Distributed and Paralel Tech: OpenMP Lab

The goal of this lab is to familiarise yourself with OpenMP, one of the most widely used shared-memory parallel programming models based on "semi-automatic parallelism". In the Lecture we saw the OpenMP directives, runtime functions and environment variables. These exercises will give you experience of writing and controlling OpenMP programs on a multicore.

The exercises use examples from the Lawrence Livermore National Laboratory (LLNL) OpenMP Tutorial. Many high-performance computing sites produce OpenMP tutorials (as it is an "easy" way to increase core utilising for scientists utilising their machines), so if you want more examples this is a great place to start!

Of course, make sure you are using a multicore machine for this lab. You can discover how many cores there are on a host with cat /proc/cpuinfo.

The first half of the lab is mainly try-it-out style exercises (worth doing ahead of the lab if you can), while the second half has some programming exercises.

Setup and Hello World example

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

int main (int argc, char* argv[]) {
    // Fork a team of threads
#pragma omp parallel
    {
        // Print id
        int tid = omp_get_thread_num();
        printf("Hello World from thread = %d\n", tid);

        // Only master thread does this
        if (tid == 0) {
            int nthreads = omp_get_num_threads();
            printf("Number of threads = %d\n", nthreads);
        }
    }  // All threads join master thread and terminate
}
```

1. Create `hello_omp.c` either from above, or using the version in the Lab files folder.
2. Review the source code and note how OpenMP directives and library routines are being used.
3. Compile `hello_omp.c` for parallel execution, with optimisation (-O2):

```
gcc -fopenmp -o hello_omp -O2 hello_omp.c.c
```

A Makefile is also available in the Lab Files.

4. Run two or more times with the default number of threads:

```
./hello_omp
```

5. Set the number of threads to 6 and rerun two or more times:

```
OMP_NUM_THREADS=6 ./hello_omp
# Alternatively
export OMP_NUM_THREADS=6
./hello_omp
```

6. What do you deduce about OpenMP thread scheduling from the results?
7. What do you notice about when the number of threads is printed? What if this is outside the openMP block?

Parallel For Example

The parallel for is probably the most widely used OpenMP directive. It introduces ***data parallelism***, i.e. the number of threads is determined by the data structure.

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#define CHUNKSIZE 10
#define N 100

int main (int argc, char *argv[]) {
    float a[N], b[N], c[N];

    // Initialise some random(ish) data
    for (int i = 0; i < N; i++) {
        a[i] = b[i] = i * 1.0;
    }

    // Create a workgroup
#pragma omp parallel shared(a,b,c)
    {
        int tid = omp_get_thread_num();
        // Run on "master" thread (arbitrary tid)
        if (tid == 0) {
            int nthreads = omp_get_num_threads();
            printf("Number of threads = %d\n", nthreads);
        }
        printf("Thread %d starting...\n",tid);

#pragma omp for schedule(dynamic, CHUNKSIZE)
        for (int i = 0; i < N; i++) {
            c[i] = a[i] + b[i];
            printf("Thread %d: c[%d] = %f\n",tid,i,c[i]);
        }
    }  /* end of parallel section */
}
```

1. Create `omp_for.c` either from above, or using the version in the Lab files folder.
2. Review the source code to understand the data parallelism introduced by the for directive. Please ask if you do not udnerstand.
3. Compile using either command line (like in part 1) or the Makefile and run
4. Think about (you might want to use `omp_for | sort` to see what computations are happening)
    - Does `i` take the values you expect from looking at the code?
    - Does `CHUNKSIZE` do what you expect?
    - What if you make `CHUNKSIZE` really small/large?
5. Another option for `schedule` is `static` (instead of `dynamic`). What happens if you use this instead?
6. This code uses two OpenMP blocks, first launch the workgroups and then add data parallelism with a for. You can also do `omp parallel for ...` which combine them both. Why didn't we do this here, and is it possible to write this code with a single `parallel omp for`?

Parallel Reduction Example

Reduction is a commonly used pattern of parallel computation where we "collapse" a set of values into a single result. Computing the sum (or product) of a list of numbers is a reduction: we reduce the list by applying + to every number and the current sum, initially 0.0. It is famously part of the Google MapReduce pattern.

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

#define N 100
#define CHUNK 10

int main (int argc, char* argv)  {
    float a[N], b[N];

    // Generate some random data
    for (int i = 0; i < N; i++) {
        a[i] = i * 1.0;
        b[i] = i * 2.0;
    }
```

```
    float result = 0.0f;
#pragma omp parallel for                        \
    schedule(static,CHUNK)                      \
    reduction(+:result)
    for (int i = 0; i < N; i++) {
        result = result + (a[i] * b[i]);
    }

    printf("Final result = %f\n",result);
}
```

1. Create `omp_reduce.c` either from above, or using the version in the Lab files folder.
2. Review the source code to understand the control parallelism introduced by the section directive.
3. Compile it and run it
4. Think about how you would write this if you **didn't** have the reduction clause?

Parallel Tasks Example

With `omp for` we introduce *data* parallelism, i.e. the loop bounds are "chunked" and passed to different threads (possibly dynamically). The task directive introduces parallel threads of control. Each task can be scheduled as required (and tasks may spawn further tasks) allowing for dynamic forms of parallelism.

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#define N 100000000

void add(float* out, float* a, float* b) {
    int tid = omp_get_thread_num();
    printf("Thread %d doing add\n",tid);

#pragma omp task
{
    int tid = omp_get_thread_num();
    printf("Thread %d doing add half\n",tid);
    for (int i = 0; i < N/2; i++) {
        out[i] = a[i] + b[i];
    }
}

#pragma omp task
{
    int tid = omp_get_thread_num();
    printf("Thread %d doing add half\n",tid);
    for (int i = N/2; i < N; i++) {
        out[i] = a[i] + b[i];
    }
}

#pragma omp taskwait
} // add

void mul(float* out, float* a, float* b) {
    int tid = omp_get_thread_num();
    printf("Thread %d doing mul\n",tid);
    for (int i = 0; i < N; i++) {
        out[i] = a[i] * b[i];
    }
}

int main (int argc, char *argv[]) {
    float *a, *b, *c, *d;
    a = malloc(sizeof(float) * N);
    b = malloc(sizeof(float) * N);
    c = malloc(sizeof(float) * N);
    d = malloc(sizeof(float) * N);

    // Initialise some random(ish) data
    for (int i = 0; i < N; i++) {
        a[i] = i * 1.5;
        b[i] = i + 22.35;
        c[i] = d[i] = 0.0;
    }

#pragma omp parallel shared(a,b,c,d)
    {
        // One thread only executes this. Other threads can execute the created tasks
#pragma omp single
        {
            int tid = omp_get_thread_num();
            printf("Thread %d launching tasks\n",tid);

#pragma omp task
```

```
        add(c, a, b);

#pragma omp task
        mul(c, a, b);

#pragma omp task
        add(d, a, b);

#pragma omp task
        mul(d, a, b);
        }  // end tasks

    }  // end parallel (threads waiting here will steal work)
    free(a); free(b); free(c); free(d);
}
```

1. Create `omp_tasks.c` either from above, or using the version in the Lab files folder.
2. Review the source code to understand the task parallelism introduced by the section directive.
3. Compile it and run it several times observe any differences in output.
4. Reflect: What happens when you have more than two threads? Think about which thread(s) are doing the work. Is the computation **deterministic**? Is the coordination **deterministic**?

Try it Yourself—Programming Exercises

Jumble

This application creates a new dataset based on combining neighbouring elements of a given dataset. The actual computation doesn't matter much here, it is just to try out parallelising the code.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main(int argc, char *argv[]) {

    // 800 million elements; Nothing a modern machine can't handle, but enough
    // to time in ms (which is more stable than ns). Adjust if needed.
    long int arraySize = 800000000;

    // Allocate some arrays
    double* x = (double*) malloc((size_t) (arraySize * sizeof(double)));
    double* y = (double*) malloc((size_t) (arraySize * sizeof(double)));

    // Initialize x with some random junk
    for (int i = 0; i < arraySize; i++) {
        x[i] = ( (double) i ) / ( i + 1000 );
    }

    struct timespec start, end;
    clock_gettime(CLOCK_MONOTONIC, &start);

    // Compute values of y based on neighbours in x
    for (int i = 0; i < (arraySize-1); i++) {
        y[i] = (x[i]/x[i+1]) + (x[i+1]*2.14) - (x[i]/5.84);
    }

    clock_gettime(CLOCK_MONOTONIC, &end);

    free(x);
    free(y);

    long long unsigned diff_ms  =
        (((double)end.tv_sec*1e9 + end.tv_nsec)
         - ((double)start.tv_sec*1e9 + start.tv_nsec))
        / 1000000;

    printf("Elapsed CPU time = %llu ms\n", diff_ms);
    return 0;
}
```

1. Create `jumble_par.c` either from above, or starting with `jumble_seq.c` in the Lab files folder.
2. Review the source code, compile, and run it sequentially.
3. Adapt the program to be parallel using OpenMP; you can decide the "best" way to parallelise it (data/sections/tasks etc).
4. Evaluate the running time and parallel performance as `OMP_NUM_THREADS` ranges over 1, 2, 4, 8, 16 and 32. What do you notice? [Remember: how many *physical* cores does your machine have?].

Primes

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

// returns 1 if argument is prime, 0 if not
// NOT an efficient prime calculator; do not use in real code!
int is_prime(unsigned long number) {
    unsigned long tNum;
    unsigned long tLim;

    if (number == 1 || number == 2) {
        return 1;
    }
    if ((number % 2) == 0) {
        return 0;
    }

    // Reminder: (initialisers, condition, incrementers)
    // Can have multiple expressions for each, as used here.
    for (tNum = 3, tLim = number; tLim > tNum; tLim = number / tNum, tNum += 2) {
        if ((number % tNum) == 0) {
            return 0;
        }
    }
    return 1;
}

int main(int argc, char *argv[]) {
    unsigned long limit = 100;
    unsigned long count = 0;

    // Let the user specify the limit they want
    if (argc > 2) {
        fprintf(stderr, "usage: ./primes [limit]\n");
        return -1;
    } else if (argc == 2) {
        sscanf(argv[1], "%lu", &limit);
    }

    printf("Computing up to limit: %lu\n", limit);

    struct timespec start, end;
    clock_gettime(CLOCK_MONOTONIC, &start);

    for (int i = 1; i < limit; i++) {
        if (is_prime(i)) {
            count++;
        }
    }

    clock_gettime(CLOCK_MONOTONIC, &end);
    long long unsigned diff_ms  =
        (((double)end.tv_sec*1e9 + end.tv_nsec)
         - ((double)start.tv_sec*1e9 + start.tv_nsec))
        / 1000000;

    printf("%lu primes computed in %llu ms, %llu ms per prime (avg)\n",
           count, diff_ms, diff_ms / limit);

    return 0;
}
```

1. Create `primes_par.c` either from above, or starting with `primes_seq.c` in the Lab files folder.
2. Run the program to compute the primes. Try to find a good value for limit that gives enough runtime to benefit from parallelism, e.g. 5–10 seconds. On my machine I needed about 10 million to get about 5ms.
3. Adapt the program to be parallel using OpenMP. Hints: Think carefully about shared and private variables; Think carefully about any critical regions.
4. Check that your threaded code finds the same (number of) primes as the sequential version.
5. Evaluate the parallel performance as `OMP_NUM_THREADS` ranges over 1, 2, 4, 8, 16 and 32.
6. Reflect: How does the programming effort of produce the OpenMP primes program compare with the effort of threaded programming in some other model you know, e.g. Java Threads, PThreads from systems programming?