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: $u_0(x, y) = f(x, y)$;

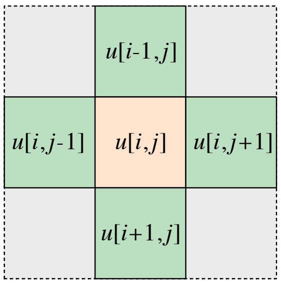
boundaries: $u_t(0, y) = u_t(L_x, y) = u_t(x, 0) = u_t(x, L_y) = 0$.

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:

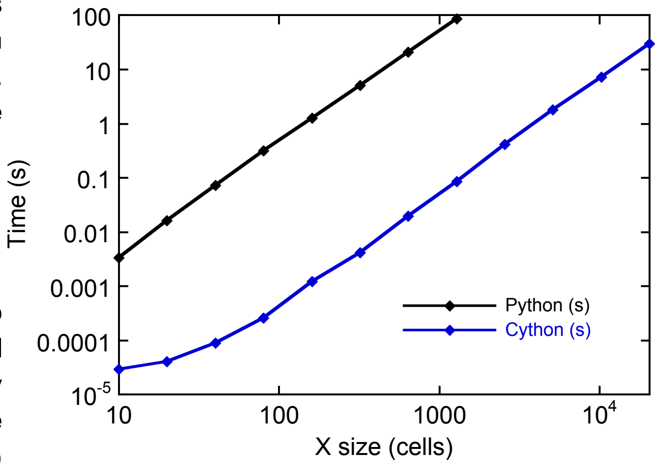
```python
import numpy as np
```

```python
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:

```python
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:

```python
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:

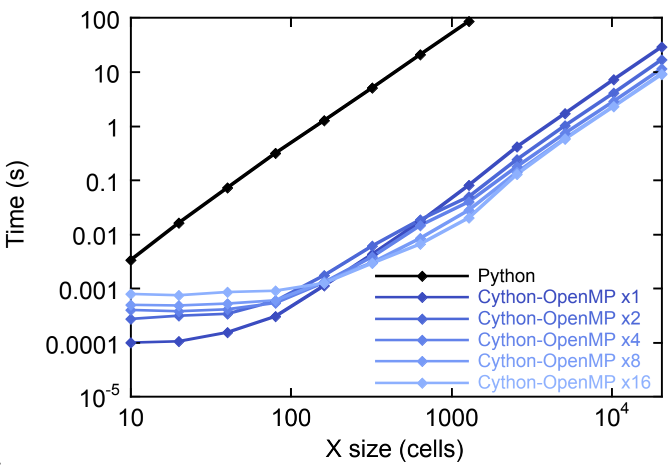
```
import numpy as np
from cython.parallel cimport prange  # import the parallel loop command
cimport openmp                       # import the OpenMP interface

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.



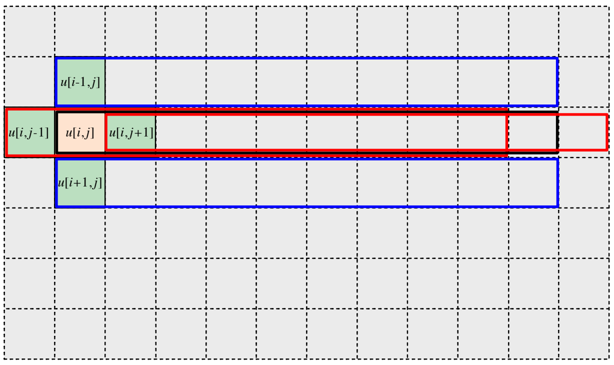
It is also worth noting that the single thread speed is almost exactly the same as the basic Cython code without OpenMP.

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:

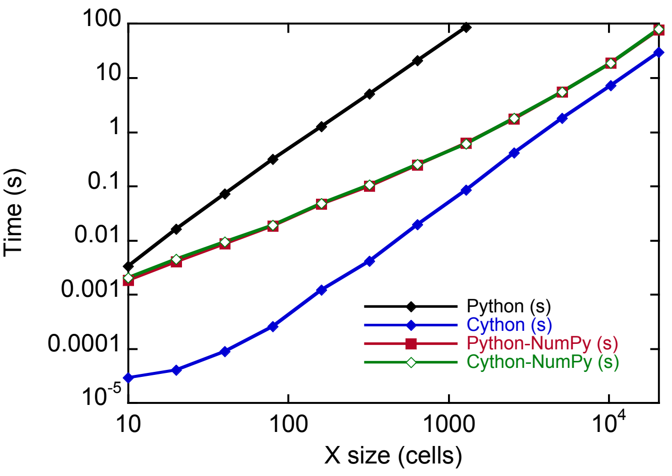


```
def update(nx, ny, u1, u2):
    temp = 1-2*(Cx+Cy)
    for ix in range(1,nx-1):
        u2[ix,1:ny-1] = u1[ix,1:ny-1]*temp +                    # note use of vectors
```

```
                        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)
```

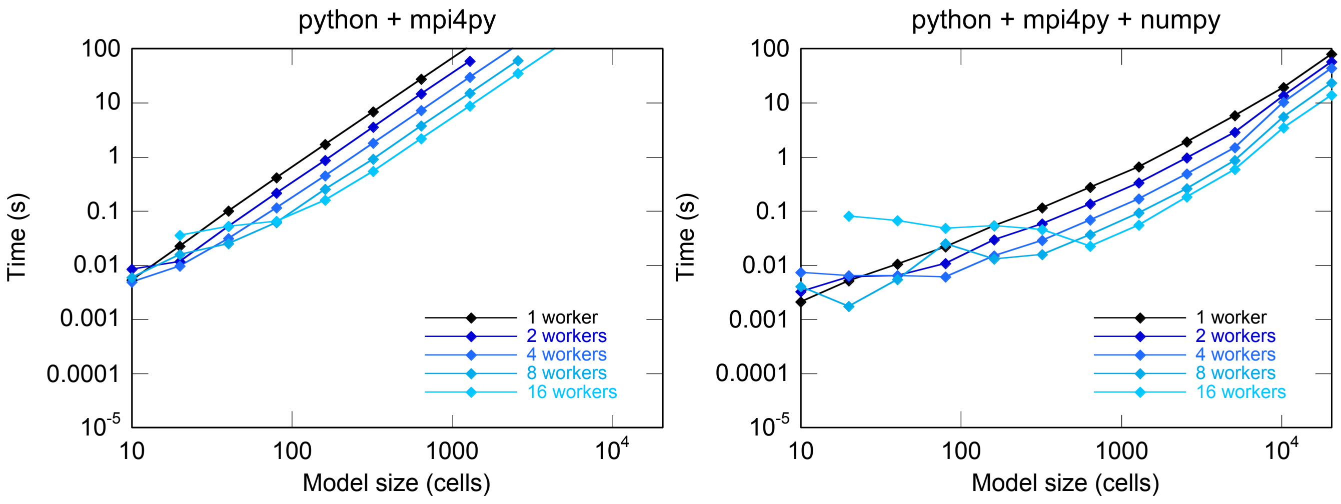
6. The master rank re-combines all the array segments into a single final array.

```
    # 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:

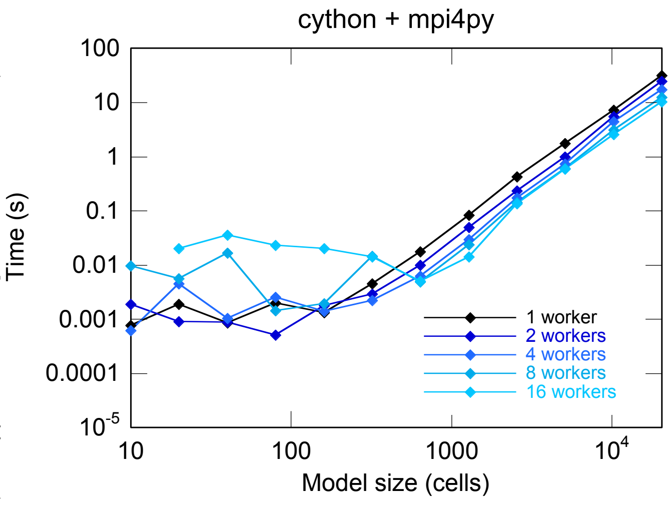
- use the Cython version of the "update" function from Section 9.3;
- move the lines of code that actually call the main program into the "run" python script, as follows:

```
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)
```

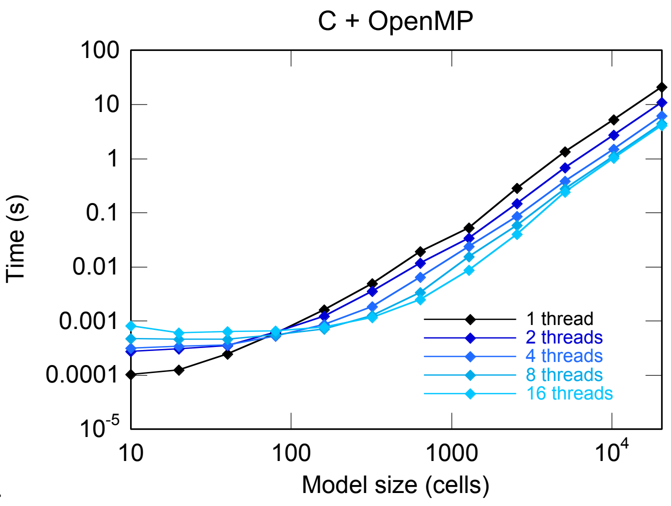
```
else:
    print("Usage: {} <ITERATIONS> <XDIM> <YDIM>".format(sys.argv[0]))
```

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).