A brief introduction to OpenMP

2: Data sharing clauses

University of BRISTOL

Outline

- Recap
- Data sharing clauses
- The Pi program
- Critical regions
- False sharing issues
- Reductions

Recap

- Fork/join execution model.
- Shared memory model:
  - All threads can read/write the *same* memory.
- Set number of threads with `OMP_NUM_THREADS` environment variable.
- Parallelise simple loops with worksharing clauses:

```
#pragma omp parallel for
for (int i = 0; i < N; ++i) {
  A[i] = ...;
}
```

- Talked about `collapse`, `nowait` and `schedule` clauses.

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

Variables declared *inside* the parallel region will be private to each thread.

Variables on the heap

- All data on the heap is shared.
- Therefore all the data allocated with `malloc` 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.

Private example

Simple for 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.

```
int x = -1;
#pragma omp parallel for private(x) / firstprivate(x) /
↪  lastprivate(x)
for (int i = 0; i < N; ++i) {
  printf("Thread %d setting x=%d to %d\n",
  ↪  omp_get_thread_num(), x, i);
  x = i;
}
```

N is set to 10. Ran using 4 threads.

Private example

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

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, 9.

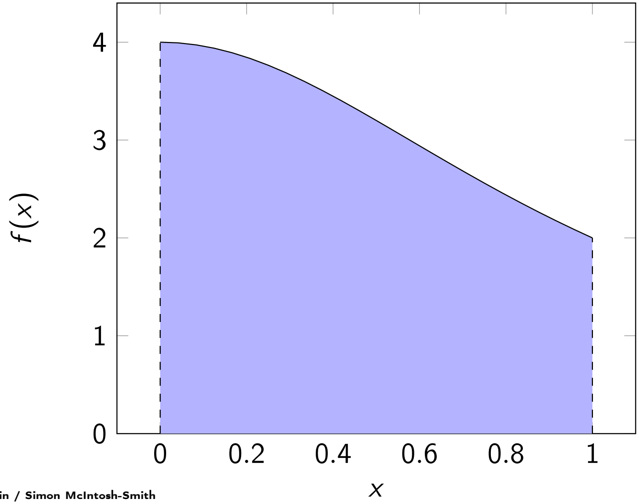
Calculating Pi

Use a simple program to numerically approximate π to explore:

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

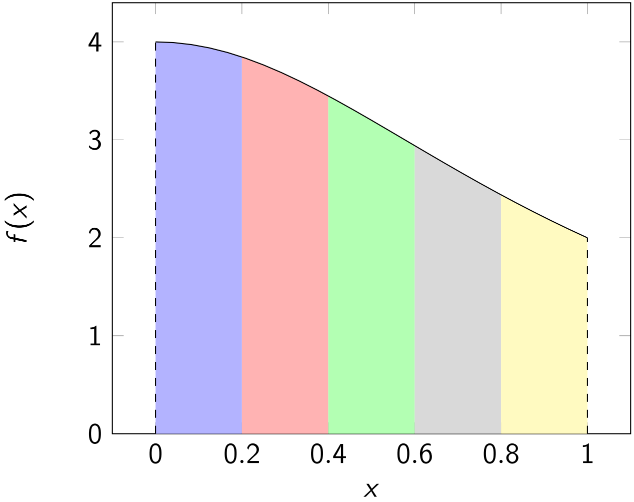
Integration to calculate Pi

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



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.

```
double step, x, sum, pi;
step = 1.0/num_steps;
for (int ii = 1; ii <= num_steps; ++ii) {
  x = (ii-0.5)*step;
  sum = sum + (4.0/(1.0+x*x));
}
pi = step * sum;
```

With 100,000,000 steps, this takes 0.368s on my laptop.

Parallelising the loop

Use a worksharing directive to parallelise the loop.

```
double step, x, sum, pi;
step = 1.0/num_steps;
#pragma omp parallel for private(x)
for (int ii = 1; ii <= num_steps; ++ii) {
  x = (ii-0.5)*step;
  sum = sum + (4.0/(1.0+x*x));
}
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.

```c
double step, x, sum, pi;
step = 1.0/num_steps;
#pragma omp parallel for private(x)
for (int ii = 1; ii <= num_steps; ++ii) {
  x = (ii-0.5)*step;
  #pragma omp critical
  {
  sum = sum + (4.0/(1.0+x*x));
  }
}
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

Slower than serial!

Independent summation

- Criticals or atomics 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;
#pragma omp parallel private(x,tid)
{
tid = omp_get_thread_num();
sum[tid] = 0.0;
#pragma omp for
for (int ii = 1; ii <= num_steps; ++ii) {
  x = (ii-0.5)*step;
  sum[tid] = sum[tid] + (4.0/(1.0+x*x));
}
}
for (int ii = 0; ii < nthreads; ++ii) {
  pi = pi + sum[ii];
}
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*

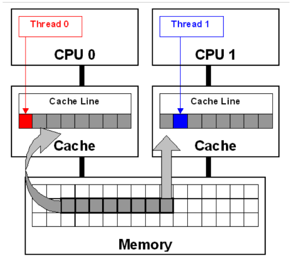
* Might be faster if have a smart compiler which avoided false sharing.

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.
- It is a *different* phenomenon to cache thrashing, as result of parallel shared memory execution.
- 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.



https://software.intel.com/en-us/articles/avoiding-and-identifying-false-sharing-among-threads

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;
#pragma omp parallel private(x) firstprivate(sum)
{
#pragma omp for
for (int ii = 1; ii <= num_steps; ++i) {
  x = (ii-0.5)*step;
  sum = sum + (4.0/(1.0+x*x));
}
#pragma omp critical
pi = pi + sum;
}
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

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(&&:var)`
- `reduction(||:var)`
- `reduction(^:var)`
- `reduction(&:var)`
- `reduction(|:var)`
- `reduction(min:var)`
- `reduction(max: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);
```

Pi reduction

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

```
step = 1.0/num_steps;
#pragma omp parallel for private(x)
↪  reduction(+:sum)
for (int ii = 1; ii <= num_steps; ++i) {
  x = (ii-0.5)*step;
  sum = sum + (4.0/(1.0+x*x));
}
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
Array	2.8
First private	0.104
Reduction	0.095

Around 3.9X faster on 4 threads!

Recommendation

Use the `reduction` clause for reductions.

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!