 for more aggressive removing.

Sometimes you want to refer to a file from two locations. This is not the same as having a copy: you want to be able to edit either one, and have the other one change too. This can be done with *`ln`*: 'link'.

This snippet creates a file and a link to it:

```
$ echo contents > arose
$ cd mydir
$ ln ../arose anyothername
$ cat anyothername
contents
$ echo morestuff >> anyothername
$ cd ..
$ cat arose
contents
morestuff
```

#### *1.2.1.6 `head, tail`*

There are more commands for displaying a file, parts of a file, or information about a file.

**Exercise** 1.7. Do `ls /usr/share/words` or `ls /usr/share/dict/words` to confirm that a file with words exists on your system. Now experiment with the commands `head`, `tail`, `more`, and `wc` using that file.

*Intended outcome.* *`head`* displays the first couple of lines of a file, *`tail`* the last, and *`more`* uses the same viewer that is used for man pages. Read the man pages for these commands and experiment with increasing and decreasing the amount of output. The *`wc`* ('word count') command reports the number of words, characters, and lines in a file.

Another useful command is *`file`*: it tells you what type of file you are dealing with.

**Exercise** 1.8. Do `file foo` for various 'foo': a text file, a directory, or the `/bin/ls` command.

*Intended outcome.* Some of the information may not be intelligible to you, but the words to look out for are 'text', 'directory', or 'executable'.

At this point it is advisable to learn to use a text *editor*, such as *emacs* or *vi*.

### 1.2.2 Directories

**Purpose.** Here you will learn about the Unix directory tree, how to manipulate it and how to move around in it.

| Commands learned in this section | |
|---|---|
| `ls` | list the contents of directories |
| `mkdir` | make new directory |
| `cd` | change directory |
| `pwd` | display present working directory |

A unix file system is a tree of directories, where a directory is a container for files or more directories. We will display directories as follows:

```
/ ..... The root of the directory tree
├── bin ..... Binary programs
└── home ..... Location of users directories
```

The root of the Unix directory tree is indicated with a slash. Do `ls /` to see what the files and directories there are in the root. Note that the root is not the location where you start when you reboot your personal machine, or when you log in to a server.

**Exercise** 1.9. The command to find out your current working directory is *`pwd`*. Your home directory is your working directory immediately when you log in. Find out your home directory.

*Intended outcome.* You will typically see something like `/home/yourname` or `/Users/yourname`. This is system dependent.

Do `ls` to see the contents of the working directory. In the displays in this section, directory names will be followed by a slash: `dir/` but this character is not part of their name. You can get this output by using `ls -F`, and you can tell your shell to use this output consistently by stating `alias ls=ls -F` at the start of your session. Example:

```
/home/you/
├── adirectory/
└── afile
```

The command for making a new directory is *`mkdir`*.

**Exercise** 1.10. Make a new directory with *`mkdir`* `newdir` and view the current directory with `ls`.

*Intended outcome.* You should see this structure:

```
/home/you/
└── newdir/ ..... the new directory
```

**Remark 4** *If you need to create a directory several levels deep, you could*

```
mkdir sub1
cd sub1
mkdir sub2
cd sub2
## et cetera
```

*but it's shorter to use the* `-p` *option (for 'parent') and write:*

```
mkdir -p sub1/sub2/sub3
```

*which creates any needed intermediate levels.*

The command for going into another directory, that is, making it your working directory, is *cd* ('change directory'). It can be used in the following ways:

- `cd` Without any arguments, `cd` takes you to your home directory.
- `cd <absolute path>` An absolute path starts at the root of the directory tree, that is, starts with `/`. The `cd` command takes you to that location.
- `cd <relative path>` A relative path is one that does not start at the root. This form of the `cd` command takes you to `<yourcurrentdir>/<relative path>`.

**Exercise** 1.11. Do `cd newdir` and find out where you are in the directory tree with `pwd`. Confirm with `ls` that the directory is empty. How would you get to this location using an absolute path?

*Intended outcome.* `pwd` should tell you `/home/you/newdir`, and `ls` then has no output, meaning there is nothing to list. The absolute path is `/home/you/newdir`.

**Exercise** 1.12. Let's quickly create a file in this directory: *touch* `onefile`, and another directory: `mkdir otherdir`. Do `ls` and confirm that there are a new file and directory.

*Intended outcome.* You should now have:

```
/home/you/
└── newdir/.....................................................you are here
    ├── onefile
    └── otherdir/
```

The `ls` command has a very useful option: with `ls -a` you see your regular files and hidden files, which have a name that starts with a dot. Doing `ls -a` in your new directory should tell you that there are the following files:

```
/home/you/
└── newdir/.....................................................you are here
    ├── .
    ├── ..
    ├── onefile
    └── otherdir/
```

The single dot is the current directory, and the double dot is the directory one level back.

**Exercise** 1.13. Predict where you will be after `cd ./otherdir/..` and check to see if you were right.

*Intended outcome.* The single dot sends you to the current directory, so that does not change anything. The `otherdir` part makes that subdirectory your current working directory. Finally, `..` goes one level back. In other words, this command puts your right back where you started.

Since your home directory is a special place, there are shortcuts for `cd`'ing to it: `cd` without arguments, `cd ~`, and `cd $HOME` all get you back to your home.

Go to your home directory, and from there do `ls newdir` to check the contents of the first directory you created, without having to go there.

**Exercise** 1.14. What does `ls ..` do?

*Intended outcome.* Recall that `..` denotes the directory one level up in the tree: you should see your own home directory, plus the directories of any other users.

Let's practice the use of the single and double dot directory shortcuts.

**Exercise** 1.15. From your home directory:

```
mkdir -p sub1/sub2/sub3
cd sub1/sub2/sub3
touch a
```

You now have a file `sub1/sub2/sub3/a`

1. How do you move it to `sub1/sub2/a`?
2. Go: `cd sub1/sub2`
   How do you now move the file to `sub1/a`?
3. Go to your home directory: `cd`
   How do you move `sub1/a` to here?

**Exercise** 1.16. Can you use `ls` to see the contents of someone else's home directory? In the previous exercise you saw whether other users exist on your system. If so, do `ls ../thatotheruser`.

*Intended outcome.* If this is your private computer, you can probably view the contents of the other user's directory. If this is a university computer or so, the other directory may very well be protected – permissions are discussed in the next section – and you get `ls: ../otheruser: Permission denied`.

Make an attempt to move into someone else's home directory with `cd`. Does it work?

You can make copies of a directory with `cp`, but you need to add a flag to indicate that you recursively copy the contents: `cp -r`. Make another directory `somedir` in your home so that you have

```
/home/you/
├── newdir/ ..... you have been working in this one
└── somedir/ ..... you just created this one
```

What is the difference between `cp -r newdir somedir` where `somedir` is an exiting directory, and `cp -r newdir thirddir` where `thirddir` is not an existing directory?

### 1.2.3 Permissions

**Purpose.** In this section you will learn about how to give various users on your system permission to do (or not to do) various things with your files.

Unix files, including directories, have permissions, indicating 'who can do what with this file'. Actions that can be performed on a file fall into three categories:

- reading `r`: any access to a file (displaying, getting information on it) that does not change the file;
- writing `w`: access to a file that changes its content, or even its metadata such as 'date modified';
- executing `x`: if the file is executable, to run it; if it is a directory, to enter it.

The people who can potentially access a file are divided into three classes too:

- the user `u`: the person owning the file;
- the group `g`: a group of users to which the owner belongs;
- other `o`: everyone else.

(For more on groups and ownership, see section 1.13.3.)

The nine permissions are rendered in sequence

| user | group | other |
|---|---|---|
| rwx | rwx | rwx |

For instance `rw-r--r--` means that the owner can read and write a file, the owner's group and everyone else can only read.

Permissions are also rendered numerically in groups of three bits, by letting `r` = 4, `w` = 2, `x` = 1:

| rwx |
|---|
| 421 |

Common codes are 7 = `rwx` and 6 = `rw`. You will find many files that have permissions 755 which stands for an executable that everyone can run, but only the owner can change, or 644 which stands for a data file that everyone can see but again only the owner can alter. You can set permissions by the *chmod* command:

```
chmod <permissions> file         # just one file
chmod -R <permissions> directory # directory, recursively
```

Examples:

```
chmod 766 file  # set to rwxrw-rw-
chmod g+w file  # give group write permission
chmod g=rx file # set group permissions
chod o-w  file  # take away write permission from others
chmod o=  file  # take away all permissions from others.
chmod g+r,o-x file # give group read permission
                   # remove other execute permission
```

The man page gives all options.

**Exercise** 1.17. Make a file `foo` and do `chmod u-r foo`. Can you now inspect its contents? Make the file readable again, this time using a numeric code. Now make the file readable to your classmates. Check by having one of them read the contents.

*Intended outcome.* 1. A file is only accessible by others if the surrounding folder is readable. Can you figure out how to do this? 2. When you've made the file 'unreadable' by yourself, you can still `ls` it, but not `cat` it: that will give a 'permission denied' message.

Make a file `com` with the following contents:

```
#!/bin/sh
echo "Hello world!"
```

This is a legitimate shell script. What happens when you type `./com`? Can you get the script executed?

In the three permission categories it is clear who 'you' and 'others' refer to. How about 'group'? We'll go into that in section 1.13.

**Exercise** 1.18. Suppose you're an instructor and you want to make a 'dropbox' directory for students to deposit homework assignments in. What would be an appropriate mode for that directory? (Assume that you have co-teachers that are in your group, and who also need to be able to see the contents. In other words, group permission should be identical to the owner permission.)

**Remark 5** *There are more obscure permissions. For instance the* `setuid` *bit declares that the program should run with the permissions of the creator, rather than the user executing it. This is useful for system utilities such* `passwd` *or* `mkdir`*, which alter the password file and the directory structure, for which* root privileges *are needed. Thanks to the setuid bit, a user can run these programs, which are then so designed that a user can only make changes to their own password entry, and their own directories, respectively. The setuid bit is set with* `chmod`*:* `chmod 4ugo file`.

### 1.2.4 Wildcards

You already saw that `ls filename` gives you information about that one file, and `ls` gives you all files in the current directory. To see files with certain conditions on their names, the *wildcard* mechanism exists. The following wildcards exist:

| | |
|---|---|
| `*` | any number of characters |
| `?` | any character. |

Example:

```
%% ls
s       sk      ski     skiing  skill
%% ls ski*
ski     skiing  skill
```

The second option lists all files whose name start with `ski`, followed by any number of other characters'; below you will see that in different contexts `ski*` means 'sk followed by any number of i characters'. Confusing, but that's the way it is.

You can use `rm` with wildcards, but this can be dangerous.

```
rm -f foo    ## remove foo if it exists
rm -r foo    ## remove directory foo with everything in it
rm -rf foo/* ## delete all contents of foo
```

*Zsh note.* Removing with a wildcard `rm foo*` is an error of there are no such files. Set `setopt +o nomatch` to allow no matches to occur.

## 1.3 Text searching and regular expressions

**Purpose.** In this section you will learn how to search for text in files.

For this section you need at least one file that contains some amount of text. You can for instance get random text from http://www.lipsum.com/feed/html.

The *grep* command can be used to search for a text expression in a file.

**Exercise** 1.19. Search for the letter `q` in your text file with `grep q yourfile` and search for it in all files in your directory with `grep q *`. Try some other searches.

*Intended outcome.* In the first case, you get a listing of all lines that contain a `q`; in the second case, `grep` also reports what file name the match was found in: `qfile:this line has q in it`.

*Things to watch out for.* If the string you are looking for does not occur, `grep` will simply not output anything. Remember that this is standard behavior for Unix commands if there is nothing to report.

In addition to searching for literal strings, you can look for more general expressions.

| | |
|---|---|
| `^` | the beginning of the line |
| `$` | the end of the line |
| `.` | any character |
| `*` | any number of repetitions |
| `[xyz]` | any of the characters `xyz` |

This looks like the wildcard mechanism you just saw (section 1.2.4) but it's subtly different. Compare the example above with:

```
%% cat s
sk
ski
skill
skiing
%% grep "ski*" s
sk
ski
skill
skiing
```

In the second case you search for a string consisting of `sk` and any number of `i` characters, including zero of them.

Some more examples: you can find

- All lines that contain the letter 'q' with `grep q yourfile`;
- All lines that start with an 'a' with `grep "^a" yourfile` (if your search string contains special characters, it is a good idea to use quote marks to enclose it);
- All lines that end with a digit with `grep "[0-9]$" yourfile`.

**Exercise** 1.20. Construct the search strings for finding

- lines that start with an uppercase character, and
- lines that contain exactly one character.

*Intended outcome.* For the first, use the range characters `[]`, for the second use the period to match any character.

**Exercise** 1.21. Add a few lines `x = 1`, `x  = 2`, `x   = 3` (that is, have different numbers of spaces between `x` and the equals sign) to your test file, and make `grep` commands to search for all assignments to `x`.

The characters in the table above have special meanings. If you want to search that actual character, you have to *escape* it.

**Exercise** 1.22. Make a test file that has both `abc` and `a.c` in it, on separate lines. Try the commands `grep "a.c" file`, `grep a\.c file`, `grep "a\.c" file`.

*Intended outcome.* You will see that the period needs to be escaped, and the search string needs to be quoted. In the absence of either, you will see that `grep` also finds the `abc` string.

### 1.3.1 Cutting up lines with `cut`

Another tool for editing lines is *`cut`*, which will cut up a line and display certain parts of it. For instance,

```
cut -c 2-5 myfile
```

will display the characters in position 2–5 of every line of `myfile`. Make a test file and verify this example.

Maybe more useful, you can give `cut` a delimiter character and have it split a line on occurrences of that delimiter. For instance, your system will mostly likely have a file `/etc/passwd` that contains user information[1], with every line consisting of fields separated by colons. For instance:

```
daemon:*:1:1:System Services:/var/root:/usr/bin/false
nobody:*:-2:-2:Unprivileged User:/var/empty:/usr/bin/false
root:*:0:0:System Administrator:/var/root:/bin/sh
```

The seventh and last field is the login shell of the user; `/bin/false` indicates that the user is unable to log in.

You can display users and their login shells with:

```
cut -d ":" -f 1,7 /etc/passwd
```

This tells `cut` to use the colon as delimiter, and to print fields 1 and 7.

## 1.4 Other useful commands: `tar`

The *`tar`* command stands for 'tape archive', that is, it was originally meant to package files on a tape. (The 'archive' part derives from the *`ar`* command.) These days, it's used to package files together for distribution on web sites and such: if you want to publish a library of hundreds of files this bundles them into a single file.

The two most common options are for

1. creating a tar file:

   ```
   tar fc package.tar directory_with_stuff
   ```

   pronounced 'tar file create', and

1. This is traditionally the case; on Mac OS information about users is kept elsewhere and this file only contains system services.

2. unpacking a tar file:

```
tar fx package.tar
# this creates the directory that was packaged
```

pronounced 'tar file extract'.

Text files can often be compressed to a large extent, so adding the `z` compressiong for *gzip* is a good idea:

```
tar fcz package.tar.gz directory_with_stuff
tar fx package.tar.gz
```

Naming the 'gzipped' file `package.tgz` is also common.

## 1.5 Command execution

If you type something in the shell, you are actually asking the underlying interpreter to execute a command. Some commands are built-ins, others can be names of programs that are stored somewhere, in a system location or in your own account. This section will go into the mechanisms.

**Remark 6** *Like any good programming language, the shell language as comments. Any line starting with a hash character* `#` *is ignored.*

*Zsh note.* In Apple's zsh, the comment character is disabled. Do

```
setopt interactivecomments
```

to enable it.

### 1.5.1 Search paths

| Commands learned in this section | |
|---|---|
| *which* | location of executable command |
| type | description of commands, functions, ... |

**Purpose.** In this section you will learn how Unix determines what to do when you type a command name.

If you type a command such as `ls`, the shell does not just rely on a list of commands: it will actually go searching for a program by the name `ls`. This means that you can have multiple different commands with the same name, and which one gets executed depends on which one is found first.

**Exercise** 1.23. What you may think of as 'Unix commands' are often just executable files in a system directory. Do *which* `ls`, and do an `ls -l` on the result.

*Intended outcome.* The location of `ls` is something like `/bin/ls`. If you `ls` that, you will see that it is probably owned by root. Its executable bits are probably set for all users.

The locations where unix searches for commands is the *search path*, which is stored in the *environment variable* (for more details see below) *PATH*.

**Exercise** 1.24. Do `echo $PATH`. Can you find the location of `cd`? Are there other commands in the same location? Is the current directory '`.`' in the path? If not, do `export PATH=".:$PATH"`. Now create an executable file `cd` in the current director (see above for the basics), and do `cd`.

*Intended outcome.* The path will be a list of colon-separated directories, for instance `/usr/bin:/usr/local/bin:/usr/X11R6/bin`. If the working directory is in the path, it will probably be at the end: `/usr/X11R6/bin:.` but most likely it will not be there. If you put '`.`' at the start of the path, unix will find the local `cd` command before the system one.

Some people consider having the working directory in the path a security risk. If your directory is writable, someone could put a malicious script named `cd` (or any other system command) in your directory, and you would execute it unwittingly.

The safest way to execute a program in the current directory is:

```
./my_program
```

This holds both for compiled programs and shell scripts; section 1.9.1.

**Remark 7** *Not all Unix commands correspond to executables. The `type` command gives more information than `which`:*

```
$ type echo
echo is a shell builtin
$ type \ls
ls is an alias for ls -F
$ unalias ls
$ type ls
ls is /bin/ls
$ type module
module is a shell function from /usr/local/Cellar/lmod/8.7.2/init/zsh
```

### 1.5.2 Aliases

It is possible to define your own commands as aliases of existing commands.

**Exercise** 1.25. Do `alias chdir=cd` and convince yourself that now `chdir` works just like `cd`. Do `alias rm='rm -i'`; look up the meaning of this in the man pages. Some people find this alias a good idea; can you see why?

*Intended outcome.* The `-i` 'interactive' option for `rm` makes the command ask for confirmation before each delete. Since unix does not have a trashcan that needs to be emptied explicitly (as on Windows or the Mac OS), this can be a good idea.

### 1.5.3 Command sequencing

There are various ways of having multiple commands on a single commandline.

#### 1.5.3.1 Simple sequencing

First of all, you can type

```
command1 ; command2
```

This is convenient if you repeat the same two commands a number of times: you only need to up-arrow once to repeat them both.

There is a problem: if you type

```
cc -o myprog myprog.c ; ./myprog
```

and the compilation fails, the program will still be executed, using an old version of the executable if that exists. This is very confusing.

A better way is:

```
cc -o myprog myprog.c && ./myprog
```

which only executes the second command if the first one was successful.

#### 1.5.3.2 Pipelining

Instead of taking input from a file, or sending output to a file, it is possible to connect two commands together, so that the second takes the output of the first as input. The syntax for this is `cmdone | cmdtwo`; this is called a pipeline. For instance, `grep a yourfile | grep b` finds all lines that contains both an `a` and a `b`.

**Exercise** 1.26. Construct a pipeline that counts how many lines there are in your file that contain the string `th`. Use the `wc` command (see above) to do the counting.

#### 1.5.3.3 Backquoting

There are a few more ways to combine commands. Suppose you want to present the result of `wc` a bit nicely. Type the following command

```
echo The line count is wc -l foo
```

where `foo` is the name of an existing file. The way to get the actual line count echoed is by the *backquote*:

```
echo The line count is `wc -l foo`
```

Anything in between backquotes is executed before the rest of the command line is evaluated.

**Exercise** 1.27. The way `wc` is used here, it prints the file name. Can you find a way to prevent that from happening?

There is another mechanism for out-of-order evaluation:

```
echo "There are $( cat Makefile | wc -l ) lines"
```

This mechanism makes it possible to nest commands, but for compatibility and legacy purposes backquotes may still be preferable when nesting is not neeeded.

#### 1.5.3.4 *Grouping in a subshell*

Suppose you want to apply output redirection to a couple of commands in a row:

```
configure ; make ; make install > installation.log 2>&1
```

This only catches the last command. You could for instance group the three commands in a subshell and catch the output of that:

```
( configure ; make ; make install ) > installation.log 2>&1
```

### 1.5.4 Exit status

Commands can fail. If you type a single command on the command line, you see the error, and you act accordingly when you type the next command. When that failing command happens in a script, you have to tell the script how to act accordingly. For this, you use the *exit status* of the command: this is a value (zero for success, nonzero otherwise) that is stored in an internal variable, and that you can access with `$?`.

Example. Suppose we have a directory that is not writable

```
[testing] ls -ld nowrite/
dr-xr-xr-x  2 eijkhout  506  68 May 19 12:32 nowrite//
[testing] cd nowrite/
```

and write try to create a file there:

```
[nowrite] cat ../newfile
#!/bin/bash
touch $1
echo "Created file: $1"
[nowrite] newfile myfile
bash: newfile: command not found
[nowrite] ../newfile myfile
touch: myfile: Permission denied
Created file: myfile
[nowrite] ls
[nowrite]
```

The script reports that the file was created even though it wasn't.

Improved script:

```
[nowrite] cat ../betterfile
#!/bin/bash
touch $1
if [ $? -eq 0 ] ; then
    echo "Created file: $1"
else
    echo "Problem creating file: $1"
fi

[nowrite] ../betterfile myfile
touch: myfile: Permission denied
Problem creating file: myfile
```

### 1.5.5 Processes and jobs

| Commands learned in this section | |
|---|---|
| `ps` | list (all) processes |
| `kill` | kill a process |
| `CTRL-c` | kill the foreground job |
| `CTRL-z` | suspect the foreground job |
| `jobs` | give the status of all jobs |
| `fg` | bring the last suspended job to the foreground |
| `fg %3` | bring a specific job to the foreground |
| `bg` | run the last suspended job in the background |

The Unix operating system can run many programs at the same time, by rotating through the list and giving each only a fraction of a second to run each time. The command *`ps`* can tell you everything that is currently running.

**Exercise** 1.28. Type `ps`. How many programs are currently running? By default `ps` gives you only programs that you explicitly started. Do `ps guwax` for a detailed list of everything that is running. How many programs are running? How many belong to the root user, how many to you?

*Intended outcome.* To count the programs belonging to a user, pipe the `ps` command through an appropriate `grep`, which can then be piped to `wc`.

In this long listing of `ps`, the second column contains the *process numbers*. Sometimes it is useful to have those: if a program misbehaves you can *`kill`* it with

```
kill 123456
```

where `12345` is the process number.

The `cut` command explained above can cut certain position from a line: type `ps guwax | cut -c 10-14`.

To get dynamic information about all running processes, use the `top` command. Read the man page to find out how to sort the output by CPU usage.

Processes that are started in a shell are known as *jobsjob (unix)*. In addition to the process number, they have a job number. We will now explore manipulating jobs.

When you type a command and hit return, that command becomes, for the duration of its run, the *foreground process*. Everything else that is running at the same time is a *background process*.

Make an executable file `hello` with the following contents:

```
#!/bin/sh
while [ 1 ] ; do
  sleep 2
  date
done
```

and type `./hello`.

**Exercise** 1.29. Type `Control-z`. This suspends the foreground process. It will give you a number like `[1]` or `[2]` indicating that it is the first or second program that has been suspended or put in the background. Now type `bg` to put this process in the background. Confirm that there is no foreground process by hitting return, and doing an `ls`.

*Intended outcome.* After you put a process in the background, the terminal is available again to accept foreground commands. If you hit return, you should see the command prompt. However, the background process still keeps generating output.

**Exercise** 1.30. Type `jobs` to see the processes in the current session. If the process you just put in the background was number 1, type `fg %1`. Confirm that it is a foreground process again.

*Intended outcome.* If a shell is executing a program in the foreground, it will not accept command input, so hitting return should only produce blank lines.

**Exercise** 1.31. When you have made the `hello` script a foreground process again, you can kill it with `Control-c`. Try this. Start the script up again, this time as `./hello &` which immediately puts it in the background. You should also get output along the lines of `[1] 12345` which tells you that it is the first job you put in the background, and that `12345` is its process ID. Kill the script with `kill %1`. Start it up again, and kill it by using the process number.

*Intended outcome.* The command `kill 12345` using the process number is usually enough to kill a running program. Sometimes it is necessary to use `kill -9 12345`.

### 1.5.6 Shell customization

Above it was mentioned that `ls -F` is an easy way to see which files are regular, executable, or directories; by typing `alias ls='ls -F'` the `ls` command will automatically expanded to `ls -F` every time it is invoked. If you would like this behavior in every login session, you can add the `alias` command to your `.profile` file. Other shells than `sh`/`bash` have other files for such customizations.

## 1.6 Input/output Redirection

**Purpose.** In this section you will learn how to feed one command into another, and how to connect commands to input and output files.

So far, the unix commands you have used have taken their input from your keyboard, or from a file named on the command line; their output went to your screen. There are other possibilities for providing input from a file, or for storing the output in a file.

### 1.6.1 Input redirection

The `grep` command had two arguments, the second being a file name. You can also write `grep string < yourfile`, where the less-than sign means that the input will come from the named file, `yourfile`. This is known as *input redirection*.

### 1.6.2 Standard files

Unix has three standard files that handle input and output:

| Standard file | |
|---|---|
| `stdin` | is the file that provides input for processes. |
| `stdout` | is the file where the output of a process is written. |
| `stderr` | is the file where error output is written. |

In an interactive session, all three files are connected to the user terminal. Using input or output redirection then means that the input is taken or the output sent to a different file than the terminal.

### 1.6.3 Output redirection

Just as with the input, you can redirect the output of your program. In the simplest case, `grep string yourfile > outfile` will take what normally goes to the terminal, and *redirect* the output to `outfile`. The output file is created if it didn't already exist, otherwise it is overwritten. (To append, use `grep text yourfile >> outfile`.)

**Exercise** 1.32. Take one of the grep commands from the previous section, and send its output to a file. Check that the contents of the file are identical to what appeared on your screen before. Search for a string that does not appear in the file and send the output to a file. What does this mean for the output file?

*Intended outcome.* Searching for a string that does not occur in a file gives no terminal output. If you redirect the output of this `grep` to a file, it gives a zero size file. Check this with `ls` and `wc`.

**Exercise** 1.33. Generate a text file that contains your information:

```
My user name is:
eijkhout
My home directory is:
/users/eijkhout
I made this script on:
isp.tacc.utexas.edu
```

where you use the commands `whoami`, `pwd`, `hostname`.
Bonus points if you can get the 'prompt' and output on the same line.
Hint: see section 1.5.3.3.

Sometimes you want to run a program, but ignore the output. For that, you can redirect your output to the system *null device*: `/dev/null`.

```
yourprogram >/dev/null
```

Here are some useful idioms:

| Idiom | |
|---|---|
| `program 2>/dev/null` | send only errors to the null device |
| `program >/dev/null 2>&1` | send output to dev-null, and errors to output<br>Note the counterintuitive sequence of specifications! |
| `program 2>&1 \| less` | send output and errors to `less` |

## 1.7 Shell environment variables

Above you encountered `PATH`, which is an example of an shell, or environment, variable. These are variables that are known to the shell and that can be used by all programs run by the shell. While `PATH` is a built-in variable, you can also define your own variables, and use those in shell scripting.

Shell variables are roughly divided in the following categories:

- Variables that are specific to the shell, such as `HOME` or `PATH`.
- Variables that are specific to some program, such as `TEXINPUTS` for TEX/LATEX.
- Variables that you define yourself; see next.
- Variables that are defined by control structures such as *`for`*; see below.

You can see the full list of all variables known to the shell by typing *`env`*.

**Remark 8** *This does not include variables you define yourself, unless you* `export` *them; see below.*

**Exercise** 1.34. Check on the value of the `PATH` variable by typing `echo $PATH`. Also find the value of `PATH` by piping `env` through `grep`.

We start by exploring the use of this dollar sign in relation to shell variables.

### 1.7.1 Use of shell variables

You can get the value of a shell variable by prefixing it with a dollar sign. Type the following and inspect the output:

```
echo x
echo $x
x=5
echo x
echo $x
```

You see that the shell treats everything as a string, unless you explicitly tell it to take the value of a variable, by putting a dollar in front of the name. A variable that has not been previously defined will print as a blank string.

Shell variables can be set in a number of ways. The simplest is by an assignment as in other programming languages.

When you do the next exercise, it is good to bear in mind that the shell is a text based language.

**Exercise** 1.35. Type `a=5` on the commandline. Check on its value with the `echo` command.
Define the variable `b` to another integer. Check on its value.
Now explore the values of `a+b` and `$a+$b`, both by `echo`'ing them, or by first assigning them.

*Intended outcome.* The shell does not perform integer addition here: instead you get a string with a plus-sign in it. (You will see how to do arithmetic on variables in section 1.10.1.)

*Things to watch out for.* Beware not to have space around the equals sign; also be sure to use the dollar sign to print the value.

### 1.7.2 Exporting variables

A variable set this way will be known to all subsequent commands you issue in this shell, but not to commands in new shells you start up. For that you need the *export* command. Reproduce the following session (the square brackets form the command prompt):

```
[] a=20
[] echo $a
20
[] /bin/bash
[] echo $a

[] exit
exit
[] export a=21
[] /bin/bash
[] echo $a
21
[] exit
```

You can also temporarily set a variable. Replay this scenario:

1. Find an environment variable that does not have a value:

   ```
   [] echo $b

   []
   ```

2. Write a short shell script to print this variable:

   ```
   [] cat > echob
   #!/bin/bash
   echo $b
   ```

   and of course make it executable: `chmod +x echob`.
3. Now call the script, preceding it with a setting of the variable `b`:

   ```
   [] b=5 ./echob
   5
   ```

   The syntax where you set the value, as a prefix without using a separate command, sets the value just for that one command.

4. Show that the variable is still undefined:

```
[] echo $b

[]
```

That is, you defined the variable just for the execution of a single command.

In section 1.8 you will see that the `for` construct also defines a variable; section 1.9.1 shows some more built-in variables that apply in shell scripts.

If you want to un-set an environment variable, there is the *unset* command.

## 1.8 Control structures

Like any good programming system, the shell has some control structures. Their syntax takes a bit of getting used to. (Different shells have different syntax; in this tutorial we only discuss the bash shell.

### 1.8.1 Conditionals

The *conditional* of the bash shell is predictably called *if*, and it can be written over several lines:

```
if [ $PATH = "" ] ; then
  echo "Error: path is empty"
fi
```

or on a single line:

```
if [ `wc -l file` -gt 100 ] ; then echo "file too long" ; fi
```

(The backquote is explained in section 1.5.3.3.) There are a number of tests defined, for instance `-f somefile` tests for the existence of a file. Change your script so that it will report `-1` if the file does not exist.

The syntax of this is finicky:

- *if* and *elif* are followed by a conditional, followed by a semicolon.
- The brackets of the conditional need to have spaces surrounding them.
- There is no semicolon after *then* of *else*: they are immediately followed by some command.

**Exercise** 1.36. Bash conditionals have an *elif* keyword. Can you predict the error you get from this:

```
if [ something ] ; then
  foo
else if [ something_else ] ; then
  bar
fi
```

Code it out and see if you were right.

*Zsh note.* The *zsh* shell has an extended conditional syntax with double square brackets. For instance, pattern matching:

```
if [[ $myvar == *substring* ]] ; then ....
```

### 1.8.2 Looping

In addition to conditionals, the shell has loops. A *for* loop looks like

```
for var in listofitems ; do
  something with $var
done
```

This does the following:

- for each item in `listofitems`, the variable `var` is set to the item, and
- the loop body is executed.

As a simple example:

```
for x in a b c ; do echo $x ; done
a
b
c
```

In a more meaningful example, here is how you would make backups of all your `.c` files:

```
for cfile in *.c ; do
  cp $cfile $cfile.bak
done
```

Shell variables can be manipulated in a number of ways. Execute the following commands to see that you can remove trailing characters from a variable:

```
[] a=b.c
[] echo ${a%.c}
b
```

(See the section 1.10 on expansion.) With this as a hint, write a loop that renames all your `.c` files to `.x` files.

The above construct loops over words, such as the output of `ls`. To do a numeric loop, use the command *seq*:

```
[shell:474] seq 1 5
1
2
3
4
5
```

Looping over a sequence of numbers then typically looks like

```
for i in `seq 1 ${HOWMANY}` ; do echo $i ; done
```

Note the *backtick*, which is necessary to have the `seq` command executed before evaluating the loop.

You can break out of a loop with *break*; this can even have a numeric argument indicating how many levels of loop to break out of.

## 1.9 Scripting

The unix shells are also programming environments. You will learn more about this aspect of unix in this section.

### 1.9.1 How to execute scripts

It is possible to write programs of unix shell commands. First you need to know how to put a program in a file and have it be executed. Make a file `script1` containing the following two lines:

```
#!/bin/bash
echo "hello world"
```

and type `./script1` on the command line. Result? Make the file executable and try again.

*Zsh note.* Bash scripts If you use the `zsh`, but you have `bash` scripts that you wrote in the past, they will keep working. The 'hash-bang' line determines which shell executes the script, and it is perfectly possible to have `bash` in your script, while using `zsh` for interactive use.

In order write scripts that you want to invoke from anywhere, people typically put them in a directory `bin` in their home directory. You would then add this directory to your *search path*, contained in *PATH*; see section 1.5.1.

### 1.9.2 Script arguments

You can invoke a shell script with options and arguments:

```
./my_script -a file1 -t -x file2 file3
```

You will now learn how to incorporate this functionality in your scripts.

First of all, all commandline arguments and options are available as variables `$1`,`$2` et cetera in the script, and the number of command line arguments is available as `$#`:

```
#!/bin/bash

echo "The first argument is $1"
echo "There were $# arguments in all"
```

Formally:

| variable | meaning |
|---|---|
| `$#` | number of arguments |
| `$0` | the name of the script |
| `$1,$2,...` | the arguments |
| `$*,$@` | the list of all arguments |

**Exercise** 1.37. Write a script that takes as input a file name argument, and reports how many lines are in that file.
Edit your script to test whether the file has less than 10 lines (use the `foo -lt bar` test), and if it does, `cat` the file. Hint: you need to use backquotes inside the test.
Add a test to your script so that it will give a helpful message if you call it without any arguments.

The standard way to parse argument is using the *`shift`* command, which pops the first argument off the list of arguments. Parsing the arguments in sequence then involves looking at `$1`, shifting, and looking at the new `$1`.

Code:

```
// arguments.sh
while [ $# -gt 0 ] ; do
    echo "argument: $1"
    shift
done
```

Output

```
[code/shell] arguments:

missing snippet
code/shell/arguments.runout :
looking in codedir=code missing
snippet code/shell/arguments.runout
: looking in codedir=code
```

**Exercise** 1.38. Write a script `say.sh` that prints its text argument. However, if you invoke it with

```
./say.sh -n 7 "Hello world"
```

it should be print it as many times as you indicated. Using the option `-u`:

```
./say.sh -u -n 7 "Goodbye cruel world"
```

should print the message in uppercase. Make sure that the order of the arguments does not matter, and give an error message for any unrecognized option.

The variables `$@` and `$*` have a different behavior with respect to double quotes. Let's say we evaluate `myscript "1 2" 3`, then

- Using `$*` is the list of arguments after removing quotes: `myscript 1 2 3`.
- Using `"$*"` is the list of arguments, with quotes removed, in quotes: `myscript "1 2 3"`.
- Using `"$@"` preserved quotes: `myscript "1 2" 3`.

### 1.9.3 Error handling

Scripts, like any other type of program, can fail with some runtime condition.

1. If there is no clear error message, you can at least rerun your script with a line

   ```
   set -x
   ```

   which echoes each command to the terminal before execution
2. Each script has a return code, contained in the variable `$?` which you can inspect.
3. It is also possible that some command in the script fails, but the script continues running. You can prevent this with

   ```
   set -e
   ```

which makes the script abort if any command fails. The additional option

```
set -o pipefail
```

will catch errors in a pipeline.

Here is an idiom for being a little more informative about errors:

```
errcode=0
some_command || errcode=$?
if [ $errcode -ne 0 ] ; then
  echo "ERROR: some_command failed with code=$errcode"
  exit $errcode
fi
```

The crucial second line contains an 'or' condition: either *some_command* succeeds, or you set *errcode* to its exit code. This conjunction always succeeds, so now you can inspect the exit code.

## 1.10 Expansion

The shell performs various kinds of expansion on a command line, that is, replacing part of the commandline with different text.

Brace expansion:

```
[] echo a{b,cc,ddd}e
abe acce addde
```

This can for instance be used to delete all extension of some base file name:

```
[] rm tmp.{c,s,o}  # delete tmp.c tmp.s tmp.o
```

Tilde expansion gives your own, or someone else's home directory:

```
[] echo ~
/share/home/00434/eijkhout
[] echo ~eijkhout
/share/home/00434/eijkhout
```

Parameter expansion gives the value of shell variables:

```
[] x=5
[] echo $x
5
```

Undefined variables do not give an error message:

```
[] echo $y
```

There are many variations on parameter expansion. Above you already saw that you can strip trailing characters:

```
[] a=b.c
[] echo ${a%.c}
b
```

Here is how you can deal with undefined variables:

```
[] echo ${y:-0}
0
```

The backquote mechanism (section 1.5.3.3 above) is known as command substitution. It allows you to evaluate part of a command and use it as input for another. For example, if you want to ask what type of file the command `ls` is, do

```
[] file `which ls`
```

This first evaluates `which ls`, giving `/bin/ls`, and then evaluates `file /bin/ls`. As another example, here we backquote a whole pipeline, and do a test on the result:

```
[] echo 123 > w
[] cat w
123
[] wc -c w
       4 w
[] if [ `cat w | wc -c` -eq 4 ] ; then echo four ; fi
four
```

### 1.10.1 Arithmetic expansion

Unix shell programming is very much oriented towards text manipulation, but it is possible to do arithmetic. Arithmetic substitution tells the shell to treat the expansion of a parameter as a number:

```
[] x=1
[] echo $((x*2))
2
```

Integer ranges can be used as follows:

```
[] for i in {1..10} ; do echo $i ; done
1
2
3
4
5
6
7
8
9
10
```

(but see also the *seq* command in section 1.8.2.)

## 1.11 Startup files

In this tutorial you have seen several mechanisms for customizing the behavior of your shell. For instance, by setting the `PATH` variable you can extend the locations where the shell looks for executables. Other environment variables (section 1.7) you can introduce for your own purposes. Many of these customizations will need to apply to every session, so you can have *shell startup files* that will be read at the start of any session.

Popular things to do in a startup file are defining `aliases`:

```
alias grep='grep -i'
alias ls='ls -F'
```

and setting a custom commandline *prompt*.

The name of the startup file depends on your shell: `.bashrc` for Bash, `.cshrc` for the C-shell, and `.zshrc` for the Z-shell. These files are read everytime you log in (see below for details), but you can also `source` them directly:

```
source ~/.bashrc
```

You would do this, for instance, if you have edited your startup file.

Unfortunately, there are several startup files, and which one gets read is a complicated functions of circumstances. Here is a good common sense guideline[2]:

- Have a `.profile` that does nothing but read the `.bashrc`:

  ```
  # ~/.profile
  if [ -f ~/.bashrc ]; then
      source ~/.bashrc
  fi
  ```

- Your `.bashrc` does the actual customizations:

  ```
  # ~/.bashrc
  # make sure your path is updated
  if [ -z "$MYPATH" ]; then
    export MYPATH=1
    export PATH=$HOME/bin:$PATH
  fi
  ```

## 1.12 Shell interaction

Interactive use of Unix, in contrast to script writing (section 1.9), is a complicated conversation between the user and the shell. You, the user, type a line, hit return, and the shell tries to interpret it. There are several cases.

- Your line contains one full command, such as `ls foo`: the shell will execute this command.

2. Many thanks to Robert McLay for figuring this out.

- You can put more than one command on a line, separated by semicolons: `mkdir foo; cd foo`. The shell will execute these commands in sequence.
- Your input line is not a full command, for instance `while [ 1]`. The shell will recognize that there is more to come, and use a different prompt to show you that it is waiting for the remainder of the command.
- Your input line would be a legitimate command, but you want to type more on a second line. In that case you can end your input line with a backslash character, and the shell will recognize that it needs to hold off on executing your command. In effect, the backslash will hide (*escape*) the return.

When the shell has collected a command line to execute, by using one or more of your input line or only part of one, as described just now, it will apply expansion to the command line (section 1.10). It will then interpret the commandline as a command and arguments, and proceed to invoke that command with the arguments as found.

There are some subtleties here. If you type `ls *.c`, then the shell will recognize the wildcard character and expand it to a command line, for instance `ls foo.c bar.c`. Then it will invoke the `ls` command with the argument list `foo.c bar.c`. Note that `ls` does not receive `*.c` as argument! In cases where you do want the unix command to receive an argument with a wildcard, you need to escape it so that the shell will not expand it. For instance, `find . -name \*.c` will make the shell invoke `find` with arguments `. -name *.c`.

## 1.13 The system and other users

### 1.13.1 System information

Most of the above material holds for any Unix or Linux system. Sometimes you need to know detailed information about the system. The following tell you something about what's going on:

*top* which processes are running on the system; use `top -u` to get this sorted the amount of cpu time they are currently taking. (On Linux, try also the `vmstat` command.)

*uptime* how long has it been since your last reboot?

Sometimes you want to know what system you are actually using. Usually you can get some information out of *uname*:

```
$ uname -a
Linux staff.frontera.tacc.utexas.edu 3.10.0-1160.45.1.el7.x86_64 #1 SMP Wed Oct 13 17:20:51
    UTC 2021 x86_64 x86_64 x86_64 GNU/Linux
```

This still doesn't tell you what *Linux distribution* you are on. For that, some of the following may work:

```
$ lsb_release -a
LSB Version:    :core-4.1-amd64:core-4.1-noarch:cxx-4.1-amd64:cxx-4.1-noarch:desktop-4.1-amd64
    :desktop-4.1-noarch:languages-4.1-amd64:languages-4.1-noarch:printing-4.1-amd64:printing
    -4.1-noarch
Distributor ID: CentOS
Description:    CentOS Linux release 7.9.2009 (Core)
Release:        7.9.2009
Codename:       Core
```

or

```
$ ls /etc/*release
/etc/centos-release  /etc/os-release@  /etc/redhat-release@  /etc/system-release@
$ cat /etc/*release
CentOS Linux release 7.9.2009 (Core)
NAME="CentOS Linux"
VERSION="7 (Core)"
ID="centos"
ID_LIKE="rhel fedora"
VERSION_ID="7"
PRETTY_NAME="CentOS Linux 7 (Core)"
ANSI_COLOR="0;31"
CPE_NAME="cpe:/o:centos:centos:7"
HOME_URL="https://www.centos.org/"
BUG_REPORT_URL="https://bugs.centos.org/"

CENTOS_MANTISBT_PROJECT="CentOS-7"
CENTOS_MANTISBT_PROJECT_VERSION="7"
REDHAT_SUPPORT_PRODUCT="centos"
REDHAT_SUPPORT_PRODUCT_VERSION="7"

CentOS Linux release 7.9.2009 (Core)
CentOS Linux release 7.9.2009 (Core)
```

### 1.13.2 Users

Unix is a multi-user operating system. Thus, even if you use it on your own personal machine, you are a user with an *account* and you may occasionally have to type in your username and password.

If you are on your personal machine, you may be the only user logged in. On university machines or other servers, there will often be other users. Here are some commands relating to them.

*whoami* show your login name.

*who* show the other users currently logged in.

*finger* `otheruser` get information about another user; you can specify a user's login name here, or their real name, or other identifying information the system knows about.

### 1.13.3 Groups and ownership

In section 1.2.3 you saw that there is a permissions category for 'group'. This allows you to open up files to your close collaborators, while leaving them protected from the wide world.

When your account is created, your system administrator will have assigned you to one or more groups. (If you admin your own machine, you'll be in some default group; read on for adding yourself to more groups.)

The command *groups* tells you all the groups you are in, and `ls -l` tells you what group a file belongs to. Analogous to `chmod`, you can use *chgrp* to change the group to which a file belongs, to share it with a user who is also in that group.

Creating a new group, or adding a user to a group needs system privileges. To create a group:

```
sudo groupadd new_group_name
```

To add a user to a group:

```
sudo usermod -a -G thegroup theuser
```

While you can change the group of a file, at least between groups that you belong to, changing the owning user of a file with *chown* needs root priviliges. See section 1.13.4.

### 1.13.4 The super user

Even if you own your machine, there are good reasons to work as much as possible from a regular user account, and use *root privileges* only when strictly needed. (The root account is also known as the *super user*.) If you have root privileges, you can also use that to 'become another user' and do things with their privileges, with the *sudo* ('superuser do') command.

- To execute a command as another user:

  ```
  sudo -u otheruser command arguments
  ```

- To execute a command as the root user:

  ```
  sudo command arguments
  ```

- Become another user:

  ```
  sudo su - otheruser
  ```

- Become the *super user*:

  ```
  sudo su -
  ```

Change the owning user of a file is done with *chown*:

```
sudo chown somefile someuser
sudo chown -R somedir someuser
```

## 1.14 Connecting to other machines: `ssh` and `scp`

No man is an island, and no computer is either. Sometimes you want to use one computer, for instance your laptop, to connect to another, for instance a supercomputer.

If you are already on a Unix computer, you can log into another with the 'secure shell' command *ssh*, a more secure variant of the old 'remote shell' command *rsh*:

```
ssh yourname@othermachine.otheruniversity.edu
```

where the `yourname` can be omitted if you have the same name on both machines.

To only copy a file from one machine to another you can use the 'secure copy' *scp*, a secure variant of 'remote copy' *rcp*. The `scp` command is much like `cp` in syntax, except that the source or destination can have a machine prefix.

To copy a file from the current machine to another, type:

```
scp localfile yourname@othercomputer:otherdirectory
```

where yourname can again be omitted, and otherdirectory can be an absolute path, or a path relative to your home directory:

```
# absolute path:
scp localfile yourname@othercomputer:/share/
# path relative to your home directory:
scp localfile yourname@othercomputer:mysubdirectory
```

Leaving the destination path empty puts the file in the remote home directory:

```
scp localfile yourname@othercomputer:
```

Note the colon at the end of this command: if you leave it out you get a local file with an 'at' in the name.

You can also copy a file from the remote machine. For instance, to copy a file, preserving the name:

```
scp yourname@othercomputer:otherdirectory/otherfile .
```

## 1.15 The sed and awk tools

Apart from fairly small utilities such as tr and cut, Unix has some more powerful tools. In this section you will see two tools for line-by-line transformations on text files. Of course this tutorial merely touches on the depth of these tools; for more information see [1, 8].

### 1.15.1 Stream editing with sed

Unix has various tools for processing text files on a line-by-line basis. The stream editor *sed* is one example. If you have used the vi editor, you are probably used to a syntax like s/foo/bar/ for making changes. With sed, you can do this on the commandline. For instance

```
sed 's/foo/bar/' myfile > mynewfile
```

will apply the substitute command s/foo/bar/ to every line of myfile. The output is shown on your screen so you should capture it in a new file; see section 1.6 for more on output *redirection*.

- If you have more than one edit, you can specify them with

  ```
  sed -e 's/one/two/' -e 's/three/four/'
  ```

- If an edit needs to be done only on certain lines, you can specify that by prefixing the edit with the match string. For instance

  ```
  sed '/^a/s/b/c/'
  ```

  only applies the edit on lines that start with an a. (See section 1.3 for regular expressions.) You can also apply it on a numbered line:

  ```
  sed '25/s/foo/bar'
  ```

- The `a` and `i` commands are for 'append' and 'insert' respectively. They are somewhat strange in how they take their argument text: the command letter is followed by a backslash, with the insert/append text on the next line(s), delimited by the closing quote of the command.

```
sed -e '/here/a\
appended text
' -e '/there/i\
inserted text
' -i file
```

- Traditionally, `sed` could only function in a stream, so the output file always had to be different from the input. The GNU version, which is standard on Linux systems, has a flag `-i` which edits 'in place':

```
sed -e 's/ab/cd/' -e 's/ef/gh/' -i thefile
```

### 1.15.2 `awk`

The *awk* utility also operates on each line, but it can be described as having a memory. An awk program consists of a sequence of pairs, where each pair consists of a match string and an action. The simplest awk program is

```
cat somefile | awk '{ print }'
```

where the match string is omitted, meaning that all lines match, and the action is to print the line. Awk breaks each line into fields separated by whitespace. A common application of awk is to print a certain field:

```
awk '{print $2}' file
```

prints the second field of each line.

Suppose you want to print all subroutines in a Fortran program; this can be accomplished with

```
awk '/subroutine/ {print}' yourfile.f
```

**Exercise** 1.39. Build a command pipeline that prints of each subroutine header only the subroutine name. For this you first use `sed` to replace the parentheses by spaces, then `awk` to print the subroutine name field.

Awk has variables with which it can remember things. For instance, instead of just printing the second field of every line, you can make a list of them and print that later:

```
cat myfile | awk 'BEGIN {v="Fields:"} {v=v " " $2} END {print v}'
```

As another example of the use of variables, here is how you would print all lines in between a `BEGIN` and `END` line:

```
cat myfile | awk '/END/ {p=0} p==1 {print} /BEGIN/ {p=1} '
```

**Exercise** 1.40. The placement of the match with `BEGIN` and `END` may seem strange. Rearrange the awk program, test it out, and explain the results you get.

## 1.16 Review questions

**Exercise** 1.41. Devise a pipeline that counts how many users are logged onto the system, whose name starts with a vowel and ends with a consonant.

**Exercise** 1.42. Pretend that you're a professor writing a script for homework submission: if a student invokes this script it copies the student file to some standard location.

```
submit_homework myfile.txt
```

For simplicity, we simulate this by making a directory `submissions` and two different files `student1.txt` and `student2.txt`. After

```
submit_homework student1.txt
submit_homework student2.txt
```

there should be copies of both files in the `submissions` directory. Start by writing a simple script; it should give a helpful message if you use it the wrong way.
Try to detect if a student is cheating. Explore the *diff* command to see if the submitted file is identical to something already submitted: loop over all submitted files and

1. First print out all differences.
2. Count the differences.
3. Test if this count is zero.

Now refine your test by catching if the cheating student randomly inserted some spaces. For a harder test: try to detect whether the cheating student inserted newlines. This can not be done with *diff*, but you could try *tr* to remove the newlines.

# Chapter 2

# Compilers and libraries

## 2.1 File types in programming

**Purpose.** In this section you will be introduced to the different types of files that you encounter while programming.

### 2.1.1 Introduction to file types

Your file system has many files, and for purposes of programming we can roughly divide them into 'text file', which are readable to you, and 'binary files', which are not meaningfully readable to you, but which make sense to the computer.

The unix command *file* can tell you what type of file you are dealing with.

```
$$ file README.txt
README.txt: ASCII text
$$ mkdir mydir
$$ file mydir
mydir: directory
$$ which ls
```

This command can also tell you about binary files. Here the output differs by operating system.

```
$$ which ls
/bin/ls

# on a Mac laptop:
$$ file /bin/ls
/bin/ls: Mach-O 64-bit x86_64 executable

# on a Linux box
$$ file /bin/ls
/bin/ls: ELF 64-bit LSB executable, x86-64
```

**Exercise** 2.1. Apply the `file` command to sources for different programming language. Can you find out how `file` figures things out?

In figure 2.1 you find a brief summary of file types. We will now discuss them in more detail.

| Text files | |
|---|---|
| Source | Program text that you write |
| Header | also written by you, but not really program text. |
| **Binary files** | |
| Object file | The compiled result of a single source file |
| Library | Multiple object files bundled together |
| Executable | Binary file that can be invoked as a command |
| Data files | Written and read by a program |

Figure 2.1: Different types of files.

### 2.1.2 About 'text' files

Readable files are sometimes called *text files*; but this is not a concept with a hard definition. One not-perfect definition is that text files are *ascii* files, meaning files where every byte uses '7-bit ascii': the first bit of every byte is zero.

This definition is incomplete, since modern programming languages can often use *unicode*, at least in character strings. (For a tutorial on ascii and unicode, see chapter 6 of [9].)

### 2.1.3 Source versus program

There are two types of programming languages:

1. In an *interpreted language* you write human-readable source code and you execute it directly: the computer translates your source line by line as it encounters it.
2. In a *compiled language* your whole source code is first compiled to a binary program file, called the *executable*, which you then execute.

Examples of interpreted languages are *Python*, *Matlab*, *Basic*, *Lisp*. Interpreted languages have some advantages: often you can write them in an interactive environment that allows you to test code very quickly. They also allow you to construct code dynamically, during runtime. However, all this flexibility comes at a price: if a source line is executed twice, it is translated twice. In the context of this book, then, we will focus on compiled languages, using *C* and *Fortran* as prototypical examples.

So now you have a distinction between the readable source code, and the unreadable, but executable, program code. In this tutorial you will learn about the translation process from the one to the other. The program doing this translation is known as a *compiler*. This tutorial will be a 'user manual' for compilers, as it were; what goes on inside a compiler is a different branch of computer science.

### 2.1.4 Binary files

Binary files fall in two categories:

- executable code,
- data

Data files can be really anything: they are typically output from a program, and their format is often specific to that program, although there are some standards, such as *hdf5*. You get a binary data file if you write out the exact bytes that integer or floating point data takes up in memory, rather than a readable representation of that number, the way you see on your screen.

**Exercise** 2.2. Why don't programs write their results to file in readable form?

*Enrichment.* How do you write/read a binary file in C and Fortran? Use the function `hexdump` to make sense of the binary file. Can you generate the file from Fortran, and read it from C? (Answer: yes, but it's not quite straightforward.) What does this tell you about binary data?

```
// binary_write.c
  FILE *binfile;
  binfile = fopen("binarydata.out","wb");
  for (int i=0; i<10; i++)
    fwrite(&i,sizeof(int),1,binfile);
  fclose(binfile);
```

```
// binary_read.c
  binfile = fopen("binarydata.out","rb");
  for (int i=0; i<10; i++) {
    int ival;
    fread(&ival,sizeof(int),1,binfile);
    printf("%d ",ival);
  }
  printf("\n");
```

```
[linking:31] make binary
clang -o binary_write_c binary_write.c
./binary_write_c
clang -o binary_read_c binary_read.c
./binary_read_c
0 1 2 3 4 5 6 7 8 9
[linking:32] hexdump binarydata.out
0000000 00 00 00 00 01 00 00 00 02 00 00 00 03 00 00 00
0000010 04 00 00 00 05 00 00 00 06 00 00 00 07 00 00 00
0000020 08 00 00 00 09 00 00 00
0000028
```

Fortran works differently: each *record*, that is, the output of each `Write` statement, has the record length (in bytes) before and after it.

```
[linking:68] make xbinary
gfortran -o binary_write_f binary_write.F90
./binary_write_f
hexdump binarydata.out
0000000 04 00 00 00 00 00 00 00 04 \
                00 00 00 04 00 00 00
0000010 01 00 00 00 04 00 00 00 04 \
                00 00 00 02 00 00 00
0000020 04 00 00 00 04 00 00 00 03 \
                00 00 00 04 00 00 00
0000030 04 00 00 00 04 00 00 00 04 \
                00 00 00 04 00 00 00
```

```
0000040 05 00 00 00 04 00 00 00 04 \
                  00 00 00 06 00 00 00
0000050 04 00 00 00 04 00 00 00 07 \
                  00 00 00 04 00 00 00
0000060 04 00 00 00 08 00 00 00 04 \
                  00 00 00 04 00 00 00
0000070 09 00 00 00 04 00 00 00
```

```
// binary_write.F90
  Open(Unit=13,File="binarydata.out",Form="
    unformatted")
  do i=0,9
     write(13) i
  end do
  Close(Unit=13)
```

In this tutorial you will mostly be concerned with executable binary files. We then distinguish between:

- program files, which are executable by themselves;
- object files, which are like bit of programs; and
- library files, which combine object files, but are not executable.

Object files come from the fact that your source is often spread over multiple source files, and these can be compiled separately. In this way, an *object file*, is a piece of an executable: by itself it does nothing, but it can be combined with other object files to form an executable.

If you have a collection of object files that you need for more than one program, it is usually a good idea to make a *library*: a bundle of object files that can be used to form an executable. Often, libraries are written by an expert and contain code for specialized purposes such as linear algebra manipulations. Libraries are important enough that they can be commercial, to be bought if you need expert code for a certain purpose.

You will now learn how these types of files are created and used.

## 2.2 Simple compilation

**Purpose.** In this section you will learn about executables and object files.

### 2.2.1 Compilers

Your main tool for turning source into a program is the *compiler*. Compilers are specific to a language: you use a different compiler for C than for Fortran. You can also have two compilers for the same language, but from different 'vendors'. For instance, while many people use the open source *gcc* or *clang* compiler families, companies like *Intel* and *IBM* offer compilers that may give more efficient code on their processors.

### 2.2.2 Compile a single file

Let's start with a simple program that has the whole source in one file.

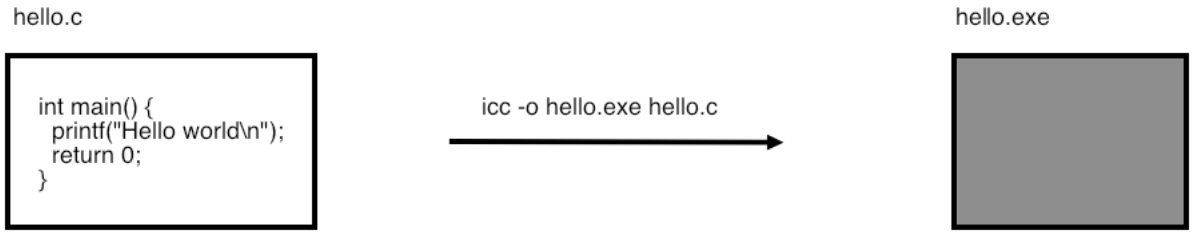



Figure 2.2: Compiling a single source file.

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

int main() {
  printf("hello world\n");
  return 0;
}
```

Compile this program with your favorite compiler; we will use `gcc` in this tutorial, but substitute your own as desired.

*TACC note.* On TACC clusters, the Intel compiler `icc` is preferred.

As a result of the compilation, a file `a.out` is created, which is the executable.

```
%% gcc hello.c
%% ./a.out
hello world
```

You can get a more sensible program name with the `-o` option:

```
%% gcc -o helloprog hello.c
%% ./helloprog
hello world
```

This process is illustrated in figure 2.2.

### 2.2.3 Compilation: the nitty gritty

Even for the simple compilation of the previous section some details have been left out. Here is an outline of the steps of the compilation process, which involves more than merely invoking the compiler. For instance, your program likely contains `#include` statements, which are processed in a separate stage.

Let's take a look at a simple 'hello world' program. Rather than transforming your source in one fell swoop into an executable, the following steps happen.

1. The preprocessor generates a file `hello.i` where all includes are inlined, and other macros expanded. This file is usually immediately deleted: you need to supply a compiler flag for it to be left behind.

2. An *assembly listing* `hello.s` of your program is generated; againi, to be immediately deleted after use. This is a sort of 'readable machine language':

```
        .arch armv8-a
        .text
        .cstring
        .align  3
lC0:
        .ascii "hello world\0"
        .text
        .align  2
        .globl _main
_main:
LFB10:
        stp     x29, x30, [sp, -16]!
```

3. The main tangible result of the compilation is `hello.o`, the object file, containing actual machine language. We will go into this more below. The object file is not directly readable, but later you'll see the *nm* tool that can give you some information.
4. Finally, the *linker* hooks together the object file and system libraries into a self-contained executable or a library file.

### 2.2.4 Multiple files: compile and link

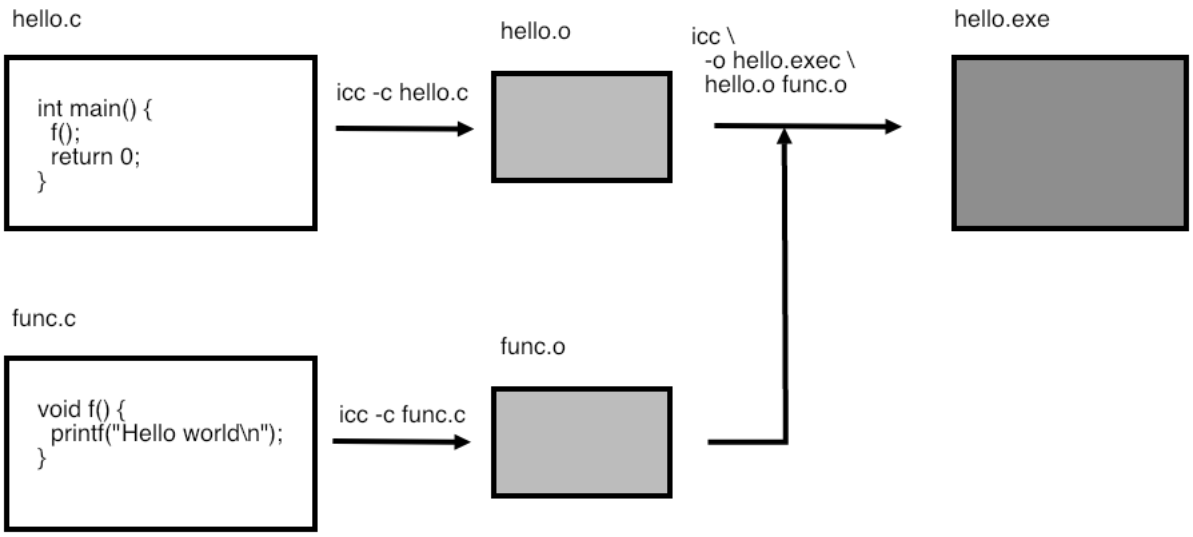



Figure 2.3: Compiling a program from multiple source files.

As an illustration of linking, let's consider a program where the source is contained in more than one file.

Main program: `fooprog.c`

```
// fooprog.c
extern void bar(char*);

int main() {
  bar("hello world\n");
```

```
  return 0;
}
```

Subprogram: `foosub.c`

```
// foosub.c
void bar(char *s) {
  printf("%s",s);
  return;
}
```

As before, you can make the program with one command.

**Output**
**[code/compile] makeoneprogram:**

```
clang -o oneprogram fooprog.c foosub.c
./oneprogram
hello world
```

However, you can also do it in steps, compiling each file separately and then linking them together. This is illustrated in figure 2.3.

**Output**
**[code/compile] makeseparatecompile:**

```
clang -g -O2 -o oneprogram fooprog.o foosub.o
./oneprogram
hello world
```

The `-c` option tells the compiler to compile the source file, giving an *object file*. The third command then acts as the *linker*, tieing together the object files into an executable.

**Exercise** 2.3.

Exercise for separate compilation. Structure:

Main program: `fooprog.c`

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

extern void bar(char*);

int main() {
  bar("hello world\n");
  return 0;
}
```

Subprogram: `foosub.c`

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

void bar(char *s) {
  printf("%s",s);
  return;
}
```

Add a second subroutine in a second file.

- Compile in one:

```
icc -o program fooprog.c foosub.c
```

- Compile in steps:

```
icc -c fooprog.c
icc -c foosub.c
```

```
icc -o program fooprog.o foosub.o
```

What files are being produced each time?
Can you write a shell script to automate this?

### 2.2.5 Paths

If your program uses libraries, maybe even of your own making, you need to tell

1. the compiler where to find the *header files*, and
2. the linker where to find the library file.

(C++ knows 'header only' libraries, so the second step is not always needed.) These locations are indicated by commandline options:

```
gcc -c mysource.cpp -I${SOMELIB_INC_DIR}
gcc -o myprogram mysource.o -L${SOMELIB_LIB_DIR} -lsomelib
```

(Instead of listing these on the commandline every time, you would of course put them in a makefile, or use CMake to generate such commandlines.) The compile line has the `-I` option for 'include', that specifies the location of the library header file. You can specify multiple include options.

The link line has the `-L` option for 'library', that specifies the location of the actual library files, and the `-l` option for the library name. You can specify multiple library directories. The `-l` option is interpreted as follows: `-l somelib` makes the linker search for files

```
libsomelib.a
libsomelib.so
libsomelib.dylib
```

### 2.2.6 Looking into binary files: `nm`

Most of a binary file consists of the same instructions that you coded in C or Fortran, just in machine language, which is much harder to understand. Fortunately, you don't need to look at machine language often. Often what interests you about object files is only knowing what functions are defined in it, and what functions are used in it.

For this, we use the *nm* command.

Each object file defines some routine names, and uses some others that are undefined in it, but that will be defined in other object files or system libraries. Use the *nm* command to display this:

```
[c:264] nm foosub.o
0000000000000000 T _bar
                 U _printf
```

Lines with `T` indicate routines that are defined; lines with `U` indicate routines that are used but not define in this file. In this case, `printf` is a system routine that will be supplied in the linker stage.

### 2.2.7 Name demangling in C++

With C++ the function names will look a little strange because of *name mangling*. For instance the function

```
void bar( string s ) {
  cout << s;
}
```

gives:

```
[] nm foosub.o
0000000000000000 T __Z3barNSt7__cxx1112basic_stringIcSt11char_traitsIcESaIcEEE
                 U __ZNSt8ios_base4InitC1Ev
                 U __ZNSt8ios_base4InitD1Ev
                 U
    __ZSt16__ostream_insertIcSt11char_traitsIcEERSt13basic_ostreamIT_T0_ES6_PKS3_l
                 U __ZSt4cout
```

You sort of recognize the `bar` function in this. Add an option `nm -C` to get de-mangled names:

```
[scientific-computing-private/code/compilecxx 1354] nm -C !$
[] nm -C foosub.o
0000000000000000 T bar(std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator
    <char> >)
                 U std::ios_base::Init::Init()
                 U std::ios_base::Init::~Init()
                 U std::basic_ostream<char, std::char_traits<char> >& std::__ostream_insert<
    char, std::char_traits<char> >(std::basic_ostream<char, std::char_traits<char> >&, char
    const*, long)
                 U std::cout
```

Another possibility is to pipe the `nm` output through `c++filt`:

```
nm foosub.o | c++file
```

You may also want to demangle names in linker errors, when your object files leave some references undefined. For this, add an option `-Wl,-demangle` to the link line:

```
Compile: g++ -g -O2 -std=c++17                -c undefined.cxx
Link line: g++ -Wl,-demangle      -o undefined undefined.o
Undefined symbols for architecture arm64:
  "f(int, int)", referenced from:
      _main in undefined.o
ld: symbol(s) not found for architecture arm64
collect2: error: ld returned 1 exit status
make[2]: *** [undefined] Error 1
```

The mangled names are available programmatically:

```
#include <typeinfo>
cout << typeid( &Myclass::method ).name() << '\n'
```

Demangling also has an Application Programmer Interface (API):

```
#include <cxxabi.h>
char* abi::__cxa_demangle
    ( const char* mangled, char* output, size_t* length,int status );
```

**Remark 9** *There are other tools with similar functionality to* `nm`*, such as* `otool`*,* `objdump`*,* `readelf`*.*

**Remark 10** *Sometimes you will come across* stripped binary *file, and* `nm` *will report* `No symbols`*. In that case* `nm -D` *may help, which displays 'dynamic symbols'.*

### 2.2.8 Compiler options and optimizations

Above you already saw some *compiler options*:

- Specifying `-c` tells the compiler to only compile, and not do the linking stage; you would do this in case of separate compilation.
- The option `-o` gives you the opportunity to specify the name of the output file; without it, the default name of an executable is `a.out`.

There are many other options, some of them a *de facto* standard, and others specific to certain compilers.

#### *2.2.8.1 Symbol table inclusion*

The `-g` option tells the compiler to include the *symbol table* in the binary. This allows you to use an interactive debugger (section 11) since it relates machine instructions to lines of code, and machine addresses to variable names.

#### *2.2.8.2 Optimization level*

Compilers can apply various levels of *optimization* to your code. The typical optimization levels are specified as `-O0` 'minus-oh-zero', `-O1`, `-O2`, `-O3`. Higher levels will typically give faster execution, as the compiler does increasingly sophisticated analysis on your code.

The following is a fairly standard set of options:

```
icc -g -O2 -c myfile.c
```

As an example, let's look at *Given's rotations*:

```
// rotate.c
void rotate(double *x,double *y,double alpha) {
  double x0 = *x, y0 = *y;
  *x = cos(alpha) * x0 - sin(alpha) * y0;
  *y = sin(alpha) * x0 + cos(alpha) * y0;
  return;
}
```

Run with optimization level 0,1,2,3 we get:

```
Done after 8.649492e-02
Done after 2.650118e-02
Done after 5.869865e-04
Done after 6.787777e-04
```

**Exercise** 2.4. From level zero to one we get (in the above example; in general this depends on the compiler) an improvement of 2× to 3×. Can you find an obvious factor of two?
Use the optimization report facility of your compiler to see what other optimizations are applied. One of them is a good lesson in benchmark design!

Many compilers can generate a report of what optimizations they perform.

| compiler | reporting option |
|---|---|
| clang | `-Rpass=.*` |
| gcc | `-fopt-info` |
| intel | `-qopt-report` |

Generally, optimizations leave the semantics of your code intact. (Makes kinda sense, not?) However, at higher levels, usually level 3, the compiler is at liberty to make transformations that are not legal according to the language standard, but that in the majority of cases will still give the right outcome. For instance, the C language specifies that arithmetic operations are evaluated left-to-right. Rearranging arithmetic expressions is usually safe, but not always. Be careful when applying higher optimization levels!

#### *2.2.8.3 Optimization reporting*

Compilers can report on certain optimizations.

- The *Intel compiler* has a parameter `-qoptreport=[0-5]`.
- The *GCC compiler* has multiple commandline options:

  ```
  -fopt-info -fopt-info-options -fopt-info-options=filename
  ```

  See: https://stackoverflow.com/a/65017597/2044454.

## 2.3 Libraries

**Purpose.** In this section you will learn about creating libraries.

If you have written some subprograms, and you want to share them with other people (perhaps by selling them), then handing over individual object files is inconvenient. Instead, the solution is to combine them into a library. This section shows you the basic Unix mechanisms. You would typically use these in a Makefile; if you use CMake it's all done for you; see chapter 4.

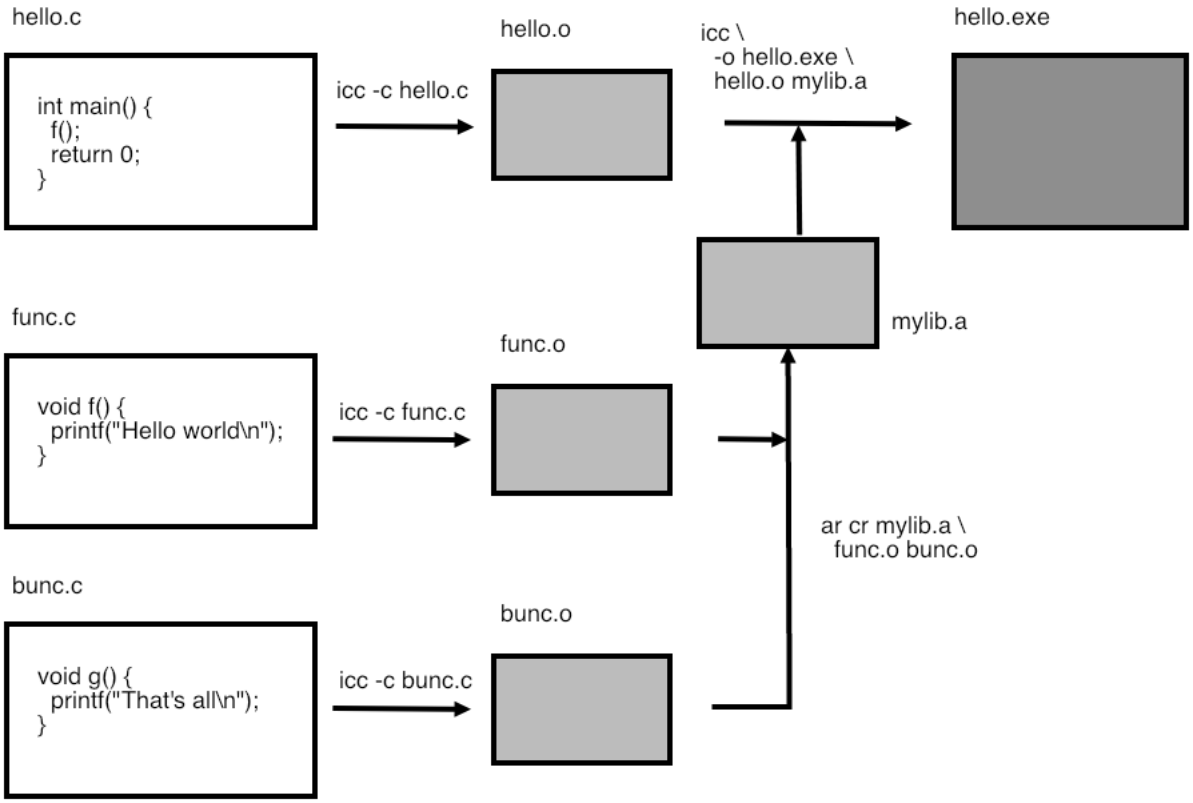



Figure 2.4: Compiling a single source file.

### 2.3.1 Static libraries

First we look at *static libraries*, for which the *archive utility* `ar` is used. A static library is linked into your executable, becoming part of it. This has the advantage that you can give such an executable to someone else, and it will immediately work. On the other hand, this may lead to large executables; you will learn about shared libraries next, which do not suffer from this problem.

The use of a library to build a program is illustrated in figure 2.4.

Create a directory to contain your library, and create the library file there. The library can be linked into your executable by explicitly giving its name, or by specifying a library path:

**Output**
**[code/compilecxx] staticprogram:**

```
==== Use of static library ====

---- Build library:
for o in foosub.o ; do ar cr libs/libfoo.a ${o} ; done
---- Linking:
g++ -o staticprogram fooprog.o -Llibs -lfoo
---- Program:
-rwxr-xr-x  1 eijkhout  staff  52042 Oct 21 10:46 staticprogram

---- running:
hello world
---- done
```

The *nm* command tells you what's in the library, just like it did with object files, but now it also tells you what object files are in the library:

We show this for C:

**Code:**

```
==== Making static library ====
for o in foosub.o ; do ar cr libs/libfoo.a ${o} ; done
nm  libs/libfoo.a
```

**Output**
**[code/compilec] staticlib:**

```
foosub.o:
0000000000000000 T bar
0000000000000000 N .debug_info_seg
                 U printf
```

For C++ we show the output in figure 2.5, where we note the `-C` flag for name *demangling*.

### 2.3.2 Shared libraries

Although they are somewhat more complicated to use, *shared libraries* have several advantages. For instance, since they are not linked into the executable but only loaded at runtime, they lead to (much) smaller executables. They are not created with `ar`, but through the compiler. For instance:

**Output**
**[code/compile] makedynamiclib:**

```
%%%%
Demonstration: making shared library
%%%%
clang -o libs/libfoo.so -shared foosub.o
```

You can again use *nm*:

```
%% nm ../lib/libfoo.so

../lib/libfoo.so(single module):
00000fc4 t __dyld_func_lookup
```

```
00000000 t __mh_dylib_header
00000fd2 T _bar
         U _printf
00001000 d dyld__mach_header
00000fb0 t dyld_stub_binding_helper
```

### 2.3.3 Finding libraries

Shared libraries are not actually linked into the executable; instead, the executable needs the information where the library is to be found at execution time. One way to do this is with *LD_LIBRARY_PATH*; see figure 2.6

Another solution is to have the path be included in the executable:

```
%% gcc -o foo fooprog.o -L../lib -Wl,-rpath,`pwd`/../lib -lfoo
%% ./foo
hello world
```

The link line now contains the library path twice:

1. once with the `-L` directive so that the linker can resolve all references, and
2. once with the linker directive `-Wl,-rpath,`pwd`/../lib` which stores the path into the executable so that it can be found at runtime.

**Remark 11** *You may also come across a syntax with* `rpath=`*:*

```
gcc -o foo fooprog.o -L../lib -Wl,-rpath=`pwd`/../lib -lfoo
```

*but beware that that is a GNU extension.*

**Remark 12** *On* Apple OS Ventura *the use of LD_LIBRARY_PATH is no longer supported for security reasons, so using the* `rpath` *is the only option.*

Use the command *ldd* to get information about what shared libraries your executable uses. (On Mac OS X, use `otool -L` instead.)

### 2.3.4 Standard library

Certain functionality, such as I/O or file and process management, is used in all programming languages, but belongs more to the OS than to the language proper. That's why this has been bundled in a *standard library*, called *libstdc* or *libstdc++*. Error messages about the contents of this library typically refer to *GLIBC* or *GLIBCXX* with a version number.

```
[] /sbin/ldconfig -p | grep stdc++
        libstdc++.so.6 (libc6,x86-64) => /lib64/libstdc++.so.6
```

```
        libstdc++.so.5 (libc6,x86-64) => /lib64/libstdc++.so.5
[] strings /lib64/libstdc++.so.6 | grep LIBCXX
GLIBCXX_3.4
GLIBCXX_3.4.1
   [...]
GLIBCXX_3.4.19
```

Sometimes functionality is requested from a later version than is present in your system.

**Code:**

```
==== Making static library ====
for o in foosub.o ; do ar cr libs/libfoo.a ${o} ; done
nm -C libs/libfoo.a
```

**Output**
**[code/compilecxx] staticlib:**

```
foosub.o:
                 U __cxa_atexit
0000000000000000 N .debug_info_seg
                 U __dso_handle
                 U
    __gxx_personality_v0
0000000000000010 t __sti__$E
0000000000000000 T bar(std::
    __cxx11::basic_string<char,
    std::char_traits<char>, std::
    allocator<char> >)
0000000000000000 b
    _INTERNALaee936d8::std::
    __ioinit
0000000000000000 W std::__cxx11::
    basic_string<char, std::
    char_traits<char>, std::
    allocator<char> >::data()
    const
0000000000000000 W std::__cxx11::
    basic_string<char, std::
    char_traits<char>, std::
    allocator<char> >::size()
    const
                 U std::ios_base::
    Init::Init()
                 U std::ios_base::
    Init::~Init()
                 U std::cout
                 U std::
    basic_ostream<char, std::
    char_traits<char> >& std::
    operator<< <char, std::
    char_traits<char>, std::
    allocator<char> >(std::
    basic_ostream<char, std::
    char_traits<char> >&, std::
    __cxx11::basic_string<char,
    std::char_traits<char>, std::
    allocator<char> > const&)
```

Figure 2.5: The `nm` output of a C++ library

**Output**
**[code/compile] dynamicprogram:**

```
Linking to shared library

clang -o libs/libfoo.so -shared foosub.o
clang -o dynamicprogram fooprog.o -Llibs -lfoo
-rwxr-xr-x  1 eijkhout  staff  49720 Nov 28 12:00 dynamicprogram

.. note the size of the program

-rwxr-xr-x  1 eijkhout  staff  49720 Nov 28 12:00 dynamicprogram

.. note unresolved link to a library

otool -L dynamicprogram | grep libfoo
        libs/libfoo.so (compatibility version 0.0.0, current version 0.0.0)

.. running by itself:

clang -o libs/libfoo.so -shared foosub.o
clang -o dynamicprogram fooprog.o -Llibs -lfoo
-rwxr-xr-x  1 eijkhout  staff  49720 Nov 28 12:00 dynamicprogram
hello world

.. note resolved with LD_LIBRARY_PATH

LD_LIBRARY_PATH=${LD_LIBRARY_PATH}:./libs otool -L dynamicprogram | grep
    libfoo
        libs/libfoo.so (compatibility version 0.0.0, current version 0.0.0)

.. running with updated library path:

clang -o libs/libfoo.so -shared foosub.o
clang -o dynamicprogram fooprog.o -Llibs -lfoo
-rwxr-xr-x  1 eijkhout  staff  49720 Nov 28 12:00 dynamicprogram
LD_LIBRARY_PATH=${LD_LIBRARY_PATH}:./libs dynamicprogram
hello world
```

Figure 2.6: Making and using a dynamic library

# Chapter 3

# Managing projects with Make

The *Make* utility helps you manage the building of projects: its main task is to facilitate rebuilding only those parts of a multi-file project that need to be recompiled or rebuilt. This can save lots of time, since it can replace a minutes-long full installation by a single file compilation.

*Make* is a Unix utility with a long history, and traditionally there are variants with slightly different behavior, for instance on the various flavors of Unix such as HP-UX, AUX, IRIX. These days, it is advisable, no matter the platform, to use the GNU version of Make which has some very powerful extensions; it is available on all Unix platforms (on Linux it is the only available variant), and it is a *de facto* standard. The manual is available at `http://www.gnu.org/software/make/manual/make.html`, or you can read the book [14].

The examples in this tutorial will be for the C and Fortran languages, but *Make* can work with any language, and in fact with things like TEX that are not really a language at all; see section 3.7.

## 3.1 A simple example

**Purpose.** In this section you will see a simple example, just to give the flavor of *Make*.

The files for this section can be found in the repository.

### 3.1.1 C++

Make the following files:

`foo.cxx`

```
#include <iostream>
using std::cout;

#include "bar.h"

int main()
{
  int a=2;
```

```
  cout << bar(a) << '\n';
  return 0;
}
```

`bar.cxx`

```
#include "bar.h"
int bar(int a)
{
  int b=10;
  return b*a;
}
```

`bar.h`

```
int bar(int);
```

and a makefile:

`Makefile`

```
fooprog : foo.o bar.o
icpc -o fooprog foo.o bar.o
foo.o : foo.cxx
icpc -c foo.cxx
bar.o : bar.cxx
icpc -c bar.cxx
clean :
rm -f *.o fooprog
```

The makefile has a number of rules like

```
foo.o : foo.cxx
<TAB>icpc -c foo.cxx
```

which have the general form

```
target : prerequisite(s)
<TAB>rule(s)
```

where the rule lines are indented by a `TAB` character.

A rule, such as above, states that a 'target' file `foo.o` is made from a 'prerequisite' `foo.cxx`, namely by executing the command `icpc -c foo.cxx`. (Here we are using the *Intel C++ compiler* `icpc`; your system could have a different compiler, such as `clang++` or `g++`.)

The precise definition of the rule is:

- if the target `foo.o` does not exist or is older than the prerequisite `foo.cxx`,
- then the command part of the rule is executed: `icpc -c foo.cxx`
- If the prerequisite is itself the target of another rule, than that rule is executed first.

**Exercise.** Call `make`.

*Expected outcome.* The above rules are applied: `make` without arguments tries to build the first target, `fooprog`. In order to build this, it needs the prerequisites `foo.o` and `bar.o`, which do not exist. However, there are rules for making them, which `make` recursively invokes. Hence you see two compilations, for `foo.o` and `bar.o`, and a link command for `fooprog`.

*Caveats.* Typos in the makefile or in file names can cause various errors. In particular, make sure you use tabs and not spaces for the rule lines. Unfortunately, debugging a makefile is not simple. *Make*'s error message will usually give you the line number in the make file where the error was detected.

**Exercise.** Do `make clean`, followed by `mv foo.cxx boo.cxx` and `make` again. Explain the error message. Restore the original file name.

*Expected outcome. Make* will complain that there is no rule to make `foo.cxx`. This error was caused when `foo.cxx` was a prerequisite for making `foo.o`, and was found not to exist. *Make* then went looking for a rule to make it and no rule for creating `.cxx` files exists.

Now add a second argument to the function `bar`. This would require you to edit all of `bar.cxx`, `bar.h`, and `foo.cxx`, but let's say we forget to edit the last two, so only edit `bar.cxx` However, it also requires you to edit `foo.c`, but let us for now 'forget' to do that. We will see how *Make* can help you find the resulting error.

**Exercise.** Call `make` to recompile your program. Did it recompile `foo.c`?

*Expected outcome.* You see that inclusion of the 'wrong' header file does not lead to an error, because C++ has polymorphism. In C this would definitely give an error. The error only shows up in the linker stage because of an unresolved reference.

**Exercise.** Update the header file, and call `make` again. What happens, and what had you been hoping would happen?

*Expected outcome.* Only the linker stage is done, and it gives the same error about an unresolved reference. Were you hoping that the main program would be recompiled?

*Caveats.*

The way out of this problem is to tie the header file to the source files in the makefile.

In the makefile, change the line

```
foo.o : foo.cxx
```

to

```
foo.o : foo.cxx bar.h
```

which adds `bar.h` as a prerequisite for `foo.o`. This means that, in this case where `foo.o` already exists, *Make* will check that `foo.o` is not older than any of its prerequisites. Since `bar.h` has been edited, it is younger than `foo.o`, so `foo.o` needs to be reconstructed.

**Remark 13** *As already noted above, in C++ fewer errors are caught by this mechanism than in C, because of polymorphism. You might wonder if it would be possible to generate header files automatically. This is of course possible with suitable shell scripts, but tools such as* `Make` *(or* `CMake`*) do not have this built in.*

### 3.1.2 C

Make the following files:

`foo.c`

```
#include "bar.h"
int c=3;
int d=4;
int main()
{
  int a=2;
  return(bar(a*c*d));
}
```

`bar.c`

```
#include "bar.h"
int bar(int a)
{
  int b=10;
  return(b*a);
}
```

`bar.h`

```
extern int bar(int);
```

and a makefile:

`Makefile`

```
fooprog : foo.o bar.o
cc -o fooprog foo.o bar.o
foo.o : foo.c
cc -c foo.c
bar.o : bar.c
cc -c bar.c
clean :
rm -f *.o fooprog
```

The makefile has a number of rules like

```
foo.o : foo.c
<TAB>cc -c foo.c
```

which have the general form

```
target : prerequisite(s)
<TAB>rule(s)
```

where the rule lines are indented by a `TAB` character.

A rule, such as above, states that a 'target' file `foo.o` is made from a 'prerequisite' `foo.c`, namely by executing the command `cc -c foo.c`. The precise definition of the rule is:

- if the target `foo.o` does not exist or is older than the prerequisite `foo.c`,

- then the command part of the rule is executed: `cc -c foo.c`
- If the prerequisite is itself the target of another rule, than that rule is executed first.

**Exercise.** Call `make`.

> *Expected outcome.* The above rules are applied: `make` without arguments tries to build the first target, `fooprog`. In order to build this, it needs the prerequisites `foo.o` and `bar.o`, which do not exist. However, there are rules for making them, which `make` recursively invokes. Hence you see two compilations, for `foo.o` and `bar.o`, and a link command for `fooprog`.
>
> *Caveats.* Typos in the makefile or in file names can cause various errors. In particular, make sure you use tabs and not spaces for the rule lines. Unfortunately, debugging a makefile is not simple. *Make*'s error message will usually give you the line number in the make file where the error was detected.

**Exercise.** Do `make clean`, followed by `mv foo.c boo.c` and `make` again. Explain the error message. Restore the original file name.

> *Expected outcome. Make* will complain that there is no rule to make `foo.c`. This error was caused when `foo.c` was a prerequisite for making `foo.o`, and was found not to exist. *Make* then went looking for a rule to make it and no rule for creating `.c` files exists.

Now add a second argument to the function `bar`. This requires you to edit `bar.c` and `bar.h`: go ahead and make these edits. However, it also requires you to edit `foo.c`, but let us for now 'forget' to do that. We will see how *Make* can help you find the resulting error.

**Exercise.** Call `make` to recompile your program. Did it recompile `foo.c`?

> *Expected outcome.* Even through conceptually `foo.c` would need to be recompiled since it uses the `bar` function, *Make* did not do so because the makefile had no rule that forced it.

In the makefile, change the line

```
foo.o : foo.c
```

to

```
foo.o : foo.c bar.h
```

which adds `bar.h` as a prerequisite for `foo.o`. This means that, in this case where `foo.o` already exists, *Make* will check that `foo.o` is not older than any of its prerequisites. Since `bar.h` has been edited, it is younger than `foo.o`, so `foo.o` needs to be reconstructed.

**Exercise.** Confirm that the new makefile indeed causes `foo.o` to be recompiled if `bar.h` is changed. This compilation will now give an error, since you 'forgot' to edit the use of the `bar` function.

### 3.1.3 Fortran

Make the following files:

`foomain.F`

```
            call func(1,2)

            end program
```

`foomod.F`

```
            contains

            subroutine func(a,b)
            integer a,b
            print *,a,b,c
            end subroutine func

            end module
```

and a makefile:

`Makefile`

```
fooprog : foomain.o foomod.o
gfortran -o fooprog foo.o foomod.o
foomain.o : foomain.F
gfortran -c foomain.F
foomod.o : foomod.F
gfortran -c foomod.F
clean :
rm -f *.o fooprog
```

If you call `make`, the first rule in the makefile is executed. Do this, and explain what happens.

**Exercise.** Call `make`.

> *Expected outcome.* The above rules are applied: `make` without arguments tries to build the first target, `foomain`. In order to build this, it needs the prerequisites `foomain.o` and `foomod.o`, which do not exist. However, there are rules for making them, which `make` recursively invokes. Hence you see two compilations, for `foomain.o` and `foomod.o`, and a link command for `fooprog`.
>
> *Caveats.* Typos in the makefile or in file names can cause various errors. Unfortunately, debugging a makefile is not simple. You will just have to understand the errors, and make the corrections.

**Exercise.** Do `make clean`, followed by `mv foomod.c boomod.c` and `make` again. Explain the error message. Restore the original file name.

> *Expected outcome. Make* will complain that there is no rule to make `foomod.c`. This error was caused when `foomod.c` was a prerequisite for `foomod.o`, and was found not to exist. *Make* then went looking for a rule to make it, and no rule for making `.F` files exists.

Now add an extra parameter to `func` in `foomod.F` and recompile.

**Exercise.** Call `make` to recompile your program. Did it recompile `foomain.F`?

*Expected outcome.* Even through conceptually `foomain.F` would need to be recompiled, *Make* did not do so because the makefile had no rule that forced it.

Change the line

```
foomain.o : foomain.F
```

to

```
foomain.o : foomain.F foomod.o
```

which adds `foomod.o` as a prerequisite for `foomain.o`. This means that, in this case where `foomain.o` already exists, *Make* will check that `foomain.o` is not older than any of its prerequisites. Recursively, *Make* will then check if `foomode.o` needs to be updated, which is indeed the case. After recompiling `foomode.F`, `foomode.o` is younger than `foomain.o`, so `foomain.o` will be reconstructed.

**Exercise.** Confirm that the corrected makefile indeed causes `foomain.F` to be recompiled.

## 3.2 Some general remarks

### 3.2.1 Rule interpretation

Probably the best way to interpret a rule is:

- if any prerequisite has changed,
- then the target needs to be remade,
- and that is done by executing the commands of the rule;
- checking the prerequisite requires a recursive application of make:
  - if the prerequisite does not exist, find a rule to create it;
  - if the prerequisite already exists, check applicable rules to see if it needs to be remade.

### 3.2.2 Make invocation

If you call `make` without any arguments, the first rule in the makefile is evaluated. You can execute other rules by explicitly invoking them, for instance `make foo.o` to compile a single file.

### 3.2.3 About the make file

The make file needs to be called `makefile` or `Makefile`; it is not a good idea to have files with both names in the same directory. If you want *Make* to use a different file as make file, use the syntax `make -f My_Makefile`.

## 3.3 Variables and template rules

**Purpose.** In this section you will learn various work-saving mechanisms in *Make*, such as the use of variables and of template rules.

### 3.3.1 Makefile variables

It is convenient to introduce variables in your makefile. For instance, instead of spelling out the compiler explicitly every time, introduce a variable in the makefile:

```
CC = gcc
FC = gfortran
```

and use `${CC}` or `${FC}` on the compile lines:

```
foo.o : foo.c
        ${CC} -c foo.c
foomain.o : foomain.F
        ${FC} -c foomain.F
```

**Exercise.** Edit your makefile as indicated. First do `make clean`, then `make foo` (C) or `make fooprog` (Fortran).

> *Expected outcome.* You should see the exact same compile and link lines as before.
>
> *Caveats.* Unlike in the shell, where braces are optional, variable names in a makefile have to be in braces or parentheses. Experiment with what happens if you forget the braces around a variable name.

One advantage of using variables is that you can now change the compiler from the commandline:

```
make CC="icc -O2"
make FC="gfortran -g"
```

**Exercise.** Invoke *Make* as suggested (after `make clean`). Do you see the difference in your screen output?

> *Expected outcome.* The compile lines now show the added compiler option `-O2` or `-g`.

*Make* also has *automatic variables*:

- `$@` The target. Use this in the link line for the main program.
- `$^` The list of prerequisites. Use this also in the link line for the program.
- `$<` The first prerequisite. Use this in the compile commands for the individual object files.
- `$*` In *template rules* (section 3.3.2) this matches the template part, the part corresponding to the `%`.

Using these variables, the rule for `fooprog` becomes

```
fooprog : foo.o bar.o
        ${CC} -o $@ $^
```

and a typical compile line becomes

```
foo.o : foo.c bar.h
        ${CC} -c $<
```

You can also declare a variable

```
THEPROGRAM = fooprog
```

and use this variable instead of the program name in your makefile. This makes it easier to change your mind about the name of the executable later.

**Exercise.** Edit your makefile to add this variable definition, and use it instead of the literal program name. Construct a commandline so that your makefile will build the executable `fooprog_v2`.

> *Expected outcome.* You need to specify the `THEPROGRAM` variable on the commandline using the syntax `make VAR=value`.
>
> *Caveats.* Make sure that there are no spaces around the equals sign in your commandline.

The full list of these automatic variables can be found at https://www.gnu.org/software/make/manual/html_node/Automatic-Variables.html.

### 3.3.2 Template rules

So far, you wrote a separate rule for each file that needed to be compiled. However, the rules for the various `.c` files are very similar:

- the rule header (`foo.o : foo.c`) states that a source file is a prerequisite for the object file with the same base name;
- and the instructions for compiling (`${CC} -c $<`) are even character-for-character the same, now that you are using *Make*'s built-in variables;
- the only rule with a difference is

  ```
  foo.o : foo.c bar.h
          ${CC} -c $<
  ```

  where the object file depends on the source file and another file.

We can take the commonalities and summarize them in one *template rule*[1]:

```
%.o : %.c
        ${CC} -c $<
%.o : %.F
        ${FC} -c $<
```

This states that any object file depends on the C or Fortran file with the same base name. To regenerate the object file, invoke the C or Fortran compiler with the `-c` flag. These template rules can function as a replacement for the multiple specific targets in the makefiles above, except for the rule for `foo.o`.

The dependence of `foo.o` on `bar.h`, or `foomain.o` on `foomod.o`, can be handled by adding a rule

```
# C
foo.o : bar.h
# Fortran
foomain.o : foomod.o
```

1. This mechanism is the first instance you'll see that only exists in GNU make, though in this particular case there is a similar mechanism in standard make. That will not be the case for the wildcard mechanism in the next section.

with no further instructions. This rule states, 'if file `bar.h` or `foomod.o` changed, file `foo.o` or `foomain.o` needs updating' too. *Make* will then search the makefile for a different rule that states how this updating is done, and it will find the template rule.

**Exercise.** Change your makefile to incorporate these ideas, and test.

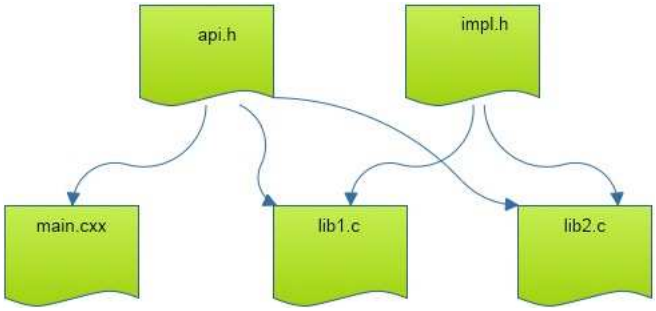



Figure 3.1: File structure with main program and two library files.

**Exercise** 3.1. Write a makefile for the following structure:

- There is one main file `libmain.cxx`, and two library files `libf.cxx` `libg.cxx`;
- There is a header file `libapi.h` that gives the prototypes for the functions in the library files;
- There is a header file `libimpl.h` that gives implementation details, only to be used in the library files.

This is illustrated in figure 3.1.

Here is how you can test it:

Changing a source file only recompiles that files:

```
clang++ -c libf.cxx
clang++ -o main \
    libmain.o libf.o libg.o
```

Changing the implementation header only recompiles the library:

```
clang++ -c libf.cxx
clang++ -c libg.cxx
clang++ -o main libmain.o libf.o
    libg.o
```

Changing the `libapi.h` recompiles everything:

```
clang++ -c libmain.cxx
clang++ -c libf.cxx
clang++ -c libg.cxx
clang++ -o main libmain.o libf.o
    libg.o
```

For Fortran we don't have header files so we use *modules* everywhere; figure 3.3. If you know how to use *submodules*, a *Fortran2008* feature, you can make the next exercise as efficient as the C version.

**Exercise** 3.2. Write a makefile for the following structure:

- There is one main file `libmain.f90`, that uses a module `api.f90`;
- There are two low level modules `libf.f90` `libg.f90` that are used in `api.f90`.

If you use modules, you'll likely be doing more compilation than needed. For the optimal solution, use submodules.

Source file `mainprog.cxx`:

```
#include <cstdio>
#include "api.h"

int main() {
    int n = f() + g();
    printf("%d\n",n);
    return 0;
}
```

Source file `libf.cxx`:

```
#include "impl.h"
#include "api.h"

int f() {
    struct impl_struct foo;
    return 1;
};
```

Source file `libg.cxx`:

```
#include "impl.h"
#include "api.h"

int g() {
    struct impl_struct foo;
    return 2;
};
```

Header file `api.h`:

```
int f();
int g();
```

Header file `impl.h`:

```
struct impl_struct {};
```

Figure 3.2: Source files for exercise 3.1.

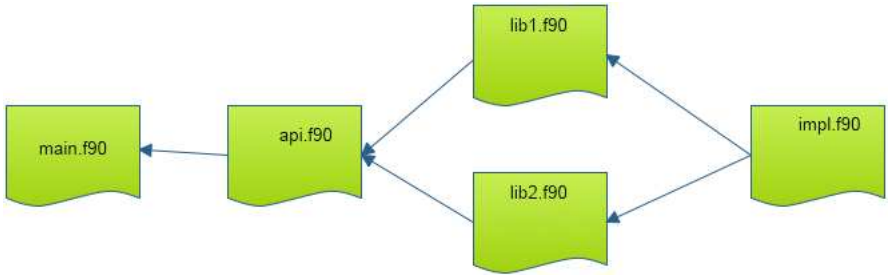



Figure 3.3: File structure with main program and two library files.

### 3.3.3 Wildcards

Your makefile now uses one general rule for compiling any source file. Often, your source files will be all the `.c` or `.F` files in your directory, so is there a way to state 'compile everything in this directory'? Indeed there is.

Add the following lines to your makefile, and use the variable `COBJECTS` or `FOBJECTS` wherever appropriate. The command *`wildcard`* gives the result of `ls`, and you can manipulate file names with *`patsubst`*.

```
# wildcard: find all files that match a pattern
CSOURCES := ${wildcard *.c}
# pattern substitution: replace one pattern string by another
COBJECTS := ${patsubst %.c,%.o,${SRC}}

FSOURCES := ${wildcard *.F}
FOBJECTS := ${patsubst %.F,%.o,${SRC}}
```

### 3.3.4 More functions

GNU make has more function that you can call inside the makefile. Some examples:

```
HOSTNAME := $(shell hostname -f)
SOURCES  := $(wildcard *.c)
OBJECTS  := $(patsubst %.c,%.o,${SOURCES})
RECURSIVE := $(foreach d,${DIRECTORIES},$(wildcard ${d}/*.c))
```

File name manipulation:

```
$(dir a/b/c.x)    # gives `a/b'
$(dir c.x)        # gives `./'
$(notdir a/b/c.x) # gives `c.x'
$(suffix a/b/c.x) # gives `.x'
```

For the full list see https://www.gnu.org/software/make/manual/html_node/Functions.html.

```
// Makefile                                    SRC: src/f1.c src/f2.c
SRC := ${wildcard src/*.c}

OBJ := ${patsubst src/%,obj/%,${patsubst %.c    OBJ: obj/f1.o obj/f2.o
    ,%.o,${SRC}}}

PRE := ${addprefix /usr/lib,${SRC} othersrc     PRE: /usr/libsrc/f1.c /usr/libsrc/f2.c /usr/
    moresrc}                                         libothersrc /usr/libmoresrc


BAK := ${addsuffix .bak,${SRC}}                 BAK: src/f1.c.bak src/f2.c.bak
```

### 3.3.5 Conditionals

There are various ways of making the behavior of a makefile dynamic. You can for instance put a shell conditional in an action line. However, this can make for a cluttered makefile; an easier way is to use makefile conditionals. There are two types of conditionals: tests on string equality, and tests on environment variables.

The first type looks like

```
ifeq "${HOME}" "/home/thisisme"
  # case where the executing user is me
else ifeq "${HOME}" "/home/buddyofmine"
  # case for other user
else
  # case where it's someone else
endif
```

and in the second case the test looks like

```
ifdef SOME_VARIABLE
```

The text in the true and false part can be most any part of a makefile. For instance, it is possible to let one of the action lines in a rule be conditionally included. However, most of the times you will use conditionals to make the definition of variables dependent on some condition.

**Exercise.** Let's say you want to use your makefile at home and at work. At work, your employer has a paid license to the Intel compiler `icc`, but at home you use the open source Gnu compiler `gcc`. Write a makefile that will work in both places, setting the appropriate value for `CC`.

## 3.4 Miscellania

### 3.4.1 Phony targets

The example makefile contained a target `clean`. This uses the *Make* mechanisms to accomplish some actions that are not related to file creation: calling `make clean` causes *Make* to reason 'there is no file called `clean`, so the following instructions need to be performed'. However, this does not actually cause a file `clean` to spring into being, so calling `make clean` again will make the same instructions being executed.

To indicate that this rule does not actually make the target, you use the *.PHONY* keyword:

```
.PHONY : clean
```

Most of the time, the makefile will actually work fine without this declaration, but the main benefit of declaring a target to be phony is that the *Make* rule will still work, even if you have a file (or folder) named `clean`.

### 3.4.2 Directories

It's a common strategy to have a directory for temporary material such as object files. So you would have a rule

```
obj/%.o : %.c
    ${CC} -c $< -o $@
```

and to remove the temporaries:

```
clean ::
    rm -rf obj
```

This raises the question how the `obj` directory is created. You could do:

```
obj/%.o : %.c
    mkdir -p obj
    ${CC} -c $< -o $@
```

but a better solution is to use *order-only prerequisite*s exist.

```
obj :
    mkdir -p obj
obj/%.o : %.c | obj
    ${CC} -c $< -o $@
```

This only tests for the existence of the object directory, but not its timestamp.

### 3.4.3 Using the target as prerequisite

Suppose you have two different targets that are treated largely the same. You would want to write:

```
PROGS = myfoo other
${PROGS} : $@.o # this is wrong!!
        ${CC} -o $@ $@.o ${list of libraries goes here}
```

and saying `make myfoo` would cause

```
cc -c myfoo.c
cc -o myfoo myfoo.o ${list of libraries}
```

and likewise for `make other`. What goes wrong here is the use of `$@.o` as prerequisite. In Gnu Make, you can repair this as follows[2]:

```
.SECONDEXPANSION:
${PROGS} : $$@.o
        ${CC} -o $@ $@.o ${list of libraries goes here}
```

**Exercise.** Write a second main program `foosecond.c` or `foosecond.F`, and change your makefile so that the calls `make foo` and `make foosecond` both use the same rule.

### 3.4.4 Predefined variables and rules

Calling `make -p yourtarget` causes make to print out all its actions, as well as the values of all variables and rules, both in your makefile and ones that are predefined. If you do this in a directory where there is no makefile, you'll see that make actually already knows how to compile `.c` or `.F` files. Find this rule and find the definition of the variables in it.

You see that you can customize make by setting such variables as `CFLAGS` or `FFLAGS`. Confirm this with some experimentation. If you want to make a second makefile for the same sources, you can call `make -f othermakefile` to use this instead of the default `Makefile`.

Note, by the way, that both `makefile` and `Makefile` are legitimate names for the default makefile. It is not a good idea to have both `makefile` and `Makefile` in your directory.

2. Technical explanation: Make will now look at lines twice: the first time `$$` gets converted to a single `$`, and in the second pass `$@` becomes the name of the target.

## 3.5 Shell scripting in a Makefile

**Purpose.** In this section you will see an example of a longer shell script appearing in a makefile rule.

In the makefiles you have seen so far, the command part was a single line. You can actually have as many lines there as you want. For example, let us make a rule for making backups of the program you are building.

Add a `backup` rule to your makefile. The first thing it needs to do is make a backup directory:

```
.PHONY : backup
backup :
        if [ ! -d backup ] ; then
          mkdir backup
        fi
```

Did you type this? Unfortunately it does not work: every line in the command part of a makefile rule gets executed as a single program. Therefore, you need to write the whole command on one line:

```
backup :
        if [ ! -d backup ] ; then mkdir backup ; fi
```

or if the line gets too long:

```
backup :
        if [ ! -d backup ] ; then \
          mkdir backup ; \
        fi
```

(Writing a long command on a single is only possible in the *bash* shell, not in the *csh* shell. This is one reason for not using the latter.)

Next we do the actual copy:

```
backup :
        if [ ! -d backup ] ; then mkdir backup ; fi
        cp myprog backup/myprog
```

But this backup scheme only saves one version. Let us make a version that has the date in the name of the saved program.

The Unix `date` command can customize its output by accepting a format string. Type the following: `date` This can be used in the makefile.

**Exercise.** Edit the `cp` command line so that the name of the backup file includes the current date.

*Expected outcome.* Hint: you need the backquote. Consult the Unix tutorial, section 1.5.3, if you do not remember what backquotes do.

If you are defining shell variables in the command section of a makefile rule, you need to be aware of the following. Extend your `backup` rule with a loop to copy the object files:

```
#### This Script Has An ERROR!
backup :
        if [ ! -d backup ] ; then mkdir backup ; fi
        cp myprog backup/myprog
        for f in ${OBJS} ; do \
          cp $f backup ; \
        done
```

(This is not the best way to copy, but we use it for the purpose of demonstration.) This leads to an error message, caused by the fact that *Make* interprets `$f` as an environment variable of the outer process. What works is:

```
backup :
        if [ ! -d backup ] ; then mkdir backup ; fi
        cp myprog backup/myprog
        for f in ${OBJS} ; do \
          cp $$f backup ; \
        done
```

(In this case *Make* replaces the double dollar by a single one when it scans the commandline. During the execution of the commandline, `$f` then expands to the proper filename.)

## 3.6 Practical tips for using Make

Here are a couple of practical tips.

- *Debugging* a makefile is often frustratingly hard. Just about the only tool is the `-p` option, which prints out all the rules that Make is using, based on the current makefile.
- You will often find yourself first typing a make command, and then invoking the program. Most Unix shells allow you to use commands from the *shell command history* by using the up arrow key. Still, this may get tiresome, so you may be tempted to write

  ```
  make myprogram ; ./myprogram -options
  ```

  and keep repeating this. There is a danger in this: if the make fails, for instance because of compilation problems, your program will still be executed. Instead, write

  ```
  make myprogram && ./myprogram -options
  ```

  which executes the program conditional upon make concluding successfully.

### 3.6.1 What does this makefile do?

Above you learned that issuing the `make` command will automatically execute the first rule in the makefile. This is convenient in one sense[3], and inconvenient in another: the only way to find out what possible actions a makefile allows is to read the makefile itself, or the – usually insufficient – documentation.

A better idea is to start the makefile with a target

3. There is a convention among software developers that a package can be installed by the sequence `./configure ; make ; make install`, meaning: Configure the build process for this computer, Do the actual build, Copy files to some system directory such as `/usr/bin`.

```
info :
        @echo "The following are possible:"
        @echo "  make"
        @echo "  make clean"
```

Now `make` without explicit targets informs you of the capabilities of the makefile.

If your makefile gets longer, you might want to document each section like this. This runs into a problem: you can not have two rules with the same target, `info` in this case. However, if you use a double colon it *is* possible. Your makefile would have the following structure:

```
info ::
        @echo "The following target are available:"
        @echo "  make install"
install :
        # ..... instructions for installing
info ::
       @echo "  make clean"
clean :
       # ..... instructions for cleaning
```

## 3.7 A Makefile for LaTeX

The *Make* utility is typically used for compiling programs, but other uses are possible too. In this section, we will discuss a makefile for LaTeX documents.

We start with a very basic makefile:

```
info :
        @echo "Usage: make foo"
        @echo "where foo.tex is a LaTeX input file"

%.pdf : %.tex
        pdflatex $<
```

The command `make myfile.pdf` will invoke `pdflatex myfile.tex`, if needed, once. Next we repeat invoking `pdflatex` until the log file no longer reports that further runs are needed:

```
%.pdf : %.tex
        pdflatex $<
        while [ `cat ${basename $@}.log | grep "Rerun to get" \
            | wc -l` -gt 0 ] ; do \
          pdflatex $< ; \
        done
```

We use the `${basename fn}` macro to extract the base name without extension from the target name.

In case the document has a bibliography or index, we run `bibtex` and `makeindex`.

```
%.pdf : %.tex
        pdflatex ${basename $@}
        -bibtex ${basename $@}
        -makeindex ${basename $@}
        while [ `cat ${basename $@}.log | grep "Rerun to get" \
                 | wc -l` -gt 0 ] ; do \
          pdflatex ${basename $@} ; \
        done
```

The minus sign at the start of the line means that *Make* should not exit if these commands fail.

Finally, we would like to use *Make*'s facility for taking dependencies into account. We could write a makefile that has the usual rules

```
mainfile.pdf : mainfile.tex includefile.tex
```

but we can also discover the include files explicitly. The following makefile is invoked with

```
make pdf TEXTFILE=mainfile
```

The `pdf` rule then uses some shell scripting to discover the include files (but not recursively), and it calls *Make* again, invoking another rule, and passing the dependencies explicitly.

```
pdf :
        export includes=`grep "^.input " ${TEXFILE}.tex \
            | awk '{v=v FS $$2".tex"} END {print v}'` ; \
        ${MAKE} ${TEXFILE}.pdf INCLUDES="$$includes"

%.pdf : %.tex ${INCLUDES}
        pdflatex $< ; \
        while [ `cat ${basename $@}.log \
                | grep "Rerun to get" | wc -l` -gt 0 ] ; do \
          pdflatex $< ; \
        done
```

This shell scripting can also be done outside the makefile, generating the makefile dynamically.

# Chapter 4

# The Cmake build system

## 4.1 CMake as build system

*CMake* is a general build system that uses other systems such as *Make* as a back-end. The general workflow is:

1. The configuration stage. Here the `CMakeLists.txt` file is parsed, and a `build` directory populated. This typically looks like:

   ```
   mkdir build
   cd build
   cmake <source location>
   ```

   Some people create the build directory in the source tree, in which case the *CMake* command is

   ```
   cmake ..
   ```

   Others put the build directory next to the source, in which case:

   ```
   cmake ../src_directory
   ```

2. The build stage. Here the installation-specific compilation in the build directory is performed. With *Make* as the 'generator' this would be

   ```
   cd build
   make
   ```

   but more generally

   ```
   cmake --build <build directory>
   ```

   Alternatively, you could use generators such as *ninja*, *Visual Studio*, or *XCode*:

   ```
   cmake -G ninja
     ## the usual arguments
   ```

3. The install stage. This can move binary files to a permanent location, such as putting library files in `/usr/lib`:

   ```
   make install
   ```

   or

   ```
   cmake --install <build directory>
   ```

| | |
|---|---|
| General directives | |
| `cmake_minimum_required` | specify minimum cmake version |
| `project` | name and version number of this project |
| `install` | specify directory where to install targets |
| Project building directives | |
| `add_executable` | specify executable name |
| `add_library` | specify library name |
| `add_subdirectory` | specify subdirectory where cmake also needs to run |
| `target_sources` | specify sources for a target |
| `target_link_libraries` | specify executable and libraries to link into it |
| `target_include_directories` | specify include directories, privately or publicly |
| `find_package` | other package to use in this build |
| Utility stuff | |
| `target_compile_options` | literal options to include |
| `target_compile_features` | things that will be translated by cmake into options |
| `target_compile_definitions` | macro definitions to be set private or publicly |
| `file` | define macro as file list |
| `message` | Diagnostic to print, subject to level specification |
| Control | |
| `if() else() endif()` | conditional |

Table 4.1: Cmake commands.

However, the install location already has to be set in the configuration stage. We will see later in detail how this is done.

Summarizing, the out-of-source workflow as advocated in this tutorial is

```
ls some_package_1.0.0 # we are outside the source
ls some_package_1.0.0/CMakeLists.txt # source contains cmake file
mkdir builddir && cd builddir # goto build location
cmake -D CMAKE_INSTALL_PREFIX=../installdir \
      ../some_package_1.0.0
make
make install
```

The resulting directory structure is illustrated in figure 4.1.


Figure 4.1: In-source (left) and out-of-source (right) build schemes.

### 4.1.1 Target philosophy

Modern *CMake* works through declaring targets and their requirements. Targets are things that will be built, and requirements can be such things as

- source files
- include paths and other paths
- compiler and linker options.

There is a distinction between build requirements, the paths/files/options used during building, and usage requirements, the paths/files/options that need to be in force when your targets are being used.

For requirements during building:

```
target_some_requirement( <target> PRIVATE <requirements> )
```

Usage requirements:

```
target_some_requirement( <target> INTERFACE <requirements> )
```

Combined:

```
target_some_requirement( <target> PUBLIC <requirements> )
```

For future reference, requirements are specified with `target_include_directories`, `target_compile_definitions`, `target_compile_options`, `target_compile_features`, `target_sources`, `target_link_libraries`, *target_link_options*, `target_link_directories`.

### 4.1.2 Languages

*CMake* is largely aimed at C++, but it easily supports C as well. For *Fortran* support, first do

```
enable_language(Fortran)
```

Note that capitalization: this also holds for all variables such as `CMAKE_Fortran_COMPILER`.

### 4.1.3 Script structure

| Commands learned in this section | |
|---|---|
| `cmake_minimum_required` | declare minimum required version for this script |
| `project` | declare a name for this project |

*CMake* is driven by the `CMakeLists.txt` file. This needs to be in the root directory of your project. (You can additionally have files by that name in subdirectories.)

Since *CMake* has changed quite a bit over the years, and is still evolving, it is a good idea to start each script with a declaration of the (minimum) required version:

```
cmake_minimum_required( VERSION 3.12 )
```

You can query the version of your *CMake* executable:

```
$ cmake --version
cmake version 3.19.2
```

You also need to declare a project name and version, which need not correspond to any file names:

```
project( myproject VERSION 1.0 )
```

## 4.2 Examples cases

### 4.2.1 Executable from sources

(The files for this examples are in `tutorials/cmake/single`.)

| Commands learned in this section | |
|---|---|
| `add_executable` | declare an executable as target |
| `target_sources` | specify sources for a target |
| `install` | indicate location where to install this project |
| `PROJECT_NAME` | macro that expands to the project name |

If you have a project that is supposed to deliver an executable, you declare in your `CMakeLists.txt`:

```
add_executable( myprogram )
target_sources( myprogram PRIVATE program.cxx )
```

Often, the name of the executable is the name of the project, so you'd specify:

```
add_executable( ${PROJECT_NAME} )
```

**Remark 14** *An older usage is to specify the sources files directly with the executable. The above usage is preferred.*

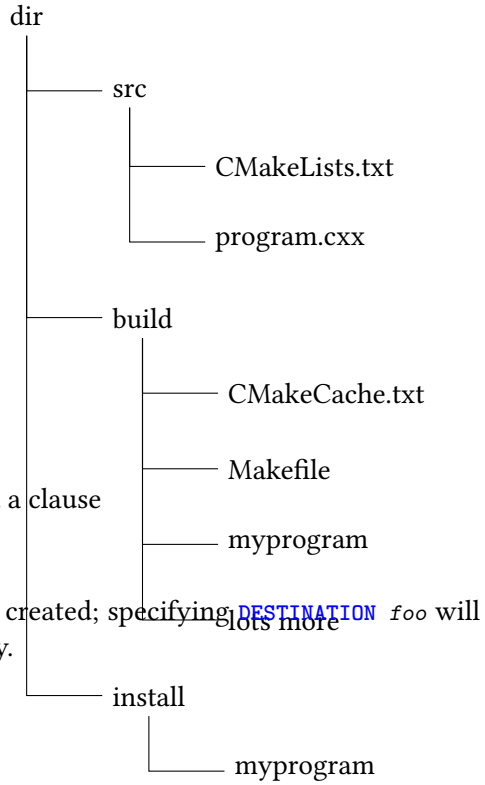
```
add_executable( myprogram program.cxx )
```

In order to move the executable to the install location, you need a clause

```
install( TARGETS myprogram DESTINATION . )
```

Without the `DESTINATION` clause, a default `bin` directory will be created; specifying `DESTINATION foo` will put the target in a `foo` sub-directory of the installation directory.

In the figure on the right we have also indicated the `build` directory, which from now on we will not show again. It contains automatically generated files that are hard to decyper, or debug. Yes, there is a `Makefile`, but even for simple projects this is too complicated to debug by hand if your *CMake* installation misbehaves.




Here is the full `CMakeLists.txt`:

```
cmake_minimum_required( VERSION 3.12 )
project( singleprogram VERSION 1.0 )

add_executable( program )
target_sources( program PRIVATE program.cxx )
install( TARGETS program DESTINATION . )
```

### 4.2.2 Making libraries

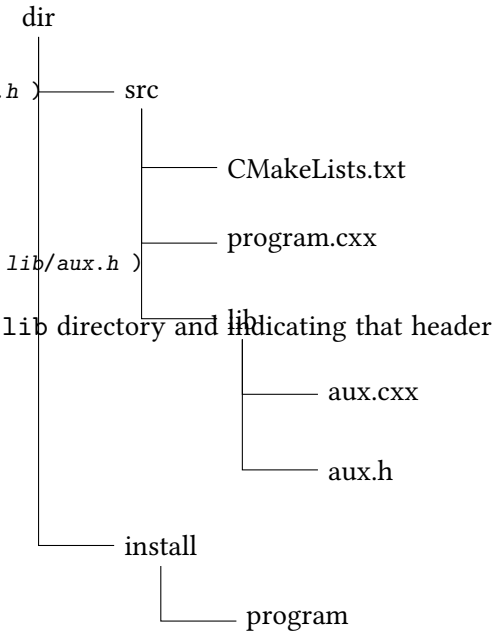
| Commands learned in this section | |
|---|---|
| `target_include_directories` | indicate location of include files |
| `add_library` | declare a library and its sources |
| `target_link_libraries` | indicate that the library belong with an executable |

#### *4.2.2.1 Multiple files*

(The files for this example are in `tutorials/cmake/multiple`.)

If there is only one source file, the previous section is all you need. If there are multiple files, you could write

```
add_executable( program )
target_sources( program PRIVATE program.cxx aux.cxx aux.h )
```




You can also put the non-main source files in a separate directory:

```
add_executable( program )
target_sources( program PRIVATE program.cxx lib/aux.cxx lib/aux.h )
```

However, often you will build libraries. We start by making a `lib` directory and indicating that header files need to be found there:

```
target_include_directories( program PRIVATE lib )
```

The actual library is declared with an `add_library` clause:

```
add_library( auxlib )
target_sources( auxlib PRIVATE aux.cxx aux.h )
```

Next, you need to link that library into the program:

```
target_link_libraries( program PRIVATE auxlib )
```

The `PRIVATE` clause means that the library is only for purposes of building the executable. (Use `PUBLIC` to have the library be included in the installation; we will explore that in section 4.2.2.3.)

The full `CMakeLists.txt`:

```
cmake_minimum_required( VERSION 3.12 )
project( cmakeprogram VERSION 1.0 )

add_executable( program )
target_sources( program PRIVATE program.cxx )

add_library( auxlib STATIC )
target_sources( auxlib PRIVATE aux1.cxx aux2.cxx aux.h )

target_link_libraries( program PRIVATE auxlib )
install( TARGETS program DESTINATION . )
```

Note that private shared libraries make no sense, as they will give runtime unresolved references.

#### *4.2.2.2 Testing the generated makefiles*

In the Make tutorial 3 you learned how Make will only recompile the strictly necessary files when a limited edit has been made. The makefiles generated by *CMake* behave similarly. With the structure above, we first touch the `aux.cxx` file, which necessitates rebuilding the library:

```
----------------
touch a source file and make:
Consolidate compiler generated dependencies of target auxlib
[ 25%] Building CXX object CMakeFiles/auxlib.dir/aux.cxx.o
[ 50%] Linking CXX static library libauxlib.a
[ 50%] Built target auxlib
Consolidate compiler generated dependencies of target program
[ 75%] Linking CXX executable program
[100%] Built target program
```

On the other hand, if we edit a header file, the main program needs to be recompiled too:

```
----------------
touch a source file and make:
Consolidate compiler generated dependencies of target auxlib
[ 25%] Building CXX object CMakeFiles/auxlib.dir/aux.cxx.o
[ 50%] Linking CXX static library libauxlib.a
[ 50%] Built target auxlib
Consolidate compiler generated dependencies of target program
[ 75%] Linking CXX executable program
[100%] Built target program
```

#### 4.2.2.3 *Making a library for release*

(The files for this example are in `tutorials/cmake/withlib`.)

| Commands learned in this section | |
|---|---|
| `SHARED` | indicated to make shared libraries |

In order to create a library we use `add_library`, and we link it into the target program with `target_link_libraries`.

By default the library is build as a static `.a` file, but adding

```
add_library( auxlib SHARED )
```

or adding a runtime flag

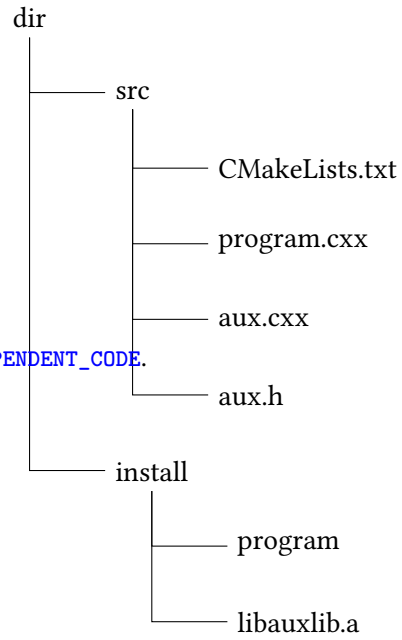
```
cmake -D BUILD_SHARED_LIBS=TRUE
```

changes that to a shared `.so` type.

Related: the *-fPIC* compile option is set by `CMAKE_POSITION_INDEPENDENT_CODE`.

The full *CMake* file:

```
cmake_minimum_required( VERSION 3.12 )
project( cmakeprogram VERSION 1.0 )

add_executable( program )
target_sources( program PRIVATE program.cxx )

add_library( auxlib STATIC )
target_sources( auxlib PRIVATE lib/aux.cxx lib/aux.h )

target_link_libraries( program PUBLIC auxlib )
target_include_directories( program PRIVATE lib )

install( TARGETS program DESTINATION bin )
install( TARGETS auxlib DESTINATION lib )
install( FILES lib/aux.h DESTINATION include )
```



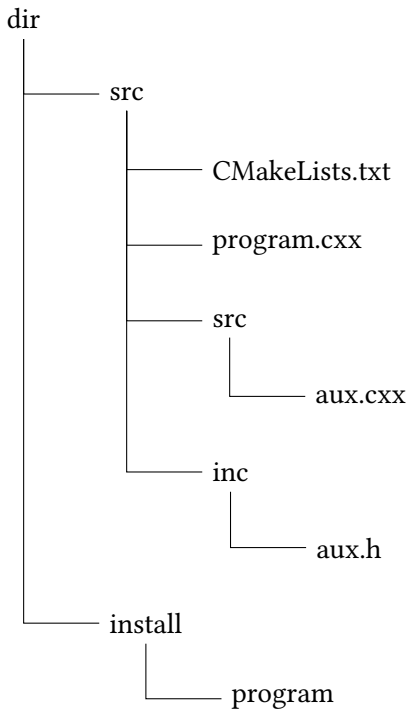

Note that we give a path to the library files. This is interpreted relative to the current directory, (as of CMake-3.13); this current directory is available as `CMAKE_CURRENT_SOURCE_DIR`.

### 4.2.3 Using subdirectories during the build

(The files for this examples are in `tutorials/cmake/includedir`.)

| Commands learned in this section | |
|---|---|
| `target_include_directories` | indicate include directories needed |
| `target_sources` | specify more sources for a target |
| `CMAKE_CURRENT_SOURCE_DIR` | variable that expands to the current directory |
| `file` | define single-name synonym for multiple files |
| `GLOB` | define single-name synonym for multiple files |

Suppose you have a directory with header files, as in the diagram on the right. The main program would have




```
#include <iostream>
using namespace std;

#include "aux.h"

int main() {
  aux1();
  aux2();
  return 0;
}
```

and which is compiled as:

```
cc -c program.cxx -I./inc
```

To make sure the header file gets found during the build, you specify that include path with `target_include_directories`:

```
target_include_directories(
        program PRIVATE
        "${CMAKE_CURRENT_SOURCE_DIR}/inc" )
```

It is best to make such paths relative to `CMAKE_CURRENT_SOURCE_DIR`, or the source root `CMAKE_SOURCE_DIR`, or equivalently `PROJECT_SOURCE_DIR`

Usually, when you start making such directory structure, you will also have sources in subdirectories. If you only need to compile them into the main executable, you could list them into a variable

```
set( SOURCES program.cxx src/aux.cxx )
```

and use that variable. However, this is deprecated practice; it is recommended to use `target_sources`:

```
target_sources( program PRIVATE src/aux1.cxx src/aux2.cxx )
```

Use of a wildcard is not trivial:

```
file( GLOB AUX_FILES "src/*.cxx" )
target_sources( program PRIVATE ${AUX_FILES} )
```

Complete *CMake* file:

```
cmake_minimum_required( VERSION 3.14 )
project( cmakeprogram VERSION 1.0 )

add_executable( program )
target_sources( program PRIVATE program.cxx )
## target_sources( program PRIVATE src/aux1.cxx src/aux2.cxx )
file( GLOB AUX_FILES "src/*.cxx" )
target_sources( program PRIVATE ${AUX_FILES} )
target_include_directories(
        program PRIVATE
        "${CMAKE_CURRENT_SOURCE_DIR}/inc" )

install( TARGETS program DESTINATION . )
```

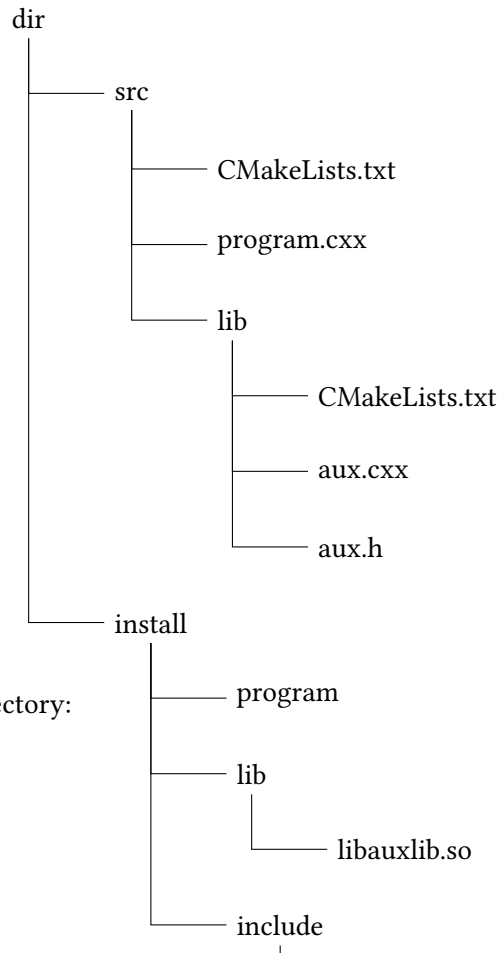
### 4.2.4 Recursive building; rpath

(The files for this examples are in `tutorials/cmake/publiclib`.)

| Commands learned in this section | |
|---|---|
| `add_subdirectory` | declare a subdirectory where cmake needs to be run |
| `CMAKE_CURRENT_SOURCE_DIR` | directory where this command is evaluated |
| `CMAKE_CURRENT_BINARY_DIR` | |
| `LIBRARY_OUTPUT_PATH` | |
| `FILES_MATCHING PATTERN` | wildcard indicator |

If your sources are spread over multiple directories, there needs to be a `CMakeLists.txt` file in each, and you need to declare the existence of those directories. Let's start with the obvious choice of putting library files in a `lib` directory with `add_subdirectory`:

```
add_subdirectory( lib )
```




For instance, a library directory would have a `CMakeLists.txt` file:

```
cmake_minimum_required( VERSION 3.14 )
project( auxlib )

add_library( auxlib SHARED )
target_sources( auxlib PRIVATE aux.cxx aux.h )
target_include_directories(
    auxlib PUBLIC "${CMAKE_CURRENT_SOURCE_DIR}" )
install( TARGETS auxlib DESTINATION lib )
install( FILES aux.h DESTINATION include )
```

to build the library file from the sources indicated, and to install it in a `lib` subdirectory.

We also add a clause to install the header files in an include directory:

```
install( FILES aux.h DESTINATION include )
```

For installing multiple files, use

```
install(DIRECTORY ${CMAKE_CURRENT_BINARY_DIR}
    DESTINATION ${LIBRARY_OUTPUT_PATH}
    FILES_MATCHING PATTERN "*.h")
```

One problem is to tell the executable where to find the library. For this we use the *rpath* mechanism. (See section 2.3.3.) By default, *CMake* sets it so that the executable in the build location can find the library. If you use a non-trivial install prefix, the following lines work:

```
set( CMAKE_INSTALL_RPATH "${CMAKE_INSTALL_PREFIX}/lib" )
set( CMAKE_INSTALL_RPATH_USE_LINK_PATH TRUE )
```

Note that these have to be specified before the target.

The whole file:

```
cmake_minimum_required( VERSION 3.14 )
project( cmakeprogram VERSION 1.0 )

set( CMAKE_INSTALL_RPATH "${CMAKE_INSTALL_PREFIX}/lib" )
set( CMAKE_INSTALL_RPATH_USE_LINK_PATH TRUE )

add_executable( program )
target_sources( program PRIVATE program.cxx )
add_subdirectory( lib )
target_include_directories(
        auxlib PUBLIC "${CMAKE_CURRECT_SOURCE_DIR}" )
target_link_libraries(
        program PUBLIC auxlib )

install( TARGETS program DESTINATION . )
```

### 4.2.5 Programs that use other libraries

So far we have discussed executables and libraries that can be used by themselves. What to if your build result needs external libraries? We will discuss how to find those libraries in section 4.3; here we point out their use.

The issue is that these libraries need to be findable when someone uses your binary. There are two strategies:

- make sure they have been added to the *LD_LIBRARY_PATH*;
- have the *linker* add their location through the *rpath* mechanism to the binary itself. This second option is in fact the only one on recent versions of *Apple Mac OS* because of 'System Integrity Protection'.

As an example, assume your binary needs the *Catch2* and *fmtlib* libraries. You would then, in addition to the `target_link_directories` specification, have:

```
set_target_properties(
        ${PROGRAM_NAME} PROPERTIES
        BUILD_RPATH   "${CATCH2_LIBRARY_DIRS};${FMTLIB_LIBRARY_DIRS}"
        INSTALL_RPATH "${CATCH2_LIBRARY_DIRS};${FMTLIB_LIBRARY_DIRS}"
    )
```

### 4.2.6 Header-only libraries

Use the `INTERFACE` keyword.

## 4.3 Finding and using external packages

If your program depends on other libraries, there is a variety of ways to let *CMake* find them.

### 4.3.1 CMake commandline options

(The files for this example are in `tutorials/cmake/usepubliclib`.)

You can indicate the location of your external library explicitly on the commandline.

```
cmake -D OTHERLIB_INC_DIR=/some/where/include
      -D OTHERLIB_LIB_DIR=/somewhere/lib
```

Example *CMake* file:

```
cmake_minimum_required( VERSION 3.12 )
project( pkgconfiglib VERSION 1.0 )

# with environment variables
# set( AUX_INCLUDE_DIR $ENV{TACC_AUX_INC} )
# set( AUX_LIBRARY_DIR $ENV{TACC_AUX_LIB} )

# with cmake -D options
```

```
option( AUX_INCLUDE_DIR "include dir for auxlib" )
option( AUX_LIBRARY_DIR   "lib dir for auxlib" )

add_executable( program )
target_sources( program PRIVATE program.cxx )
target_include_directories(
        program PUBLIC
        ${AUX_INCLUDE_DIR} )
target_link_libraries( program PUBLIC auxlib )
target_link_directories(
        program PUBLIC
        ${AUX_LIBRARY_DIR} )
install( TARGETS program DESTINATION . )
```

### 4.3.2 Package finding through 'find library' and 'find package'

| Commands learned in this section | |
|---|---|
| `find_library` | find a library with a `FOOConfig.cmake` file |
| `CMAKE_PREFIX_PATH` | used for finding `FOOConfig.cmake` files |
| `find_package` | find a library with a `FindFOO` module |
| `CMAKE_MODULE_PATH` | location for `FindFOO` modules |

The `find_package` command looks for files with a name `FindXXX.cmake`, which are searched on the `CMAKE_MODULE_PATH`. Unfortunately, the working of `find_package` depend somewhat on the specific package. For instance, most packages set a variable *FooFound* that you can test

```
find_package( Foo )
if ( FooFound )
    # do something
else()
    # throw an error
endif()
```

Some libraries come with a `FOOConfig.cmake` file, which is searched on the `CMAKE_PREFIX_PATH` through `find_library`. You typically set this variable to the root of the package installation, and CMake will find the directory of the `.cmake` file.

You can test the variables set:

```
find_library( FOOLIB foo )
if (FOOLIB)
  target_link_libraries( myapp PRIVATE ${FOOLIB} )
else()
  # throw an error
endif()
```

#### 4.3.2.1 *Example: Range-v3*

```
find_package( range-v3 REQUIRED )
target_link_libraries( ${PROGRAM_NAME} PUBLIC range-v3::range-v3 )
```

### 4.3.3 Use of other packages through 'pkg config'

These days, many package support the *pkgconfig* mechanism.

1. Suppose you have a library `mylib`, installed in `/opt/local/mylib`.
2. If `mylib` supports `pkgconfig`, there is most likely a path `/opt/local/mylib/lib/pkgconfig`, containing a file `mylib.pc`.
3. Add the path that contains the *.pc* file to the *PKG_CONFIG_PATH* environment variable.

Cmake is now able to find `mylib`:

```
find_package( PkgConfig REQUIRED )
pkg_check_modules( MYLIBRARY REQUIRED mylib )
```

This defines variables

```
MYLIBRARY_INCLUDE_DIRS
MYLIBRARY_LIBRARY_DIRS
MYLIBRARY_LIBRARIES
```

which you can then use in the `target_include_directories` and `target_link_directories target_link_libraries` commands.

### 4.3.4 Writing your own pkg config

We extend the configuration of section 4.2.4 to generate a `.pc` file.

First of all we need a template for the `.pc` file:

```
prefix="@CMAKE_INSTALL_PREFIX@"
exec_prefix="${prefix}"
libdir="${prefix}/lib"
includedir="${prefix}/include"

Name: @PROJECT_NAME@
Description: @CMAKE_PROJECT_DESCRIPTION@
Version: @PROJECT_VERSION@
Cflags: -I${includedir}
Libs: -L${libdir} -l@libtarget@
```

Here the at-signs delimit CMake macros that will be substituted when the `.pc` file is generated; Dollar macros go into the file unchanged.

Generating the file is done by the following lines in the CMake configuration:

```
set( libtarget auxlib )
configure_file(
    ${CMAKE_CURRENT_SOURCE_DIR}/${PROJECT_NAME}.pc.in
    ${CMAKE_CURRENT_BINARY_DIR}/${PROJECT_NAME}.pc
    @ONLY
)
install(
    FILES ${CMAKE_CURRENT_BINARY_DIR}/${PROJECT_NAME}.pc
    DESTINATION share/pkgconfig
)
```

### 4.3.5 Libraries

#### *4.3.5.1 Example: MPI*

(The files for this example are in `tutorials/cmake/mpiprog`.)

While many MPI implementations have a `.pc` file, it's better to use the FindMPI module. This package defines a number of variables that can be used to query the MPI found; for details see https://cmake.org/cmake/help/latest/module/FindMPI.html Sometimes it's necessary to set the *MPI_HOME* environment variable to aid in discovery of the MPI package.

C version:

```
cmake_minimum_required( VERSION 3.12 )
project( cxxprogram VERSION 1.0 )

add_executable( ${PROJECT_NAME} ${PROJECT_NAME}.cxx )
find_package( MPI )
target_include_directories(
        ${PROJECT_NAME} PUBLIC
        ${MPI_C_INCLUDE_DIRS} ${CMAKE_CURRENT_SOURCE_DIR} )
target_link_libraries(
        ${PROJECT_NAME} PUBLIC
        ${MPI_C_LIBRARIES} )

install( TARGETS ${PROJECT_NAME} DESTINATION . )
```

Fortran version:

```
cmake_minimum_required( VERSION 3.12 )
project( ${PROJECT_NAME} VERSION 1.0 )

enable_language(Fortran)

find_package( MPI )

if( MPI_Fortran_HAVE_F08_MODULE )
else()
  message( FATAL_ERROR "No f08 module for this MPI" )
endif()

add_executable( ${PROJECT_NAME} ${PROJECT_NAME}.F90 )
target_include_directories(
        ${PROJECT_NAME} PUBLIC
        ${MPI_Fortran_INCLUDE_DIRS} ${CMAKE_CURRENT_SOURCE_DIR} )
target_link_directories(
        ${PROJECT_NAME} PUBLIC
        ${MPI_LIBRARY_DIRS} )
target_link_libraries(
        ${PROJECT_NAME} PUBLIC
        ${MPI_Fortran_LIBRARIES} )

install( TARGETS ${PROJECT_NAME} DESTINATION . )
```

The *MPL* package also uses `find_package`:

```
find_package( mpl REQUIRED )
add_executable( ${PROJECT_NAME} )
target_sources( ${PROJECT_NAME} PRIVATE ${PROJECT_NAME}.cxx )
target_include_directories(
        ${PROJECT_NAME} PUBLIC
        mpl::mpl )
target_link_libraries(
        ${PROJECT_NAME} PUBLIC
        mpl::mpl )
```

#### 4.3.5.2 *Example: OpenMP*

```
find_package(OpenMP)
if(OpenMP_C_FOUND) # or CXX
else()
        message( FATAL_ERROR "Could not find OpenMP" )
endif()
# for C:
add_executable( ${program} ${program}.c )
target_link_libraries( ${program} PUBLIC OpenMP::OpenMP_C )
# for C++:
add_executable( ${program} ${program}.cxx )
target_link_libraries( ${program} PUBLIC OpenMP::OpenMP_CXX)
# for Fortran
enable_language(Fortran)
# test: if( OpenMP_Fortran_FOUND )
add_executable( ${program} ${program}.F90 )
target_link_libraries( ${program} PUBLIC OpenMP::OpenMP_Fortran )
```

#### 4.3.5.3 *CUDA*

```
FindCUDA

Changed in version 3.27: This module is available only if policy CMP0146 is not set to N

Deprecated since version 3.10: Do not use this module in new code.

It is no longer necessary to use this module or call find_package(CUDA) for compiling CU

New in version 3.17: To find and use the CUDA toolkit libraries manually, use the FindCU
```

#### 4.3.5.4 *Example: MKL*

(The files for this example are in `tutorials/cmake/mklcmake`.)

Intel compiler installations come with *CMake* support: there is a file `MKLConfig.cmake`.

Example program using *Cblas* from MKL:

```
#include <iostream>
```

```
#include <vector>
using namespace std;

#include "mkl_cblas.h"

int main() {
  vector<double> values{1,2,3,2,1};
  auto maxloc = cblas_idamax ( values.size(),values.data(),1);
  cout << "Max abs at: " << maxloc << " (s/b 2)" << '\n';

  return 0;
}
```

The following configuration file lists the various options and such:

```
cmake_minimum_required( VERSION 3.12 )
project( mklconfigfind VERSION 1.0 )

## https://www.intel.com/content/www/us/en/develop/documentation/onemkl-linux-developer-guide/
    top/getting-started/cmake-config-for-onemkl.html

find_package( MKL CONFIG REQUIRED )

add_executable( program program.cxx )
target_compile_options(
        program PUBLIC
        $<TARGET_PROPERTY:MKL::MKL,INTERFACE_COMPILE_OPTIONS> )
target_include_directories(
        program PUBLIC
        $<TARGET_PROPERTY:MKL::MKL,INTERFACE_INCLUDE_DIRECTORIES> )
target_link_libraries(
        program PUBLIC
        $<LINK_ONLY:MKL::MKL>)

install( TARGETS program DESTINATION . )
```

#### 4.3.5.5 *Example: PETSc*

(The files for this example are in `tutorials/cmake/petscprog`.)

This *CMake* setup searches for `petsc.pc`, which is located in `$PETSC_DIR/$PETSC_ARCH/lib/pkgconfig`:

```
cmake_minimum_required( VERSION 3.12 )
project( pkgconfiglib VERSION 1.0 )

find_package( PkgConfig REQUIRED )
pkg_check_modules( PETSC REQUIRED petsc )
message( STATUS "PETSc includes: ${PETSC_INCLUDE_DIRS}" )
message( STATUS "PETSc libraries: ${PETSC_LIBRARY_DIRS}" )

add_executable( program program.cxx )
target_include_directories(
        program PUBLIC
        ${PETSC_INCLUDE_DIRS} )
```

```
target_link_directories(
        program PUBLIC
        ${PETSC_LIBRARY_DIRS} )
target_link_libraries(
        program PUBLIC petsc )

install( TARGETS program DESTINATION . )
```

#### 4.3.5.6 Example: Eigen

(The files for this example are in `tutorials/cmake/eigen`.)

The *eigen* package uses *pkgconfig*.

```
cmake_minimum_required( VERSION 3.12 )
project( eigentest )

find_package( PkgConfig REQUIRED )
pkg_check_modules( EIGEN REQUIRED eigen3 )

add_executable( eigentest eigentest.cxx )
target_include_directories(
        eigentest PUBLIC
        ${EIGEN_INCLUDE_DIRS})
```

#### 4.3.5.7 Example: cxxopts

(The files for this example are in `tutorials/cmake/cxxopts`.)

The *cxxopts* package uses *pkgconfig*.

```
cmake_minimum_required( VERSION 3.12 )
project( pkgconfiglib VERSION 1.0 )

find_package( PkgConfig REQUIRED )
pkg_check_modules( OPTS REQUIRED cxxopts )
message( STATUS "cxxopts includes: ${OPTS_INCLUDE_DIRS}" )

add_executable( program program.cxx )
target_include_directories(
        program PUBLIC
        ${OPTS_INCLUDE_DIRS})

install( TARGETS program DESTINATION . )
```

#### 4.3.5.8 Example: fmtlib

(The files for this example are in `tutorials/cmake/fmtlib`.)

In the following example, we use the *fmtlib*. The main *CMake* file:

```
cmake_minimum_required( VERSION 3.12 )
project( pkgconfiglib VERSION 1.0 )
```

```
find_package( PkgConfig REQUIRED )
pkg_check_modules( FMTLIB REQUIRED fmt )
message( STATUS "fmtlib includes: ${FMTLIB_INCLUDE_DIRS}" )

add_executable( program program.cxx )
target_include_directories(
        program PUBLIC
        ${FMTLIB_INCLUDE_DIRS})

install( TARGETS program DESTINATION . )
```

#### 4.3.5.9 *Example: fmtlib used in library*

(The files for this example are in `tutorials/cmake/fmtliblib`.)

We continue using the *fmtlib* library, but now the generated library also has references to this library, so we use `target_link_directories` and `target_link_library`.

Main file:

```
cmake_minimum_required( VERSION 3.12 )
project( pkgconfiglib VERSION 1.0 )

find_package( PkgConfig REQUIRED )
pkg_check_modules( FMTLIB REQUIRED fmt )
message( STATUS "fmtlib includes : ${FMTLIB_INCLUDE_DIRS}" )
message( STATUS "fmtlib lib dirs : ${FMTLIB_LIBRARY_DIRS}" )
message( STATUS "fmtlib libraries: ${FMTLIB_LIBRARIES}" )

add_executable( program program.cxx )
target_include_directories(
        program PUBLIC
        ${FMTLIB_INCLUDE_DIRS})

add_subdirectory( prolib )
target_link_libraries( program PUBLIC prolib )

install( TARGETS program DESTINATION . )
```

Library file:

```
project( prolib )

add_library( prolib SHARED aux.cxx aux.h )
target_include_directories(
        prolib PUBLIC
        ${FMTLIB_INCLUDE_DIRS})
target_link_directories(
        prolib PUBLIC
        ${FMTLIB_LIBRARY_DIRS})
target_link_libraries(
        prolib PUBLIC fmt )
```

## 4.4 Customizing the compilation process

| Commands learned in this section | |
|---|---|
| `add_compile_options` | global compiler options |
| `target_compile_features` | compiler-independent specification of compile flags |
| `target_compile_definitions` | pre-processor flags |

### 4.4.1 Customizing the compiler

It's probably a good idea to tell *CMake* explicitly what compiler you are using, otherwise it may find some default gcc version that came with your system. Use the variables `CMAKE_CXX_COMPILER`, `CMAKE_C_COMPILER`, `CMAKE_Fortran_COMPILER`, `CMAKE_LINKER`.

Alternatively, set environment variables *CC*, *CXX*, *FC* by the explicit paths of the compilers. For examples, for Intel compilers:

```
export CC=`which icc`
export CXX=`which icpc`
export FC=`which ifort`
```

### 4.4.2 Global and target flags

Most of the time, compile options should be associated with a target. For instance, some file could need a higher or lower optimization level, or a specific C++ standard. In that case, use `target_compile_features`.

Certain options may need to be global, in which case you use `add_compile_options`. Example:

```
## from https://youtu.be/eC9-iRN2b04?t=1548
if (MVSC)
    add_compile_options(/W3 /WX)
else()
    add_compile_options(-W -Wall -Werror)
endif()
```

#### *4.4.2.1 Universal flags*

Certain flags have a universal meaning, but compiler-dependent realization. For instance, to specify the C++ standard:

```
target_compile_features( mydemo PRIVATE cxx_std_17 )
```

Alteratively, you can set this one the commandline:

```
cmake -D CMAKE_CXX_STANDARD=20
```

The variable `CMAKE_CXX_COMPILE_FEATURES` contains the list of all features you can set.

Optimization flags can be set by specifying the `CMAKE_BUILD_TYPE`:

- *Debug* corresponds to the `-g` flag;
- *Release* corresponds to `-O3 -DNDEBUG`;
- *MinSizeRel* corresponds to `-Os -DNDEBUG`
- *RelWithDebInfo* corresponds to `-O2 -g -DNDEBUG`.

This variable will often be set from the commandline:

```
cmake .. -DCMAKE_BUILD_TYPE=Release
```

Unfortunately, this seems to be the only way to influence optimization flags, other than explicitly setting compiler flags; see next point.

#### 4.4.2.2 *Custom compiler flags*

Set the variable `CMAKE_CXX_FLAGS` or `CMAKE_C_FLAGS`; also `CMAKE_LINKER_FLAGS` (but see section 4.2.4 for the popular *rpath* options.)

### 4.4.3 Macro definitions

*CMake* can provide macro definitions:

```
target_compile_definitions
  ( programname PUBLIC
    HAVE_HELLO_LIB=1 )
```

and your source could test these:

```
#ifdef HAVE_HELLO_LIB
#include "hello.h"
#endif
```

## 4.5 CMake scripting

| Commands learned in this section | |
|---|---|
| `option` | query a commandline option |
| `message` | trace message during cmake-ing |
| `set` | set the value of a variable |
| `CMAKE_SYSTEM_NAME` | variable containing the operating system name |
| `STREQUALS` | string comparison operator |

The *CMakeLists.txt* file is a script, though it doesn't much look like it.

- Instructions consist of a command, followed by a parenthesized list of arguments.
- (All arguments are strings: there are no numbers.)
- Each command needs to start on a new line, but otherwise whitespace and line breaks are ignored.

Comments start with a hash character.

### 4.5.1 System dependencies

```
if (CMAKE_SYSTEM_NAME STREQUALS "Window")
  target_compile_options( myapp PRIVATE /W4 )
elseif (CMAKE_SYSTEM_NAME STREQUALS "Darwin" -Wall -Wextra -Wpedantic)
  target_compile_options( myapp PRIVATE /W4 )
endif()
```

### 4.5.2 Messages, errors, and tracing

The `message` command can be used to write output to the console. This command has two arguments:

```
message( STATUS "We are rolling!")
```

Instead of `STATUS` you can specify other logging levels (this parameter is actually called 'mode' in the documentation); running for instance

```
cmake --log-level=NOTICE
```

will display only messages of 'notice' status or higher.

The possibilities here are: `FATAL_ERROR`, `SEND_ERROR`, `WARNING`, `AUTHOR_WARNING`, `DEPRECATION`, `NOTICE`, `STATUS`, `VERBOSE`, `DEBUG`, `TRACE`.

The `NOTICE`, `VERBOSE`, `DEBUG`, `TRACE` options were added in CMake-3.15.

For a complete trace of everything *CMake* does, use the commandline option `--trace`.

You can get a verbose make file by using the option

```
-D CMAKE_VERBOSE_MAKEFILE=ON
```

on the *CMake* invocation. You still need `make V=1`.

### 4.5.3 Variables

Variables are set with `set`, or can be given on the commandline:

```
cmake -D MYVAR=myvalue
```

where the space after `-D` is optional.

Using the variable by itself gives the value, except in strings, where a shell-like notation is needed:

```
set(SOME_ERROR "An error has occurred")
message(STATUS "${SOME_ERROR}")
set(MY_VARIABLE "This is a variable")
message(STATUS "Variable MY_VARIABLE has value ${MY_VARIABLE}")
```

Variables can also be queried by the *CMake* script using the `option` command:

```
option( SOME_FLAG "A flag that has some function" defaultvalue )
```

Some variables are set by other commands. For instance the `project` command sets `PROJECT_NAME` and `PROJECT_VERSION`.

#### 4.5.3.1 *Environment variables*

*Environment variables* can be queried with the `ENV` command:

```
set( MYDIR $ENV{MYDIR} )
```

#### 4.5.3.2 *Numerical variables*

```
math( EXPR lhs_var "math expr" )
```

### 4.5.4 Control structures

#### 4.5.4.1 *Conditionals*

```
if ( MYVAR MATCHES "value$" )
    message( NOTICE "Variable ended in 'value'" )
elseif( stuff )
    message( stuff )
else()
    message( NOTICE "Variable was otherwise" )
endif()
```

#### 4.5.4.2 *Looping*

```
while( myvalue LESS 50 )
    message( stuff )
endwhile()

foreach ( var IN ITEMS item1 item2 item3 )
  ## something wityh ${var}
endforeach()
foreach ( var IN LISTS list1 list2 list3 )
  ## something wityh ${var}
endforeach()
```

Integer range, with inclusive bounds, upper bound zero by default:

```
foreach ( idx RANGE 10 )
foreach ( idx RANGE 5 10 )
foreach ( idx RANGE 5 10 2 )
endforeach()
```

# Chapter 5

# Source code control through Git

In this tutorial you will learn *git*, the currently most popular *version control* (also *source code control* or *revision control*) systems. Other similar systems are *Mercurial* and *Microsoft Sharepoint*. Earlier systems were *SCCS*, *CVS*, *Subversion*, *Bitkeeper*.

Version control is a system that tracks the history of a software project, by recording the successive versions of the files of the project. These versions are recorded in a *repository*, either on the machine you are working on, or remotely.

This has many practical advantages:

- It becomes possible to undo changes;
- Sharing a repository with another developer makes collaboration possible, including multiple edits on the same file.
- A repository records the history of the project.
- You can have multiple versions of the project, for instance for exploring new features, or for customization for certain users.

The use of a version control system is industry standard practice, and git is by far the most popular system these days.

## 5.1 Concepts and overview

Older systems were based on having one *central repository*, that all developers coordinated with. These days a setup is popular where each developer (or a small group) has a *local repository*, which gets synchronized to a *remote repository*. In so-called *distributed version control* systems there can even be multiple remote repositories to synchronize with.

It is possible to track the changes of a single file, but often it makes sense to bundle a group of changes together in a *commit*. That also makes it easy to roll back such a group of changes.

If a project is in a state that constitutes some sort of a milestone, it is possible to attach a *tag*, or mark the state as a *release*.

Modern version control systems allow you to have multiple *branch*es, even in the same local repository. This way you can have a main branch for the release version of the project, and one or more development

branches for exploring new features. These branches can be merged when you're satisfied that a new feature has been sufficiently tested.

## 5.2 Git

This lab should be done two people, to simulate a group of programmers working on a joint project. You can also do this on your own by using two clones of the repository, preferably opening two windows on your computer.

## 5.3 Create and populate a repository

**Purpose.** In this section you will create a repository and make a local copy to work on.

You can create a repository two different ways:

1. Create the remote repository and do a clone.
2. Create the repository locally, and then connect it to a remote; this is a lot more work.

### 5.3.1 Create a repository by cloning

The easiest way to start a new repository is use a website such as *github.com* or *gitlab.com*, create the repository there, and then clone it to your local machine. Syntax:

```
git clone URL [ localname ]
```

This gives you a directory with the contents of the repository. If you leave out the local name, the directory will have the name of the repository.

```
Cmd >> git clone https://github.com/TACC/empty.git
    ↪empty
Out >>
Cloning into 'empty'...
warning: You appear to have cloned an empty repository.
Cmd >> cd empty
Cmd >> ls -a
Out >>
.
..
.git
Cmd >> git status
Out >>
On branch main
No commits yet
nothing to commit (create/copy files and use "git add"
    ↪to track)
```

Clone an empty repository and check that it is indeed empty

As you see, even an empty repository contains a directory `.git`. This contains bookkeeping information about your repository; you will hardly ever look into it.

### 5.3.2 Create a repository locally

You can also create a directory for your repository, and connect it to a remote site later. For this you do `git init` in the directory that is either empty, or already contains the material that you want to add later. Here, we start with an empty directory.

```
Cmd >> mkdir newrepo
Cmd >> cd newrepo
Cmd >> git init
Out >>
Initialized empty Git repository in
    ↪/users/demo/git/newrepo/.git/
Cmd >> ls -a
Out >>
.
..
.git
Cmd >> git status
Out >>
On branch master
No commits yet
nothing to commit (create/copy files and use "git add"
    ↪to track)
```

Create a directory, and nake it into a repository

The disadvantage of this method, over cloning an empty repo, is that you now have to connect your directory to a remote repository. See section 5.6.

### 5.3.3 Main vs master

It used to be that the default branch (yes, I know, we haven't discussed branches yet) was called 'master'. In a shift of terminology, the preferred name is now 'main'. Sites such as *github* and *gitlab* may already create this name by default; the git software does not do this, as of this writing in 2022.

Renaming a branch is possible. You can use `git status` to see what branch you are on.

```
Cmd >> git status
Out >>
On branch master
No commits yet
nothing to commit (create/copy files and use "git add"
    ↪to track)
```

See what the main branch is

Move the current branch:

```
Cmd >> git branch -m main
Cmd >> git status
Out >>
On branch main
No commits yet
nothing to commit (create/copy files and use "git add"
    ↪to track)
```

Rename the branch

For good measure, check the name again with `git status`.

```
Cmd >> git branch -m main
Cmd >> git status
Out >>
On branch main
No commits yet
nothing to commit (create/copy files and use "git add"
    ↪to track)
```

Check branch names

## 5.4 Adding and changing files

### 5.4.1 Creating a new file

If you create a file it does not automatically become part of your repository. This takes a sequence of steps. If you create a new file, and you run `git status`, you'll see that the file is listed as 'untracked'.

```
Cmd >> echo foo > firstfile
Cmd >> git status
Out >>
On branch main
No commits yet
Untracked files:
(use "git add <file>..." to include in what will be
    ↪committed)
firstfile
nothing added to commit but untracked files present
    ↪(use "git add" to track)
```

Create a file with the usual tools such as an editor. Here we use `touch` as a short cut. The file will initially be untracked.

You need to `git add` on your file to tell git that the file belongs to the repository. (You can add a single file, or use a wildcard to add multiple.) However, this does not actually add the file: it moves it to the *staging area*. The status now says that it is a change to be committed.

```
Cmd >> git add firstfile
Cmd >> git status
Out >>
On branch main
No commits yet
Changes to be committed:
(use "git rm --cached <file>..." to unstage)
new file:   firstfile
```

Add the file to the local repository

Use `git commit` to add these changes to the repository.

```
Cmd >> git commit -m "adding first file"
Out >>
[main (root-commit) f968ac6] adding first file
1 file changed, 1 insertion(+)
create mode 100644 firstfile
Cmd >> git status
Out >>
On branch main
nothing to commit, working tree clean
```

Commit these changes

### 5.4.2 Changes to a file in the repository

Let us investigate the process of adding changes to a file to the repository.

```
Cmd >> echo bar >> firstfile
Cmd >> cat firstfile
Out >>
foo
bar
Cmd >> git status
Out >>
On branch main
Changes not staged for commit:
(use "git add <file>..." to update what will be
    ↪committed)
(use "git restore <file>..." to discard changes in
    ↪working directory)
modified:   firstfile
no changes added to commit (use "git add" and/or "git
    ↪commit -a")
```

Make changes to a file that is tracked. You would typically use an editor, but we use `cat` to append.

If you need to check what changes you have made, `git diff` on that file will tell you the differences the between the edited, but not yet added or committed, file and the previous commit version.

```
Cmd >> git diff firstfile
Out >>
diff --git a/firstfile b/firstfile
index 257cc56..3bd1f0e 100644
--- a/firstfile
+++ b/firstfile
@@ -1 +1,2 @@
foo
+bar
```

See what the changes were wrt the previously commit version.

You now need to repeat `git add` and `git commit` on that file.

```
Cmd >> git add firstfile
Cmd >> git commit -m "changes to first file"
Out >>
[main b1edf77] changes to first file
1 file changed, 1 insertion(+)
Cmd >> git status
Out >>
On branch main
nothing to commit, working tree clean
```

Commit the changes to the local repo.

The changes are now in your local repo; you need to `git push` to update the upstream repo; see section 5.6.

Doing `git log` will give you the history of the repository, listing the commit numbers, and the messages that you entered on those commits.

```
Cmd >> git log
Out >>
commit b1edf778c17b7c7e6cb1a8ac73fa9b61464eba14
Author: Victor Eijkhout <eijkhout@tacc.utexas.edu>
Date:   Sat Jan 29 14:14:40 2022 -0600
changes to first file
commit f968ac6c05dd877db84705a4dcdadbc0bed2c535
Author: Victor Eijkhout <eijkhout@tacc.utexas.edu>
Date:   Sat Jan 29 14:14:39 2022 -0600
adding first file
```

Get the log of all commits so far.

## 5.5 Undoing changes

There are various levels of undo, depending on whether you have added or committed those changes.

### 5.5.1 Undo uncommitted change

Scenario: you have edited a file that is in the repo, you want to undo that change, and you have not yet committed the change.

First of all, do `git diff` to confirm:

```
Cmd >> echo bar >> firstfile
Cmd >> cat firstfile
Out >>
foo
bar
bar
Cmd >> git status
Out >>
On branch main
Changes not staged for commit:
(use "git add <file>..." to update what will be
    ↪committed)
(use "git restore <file>..." to discard changes in
    ↪working directory)
modified:   firstfile
no changes added to commit (use "git add" and/or "git
    ↪commit -a")
Cmd >> git diff firstfile
Out >>
diff --git a/firstfile b/firstfile
index 3bd1f0e..58ba28e 100644
--- a/firstfile
+++ b/firstfile
@@ -1,2 +1,3 @@
foo
bar
+bar
```

Make regrettable changes.

Doing `git checkout` on that file gets the last committed version and puts it back in your working directory.

```
Cmd >> git checkout firstfile
Out >>
Updated 1 path from the index
Cmd >> cat firstfile
Out >>
foo
bar
Cmd >> git status
Out >>
On branch main
nothing to commit, working tree clean
```

Restore previously committed version.

### 5.5.2 Restore a file from a previous commit

A more complicated scenario is where you have committed the change. Then you need to find the commit id.

You can use `git log` to get the ids of all commits. This is useful if you want to roll back to pretty far in the past. However, if you only want to roll back the last commit, use `git show HEAD` to get a description of just that last commit.

```
Cmd >> git log
Out >>
commit e411fad261fd82eb93c328c44978699e946abc0d
Author: Victor Eijkhout <eijkhout@tacc.utexas.edu>
Date:   Sat Jan 29 14:14:42 2022 -0600
changes to first file
commit 63d6ad16beb4e2d12574fb238c29e8ba11fc6732
Author: Victor Eijkhout <eijkhout@tacc.utexas.edu>
Date:   Sat Jan 29 14:14:41 2022 -0600
adding first file
Cmd >> git show HEAD
Out >>
commit e411fad261fd82eb93c328c44978699e946abc0d
Author: Victor Eijkhout <eijkhout@tacc.utexas.edu>
Date:   Sat Jan 29 14:14:42 2022 -0600
changes to first file
diff --git a/firstfile b/firstfile
index 257cc56..3bd1f0e 100644
--- a/firstfile
+++ b/firstfile
@@ -1 +1,2 @@
foo
+bar
```

Find the commit id that you want to roll back.

Now do:

```
git checkout sdf234987238947 -- myfile myotherfile
```

### 5.5.3 Undo a commit

As above, find the commit number.

Then do `git revert sdlksdfkl2343` (with the right id). This will normally open an editor for you to leave comments; you can prevent this with the `--no-edit` option.

```
Cmd >> git revert $commit --no-edit
Out >>
[main 3dca724] Revert "changes to first file"
Date: Sat Jan 29 14:14:42 2022 -0600
1 file changed, 1 deletion(-)
```

Use 'git revert' to roll back.

This will restore the file to its state before the last add and commit, and it will in generally leave the repository back in the state it was before that commit.

```
Cmd >> cat firstfile
Out >>
foo
Cmd >> git status
Out >>
On branch main
nothing to commit, working tree clean
```

See that we have indeed undone the commit.

However, the log will show that you have reverted a certain commit.

```
Cmd >> git log
Out >>
commit 3dca724a1902e8a5e3dba007c325542c6753a424
Author: Victor Eijkhout <eijkhout@tacc.utexas.edu>
Date:   Sat Jan 29 14:14:42 2022 -0600
Revert "changes to first file"

This reverts commit
    ↪e411fad261fd82eb93c328c44978699e946abc0d.
commit e411fad261fd82eb93c328c44978699e946abc0d
Author: Victor Eijkhout <eijkhout@tacc.utexas.edu>
Date:   Sat Jan 29 14:14:42 2022 -0600
changes to first file
commit 63d6ad16beb4e2d12574fb238c29e8ba11fc6732
Author: Victor Eijkhout <eijkhout@tacc.utexas.edu>
Date:   Sat Jan 29 14:14:41 2022 -0600
adding first file
```

But there will be an entry in the log.

The `git reset` command can also be used for various types of undo.

## 5.6 Remote repositories and collaboration

The repository where you have been adding files and changes with `git commit` is the *local repository*. This is great for tracking changes, and reverting them when needed, but a major reason for using source code control is collaboration with others. For this you need a *remote repository*. This involves a set of commands

```
git remote [ other keywords ] [ arguments ]
```

We have some changes, added to the local repository with `git add` and `git commit`

```
Cmd >> git add newfile && git commit -m "adding first
    ↪file"
Out >>
[main 8ce1de4] adding first file
1 file changed, 1 insertion(+)
create mode 100644 newfile
```

Committed changes.

If the repository was created with `git init`, we need to connect it to some remote repository with

```
git remote add servername url
```

Often, the remote name is *origin* by convention, so you often see:

```
git remote add origin url
```

If you want to see what your remote is, do

```
git remote -v
```

```
Cmd >> git remote add mainserver
    ↪git@github.com:TACC/tinker.git
Cmd >> git remote -v
Out >>
mainserver      git@github.com:TACC/tinker.git (fetch)
mainserver      git@github.com:TACC/tinker.git (push)
```

Connect local repo to a remote one.

Finally, you can `git push` committed changes to this remote. Git doesn't just push everything here: since you can have multiple branches locally, and multiple *upstream*s remotely, you intially specify both:

```
git push -u servername branchname
```

```
Cmd >> git push -u mainserver main
Out >>
To github.com:TACC/tinker.git
8333bc1..8ce1de4  main -> main
Branch 'main' set up to track remote branch 'main'
    ↪from 'mainserver'.
```

Push changes.

### 5.6.1 Changing the transport

You may have made your clone with

```
git clone https://....
```

but when you `git push` for the first time you get some permission-related errors.

Do

```
git remote -v
# output: origin  https://username@bitbucket.org/username/reponame.git
git remote set-url origin git@bitbucket.org:username/reponame.git
```

### 5.6.2 Collaboration on the same repository

Let's see how changes from one clone of a repository can propagate to another clone. This can be because more than one person is working on a project, or because one person is working from more than one machine.

We make one local repository in directory *person1*.

```
Cmd >> git clone git@github.com:TACC/tinker.git person1
Out >>
Cloning into 'person1'...
```

Person 1 makes a clone.

Create another clone in *person2*. Normally the cloned repositories would be two user accounts, or the accounts of one user on two machines.

```
Cmd >> git clone git@github.com:TACC/tinker.git person2
Out >>
Cloning into 'person2'...
```

Person 2 makes a clone.

Now the first user creates a file, adds, commits, and pushes it. (This of course requires an upstream to be set, but since we did a `git clone`, this is automatically done.)

```
Cmd >> ( cd person1 && echo 123 >> p1 && git add p1 &&
    ↪git commit -m "add p1" && git push )
Out >>
[main 6f6b126] add p1
1 file changed, 1 insertion(+)
create mode 100644 p1
To github.com:TACC/tinker.git
8863559..6f6b126  main -> main
```

Person 1 adds a file and pushes it.

The second user now does

```
git pull
```

to get these changes. Again, because we create the local repository by `git clone` it is clear where the pull is coming from. The pull message will tell us what new files are created, or how many other files were changes.

```
Cmd >> ( cd person2 && git pull )
Out >>
From github.com:TACC/tinker
8863559..6f6b126  main       -> origin/main
Updating 8863559..6f6b126
Fast-forward
p1 | 1 +
1 file changed, 1 insertion(+)
create mode 100644 p1
```

Person 2 pulls, getting the new file.

### 5.6.3 Merging changes

If you work with someone else, or even if you work solo on a project, but from more than one machine, it may happen that there will be multiple changes on a single file. That is, two local repositories have changes committed, and are now pushing to the same remote.

In the following script we start with the same situation of the previous example, where we have two local repositories, as a stand-in for two users, or two different machines.

We have a file of four lines.

```
Cmd >> cat person1/fourlines
Out >>
1
2
3
4
```

We have a four line file.

The first user makes an edit on the first line; we confirm the state of the file;

```
Cmd >> ( cd person1 && sed -i -e '1s/1/one/' fourlines
    ↪&& cat fourlines )
Out >>
one
2
3
4
```

Person 1 makes a change.

This user pushes the change.

```
Cmd >> ( cd person1 && git add fourlines && git commit
    ↪-m "edit line one" && git push )
Out >>
[main 6767e3f] edit line one
1 file changed, 1 insertion(+), 1 deletion(-)
To github.com:TACC/tinker.git
fdd70b7..6767e3f  main -> main
```

Person 1 pushes the change.

The other user also makes a change, but on line 4, so that there is no conflict;

```
Cmd >> ( cd person2 && sed -i -e '4s/4/four/'
    ↪fourlines && cat fourlines )
Out >>
1
2
3
four
```

Person 2 makes a different change to the same file.

This change is added with `git add` and `git commit`, but we proceed more cautiously in pushing: first we pull any changes made by others with

```
git pull --no-edit
git push
```

```
Cmd >> ( cd person2 && git add fourlines && git commit
    ↪-m "edit line four" && git pull --no-edit && git
    ↪push )
Out >>
[main 27fb2b2] edit line four
1 file changed, 1 insertion(+), 1 deletion(-)
From github.com:TACC/tinker
fdd70b7..6767e3f  main       -> origin/main
Auto-merging fourlines
Merge made by the 'recursive' strategy.
fourlines | 2 +-
1 file changed, 1 insertion(+), 1 deletion(-)
To github.com:TACC/tinker.git
6767e3f..62bd424  main -> main
```

This change does not conflict, we can pull/push.

Now if the first user does a pull, they see all the merged changes.

```
Cmd >> ( cd person1 && git pull && cat fourlines )
Out >>
From github.com:TACC/tinker
6767e3f..62bd424  main       -> origin/main
Updating 6767e3f..62bd424
Fast-forward
fourlines | 2 +-
1 file changed, 1 insertion(+), 1 deletion(-)
one
2
3
four
```

Person 1 pulls to get all the changes.

### 5.6.4 Conflicting changes

There can be various reasons for git to report a conflict.

- You are trying to pull changes from another developer, but you have changes yourself that you haven't yet committed. If you don't want to commit your changes (maybe you're still busy editing) you can use `git stash` to set your edits aside. You can later retrieve them with `git stash pop`, or decide to forget all about them with `git stash drop`.
- There is a conflict between your changes, and those you are trying to pull from another developer, or from yourself on another machine. This is the case we will look at here.
- Very similar, there can also be conflicts between two branches you try to merge. We will look at that in section 5.7.1.

As before, we have directories *person1* and *person2* containing independent clones of a repository. We have a file of four lines long, but contrary to above, we now make edits that are too close together for git's auto-merge to deal with them.

```
Cmd >> cat person1/fourlines
Out >>
1
2
3
4
```

The original file.

Now developer 1 makes a change on line 1.

```
Cmd >> ( cd person1 && sed -i -e '1s/1/one/' fourlines
    ↪&& git add fourlines && git commit -m "edit line
    ↪one" && git push )
Out >>
[main a10216d] edit line one
1 file changed, 1 insertion(+), 1 deletion(-)
To github.com:TACC/tinker.git
4955e50..a10216d  main -> main
```

With apologies for some scripting trickery, we use an edit by *sed*, changing 1 on line 1 to *one*. We add, commit, and push this change.

In the meantime, developer 2 makes another change, to the original file. This change can be added and committed to the local repository without any problem.

```
Cmd >> ( cd person2 && sed -i -e '2s/2/two/' fourlines
    ↪&& cat fourlines  && git add fourlines && git
    ↪commit -m "edit line two" )
Out >>
1
two
3
4
[main c9b6ded] edit line two
1 file changed, 1 insertion(+), 1 deletion(-)
```

Change the 2 on line two to *two*. We add and commit this to the local repository.

However, if we try to `git push` this change to the remote repository, we get an error that the remote is ahead of the local repository. So we first pull the state of the remote repository. In the previous section this led to an automatic merge; not so here.

```
Cmd >> ( cd person2 && git pull --no-edit || echo )
Out >>
From github.com:TACC/tinker
4955e50..a10216d  main       -> origin/main
Auto-merging fourlines
CONFLICT (content): Merge conflict in fourlines
Automatic merge failed; fix conflicts and then commit
    ↪the result.
```

The `git pull` call results in a message that the automatic mere failed, indicating what file was the problem.

You can now edit the file by hand, or using some merge tool.

```
Cmd >> ( cd person2 && cat fourlines )
Out >>
<<<<<<< HEAD
1
two
=======
one
2
>>>>>>> a10216da358649df80aaaeb94f1ceef909c2ed83
3
4
```

In between the chevron'ed lines you first get the `HEAD`, that is the local state, followed by the pulled remote state. Edit this file, commit the merge and push.

### 5.6.5 Pull requests

The previous sections described collaboration on a repository that you have write permission for. For large projects you may not have this: the owner of the project may want a safer mechanism where you submit changes for approval in the form of a *pull request*, and they will do the merge of your changes into the repository.

This involves the following steps:

1. You make a *fork* of the repository;
2. you clone this fork;
3. make a branch with your changes;

## 5.7 Branching

With a *branch* you can keep a completely separate version of all the files in your project.

Initially we have a file on the `main` branch.

```
Cmd >> cat firstfile
Out >>
foo
Cmd >> git status
Out >>
On branch main
nothing to commit, working tree clean
```

We have a file, committed and all.

We create a new branch, named `dev` and check it out

```
git branch dev
git checkout dev
```

This initially has the same content.

```
Cmd >> git branch dev && git branch -a
Out >>
dev
* main
Cmd >> git checkout dev && git branch -a
Out >>
Switched to branch 'dev'
* dev
main
```

Make a development branch.

We make changes, and commit them to the current branch.

```
Cmd >> cat firstfile
Out >>
foo
Cmd >> echo bar > firstfile && cat firstfile
Out >>
bar
Cmd >> git status
Out >>
On branch dev
Changes not staged for commit:
(use "git add <file>..." to update what will be
    ↪committed)
(use "git restore <file>..." to discard changes in
    ↪working directory)
modified:   firstfile
no changes added to commit (use "git add" and/or "git
    ↪commit -a")
Cmd >> git add firstfile && git commit -m "dev changes"
Out >>
[dev b07cd2e] dev changes
1 file changed, 1 insertion(+), 1 deletion(-)
```

Make changes and commit them to the dev branch.

If we switch back to the *main* branch, everything is as before when we made the *dev* branch.

```
Cmd >> git checkout main && cat firstfile && git status
Out >>
Switched to branch 'main'
foo
On branch main
nothing to commit, working tree clean
```

The other branch is still unchanged.

The first time you try to push a new branch you need to establish it upstream:

```
git push --set-upstream origin mynewbranch
```

We can inspect differences between branches with

```
  git diff branch1 branch2
```

```
Cmd >> git diff main dev
Out >>
diff --git a/firstfile b/firstfile
index 257cc56..5716ca5 100644
--- a/firstfile
+++ b/firstfile
@@ -1 +1 @@
-foo
+bar
```

We can check differences between branches.

### 5.7.1 Branch merging

One of the points of having branches is to merge them after you have done some development in one branch.

We start with the four line file from before.

```
Cmd >> cat fourlines
Out >>
1
2
3
4
Cmd >> git status
Out >>
On branch main
nothing to commit, working tree clean
```

The main branch is up to date.

Also as before, we have a *dev* branch that contains these contents.

We switch back to the *main* branch and make a change. This will not be visible in the *dev* branch.

```
Cmd >> git checkout main
Out >>
Switched to branch 'main'
Cmd >> sed -i -e '1s/1/one/' fourlines && cat fourlines
Out >>
one
2
3
4
Cmd >> git add fourlines && git commit -m "edit line 1"
Out >>
[main c51d4ff] edit line 1
1 file changed, 1 insertion(+), 1 deletion(-)
```

On line 1, change `1` to *one*.

We switch to the *dev* branch and make another file. The change in the main branch is indeed not here.

```
Cmd >> git checkout dev
Out >>
Switched to branch 'dev'
Cmd >> sed -i -e '4s/4/four/' fourlines && cat
    ↪fourlines
Out >>
1
2
3
four
Cmd >> git add fourlines && git commit -m "edit line 4"
Out >>
[dev dbb0c03] edit line 4
1 file changed, 1 insertion(+), 1 deletion(-)
```

On line 4, change `4` to *four*. This change is far enough away from the other change, that there should be no conflict.

Switching back to the *main* branch, we use

```
git merge dev
```

to merge the *dev* changes into *main*.

```
Cmd >> git checkout main
Out >>
Switched to branch 'main'
Cmd >> git merge dev
Out >>
Auto-merging fourlines
Merge made by the 'recursive' strategy.
fourlines | 2 +-
1 file changed, 1 insertion(+), 1 deletion(-)
Cmd >> cat fourlines
Out >>
one
2
3
four
```

Merge the dev branch into the main one with `git merge`. Note the 'auto-merge' message, and confirm that both changes to the file are there.

If two developers make changes on the same line, or on adjacent lines, git will not be able to merge and you have to edit the file as in section 5.6.4.

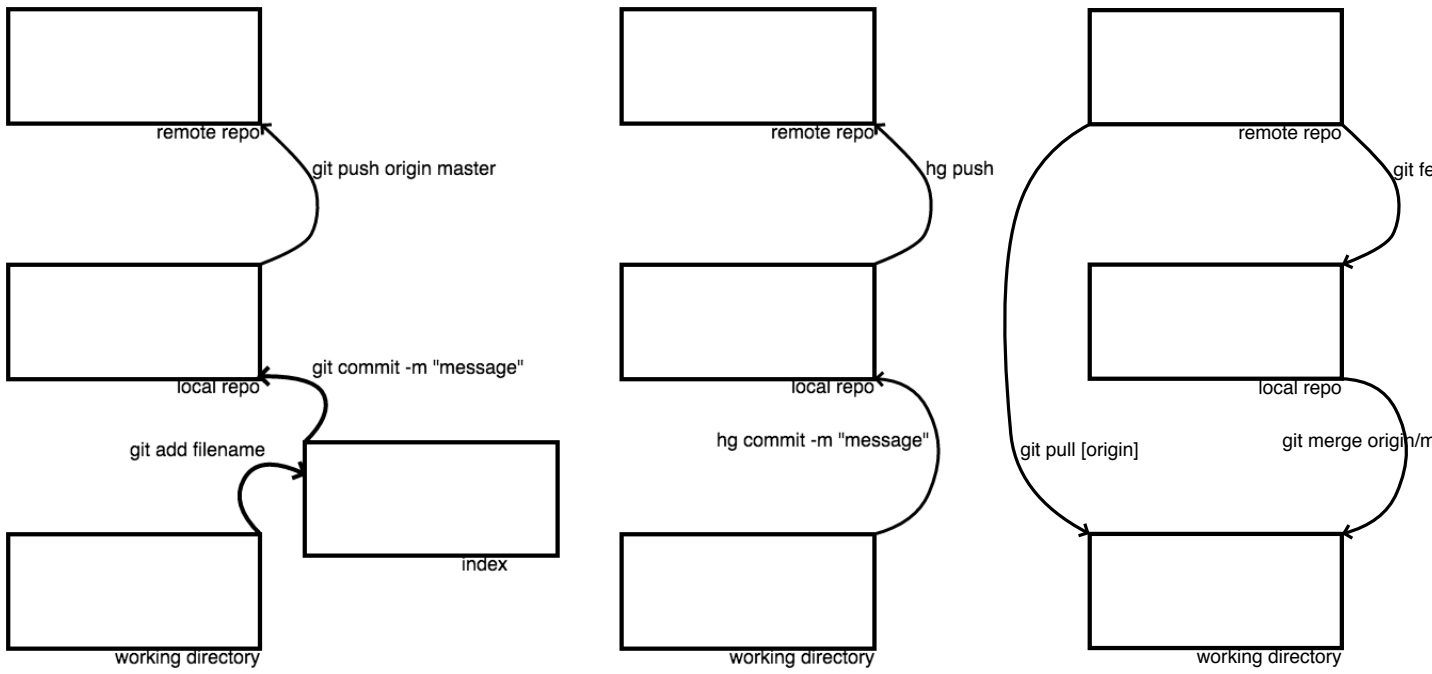



Figure 5.1: Add local changes to the remote repository (left); Get changes that were made to the remote repository (right).

## 5.8 Releases

At certain point in your development process you may want to mark the current state of the repository as 'finished'. You can do this by

1. Attaching *tag* to the state of the repository, or
2. Creating an *archive*: a released version that has the repo information stripped.

### 5.8.1 Tags

A tag is a marked state of the repository. There are two types of tags:

1. light-weight tags are no more than a synonym for a commit:

```
git tag v0.09
```

2. annotated tags, which carry an information message:

```
git tag -a v0.1 "base classes finished"
```

You list all tags with `git tag`, you get information on a tag with `git show v0.1`, and you push a tag to a remote server with

```
git push origin v0.1
```

You can retrieve the tagged state of the repository with

```
git checkout v0.1
```

but beware that changes you now make can not be pushed to anything: this is a 'detached HEAD' state. If you want to fix bugs in a tagged state, you can create a branch based on the tag:

```
git checkout -b version0.2 v0.1
```

### 5.8.2 Archives, releases

If you want to make a released version of your software, that can be downloaded and does not rely on the git software, use the *archive* command:

```
git archive master --format=tgz --prefix=MyProject-v1 -o MyProject-v1.tgz
```

# Chapter 6

# Dense linear algebra: BLAS, LAPACK, SCALAPACK

In this section we will discuss libraries for dense linear algebra operations.

Dense linear algebra, that is linear algebra on matrices that are stored as two-dimensional arrays (as opposed to sparse linear algebra; see HPC book, section 5.4, as well as the tutorial on PETSc *Parallel Programming book, part III*) has been standardized for a considerable time. The basic operations are defined by the three levels of *Basic Linear Algebra Subprograms (BLAS)*:

- Level 1 defines vector operations that are characterized by a single loop [13].
- Level 2 defines matrix vector operations, both explicit such as the matrix-vector product, and implicit such as the solution of triangular systems [7].
- Level 3 defines matrix-matrix operations, most notably the matrix-matrix product [6].

The name 'BLAS' suggests a certain amount of generality, but the original authors were clear [13] that these subprograms only covered dense linear algebra. Attempts to standardize sparse operations have never met with equal success.

Based on these building blocks, libraries have been built that tackle the more sophisticated problems such as solving linear systems, or computing eigenvalues or singular values. *Linpack*[1] and *Eispack* were the first to formalize these operations involved, using Blas Level 1 and Blas Level 2 respectively. A later development, *Lapack* uses the blocked operations of Blas Level 3. As you saw in section HPC book, section 1.6.1, this is needed to get high performance on cache-based CPUs.

**Remark 15** *The reference implementation* `https://netlib.org/blas/index.html` *of the BLAS [3] will not give good performance with any compiler; most platforms have vendor-optimized implementations, such as the* MKL *library from Intel.*

With the advent of parallel computers, several projects arose that extended the Lapack functionality to distributed computing, most notably *Scalapack* [4, 2], *PLapack* [23, 22], and most recently Elemental [19]. These packages are harder to use than Lapack because of the need for a two-dimensional cyclic distribution; sections HPC book, section 7.2.3 and HPC book, section 7.3.2. We will not go into the details here.

1. The linear system solver from this package later became the *Linpack benchmark*; see section HPC book, section 2.11.5.

## 6.1 Some general remarks

### 6.1.1 The Fortran heritage

The original BLAS routines were written in Fortran, and the reference implementation is still in Fortran. For this reason you will see the routine definitions first in Fortran in this tutorial It is possible to use the Fortran routines from a C/C++ program:

- You typically need to append an underscore to the Fortran name;
- You need to include a prototype file in your source, for instance `mkl.h`;
- Every argument needs to be a 'star'-argument, so you can not pass literal constants: you need to pass the address of a variable.
- You need to create a column-major matrix.

There are also C/C++ interfaces:

- The C routine names are formed by prefixing the original name with `cblas_`; for instance `dasum` becomes `cblas_dasum`.
- Fortran character arguments have been replaced by enumerated constants, for instance `CblasNoTrans` instead of the `'N'` parameter.
- The Cblas interface can accommodate both row-major and column-major storage.
- Array indices are 1-based, rather than 0-based; this mostly becomes apparent in error messages and when specifying pivot indices.

### 6.1.2 Routine naming

Routines conform to a general naming scheme: `XYYZZZ` where

**X** precision: `S,D,C,Z` stand for single and double, single complex and double complex, respectively.
**YY** storage scheme: general rectangular, triangular, banded.
**ZZZ** operation. See the manual for a list.

### 6.1.3 Data formats

Lapack and Blas use a number of data formats, including

**GE** General matrix: stored two-dimensionally as `A(LDA,*)`
**SY/HE** Symmetric/Hermitian: general storage; `UPLO` parameter to indicate upper or lower (e.g. `SPOTRF`)
**GB/SB/HB** General/symmetric/Hermitian band; these formats use column-major storage; in `SGBTRF` overallocation needed because of pivoting
**PB** Symmetric of Hermitian positive definite band; no overallocation in `SPDTRF`

### 6.1.4 Lapack operations

For Lapack, we can further divide the routines into an organization with three levels:

- Drivers. These are powerful top level routine for problems such as solving linear systems or computing an SVD. There are simple and expert drivers; the expert ones have more numerical sophistication.

- Computational routines. These are the routines that drivers are built up out of. A user may have occasion to call them by themselves.
- Auxiliary routines.

Expert driver names end on 'X'.

- Linear system solving. Simple drivers: `-SV` (e.g., `DGESV`) Solve $AX = B$, overwrite A with LU (with pivoting), overwrite B with X.
  Expert driver: `-SVX` Also transpose solve, condition estimation, refinement, equilibration
- Least squares problems. Drivers:
  `xGELS` using QR or LQ under full-rank assumption
  `xGELSY` "complete orthogonal factorization"
  `xGELSS` using SVD
  `xGELSD` using divide-conquer SVD (faster, but more workspace than `xGELSS`)
  Also: LSE & GLM linear equality constraint & general linear model
- Eigenvalue routines. Symmetric/Hermitian: `xSY` or `xHE` (also `SP`, `SB`, `ST`)
  simple driver `-EV`
  expert driver `-EVX`
  divide and conquer `-EVD`
  relative robust representation `-EVR`
  General (only `xGE`)
  Schur decomposition `-ES` and `-ESX`
  eigenvalues `-EV` and `-EVX`
  SVD (only `xGE`)
  simple driver `-SVD`
  divide and conquer `SDD`
  Generalized symmetric (`SY` and `HE`; `SP`, `SB`)
  simple driver `GV`
  expert `GVX`
  divide-conquer `GVD`
  Nonsymmetric:
  Schur: simple `GGES`, expert `GGESX`
  eigen: simple `GGEV`, expert `GGEVX`
  svd: `GGSVD`

## 6.2 BLAS matrix storage

There are a few points to bear in mind about the way matrices are stored in the BLAS and LAPACK[2]:

### 6.2.1 Array indexing

Since these libraries originated in a Fortran environment, they use 1-based indexing. Users of languages such as C/C++ are only affected by this when routines use index arrays, such as the location of pivots in LU factorizations.

2. We are not going into band storage here.

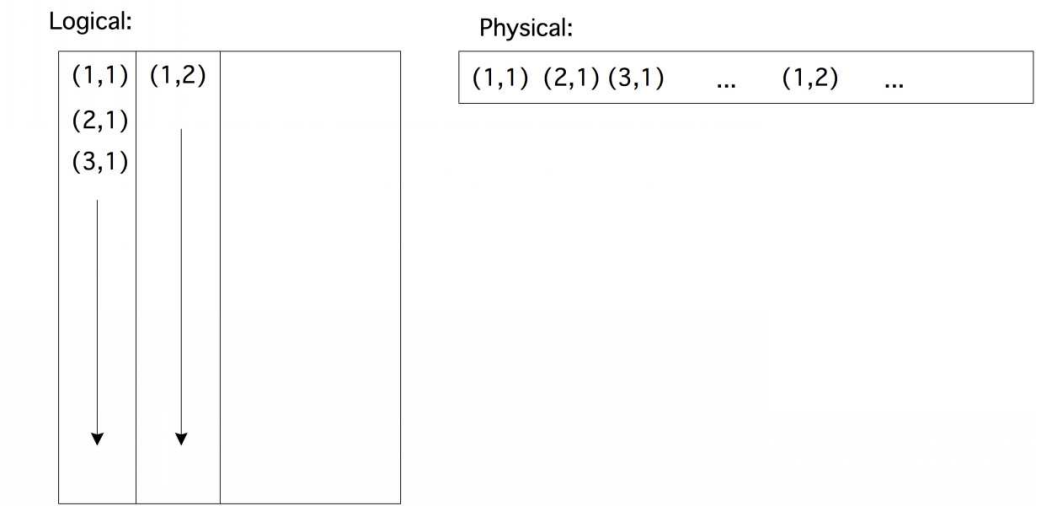



Figure 6.1: Column-major storage of an array in Fortran.

### 6.2.2 Fortran column-major ordering

Since computer memory is one-dimensional, some conversion is needed from two-dimensional matrix coordinates to memory locations. The *Fortran* language uses *column-major* storage, that is, elements in a column are stored consecutively; see figure 6.1. This is also described informally as 'the leftmost index varies quickest'.

Arrays in C, on the other hand, are laid out in *row-major* order.

### 6.2.3 Submatrices and the `LDA` parameter

Using the storage scheme described above, it is clear how to store an $m \times n$ matrix in $mn$ memory locations. However, there are many cases where software needs access to a matrix that is a subblock of another, larger, matrix. As you see in figure 6.2 such a subblock is no longer contiguous in memory. The way to describe this is by introducing a third parameter in addition to `M,N`: we let `LDA` be the 'leading dimension of `A`', that is, the allocated first dimension of the surrounding array. This is illustrated in figure 6.3. To pass the subblock to a routine, you would specify it as

```
call routine( A(3,2), /* M= */ 2, /* N= */ 3, /* LDA= */ Mbig, ... )
```

## 6.3 Performance issues

The collection of BLAS and LAPACK routines are a *de facto* standard: the API is fixed, but the implementation is not. You can find reference implementations on the *netlib* website (`netlib.org`), but these will be very low in performance.

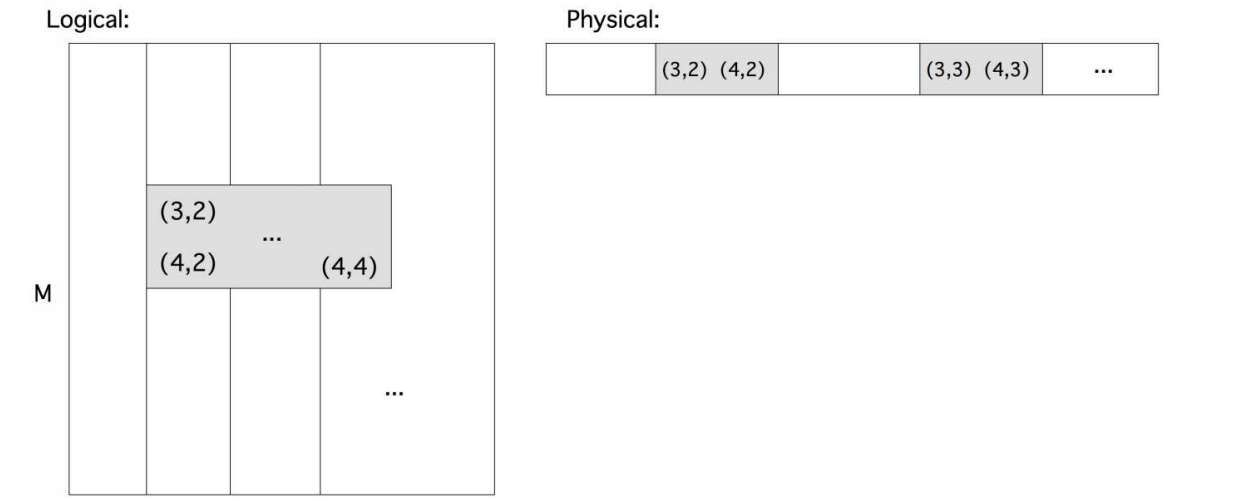



Figure 6.2: A subblock out of a larger matrix.

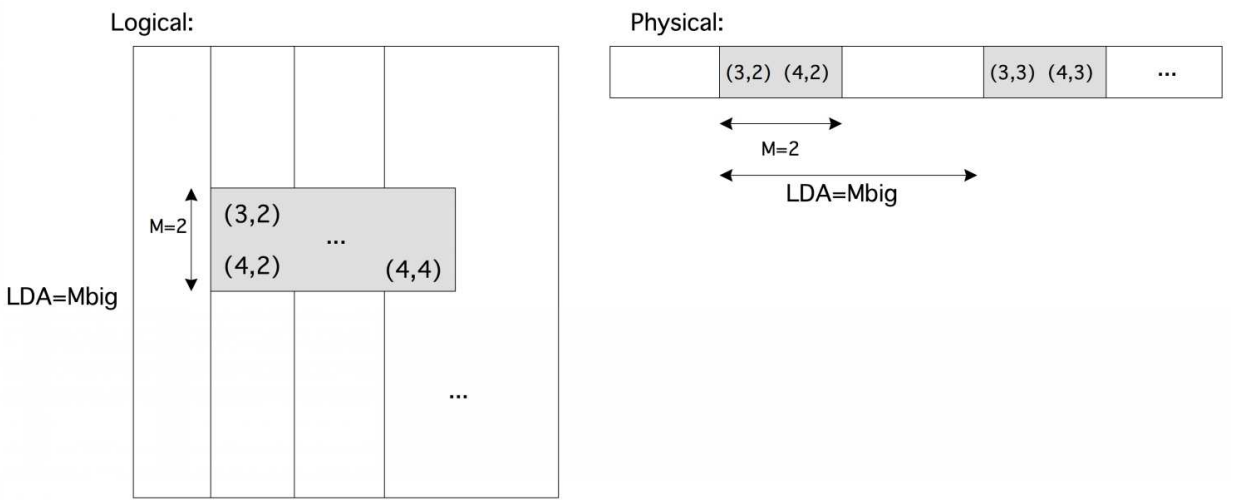



Figure 6.3: A subblock out of a larger matrix, using `LDA`.

On the other hand, many LAPACK routines can be based on the matrix-matrix product (BLAS routine `gemm`), which you saw in section HPC book, section 7.4.1 has the potential for a substantial fraction of peak performance. To achieve this, you should use an optimized version, such as

- *MKL*, the Intel math-kernel library;
- OpenBlas (`http://www.openblas.net/`), an open source version of the original *Goto BLAS*; or
- *blis* (`https://code.google.com/p/blis/`), a BLAS replacement and extension project.

## 6.4 Some simple examples

Let's look at some simple examples.

The routine `xscal` scales a vector in place.

```
! Fortran
subroutine dscal(integer N, double precision DA,
    double precision, dimension(*) DX, integer INCX )
// C
void cblas_dscal (const MKL_INT n, const double a,
    double *x, const MKL_INT incx);
```

A simple example:

```
// example1.F90
  do i=1,n
     xarray(i) = 1.d0
  end do
  call dscal(n,scale,xarray,1)
  do i=1,n
     if (.not.assert_equal( xarray(i),scale )) print *,"Error in index",i
  end do
```

The same in C:

```
// example1c.cxx
  xarray = new double[n]; yarray = new double[n];

  for (int i=0; i<n; i++)
    xarray[i] = 1.;
  cblas_dscal(n,scale,xarray,1);
  for (int i=0; i<n; i++)
    if (!assert_equal( xarray[i],scale ))
      printf("Error in index %d",i);
```

Many routines have an increment parameter. For `xscale` that's the final parameter:

```
// example2.F90
  integer :: inc=2
  call dscal(n/inc,scale,xarray,inc)
  do i=1,n
     if (mod(i,inc)==1) then
        if (.not.assert_equal( xarray(i),scale )) print *,"Error in index",i
     else
        if (.not.assert_equal( xarray(i),1.d0 )) print *,"Error in index",i
     end if
  end do
```

The matrix-vector product `xgemv` computes $y \leftarrow \alpha Ax + \beta y$, rather than $y \leftarrow Ax$. The specification of the matrix takes the `M,N` size parameters, and a character argument `'N'` to indicate that the matrix is not transposed. Both of the vectors have an increment argument.

```
subroutine dgemv(character TRANS,
```

```
  integer M,integer N,
  double precision ALPHA,
  double precision, dimension(lda,*) A,integer LDA,
  double precision, dimension(*) X,integer INCX,
  double precision BETA,double precision, dimension(*) Y,integer INCY
  )
```

An example of the use of this routine:

```
// example3.F90
  do j=1,n
     xarray(j) = 1.d0
     do i=1,m
        matrix(i,j) = 1.d0
     end do
  end do

  alpha = 1.d0; beta = 0.d0
  call dgemv( 'N',M,N, alpha,matrix,M, xarray,1, beta,yarray,1)
  do i=1,m
     if (.not.assert_equal( yarray(i),dble(n) )) &
          print *,"Error in index",i,":",yarray(i)
  end do
```

The same example in C has an extra parameter to indicate whether the matrix is stored in row or column major storage:

```
// example3c.cxx
  for (int j=0; j<n; j++) {
    xarray[j] = 1.;
    for (int i=0; i<m; i++)
      matrix[ i+j*m ] = 1.;
  }

  alpha = 1.; beta = 0.;
  cblas_dgemv(CblasColMajor,
              CblasNoTrans,m,n, alpha,matrix,m, xarray,1, beta,yarray,1);

  for (int i=0; i<m; i++)
    if (!assert_equal( yarray[i],(double)n ))
      printf("Error in index %d",i);
```

# Chapter 7

# Scientific Data Storage with HDF5

There are many ways of storing data, in particular data that comes in arrays. A surprising number of people stores data in spreadsheets, then exports them to ascii files with comma or tab delimiters, and expects other people (or other programs written by themselves) to read that in again. Such a process is wasteful in several respects:

- The ascii representation of a number takes up much more space than the internal binary representation. Ideally, you would want a file to be as compact as the representation in memory.
- Conversion to and from ascii is slow; it may also lead to loss of precision.

For such reasons, it is desirable to have a file format that is based on binary storage. There are a few more requirements on a useful file format:

- Since binary storage can differ between platforms, a good file format is platform-independent. This will, for instance, prevent the confusion between *big-endian* and *little-endian* storage, as well as conventions of 32 versus 64 bit floating point numbers.
- Application data can be heterogeneous, comprising integer, character, and floating point data. Ideally, all this data should be stored together.
- Application data is also structured. This structure should be reflected in the stored form.
- It is desirable for a file format to be *self-documenting*. If you store a matrix and a right-hand side vector in a file, wouldn't it be nice if the file itself told you which of the stored numbers are the matrix, which the vector, and what the sizes of the objects are?

This tutorial will introduce the HDF5 library, which fulfills these requirements. HDF5 is a large and complicated library, so this tutorial will only touch on the basics. For further information, consult http://www.hdfgroup.org/HDF5/. While you do this tutorial, keep your browser open on http://www.hdfgroup.org/HDF5/doc/ or http://www.hdfgroup.org/HDF5/RM/RM_H5Front.html for the exact syntax of the routines.

## 7.1 Setup

As described above, HDF5 is a file format that is machine-independent and self-documenting. Each HDF5 file is set up like a directory tree, with subdirectories, and leaf nodes which contain the actual data. This means that data can be found in a file by referring to its name, rather than its location in the file. In this

section you will learn to write programs that write to and read from HDF5 files. In order to check that the files are as you intend, you can use the `h5dump` utility on the command line.

Just a word about compatibility. The HDF5 format is not compatible with the older version HDF4, which is no longer under development. You can still come across people using hdf4 for historic reasons. This tutorial is based on HDF5 version 1.6. Some interfaces changed in the current version 1.8; in order to use 1.6 APIs with 1.8 software, add a flag `-DH5_USE_16_API` to your compile line.

### 7.1.1 Compilation

Include file for C:

```
#include <netcdf.h>
```

CMake for C:

```
find_package( PkgConfig REQUIRED )
pkg_check_modules( NETCDF REQUIRED netcdf )

target_include_directories(
        ${PROJECTNAME} PUBLIC
        ${NETCDF_INCLUDE_DIRS} )
target_link_libraries(
        ${PROJECTNAME} PUBLIC
        ${NETCDF_LIBRARIES} )
target_link_directories(
        ${PROJECTNAME} PUBLIC
        ${NETCDF_LIBRARY_DIRS} )
target_link_libraries(
        ${PROJECTNAME} PUBLIC netcdf )
```

Include for Fortran:

```
use netcdf
```

CMake for Fortran:

```
find_package( PkgConfig REQUIRED )
pkg_check_modules( NETCDFF REQUIRED netcdf-fortran )
pkg_check_modules( NETCDF REQUIRED netcdf )

target_include_directories(
        ${PROJECTNAME} PUBLIC
        ${NETCDFF_INCLUDE_DIRS}
        )
target_link_libraries(
        ${PROJECTNAME} PUBLIC
        ${NETCDFF_LIBRARIES} ${NETCDF_LIBRARIES}
        )
target_link_directories(
        ${PROJECTNAME} PUBLIC
        ${NETCDFF_LIBRARY_DIRS} ${NETCDF_LIBRARY_DIRS}
        )
```

```
target_link_libraries(
        ${PROJECTNAME} PUBLIC netcdf )
```

### 7.1.2 General design

Many HDF5 routines are about creating objects: file handles, members in a dataset, et cetera. The general syntax for that is

```
hid_t h_id;
h_id = H5Xsomething(...);
```

Failure to create the object is indicated by a negative return parameter, so it would be a good idea to create a file `myh5defs.h` containing:

```
#include "hdf5.h"
#define H5REPORT(e) \
  {if (e<0) {printf("\nHDF5 error on line %d\n\n",__LINE__); \
   return e;}}
```

and use this as:

```
#include "myh5defs.h"

hid_t h_id;
h_id = H5Xsomething(...); H5REPORT(h_id);
```

## 7.2 Creating a file

First of all, we need to create an HDF5 file.

```
hid_t file_id;
herr_t status;

file_id = H5Fcreate( filename, ... );
    ...
status = H5Fclose(file_id);
```

This file will be the container for a number of data items, organized like a directory tree.

**Exercise.** Create an HDF5 file by compiling and running the `create.c` example below.

*Expected outcome.* A file `file.h5` should be created.

*Caveats.* Be sure to add HDF5 include and library directories:
`cc -c create.c -I. -I/opt/local/include`
and
`cc -o create create.o -L/opt/local/lib -lhdf5`. The include and lib directories will be system dependent.

```
#include "myh5defs.h"
#define FILE "file.h5"

main() {

   hid_t       file_id;   /* file identifier */
   herr_t      status;

   /* Create a new file using default properties. */
   file_id = H5Fcreate(FILE, H5F_ACC_TRUNC, H5P_DEFAULT, H5P_DEFAULT);
   H5REPORT(file_id);

   /* Terminate access to the file. */
   status = H5Fclose(file_id);
}
```

You can display hdf5 files on the commandline:

```
%% h5dump file.h5
HDF5 "file.h5" {
GROUP "/" {
}
}
```

Note that an empty file corresponds to just the root of the directory tree that will hold the data.

## 7.3 Datasets

Next we create a dataset, in this example a 2D grid. To describe this, we first need to construct a dataspace:

```
dims[0] = 4; dims[1] = 6;
dataspace_id = H5Screate_simple(2, dims, NULL);
dataset_id = H5Dcreate(file_id, "/dset", dataspace_id, .... );
....
status = H5Dclose(dataset_id);
status = H5Sclose(dataspace_id);
```

Note that datasets and dataspaces need to be closed, just like files.

**Exercise.** Create a dataset by compiling and running the `dataset.c` code below

*Expected outcome.* This creates a file `dset.h` that can be displayed with `h5dump`.

```
#include "myh5defs.h"
#define FILE "dset.h5"

main() {

   hid_t       file_id, dataset_id, dataspace_id;  /* identifiers */
   hsize_t     dims[2];
   herr_t      status;

   /* Create a new file using default properties. */
   file_id = H5Fcreate(FILE, H5F_ACC_TRUNC, H5P_DEFAULT, H5P_DEFAULT);

   /* Create the data space for the dataset. */
   dims[0] = 4;
   dims[1] = 6;
   dataspace_id = H5Screate_simple(2, dims, NULL);

   /* Create the dataset. */
   dataset_id = H5Dcreate(file_id, "/dset", H5T_NATIVE_INT,
                          dataspace_id, H5P_DEFAULT);
   /*H5T_STD_I32BE*/

   /* End access to the dataset and release resources used by it. */
   status = H5Dclose(dataset_id);

   /* Terminate access to the data space. */
   status = H5Sclose(dataspace_id);

   /* Close the file. */
   status = H5Fclose(file_id);
}
```

We again view the created file online:

```
%% h5dump dset.h5
HDF5 "dset.h5" {
GROUP "/" {
   DATASET "dset" {
      DATATYPE  H5T_STD_I32BE
      DATASPACE  SIMPLE { ( 4, 6 ) / ( 4, 6 ) }
      DATA {
      (0,0): 0, 0, 0, 0, 0, 0,
      (1,0): 0, 0, 0, 0, 0, 0,
      (2,0): 0, 0, 0, 0, 0, 0,
      (3,0): 0, 0, 0, 0, 0, 0
      }
   }
}
```

```
}
```

The datafile contains such information as the size of the arrays you store. Still, you may want to add related scalar information. For instance, if the array is output of a program, you could record with what input parameter was it generated.

```
parmspace = H5Screate(H5S_SCALAR);
parm_id = H5Dcreate
  (file_id,"/parm",H5T_NATIVE_INT,parmspace,H5P_DEFAULT);
```

**Exercise.** Add a scalar dataspace to the HDF5 file, by compiling and running the `parmwrite.c` code below.

*Expected outcome.* A new file `wdset.h5` is created.

```
#define FILE "pdset.h5"

main() {

   hid_t       file_id, dataset_id, dataspace_id;  /* identifiers */
   hid_t       parm_id,parmspace;
   hsize_t     dims[2];
   herr_t      status;

   /* Create a new file using default properties. */
   file_id = H5Fcreate(FILE, H5F_ACC_TRUNC, H5P_DEFAULT, H5P_DEFAULT);

   /* Create the data space for the dataset. */
   dims[0] = 4;
   dims[1] = 6;
   dataspace_id = H5Screate_simple(2, dims, NULL);

   /* Create the dataset. */
   dataset_id = H5Dcreate
     (file_id, "/dset", H5T_STD_I32BE, dataspace_id, H5P_DEFAULT);

   /* Add a descriptive parameter */
   parmspace = H5Screate(H5S_SCALAR);
   parm_id = H5Dcreate
     (file_id,"/parm",H5T_NATIVE_INT,parmspace,H5P_DEFAULT);

   /* End access to the dataset and release resources used by it. */
   status = H5Dclose(dataset_id);
   status = H5Dclose(parm_id);

   /* Terminate access to the data space. */
   status = H5Sclose(dataspace_id);
   status = H5Sclose(parmspace);

   /* Close the file. */
   status = H5Fclose(file_id);
}
```

```
%% h5dump wdset.h5
HDF5 "wdset.h5" {
GROUP "/" {
   DATASET "dset" {
      DATATYPE  H5T_IEEE_F64LE
      DATASPACE  SIMPLE { ( 4, 6 ) / ( 4, 6 ) }
      DATA {
      (0,0): 0.5, 1.5, 2.5, 3.5, 4.5, 5.5,
      (1,0): 6.5, 7.5, 8.5, 9.5, 10.5, 11.5,
      (2,0): 12.5, 13.5, 14.5, 15.5, 16.5, 17.5,
      (3,0): 18.5, 19.5, 20.5, 21.5, 22.5, 23.5
      }
   }
   DATASET "parm" {
      DATATYPE  H5T_STD_I32LE
      DATASPACE  SCALAR
      DATA {
      (0): 37
      }
   }
}
}
```

## 7.4 Writing the data

The datasets you created allocate the space in the hdf5 file. Now you need to put actual data in it. This is done with the `H5Dwrite` call.

```
/* Write floating point data */
for (i=0; i<24; i++) data[i] = i+.5;
status = H5Dwrite
  (dataset,H5T_NATIVE_DOUBLE,H5S_ALL,H5S_ALL,H5P_DEFAULT,
   data);
/* write parameter value */
parm = 37;
status = H5Dwrite
  (parmset,H5T_NATIVE_INT,H5S_ALL,H5S_ALL,H5P_DEFAULT,
   &parm);
```

```
#include "myh5defs.h"
#define FILE "wdset.h5"
```

```
main() {

   hid_t       file_id, dataset, dataspace;  /* identifiers */
   hid_t       parmset,parmspace;
   hsize_t     dims[2];
   herr_t      status;
   double data[24]; int i,parm;

   /* Create a new file using default properties. */
   file_id = H5Fcreate(FILE, H5F_ACC_TRUNC, H5P_DEFAULT, H5P_DEFAULT);

   /* Create the dataset. */
   dims[0] = 4; dims[1] = 6;
   dataspace = H5Screate_simple(2, dims, NULL);
   dataset = H5Dcreate
     (file_id, "/dset", H5T_NATIVE_DOUBLE, dataspace, H5P_DEFAULT);

   /* Add a descriptive parameter */
   parmspace = H5Screate(H5S_SCALAR);
   parmset = H5Dcreate
     (file_id,"/parm",H5T_NATIVE_INT,parmspace,H5P_DEFAULT);

   /* Write data to file */
   for (i=0; i<24; i++) data[i] = i+.5;
   status = H5Dwrite
     (dataset,H5T_NATIVE_DOUBLE,H5S_ALL,H5S_ALL,H5P_DEFAULT,
      data); H5REPORT(status);

   /* write parameter value */
   parm = 37;
   status = H5Dwrite
     (parmset,H5T_NATIVE_INT,H5S_ALL,H5S_ALL,H5P_DEFAULT,
      &parm); H5REPORT(status);

   /* End access to the dataset and release resources used by it. */
   status = H5Dclose(dataset);
   status = H5Dclose(parmset);

   /* Terminate access to the data space. */
   status = H5Sclose(dataspace);
   status = H5Sclose(parmspace);

   /* Close the file. */
   status = H5Fclose(file_id);
}


%% h5dump wdset.h5
HDF5 "wdset.h5" {
GROUP "/" {
   DATASET "dset" {
      DATATYPE  H5T_IEEE_F64LE
      DATASPACE  SIMPLE { ( 4, 6 ) / ( 4, 6 ) }
```

```
      DATA {
      (0,0): 0.5, 1.5, 2.5, 3.5, 4.5, 5.5,
      (1,0): 6.5, 7.5, 8.5, 9.5, 10.5, 11.5,
      (2,0): 12.5, 13.5, 14.5, 15.5, 16.5, 17.5,
      (3,0): 18.5, 19.5, 20.5, 21.5, 22.5, 23.5
      }
   }
   DATASET "parm" {
      DATATYPE  H5T_STD_I32LE
      DATASPACE  SCALAR
      DATA {
      (0): 37
      }
   }
}
}
```

If you look closely at the source and the dump, you see that the data types are declared as 'native', but rendered as LE. The 'native' declaration makes the datatypes behave like the built-in C or Fortran data types. Alternatively, you can explicitly indicate whether data is *little-endian* or *big-endian*. These terms describe how the bytes of a data item are ordered in memory. Most architectures use little endian, as you can see in the dump output, but, notably, IBM uses big endian.

## 7.5 Reading

Now that we have a file with some data, we can do the mirror part of the story: reading from that file. The essential commands are

```
h5file = H5Fopen( .... )
....
H5Dread( dataset, .... data .... )
```

where the `H5Dread` command has the same arguments as the corresponding `H5Dwrite`.

**Exercise.** Read data from the `wdset.h5` file that you create in the previous exercise, by compiling and running the `allread.c` example below.

> *Expected outcome.* Running the `allread` executable will print the value 37 of the parameter, and the value `8.5` of the `(1,2)` data point of the array.
>
> *Caveats.* Make sure that you run `parmwrite` to create the input file.

```
#include "myh5defs.h"
#define FILE "wdset.h5"

main() {

   hid_t       file_id, dataset, parmset;
```

```
   herr_t      status;
   double data[24]; int parm;

   /* Open an existing file */
   file_id = H5Fopen(FILE, H5F_ACC_RDONLY, H5P_DEFAULT);
   H5REPORT(file_id);

   /* Locate the datasets. */
   dataset = H5Dopen(file_id, "/dset"); H5REPORT(dataset);
   parmset = H5Dopen(file_id,"/parm");  H5REPORT(parmset);

   /* Read data back */
   status = H5Dread
     (parmset,H5T_NATIVE_INT,H5S_ALL,H5S_ALL,H5P_DEFAULT,
      &parm); H5REPORT(status);
   printf("parameter value: %d\n",parm);

   status = H5Dread
     (dataset,H5T_NATIVE_DOUBLE,H5S_ALL,H5S_ALL,H5P_DEFAULT,
      data); H5REPORT(status);
   printf("arbitrary data point [1,2]: %e\n",data[1*6+2]);

   /* Terminate access to the datasets */
   status = H5Dclose(dataset); H5REPORT(status);
   status = H5Dclose(parmset); H5REPORT(status);

   /* Close the file. */
   status = H5Fclose(file_id);
}
```

```
 %% ./allread
parameter value: 37
arbitrary data point [1,2]: 8.500000e+00
```

# Chapter 8

# Parallel I/O

Parallel I/O is a tricky subject. You can try to let all processors jointly write one file, or to write a file per process and combine them later. With the standard mechanisms of your programming language there are the following considerations:

- On clusters where the processes have individual file systems, the only way to write a single file is to let it be generated by a single processor.
- Writing one file per process is easy to do, but
  - You need a post-processing script;
  - if the files are not on a shared file system (such as *Lustre*), it takes additional effort to bring them together;
  - if the files *are* on a shared file system, writing many files may be a burden on the metadata server.
- On a shared file system it is possible for all files to open the same file and set the file pointer individually. This can be difficult if the amount of data per process is not uniform.

Illustrating the last point:

```
// pseek.c
FILE *pfile;
pfile = fopen("pseek.dat","w");
fseek(pfile,procid*sizeof(int),SEEK_CUR);
//  fseek(pfile,procid*sizeof(char),SEEK_CUR);
fprintf(pfile,"%d\n",procid);
fclose(pfile);
```

MPI also has its own portable I/O: *MPI I/O*, for which see chapter *Parallel Programming book, chapter 10*.

Alternatively, one could use a library such as *hdf5*; see 7.

For a great discussion see [15], from which figures here are taken.

## 8.1 Use sequential I/O

MPI processes can do anything a regular process can, including opening a file. This is the simplest form of parallel I/O: every MPI process opens its own file. To prevent write collisions,

- you use `MPI_Comm_rank` to generate a unique file name, or
- you use a local file system, typically `/tmp`, that is unique per process, or at least per the group of processes on a node.

For reading it is actually possible for all processes to open the same file, but for reading this is not really feasible. Hence the unique files.

## 8.2 MPI I/O

In chapter *Parallel Programming book, section 10* we discuss MPI I/O. This is a way for all processes on a communicator to open a single file, and write to it in a coordinated fashion. This has the big advantage that the end result is an ordinary Unix file.

## 8.3 Higher level libraries

Libraries such as *NetCDF* or *HDF5* (see chapter 7) offer advantages over MPI I/O:

- Files can be OS-independent, removing worries such as about *little-endian* storage.
- Files are self-documenting: they contain the metadata describing their contents.

# Chapter 9

# Plotting with GNUplot

The *gnuplot* utility is a simple program for plotting sets of points or curves. This very short tutorial will show you some of the basics. For more commands and options, see the manual http://www.gnuplot.info/docs/gnuplot.html.

## 9.1 Usage modes

The two modes for running `gnuplot` are *interactive* and *from file*. In interactive mode, you call `gnuplot` from the command line, type commands, and watch output appear; you terminate an interactive session with `quit`. If you want to save the results of an interactive session, do `save "name.plt"`. This file can be edited, and loaded with `load "name.plt"`.

Plotting non-interactively, you call `gnuplot <your file>`.

The output of `gnuplot` can be a picture on your screen, or drawing instructions in a file. Where the output goes depends on the setting of the *terminal*. By default, `gnuplot` will try to draw a picture. This is equivalent to declaring

```
set terminal x11
```

or `aqua`, `windows`, or any choice of graphics hardware.

For output to file, declare

```
set terminal pdf
```

or `fig`, `latex`, `pbm`, et cetera. Note that this will only cause the pdf commands to be written to your screen: you need to direct them to file with

```
set output "myplot.pdf"
```

or capture them with

```
gnuplot my.plt > myplot.pdf
```

## 9.2 Plotting

The basic plot commands are `plot` for 2D, and `splot` ('surface plot') for 3D plotting.

### 9.2.1 Plotting curves

By specifying

```
plot x**2
```

you get a plot of $f(x) = x^2$; `gnuplot` will decide on the range for $x$. With

```
set xrange [0:1]
plot 1-x title "down", x**2 title "up"
```

you get two graphs in one plot, with the $x$ range limited to $[0, 1]$, and the appropriate legends for the graphs. The variable `x` is the default for plotting functions.

Plotting one function against another – or equivalently, plotting a parametric curve – goes like this:

```
set parametric
plot [t=0:1.57] cos(t),sin(t)
```

which gives a quarter circle.

To get more than one graph in a plot, use the command `set multiplot`.

### 9.2.2 Plotting data points

It is also possible to plot curves based on data points. The basic syntax is `plot 'datafile'`, which takes two columns from the data file and interprets them as $(x, y)$ coordinates. Since data files can often have multiple columns of data, the common syntax is `plot 'datafile' using 3:6` for columns 3 and 6. Further qualifiers like `with lines` indicate how points are to be connected.

Similarly, `splot "datafile3d.dat" 2:5:7` will interpret three columns as specifying $(x, y, z)$ coordinates for a 3D plot.

If a data file is to be interpreted as level or height values on a rectangular grid, do `splot "matrix.dat" matrix` for data points; connect them with

```
split "matrix.dat" matrix with lines
```

### 9.2.3 Customization

Plots can be customized in many ways. Some of these customizations use the `set` command. For instance,

```
  set xlabel "time"
  set ylabel "output"
  set title "Power curve"
```

You can also change the default drawing style with

```
set style function dots
```

(`dots`, `lines`, `dots`, `points`, et cetera), or change on a single plot with

```
plot f(x) with points
```

## 9.3 Workflow

Imagine that your code produces a dataset that you want to plot, and you run your code for a number of inputs. It would be nice if the plotting can be automated. Gnuplot itself does not have the facilities for this, but with a little help from shell programming this is not hard to do.

Suppose you have data files

```
  data1.dat data2.dat data3.dat
```

and you want to plot them with the same gnuplot commands. You could make a file `plot.template`:

```
set term pdf
set output "FILENAME.pdf"
plot "FILENAME.dat"
```

The string `FILENAME` can be replaced by the actual file names using, for instance `sed`:

```
for d in data1 data2 data3 ; do
  cat plot.template | sed s/FILENAME/$d/ > plot.cmd
  gnuplot plot.cmd
done
```

Variations on this basic idea are many.

# Chapter 10

# Good coding practices

Sooner or later, and probably sooner than later, every programmer is confronted with code not behaving as intended. In this section you will learn some techniques of dealing with this problem. At first we will see a number of techniques for *preventing* errors; in the next chapter we will discuss debugging, the process of finding the inevitable errors in a program, once they have occurred.

## 10.1 Defensive programming

In this section we will discuss a number of techniques that are aimed at preventing the likelihood of programming errors, or increasing the likelihood of them being found at runtime. We call this *defensive programming*.

Scientific codes are often large and involved, so it is a good practice to code knowing that you are going to make mistakes and prepare for them. Another good coding practice is the use of tools: there is no point in reinventing the wheel if someone has already done it for you. Some of these tools are be described in other sections:

- Build systems, such as Make, Scons, Bjam; see section 3.
- Source code management with Git; see section 5.
- Regression testing and designing with testing in mind (unit testing)

First we will have a look at runtime sanity checks, where you test for things that can not or should not happen.

### 10.1.1 Assertions

In the things that can go wrong with a program we can distinguish between errors and bugs. Errors are things that legitimately happen but that should not. File systems are common sources of errors: a program wants to open a file but the file doesn't exist because the user mistyped the name, or the program writes to a file but the disk is full. Other errors can come from arithmetic, such as *overflow* errors.

On the other hand, a *bug* in a program is an occurrence that cannot legitimately occur. Of course, 'legitimately' here means 'according to the programmer's intentions'. Bugs can often be described as 'the computer always does what you ask, not necessarily what you want'.

Assertions serve to detect bugs in your program: an *assertion* is a predicate that should be true at a certain point in your program. Thus, an assertion failing means that you didn't code what you intended to code. An assertion is typically a statement in your programming language, or a preprocessor macro; upon failure of the assertion, your program will stop.

Some examples of assertions:

- If a subprogram has an array argument, it is a good idea to test whether the actual argument is a null pointer before indexing into the array.
- Similarly, you could test a dynamically allocated data structure for not having a null pointer.
- If you calculate a numerical result for which certain mathematical properties hold, for instance you are writing a sine function, for which the result has to be in $[-1, 1]$, you should test whether this property indeed holds for the result.

Assertions are often disabled in a program once it's sufficiently tested. The reason for this is that assertions can be expensive to execute. For instance, if you have a complicated data structure, you could write a complicated integrity test, and perform that test in an assertion, which you put after every access to the data structure.

Because assertions are often disabled in the 'production' version of a code, they should not affect any stored data . If they do, your code may behave differently when you're testing it with assertions, versus how you use it in practice without them. This is also formulated as 'assertions should not have *side-effects*'.

#### *10.1.1.1 The C* `assert` *macro*

The C standard library has a file `assert.h` which provides an `assert()` macro. Inserting `assert(foo)` has the following effect: if `foo` is zero (false), a diagnostic message is printed on standard error:

```
Assertion failed: foo, file filename, line line-number
```

which includes the literal text of the expression, the file name, and line number; and the program is subsequently stopped. Here is an example:

```
#include<assert.h>

void open_record(char *record_name)
{
  assert(record_name!=NULL);
  /* Rest of code */
}

int main(void)
{
  open_record(NULL);
}
```

The `assert` macro can be disabled by defining the `NDEBUG` macro.

#### *10.1.1.2 An assert macro for Fortran*

(Thanks to Robert Mclay for this code.)

```
#if (defined( GFORTRAN ) || defined( G95 ) || defined ( PGI) )
# define MKSTR(x) "x"
#else
# define MKSTR(x) #x
#endif
#ifndef NDEBUG
# define ASSERT(x, msg) if (.not. (x) ) \
                call assert( FILE , LINE ,MKSTR(x),msg)
#else
# define ASSERT(x, msg)
#endif
subroutine assert(file, ln, testStr, msgIn)
implicit none
character(*) :: file, testStr, msgIn
integer :: ln
print *, "Assert: ",trim(testStr)," Failed at ",trim(file),":",ln
print *, "Msg:", trim(msgIn)
stop
end subroutine assert
```

which is used as

```
ASSERT(nItemsSet.gt.arraySize,"Too many elements set")
```

### 10.1.2 Use of error codes

In some software libraries (for instance MPI or PETSc) every subprogram returns a result, either the function value or a parameter, to indicate success or failure of the routine. It is good programming practice to check these error parameters, even if you think that nothing can possibly go wrong.

It is also a good idea to write your own subprograms in such a way that they always have an error parameter. Let us consider the case of a function that performs some numerical computation.

```
float compute(float val)
{
  float result;
  result = ... /* some computation */
  return result;
}

float value,result;
result = compute(value);
```

Looks good? What if the computation can fail, for instance:

```
result = ... sqrt(val) ... /* some computation */
```

How do we handle the case where the user passes a negative number?

```
float compute(float val)
{
  float result;
  if (val<0) { /* then what? */
  } else
    result = ... sqrt(val) ... /* some computation */
  return result;
}
```

We could print an error message and deliver some result, but the message may go unnoticed, and the calling environment does not really receive any notification that something has gone wrong.

The following approach is more flexible:

```
int compute(float val,float *result)
{
  float result;
  if (val<0) {
    return -1;
  } else {
    *result = ... sqrt(val) ... /* some computation */
  }
  return 0;
}

float value,result; int ierr;
ierr = compute(value,&result);
if (ierr!=0) { /* take appropriate action */
}
```

You can save yourself a lot of typing by writing

```
#define CHECK_FOR_ERROR(ierr) \
  if (ierr!=0) { \
    printf("Error %d detected\n",ierr); \
    return -1 ; }
....
ierr = compute(value,&result); CHECK_FOR_ERROR(ierr);
```

The C Preprocessor (CPP) has built-in macros that lend themselves to informative error reporting. The following macro not only checks on an error condition, but also reports where the error occurred:

```
#define CHECK_FOR_ERROR(ierr) \
  if (ierr!=0) { \
    printf("Error %d detected in line %d of file %s\n",\
           ierr,__LINE__,__FILE__); \
    return -1 ; }
```

Note that this macro not only prints an error message, but also does a further return. This means that, if you adopt this use of error codes systematically, you will get a full backtrace of the calling tree if an error occurs. (In the Python language this is precisely the wrong approach since the backtrace is built-in.)

## 10.2 Guarding against memory errors

In scientific computing it goes pretty much without saying that you will be working with large amounts of data. Some programming languages make managing data easy, others, one might say, make making errors with data easy.

The following are some examples of *memory violations*.

- Writing outside array bounds. If the address is outside the user memory, your code may exit with an error such as *segmentation violation*, and the error is reasonably easy to find. If the address is just outside an array, it will corrupt data but not crash the program; such an error may go undetected for a long time, as it can have no effect, or only introduce subtly wrong values in your computation.
- Reading outside array bounds can be harder to find than errors in writing, as it will often not stop your code, but only introduce wrong values.
- The use of uninitialized memory is similar to reading outside array bounds, and can go undetected for a long time. One variant of this is through attaching memory to an unallocated pointer.
  This particular kind of error can manifest itself in interesting behavior. Let's say you notice that your program misbehaves, you recompile it with debug mode to find the error, and now the error no longer occurs. This is probably due to the effect that, with low optimization levels, all allocated arrays are filled with zeros. Therefore, your code was originally reading a random value, but is now getting a zero.

This section contains some techniques to prevent errors in dealing with memory that you have reserved for your data.

### 10.2.1 Array bound checking and other memory techniques

Array bound checking, that is, detecting whether an array access is indeed to a legal location, carries runtime overhead. For that reason you may want to do this only in the testing phrase of a code, or keep it out of compute-intensive loops.

#### *10.2.1.1 C*

The C language has arrays, but they suffer from 'pointer decay': they behave largely like pointers in memory. Thus, bounds checking is hard, other than with external tools like *Valgrind*.

#### *10.2.1.2 C++*

C++ has the containers such as `std::vector` which support bound checking:

```
vector<float> x(25);
x.at(26) = y; // throws an exception
```

On the other hand, the C-style `x[26]` does not perform such checks.

#### *10.2.1.3 Fortran*

Fortran arrays are more restricted than C arrays, so compilers often support a flag for activating runtime bounds checking. For *gfortran* that is `-fbounds-check`.

### 10.2.2 Memory leaks

We say that a program has a *memory leak*, if it allocates memory, and subsequently loses track of that memory. The operating system then thinks the memory is in use, while it is not, and as a result the computer memory can get filled up with allocated memory that serves no useful purpose.

In this example data is allocated inside a lexical scope:

```
for (i=.... ) {
  real *block = malloc( /* large number of bytes */ )
  /* do something with that block of memory */
  /* and forget to call "free" on that block */
}
```

The block of memory is allocated in each iteration, but the allocation of one iteration is no longer available in the next. A similar example can be made with allocating inside a conditional.

It should be noted that this problem is far less serious in Fortran, where memory is deallocated automatically as a variable goes out of scope.

There are various tools for detecting memory errors: Valgrind, DMALLOC, Electric Fence. For valgrind, see section 11.8.

### 10.2.3 Roll-your-own malloc

Many programming errors arise from improper use of dynamically allocated memory: the program writes beyond the bounds, or writes to memory that has not been allocated yet, or has already been freed. While some compilers can do bound checking at runtime, this slows down your program. A better strategy is to write your own memory management. Some libraries such as PETSc already supply an enhanced malloc;

if this is available you should certainly make use of it. (The *gcc* compiler has a function `mcheck`, defined in `mcheck.h`, that has a similar function.)

If you write in C, you will probably know the `malloc` and `free` calls:

```
int *ip;
ip = (int*) malloc(500*sizeof(int));
if (ip==0) {/* could not allocate memory */}
..... do stuff with ip .....
free(ip);
```

You can save yourself some typing by

```
#define MYMALLOC(a,b,c) \
  a = (c*)malloc(b*sizeof(c)); \
  if (a==0) {/* error message and appropriate action */}

int *ip;
MYMALLOC(ip,500,int);
```

Runtime checks on memory usage (either by compiler-generated bounds checking, or through tools like valgrind or Rational Purify) are expensive, but you can catch many problems by adding some functionality to your malloc. What we will do here is to detect memory corruption after the fact.

We allocate a few integers to the left and right of the allocated object (line 1 in the code below), and put a recognizable value in them (line 2 and 3), as well as the size of the object (line 2). We then return the pointer to the actually requested memory area (line 4).

```
#define MEMCOOKIE 137
#define MYMALLOC(a,b,c) { \
  char *aa; int *ii; \
  aa = malloc(b*sizeof(c)+3*sizeof(int)); /* 1 */ \
  ii = (int*)aa; ii[0] = b*sizeof(c); \
          ii[1] = MEMCOOKIE;              /* 2 */ \
  aa = (char*)(ii+2); a = (c*)aa ;        /* 4 */ \
  aa = aa+b*sizesof(c); ii = (int*)aa; \
          ii[0] = MEMCOOKIE;              /* 3 */ \
  }
```

Now you can write your own `free`, which tests whether the bounds of the object have not been written over.

```
#define MYFREE(a) { \
  char *aa; int *ii,; ii = (int*)a; \
  if (*(--ii)!=MEMCOOKIE) printf("object corrupted\n"); \
  n = *(--ii); aa = a+n; ii = (int*)aa; \
  if (*ii!=MEMCOOKIE)  printf("object corrupted\n"); \
```

```
}
```

You can extend this idea: in every allocated object, also store two pointers, so that the allocated memory areas become a doubly linked list. You can then write a macro `CHECKMEMORY` which tests all your allocated objects for corruption.

Such solutions to the memory corruption problem are fairly easy to write, and they carry little overhead. There is a memory overhead of at most 5 integers per object, and there is practically no performance penalty.

(Instead of writing a wrapper for `malloc`, on some systems you can influence the behavior of the system routine. On linux, `malloc` calls hooks that can be replaced with your own routines; see http://www.gnu.org/s/libc/manual/html_node/Hooks-for-Malloc.html.)

## 10.3 Testing

There are various philosophies for testing the correctness of a code.

- Correctness proving: the programmer draws up predicates that describe the intended behavior of code fragments and proves by mathematical techniques that these predicates hold [10, 5].
- Unit testing: each routine is tested separately for correctness. This approach is often hard to do for numerical codes, since with floating point numbers there is essentially an infinity of possible inputs, and it is not easy to decide what would constitute a sufficient set of inputs.
- Integration testing: test subsystems
- System testing: test the whole code. This is often appropriate for numerical codes, since we often have model problems with known solutions, or there are properties such as bounds that need to hold on the global solution.
- Test-driven design: the program development process is driven by the requirement that testing is possible at all times.

With parallel codes we run into a new category of difficulties with testing. Many algorithms, when executed in parallel, will execute operations in a slightly different order, leading to different roundoff behavior. For instance, the parallel computation of a vector sum will use partial sums. Some algorithms have an inherent damping of numerical errors, for instance stationary iterative methods (section HPC book, section 5.5.1), but others have no such built-in error correction (nonstationary methods; section HPC book, section 5.5.8). As a result, the same iterative process can take different numbers of iterations depending on how many processors are used.

### 10.3.1 Unit testing

Unit testing is a way to ensure correctness of a code. For that it's necessary that the tests have full *coverage* of the code: all statements in your code should be part of a test.

Unit tests are also a way to document the use of a code: they show the intended use of the code.

A few notes:

- Global state in your program makes it hard to test, since it carries information between tests.
- Tests should not reproduce the logic of your code: if the program logic is faulty, the test will be too.
- Tests should be short, and obey the *single-responsibility* principle. Naming your tests is good to keep them focused.

### 10.3.2 Test-driven design and development

In test-driven design there is a strong emphasis on the code always being testable. The basic ideas are as follows.

- Both the whole code and its parts should always be testable.
- When extending the code, make only the smallest change that allows for testing.
- With every change, test before and after.
- Assure correctness before adding new features.

Volume 3 of this series discusses Test-Drive Development (TDD) and Unit Testing, using the *Catch2* framework. See *Introduction to Scientific Programming book, chapter 70*.

# Chapter 11

# Debugging

> Debugging is like being the detective in a crime movie where you are also the murderer.
> (Filipe Fortes, 2013)

When a program misbehaves, *debugging* is the process of finding out *why*. There are various strategies of finding errors in a program. The crudest one is debugging by print statements. If you have a notion of where in your code the error arises, you can edit your code to insert print statements, recompile, rerun, and see if the output gives you any suggestions. There are several problems with this:

- The edit/compile/run cycle is time consuming, especially since
- often the error will be caused by an earlier section of code, requiring you to edit, compile, and rerun repeatedly. Furthermore,
- the amount of data produced by your program can be too large to display and inspect effectively, and
- if your program is parallel, you probably need to print out data from all processors, making the inspection process very tedious.

For these reasons, the best way to debug is by the use of an interactive *debugger*, a program that allows you to monitor and control the behavior of a running program. In this section you will familiarize yourself with *gdb* and *lldb*, the open source debuggers of the *GNU* and *clang* projects respectively. Other debuggers are proprietary, and typically come with a compiler suite. Another distinction is that gdb is a commandline debugger; there are graphical debuggers such as *ddd* (a frontend to gdb) or *DDT* and *TotalView* (debuggers for parallel codes). We limit ourselves to gdb, since it incorporates the basic concepts common to all debuggers.

In this tutorial you will debug a number of simple programs with gdb and valgrind. The files can be found in the repository in the directory `code/gdb`.

## 11.1 Compiling for debug

You often need to recompile your code before you can debug it. A first reason for this is that the binary code typically knows nothing about what variable names corresponded to what memory locations, or what lines in the source to what instructions. In order to make the binary executable know this, you have to include the *symbol table* in it, which is done by adding the `-g` option to the compiler line.

Table 11.1: List of common gdb / lldb commands.

| gdb | lldb |
|---|---|
| Starting a debugger run | |
| `$ gdb program`<br>`(gdb) run` | `$ lldb program`<br>`(lldb) run` |
| Displaying a stack trace | |
| `(gdb) where` | `(lldb) thread backtrace` |
| Investigate a specific frame | |
| `frame 2` | `frame select 2` |
| Run/step | |
| `run / step / continue` | thread continue / step-in/over/out |
| Set a breakpoint at a line | |
| `break foo.c:12`<br>`break foo.c:12 if n>0`<br>`info breakpoints` | `breakpoint set [ -f foo.c ] -l 12` |
| Set a breakpoint for exceptions | |
| `catch throw` | `break set -E C++` |

Usually, you also need to lower the *compiler optimization level*: a production code will often be compiled with flags such as `-O2` or `-Xhost` that try to make the code as fast as possible, but for debugging you need to replace this by `-O0` ('oh-zero'). The reason is that higher levels will reorganize your code, making it hard to relate the execution to the source[1].

## 11.2 Invoking the debugger

There are three ways of using gdb: using it to start a program, attaching it to an already running program, or using it to inspect a *core dump*. We will only consider the first possibility.

Starting a debugger run

| gdb | lldb |
|---|---|
| `$ gdb program` | `$ lldb program` |
| `(gdb) run` | `(lldb) run` |

Here is an example of how to start gdb with program that has no arguments (Fortran users, use `hello.F`):

tutorials/gdb/c/hello.c

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

%% cc -g -o hello hello.c
# regular invocation:
%% ./hello
hello world
# invocation from gdb:
%% gdb hello
GNU gdb 6.3.50-20050815 # ..... [version info]
Copyright 2004 Free Software Foundation, Inc. .... [copyright info] ....
(gdb) run
Starting program: /home/eijkhout/tutorials/gdb/hello
Reading symbols for shared libraries +. done
hello world

Program exited normally.
(gdb) quit
%%
```

1. Typically, actual code motion is done by `-O3`, but at level `-O2` the compiler will inline functions and make other simplifications.

Important note: the program was compiled with the *debug flag* `-g`. This causes the *symbol table* (that is, the translation from machine address to program variables) and other debug information to be included in the binary. This will make your binary larger than strictly necessary, but it will also make it slower, for instance because the compiler will not perform certain optimizations[2].

To illustrate the presence of the symbol table do

```
%% cc -g -o hello hello.c
%% gdb hello
GNU gdb 6.3.50-20050815 # ..... version info
(gdb) list
```

and compare it with leaving out the `-g` flag:

```
%% cc -o hello hello.c
%% gdb hello
GNU gdb 6.3.50-20050815 # ..... version info
(gdb) list
```

For a program with commandline input we give the arguments to the `run` command (Fortran users use `say.F`):

tutorials/gdb/c/say.c

```
#include <stdlib.h>
#include <stdio.h>
int main(int argc,char **argv) {
  int i;
  for (i=0; i<atoi(argv[1]); i++)
    printf("hello world\n");
  return 0;
}
```

```
%% cc -o say -g say.c
%% ./say 2
hello world
hello world
%% gdb say
.... the usual messages ...
(gdb) run 2
Starting program: /home/eijkhout/tutorials/gdb
Reading symbols for shared libraries +. done
hello world
hello world

Program exited normally.
```

## 11.3 Finding errors: where, frame, print

Let us now consider some programs with errors.

2. Compiler optimizations are not supposed to change the semantics of a program, but sometimes do. This can lead to the nightmare scenario where a program crashes or gives incorrect results, but magically works correctly with compiled with debug and run in a debugger.

### 11.3.1 C programs

The following code has several errors. We will use the debugger to uncover them.

```
// square.c
  int nmax,i;
  float *squares,sum;

  fscanf(stdin,"%d",nmax);
  for (i=1; i<=nmax; i++) {
    squares[i] = 1./(i*i); sum += squares[i];
  }
  printf("Sum: %e\n",sum);
```

```
%% cc -g -o square square.c
%% ./square
5000
Segmentation fault
```

The *segmentation fault* (other messages are possible too) indicates that we are accessing memory that we are not allowed to, making the program exit. A debugger will quickly tell us where this happens:

```
%% gdb square
(gdb) run
50000

Program received signal EXC_BAD_ACCESS, Could not access memory.
Reason: KERN_INVALID_ADDRESS at address: 0x000000000000eb4a
0x00007fff824295ca in __svfscanf_l ()
```

Apparently the error occurred in a function `__svfscanf_l`, which is not one of ours, but a system function. Using the `backtrace` (or `bt`, also `where` or `w`) command we display the *call stack*. This usually allows us to find out where the error lies:

Displaying a stack trace

| gdb | lldb |
|---|---|
| `(gdb) where` | `(lldb) thread backtrace` |

```
(gdb) where
#0  0x00007fff824295ca in __svfscanf_l ()
#1  0x00007fff8244011b in fscanf ()
#2  0x0000000100000e89 in main (argc=1, argv=0x7fff5fbfc7c0) at square.c:7
```

We inspect the actual problem:

Investigate a specific frame

| gdb | clang |
|---|---|
| `frame 2` | `frame select 2` |

We take a close look at line 7, and see that we need to change `nmax` to `&nmax`.

There is still an error in our program:

```
(gdb) run
50000

Program received signal EXC_BAD_ACCESS, Could not access memory.
Reason: KERN_PROTECTION_FAILURE at address: 0x000000010000f000
0x0000000100000ebe in main (argc=2, argv=0x7fff5fbfc7a8) at square1.c:9
9           squares[i] = 1./(i*i); sum += squares[i];
```

We investigate further:

```
(gdb) print i
$1 = 11237
(gdb) print squares[i]
Cannot access memory at address 0x10000f000
(gdb) print squares
$2 = (float *) 0x0
```

and we quickly see that we forgot to allocate `squares`.

By the way, we were lucky here: this sort of memory errors is not always detected. Starting our programm with a smaller input does not lead to an error:

```
(gdb) run
50
Sum: 1.625133e+00

Program exited normally.
```

Memory errors can also occur if we have a legitimate array, but we access it outside its bounds. The following program fills an array, forward, and reads it out, backward. However, there is an indexing error in the second loop.

```
// up.c
  int nlocal = 100,i;
  double s, *array = (double*) malloc(nlocal*sizeof(double));
  for (i=0; i<nlocal; i++) {
    double di = (double)i;
    array[i] = 1/(di*di);
  }
  s = 0.;
  for (i=nlocal-1; i>=0; i++) {
    double di = (double)i;
    s += array[i];
  }
```

```
  Program received signal EXC_BAD_ACCESS, Could not access memory.
```

```
Reason: KERN_INVALID_ADDRESS at address: 0x0000000100200000
0x0000000100000f43 in main (argc=1, argv=0x7fff5fbfe2c0) at up.c:15
15          s += array[i];
(gdb) print array
$1 = (double *) 0x100104d00
(gdb) print i
$2 = 128608
```

You see that the index where the debugger finally complains is quite a bit larger than the size of the array.

**Exercise** 11.1. Can you think of a reason why indexing out of bounds is not immediately fatal? What would determine where it does become a problem? (Hint: how is computer memory structured?)

In section 11.8 you will see a tool that spots any out-of-bound indexing.

### 11.3.2 Fortran programs

Compile and run the following program:

MISSING SNIPPET gdb-squaref

It should end prematurely with a message such as 'Illegal instruction'. Running the program in gdb quickly tells you where the problem lies:

```
(gdb) run
Starting program: tutorials/gdb//fsquare
Reading symbols for shared libraries ++++. done

Program received signal EXC_BAD_INSTRUCTION,
Illegal instruction/operand.
0x0000000100000da3 in square () at square.F:7
7                 sum = sum + squares(i)
```

We take a close look at the code and see that we did not allocate `squares` properly.

## 11.4 Stepping through a program

Stepping through a program

| gdb | lldb | meaning |
|---|---|---|
| `run` | | start a run |
| `cont` | | continue from breakpoint |
| `next` | | next statement on same level |
| `step` | | next statement, this level or next |

Often the error in a program is sufficiently obscure that you need to investigate the program run in detail. Compile the following program

```
// roots.c
float root(int n)
{
  float r;
  r = sqrt(n);
  return r;
}

int main() {
  feenableexcept(FE_INVALID | FE_OVERFLOW);
  int i;
  float x=0;
  for (i=100; i>-100; i--)
    x += root(i+5);
  printf("sum: %e\n",x);
```

and run it:

```
%% ./roots
sum: nan
```

Start it in gdb as before:

```
%% gdb roots
GNU gdb 6.3.50-20050815
Copyright 2004 Free Software Foundation, Inc.
....
```

but before you run the program, you set a *breakpoint* at `main`. This tells the execution to stop, or 'break', in the main program.

```
(gdb) break main
Breakpoint 1 at 0x100000ea6: file root.c, line 14.
```

Now the program will stop at the first executable statement in `main`:

```
(gdb) run
Starting program: tutorials/gdb/c/roots
Reading symbols for shared libraries +. done

Breakpoint 1, main () at roots.c:14
14        float x=0;
```

Most of the time you will set a breakpoint at a specific line:

| Set a breakpoint at a line | |
|---|---|
| gdb | lldb |
| `break foo.c:12` | `breakpoint set [ -f foo.c ] -l 12` |

If execution is stopped at a breakpoint, you can do various things, such as issuing the `step` command:

```
Breakpoint 1, main () at roots.c:14
14        float x=0;
(gdb) step
15        for (i=100; i>-100; i--)
(gdb)
16          x += root(i);
(gdb)
```

(if you just hit return, the previously issued command is repeated). Do a number of `steps` in a row by hitting return. What do you notice about the function and the loop?

Switch from doing `step` to doing `next`. Now what do you notice about the loop and the function?

Set another breakpoint: `break 17` and do `cont`. What happens?

Rerun the program after you set a breakpoint on the line with the `sqrt` call. When the execution stops there do `where` and `list`.

- If you set many breakpoints, you can find out what they are with `info breakpoints`.
- You can remove breakpoints with `delete n` where `n` is the number of the breakpoint.
- If you restart your program with `run` without leaving gdb, the breakpoints stay in effect.
- If you leave gdb, the breakpoints are cleared but you can save them: `save breakpoints <file>`. Use `source <file>` to read them in on the next gdb run.

## 11.5 Inspecting values

Run the previous program again in gdb: set a breakpoint at the line that does the `sqrt` call before you actually call `run`. When the program gets to line 8 you can do `print n`. Do `cont`. Where does the program stop?

If you want to repair a variable, you can do `set var=value`. Change the variable `n` and confirm that the square root of the new value is computed. Which commands do you do?

## 11.6 Breakpoints

If a problem occurs in a loop, it can be tedious keep typing `cont` and inspecting the variable with `print`. Instead you can add a condition to an existing breakpoint. First of all, you can make the breakpoint subject to a condition: with

```
condition 1 if (n<0)
```

breakpoint 1 will only obeyed if `n<0` is true.

You can also have a breakpoint that is only activated by some condition. The statement

```
break 8 if (n<0)
```

means that breakpoint 8 becomes (unconditionally) active after the condition `n<0` is encountered.

Set a breakpoint

| gdb | lldb |
|---|---|
| `break foo.c:12` | `breakpoint set [ -f foo.c ] -l 12` |
| `break foo.c:12 if n>0` | |

**Remark 16** *You can break on* `NaN` *with the following trick:*

```
break foo.c:12 if x!=x
```

*using the fact that* `NaN` *is the only number not equal to itself.*

Another possibility is to use `ignore 1 50`, which will not stop at breakpoint 1 the next 50 times.

Remove the existing breakpoint, redefine it with the condition `n<0` and rerun your program. When the program breaks, find for what value of the loop variable it happened. What is the sequence of commands you use?

You can set a breakpoint in various ways:

- `break foo.c` to stop when code in a certain file is reached;
- `break 123` to stop at a certain line in the current file;
- `break foo` to stop at subprogram `foo`
- or various combinations, such as `break foo.c:123`.

Information about breakpoints:

- If you set many breakpoints, you can find out what they are with `info breakpoints`.
- You can remove breakpoints with `delete n` where `n` is the number of the breakpoint.
- If you restart your program with `run` without leaving gdb, the breakpoints stay in effect.
- If you leave gdb, the breakpoints are cleared but you can save them: `save breakpoints <file>`. Use `source <file>` to read them in on the next gdb run.
- In languages with *exceptions*, such as *C++*, you can set a *catchpoint*:

Set a breakpoint for exceptions

| gdb | clang |
|---|---|
| `catch throw` | `break set -E C++` |

Finally, you can execute commands at a breakpoint:

```
break 45
command
print x
cont
end
```

This states that at line 45 variable `x` is to be printed, and execution should immediately continue.

If you want to run repeated gdb sessions on the same program, you may want to save an reload breakpoints. This can be done with

```
save-breakpoint filename
source filename
```

## 11.7 Memory debugging

Many problems in programming stem from memory errors. We start with a sort description of the most common types, and then discuss tools that help you detect them.

### 11.7.1 Type of memory errors

#### *11.7.1.1 Invalid pointers*

Dereferencing a pointer that does not point to an allocated object can lead to an error. If your pointer points into valid memory anyway, your computation will continue but with incorrect results.

However, it is more likely that your program will probably exit with a *segmentation violation* or a *bus error*.

#### *11.7.1.2 Out-of-bounds errors*

Addressing outside the bounds of an allocated object is less likely to crash your program and more likely to give incorrect results.

Exceeding bounds by a large enough amount will again give a segmentation violation, but going out of bounds by a small amount may read invalid data, or corrupt data of other variables, giving incorrect results that may go undetected for a long time.

#### *11.7.1.3 Memory leaks*

We speak of a *memory leak* if allocated memory becomes unreachable. Example:

```
if (something) {
  double *x = malloc(10*sizeofdouble);
  // do something with x
}
```

After the conditional, the allocated memory is not freed, but the pointer that pointed to has gone away.

This last type especially can be hard to find. Memory leaks will only surface in that your program runs out of memory. That in turn is detectable because your allocation will fail. It is a good idea to always check the return result of your `malloc` or `allocate` statement!

## 11.8 Memory debugging with Valgrind

Errors leading to memory problems are easy to make. In this section we will see how *valgrind* makes it possible to track down these errors. The use of valgrind is simplicity itself:

```
valgrind yourprogram yourargs
```

As a first example, consider out of bound addressing, also known as *buffer overflow*:

MISSING SNIPPET corruptbound

This is unlikely to crash your code, but the results are unpredictable, and this is certainly a failure of your program logic.

Valgrind indicates that this is an invalid read, what line it occurs on, and where the block was allocated:

```
==9112== Invalid read of size 4
==9112==    at 0x40233B: main (outofbound.cpp:10)
==9112==  Address 0x595fde8 is 0 bytes after a block of size 40 alloc'd
==9112==    at 0x4C2A483: operator new(unsigned long) (vg_replace_malloc.c:344)
==9112==    by 0x4023CD: allocate (new_allocator.h:111)
==9112==    by 0x4023CD: allocate (alloc_traits.h:436)
==9112==    by 0x4023CD: _M_allocate (stl_vector.h:296)
==9112==    by 0x4023CD: _M_create_storage (stl_vector.h:311)
==9112==    by 0x4023CD: _Vector_base (stl_vector.h:260)
==9112==    by 0x4023CD: _Vector_base (stl_vector.h:258)
==9112==    by 0x4023CD: vector (stl_vector.h:415)
==9112==    by 0x4023CD: main (outofbound.cpp:9)
```

**Remark 17** *Buffer overflows are a well-known security risk, typically associated with reading string input from a user source. Buffer overflows can be largely avoided by using C++ constructs such as* `cin` *and* `string` *instead of* `sscanf` *and character arrays.*

Valgrind is informative but cryptic, since it works on the bare memory, not on variables. Thus, these error messages take some exegesis. They state that line 10 reads a 4-byte object immediately after a block of 40 bytes that was allocated. In other words: the code is writing outside the bounds of an allocated array.

The next example performs a read on an array that has already been free'd. In this simple case you will actually get the expected output, but if the read comes much later than the free, the output can be anything.

MISSING SNIPPET corruptfree

Valgrind again states that this is an invalid read; it gives both where the block was allocated and where it was freed.

```
==11431== Invalid read of size 4
==11431==    at 0x40230C: main (free.cpp:13)
==11431==  Address 0x595fdcc is 12 bytes inside a block of size 40,000 free'd
==11431==    at 0x4C2B5CF: operator delete(void*, unsigned long) (vg_replace_malloc.c:595)
==11431==    by 0x402301: main (free.cpp:12)
==11431==  Block was alloc'd at
==11431==    at 0x4C2AB28: operator new[](unsigned long) (vg_replace_malloc.c:433)
==11431==    by 0x4022E0: main (free.cpp:10)
```

On the other hand, if you forget to free memory you have a *memory leak* (just imagine allocation, and not free'ing, in a loop)

MISSING SNIPPET corruptleak

which valgrind reports on:

```
==283234== LEAK SUMMARY:
==283234==    definitely lost: 40,000 bytes in 1 blocks
==283234==    indirectly lost: 0 bytes in 0 blocks
==283234==      possibly lost: 0 bytes in 0 blocks
==283234==    still reachable: 8 bytes in 1 blocks
==283234==         suppressed: 0 bytes in 0 blocks
```

Memory leaks are much more rare in C++ than in C because of containers such as `std::vector`. However, in sophisticated cases you may still do your own memory management, and you need to be aware of the danger of memory leaks.

If you do your own memory management, there is also a danger of writing to an array pointer that has not been allocated yet:

MISSING SNIPPET corruptinit

The behavior of this code depends on all sorts of things: if the pointer variable is zero, the code will crash. On the other hand, if it contains some random value, the write may succeed; provided you are not writing too far from that location.

The output here shows both the valgrind diagnosis, and the OS message when the program aborted:

```
==283234== LEAK SUMMARY:
==283234==    definitely lost: 40,000 bytes in 1 blocks
==283234==    indirectly lost: 0 bytes in 0 blocks
==283234==      possibly lost: 0 bytes in 0 blocks
==283234==    still reachable: 8 bytes in 1 blocks
==283234==         suppressed: 0 bytes in 0 blocks
```

### 11.8.1 Electric fence

The *electric fence* library is one of a number of tools that supplies a new `malloc` with debugging support. These are linked instead of the `malloc` of the standard `libc`.

```
cc -o program program.c -L/location/of/efence -lefence
```

Suppose your program has an out-of-bounds error. Running with gdb, this error may only become apparent if the bounds are exceeded by a large amount. On the other hand, if the code is linked with `libefence`, the debugger will stop at the very first time the bounds are exceeded.

## 11.9 Further reading

A good tutorial: http://www.dirac.org/linux/gdb/.

Reference manual: http://www.ofb.net/gnu/gdb/gdb_toc.html.

# Chapter 12

# Parallel debugging

When a program misbehaves, *debugging* is the process of finding out *why*. There are various strategies of finding errors in a program. The crudest one is debugging by print statements. If you have a notion of where in your code the error arises, you can edit your code to insert print statements, recompile, rerun, and see if the output gives you any suggestions. There are several problems with this:

- The edit/compile/run cycle is time consuming, especially since
- often the error will be caused by an earlier section of code, requiring you to edit, compile, and rerun repeatedly. Furthermore,
- the amount of data produced by your program can be too large to display and inspect effectively, and
- if your program is parallel, you probably need to print out data from all proccessors, making the inspection process very tedious.

For these reasons, the best way to debug is by the use of an interactive *debugger*, a program that allows you to monitor and control the behaviour of a running program. In this section you will familiarize yourself with *gdb*, which is the open source debugger of the *GNU* project. Other debuggers are proprietary, and typically come with a compiler suite. Another distinction is that gdb is a commandline debugger; there are graphical debuggers such as *ddd* (a frontend to gdb) or *DDT* and *TotalView* (debuggers for parallel codes). We limit ourselves to gdb, since it incorporates the basic concepts common to all debuggers.

In this tutorial you will debug a number of simple programs with gdb and valgrind. The files can be found in the repository in the directory `tutorials/debug_tutorial_files`.

## 12.1 Parallel debugging

Debugging parallel programs is harder than than sequential programs, because every sequential bug may show up, plus a number of new types, caused by the interaction of the various processes.

Here are a few possible parallel bugs:

- Processes can *deadlock* because they are waiting for a message that never comes. This typically happens with blocking send/receive calls due to an error in program logic.
- If an incoming message is unexpectedly larger than anticipated, a memory error can occur.
- A collective call will hang if somehow one of the processes does not call the routine.

There are few low-budget solutions to parallel debugging. The main one is to create an xterm for each process. We will describe this next. There are also commercial packages such as *DDT* and *TotalView*, that offer a GUI. They are very convenient but also expensive. The *Eclipse* project has a parallel package, *Eclipse PTP*, that includes a graphic debugger.

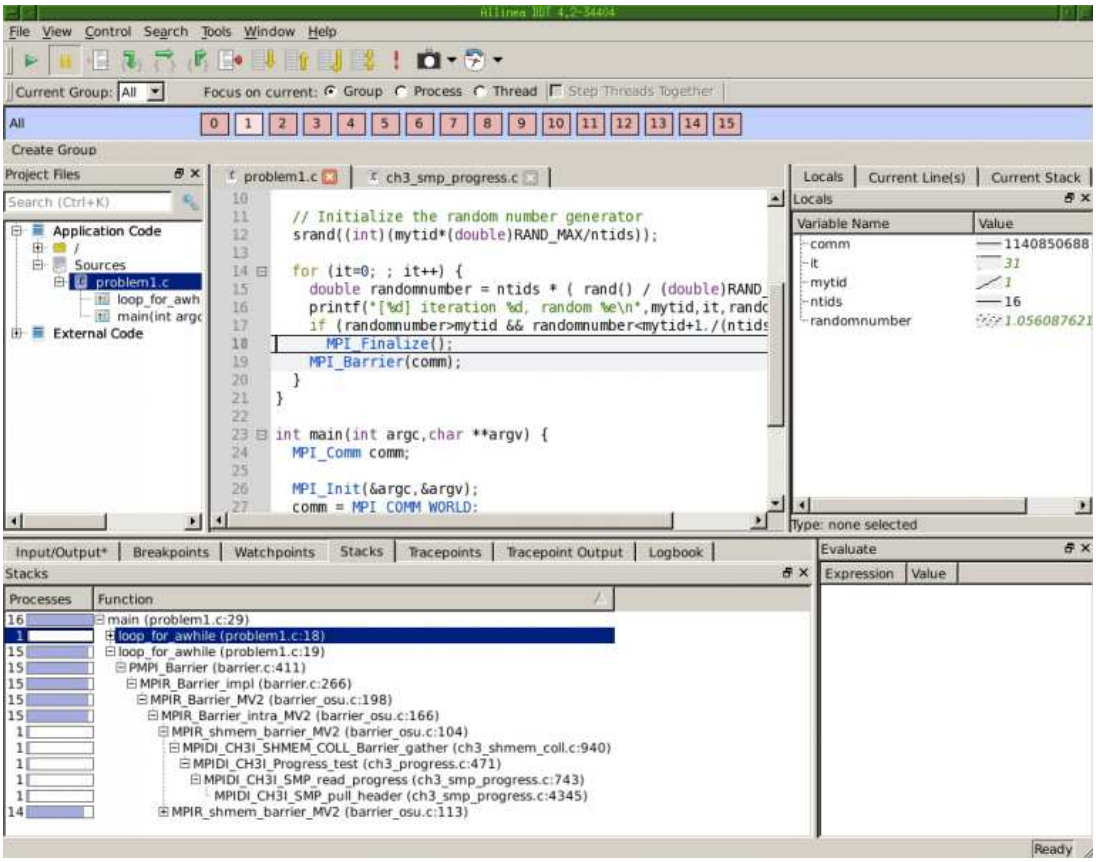



Figure 12.1: Display of 16 processes in the DDT debugger.

Debugging in parallel is harder than sequentially, because you will run errors that are only due to interaction of processes such as *deadlock*; see section HPC book, section 2.6.3.6.

As an example, consider this segment of MPI code:

```
MPI_Init(0,0);
// set comm, ntids, mytid
for (int it=0; ; it++) {
  double randomnumber = ntids * ( rand() / (double)RAND_MAX );
  printf("[%d] iteration %d, random %e\n",mytid,it,randomnumber);
  if (randomnumber>mytid && randomnumber<mytid+1./(ntids+1))
    MPI_Finalize();
}
MPI_Finalize();
```

Each process computes random numbers until a certain condition is satisfied, then exits. However, consider introducing a barrier (or something that acts like it, such as a reduction):

```
for (int it=0; ; it++) {
  double randomnumber = ntids * ( rand() / (double)RAND_MAX );
  printf("[%d] iteration %d, random %e\n",mytid,it,randomnumber);
  if (randomnumber>mytid && randomnumber<mytid+1./(ntids+1))
    MPI_Finalize();
  MPI_Barrier(comm);
}
MPI_Finalize();
```

Now the execution will hang, and this is not due to any particular process: each process has a code path from init to finalize that does not develop any memory errors or other runtime errors. However as soon as one process reaches the finalize call in the conditional it will stop, and all other processes will be waiting at the barrier.

Figure 12.1 shows the main display of the Allinea *DDT* debugger (http://www.allinea.com/products/ddt) at the point where this code stops. Above the source panel you see that there are 16 processes, and that the status is given for process 1. In the bottom display you see that out of 16 processes 15 are calling `MPI_Barrier` on line 19, while one is at line 18. In the right display you see a listing of the local variables: the value specific to process 1. A rudimentary graph displays the values over the processors: the value of `ntids` is constant, that of `mytid` is linearly increasing, and `it` is constant except for one process.

**Exercise** 12.1. Make and run `ring_1a`. The program does not terminate and does not crash. In the debugger you can interrupt the execution, and see that all processes are executing a receive statement. This is probably a case of deadlock. Diagnose and fix the error.

**Exercise** 12.2. The author of `ring_1c` was very confused about how MPI works. Run the program. While it terminates without a problem, the output is wrong. Set a breakpoint at the send and receive statements to figure out what is happening.

## 12.2 MPI debugging with gdb

You can not run parallel programs in gdb, but you can start multiple gdb processes that behave just like MPI processes! The command

```
mpirun -np <NP> xterm -e gdb ./program
```

create a number of `xterm` windows, each of which execute the commandline `gdb ./program`. And because these xterms have been started with `mpirun`, they actually form a communicator.

## 12.3 Full-screen parallel debugging with DDT

In this tutorial you will run and diagnose a few incorrect MPI programs using DDT. You can start a session with `ddt yourprogram &`, or use `File > New Session > Run` to specify a program name, and possibly parameters. In both cases you get a dialog where you can specify program parameters. It is also important to check the following:

- You can specify the number of cores here;
- It is usually a good idea to turn on memory checking;
- Make sure you specify the right MPI.

When DDT opens on your main program, it halts at the `MPI_Init` statement, and need to press the forward arrow, top left of the main window.

***Problem1*** This program has every process independently generate random numbers, and if the number meets a certain condition, stops execution. There is no problem with this code as such, so let's suppose you simply want to monitor its execution.

- Compile `abort.c`. Don't forget about the `-g -O0` flags; if you use the makefile they are included automatically.
- Run the program with DDT, you'll see that it concludes succesfully.
- Set a breakpoint at the Finalize statement in the subroutine, by clicking to the left of the line number. Now if you run the program you'll get a message that all processes are stopped at a breakpoint. Pause the execution.
- The 'Stacks' tab will tell you that all processes are the same point in the code, but they are not in fact in the same iteration.
- You can for instance use the 'Input/Output' tabs to see what every process has been doing.
- Alternatively, use the variables pane on the right to examine the `it` variable. You can do that for individual processes, but you can also control click on the `it` variable and choose `View as Array`. Set up the display as a one-dimensional array and check the iteration numbers.
- Activate the barrier statement and rerun the code. Make sure you have no breakpoints. Reason that the code will not complete, but just hang.
- Hit the general Pause button. Now what difference do you see in the 'Stacks' tab?

***Problem2*** Compile `problem1.c` and run it in DDT. You'll get a dialog warning about an error condition.

- Pause the program in the dialog. Notice that only the root process is paused. If you want to inspect other processes, press the general pause button. Do this.
- In the bottom panel click on `Stacks`. This gives you the 'call stack', which tells you what the processes were doing when you paused them. Where is the root process in the execution? Where are the others?
- From the call stack it is clear what the error was. Fix it and rerun with `File > Restart Session`.

***Problem2***

### 12.3.1 DDT running modes

DDT can be run several different ways.

1. If you are on a cluster with login nodes, compute nodes, and a batch system, you can run the DDT Graphical User Interface (GUI) and let it submit a batch job to the queue. The GUI will then pause until your job starts.
2. If your system does not have the login/compute node distinction, or if you are interactively on a compute node (such as with the `idev` command at *TACC*) you can start the GUI and let it run your program, bypassing the queue.
3. Using the *DDT reverse connect* mode you
   (a) Start the GUI, telling it to wait for a connection
   (b) Submit a batch job, adding an option for the connection.
4. Finally, you can run DDT completely off-line in a batch job, letting it output its result as an HTML file.

## 12.4 Further reading

A good tutorial: http://www.dirac.org/linux/gdb/.

Reference manual: http://www.ofb.net/gnu/gdb/gdb_toc.html.

# Chapter 13

# Language interoperability

Most of the time, a program is written is written in a single language, but in some circumstances it is necessary or desirable to mix sources in more than one language for a single executable. One such case is when a library is written in one language, but used by a program in another. In such a case, the library writer will probably have made it easy for you to use the library; this section is for the case that you find yourself in the place of the library writer. We will focus on the common case of *interoperability* between C/C++ and Fortran or Python.

This issue is complicated by the fact that both languages have been around for a long time, and various recent language standards have introduced mechanisms to facilitate interoperability. However, there is still a lot of old code around, and not all compilers support the latest standards. Therefore, we discuss both the old and the new solutions.

## 13.1 C/Fortran interoperability

### 13.1.1 Linker conventions

As explained above, a compiler turns a source file into a binary, which no longer has any trace of the source language: it contains in effect functions in machine language. The linker will then match up calls and definitions, which can be in different files. The problem with using multiple languages is then that compilers have different notions of how to translate function names from the source file to the binary file.

Let's look at codes (you can find example files in `tutorials/linking`):

```
// C:
void foo() {
  return;
}
! Fortran
      Subroutine foo()
      Return
      End Subroutine
```

After compilation you can use *nm* to investigate the binary *object file*:

```
%% nm fprog.o
0000000000000000 T _foo_
....
%% nm cprog.o
0000000000000000 T _foo
....
```

You see that internally the `foo` routine has different names: the Fortran name has an underscore appended. This makes it hard to call a Fortran routine from C, or vice versa. The possible name mismatches are:

- The Fortran compiler appends an underscore. This is the most common case.
- Sometimes it can append two underscores.
- Typically the routine name is lowercase in the object file, but uppercase is a possibility too.

Since C is a popular language to write libraries in, this means that the problem is often solved in the C library by:

- Appending an underscore to all C function names; or
- Including a simple wrapper call:

```
int SomeCFunction(int i,float f)
{
  // this is the actual function
}
int SomeCFunction_(int i,float f)
{
  return SomeCFunction(i,f);
}
```

### 13.1.2 Complex numbers

The *complex data types in C/C++ and Fortran* are compatible with each other. Here is an example of a C++ program linking to Lapack's complex vector scaling routine *zscal*.

```
// zscale.cxx
extern "C" {
void zscal_(int*,double complex*,double complex*,int*);
}
  complex double *xarray,*yarray, scale=2.;
  xarray = new double complex[n]; yarray = new double complex[n];
  zscal_(&n,&scale,xarray,&ione);
```

### 13.1.3 C bindings in Fortran 2003

With the latest Fortran standard there are explicit *C bindings*, making it possible to declare the external name of variables and routines:

```
module operator
  real, bind(C) :: x
contains
  subroutine s() bind(C,name='s')
  return
  end subroutine
end module

%% ifort -c fbind.F90
%% nm fbind.o
.... T _s
.... C _x
```

It is also possible to declare data types to be C-compatible:

```
Program fdata

  use iso_c_binding

  type, bind(C) :: c_comp
    real (c_float)  :: data
    integer (c_int) :: i
    type (c_ptr)    :: ptr
  end type

end Program fdata
```

The latest version of Fortran, unsupported by many compilers at this time, has mechanisms for interfacing to C.

- There is a module that contains named kinds, so that one can declare
  ```
  INTEGER,KIND(C_SHORT) :: i
  ```
- Fortran pointers are more complicated objects, so passing them to C is hard; Fortran2003 has a mechanism to deal with C pointers, which are just addresses.
- Fortran derived types can be made compatible with C structures.

## 13.2 C/C++ linking

Libraries written in C++ offer further problems. The C++ compiler makes external symbols by combining the names a class and its methods, in a process known as *name mangling*.

### 13.2.1 Mangling and demangling

Consider a simple C program:

```
#include <stdlib.h>
#include <stdio.h>
void bar(char *s) {
  printf("%s",s);
  return;
}
```

If you compile this and inspect the output with `nm` you get:

```
$ gcc -c foochar.c && nm foochar.o | grep bar
0000000000000000 T _bar
```

That is, apart from a leading underscore the symbol name is clear.

On the other hand, the identical program compiled as C++ gives

```
$ g++ -c foochar.c && nm foochar.o | grep bar
0000000000000000 T __Z3barPc
```

Why is this? Well, because of polymorphism, and the fact that methods can be included in classes, you can not have a unique linker symbol for each function name. Instead this mangled symbol includes enough information to make the symbol unique.

You can retrieve the meaning of this mangled symbol a number of ways. First of all, there is a demangling utility *`c++filt`*:

```
c++filt __Z3barPc
bar(char*)
```

But maybe easier is to use the `-C` flag on *`nm`*

```
$ g++ -c foochar.c && nm -C foochar.o | grep bar
0000000000000000 T bar(char*)
```

### 13.2.2 Extern naming

You can force the compiler to generate names that are intelligible to other languages by

```
#ifdef __cplusplus
  extern"C" {
#endif
  .
  .
  place declarations here
```

```
  .
  .
#ifdef __cplusplus
  }
#endif
```

You again get the same linker symbols as for C, so that the routine can be called from both C and Fortran.

If your main program is in C, you can use the C++ compiler as linker. If the main program is in Fortran, you need to use the Fortran compiler as linker. It is then necessary to link in extra libraries for the C++ system routines. For instance, with the Intel compiler `-lstdc++ -lc` needs to be added to the link line.

The use of `extern` is also needed if you link other languages to a C++ main program. For instance, a Fortran subprogram `foo` should be declared as

```
extern "C" {
void foo_();
}
```

In that case, you again use the C++ compiler as linker.

## 13.3 Strings

Programming languages differ widely in how they handle strings.

- In C, a string is an array of characters; the end of the string is indicated by a null character, that is the ascii character zero, which has an all zero bit pattern. This is called *null termination*.
- In Fortran, a string is an array of characters. The length is maintained in a internal variable, which is passed as a hidden parameter to subroutines.
- In Pascal, a string is an array with an integer denoting the length in the first position. Since only one byte is used for this, strings can not be longer than 255 characters in Pascal.

As you can see, passing strings between different languages is fraught with peril. This situation is made even worse by the fact that passing strings as subroutine arguments is not standard.

Example: the main program in Fortran passes a string

```
Program Fstring
  character(len=5) :: word = "Word"
  call cstring(word)
end Program Fstring
```

and the C routine accepts a character string and its length:

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

```
void cstring_(char *txt,int txtlen) {
  printf("length = %d\n",txtlen);
  printf("<<");
  for (int i=0; i<txtlen; i++)
    printf("%c",txt[i]);
  printf(">>\n");
}
```

which produces:

```
length = 5
<<Word >>
```

To pass a Fortran string to a C program you need to append a null character:

```
call cfunction ('A string'//CHAR(0))
```

Some compilers support extensions to facilitate this, for instance writing

```
DATA forstring /'This is a null-terminated string.'C/
```

Recently, the 'C/Fortran interoperability standard' has provided a systematic solution to this.

## 13.4 Subprogram arguments

In C, you pass a `float` argument to a function if the function needs its value, and `float*` if the function has to modify the value of the variable in the calling environment. Fortran has no such distinction: every variable is passed *by reference*. This has some strange consequences: if you pass a literal value 37 to a subroutine, the compiler will allocate a nameless variable with that value, and pass the address of it, rather than the value[1].

For interfacing Fortran and C routines, this means that a Fortran routine looks to a C program like all its argument are 'star' arguments. Conversely, if you want a C subprogram to be callable from Fortran, all its arguments have to be star-this or that. This means on the one hand that you will sometimes pass a variable by reference that you would like to pass by value.

Worse, it means that C subprograms like

```
void mysub(int **iarray) {
 *iarray = (int*)malloc(8*sizeof(int));
 return;
}
```

can not be called from Fortran. There is a hack to get around this (check out the Fortran77 interface to the Petsc routine `VecGetValues`) and with more cleverness you can use `POINTER` variables for this.

1. With a bit of cleverness and the right compiler, you can have a program that says `print *,7` and prints 8 because of this.

## 13.5 Input/output

Both languages have their own system for handling input/output, and it is not really possible to meet in the middle. Basically, if Fortran routines do I/O, the main program has to be in Fortran. Consequently, it is best to isolate I/O as much as possible, and use C for I/O in mixed language programming.

## 13.6 Python calling C code

Because of its efficiency of computing, C is a logical language to use for the lowest layers of a program. On the other hand, because of its expressiveness, Python is a good candidate for the top layers. It is then a logical thought to want to call C routines from a python program. This is possible using the python *ctypes* module.

1. You write your C code, and compile it to a dynamic library as indicated above;
2. The python code loads the library dynamically, for instance for *libc*:
```
path_libc = ctypes.util.find_library("c")
libc = ctypes.CDLL(path_libc)
libc.printf(b"%s\n", b"Using the C printf function from Python ... ")
```
3. You need to declare what the types are of the C routines in python:
```
test_add = mylib.test_add
test_add.argtypes = [ctypes.c_float, ctypes.c_float]
test_add.restype = ctypes.c_float
test_passing_array = mylib.test_passing_array
test_passing_array.argtypes = [ctypes.POINTER(ctypes.c_int), ctypes.c_int]
test_passing_array.restype = None
```
4. Scalars can be passed simply; arrays need to be constructed:
```
data = (ctypes.c_int * Nelements)(*[x for x in range(numel)])
```

### 13.6.1 Swig

Another way to let C and python interact is through *Swig*.

Let's assume you have C code that you want to use from Python. First of all, you need to supply an interface file for the routines you want to use.

Source file:

```
#include <time.h>
double My_variable = 3.0;

int fact(int n) {
  if (n <= 1) return 1;
  else return n*fact(n-1);
}

int my_mod(int x, int y) {
  return (x%y);
}

char *get_time()
```

```
{
  time_t ltime;
  time(&ltime);
  return ctime(&ltime);
}
```

Interface file:

```
%module example
%{
  /* Put header files here or function declarations like
  extern double My_variable;
  extern int fact(int n);
  extern int my_mod(int x, int y);
  extern char *get_time();
  %}

extern double My_variable;
extern int fact(int n);
extern int my_mod(int x, int y);
extern char *get_time();
```

You now use a combination of Swig and the regular compiler to generate the interface:

```
swig -python example.i
${TACC_CC} -c example.c example_wrap.c \
    -g -fPIC \
    -I${TACC_PYTHON_INC}/python3.9
ld -shared example.o example_wrap.o -o _example.so
```

Testing the generated interface:

### 13.6.2 Boost

Another way to let C and python interact is through the *Boost* library.

Let's start with a C/C++ file that was written for some other purpose, and with no knowledge of Python or interoperability tools:

```
char const* greet()
{
  return "hello, world";
}
```

With it, you should have a `.h` header file with the function signatures.

Next, you write a C++ file that uses the Boost tools:

```
#include <boost/python.hpp>

#include "hello.h"

BOOST_PYTHON_MODULE(hello_ext)
{
  using namespace boost::python;
  def("greet", greet);
}
```

The crucial step is compiling both C/C++ files together into a *dynamic library*:

```
icpc -shared -o hello_ext.so hello_ext.o hello.o \
    -Wl,-rpath,/pythonboost/lib  -L/pythonboost/lib  -lboost_python39 \
    -Wl,-rpath,/python/lib       -L/python/lib       -lpython3
```

You can now import this library in python, giving you access to the C function:

```
import hello_ext
print(hello_ext.greet())
```

# Chapter 14

# Bit operations

In most of this book we consider numbers, such as integer or floating point representations of real numbers, as our lowest building blocks. Sometimes, however, it is necessary to dig deeper and consider the actual representation of such numbers in terms of bits.

Various programming languages have support for bit operations. We will explore the various options. For details on C++ and Fortran, see *Introduction to Scientific Programming book, section 5.2.1* and *Introduction to Scientific Programming book, section 30.7* respectively.

## 14.1 Construction and display

### 14.1.1 C/C++

The built-in possibilities for display are limited:

```
printf("Octal: %o",i);
printf("Hex  : %x",i);
```

gives octal and hexadecimal representation, but there is no *format specifier* for binary. Instead use the following bit of magic:

```
void printBits(size_t const size, void const * const ptr)
{
  unsigned char *b = (unsigned char*) ptr;
  unsigned char byte;
  for (int i=size-1; i>=0; i--) {
      for (int j=7; j>=0; j--) {
          byte = (b[i] >> j) & 1;
          printf("%u", byte);
        }
    }
}
/* ... */
printBits(sizeof(i),&i);
```

### 14.1.2 Python

- The python `int` function converts a string to int. A second argument can indicate what base the string is to be interpreted in:

```
five     = int('101',2)
maxint32 = int('0xffffffff',16)
```

- Function `bin hex` convert an int to a string in base 2,8,16 respectively.
- Since python integers can be of unlimited length, there is a function to determine the bit length (Python version 3.1): `i.bit_length()`.

## 14.2 Bit operations

Boolean operations are usually applied to the boolean datatype of the programming language. Some languages allow you to apply them to actual bits.

| | boolean | bitwise (C) | bitwise (Py) |
|---|---|---|---|
| and | && | & | & |
| or | \|\| | \| | \| |
| not | ! | | ~ |
| xor | | ^ | |

Additionally, there are operations on the bit string as such:

| | |
|---|---|
| left shift | << |
| right shift | >> |

**Exercise** 14.1. Use bit operations to test whether a number is odd or even.

The shift operations are sometimes used as an efficient shorthand for arithmetic. For instance, a left shift by one position corresponds to a multiplication by two.

**Exercise** 14.2. Given an integer $n$, find the largest multiple of 8 that is $\leq n$.

This mechanism is sometimes used to allocate aligned memory. Write a routine

```
aligned_malloc( int Nbytes, int aligned_bits );
```

that allocates `Nbytes` of memory, where the first byte has an address that is a multiple of `aligned_bits`.

# Chapter 15

# LaTeX for scientific documentation

## 15.1 The idea behind LaTeX, some history of TeX

TeX is a typesetting system that dates back to the late 1970s. In those days, graphics terminals where you could design a document layout and immediately view it, the way you can with for instance Microsoft Word, were rare. Instead, TeX uses a two-step workflow, where you first type in your document with formatting instructions in an ascii document, using your favorite text editor. Next, you would invoke the `latex` program, as a sort of compiler, to translate this document to a form that can be printed or viewed.

```
%% edit mydocument.tex
%% latex mydocument
%% # print or view the resulting output
```

The process is comparable to making web pages by typing HTML commands.

This way of working may seem clumsy, but it has some advantages. For instance, the TeX input files are plain ascii, so they can be generated automatically, for instance from a database. Also, you can edit them with whatever your favorite editor happens to be.

Another point in favor of TeX is the fact that the layout is specified by commands that are written in a sort of programming language. This has some important consequences:

- Separation of concerns: when you are writing your document, you do not have to think about layout. You give the 'chapter' command, and the implementation of that command will be decided independently, for instance by you choosing a document style.
- Changing the layout of a finished document is then done by choosing a different realization of the layout commands in the input file: the same 'chapter' command is used, but by choosing a different style the resulting layout is different. This sort of change can be as simple as a one-line change to the document style declaration.
- If you have unusual typesetting needs, it is possible to write new TeX commands for this. For many needs such extensions have in fact already been written; see section 15.4.

The commands in TeX are fairly low level. For this reason, a number of people have written systems on top of TeX that offer powerful features, such as automatic cross-referencing, or generation of a table of contents. The most popular of these systems is LaTeX. Since TeX is an interpreted system, all of its mechanisms are still available to the user, even though LaTeX is loaded on top of it.

### 15.1.1 Installing LaTeX

The easiest way to install LaTeX on your system is by downloading the TeXlive distribution from http://tug.org/texlive. Apple users can also use `fink` or `macports`. Various front-ends to TeX exist, such as TeXshop on the Mac.

### 15.1.2 Running LaTeX

**Purpose.** In this section you will run the LaTeX compiler

Originally, the `latex` compiler would output a device independent file format, named `dvi`, which could then be translated to PostScript or PDF, or directly printed. These days, many people use the `pdflatex` program which directly translates `.tex` files to `.pdf` files. This has the big advantage that the generated PDF files have automatic cross linking and a side panel with table of contents. An illustration is found below.

Let us do a simple example.

```
\documentclass{article}
\begin{document}
Hello world!
\end{document}
```

Figure 15.1: A minimal LaTeX document.

**Exercise** 15.1. Create a text file `minimal.tex` with the content as in figure 15.1. Try the command `pdflatex minimal` or `latex minimal`. Did you get a file `minimal.pdf` in the first case or `minimal.dvi` in the second case? Use a pdf viewer, such as Adobe Reader, or `dvips` respectively to view the output.

*Things to watch out for.* If you make a typo, TeX can be somewhat unfriendly. If you get an error message and TeX is asking for input, typing `x` usually gets you out, or `Ctrl-C`. Some systems allow you to type `e` to go directly into the editor to correct the typo.

## 15.2 A gentle introduction to LaTeX

Here you will get a very brief run-through of LaTeX features. There are various more in-depth tutorials available, such as the one by Oetiker [18].

### 15.2.1 Document structure

Each LaTeX document needs the following lines:

```
\documentclass{ .... } % the dots will be replaced

\begin{document}

\end{document}
```

The 'documentclass' line needs a class name in between the braces; typical values are 'article' or 'book'. Some organizations have their own styles, for instance 'ieeeproc' is for proceedings of the IEEE.

All document text goes between the `\begin{document}` and `\end{document}` lines. (Matched 'begin' and 'end' lines are said to denote an 'environment', in this case the document environment.)

The part before `\begin{document}` is called the 'preamble'. It contains customizations for this particular document. For instance, a command to make the whole document double spaced would go in the preamble. If you are using `pdflatex` to format your document, you want a line

```
\usepackage{hyperref}
```

here.

Have you noticed the following?

- The backslash character is special: it starts a LaTeX command.
- The braces are also special: they have various functions, such as indicating the argument of a command.
- The percent character indicates that everything to the end of the line is a comment.

### 15.2.2 Some simple text

**Purpose.** In this section you will learn some basics of text formatting.

**Exercise** 15.2. Create a file `first.tex` with the content of figure 15.1 in it. Type some text in the preamble, that is, before the `\begin{document}` line and run `pdflatex` on your file.

*Intended outcome.* You should get an error message because you are not allowed to have text in the preamble. Only commands are allowed there; all text has to go after `\begin{document}`.

**Exercise** 15.3. Edit your document: put some text in between the `\begin{document}` and `\end{document}` lines. Let your text have both some long lines that go on for a while, and some short ones. Put superfluous spaces between words, and at the beginning or end of lines. Run `pdflatex` on your document and view the output.

*Intended outcome.* You notice that the white space in your input has been collapsed in the output. TeX has its own notions about what space should look like, and you do not have to concern yourself with this matter.

**Exercise** 15.4. Edit your document again, cutting and pasting the paragraph, but leaving a blank line between the two copies. Paste it a third time, leaving several blank lines. Format, and view the output.

*Intended outcome.* TEX interprets one or more blank lines as the separation between paragraphs.

**Exercise** 15.5. Add `\usepackage{pslatex}` to the preamble and rerun `pdflatex` on your document. What changed in the output?

*Intended outcome.* This should have the effect of changing the typeface from the default to Times Roman.

*Things to watch out for.* Typefaces are notoriously unstandardized. Attempts to use different typefaces may or may not work. Little can be said about this in general.

Add the following line before the first paragraph:

```
\section{This is a section}
```

and a similar line before the second. Format. You see that LATEX automatically numbers the sections, and that it handles indentation different for the first paragraph after a heading.

**Exercise** 15.6. Replace `article` by `artikel3` in the documentclass declaration line and reformat your document. What changed?

*Intended outcome.* There are many documentclasses that implement the same commands as `article` (or another standard style), but that have their own layout. Your document should format without any problem, but get a better looking layout.

*Things to watch out for.* The `artikel3` class is part of most distributions these days, but you can get an error message about an unknown documentclass if it is missing or if your environment is not set up correctly. This depends on your installation. If the file seems missing, download the files from http://tug.org/texmf-dist/tex/latex/ntgclass/ and put them in your current directory; see also section 15.2.9.

### 15.2.3 Math

**Purpose.** In this section you will learn the basics of math typesetting

One of the goals of the original TEX system was to facilitate the setting of mathematics. There are two ways to have math in your document:

- Inline math is part of a paragraph, and is delimited by dollar signs.
- Display math is, as the name implies, displayed by itself.

**Exercise** 15.7. Put `$x+y$` somewhere in a paragraph and format your document. Put `\[x+y\]` somewhere in a paragraph and format.

*Intended outcome.* Formulas between single dollars are included in the paragraph where you declare them. Formulas between `\[...\]` are typeset in a display.

For display equations with a number, use an `equation` environment. Try this.

Here are some common things to do in math. Make sure to try them out.

- Subscripts and superscripts: `$x_i^2$`. If the sub or superscript is more than a single symbol, it needs to be grouped: `$x_{i+1}^{2n}$`. If you need a brace in a formula, use `$\{ \}$`.

- Greek letters and other symbols: `$\alpha\otimes\beta_i$`.
- Combinations of all these `$\int_{t=0}^\infty tdt$`.

**Exercise** 15.8. Take the last example and typeset it as display math. Do you see a difference with inline math?

*Intended outcome.* TEX tries not to include the distance between text lines, even if there is math in a paragraph. For this reason it typesets the bounds on an integral sign differently from display math.

### 15.2.4 Referencing

**Purpose.** In this section you will see TEX's cross referencing mechanism in action.

So far you have not seen LATEX do much that would save you any work. The cross referencing mechanism of LATEX will definitely save you work: any counter that LATEX inserts (such as section numbers) can be referenced by a label. As a result, the reference will always be correct.

Start with an example document that has at least two section headings. After your first section heading, put the command `\label{sec:first}`, and put `\label{sec:other}` after the second section heading. These label commands can go on the same line as the `section` command, or on the next. Now put

```
As we will see in section~\ref{sec:other}.
```

in the paragraph before the second section. (The tilde character denotes a non-breaking space.)

**Exercise** 15.9. Make these edits and format the document. Do you see the warning about an undefined reference? Take a look at the output file. Format the document again, and check the output again. Do you have any new files in your directory?

*Intended outcome.* On a first pass through a document, the TEX compiler will gather all labels with their values in a `.aux` file. The document will display a double question mark for any references that are unknown. In the second pass the correct values will be filled in.

*Things to watch out for.* If after the second pass there are still undefined references, you probably made a typo. If you use the `bibtex` utility for literature references, you will regularly need three passes to get all references resolved correctly.

Above you saw that the `equation` environment gives displayed math with an equation number. You can add a label to this environment to refer to the equation number.

**Exercise** 15.10. Write a formula in an `equation` environment, and add a label. Refer to this label anywhere in the text. Format (twice) and check the output.

*Intended outcome.* The `\label` and `\ref` command are used in the same way for formulas as for section numbers. Note that you must use `\begin/end{equation}` rather than `\[...\]` for the formula.

### 15.2.5 Lists

**Purpose.** In this section you will see the basics of lists.

Bulleted and numbered lists are provided through an environment.

```
\begin{itemize}
\item This is an item;
\item this is one too.
\end{itemize}
\begin{enumerate}
\item This item is numbered;
\item this one is two.
\end{enumerate}
```

**Exercise** 15.11. Add some lists to your document, including nested lists. Inspect the output.

*Intended outcome.* Nested lists will be indented further and the labeling and numbering style changes with the list depth.

**Exercise** 15.12. Add a label to an item in an `enumerate` list and refer to it.

*Intended outcome.* Again, the `\label` and `\ref` commands work as before.

### 15.2.6 Source code and algorithms

As a computer scientist, you will often want to include algorithms in your writings; sometimes even source code.

In this tutorial so far you have seen that some characters have special meaning to LaTeX, and just can not just type them and expect them to show up in the output. Since funny characters appear quite regularly in programming languages, we need a tool for this: the *verbatim mode*.

To display bits of code inside a paragraph, you use the `\verb` command. This command delimits its argument with two identical characters that can not appear in the verbatim text. For instance, the output `if (x%5>0) { ... }` is produced by `\verb+if (x%5>0) { ... }+`. (Exercise: how did the author of this book get that verbatim command in the text?)

For longer stretches of verbatim text, that need to be displayed by themselves you use

```
\begin{verbatim}
stuff
\end{verbatim}
```

Finally, in order to include a whole file as verbatim listing, use .

Verbatim text is one way of displaying algorithms, but there are more elegant solutions. For instance, in this book the following is used:

```
\usepackage[algo2e,noline,noend]{algorithm2e}
```

### 15.2.7 Graphics

Since you can not immediately see the output of what you are typing, sometimes the output may come as a surprise. That is especially so with graphics. LaTeX has no standard way of dealing with graphics, but the following is a common set of commands:

```
\usepackage{graphicx} % this line in the preamble

\includegraphics{myfigure} % in the body of the document
```

The figure can be in any of a number of formats, except that PostScript figures (with extension `.ps` or `.eps`) can not be used if you use pdflatex.

Since your figure is often not the right size, the include line will usually have something like:

```
\includegraphics[scale=.5]{myfigure}
```

A bigger problem is that figures can be too big to fit on the page if they are placed where you declare them. For this reason, they are usually treated as 'floating material'. Here is a typical declaration of a figure:

```
\begin{figure}[ht]
  \includegraphics{myfigure}
  \caption{This is a figure.}
  \label{fig:first}
\end{figure}
```

It contains the following elements:

- The `figure` environment is for 'floating' figures; they can be placed right at the location where they are declared, at the top or bottom of the next page, at the end of the chapter, et cetera.
- The `[ht]` argument of the `\begin{figure}` line states that your figure should be attempted to be placed here; it that does not work, it should go top of the next page. The remaining possible specifications are `b` for placement at the bottom of a page, or `p` for placement on a page by itself. For example

  ```
  \begin{figure}[hbp]
  ```

  declares that the figure has to be placed here if possible, at the bottom of the page if that's not possible, and on a page of its own if it is too big to fit on a page with text.
- A caption to be put under the figure, including a figure number;
- A label so that you can refer to the figure number by its label: `figure~\ref{fig:first}`.
- And of course the figure material. There are various ways to fine-tune the figure placement. For instance

  ```
  \begin{center}
    \includegraphics{myfigure}
  \end{center}
  ```

  gives a centered figure.

### 15.2.8 Bibliography references

The mechanism for citing papers and books in your document is a bit like that for cross referencing. There are labels involved, and there is a `\cite{thatbook}` command that inserts a reference, usually numeric. However, since you are likely to refer to a paper or book in more than one document your write, LaTeX allows you to have a database of literature references in a file by itself, rather than somewhere in your document.

Make a file `mybibliography.bib` with the following content:

```
@article{JoeDoe1985,
author = {Joe Doe},
title = {A framework for bibliography references},
journal = {American Library Assoc. Mag.},
year = {1985}
}
```

In your document `mydocument.tex`, put

```
For details, refer to Doe~\cite{JoeDoe1985} % somewhere in the text

\bibliography{mybibliography} % at the end of the document
\bibliographystyle{plain}
```

Format your document, then type on the commandline

```
bibtex mydocument
```

and format your document two more times. There should now be a bibliography in it, and a correct citation. You will also see that files `mydocument.bbl` and `mydocument.blg` have been created.

### 15.2.9 Environment variables

On Unix systems, TeX investigates the `TEXINPUTS` *environment variable* when it tries to find an include file. Consequently, you can create a directory for your styles and other downloaded include files, and set this variable to the location of that directory. Similarly, the `BIBINPUTS` variable indicates the location of bibliography files for bibtex (section 15.2.8).

## 15.3 A worked out example

The following example `demo.tex` contains many of the elements discussed above.

You also need the file `math.bib`:

The following sequence of commands

```
pdflatex demo
bibtex demo
pdflatex demo
pdflatex demo
```

gives the output of figures 15.2, 15.3.

### 15.3.1 Listings

The 'listings' package is makes it possible to have source code included, with coloring and indentation automatically taken care of.

```
\documentclass{article}

\usepackage[pdftex]{hyperref}
\usepackage{pslatex}


%%%%
%%%% Import the listings package
%%%%
\usepackage{listings,xcolor}

%%%%
%%%% Set a basic code style
%%%% (see documentation for more options)
%%%%
\lstdefinestyle{reviewcode}{
  belowcaptionskip=1\baselineskip, breaklines=true,
  xleftmargin=\parindent, showstringspaces=false,
  basicstyle=\footnotesize\ttfamily,
  keywordstyle=\bfseries\color{blue},
  commentstyle=\color{red!60!black},
  identifierstyle=\slshape\color{black},
  stringstyle=\color{green!60!black}, columns=fullflexible,
  keepspaces=true,tabsize=8,
}
\lstset{style=reviewcode}

\lstset{emph={ %% MPI commands
    MPI_Init,MPI_Initialized,MPI_Finalize,MPI_Finalized,MPI_Abort,
    MPI_Comm_size,MPI_Comm_rank,
    MPI_Send,MPI_Isend,MPI_Rsend,MPI_Irsend,MPI_Ssend,MPI_Issend,
    MPI_Recv,MPI_Irecv,MPI_Mrecv,MPI_Sendrecv,MPI_Sendrecv_replace,
  },emphstyle={\color{red!70!black}\bfseries}
}
\lstset{emph={[2] %% constants
    MPI_COMM_WORLD,MPI_STATUS_IGNORE,MPI_STATUSES_IGNORE,MPI_SUCCESS,
    MPI_INT,MPI_INTEGER,
  },emphstyle={[2]\color{green!40!black}}
}
\lstset{emph={[3] %% types
    MPI_Aint,MPI_Comm,MPI_Count,MPI_Datatype,MPI_Errhandl
  },emphstyle={[3]\color{yellow!30!brown}\bfseries},
}

\begin{document}
\title{SSC 335: listings demo}
\author{Victor Eijkhout}
\date{today}
\maketitle

\section{C examples}

\lstset{language=C}
\begin{lstlisting}
int main() {
  MPI_Init();
  MPI_Comm comm = MPI_COMM_WORLD;
  if (x==y)
    MPI_Send( &x,1,MPI_INT,0,0,comm);
  else
    MPI_Recv( &y,1,MPI_INT,1,1,comm,MPI_STATUS_IGNORE);
  MPI_Finalize();
}
\end{lstlisting}

\section{Fortran examples}

\lstset{language=Fortran}
\begin{lstlisting}
Program myprogram
  Type(MPI_Comm) :: comm = MPI_COMM_WORLD
  call MPI_Init()
  if (x==y) then
    call MPI_Send( x,1,MPI_INTEGER,0,0,comm);
```

# SSC 335: demo

**Victor Eijkhout**

**today**

## 1 This is a section

This is a test document, used in [2]. It contains a discussion in section 2.

**Exercise 1.** Left to the reader.

**Exercise 2.** Also left to the reader, just like in exercise 1

**Theorem 1** *This is cool.*

This is a formula: $a \Leftarrow b$.

$$x_i \leftarrow y_{ij} \cdot x_j^{(k)} \tag{1}$$

Text: $\int_0^1 \sqrt{x} dx$

$$\int_0^1 \sqrt{x} dx$$

## 2 This is another section

| | |
|---|---|
| one | value |
| another | values |

Table 1: This is the only table in my demo

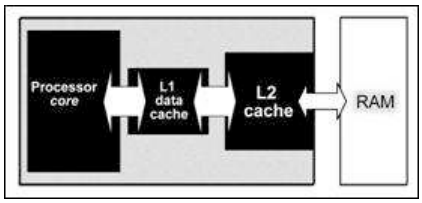



Figure 1: this is the only figure

As I showed in the introductory section 1, in the paper [1], it was shown that equation (1)

- There is an item.





Figure 15.2: First page of LaTeX demo output

- There is another item
  - sub one
  - sub two

1. item one
2. item two
   (a) sub one
   (b) sub two

**Contents**



**List of Figures**



**References**

[1] Loyce M. Adams and Harry F. Jordan. Is SOR color-blind? *SIAM J. Sci. Stat. Comput.*, 7:490–506, 1986.
[2] Victor Eijkhout. Short LaTeX demo. SSC 335, oct 1, 2008.



Figure 15.3: First page of LaTeX demo output

```
  else                                                    \end{lstlisting}
    call MPI_Recv( y,1,MPI_INTEGER,1,1,comm,MPI_STATUS_IGNORE)
  end if                                                  \end{document}
  call MPI_Finalize()
End Program myprogram
```

See the output in figure 15.4.

### 15.3.2 Native graphing

You have seen how to include graphics files, but it is also possible to let LaTeX do the drawing. For this, there is the *tikz* package. Here we show another package *pgfplots* that uses `tikz` to draw numerical plots.

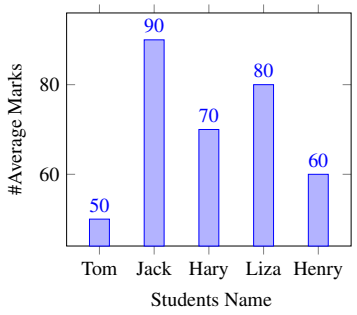
```
\documentclass{artikel3}

\usepackage[pdftex]{hyperref}
\usepackage{pslatex}

\usepackage{wrapfig}
\usepackage{pgfplots}
\pgfplotsset{width=6.6cm,compat=1.7}

\usepackage{geometry}
\addtolength{\textwidth}{.75in}
\addtolength{\textheight}{.75in}

\begin{document}
\title{SSC 335: barchart demo}
\author{Victor Eijkhout}
\date{today}
\maketitle

\section{Two graphs}

\begin{wrapfigure}{l}{2in}
  \hrule width 3in height 0pt
  \begin{tikzpicture}
    \begin{axis}
      [
        ybar,
        enlargelimits=0.15,
        ylabel={\#Average Marks},
        xlabel={\ Students Name},
        symbolic x coords={Tom, Jack, Hary, Liza, Henry},
        xtick=data,
        nodes near coords,
        nodes near coords align={vertical},
      ]
      \addplot coordinates {(Tom,50) (Jack,90) (Hary,70)
    \end{axis}
  \end{tikzpicture}
\end{wrapfigure}
Lorem ipsum dolor sit amet, consectetur adipiscing elit,

\begin{wrapfigure}{r}{2in}
  \hrule width 3in height 0pt % kludge: picture is not w
  \begin{tikzpicture}
    \begin{axis}
      [
        ybar,
        enlargelimits=0.15,
        legend style={at={(0.4,-0.25)},anchor=north,lege
        ylabel={\#Annual Growth Percentage},
        symbolic x coords={2016, 2017, 2018},
        xtick=data,
        nodes near coords,
        nodes near coords align={vertical},
      ]
      \addplot coordinates {(2016, 75) (2017, 78) (2018,
      \addplot coordinates {(2016, 70) (2017, 63) (2018,
      \addplot coordinates {(2016, 61) (2017, 55) (2018,
      \legend{Wheat, Tea, Rice}
    \end{axis}
  \end{tikzpicture}
\end{wrapfigure}
Sem nulla pharetra diam sit amet. Vel pharetra vel turpi
\end{document}
```

# SSC 335: barchart demo

**Victor Eijkhout**

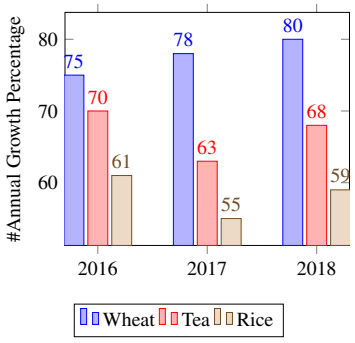
**today**

## 1 Two graphs




Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Pharetra massa massa ultricies mi quis hendrerit. Tempor nec feugiat nisl pretium fusce id velit ut tortor. Eget nulla facilisi etiam dignissim diam quis enim. Cursus sit amet dictum sit amet justo donec. Tortor consequat id porta nibh venenatis cras sed felis eget. Senectus et netus et malesuada fames ac turpis egestas integer. Ultricies mi quis hendrerit dolor magna eget est. A iaculis at erat pellentesque adipiscing. Sagittis orci a scelerisque purus. Quisque non tellus orci ac. Nisl nunc mi ipsum faucibus. Vivamus at augue eget arcu dictum varius duis. Maecenas ultricies mi eget mauris pharetra et ultrices neque ornare. Pulvinar neque laoreet suspendisse interdum consectetur. Nunc id cursus metus aliquam eleifend mi. Tristique sollicitudin nibh sit amet commodo nulla. Massa tincidunt nunc pulvinar sapien et ligula ullamcorper malesuada. Justo laoreet sit amet cursus sit. Laoreet id donec ultrices tincidunt arcu non sodales.

Sem nulla pharetra diam sit amet. Vel pharetra vel turpis nunc eget. Vulputate dignissim suspendisse in est ante in nibh mauris cursus. Sem viverra aliquet eget sit amet tellus cras. Rhoncus aenean vel elit scelerisque mauris pellentesque pulvinar pellentesque. Fusce ut placerat orci nulla pellentesque. Vel risus commodo viverra maecenas accumsan lacus vel facilisis volutpat. Enim ut tellus elementum sagittis vitae et. In nibh mauris cursus mattis molestie. Curabitur gravida arcu ac tortor dignissim convallis aenean et tortor. Mauris commodo quis imperdiet massa.



## 15.4 Where to take it from here

This tutorial touched only briefly on some essentials of TEX and LATEX. You can find longer intros online [18], or read a book [12, 11, 16]. Macro packages and other software can be found on the Comprehensive TEX Archive `http://www.ctan.org`. For questions you can go to the newsgroup `comp.text.tex`, but the most common ones can often already be found on web sites [21].

## 15.5 Review questions

**Exercise** 15.13. Write a one or two page document about your field of study. Show that you have mastered the following constructs:

- formulas, including labels and referencing;
- including a figure;
- using bibliography references;
- construction of nested lists.

# SSC 335: listings demo

Victor Eijkhout

today

## 1 C examples

```c
int main() {
  MPI_Init();
  MPI_Comm comm = MPI_COMM_WORLD;
  if (x==y)
    MPI_Send( &x,1,MPI_INT,0,0,comm);
  else
    MPI_Recv( &y,1,MPI_INT,1,1,comm,MPI_STATUS_IGNORE);
  MPI_Finalize();
}
```

## 2 Fortran examples

```fortran
Program myprogram
  Type(MPI_Comm) :: comm = MPI_COMM_WORLD
  call MPI_Init()
  if (.not. x==y ) then
    call MPI_Send( x,1,MPI_INTEGER,0,0,comm);
  else
    call MPI_Recv( y,1,MPI_INTEGER,1,1,comm,MPI_STATUS_IGNORE)
  end if
  call MPI_Finalize()
End Program myprogram
```

1

Figure 15.4: Output of the listings demo

# Chapter 16

# Profiling and benchmarking

Much of the teaching in this book is geared towards enabling you to write fast code, whether this is through the choice of the right method, or through optimal coding of a method. Consequently, you sometimes want to measure just *how fast* your code is. If you have a simulation that runs for many hours, you'd think just looking on the clock would be enough measurement. However, as you wonder whether your code could be faster than it is, you need more detailed measurements. This tutorial will teach you some ways to measure the behavior of your code in more or less detail.

Here we will discuss

- timers: ways of measuring the execution time (and sometimes other measurements) of a particular piece of code, and
- profiling tools: ways of measuring how much time each piece of code, typically a subroutine, takes during a specific run.

## 16.1 Timers

There are various ways of timing your code, but mostly they come down to calling a *timer* routine twice that tells you the clock values:

```
tstart = clockticks()
....
tend = clockticks()
runtime = (tend-tstart)/ticks_per_sec
```

Many systems have their own timers:

- MPI see section *Parallel Programming book, section 15.6.1*;
- OpenMP see section *Parallel Programming book, section 28.2*;
- PETSc see section *Parallel Programming book, section 38.4*.

### 16.1.1 Fortran

For instance, in *Fortran* there is the `system_clock` routine:

```
implicit none
INTEGER :: rate, tstart, tstop
REAL    :: time
real    :: a
integer :: i

CALL SYSTEM_CLOCK(COUNT_RATE = rate)
if (rate==0) then
   print *,"No clock available"
   stop
else
   print *,"Clock frequency:",rate
end if
CALL SYSTEM_CLOCK(COUNT = tstart)
a = 5
do i=1,1000000000
   a = sqrt(a)
end do
CALL SYSTEM_CLOCK(COUNT = tstop)
time = REAL( ( tstop - tstart )/ rate )
print *,a,tstart,tstop,time
end
```

with output

```
 Clock frequency:       10000
   1.000000       813802544   813826097   2.000000
```

### 16.1.2 C

In *C* there is the `clock` function: with output

```
clock resolution: 1000000
res: 1.000000e+00
start/stop: 0.000000e+00,2.310000e+00
Time: 2.310000e+00
```

Do you see a difference between the Fortran and C approaches? Hint: what happens in both cases when the execution time becomes long? At what point do you run into trouble?

### 16.1.3 C++

While C routines are available in *C++*, there is also a new *chrono* library that can do many things, including handling different time formats.

```
std::chrono::system_clock::time_point start_time;
start_time = std::chrono::system_clock::now();
// ... code ...
```

```
auto duration =
    std::chrono::system_clock::now()-start_time;
auto millisec_duration =
    std::chrono::duration_cast<std::chrono::milliseconds>(duration);
std::cout << "Time in milli seconds: "
          << .001 * millisec_duration.count() << endl;
```

For more details, see *Introduction to Scientific Programming book, section 24.8*.

### 16.1.4 System utilities

There are unix system calls that can be used for timing: *getrusage*

```
#include <sys/resource.h>
double time00(void)
{
   struct rusage ruse;
   getrusage(RUSAGE_SELF, &ruse);
   return( (double)(ruse.ru_utime.tv_sec+ruse.ru_utime.tv_usec
    / 1000000.0) );
}
```

and *gettimeofday*

```
#include <sys/time.h>
double time00(void)
{
   struct timeval tp;
   gettimeofday(&tp, NULL);
   return( (double) (tp.tv_sec + tp.tv_usec/1000000.0) ); /* wall
}
```

These timers have the advantage that they can distinguish between user time and system time, that is, exclusively timing program execution or giving *wallclock time* including all system activities.

### 16.1.5 Accurate counters

The timers in the previous section had a resolution of at best a millisecond, which corresponds to several thousand cycles on a modern CPU. For more accurate counting it is typically necessary to use assembly language, such as the Intel *RDTSC* (ReaD Time Stamp Counter) instruction http://developer.intel.com/drg/pentiumII/appnotes/RDTSCPM1.HTM.

```
static inline void microtime(unsigned *lo, unsigned *hi)
{
  __asm __volatile (
        ".byte 0x0f; .byte 0x31   # RDTSC instruction
         movl    %%edx,%0          # High order 32 bits
         movl    %%eax,%1          # Low order 32 bits"
                 : "=g" (*hi), "=g" (*lo) :: "eax", "edx");
}
```

However, this approach of using processor-specific timers is not portable. For this reason, the *PAPI* package (http://icl.cs.utk.edu/papi/) provides a uniform interface to *hardware counters*. You can see this package in action in the codes in appendix HPC book, section 31.

In addition to timing, hardware counters can give you information about such things as cache misses and instruction counters. A processor typically has only a limited number of counters, but they can be assigned to various tasks. Additionally, PAPI has the concept of *derived metrics*.

## 16.2 Parallel timing

Timing parallel operations is fraught with peril, as processes or threads can interact with each other. This means that you may be measuring the wait time induced by synchronization. Sometimes that is actually what you want, as in the case of a *ping-pong* operation; section *Parallel Programming book, section 4.1.1*.

Other times, this is not what you want. Consider the code

```
if (procno==0)
  do_big_setup();
t = timer();
mpi_some_collective();
duration = timer() - t;
```

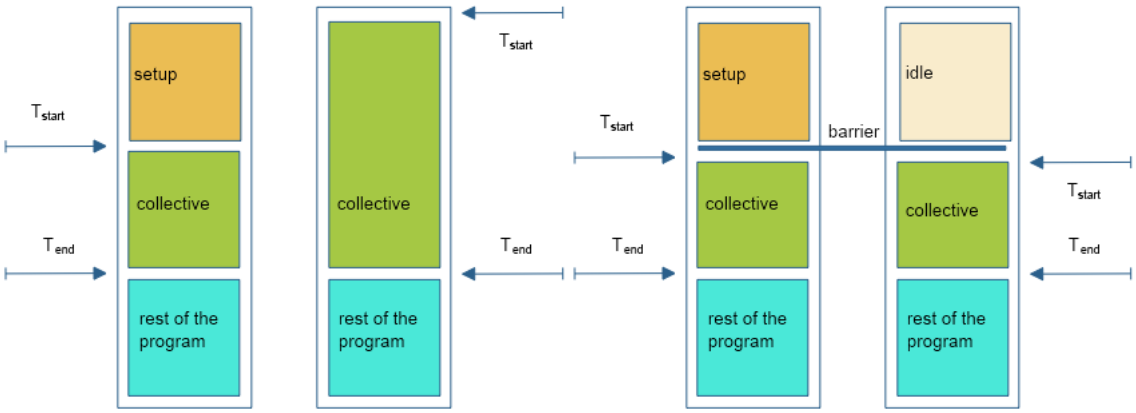



Figure 16.1: Timing a parallel code without and with barrier

Figure 16.1 illustrates this:

- in the naive scenario, processes other than zero start the collective immediately, but process zero first does the setup;
- all processes presumably finish more or less together.

On the non-zero processes we now get a time measurement, which we intended to be just the collective operation, that includes the setup time of process zero.

The solution is to put a barrier around the section that you want to time.

```
Barrier();
tstart = Wtime();
Barrier();
duration = Wtime()-tstart;
```

See also figure 16.1.

### 16.2.1 Parallel timers in MPI and OpenMP

Many packages have their own timers. For instance for *MPI*

```
double MPI_Wtime(void);
double MPI_Wtick(void);
```

See *Parallel Programming book, section 15.6.1*.

For *OpenMP*

```
double omp_get_wtime()
double omp_get_wtick()
```

See *Parallel Programming book, section 28.2*.

## 16.3 Profiling tools

Profiling tools will give you the time spent in various events in the program, typically functions and subroutines, or parts of the code that you have declared as such. The tool will then report how many time the event occurred, total and average time spent, et cetera.

Here we discuss two simple tools:

- `gprof`, which requires instrumentation, and
- `perf`, which doesn't.
- Intel VTune.

The TAU tool, discussed in section 17 for the purposes of tracing, also has profiling capabilities, presented in a nice graphic way. Finally, we mention that the *PETSc* library allows you to define your own timers and events.

### 16.3.1 gprof

The profiler of the *GNU* compiler, `gprof` requires recomplication and linking with an extra flag:

```
% gcc  -g -pg -c ./srcFile.c
% gcc  -g -pg -o MyProgram ./srcFile.o
```

The program is then run by itself:

```
% ./MyProgram
```

leaving behind a file *gmon.out*. This can be post-processed and displayed:

```
% gprof    ./exeFile gmon.out > profile.txt
% gprof -l ./exeFile gmon.out > profile_line.txt
% gprof -A ./exeFile gmon.out > profile_anotated.txt
```

### 16.3.2 perf

Coming with most Unix distributions, *perf* does not require any instrumentation.

Run:

```
perf record myprogram myoptions
perf record --call-graph fp myprogram myoptions
```

This gathers event information into a file `perf.data`, or do

```
perf record -o myoutputfile myprogram myoptions
```

for a custom output file.

Post-process and display:

```
perf report
perf report --demangle ## for C++
perf report -i myoutputfile
```

The display may be interactive; the following gives a pure ascii display, limiting to events amount to more than one percent, and printing out only the columns of percentage and routine name:

```
perf report --stdio \
    --percent-limit=1 \
    --fields=Overhead,Symbol
```

Example:

```
+   14.15%     4.07%  fsm.exe  fsm.exe              [.] std::vector<richdem::dephier::
    Depression<double>, std::allocator<richdem::dephier::Depression<double> > >::at
+    8.92%     4.58%  fsm.exe  fsm.exe              [.] std::vector<richdem::dephier::
    Depression<double>, std::allocator<richdem::dephier::Depression<double> > >::
    _M_range_check
```

This shows that 14% of the time is spent in indexing with `at`, and that more than half of that went into the range checking.

### 16.3.3 Intel VTune

The *Intel VTune* profiler also needs no instrumentation.

```
vtune -collect hotspots yourprogram options
## result are in a directory: r000hs
vtune -report hotspots -r r000hs
```

For graphical output you can use *vtune-gui*. Rather than analyzing results, this lets you set up, run, and analyze an application.

### 16.3.4 MPI profiling

The MPI library has been designed to make it easy to profile. See *Parallel Programming book, section 15.6*.

## 16.4 Tracing

In profiling we are only concerned with aggregate information: how many times a routine was called, and with what total/average/min/max runtime. However sometimes we want to know about the exact timing of events. This is especially relevant in a parallel context when we care about *load unbalance* and *idle time*.

Tools such as Vampyr can collect trace information about events and in particular messages, and render them in displays such as figure 16.2.

Figure 16.2: A Vampyr timeline diagram of a parallel process.

# Chapter 17

# TAU

The TAU tool [20] (see http://www.cs.uoregon.edu/research/tau/home.php for the official documentation) uses *instrumentation* to profile and trace your code. That is, it adds profiling and trace calls to your code. You can then inspect the output after the run.

Profiling is the gathering and displaying of bulk statistics, for instance showing you which routines take the most time, or whether communication takes a large portion of your runtime. When you get concerned about performance, a good profiling tool is indispensible.

Tracing is the construction and displaying of time-dependent information on your program run, for instance showing you if one process lags behind others. For understanding a program's behaviour, and the reasons behind profiling statistics, a tracing tool can be very insightful.

## 17.1 Usage modes

There are two ways to instrument your code:

- You can use *dynamic instrumentation*, where TAU adds the measurement facility at runtime:
  ```
  # original commandline:
  % mpicxx wave2d.cpp -o wave2d
  # with TAU dynamic instrumentation:
  % mpirun -np 12 tau_exec ./wave2d 500 500 3 4 5
  ```
- You can have the instrumentation added at compile time. For this, you need to let TAU take over the compilation in some sense.
  1. TAU has its own makefiles. The names and locations depend on your installation, but typically it will be something like
     ```
     export TAU_MAKEFILE=$TAU_HOME/lib/Makefile.tau-mpi-pdt
     ```
  2. Now you can invoke the TAU compilers `tau_cc,sh`, `tau_cxx.sh`, `tau_f90.sh`.

When you run your program you need to tell TAU what to do:

```
export TAU_TRACE=1
export TAU_PROFILE=1
export TRACEDIR=/some/dir
export PROFILEDIR=/some/dir
```

In order to generate trace plots you need to convert TAU output:

```
cd /some/dir # where the trace and profile output went
tau_treemerge.pl
tau2slog2 tau.trc tau.edf -o yourrun.slog2
```

The `slog2` file can be displayed with *jumpshot*.

## 17.2 Instrumentation

Unlike such tools as *VTune* which profile your binary as-is, TAU can work by adding *instrumentation* to your code: in effect it is a source-to-source translator that takes your code and turns it into one that generates run-time statistics.

This instrumentation is largely done for you; you mostly need to recompile your code with a script that does the source-to-source translation, and subsequently compiles that instrumented code. You could for instance have the following in your makefile:

```
ifdef TACC_TAU_DIR
  CC = tau_cc.sh
else
  CC = mpicc
endif

% : %.c
<TAB>${CC} -o $@ $^
```

If TAU is to be used (which we detect here by checking for the environment variable *TACC_TAU_DIR*), we define the *CC* variable as one of the TAU compilation scripts; otherwise we set it to a regular MPI compiler.

*Fortran note.* Cpp includes If your source contains

```
#include "something.h"
```

directives, add the option

```
-optPreProcess
```

to the TAU compiler.

**Remark 18** *The* PETSc *library can be compiled with TAU instrumentation enabled by adding the* `--with-perfstubs-tau=1` *option at configuration time.*

To use *TAU* on *TACC* resources, do `module load tau`.

## 17.3 Running

You can now run your instrumented code; trace/profile output will be written to file if environment variables *TAU_PROFILE* and/or *TAU_TRACE* are set:

```
export TAU_PROFILE=1
export TAU_TRACE=1
```

A TAU run can generate many files: typically at least one per process. It is therefore advisabe to create a directory for your tracing and profiling information. You declare them to TAU by setting the environment variables *PROFILEDIR* and *TRACEDIR*.

```
mkdir tau_trace
mkdir tau_profile
export PROFILEDIR=tau_profile
export TRACEDIR=tau_trace
```

The actual program invocation is then unchanged:

```
mpirun -np 26 myprogram
```

*TACC note.* At TACC, use *ibrun* without a processor count; the count is derived from the queue submission parameters.

While this example uses two separate directories, there is no harm in using the same for both.

## 17.4 Output

The tracing/profiling information is spread over many files, and hard to read as such. Therefore, you need some further programs to consolidate and display the information.

You view profiling information with *paraprof*

```
paraprof tau_profile
```

Viewing the traces takes a few steps:

```
cd tau_trace
rm -f tau.trc tau.edf align.trc align.edf
tau_treemerge.pl
```

```
tau_timecorrect tau.trc tau.edf align.trc align.edf
tau2slog2 align.trc align.edf -o yourprogram.slog2
```

If you skip the *tau_timecorrect* step, you can generate the slog2 file by:

```
tau2slog2 tau.trc tau.edf -o yourprogram.slog2
```

The slog2 file can be viewed with *jumpshot*:

```
jumpshot yourprogram.slog2
```

## 17.5 Without instrumentation

Event-based sampling on uninstrumented code:

```
tau_exec -ebs yourprogram
```

The resulting .trc file can be viewed with *paraprof*.

## 17.6 Examples

### 17.6.1 Bucket brigade

Let's consider a *bucket brigade* implementation of a broadcast: each process sends its data to the next higher rank.

```
int sendto =
    ( procno<nprocs-1 ? procno+1 : MPI_PROC_NULL )
    ;
int recvfrom =
    ( procno>0 ? procno-1 : MPI_PROC_NULL )
    ;

MPI_Recv( leftdata,1,MPI_DOUBLE,recvfrom,0,comm,MPI_STATUS_IGNORE);
myvalue = leftdata
MPI_Send( myvalue,1,MPI_DOUBLE,sendto,0,comm);
```

We implement the bucket brigade with blocking sends and receives: each process waits to receive from its predecessor, before sending to its successor.

```
// bucketblock.c
if (procno>0)
  MPI_Recv(leftdata, N, MPI_DOUBLE,recvfrom,0, comm, MPI_STATUS_IGNORE);
for (int i=0; i<N; i++)
  myvalue[i] = (procno+1)*(procno+1) + leftdata[i];
if (procno<nprocs-1)
  MPI_Send(myvalue,N, MPI_DOUBLE,sendto,0, comm);
```

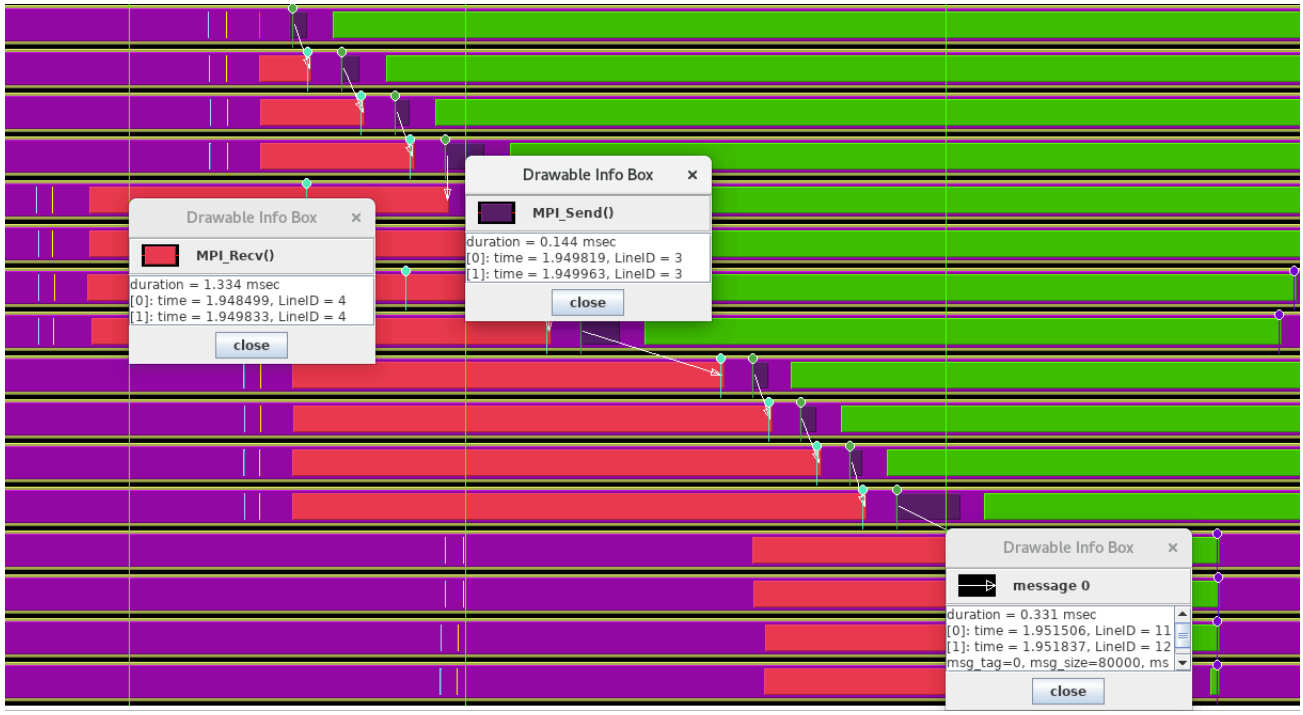



Figure 17.1: Trace of a bucket brigade broadcast

The TAU trace of this is in figure 17.1, using 4 nodes of 4 ranks each. We see that the processes within each node are fairly well synchronized, but there is less synchronization between the nodes. However, the bucket brigade then imposes its own synchronization on the processes because each has to wait for its predecessor, no matter if it posted the receive operation early.

Next, we introduce pipelining into this operation: each send is broken up into parts, and these parts are sent and received with non-blocking calls.

```
// bucketpipenonblock.c
MPI_Request rrequests[PARTS];
for (int ipart=0; ipart<PARTS; ipart++) {
  MPI_Irecv
    (
     leftdata+partition_starts[ipart],partition_sizes[ipart],
     MPI_DOUBLE,recvfrom,ipart,comm,rrequests+ipart);
}
```

The TAU trace is in figure 17.2.

### 17.6.2 Butterfly exchange

The NAS Parallel Benchmark suite [17] contains a Conjugate Gradients (CG) implementation that spells out its all-reduce operations as a *butterfly exchange*.

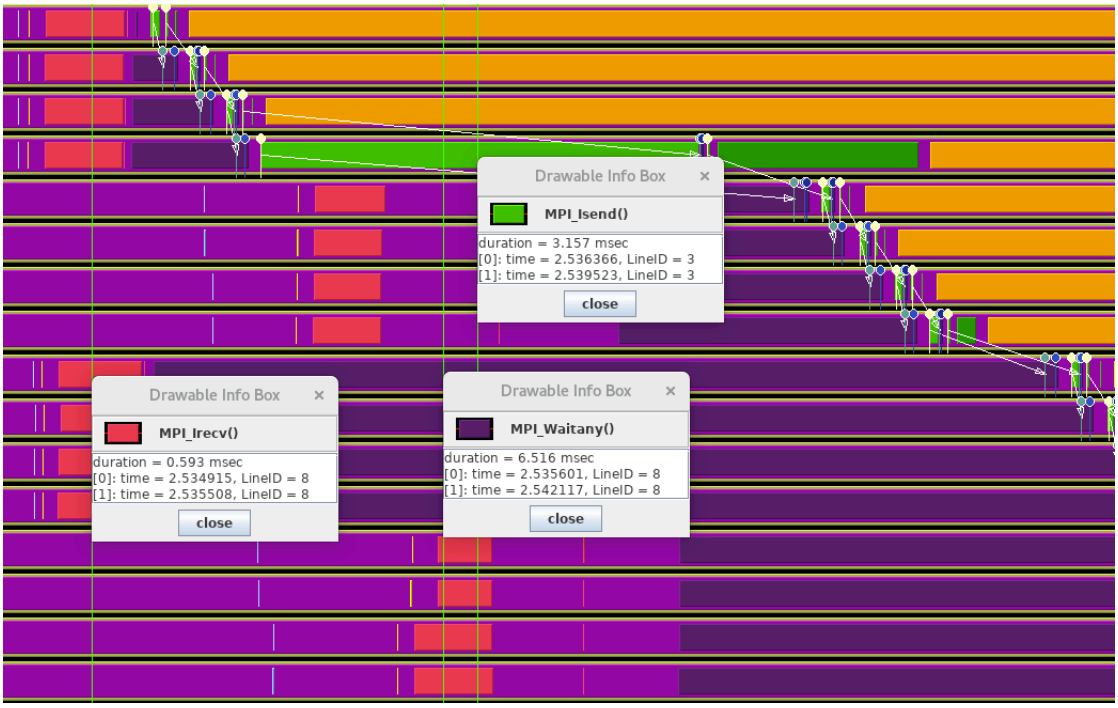



Figure 17.2: Trace of a pipelined bucket brigade broadcast

```
!! cgb.f
         do i = 1, l2npcols
            call mpi_irecv( d,
     >                      1,
     >                      dp_type,
     >                      reduce_exch_proc(i),
     >                      i,
     >                      mpi_comm_world,
     >                      request,
     >                      ierr )
            call mpi_send(  sum,
     >                      1,
     >                      dp_type,
     >                      reduce_exch_proc(i),
     >                      i,
     >                      mpi_comm_world,
     >                      ierr )

            call mpi_wait( request, status, ierr )

            sum = sum + d
         enddo
```

We recognize this structure in the TAU trace: figure 17.3. Upon closer examination, we see how this particular algorithm induces a lot of wait time. Figures 17.5 and 17.6 show a whole cascade of processes

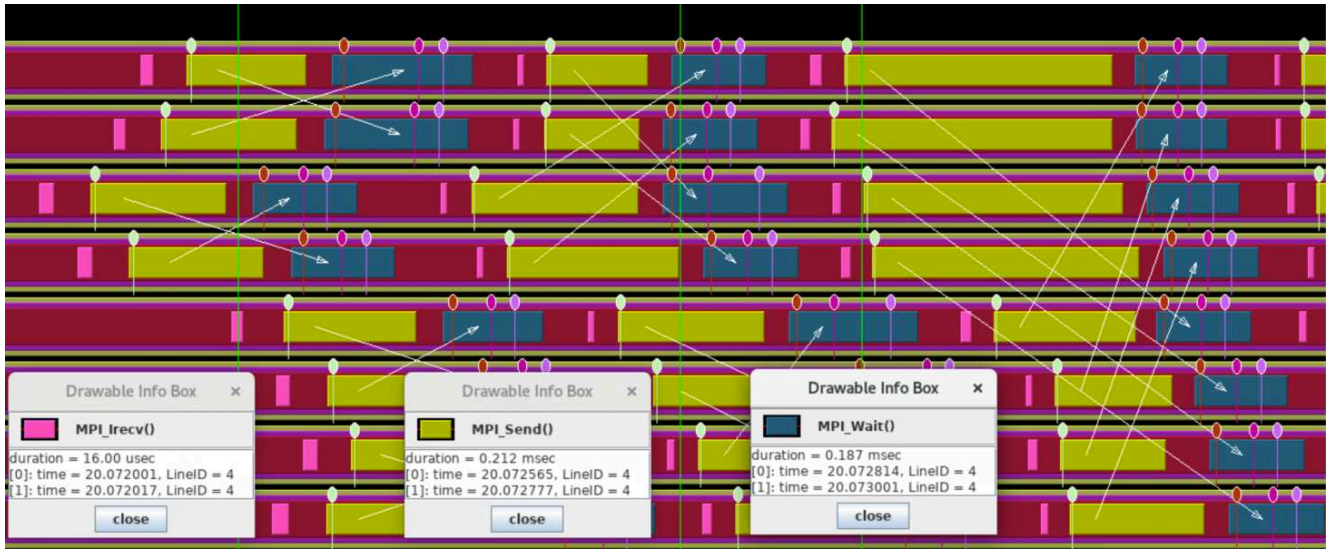



Figure 17.3: Trace of a butterfly exchange

waiting for each other.

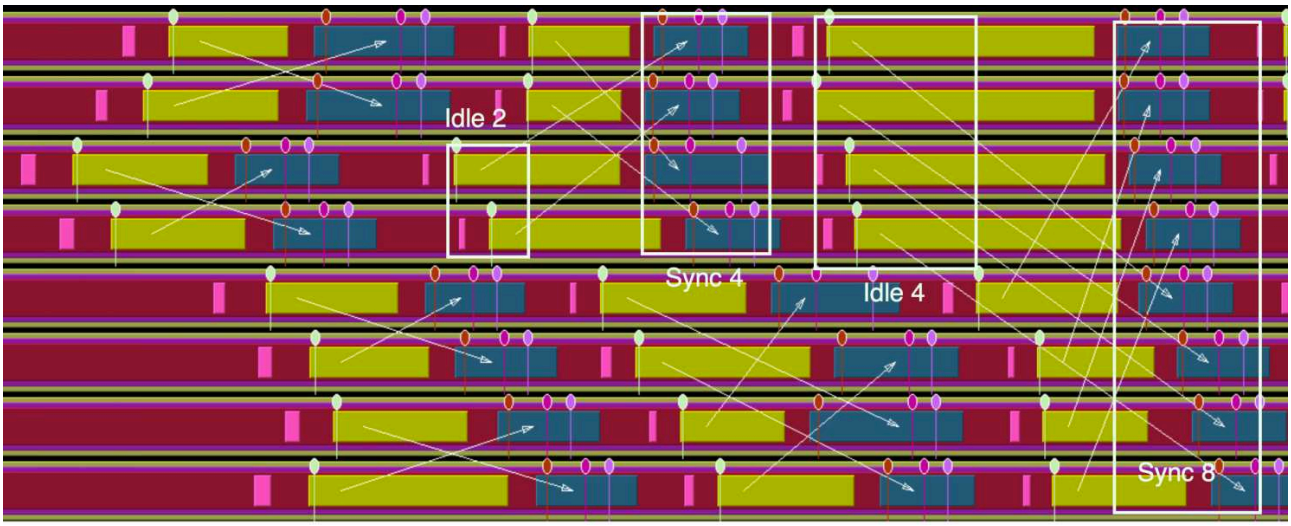



Figure 17.4: Trace of a butterfly exchange

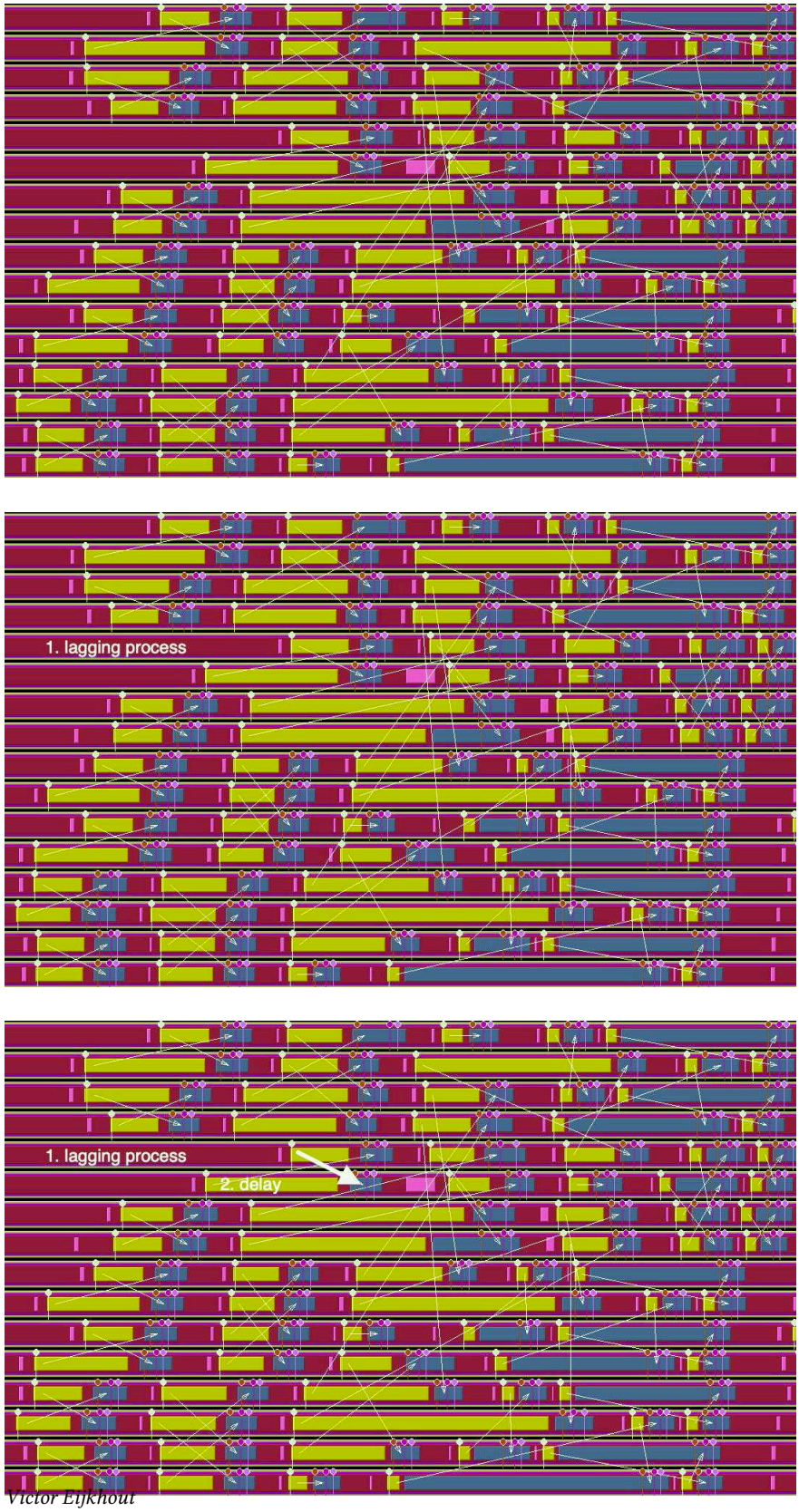



Figure 17.5: Four stages of processes waiting caused by a single lagging process

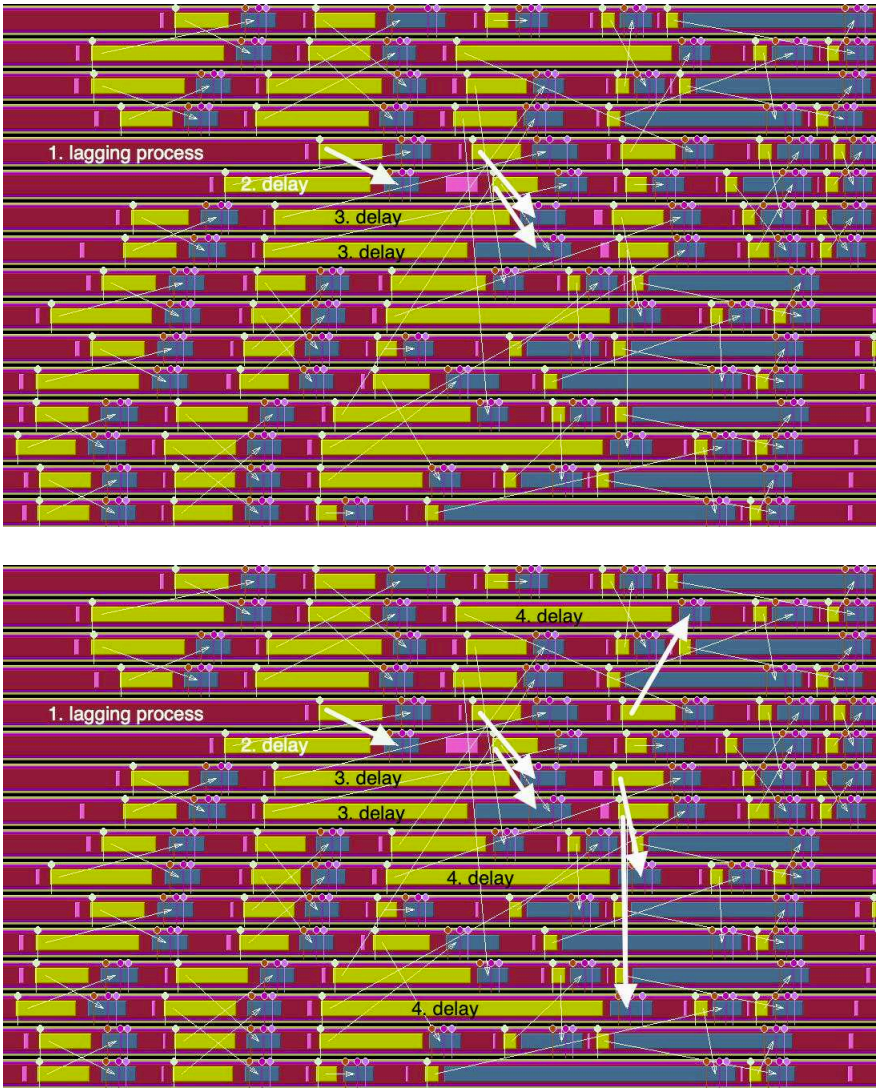



Figure 17.6: Four stages of processes waiting caused by a single lagging process

# Chapter 18

# SLURM

Supercomputer *clusters* can have a large number of *nodes*, but not enough to let all their users run simultaneously, and at the scale that they want. Therefore, users are asked to submit *jobs*, which may start executing immediately, or may have to wait until resources are available.

The decision when to run a job, and what resources to give it, is not done by a human operator, but by software called a *batch system*. (The *Stampede* cluster at *TACC* ran close to 10 million jobs over its lifetime, which corresponds to starting a job every 20 seconds.)

This tutorial will cover the basics of such systems, and in particular Simple Linux Utility for Resource Management (SLURM).

## 18.1 Cluster structure

A supercomputer cluster usually has two types of nodes:

- *login node*, and
- *compute node*.

When you make an *ssh connection* to a cluster, you are connecting to a login node. The number of login nodes is small, typically less than half a dozen.

**Exercise** 18.1. Connect to your favourite cluster. How many people are on that login node? If you disconnect and reconnect, do you find yourself on the same login node?

Compute nodes are where your jobs are run. Different clusters have different structures here:

- Compute nodes can be shared between users, or they can be assigned exclusively.
  - Sharing makes sense if user jobs have less parallelisn than the core count of a node.
  - ... on the other hand, it means that users sharing a node can interfere with each other's jobs, with one job using up memory or bandwidth that the other job needs.
  - With exclusive nodes, a job has access to all the memory and all the bandwidth of that node.
- Clusters can homogeneous, having the same processor type on each compute node, or they can have more than one processor type. For instance, the TACC *Stampede2* cluster has *Intel Knights-landing* and *Intel Skylake* nodes.
- Often, clusters have a number of 'large memory' nodes, on the order of a Terabyte of memory or more. Because of the cost of such hardware, there is usually only a small number of these nodes.

## 18.2 Queues

Jobs often can not start immediately, because not enough resources are available, or because other jobs may have higher priority (see section 18.7). It is thus typical for a job to be put on a *queue*, scheduled, and started, by a batch system such as SLURM.

Batch systems do not put all jobs in one big pool: jobs are submitted to any of a number of queues, that are all scheduled separately.

Queues can differ in the following ways:

- If a cluster has different processor types, those are typically in different queues. Also, there may be separate queues for the nodes that have a Graphics Processing Unit (GPU) attched. Having multiple queues means you have to decide what processor type you want your job to run on, even if your executable is binary compatible with all of them.
- There can be 'development' queues, which have restrictive limits on runtime and node count, but where jobs typically start faster.
- Some clusters have 'premium' queues, which have a higher charge rate, but offer higher priority.
- 'Large memory nodes' are typically also in a queue of their own.
- There can be further queues for jobs with large resource demands, such as large core counts, or longer-than-normal runtimes.

For slurm, the `sinfo` command can tell you much about the queues.

```
# what queues are there?
sinfo -o "%P"
# what queues are there, and what is their status?
sinfo -o "%20P %.5a"
```

**Exercise** 18.2. Enter these commands. How many queues are there? Are they all operational at the moment?

### 18.2.1 Queue limits

Queues have limits on

- the runtime of a job;
- the node count of a job; or
- how many jobs a user can have in that queue.

## 18.3 Job running

There are two main ways of starting a job on a cluster that is managed by slurm. You can start a program run synchronously with `srun`, but this may hang until resources are available. In this section, therefore, we focus on asynchronously executing your program by submitting a job with `sbatch`.

### 18.3.1 The job submission cycle

In order to run a *batch job*, you need to write a *job script*, or *batch script*. This script describes what program you will run, where its inputs and outputs are located, how many processes it can use, and how long it will run.

In its simplest form, you submit your script without further parameters:

```
sbatch yourscript
```

All options regarding the job run are contained in the script file, as we will now discuss.

As a result of your job submission you get a job id. After submission you can queury your job with *squeue*:

```
squeue -j 123456
```

or queury all your jobs:

```
squeue -u yourname
```

The *squeue* command reports various aspects of your job, such as its status (typically pending or running); and if it is running, the queue (or 'partition') where it runs, its elapsed time, and the actual nodes where it runs.

```
squeue -j 5807991
  JOBID   PARTITION     NAME     USER ST   TIME  NODES NODELIST(REASON)
5807991 development packingt eijkhout  R   0:04      2 c456-[012,034]
```

If you discover errors in your script after submitting it, including when it has started running, you can cancel your job with *scancel*:

```
scancel 1234567
```

## 18.4 The script file

A job script looks like an executable shell script:

- It has an 'interpreter' line such as
  ```
  #!/bin/bash
  ```
  at the top, and
- it contains ordinary unix commands, including
- the (parallel) startup of you program:
  ```
  # sequential program:
  ./yourprogram youroptions
  # parallel program, general:
  mpiexec -n 123 parallelprogram options
  ```

```
# parallel program, TACC:
ibrun parallelprogram options
```

- ... and then it has many options specifying the parallel run.

### 18.4.1 sbatch options

In addition to the regular unix commands and the interpreter line, your script has a number of SLURM directives, each starting with `#SBATCH`. (This makes them comments to the shell interpreter, so a batch script is actually a legal shell script.)

Directives have the form

```
#SBATCH -option value
```

Common options (except parallelism related options which are discussed in section 18.5) are:

- `-J`: the jobname. This will be displayed when you call *squeue*.
- `-o`: name of the output file. This will contain all the stdout output of the script.
- `-e`: name of the error file. This will contain all the stderr output of the script, as well as slurm error messages.
  It can be a good idea to make the output and error file unique per job. To this purpose, the macro `%j` is available, which at execution time expands to the job number. You will then get an output file with a name such as `myjob.o2384737`.
- `-p`: the *partition* or queue. See above.
- `-t hh:mm:ss`: the maximum running time. If your job exceeds this, it will get *cancelled*. Two considerations:
  1. You can not specify a duration here that is longer than the queue limit.
  2. The shorter your job, the more likely it is to get scheduled sooner rather than later.
- `-w c452-[101-104,111-112,115]` specific nodes to place the job.
- `-A`: the name of the account to which your job should be billed.
- `--mail-user=you@where` Slurm can notify you when a job starts or ends. You may for instance want to connect to a job when it starts (to run *top*), or inspect the results when it's done, but not sit and stare at your terminal all day. The action of which you want to be notified is specified with (among others) `--mail-type=begin/end/fail/all`
- `--dependency=after:123467` indicates that this job is to start after jobs `1234567` finished. Use `afterok` to start only if that job successfully finished. (See https://cvw.cac.cornell.edu/slurm/submission_depend for more options.)
- `--nodelist` allows you to specify specific nodes. This can be good for getting reproducible timings, but it will probably increase your wait time in the queue.
- `--array=0-30` is a specification for 'array jobs': a task that needs to be executed for a range of parameter values.
  *TACC note.* Arry jobs are not supported at TACC; use a launcher instead; section 18.5.3.
- `--mem=10000` specifies the desired amount of memory per node. Default units are megabytes, but can be explicitly indicated with `K/M/G/T`.

*TACC note.* This option can not be used to request arbitrary memory: jobs always have access to all available physical memory, and use of shared memory is not allowed.

See https://slurm.schedmd.com/sbatch.html for a full list.

**Exercise** 18.3. Write a script that executes the *date* command twice, with a *sleep* in between. Submit the script and investigate the output.

### 18.4.2 Environment

Your job script acts like any other shell script when it is executed. In particular, it inherits the calling *environment* with all its environment variables. Additionally, slurm defines a number of environment variables, such as the job ID, the hostlist, and the node and process count.

## 18.5 Parallelism handling

We discuss parallelism options separately.

### 18.5.1 MPI jobs

On most clusters there is a structure with compute nodes, that contain one or more multi-core processors. Thus, you want to specify the node and core count. For this, there are options `-N` and `-n` respectively.

```
#SBATCH -N 4              # Total number of nodes
#SBATCH -n 4              # Total number of mpi tasks
```

It would be possible to specify only the node count or the core count, but that takes away flexibility:

- If a node has 40 cores, but your program stops scaling at 10 MPI ranks, you would use:
  ```
  #SBATCH -N 1
  #SBATCH -n 10
  ```
- If your processes use a large amount of memory, you may want to leave some cores unused. On a 40-core node you would either use
  ```
  #SBATCH -N 2
  #SBATCH -n 40
  ```
  or
  ```
  #SBATCH -N 1
  #SBATCH -n 20
  ```

Rather than specifying a total core count, you can also specify the core count per node with `--ntasks-per-node`.

**Exercise** 18.4. Go through the above examples and replace the `-n` option by an equivalent `--ntasks-per-node` values.

*Python note.* Python MPI programs Python programs using *mpi4py* should be treated like other MPI programs, except that instead of an executable name you specify the python executable and the script name:

```
ibrun python3 mympi4py.py
```

### 18.5.2 Threaded jobs

The above discussion was mostly of relevance to MPI programs. Some other cases:

- For pure-OpenMP programs you need only one node, so the `-N` value is 1. Maybe surprisingly, the `-n` value is also 1, since only one process needs to be created: OpenMP uses thread-level parallelism, which is specified through the *OMP_NUM_THREADS* environment variable.
- A similar story holds for the *Matlab parallel computing toolbox* (note: note the distributed computing toolbox), and the *Python multiprocessing* module.

**Exercise** 18.5. What happens if you specify an `-n` value greater than 1 for a pure-OpenMP program?

For *hybrid computing* MPI-OpenMP programs, you use a combination of slurm options and enviroment variables, such that, for instance, the product of the `--tasks-per-node` and `OMP_NUM_THREADS` is less than the core count of the node.

### 18.5.3 Parameter sweeps / ensembles / massively parallel

So far we have focused on jobs where a single parallel executable is scheduled. However, there are use cases where you want to run a sequential (or very modestly parallel) executable for a large number of inputs. This is called variously a *parameter sweep* or an *ensemble*.

Slurm can support this itself with *array jobs*, though there are more sophisticated *launcher* tools for such purposes.

*TACC note.* TACC clusters do not support array jobs. Instead, use the *launcher* or *pylauncher* modules.

## 18.6 Job running

When your job is running, its status is reported as `R` by *squeue*. That command also reports which nodes are allocated to it.

```
squeue -j 5807991
       JOBID   PARTITION     NAME     USER ST       TIME  NODES NODELIST(REASON)
     5807991 development packingt eijkhout  R       0:04      2 c456-[012,034]
```

You can then *ssh* into the compute nodes of your job; normally, compute nodes are off-limits. This is useful if you want to run *top* to see how your processes are doing.

## 18.7 Scheduling strategies

Such a system looks at resource availability and the user's priority to determine when a job can be run.

Of course, if a user is requesting a large number of nodes, it may never happen that that many become available simultaneously, so the batch system will force the availability. It does so by determining a time when that job is set to run, and then let nodes go *idle* so that they are available at that time.

An interesting side effect of this is that, right before the really large job starts, a 'fairly' large job can be run, if it only has a short running time. This is known as *backfill*, and it may cause jobs to be run earlier than their priority would warrant.

## 18.8 File systems

File systems come in different types:

- They can be backed-up or not;
- they can be shared or not; and
- they can be permanent or purged.

On many clusters each node has as local disc, either spinning or a *RAM disc*. This is usually limited in size, and should only be used for temporary files during the job run.

Most of the file system lives on discs that are part of *RAID array*. These discs have a large amount of redundancy to make them fault-tolerant, and in aggregate they form a *shared file system*: one unified file system that is accessible from any node and where files can take on any size, or at least much larger than any individual disc in the system.

> *TACC note.* The `HOME` file system is limited in size, but is both permanent and backed up. Here you put scripts and sources.
>
> The `WORK` file system is permanent but not backed up. Here you can store output of your simulations. However, currently the work file system can not immediately sustain the output of a large parallel job.

The `SCRATCH` file system is purged, but it has the most bandwidth for accepting program output. This is where you would write your data. After post-processing, you can then store on the work file system, or write to tape.

**Exercise** 18.6. If you install software with *cmake*, you typically have

1. a script with all your cmake options;
2. the sources,
3. the installed header and binary files
4. temporary object files and such.

How would you orgnize these entities over your available file systems?

## 18.9 Examples

Very sketchy section.

### 18.9.1 Job dependencies

```
JOB=`sbatch my_batchfile.sh | egrep -o -e "\b[0-9]+$"`

#!/bin/sh

# Launch first job
JOB=`sbatch job.sh | egrep -o -e "\b[0-9]+$"`

# Launch a job that should run if the first is successful
sbatch --dependency=afterok:${JOB} after_success.sh

# Launch a job that should run if the first job is unsuccessful
sbatch --dependency=afternotok:${JOB} after_fail.sh
```

### 18.9.2 Multiple runs in one script

```
ibrun stuff &
sleep 10
for h in hostlist ; do
  ssh $h "top"
done
wait
```

## 18.10 Review questions

For all true/false questions, if you answer False, what is the right answer and why?

**Exercise** 18.7. T/F? When you submit a job, it starts running immediately once sufficient resources are available.

**Exercise** 18.8. T/F? If you submit the following script:

```
#!/bin/bash
#SBATCH -N 10
#SBATCH -n 10
echo "hello world"
```

you get 10 lines of 'hello world' in your output.

**Exercise** 18.9. T/F? If you submit the following script:

```
#!/bin/bash
#SBATCH -N 10
#SBATCH -n 10
hostname
```

you get the hostname of the login node from which your job was submitted.

**Exercise** 18.10. Which of these are shared with other users when your job is running:

- Memory;
- CPU;
- Disc space?

**Exercise** 18.11. What is the command for querying the status of your job?

- `sinfo`
- `squeue`
- `sacct`

**Exercise** 18.12. On 4 nodes with 40 cores each, what's the largest program run, measured in

- MPI ranks;
- OpenMP threads?

# Chapter 19

# SimGrid

Many readers of this book will have access to some sort of parallel machine so that they can run simulations, maybe even some realistic scaling studies. However, not many people will have access to more than one cluster type so that they can evaluate the influence of the *interconnect*. Even then, for didactic purposes one would often wish for interconnect types (fully connected, linear processor array) that are unlikely to be available.

In order to explore architectural issues pertaining to the network, we then resort to a simulation tool, *SimGrid*.

### *Installation*

***Compilation*** You write plain MPI files, but compile them with the *SimGrid compiler* `smpicc`.

***Running*** SimGrid has its own version of `mpirun`: `smpirun`. You need to supply this with options:

- `-np 123456` for the number of (virtual) processors;
- `-hostfile simgridhostfile` which lists the names of these processors. You can basically make these up, but are defined in:
- `-platform arch.xml` which defines the connectivity between the processors.

For instance, with a hostfile of 8 hosts, a linearly connected network would be defined as:

```
<?xml version='1.0'?>
<!DOCTYPE platform SYSTEM "http://simgrid.gforge.inria.fr/simgrid/simgrid.dtd">

<platform version="4">

<zone id="first zone" routing="Floyd">
  <!-- the resources -->
  <host id="host1" speed="1Mf"/>
  <host id="host2" speed="1Mf"/>
  <host id="host3" speed="1Mf"/>
```

```
    <host id="host4" speed="1Mf"/>
    <host id="host5" speed="1Mf"/>
    <host id="host6" speed="1Mf"/>
    <host id="host7" speed="1Mf"/>
    <host id="host8" speed="1Mf"/>
    <link id="link1" bandwidth="125MBps" latency="100us"/>
    <!-- the routing: specify how the hosts are interconnected -->
    <route src="host1" dst="host2"><link_ctn id="link1"/></route>
    <route src="host2" dst="host3"><link_ctn id="link1"/></route>
    <route src="host3" dst="host4"><link_ctn id="link1"/></route>
    <route src="host4" dst="host5"><link_ctn id="link1"/></route>
    <route src="host5" dst="host6"><link_ctn id="link1"/></route>
    <route src="host6" dst="host7"><link_ctn id="link1"/></route>
    <route src="host7" dst="host8"><link_ctn id="link1"/></route>
  </zone>

  </platform>
```

(such files can be generated with a short shell script).

The `Floyd` designation of the routing means that any route using the transitive closure of the paths given can be used. It is also possible to use `routing="Full"` which requires full specification of all pairs that can communicate.

# Chapter 20

# Bibliography


[1] Alfred V. Aho, Brian W. Kernighan, and Peter J. Weinberger. *The Awk Programming Language*. Addison-Wesley Series in Computer Science. Addison-Wesley Publ., 1988. ISBN 020107981X, 9780201079814. [Cited on page 37.]

[2] L.S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammerling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R.C. Whaley. *ScaLAPACK Users' Guide*. SIAM, 1997. [Cited on page 122.]

[3] Netlib.org BLAS reference implementation. http://www.netlib.org/blas. [Cited on page 122.]

[4] Yaeyoung Choi, Jack J. Dongarra, Roldan Pozo, and David W. Walker. Scalapack: a scalable linear algebra library for distributed memory concurrent computers. In *Proceedings of the fourth symposium on the frontiers of massively parallel computation (Frontiers '92), McLean, Virginia, Oct 19–21, 1992*, pages 120–127, 1992. [Cited on page 122.]

[5] Edsger W. Dijkstra. Programming as a discipline of mathematical nature. *Am. Math. Monthly*, 81:608–612, 1974. [Cited on page 151.]

[6] Jack J. Dongarra, Jeremy Du Croz, Sven Hammarling, and Iain Duff. A set of level 3 basic linear algebra subprograms. *ACM Transactions on Mathematical Software*, 16(1):1–17, March 1990. [Cited on page 122.]

[7] Jack J. Dongarra, Jeremy Du Croz, Sven Hammarling, and Richard J. Hanson. An extended set of FORTRAN basic linear algebra subprograms. *ACM Transactions on Mathematical Software*, 14(1):1–17, March 1988. [Cited on page 122.]

[8] Dale Dougherty and Arnold Robbins. *sed & awk*. O'Reilly Media, 2nd edition edition. Print ISBN: 978-1-56592-225-9 , ISBN 10:1-56592-225-5; Ebook ISBN: 978-1-4493-8700-6, ISBN 10:1-4493-8700-4. [Cited on page 37.]

[9] Victor Eijkhout. *The Science of TEX and LATEX*. lulu.com, 2012. [Cited on page 41.]

[10] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, pages 576–580, October 1969. [Cited on page 151.]

[11] Helmut Kopka and Patrick W. Daly. *A Guide to LATEX*. Addison-Wesley, first published 1992. [Cited on page 196.]

[12] L. Lamport. *LATEX, a Document Preparation System*. Addison-Wesley, 1986. [Cited on page 196.]

[13] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. Basic linear algebra subprograms for fortran usage. *ACM Trans. Math. Softw.*, 5(3):308–323, September 1979. [Cited on page 122.]

[14] Robert Mecklenburg. *Managing Projects with GNU Make*. O'Reilly Media, 3rd edition edition, 2004. Print ISBN:978-0-596-00610-5 ISBN 10:0-596-00610-1 Ebook ISBN:978-0-596-10445-0 ISBN 10:0-596-10445-6. [Cited on page 57.]

[15] Sandra Mendez, Sebastian Lührs, Volker Weinberg, Dominic Sloan-Murphy, and Andrew Turner. Best practice guide - parallel i/o. https://prace-ri.eu/training-support/best-practice-guides/best-practice-guide-parallel-io/, 02 2019. [Cited on page 139.]

[16] Frank Mittelbach, Michel Goossens, Johannes Braams, David Carlisle, and Chris Rowley. *The LaTeX Companion, 2nd edition*. Addison-Wesley, 2004. [Cited on page 196.]

[17] NASA Advaned Supercomputing Division. NAS parallel benchmarks. https://www.nas.nasa.gov/publications/npb.html. [Cited on page 209.]

[18] Tobi Oetiker. The not so short introduction to LaTeX. http://tobi.oetiker.ch/lshort/. [Cited on pages 184 and 196.]

[19] Jack Poulson, Bryan Marker, Jeff R. Hammond, and Robert van de Geijn. Elemental: a new framework for distributed memory dense matrix computations. *ACM Transactions on Mathematical Software*. submitted. [Cited on page 122.]

[20] S. Shende and A. D. Malony. *International Journal of High Performance Computing Applications*, 20:287–331, 2006. [Cited on page 205.]

[21] TeX frequently asked questions. [Cited on page 196.]

[22] R. van de Geijn, Philip Alpatov, Greg Baker, Almadena Chtchelkanova, Joe Eaton, Carter Edwards, Murthy Guddati, John Gunnels, Sam Guyer, Ken Klimkowski, Calvin Lin, Greg Morrow, Peter Nagel, James Overfelt, and Michelle Pal. Parallel linear algebra package (PLAPACK): Release r0.1 (beta) users' guide. 1996. [Cited on page 122.]

[23] Robert A. van de Geijn. *Using PLAPACK: Parallel Linear Algebra Package*. The MIT Press, 1997. [Cited on page 122.]

[24] Greg Wilson, D. A. Aruliah, C. Titus Brown, Neil P. Chue Hong, Matt Davis, Richard T. Guy, Steven H. D. Haddock, Kathryn D. Huff, Ian M. Mitchell, Mark D. Plumbley, Ben Waugh, Ethan P. White, and Paul Wilson. Best practices for scientific computing. *PLOS Biology*, 12(1):1–7, 01 2014. [Cited on page 6.]

# Chapter 21

# List of acronyms

**ABI** Application Binary Interface
**ADL** Argument-Dependent Lookup
**AMR** Adaptive Mesh Refinement
**AOS** Array-Of-Structures
**API** Application Programmer Interface
**AVX** Advanced Vector Extensions
**BEM** Boundary Element Method
**BFS** Breadth-First Search
**BLAS** Basic Linear Algebra Subprograms
**BM** Bowers-Moore
**BSP** Bulk Synchronous Parallel
**BVP** Boundary Value Problem
**CAF** Co-array Fortran
**CCS** Compressed Column Storage
**CG** Conjugate Gradients
**CGS** Classical Gram-Schmidt
**COO** Coordinate Storage
**CPP** C Preprocessor
**CPU** Central Processing Unit
**CRS** Compressed Row Storage
**CSV** comma-separated values
**CUDA** Compute-Unified Device Architecture
**DAG** Directed Acyclic Graph
**DL** Deep Learning
**DRAM** Dynamic Random-Access Memory
**DSP** Digital Signal Processing
**FD** Finite Difference
**FE** Finite Elements
**FMA** Fused Multiply-Add
**FDM** Finite Difference Method
**FEM** Finite Element Method
**FMM** Fast Multipole Method
**FOM** Full Orthogonalization Method
**FPU** Floating Point Unit
**FFT** Fast Fourier Transform
**FSA** Finite State Automaton
**FSB** Front-Side Bus
**FPGA** Field-Programmable Gate Array
**GMRES** Generalized Minimum Residual
**GPU** Graphics Processing Unit
**GPGPU** General Purpose Graphics Processing Unit
**GS** Gram-Schmidt
**GSL** Guideline Support Library
**GnuSL** GNU Scientific Library
**GUI** Graphical User Interface
**HDFS** Hadoop File System
**HPC** High-Performance Computing
**HPF** High Performance Fortran
**IBVP** Initial Boundary Value Problem
**IDE** Integrated Development Environment
**ILP** Instruction Level Parallelism
**ILU** Incomplete LU
**IMP** Integrative Model for Parallelism
**IVP** Initial Value Problem
**LAPACK** Linear Algebra Package
**LAN** Local Area Network
**LBM** Lattice Boltzmann Method
**LRU** Least Recently Used
**MGS** Modified Gram-Schmidt
**MIC** Many Integrated Cores
**MIMD** Multiple Instruction Multiple Data

**MGS** Modified Gram-Schmidt
**ML** Machine Learning
**MPI** Message Passing Interface
**MPL** Message Passing Library
**MSI** Modified-Shared-Invalid
**MTA** Multi-Threaded Architecture
**MTSP** Multiple Traveling Salesman Problem
**NUMA** Non-Uniform Memory Access
**ODE** Ordinary Diffential Equation
**OO** Object-Oriented
**OOP** Object-Oriented Programming
**OS** Operating System
**PGAS** Partitioned Global Address Space
**PDE** Partial Diffential Equation
**PRAM** Parallel Random Access Machine
**RDMA** Remote Direct Memory Access
**RNG** Random Number Generator
**SAN** Storage Area Network
**SAS** Software As a Service
**SCS** Shortest Common Superset
**SFC** Space-Filling Curve
**SGD** Stochastic Gradient Descent
**SIMD** Single Instruction Multiple Data
**SIMT** Single Instruction Multiple Thread
**SLURM** Simple Linux Utility for Resource Management
**SM** Streaming Multiprocessor
**SMP** Symmetric Multi Processing
**SMT** Symmetric Multi Threading
**SOA** Structure-Of-Arrays
**SOR** Successive Over-Relaxation
**SSOR** Symmetric Successive Over-Relaxation
**SP** Streaming Processor
**SPMD** Single Program Multiple Data
**SPD** symmetric positive definite
**SRAM** Static Random-Access Memory
**SSE** SIMD Streaming Extensions
**SSSP** Single Source Shortest Path
**STL** Standard Template Library
**TBB** Threading Building Blocks (Intel)
**TDD** Test-Drive Development
**TLB** Translation Look-aside Buffer
**TSP** Traveling Salesman Problem
**UB** Undefined Behavior
**UMA** Uniform Memory Access
**UPC** Unified Parallel C
**WAN** Wide Area Network

# Chapter 22

# Index

ISBN 978-1-257-99254-6 90000
9 781257 992546