OpenMP for Computational Scientists

2: Data sharing and Reductions

Dr Tom Deakin
University of Bristol

Tuesday 1 December, 2020

The first exercise

- Parallelise a serial 5-point stencil code using OpenMP.
- Solution is adding an OpenMP worksharing construct:

```fortran
!$omp parallel do collapse(2)
do i = 1, nx
  do j = 1, ny
    Anew(i,j) = (A(i-1,j) + A(i+1,j) + A(i,j) + A(i,j-1) + A(i,j+1))
     ↪  / 5.0
  end do
end do
!$omp end parallel do
```

- OpenMP threads are created.
- Loops are collapsed and iterations shared between threads.
- Each thread computes its assigned portion of iteration space.
- Threads synchronise and join.

Data sharing

Remember: OpenMP is a *shared memory* programming model.

- By default, all data is available to all threads.
- There is a single copy of *shared* data.

You must specify which data should be *private* to each thread.

- Each thread then has local (stack) space for each private variable.
- Each copy is only visible to its associated thread.

Notice

Fortran variables being declared at the top of the routine mean you must think about this.

Variables on the heap

- All data on the heap is shared.
- Therefore all the Fortran `allocatable` data is shared.
- You must ensure that different threads do not write to the same element of these arrays.

Caution

Setting a data sharing clause on a heap variable only effects the metadata of the variable. The pointer could be private, but the target will still be shared.

Data clauses

- `shared(x)` There is one copy of the `x` variable. The programmer must ensure synchronisation.
- `private(x)` Each thread gets its own local `x` variable. It is not initialised. The value of the original `x` variable is undefined on region exit.
- `firstprivate(x)` Each thread gets its own `x` variable, and it is initialised to the value of the original variable entering the region.
- `lastprivate(x)` Used for loops. Each thread gets its own `x` variable, and on exiting the region the original variable is updated taking the value from the sequentially last iteration.

These are the most common clauses that are needed.

Data clauses

There is also the `threadprivate(x)` directive (not a clause).

- This says to take a copy of the data in *thread local storage* which is persistent across parallel regions.
- The `copyin` directive is a means to initialise `threadprivate` data, copying from the master thread.

Unlikely to use this clause. Might be useful if using `common` blocks (or `static` variables in C).

Private example

Simple do loop, which just sets a variable to the iteration number. Each iteration prints out the current and next value of x, along with the thread number. Will see what happens with different data sharing clauses.

```fortran
!$omp parallel do private(x) / firstprivate(x) / lastprivate(x)
do i = 1, N
  write (*,"(2X,A,I0,A,I0,A,I0)") "Thread ", omp_get_thread_num(), "
  ↪  setting x=", x, " to ", i
  x = i
end do
!$omp end parallel do
```

N is set to 10. Ran using 4 threads. Full implementation: `private.f90`.

Private example

```
private:
 before: x=-1
  Thread 1 setting x=0 to 4
  Thread 2 setting x=0 to 7
  Thread 3 setting x=0 to 9
  Thread 0 setting x=0 to 1
  Thread 1 setting x=4 to 5
  Thread 2 setting x=7 to 8
  Thread 3 setting x=9 to 10
  Thread 0 setting x=1 to 2
  Thread 1 setting x=5 to 6
  Thread 0 setting x=2 to 3
 after: x=-1
```

Each thread starts with its own `x`. No guarantees of initial value, but happened to be zero this time.

Private example

```
firstprivate:
 before: x=-1
  Thread 3 setting x=-1 to 9
  Thread 2 setting x=-1 to 7
  Thread 1 setting x=-1 to 4
  Thread 0 setting x=-1 to 1
  Thread 3 setting x=9 to 10
  Thread 2 setting x=7 to 8
  Thread 1 setting x=4 to 5
  Thread 0 setting x=1 to 2
  Thread 1 setting x=5 to 6
  Thread 0 setting x=2 to 3
 after: x=-1
```

Each thread starts with its own `x`, which set to the value of `x` before entering the `parallel` region, -1.

Private example

```
lastprivate:
 before: x=-1
  Thread 3 setting x=3 to 9
  Thread 2 setting x=2 to 7
  Thread 1 setting x=1 to 4
  Thread 3 setting x=9 to 10
  Thread 0 setting x=0 to 1
  Thread 2 setting x=7 to 8
  Thread 1 setting x=4 to 5
  Thread 0 setting x=1 to 2
  Thread 1 setting x=5 to 6
  Thread 0 setting x=2 to 3
 after: x=10
```

Each thread starts with its own `x`, which set to to a garbage value. On exiting the region, the original `x` is set to the value of the last iteration of the loop, 10.

Choosing default data sharing

Note

It is especially important to list private variables in Fortran. All variables have *global* scope within each `subroutine` so *everything* is shared by default. In C, local scoping rules makes this easier.

- You can force yourself to specify everything manually by using the `default(none)` attribute. This is good practice.
- You can also `default(private)` or `default(firstprivate)` to make everything private by default — this might save a lot of typing in an old code with many temporary variables.

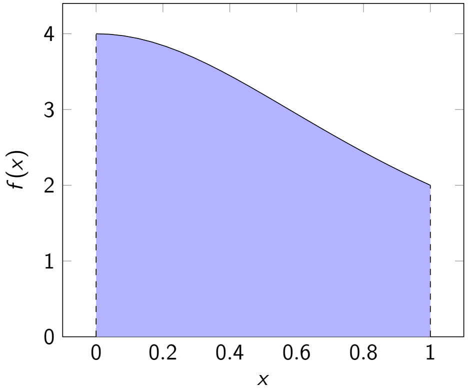
Calculating Pi

Use a simple program to numerically approximate π to explore:

- Use of data sharing clauses.
- Updating a shared variable in parallel.
- Reductions.

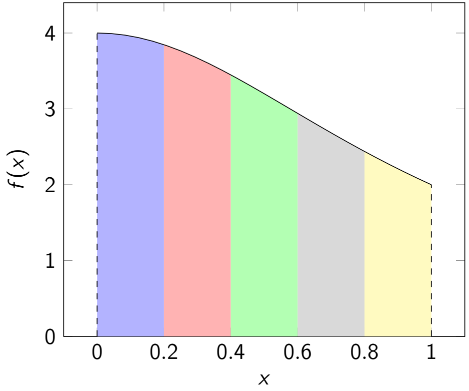
Integration to calculate Pi

$$\int_0^1 \frac{4}{1+x^2}\,dx = \pi$$



Trapezoidal rule

Sum the area of the boxes. Choose a small *step* size to generate lots of boxes, and increase accuracy.



Code

We will use this code which calculates the value of π as an example for the remainder of this session.

```
step = 1.0/num_steps
do ii = 1, num_steps
  x = (ii-0.5)*step
  sum = sum + (4.0/(1.0+x*x))
end do
pi = step * sum
```

With 100,000,000 steps, this takes 0.368s on my laptop.
Full implementation: `pi.f90`.

Parallelising the loop

Use a worksharing directive to parallelise the loop.

```
step = 1.0/num_steps
!$omp parallel do private(x)
do ii = 1, num_steps
  x = (ii-0.5)*step
  sum = sum + (4.0/(1.0+x*x))
end do
!$omp end parallel do
pi = step * sum
```

What about data sharing?

- `x` needs to be used independently by each thread, so mark as `private`.
- `sum` needs to be updated by *all* threads, so leave as `shared`.

Parallelising with critical

- But need to be careful changing the shared variable, sum.
- All threads can update this value directly!
- A critical region only allows one thread to execute at any one time. No guarantees of ordering.

```
step = 1.0/num_steps
!$omp parallel do private(x)
do ii = 1, num_steps
  x = (ii-0.5)*step
  !$omp critical
  sum = sum + (4.0/(1.0+x*x))
  !$omp end critical
end do
!$omp end parallel do
pi = step * sum
```

Runtimes

Run on a MacBook Pro (Intel Core i7-4980HQ CPU @ 2.80GHz) with 4 threads.

Implementation	Runtime (s)
Serial	0.368
Critical	426.1

Full implementation: `pi_critical.f90`.

Really slow!

Atomics

A `critical` region protects a whole block of code. For a single operation, can use `atomic` instead.

Atomic operations are with respect to the memory access of a scalar variable `x`.

- `read` for `v = x`
- `write` for `x = expr`
- `update` for `x = x op expr`
- `capture` for read and write/update. The result is retained:
  `x = x op expr; v = x`

Not specifying an atomic clause defaults to `update`.

Atomic pi

```fortran
step = 1.0/num_steps
!$omp parallel do private(x)
do ii = 1, num_steps
  x = (ii-0.5)*step
  !$omp atomic
  sum = sum + (4.0/(1.0+x*x))
end do
!$omp end parallel do
pi = step * sum
```

Runtimes

Run on a MacBook Pro (Intel Core i7-4980HQ CPU @ 2.80GHz) with 4 threads.

Implementation	Runtime (s)
Serial	0.368
Critical	426.1
Atomic	8.3

Full implementation: `pi_atomic.f90`.

Faster, but still slower than serial.

Independent summation

- Both methods cause threads to synchronise for every update to `sum`.
- But each thread could compute a partial sum independently, synchronising once to total at the end.

Make `sum` an array of length equal to the number of threads.

- Each thread stores its partial sum, and the array is totalled by the master thread serially at the end.
- As it's *shared memory*, the `sum` array can be read just fine on the master rank.

Independent summation

```
step = 1.0/num_steps
!$omp parallel private(x,tid)
tid = omp_get_thread_num()
sum(tid+1) = 0.0
!$omp do
do ii = 1, num_steps
  x = (ii-0.5)*step
  sum(tid+1) = sum(tid+1) + (4.0/(1.0+x*x))
  !$omp flush(sum)
end do
!$omp end do
!$omp end parallel
do ii = 1, nthreads
  pi = pi + sum(ii)
end do
pi = pi * step
```

Runtimes

Run on a MacBook Pro (Intel Core i7-4980HQ CPU @ 2.80GHz) with 4 threads.

Implementation	Runtime (s)
Serial	0.368
Critical	426.1
Atomic	8.3
Array	2.8

Full implementation: `pi_array.f90`.

Fastest parallel version so far, but still slow.

False sharing

This code is susceptible to *false sharing*.

- False sharing occurs when different threads update data on the same cache line.
- Cache system is coherent between cores, so data consistency must be maintained.
- The cache line is no longer up to date because another thread changed it (in their local cache).
- Therefore, cache line must be flushed to memory and reread into the other thread every time.
- This is an example of *cache thrashing*.
- The performance is reduced as threads must wait for the cache lines to refresh.

Flush

- The `flush()` construct ensures that the variables are consistent between the thread's memory and main memory.
- Don't want to go into complicated parts of the OpenMP memory model.
- In general, don't need to worry about this stuff.
- Without the flush, the write to memory will be lowered to after the loop, so false sharing only occurs once at the end.
- Here we use it to *ensure* that false sharing occurs every time to highlight the performance hit.

Firstprivate pi

Can use data sharing clauses to our advantage here:
Give each thread a *scalar* copy of sum to compute their partial sum, and reduce with only one critical (or atomic) region at the end. No false sharing, as value is just a single number (i.e. a register).

```
step = 1.0/num_steps
!$omp parallel private(x) firstprivate(sum)
!$omp do
do ii = 1, num_steps
  x = (ii-0.5)*step
  sum = sum + (4.0/(1.0+x*x))
end do
!$omp end do
!$omp critical
pi = pi + sum
!$omp end critical
!$omp end parallel
pi = pi * step
```

Runtimes

Run on a MacBook Pro (Intel Core i7-4980HQ CPU @ 2.80GHz) with 4 threads.

Implementation	Runtime (s)
Serial	0.368
Critical	426.1
Atomic	8.3
Array	2.8
First private	0.104

Full implementation: `pi_private.f90`.

Finally faster than serial! Around 3.5X faster on 4 threads.

Reductions

Much simpler to use the OpenMP `reduction` clause on a worksharing loop. Specify the operation and the variable.

- `reduction(+:var)`
- `reduction(-:var)`
- `reduction(*:var)`
- `reduction(.and.:var)`
- `reduction(.or.:var)`
- `reduction(.eqv.:var)`
- `reduction(.neqv.:var)`
- `reduction(.max.:var)`
- `reduction(.min.:var)`
- `reduction(.iand.:var)`
- `reduction(.ior.:var)`
- `reduction(.ieor.:var)`

Can also do array reductions. Each element of array is treated as own, separate, reduction. Similar to:

```
MPI_Allreduce(MPI_IN_PLACE, arr, N, MPI_DOUBLE, MPI_SUM, 0,
↪  MPI_COMM_WORLD, ierr)
```

Pi reduction

Much simpler to write using the `reduction` clause — just need a single directive:

```
step = 1.0/num_steps
!$omp parallel do private(x) reduction(+:sum)
do ii = 1, num_steps
  x = (ii-0.5)*step
  sum = sum + (4.0/(1.0+x*x))
end do
!$omp end parallel do
pi = step * sum
```

Full implementation: `pi_reduction.f90`.

Runtimes

Run on a MacBook Pro (Intel Core i7-4980HQ CPU @ 2.80GHz) with 4 threads.

Implementation	Runtime (s)
Serial	0.368
Critical	426.1
Atomic	8.3
Array	2.8
First private	0.104
Reduction	0.095

Around 3.9X faster on 4 threads!

Recommendation

Use the `reduction` clause for reductions.

Exercise

- Start with your parallel 5-point stencil code from last time.
- Change the code to print out the total of the cells (excluding halo) every timestep.
- You'll need to implement a parallel reduction to do this.
- Try the different techniques shown to implement reductions:
  - Critical sections.
  - Atomics.
  - Reduction clause.
- Extension: there is also a Jacobi code to parallelise — it needs a reduction too.

Summary

- Have now covered the most common parts of OpenMP.
- 80/20 rule: Most programs will only use what you know so far.
- OpenMP is deceptively simple!
- In the remaining sessions you'll learn to program OpenMP on NUMA and GPU architectures.