Level 7 Advanced Computational Physics
Software Installation and Programming Notes
Version 1.3a[1]

Dr. Simon Hanna
Rm. 3.44

October 7, 2021

[1]Revision 1: Substantial revisions to Chapter 2 Mac installations, with more information about Open-MPI.

Preface

This is a collection of notes that were put together while supporting students taking the fourth year Advanced Computational Physics PHYSM0032 course at the University of Bristol. They have evolved over time, as I have attempted to keep up with the changing needs of students, and the changing whims of multiple operating systems and types of computer. I don't guarantee that all of the information is either correct or up-to-date, so please do let me know if you find any mistakes.

The Advanced Computational Physics course is a course in parallel programming. While it is expected that the University supercomputer, BlueCrystal phase 4, will be used for the mini-project assessment, having a working local development environment will be crucial for writing, debugging and testing code. Mini-projects will make use of the OpenMP and MPI libraries, which are available for Windows, MacOS and Linux. OpenMP and MPI library routines are accessible from a range of languages, and you are free to use Python (Cython), C/C++ or Fortran. Notes on installation of OpenMP and MPI, various compilers and Cython will be included below.

The first chapter of these notes covers obtaining an account on BlueCrystal phase 4, and explains how to set up your working environment so that you can run parallel programs on the job queue. Subsequent chapters then cover setting up similar environments on your own Mac, PC or linux box, so that you can do the bulk of your development offline. Later chapters cover a bunch of useful topics including an introduction to Cython, tips and tricks in code optimisation and examples of benchmarking different parallelisation techniques. I hope you find them useful.

Simon Hanna
July 2021

Contents

TOPIC 1

Getting started with BlueCrystal phase 4

1.1 Obtaining an account on BlueCrystal 4

Although all of the methods discussed in the Advanced Computational Physics course can be used on a personal computer or laptop, ultimately, you will want to scale up your work to use a supercomputer. The University of Bristol currently has five such machines: BlueCrystal phases 3 & 4 (Intel-based clusters with fast interconnections); Blue Pebble (loosely-coupled Intel-based cluster); Catalyst & Isambard (Arm-based clusters). This course will use BlueCrystal phase 4 (BC4), which has several hundred compute nodes, each with 28 compute cores, making around 5000 cores in total.

You will need to request an account on BlueCrystal phase 4. Go to:

https://www.bristol.ac.uk/acrc/high-performance-computing/

and follow one of the links to the application form (in the left-hand sidebar, or the right-hand box). You will need to join the existing project for the PHYSM0032 unit—the title of the project is “PHYSM0032 Advanced Computational Physics 2021” and the project code is “IFAC025342”. Make sure that you request BlueCrystal phase 4 in the “Additional information” box, and select the “bash” preferred login shell. Accounts are usually generated within 24–48 hours.

1.2 Using BlueCrystal phase 4

BlueCrystal 4 is a Linux based cluster. It consists of over 500 compute nodes, each with two 14 core 2.4 GHz Intel E5-2680 v4 (Broadwell) Xeon CPUs, and 128 GB of RAM. In other words, each node has 28 compute cores, and approximately 4GB of memory per core. Generally commands are issued on the command line in a terminal window. Information on connecting a terminal window to BlueCrystal 4, from Macs, PCs and other computers, as well as transferring files from your own computer to BC4 and back again, is available in the various user guides and tutorials on the ACRC website. You will also find some basic Linux tutorials there:

https://www.bristol.ac.uk/acrc/high-performance-computing/hpc-documentation-support-and-training/

Once you have an account, you will need to set up your user environment, either for programming in Python, or for programming in C/C++ or Fortran. The procedure is a little different, depending on which you choose. In both cases, you are installing the programming language as well as ensuring you have access to the multi-threading library OpenMP, and the distributed processing Message Passing Interface (MPI).

1.3 Setting up a Python environment on BC4

If you are intending to work with Python / Cython in your mini project, you should follow this section. If you are intending to use C/C++ or Fortran, please refer to Section 1.4. It is not a good idea to mix the environments.

1.3.1 Installing Python

When you first login to BC4, you will need to install a module containing a modern Python installation. If you type:

```
python --version
```

at the command prompt, you will most likely see: `Python 2.7.5`, which is of no use to us. Similarly, if you type:

```
module list
```

you will most likely see the response: `No modules loaded`. To install the correct version of Python, type the command:

```
module add languages/anaconda3/2020-3.8.5
```

and you should now see several modules appear when you type `module list`:

```
Currently Loaded Modules:
  1) languages/java/sdk-1.8.0.141    5) libs/cuda/10.0-gcc-5.4.0-2.26
  2) GCCcore/5.4.0                   6) libs/cudnn/10.1-cuda-10.0
  3) binutils/2.26-GCCcore-5.4.0     7) languages/anaconda3/2020-3.8.5
  4) GCC/5.4.0-2.26
```

This collection of modules includes everything you need for running multi-threaded and distributed programs using Anaconda python. To ensure these modules are available everytime you login to BC4, you need to add a line to your `.bashrc` file.[1] In your login directory, type the command:

```
nano .bashrc
```

[1]The `.bashrc` file is an example of a hidden file in Linux. Any file beginning with a dot is hidden from normal directory listings. In order to see it in a directory listing use the command: `ls -a`.

Listing 1.1: A basic `.bashrc` file for use with Python on BC4.

```
# .bashrc
# Source global definitions
if [ -f /etc/bashrc ]; then
        . /etc/bashrc
fi

# User specific aliases and functions

# Add locally built modules
module use /mnt/storage/easybuild/modules/local

# my modules follow:
module add languages/anaconda3/2020-3.8.5
```

to open a very simple editor. You can move around the screen using the cursor keys, and type where you like. Add the line:

```
module add languages/anaconda3/2020-3.8.5
```

at the bottom of the file, and hit control-x to leave. A copy of my `.bashrc` file is given in Listing 1.1 for reference. Next time you login, your python installation will be ready for you. You can confirm this by typing `python --version` again, and you should have version 3.8.5.

1.3.2 Testing Python

As already mentioned, BC4 is a terminal-based system, so all programs need to be run from the command prompt, or via the job queue. Typically any program that runs for more than a second, needs to be executed via the job queue. This will be discussed in a later section. To quickly demonstrate that python is working, copy and paste the code in Listing 1.2 into a new file on BC4. You can do this by using `nano` with a new file name e.g. `nano newprog.py` and saving the result. Now, type:

```
python newprog.py
```

to run the program. You should see a typical "Hello, World!" output.

1.3.3 Testing the message passing interface (MPI)

Next you can test your MPI installation. In python, MPI is provided through the module mpi4py. Note that the version of mpi4py you will be using is version 3.0.3, and this is already installed on the version of Anaconda mentioned above. To verify that mpi4py is working, proceed as follows:

1. Check that you have a working `mpiexec` command by typing:

   ```
   which mpiexec
   ```

   This should point to a folder in the Anaconda Python directory i.e.:

   ```
   /mnt/storage/software/languages/anaconda/Anaconda3.8.5/bin/mpiexec
   ```

   If it doesn't, check that you haven't loaded any alternative versions of MPI through use of `module add` that might be conflicting.

2. At the command prompt, issue the command:

   ```
   mpiexec -n 5 python -m mpi4py.bench helloworld
   ```

   and check that you see output resembling:

   ```
   Hello, World! I am process 0 of 5 on bc4login1.bc4.acrc.priv.
   Hello, World! I am process 1 of 5 on bc4login1.bc4.acrc.priv.
   Hello, World! I am process 2 of 5 on bc4login1.bc4.acrc.priv.
   Hello, World! I am process 3 of 5 on bc4login1.bc4.acrc.priv.
   Hello, World! I am process 4 of 5 on bc4login1.bc4.acrc.priv.
   ```

Listing 1.2: A simple python test script.

```python
#
# Very basic python code
#
print("Hello, World!")
```

1.3.4 Testing multi-threaded Python

Multi-threading is a programming paradigm in which multiple program "threads" are executed simultaneously, ideally on separate cores of your multi-core processor. As you will hear in lectures, multi-threading in Python is a painful process. Essentially, although various multi-threading schemes exist, it is not possible to have *simultaneous* parallel threads in ordinary Python and there is no easy way around this. Various modules exist which use parallel threads "under the hood", but these are not programmed in Python. For example, `NumPy` and `SciPy` make extensive use of parallel code, but these are programmed in C or even Fortran.

The approach we will take is to use a language known as Cython. Cython is a pre-processor, which converts Python-like code into C, which is then compiled with a C compiler. The advantages of using this system are:

1. You are programming a very Python-esque code;
2. Your program runs at the speed of C, which typically gives a hundredfold speedup;
3. Your program can use the OpenMP library and run multi-threaded with further speed improvements.

Unfortunately, it turns out that there are severe limitations on how the multi-threading is used, such that it is best reserved for accelerating simple loops deep in the core of your code. This will be discussed further in lectures. For now, we want to be sure that the system is working.

Cython is part of Anaconda, and will be available if you have loaded the Anaconda module as described above. It is important to have a C compiler which is compatible with this version of Anaconda.[2] You may have noticed that one was installed alongside Anaconda when you loaded it. You can check this by typing:

```
gcc --version
```

which should give you the output:

```
gcc (GCC) 5.4.0
Copyright (C) 2015 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

`gcc` is the Gnu C compiler, and version 5.4 contains the OpenMP libraries that we need, so this looks good. Next we want to test the build process. Typically three files are needed to build and run a Cython script. These are:

1. The Cython script itself, which looks similar to Python, but has a file ending `.pyx` rather than `.py`.
2. A "setup" script, which is written in Python and has instructions on how to compile our Cython into an executable library.
3. A "run" script, also written in Python, which loads the Cython library and executes it.

Examples of each are shown in Listings 1.3, 1.4 and 1.5. You should copy each of these scripts into files on BC4, giving them the names specified in the comments.

First you need to build your Cython library from the `picalc_pyx_omp.pyx` file. You do this with the following command:

```
python setup_picalc_pyx_omp.py build_ext -fi
```

What you will notice is that the build process produced lots of warnings but, at the end of it all, if you list your files (type `ls`) you should see:

[2] Essentially this means that the compiler must be compatible with the binary libraries contained within Anaconda. On Linux systems and MacOS, this is usually gcc, whereas on Windows PCs it will be a particular version of Microsoft Visual C Compiler that is required.

Listing 1.3: A multi-threaded Cython script for calculating π.

```
#===================
# picalc_pyx_omp.pyx
#===================
import time
from libc.math cimport sqrt
from cython.parallel cimport prange
cimport openmp

def main( int nmax, int threads ):
    cdef:
        double pibyfour = 0.0
        double dx = 1.0 / nmax
        double initial , final
        int i
    print("Threads set to {:2d}".format(threads))
    initial = openmp.omp_get_wtime()

    for i in prange(nmax, nogil=True, num_threads=threads):
        pibyfour += sqrt(1-(i*dx)**2)

    final = openmp.omp_get_wtime()
    print("Elapsed time: {:8.6f} s".format(final-initial))
    pi = 4.0 * pibyfour * dx
    print("Pi = {:18.16f}".format(pi))
    return 0
```

Listing 1.4: A Python script for building the Cython library for our π calculation.

```
#=========================
# setup_picalc_pyx_omp.py
#=========================
from distutils.core import setup
from distutils.extension import Extension
from Cython.Build import cythonize

ext_modules = [
    Extension(
        "picalc_pyx_omp",
        ["picalc_pyx_omp.pyx"],
        extra_compile_args=['-fopenmp'],
        extra_link_args=['-fopenmp'],
    )
]

setup(name="picalc_pyx_omp",
      ext_modules=cythonize(ext_modules))
```

Listing 1.5: A Python script for running the π calculation.

```
#=======================
# run_picalc_pyx_omp.py
#=======================
import sys
from picalc_pyx_omp import main

if int(len(sys.argv)) == 3:
    main(int(sys.argv[1]),int(sys.argv[2]))
else:
    print("Usage: python {} <ITERATIONS> <THREADS>".format(sys.argv[0]))
```

```
build                                        picalc_pyx_omp.pyx
picalc_pyx_omp.c                             run_picalc_pyx_omp.py
picalc_pyx_omp.cpython-38-x86_64-linux-gnu.so   setup_picalc_pyx_omp.py
```

There are 3 new files. `build` is a folder with temporary files that may be ignored. `picalc_pyx_omp.c` is the pre-processed C code that has been compiled into your binary library, and `picalc_pyx_omp.cpython-38-x86_64-linux-gnu.so` is the library itself. You can ignore `picalc_pyx_omp.c` as it is no longer needed. Although the library has a convoluted name, it is quite meaningful. It tells us the library was created using Python 3.8, is suitable for an Intel x86 architecture, is 64 bit and was compiled using a Gnu compiler for use with a Linux operating system. The `.so` extension tells us it is a shared binary object. In fact only the first part of the name, before the first dot, is needed when we load the library into our Python script for execution, as you can see from Listing 1.5.

Run the program by typing the following:

```
python run_picalc_pyx_omp.py
```

and you should see the following, because we have forgotten to enter some command line arguments.

```
Usage: python run_picalc_pyx_omp.py <ITERATIONS> <THREADS>
```

The program has been written to take arguments from the command line; we need to include the number of iterations and the number of threads we want to use. Try the following:

```
python run_picalc_pyx_omp.py 10000000 10
```

and you should see something like:

```
Threads set to 10
Elapsed time: 0.007168 s
Pi = 3.1415928535526194
```

which means that your Python environment is all set up and ready for you to start coding. It was mentioned above that any program running for more than a second needs to be run on the job queue. This is a courtesy to other users, and for practical reasons because the login node is not capable of running 100's of programs at once. Use of the job queue gives you access to the full 10,000 compute cores of BC4, and will be covered in detail in Section 1.5.

1.3.5 Using NumPy and SciPy on BlueCrystal phase 4

This section will make more sense after you have read about the job queue (Section 1.5). As should have been mentioned in lectures, the `NumPy` and `SciPy` modules are very efficient in their use of resources. Unfortunately, this includes taking advantage of all available cores in the computer system on which they are running, irrespective of the resources actually requested when you put your program on the job queue. In the context of BC4, this means attempting to use all 28 cores on a node, even if these have been allocated to other users. This type of antisocial behaviour is usually blamed on the student, and has resulted in account suspension in the past.

There are two ways to avoid this situation:

1. Set an environment variable in your python script to prevent `NumPy` from being able to run on cores that have not been granted via the job queue;

2. Always ask for whole nodes (i.e. use multiples of 28 cores) so that other users' programs cannot be affected by yours.

It seems that `NumPy` on BC4 does not always respect the environment variables set in your job control script, so set them within your Python by using, for example:

```
import os
os.environ["MKL_NUM_THREADS"] = "1"
os.environ["NUMEXPR_NUM_THREADS"] = "1"
os.environ["OMP_NUM_THREADS"] = "1"
```

These lines must appear **before** the `import NumPy` statement.

1.4 Setting up a C/C++ or Fortran environment on BC4

If you are intending to work with C/C++ or Fortran in your mini project, you should follow this section. If you are intending to use Python / Cython, please refer to Section 1.3. It is not a good idea to mix the environments.

1.4.1 Installing C/C++ and Fortran

When undertaking the Advanced Computational Physics course, you need access to a modern compiler, the multi-threaded OpenMP library and the Message Passing Interface (MPI) for distributed computing. This section will outline how to make these available in BC4. The instructions are essentially the same, whether you want to use C/C++ or Fortran, because all the compilers are included in a single module. There are two sets of compilers available on BC4: the Gnu compilers and the Intel compilers. We will use the Intel compilers, because in my tests they produce code which runs at least twice as fast the the Gnu compilers. The Intel compilers are named `icc` (C), `icpc` (C++) and `ifort` (Fortran). I will illustrate this guide with C only.

If you type:

```
icc --version
```

at the command prompt, you will most likely see: `-bash: icc: command not found`, which is perfectly fine, because we have not loaded any compilers yet. Also, if you type:

```
module list
```

you will most likely see the response: `No modules loaded`. To install the correct version of C, we will install the Intel compiler module. When we do this, the system loads in all the dependencies, including everything we need for OpenMP and MPI. Type the command:

```
module load languages/intel/2017.01
```

and you should now see several modules appear when you type `module list`:

```
Currently Loaded Modules:
  1) GCCcore/5.4.0                    5) impi/2017.1.132-GCC-5.4.0-2.26
  2) binutils/2.26-GCCcore-5.4.0      6) vtune/2017.1.132-GCC-5.4.0-2.26
```

Listing 1.6: A basic `.bashrc` file for use with C/C++ or Fortran on BC4.

```
# .bashrc
# Source global definitions
if [ -f /etc/bashrc ]; then
        . /etc/bashrc
fi

# User specific aliases and functions

# Add locally built modules
module use /mnt/storage/easybuild/modules/local

# my modules follow:
module load languages/intel/2017.01
```

```
  3) icc/2017.1.132-GCC-5.4.0-2.26      7) imkl/2017.1.132-GCC-5.4.0-2.26
  4) ifort/2017.1.132-GCC-5.4.0-2.26    8) languages/intel/2017.01
```

This collection of modules includes everything you need for running multi-threaded and distributed programs using C/C++ or Fortran. To ensure these modules are available everytime you login to BC4, you need to add a line to your `.bashrc` file.[3] In your login directory, type the command:

```
nano .bashrc
```

to open a very simple editor. You can move around the screen using the cursor keys, and type where you like. Add the line:

```
module load languages/intel/2017.01
```

at the bottom of the file, and hit control-x to leave. A copy of my `.bashrc` file is given in Listing 1.6 for reference. Next time you login, your C/C++ and Fortran installation will be ready for you. You can confirm this by typing `icc --version` again, and you should have version 17.0.1.

1.4.2 Testing single or multithreaded C/C++ or Fortran

We will demonstrate the use of the compilers using C as an example. If you are planning to use C++ or Fortran, the commands are very similar, but talk to Dr Hanna for advice.

Listing 1.7 shows a simple multi-threaded C program, using the OpenMP library. You should copy and paste this into a file on BC4, for example using the `nano` editor, saving it with the name given in the comments.

You can compile the C code into an executable using the command:

```
icc picalc_c_omp.c -o picalc_c_omp -qopenmp -O3 -xHost
```

This produces an executable called `picalc_c_omp` which you can run by typing:[4]

```
./picalc_c_omp
```

You should see the following, because we have forgotten to enter some command line arguments.

```
Usage: ./picalc_c_omp <ITERATIONS> <THREADS>
```

[3]The `.bashrc` file is an example of a hidden file in Linux. Any file beginning with a dot is hidden from normal directory listings. In order to see it in a directory listing use the command: `ls -a`.

[4]Note that the `./` are important here as they tell the system to look in the current working directory for the executable.

Listing 1.7: A multi-threaded C program using OpenMP for calculating π.

```c
//================
// picalc_c_omp.c
//================
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <omp.h>

int main( int argc, char* argv[] )
{
    if (argc < 3){
        printf("Usage: %s <ITERATIONS> <THREADS>\n",argv[0]);
        return 0;
        }
    int nmax = atoi(argv[1]);
    int threads = atoi(argv[2]);
    double pi, pibyfour = 0.0;
    double initial, final;
    double dx = 1.0 / nmax;

    omp_set_num_threads( threads );
    printf("Threads set to %d\n",threads);
    initial = omp_get_wtime();
    int i;
#pragma omp parallel private(i)
{
#pragma omp for reduction(+:pibyfour)
    for( i=0; i < nmax ; i++ ){
        pibyfour += sqrt(1-pow(i*dx,2));
    }
}
    final = omp_get_wtime();
    printf("Elapsed %8.6f s\n",final-initial);
    pi = 4.0 * pibyfour * dx;
    printf("Pi = %18.16f\n",pi);
    return 0;
}
```

Listing 1.8: A distributed C program using MPI to say "hello" from different processors.

```
//=============
// hello_mpi.c
//=============
#include <stdio.h>
#include "mpi.h"

int main(int argc, char *argv[]) {
  int numprocs, rank, namelen;
  char processor_name[MPI_MAX_PROCESSOR_NAME];

  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Get_processor_name(processor_name, &namelen);

  {
    printf("Hello from process %d out of %d on %s\n",
           rank, numprocs, processor_name);
  }

  MPI_Finalize();
}
```

The program has been written to take arguments from the command line; we need to include the number of iterations and the number of threads we want to use. Try the following:

```
./picalc_c_omp 10000000 10
```

and you should see something like:

```
Threads set to 10
Elapsed 0.003554 s
Pi = 3.1415928535526096
```

which means that your C environment is all set up and ready for you to start coding. It was mentioned above that any program running for more than a second needs to be run on the job queue. This is a courtesy to other users, and for practical reasons because the login node is not capable of running 100's of programs at once. Use of the job queue gives you access to the full 10,000 compute cores of BC4, and will be covered in detail in Section 1.5. The following section explains how to compile and run using the MPI library.

1.4.3 Running distributed codes using MPI with C/C++ or Fortran

Special commands are available for compiling programs with the MPI libraries. These are `mpiicc` for C, `mpiicpc` for C++ and `mpiifort` for Fortran. These commands are actually just wrappers, which run the underlying Intel compiler with the appropriate set of flags. For testing MPI, we need a simple example, such as that in Listing 1.8. Copy this code into a file on BC4, and issue the following command to compile it:

```
mpiicc -O3 -o hello_mpi -xHost hello_mpi.c
```

This will produce an executable called `hello_mpi` which must be executed using the command `mpiexec`. To verify that `mpiexec` is working, proceed as follows:

1. Check that you have a working `mpiexec` command by typing:

   ```
   which mpiexec
   ```

This should point to a folder in the Intel MPI directory i.e.:

```
/mnt/storage/apps/intel/impi/2017.1.132/bin64/mpiexec
```

If it doesn't, check that you haven't loaded any alternative versions of MPI through use of `module add` that might be conflicting.

2. At the command prompt, issue the command:

```
mpiexec -n 5 ./hello_mpi
```

and check that you see output resembling:

```
Hello from process 0 out of 5 on bc4login1.bc4.acrc.priv
Hello from process 2 out of 5 on bc4login1.bc4.acrc.priv
Hello from process 4 out of 5 on bc4login1.bc4.acrc.priv
Hello from process 1 out of 5 on bc4login1.bc4.acrc.priv
Hello from process 3 out of 5 on bc4login1.bc4.acrc.priv
```

If you have any problems seeing this output, please consult Dr Hanna.

1.5 Using the BC4 job queue

1.5.1 Introduction to the queueing system and some terminology

All programs lasting more than about a second need to be run on the BlueCrystal job queue. The job queue is the mechanism by which programs are executed on the supercomputer. The idea is that you submit a script (or job) to the queue, which contains instructions on how to run your program. Note, you do not submit the program itself, just the instructions. We often call this script the “run script”. The queueing system then looks at the available resources, and matches jobs to resources to make best use of the supercomputer. Our job script therefore needs to carefully detail exactly what resources we require.

The main resources that are of interest are (a) the number of compute cores needed and (b) the amount of time needed, as measured in hours, minutes and seconds by the clock on the wall. Secondary resources that may be of interest include the amount of memory, and whether one or more GPUs are needed. We will focus on compute cores and wall time.

There are several different queues which access different groups of processors dedicated to different types of job. One section of BC4 is dedicated to single processor (serial) jobs, and another section is for parallel jobs. Two other sections have processors with extra memory and GPUs. These different sections are accessed by queues with different names.

The queueing system in use on BC4 is known as Slurm, and there is some important terminology to understand in using it:

Node: A physical computer sitting in a rack in the computer room. There are around 500 of these in BC4. A typical node contains 2 CPUs and some memory (RAM). Some nodes have GPUs as well. Each CPU in BC4 contains 14 compute cores, which equates to 28 CPU cores per node. Multi-threaded programs will generally be limited to a single node, and hence to 28 threads.

Task: Each compute task in Slurm refers to a process running on a node. You might run 28 tasks on a node which communicate with each other via MPI or you might run 1 task which itself spins up 28 threads. A serial program will consist of a single task with a single thread, running on a single compute core of one node.

CPU core: A CPU core is the compute core that actually does the work. Most of the time, Slurm refers to a CPU core as a CPU. For example, the compute nodes on BC4 have two physical CPU chips (some would describe this as having two sockets), with 14 processing cores inside each; Slurm would say that this node has 28 CPUs.

Partition or queue: A set of nodes with associated restrictions on use. This is called a queue in some systems. For example we have a test partition which only allows small, short jobs and we have a cpu partition which allows large, long parallel jobs. Type `sinfo -s` to see a list of all the partitions you have access to:

```
PARTITION        AVAIL  TIMELIMIT   NODES(A/I/O/T)  NODELIST
veryshort           up    6:00:00        9/14/1/24  compute[084-103],highmem[10-13]
test                up    1:00:00      418/1/7/426  compute[104-525],highmem[10-13]
cpu_test            up    1:00:00          4/0/0/4  compute[080-083]
cpu*                up 14-00:00:0      415/0/7/422  compute[104-525]
hmem                up 14-00:00:0          7/1/0/8  highmem[10-17]
gpu                 up 7-00:00:00        21/9/1/31  gpu[01-31]
gpu_veryshort       up    2:00:00          0/1/0/1  gpu32
serial              up 3-00:00:00        12/0/0/12  compute[068-079]
dcv                 up 14-00:00:0          1/0/0/1  bc4vis1
serial_verylong     up 14-00:00:0          8/0/0/8  compute[068-075]
```

The time limit on each partition is given (dd-hh:mm:ss), as are the nodes allocated to each partition. Note that the `test` partition overlaps with the `cpu` partition, so they are both equally busy (but with different priorities), but some of the other partitions have dedicated nodes. If you are running serial (single compute core) programs, you can consider using either the `test`, `serial` or `serial_verylong` partitions. On the other hand, if you are running parallel jobs with 14 or more cores, use the `test`, `cpu_test` or `cpu` partitions. The `cpu` partition is the default partition – the one you get if you forget to specify it. For jobs using 2 to 13 cores, you will need to use the `test` partition, or else request 14 cores on one of the parallel partitions, but within your code, tell it to use fewer cores.

There are a number of parameters to be set for Slurm, for every job that is submitted. These are:

- the name of the job (`--job-name`), which you choose for easy identification,
- the number of nodes the job requires (`--nodes`),
- the number of tasks to run on each node (`--ntasks-per-node`),
- the number of CPUs (CPU cores) to devote to each task (`--cpus-per-task`),
- the time the job will run for (`--time`),
- and the amount of memory the job will require per node (`--mem`) or per CPU core (`--mem-per-cpu`).

How you chose these parameters will vary depending on whether your code is serial, multi-threaded (OpenMP) or distributed (MPI), or some hybrid of all of these. The job time is the wall-time required for completion of the program, and your code will abort if you exceed this. On the other hand, you should choose as low a value as reasonable, to ensure Slurm schedules your program to run quickly. Similarly your program will abort if you exceed the memory requested. We will go through the options for each of the variables in the sections below. For more detailed in formation on the Slurm scheduler, please see:

https://www.acrc.bris.ac.uk/protected/bc4-docs/scheduler/index.html

1.5.2 Running serial code on the BC4 job queue

When running serial code, the key parameters are:

number of nodes required:	1
number of tasks per node:	1
number of cpus (cores) per task:	1

Listings 1.9 and 1.10 show typical scripts for running serial jobs, for Python and C codes respectively. You should copy the appropriate script to your BC4 account and then you can submit your job using:

```
sbatch myserialscript.sh
```

and you should see a message looking similar to this:

```
Submitted batch job 4550665
```

where the number given is unique for each job you run. There are a number of points to bear in mind:

- As indicated above, serial programs can be run on either the `test` (up to an hour), `serial` (up to 3 days) or `serial_verylong` (up to 14 days) partitions. Serial jobs cannot be run on the default `cpu` partition.
- The line: `cd $SLURM_SUBMIT_DIR` changes directory to the one you were looking at when you submitted your job. This is a useful way of ensuring that the script, your program, and your output all end up in the same directory. Typically I make a separate directory for each job I want to run, as a convenient way of keeping track of simulations and results.
- The time specified is 10 seconds, which is very brief, but this is only a test.
- The job name can be anything you like to help keep track of your jobs on the queue.
- You should include any modules you need within the script i.e. don't rely on your `.bashrc` as this may not include everything you need for a specific job.
- If you are running a python script, don't forget to issue the python command e.g.:

  ```
  python picalc_py.py 10000000
  ```

- If you are running a C/C++ or Fortran program, it is the compiled executable file that needs to be run, and you need to specify the path to the file (typically `./`).
- In the examples given, I am running a program for calculating π which takes a command line argument – 10000000 in the present case. Command line arguments are a useful way to get parameters into your code, to avoid you having to keep changing your program.
- All output from your program will be sent to a file, rather than the terminal window. The file will be named `slurm-<jobid>.out` (e.g. `slurm-4550665.out`) and will be found in the directory where your code is running.

Once your job is on the queue, there are a number of useful commands for following it's progress:

List all current jobs for a user:	squeue -u <username>
List all running jobs for a user:	squeue -u <username> -t RUNNING
List all pending jobs for a user:	squeue -u <username> -t PENDING
List detailed information for a job (useful for troubleshooting):	scontrol show jobid -dd <jobid>
To cancel one job:	scancel <jobid>
To cancel all the jobs for a user:	scancel -u <username>
To cancel all the pending jobs for a user:	scancel -t PENDING -u <username>

where <jobid> refers to the unique job number and <username> is your Bristol user ID.

Listing 1.9: A Slurm bash script for running a serial python program on the job queue.

```bash
#!/bin/bash
# ======================
# pythonserialscript.sh
# ======================

#SBATCH --job-name=test_job
#SBATCH --partition=test
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=1
#SBATCH --cpus-per-task=1
#SBATCH --time=0:0:10
#SBATCH --mem=100M

# Load modules required for runtime e.g.
module add languages/anaconda3/2020-3.8.5

cd $SLURM_SUBMIT_DIR

# Now run your program with the usual command
python picalc_py.py 10000000
```

Listing 1.10: A Slurm bash script for running a serial C program on the job queue.

```bash
#!/bin/bash
# =================
# cserialscript.sh
# =================

#SBATCH --job-name=test_job
#SBATCH --partition=test
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=1
#SBATCH --cpus-per-task=1
#SBATCH --time=0:0:10
#SBATCH --mem=100M

# Load modules required for runtime e.g.
module load languages/intel/2017.01

cd $SLURM_SUBMIT_DIR

# Now run your program with the usual command
./picalc_c 10000000
```

1.5.3 Running multi-threaded (OpenMP) jobs on BC4

You should first familiarise yourself with the operation of the Slurm job queue, described in Sections 1.5.1 and 1.5.2. When running multi-threaded code, the key parameters are:

number of nodes required:	1
number of tasks per node:	1
number of cpus (cores) per task:	p, where $\begin{cases} 14 \leqslant p \leqslant 28 & \text{cpu and } \texttt{cpu\_test} \text{ partition} \\ 1 < p \leqslant 28 & \texttt{test} \text{ partition} \end{cases}$

Example scripts for submitting Python (Cython) and C code using OpenMP are given in Listings 1.11 and 1.12. There are a few points to bear in mind here:

- p will be the number of threads used, and you are limited to running on a single node of BC4.
- If you are using a Cython script, make sure you have built this first at the command line, and that the shared library generated is available in the same directory where you are executing your code.
- If you are using a compiled language, make sure you have compiled your executable with the appropriate (–qopenmp) flag.
- The `export` statement is used to set the maximum number of threads available to your program. If you query the maximum number of threads in your code, this is the number which will be reported.
- The programs in the examples both read in command line arguments. The last of these is used to determine how many threads your code actually uses. This is a handy way of controlling the number of threads for benchmarking purposes; if you place repeated execution lines in your script, they will run one after the other, and you can pass different parameters each time.

1.5.4 Running distributed (MPI) jobs on BC4

You should first familiarise yourself with the operation of the Slurm job queue, described in Sections 1.5.1 and 1.5.2. When running distributed (MPI) code, the key parameters are:

number of nodes required:	n
number of tasks per node:	t, where $\begin{cases} 14 \leqslant t \leqslant 28 & \text{cpu and } \texttt{cpu\_test} \text{ partition} \\ 1 < t \leqslant 28 & \texttt{test} \text{ partition} \end{cases}$
number of cpus (cores) per task:	1

Example scripts for submitting Python (mpi4py) and C code using MPI are given in Listings 1.13 and 1.14. There are a couple of points to bear in mind compared with previous sections:

- The total number of MPI tasks used will be $n \times t$.
- It's probably best to limit n to about 5, or you will wait a very long time for your programs to run.
- Slurm makes use of the `srun` command rather than `mpiexec` when running code compiled with the Intel compilers. The `export` command is used to link `srun` to the correct libraries.
- In the `mpi4py` example, it is very important the the number of tasks requested in the `mpiexec` command is equal to $n \times t$.
- In the C example, it is important that the C code has been compiled using `mpiicc`.

Listing 1.11: A Slurm bash script for running a multi-threaded Cython program with OpenMP on the job queue.

```
#!/bin/bash
# ======================
# pythonopenmpscript.sh
# ======================

#SBATCH --job-name=test_job
#SBATCH --partition=test
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=1
#SBATCH --cpus-per-task=28
#SBATCH --time=0:0:10
#SBATCH --mem=100M

# Load modules required for runtime e.g.
module add languages/anaconda3/2020-3.8.5

cd $SLURM_SUBMIT_DIR
export OMP_NUM_THREADS=${SLURM_CPUS_PER_TASK}

# Now run your program with the usual command
python run_picalc_pyx_omp.py 1000000000 28
```

Listing 1.12: A Slurm bash script for running a multi-threaded C program with OpenMP on the job queue.

```
#!/bin/bash
# =================
# copenmpscript.sh
# =================

#SBATCH --job-name=test_job
#SBATCH --partition=test
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=1
#SBATCH --cpus-per-task=28
#SBATCH --time=0:1:00
#SBATCH --mem=100M

# Load modules required for runtime e.g.
module load languages/intel/2017.01

cd $SLURM_SUBMIT_DIR
export OMP_NUM_THREADS=${SLURM_CPUS_PER_TASK}

# Now run your program with the usual command
./picalc_c_omp 1000000000 28
```

Listing 1.13: A Slurm bash script for running a distributed Python program with mpi4py on the job queue.

```bash
#!/bin/bash
# =================
# mpi4pyscript.sh
# =================

#SBATCH --job-name=test_job
#SBATCH --partition=test
#SBATCH --nodes=4
#SBATCH --ntasks-per-node=4
#SBATCH --cpus-per-task=1
#SBATCH --time=0:0:10
#SBATCH --mem-per-cpu=100M

# Load modules required for runtime e.g.
module add languages/anaconda3/2020-3.8.5

cd $SLURM_SUBMIT_DIR

# Note the need to MANUALLY specify the number of tasks
# (in this case 4 x 4)
mpiexec -n 16 python mpi_heat2D.py
# mpiexec -n 16 python -m mpi4py.bench helloworld
```

Listing 1.14: A Slurm bash script for running a distributed C program with MPI on the job queue.

```bash
#!/bin/bash
# =================
# cmpiscript.sh
# =================

#SBATCH --job-name=test_job
#SBATCH --partition=test
#SBATCH --nodes=4
#SBATCH --ntasks-per-node=4
#SBATCH --cpus-per-task=1
#SBATCH --time=0:0:10
#SBATCH --mem-per-cpu=100M

# Load modules required for runtime e.g.
module load languages/intel/2017.01

export I_MPI_PMI_LIBRARY=/usr/lib64/libpmi.so

cd $SLURM_SUBMIT_DIR

srun ./hello_mpi
```

Listing 1.15: A simple example combining OpenMP and MPI in C.

```c
//================
// compmpihello.c
//================

#include <stdio.h>
#include "mpi.h"
#include <omp.h>

int main(int argc, char *argv[]) {
  int numprocs, rank, namelen;
  char processor_name[MPI_MAX_PROCESSOR_NAME];
  int iam, np;

  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Get_processor_name(processor_name, &namelen);

  #pragma omp parallel default(shared) private(iam, np)
  {
    np = omp_get_num_threads();
    iam = omp_get_thread_num();
    printf("Hello from thread %d out of %d from process %d out of %d on %s\n",
           iam, np, rank, numprocs, processor_name);
  }

  MPI_Finalize();
}
```

1.5.5 Combined OpenMP and MPI jobs on BlueCrystal phase 4

You should first familiarise yourself with the operation of the Slurm job queue, described in Sections 1.5.1 and 1.5.2. When running a combination of MPI and OpenMP code, the key parameters are:

number of nodes required:	n
number of tasks per node:	t
number of cpus (cores) per task:	p, where $p \times t \leqslant 28$

- The total number of CPU cores in use with this approach will be $n \times t \times p$.
- A common model would be one in which each MPI task would be allocated to a separate node, while each node could run up to 28 OpenMP threads with shared memory. In this scheme, we would set $t = 1$.
- In python, you can only use OpenMP as part of Cython. This means that you will need to Cythonise your mpi4py code. In practice this is not difficult, as will be discussed in lectures.

We will look at the C and Python solutions separately.

OpenMP and MPI in a C program

Listing 1.15 shows an example C program using both OpenMP and MPI. The code should be compiled on BC4 using both the `mpiicc` compiler and the `-qopenmp` flag:

Listing 1.16: A Slurm bash script for running a C program combining OpenMP with MPI on the job queue.

```
#!/bin/bash
# ================
# ompmpiscript.sh
# ================

#SBATCH --job-name=test_job
#SBATCH --partition=test
#SBATCH --nodes=2
#SBATCH --ntasks-per-node=4
#SBATCH --cpus-per-task=7
#SBATCH --time=0:0:10
#SBATCH --mem=100M

# Load modules required for runtime e.g.
module load languages/intel/2017.01

export I_MPI_PMI_LIBRARY=/usr/lib64/libpmi.so
export OMP_NUM_THREADS=${SLURM_CPUS_PER_TASK}

cd $SLURM_SUBMIT_DIR

srun ./compmpihello
```

```
mpiicc -O3 -o compmpihello -qopenmp -xHost compmpihello.c
```

If you run this code on the command line on the login node, specifying the number of tasks, t, `mpiexec` will choose the number of threads p per task so that $t \times p \leqslant 28$. Listing 1.16 gives a suitable job submission script. The point to note here is that `sbatch` is fussy about which combinations of n, t and p it will allow. Generally, it is preferable to work with complete nodes, but there are exceptions. For example, $n = t = p = 2$ works fine (8 threads total) and so does $n = 2$, $t = 4$, $p = 7$ (56 threads total), but $n = t = p = 4$ is not accepted. Some experimentation is clearly needed.

OpenMP and MPI in a Cython program

Running OpenMP and MPI together in Python is a little more tricky than in C. One approach is to use Cython on a full mpi4py-based program, using OpenMP in one of the inner loops to get additional speed. However, in this example, we take a slightly simpler approach: we use Cython for an inner function which includes threads, and then we call this from a Python and mpi4py-based outer wrapper. Both approaches are equally valid - I understand mpi4py was written and compiled using Cython, so there is no problem with compatibility.

Listing 1.17 shows a Cython script for a function that opens a parallel multi-threaded section so that each thread can print a message. The number of threads used is the default for the system, which is set with the environment variable `OMP_NUM_THREADS`. Once it has been compiled into a library (not forgetting to load Anaconda first), it can be called from the mpi4py-based Python program in Listing 1.18. This program simply runs the same set of queries on each MPI task and passes the information into the Cython function. You can run this quite easily at the command prompt using for example:

```
mpiexec -n 2 python run_hellocombi.py
```

If you haven't set `OMP_NUM_THREADS`, it will default to 28, and you will see 56 messages on your screen i.e. the number of threads allocated is not limited to the number of cores available. *For this reason it is very important to set the number of threads correctly in your job submission script, to avoid grabbing cores from other users!*

Listing 1.17: A multi-threaded Cython/OpenMP program for use with an mpi4py wrapper.

```
#================
# hellocombi.pyx
#================
from cython.parallel cimport parallel,threadid
cimport openmp

def printinfo( rank, size, name ):
    cdef int num_threads,thread
    with nogil, parallel():
        num_threads = openmp.omp_get_num_threads()
        thread = threadid()
        with gil:
            print("This is thread {} of {} threads, on rank {} out of {} tasks, running on
                {}".format(thread,num_threads,rank,size,name))
    return 0
```

Listing 1.18: An mpi4py-based Python script for calling a Cython/OpenMP function.

```
#=======================
# run_picalc_pyx_omp.py
#=======================
from hellocombi import printinfo
from mpi4py import MPI

size = MPI.COMM_WORLD.Get_size()
rank = MPI.COMM_WORLD.Get_rank()
name = MPI.Get_processor_name()

printinfo(rank,size,name)
```

Listing 1.19: A Slurm bash script for running a combined Cython/OpenMP/mpi4py program on the job queue.

```
#!/bin/bash
# ===================
# ompmpi4pyscript.sh
# ===================

#SBATCH --job-name=test_job
#SBATCH --partition=test
#SBATCH --nodes=2
#SBATCH --ntasks-per-node=4
#SBATCH --cpus-per-task=7
#SBATCH --time=0:0:10
#SBATCH --mem=100M

# Load modules required for runtime e.g.
module add languages/anaconda3/2020-3.8.5
export OMP_NUM_THREADS=${SLURM_CPUS_PER_TASK}
cd $SLURM_SUBMIT_DIR

# Note the need to MANUALLY specify the number of tasks (in this case 2 x 4 = 8)
# Each task will run with OMP_NUM_THREADS = cpus_per_task = 7
# so we will use 2 full nodes, 28 cores per node.
mpiexec -n 8 python run_hellocombi.py
```

A suitable job submission script is shown in Listing 1.19. As noted in the previous section, the total number of cores reserved will be $n \times t \times p$; not every combination of n, t and p works, but multiples of whole nodes (28 cores) are favoured. Running this script on the queue results in the following output:

```
This is thread 0 of 7 threads, on rank 4 out of 8 tasks, running on compute095.bc4.acrc.priv
This is thread 5 of 7 threads, on rank 4 out of 8 tasks, running on compute095.bc4.acrc.priv
This is thread 1 of 7 threads, on rank 4 out of 8 tasks, running on compute095.bc4.acrc.priv
This is thread 4 of 7 threads, on rank 4 out of 8 tasks, running on compute095.bc4.acrc.priv
This is thread 6 of 7 threads, on rank 4 out of 8 tasks, running on compute095.bc4.acrc.priv
This is thread 2 of 7 threads, on rank 4 out of 8 tasks, running on compute095.bc4.acrc.priv
This is thread 3 of 7 threads, on rank 4 out of 8 tasks, running on compute095.bc4.acrc.priv
This is thread 0 of 7 threads, on rank 5 out of 8 tasks, running on compute095.bc4.acrc.priv
This is thread 3 of 7 threads, on rank 5 out of 8 tasks, running on compute095.bc4.acrc.priv
This is thread 2 of 7 threads, on rank 5 out of 8 tasks, running on compute095.bc4.acrc.priv
This is thread 6 of 7 threads, on rank 5 out of 8 tasks, running on compute095.bc4.acrc.priv
This is thread 5 of 7 threads, on rank 5 out of 8 tasks, running on compute095.bc4.acrc.priv
This is thread 4 of 7 threads, on rank 5 out of 8 tasks, running on compute095.bc4.acrc.priv
This is thread 1 of 7 threads, on rank 5 out of 8 tasks, running on compute095.bc4.acrc.priv
This is thread 0 of 7 threads, on rank 7 out of 8 tasks, running on compute095.bc4.acrc.priv
This is thread 6 of 7 threads, on rank 7 out of 8 tasks, running on compute095.bc4.acrc.priv
This is thread 1 of 7 threads, on rank 7 out of 8 tasks, running on compute095.bc4.acrc.priv
This is thread 4 of 7 threads, on rank 7 out of 8 tasks, running on compute095.bc4.acrc.priv
This is thread 5 of 7 threads, on rank 7 out of 8 tasks, running on compute095.bc4.acrc.priv
This is thread 3 of 7 threads, on rank 7 out of 8 tasks, running on compute095.bc4.acrc.priv
This is thread 2 of 7 threads, on rank 7 out of 8 tasks, running on compute095.bc4.acrc.priv
This is thread 0 of 7 threads, on rank 6 out of 8 tasks, running on compute095.bc4.acrc.priv
This is thread 1 of 7 threads, on rank 6 out of 8 tasks, running on compute095.bc4.acrc.priv
This is thread 6 of 7 threads, on rank 6 out of 8 tasks, running on compute095.bc4.acrc.priv
This is thread 4 of 7 threads, on rank 6 out of 8 tasks, running on compute095.bc4.acrc.priv
This is thread 3 of 7 threads, on rank 6 out of 8 tasks, running on compute095.bc4.acrc.priv
This is thread 2 of 7 threads, on rank 6 out of 8 tasks, running on compute095.bc4.acrc.priv
This is thread 5 of 7 threads, on rank 6 out of 8 tasks, running on compute095.bc4.acrc.priv
This is thread 0 of 7 threads, on rank 1 out of 8 tasks, running on compute094.bc4.acrc.priv
This is thread 3 of 7 threads, on rank 1 out of 8 tasks, running on compute094.bc4.acrc.priv
This is thread 2 of 7 threads, on rank 1 out of 8 tasks, running on compute094.bc4.acrc.priv
This is thread 4 of 7 threads, on rank 1 out of 8 tasks, running on compute094.bc4.acrc.priv
This is thread 1 of 7 threads, on rank 1 out of 8 tasks, running on compute094.bc4.acrc.priv
This is thread 5 of 7 threads, on rank 1 out of 8 tasks, running on compute094.bc4.acrc.priv
This is thread 6 of 7 threads, on rank 1 out of 8 tasks, running on compute094.bc4.acrc.priv
This is thread 0 of 7 threads, on rank 2 out of 8 tasks, running on compute094.bc4.acrc.priv
This is thread 1 of 7 threads, on rank 2 out of 8 tasks, running on compute094.bc4.acrc.priv
This is thread 3 of 7 threads, on rank 2 out of 8 tasks, running on compute094.bc4.acrc.priv
This is thread 6 of 7 threads, on rank 2 out of 8 tasks, running on compute094.bc4.acrc.priv
This is thread 5 of 7 threads, on rank 2 out of 8 tasks, running on compute094.bc4.acrc.priv
This is thread 2 of 7 threads, on rank 2 out of 8 tasks, running on compute094.bc4.acrc.priv
This is thread 4 of 7 threads, on rank 2 out of 8 tasks, running on compute094.bc4.acrc.priv
This is thread 0 of 7 threads, on rank 0 out of 8 tasks, running on compute094.bc4.acrc.priv
This is thread 2 of 7 threads, on rank 0 out of 8 tasks, running on compute094.bc4.acrc.priv
This is thread 5 of 7 threads, on rank 0 out of 8 tasks, running on compute094.bc4.acrc.priv
This is thread 3 of 7 threads, on rank 0 out of 8 tasks, running on compute094.bc4.acrc.priv
This is thread 1 of 7 threads, on rank 0 out of 8 tasks, running on compute094.bc4.acrc.priv
This is thread 6 of 7 threads, on rank 0 out of 8 tasks, running on compute094.bc4.acrc.priv
This is thread 4 of 7 threads, on rank 0 out of 8 tasks, running on compute094.bc4.acrc.priv
This is thread 0 of 7 threads, on rank 3 out of 8 tasks, running on compute094.bc4.acrc.priv
This is thread 4 of 7 threads, on rank 3 out of 8 tasks, running on compute094.bc4.acrc.priv
This is thread 3 of 7 threads, on rank 3 out of 8 tasks, running on compute094.bc4.acrc.priv
This is thread 6 of 7 threads, on rank 3 out of 8 tasks, running on compute094.bc4.acrc.priv
This is thread 5 of 7 threads, on rank 3 out of 8 tasks, running on compute094.bc4.acrc.priv
This is thread 1 of 7 threads, on rank 3 out of 8 tasks, running on compute094.bc4.acrc.priv
This is thread 2 of 7 threads, on rank 3 out of 8 tasks, running on compute094.bc4.acrc.priv
```

As you can see, there are 56 threads distributed between 2 nodes, `compute094` and `compute095`. Getting jobs to work on the BC4 queue can be a bit puzzling or even frustrating at first. So, if you have any problems using these scripts, please contact Dr. Hanna or a demonstrator for advice.

TOPIC 2

Using OpenMP and MPI on a Mac

2.1 Introduction – a warning about Macs

While it is expected that most simulations for this unit will be carried out on Blue Crystal, it is important to have a development environment on your local computer, to speed the writing and debugging process. Macs are very popular development machines, and are perfectly capable of running all of the libraries we need for the PHYSM0032 Advanced Computational Physics unit. However, there are a few issues with them, the biggest being that every update to the operating system causes changes to where libraries are stored and this will lead to some initial confusion until everything is sorted out. Apple updates its operating system just at the start of the academic year, which means that it is hard to provide fully up-to-date instructions. This year, Apple have also introduced machines with an Arm-based M1 chip, for which there is currently no version of Anaconda.

Therefore, if you are using a Mac, as increasing numbers of students do, please be aware that the following notes are probably out-of-date. I hope they will at least point you in the right direction, and I will be happy to work with you to get your development environments working, and to update these notes.

There are two main sections to follow, depending on whether you intend to program with Python, or whether you prefer a compiled language, such as C or C++.

2.2 Setting up your Python environment on a Mac

The School of Physics recommends Anaconda Python, and this includes Cython by default which gives you access to OpenMP. Anaconda is also available on BC4, and on the computers in 1.14 so it is definitely the way to proceed. To get Cython working with OpenMP, we will need to make sure you have the correct C compiler (not the Apple one), and to get mpi4py working, you will need a working MPI library. The following sections detail how to achieve this.

2.2.1 Installing mpi4py and using message passing in Python

You most likely already have a working installation of Anaconda Python from a previous course. If not, the latest version is available from: https://www.anaconda.com/distribution/#download-section.

The message passing library, MPI, is available in Anaconda Python as the module mpi4py. You must have a compatible MPI library already installed on your system, before attempting to install mpi4py and, in the past, this has caused some issues due to later versions of the operating system being incompatible with earlier versions of mpi4py (and vice versa).

With the latest operating system (Big Sur) and Python 3.8.5, all appears to work again. However, if you are using older software and have difficulties, please contact Dr Hanna for support.

Once Anaconda is installed, go to the Mac command prompt, and type:

```
conda list
```

and look for “mpi4py” in the list. Assuming it is missing, type:

```
conda install mpi4py
```

which should install version 3.0.3. If mpi4py is present with a lower version number, update it using:

```
conda update mpi4py
```

During the installation process, you should see several libraries installed, in particular, you might see:

```
The following NEW packages will be INSTALLED:

  mpi                  pkgs/main/osx-64::mpi-1.0-mpich
  mpi4py               pkgs/main/osx-64::mpi4py-3.0.3-py38h77202c6_1
  mpich                pkgs/main/osx-64::mpich-3.3.2-hc856adb_0
```

Note, this means that Anaconda has installed an underlying set of libraries for the MPI-CH version of MPI. This is a recent change – older versions of Anaconda relied on Open-MPI instead. If you don’t see these libraries there may be a problem. You can test your installation using:

```
mpiexec -n 2 python -m mpi4py.bench helloworld
```

2.2.2 Installing and running Cython on a Mac

As previously noted, Cython is part of the standard Anaconda installation, so you should have all you need to get started. The difficulty is that the standard Apple compiler, known as Apple Clang, but aliased to the name `gcc`, is not compatible with OpenMP. This is a shame, because Cython automatically looks for a compiler called `gcc` with which to interact. So, if we want to run multi-threaded Cython code, we need a different compiler, and we need to specify a name other than `gcc`.

Instructions for installing the Gnu `gcc` compiler are given in Section 2.3.2. (Note that the `gcc` compiler is essential because it is the same compiler that was used to build Anaconda Python, and hence uses compatible libraries). Once you have followed those instructions, you should have an OpenMP compatible version of `gcc` which will be named something like “`gcc-9`” or “`gcc-11`”.

Assuming you have a `.pyx` program file, and a suitable setup file (download examples from Blackboard) you should build the executable using the command:

```
CC=gcc-11 python setup_hello_omp.py build_ext -fi
```

where `CC=gcc-11` selects your new compiler, and the setup file used contains the `-fopenmp` flag.

Run this by typing:

```
python run_hello_omp.py
```

2.3 Setting up a C/C++ environment on a Mac

2.3.1 Development Environments on a Mac

There are several options for compiling and building programs on the Apple Macintosh.

1. Because the Mac is built on a Linux kernel, you can open a terminal window and issue commands from there, and use one of the linux-based editors.
2. Alternatively, you can install Xcode, available from the app store, or an alternative development environment.

Personally, I tend to use Xcode as an editor only, issuing compilation commands from the terminal window. Either way, your first requirement will be to install the command-line tools which, among other things, will install the Apple C compiler. This is achieved by typing, at the terminal prompt:

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

The Apple C compiler will now be available and a typical command line compilation might be given as:

```
gcc -O3 -o myprog myprog.c
```

Troubleshooting Mojave

If you are using MacOS Mojave[1], you may find that the above compilation step fails with a message that various include-files cannot be found. This is because the Command Line Tools for Mojave appear to have changed their location. You should follow these instructions:

1. First, double-check you have the Command Line Tools installed: open a Terminal and write:

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

   which will guide you through the installation process or tell you they are already in place.
2. macOS Mojave changed the location of the system headers. In order to install the required header files in the old location, type:

   ```
   cd /Library/Developer/CommandLineTools/Packages/
   open .
   ```

   A window should open. Double-click the “pkg” file, and accept the defaults. This should solve the problem of the missing header files.

2.3.2 Installing OpenMP on a Mac

Unfortunately, the Apple compiler, known as “gcc” is not in fact the Gnu compiler, but another known as Clang which is built on another open-source code known as LLVM. You will see this if you type:

```
gcc --version
```

[1] I know this is quite out of date. These comments only apply to Mojave. The latest version i.e. MacOS Big Sur, appears to work fine.

Although Clang **is** compatible with OpenMP, Apple do not provide the necessary libraries, and I have not found an easy way to install them independently. The easiest approach, therefore, is to install the *real* Gnu compiler. Versions from 5.0 onwards will include the OpenMP libraries by default.

One method for installing a working gcc is to use HomeBrew. The approach consists of two steps in a terminal window:

1. Install the Homebrew MacOS package manager:

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

2. Install the latest available version of gcc (currently 11.20):

   ```
   brew install gcc
   ```

This also automatically loads the Apple command line tools (see above), if you don't already have them. To check the new compiler is loaded, type: `gcc-11 --version`, and you will see text from Gnu with version 11.2 given.

2.3.3 Compiling and running an OpenMP program on a Mac

In order to use the OpenMP library, you must first include the "omp.h" header file at the top of your program:

```
#include <omp.h>
```

On the command line, the simplest compilation command will be:

```
gcc-11 myprog.c -fopenmp
```

which will produce an executable called "a.out" which is run by typing:

```
./a.out
```

Typically, you will want to include other command line options, particularly for optimisation. For example:

```
gcc-11 -march=core2 -O3 -fopenmp -m64 -std=c99 -o myprog myprog.c
```

which will produce an optimised executable called "myprog" for running on an Intel 64bit Core2 architecture. This is run by typing:

```
./myprog
```

If you are familiar with "make" files, you could use the following (entered into a text file called e.g. "myprog.make" and being careful to maintain the indentation):

```
CC := gcc-11 # replace this with your correct compiler as identified above
ARCH := core2 # Replace this with your CPU architecture.
CFLAGS := -march=$(ARCH) -O3 -fopenmp -m64 -std=c99
COMPILE_COMMAND := $(CC) $(CFLAGS)
OUTPUT := myprog

all: myprog.c
 $(COMPILE_COMMAND) -o $(OUTPUT) myprog.c

clean:
 rm -f *.o $(OUTPUT).*
```

Typing:

```
make -f myprog.make
```

will compile the program for you. “make” files are particularly useful if you have several files to compile and link into a single executable, or when you need to specify different locations for “include” files and libraries.

2.3.4 Installing and using MPI on the Mac

The Open-MPI libraries no longer appear to be available for download and instead need to be built. This is a lengthy process. Before you begin, please note that these MPI libraries and tools are not compatible with those installed with Anaconda. If you have previously installed Anaconda in such a way that you automatically join the base, or any other, Conda environment when you open a terminal window, you should drop out of this by typing:

```
conda deactivate
```

before attempting this section. N.B. This installation method requires certain libraries that are **only** available with the Gnu compilers. I will assume you have gcc-11 installed in what follows.

1. Download the latest version of Open-MPI from:

   ```
   https://www.open-mpi.org
   ```

   You should choose the file ending `.tar.gz`. The current version is `openmpi-4.1.1.tar.gz`. MacOS will automatically expand this to `openmpi-4.1.1.tar` – if this doesn't happen you can expand it yourself using the `gunzip` command.

2. Expand the `.tar` file using:

   ```
   tar -xvf openmpi-4.1.1.tar
   ```

   This takes a while, but you should be able to see all the files being created, so you know if it is working.

3. Change directory to the newly expanded folder:

   ```
   cd openmpi-4.1.1
   ```

4. Run the configuration utility. This is the part that calls in the Gnu compiler:

   ```
   ./configure CC=gcc-11 CXX=g++-11
   ```

   This takes a very long time, but prepares the ground for building and installing the libraries.

5. Finally, build and install the libraries:

   ```
   make all install
   ```

   This takes forever (probably an hour or more). The final two stages may need to be run with `sudo` but I didn't need it.

The Open-MPI libraries should now be installed. You can check this by compiling a C/C++ MPI-based program, which you must do using `mpicc` for C or `mpicxx` for C++, and then run using `mpiexec`. Hint: if you type `which mpiexec` the result should be `/usr/local/bin/mpiexec`. If the result is `/opt/anaconda3/bin/mpiexec` you are seeing the version needed for Anaconda and mpi4py and should use `conda deactivate` before continuing.

TOPIC 3

Using OpenMP and MPI on a Windows PC

3.1 Introduction – a warning about Windows PCs

While it is expected that most simulations for this unit will be carried out on Blue Crystal, it is important to have a development environment on your local computer, to speed the writing and debugging process. Windows PCs are popular among students, and are perfectly capable of running all of the libraries we need for the PHYSM0032 Advanced Computational Physics unit. However, there are a few issues with them, the biggest being the need to work at the command prompt, but a *different* command prompt depending on which type of code you are working on (Python or C/C++). There is also the problem that very specific versions of the Microsoft compiler are needed for each different version of Anaconda when using Cython, and there are lots of potential incompatibilities e.g. with 32-bit versus 64-bit libraries etc., as well as Microsoft's insistence on making its own versions of key libraries, and then changing them so they are no longer compatible with other code.

In recent years, some students have abandoned Windows, almost entirely, and instead down their development using Windows Subsystem for Linux (WSL2). This is a brilliant bit of code, available for free as part of Windows 10, which allows you to install a fully featured copy of Linux in a Window in Windows 10. If you decide to go down this route, you should look at Chapter 4 for information on setting up your environment.

If you are sticking with Windows, there are two main sections to follow in the current chapter, depending on whether you intend to program with Python, or whether you prefer a compiled language, such as C or C++.

3.2 Setting up your C/C++ environment on a Windows PC

To set up your programming environment on Windows, you need a compiler compatible with OpenMP and a set of libraries for MPI. A suitable development environment, that will be compatible with MPI and Anaconda Python, is the Microsoft Windows Visual Studio, a free version of which is readily available. Loading the full studio will get you all the libraries you need. However, for most of the simple programs (simple=1 or few files) you are likely to build, use of the command line should be sufficient for compiling code, in conjunction with a suitable editor such as NotePad++ (available from https://notepad-plus-plus.org/).

The options for running OpenMP and MPI on a Windows-based machine are limited by the availability of working distributions of MPI. The popular distributions, Open MPI and MPICH, are no longer maintained for Windows, so that the best choice now is Microsoft's offering, MS-MPI. However, there are several alternative approaches with varying degrees of compatibility, as listed here:

1. Use the Microsoft compiler MSVC2017 or later, in conjunction with the developer's command shell. This is actually the simplest approach, although the MSVC2017 compiler seems to produce slow-running code in my tests. Fortran is **not** supported, but *this is the easiest way to get compatibility with Anaconda Python/-Cython*.
2. Install Windows Subsystem for Linux (available from Microsoft), and work in a Linux environment.
3. Install the Intel development tools for high performance computing. These offer an integrated environment for C/C++/Fortran and Anaconda Python/Cython. They are free for students, but I have no experience in using them. Taken at face value, this appears to be an interesting option.

I will give details of the first method here.

3.2.1 Installing the Microsoft MSVC compiler and MPI

These notes are one possible way to set up an MPI environment under Windows and compile and run MPI programs on the command line. I have tested this under Windows 10.

1. First you need to install the MSVC compiler. The easiest way to do this is to install Windows Visual Studio. This is a free download from Microsoft. I have downloaded the Visual Studio 2017 Community Edition, which includes the *cl* compiler.
2. If you don't want to install the whole Visual Studio, you should download and install the Visual C++ Build Tools 2017, which includes the compiler.
3. Next download and install the latest version of Microsoft MPI – the latest I have used is version 10.0 – and the MS-MPI SDK. These are available at https://docs.microsoft.com/en-us/message-passing-interface/microsoft-mpi and you should download two separate components:

   - msmpisdk.msi
   - msmpisetup.exe

   Double click on each in turn and allow the packages to install their components (this is very quick). Next, you must add the appropriate search paths to the Windows path environment variable. To do this, type "path" in the Windows search box, and then choose the "Edit environment variables" entry. Note, you can choose whether you do this for the whole system or just your account. In the dialogue box that opens, click on "Environment Variables" and then double-click on "Path". You will need to add two new entries (on separate lines):

   ```
   C:\Program Files (x86)\Microsoft SDKs\MPI
   C:\Program Files\Microsoft MPI\Bin
   ```

   Once this is done, you are ready to test the compiler.
4. To open a command line, **do not** use the standard Windows command line, but the **Developers Command** Line which will be found from the Start menu in the Visual C++ Build Tools folder. Choose "Visual C++ 2017 x64 Native Tools Command Prompt" **or** "Visual C++ 2017 x64 Native Tools Command Prompt" (they are the same) to open a 64-bit command prompt.
5. You should find the command prompt opens in the Visual Studio folder e.g.

   ```
   C:\Program Files (x86)\Microsoft Visual Studio 14.0\VC>
   ```

(Version 14 is 2015). Type the command "cl" and if all is well you will see something like:

```
Microsoft (R) C/C++ Optimizing Compiler Version 19.00.24210 for x64
Copyright (C) Microsoft Corporation.  All rights reserved.

usage: cl [ option... ] filename... [ /link linkoption... ]
```

6. If you have a 32-bit system,[1] make sure you download the 32-bit x686 versions of the above libraries, and substitute x64 with x86 in steps 4 and 5.

3.2.2 Compiling and running OpenMP programs

1. Open the Developers Command prompt from the Start Menu (see above).
2. Create a small OpenMP test program using, e.g., notepad or Notepad++. For example, create a folder called "simple" under c:\Users\<my-user-name>, change directory to it, and create a program file containing:

```
#include <stdio.h>
#include <omp.h>
int main(){
#pragma omp parallel
    printf("Hello from thread %d \n",omp_get_thread_num());
}
```

3. Compile the program using:

```
cl /openmp my_prog.c
```

4. Run the program by typing the name of the executable: "my_openmp_prog". You should see output similar to:

```
Hello from thread 0
Hello from thread 2
Hello from thread 3
Hello from thread 1
```

3.2.3 Compiling and running MPI programs

1. Open the Developers Command prompt from the Start Menu (see above).
2. Create a small MPI test program using, e.g., notepad or Notepad++. For example, create a folder called "simple" under c:\Users\<my-user-name>, change directory to it, and create a program file containing:

```
#include <mpi.h>
#include <stdio.h>
int main()
{
    MPI_Init(NULL,NULL);
    printf("Testing MPI C program compiled on the command line.\n");
    MPI_Finalize();
    return 0;
}
```

[1]Actually 32-bit systems are now very rare, and you will struggle to find compatible libraries and a compatible version of Anaconda. I would strongly recommend in this case that you work solely on Blue Crystal, using your local machine as a terminal.

3. Compile the program using the cl compiler:

   ```
   C:\simple>cl /I"C:\Program Files (x86)\Microsoft SDKs\MPI\Include" /c simple_mpi.c
   ```

   where the `/I` option tells the compiler where to find mpi.h and `/c` means create an object file only.

4. Next link the object file with the MPI libraries:

   ```
   C:\simple>link /machine:x64 /out:simple_mpi.exe /dynamicbase "msmpi.lib"
             /libpath:"C:\Program Files (x86)\Microsoft SDKs\MPI\Lib\x64" simple_mpi.obj
   ```

5. Finally run the program using mpiexec:

   ```
   C:\simple>mpiexec simple_mpi
   C:\simple>mpiexec -n 4 simple_mpi
   ```

   where the -n option determines the number of tasks allocated. You should see the same output printed by every task.

In principle the same stages are possible within Visual Studio, provided the correct include and library paths are added to the build menus. If your OpenMP and MPI programs executed successfully, then you are done. If you have any problems, please contact Dr Hanna or a demonstrator.

3.3 Setting up your Python environment on a Windows PC

The School of Physics recommends Anaconda Python, and this includes Cython by default which gives you access to OpenMP. Anaconda is also available on BC4, and on the computers in 1.14 so it is definitely the way to proceed. To get Cython working with OpenMP, we will need to make sure you have the correct C compiler, and to get mpi4py working, you will need a working Microsoft MS-MPI library.

For a fully working installation of Python under Windows 10, you will need to install the following:

1. A full installation of Python 3 from Anaconda (Python version 3.5 or later).
2. The Microsoft Visual C compiler (2017 edition or later) for running Cython.
3. The latest version of MPI for Python (mpi4py).
4. The Microsoft MPI libraries and SDK (version 10.0 or later)[2].

The following sections detail how to achieve this.

3.3.1 Anaconda Python

You most likely already have a working installation of Anaconda Python from a previous course. This will generally work fine. If you are installing from scratch, note that Anaconda is available from:

https://www.anaconda.com/distribution/#download-section.

[2]Note: These libraries might not be required. I suggest testing mpi4py *before* installing them, in case they are not needed.

3.3.2 Microsoft Visual C compiler

Instructions for installing this are given in Section 3.2.1. The Microsoft compiler is mandatory with this configuration, because it is the same compiler that was used to build Anaconda Python, and hence uses compatible libraries.

3.3.3 Installing the mpi4py module

Once the latest Anaconda is installed, go to the Windows command prompt, and type:

```
conda list
```

and look for “mpi4py” in the list. Assuming it is missing, type:

```
conda install mpi4py
```

which should install version 2.0.0. If mpi4py is present with a lower version number, update it using:

```
conda update mpi4py
```

3.3.4 Installing the Microsoft MPI libraries

Installation of the MS-MPI libraries is described in Section 3.2.1.

3.3.5 Testing Cython under Windows 10, with OpenMP

To test that your Cython installation is working correctly, copy the files:

```
picalc_pyx_omp.pyx
setup_picalc_pyx_omp.py
run_picalc_pyx_omp.py
```

to a folder in your personal file space.

Edit the setup file, changing instances of “-fopenmp” to “/openmp”. Build the cythonized code by typing:

```
python setup_picalc_pyx_omp.py build_ext -fi --compiler=msvc
```

Note: use of the “/openmp” flag in the setup file, ensures your executable will be multithreaded. Test the program by typing:

```
python run_picalc_pyx_omp.py 1000000 8
```

The two numbers on the command line are the number of iterations (1000000) and number of threads (8). You should see output that resembles:

```
Threads set to  8
Elapsed time: 0.011970 s
Pi = 3.1415946524137999
```

TOPIC 4

Using OpenMP and MPI on a Linux PC

4.1 Introduction – a warning about Linux PCs

While it is expected that most simulations for this unit will be carried out on Blue Crystal, it is important to have a development environment on your local computer, to speed the writing and debugging process. Linux PCs are ideal for running all of the libraries we need for the PHYSM0032 Advanced Computational Physics unit, but are relatively rare among students. This means there will be less help available from your peers, and you may end up as the *local expert* if you go down this route. However, there are advantages, including the fact that familiarity with Linux will help you when you are using Blue Crystal, and that it is relatively easy to keep a consistent set of libraries up to date. These notes will be very brief, because I generally feel that, if you are using Linux, you probably know what you are doing.

If you have Windows on your PC, but would prefer to have Linux, I strongly recommend against playing around with dual-boot systems. In recent years I have known students lose all their files this way! Instead, I suggest you consider using the Windows Subsystem for Linux (WSL2). This is a brilliant bit of code, available for free as part of Windows 10, which allows you to install a fully featured copy of Linux in a Window in Windows 10. If you decide to go down this route, you should look at the next section for information on getting started.[1]

There are three sections to follow in this chapter. First, I briefly outline the process for loading Windows Subsystem for Linux (WSL2), and then there are different sections depending on whether you intend to program with Python, or whether you prefer a compiled language, such as C or C++.

4.2 How to install Windows Subsystem for Linux (WSL2)

WSL2 is available for free as part of Windows 10. The installation process is easy to follow, but it is best to search online for the latest instructions. The process consists of three stages:

1. Change a setting in Windows 10, opening it up to developer installations;
2. Update Windows to install WSL2;

[1]Of course, if you are using a Mac, you already have a Linux-like command line, but you should definitely NOT follow the instructions here - they are for Ubuntu Linux only.

3. Visit the Microsoft shop and download a free copy of Linux. There are a few available, and I have tested and recommend Ubuntu.

4. Follow the instructions in the following sections to set up your programming environment.

4.3 Setting up a C/C++ environment in Ubuntu Linux

There are two main libraries you will need (OpenMP and MPI), together with a set of compilers. Fortunately, OpenMP is installed with the compiler, so there are just two installations needed. There are many variations of linux and all have different ways of installing packages. I will illustrate only with Ubuntu linux, as I have it installed on numerous machines. There are several stages required in the installation.

C and C++ compilers: First update your package lists:

```
sudo apt-get update
```

We use the GNU compiler family which is free, open source, and easy to install:

```
sudo apt-get install gcc g++
```

Open MPI package: Install Open MPI with package manager:

```
sudo apt-get install libopenmpi-dev
```

Check that it has installed correctly using:

```
which mpirun mpicc mpic++
```

which should produce the output:

```
/usr/bin/mpirun
/usr/bin/mpicc
/usr/bin/mpic++
```

If mpirun is missing, install it using:

```
sudo apt install openmpi-bin
```

Note: MPICH can alternatively be installed by using `sudo apt-get install libmpich-dev`. However, to avoid confusion you should **not** install both!

Check MPI version by typing:

```
mpirun --version
```

which should produce output similar to:

```
mpirun (Open MPI) 2.1.1 Report bugs to http://www.open-mpi.org/community/help/
```

You can test your OpenMPI installation using a piece of sample MPI code. Obtain the sample code, build and run it as follows:

```
wget https://www.open-mpi.org/papers/workshop-2006/hello.c
mpicc -o hello.exe hello.c
mpirun -np 2 ./hello.exe
```

The output from the mpirun command should look like this:

```
Hello, World.  I am 1 of 2
Hello, World.  I am 0 of 2
```

4.4 Setting up a Python environment in Ubuntu Linux

There are three steps in preparing your Python environment:

1. In preparing the Python environment, we need access to Cython, which needs a compiler with OpenMP, and we need to install mpi4py, which needs access to the underlying MPI libraries. Therefore, you should first follow the instructions in the previous section (Section 4.3) before continuing. Once you have the compilers, and MPI libraries, you can proceed to the next stage.
2. How you proceed depends on the type of Python installation you have. Every linux installation has a "system" python, which is best left alone, unless you don't mind taking risks. You most likely already have a working installation of Anaconda Python from a previous course. If you don't, the latest version of Anaconda Python is available from: https://www.anaconda.com/distribution/#download-section.
3. Now you can install a copy of mpi4py:
   (a) If you have Anaconda python:
   - Use `conda list` to check for existing copies of mpi4py. If you find any, remove using `conda uninstall mpi4py`
   - Install the latest mpi4py using:
     `conda install -c conda-forge mpi4py`
     This should get you version 3.0.2.
   - Finally type:
     `mpiexec -n 2 python -m mpi4py.bench helloworld`
     to verify that the installation has worked.

   (b) If you have other python: You will need to use the pip installer to load mpi4py. You may need to download this separately. Once you have it, install mpi4py using:
   `pip install mpi4py`
   which should get you the latest version.

TOPIC 5

Getting started with Cython: building and running basic Cython programs

5.1 Introduction

As discussed in lectures, Cython is a system for taking Python code and compiling it into a binary executable, which will potentially run many times more quickly than the original script. At the barest minimum, to use Cython you require:

- Anaconda Python
- A C/C++ compiler such as gcc or MSVC

In addition, to run the OpenMP and MPI libraries discussed in the lectures, you will need:

- Your C/C++ compiler must have working OpenMP libraries
- An installation of MPI
- The Python module mpi4py version 2.0.0 or later.

Full installation details are given in Section **??** and the following sections. I will assume you are issuing all commands from the command prompt. There may be some opportunity for automating these through the menu items of whatever development environment you have chosen, but working form the command line has the advantage that it transfers directly to BlueCrystal where the bulk of your computation is likely to take place.

What follows are some brief notes to get started with Cython. More comprehensive documentation is available at: https://cython.readthedocs.io/en/latest/

5.2 Typical scheme for running Cython

You will typically need a minimum of 3 files for running each Cython script you write:

1. Your Cython script, which will look very much like Python with some extra commands, and which must have the filename extension ".pyx"

2. A “setup” script which will contain instructions for compiling the Cython script into an executable library
3. A “run” script which will be written in Python and will load and use the library you created using the previous script.

5.2.1 A basic Cython script

Here is a simple Cython script (picalc_pyx1.pyx) which is really just Python:

```
import time
from math import sqrt

def main( nmax ):
    pibyfour = 0.0
    dx = 1.0 / nmax
    initial = time.time()

    for i in range(nmax):
        pibyfour += sqrt(1-(i*dx)**2)

    final = time.time()
    print("Elapsed time: {:8.6f} s".format(final-initial))
    pi = 4.0 * pibyfour * dx
    print("Pi = {:18.16f}".format(pi))
    return 0
```

Note:

1. The script consists of a single function, “main()”, that will ultimately be called from your “run” script
2. The script could actually contain multiple functions with different names that could all be called separately from your “run” script.

5.2.2 A basic “setup” file

The “setup” file contains instructions for building the binary library. Here is a typical example (setup_picalc_pyx1.py):

```
from distutils.core import setup
from Cython.Build import cythonize

setup(name="picalc_pyx1",
      ext_modules=cythonize("picalc_pyx1.pyx"))
```

This script should be executed by typing:

```
python setup_picalc_pyx1.py build_ext -fi
```

if you are using the gcc compiler or:

```
python setup_picalc_pyx1.py build_ext -fi --compiler=msvc
```

for the Microsoft compiler.

Assuming this works, you will be left with a library file with the extension “.so” on a Mac, or ending with “.pyd” under Windows.

5.2.3 A simple “run” script for Cython

The “run” script can be thought of as a wrapper for the Cython code. In this simple case (run_picalc_pyx1.py), the script takes its argument from the command line and calls the compiled binary code in the library:

```
import sys
from picalc_pyx1 import main

if int(len(sys.argv)) == 2:
    main(int(sys.argv[1]))
else:
    print("Usage: {} <ITERATIONS>".format(sys.argv[0]))
```

To run the program, type the following:

```
python run_picalc_pyx1.py 1000000
```

Note:

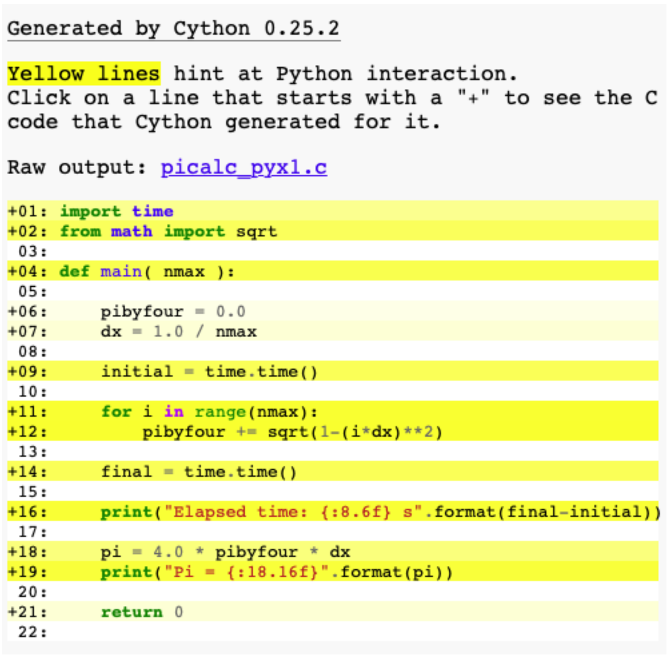
1. The function “main()” is imported from the module “picalc_pyx1” that was created in the “setup” process.
2. “sys.argv[]” is a list of command line arguments; “sys.argv[0]” is the command itself. This is a concept imported from C.
3. An error message is generated if the wrong number of arguments is presented
4. The whole “sys.argv[]” part is optional, but useful especially when running the code on BlueCrystal.

5.2.4 A useful Cython diagnostic tool

When analysing Cython code with a view to making it run faster, the following diagnostic command is very useful:

```
cython -a picalc_pyx1.pyx
```

This generates an html file that can be viewed in a browser:



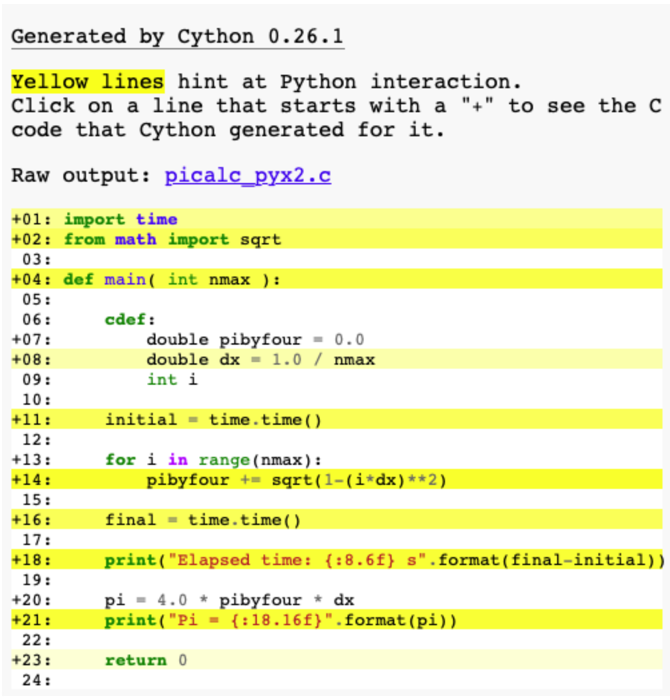
```
Generated by Cython 0.25.2

Yellow lines hint at Python interaction.
Click on a line that starts with a "+" to see the C
code that Cython generated for it.

Raw output: picalc_pyx1.c

+01: import time
+02: from math import sqrt
 03:
+04: def main( nmax ):
 05:
+06:     pibyfour = 0.0
+07:     dx = 1.0 / nmax
 08:
+09:     initial = time.time()
 10:
+11:     for i in range(nmax):
+12:         pibyfour += sqrt(1-(i*dx)**2)
 13:
+14:     final = time.time()
 15:
+16:     print("Elapsed time: {:8.6f} s".format(final-initial))
 17:
+18:     pi = 4.0 * pibyfour * dx
+19:     print("Pi = {:18.16f}".format(pi))
 20:
+21:     return 0
 22:
```

The more yellow that appears, the greater the chance the code will run slowly. Clicking on a line will show the auto-generated C code beneath, and this can sometimes be surprising in its extent. The process of optimisation will consist of minimising the amount of yellow in the key areas – in the present case the main loop, lines 11 & 12.

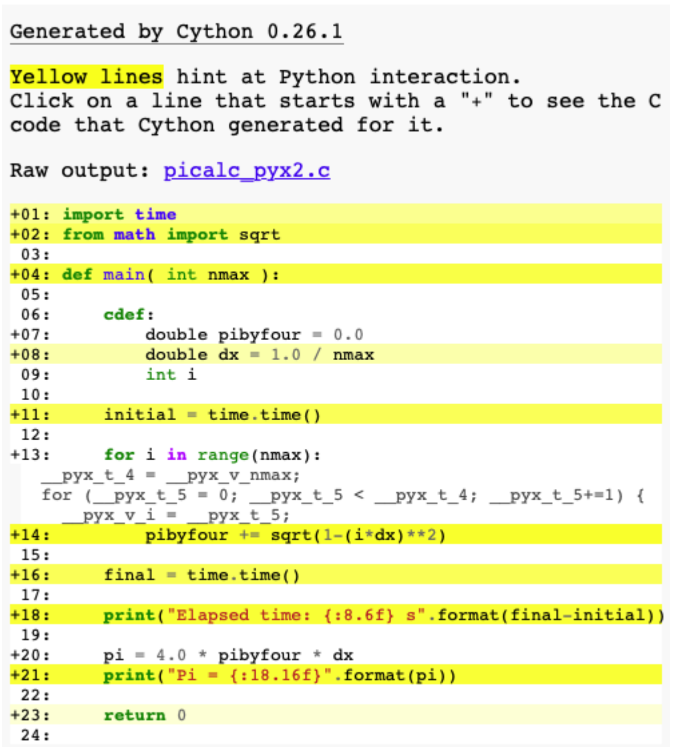
The execution time of picalc_pyx1.pyx with 10^9 iterations on my laptop is 209 s compared to 263 s for the original Python.

5.3 First stage optimisation of Cython code

The first level of optimisation that can be applied is the use of the `cdef` command. `cdef` is used to specify the type of a variable. Much of the execution time of a Python script is spent in determining variable types from the context, so giving a pointer can be very helpful. The following code (picalc_pyx2.pyx) illustrates the use of `cdef`:



```
Generated by Cython 0.26.1

Yellow lines hint at Python interaction.
Click on a line that starts with a "+" to see the C
code that Cython generated for it.

Raw output: picalc_pyx2.c

+01: import time
+02: from math import sqrt
 03:
+04: def main( int nmax ):
 05:
 06:     cdef:
+07:         double pibyfour = 0.0
+08:         double dx = 1.0 / nmax
 09:         int i
 10:
+11:     initial = time.time()
 12:
+13:     for i in range(nmax):
+14:         pibyfour += sqrt(1-(i*dx)**2)
 15:
+16:     final = time.time()
 17:
+18:     print("Elapsed time: {:8.6f} s".format(final-initial))
 19:
+20:     pi = 4.0 * pibyfour * dx
+21:     print("Pi = {:18.16f}".format(pi))
 22:
+23:     return 0
 24:
```



```
Generated by Cython 0.26.1

Yellow lines hint at Python interaction.
Click on a line that starts with a "+" to see the C
code that Cython generated for it.

Raw output: picalc_pyx2.c

+01: import time
+02: from math import sqrt
 03:
+04: def main( int nmax ):
 05:
 06:     cdef:
+07:         double pibyfour = 0.0
+08:         double dx = 1.0 / nmax
 09:         int i
 10:
+11:     initial = time.time()
 12:
+13:     for i in range(nmax):
   __pyx_t_4 = __pyx_v_nmax;
   for (__pyx_t_5 = 0; __pyx_t_5 < __pyx_t_4; __pyx_t_5+=1) {
     __pyx_v_i = __pyx_t_5;
+14:         pibyfour += sqrt(1-(i*dx)**2)
 15:
+16:     final = time.time()
 17:
+18:     print("Elapsed time: {:8.6f} s".format(final-initial))
 19:
+20:     pi = 4.0 * pibyfour * dx
+21:     print("Pi = {:18.16f}".format(pi))
 22:
+23:     return 0
 24:
```

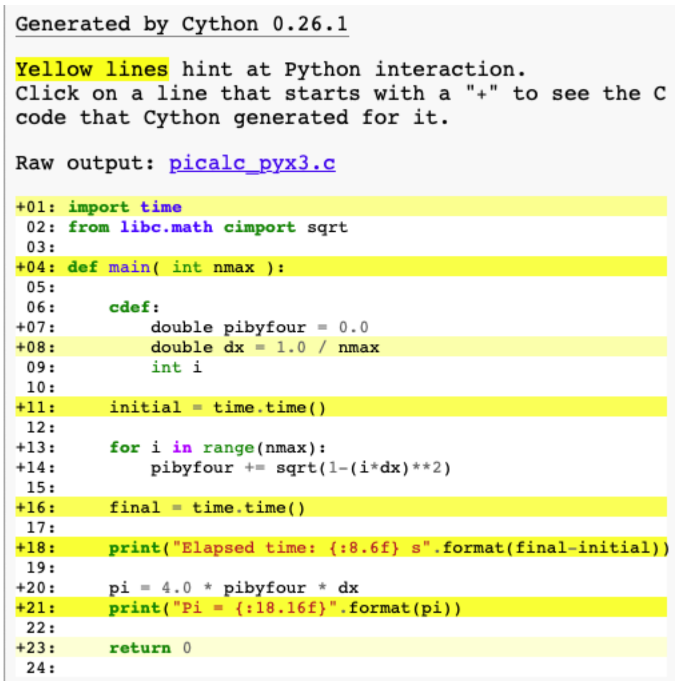
Note the use of the `cdef` block. Integers `nmax` and `i` are declared as `int`, while floating point variables are declared as `double`, which is short for double precision. These are standard designations from C/C++.

Note also that the `for` loop is no longer yellow. Clicking on it reveals a very C-like construction beneath.

The execution time of picalc_pyx2.pyx with 10^9 iterations on my laptop is 74 s compared to 209 s for picalc_pyx1.pyx.

5.4 Second stage optimisations

The next easy thing to try is to replace any Python maths library calls with their C equivalents. In the present case, this means replacing the square root with the version in the C maths library. This is illustrated here (picalc_pyx3.pyx):



Generated by Cython 0.26.1

Yellow lines hint at Python interaction.
Click on a line that starts with a "+" to see the C code that Cython generated for it.

Raw output: picalc_pyx3.c

```
import time
from libc.math cimport sqrt

def main( int nmax ):

    cdef:
        double pibyfour = 0.0
        double dx = 1.0 / nmax
        int i

    initial = time.time()

    for i in range(nmax):
        pibyfour += sqrt(1-(i*dx)**2)

    final = time.time()

    print("Elapsed time: {:8.6f} s".format(final-initial))

    pi = 4.0 * pibyfour * dx
    print("Pi = {:18.16f}".format(pi))

    return 0

```

Note that the line containing the square root is no longer yellow. Note also the use of `cimport` to import the C library function. The execution time of this program on my laptop is 5.6 s for 10^9 iterations.

5.5 Cython with OpenMP

As discussed elsewhere, the OpenMP library is built into Cython through use of `prange`. The important thing to note is the need to include a flag for OpenMP on the C compiler. This is done as follows (setup_picalc_pyx_omp.py):

```
from distutils.core import setup
from distutils.extension import Extension
from Cython.Build import cythonize

ext_modules = [
    Extension(
        "picalc_pyx_omp",
        ["picalc_pyx_omp.pyx"],
        extra_compile_args=['-fopenmp'],
        extra_link_args=['-fopenmp'],
    )
]

setup(name="picalc_pyx_omp",
      ext_modules=cythonize(ext_modules))
```

For the Microsoft MSVC compiler change the '-fopenmp' flag to '/openmp'.

Here is the Cython script (picalc_pyx_omp.pyx):

```
import time
from libc.math cimport sqrt
from cython.parallel cimport prange
cimport openmp

def main( int nmax, int threads ):

    cdef:
        double pibyfour = 0.0
        double dx = 1.0 / nmax
        double initial, final
        int i

    print("Threads set to {:2d}".format(threads))
    initial = openmp.omp_get_wtime()

    for i in prange(nmax, nogil=True, num_threads=threads):
        pibyfour += sqrt(1-(i*dx)**2)

    final = openmp.omp_get_wtime()
    print("Elapsed time: {:8.6f} s".format(final-initial))
    pi = 4.0 * pibyfour * dx
    print("Pi = {:18.16f}".format(pi))
    return 0
```

Things to note:

1. The parallel loop construction uses `prange` rather than `range` and this needs to be loaded using `cimport`.
2. `nogil=True` is used to allow the use of multiple threads. This is needed because Python normally runs using a single thread. However, this may cause problems later for certain Python commands, in which case it is possible to run them "with gil".
3. Note also the use of the native OpenMP functions for timing.

The run script (run_picalc_pyx_omp.pi) for this file is:

```
import sys
from picalc_pyx_omp import main

if int(len(sys.argv)) == 3:
    main(int(sys.argv[1]),int(sys.argv[2]))
else:
    print("Usage: {} <ITERATIONS> <THREADS>".format(sys.argv[0]))
```

- Note the use of an additional command line argument to specify them number of threads to use.
- More details on using Cython with OpenMP will be found in Section **??**.

5.6 Additional Cython features

5.6.1 cdef for functions

There are several different ways to define a function in Cython:

def myfunc(myobj): This treats the function as a normal Python function with normal Python object as argument;

def myfunc(int myvar): This treats the function as a normal Python function with a typed argument, in this case an integer variable. The “cdef” is implicit in the argument list;

cdef myfunc(int myvar): This allows the function to be called in the same way as a C function. It is very efficient and can give good speed-ups;

cpdef myfunc(int myvar): This gives the best of both worlds. The single definition can be called as a Python function from Python, or as a Cython function from Cython.

5.6.2 Typed Memory Views

This is the recommended way to handle arrays in Cython. The typed memory view is the most efficient way of accessing data in an array. The standard approach is to create the array using NumPy and then map this onto a memory view as follows:

```
cdef int[:,:,:] mymemview
mymemview = np.zeros((10,15,20),dtype=np.int32)
```

The above creates a 3-dimensional memory view which acquires a memory buffer from the NumPy array specified, which consists of 32-bit integers.

The following generates a 2-dimensional double precision buffer and assigns it to a 2-d memory view in a single statement.

```
cdef double[:,:] mymemview = np.zeros((20,20),dtype=np.double)
```

There is lots of flexibility in the way memory views map onto actual buffers – see the Cython documentation for more details.

TOPIC 6

Optimisation tips and tricks

Before trying anything "clever" in a program, there are a few things worth bearing in mind which can sometimes make a large difference. This list is not comprehensive, and not all tricks will work in all circumstances.

6.1 Compilers

Compiler flags: Many students are unaware that compiler flags exist to control the degree of optimisation of the code. The simplest optimisers are `-O, -O0, -O1, -O2, -O3`, where the last of these offers the greatest speed-up. There is also `-Ofast` but this can be dangerous, as some of the optimisations will affect the accuracy e.g. using 1.5 precision instead of double precision. Optimisations can include:

- unrolling loops
- auto-vectorisation
- throwing out redundant code

and there are many other flags possible to fine-tune these. It is possible to craft your code to take advantage of the available optimisations. e.g. grouping variables in 4's in a loop may encourage vectorisation.

Processor optimisation: There are also compiler flags to select particular processor architecture e.g. `-march=core2` works for my aged laptop, while `-march=skylake` works for a more modern, 6th generation i7 processor. `-mavx` encourages the compiler to try autovectorisation.

Choice of compiler: Different compilers have different available optimisations, and may cope differently with different code. I observed a 20% performance difference between the Apple Clang compiler and gcc 5.4 for one particular example with default compiler options.

6.2 Memory management

Order of storage of arrays: In C/C++, multi-dimensional arrays are stored in a row-major order. (In Fortran, it is column major). This means that, for a two-dimensional array, the first row is stored first, followed by the second, then the third etc. In terms of memory layout, the elements of a 3x3 array, `A[i][j]`, will be stored in the order: `A[1][1], A[1][2], A[1][3], A[2][1], A[2][2], A[2][3], A[3][1],` A[3][2], A[3][3]. It will be more efficient to access the elements of A row by row, rather than column by column, because this will involve sequential memory accesses, and will make better use of cache lines.

In Python, NumPy arrays are stored in row-major form, similar to C/C++. However, it is possible to request Fortran ordering when initially defining the array. This may be advantageous e.g. when forming a matrix product, where rows from one matrix are multiplied by columns from the other.

AOS versus SOA: Consider an n-body simulation where you need to store coordinates x, y, z and mass m for each body. A programmatically convenient approach might be to pack x, y, z and m into a structure or class, and create an array of such structures (AOS). This should be efficient for, each time you need to compute, for example, the acceleration of a body, you might need x, y, z and m simultaneously, and having them side by side in memory will allow efficient memory access. It works well with GPUs because different structures will be worked on by different processing cores. However, if you want to take advantage of vectorisation it turns out to be more efficient in terms of memory access to have a data structure of arrays (SOA), with one array for each variable. Also, the arrangement of your memory may impact on how readily you can transmit your data using MPI.

6.3 Python specific issues

'for' loops and NumPy vectorisation: The 'for' loop in Python is notoriously slow. Wherever possible, avoid 'for' loops by taking advantage of NumPy vectorisation (see lectures). NumPy vectorisation can be a natural fit to your algorithm if you are processing your data in a form of an array, as for example the 2D heat equation example given in lectures. If you are not using an array, consider modifying your algorithm to take advantage of one. For example, the PiCalc.py program could be written to place intermediate results in an array, whose values are then summed:

```
from math import sqrt
pibyfour = 0.0
dx = 1.0 / nmax

for i in range(nmax):
    pibyfour += sqrt(1-(i*dx)**2)

pi = 4.0 * pybyfour * dx
```

```
import numpy as np
pibyfour = 0.0
dx = 1.0 / nmax
chunk = 100000

idx = np.linspace(0,chunk*dx,num=chunk,
                         endpoint=False)

for i in range(0,nmax,chunk):
    temp_array = np.sqrt(1-(i*dx+idx)**2)
    pibyfour += np.sum(temp_array)
pi = 4.0 * pybyfour * dx
```

The array idx contains a set of values of i*dx at which the function $\sqrt{1 - i.dx}$ needs to be evaluated. The second array temp_array holds the results of these evaluations which are summed using the np.sum() function. For nmax = 10^8, the left hand code runs in about 27 seconds on my laptop, whereas the right hand code runs in about 0.33 seconds. You can experiment with different values for the array size (variable chunk) but on my system 10^5 appeared to be optimum i.e. fastest.

More problems with 'for' loops: Avoid performing algebra inside a 'for' loop that could be done outside the loop. If a C compiler spots such repetitive and redundant computations, it will optimise the code in question and move the lines of code outside the loop. However, Python will not do this. In the following left-hand example, all the computations will be performed repeatedly, slowing code significantly. However, the right-hand version is just as correct, but runs in 2/3 of the time (1.9 seconds compared with 3.2 seconds):

```
import numpy as np
nmax = 10000000
dd = np.zeros(nmax)
delta = 0.001

for i in range(nmax):
    dd[i] = i*(delta**3)
result = np.sum(dd)
```

```
import numpy as np
nmax = 10000000
dd = np.zeros(nmax)
delta = 0.001

for i in range(nmax):
    dd[i] = i
result = np.sum(dd)*(delta**3)
```

Even more problems with ‘for’ loops: Functions inside loops in python also can be a cause of slowness. Although we encourage the use of user-defined functions to improve legibility of code, such a function in the heart of nested loops can add a significant overhead to the running of your code. For example, in the following code, the right-hand version with a function runs in over 4 seconds, while the original left-hand version runs in 3.2 seconds:

```
import numpy as np
nmax = 10000000
dd = np.zeros(nmax)
delta = 0.001

for i in range(nmax):
    dd[i] = i*(delta**3)
result = np.sum(dd)
```

```
import numpy as np
nmax = 10000000
dd = np.zeros(nmax)
delta = 0.001

def kernel(a,b):
    return a*(b**3)

for i in range(nmax):
    dd[i] = kernel(i,delta)
result = np.sum(dd)
```

6.4 Miscellaneous

Beware square roots: Many students use the power function, `pow(x,y)` to calculate x^y. However, `sqrt(x)` is much more efficient than `pow(x,0.5)`, and `x*sqrt(x)` is much more efficient than `pow(x,1.5)` – useful to know if you are doing an n-body gravity simulation. On the other hand, `x*x` may, or may not, be quicker than `pow(x,2)` depending on the language used and other optimisations in place.

In Python, the math library `math.sqrt(x)` can run significantly more quickly than the NumPy form `numpy.sqrt(x)`. However, the NumPy form is vectorised, and easily wins when processing arrays of numbers.

6.5 Profiling your code

There are various methods available for profiling your code, which means, analysing your code automatically to see where the computer is spending most of its time. In general, profiling will work best in code with lots of user-defined functions, as the output generally indicates the time, in seconds, spent in each. This helps to establish where bottle-necks are occurring and enables the programmer to direct their efforts more effectively.

6.5.1 Profiling a Python program

The simplest way to profile a Python program is using the cProfile module. This can be included as part of your program, or can be run from the command line. For example, in a version of my 2D heat equation program, I have the following functions:

`inidat()`	initialises the data arrays
`update()`	the main function for updating the arrays, called once per iteration
`do_a_line()`	updates one line of the array
`do_a_point()`	updates one cell on one line of the array

If I run the program with 50 iterations, and a 100×100 data array, I expect `inidat()` to be called once, `update()` to be called 50 times, `do_a_line()` to be called $(100-2) \times 50 = 4900$ times, and `do_a_point()` to be called $(100-2) \times (100-2) \times 50 = 480200$ times. Running the program form the command-line with the profiler, using the following command:

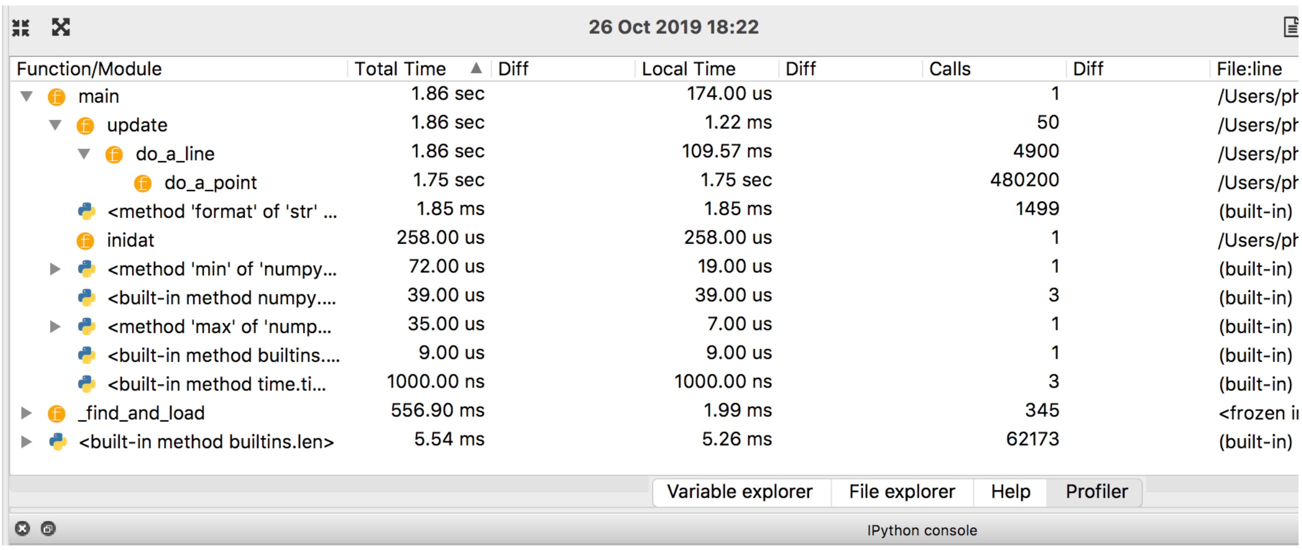
```
python -m cProfile heat2Dnv0.py 50 100 100
```

produces a couple of thousand lines of output, mostly concerning system-level functions that have been called. However, searching through this for the above set of functions, we find:

```
heat2Dnv0.py: X= 100  Y= 100  Steps= 50 time: 1.846391 s
         938429 function calls (930726 primitive calls) in 2.418 seconds

   Ordered by: standard name

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000    2.419    2.419 heat2Dnv0.py:21(<module>)
        1    0.000    0.000    0.000    0.000 heat2Dnv0.py:31(inidat)
   480200    1.737    0.000    1.737    0.000 heat2Dnv0.py:61(do_a_point)
     4900    0.108    0.000    1.845    0.000 heat2Dnv0.py:64(do_a_line)
       50    0.001    0.000    1.846    0.037 heat2Dnv0.py:68(update)
        1    0.000    0.000    1.847    1.847 heat2Dnv0.py:94(main)
```

from which we can confirm that the functions have indeed been called the correct number of times, but also, we can see how long, in seconds, is spent in each function. Clearly, in this case, any efforts at optimisation should be focussed on the `do_a_point()` function, which takes up about 1.7 seconds out of the total run time of 1.8 seconds.

Similar functionality is available directly from within Spyder, and is rather easier to interpret as much of the system-related information is restricted. To use the profiler, simply select "Profile" from the run menu, and wait while the program runs. The output for the same heat equation program as above looks like this:



Function/Module	Total Time ▲	Diff	Local Time	Diff	Calls	Diff	File:line
▼ main	1.86 sec		174.00 us		1		/Users/ph
▼ update	1.86 sec		1.22 ms		50		/Users/ph
▼ do_a_line	1.86 sec		109.57 ms		4900		/Users/ph
do_a_point	1.75 sec		1.75 sec		480200		/Users/ph
<method 'format' of 'str' ...	1.85 ms		1.85 ms		1499		(built-in)
inidat	258.00 us		258.00 us		1		/Users/ph
▶ <method 'min' of 'numpy...	72.00 us		19.00 us		1		(built-in)
<built-in method numpy....	39.00 us		39.00 us		3		(built-in)
▶ <method 'max' of 'nump...	35.00 us		7.00 us		1		(built-in)
<built-in method builtins....	9.00 us		9.00 us		1		(built-in)
<built-in method time.ti...	1000.00 ns		1000.00 ns		3		(built-in)
▶ _find_and_load	556.90 ms		1.99 ms		345		<frozen i
▶ <built-in method builtins.len>	5.54 ms		5.26 ms		62173		(built-in)

6.5.2 Profiling C code

Profiling tools for C are available from various sources. Commercially, these codes tend to be expensive and sold by compiler vendors to large companies: e.g. the profiling tools from Intel form a major part of its development suite and will retail for tens of thousands of pounds. Fortunately they are available to students under academic licence should you want to try them out. However, the most general approach to profiling will be to use the tools available with the Gnu compilers (sadly, this is not an option under Windows 10). The Gnu profiler is known as “gprof” and gives similar information to the cProfile module in Python. On the Mac, gprof can be loaded using “brew” while, on linux, your standard package manager should be used. As an alternative to gprof, on linux or Mac, you could try the Google Performance Tools, which will get you “pprof”. None of these solutions are particularly easy to use.

TOPIC 7

Using “make” files

7.1 Introduction

“make” is a unix-based command-line utility that is very useful for compiling and linking C/C++, Fortran, Cython and other programs that require the setting of multiple flags, or the handling of multiple files. Versions of “make” are available under Linux (including BlueCrystal), on the Mac, and on Windows – this section will describe use of “make” and give specific instructions for running it on different operating systems.

7.2 Invoking “make”

The “make” utility operates on a text file, known as a makefile, that contains all of the instructions necessary to build a particular program. Before discussing the detail of implementing a makefile, the syntax of the “make” command is given including, where necessary, instructions on installing the utility.

7.2.1 “make” on Linux and MacOS systems

“make” is readily available on all Linux and MacOS systems. There are two basic ways of invoking it:

1. In a folder with a single makefile named “Makefile” (with no filename extension), type the single word:

   ```
   make
   ```

2. In a folder containing multiple makefiles, specify the name of the makefile using:

   ```
   make -f myMakefile.txt
   ```

7.2.2 “make” on a Windows system

- If you have installed the MSYS command subsystem, as described in Section **??** above, the Unix “make” utility will be available to you at the command prompt, and you can use the tutorial following below.

- If, however, you have not installed MSYS and are relying on the Microsoft compilers and Visual Studio, you will not have access to “make” but can, instead, use Microsoft’s “nmake” utility, which is available on the command line. “nmake” reproduces the functionality of “make” but is sufficiently incompatible that a makefile written for one system will throw up errors when used for the other. If you need to use “nmake”, documentation is available at: `https://msdn.microsoft.com/en-us/library/dd9y37ha.aspx` .

7.3 A “make” tutorial

7.3.1 The general philosophy behind “make”

As already notes, “make” operates on a text file containing instructions for building a particular program or programs. The process often consists of multiple stages, and at each stage, “make” will look at the time stamps on files to see whether or not they need to be re-built. For example, suppose two program (source) files, `myprog1.c` and `myprog2.c`, are compiled into binary (object) files, `myprog1.o` and `myprog2.o`, and then linked into a single executable, `myprog.exe`. The makefile will contain instructions for two compilations and one linking stage. If `myprog1.c` is changed, “make” will spot that the timestamp on `myprog1.o` is older than `myprog1.c`, and will re-compile it. Then “make” will see that the timestamp on `myprog.exe` is older than `myprog1.o`, and it will re-link the executable. Only steps that are required by out-of-date timestamps will be performed. When working on large projects with multiple source files, this can result in a considerable time saving. For us, it is more of a convenience.

7.3.2 Basic makefile syntax

As noted above, the makefile is a text file containing rules for building programs or parts of programs. A typical simple makefile would appear as follows:

```
#
# Any line beginning with a hash is a comment
#
# Indented lines must be indented with a single <TAB>
#
myprog1.o: myprog1.c
	gcc -c myprog1.c

myprog2.o: myprog2.c
	gcc -c myprog2.c

myprog.exe: myprog1.o myprog2.o
	gcc myprog1.o myprog2.o -o myprog.exe
```

Notes:

- The rules are referred to as *dependencies*, and the names appearing on the lines with colons are labels. Having the labels correspond exactly with the filenames is a convenience but not essential i.e. the same could be achieved with:

  ```
  myobject1: myprog1.c
  	gcc -c myprog1.c
  myobject2: myprog2.c
  	gcc -c myprog2.c
  myexecutable: myobject1 myobject2
  	gcc myprog1.o myprog2.o -o myprog.exe
  ```

but with rather less clarity.

- Each rule (dependency) is followed by indented lines containing the commands to be executed. The indents should be a single <TAB> – if copying and pasting from the above, these white-spaces will need to be edited manually.
- Due to the use of timestamps, if “make” thinks the executable file is up-to-date, it will say so:

  ```
  make: ‘myprog.exe’ is up to date.
  ```

  If this is not correct, or you just want to recompile to be on the safe-side e.g. if you have changed some compiler flags in the makefile itself, you can invoke “make” with the -B option:

  ```
  make -B
  ```

- If you only want to re-build one item from a makefile, refer to it by label, as in for example:

  ```
  make myprog1.o
  ```

- It is common to specify the final executable with the “all” label, as in:

  ```
  myprog1.o: myprog1.c
      gcc -c myprog1.c
  myprog2.o: myprog2.c
      gcc -c myprog2.c
  all: myprog1.o myprog2.o
      gcc myprog1.o myprog2.o -o myprog.exe
  ```

  in which case a call to:

  ```
  make all
  ```

  will ensure that everything is recompiled.

- It is also common to include a “clean” label in the makefile, to enable a clean-start when rebuilding the code. For example:

  ```
  myprog1.o: myprog1.c
      gcc -c myprog1.c
  myprog2.o: myprog2.c
      gcc -c myprog2.c
  all: myprog1.o myprog2.o
      gcc myprog1.o myprog2.o -o myprog.exe
  clean:
      /bin/rm -f *.o myprog.exe
  # Use the del command on Windows
  ```

  followed by the calls:

  ```
  make clean
  make
  ```

  will result in all .o files and the executable being deleted before the program is rebuilt. **N.B. Exercise extreme caution in using this command in a folder containing multiple makefiles, as it is very easy to delete more than you intended. I speak from experience.**

7.3.3 Variables in makefiles

The use of variables in makefiles saves a lot of typing, and makes it easy to include the same sets of compiler flags for each build stage or, indeed, to change them if necessary. Consider the following example, which I used to build a vectorised version of my pi program:

```
CC := gcc-6
# replace this with your correct compiler as identified above

ARCH := core2 # Replace this with your CPU architecture.
# core2 is pretty safe for most modern machines.

CFLAGS := -march=$(ARCH) -mavx -O3 -m64 -std=c99 -fopenmp
# add -fopenmp for OpenMP support (won’t work with Apple Clang).
# -mavx for AVX support.
# -mfma for fused multiply/add functions from AVX2 (2013 onwards).

COMPILE_COMMAND := $(CC) $(CFLAGS)

OUTPUT := picalc_c_avx_omp

all:
    $(COMPILE_COMMAND) -o $(OUTPUT) picalc_c_avx_omp.c

clean:
    rm -f *.o $(OUTPUT)
```

- CC, ARCH, CFLAGS, COMPILE_COMMAND and OUTPUT are all variables, which are assigned with the lines:

  ```
  CC := gcc-6
  ARCH := core2
  CFLAGS := -march=$(ARCH) -mavx -O3 -m64 -std=c99 -fopenmp
  COMPILE_COMMAND := $(CC) $(CFLAGS)
  OUTPUT := picalc_c_avx_omp
  ```

- Note the use of “:=” in the assignments.
- The values of the variables, which will be text strings, will be inserted wherever the `$(variable)` syntax is used.
- It is common to have variables for:
  - the compiler name;
  - compiler flags;
  - separate linker flags;
  - lists of “include” directories;
  - locations of libraries.
- In the above example, the statement:

  ```
  $(COMPILE_COMMAND) -o $(OUTPUT) picalc_c_avx_omp.c
  ```

  will translate to:

  ```
  gcc-6 -march=core2 -mavx -O3 -m64 -std=c99 -fopenmp -o picalc_c_avx_omp picalc_c_avx_omp.c
  ```

7.3.4 Multiple targets in one makefile

Here is an example in which one makefile is used to build two separate programs:

```
prog1: myprog1.c
    gcc -c myprog1.c -o prog1

prog2: myprog2.c
    gcc -c myprog2.c -o prog2

all: prog1 prog2
```

Call this makefile with `make prog1` or `make prog2` to build the individual programs, or `make all` to build them both.

7.3.5 Conditional statements in makefiles

Sometimes you might want to create a makefile that can do different things depending on the circumstances. For example, a makefile might check the value of an environment variable, and alter its behaviour accordingly. The syntax for such a conditional statement in the general case is:

```
First-conditional-test
text if first test is true
else second-conditional-test
text if second test is true
else
text if first and second tests are false
endif
```

The four possible conditional tests are: `ifeq`, `ifneq`, `ifdef`, `ifndef`.

ifeq and **ifneq:** testing variables or environment variables for equality or lack of it.

ifdef and **ifndef:** checking whether or not a variable or environment variable is defined.

Environment variables are automatically passed into “make” and can be tested and used by name. For example:

```
ifdef CLANG
CC:= gcc
else
CC:= gcc-6
endif
```

or, alternatively:

```
CLANG:=no
ifeq ($(CLANG),yes)
CC:= gcc
else
CC:= gcc-6
endif
```

In the first case, the existence of the environment variable CLANG is used to choose the compiler to use while, in the second, the value of the local variable CLANG is used instead.

To set an environment variable outside a makefile, in the Linux bash shell, use for example:

```
export CLANG=yes
```

7.3.6 Makefiles for Fortran and Cython

The same rules as indicated above work equally well when compiling Fortran programs, or generating a Cython module:

Fortran :

```
myprog1.o: myprog1.for
    gfortran -c myprog1.for
myprog2.o: myprog2.for
    gfortran -c myprog2.for
all: myprog1.o myprog2.o
    gfortran myprog1.o myprog2.o -o myprog.exe
```

Cython :

```
all: picalc_pyx1.pyx
    cython -a picalc_pyx1.pyx
    python setup_picalc_pyx1.py build_ext -fi
```

TOPIC 8

Installing and using the Gnu Scientific Library (GSL)

8.1 Introduction

This section is likely to be of interest to C/C++ and Fortran programmers only. The GNU Scientific Library (GSL) is a collection of routines covering a wide range of topics in numerical computing. Routines are available in many areas, including:

> Complex Numbers, Vectors and Matrices, Linear Algebra, Fast Fourier Transforms, Eigensystems, Random Numbers, Differential Equations, Simulated Annealing, Numerical Differentiation, Interpolation, Root-Finding, and Least-Squares Fitting.

Full documentation is available in the manual at: `https://www.gnu.org/software/gsl/manual/gsl-ref.html` . The current version of GSL (Oct 2019) is version 2.6.

GSL is also available for Python, but see Section 8.6 for caveats regarding its use.

8.2 Installing and using GSL on Linux systems

Although there are many automated package managers on Linux systems for downloading and installing a range of applications and libraries, a simple approach that should work on any system is as follows:

1. Download the package from: `https://ftp.gnu.org/gnu/gsl/gsl-2.5.tar.gz`
2. Extract the archive using:

   ```
   tar -xf gsl-2.5.tar.gz
   ```

3. Change directory to the newly created gsl-2.5:

   ```
   cd gsl-2.5
   ```

4. Build GSL by running the following commands:

```
./configure --prefix=/usr/local --disable-static &&
make
```

Nothing will happen until you type the "make" command, but after you have started this, the building process will take quite some time.

5. Finally, as the *root* user, type:

```
make install
```

i.e. instead of logging in as root, you can just type:

```
sudo make install
```

and enter the root password when requested.

To add GSL libraries when compiling and linking with gcc, find the appropriate include directory path by typing:

```
gsl-config --cflags
```

and the necessary libraries and their location using:

```
gsl-config --libs
```

So, a simple compilation might look like:

```
gcc myprog.c -I/usr/local/include -L/usr/local/lib -lgsl -lgslcblas
```

Alternatively, include `gsl-config` directly within the compilation command as in:

```
gcc myprog.c $(gsl-config --cflags) $(gsl-config --libs)
```

The `gsl-config` commands can also be used in a makefile to specify the correct compiler flags.

8.3 Using GSL on BlueCrystal

GSL version 2.3 is already installed on BlueCrystal phase 4, and can be accessed by using the command:

```
module add GSL/2.3-foss-2017a
```

To add GSL libraries when compiling and linking with gcc, add the flags "-lgsl -lgslcblas" to your gcc or icc command.

8.4 Installing and using GSL on the Mac

Follow the instructions given above in Section 8.2 for Linux systems.

8.5 GSL on Windows with Microsoft Visual Compilers

If you are using the Microsoft MSVC compilers, you will need a version of GSL that is specially compiled for this system. Unfortunately, there are many incompatibilities ready to catch the unwary relating to 32-bit versus 64-bit, and different versions of the MSVC compiler so that, undoubtedly, the simplest approach is to use some sort of package manager. A simple command-line approach is to use the “nuget” utility, available from:

```
https://www.nuget.org/downloads
```

Be sure to download the latest x86 command-line version of nuget.exe; place this somewhere that your search path will be able to find it. Search on the www.nuget.org website for versions of GSL. You should find 64-bit versions available for both MSVC 2015 and MSVC 2017. Using the information on the webpage, type the appropriate nuget command at the standard Windows command prompt. In my case, I downloaded a 64-bit version of GSL 2.4 using the command:

```
nuget gsl-msvc14-x64 -Version 2.4.0.8788
```

The resulting installation will be placed in a directory with name derived from the version information above. In my case, it was called:

```
gsl-msvc-x64.2.4.0.8788
```

To compile and link a GSL-based program proceed as follows:

1. First open the **Developers Command** Line which will be found from the Start menu in the Visual C++ Build Tools folder. Choose “Visual C++ 2015 x64 Native Tools Command Prompt” **or** “Visual C++ 2015 x64 Native Tools Command Prompt” (they are the same) to open a 64-bit command prompt (or the 2017 equivalents if you have those).

2. Identify the locations of the `gsl` include directory, and the `gsl.lib` and `cblas.lib` libraries. In my installation they are found at:

   ```
   c:\Users\simon\gsl-msvc-x64.2.4.0.8788\build\native
   ```

   and:

   ```
   c:\Users\simon\gsl-msvc-x64.2.4.0.8788\build\native\lib\x64\Release
   ```

   respectively.

3. Compile and link using, for example:

   ```
   cl test.c /I:"c:\Users\simon\gsl-msvc-x64.2.4.0.8788\build\native" /link
      /libpath:"c:\Users\simon\gsl-msvc-x64.2.4.0.8788\build\native\lib\x64\Release"
      gsl.lib cblas.lib
   ```

8.6 Using GSL with Python

Although GSL has been made available to Python and Cython users, through modules PyGSL and CythonGSL, these are not available on BlueCrystal phase 4. Most of the features of GSL are also available in the Python modules NumPy, and SciPy, so it is recommended that they are used instead.

TOPIC 9

Example: Parallel methods for solving the heat equation using Cython and mpi4py

9.1 Introduction

The equation to be solved is:

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = \frac{1}{k}\frac{\partial u}{\partial t}$$

which is to be solved at each point on a rectangular lattice, taking differences between lattice points to provide estimates of gradients. The variable u typically represents temperature but, since this equation is also known as the diffusion equation, it might be some other physical quantity such as concentration. The boundary conditions are typically given as:

initial temperature distribution: $u_0(x, y) = f(x, y)$;

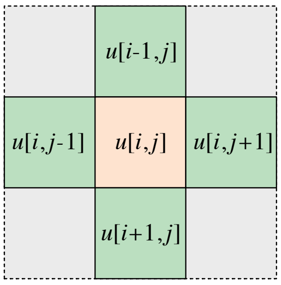
boundaries: $u_t(0, y) = u_t(L_x, y) = u_t(x, 0) = u_t(x, L_y) = 0.$

On discretisation the time stepping equation for each lattice site $[i, j]$ looks like:

$$u_{t+\Delta t}[i,j] = u_t[i,j] + c_x\left(u_t[i+1,j] - 2u_t[i,j] + u_t[i-1,j]\right) + c_y\left(u_t[i,j+1] - 2u_t[i,j] + u_t[i,j-1]\right)$$

$$\text{i.e.}\quad u_{t+\Delta t}[i,j] = u_t[i,j](1 - 2c_x - 2c_y) + c_x\left(u_t[i+1,j] + u_t[i-1,j]\right) + c_y\left(u_t[i,j+1] + u_t[i,j-1]\right)$$

where elements of a 2-dimensional array $u[i, j]$ represent the temperature at points on the lattice, and $c_\eta = k\Delta t/(\Delta\eta)^2$.
Clearly, each update of $u[i, j]$ involves 4 neighbours only, as shown in the figure. Note the designation i = row, j = column.



9.2 Basic method

For the full Python listing see `heat2D_python.py`. The method relies on an array `u[2,NXPROB,NYPROB]` where `NXPROB,NYPROB` represent the dimensions of the rectangular lattice, and the first index, 0 or 1, is used for alternate iterations of the method:

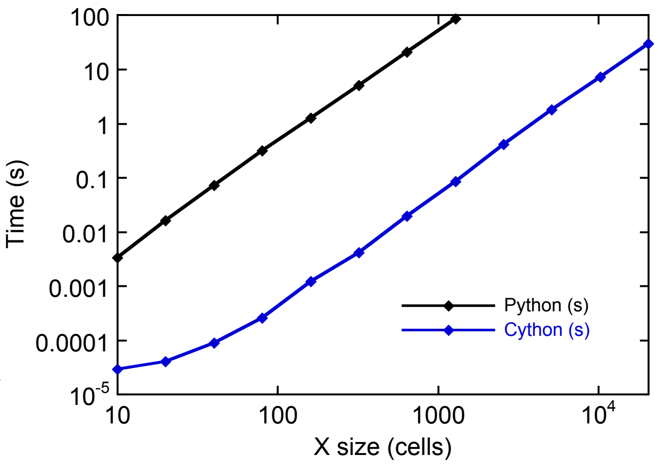
```
import numpy as np
```

```
def main( PROGNAME, STEPS, NXPROB, NYPROB ):
    u = np.zeros((2,NXPROB,NYPROB))  # array for lattice
    inidat(NXPROB, NYPROB, u)        # initialise lattice
    iz = 0
    # Now call update() repeatedly to update the value of grid points
    for it in range(STEPS):
        update(NXPROB,NYPROB,u[iz],u[1-iz])  # passing two sub-arrays separately
        iz = 1 - iz                          # swap the old and new arrays
```

Where the update function is defined by:

```
def update(nx, ny, u1, u2):
    temp=1-2.0*(Cx+Cy)         # this variable  saves time in the loops
    for ix in range(1,nx-1):
        for iy in range(1,ny-1):
            u2[ix,iy] = u1[ix,iy]*temp +
                        Cx*(u1[ix+1,iy] + u1[ix-1,iy]) +
                        Cy*(u1[ix,iy+1] + u1[ix,iy-1])
```

Note the use of the variable `temp` to reduce the number of multiplications performed within the nested loops. This knocks about 20% off the run times for the code. Run times for this program are shown in the figure (black line). The times scale approximately with the square of the side length, as expected (slope of 2 on the graph).



9.3 Basic Cython conversion

(See file `heat2D_cython.pyx`). A basic conversion to Cython consists of (a) introducing `cdef` statements for all variables and (b) using typed memory views for the key arrays. Point (a) makes substantial improvements to the speed of **for**-loops, while the combination of (a) and (b) is essential for quick data access in arrays.

The only change required in the main program, apart from use of `cdef` and separating the code into a '.pyx' file and a 'run.py' file, is the typed memory view for the main array. This is written as:

```
def main( PROGNAME, int STEPS, int NXPROB, int NYPROB ):
    cdef double[:,:,::1] u = np.zeros((2,NXPROB,NYPROB),dtype=np.double) # array for grid
        # note that u[] is defined as a double precision floating point NumPy array
        # assigned to a 3-dimensional double precision memory view, with C style
        # memory access implied by the ::1 in the final dimension
    inidat(NXPROB, NYPROB, u) # initialise lattice
    cdef:
        int iz = 0
        int it
    # Now call update() repeatedly to update the value of grid points
    # (this part has not changed)
    for it in range(STEPS):
        update(NXPROB,NYPROB,u[iz],u[1-iz]);
        iz = 1 - iz
```

The changes to the update() function include the use of correct C type definitions for the loops variables and double precision arrays, the use of the typed memory view and the decorators that turn off checking of the array indices. Taken together, these can account for orders of magnitude improvements in speed.

```
cimport cython
import numpy as np
cimport numpy as np
ctypedef np.float64_t dtype_t   # the correct C type for a double precision variable

@cython.boundscheck(False)    # this is a good way of saving time
@cython.wraparound(False)     # if you are confident in your indexing
def update(Py_ssize_t nx, Py_ssize_t ny, double [:,::1] u1, double[:,::1] u2): # type defs
    cdef:
        dtype_t Cx = 0.1          # put blend factors in here as
        dtype_t Cy = 0.1          # not needed elsewhere;
        Py_ssize_t ix, iy         # this is correct type for loop variables
        dtype_t temp = 1-2.0*(Cx+Cy)
    for ix in range(1,nx-1):
        for iy in range(1,ny-1):
            u2[ix,iy] = u1[ix,iy]*temp +
                        Cx*(u1[ix+1,iy] + u1[ix-1,iy]) +
                        Cy*(u1[ix,iy+1] + u1[ix,iy-1])
```

Note in the above how `u1[]` and `u2[]` are received as 2-dimensional double precision memory views.

The Cython code is compiled and run in the usual way with appropriate ‘setup.py’ and ‘run.py’ scripts. The results of running it are shown in the graph on the previous page (blue curve). It can be seen that the basic Cython version runs approximately 2 orders of magnitude more quickly than the original Python code, while maintaining the same scaling behaviour of run time with model size.

9.4 Cython with OpenMP

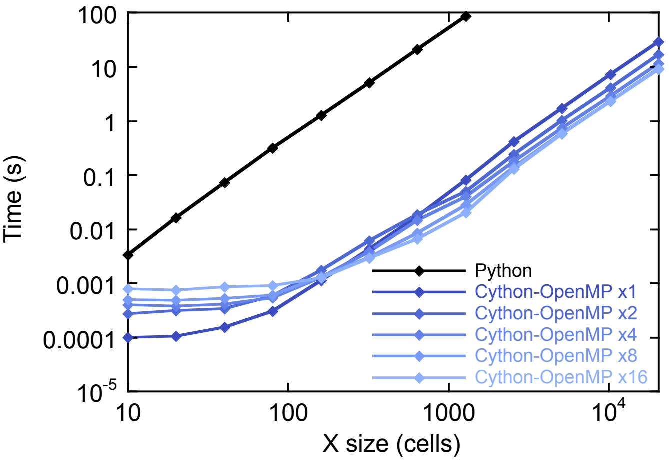
Conversion of Cython code to use OpenMP requires very little additional change (see file `heat2D_cython_omp.pyx`). It is necessary to import the `cython.parallel` module to access the **prange** command, and to import the `openmp` module to access OpenMP functions such as the timers. These are both C libraries – note the need to use OpenMP timing routines when timing multi-threaded code:

```
import numpy as np
from cython.parallel cimport prange  # import the parallel loop command
cimport openmp                       # import the OpenMP interface

def main( PROGNAME, int STEPS, int NXPROB, int NYPROB, int threads ):
    # no changes here
    initial = openmp.omp_get_wtime()
    #
    # remember to use the OpenMP timing routines
    #
    final = openmp.omp_get_wtime()
```

Note also the introduction of a `threads` variable that is required by `prange`. The only other change is the use of the `prange` command in the update() function:

```
@cython.boundscheck(False)
@cython.wraparound(False)
def update(Py_ssize_t nx, Py_ssize_t ny, double [:,::1] u1, double[:,::1] u2,
           int threads): # threads added
    cdef:
        dtype_t Cx = 0.1          # no change here
        dtype_t Cy = 0.1
        Py_ssize_t ix, iy
        dtype_t temp = 1-2.0*(Cx+Cy)
    for ix in prange(1,nx-1, nogil=True, num_threads=threads): # note use of prange in
        for iy in range(1,ny-1):                               # outer loop
            u2[ix,iy] = u1[ix,iy]*temp +
                        Cx*(u1[ix+1,iy] + u1[ix-1,iy]) +       # this part not changed
                        Cy*(u1[ix,iy+1] + u1[ix,iy-1])
```

Note that the `prange` command is very slow to initialise and, as a result, it is best kept to the outer loop of nested loops. The overhead associated with using `prange` can be appreciated by looking at the timings for 4 threads and 16 threads in the figure to the right. It is clear that there is a disadvantage in using multiple threads for model sizes less than approximately 500×500, with run times getting worse as the number of threads increases – clearly this is back to front! On the other hand, for the larger models, we observe the correct scaling of run time with model size. However, the run times don't decrease linearly with the number of threads and, overall, there appears no benefit in using 16 threads compared with 8 threads, say. This is probably both model and hardware dependent. Overall, we see approximately a 3- to 4-fold improvement of speed when using 16 threads.



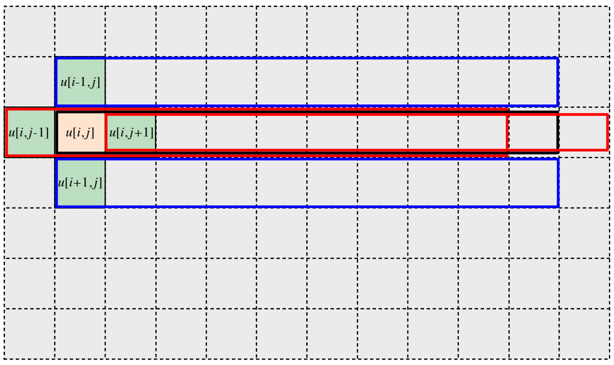
It is also worth noting that the single thread speed is almost exactly the same as the basic Cython code without OpenMP.

9.5 Python code with NumPy vectorisation

As noted elsewhere, considerable speed-ups are possible in Python code when exploiting the use of NumPy vectors. The requirements for vectorisation are:

1. Arrays should be declared as NumPy arrays e.g. using `numpy.zeros()` or other similar forms;
2. The arrays used should be the same size and shape;
3. Any maths functions used in vector commands should the NumPy variants i.e. use `numpy.sqrt()` but not `math.sqrt()`.

The aim here will be to process a single row of the 2-dimensional lattice in each step, reducing the 2 nested loops in the update() function to a single loop. (As noted previously, a two-dimensional vectorisation is possible but is less efficient overall). In the NumPy variant of the code (see `heat2D_python_numpy.py`) the only change from the basic Python version is within the update() function where, instead of the inner loop, we specify 5 overlapping rows from `u1[]`, each of size $(1, ny-2)$, but each with a different offset as indicated in the figure and the code clip:

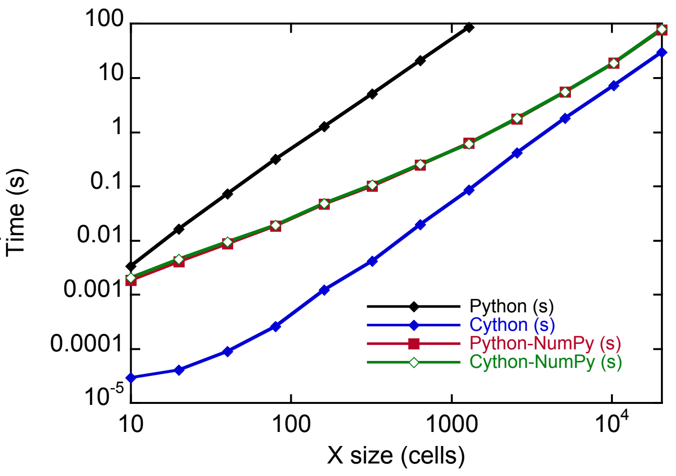


```
def update(nx, ny, u1, u2):
    temp = 1-2*(Cx+Cy)
    for ix in range(1,nx-1):
        u2[ix,1:ny-1] = u1[ix,1:ny-1]*temp +                  # note use of vectors
```

```
                        Cx*(u1[ix+1,1:ny-1] + u1[ix-1,1:ny-1]) +    # all same length but
                        Cy*(u1[ix,0:ny-2] + u1[ix,2:ny])            # offset as per diagram
```

The timings for the NumPy variant of the code are shown in the following graph, compared with the raw Python version and the Cython version. It can be seen that NumPy beats the raw Python for all model sizes, with the improvement increasing for the larger models. At no point does it quite equal the performance of Cython, but it comes close with the largest model simulated. Also shown in the graph is the performance of a Cythonised NumPy code. The timings are essentially identical for all model sizes, suggesting that there is no additional benefit to be gained from combining the two approaches (see following section).

9.6 Combining NumPy with Cython – a lot of work for no tangible benefit



(See file `heat2D_cython_numpy.pyx`). Running Cython on NumPy vectorised code brings added complexity to the program with no obvious benefit. Although it is remarkable that Cython is able to handle the vector code, the biggest issue is that the vectors are incompatible with the typed memory views that were introduced to make the Cythonised array access more efficient. To get around this, it is necessary to convert between typed memory views and NumPy arrays on each iteration of the method, as shown in the update() function code here:

```
def update(int nx, int ny, double [:,::1] u1p, double[:,::1] u2p):
    cdef:
        int ix
        double temp = 1-2*(Cx+Cy)
    u1 = np.asarray(u1p,dtype=np.float64)  # converting from typed memory
    u2 = np.asarray(u2p,dtype=np.float64)  # views to NumPy arrays
    for ix in range(1,nx-1):
        u2[ix,1:ny-1] = u1[ix,1:ny-1]*temp +
                        Cx*(u1[ix+1,1:ny-1] + u1[ix-1,1:ny-1]) +
                        Cy*(u1[ix,0:ny-2] + u1[ix,2:ny])
    u1p = u1  # converting back to
    u2p = u2  # memory views
```

9.7 Distributed processing with mpi4py (including NumPy)

The basic functioning of the MPI version of the Python code is as follows (see file `heat2D_python_mpi4py.py`):

1. The master rank initialises the data array, and decides how many rows of the array to send to each worker. In this version, all ranks initialise the same size data array, with each worker then just using a part of the array. A more elegant (and memory efficient) approach would be for each rank to only create a sub-array, large enough to handle just the amount of data it needs and no more:

```
def main( PROGNAME, STEPS, NXPROB, NYPROB ):
    u = np.zeros((2,NXPROB,NYPROB)) # Array for grid; all ranks have same size array
    comm = MPI.COMM_WORLD
    taskid = comm.Get_rank()         # First, find out my taskid and
    numtasks = comm.Get_size()       # how many tasks are running
    numworkers = numtasks-1
    if taskid == MASTER:             # --------- Master only ---------
        inidat(NXPROB, NYPROB, u)    # Initialize grid
        averow = NXPROB//numworkers          # Figure out how many rows to
        extra = NXPROB%numworkers            # send to each worker and what
        for i in range(1,numworkers+1):      # to do with extra rows
            rows = averow
            if i <= extra:
                rows+=1
```

2. The master sends messages to each worker containing the number of rows, the offset of the rows into the data array, the ranks of the neighbours above and below in the array, and finally the segment of the data itself. Note use of `.send` and `.Send` for Python objects and NumPy arrays respectively.

```
    if taskid == MASTER:                # --------- Master only ---------
    ...
            if i == 1:                  # Figure out the neighbours
                left = NONE             # for each worker
            else:
                left = i - 1
            if i == numworkers:
                right = NONE
            else:
                right = i + 1
            comm.send(offset, dest=i, tag=BEGIN)     # Send offset, number of rows,
            comm.send(rows, dest=i, tag=BEGIN)       # above and below neighbours
            comm.send(above, dest=i, tag=BEGIN)       # and a section of the data array
            comm.send(below, dest=i, tag=BEGIN)      # to each of the workers
            comm.Send(u[0,offset:offset+rows,:], dest=i, tag=BEGIN)
            offset += rows
```

3. The workers receive their data, and iterate through their section of the array.

```
    elif taskid != MASTER:                            # --------- Worker only ---------
        offset = comm.recv(source=MASTER, tag=BEGIN) # Receive parameters and
        rows = comm.recv(source=MASTER, tag=BEGIN)   # data array from Master
        above = comm.recv(source=MASTER, tag=BEGIN)
        below = comm.recv(source=MASTER, tag=BEGIN)
        comm.Recv([u[0,offset,:],rows*NYPROB,MPI.DOUBLE], source=MASTER, tag=BEGIN)
        ...
        for it in range(STEPS):     # Begin doing STEPS iterations
```

4. On each timestep, the workers exchange their bordering rows with their neighbours, to keep them up to date.

```
    if above != NONE:   # --------- Workers only ---------
        req=comm.Isend([u[iz,offset,:],NYPROB,MPI.DOUBLE], dest=above, tag=RTAG)
        comm.Recv([u[iz,offset-1,:],NYPROB,MPI.DOUBLE], source=above, tag=LTAG)
    if below != NONE:   # The top and bottom workers only have one neighbour
        req=comm.Isend([u[iz,offset+rows-1,:],NYPROB,MPI.DOUBLE], dest=below, tag=LTAG)
        comm.Recv([u[iz,offset+rows,:],NYPROB,MPI.DOUBLE], source=below, tag=RTAG)
```

5. When the iterations are complete, the workers send their data back to the master rank;

```
    # Finally, send workers portion of final results back to master
        comm.send(offset, dest=MASTER, tag=DONE)
        comm.send(rows, dest=MASTER, tag=DONE)
        comm.Send([u[iz,offset,:],rows*NYPROB,MPI.DOUBLE], dest=MASTER, tag=DONE)
```

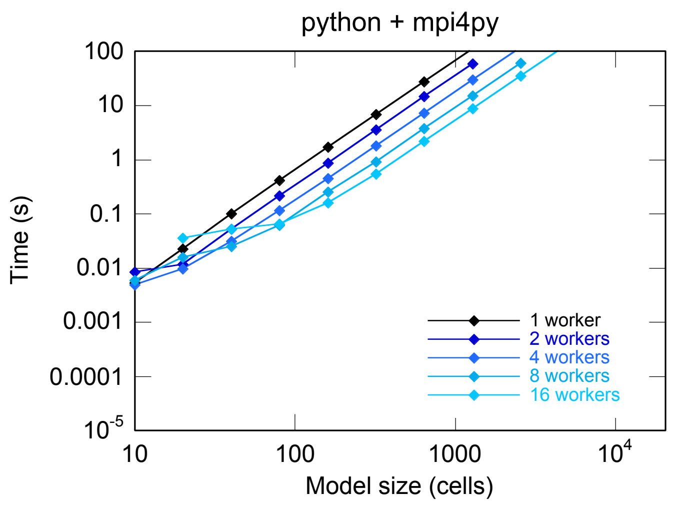
6. The master rank re-combines all the array segments into a single final array.

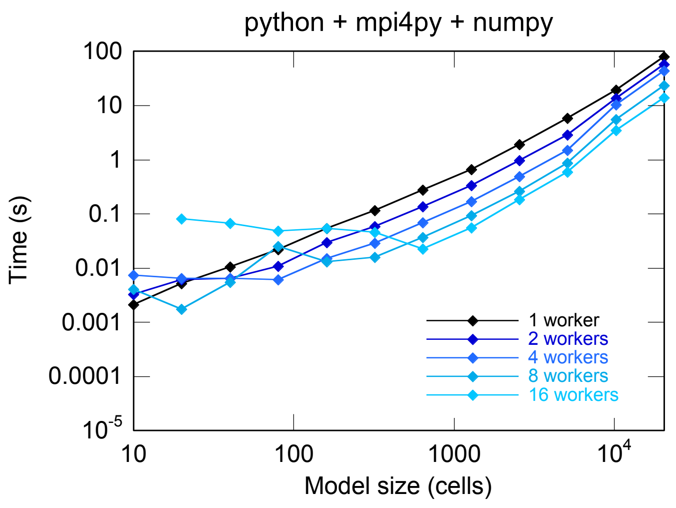
```
        # Now wait for results from all worker tasks
        for i in range(1,numworkers+1):
            offset = comm.recv(source=i, tag=DONE)
            rows = comm.recv(source=i, tag=DONE)
            comm.Recv([u[0,offset,:],rows*NYPROB,MPI.DOUBLE], source=i, tag=DONE)
```

In the method described, the communication is all performed using point-to-point messages with simple send and receive messages. An alternative using 'broadcast' and 'gather' for the initial distribution of data would also be possible but, since this is only required once at the start and end, there would be little performance benefit.

Because the method described works on whole numbers of rows, it is well suited to NumPy vectorisation, and this is illustrated in the file `heat2D_python_mpi4py_numpy.py`.

Timings for both programs, together with the raw Python code, are illustrated in the following graphs.





It can be seen that the mpi4py code without numpy (left plot) scales very well with the number of workers, once the model size exceeds about 200×200 cells, with the 16 worker version showing a speed-up of 12.5 times the single worker version. The numpy version of the code (right plot) shows similar speed-ups for models in the mid-size range, but drops to only 6 times speed up for the larger models. However, the NumPy version of the code is typically 100 to 200 times faster than the plain python version.

9.8 mpi4py combined with Cython

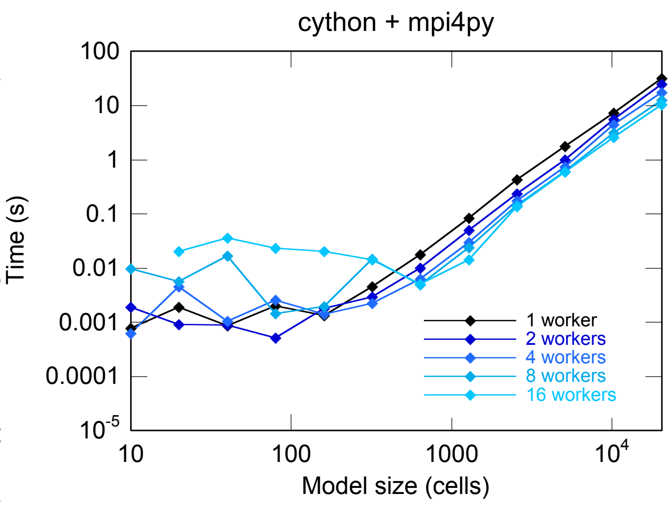
Using Cython with the mpi4py version of the code is very straightforward (see `heat2D_cython_mpi4py.pyx`). The main adaptations are:

- use the Cython version of the "update" function from Section 9.3;
- move the lines of code that actually call the main program into the "run" python script, as follows:

```
import sys
from heat2D_cython_mpi4py import main

if int(len(sys.argv)) == 4:
    PROGNAME = sys.argv[0]
    STEPS = int(sys.argv[1])
    NXPROB = int(sys.argv[2])
    NYPROB = int(sys.argv[3])
    main(PROGNAME, STEPS, NXPROB, NYPROB)
```

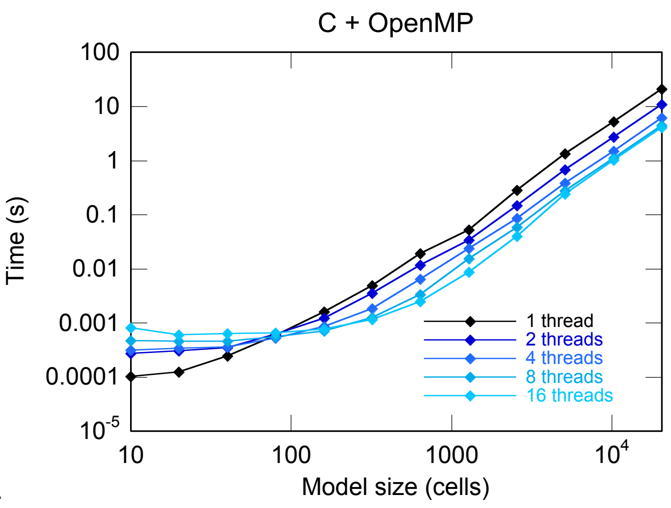
```
else:
    print("Usage: {} <ITERATIONS> <XDIM> <YDIM>".format(sys.argv[0]))
```

Fully cythonised mpi4py code: Timings are shown on the right for mpi4py + Cython. Overall it appears that there is a modest advantage over using mpi4py + NumPy. However, using 16 workers only results in a speed-up of a factor of about 3 compared with the single worker code.



Cythonised update function only: It is also possible to compromise, and only use Cython on the "update" function. In this case the "run" python script will contain most of the program including all of the mpi4py calls (see `heat2D_cython_mpi4py_kernel.pyx` and associated files). In fact, no performance benefit was observed in the present case by doing this. However, the advantage to the programmer in only needing to cythonise a single function is considerable.

C + MPI comparison: Timings were performed using a native C version of the code, using the standard MPI libraries. No performance benefit was found compared with the Cython + mpi4py code, suggesting that the communication time was dominating.



C + OpenMP comparison: Finally, a multi-threaded native C version of the code was tested. Timings are shown on the right, and should be compared with the graph in Section 9.4. Modest speed-improvements are observed, mainly for the larger models.

9.9 Conclusions

The overall conclusions are the following:

- The difference in speed between the slowest and fastest codes displayed above is about a factor of 10,000.
- The difference in speed between the fastest cythonised multi-threaded python and the equivalent native C code is only about a factor of 2.
- In some instances use of NumPy vectorisation can be almost as effective as using Cython, but it is not worth combining the two methods.
- The speed-ups from using mpi4py are comparable with (but usually a little worse than) those available from multi-threading. However, mpi4py should scale to 100's of workers, whereas multi-threading is limited to the number of cores available on a single motherboard (28 threads on BlueCrystal phase 4).