

# Part 1

## TASK 1

### *1. Initial hello program compilation and run:*

```
nwade@ubuntu:~/Documents/itcs_4145/OpenMP$ cc -fopenmp hello.c -o hello
nwade@ubuntu:~/Documents/itcs_4145/OpenMP$ ./hello
Hello World from thread = 0
Number of threads = 8
Hello World from thread = 6
Hello World from thread = 2
Hello World from thread = 4
Hello World from thread = 1
Hello World from thread = 7
Hello World from thread = 3
Hello World from thread = 5
nwade@ubuntu:~/Documents/itcs_4145/OpenMP$
```

Each thread runs what is in the base parallel section. Thread order is based on whichever thread is tasked to run next.

### *2. Program listing with thread count altered:*

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

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

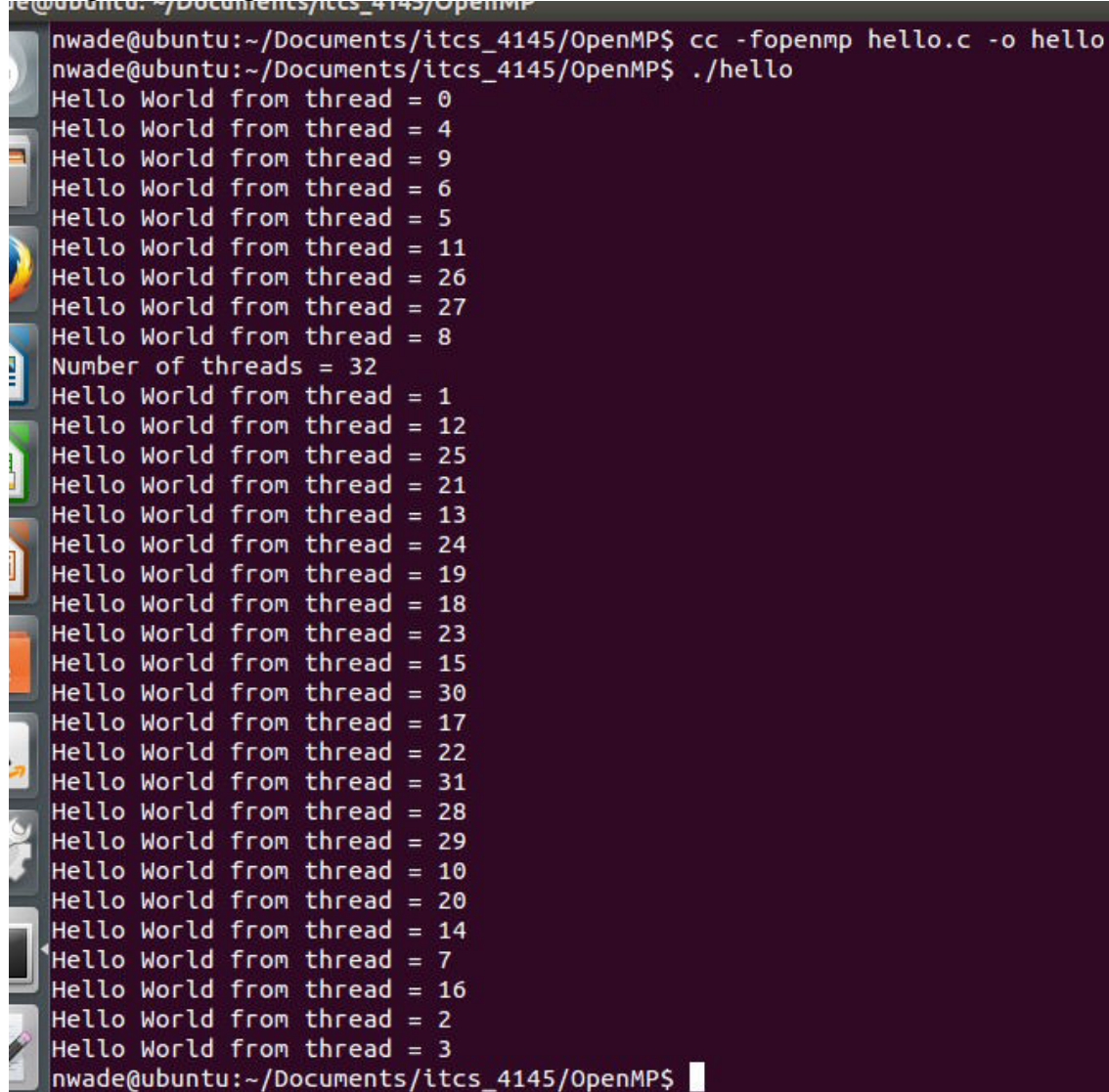
    omp_set_num_threads (32);

    #pragma omp parallel private (nthreads, tid)
    {
        tid = omp_get_thread_num();
        printf ("Hello World from thread = %d\n", tid);

        if (tid == 0)
        {
            nthreads = omp_get_num_threads();
            printf ("Number of threads = %d\n", nthreads);
        }
    }
}
```

```
    return 0;  
}
```

### 3. Program running with 32 threads:

A terminal window with a dark purple background and light green text. The prompt is 'nwade@ubuntu:~/Documents/itcs\_4145/OpenMP\$'. The user enters 'cc -fopenmp hello.c -o hello' and then './hello'. The output shows 32 'Hello World from thread = [id]' messages, where the thread IDs are: 0, 4, 9, 6, 5, 11, 26, 27, 8, 1, 12, 25, 21, 13, 24, 19, 18, 23, 15, 30, 17, 22, 31, 28, 29, 10, 20, 14, 7, 16, 2, 3. The prompt returns at the end.

```
nwade@ubuntu:~/Documents/itcs_4145/OpenMP$ cc -fopenmp hello.c -o hello  
nwade@ubuntu:~/Documents/itcs_4145/OpenMP$ ./hello  
Hello World from thread = 0  
Hello World from thread = 4  
Hello World from thread = 9  
Hello World from thread = 6  
Hello World from thread = 5  
Hello World from thread = 11  
Hello World from thread = 26  
Hello World from thread = 27  
Hello World from thread = 8  
Number of threads = 32  
Hello World from thread = 1  
Hello World from thread = 12  
Hello World from thread = 25  
Hello World from thread = 21  
Hello World from thread = 13  
Hello World from thread = 24  
Hello World from thread = 19  
Hello World from thread = 18  
Hello World from thread = 23  
Hello World from thread = 15  
Hello World from thread = 30  
Hello World from thread = 17  
Hello World from thread = 22  
Hello World from thread = 31  
Hello World from thread = 28  
Hello World from thread = 29  
Hello World from thread = 10  
Hello World from thread = 20  
Hello World from thread = 14  
Hello World from thread = 7  
Hello World from thread = 16  
Hello World from thread = 2  
Hello World from thread = 3  
nwade@ubuntu:~/Documents/itcs_4145/OpenMP$
```

## TASK 2

### 1. Source code:

```
#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];
    double start, end;

    for (i = 0; i < N; ++i)
        a[i] = b[i] = i * 1.0;

    chunk = CHUNKSIZE;

    start = omp_get_wtime();

    #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);
        }

        #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 = omp_get_wtime();
    printf("Time of computation: %f seconds\n", end - start);
    return 0;
}

```

## 2a. Compiling and running original code:

```
nwade@ubuntu:~/Documents/itcs_4145/OpenMP$ cc -fopenmp worksharing.c -o worksharing
nwade@ubuntu:~/Documents/itcs_4145/OpenMP$ ./worksharing
Number of threads = 8
thread 0: c[50] = 100.000000
thread 0: c[51] = 102.000000
thread 0: c[52] = 104.000000
thread 0: c[53] = 106.000000
thread 0: c[54] = 108.000000
thread 0: c[55] = 110.000000
thread 0: c[56] = 112.000000
thread 0: c[57] = 114.000000
thread 0: c[58] = 116.000000
thread 0: c[59] = 118.000000
thread 0: c[60] = 120.000000
thread 0: c[61] = 122.000000
thread 0: c[62] = 124.000000
thread 0: c[63] = 126.000000
thread 0: c[64] = 128.000000
thread 0: c[65] = 130.000000
thread 0: c[66] = 132.000000
thread 0: c[67] = 134.000000
thread 0: c[68] = 136.000000
thread 0: c[69] = 138.000000
thread 0: c[70] = 140.000000
thread 0: c[71] = 142.000000
thread 0: c[72] = 144.000000
thread 0: c[73] = 146.000000
thread 0: c[74] = 148.000000
thread 0: c[75] = 150.000000
thread 0: c[76] = 152.000000
thread 0: c[77] = 154.000000
thread 0: c[78] = 156.000000
thread 0: c[79] = 158.000000
thread 0: c[80] = 160.000000
thread 0: c[81] = 162.000000
thread 0: c[82] = 164.000000
thread 0: c[83] = 166.000000
thread 0: c[84] = 168.000000
thread 0: c[85] = 170.000000
thread 0: c[86] = 172.000000
thread 0: c[87] = 174.000000
thread 0: c[88] = 176.000000
thread 0: c[89] = 178.000000
thread 0: c[90] = 180.000000
thread 0: c[91] = 182.000000
thread 0: c[92] = 184.000000
thread 0: c[93] = 186.000000
thread 0: c[94] = 188.000000
thread 0: c[95] = 190.000000
```

2b. Compiling and running original code (continued):

```
thread 0: c[96] = 192.000000
thread 0: c[98] = 196.000000
thread 0: c[99] = 198.000000
thread 3: c[10] = 20.000000
thread 2: c[30] = 60.000000
thread 3: c[11] = 22.000000
thread 3: c[12] = 24.000000
thread 3: c[13] = 26.000000
thread 3: c[14] = 28.000000
thread 3: c[15] = 30.000000
thread 3: c[16] = 32.000000
thread 3: c[17] = 34.000000
thread 3: c[18] = 36.000000
thread 3: c[19] = 38.000000
thread 1: c[20] = 40.000000
thread 1: c[21] = 42.000000
thread 1: c[22] = 44.000000
thread 1: c[23] = 46.000000
thread 1: c[24] = 48.000000
thread 1: c[25] = 50.000000
thread 1: c[26] = 52.000000
thread 1: c[27] = 54.000000
thread 1: c[28] = 56.000000
thread 1: c[29] = 58.000000
thread 4: c[40] = 80.000000
thread 4: c[41] = 82.000000
thread 4: c[42] = 84.000000
thread 4: c[43] = 86.000000
thread 4: c[44] = 88.000000
thread 4: c[45] = 90.000000
thread 4: c[46] = 92.000000
thread 4: c[47] = 94.000000
thread 4: c[48] = 96.000000
thread 4: c[49] = 98.000000
thread 7: c[0] = 0.000000
thread 7: c[1] = 2.000000
thread 7: c[2] = 4.000000
thread 7: c[3] = 6.000000
thread 7: c[4] = 8.000000
thread 7: c[5] = 10.000000
thread 7: c[6] = 12.000000
thread 7: c[7] = 14.000000
thread 7: c[8] = 16.000000
thread 7: c[9] = 18.000000
thread 2: c[31] = 62.000000
thread 2: c[32] = 64.000000
thread 2: c[33] = 66.000000
thread 2: c[34] = 68.000000
thread 2: c[35] = 70.000000
thread 2: c[36] = 72.000000
thread 2: c[37] = 74.000000
thread 2: c[38] = 76.000000
thread 2: c[39] = 78.000000
Time of computation: 0.073771 seconds
```

### 3. Running with static scheduling:

```
thread 5: c[59] = 118.000000
thread 6: c[60] = 120.000000
thread 6: c[61] = 122.000000
thread 6: c[62] = 124.000000
thread 6: c[63] = 126.000000
thread 6: c[64] = 128.000000
thread 6: c[65] = 130.000000
thread 6: c[66] = 132.000000
thread 6: c[67] = 134.000000
thread 6: c[68] = 136.000000
thread 6: c[69] = 138.000000
thread 3: c[31] = 62.000000
thread 3: c[32] = 64.000000
thread 3: c[33] = 66.000000
thread 3: c[34] = 68.000000
thread 3: c[35] = 70.000000
thread 3: c[36] = 72.000000
thread 3: c[37] = 74.000000
thread 3: c[38] = 76.000000
thread 3: c[39] = 78.000000
thread 7: c[71] = 142.000000
thread 7: c[72] = 144.000000
thread 7: c[73] = 146.000000
thread 7: c[74] = 148.000000
thread 7: c[75] = 150.000000
thread 7: c[76] = 152.000000
thread 7: c[77] = 154.000000
thread 7: c[78] = 156.000000
thread 7: c[79] = 158.000000
thread 2: c[21] = 42.000000
thread 2: c[22] = 44.000000
thread 2: c[23] = 46.000000
thread 2: c[24] = 48.000000
thread 2: c[25] = 50.000000
thread 2: c[26] = 52.000000
thread 2: c[27] = 54.000000
Workspace Switcher 56.000000
thread 2: c[29] = 58.000000
thread 1: c[15] = 30.000000
thread 1: c[16] = 32.000000
thread 1: c[17] = 34.000000
thread 1: c[18] = 36.000000
thread 1: c[19] = 38.000000
thread 1: c[90] = 180.000000
thread 1: c[91] = 182.000000
thread 1: c[92] = 184.000000
thread 1: c[93] = 186.000000
thread 1: c[94] = 188.000000
thread 1: c[95] = 190.000000
thread 1: c[96] = 192.000000
thread 1: c[97] = 194.000000
thread 1: c[98] = 196.000000
thread 1: c[99] = 198.000000
Time of computation: 0.057190 seconds
```

#### 4. Running with guided scheduling:

```
thread 6: c[51] = 102.000000
thread 6: c[52] = 104.000000
thread 6: c[53] = 106.000000
thread 0: c[78] = 156.000000
thread 0: c[79] = 158.000000
thread 0: c[80] = 160.000000
thread 0: c[81] = 162.000000
thread 0: c[82] = 164.000000
thread 0: c[83] = 166.000000
thread 7: c[64] = 128.000000
thread 7: c[65] = 130.000000
thread 7: c[66] = 132.000000
thread 7: c[67] = 134.000000
thread 7: c[68] = 136.000000
thread 7: c[69] = 138.000000
thread 7: c[70] = 140.000000
thread 7: c[71] = 142.000000
thread 7: c[72] = 144.000000
thread 7: c[73] = 146.000000
thread 5: c[15] = 30.000000
thread 5: c[16] = 32.000000
thread 5: c[17] = 34.000000
thread 5: c[18] = 36.000000
thread 5: c[19] = 38.000000
thread 5: c[20] = 40.000000
thread 5: c[21] = 42.000000
thread 5: c[22] = 44.000000
thread 5: c[23] = 46.000000
thread 4: c[0] = 0.000000
thread 3: c[25] = 50.000000
thread 3: c[26] = 52.000000
thread 3: c[27] = 54.000000
thread 3: c[28] = 56.000000
thread 3: c[29] = 58.000000
thread 3: c[30] = 60.000000
thread 3: c[31] = 62.000000
thread 3: c[32] = 64.000000
thread 3: c[33] = 66.000000
thread 4: c[1] = 2.000000
thread 4: c[2] = 4.000000
thread 4: c[3] = 6.000000
thread 4: c[4] = 8.000000
thread 4: c[5] = 10.000000
thread 4: c[6] = 12.000000
thread 4: c[7] = 14.000000
thread 4: c[8] = 16.000000
thread 4: c[9] = 18.000000
thread 4: c[10] = 20.000000
thread 4: c[11] = 22.000000
thread 4: c[12] = 24.000000
Time of computation: 0.069674 seconds
```

#### 4. Conclusions:

Static appears to run the fastest by a fraction of a second, however, guided does a better job at load balancing



## TASK 3

### 1. Compilation and execution of program:

```
nwade@ubuntu:~/Documents/itcs_4145/OpenMP$ cc -fopenmp sections.c -o sections
nwade@ubuntu:~/Documents/itcs_4145/OpenMP$ ./sections
Thread 7 starting...
Thread 3 starting...
Thread 3 doing section 2
Thread 3: d[0] = 0.000000
Thread 3: d[1] = 33.525002
Thread 2 starting...
Thread 2 done.
Number of threads = 8
Thread 0 starting...
Thread 0 done.
Thread 4 starting...
Thread 4 done.
Thread 6 starting...
Thread 6 done.
Thread 5 starting...
Thread 5 done.
Thread 3: d[2] = 134.100006
Thread 3: d[3] = 301.725006
Thread 3: d[4] = 536.400024
Thread 3: d[5] = 838.125000
Thread 3: d[6] = 1206.900024
Thread 3: d[7] = 1642.724976
Thread 3: d[8] = 2145.600098
Thread 3: d[9] = 2715.524902
Thread 3: d[10] = 3352.500000
Thread 3: d[11] = 4056.525146
Thread 3: d[12] = 4827.600098
Thread 3: d[13] = 5665.724609
Thread 3: d[14] = 6570.899902
Thread 3: d[15] = 7543.125000
Thread 3: d[16] = 8582.400391
Thread 3: d[17] = 9688.725586
Text Editor 18] = 10862.099609
Thread 3: d[19] = 12102.524414
Thread 3: d[20] = 13410.000000
Thread 3: d[21] = 14784.525391
Thread 3: d[22] = 16226.100586
Thread 3: d[23] = 17734.724609
Thread 3: d[24] = 19310.400391
Thread 3: d[25] = 20953.125000
Thread 3: d[26] = 22662.898438
Thread 3: d[27] = 24439.724609
Thread 3: d[28] = 26283.599609
Thread 3: d[29] = 28194.525391
Thread 3: d[30] = 30172.500000
Thread 3: d[31] = 32217.523438
Thread 3: d[32] = 34329.601562
Thread 3: d[33] = 36508.722656
Thread 3: d[34] = 38754.902344
Thread 3: d[35] = 41068.125000
Thread 3: d[36] = 43448.398438
Thread 3: d[37] = 45895.726562
Thread 3: d[38] = 48410.097656
```

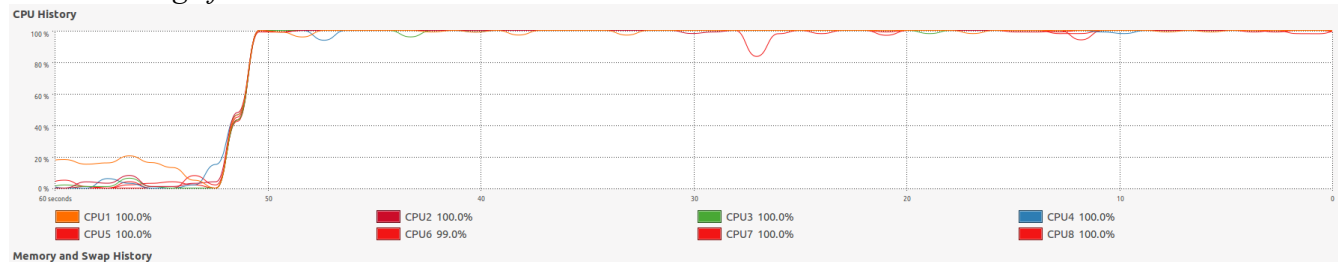


## 2. Conclusions:

One thread does one section. If a thread does not have a section it continues on to the executable line if `nowait` was added to the section directive.

## TASK 4

### 1a. CPU usage for 8 threads:



### 1b. Number threads | time to execute:

Threads	Time (Seconds)
1	27.354891
2	91.796902
4	200.657403
8	253.384475

### 2a. Execute time with private x:

71.600604s

### 2b. Execute time with public x:

100.257486s

### 2c. Explanation of difference:

Execution time increased. I expected it to go down because the threads are working together, but I think it is the communication between the 2 threads that slowed it down.

# Part 2

## Task 1

### 1. program source:

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

```
#define N 512
```

```
int main (int argc, char *argv)
{
    omp_set_num_threads(4);

    int i, j, k, x;
```

```

double sum;
double start, end;
double A[N][N], B[N][N], C[N][N], D[N][N];
int count = 1;
double average_seq, average_para;

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

start = omp_get_wtime();
for (i = 0; i < N; ++i)
{
    for (j = 0; j < N; ++j)
    {
        sum = 0;
        for (k = 0; k < N; ++k)
        {
            sum += A[i][k] * B[k][j];
        }
        C[i][j] = sum;
    }
}
end = omp_get_wtime();
average_seq += end - start;

//parallelized multiplication
start = omp_get_wtime();
#pragma omp parallel for private(j, sum, k)
for (i = 0; i < N; ++i)
{
    for (j = 0; j < N; ++j)
    {
        sum = 0;
        for (k = 0; k < N; ++k)
        {
            sum += A[i][k] * B[k][j];
        }
        D[i][j] = sum;
    }
}
end = omp_get_wtime();
average_para += end - start;

```

```

int error = 0;
for (i = 0; i < N; ++i)
{
    for (j = 0; j < N; ++j)
    {
        if ((C[i][j] - D[i][j] > 0.001) || (D[i][j] - C[i][j] > 0.001))
            error = -1;
    }
}
if(error == -1) {printf("ERROR, sequential and parallel versions give different answers\n"); }

printf("average sequential runtime: %f seconds\n", average_seq / count);
printf("average parallel runtime: %f seconds\n", average_para / count);

return 0;
}

```

## 2. compilation and execution:

```

nwade@ubuntu:~/Documents/itcs_4145/OpenMP$ cc -fopenmp matrixmult.c -o matrixmult
nwade@ubuntu:~/Documents/itcs_4145/OpenMP$ ./matrixmult
average sequential runtime: 0.100383 seconds
average parallel runtime: 0.032851 seconds
nwade@ubuntu:~/Documents/itcs_4145/OpenMP$ _

```

## 3. threads/execution time:

Threads	Sequential	Parallel
1	0.115914s	0.088233s
4	0.100383s	0.032851s
8	0.102037s	0.067744s
16	0.104859s	0.025751s

## Task 2

### 1. program source:

```

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

```

```

#define N 256

```

```

int main (int argc, char *argv)
{
    omp_set_num_threads(16);

    int i, j, k, x;
    double sum;
    double start, end;

```

```

double A[N][N], B[N][N], C[N][N], D[N][N];
int count = 1;
double average_seq, average_para;

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

start = omp_get_wtime();
for (i = 0; i < N; ++i)
{
    for (j = 0; j < N; ++j)
    {
        sum = 0;
        for (k = 0; k < N; ++k)
        {
            sum += A[i][k] * B[k][j];
        }
        C[i][j] = sum;
    }
}
end = omp_get_wtime();
average_seq += end - start;

//parallelized multiplication
start = omp_get_wtime();
for (i = 0; i < N; ++i)
{
    #pragma omp parallel for private (k, sum)
    for (j = 0; j < N; ++j)
    {
        sum = 0;
        for (k = 0; k < N; ++k)
        {
            sum += A[i][k] * B[k][j];
        }
        D[i][j] = sum;
    }
}
end = omp_get_wtime();
average_para += end - start;

int error = 0;
for (i = 0; i < N; ++i)

```

```

    {
        for (j = 0; j < N; ++j)
        {
            if ((C[i][j] - D[i][j] > 0.001) || (D[i][j] - C[i][j] > 0.001))
                error = -1;
        }
    }
    if(error == -1) {printf("ERROR, sequential and parallel versions give different answers\n"); }

    printf("average sequential runtime: %f seconds\n", average_seq / count);
    printf("average parallel runtime: %f seconds\n", average_para / count);

    return 0;
}

```

## 2. compilation and execution

```

nwade@ubuntu:~/Documents/itcs_4145/OpenMP$ ./matrixmult
average sequential runtime: 0.080585 seconds
average parallel runtime: 0.093711 seconds
nwade@ubuntu:~/Documents/itcs_4145/OpenMP$ _

```

## 3. threads/execution time:

Threads	Sequential	Parallel
1	0.080585s	0.093711s
4	0.079588s	0.044445s
8	0.092093s	0.227464s
16	0.093493s	0.126404s

## Task 3

### 1. program source:

```

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

#define N 512

int main (int argc, char *argv)
{
    omp_set_num_threads(1);

    int i, j, k, x;
    double sum;
    double start, end;
    double A[N][N], B[N][N], C[N][N], D[N][N];
    int count = 1;
    double average_seq, average_para;

```

```

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

start = omp_get_wtime();
for (i = 0; i < N; ++i)
{
    for (j = 0; j < N; ++j)
    {
        sum = 0;
        for (k = 0; k < N; ++k)
        {
            sum += A[i][k] * B[k][j];
        }
        C[i][j] = sum;
    }
}
end = omp_get_wtime();
average_seq += end - start;

//parallelized multiplication
start = omp_get_wtime();
#pragma omp parallel for private(j)
for (i = 0; i < N; ++i)
{
    #pragma omp parallel for private(sum, k)
    for (j = 0; j < N; ++j)
    {
        sum = 0;
        for (k = 0; k < N; ++k)
        {
            sum += A[i][k] * B[k][j];
        }
        D[i][j] = sum;
    }
}
end = omp_get_wtime();
average_para += end - start;

int error = 0;
for (i = 0; i < N; ++i)
{
    for (j = 0; j < N; ++j)
    {

```



```

        if ((C[i][j] - D[i][j] > 0.001) || (D[i][j] - C[i][j] > 0.001))
            error = -1;
    }
}
if(error == -1) {printf("ERROR, sequential and parallel versions give different answers\n"); }

printf("average sequential runtime: %f seconds\n", average_seq / count);
printf("average parallel runtime: %f seconds\n", average_para / count);

return 0;
}

```

## 2. compilation and execution:

```

nwade@ubuntu:~/Documents/itcs_4145/OpenMP$ ./matrixmult
average sequential runtime: 0.090886 seconds
average parallel runtime: 0.090880 seconds
nwade@ubuntu:~/Documents/itcs_4145/OpenMP$ _

```

## 3. Threads/Execution Time:

Threads	Sequential	Parallel
1	0.090886s	0.090880s
4	0.089328s	0.032245s
8	0.090206s	0.070210s
16	0.092261s	0.122470s

## Task 4

### 1. result of increasing the size:

Increasing the size of the matrices resulted in a segmentation fault. This occurs because when I run the 512 x 512, C allocates space for the arrays. Since the total amount of memory being used is  $2^{22}$ , it exceeds the amount of memory my VM has access to.

### 2. conclusions:

It seems parallelization of this program has a negative parabolic shape, meaning that efficiency goes up as you increase the number of threads before it goes back down as more are being introduced. The maximum efficiency in this case would be executing with 4 threads.

# Part 3

## 1. output from cci-gridgw:

```

[nwade3@cci-gridgw OpenMP]$ ./matrixmult
average sequential runtime: 0.116496 seconds
average parallel runtime: 0.043519 seconds
[nwade3@cci-gridgw OpenMP]$ _

```

## *2. Threads/Execution Times:*

Threads	My PC	CCI-Gridgw
1	0.090880s	0.141259s
4	0.032245s	0.039835s
8	0.070210s	0.049158s
16	0.122470s	0.043519s

## *3. conclusions:*

With the exception of the 1 thread run, cci-grid was faster as I would expect it to be a more powerful machine. Also, my conclusions about the negative parabolic shape still mostly holds true, and maximum efficiency is gained at 4 threads.