Setting up the computing environment

• I will be demonstrating on MAC.
• Linux environment is very similar to MAC and should be easy to follow
• Windows might be hardest to set up

In the past we provided virtual machine, light linux ubuntu, with all software installed. Such virtual machine can be run on any operating system. If there is sufficient interest for a virtual machine, we will create it. Native installation is more efficient, and probably a good idea to try it out.

Essential Software:

• Python, and its packages: numpy, scipy, matplotlib (easy installation with Anaconda  www.anaconda.com)

• C++ compiler, such as gcc, but other C++ compilers should be fine too.

• Text editor for coding (anaconda includes Spyder, which I am learning, Emacs, Aquamacs, or similar)

• make to execute makefiles
Recommended Hardware:

It is impossible to demonstrate these algorithms on different platforms, therefore we need to choose one of the common platforms.

I will be demonstrating algorithms on Mac platform, and if you have Windows or Linux machine, you will need to translate installation to your platform, which might be technically demanding at times.

If you want to follow the course on a similar system, you can choose, for example:

MacBook Pro, Intel i7, 16GB DDR RAM, and 512 GB hard drive.
Recommended Software:

- **Fortran** compiler, such as `gfortran`, or intel fortran (it is getting harder to install nowadays).

- **blas&lapack** libraries. They come preinstalled in most computers or one needs to install vendors libraries (intel mkl for linux). On mac it is contained in Xcode

- **openMP enabled C++** compiler (native gcc rather than apple clang, which is invoked by gcc,g++)

- **gnuplot** for fast plotting.
Installation on MAC:

• Install Xcode package from App Store (Essential)

• We will need specific part of Xcode namely “Command Line Tools”, which might be already installed in your distribution. To check if they are, type:

  xcode-select --print-path

If you do not find them, install by:

  xcode-select --install

Xcode contains C/C++ compiler (gcc/g++). It is just a link to apple's native Clang.

Xcode also contains many libraries. For example, it should include BLAS and LAPACK libraries. To use these libraries, we will use a linker option: -framework Accelerate.

For more information see (https://developer.apple.com/accelerate/)

Xcode also includes make.
Installation on MAC:

Multicore openMP execution:
However, Xcode does not come anymore with gnu compilers (such as gnu-c==gcc and gnu-c++==g++). Instead gcc and g++ point to apple’s own Clang compiler. Unfortunately Clang does not yet support openMP (multicore) instructions. Moreover, Clang does not include fortran compiler, such as gnu compiler (gfortran).

Recommended:
To install openMP, fortran & gnuplot we will use homebrew.
The long instructions of how to install the homebrew are available at http://brew.sh

In summary, you need to paste the following into the terminal prompt:

```
/bin/bash -c "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/HEAD/install.sh)"
```

and follow the instructions.

After installing homebrew, you can check for any issues with the install by typing

`brew doctor`
On some OSX versions you may run into file permission issues with OSX’s SIP process. If you see permission issues that does not allow you to install homebrew, you might need to change ownership on the homebrew directory by typing:

```
sudo chown -R $(whoami):admin /usr/local
```

**Recommended:**
Now that homebrew is installed, we can install gcc and gfortran

First check that gcc package exists, you can type in the Terminal prompt

```
brew search gcc
```

To actually install gcc and gfortran, please type:

```
brew install gcc --without-multilib
```

The extra option “--without-multilib” adds openmp support.
You no longer need to include --without-multilib on 64-bit-only systems as it is automatically disabled but it doesn’t hurt to add it.

If you already installed gcc before, but you do not have openMP support, you will need to rather type:

```
brew reinstall gcc --without-multilib
```

Warning: the installation of gcc will take a lot of time.

Check installation after complete:

```
gcc-10 --version
gfortran --version
```
Installation on MAC:

Recommended:

Installing gnuplot using homebrew:

```
brew install gnuplot --with-x11
```

If error is returned, just type:

```
brew install gnuplot
```

If you previously unsuccessfully installed a package such as gnuplot, then first type:

```
brew uninstall gnuplot
```

followed by command

```
brew install gnuplot --with-x11
```
Finally, we will install Python with its packages. It should always be installed after Xcode was installed. Xcode comes with a stripped-down version of Python, however, this native version of python usually does not contain scipy support. To check the current python installation, you can type

```python
import scipy
scipy.test()
```

If it survives a minute or more, the installation is OK. If not, follow the instructions below.

As of this writing, the Anaconda distribution is most user friendly and easy to install, available for all operating systems:

https://www.anaconda.com

“Get Started”, “Download Anaconda installers”, “Graphical installer”

Once installed, open:

It already includes all packages we need: scipy, numpy, matplotlib, ipython.jupyter
Installation for MAC:

Recommended:

Installing `pybind11` in anaconda:

```bash
conda install -c conda-forge pybind11
```

This is one of the easier methods to speed-up python codes through C++


Installing `numba` in anaconda:

Usually it is already installed. Check:

```bash
conda list|grep numba
```

In nothing is listed, type:

```bash
conda install numba
```
Testing of our installation with a concrete example
Which language is most popular?

https://www.tiobe.com/tiobe-index/

Python is climbing up fast

C/C++ are basis of operating systems and core of most software
Comparison of languages by generating Mandelbrot set:

Wikipedia: The Mandelbrot set $M$ is defined by a family of complex quadratic polynomials $f(z) = z^2 + z_0$ where $z_0$ is a complex parameter. For each $z_0$, one considers the behavior of the sequence $(0, f(0), f(f(0)), f(f(f(0))), \cdots)$ obtained by iterating $f(z)$ starting at $z = 0$, which either escapes to infinity or stays within a disk of some finite radius. The Mandelbrot set is defined as the set of all points $z_0$ such that the above sequence does not escape to infinity.
Implementation in Fortran (mandf.f90)

INTEGER Function Mandelb(z0, max_steps)
  IMPLICIT NONE ! Every variable needs to be declared. It is very prudent to use that.
  COMPLEX*16, intent(in) :: z0
  INTEGER, intent(in)    :: max_steps
  ! locals
  COMPLEX*16 :: z
  INTEGER    :: i
  z=0.
  do i=1,max_steps
    if (abs(z)>2.) then
      Mandelb = i-1
      return
    end if
    z = z*z + z0
  end do
  Mandelb = max_steps
  return
END Function Mandelb

Main function which defines how many steps are needed before the value f(f(....f(z0))) explodes.
This is how we use the above function in the main part of the program.

Note !$OMP directives for multicore execution.

We print 2D array to the standard output, which contains 1/#steps needed before value explodes.
gfortran -O3 -fopenmp -o mandf mandf.f90

Execute and check the time:

./mandf > mand.dat

clock time : 1.30075s with wall time=0.218432s

Finally plot:

gnuplot gnu.sh
The codes produce three column output $x, y, \text{color}$ and need a plotting program to display results. In gnuplot the following command plots the output:

```
set view map
splot 'mand.dat' with p ps 3 pt 5 palette
```

or call the script by

```
gnuplot gnu.sh
```
Why do we still bother with Fortran?

↑ A lot of scientific code written in fortran → we need to use it and be able to occasionally adapt it.
↑ It is very easily integrated with Python/numpy through f2py/f2py3, hence useful in combination with Python.

↓ For todays standards, it is obsolete, i.e, developed by IBM in the 1950s (John W. Backus 1953).
↓ The language keeps changing substantially, but maintains backward compatibility, with several implementations, but no standard compiler: Intel fortran, gnu, PGI fortran,...
Bunch of includes from header files, which contain function/class definitions need to state that complex and basic printing is in std namespace

**for** loop is almost the same as **do** loop in fortran, except that in C/C++ we always start at 0 rather than 1. This is because of array index starts with a(0) and not a(1) as in fortran.

```cpp
#include <iostream>
#include <complex>
#include <ctime>
#include <vector>
#include <omp.h>
using namespace std;

int Mandelb(const complex<double>& z0, int max_steps)
{
    complex<double> z=0;
    for (int i=0; i<max_steps; i++){
        if (abs(z)>2.) return i;
        z = z*z + z0;
    }
    return max_steps;
}
```
This is how we use the above function in the main part of the program.

We use native vector<int> data, which is 1D array. C++ still does not have native 2D arrays! Blitz++ or puma can be used, but is not included in standard C++.

Note #pragma omp directives for multicore execution.

We print 1D/2D array to the standard output, which contains 1/#steps needed before value explodes.
Testing example

Compile:  
g++-10 -fopenmp -O3 -o mandc mandc.cc

Execute and check the time:

mandf > mand.dat  
clock time : 1.30075s with wall time=0.218432s

time mandc > mand.dat  
clock time : 1.256s  wall time= 0.219s

Finally plot:  
gnuplot gnu.sh

• C++ and fortran timings very similar
Alternative compilation with makefile

Both C++ and fortran code can be simultaneously compiled with a help of makefile (compilation allows optimization):

```makefile
CC = g++-10
F90 = gfortran

all : mandc mandf
    $(CC) -O3 -fopenmp -o mandc mandc.cc
    $(F90) -O3 -fopenmp -o mandf mandf.f90

mandc : mandc.cc
    $(CC) -O3 -fopenmp -o mandc mandc.cc

mandf : mandf.f90
    $(F90) -O3 -fopenmp -o mandf mandf.f90

clean :
    rm mandc mandf
```

my C++ executable is g++-10
my fortran compiler
all instructions that need to be processed
both instructions defined above but specified here
useful to know how to clean compilation
Perl code does not need to be compiled. It is interpreter. The call to subroutine is skipped due to optimization. The execution very very slow.

```perl
#!/usr/bin/perl
use Math::Complex;

$Nx=100;
$Ny=100;
$max_steps=50;

for ($i=0; $i<$Nx; $i++){
    for ($j=0; $j<$Ny; $j++){
        $x = -2. + 3.*$i/($Nx-1.);
        $y = -1. + 2.*$j/($Ny-1.);
        $z0 = $x + $y*$i;
        $z=0;
        for ($itr=0; $itr<$max_steps; $itr++){
            if (abs($z)>2.){last;}
            $z = $z*$z + $z0;
        }
        print "$x  $y  $itr \n";
    }
}
```
from scipy import *
from pylab import *
import time

def Mand(z0, max_steps):
    z = 0j
    for itr in range(max_steps):
        if abs(z)>2.:
            return itr
        z = z*z + z0
    return max_steps

if __name__ == '__main__':
    Nx = 1000
    Ny = 1000
    max_steps = 1000  #50

    ext = [-2,1,-1,1]
    t0 = time.time()

    data = zeros((Nx,Ny))
    for i in range(Nx):
        for j in range(Ny):
            x = ext[0] + (ext[1]-ext[0])*i/(Nx-1.)
            y = ext[2] + (ext[3]-ext[2])*j/(Ny-1.)
            data[i,j] = Mand(x + y*1j, max_steps)
    print('clock time: ' + str(time.time()-t0))
imshow(transpose(1./data), extent=ext)
show()
Testing examples

Execute and check the time:

```bash
./mandf > mand.dat
clock time : 1.30075s with wall time=0.218432s

./mandc > mand.dat
clock time : 1.256s wall time = 0.219s

perl mandp.pl > mand.dat
clock time : 5361.4s

python manp.py
clock time : 24.6s
```

- Both interpreters are substantially slower than compilers.
- Python is substantially faster than perl.
- C++ and fortran timings very similar, and much faster than interpreters.
Homework:

Set up your environment: C++, Python, fortran, BLAS&LAPACK (or vecLib).

- If you are familiar with coding, write your own mandelbrot version of the code.
  If not, download Mandelbrot code written in fortran, C++, perl and python. Execute them and check that they work properly.
- Test your gnuplot by plotting mandelbrot set from generated file mand.dat.
Improving Python

Python can be speed up. Here are the main strategies:

1) Find numpy/scipy routines which can replace slow python code and for loops
2) Try using numba: https://numba.pydata.org
   1) helps when there are slow for loops
   2) many repeated operations in an area
   3) one needs to experiment with numba/Python code (sometimes slower Python leads to faster numba)
3) Recode the slow loop in fortran, and use f2py
4) Recode the slow part in C++, and use pybind11
5) [In Python2 there was weave “small inline C++ code”]

For mandelbrot we do not have scipy/numpy routine.
We will first try numba: https://numba.pydata.org
Improving Python: Numba

Just need to add two lines: import + single line before function

```python
from numba import jit

@jit(nopython=True)
```

Limitations of Numba

- Numba only accelerates code that uses scalars or (N-dimensional) arrays. You can’t use built-in types like `list` or `dict` or your own custom classes.
- You can’t allocate new arrays in accelerated code.
- You can’t use recursion.

Most of those limitations are removed if using Cython.

Numba has been getting a lot better, even just over the past few months (e.g., they recently added support for generating random numbers).

But first we will rewrite Python code into a single function with two loops, which will make Numba faster and can also be used with C++/fortran to substantially speed up the code.
We now have three nested for loops
All three can be optimized with any of
available tools

The code appears slower now (90s versus 25s).
This might vary between different computers.
Improving Python:
Numba

The two lines added here

```python
from scipy import *
from numpy import *
from pylab import *
import time
from numba import jit  # This is the new line with numba

@jit(nopython=True)  # This is the second new line with numba
def MandNumba(ext, max_steps, Nx, Ny):
data = ones( (Nx,Ny) )*max_steps
    for i in range(Nx):
        for j in range(Ny):
            x = ext[0] + (ext[1]-ext[0])*i/(Nx-1.)
            y = ext[2] + (ext[3]-ext[2])*j/(Ny-1.)
            z0 = x+y*1j
            z = 0j
            for itr in range(max_steps):
                if abs(z)>2.:
                    data[j,i]=itr
                    break
                z = z*z + z0
    return data
```

Speed goes from 90s to 1.5s, i.e., 60-times.
Also, compared to previous code (25s) 16-times.

Impressive performance for the effort
*but it does not always perform so great*
Dynamic update of selected region: self-similarity

We want to create a figure, which can be dynamically zoomed-in and enlarged, to follow self-similarity of fractal plot.

Example:
Dynamic update of selected region: self-similarity

We want to create a figure, which can be dynamically zoomed-in and enlarged, to follow self-similarity of fractal plot.

We will connect a function `ax_update(ax)` with the event of changing xlim or ylim using `callbacks.connect` function, which is defined for axis class in matplotlib library.

Below are the lines we will add/change in the existing Python/Numba implementation:

```python
......
def ax_update(ax):  # actual plotting routine
    ax.set_autoscale_on(False)  # Otherwise, infinite loop

if __name__ == '__main__':
    ......
    imshow(1./data, extent=ext)
    fig, ax = subplots(1, 1)
    ax.imshow(data, extent=ext, aspect='equal', origin='lower')
    ax.callbacks.connect('xlim_changed', ax_update)
    ax.callbacks.connect('ylim_changed', ax_update)
    show()
```

Current code does not work yet, because we did not replot the fractal once the function `ax_update` is called. Now we need to find the size of the region we zoom-in, and replot the fractal for the new region.
def ax_update(ax):  # actual plotting routine
    ax.set_autoscale_on(False)  # Otherwise, infinite loop
    # Get the range for the new area
    xstart, ystart, xdelta, ydelta = ax.viewLim.bounds
    xend = xstart + xdelta
    yend = ystart + ydelta
    ext = array([[xstart, xend, ystart, yend]])
    data = MandNumba(ext, max_steps, Nx, Ny)  # actually producing new fractal

    # Update the image object with our new data and extent
    im = ax.images[-1]  # take the latest object
    im.set_data(data)  # update it with new data
    im.set_extent(ext)  # change the extent
    ax.figure.canvas.draw_idle()  # finally redraw

if __name__ == '__main__':
    ........
    #imshow(1./data, extent=ext)
    fig, ax = subplots(1, 1)
    ax.imshow(data, extent=ext, aspect='equal', origin='lower')

connecting the mouse event
with the above function
ax.callbacks.connect('xlim_changed', ax_update)
ax.callbacks.connect('ylim_changed', ax_update)
show()
Homework:

Implement the dynamic version of the mandelbrot cell
Improving Python

The idea: Write most of the code in Python. Allocate all arrays in Python, to avoid annoying bookkeeping of allocation/deallocation of memory. Speed-up the innermost loop by fortran/C++.

Python is used as ”glue” for C++ and fortran code. Can combined modules obtained by either of the two or other tools.

Several available tools:
• f2py for fortran
• **pybind11**: [https://github.com/pybind/pybind11](https://github.com/pybind/pybind11) very powerful for C++ to Python-library conversion. Needs newer C++-11 compiler. It requires only a few header files, and no libraries or compilation. Efficient, and not too hard to use.
• **weave**: was removed in Python3. It used to be part of scipy, later removed from scipy, but included in stand alone Python packages. In python3 abandoned. Very simple to use and very efficient results. But code is a string, which is very clumsy for writting more than 10 lines of code.
• **Swig**: very general. It can glue almost everything with everything. It is demanding to master.
• **PyCXX**: smaller, intented only for C/C++ -> Python conversion. Looks quite simple, but very limited numpy support.
• **Cython**: [http://cython.org/](http://cython.org/) (very popular with similar performance as numba, but much easier to port. However, a bit harder to use — almost like a new compiler for python). We do not write real C++ code, but code similar to C++, which is being compiled.
Improving Python with fortran (f2py)

Normal fortran subroutine, which can be called by fortran main code.

We can add some comments that make nicer python modules
!f2py optional
!f2py intent(hide)

!$OMP directive allows openMP parallelization
Improving Python with fortran (f2py)

We can compile fortran mandel.f90 with:

```
f2py -c mandel.f90 -f90flags='-fopenmp' -m mandel
```

which should produce mandel.so

Note: -fopenmp switches on openMP parallelization

To use openMP we need to set OMP_NUM_THREADS, for example

```
export OMP_NUM_THREADS=4
```

Finally, we write short python script and import the module like it was Python module

```
#!/usr/bin/env python
from scipy import *  # for arrays
from pylab import *  # for plotting
import mandel  # importing module created by f2py
import time

# The range of the mandelbrot plot [x0,x1,y0,y1]
ext=[-2,1,-1,1]

tc = time.process_time()  # cpu time
hw = time.time()  # wall time

data = mandel.mandelb(ext,1000,1000).transpose()

print('# wall time : ', time.time()-tw, 's  clock time : ', time.process_time()-tc, 's')

# Using python's pylab, we display pixels to the screen!
imshow(data, interpolation='bilinear', origin='lower', extent=ext, aspect=1.)
show()
```

Wall time essentially the same as for fortran native code:  wall time :  0.273 s  clock time :  3.901 s
Improving Python with C++ (pybind11)

Normal C++ code, but needs some extra header files “pybind11/“

```cpp
#include "pybind11/pybind11.h"
#include "pybind11/numpy.h"
#include "pybind11/stl.h"
#include <cstdint>
namespace py = pybind11;
namespace std;
void mand(py::array_t<double>& data, int Nx, int Ny, int max_steps, const vector<int>& ext) {
  auto dat = data.mutable_unchecked<2>();
  #pragma omp parallel for
  for (int i=0; i<Nx; i++){
    for (int j=0; j<Ny; j++){
      dat(j,i) = max_steps;
      double x = ext[0] + (ext[1]-ext[0])*i/(Nx-1.);
      double y = ext[2] + (ext[3]-ext[2])*j/(Ny-1.);
      complex<double> z0(x,y);
      complex<double> z=0;
      for (int itr=0; itr<max_steps; itr++){
        if (norm(z)>4.){
          dat(j,i) = itr;
          break;
        }
        z = z*z + z0;
      }
    }
  }
}
PYBIND11_MODULE(imanc,m) {
  m.doc() = "pybind11 wrap for mandelbrot";
  m.def("mand", &mand);
}
```

This is how to access numpy arrays in C++ through pybind11

This code is instead of main()
It is specific to pybind11 and needs some learning from pybind11 manual.
Improving Python with C++ (pybind11)

To create python module from C++ code, we type:

```
g++-10 `python3 -m pybind11 --includes` -undefined dynamic_lookup -O3 -fopenmp -shared -std=c++11 -fPIC imanc.cc -o imanc.so
```

`g++-10` must be replaced by your C++ compiler.
Notice that “python3 -m pybind11 — includes” should give correct include files of your python installation, provided pybind11 was properly installed. Otherwise one needs to find python include files manually.

Notice also that “-undefined dynamic_lookup” is needed only in mac (not linux). `-fopenmp` is for openMP parallelization.
Improving Python with C++ (pybind11)

Here we import C++ module imanc.so

This is old Numba code

Comparing pybind11 and Numba:

pybind11: walltime:  0.153 cputime:  0.852
numba:    walltime:  1.241 cputime:  1.212

C++ with pybind11 is 8-times faster, mostly because we use multiple cores

Call to C++ module

On a single core:

pybind11: walltime:  0.776 cputime:  0.776
numba:    walltime:  1.252 cputime:  1.220

C++ with pybind11 is still 60% faster, sometimes more.
Homework:

Speed up the implement mandelbrot with f2py or with pybind11.
Compilation and Linking Instructions + creating Makefiles

C/C++ and fortran code needs to be compiled before it can be run. The compilation takes two steps: producing object (machine) code from the source code, and linking objects into executable. The commands are:

- compile: `g++ [options] -c <source1>.cc`
  `g++ [options] -c <source2>.cc`
- link: `g++ [options] -o <executable> <source1>.o <source2>.o`
- execute: `./<executable>`

If compiling a single source file, we can achieve both steps with one command

```
compile&link: g++ [options] -o <executable> <source>.cc
execute: ./<executable>
```

Options can be omitted, but we will many times use options for optimization (`-O`, or `-O3`) adding debugging information (`-g`), or additig profiling information (`-p` or `-pg`)

For fortran, code we can use identical process, except `g++` is replaced by fortran compiler, i.e., either gnu-fortram `gfortran` or intel’s `ifort`.

Compilation and Linking Instructions + creating Makefiles

Python is interpreter. The code does not need explicit compilation. By invoking Python interpreter, the code is compiled on the fly and executed at the same time

```
   compile&execute: python  <script>.py
```

If we want to avoid invoking python interpreter explicitly, we need to do the following:

- change script permission:  chmod +x  <script>.py
- the first line needs to be:  #!/usr/bin/env python
- execute:  ./<script>.py
Creating Makefiles

It is a good practice to write a makefile for every project. Makefile typically contains information about the default compilers, location of necessary include files and necessary libraries to link to the executable.

There are many nice tutorials available on the Web including

https://cs.colby.edu/maxwell/courses/tutorials/maketutor/
https://www.tutorialspoint.com/makefile/index.htm
https://www.gnu.org/software/make/manual/

We will briefly describe the steps in writing simple makefiles.

But first remember:

- The name of the makefile can be ”Makefile” or ”makefile” and is typically located in the same directory as other source files.
- User types ”make” in the source directory and makefile is executed producing the executable file.
Creating Makefiles

Lets call our project **manc**. The C++ source file is **manc.cc**. The simplest makefile contains the following two lines:

```
dependency rule
manc : manc.cc
  g++ -o manc manc.cc
```

Note: Each line in the commands list must begin with a **TAB** character!

The dependency rule defines under what conditions a given file needs to be recompiled, and how to compile it.

The above rule states that the executable **manc** (is a target) has to be recompiled whenever **manc.cc** (prerequisite) is modified. The rule tells us that **manc** can be obtained by the command **g++ -o manc manc.cc** (the recipe).

We can have multiple rules, which are executed recursively.
By default, make always executes the first rule in the makefile. The other rules are executed, if they are called by some other rule (starting from the first rule).
The exception is the case when we give an argument to the make command, make will start at the rule with such name.
Creating Makefiles

Here is an example with multiple rules

```
all : manc manf  # if all does not exists, manc and manf are envoked

manc : manc.cc  # target : dependencies // time1 > time2 -> execute
g++ -o manc manc.cc  # commands

manf : manf.f90  # target : dependencies
       gfortran -o manf manf.f90  # command
```

The first rule is `all`, and make will start evaluating it.

The first lines says that `all` depends on `manc` and `manf`. If the two files do not exist, make will create them by finding and executing rules for `manc` and `manf`. Even if the two files (`manc & manf`) exist, make will check if they are up to date, otherwise it will evaluate the rules. Up to date means that prerequisites (on the right) are older than targets (on the left). For example, if `manc.cc` is newer than `manc`, the rule for `manc` will be evaluated even though `manc` exists. We could say that if the file does not exists, it is equivalent to be very old for the purpose of makefile rules evaluation.
Creating Makefiles

Next we could define some constants for compiler names and compiler flags (optimization).
For example

\begin{verbatim}
C++ = g++       # define variable C++
FORT = gfortran # define variable FORT
CFLAGS = -O3    
FFLAGS = -O3

# rules below

all : manc manf # target : prerequisite
manc : manc.cc  # target : prerequisite // time1 > time2 -> execute
  $(C++) $(CFLAGS) -o manc manc.cc # commands
manf : manf.f90 # target : prerequisite
  $(FORT) $(FFLAGS) -o manf manf.f90 # command
\end{verbatim}

This is useful for porting makefiles to different computer/operating system, as only a few variables needs to be changed on different system.
Most makefiles have a rule named `clean`. This will remove all object files and all executables, so that a fresh compilation can be started after `clean` is invoked. We would add a rule like that

```plaintext
clean :
    rm -f manc manf
```

Notice that the dependency list is empty, hence the rule is always executed when invoked.

To invoke the `clean` rule, we need to call `make` with the argument: `make clean`

Make also defined many special variables, such as `$@`, `$<`, `$*`. The variable `$@` stands for the target on the left hand side, and `$<` is the first item in the prerequisites list.

We could rewrite the `manc` and `manf` rules in the following way:

```plaintext
manc : manc.cc
    $(C++) $(CFLAGS) -o $@ $< # commands
```

```plaintext
manf : manf.f90
    $(FORT) $(FFLAGS) -o $@ $< # command
```

Notice how the two commands become almost identical when using special variables.

We can exploit this similarity of commands by writing generic rules for any pair of target:prerequisite.
Creating Makefiles

When we have many C++ and fortran files, which need to be compiled in a similar way, we can define generic rule. For example, we can define a rule that produces *xxx.o* from a corresponding *xxx.f90* file. We can achieve that by so called pattern rules, which can be added at the end of the makefile.

```
#.................
# Patter rules
#.................
%.o : %.cc
    $(C++) $(CFLAGS) -c $<
%.o : %.f90
    $(FORT) $(CFLAGS) -c $<
```

Now we do not need to write a rule for obtaining *xxx.o* from *xxx.f90* or *xxx.cc*. The pattern rule will find such files in the current directory and execute the commands.

Note that if we have multiple files with the same name, for example *xxx.f90* and also *xxx.cc*, makefile will get confused of which file to use in order to generate *xxx.o*. It will just use one of the two source files. Therefore always avoid naming multiple files with the same name.
## Why parallelization?

Top 500 computers in the world: [www.top500.org](http://www.top500.org)

<table>
<thead>
<tr>
<th>Rank</th>
<th>System</th>
<th>Cores</th>
<th>Rmax [TFlop/s]</th>
<th>Rpeak [TFlop/s]</th>
<th>Power [kW]</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Supercomputer Fugaku - Supercomputer Fugaku, A64FX 48C 2.26GHz, Tofu Interconnect D, Fujitsu RIKEN Center for Computational Science Japan</td>
<td>7,630,848</td>
<td>642,010.0</td>
<td>537,212.0</td>
<td>29,899</td>
</tr>
<tr>
<td>2</td>
<td>Summit - IBM Power System AC922, IBM POWER9 22C 3.07GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband, IBM DOE/SC/Oak Ridge National Laboratory United States</td>
<td>2,414,592</td>
<td>148,600.0</td>
<td>200,794.9</td>
<td>10,096</td>
</tr>
<tr>
<td>3</td>
<td>Sierra - IBM Power System AC922, IBM POWER9 22C 3.10GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband, IBM / NVIDIA / Mellanox DOE/NNSA/LLNL United States</td>
<td>1,572,480</td>
<td>94,640.0</td>
<td>125,712.0</td>
<td>7,438</td>
</tr>
<tr>
<td>4</td>
<td>Sunway TaihuLight - Sunway MPP, Sunway SW26010 260C 1.45GHz, Sunway, NRCPC National Supercomputing Center in Wuxi China</td>
<td>10,649,600</td>
<td>93,014.6</td>
<td>125,435.9</td>
<td>15,371</td>
</tr>
</tbody>
</table>

Fast computers have several million cores, which need to be used efficiently & simultaneously

my laptop: 8 cores, 2.4 GHz with 8 single-precision FLOPS’s per second
hence theoretical performance = 8*2.4GHz*8 = 38.4GFLOPS/s= 0.0384TFLOPS/s
This is theoretical not actual speed, the list contains actual TFLOPS by running LINPACK benchmark
Why parallelization?

Top 500 computers in the world: [www.top500.org](http://www.top500.org)

Kilo $10^3$
Mega $10^6$
Giga $10^9$
Tera $10^{12}$
Peta $10^{15}$
Exa $10^{18}$

The number of cores is exploding in the list of top 500
Why parallelization?

Processor’s speed increased linearly with small slope between 1980-1985 (1.25/year), and larger slope between 1985-2000 (1.52/year).

Processor’s speed plateaued in 2005 (people were predicting Moor’s law to break).

Instead of increasing the speed of single processor, number of processors and cores is now increasing exponentially.
Moore’s law still works!

Number of transistors is still exploding, which defines Moore’s law.

Quantum limit for single core, it cannot be too small, because it stops behaving classically.

Number of cores is exploding.
Why parallelization?

Also important is memory latency, which is improving slowly with 1.07/year. Hence memory speed is substantially slower than processor speed, and it will remain so for foreseeable future.
OpenMP and multicore execution

- OpenMP is designed for multi-processor/core to run a program on several cores (using several "threads")
- OpenMP programs accomplish parallelism exclusively through the use of threads. Typically, the number of threads matches the number of machine processors/cores. However, the actual use of threads is up to the application.
- OpenMP is a shared memory programming model, most variables in OpenMP code are visible to all threads by default.
- But sometimes private variables are necessary to avoid race conditions
- OpenMP is an explicit (not automatic) programming model, offering the programmer full control over parallelization.
- Parallelization can be as simple as taking a serial program and inserting compiler directives... Or as complex as inserting subroutines to set multiple levels of parallelism, locks and even nested locks.
openMP and multicore execution

The simplest case of parallel mandelbrot calculation:

```cpp
#pragma omp parallel for
for (int i=0; i<Nx; i++){
    for (int j=0; j<Ny; j++){
        double x = ext[0] + (ext[1]-ext[0])*i/(Nx-1.);
        double y = ext[2] + (ext[3]-ext[2])*j/(Ny-1.);
        mand[i*Ny+j] = Mandelb(complex<double>(x,y), max_steps);
    }
}
```

The loop over $i$ is parallelized. Each core is calculating different $i$ term.

Note that $\text{mand}$ array is shared across all cores, because all cores have access to the entire array, but each core is changing only its own slice of the array.

Note that $x$ and $y$ must be different on each core. As they are declared inside the loop, compiler makes them private to each core.

In more general case, the `omp parallel` statement is

```cpp
#pragma omp parallel shared(mand,ax,ay) private(beta,pi)
```

By default all variables are shared, hence `shared` statement is not really needed.
openMP and multicore execution

The same loop in fortran is:

```fortran
!$OMP PARALLEL DO PRIVATE(j,x,y,z0)
   do i=1,Nx
      do j=1,Ny
         x = ext(1) + (ext(2)-ext(1))*(i-1.)/(Nx-1.)
         y = ext(3) + (ext(4)-ext(3))*(j-1.)/(Ny-1.)
         z0 = dcmplx(x,y)
         mande(i,j) = Mandelb(z0, max_steps)
      enddo
   enddo
!$OMP END PARALLEL DO
```

Note that in fortran all variables are declared at the top of the program, hence x, y, z0, j need to be declared private. Also i is private, but the first loop counter does not need to be added to the private list, as compiler will add it automatically.

The code is compiled by adding a flag `-fopenmp`:

```bash
g++ -fopenmp -O3 -o mandc mandc.cc
```

or

```bash
gfortran -fopenmp -O3 -o mandf mandf.f90
```

Also the environment variable `OMP_NUM_THREADS` should be set to the number of cores (threads) we want to use. We can issue a command

```bash
export OMP_NUM_THREADS=4
```
openMP and multicore execution

Example of time for mandelbrot set on multiple cores for Intel Core i9 processor:
speed improves, but not close to theoretical (1/core) estimate. Why?
speed improves even beyond 8 threads, even though we have 8 cores. Why?
One more openMP example

\[ \pi = \int_0^1 \frac{4}{1 + x^2} \, dx \]

#include <iostream>
#include <ctime>
#include <cmath>
#include <omp.h>
using namespace std;

double f(double x) {
    return 4.0/(1.0+x*x);
}

double calcPi(int n) {
    const double dx = 1.0/n;
    double fSum = 0.0;
    #pragma omp parallel for reduction(+:fSum)
    for (int i=0; i<n; ++i) {
        double x = (i+0.5)*dx;
        fSum += f(x);
    }
    return fSum*dx;
}
One more openMP example

The alternative, but worse implementation:
We do not specify that fSum is obtained by reduction, but we specify that a particular line “fSum+=df” should be done without parallelization.

omp critical can be used for any line that cannot be parallelized.
openMP and multicore execution

Memory access is slow. When several cores need to manipulate few MB of data, several cores compete for the bandwidth/access to RAM and L3 cache.

CPU: ~3GHz ~ 0.3ns per tick ~ 0.04ns for floating point operation (8FP per tick)
L1 cache: latency~ 1ns, size ~16KB
L2 cache: latency~ 3ns, size ~256KB
L3 cache: latency~ 6ns, size ~2MB
RAM: latency~20ns, size ~GB, bandwidth~0.3GHz, corresponding to 3.3ns

Latency: Delay incurred when a processor accesses data inside the memory (even when reading just one number)

Bandwidth: Rate at which data can be read from or stored into memory by a processor
More realistic multicore architecture

- ~32KB L1 cache per core
- ~256KB L2 cache per core
- ~2MB L3 cache per core, but shared by all cores
- several GB RAM

Since we write data into common variable, speed is limited by memory access and not computation, hence we do not get theoretical performance.

Why do we get speedup when using more threads than cores?
Design of modern CPU

Access to memory is arranged to be staggered: some threads are doing computation and some are writing, so we can squeeze out a bit of performance by flooding CPU with threads. Notice that this is not necessary the case. Sometimes the execution is slowed down when number of threads exceeds number of cores.

If you want to learn more about openMP, consult these resources
https://www.openmp.org
https://www.openmp.org/resources/tutorials-articles/
https://www.youtube.com/channel/UCtdrEoe46tD2IvJJRs_JH1A/videos
How to improve memory management?

To squeeze out best performance can be a very hard software engineering problem, which is handled by compiler, and user does not have complete overview how memory access is handled.

However, there are some general ideas tips of how to access memory to allow compiler well optimize the code.

- Do not use hard-disc for data manipulation if possible. Keep data in RAM. If you need a lot of RAM, estimate whether it fits into RAM. Rethink your algorithm before you start writing data to hard-disc.
- Try avoiding random access of data in RAM to reduce cache misses.
- The data which you need in the innermost loop should be stored in a way that the access is maximally continuous.

Why should we access memory continuously?
Because CPU does not load a single number, but a page, which is 64 byte (8 double’s).
We can use data already present.
For reading or writing one element in the memory, a complete page of memory has to be loaded into cache.
How does memory work?

For reading or writing one element in the memory, a complete page of memory has to be loaded into cache.
How does memory work?

For reading or writing one element in the memory, a complete page of memory has to be loaded into cache.

Now the processor can read and write the elements.
How does memory work?

For reading or writing one element in the memory, a complete page of memory has to be loaded into cache.

Now the processor can read and write the elements.

If the next element is outside the loaded cache pages, another page needs to be loaded.
How does memory work?

For reading or writing one element in the memory, a complete page of memory has to be loaded into cache.

Now the processor can read and write the elements.

If the next element is outside the loaded cache pages, another page needs to be loaded.
How does memory work?

For reading or writing one element in the memory, a complete page of memory has to be loaded into cache.

Now the processor can read and write the elements.

If the next element is outside the loaded cache pages, another page needs to be loaded.
How does memory work?

For reading or writing one element in the memory, a complete page of memory has to be loaded into cache.

Now the processor can read and write the elements.

If the next element is outside the loaded cache pages, another page needs to be loaded.

Accessing an element already loaded in cache is very fast and does not cost extra cycles.
How does memory work?

For reading or writing one element in the memory, a complete page of memory has to be loaded into cache.

Now the processor can read and write the elements.

If the next element is outside the loaded cache pages, another page needs to be loaded.

Accessing an element already loaded in cache is very fast and does not cost extra cycles.

If the cache is full and a new cache page should be loaded, an old one must be dropped, which costs several hundred cycles, and is called cache miss.
How to improve memory management?

Typical example is a matrix manipulation. In C or C++, one needs to access multidimensional arrays in the following order since the data is stored in a row major order.

In Fortran, the same loop should be written in the following way

```
for (int i=0; i<size; i++)
    for (int j=0; j<size; j++)
        A[i][j] = ..... 
```

```
do i=1, size
    do j=1, size
        A(j,i) = ..... 
    enddo
endo
```

This is because Fortran (C) uses column (row) major storage. The figure explains it all.

Fig. 15.2 (Left) Row-major order used for matrix storage in C and Pascal; (right) column-major order used for matrix storage in Fortran. On the bottom is shown how successive matrix elements are stored in a linear fashion in memory.
Multi-node parallelization : MPI

When parallel execution uses several nodes (not just several cores on a single node), we need to use MPI parallelization. MPI requires one to call specialized MPI routines to communicate and exchange data. This is more technically involved programming.

Inter-node (2nd level interconnect) speed:
- InfiniBand: latency ~5μs, bandwidth ~1Gb/s
- GigaBit Ethernet: latency 60μs, bandwidth ~0.1Gb/s

Latency: Time required to send a message of size zero (time to set up communication)
Bandwidth: Rate at which large messages (>=2Mb) are transferred
Virtual box from past years (which should work if other installations fail):

If you do not want/succeed to install the necessary software, you should download the file:
http://hauleweb.rutgers.edu/downloads/509/509.ova
(warning: 4.8GB file, it might take a while)

Then you should install VirtualBox to run the provided virtual machine:
https://www.virtualbox.org

Finally, start the VirtualBox and navigate to File/Import Appliance, and choose the downloaded 509.ova file.

Then click Start and wait for the linux to start. Once linux is running, you can start a terminal Konsole and start emacs in the terminal. You can navigate to

```
cd ~/ComputationalPhysics/mandelbroat
```

and examine the files we will discuss in the first lecture. If you need username, use student, and passwd student123.
Learning Python

Next learning python from the following lectures:

https://github.com/jrjohansson/scientific-python-lectures

If you prefer video, this might be very good one:

https://www.youtube.com/watch?v=xCKfR80E8ZA