What is OpenMP?

- OpenMP is an API for developing shared memory application
- Cross platform — supported by most vendors
- C, C++ and Fortran
- Based on thread model of programming
- Formalized in 1997

What is OpenMP?

- Compiler directives to generate parallel code
- Runtime library that implements the functionality
- Environment variables control parallel execution at runtime

Compiler Directives

- `#pragma omp directive [clause] [clause] ...`
- The directive is applied to the next statement — often a new block
- The directive can be split over more than one line if required by using a backslash

Hello world

```c
#include <omp.h>
#include <stdio.h>

int main() {

#pragma omp parallel
  {
    int id;
    id = omp_get_thread_num();
    printf("%d: Hello world\n", id);
  }

  return 0;
}
```

- `gcc -fopenmp hello_omp.c`

A note on compilers

- Most modern compilers support OpenMP
- GNU — `gcc -fopenmp foo.c`
- Intel — `icc -openmp foo.c`
- Pathscale — `pathcc -openmp foo.c`
- Potential issue with gcc built using homebrew on Mac

Creating Threads

- By default OpenMP creates a number of threads equal to the number of processors in the system
- There are several ways to modify this
  - Library call — `omp_set_num_threads(3);`
  - Directive — `#pragma omp parallel num_threads(3)`
  - Environment variable — `% export OMP_NUM_THREADS=3`
- Can use `omp_get_num_procs()` to find out how many processors are available for you to use

Loop Parallelization

- Often the evaluation of statements in a loop are independant of each other
- The order of execution is not important
- These can be easily parallelized in OpenMP using the `for` pragma

```
#pragma omp parallel for
for(i=0;i<N;i++)
    a[i] = b[i]*c[i];
```

- The for loop conditions must be relatively normal
- No `break`, `return`, `exit`
- Each thread executes a section of the loop index range

Loop Parallelization

- What if we have a double loop?

```
#pragma omp parallel for
for(i=0;i<N;i++) {
    for(j=0;j<N;j++) {
        a[i][j] = b[i][j] * c[i][j];
    }
}
```

- Each thread will execute their own section of the `i` loop
- Since `j` is shared things will get very confused
- Could move declaration of `j` inside the `i` loop
- Could just do the omp on the inner loop — inefficient

Per Thread Variables

- Unless otherwise specified, all variables declared outside the block are shared
- One exception is the `for` loop index variable
- Variables declared inside the block are unique for that thread
- Make variable private to the thread by modifying the construct
- `#pragma omp parallel private(var)`
- This makes a new copy of the variable `var` for each thread
- The value of the new `var` is undefined

Loop Parallelization

- Now our double loop works fine

```
#pragma omp parallel for private(j)
for(i=0;i<N;i++) {
    for(j=0;j<N;j++) {
        a[i][j] = b[i][j] * c[i][j];
    }
}
```

Per Thread Variables

- `#pragma omp parallel firstprivate(var)`
- This will create a new variable for each thread but initialize to the value of the already declared variable `var`
- Using `private` and `firstprivate`, at the end of the block the old value of `var` is unchanged

```
var = 101;
#pragma omp parallel for firstprivate(var)
for(i=0;i<N;i++) {
    printf("var = %d\n", var);
    var  = i;
}
printf("var = %d\n", var); /* This will print 101 */
```

Per Thread Variables

- Sometimes we want to be able to use the value of variables after the loop has completed
- Adding a `lastprivate` clause to the construct will set the value of the variable after the parallel region to the value it would have, if the loop had been executed serially

```
var = 4;
printf("%d\n", var);

#pragma omp parallel for lastprivate(var)
for(i=0;i<N;i++) {
    var = i;
}

printf("%d\n", var); /* Will print value N-1 */
```

Accessing Shared Variables

- As with pthreads, need a way to control updates to shared variables
- Can introduce race conditions that give the wrong answer

```
max = INT_MIN;
#pragma omp parallel for
for(i=0;i<N;i++) {
    if(a[i] > max)
        max = a[i];
}
```

- Without OpenMP, this loop finds the maximum value in an array
- With multiple threads, a max value may be skipped due to order of read/write instructions in different threads

Critical Sections

- Can fix this using a `critical` directive
- Only one thread can be executing in a `critical` section at a time
- This version, while correct is terribly inefficient
- We have pretty much serialized the code again

```
max = INT_MIN;
#pragma omp parallel for
for(i=0;i<N;i++) {
#pragma omp critical
     {
     if(a[i] > max)
          max = a[i];
     }
}
```

Critical Sections

- We can't just move the `critical` directive to after the test
- The value of `max` increases monotonically
- Most loop trips will not update max

```
max = INT_MIN;
#pragma omp parallel for
for(i=0;i<N;i++) {
    if(a[i] > max) {
#pragma omp critical
        {
        if(a[i] > max)
            max = a[i];
        }
    }
}
```

Reduction Clauses

- Often we want to carry out a reduction on a variable in a loop
- OpenMP allows us to add this to the directive
- Sum, product, bitwise and logical operations

critical

```c
total = 0;
#pragma omp parallel for
for(i=0;i<N;i++) {
    a[i] = some_func();
#pragma omp critical
    total += a[i];
}
```

reduction

```c
total = 0;
#pragma omp parallel for reduction(+:total)
for(i=0;i<N;i++) {
    a[i] = some_func();
    total += a[i];
}
```

Conditional Loop Parallelization

- There is an overhead in creating and joining threads
- If the trip count of a loop is too small it may just be quicker to execute in serial
- We can add a condition to the directive to decide at runtime whether or not to parallelize the loop
- Need to do some experiments to determine what the threshold should be

```
total = 0;
#pragma omp parallel for reduction(+:total) if(N > 200)
for(i=0;i<N;i++) {
    a[i] = some_func();
    total += a[i];
}
```

Work Scheduling

- With the default `for` division of work between the threads we can have imbalances
- For example - initialising an upper triangular matrix
- Thread 0 will have $N + N - 1 + N - 2 \ldots$ calls to `func` whereas the last thread will have $\ldots 3 + 2 + 1$ calls to `func`

```
#pragma omp parallel for private(j)
for(i=0;i<N;i++)
    for(j=i;j<N;j++)
        a[i][j] = func(i, j);
```

Work Scheduling

- We can have three types of schedule in OpenMP
  - static — all iterations are assigned to a thread before the start execution
  - dynamic — threads execute some iterations and are then assigned more until all work is complete
  - guided — like dynamic but the size of work packages changes over time heuristically
- We also define a `chunk` to be a range of contiguous iterations of a loop
- Now we can add the clause `schedule(type, chunk)` to the OpenMP directive. The chunk is optional

Work Scheduling

- schedule(static) — every thread gets about n/t iterations
- schedule(static, N) — every thread gets given an interleaved allocation of sets of N iterations
- schedule(dynamic) — threads are given iterations one at a time
- schedule(dynamic, N) — threads are given iterations N at a time
- schedule(guided) — the system uses a heuristic function to allocate threads down to chunks of size one
- schedule(guided, N) — as guided but to minimum chunk size of N
- schedule(runtime) — loooks at the environment variable `OMP_SCHEDULE` to determine what to use

Backward Compatability

- Even though most compilers now support OpenMP you may for some reason want to compile and run your code without OpenMP
- Recall there are two ways your code has been modified
  - Directives
  - Library Calls
- If the compiler reaches a directive it doesn't understand it silently ignores it
- However you will need to work around any library calls

```
#ifdef _OPENMP
    nthreads = omp_get_num_procs();
#else
    nthreads = 1;
#endif
```

Programming Models

- Pure MPI — one MPI task on each core
- Pure OpenMP — limited to 1 machine or use distributed shared memory
- Hybrid — MPI for internode, OpenMP within the node
  - No overlap of comms — MPI outside OpenMP regions
  - Overlap of comms — MPI within OpenMP regions
- Code may not be portable. Different MPI implementations support threads in different ways

Why mix MPI and OpenMP

- Modern compute nodes have many cores — 8 to 24 being common
- Still tend to only have a single network connection
- With 24 MPI tasks per node there will be lots of contention for the available network resources
- Might be better to reduce the amount of comms outside the node — 2 MPI tasks with 12 threads or 4 MPI tasks with 6 threads
- Avoids unnecessary overhead doing MPI within the node

Support for Hybrid programming

- Your MPI implementation may have limited or no support for hybrid programming
- There are four levels of thread support
  - MPI_THREAD_SINGLE — only one thread
  - MPI_THREAD_FUNNELED — only the main thread can issue MPI calls
  - MPI_THREAD_SERIALIZED — all threads can issue MPI calls but only one thread at a time
  - MPI_THREAD_MULTIPLE — all threads can issue MPI calls at any time
- Each is represented by an integer and they are strictly ordered. The higher the value the more support there is
- Replace `MPI_Init` with `MPI_Init_thread` to see what your MPI supports

Determining your level of support

```
requested = MPI_THREAD_MULTIPLE;
MPI_Init_thread(&argc, &argv, requested, &provided);

switch(provided) {
    case MPI_THREAD_SINGLE:
        printf("Support for MPI_THREAD_SINGLE\n");
        break;
    case MPI_THREAD_FUNNELED:
        printf("Support for MPI_THREAD_FUNNELED\n");
        break;
    case MPI_THREAD_SERIALIZED:
        printf("Support for MPI_THREAD_SERIALIZED\n");
        break;
    case MPI_THREAD_MULTIPLE:
        printf("Support for MPI_THREAD_MULTIPLE\n");
        break;
}
```

Mixing MPI and OpenMP

- In order to mix MPI and OpenMP we need at least MPI_THREAD_FUNNELED support
- We build our MPI application and then use OpenMP directives to parallelize the compute parts
- MPI calls are only used in the main MPI thread
- We can use the `MPI_Is_thread_main(int *flag)` to check if we are the thread that is allowed to make MPI calls

Other thread libraries

- We can use other thread libraries such as `pthreads` instead of OpenMP to spawn threads
- More flexible but introduces the usual problem of controlling access to shared data
- Best if MPI configured to provied MPI_THREAD_MULTIPLE support
- Note MPI Send/Recv can address only a process, not a thread within the process
- Can use tags as an additional identifier
- Be very careful when mixing MPI, threads and non-blocking calls
- Data will probably not end up where you expect it to!!