Lecture 3: More on OpenMP with application to a Monte Carlo solver

Mike Giles

Mathematical Institute

False sharing

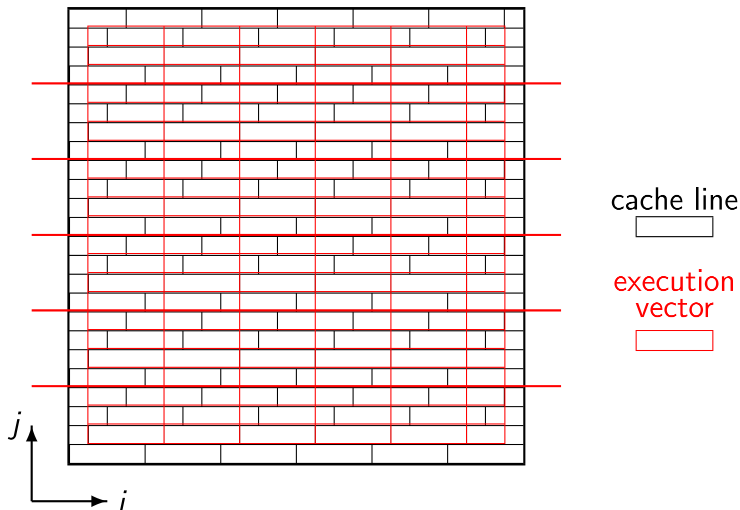
In the first lecture, we discussed cache lines and the problem of **false sharing** in which different threads try to update different parts of the same cache line.

In Practical 1 you saw the potential consequences of this for OpenMP performance.

In real applications this is probably my most common concern, and it's hard to diagnose. This is why I check to see whether I am getting a good fraction of peak performance; if I'm not, then I look to see if I might have a false sharing problem.

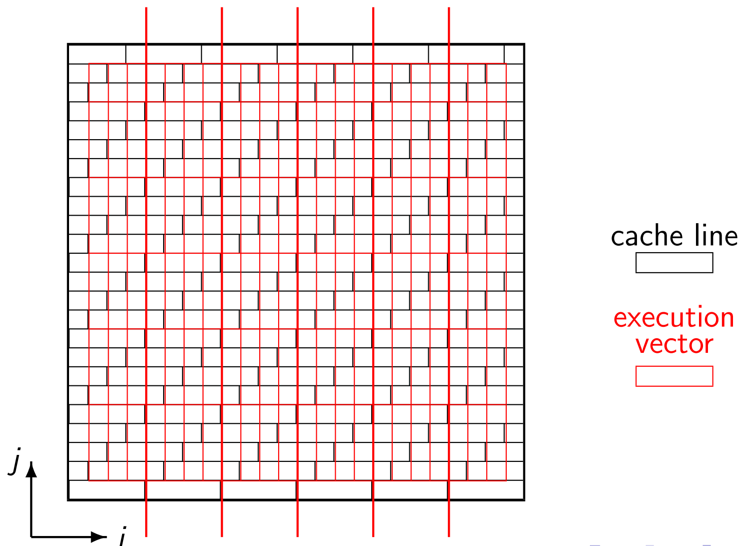
False sharing

Optimal approach – parallel outer j loop, vector inner i loop



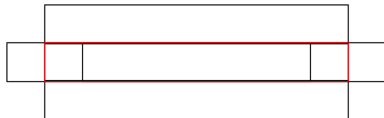
False sharing

Poor approach – parallel outer i loop, vector inner j loop



Data movement

Going back to the optimal approach, for each vector the core loads 5 vectors of data from the L1 cache into vector registers, and writes 1.



This is 6 variables for each interior grid point, so in total it is approximately

$$6 \times m^2 \times 8 \text{ bytes} \equiv \texttt{bytes1}$$

Data movement

The data needed for 1 row of the calculation is 3 rows of `v1` plus 1 row of `v2`

$$4 \times m \times 8\,\text{bytes} \approx 13\text{kB}$$

This sits comfortably within L1 cache, so when going back to calculate the next row the only new information that needs to be loaded in is 1 row of `v1` plus 1 row of `v2`

$$2 \times m \times 8\,\text{bytes}$$

Summing over all of the calculation rows, the total amount of data being moved from L2 cache into L1 cache is

$$2 \times m^2 \times 8\,\text{bytes} \equiv \texttt{bytes2}$$

Data movement

The total amount of data is

$$2 \times m^2 \times 8\,\text{bytes} \approx 2.5\text{MB}$$

This will easily sit inside the L3 cache, so there will be no data transfer between the L3 cache and the main external DDR memory.

Dividing by the number of cores, this may be small enough that the data being used by each thread can live within its private L2 cache for that thread, so the L3 does not get used either.

Practical 1

Parallelising the inner loop is much worse.

Starting and stopping a large team of threads is expensive, so generally best to parallelise outer loop **unless** that doesn't offer enough parallelism.

Following on from this brings us to ideas of parallel sections and work sharing.

Parallel sections

The OpenMP construct

```
#pragma omp parallel for ...
```

is really a concatenation of two different constructs:

```
#pragma omp parallel shared(...) private (...)
{


}
```

which defines a **parallel section** of code which is to be executed by a team of threads, and

```
#pragma omp for
```

which is a work sharing directive to say that different threads in the team should do different parts of the loop

Parallel sections

Some experts argue strongly that it is bad programming practice to use the concatenated form, but I like its simplicity and find it very useful for lots of applications.

However, there are some applications for which the parallel section approach is ideal.

Note that if there are no other pragmas in the parallel section then all threads will execute the same code identically.

The RTL function `omp_get_thread_num` can be used to get a thread ID (0 up to #threads−1), and this can be used to change what each thread does.

Work sharing

By default,

```
#pragma omp for
```

splits the execution of the subsequent loop into chunks of equal size (or as close as possible) for the different threads to execute.

This can be changed by using a `schedule` "clause"

- `schedule(static)` – the default
- `schedule(static,chunk_size)` – similar but assigns the specified chunk sizes round-robin to each thread (useful sometimes when work per loop element varies)
- `schedule(dynamic,chunk_size)` – similar again, but new chunks are assigned when previous ones are completed (again useful for variable work, but loses cache reuse between loops)
- see documentation for other alternatives (guided, auto, runtime)

Critical section

Within a parallel section, a critical sub-section is a piece of code which must be executed one thread at a time.

```
#pragma omp critical
{


}
```

I've used it before for array reductions, but now that those are supported in version 4.5 it's better to use the OpenMP reduction.

But there may be other times when it's useful to avoid conflicts when different threads are updating the same variables in memory.

There is also a similar `#pragma omp atomic` capability for single instructions (e.g. to increment a shared counter)

SIMD parallelism

In Practical 1, the compiler automatically vectorised the innermost loop.

Sometimes it is necessary to force (or at least strongly encourage) the compiler to use vectorisation by using the pragma

```
#pragma omp simd
```

which has an optional clause `simdlen(length)`

(SIMD = Single Instruction Multiple Data)

Global scope variables

Global variables are defined outside the main program or any function, and have a global scope – means they can be referenced anywhere.

I often use this for constants, set once and then used throughout the application. I almost never use it for variables which are repeatedly changed – I pass these through function argument lists.

In OpenMP, by default such variables are shared variables.

There is a special `threadprivate` capability to declare them as private, creating separate copies for each thread – very rarely needed but very useful when it is.

OpenMP RTL

The full list of functions is available here:

https://software.intel.com/en-us/cpp-compiler-18.0-developer-guide-and-reference-openmp-run-time-library-routines

There are about 30, but I have already mentioned all of those that I have used myself.

Environment variables

The full list of environment variables is available here, as part of a bigger list of environment variables recognised at run-time:

https://software.intel.com/en-us/cpp-compiler-18.0-developer-guide-and-reference-supported-environment-variables

This includes the following:

- `OMP_DYNAMIC` which controls whether the number of threads can be adjusted dynamically (default: FALSE)
- `OMP_NESTED` which controls whether nested parallelism is allowed (default: FALSE)
- `OMP_PLACES` which controls the placement of threads on cores

Practical 2

Practical 2 is a Monte Carlo solver in which we want to compute

$$\sum_n X_n, \quad \sum_n X_n^2$$

where the X_n are each generated using different random numbers.

High-level view:

- trivially parallel, just needs final top-level reduction
- each threads needs its own random number generator
- should use Intel MKL library to generate random numbers efficiently in chunks
- chunks should be big enough for good compute performance, but small enough to stay in L2 cache for good bandwidth

Practical 2

```
/* each OpenMP thread has its own VSL RNG and storage */

#define NRV 16384  // number of random variables
VSLStreamStatePtr stream;
float *uniforms,      *normals;
int    uniforms_count, normals_count;
#pragma omp threadprivate(stream, uniforms,uniforms_count
                                  normals, normals_count)
```

Here we use `threadprivate` to define separate generators for each thread. `NRV` will be the size of arrays `uniforms` and `normals`, chosen to hold the random numbers within each L2 cache.

Practical 2

```
//
// RNG routines
//

void rng_initialisation(){
  int tid = omp_get_thread_num();
  vslNewStream(&stream, VSL_BRNG_MRG32K3A,1337);
  long long skip = ((long long) (tid+1)) << 48;
  vslSkipAheadStream(stream,skip);
  uniforms     = (float *)malloc(NRV*sizeof(float));
  normals      = (float *)malloc(NRV*sizeof(float));
  uniforms_count      = 0;  // means no random numbers
  normals_count       = 0;  // in the arrays currently
}
```

Practical 2

```
void rng_termination(){
  vslDeleteStream(&stream);
  free(uniforms);
  free(normals);
}

inline float next_normal(){
  if (normals_count==0) {
    vsRngGaussian(VSL_RNG_METHOD_GAUSSIAN_BOXMULLER2,
                  stream,NRV,normals,0.0f,1.0f);
    normals_count = NRV;
  }
  return normals[--normals_count];
}
```

Practical 2

```
int main(int argc, char **argv)
{
  float  T=1.0f, X0=1.0f, mu=0.05f, sigma=0.2f, dt;
  double sum1=0.0, sum2=0.0;
  int    M = 200;       /* number of timesteps */
  int    N = 9600000;   /* total number of MC samples */

  dt  = T / ((float) M);

// initialise generator, with separate storage for each
// thread when compiled for OpenMP
#pragma omp parallel
    rng_initialisation();
```

Practical 2

```
#pragma omp parallel for default(none) \
                         shared(T,X0,mu,sigma,dt,M,N) \
                         reduction(+:sum1,sum2)
  for (int n=0; n<N; n++) {
    float X = X0;

    for (int m=0; m<M; m++) {
      float delW = sqrtf(dt)*next_normal();
      X = X + X*(mu*dt + sigma*delW);
    }

    sum1 += X;
    sum2 += X*X;
  }
```

Final comments

- this lecture covered more advanced / exotic features – don't feel that you need to understand and use all of them
- concentrate on the fundamentals, and then expand into using the more advanced ones purely as needed
- remember to check on the web for examples, or talk to others with more experience
- always monitor your performance to assess whether you are doing a good job