

# Assignment 4

## OpenMP and hybrid OpenMP/MPI Programming Assignment

B. Wilkinson and C Ferner: Modification date October 15, 2012

### Overview

This assignment explores using OpenMP alone and with MPI.

Part 1 provides basic practice in coding, compiling and running OpenMP programs, covering hello world program, timing, using work sharing `for`, and sections constructs. OpenMP code is given. You are also asked to parallelize matrix multiplication using the work sharing `for` construct and draw conclusions. Code for sequential matrix multiplication is given.

Part 2 asks you to write an OpenMP program for the astronomical  $n$ -body problem to include forwarding graphical output from the server to the client PC. Basic code for the graphics is given.

Part 3 is for graduate students only (extra credit for undergraduates). It asks you to modify the astronomical  $n$ -body program in part 2 to use both OpenMP and MPI together in one program, taking advantage of both shared memory systems and a message-passing cluster. A sample hybrid program is given to illustrate how hybrid OpenMP/MPI programs can be constructed and compiled.

### Preliminaries

For this assignment, **coit-grid05.uncc.edu** at UNC-Charlotte will be used for OpenMP– four quad-core processor (16-core) shared memory system. You cannot ssh directly into this computer. First log onto **coit-grid01.uncc.edu**. From **coit-grid01.uncc.edu**, ssh into **coit-grid05.uncc.edu** with the command:

```
ssh coit-grid05.uncc.edu
```

It will prompt for your password. Create a directory called **OpenMP** to hold all the files for this assignment and `cd` into this directory.

This assignment requires a compiler that compiles OpenMP programs. The newer versions of the **gcc** compiler will compile OpenMP programs but we have also installed the Intel C++ compiler called **icc** as this has good support for multithreaded OpenMP programs.

### Part 1 – OpenMP Tutorial (undergraduates 40%, graduates 30%)

The purpose of this part is to become familiar with using OpenMP constructs. Task 1, 2, and 3 are derived from the OpenMP tutorial at <https://computing.llnl.gov/tutorials/openMP/exercise.html>. Task 4 is derived from material provided by Professor B. Kurtz, Appalachian State University.

## Task 1 – Compile a “hello world” program

An 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
        printf("Hello World from thread = %d\n", tid);

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

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.*

Compile the program using the command:

```
icc -openmp omp_hello.c -o omp_hello
```

Execute the program with the command:

```
./omp_hello
```

You should get a listing showing 16 threads such as:

```
Hello World from thread = 0
Number of threads = 16
Hello World from thread = 4
Hello World from thread = 3
Hello World from thread = 2
Hello World from thread = 7
Hello World from thread = 1
Hello World from thread = 6
Hello World from thread = 5
Hello World from thread = 8
```

Hello World from thread = 15  
Hello World from thread = 9  
Hello World from thread = 14  
Hello World from thread = 10  
Hello World from thread = 13  
Hello World from thread = 12  
Hello World from thread = 11

Alter the number of threads to 4. There are three ways to do this and you should try all three (separately).

### Method 1:

Altering the environment variable with the command (at the command prompt):

```
export OMP_NUM_THREADS=4
```

Check value is correct with the command:

```
echo $OMP_NUM_THREADS
```

Re-execute the program.

### Method 2:

Undo method 1 using the command:

```
unset OMP_NUM_THREADS
```

Add the following function *BEFORE* the parallel region pragma:

```
omp_set_num_threads(4);
```

Re-execute the program.

### Method 3:

Undo method 2 by removing the `omp_set_num_threads()` function. Add the following clause to the parallel region pragma statement:

```
num_threads(4)
```

Re-execute the program.

## What to submit from this task

Your submission document should include the following:

- 1) A copy of the hello world source program;
- 2) Screenshot from compiling and running your hello world program;

- 3) Screenshots from using the 3 above methods to alter the number of threads (this should include evidence of the modification as well as the output from running the program).

## Task 2 – Work sharing with the for construct

This task explores the use of the **for** work-sharing construct. The following program 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;
    float 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();
            printf("Number of threads = %d\n", nthreads);
        }
        printf("Thread %d starting...\n",tid);

        #pragma omp for schedule(dynamic,chunk)
        for (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 */
}
```

This program has an overall **parallel** region within which there is a work-sharing **for** construct. Compile and execute the program. Depending upon 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.

## 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?

## Time of execution

Measure the execution time by instrumenting the MPI code with the OpenMP 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**.

## What to submit from this task

Your submission document should include the following:

- 1) A copy of the source program;
- 2) Screenshot from compiling and running the program;
- 3) Screenshots from of running the program with dynamic and static scheduling
- 4) Screenshot of output of execution time from instrumenting the program.

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

This task explores the use of the **sections** construction. The program 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;
    float 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();
```

```

        printf("Number of threads = %d\n", nthreads);
    }
    printf("Thread %d starting...\n",tid);

#pragma omp sections nowait
{
    #pragma omp section
    {
        printf("Thread %d doing section 1\n",tid);
        for (i=0; i<N; i++) {
            c[i] = a[i] + b[i];
            printf("Thread %d: c[%d]= %f\n",tid,i,c[i]);
        }
    }

    #pragma omp section
    {
        printf("Thread %d doing section 2\n",tid);
        for (i=0; i<N; i++) {
            d[i] = a[i] * b[i];
            printf("Thread %d: d[%d]= %f\n",tid,i,d[i]);
        }
    }
} // end of sections
    printf("Thread %d done.\n",tid); // end of parallel section
}
}

```

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.

## What to submit from this task

Your submission document should include the following:

- 1) A copy of the source program;
- 2) Screenshot from compiling and running the program;

## Task 4 – Matrix Multiplication

In this part, you are to add OpenMP constructs to the sequential program for matrix multiplication given overleaf:

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#define M 500
#define N 500

int main(int argc, char *argv) {
    omp_set_num_threads(8); //set number of threads here
    int i, j, k;
    double sum;
    double start, end;                // used for timing
    double **A, **B, **C;

    A = malloc(M*sizeof(double *));
    B = malloc(M*sizeof(double *));
    C = malloc(M*sizeof(double *));

    for (i = 0; i < M; i++) {
        A[i] = malloc(N*sizeof(double));
        B[i] = malloc(N*sizeof(double));
        C[i] = malloc(N*sizeof(double));
    }

    for (i = 0; i < M; i++) {
        for (j = 0; j < N; j++) {
            A[i][j] = j*1;
            B[i][j] = i*j+2;
            C[i][j] = j-i*2;
        }
    }

    start = omp_get_wtime();          //start time measurement

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

```

    }
}

end = omp_get_wtime(); //end time measurement
printf("Time of computation: %f\n", end-start);
}

```

You are to parallelize this algorithm in two different ways:

1. Add the necessary pragma to parallelize the outer **for** loop
2. Remove the pragma for the outer **for** loop and create a pragma for the middle **for** loop

Collect timing data given 1, 4, 8, and 16 threads and two different matrix sizes. You will find that when you run the same program several times, the timing values can vary significantly. Therefore, for each set of conditions, collect ten data values and average them. You are encouraged to use a spreadsheet program either from MS Office or OpenOffice to store your data and perform the necessary calculations.

Here are the conditions you should collect data for:

1. No parallelization at all (that is, the given program)
2. Parallelizing the outer loop with 1, 4, 8, and 16 threads using matrix sizes 50x50 and 500x500
3. Parallelizing the middle loop with 1, 4, 8, and 16 threads using matrix sizes 50x50 and 500x500

Collect timing data for each case; average the result based on ten test runs each.

Make tables and graphs of your average timing data and put them in the submitted report. After you have reported your results, try to explain them as best as possible. Include two source code files (outer loop - 8 threads-500x500 and middle loop - 8 threads-500x500).

## What to submit from this task

Your submission document should include the following:

- 1) Two copies of the source program: (outer loop - 8 threads-500x500 and middle loop - 8 threads-500x500);
- 2) Two screenshot from compiling and running the program: one for each of the two ways;
- 3) Results of your graphs of the average timings;
- 4) Your conclusions and explanations of the data.



## Part 2 Astronomical $n$ -Body Problem (60% undergraduates, 40% graduates)

### The Problem

The objective is to find the positions and movements of bodies in space (e.g., planets) that are subject to gravitational forces from other bodies using Newtonian laws of physics. The gravitational force between two bodies of masses  $m_a$  and  $m_b$  is given by:

$$F = \frac{Gm_a m_b}{r^2}$$

where  $G$  is the gravitational constant and  $r$  is the distance between the bodies. When there are multiple bodies, each body will feel the influence of each of the other bodies and the forces will sum together (taking into account the direction of each force). Subject to forces, a body will accelerate according to Newton's second law:

$$F = ma$$

where  $m$  is the mass of the body,  $F$  is the force it experiences, and  $a$  is the resultant acceleration. All the bodies will move to new positions due to these forces and have new velocities. Written as differential equations, we have:

$$F = m \frac{dv}{dt}$$

and

$$v = \frac{dx}{dt}$$

where  $v$  is the velocity. For a computer simulation, we use values at particular times,  $t_0$ ,  $t_1$ ,  $t_2$ , and so on, the time intervals being as short as possible to achieve the most accurate solution. Let the time interval be  $\Delta t$ . Then, for a particular body of mass  $m$ , the force is given by:

$$F = \frac{m(v^{t+1} - v^t)}{\Delta t}$$

and a new velocity

$$v^{t+1} = v^t + \frac{F\Delta t}{m}$$

where  $v^{t+1}$  is the velocity of the body at time  $t + 1$ , and  $v^t$  is the velocity of the body at time  $t$ . If a body is moving at a velocity  $v$  over the time interval  $\Delta t$ , its position changes by

$$x^{t+1} - x^t = v\Delta t$$

where  $x^{t+1}$  is its position at time  $t+1$  and  $x^t$  is its position at time  $t$ . Once bodies move to new positions, the forces change and the computation has to be repeated. The velocity is not actually constant over the time interval,  $\Delta t$ , so only an approximate answer is obtained.

**Three-Dimensional Space.** If the bodies are in a three-dimensional space, all values are vectors and have to be resolved into three directions, x, y, and z. In a three-dimensional space having a coordinate system (x, y, z), the distance between the bodies at  $(x_a, y_a, z_a)$  and  $(x_b, y_b, z_b)$  is given by:

$$r = \sqrt{(x_b - x_a)^2 + (y_b - y_a)^2 + (z_b - z_a)^2}$$

The forces are resolved in the three directions, using:

$$F_x = \frac{Gm_a m_b}{r^2} \left( \frac{x_b - x_a}{r} \right)$$

$$F_y = \frac{Gm_a m_b}{r^2} \left( \frac{y_b - y_a}{r} \right)$$

$$F_z = \frac{Gm_a m_b}{r^2} \left( \frac{z_b - z_a}{r} \right)$$

where the particles have the coordinates  $(x_a, y_a, z_a)$  and  $(x_b, y_b, z_b)$ .

The forces due to all the bodies on each body are added together in each dimension to obtain the final force on each body. Finally, the new position and velocity of each body are computed due to the forces. The velocity must also be resolved in three directions. For a simple computer solution, we usually assume a three-dimensional space with fixed boundaries. Actually, the universe is continually expanding and does not have fixed boundaries!

*For this assignment, you will use two-dimensional space so you will only need  $F_x$  and  $F_y$ .*

**Sequential Code.** The overall gravitational  $N$ -body computation can be described by the following steps:

```

for (t = 0; t < tmax; t++) { // for each time period

    for (a = 0; a < N; a++) { // for each pair of bodies calculate force on body due to other bodies
        for (i = 0; i < N; i++) {
            if (a != i) { // for different bodies
                x_diff = ... ; // compute distance between body a and body i in x direction
                y_diff = ... ; // compute distance between body a and body i in y direction
                r = ... ; // compute distance r
                F = ... ; // compute force on bodies
                Fx += ... ; // resolve and accumulate force in x direction
                Fy += ... ; // resolve and accumulate force in y direction
            }
        }
    }

    for (i = 0; i < N; i++) { // for each body, compute and update positions and velocity
        A[i][x_velocity] = ... ; // new velocity in x direction
        A[i][y_velocity] = ... ; // new velocity in y direction
        A[i][x_position] = ... ; // new position in x direction
        A[i][y_position] = ... ; // new position in y direction
    }
}

```

```
}
```

```
} // end time period
```

### Task 1

Write a sequential C program that computes the movement on  $N$  bodies in two dimensions of a time period, where  $N$  is a constant. Set  $N = 8$ . Let the number of iterations be another constant  $T$ . Set  $T = 100$ . Choose suitable masses, initial positions and velocities and store in an array, i.e.

**Table 1: Input data**

Body	Mass	Position in x direction	Position in y direction	Velocity in x direction	Velocity in y direction
0					
1					
2					
3					
4					
5					
6					
7					

Update this array after each iteration and display the values.

### Task 2

Add code to that of Task 1 to display the movement of the bodies on the client computer using X11 graphics (or another way). How to create and forward X11 graphical output is given in a separate document “Notes on creating graphical output - X-11 graphics.” Execute the code and provide screen shots of the output (video better).

### Task 3

Re-write your program as an OpenMP program to execute on **coit-grid05.uncc.edu** using all 4 processors (16 cores in total). Incorporate code to measure the time of execution.

### What to submit from this part

Your submission document should include the following:

- 1) A copy of the source program;
- 2) A screenshot of compiling and executing the program
- 3) A screenshot of the X11 display on your machine showing the bodies
- 4) Comparison of the time of execution for sequential and parallel versions along with your conclusions and explanation of this data.

### Part 3 Hybrid OpenMP/MPI Astronomical n-Body Program

**(For graduate students only 30%, extra credit for undergraduates)**

In this part, you are to write a C program that has both MPI message-passing routines and OpenMP threads. This is a form of “hybrid programming” and extremely relevant for clusters of multiprocessor and multicore computer systems. The message-passing routines are used to pass messages between computer systems and the threads are run using the multiple cores on each system. The overall structure of the program is illustrated below:

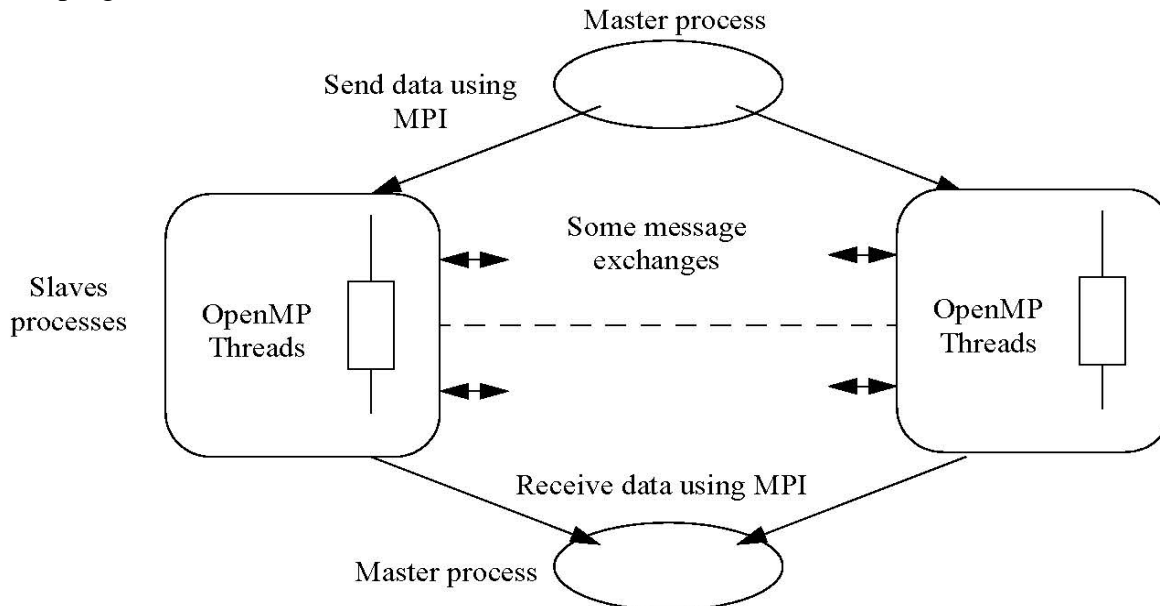


Figure 1 Hybrid MPI message passing and OpenMP threads

**Sample MPI/OpenMP program and compiling.** A sample MPI/OpenMP program is given below:

```
#include <stdio.h>
#include <string.h>
#include <stddef.h>
```

```

#include <stdlib.h>
#include "mpi.h"

#define CHUNKSIZE 10
#define N 100

void openmp_code() {
    int nthreads, tid, i, chunk;
    float 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();
            printf("Number of threads = %d\n", nthreads);
        }

        printf("Thread %d starting...\n",tid);

        #pragma omp for schedule(dynamic,chunk)
        for (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 */
}

main(int argc, char **argv ) {
    char message[20];
    int i,rank, size, type=99;
    MPI_Status status;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD,&size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    if (rank == 0) {
        strcpy(message, "Hello, world");
        for (i=1; i<size; i++)
            MPI_Send(message, 13, MPI_CHAR, i, type, MPI_COMM_WORLD);
    }
    else
        MPI_Recv(message, 20, MPI_CHAR, 0, type, MPI_COMM_WORLD, &status);

    openmp_code();          //all MPI processes run OpenMP code, no message passing
}

```

```
printf( "Message from process =%d : %.13s\n", rank,message);
MPI_Finalize();
}
```

Compile with:

```
mpicc mpi_test.c -o mpi_out -fopenmp
```

Note this command invokes the regular cc compiler not the Intel icc compiler.

Modify your OpenMP program in Part 2 to be a hybrid program using both OpenMP and MPI. Use both **coit-grid05.uncc.edu** and **coit-grid02.uncc.edu** with MPI message passing between them and OpenMP threads on each. Use the following approach:

1. The master process holds the array and broadcast the array to the slaves
2. Each slave presents 4 bodies and computes the new position for those bodies
3. All positions are gathered by the master process and the array is updated
4. Steps 1 - 3 are repeated

## What to submit from this part

Your submission document should include the following:

- 1) A copy of the source program to perform the n-body problem using both MPI and OpenMP;
- 2) A screenshot of compiling and executing the program
- 3) A screenshot of the X11 display on your machine showing the bodies
- 4) Comparison of the time of execution for sequential and parallel versions along with your conclusions and explanation of this data.

## Assignment Preparation and Submission

Produce a document that provides the following details:

- 1) Your name and school;
- 2) Whether you are a graduate or undergraduate student;
- 3) A full explanation of your code;
- 4) Code listing;
- 5) Sample output;
- 6) Insightful conclusions.

Submit to Moodle by the due date (see home page). Combine everything into one PDF file. *All students must work individually.* **Include all code, not as screen shots of the listings but as complete properly documented code listing.**