HPSC 101 — Lecture 15

Outline:

- OpenMP:
- Nested loops, reductions
- Monte Carlo Pi - exercise

Reading:

- `codes/openmp`

Nested loops

```
!$omp parallel do private(i)
do j=1,m
   do i=1,n
      a(i,j) = 0.d0
   enddo
enddo
```

The loop on `j` is split up between threads.

The thread handling `j=1` does the entire loop on `i`,
sets `a(1,1), a(2,1), ..., a(n,1).`

Nested loops

```
!$omp parallel do private(i)
do j=1,m
   do i=1,n
      a(i,j) = 0.d0
   enddo
enddo
```

The loop on `j` is split up between threads.

The thread handling `j=1` does the entire loop on `i`, sets `a(1,1), a(2,1), ..., a(n,1)`.

Note: The loop iterator `i` must be declared private!

`j` is private by default, `i` is shared by default.

Nested loops

Which is better? (assume $m \approx n$)

```
!$omp parallel do private(i)
do j=1,m
   do i=1,n
      a(i,j) = 0.d0
   enddo
enddo
```

or

```
do j=1,m
   !$omp parallel do
   do i=1,n
      a(i,j) = 0.d0
   enddo
enddo
```

Nested loops

Which is better? (assume $m \approx n$)

```
!$omp parallel do private(i)
do j=1,m
   do i=1,n
      a(i,j) = 0.d0
   enddo
enddo
```

or

```
do j=1,m
   !$omp parallel do
   do i=1,n
      a(i,j) = 0.d0
   enddo
enddo
```

The first has less overhead: Threads created only once.
The second has more overhead: Threads created m times.

Nested loops

But have to make sure loop can be parallelized!

Incorrect code for replicating first column:

```
!$omp parallel do private(j)
do i=2,n
   do j=1,m
      a(i,j) = a(i-1,j)
   enddo
enddo
```

Corrected: (j's can be done in any order, i's cannot)

```
!$omp parallel do private(i)
do j=1,m
   do i=2,n
      a(i,j) = a(i-1,j)
   enddo
enddo
```

Reductions

Incorrect code for computing $\|x\|_1 = \sum_i |x_i|$:

```
norm = 0.d0
!$omp parallel do
do i=1,n
    norm = norm + abs(x(i))
    enddo
```

There is a race condition: each thread is updating same shared variable `norm`.

Correct code:

```
!$omp parallel do reduction(+ : norm)
do i=1,n
    norm = norm + abs(x(i))
    enddo
```

A reduction reduces an array of numbers to a single value.

Reductions

A more complicated way to do this:

```
norm = 0.d0
!$omp parallel private(mysum) shared(norm)
mysum = 0
!$omp do
do i=1,n
    mysum = mysum + abs(x(i))
    enddo

!$omp critical
norm = norm + mysum
!$omp end critical
!$omp end parallel
```

Some other reductions

Can do reductions using $+, -, *$,min, max, .and., .or., some others

General form:

```
!$omp parallel do reduction(operator : list)
```

Example with max:

```
y = -1.d300  ! very negative value
!$omp parallel do reduction(max: y)
do i=1,n
   y = max(y,x(i))
enddo
print *, 'max of x = ',y
```

Some other reductions

General form:

```
!$omp parallel do reduction(operator : list)
```

Example with .or.:

```
logical anyzero

! set x...
anyzero = .false.

!$omp parallel do reduction(.or.: anyzero)
do i=1,n
   anyzero = anyzero .or. (x(i) == 0.d0)
   enddo
print *, 'anyzero = ',anyzero
```

Prints `T` if any `x(i)` is zero, `F` otherwise.

Timing fortran codes

Outline:

- Timing Fortran codes

Codes:

- `codes/fortran/timings.f90`
- `codes/openmp/timings.f90`

Determining CPU and execution time

Unix time command, e.g.

```
$ time ./a.out
<output from code>

real    0m5.279s
user    0m1.915s
sys     0m0.006s
```

Means the elapsed (wall clock) time was 5.279 seconds, CPU time dedicated to your code was ≈ 1.915 seconds. System time ≈ 0.006 seconds.

Determining CPU and execution time

Unix time command, e.g.

```
$ time ./a.out
<output from code>

real    0m5.279s
user    0m1.915s
sys     0m0.006s
```

Means the elapsed (wall clock) time was 5.279 seconds, CPU time dedicated to your code was ≈ 1.915 seconds. System time ≈ 0.006 seconds.

Doesn't allow examining parts of code, not always very accurate.

Note that timing small codes can be deceptive

Fortran timing utilities

`system_clock`: elapsed time between 2 calls.

`cpu_time`: CPU time used between 2 calls.

See timings code

Exercise

1. Implement the Monte Carlo Pi code (as given in the midsem exam) in Fortran
    1. Input data from command line
2. Parallelise the code using OpenMP
3. Strong and weak scaling of the code
4. Test execution on HPC, perform scaling on HPC