Setting up your computer for IN3200/IN4200

Kristian Gregorius Hustad[1,2,3] (krihus@ifi.uio.no)

Xing Cai[3,2] (xingca@ifi.uio.no)

[1]Centre for Computing in Science Education, University of Oslo
[2]Department of Informatics, University of Oslo
[3]Department of High-Performance Computing, Simula Research Laboratory

Jan 16, 2020

Warning.

This document is a work in progress. Please contact the authors if you should find any errors.

Contents

1 Introduction **3**

2 macOS **5**
2.1 Prerequisites 5
2.1.1 Installing Xcode Command Line Tools 5
2.1.2 Installing Homebrew 6
2.1.3 Setting up conda 6
2.1.4 Obtaining a C compiler with OpenMP support 6
2.1.5 Installing OpenMPI 8

3 Ubuntu **10**
3.1 Installing programs with `apt` 10
3.1.1 Compilation 10
3.1.2 Debugging 11
3.1.3 Profiling 11
3.1.4 Python 11

4 Windows **12**
4.1 Setting up Windows Subsystem for Linux (WSL) 12
4.2 Installing Virtualbox 12

5 Building from source **13**
5.1 Building GCC 13
5.2 Building Open MPI 15

Bibliography **17**

Chapter 1

Introduction

This guide mentions a wide array of software. We will therefore start by giving a brief description for each piece of software as a kind of index for later reference.

Anaconda: A distribution of Python packages and scientific software. Comes with its own package manager, `conda`.

Clang: A C (`clang`) and C++ (`clang++`) compiler based on LLVM. Default compiler on macOS.

GCC: The GNU compiler collection. Includes compilers for C (`gcc`), C++ (`g++`) and Fortran (`gfortran`). Default compiler on most Linux distributions.

GDB: The GNU Debugger. Command: `gdb`

GNU Make: The official documentation describes GNU Make as a *tool which controls the generation of executables and other non-source files of a program from the program's source files.* Technically, Make executes tasks specified in a task-dependency graph, and can be used more broadly to produce files from existing files (which do not need to be program source code). Command: `make`

gperftools: Profiling library developed by Google. https://github.com/gperftools/gperftools

Homebrew: Package manager for macOS. Command: `brew`

Intel compiler suite: The CPU manufacturer Intel makes commercial compilers. These are installed on Abel (in addition to GCC). We will not use them elsewhere since they are not available for free.

LLVM: LLVM is a compiler toolchain released under a BSD license. It is an alternative to GPL-licensed compiler software developed by GNU.

MPICH: MPI library. Installs wrappers around the chosen C compiler. Compile C with `mpicc`, C++ with `mpic++`, and Fortran with `mpifort`.

Open MPI: MPI library. Installs wrappers around the chosen C compiler. Compile C with `mpicc`, C++ with `mpic++`, and Fortran with `mpifort`.

perf: A tool for profiling programs on Linux. https://perf.wiki.kernel.org/index.php/Main_Page

Note that Open MPI and MPICH are two different implementations of MPI. We will only be using Open MPI in this course.

Further instructions depend on which operating system you are using.

- macOS: Go to 2
- Ubuntu: Go to 3
- Windows: Go to 4

Chapter 2

macOS

2.1 Prerequisites

macOS does not include any C compiler by default, but Clang can easily be installed as part of the Xcode Command Line Tools.

2.1.1 Installing Xcode Command Line Tools

Install Xcode Command Line Tools with the following command

```
xcode-select --install
```

Afterwards, try running

```
cc --version
```

You should see a few lines of output starting with "Apple LLVM ...". `cc` will conventionally point to the system compiler, which is Apple Clang in the case of macOS. By closer inspection we see that `/usr/bin/cc` is indeed a symlink (symbolic link) to `/usr/bin/clang`.

```
$ which cc
/usr/bin/cc
$ ls -lh /usr/bin/cc
lrwxr-xr-x 1 root wheel 5 Dec 15  2017 /usr/bin/cc -> clang
```

Note that Xcode Command Line Tools also installs `/usr/bin/gcc`, which is only a wrapper around Clang.

Installing system headers for Xcode *geq* **10.0.** With Xcode 10 and later, Apple decided that system header files should not be installed to `/usr/include` by default, and this causes problems when building many programs e.g. GCC.

There is a workaround for this problem that is described here: https://stackoverflow.com/a/52530212

2.1.2 Installing Homebrew

Homebrew is a package manager for macOS which greatly simplifies installation of various programs. Note that Homebrew is community-driven, and sometimes packages are removed or break.

See instructions for installing Homebrew at https://brew.sh/

2.1.3 Setting up conda

Conda is a package manager, perhaps best known for being bundled with Anaconda, a distribution of various Python packages used in scientific computing. We don't need all of Anaconda, so we will install a minimal conda bundle, called Miniconda, instead. (If you have Anaconda installed, you don't need to install Miniconda.)

Run the commands below to install Miniconda.

```
# download installation script
curl -O https://repo.continuum.io/miniconda/Miniconda3-latest-MacOSX-x86_64.sh

# install (follow the instructions, the defaults are usually sensible)
bash Miniconda3-latest-MacOSX-x86_64.sh
```

See conda's documentation for more details.

Creating a conda environment. We recommend setting up a conda environment for this course. Here we call the environment `hpc`, but you may choose a different name if you wish. We create the environment with Python 3 and `scipy` since `scipy` is used to preprocess some of the datasets.

```
# create the environment
conda create --name hpc python=3 scipy

# and then activate it
source activate hpc
```

You should see `(hpc)` prefixed to your bash prompt when the environment is active. You can deactivate it with

```
source deactivate
```

You can read more about conda environments at https://conda.io/docs/user-guide/tasks/manage-environments.html.

2.1.4 Obtaining a C compiler with OpenMP support

While Xcode Command Line Tools includes a C compiler, `clang`, it does not support OpenMP out of the box. There are several ways to obtain a C compiler with OpenMP support.

1. Build GCC (easy for macOS <= 10.14)

2. Install GCC via brew (best option for macOS >= 10.15)
3. Make Apple Clang use `libomp`
   (a) Install `libomp` with `brew`
   (b) Build `libomp` from source
4. Install an OpenMP-enabled version of Clang with `conda`

For macOS 10.14 (Mojave) and older, we recommend building GCC from source, since that requires fewer workarounds further down the line. For macOS 10.15 (Catalina) and newer, building GCC from source (while still possible) is more tricky, and we recommend installing it via Homebrew. Additionally, GCC includes a Fortran compiler, which can be used to try out other examples from the course textbook [1].

If you want to have both Clang and GCC set up with OpenMP support, you should do both option 1 or 2 and option 3.

Option 3 has two sub-obtions. `libomp` is available from Homebrew at the time of writing, but in case the Homebrew package should break or be removed, it is also easy to build `libomp` from source.

If you are using an older version of macOS and want to install a more recent version of Clang, you should choose option 3 over option 2.

Option 1: Build GCC from source. See section 5.1 for instructions.

Option 2: Installing GCC via Homebrew. With Homebrew installed (see 2.1.2), run

```
brew install gcc
```

Option 3: Make Apple Clang use `libomp`. Apple Clang is actually capable of processing OpenMP directives correctly, but it needs an OpenMP library, `libomp`, to link the executable with.

3a) Install `libomp` from Homebrew: `libomp` can be installed via `brew` with

```
brew install libomp
```

3b) Build `libomp` from source: Alternatively, `libomp` can also be built from source in under a minute. In this guide, we will build the library under `/dev/openmp`

```
mkdir -p ~/dev
cd ~/dev

# check out the code with Subversion
svn co http://llvm.org/svn/llvm-project/openmp/trunk openmp

cd ~/dev/openmp
mkdir build
cd build

# configure build
cmake ~/dev/openmp

# build
make -j 2

# install the library to /usr/local
make install

# optionally we can clean up afterwards
cd ~/dev
rm -rf ~/dev/openmp
```

For more details, see https://openmp.llvm.org/.

When compiling, you must specify `-Xpreprocessor -fopenmp -lomp` (`-lomp` is only required when linking the final executable) as flags to Clang instead of the usual `-fopenmp`.

Option 4: Installing OpenMP-enabled Clang with `conda`. Install the `openmp` and `clangdev` packages from the `conda-forge` channel.

```
conda install -c conda-forge openmp clangdev
```

2.1.5 Installing OpenMPI

See section 5.2 for instructions.

Note that the environment variables `CC` and `FC` can be set (prior to building Open MPI) to control the compiler used by `mpicc` and `mpifort`. If you built GCC from source, and want OpenMPI to use GCC, you must set `CC` and `FC` to point to the `gcc` and `gfortran` executables from your GCC installation.

For instance, if you installed GCC 8.2.0 and set up `gcc-8.2.0` and `gfortran-8.2.0` to point to that installation, set the following environment variables **before** building Open MPI.

```
export CC=gcc-8.2.0
export FC=gfortran-8.2.0
```

These environment variables can be unset afterwards with

```
unset CC
unset FC
```

Chapter 3

Ubuntu

3.1 Installing programs with `apt`

We will use Ubuntu's package manager `apt` to install the programs we need. First we need to update the local package cache.

```
sudo apt-get update
```

Then we can install packages with the following command

```
sudo apt-get install <list of packages>
```

For instance, we can install `gcc` and `make` with

```
sudo apt-get install gcc make
```

Below is a list of some relevant programs for this course.

3.1.1 Compilation

- `make`
- `gcc`
- `gfortran` (optional)
  - Required if you want to compile Fortran examples from the course textbook [1].
- MPI compiler and runtime
  - `openmpi-bin`
  - `libopenmpi-dev`

3.1.2 Debugging

- `gdb`

3.1.3 Profiling

- `perf`
- `gprof` (optional)

3.1.4 Python

Additionally, Python 3 with scipy is required to preprocess some of the datasets. These can be installed from apt as `python3` and `python3-scipy`. Alternatively they can also be installed via conda. (See instructions in section 2.1.3).

Chapter 4

Windows

We recommend programming in a Linux-like environment. There are two ways to use Linux on Windows:

1. Use Windows Subsystem for Linux (WSL)
2. Set up a virtual machine with Linux

For Windows 10 users, we recommend enabling Windows Subsystem for Linux (WSL) and installing Ubuntu. For older Windows versions, where WSL is not available, we recommend setting up a virtual machine with Virtualbox.

4.1 Setting up Windows Subsystem for Linux (WSL)

Follow the instruction at https://docs.microsoft.com/en-us/windows/wsl/install-win10. Note that WSL is not available for older versions of Windows. If you are using Windows 8 or older, see 4.2.

With Ubuntu installed, you can follow the instructions in 3.

4.2 Installing Virtualbox

Install Virtualbox by following the instructions at https://www.virtualbox.org/manual/UserManual.html#intro-installing.

When you have installed Virtualbox, you need an ISO file with a Linux distribution. For novice Linux users, we recommend that you choose to install the latest LTS release from Ubuntu, which can be obtained at https://www.ubuntu.com/download/desktop.

With Ubuntu installed, you can follow the instructions in 3.

Chapter 5

Building from source

5.1 Building GCC

There's a script for building GCC from source at `dependencies/build_from_source/build_gcc.sh`. It is also listed below.

```
#!/usr/bin/env bash
set -e

# find wget or equivalent program
if command -v wget >/dev/null 2>&1; then
    WGET="wget"
elif command -v curl >/dev/null 2>&1; then
    WGET="curl -O"
else
    echo "ERROR: Could not find wget or curl"
    exit 1
fi

# determine number of logical CPUs
if command -v nproc >/dev/null 2>&1; then
    NUM_PROCS=$(nproc)
elif command -v sysctl >/dev/null 2>&1; then
    NUM_PROCS=$(sysctl -n hw.ncpu)
else
    NUM_PROCS=1
fi


GCC_MIRROR=http://gnuftp.uib.no/gcc
GCC_VERSION=8.2.0

DEV_DIR=$HOME/dev
mkdir -p ${DEV_DIR}
cd ${DEV_DIR}


GCC_SOURCE_TARBALL=gcc-${GCC_VERSION}.tar.gz
GCC_SOURCE_DIR=${DEV_DIR}/gcc-${GCC_VERSION}
GCC_BUILD_DIR=${DEV_DIR}/gcc-build
```

```
GCC_INSTALL_DIR=${HOME}/local/gcc-${GCC_VERSION}

# Download source tarball
if [ ! -f ${GCC_SOURCE_TARBALL} ]; then
    echo "Downloading source tarball"
    ${WGET} ${GCC_MIRROR}/gcc-${GCC_VERSION}/${GCC_SOURCE_TARBALL}
fi

# Extract tarball to directory
if [ ! -d ${GCC_SOURCE_DIR} ]; then
    echo "Extracting source tarball"
    tar xf ${GCC_SOURCE_TARBALL}
fi

# Download dependencies
cd ${GCC_SOURCE_DIR}
echo "Downloading dependencies"
./contrib/download_prerequisites
cd ${DEV_DIR}

echo "Cleaning any previous build directory"
rm -rf ${GCC_BUILD_DIR}
mkdir -p ${GCC_BUILD_DIR}
cd ${GCC_BUILD_DIR}

# configure GCC build
echo "Configuring GCC"
${GCC_SOURCE_DIR}/configure \
    --enable-languages=c,c++,fortran,lto    \
    --prefix=${GCC_INSTALL_DIR} \
    --disable-multilib

# build GCC (this can take an hour or two)
echo "Building GCC"
make -j ${NUM_PROCS} -l ${NUM_PROCS}

# install into ${GCC_INSTALL_DIR}
echo "Installing GCC into ${GCC_INSTALL_DIR}"
make install

# remove built files
echo "Removing build directory"
cd ..
rm -rf ${GCC_BUILD_DIR}


## optionally, add symlinks to make it easy to switch between gcc versions
## note that ~/bin must be an existing directory on PATH for this to work
## uncomment the lines below if you want to set up these symlinks
#echo "Setting up symlinks from ~/bin"
#ln -s ${GCC_INSTALL_DIR}/bin/gcc ~/bin/gcc-${GCC_VERSION}
#ln -s ${GCC_INSTALL_DIR}/bin/g++ ~/bin/g++-${GCC_VERSION}
#ln -s ${GCC_INSTALL_DIR}/bin/gfortran ~/bin/gfortran-${GCC_VERSION}

## if GCC_VERSION was set to 8.2.0, you should now be able to run
## $ gcc-8.2.0 --version

echo "Done"
```

5.2 Building Open MPI

There's a script for building Open MPI from source at `dependencies/build_from_source/build_openmpi.sh`. It is also listed below.

```
#!/usr/bin/env bash
set -e

# find wget or equivalent program
if command -v wget >/dev/null 2>&1; then
    WGET="wget"
elif command -v curl >/dev/null 2>&1; then
    WGET="curl -O"
else
    echo "ERROR: Could not find wget or curl"
    exit 1
fi

# determine number of logical CPUs
if command -v nproc >/dev/null 2>&1; then
    NUM_PROCS=$(nproc)
elif command -v sysctl >/dev/null 2>&1; then
    NUM_PROCS=$(sysctl -n hw.ncpu)
else
    NUM_PROCS=1
fi


OPENMPI_VERSION=3.1.1
OPENMPI_SOURCE_TARBALL=openmpi-${OPENMPI_VERSION}.tar.gz
OPENMPI_SOURCE_TARBALL_URL=https://download.open-mpi.org/release/open-mpi/v3.1/${OPENMPI_SOURCE_TAR

DEV_DIR=$HOME/dev
mkdir -p ${DEV_DIR}
cd ${DEV_DIR}


OPENMPI_SOURCE_TARBALL=openmpi-${OPENMPI_VERSION}.tar.gz
OPENMPI_SOURCE_DIR=${DEV_DIR}/openmpi-${OPENMPI_VERSION}
OPENMPI_BUILD_DIR=${DEV_DIR}/openmpi-build
OPENMPI_INSTALL_DIR=${HOME}/local/openmpi-${OPENMPI_VERSION}

# Download source tarball
if [ ! -f ${OPENMPI_SOURCE_TARBALL} ]; then
    echo "Downloading source tarball"
    ${WGET} ${OPENMPI_SOURCE_TARBALL_URL}
fi

# Extract tarball to directory
if [ ! -d ${OPENMPI_SOURCE_DIR} ]; then
    echo "Extracting source tarball"
    tar xf ${OPENMPI_SOURCE_TARBALL}
fi


echo "Cleaning any previous build directory"
rm -rf ${OPENMPI_BUILD_DIR}
mkdir -p ${OPENMPI_BUILD_DIR}
cd ${OPENMPI_BUILD_DIR}
```

```
# configure OPENMPI build
echo "Configuring Open MPI"
${OPENMPI_SOURCE_DIR}/configure \
    --prefix=${OPENMPI_INSTALL_DIR} \
    --with-hwloc \
    --disable-java

# build Open MPI (this can take a couple of minutes)
echo "Building Open MPI"
make -j ${NUM_PROCS} -l ${NUM_PROCS}

# install into ${OPENMPI_INSTALL_DIR}
echo "Installing Open MPI into ${OPENMPI_INSTALL_DIR}"
make install

# remove built files
echo "Removing build directory"
cd ..
rm -rf ${OPENMPI_BUILD_DIR}

echo "Done"
```

Warning.

`mpicc` and the other wrappers don't work when pointed to by a symlink with a different filename. This makes switching between two Open MPI installations harder, but it can be done by editing the `PATH` environment variable.

Bibliography

[1] G. Hager and G. Wellein, *Introduction to High Performance Computing for Scientists and Engineers*, ser. Computational Science Series. CRC Press, 2011.