MinGW coding under Windows (C, C++, OpenMP, MPI)

This tutorial helps you set up a coding environment on Windows with the support for C/C++, Fortran, OpenMP, MPI, as well as compiling and running the TMAC package. If you use `cygwin`, please use this tutorial instead.

Overview

- **MSYS2** is a unix-like command-line evironment for Windows. It comes with the `pacman` package manager, which helps you install packages for coding and other purposes.
- **MINGW32** and **MINGW64** include the GCC (GNU Compiler Collection), which you can use to compile C/C++, Fortran, and other source codes. As their names suggest, they are 32-bit and 64-bit versions, respectively. Codes that they compile can execute natively under Windows without runtime libraries from MINGW32, MINGW64, or MSYS2. GCC compilers can be called under both MSYS2 and Window's native CMD. I prefer MSYS2 because its sets up the environment, provides a package manager, and installs other coding tools (e.g., autoconf and automake). MINGW32 and MINGW64 can be also installed alone without MSYS2, but I suggest you install MSYS2 first and then use its package manager to install either MINGW32 (for 32-bit Windows) or MINGW64 (for 64-bit Windows).
- **OpenMP** is an inter-thread communication specification; it often comes with compilers (e.g., latest GCC). In principle, threads can share memory, but processes cannot. To share data, processes use message passing.
- **MPI** (Message Passing Interface) is a specification for inter-process communication via message passing. MS-MPI is Microsoft's implementation of MPI. It comes with header and library files, as well as some exe's, that you need to compile and execute your codes with the MPI support. Besides MS-MPI, Windows supports other MPI implementations.
- **BLAS** is a collection of routines you can call from your codes to perform basic vector and matrix operations. They come with header and library files, and they are typically more efficiently than your own implementations. Besides the original NetLib implementation, BLAS has other implementations from ATLAS, INTEL MKL, Open BLAS, etc. Typically, you can use the include and library switches of your compilers to select the BLAS implementation that you code will use.
- **LAPACK** a library of matrix algorithms. It implements algorithms for common matrix factorizations and solving linear systems, least squares problems, eigenvalue and singular value problems. LAPACK uses BLAS.
- **CMake** a tool that manages the build process on many platforms including MSYS2 and MINGW32/64. It generates configure script and Makefiles.
- **Eigen** a template-style library for matrix and linear algebra operations.
- **GSL** is GNU Scientific Library, a software library for numerical computations.
- **FFTW** is the fastest free C library of the fast Fourier transform (FFT).
- **TMAC** is a C++11 framework that lets you quickly develop your own codes for solving a variety of optimization problems in a parallel fashion. In particular, you can add your operators to TMAC and run your algorithms based on operator splitting and coordinate update methods. TMAC makes it easy to test both single-threaded and synchronous, as well as asynchronous, parallel algorithms.

Install MSYS2 and MINGW32 / MINGW64

- start with no mingw or msys on my system (otherwise, please uninstall them)
- run the MSYS2 installer, or use the MSYS2 installer at sourceforge (choose i686 for 32-bit MSYS2 and x86_64 for 64-bit MSYS2)
- open MSYS2 at `C:\msys64\msys2_shell.bat`
- let `pacman` update MSYS2 and refreshe its package list

```
> pacman -Syuu   # update the package list, will take a while
```

- close and reopen MSYS2
- install coding tools (from packages `base-devel`, `git`, and `toolchain`) as follows. Note that the three packages are large and have more than what you need; afterward, the msys folder takes 1.77GB of disk space; you can install only the modules you will need. You can always uninstall a package using `pacman -Rns <package_name>`:

If you installed 64-bit MSYS2, then do

```
> pacman -S base-devel git mingw-w64-x86_64-toolchain   # will take a while
```

- in fact, it installs both MINGW64 and MINGW32. So, you can create 32-bit codes with MINGW32 in 64-bit Windows.
- close MSYS2; open MSYS2-MINGW64 by running `C:\msys64\mingw64.exe`

If you installed 32-bit MSYS2, then do

```
> pacman -S base-devel git mingw-w64-i686-toolchain   # will take a while
```

- close MSYS2; open MSYS2-MINGW32 by running`C:\msys64\mingw32.exe`

Verify installation

```
> gcc -v    # test gcc
```

Hello world and OpenMP

- create a folder under home

```
> cd ~
> mkdir omp_hello
> cd omp_hello
```

- next, download and compile the demo code

```
> wget https://computing.llnl.gov/tutorials/openMP/samples/C/omp_hello.c
> gcc -fopenmp omp_hello.c -o omp_hello     # generate the executable file omp_hello.exe
```

- try running it

```
> ./omp_hello       # By default, gcc creates 1 thread for each core. My PC has 2 physical cores (4 logical cores under hyperthreading)
                    # The order of the five output lines will be random
Hello World from thread = 1
Hello World from thread = 2
Hello World from thread = 3
Hello World from thread = 0
Number of threads = 4
```

- try it with more threads

```
> export OMP_NUM_THREADS=8  # explicitly set 8 threads by the OMP_NUM_THREADS environment variable
> ./omp_hello
Hello World from thread = 3
Hello World from thread = 0
Number of threads = 8
Hello World from thread = 6
Hello World from thread = 5
Hello World from thread = 4
Hello World from thread = 1
Hello World from thread = 2
Hello World from thread = 7
```

Install Microsoft MPI (MS-MPI)

- download MS MPI V7 (newer versions are available but not tested), and install both `msmpisdk.msi` and `MSMpiSetup.exe`
- execute `C:\msys64\mingw64.exe` and locate the environment variables WINDIR, MSMPI_INC, and MSMPI_LIB64 by running:

```
> printenv | grep "WIN\|MSMPI"
```

If you don't see them, then your Windows environment variables are not passed to MSYS2-MINGW64; you need to correct this before proceeding to the next step.

- add/create the header and library files for MS-MPI for later use:

```
> mkdir ~/msmpi                     # create a temporary folder under your home directory
> cd ~/msmpi                        # enter the folder
> cp "$MSMPI_LIB64/msmpi.lib" .     # copy msmpi.lib to ~/msmpi/; the import library, which is a placeholder for dll
> cp "$WINDIR/system32/msmpi.dll" . # copy msmpi.dll to ~/msmpi/; the runtime library
> gendef msmpi.dll                  # generate msmpi.def. For 32-bit, use: gendef -a msmpi.dll, which specifies the stdcall format
> dlltool -d msmpi.def -D msmpi.dll -l libmsmpi.a   # generate the (static) library file libmsmpi.a
> cp libmsmpi.a /mingw64/lib        # copy this library file to where g++ looks for them;
                                    # try "g++ --print-search-dirs"
> cp "$MSMPI_INC/mpi.h" .           # copy the header file mpi.h to ~/msmpi/
```

- open mpi.h under `~/msmpi` in an editor, look for "`typedef __int64 MPI_Aint`". Just above it, add a new line with "`#include <stdint.h>`" (without the quotes), which we need for the definition `__int64`.

```
> cp mpi.h /mingw64/include         # copy the header file to the default include folder
```

- now you can delete the folder `~/msmpi`

Hello world / MPI

- create a folder under home

```
> cd ~
> mkdir mpi_hello
> cd mpi_hello
```

- next, save mpi_hello_world.c to `/mpi_hello` and compile it

```
> gcc mpi_hello_world.c -lmsmpi -o mpi_hello
       # -lmsmpi: links with the msmpi library, the file libmsmpi.a that we generated above
```

```
> mpiexec -n 4 mpi_hello.exe        # run it with 4 processes
```

Compile BLAS from source (the generated code usually runs slower than OpenBLAS below)

if you want to install the original BLAS, do

```
> wget http://www.netlib.org/blas/blas-3.6.0.tgz
> tar xf blas-3.6.0.tgz
> cd BLAS-3.6.0
> gfortran.exe -c *.f       # compile each .f file and produce a .o fils
                            # (you can also add the Optimization switch -O3).
> ar rv libblas.a *.o       # combine all the .o files into a library file.
> cp libblas.a /mingw64/lib # copy the library file to where g++ looks for them;
                            # try "g++ --print-search-dirs"
```

- you must add `-lgfortran` during compiling/linking (see below) because BLAS requires the gfortran library (there might be a way to avoid this, but I didn't make an attempt)

Install OpenBLAS (preferred over compiling BLAS for code efficiency)

```
> pacman -S mingw-w64-x86_64-openblas   # use "pacman -S mingw-w64-i686-openblas" for 32-bit
```

- continuing this tutorial, you must replace `-lblas` by `-lopenblas` during compiling/linking below

Install BLAS and LAPACK together (this BLAS code is typically slower than OpenBLAS above)

Let us use the scripts for building a set of packages from a Github Repo

```
> pacman -S  mingw-w64-x86_64-cmake                    # install CMake
> git clone https://github.com/msys2/MINGW-packages.git # clone the scripts
> cd MINGW-packages/mingw-w64-lapack                   # locate the LAPACK script
```

Open the file `PKGBUILD` in a text editor and replace "RESPOSE" to "RESPONSE" (to avoid an error message "ar.exe: Argument list too long")

```
> makepkg-mingw                                # build BLAS and LAPACK
> pacman -U mingw-w64-x86_64-lapack*.pkg.tar.xz   # install BLAS and LAPACK
```

Download lapack_test.cpp to test LAPACK:

```
> cd ~
> wget http://www.math.ucla.edu/~wotaoyin/software/lapack_test.cpp  # download
> g++ lapack_test.cpp -llapack -o lapack_test     # build
> ./lapack_test                                  # run
```

CMake

We first need to install make and CMake for MINGW:

```
> pacman -S make mingw-w64-x86_64-cmake
```

You must close and reopen MSYS2-MINGW64 by running `C:\msys64\mingw64.exe`.

You can test CMake as follows:

```
> cd ~
> git clone https://github.com/bast/cmake-example.git   # download a demo code
> cd cmake-example
> git checkout 7931bf4                              # revert to a 2016 version; (courtesy of Filip Sund) the 2018 version will let
                                                    # CMake download the 2018 googletest, which has a bug in "internal\gtest-port.h."
> mkdir build                                       # you cannot build a project in the source-code folder, so we create a subfolder.
> cd build                                          # ensure this folder is empty, no previous CMake* files or folder.
> cmake .. -G "MSYS Makefiles" -DCMAKE_INSTALL_PREFIX=$MINGW_PREFIX     # ".." locates CMakeLists.txt, "MSYS Makefiles" is needed MSYS2,
                                                                        # -D... tells "make install" where to install files.
> ls                                                # CMake creates many files including Makefile for MSYS2's make command
> make                                              # create three exe files
> ls *.exe
> ./hello.x                                         # you should get Hello World
> ./main.x
> ./unit_tests
```

Install and test Eigen

Download Eigen

```
> pacman -S mercurial   # Mercurial is a distributed revision-control tool a distributed revision-control tool
> hg clone https://bitbucket.org/eigen/eigen/   # downloads and installs latest Eigen
```

Install Eigen to the system

```
> cmake ~/eigen -G"MSYS Makefiles" -DCMAKE_INSTALL_PREFIX=$MINGW_PREFIX
```

Install and test GSL

Download and install GSL

```
> cd ~
> wget -qO- http://gnu.mirrors.pair.com/gsl/gsl-latest.tar.gz | tar xvz   # this creates the folder ~/gsl-2.4/
> cd gsl-2.4
> ./configure --prefix=/mingw64     # I am unsure about the necessity of --prefix=/mingw64, but it should not hurt
> make                              # this will take 15 minutes or so to complete
> make install                      # install header and library files to the system, you can use "make uninstall" to do the opposite
```

Save the example program as `gsl_test.c+` in your homefolder.

```
> cd ~
> gcc $(gsl-config --cflags) gsl_test.c $(gsl-config --libs) -o gsl_test    # gsl-config automatically generates proper compiling and linke flags
> ./gsl_test
```

Install and test Google Test

Download and install Google Test

```
> pacman -S mingw-w64-x86_64-googletest-git
```

Run a sample code made by dmonopoly as follows:

```
> cd ~
> git clone https://github.com/dmonopoly/gtest-cmake-example.git
> cd gtest-cmake-example/
> mkdir build
> cd build/
> cmake .. -G "MSYS Makefiles" -Dtest=ON    # dmonopoly's instruction to add -Dtest=ON
> make
> ls *.exe
> ./project1        # run the binary code of project1.cpp
> ./runUnitTests    # run the binary code that tests project1.cpp
> make test         # run all the tests easily
```

Install and test FFTW

Download, compile, and install FFTW. Apply some configure options.

```
> cd ~
> wget http://www.fftw.org/fftw-3.3.8.tar.gz      # the latest stable build as of September 2018
> tar xf fftw-3.3.8.tar.gz
> cd fftw-3.3.8/
> ./configure --with-our-malloc16 --enable-shared --enable-threads --with-combined-threads --enable-sse2
> make
> make install
```

Test FFTW

```
> cd ~
> git clone https://github.com/undees/fftw-example.git
> cd fftw-example/
> gcc fftw_example.c -lfftw3 -o fftw_example.exe    # the c code uses FFTW.
> ./fftw_example.exe                                # run and see some results.
```

Run TMAC

- download the code from GitHub

```
> pacman -S git    # install git
> git clone https://github.com/uclaopt/tmac.git  # download TMAC from GitHub
```

- open `Makefile` in the project root in an editor (e.g., Notepad, WinEdt)
- if you installed BLAS, add `-lgfortran` (if not already there) to the end of the line starting "`LIB := `" because BLAS requires the gfortran library; otherwise, make will run into an error
- if you installed OpenBLAS, replace `-lblas` by `-lopenblas` in the line starting "`LIB := `"; otherwise, make will run into an error

```
> make
> ./bin/tmac_prs_demo -problem_size 1500 -nthread 1    # run Peaceman-Rachford Splitting algorithm with 1 thread
> ./bin/tmac_prs_demo -problem_size 1500 -nthread 2    # run 2 threads
> ./bin/tmac_prs_demo -problem_size 1500 -nthread 4    # run 4 threads
```