



Parallel Programming

Homework 5

Student Name : ABID ALI
Student ID : 2019380141
Student Email : abiduu354@gmail.com / 3616990795@qq.com
Lecture : Professor Tianhai Zhao
Submission : 12/22/2022
Submitted Email : zhaoth@nwpu.edu.cn

1. Try the codes in directories pi and hybrid.

Solution:

Pi

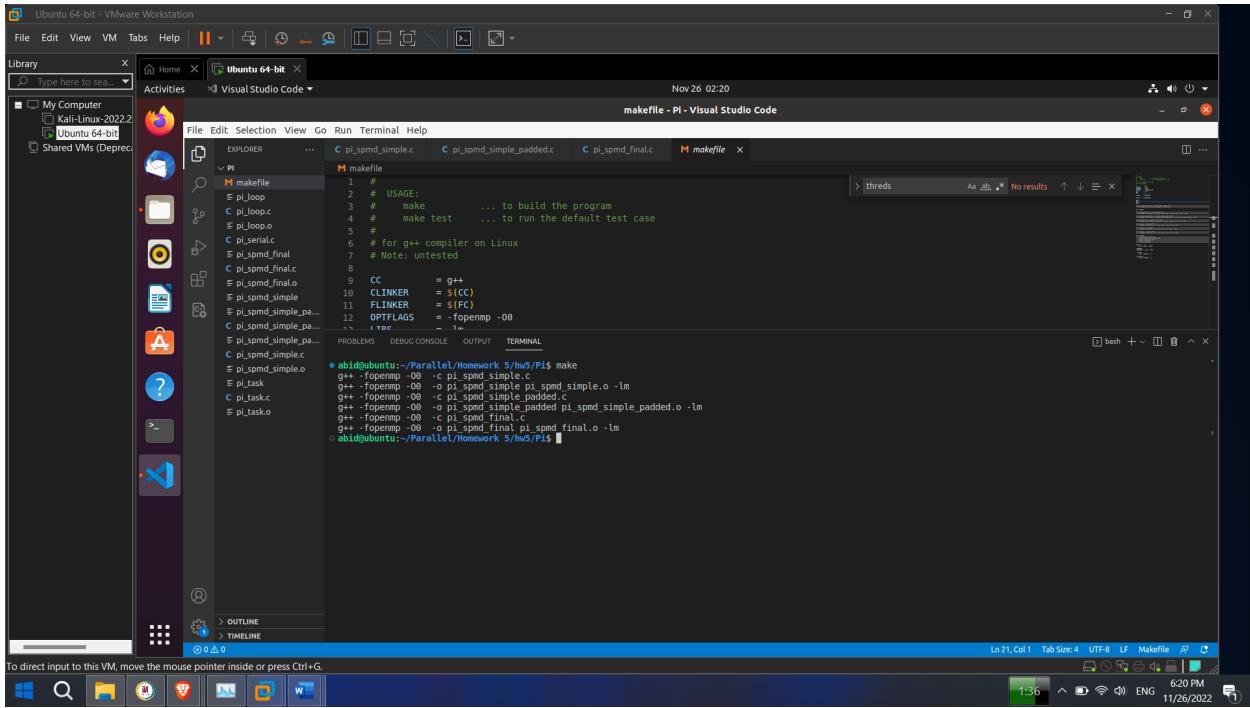


Fig : Make

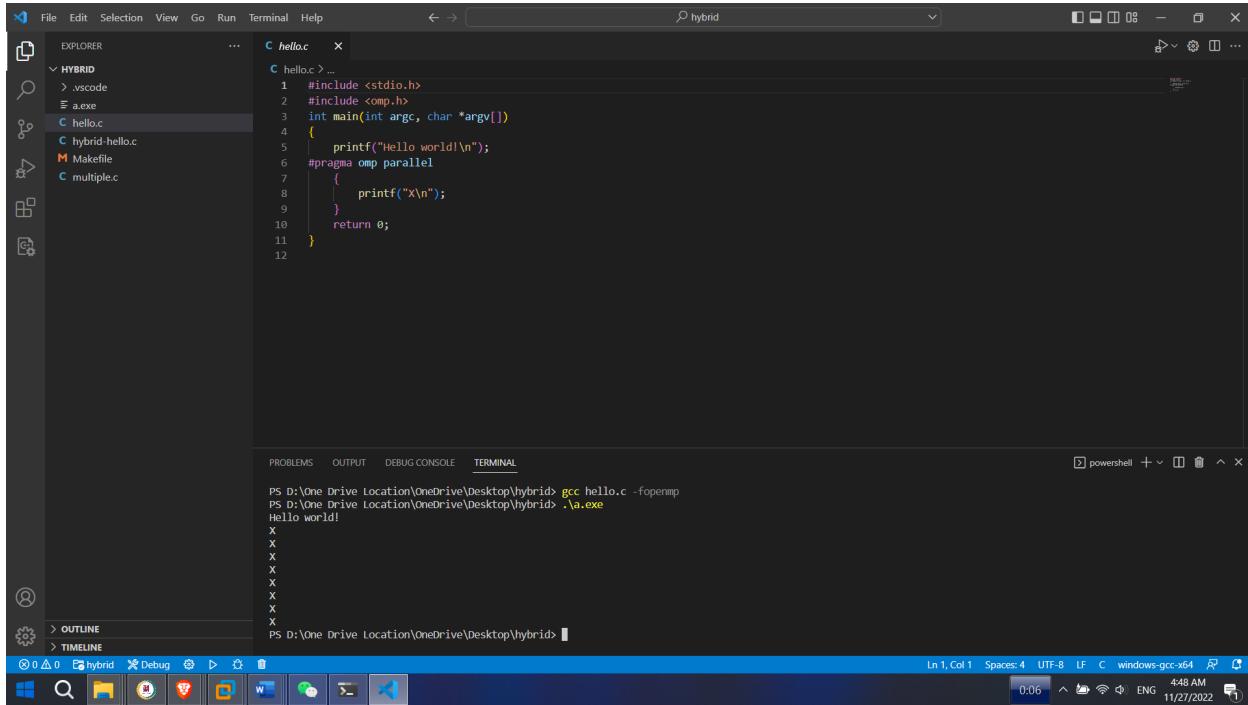
```
./pi_spmd_simple
pi is 3.141593 in 2.868942 seconds 1 threads
num threads = 1
pi is 3.141593 in 2.868942 seconds 1 threads
num threads = 2
pi is 3.141593 in 3.966735 seconds 2 threads
num threads = 2
pi is 3.141593 in 3.521823 seconds 3 threads
num threads = 3
pi is 3.141593 in 2.933224 seconds 4 threads
num threads = 4
pi is 3.141593 in 2.567633 seconds 5 threads
num threads = 5
pi is 3.141593 in 2.158360 seconds 6 threads
num threads = 6
pi is 3.141593 in 1.917942 seconds 7 threads
num threads = 7
pi is 3.141593 in 1.748899 seconds 8 threads
num threads = 8
pi is 3.141593 in 1.440747 seconds 2 threads
./pi_spmd_simple_padded
num threads = 1
pi is 3.141593 in 2.561821 seconds 1 threads
num threads = 2
pi is 3.141593 in 1.372613 seconds 2 threads
num threads = 3
pi is 3.141593 in 0.916270 seconds 3 threads
num threads = 4
pi is 3.141593 in 0.720246 seconds 4 threads
num threads = 5
pi is 3.141593 in 0.609757 seconds 5 threads
num threads = 6
pi is 3.141593 in 0.625322 seconds 6 threads
num threads = 7
pi is 3.141593 in 0.553704 seconds 7 threads
num threads = 8
pi is 3.141593 in 0.591107 seconds 8 threads
./pi_spmd_final
```

Fig : Make Test(1)

```
pi is 3.141593 in 1.440747 seconds 2 threads
pi is 3.141593 in 0.959962 seconds 3 threads
pi is 3.141593 in 0.768491 seconds 4 threads
pi is 3.141593 in 0.666807 seconds 5 threads
pi is 3.141593 in 0.623880 seconds 6 threads
pi is 3.141593 in 0.627072 seconds 7 threads
pi is 3.141593 in 0.486671 seconds 8 threads
./pi_spmd_simple
num threads = 1
pi is 3.141593 in 2.737140 seconds and 1 threads
num threads = 2
pi is 3.141593 in 1.413130 seconds and 2 threads
num threads = 3
pi is 3.141593 in 1.008401 seconds and 3 threads
num threads = 4
pi is 3.141593 in 0.785117 seconds and 4 threads
num threads = 5
pi is 3.141593 in 0.678366 seconds and 5 threads
num threads = 6
pi is 3.141593 in 0.639505 seconds and 6 threads
num threads = 7
pi is 3.141593 in 0.590847 seconds and 7 threads
num threads = 8
pi is 3.141593 in 0.683133 seconds and 8 threads
./pi_task
num threads=1 for 1073741824 steps pi = 3.141593 in 27.697250 secs
num threads=2 for 1073741824 steps pi = 3.141593 in 14.612770 secs
num threads=3 for 1073741824 steps pi = 3.141593 in 9.835000 secs
num threads=4 for 1073741824 steps pi = 3.141593 in 9.116960 secs
num threads=5 for 1073741824 steps pi = 3.141593 in 6.933383 secs
num threads=6 for 1073741824 steps pi = 3.141593 in 7.426896 secs
num threads=7 for 1073741824 steps pi = 3.141593 in 5.982972 secs
num threads=8 for 1073741824 steps pi = 3.141593 in 5.884638 secs
```

Fig : Make Test(2)

Hybrid



```
File Edit Selection View Go Run Terminal Help
EXPLORER ... C hello.c x
HYBRID > .vscode
  a.exe
  C hello.c
  C hybrid-hello.c
  M Makefile
  C multiple.c
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
powershell + ▾
PS D:\One Drive Location\OneDrive\Desktop\hybrid> gcc hello.c -fopenmp
PS D:\One Drive Location\OneDrive\Desktop\hybrid> .\a.exe
Hello world!
X
X
X
X
X
X
X
X
X
X
TERMINAL
Ln 1, Col 1 Spaces: 4 UTF-8 LF C windows-gcc-x64 4:48 AM 0:06 11/27/2022
```

Fig : Hello.c

2. Implement the matrix multiplication program with OpenMP
you are to add OpenMP constructs to the sequential program for matrix multiplication given here:

```

#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#define M 500
#define N 500
#define Max_threads 8

int main(int argc, char *argv) {
    omp_set_num_threads(Max_threads); //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));
    }
    //initialize matrix A, B, C
    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);
}

```

1. Add the necessary pragma to parallelize the outer for loop;

Theory:

Typically, we want to parallelize the outermost loop that makes sense. If we parallelize the inner loops, we will be introducing extra overhead. By having the "loop bodies" be as large as possible, we will get better overall throughput. This really boils down to Amdahl's law - in this case, the overhead involved in scheduling the parallel work items is not parallelizable, so the more of that

work you do, the lower the potential efficiency overall.

The risk is that, if there are too few items in the outer loop, you may end up where work items can't be run in parallel, since there will be a point where there are fewer items than processing cores in your system.

Provided that our outer loop has enough to keep the cores busy, it's the best place to go - especially if the amount of work done in each loop body is relatively consistent.

Solution:

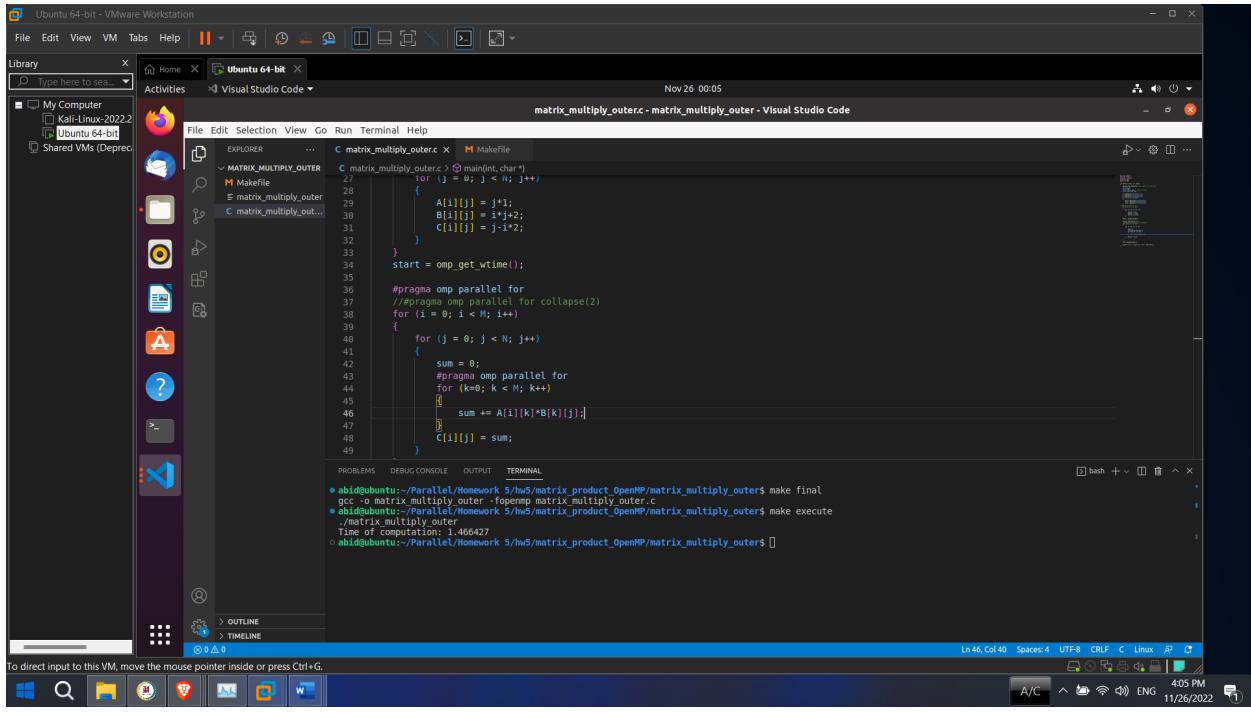
```
start = omp_get_wtime();

#pragma omp parallel for
//#pragma omp parallel for collapse(2)
for (i = 0; i < M; i++)
{
    for (j = 0; j < N; j++)
    {
        sum = 0;
        #pragma omp parallel for
        for (k=0; k < M; k++)
        {
            sum += A[i][k]*B[k][j];
        }
        C[i][j] = sum;
    }
}

end = omp_get_wtime();
```

Result:

```
1st time time we got 1.466427
2nd time time we got 1.351925
3rd time time we got 1.642867
4th time time we got 1.827936
```



2. Remove the pragma for the outer for loop and create a pragma for the middle for loop;

Solution:

```
start = omp_get_wtime();

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

end = omp_get_wtime();
```

Result:

1st time time we got 1.500284
 2nd time time we got 1.896121

3rd time time we got 2.022340
4th time time we got 2.365350

```
Ubuntu 64-bit - VMware Workstation
File Edit Selection View Go Run Terminal Help
Activities Ubuntu 64-bit Nov 25 23:32
matrix_multiply_middle.c - matrix_multiply_middle - Visual Studio Code
File Edit Selection View Go Run Terminal Help
EXPLORER ... MATRIX_MULTIPLY_MIDDLE C matrix_multiply_middle.c > main(int, char*)
MATRIX_MULTIPLY_MIDDLE C matrix_multiply_middle.c > main(int, char*)
31     | C[i][j] = j-1*i;
32     |
33 }
34 start = omp_get_wtime();
35
36 for (i = 0; i < M; i++)
37 {
38     #pragma omp parallel for
39     for (j = 0; j < N; j++)
40     {
41         sum = 0;
42         #pragma omp parallel for
43         for (k=0; k < M; k++)
44         {
45             sum += A[i][k]*B[k][j];
46         }
47         C[i][j] = sum;
48     }
49
50 end = omp_get_wtime();
51
52 printf("Time of computation: %f\n", end-start);
53
PROBLEMS DEBUG CONSOLE OUTPUT TERMINAL
● abid@ubuntu:~/Parallel/Homework 5/hw5/matrix_product_OpenMP/matrix_multiply_middle$ make final
● abid@ubuntu:~/Parallel/Homework 5/hw5/matrix_product_OpenMP/matrix_multiply_middle$ ./matrix_multiply_middle
Time of computation: 1.500284
● abid@ubuntu:~/Parallel/Homework 5/hw5/matrix_product_OpenMP/matrix_multiply_middle$ bash +v
To direct input to this VM, move the mouse pointer inside or press Ctrl+G.
A/C 3:32 PM 11/26/2022
```

3. Add the necessary pragma's to parallelize both the outer and middle for loops.

Solution:

```
start = omp_get_wtime();

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

end = omp_get_wtime();
```

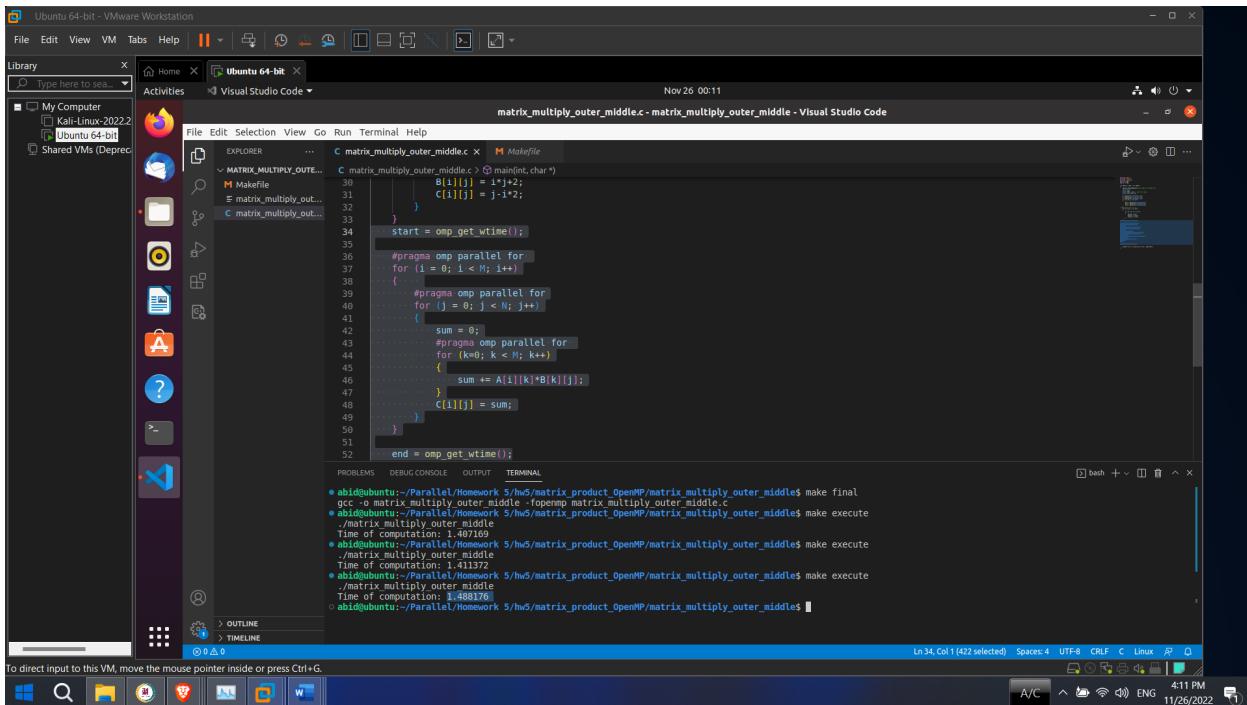
Result:

1st time we got 1.407169

2nd time we got 1.411372

3rd time we got 1.488176

4th time we got 1.536947



4. Add an assert function that compares the parallel computing result with serial computing result in order to make sure that you get correct result.

You need to collect timing data given one thread, four threads, eight threads, and 16 threads and two matrix sizes, 50x50 and 500x500. 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 should write a report and analyze the results.

Here are the conditions you should collect data for:

1. No parallelization at all (that is, the given program)

Solution:

Code

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

```

// Set two matrix sizes 500x500 and 50x50
#define M 500
#define N 500
// Threads set to 1,4,8,16
#define Max_threads 8

int main(int argc, char *argv) {
    omp_set_num_threads(Max_threads); //set number of threads here
    int i, j, k;
    double sum;
    double start, end; // used for timing
    double **A, **B, **C;
    /* FILE *fp; C provides a number of build-in function to perform basic file operations:
       fopen() - create a new file or open a existing file.
       fclose() - close a file */
    FILE *fp; // Create file pointers.
    A = malloc(M*sizeof(double *));
    B = malloc(M*sizeof(double *));
    C = malloc(M*sizeof(double *));
    /* After a call to freopen, it associates STDOUT to file file.txt, so whatever we write at STDOUT that
    goes inside file.txt */

    fp = freopen("file.txt", "w+", stdout);

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

```

```
// Closing the file using fclose() function.  
fclose(fp);  
}
```

Result

1 threads and two matrix sizes 500x500

```
#define M 500  
#define N 500  
#define Max_threads 1  
  
omp_set_num_threads(Max_threads)
```

Time of computation: 0.594810

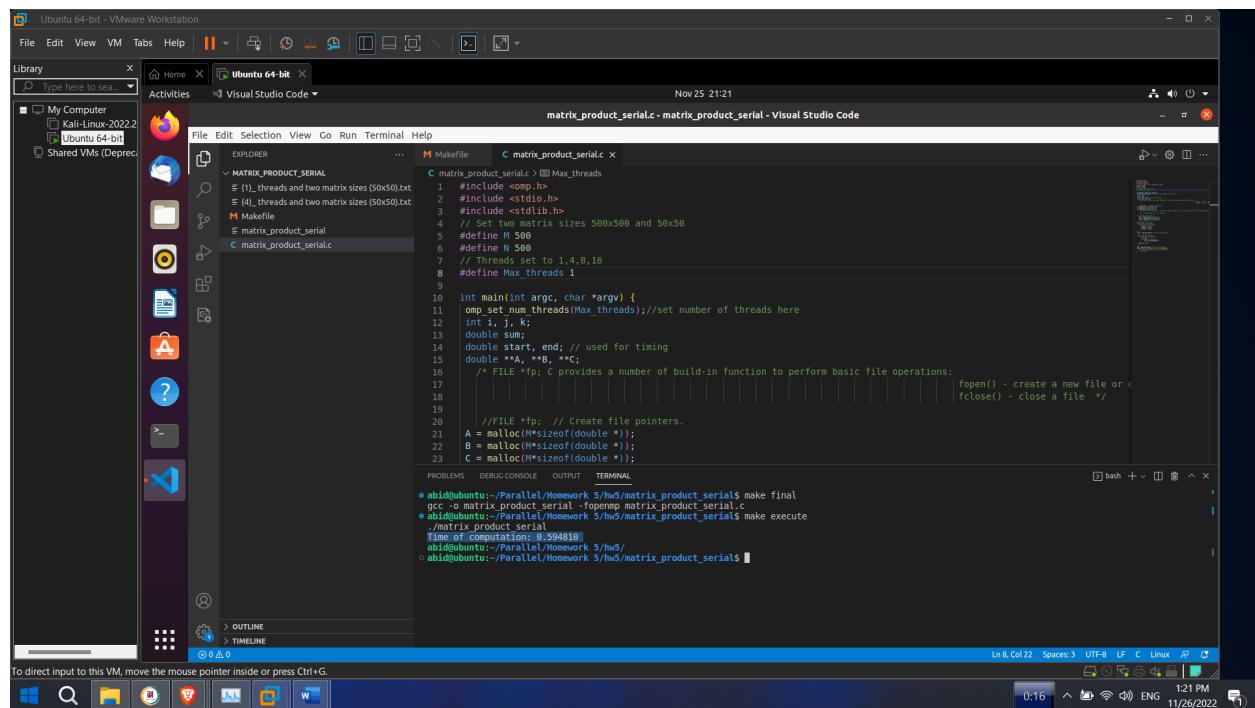


Fig : 1 threads and two matrix sizes 500x500

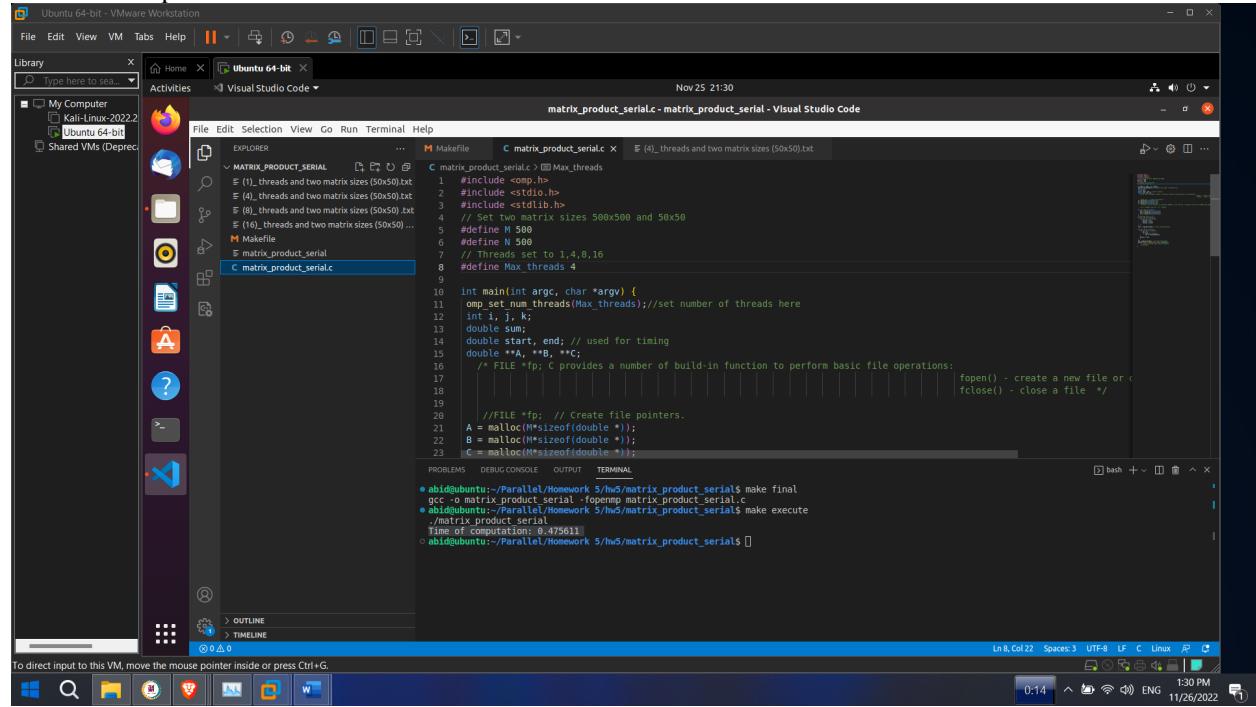
Result

4 threads and two matrix sizes 500x500

```
#define M 500  
#define N 500  
#define Max_threads 4
```

```
omp_set_num_threads(Max_threads)
```

Time of computation: 0.475611



```
matrix_product_serial.c - matrix_product_serial - Visual Studio Code
Nov 25 21:30
matrix_product_serial.c
matrix_product_serial.c (4)_threads and two matrix sizes (50x50).txt

File Edit Selection View Go Run Terminal Help
EXPLORER C matrix_product_serial.c E (4)_threads and two matrix sizes (50x50).txt
MATRIX_PRODUCT_SERIAL E (1)_threads and two matrix sizes (50x50).txt
E (4)_threads and two matrix sizes (50x50).txt
E (8)_threads and two matrix sizes (50x50).txt
E (16)_threads and two matrix sizes (50x50).txt
Makefile matrix_product_serial C matrix_product_serial.c

1 #include <omp.h>
2 #include <stdio.h>
3 #include <stdlib.h>
4 // Set two matrix sizes 500x500 and 50x50
5 #define M 500
6 #define N 500
7 // Threads set to 1,4,8,16
8 #define Max_threads 4
9
10 int main(int argc, char *argv) {
11     omp_set_num_threads(Max_threads); //set number of threads here
12     int i, j, k;
13     double sum;
14     double start, end; // used for timing
15     double *A, *B, *C;
16     /* FILE *fp; // Create file pointers.
17      fopen() - create a new file or open an existing one
18      | fclose() - close a file */
19
20     //FILE *fp;
21     A = malloc(M * sizeof(double *));
22     B = malloc(M * sizeof(double *));
23     C = malloc(M * sizeof(double *));
24
25     // Initialize matrix A
26     for (i = 0; i < M; i++) {
27         for (j = 0; j < M; j++) {
28             A[i][j] = (double)(i * j);
29         }
30     }
31
32     // Initialize matrix B
33     for (i = 0; i < M; i++) {
34         for (j = 0; j < N; j++) {
35             B[i][j] = (double)(i * j);
36         }
37     }
38
39     // Initialize matrix C
40     for (i = 0; i < M; i++) {
41         for (j = 0; j < N; j++) {
42             C[i][j] = 0.0;
43         }
44     }
45
46     // Start timer
47     start = omp_get_wtime();
48
49     // Compute matrix multiplication
50     #pragma omp parallel for
51     for (i = 0; i < M; i++) {
52         for (j = 0; j < N; j++) {
53             #pragma omp parallel for
54             for (k = 0; k < M; k++) {
55                 C[i][j] += A[i][k] * B[k][j];
56             }
57         }
58     }
59
60     // Stop timer
61     end = omp_get_wtime();
62
63     // Print result
64     for (i = 0; i < M; i++) {
65         for (j = 0; j < N; j++) {
66             printf("%f ", C[i][j]);
67         }
68         printf("\n");
69     }
70
71     // Free memory
72     free(A);
73     free(B);
74     free(C);
75
76     // Print time taken
77     printf("Time of computation: %f\n", end - start);
78
79     return 0;
80 }
```

To direct input to this VM, move the mouse pointer inside or press Ctrl+G.

PROBLEMS DEBUG CONSOLE OUTPUT TERMINAL

● abid@ubuntu:~/Parallel/Homework 5/n5/matrix_product_serial\$ make final
● gcc -o matrix_product_serial -fopenmp matrix_product_serial.c
● abid@ubuntu:~/Parallel/Homework 5/n5/matrix_product_serial\$ make execute
● matrix_product_serial
● Time of computation: 0.475611
● abid@ubuntu:~/Parallel/Homework 5/n5/matrix_product_serial\$ []

Ln 8, Col 22 Spaces:3 UTF-8 LF C Linux F C
0:14 1:30 PM ENG 11/26/2022

Fig : 4 threads and two matrix sizes 500x500

Result

8 threads and two matrix sizes 500x500

```
#define M 500
#define N 500
#define Max_threads 8

omp_set_num_threads(Max_threads)
```

Time of computation: 0.577183

The screenshot shows a dual-monitor setup. The left monitor displays a Windows desktop environment with icons for File Explorer, Task View, and Start. The right monitor displays an Ubuntu 64-bit VM in VMware Workstation. The VM window has a title bar "Ubuntu 64-bit" and shows a terminal session in Visual Studio Code. The terminal output is as follows:

```
Nov 25 21:33
matrix_product_serial.c - matrix_product_serial - Visual Studio Code

matrix_product_serial.c
1 //include <omp.h>
2 //include <stdio.h>
3 //include <stdlib.h>
4 // Set two matrix sizes 500x500 and 50x50
5 #define M 500
6 #define N 500
7 // Threads set to 1,4,8,16
8 #define Max_threads 8
9
10 int main(int argc, char *argv) {
11     omp_set_num_threads(Max_threads); //set number of threads here
12     int i, j, k;
13     double sum;
14     double start, end; // used for timing
15     double **A, **B, **C;
16     /* FILE *fp; C provides a number of build-in function to perform basic file operations:
17     fopen() - create a new file or open an existing one
18     fclose() - close a file */
19
20     //FILE *fp; // Create file pointers.
21     A = malloc(M*sizeof(double *));
22     B = malloc(M*sizeof(double *));
23     C = malloc(M*sizeof(double *));
24
25     for(i=0; i<M; i++)
26         for(j=0; j<M; j++)
27             A[i][j] = B[j][i] = C[i][j] = 1.0;
28
29     start = omp_get_wtime();
30
31     #pragma omp parallel for
32     for(i=0; i<M; i++)
33         for(j=0; j<M; j++)
34             for(k=0; k<N; k++)
35                 C[i][j] += A[i][k]*B[k][j];
36
37     end = omp_get_wtime();
38
39     printf("Time of execution %f\n", end - start);
40
41     free(A);
42     free(B);
43     free(C);
44
45     return 0;
46 }
```

Fig : 8 threads and two matrix sizes 50x50

Result

16 threads and two matrix sizes 500x500

```
#define M 500  
#define N 500  
#define Max_threads 16  
  
omp set num threads(Max_threads)
```

Time of computation: 0.578601

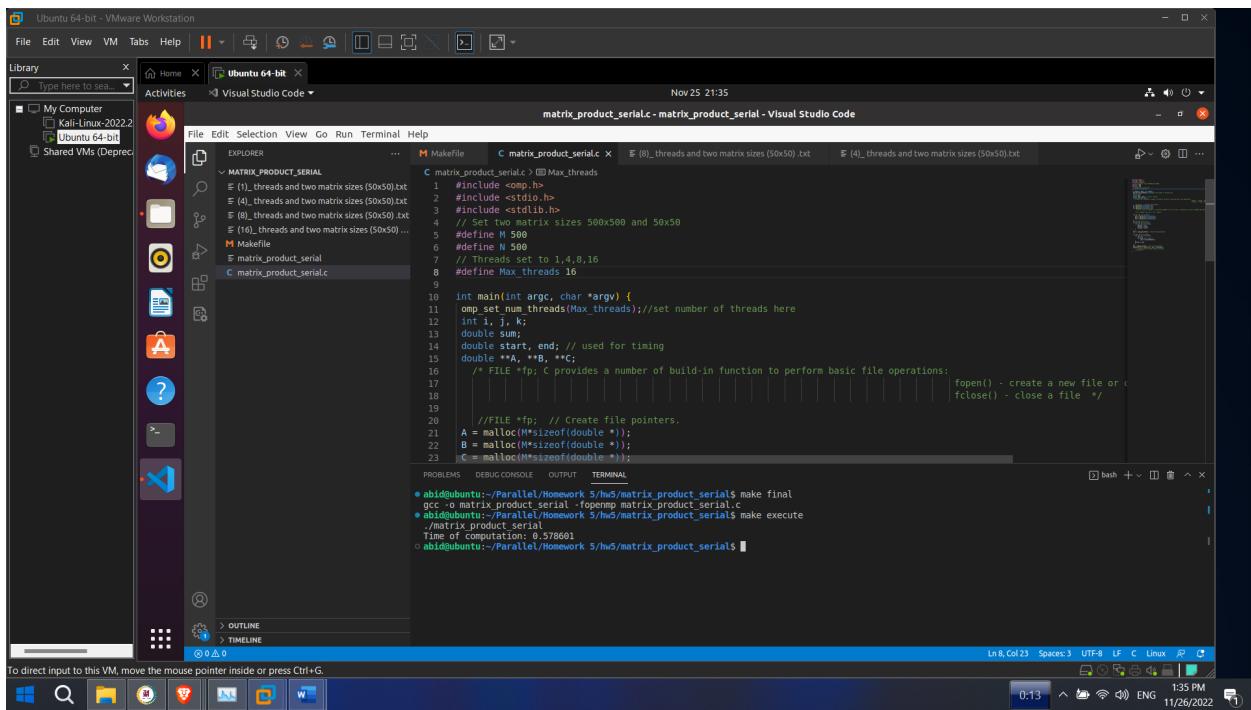


Fig : 16 threads and two matrix sizes 500x500

1 threads and two matrix sizes 50x50

```
#define M 50
#define N 50
#define Max_threads 1

omp_set_num_threads(Max_threads)
```

Time of computation: 0.001033

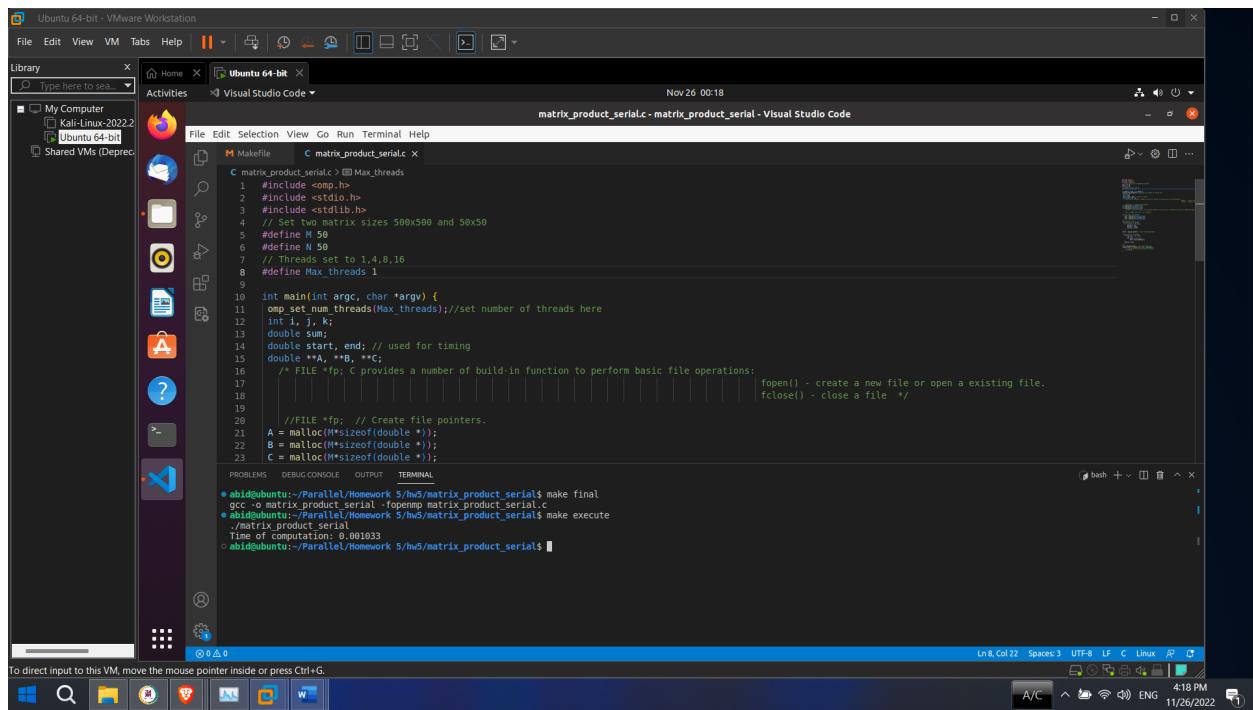


Fig : 1 threads and two matrix sizes 50x50

4 threads and two matrix sizes 50x50

```
#define M 50
#define N 50
#define Max_threads 4

omp_set_num_threads(Max_threads)
```

Time of computation: 0.001341

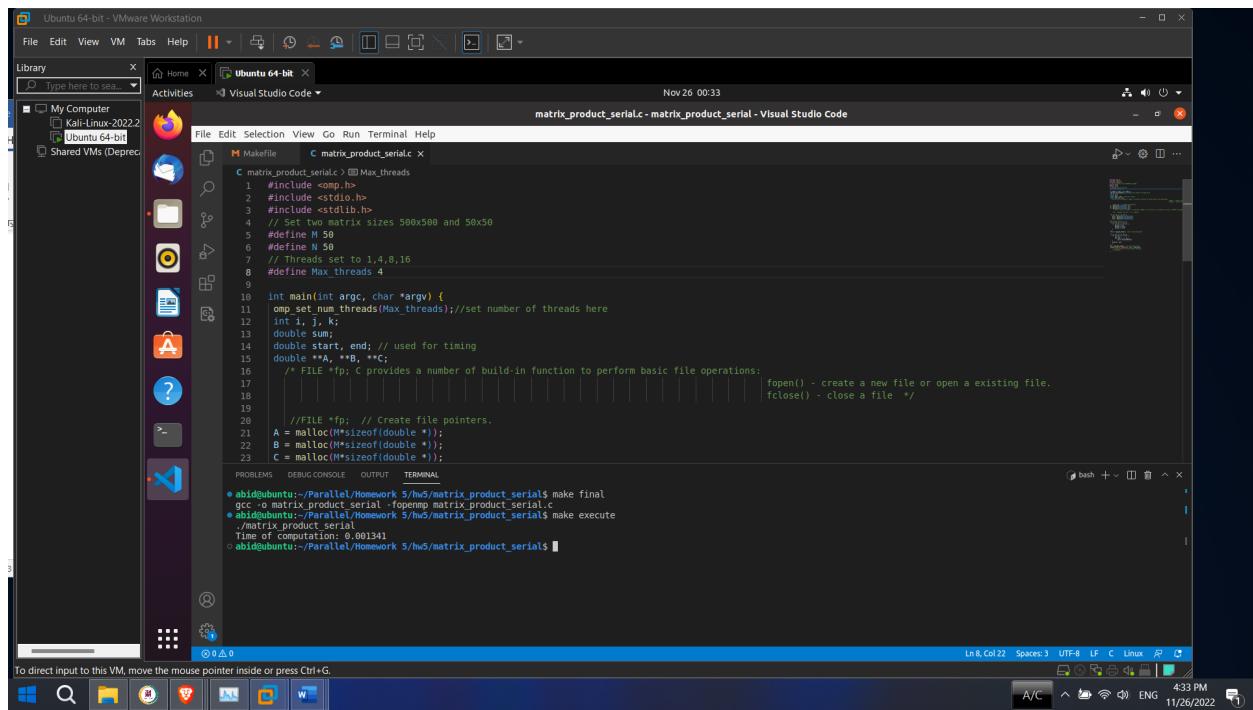


Fig : 4 threads and two matrix sizes 50x50

8 threads and two matrix sizes 50x50

```
#define M 50
#define N 50
#define Max_threads 8

omp_set_num_threads(Max_threads)
```

Time of computation: 0.001732

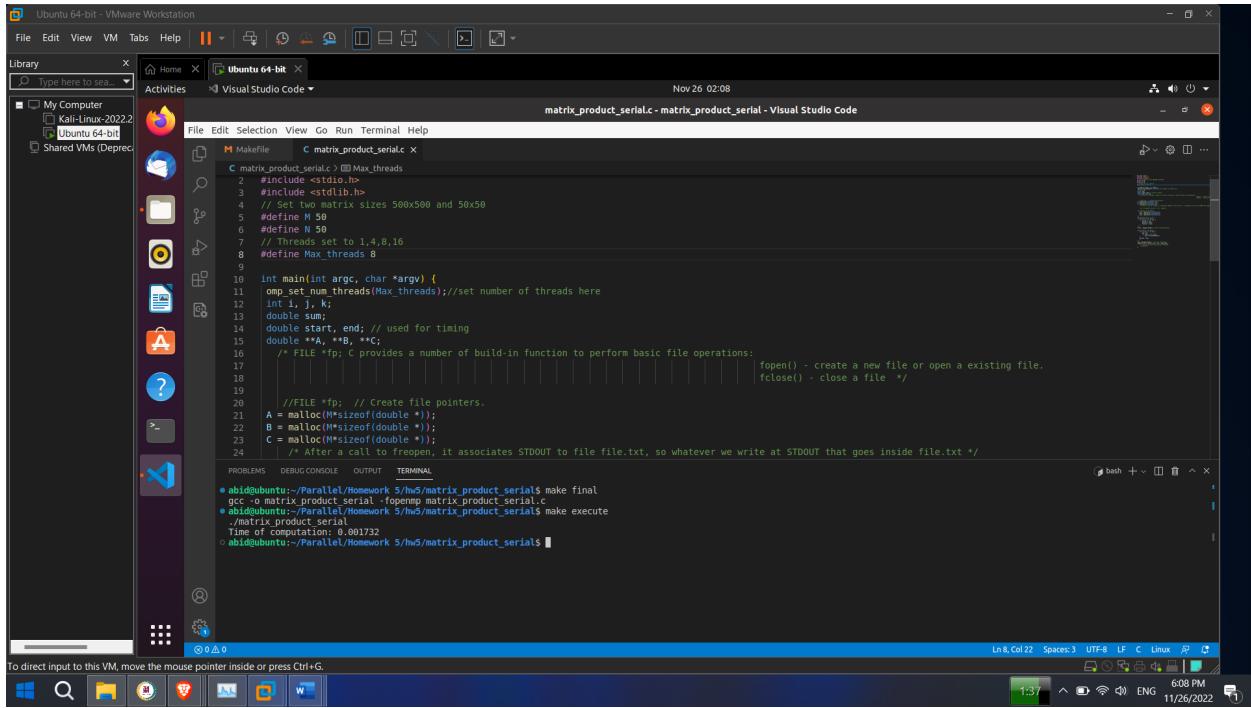


Fig : 8 threads and two matrix sizes 50x50

16 threads and two matrix sizes 50x50

```
#define M 50
#define N 50
#define Max_threads 16

omp_set_num_threads(Max_threads)
```

Time of computation: 0.001313

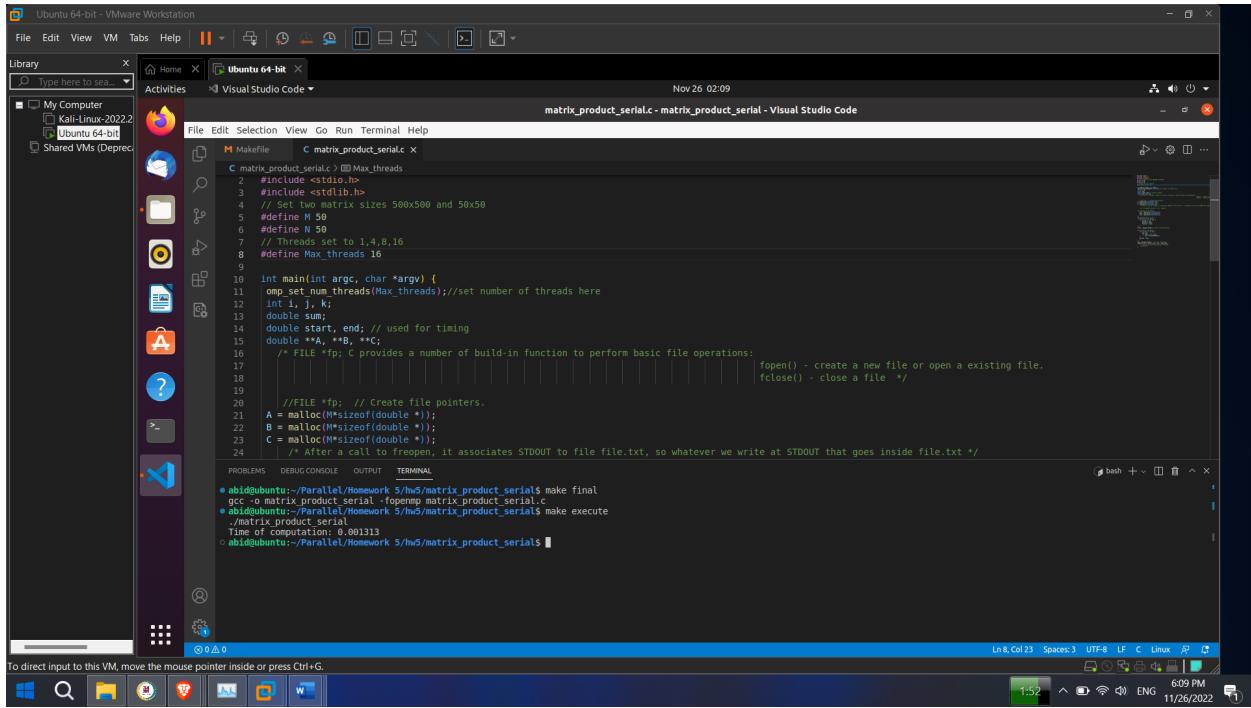


Fig : 16 threads and two matrix sizes 50x50

2. Parallelizing the loops as above with 1, 4, 8, and 16 threads using matrix sizes 50x50 and 500x500

Solution:

Code

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
// Set two matrix sizes 500x500 and 50x50
#define M 500
#define N 500

int main(int argc, char *argv)
{
    //set number of threads here. That is 1,4,8,16
    omp_set_num_threads(8);
    int i,j,k;
    double sum;
    double start, end, start_p, end_p;
    double **A, **B, **C;
    /* FILE *fp; C provides a number of build-in function to perform basic file operations:
       fopen() - create a new file or open a existing file.
       fclose() - close a file */
    FILE *fp; // Create file pointers.
    A= malloc(M*sizeof(double *));
    B= malloc(M*sizeof(double *));
    C= malloc(M*sizeof(double *));
    /* After a call to freopen, it associates STDOUT to file file.txt, so whatever we write at STDOUT that goes inside file.txt */

    //FILE *fp; // Create file pointers.
    A = malloc(M*sizeof(double *));
    B = malloc(M*sizeof(double *));
    C = malloc(M*sizeof(double *));
    /* After a call to freopen, it associates STDOUT to file file.txt, so whatever we write at STDOUT that goes inside file.txt */
    /* FILE *fp; C provides a number of build-in function to perform basic file operations:
       fopen() - create a new file or open a existing file.
       fclose() - close a file */
}
```

```

C= malloc(M*sizeof(double *));
/* After a call to freopen, it associates STDOUT to file file.txt, so whatever we write at STDOUT that
goes inside file.txt */

fp = freopen("file.txt", "w+", stdout);
for(i=0; i<M; i++)
{
    A[i]= malloc(N*sizeof(double));
    B[i]= malloc(N*sizeof(double));
    C[i]= malloc(N*sizeof(double));
}
//initialize matrix A, B, C
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;
    }
}
printf("MATRIX A:\n");
for(i=0; i<M; i++)
{
    for(j=0; j<N; j++)
    {
        printf("%f ",A[i][j]);
    }
    printf("\n");
}
printf("\n");
printf("MATRIX B:\n");
for(i=0; i<M; i++)
{
    for(j=0; j<N; j++)
    {
        printf("%f ",B[i][j]);
    }
    printf("\n");
}
printf("\n");
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();
printf("Serial Execution Time: %f\n\n", (end-start));
///////////////////////////////
start_p= omp_get_wtime();
#pragma omp parallel for
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_p= omp_get_wtime(); //end time measurement
printf("Parallel Execution Time: %f\n\n", (end_p-start_p));
printf("MATRIX MULTIPLICATION, MATRIX C:\n");
for(i=0; i<M; i++)
{
    for(j=0; j<N; j++)
    {
        printf("%f ",C[i][j]);
    }
    printf("\n");
}
// Closing the file using fclose() function.
fclose(fp);

return 0;
}

```

Result

1 threads and two matrix sizes 500x500

```
#define M 500
#define N 500
```

```
omp_set_num_threads(1);
```



(1)_threads and two
matrix sizes (500x500)

Serial Execution Time: 1.500378

Parallel Execution Time: 2.339876

Fig : 1 threads and two matrix sizes 500x500

Result

4 threads and two matrix sizes 500x500

```
#define M 500  
#define N 500
```

```
omp_set_num_threads(4);
```



(4) threads and two matrix sizes (500x500)

Serial Execution Time: 0.047728

Parallel Execution Time: Didn't get due to segmentation fault. Don't know the exact reason why the parallel part function couldn't get the value. I think there is a limitation of processor and core that could be the reason.

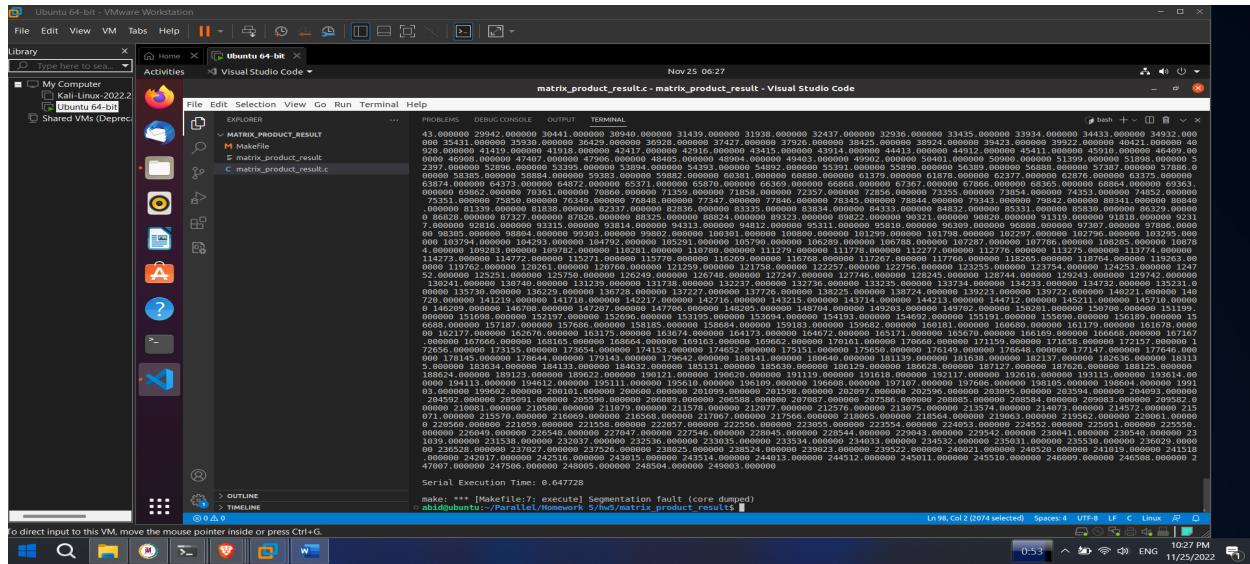


Fig : 4 threads and two matrix sizes 500x500

```
abid@Abid-/mnt/d/matrix_p > + -  
2.000000 43.000000 84.000000 125.000000 166.000000 207.000000 248.000000 289.000000 338.000000 371.000000 412.000000 453.000000 494.000000 535.000000 576.00  
00000 617.000000 658.000000 699.000000 748.000000 781.000000 822.000000 863.000000 904.000000 945.000000 986.000000 1027.000000 1068.000000 1109.000000 1158.  
00000 1191.000000 1232.000000 1273.000000 1314.000000 1355.000000 1396.000000 1437.000000 1478.000000 1519.000000 1560.000000 1601.000000 1642.000000 1683.  
00000 1724.000000 1765.000000 1866.000000 1847.000000 1888.000000 1929.000000 1970.000000 2011.000000  
2.000000 44.000000 86.000000 128.000000 170.000000 212.000000 254.000000 296.000000 338.000000 380.000000 422.000000 464.000000 506.000000 548.000000 590.00  
00000 632.000000 674.000000 716.000000 758.000000 806.000000 842.000000 884.000000 926.000000 968.000000 1010.000000 1052.000000 1094.000000 1136.000000 1178.  
00000 1196.000000 1237.000000 1278.000000 1318.000000 1359.000000 1399.000000 1439.000000 1479.000000 1519.000000 1556.000000 1596.000000 1648.000000 1682.000000 1724.  
00000 1766.000000 1808.000000 1850.000000 1892.000000 1924.000000 1976.000000 2018.000000 2060.000000  
2.000000 45.000000 88.000000 131.000000 174.000000 217.000000 268.000000 303.000000 346.000000 389.000000 432.000000 475.000000 518.000000 561.000000 604.00  
00000 647.000000 690.000000 733.000000 776.000000 819.000000 862.000000 905.000000 948.000000 991.000000 1034.000000 1077.000000 1120.000000 1163.000000 1206.  
00000 1249.000000 1292.000000 1335.000000 1378.000000 1421.000000 1464.000000 1507.000000 1550.000000 1593.000000 1636.000000 1679.000000 1722.000000 1765.  
.00000 1888.000000 1851.000000 1894.000000 1937.000000 1989.000000 2023.000000 2066.000000 2109.000000  
2.000000 46.000000 90.000000 134.000000 178.000000 222.000000 266.000000 310.000000 354.000000 398.000000 442.000000 486.000000 530.000000 574.000000 618.00  
00000 662.000000 706.000000 756.000000 794.000000 838.000000 882.000000 926.000000 970.000000 1014.000000 1058.000000 1102.000000 1146.000000 1190.000000 123.  
00000 1288.000000 1332.000000 1369.000000 1418.000000 1454.000000 1498.000000 1544.000000 1588.000000 1636.000000 1674.000000 1718.000000 1762.000000 180.  
00000 1856.000000 1897.000000 1937.000000 1977.000000 2019.000000 2059.000000 2114.000000 2158.000000  
2.000000 47.000000 92.000000 137.000000 182.000000 227.000000 272.000000 317.000000 362.000000 407.000000 452.000000 497.000000 542.000000 587.000000 632.00  
00000 677.000000 722.000000 767.000000 812.000000 857.000000 902.000000 947.000000 992.000000 1037.000000 1082.000000 1127.000000 1172.000000 1217.000000 126.  
00000 1307.000000 1352.000000 1397.000000 1424.000000 1487.000000 1532.000000 1577.000000 1622.000000 1667.000000 1712.000000 1757.000000 1802.000000 184.  
00000 1892.000000 1937.000000 1982.000000 2027.000000 2072.000000 2117.000000 2162.000000 2207.000000  
2.000000 48.000000 94.000000 140.000000 186.000000 232.000000 278.000000 324.000000 370.000000 416.000000 462.000000 508.000000 554.000000 600.000000 646.00  
00000 692.000000 738.000000 784.000000 830.000000 876.000000 922.000000 968.000000 1014.000000 1060.000000 1106.000000 1152.000000 1198.000000 1244.000000 128.  
00000 1336.000000 1389.000000 1426.000000 1471.000000 1519.000000 1566.000000 1603.000000 1658.000000 1704.000000 1750.000000 1796.000000 1842.000000 188.000000 192.  
00000 1896.000000 1942.000000 1989.000000 2036.000000 2083.000000 2130.000000 2178.000000 2226.000000  
2.000000 49.000000 95.000000 142.000000 188.000000 235.000000 281.000000 321.000000 378.000000 424.000000 472.000000 519.000000 566.000000 613.000000 656.00  
00000 707.000000 754.000000 801.000000 848.000000 895.000000 942.000000 989.000000 1036.000000 1083.000000 1130.000000 1177.000000 1224.000000 1271.000000 131.  
00000 1365.000000 1412.000000 1459.000000 1506.000000 1553.000000 1600.000000 1647.000000 1694.000000 1741.000000 1788.000000 1835.000000 1882.000000 192.  
00000 1899.000000 1977.000000 2023.000000 2070.000000 2117.000000 2164.000000 2211.000000 2258.000000 2305.000000  
2.000000 50.000000 98.000000 146.000000 194.000000 242.000000 290.000000 338.000000 386.000000 434.000000 482.000000 530.000000 578.000000 626.000000 674.00  
00000 722.000000 770.000000 818.000000 866.000000 914.000000 962.000000 1010.000000 1058.000000 1106.000000 1154.000000 1202.000000 1250.000000 1298.000000 1  
346.000000 1394.000000 1442.000000 1499.000000 1538.000000 1586.000000 1634.000000 1682.000000 1730.000000 1778.000000 1826.000000 1874.000000 1922.000000 1  
371.000000 1408.000000 1516.000000 1564.000000 1612.000000 1660.000000 1708.000000 1756.000000 1804.000000 1852.000000 1890.000000 1938.000000 1976.  
00000 1988.000000 2035.000000 2083.000000 2131.000000 2179.000000 2227.000000 2275.000000 2323.000000 2371.000000  
2.000000 51.000000 99.000000 147.000000 200.000000 249.000000 296.000000 343.000000 390.000000 438.000000 486.000000 534.000000 582.000000 630.000000 678.00  
00000 727.000000 784.000000 826.000000 884.000000 923.000000 982.000000 1021.000000 1068.000000 1119.000000 1178.000000 1227.000000 1276.000000 1325.000000 1374.  
00000 1423.000000 1474.000000 1521.000000 1570.000000 1619.000000 1668.000000 1717.000000 1766.000000 1815.000000 1864.000000 1913.000000 1962.000000  
2011.000000 2060.000000 2109.000000 2158.000000 2207.000000 2256.000000 2305.000000 2354.000000 2403.000000  
Serial Execution Time: 0.007074  
.Makefile: line 7: 13209 Segmentation fault      ./matrix_product_result
```

Fig (Extra) : 4 threads and two matrix sizes, 500x500

8 threads and two matrix sizes 500x500

```
#define M 500  
#define N 500
```

```
omp set num threads(8);
```



(8) threads and two matrix sizes (500x500)

Serial Execution Time: 3.235702

Parallel Execution Time: Didn't get due to segmentation fault. Don't know the exact reason why the parallel part function couldn't get the value. I think there is a limitation of processor and core that could be the reason.

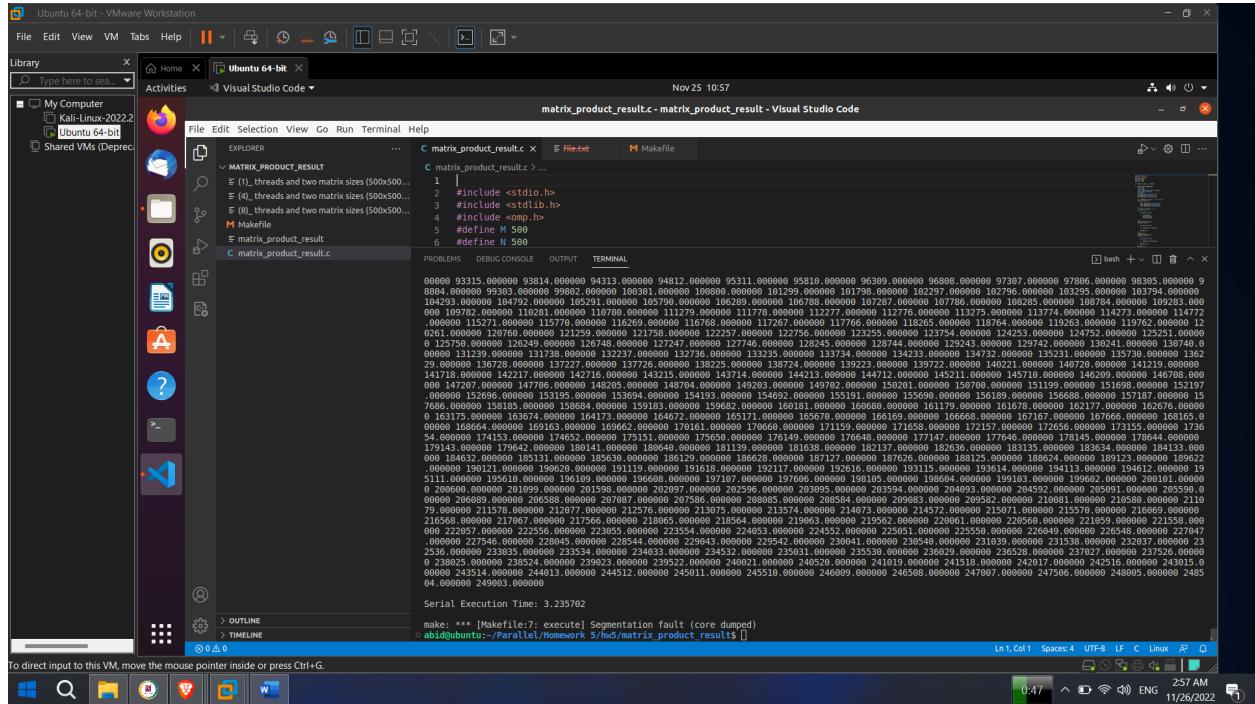


Fig : 8 threads and two matrix sizes 500x500

16 threads and two matrix sizes 500x500

```
#define M 500
#define N 500
```

```
omp_set_num_threads(16);
```



(16)_threads and two
matrix sizes (500x500)

Serial Execution Time: 2.924644

Parallel Execution Time: Didn't get due to segmentation fault. Don't know the exact reason why the parallel part function couldn't get the value. I think there is a limitation of processor and core that could be the reason.

Fig : 16 threads and two matrix sizes 500x500

1 threads and two matrix sizes 50x50

```
#define M 50  
#define N 50
```

```
omp_set_num_threads(1);
```



(1)_ threads and two
matrix sizes (50x50).tx

Serial Execution Time: 0.000455

Parallel Execution Time: 0.000538

The screenshot shows a Linux desktop environment (Ubuntu 64-bit) running in a VMware Workstation window. The desktop has a standard Unity interface with a dock at the bottom containing icons for various applications like Dash, Home, Activities, and Dash to Dock.

A terminal window titled "matrix_product_result.c - matrix_product_result - Visual Studio Code" is open, displaying the source code for a C program named `matrix_product_result.c`. The code implements matrix multiplication using OpenMP threads. It includes defines for thread counts (M_1, M_2, M_3, M_4), a main function that initializes arrays and prints them, and an OpenMP directive to set the number of threads.

Below the terminal, two execution times are shown:

- Serial Execution Time: 0.000577
- Parallel Execution Time: 0.000803

The terminal also displays the output of the program, which is a large matrix multiplication result. The output is as follows:

```
MATRIX MULTIPLICATION, MATRIX C:  
2.000000 49.000000 56.000000 143.000000 109.000000 237.000000 284.000000 331.000000 378.000000 425.000000 472.000000 513.000000 566.000000 613.000000  
666.000000 707.000000 756.000000 801.000000 845.000000 855.000000 942.000000 989.000000 1035.000000 1083.000000 1139.000000 1177.000000 1224.000000 1285.000000  
12.000000 1318.000000 1365.000000 1412.000000 1450.000000 1506.000000 1553.000000 1600.000000 1647.000000 1694.000000 1741.000000 1788.000000 1835.000000  
1882.000000 1929.000000 1976.000000 2023.000000 2070.000000 2117.000000 2164.000000 2211.000000 2258.000000 2305.000000  
2.000000 50.000000 98.000000 146.000000 194.000000 242.000000 290.000000 338.000000 386.000000 434.000000 482.000000 530.000000 578.000000 626.000000  
674.000000 722.000000 776.000000 818.000000 864.000000 914.000000 962.000000 1010.000000 1058.000000 1106.000000 1154.000000 1202.000000 1250.000000 1298.000000  
13.000000 1394.000000 1442.000000 1489.000000 1536.000000 1584.000000 1634.000000 1682.000000 1730.000000 1778.000000 1826.000000 1874.000000  
1922.000000 1970.000000 2018.000000 2066.000000 2114.000000 2162.000000 2210.000000 2258.000000 2306.000000 2354.000000  
2.000000 51.000000 109.000000 149.000000 198.000000 247.000000 295.000000 345.000000 394.000000 443.000000 492.000000 541.000000 590.000000 639.000000  
688.000000 737.000000 786.000000 835.000000 884.000000 933.000000 982.000000 1031.000000 1088.000000 1129.000000 1178.000000 1227.000000 1276.000000  
1325.000000 1374.000000 1423.000000 1472.000000 1521.000000 1570.000000 1619.000000 1668.000000 1717.000000 1766.000000 1815.000000 1864.000000 1913.000000  
1962.000000 2011.000000 2066.000000 2109.000000 2158.000000 2207.000000 2256.000000 2305.000000 2354.000000 2403.000000  
Serial Execution Time: 0.000577  
Parallel Execution Time: 0.000803
```

The terminal also displays the output of the program, which is a large matrix multiplication result. The output is as follows:

```
MATRIX MULTIPLICATION, MATRIX C:  
2.000000 49.000000 56.000000 143.000000 109.000000 237.000000 284.000000 331.000000 378.000000 425.000000 472.000000 513.000000 566.000000 613.000000  
666.000000 707.000000 756.000000 801.000000 845.000000 855.000000 942.000000 989.000000 1035.000000 1083.000000 1139.000000 1177.000000 1224.000000 1285.000000  
12.000000 1318.000000 1365.000000 1412.000000 1450.000000 1506.000000 1553.000000 1600.000000 1647.000000 1694.000000 1741.000000 1788.000000 1835.000000  
1882.000000 1929.000000 1976.000000 2023.000000 2070.000000 2117.000000 2164.000000 2211.000000 2258.000000 2305.000000  
2.000000 50.000000 98.000000 146.000000 194.000000 242.000000 290.000000 338.000000 386.000000 434.000000 482.000000 530.000000 578.000000 626.000000  
674.000000 722.000000 776.000000 818.000000 864.000000 914.000000 962.000000 1010.000000 1058.000000 1106.000000 1154.000000 1202.000000 1250.000000 1298.000000  
13.000000 1394.000000 1442.000000 1489.000000 1536.000000 1584.000000 1634.000000 1682.000000 1730.000000 1778.000000 1826.000000 1874.000000  
1922.000000 1970.000000 2018.000000 2066.000000 2114.000000 2162.000000 2210.000000 2258.000000 2306.000000 2354.000000  
2.000000 51.000000 109.000000 149.000000 198.000000 247.000000 295.000000 345.000000 394.000000 443.000000 492.000000 541.000000 590.000000 639.000000  
688.000000 737.000000 786.000000 835.000000 884.000000 933.000000 982.000000 1031.000000 1088.000000 1129.000000 1178.000000 1227.000000 1276.000000  
1325.000000 1374.000000 1423.000000 1472.000000 1521.000000 1570.000000 1619.000000 1668.000000 1717.000000 1766.000000 1815.000000 1864.000000 1913.000000  
1962.000000 2011.000000 2066.000000 2109.000000 2158.000000 2207.000000 2256.000000 2305.000000 2354.000000 2403.000000  
Serial Execution Time: 0.000577  
Parallel Execution Time: 0.000803
```

Fig : 1 threads and two matrix sizes 50x50

4 threads and two matrix sizes 50x50

```
#define M 50  
#define N 50
```

```
omp_set_num_threads(4);
```



(4)_ threads and two matrix sizes (50x50).tx

Serial Execution Time: 0.001381

Parallel Execution Time: Didn't get due to segmentation fault. Don't know the exact reason why the parallel part function couldn't get the value. I think there is a limitation of processor and core that could be the reason.

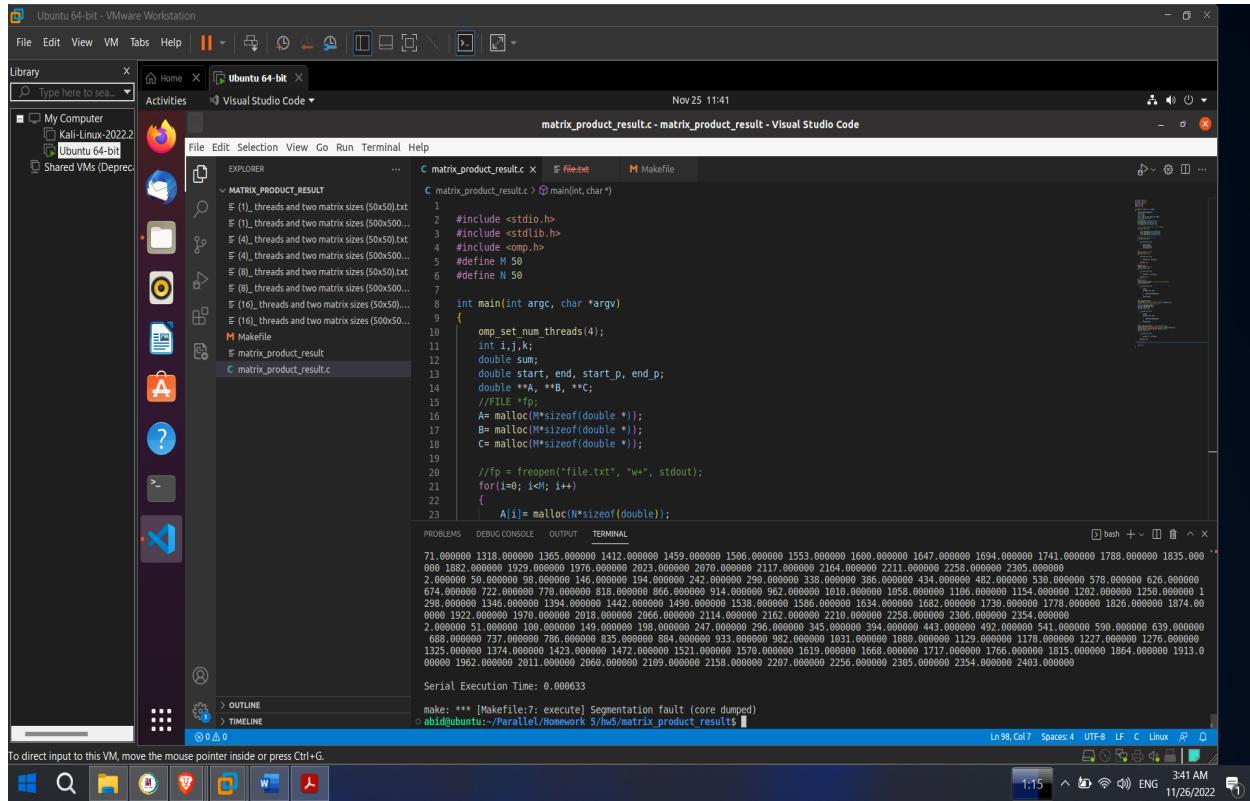


Fig : 4 threads and two matrix sizes 50x50

8 threads and two matrix sizes 50x50

```
#define M 50
#define N 50

omp_set_num_threads(8);
```

(8)_threads and two
matrix sizes (50x50).tx

Serial Execution Time: 0.000539

Parallel Execution Time: Didn't get due to segmentation fault. Don't know the exact reason why the parallel part function couldn't get the value. I think there is a limitation of processor and core that could be the reason.

Ubuntu 64-bit - VMware Workstation

File Edit View VM Tabs Help

Activities

Ubuntu 64-bit

Visual Studio Code

Nov 25 11:43

matrix_product_result.c - matrix_product_result - Visual Studio Code

File Edit Selection View Go Run Terminal Help

EXPLORER

MATRIX_PRODUCT_RESULT

matrix_product_result.c

matrix_product_result.h

Makefile

matrix_product_result.c

matrix_product_result.h

matrix_product_result.c

```
1 // (1), threads and two matrix sizes (50x50).txt
2 // (1), threads and two matrix sizes (500x500).txt
3 // (4), threads and two matrix sizes (50x50).txt
4 // (4), threads and two matrix sizes (500x500).txt
5 // (8), threads and two matrix sizes (50x50).txt
6 // (8), threads and two matrix sizes (500x500).txt
7 // (16), threads and two matrix sizes (50x50).txt
8 // (16), threads and two matrix sizes (500x500).txt
9 // (32), threads and two matrix sizes (50x50).txt
10 // (32), threads and two matrix sizes (500x500).txt
11 // (64), threads and two matrix sizes (50x50).txt
12 // (64), threads and two matrix sizes (500x500).txt
13 // (128), threads and two matrix sizes (50x50).txt
14 // (128), threads and two matrix sizes (500x500).txt
15 // (256), threads and two matrix sizes (50x50).txt
16 // (256), threads and two matrix sizes (500x500).txt
17 // (512), threads and two matrix sizes (50x50).txt
18 // (512), threads and two matrix sizes (500x500).txt
19 // (1024), threads and two matrix sizes (50x50).txt
20 // (1024), threads and two matrix sizes (500x500).txt
21 // (2048), threads and two matrix sizes (50x50).txt
22 // (2048), threads and two matrix sizes (500x500).txt
23 A[1]= malloc(N*sizeof(double));
```

PROBLEMS DEBUG CONSOLE OUTPUT TERMINAL

```
71.000000 1318.000000 1365.000000 1412.000000 1459.000000 1506.000000 1553.000000 1600.000000 1647.000000 1694.000000 1741.000000 1788.000000 1835.000000
000 1882.000000 1929.000000 1976.000000 2023.000000 2070.000000 2117.000000 2164.000000 2211.000000 2258.000000 2305.000000
2.000000 50.000000 98.000000 146.000000 194.000000 242.000000 290.000000 338.000000 386.000000 434.000000 482.000000 530.000000 578.000000
674.000000 722.000000 770.000000 818.000000 866.000000 914.000000 962.000000 1018.000000 1056.000000 1104.000000 1154.000000 1202.000000 1250.000000 1298.000000 1346.000000 1394.000000 1442.000000 1490.000000 1538.000000 1586.000000 1634.000000 1682.000000 1730.000000 1778.000000 1826.000000 1874.000000
0000 1922.000000 1970.000000 2066.000000 2114.000000 2162.000000 2210.000000 2258.000000 2306.000000 2354.000000
2.000000 51.000000 100.000000 149.000000 198.000000 247.000000 296.000000 345.000000 394.000000 443.000000 492.000000 541.000000 590.000000
639.000000 688.000000 737.000000 786.000000 835.000000 884.000000 933.000000 982.000000 1031.000000 1080.000000 1129.000000 1177.000000 1227.000000 1276.000000
1325.000000 1374.000000 1423.000000 1472.000000 1521.000000 1570.000000 1619.000000 1668.000000 1697.000000 1717.000000 1765.000000 1815.000000 1864.000000 1913.0
0000 1962.000000 2011.000000 2066.000000 2109.000000 2158.000000 2207.000000 2256.000000 2305.000000 2354.000000 2403.000000
Serial Execution Time: 0.00396
```

make: *** [Makefile:7: execute] Segmentation fault (core dumped)

abid@ubuntub:~/Parallel/Homework 5/Hw5/matrix product results\$

Ln Col 26 Space 4 UTF-8 LF C Linux JP

To direct input to this VM, move the mouse pointer inside or press Ctrl+G.

Fig : 8 threads and two matrix sizes 50x50

16 threads and two matrix sizes 50x50

```
#define M 50  
#define N 50
```

```
omp_set_num_threads(16);
```



(16)_threads and two matrix sizes (50x50).tx

Serial Execution Time: 0.001175

Parallel Execution Time: Didn't get due to segmentation fault. Don't know the exact reason why the parallel part function couldn't get the value. I think there is a limitation of processor and core that could be the reason.

Fig : 16 threads and two matrix sizes 50x50