PHYSM0032 Advanced Computational Physics
Lecture 2

Dr. Simon Hanna

September 28, 2021

3 Compilers and Interpreters: Introduction to Cython

3.1 Interpreted languages e.g. BASIC, Python

- Running the program consists of two passes:
  1. the whole program is interpreted into a set of "byte-codes"
  2. the byte-codes are passed through the emulator, which operates by calling binary functions which run on the processor.
- The need to determine the data-type for each variable adds a huge overhead to the Python program and generally makes it orders of magnitude slower than the equivalent compiled program.

3.2 Compiled languages

- There are many of these e.g. C/C++, Fortran, Algol, Cobol, designed with different aims in mind.
- They share one feature: the program is "compiled" (converted) into a binary "object" file, which is then "linked" with existing low-level binary libraries to create an "executable".
- The compiler usually aims to streamline the binary it produces, to produce a fast-running binary executable. This is often facilitated by strong typing of all variables.
- Compilers usually have a range of tricks that can be applied via compiler flags (see below).
- Compiled languages normally very fast.

3.3 Cython – the best of all worlds?

- Cython offers many of the advantages of Python, while offering the opportunity to compile code into an efficient binary with speed comparable to C/C++ or Fortran.
- Several levels of optimisation are possible:
  1. Do nothing. Cythonize the raw Python code. Some modest speed-ups (10-20%) may be observed.
  2. Type your variables. By using the `cdef` keyword, variables can be given specific C-like types e.g. int, float, double etc. This can produce a significant speed-up, especially for variables involved with loops. `cdef` may also be used for functions. Use of `cdef` can lead to 2x or 3x speed-ups.
  3. Call C functions. Rather than calling the Python maths or NumPy maths functions, use `cimport` to import C functions from the C maths library. Cythonizing this code can lead to execution times comparable to those from C/C++.
- Cython is also compatible with OpenMP and can be used to generate efficient multithreaded parallel code.
- Further details are available from the Cython website:
  http://www.cython.org

3.4 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.

3.4.1 A basic Cython script

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

```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.

3.4.2 A basic “setup” file

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

```python
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 MacOS or Linux (slightly different on Windows).

Assuming this works, you will be left with a library file ready to use with your run script.

3.4.3 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:

```python
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:

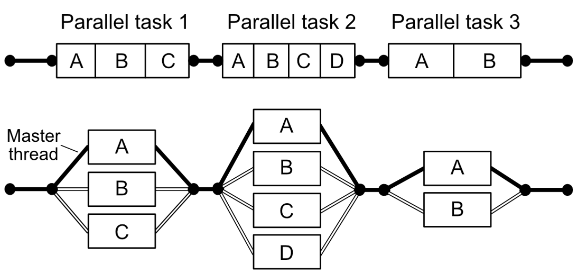
```
python run_picalc_pyx1.py 1000000
```

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.

4 Shared memory programming with threads

4.1 Introduction



- Programming with threads
- Threads have shared address space (unlike processes)
- Great for parallel processing on shared memory systems
- E.g.: quad-core—use 4 threads (8 with hyperthreading); threads closely tied to physical cores, but need not be (e.g. through multitasking)

4.2 Different threading models

- "old" pthreads library in C; "new" threads library in C++; generally awkward to use
- several options in python:
  - `threading` module: multiple threads on one processor
  - `asyncio` module: as above but ideal when waiting for i/o
  - `multiprocessing` module: multiple threads on multiple cores
- Here is a simple multiprocessing example:

```python
import multiprocessing
import time

def cpu_bound(number):
    return sum(i * i for i in range(number))

def find_sums(numbers):
    with multiprocessing.Pool() as pool:
        pool.map(cpu_bound, numbers)

if __name__ == "__main__":
    numbers = [5000000 + x for x in range(20)]

    start_time = time.time()
    find_sums(numbers)
    duration = time.time() - start_time
    print(f"Duration {duration} seconds")
```

4.3 OpenMP:

4.3.1 the de facto standard in parallel scientific programming

- OpenMP declares parallel tasks; the threads execute them in some order (shared memory essential)
- "pragma"-based directives are passed to the compiler
- Memory is shared – variables are shared as well unless declared as "private"
- Available in C/C++, Fortran, Cython but not Python
- GPUs handle threads with ease, and so will be included in OpenMP from version 4.5 onwards.

4.3.2 C example: loop iterations can be parallel

```c
#include <omp.h>
#pragma omp parallel default(none) \
        shared(n,x,y) private(i)
{
#pragma omp for
    for (i=0; i<n; i++){
        x[i] += y[i];
        }
    }
```

- the first `#pragma omp` indicates the start of a parallel block; in C the parallel block is bounded by braces {}.
- `#pragma omp for` indicates a parallel loop. The `for`-loop will be rolled out to the available threads, each taking different values of `i`.

- `default(none)` means there are no assumptions about which variables are shared between threads and which are private to individual threads
- `shared` lists those variables that all threads can access. Care is needed here to avoid threads trying to change the same variable simultaneously.
- `private` lists those variables that have a separate and independent copy in each thread.

4.3.3 Cython example:

In Cython parallel features are built upon the OpenMP library, but lots of things are implied i.e. the user has little control. The same code fragment could look something like:

```python
from cython.parallel cimport prange
cimport openmp

for i in prange(n, nogil=True):
    x[i] += y[i]
```

- `prange()` is a parallel loop
- `nogil=True` releases the global instruction lock, which prevents Python normally from running in parallel.
- variables are private or shared, dependent on whether they are written to, or only read, during the parallel section.

4.3.4 Fortran example:

For completeness, the Fortran equivalent is:

```fortran
!$OMP PARALLEL DEFAULT(NONE)
!$OMP& SHARED(N,X,Y) PRIVATE(I)
!$OMP DO
      DO I = 1, N
```

```
        X(I) = X(I)+Y(I)
      ENDDO
!$OMP END DO
!$OMP END PARALLEL
```

(Note: capitals no longer required in Fortran, but I will use to distinguish from C; Fortran is now case *in*-sensitive).

4.4 Practical considerations

- OpenMP available as a library for C/C++ and Fortran.
- Supplied with Gnu compiler gcc version 5 onwards and compatible versions of gfortran.
- Cython requires an OpenMP compatible C compiler to operate correctly.
- Resources:
  - Very useful online documentation from Lawrence-Livermore Laboratory
    https://hpc.llnl.gov/tuts/openMP/
    covers both C/C++ and Fortran interfaces.
  - Limited but sufficient Cython documentation is available here:
    https://cython.readthedocs.io/en/latest/src/userguide/parallelism.html
- See the PHYSM0032 Installation & Programming Notes on Blackboard for details of how to build and run Cython programs using OpenMP.

4.5 Program example: Hello World

4.5.1 C version

There is some value in trying the simple "Hello World" program as it is a useful diagnostic of whether or not you have working threads. Here is an example of a C program to demonstrate use of threads:

```c
#include <stdlib.h>
#include <stdio.h>
#include <omp.h>

int main()
{
#pragma omp parallel
  {
    int t = omp_get_thread_num();
    printf("Hello from thread %d \n",t);
  }
}
```

4.5.2 Cython version

The equivalent in Cython is:

```python
from cython.parallel cimport parallel
cimport openmp

cdef int thread

with nogil, parallel(num_threads=8):
    thread = openmp.omp_get_thread_num()
    with gil:
        print("Hello from thread %2d}" % (thread)
            )
```

- Note that the print function requires the gil, so the only code run in parallel is the call to `openmp.omp_get_thread_num()`.
- Note that parallel Cython is built using OpenMP, and many OpenMP constructs are performed implicitly. However, not all options are yet available.

4.5.3 Fortran version

The Fortran translation of the problem is:

```fortran
      PROGRAM HELLO
      INTEGER TID, OMP_GET_THREAD_NUM

!$OMP PARALLEL PRIVATE(TID)
      TID = OMP_GET_THREAD_NUM()
      PRINT *, 'Hello from thread = ', TID
!$OMP END PARALLEL

      END
```

4.5.4 Program output

Typical output from the above:

```
Hello from thread 2
Hello from thread 1
Hello from thread 0
Hello from thread 6
Hello from thread 4
Hello from thread 3
Hello from thread 5
Hello from thread 7
```

Note the lack of ordering as threads compete to get their output to the screen!

4.6 Program example: Pi calculation

4.6.1 C version

The most obvious candidate for parallelisation is a loop. Here is a program to calculate π using threads in a `for`-loop. The method is to integrate over a quarter circle using Riemann sums. It is very inefficient, but ideal as a demonstration.

```c
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <time.h>
#include <omp.h>
#define THREADS 8

int main( int argc, char* argv[] )
{
    omp_set_num_threads( THREADS );
```

```c
    double pibyfour = 0.0;
    double pi;
    double initial, final;
    int nmax = 1000000000;
    double dx = 1.0/nmax;

    printf("Threads set to %d\n",THREADS);
    initial = omp_get_wtime();

#pragma omp parallel for reduction(+:pibyfour)
    for( int 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;
}
```

4.6.2 Fortran version

An equivalent Fortran program is:

```fortran
      PROGRAM PICALC

      INTEGER NMAX, NTHREADS, I
      DOUBLE PRECISION DX, PI, PIBYFOUR
      DOUBLE PRECISION INITIAL, FINAL
      DOUBLE PRECISION OMP_GET_WTIME

      NMAX = 1000000000
      NTHREADS = 8
      DX = 1.0 / NMAX
      PIBYFOUR = 0.0

      CALL OMP_SET_NUM_THREADS (NTHREADS)
      PRINT *, 'Threads set to ',NTHREADS
      INITIAL = OMP_GET_WTIME()
```

```fortran
!$OMP PARALLEL DO PRIVATE(I) REDUCTION(+:PIBYFOUR
     )
      DO I = 0, NMAX-1
         PIBYFOUR = PIBYFOUR + SQRT(1-(I*DX)**2)
      ENDDO
!$OMP END PARALLEL DO

      FINAL = OMP_GET_WTIME()
      PRINT *, 'Elapsed time (s): ',FINAL-INITIAL
      PI = 4.0 * PIBYFOUR * DX
      PRINT *, 'Pi = ', PI
      END
```

4.6.3 Cython version

A Cython function to perform the same role is given by:

```python
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
```

```python
    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
```

Note that the “reduction” clause (see C version) is implicit in Cython.

4.7 Cython with OpenMP

- As discussed above, 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 used during the setup process (see file setup_picalc_pyx_omp.py).
- For the Microsoft MSVC compiler change the `-fopenmp` flag to `/openmp`.

```
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))
```

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.py) 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 the number of threads to use.

Exercise Week 1

Exercise Week 1

- Apply for an account on BlueCrystal Phase 4 (see Installation and Programming notes on Blackboard). Your account may take a day or two to come through.
- Make sure you are able to load the appropriate programming environment for your chosen language i.e. the correct version of Anaconda python, or the appropriate C/C++ (or Fortran) compiler.
- Try uploading and running some of the test programs provided on Blackboard, especially the multithreaded code for calculating π. *Hint: you can do very quick test runs at the terminal, but anything longer than about 1s should be submitted to the job queue.*
- Try submitting longer jobs to the queue and experiment with jobs requiring different numbers of threads.
- Produce a timing graph for calculating π with different numbers of threads, for up to 28 threads (one node).
- Email your graph to Dr Hanna before the next lecture.