PHYSM0032 Advanced Computational Physics Lectures 3 & 4

Dr. Simon Hanna

October 5, 2021

5 More OpenMP: directives and functions

5.1 OpenMP directives

This is a brief survey of the use of OpenMP directives. For full details of the C and Fortran interfaces see *https://hpc.llnl.gov/tuts/openMP/*

More details of the use of threads in Cython may be found at *http://docs.cython.org/en/latest/src/userguide/parallelism.html*

5.1.1 parallel

This is the simplest and most useful directive. In a parallel code segment, all threads will execute the same statements as seen previously in `C/C++`, `Fortran` and `Cython`:

```
C/C++
#pragma omp parallel num_threads(8)
  {
   // this is executed by a team of threads
  }
```

```
Fortran
!$OMP PARALLEL NUM_THREADS(8)

!this is executed by a team of threads

!$OMP END PARALLEL
```

```
Cython
from cython.parallel import parallel

with nogil, parallel(num_threads=8):
    # this is executed by a
    # team of threads
```

In the following example, a function call is used to obtain the thread number and direct the program flow accordingly:

```
C/C++
double result,fresult,gresult,hresult;

#pragma omp parallel num_threads(3) private(num)
      { int num = omp_get_thread_num();
        if (num==0)       fresult = f(x);
        else if (num==1) gresult = g(x);
        else if (num==2) hresult = h(x);
      }
result = fresult + gresult + hresult;
```

```
Cython
from cython.parallel import parallel
cimport openmp

with nogil, parallel(num_threads=3):
    num = openmp.omp_get_thread_num()
    if (num==0):
        fresult = f(x)
    elif (num==1):
        gresult = g(x)
    elif (num==2):
        hresult = h(x)
result = fresult + gresult + hresult
```

5.1.2 critical

```
C/C++
#pragma omp critical
  {
  // Executed by one thread at a time
  }
```

```
Cython
from cython.parallel import parallel
with nogil, parallel:
    with gil:
        # One thread at a time
```

- The typical application of a critical section is to update a variable, avoiding a *race* condition (i.e. multiple threads trying to update same variable at the same time):

```
#pragma omp parallel
  {
```

```
  int mytid = omp_get_thread_num();
  double tmp = some_function(mytid);
#pragma omp critical
  sum += tmp;
  }
```

```
from cython.parallel import parallel

with nogil, parallel:
    mytid = openmp.omp_get_thread_num()
    tmp = some_function(mytid)
    with gil:
        sum += tmp
```

- Can be inefficient, as a “lock” is required on the other threads which takes many processor cycles.
- In a loop, the “reduction” clause is more efficient (see below).

5.1.3 parallel for

A “for” loop can be placed in a parallel section, and manually divided between the threads. However, this can be achieved more naturally using:

```
C/C++
#pragma omp parallel
#pragma omp for
     for (int i=0; i<N; i++) {
       // do something with i
     }
```

```
Cython
from cython.parallel import parallel, prange

for i in prange(n, nogil=True):
    # do something with i
```

- Each value of “i” will be taken by the next available thread.
- The results for each “i” will not necessarily be available in sequence.
- Without the “for” directive, each thread would run the whole loop.

In C or Fortran, if the loop is the only code within the parallel segment, the two directives may be combined as in:

```
C/C++
#pragma omp parallel for
     for (i=0; i<N; i++) {
       // do something with i
     }
```

```
Fortran
!$OMP PARALLEL DO PRIVATE(I)
        DO I = 1, N
!         // do something with I
        ENDDO
!$OMP END PARALLEL DO
```

- There are some constraints on the operation of the loop:
  - OpenMP needs to know how many iterations there will be, which means...
  - No break, return or exit statements within the loop;
  - The index update must be a constant;
  - The loop index is private to the loop and cannot be changed.
- If all iterations of the loop are incrementing a single variable, the “reduction” clause is used:

```
C/C++
#pragma omp parallel for reduction(+:pibyfour
    )
    for( int i=0; i < nmax ; i++ ) {
        pibyfour += sqrt(1-pow(i*dx,2));
        // Reduction explicitly requested
    }
```

```
Cython
from cython.parallel import prange

for i in prange(nmax, nogil=True):
    pibyfour += sqrt(1-(i*dx)**2)
    # Cython infers the reduction
        automatically
    # provided '+=' is used
```

- Reduction clauses are available for the $+$, $-$, $*$, min, max operators.

5.1.4 Loop scheduling

The “schedule” clause is used to control the number of iterations taken by each thread, as in:

```
C/C++
#pragma omp for schedule(static[,chunk])
#pragma omp for schedule(dynamic[,chunk])
#pragma omp for schedule(guided[,chunk])
```

```
Cython
for i in prange(n,nogil=True,schedule='static'
                            [,chunksize=value]):
for i in prange(n,nogil=True,schedule='dynamic'
                            [,chunksize=value]):
for i in prange(n,nogil=True,schedule='guided'
                            [,chunksize=value]):
```

- The optional “chunk” controls the number of loop iterations assigned to each thread, in each pass.
- “chunk” needs to be tuned and may be dependent on cache sizes etc.

Static schedule is fixed at compile time, and is safe if all iterations roughly the same length.

Dynamic schedule is determined at run time. Blocks of iterations are placed in queue and run by whichever thread becomes free next.

Guided scheduling gradually reduces the chunk size on basis that this will ease load balancing towards the end of the loop.

5.1.5 Nested loops

- In C (or Fortran), the “for” directive only operates on the outer loop of nested loops.
- In C (or Fortran), if loops are perfectly nested i.e. no other commands are between the “for” statements, loop collapsing can be used:

```
C/C++
#pragma omp for collapse(2)
    for ( i=0; i<N; i++ )
        for ( j=0; j<N; j++ )
            A[i][j] = B[i][j] + C[i][j];
```

```
Cython
# Cython does not currently support
# nested pranges
```

- Note: this is **only allowed** when all iterations are independent of each other and order of calculation is not important.

5.1.6 Sections

If parallel tasks do not make use of a loop, the “sections” directive may be used:

```
C/C++
#pragma omp sections
    {
#pragma omp section
      // one calculation
#pragma omp section
      // another independent calculation
    }
```

```
Cython
# Cython does not currently support sections.
# Similar behaviour can be achieved using
# parallel() and allocating work based on
# thread ID.
```

Within the “sections” code segment, the individual “section” directives indicate regions that will be taken by different threads, hopefully simultaneously.

5.2 OpenMP functions

OpenMP has a number of settings that can be set through environment variables, and both queried and set through library functions. The functions available are:

```
omp_set_num_threads              omp_get_num_threads
omp_get_max_threads              omp_get_thread_num
omp_get_num_procs                omp_in_parallel
omp_set_dynamic                  omp_get_dynamic
omp_set_nested                   omp_get_nested
omp_get_wtime                    omp_get_wtick
omp_set_schedule                 omp_get_schedule
omp_set_max_active_levels
    omp_get_max_active_levels
omp_get_thread_limit             omp_get_level
omp_get_active_level
    omp_get_ancestor_thread_num
omp_get_team_size
```

Further details can be found at:
https://hpc.llnl.gov/tuts/openMP/#RunTimeLibrary

- In C, use `#include <omp.h>` to access these functions;
- Cython equivalents are contained in the “openmp” module.

6 Parallel Linear Algebra with OpenMP

6.1 Dense products

- Matrix-vector and matrix-matrix products easy to parallelise because order of calculations not important.
- Shared memory suited to such calculations.
- Distributed memory systems introduce complications due to parts of matrix being held by different processors.

- Need to think about row-column ordering when accessing large amounts of memory.

6.1.1 Memory access

- For example, in C a 2-d array will be stored in *row-major* order i.e. the arrays `A[i][j]` and `x[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],...` `x[1], x[2], x[3],...` and the matrix-vector product:

  $$A_{ij}x_j = y_i \qquad (j = 1 \dots n)$$

  will involve the multiplication of two contiguous blocks of memory (each row of A will be contiguous) to generate each element of y.

- In Fortran however, a 2-d array will be stored in *column-major* order i.e.

  `A[1][1], A[2][1], A[3][1],...,`

  `A[1][2], A[2][2], A[3][2],...,`

  `A[1][3], A[2][3], A[3][3],...`

  `x[1], x[2], x[3],...` and strided access to A will be needed for the above product.

- In Python, the NumPy array normally stores data in the same order as C/C++, although the Fortran ordering is available as an option.

- For matrix-matrix multiplication such as:

  $$C_{ij} = A_{ik}B_{kj} \qquad (k = 1 \dots n)$$

  inevitably a row of A will multiply a column of B, which is likely to lead to memory access issues for larger matrices.

6.1.2 Matrix-vector examples

C/C++
```
// Matrix—vector multiplication
// b_i = a_ij.x_j
#pragma omp parallel for
  {
  for (int i=0; i<n; i++){
    b[i] = 0.0;
    for (int j=0; j<n; j++)
      b[i] += a[i][j] * x[j];
    }
  }
```

Cython
```
# Matrix—vector multiplication
# b_i = a_ij.x_j
import numpy as np
from cython.parallel import prange
cdef int i, j
cdef double[:,:] a = np.zeros((n,n), dtype=np.
    double)
cdef double[:] b = np.zeros(n, dtype=np.double)
cdef double[:] x = np.zeros(n, dtype=np.double)

for i in prange(n, nogil=True):
    b[i] = 0.0
    for j in range(n):
        b[i] += a[i][j] * x[j]
```

- Note the use of the Cython "typed memory view" which is Cython's very efficient way of accessing the elements of a NumPy array.
- In each case, only the outer loop is parallelised.
- No reduction clause is required because each b[i] belongs to only one thread.

6.1.3 Matrix-matrix examples

C/C++
```
// Matrix—matrix multiplication
// c_ij = a_ik.b_kj
#pragma omp parallel for collapse(2)
  {
  for (int i=0; i<n; i++)
    for (int j=0; j<n; j++){
      c[i][j] = 0.0;
      for (int k=0; k<n; k++)
        c[i][j] += a[i][k] * b[k][j];
      }
  }
```

- Both outer loops can be parallelised using `collapse` (not Cython).
- Inner loop cannot be included because it needs a `reduction`.
- In this approach, no `reduction` clause required because each `c[i][j]` belongs to only one thread.

Cython
```
# Matrix—vector multiplication
# c_ij = a_ik.b_kj
import numpy as np
from cython.parallel import prange

cdef int i, j, k
cdef double[:,::1] a = np.zeros((n,n),
            dtype=np.double) # [::1] forces
            # unstrided access for even greater
            # efficiency
cdef double[::1,:] b = np.zeros((n,n),
            dtype=np.double,order='F') # Fortran
            # ordering for more efficient memory
            # access
cdef double[:,::1] c = np.zeros((n,n),
            dtype=np.double)

for i in prange(n, nogil=True):
    for j in range(n):
        for k in range(n):
            c[i][j] += a[i][k]*b[k][j]
```

- Tune parallel loops with `schedule` clause.

6.1.4 Matrix-matrix timings

- Sample timings for multiplying two 1000x1000 matrices:

Basic Python (1 core):	185 s
Best Cython (4 cores):	0.06 s
Python with NumPy (4 cores):	0.02 s

- In fact, ideal scaling for $n \times n$ matrices should be n^3, so larger matrix sizes become prohibitively slow.

- Strassen's divide and conquer algorithm – recursively dividing down into block matrices – reduces this to $n^{2.81}$.

- Winograd's improvement to the method claims scaling of $n^{2.37}$ by re-use of common terms – this is the fastest known method.

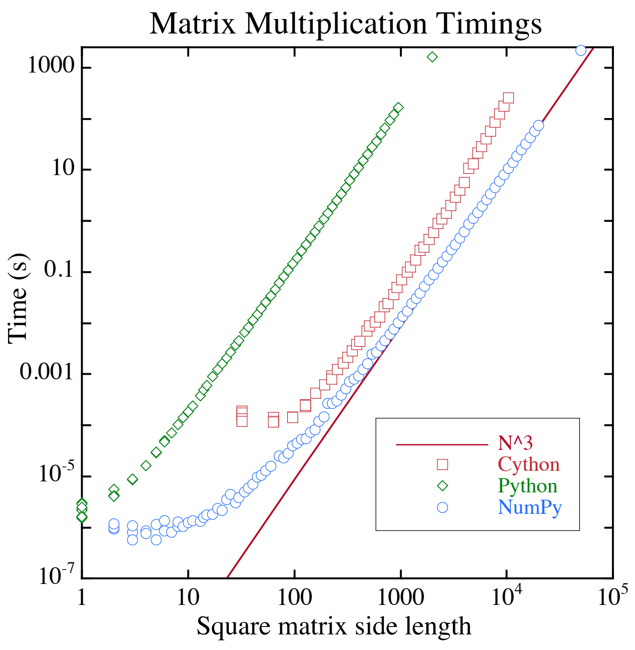
- Non-square matrices can be handled by "padding and peeling" rows or columns.

- In practice any method is extremely sensitive to the arrangement of caches in the cpu.

- For example, here are timings for:

  1. raw Python code,
  2. the best Cython code I could produce (8 threads and using 4-vectors) and
  3. a simple NumPy call.

- Clearly NumPy wins here, although the overall scaling is still $\sim n^3$.



6.2 Solving linear equations

- Typical problem:
  - Solution of the matrix equation $\mathbf{A} \cdot \mathbf{x} = \mathbf{b}$ for an unknown vector $\mathbf{x}$, where $\mathbf{A}$ is a square matrix of coefficients;

- Generally, we do not solve large systems of linear equations using standard matrix inversion formulae

- The evaluation of the standard formula is *factorial* in the size of the matrix e.g. for the determinant of matrix $\mathbf{M}$:

$$|\mathbf{M}| = \sum_i (-1)^i m_{1i} \left|\mathbf{M}^{[1,i]}\right|$$

  where $\mathbf{M}^{[1,i]}$ is the matrix $\mathbf{M}$ with row 1 and column i eliminated.

- Therefore consider Gaussian elimination.

6.2.1 Gaussian elimination

$$\begin{array}{llllll} a_{11}x_1 & +a_{12}x_2 & +a_{13}x_3 & = & b_1 \\ a_{21}x_1 & +a_{22}x_2 & +a_{23}x_3 & = & b_2 \\ a_{31}x_1 & +a_{32}x_2 & +a_{33}x_3 & = & b_3 \end{array}$$

- Multiply 1st equation by a_{21}/a_{11} and subtract from 2nd equation;

- Multiply 1st equation by a_{31}/a_{11} and subtract from 3rd equation:

$$\begin{array}{llllll} a_{11}x_1 & +a_{12}x_2 & +a_{13}x_3 & = & b_1 \\ & a'_{22}x_2 & +a'_{23}x_3 & = & b'_2 \\ & a'_{32}x_2 & +a'_{33}x_3 & = & b'_3 \end{array}$$

- Multiply 2nd equation by a'_{32}/a'_{22} and subtract from 3rd equation:

$$\begin{array}{rrrcl} a_{11}x_1 & +a_{12}x_2 & +a_{13}x_3 & = & b_1 \\ & a'_{22}x_2 & +a'_{23}x_3 & = & b'_2 \\ & & a''_{33}x_3 & = & b''_3 \end{array}$$

- Resulting matrix is upper diagonal.
- Back-substitute from bottom to top to obtain the x_i values.
- The diagonal terms, a_{11}, a'_{22} and a''_{33} are referred to as *pivots*.
- The method fails if a pivot is zero. But, for non-singular equations, you can always swap 2 rows to avoid zero pivots: called *pivoting*.
- Pivoting is good for numerical stability: best to always pivot to place largest remaining non-zero element into the pivot position.

6.2.2 Practical example – no pivots

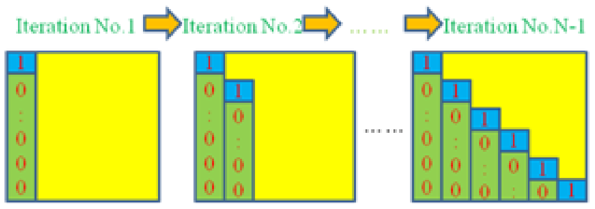
- Assume matrix $\mathbf{A}$ is put into upper diagonal form $\mathbf{A}'$ such that:

$$\begin{bmatrix} a'_{11} & a'_{12} & \cdots & a'_{1k} \\ 0 & a'_{22} & \cdots & a'_{2k} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & a'_{kk} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_k \end{bmatrix} = \begin{bmatrix} b'_1 \\ b'_2 \\ \vdots \\ b'_k \end{bmatrix}$$

- Solutions obtained from *backward substitution*:

$$x_i = \frac{1}{a'_{ii}}\left(b'_i - \sum_{j=i+1}^{k} a'_{ij}x_j\right)$$

- Graphically, first process is (note, scaling diagonal as well):

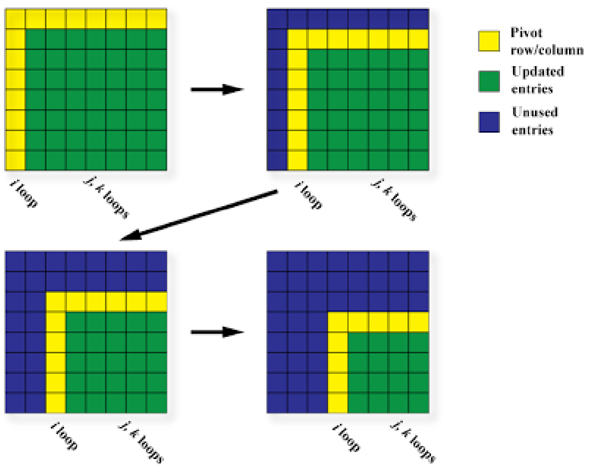


Code for the first part:

```
for (int i = 0; i < N-1; i++) {
    for (int j = i; j < N; j++) {
        double ratio = A[j][i]/A[i][i];
        for (int k = i; k < N; k++) {
            A[j][k] -= (ratio*A[i][k]);
            b[j] -= (ratio*b[i]);
        }
    }
}
```

```
cdef:
    int i, j, k
    double ratio
for i in range(N-1):
    for j in range(i, N):
        ratio = A[j][i]/A[i][i]
        for k in range(i, N):
            A[j][k] -= (ratio*A[i][k])
            b[j] -= (ratio*b[i])
```

- Which loop(s) to parallelise?
  - Loops where number of iterations known upon entry, and does not change.
  - Loops where each iteration independent of all others.
  - Loops that contain no data dependence.
  - Consider this picture:



- ***The i loop*** is shown in yellow. Yellow entries are being used to update the green sub matrix before going on to row/column i+1, meaning the values of the entries in the (i+1)st yellow area depend on operations performed on them at previous i. Therefore can't use OpenMP to parallelize this loop – data dependence.
- ***The j loop:*** iterations vary with i, but we know the number of iterations each time we enter the loop. Later iterations ***do not*** depend on earlier ones and can be computed in any order. The j loop is parallelizable.
- ***The k loop:*** like the j loop, iterations vary but are calculable for each i. Later iterations ***do not*** depend on earlier ones and can be computed in any order. Therefore the k loop is also parallelizable.

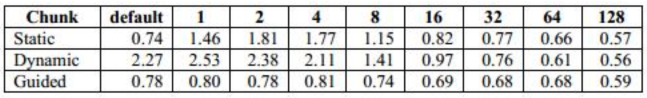
- Suggest selecting the middle loop (j), because the chunks for each thread will be larger:

```
for (int i = 0; i < N-1; i++) {
#pragma omp parallel for
    for (int j = i; j < N; j++) {
        //Parallel j loop
        double ratio = A[j][i]/A[i][i];
        for (int k = i; k < N; k++) {
            A[j][k] -= (ratio*A[i][k]);
            b[j] -= (ratio*b[i]);
        }
    }
}
```

```
cdef:
    int i, j, k
    double ratio
for i in range(N-1):
    for j in prange(i, N, nogil=True):
        # Parallel j loop
        ratio = A[j][i]/A[i][i]
        for k in range(i, N):
            A[j][k] -= (ratio*A[i][k])
            b[j] -= (ratio*b[i])
```

- In practice, division is much more expensive than multiplication, so the factor (1/A[i][i]) could be computed in the outer loop.

- Example timings (seconds) for n = 400 and p = 4 threads:



Chunk	default	1	2	4	8	16	32	64	128
Static	0.74	1.46	1.81	1.77	1.15	0.82	0.77	0.66	0.57
Dynamic	2.27	2.53	2.38	2.11	1.41	0.97	0.76	0.61	0.56
Guided	0.78	0.80	0.78	0.81	0.74	0.69	0.68	0.68	0.59

- N.B. for static scheduling, chunk size defaults to n/p.

Exercise Week 2

Exercise Week 2

- The next stage in your preparation for the Advanced Computational Physics course is to set up a Cython environment if you are intending to use Python as your language, or installing / verifying the availability of the OpenMP library if you are using C/C++. This is described in the latest version of the installation notes and can be prone to unexpected issues, which is why you should troubleshoot this now.

- Compile and run the "hello_world" and "pi_calc" programs given on Blackboard, to test that they work, using either Cython or C/C++.

- Plot a graph of the compute time versus number of threads for the π program running on your own computer, and bring it to the next lecture or email it to Dr Hanna.

- If you haven't already done so, follow the instructions in the guide and repeat the above on BlueCrystal 4 (see the guide for instructions on setting up your account.