# ACSE6- Assignment 1
# OpenMP Programming Coursework

## Task 1 – "hello world" program

An OpenMP hello world program is given below:

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

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

    // Fork a team of threads giving them their own copies of variables
    #pragma omp parallel private(nthreads, tid)
    {
        tid = omp_get_thread_num();     // Obtain thread number
        //print Hello World from thread tid – COMPLETE HERE

        if (tid == 0)
        { // Only master thread does this nthreads =
            omp_get_num_threads();
            //print number of threads – COMPLETE HERE
        }
    }   // All threads join master thread and disband
    return(0);
}
```

Complete the code in the specified places.

This program has the basic **parallel** construct for defining a single parallel region for multiple threads. It also has a **private** clause for defining a variable local to each thread. *Remember that OpenMP constructs such as parallel have their opening braces on the next line and not on the same line.*

Create this program and call it **main_task1.cpp**. Compile the program on your own computer. Execute the program. You should get a listing showing a number of threads such as:

```
Hello World from thread = 0
Number of threads = 4
Hello World from thread = 3
Hello World from thread = 2
Hello World from thread = 1
```

The number of threads will depend on your computer system. Execute the program at least four times. Explain your output. Does the thread order change? Why?

Alter the number of threads to 32. Just try adding the following function:

**omp_set_num_threads(32);**//***BEFORE*** the parallel region **pragma**. Re-execute the program.

## What you should attempt to generate for each task

Your submission document should include the following:

1) Compiling and running the program on your computer and understanding of output and thread order.
2) Running the program with 1, 2, 10, and 32 threads

## Task 2 – Work sharing with the "for" construct

This task explores the use of the **for** work-sharing construct. The following program **main_task2.cpp** adds two vectors together using a work-sharing approach to assign work to threads:

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

int main (int argc, char *argv[])
{
    int nthreads, tid, i, chunk; double a[N], b[N], c[N];

    for (i=0; i < N; i++)
        a[i] = b[i] = i * 1.0;   // initialize arrays chunk = CHUNKSIZE;

    #pragma omp parallel shared(a,b,c,nthreads,chunk) private(i,tid)
    {
        tid = omp_get_thread_num();
        if (tid == 0) {
            nthreads = omp_get_num_threads();
            COMPLETE here, PRINT nthreads
        }
       COMPLETE here, PRINT: "Thread" << tid << "starting …"

        #pragma omp for schedule(dynamic,chunk)
        for (i=0; i<N; i++){
            c[i] = a[i] + b[i];
            COMPLETE here, PRINT: tid, i, c[i] //What are these VARIABLES?
        }
    } /* end of parallel section */
    return(0);
}
```

Complete the code in the specified places. This program has an overall **parallel** region within which there is a work-sharing **for** construct. Compile and execute the program. Depending on the scheduling of work different threads might add elements of the vector. It may be that one thread does all the work. Execute the program several times to see any different thread scheduling. In the case that multiple threads are being used, observe how they may interleave.

**Time of execution:** Measure the execution time by instrumenting the OpenMP code with routine **omp_get_wtime()** at the beginning and end of the program and finding the elapsed in time. The function **omp_get_wtime()** returns a **double**.

**Experimenting with Scheduling:** Alter the code from **dynamic** scheduling to **static** scheduling and repeat. What are your conclusions? Alter the code from **static** scheduling to **guided** scheduling (chunk size is irrelevant) and repeat. What are your conclusions?

**Task 3 – Work-sharing with the sections construct**

This task explores the use of the **sections** construction. The program **main_task3.cpp** below adds elements of two vectors to form a third and also multiplies the elements of the arrays to produce a fourth vector.

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#define N 50

int main (int argc, char *argv[]) {
    int i, nthreads, tid;
    double a[N], b[N], c[N], d[N];

    for (i=0; i<N; i++) {  // Some initializations, arbitrary values a[i] = i * 1.5;
                    b[i] = i + 22.35;
                     c[i] = d[i] = 0.0;
    }

    #pragma omp parallel shared(a,b,c,d,nthreads) private(i,tid)
    {
        tid = omp_get_thread_num();
        if (tid == 0) {
            nthreads = omp_get_num_threads();
            COMPLETE here, PRINT nthreads
        }
       COMPLETE here, PRINT: Thread tid starting …

        #pragma omp sections nowait
        {
            #pragma omp section
            {
                COMPLETE Thread tid doing section 1
                for (i=0; i<N; i++) {
                    c[i] = a[i] + b[i];
                    COMPLETE PRINT tid,i,c[i]) – what are these variables?
                }
            }

            #pragma omp section
            {
                COMPLETE Thread tid doing section 2
                for (i=0; i<N; i++) {
                    d[i] = a[i] * b[i];
                    COMPLETE PRINT tid,i,d[i]) – what are these variables?
                }
            }
        } // end of sections

       COMPLETE here, PRINT: Thread tid done …
    }                                   // end of parallel section
    return(0);
}
```

Complete the code in the specified places. This program has a parallel region but now with variables declared as shared among the threads as well as private variables. Also, there is a sections work sharing construct. Within the sections construct, there are individual section blocks that are to be executed once by one member of the team of threads. *Remember that OpenMP constructs such as sections and section have their opening braces on the next line and not on the same line.*

Compile and execute the program and make conclusions on its execution.

**Task 4 – Matrix Multiplication**

A sequential program for matrix multiplication given here:

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

int main(int argc, char *argv)
{
    omp_set_num_threads(8);//set number of threads here

    double A[N][N], B[N][N], C[N][N];

    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++)  {
            A[i][j] = j*1; B[i][j] = i*j+2;
            C[i][j] = j-i*2;
        }
    }
    double start = omp_get_wtime(); //start time measurement

    double sum(0);
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            sum = 0;
            for (int k=0; k < N; k++) {
                sum += A[i][k]*B[k][j];
            }
            C[i][j] = sum;
        }
    }

    double end = omp_get_wtime();  //end time measurement
    COMPLETE PRINT Time of computation: end-start seconds
    return(0);
}
```

The size of the *N* x *N* matrices in the program is set to 100 x 100. Change this to the maximum size you can use on your system without getting a segmentation fault.

Please parallelize this matrix multiplication program in two different ways:

1. Add the necessary **pragma** to parallelize the outer **for** loop in the matrix multiplication;
2. Remove the **pragma** for the outer **for** loop and add the necessary **pragma** to parallelize the middle **for** loop in the matrix multiplication;

In both cases, collect timing data with 1 thread, 4 threads, 8 threads, and 16 threads. You will find that when you run the same program several times, the timing values can vary significantly. Therefore, add a loop in the code to execute the program 10 times and display the average time.