Introduction to Supercomputing

TMA4280 · Introduction to development tools

0.1 Development tools

During this course, only the `make` tool, compilers, and the GIT tool will be used for the sake of simplicity: no integrated development environment is required. The purpose of the computers labs and project is to get you to do fairly low-level programming with C/C++ or FORTRAN using OpenMP and MPI. You need to be able to write simple algorithms involving vectors and matrice, manage the allocation/deallocation of memory, and compile/run the code. Of course you are free to explore more advanced aspects of the development of numerical algorithms on a computer, feel free to ask questions.

Revision control system

In this section, we will use the `GIT` revision control system tool to create a local repository and push it to Github. You need to create an account on GitHub to be able to create source code repositories. Although, not crucial for this course, it will make your work more efficient and ease working with your teammate(s) on the projects.

Motivations and goals

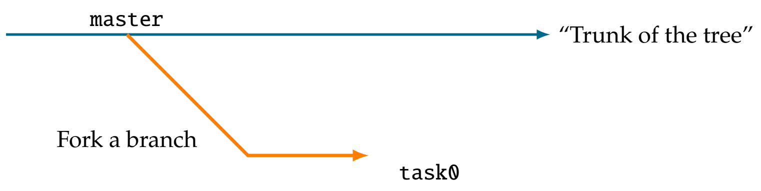
The rationale behind Revision Control Systems (RCS) is that the history of code development should be kept to understand the evolution of the software, and that several individuals should be able to work on the same code base simultaneously. Different RCS exist and adopt a more or less distributed collaborative approach. GIT has become the most popular in the recent years (mainly because it is used by GNU/Linux communities), and offers a distributed environment for collaborating on projects.

A collection of files can be *tracked* using a RCS: the history of modifications of each file is recorded and can be reviewed at any time. The collection of files *tracked* and the metadata used to do so consist of a *repository*. In your case, the files will be source code, makefiles, reports, . . . Whenever the collection of files tracked contains source code, it is usually called a *source repository*.

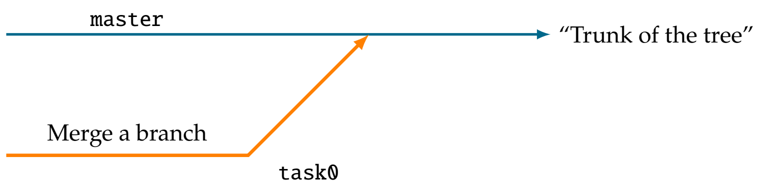
Concepts

The *source repository* consists of a tree of directories containing the files (source code, build tools, documentation), . . .) The history of changes to this collection of files will be kept and represented as a sequence of *changesets*: any *changeset* contains the modification to the files. A new *changeset* is created any time, you *commit* the modifications.

To allow simulaneous work on the code, the notion of *branch* should be introduced. You can represent working on a project with a *tree*: the work is added to a reference version sometimes called a *trunk*, but people should be able to work independently and simulaneously on the project, so they are able to create a local *branch* containing their modifications. Usually for software taska requiring modifications fall in the "bug fix" or "feature" category. Sometimes the reference version changes and the *local branch* needs be synchronized with a *remote branch*. When the task is completed, the *local branch* can be synchronized with the *remote branch*, possible conflicts solved (modifications to the same file), and finally merged.



By default, a *branch* name `master` is created: you can create a new branch from `master` to work on your own version of the code, then merge the modifications at a later stage.



The workflow can be summarized as:

1. Fork a branch
2. Add modifications and synchronize if necessary
3. Solve conflicts
4. Merge

The merge step on GitHub is performed using a "Pull Request": an issue is opened, the code is subject to review, and finally merged if approved.

Exercise 1. Open a terminal and go to the directory you created last week, it does not contain a GIT repository yet:

```
$ git status
fatal: Not a git repository (or any parent up to mount point /home)
Stopping at filesystem boundary (GIT_DISCOVERY_ACROSS_FILESYSTEM not set).
```

Follow the steps:

1. Use `git init` to create a local repository in this directory: you should see now a `.git` directory appear, do not delete it!

   ```
   $ git init
   Initialized empty Git repository in /home/<user>/TMA4280/.git/
   ```

2. Use `git status` to list files in the directory: they should be shown as *untracked*.

   ```
   $ git status
   On branch master

   Initial commit

   Untracked files:
     (use "git add <file>..." to include in what will be committed)

           README
           cat/
           shell/

   nothing added to commit but untracked files present (use "git add" to track)
   ```

3. Create an empty file named `README` and add it to the repository

   ```
   $ git add README
   $ git status
   On branch master

   Initial commit

   Changes to be committed:
     (use "git rm --cached <file>..." to unstage)

           new file:   README

   Untracked files:
     (use "git add <file>..." to include in what will be committed)

           cat/
           shell/
   ```

4. Create your first commit containing the README file (you can write your name in this file for example).

   ```
   $ git commit -m "Initial commit"
   [master (root-commit) 4a177e4] Initial commit
    1 file changed, 1 insertion(+)
    create mode 100644 README
   ```

5. Add all the files you created in the project directory `TMA4280` and commit them.
6. Create an account on GitHub then create a repository named `TMA4280v2018` in the user interface: this remote *repository* will contain your files.
7. Follow the instruction on GitHub and upload (`push`) the local branch to the remote repository.

   ```
   $ git remote add origin https://github.com/<username>/TMA4280v2018.git
   $ git push -u origin master
   ```

Now that your remote repository is created and updated with your initial files, let us have a quick look at the local repository.

Exercise 2. To understand the principle of commits and branches, follow the steps:

1. Use `git log` to look at the history of your repository.
2. Use `git branch` to list local branches.
3. Use `git checkout -b` to create a branch named `lab01` and list again the branches.
4. Use `git push` to push the branch to the remote repository and check on GitHub that it is listed.

With this simple introduction you should be able to manage the exercises and projects without risking losing your work.

GIT Reference

This list presents basic actions that you may need to perform for managing your projects.

Initialize a repository

```
git init
```

Clone a repository

```
git clone <repository_location>
```

Create a branch

```
git checkout -b <branch_name>
```

Checkout file from other branch if necessary

```
git checkout <branch_name> -- <paths>
```

Synchronize feature branch with master

```
git checkout <my_branch_name>
git pull --rebase origin master
```

Cleanup commit history before merge

WARNING: Don't cleanup history if more than one person works on the branch!

Find the number of last commits to be reordered with:

```
git log
```

Then:

```
git rebase -i HEAD~n
```

with n the number of commits to be treated (until now nothing is modified), or until the commit with given hash.

```
git rebase -i hash

Use the following commands to change to first keyword

# Commands:
#  p, pick = use commit
#  r, reword = use commit, but edit the commit message
#  e, edit = use commit, but stop for amending
#  s, squash = use commit, but meld into previous commit
#  f, fixup = like "squash", but discard this commit's log message
#  x, exec = run command (the rest of the line) using shell
```

Using squash ('s') merges commits together.

Possibly use this command to change the last commit message

```
git commit --amend
```

Cleanup master branch locally

```
git pull --rebase origin master
git reset --hard
```

Fetch all the branches from the remotes

```
git fetch --all
```

Pick specific commits from any branch

```
git cherry-pick <commit_hash>
```

Remove untracked files from working directory

```
git clean -f -d
```

Adding a reference to a bug
Add a keyword and the ID in the commit message, like:

```
git commit -m "This is nice commit message"
```

0.2 C Programming

Table 0.1: Data types

Type	Description
`char`	Signed or unsigned integer stored on 8 bits, a *byte*.
`short`	Integer of a least 16-bit long.
`int`	Integer of a least 16-bit long, usually 32-bit long.
`long`	Integer of a least same size as `int`, usually 32-bit long.
`float`	Single precision floating-point number of size 32 bits
`double`	Double precision floating-point number of size 64 bits

Integer types can be `signed` or `unsigned`.

Table 0.2: Control flow statements and loops

Name	Description
`if {` *condition* `} ( ...)`	Conditional.
`if {` *condition* `} ( ...)  else ( ...)`	Conditional.
`for (`*pre*`;`*condition*`;`*post*`) { ...}`	Loop.
`while (`*condition*`) { ...}`	Loop.
`do { ...}` while (*condition*)	Loop.
`switch (`*variable*`) { case` *value*`:` *body*`; break; ...}`	Switch between cases.

Exercise 3. Let us create our first program, the famous "`Hello world!`":

1. Go to the `progs` directory and create a `hello0` subdirectory.
2. Create a `main.c` file, implement a program printing `Hello world!` in `C` using the `printf` function.
3. Compile and execute it.
   - The code consists of a single source file.
   - The compilation and linking can be done in a single command.

     ```
     gcc -o hello hello.c
     ```
   - We might want to turn on compiler optimizations.

     ```
     gcc -o hello hello.c -O2
     ```
   - We might want to include debugging info.

```
gcc -o hello hello.c -g
```

4. Copy the directory to `hello1`, create a header and source file to separate the implementation from the main file: list the commands required to build the program and write a Makefile.

   ```
   gcc -c -o hello.o hello.c
   gcc -c -o utils.o utils.c
   gcc -o hello hello.o utils.o
   ```

5. Modify the program to print `Goodbye world!` depending on an input argument, then add support for a number of times the message should be printed using `int ntimes = atoi(argv[1]);` to convert the argument.

6. Copy the directory to `hello2`, modify the Makefile to create a static library and link.

   - We first link the printing functions into a library

     ```
     gcc -c -o hello_utils.o hello_utils.c
     gcc -c -o goodbye_utils.o goodbye_utils.c
     ar r libutils.a hello_utils.o goodbye_utils.o
     ```

   - Then we build the program.

     ```
     gcc -c -o main.o main.c
     gcc -o hello main.o libutils.a
     ```

Table 0.3: Main function

```c
#include <stdio.h>

int main(int argc, char **argv)
{
  return 0;
}
```

Three directories `hello0`, `hello1`, and `hello2` provided in the repository will show you how to use Makefiles. Makefile are recipes to build programs, designed to overcome two difficulties:

- Compiler commands and options vary across systems and depend on the programming language,

- Binaries are created in three main stages: the source code is pre-processed (ex: `cpp`), then compiled into objects (ex: `cc`) and finally linked into a library or executable (ex: `ld`).

Small programming exercises

Exercise 4. *Data model*. Implement a C/C++ program that creates four integer variables $i0, i1, i2, i3$ of type `char`, `short`, `int`, `long` and two real variable $r0, r1$ of type `float`, `double`. Using the functions `printf` and `sizeof()` to print the number of bytes used for each data type. Compile the program in 32− and 64−bit using the `-m32` and `-m64` options, and note the values in a file. Which model is used in the 64−bit case?

Exercise 5. *Matrix-Vector product*. Implement a C/C++ program that computes $\boldsymbol{y} = \boldsymbol{A}\boldsymbol{x}$.

$$\boldsymbol{A} = \begin{pmatrix} 0.3 & 0.4 & 0.3 \\ 0.7 & 0.1 & 0.2 \\ 0.5 & 0.5 & 0.0 \end{pmatrix}, \qquad \boldsymbol{x} = \begin{pmatrix} 1.0 \\ 1.0 \\ 1.0 \end{pmatrix}.$$

Two versions possible.

Exercise 6. *Vector sum*. Implement a C/C++ program that creates three real vectors $\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{z}$, one real number α and computes $\boldsymbol{z} = \alpha\boldsymbol{x} + \boldsymbol{y}$ for given $\boldsymbol{x}$ and $\boldsymbol{y}$ (double-precision). First use a fixed-size of 10 then implement a version with dynamic size.

Exercise 7. *Dot product*. Implement a C/C++ program that creates two real vectors $\boldsymbol{x}, \boldsymbol{y} \in \mathbb{R}^N$ and one real number α and computes $\alpha = \sum_{i=1}^{N} \boldsymbol{x}_i \boldsymbol{y}_i$ for given $\boldsymbol{x}$ and $\boldsymbol{y}$ (double-precision).