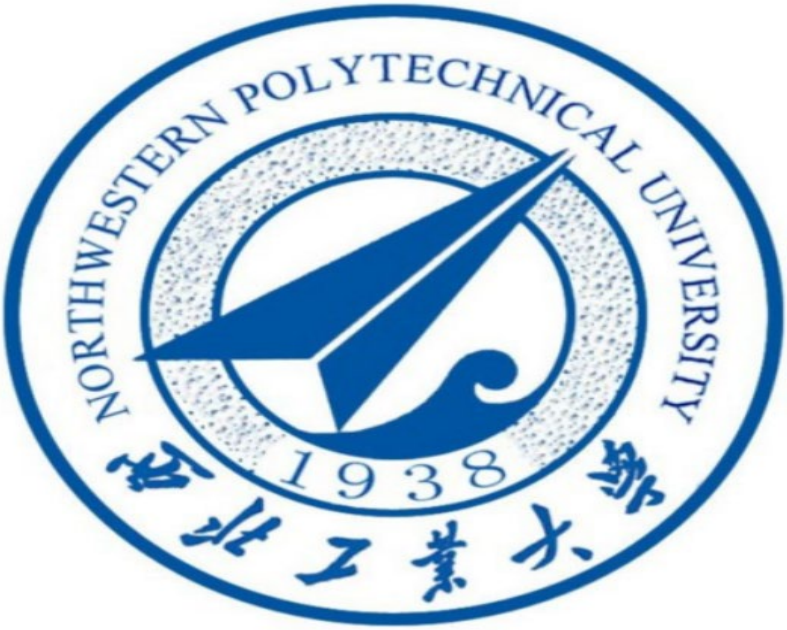


Parallel Programming

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

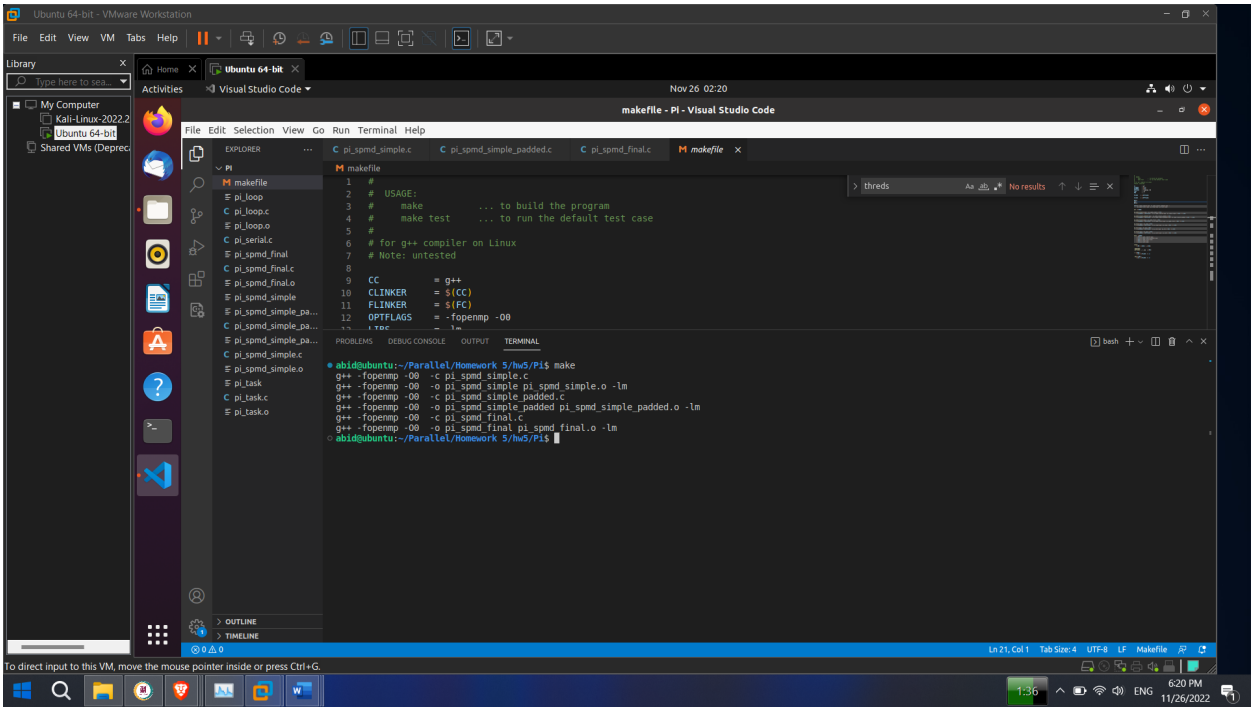


Fig : Make

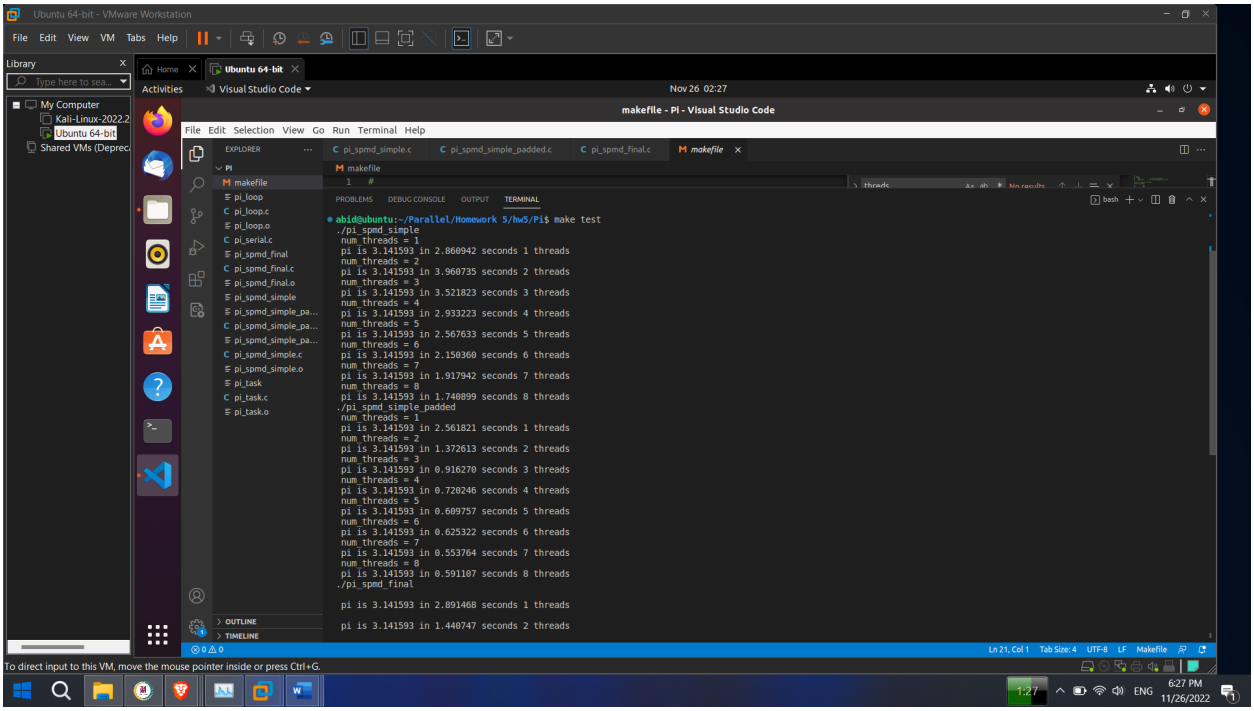


Fig : Make Test(1)

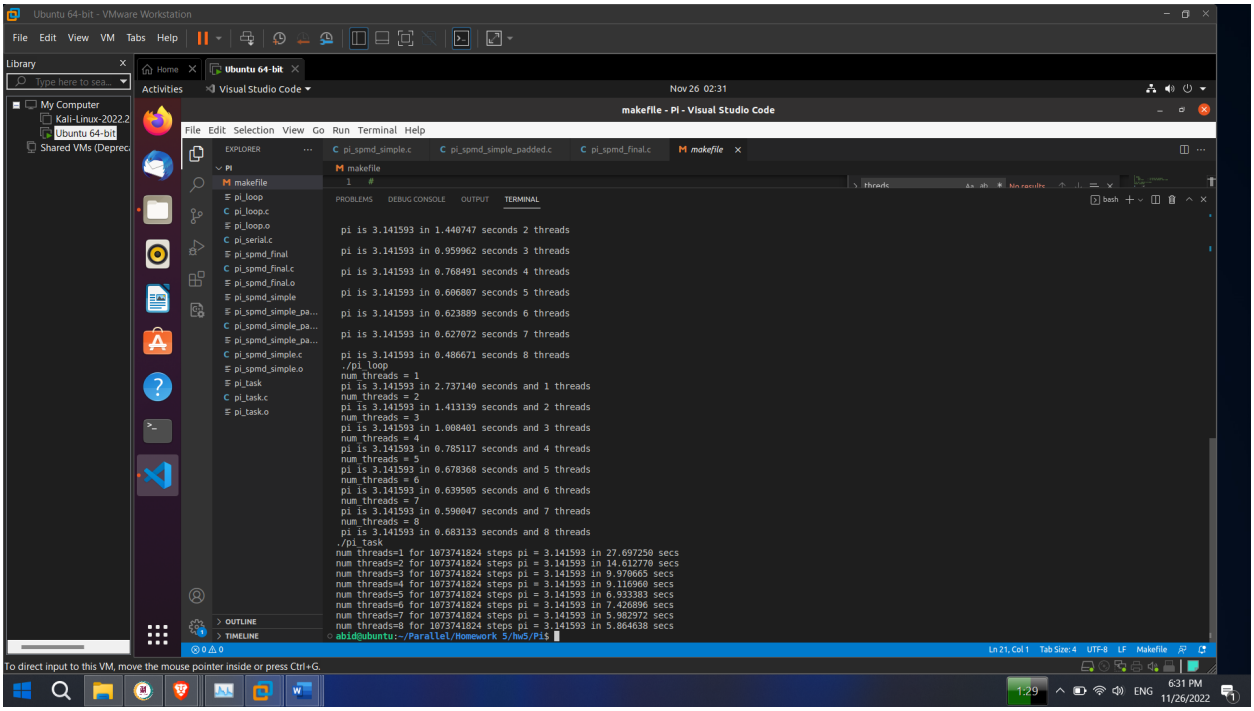


Fig : Make Test(2)

Hybrid

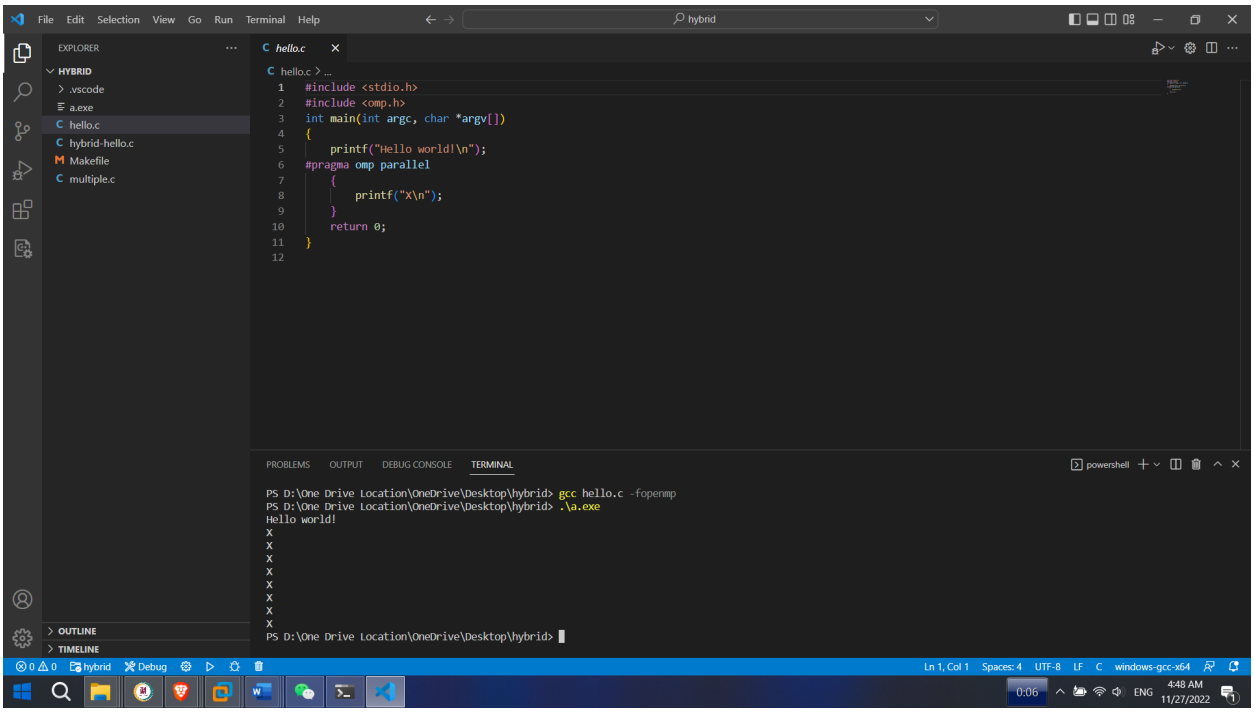


Fig : Hello.c

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.

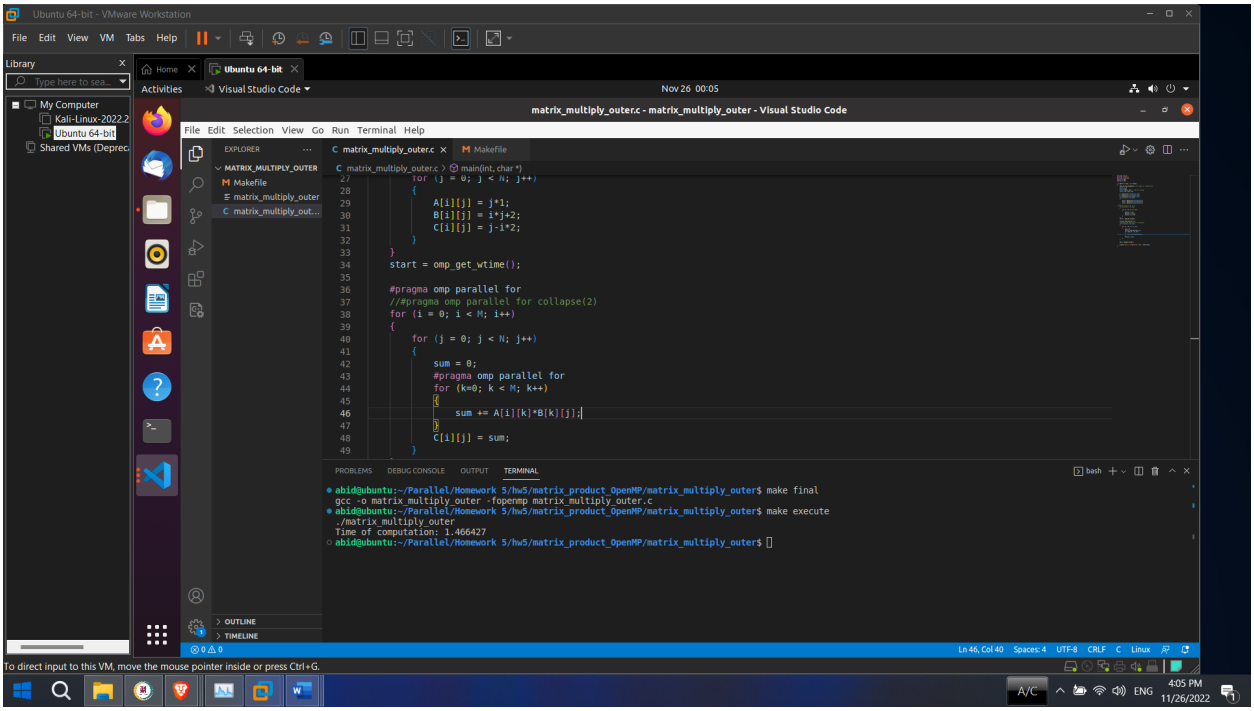
Provided that our outer loop has enough to keep the cores busy, it's the best place to go - especially if the amount of work done in each loop body is relatively consistent.

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;

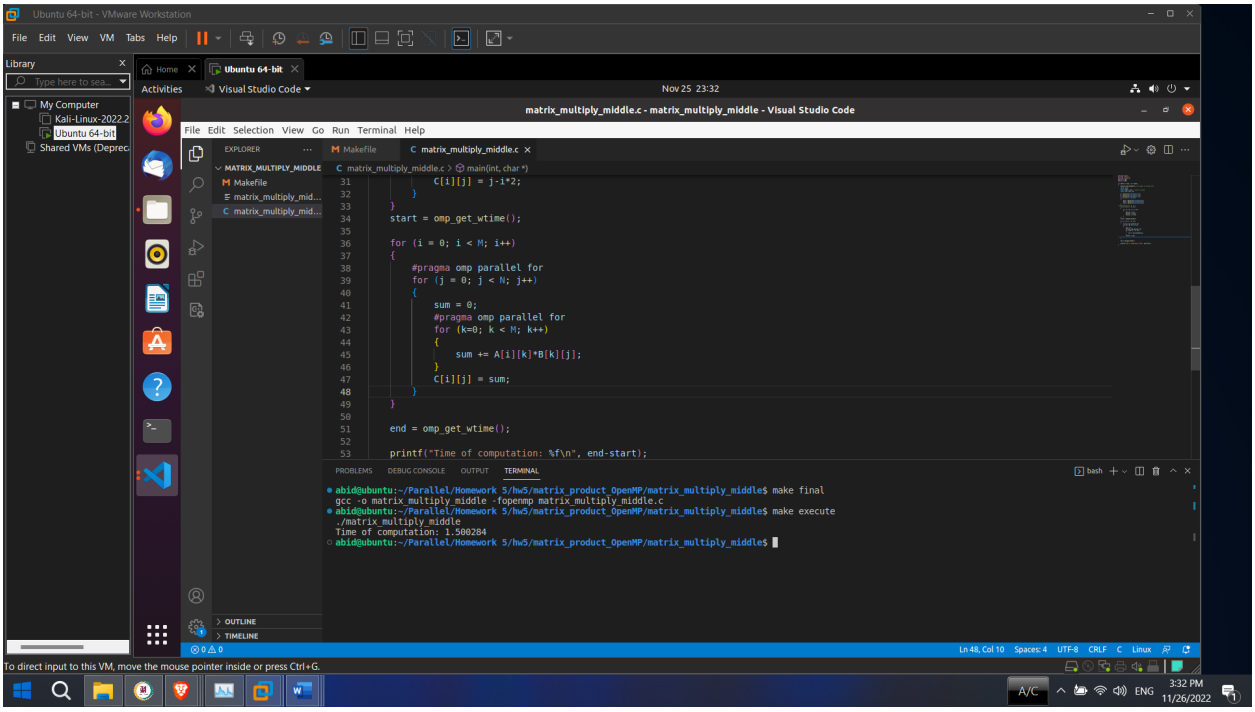
Solution:

```
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



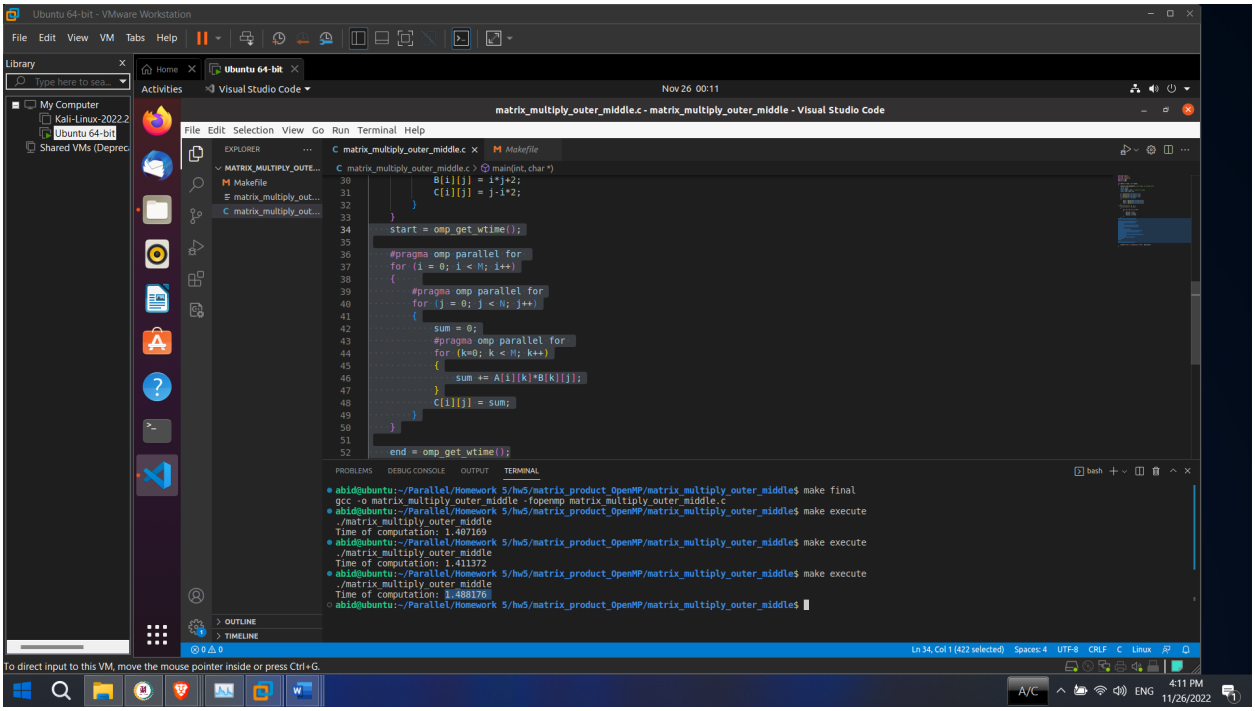
3. Add the necessary pragma's to parallelize both the outer and middle for loops.

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 time we got 1.407169
2nd time time we got 1.411372
3rd time time we got 1.488176
4th time 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

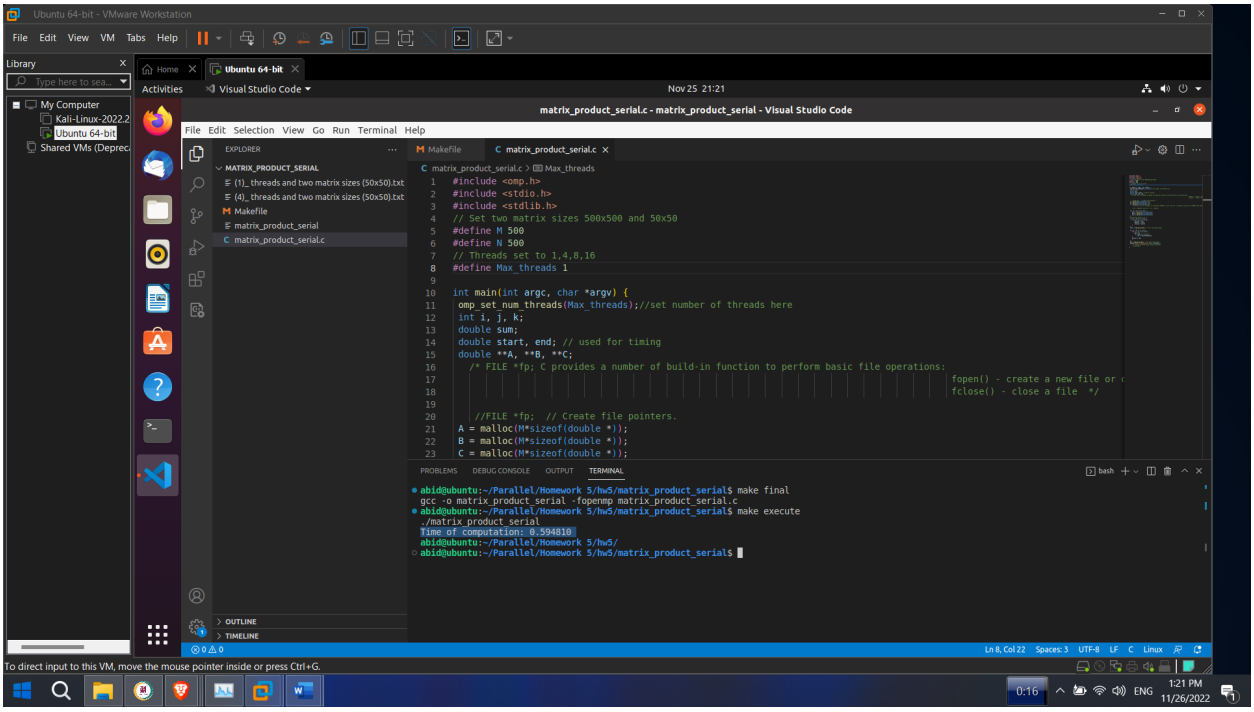


Fig : 1 threads and two matrix sizes 500x500

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

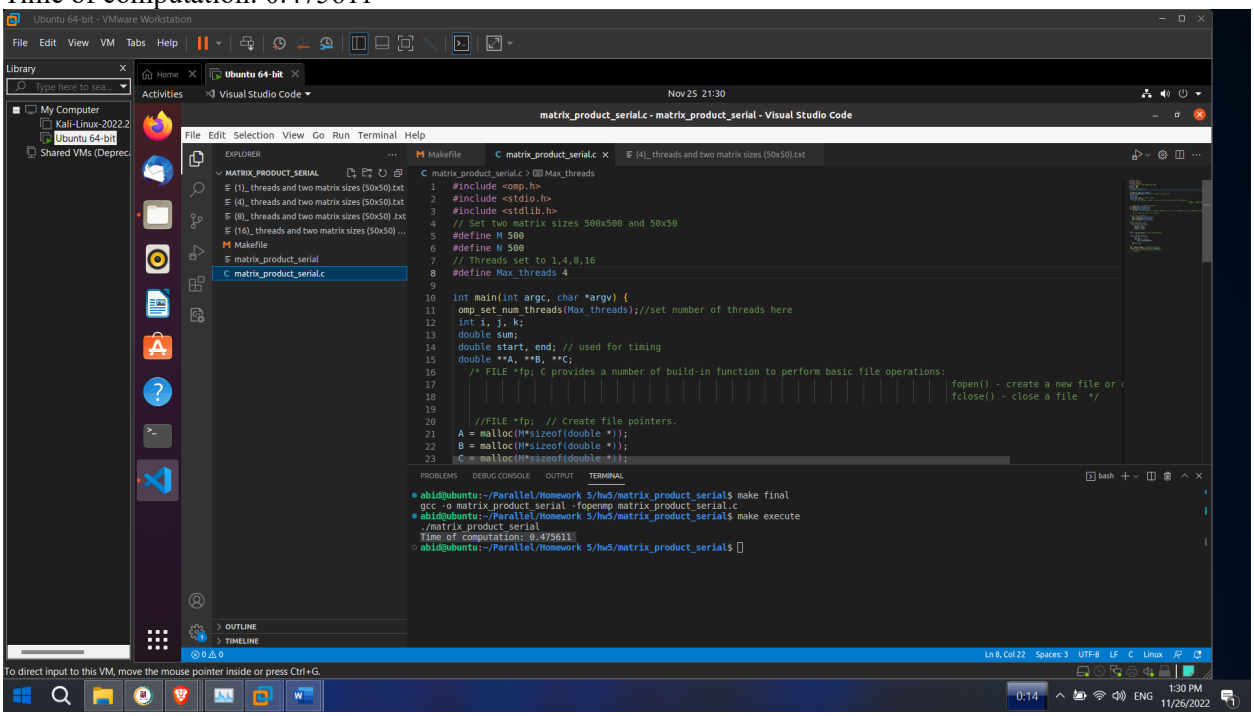


Fig : 4 threads and two matrix sizes 500x500

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

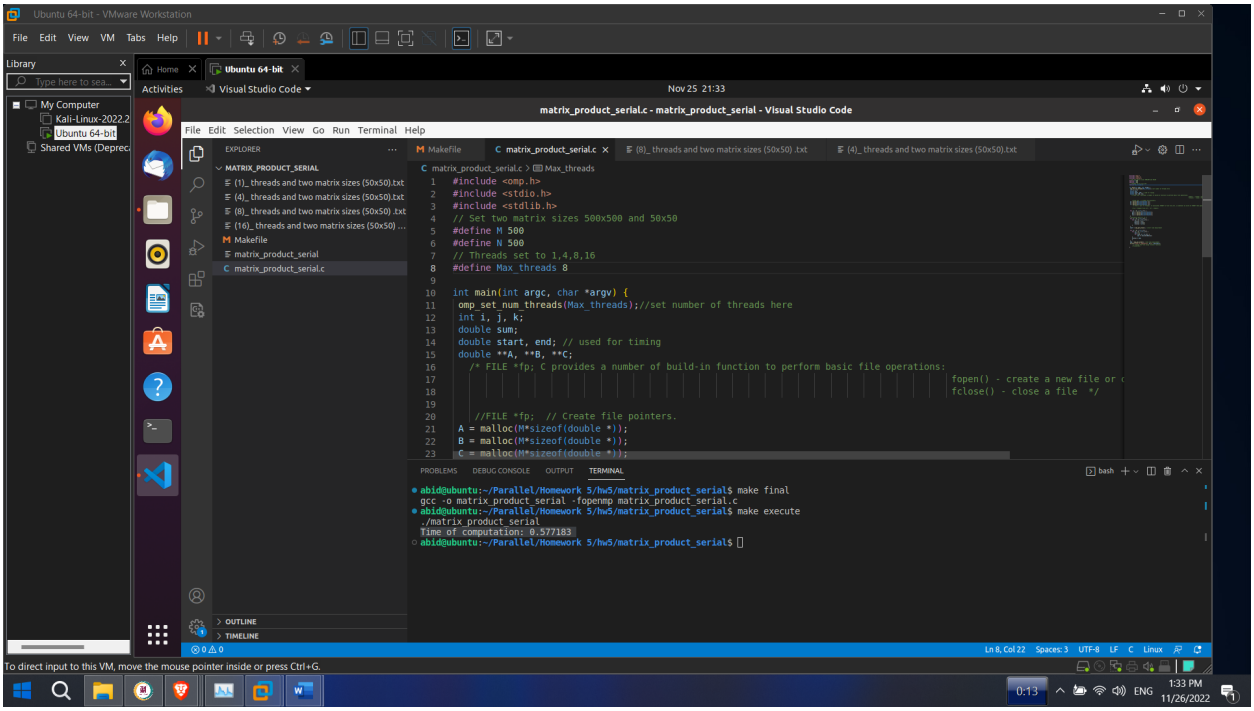


Fig : 8 threads and two matrix sizes 50x50

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

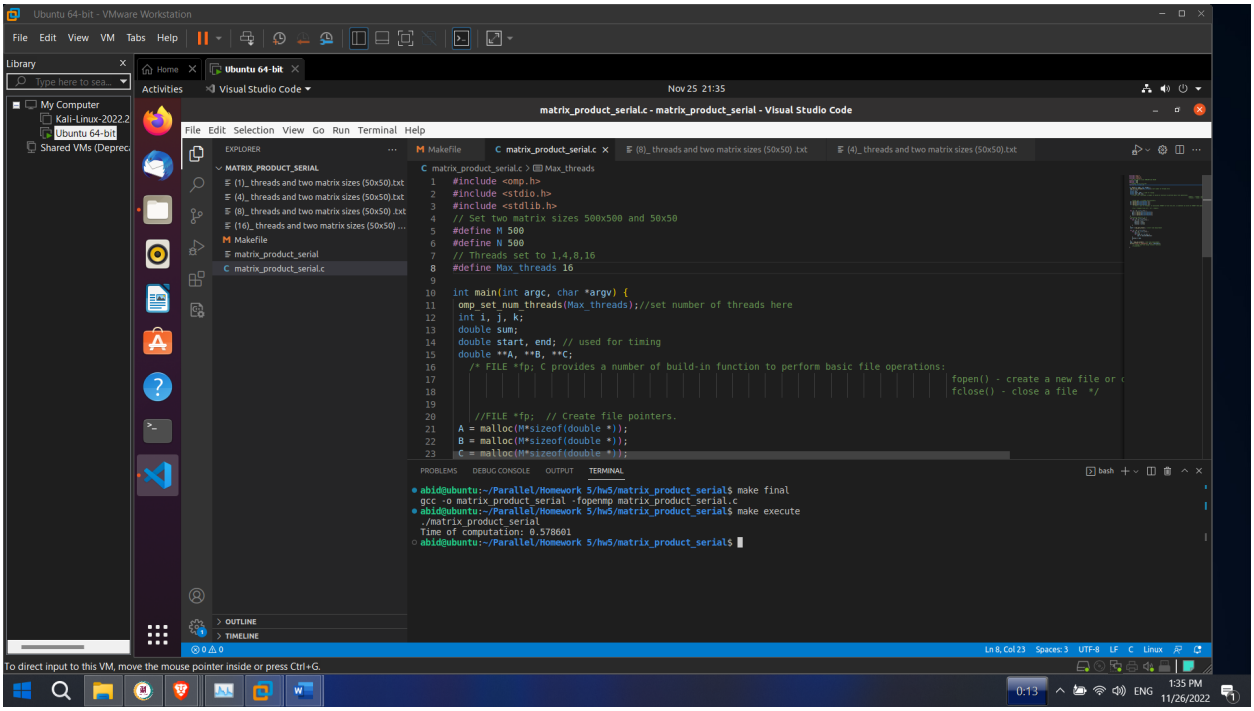


Fig : 16 threads and two matrix sizes 500x500

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

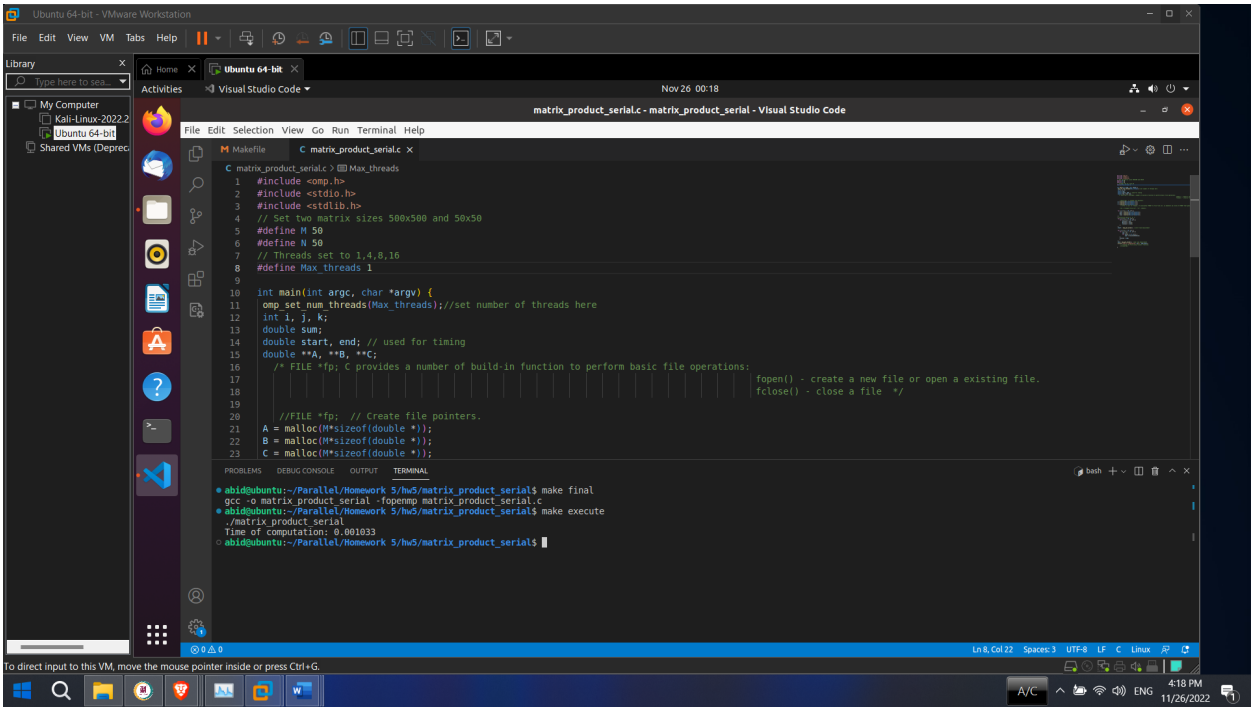


Fig : 1 threads and two matrix sizes 50x50

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

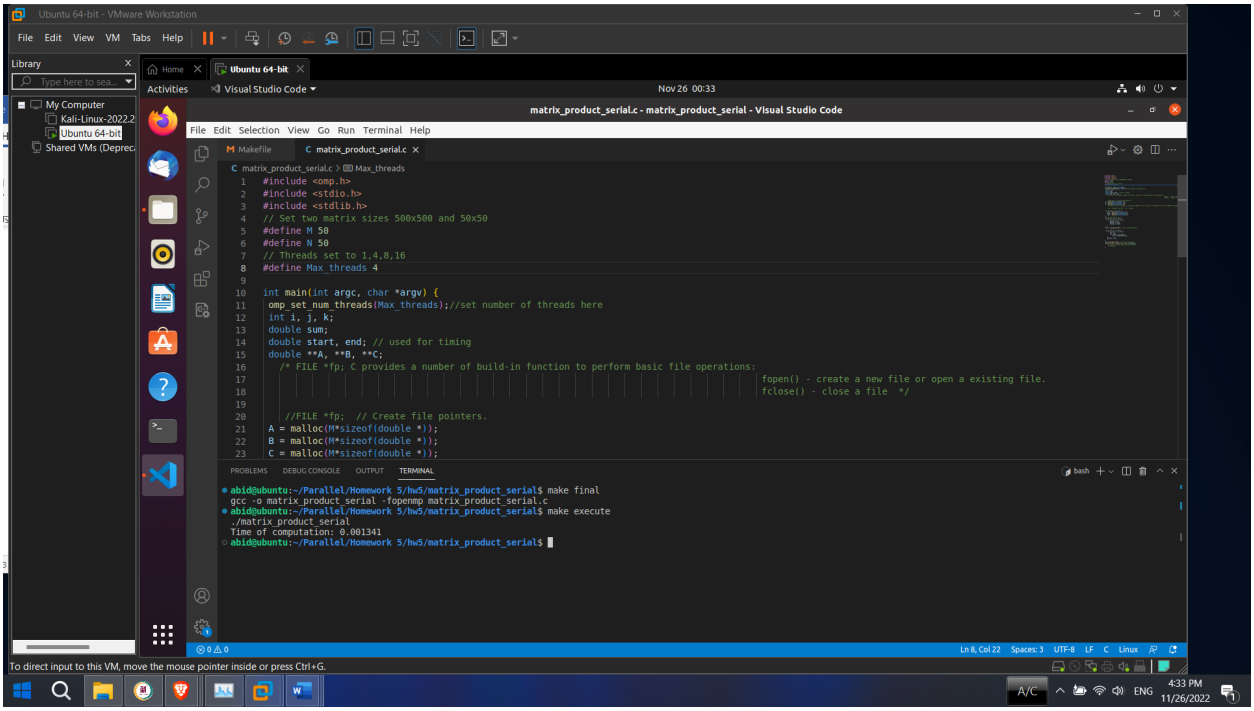


Fig : 4 threads and two matrix sizes 50x50

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

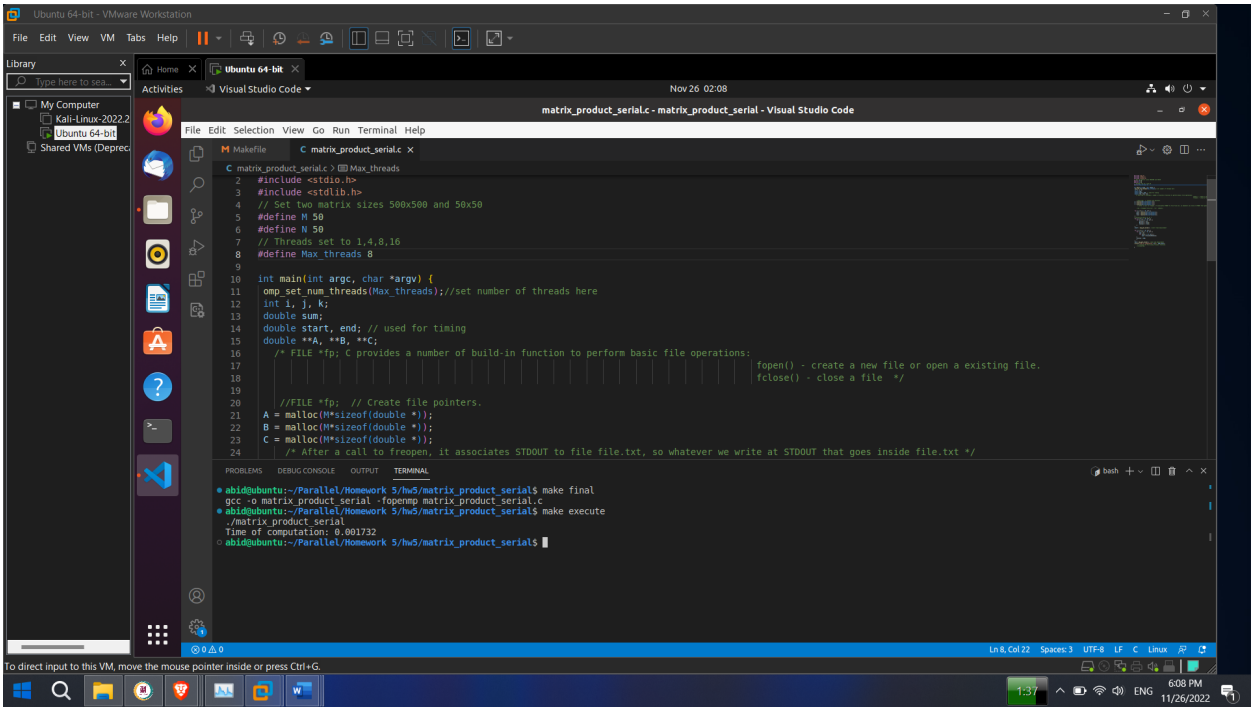


Fig : 8 threads and two matrix sizes 50x50

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

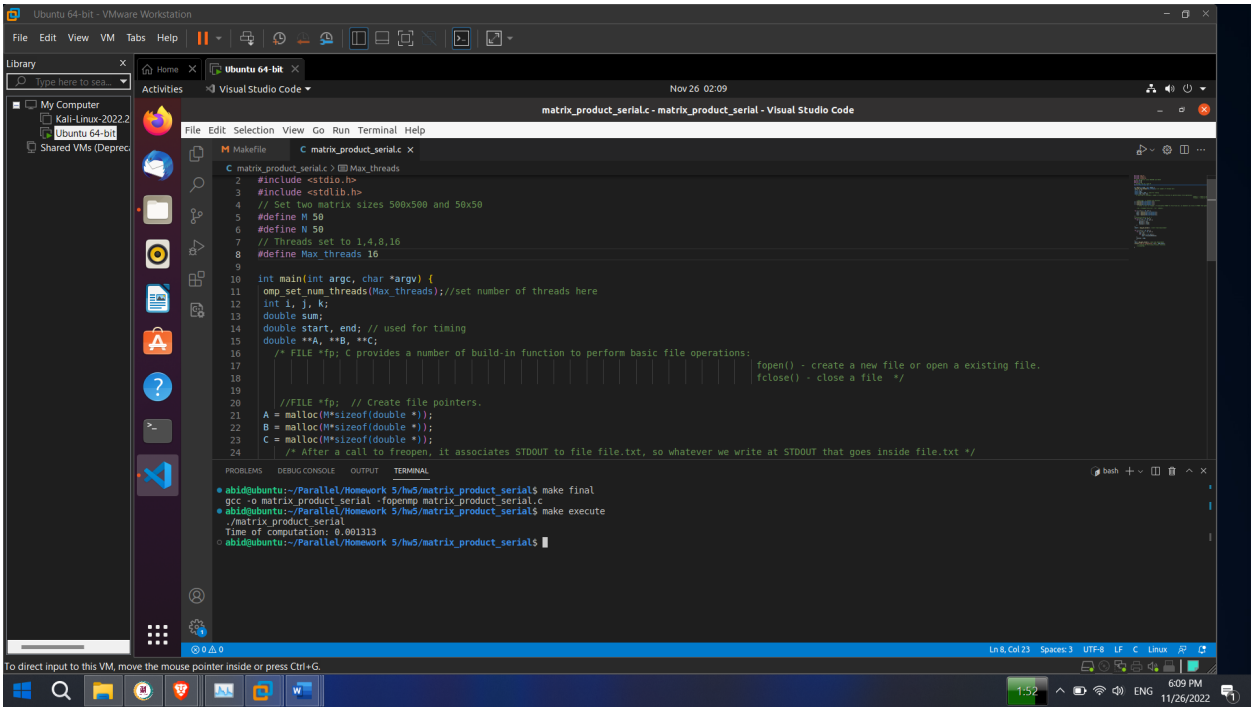


Fig : 16 threads and two matrix sizes 50x50

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 */

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

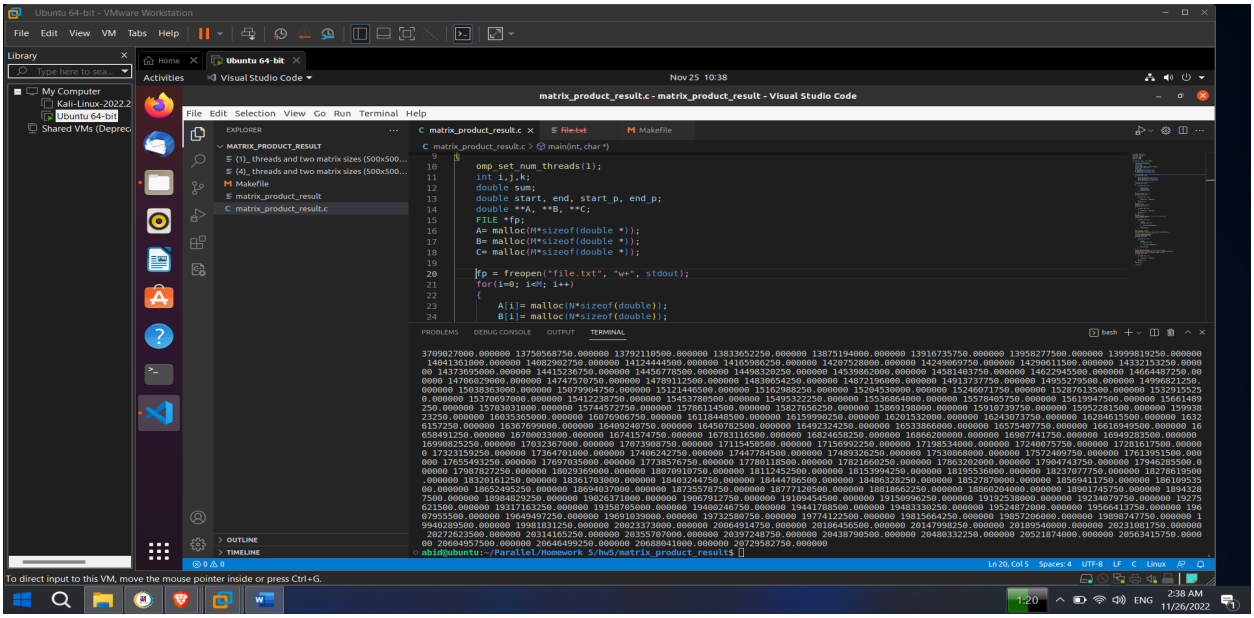


Fig : 1 threads and two matrix sizes 500x500

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.

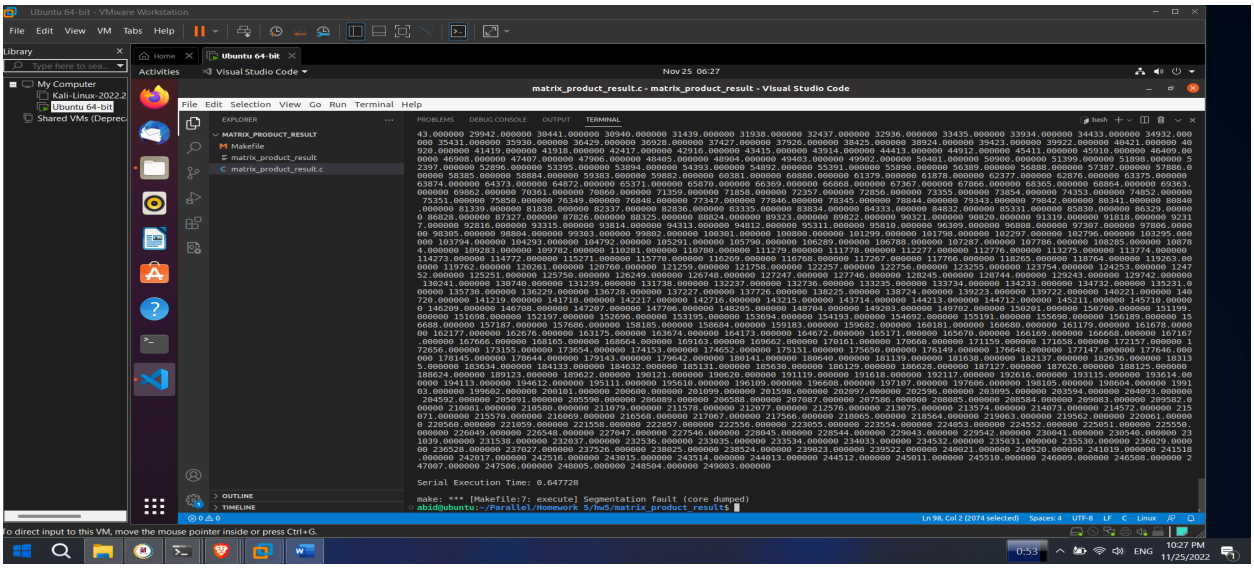


Fig : 4 threads and two matrix sizes 500x500

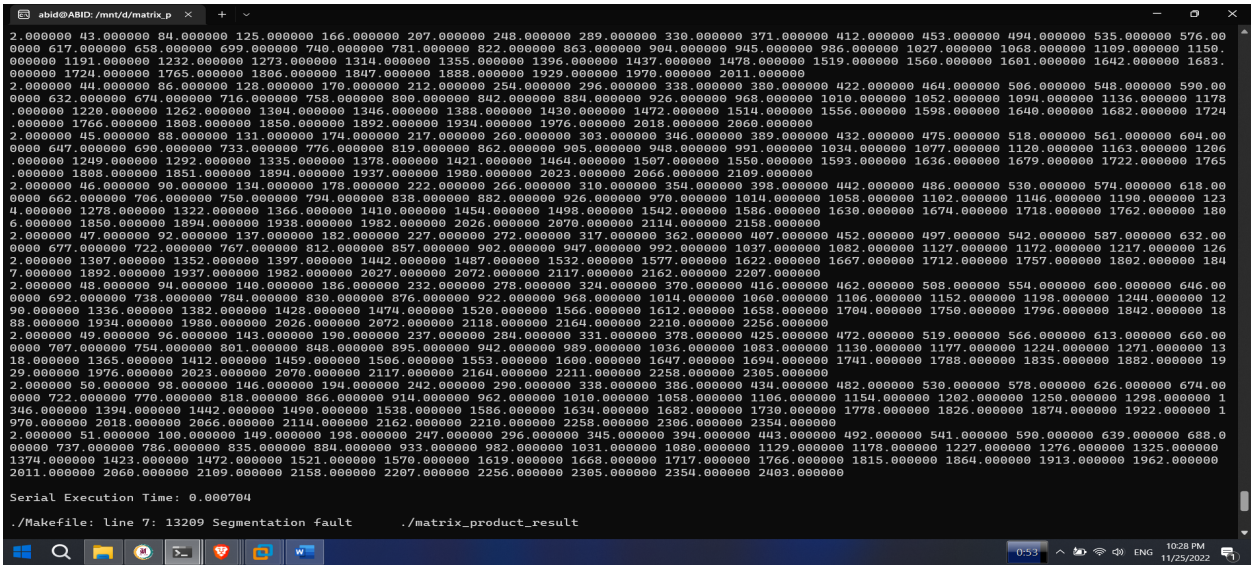


Fig (Extra) : 4 threads and two matrix sizes, 500x500

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.

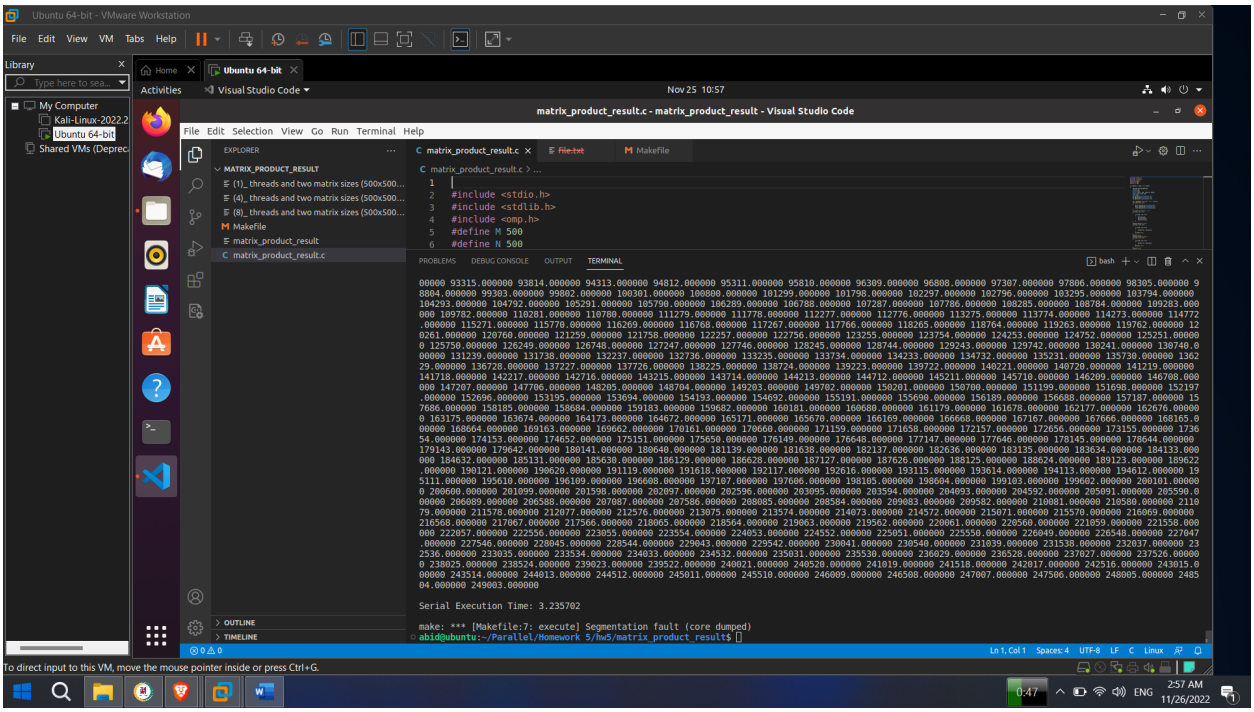


Fig : 8 threads and two matrix sizes 500x500

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.

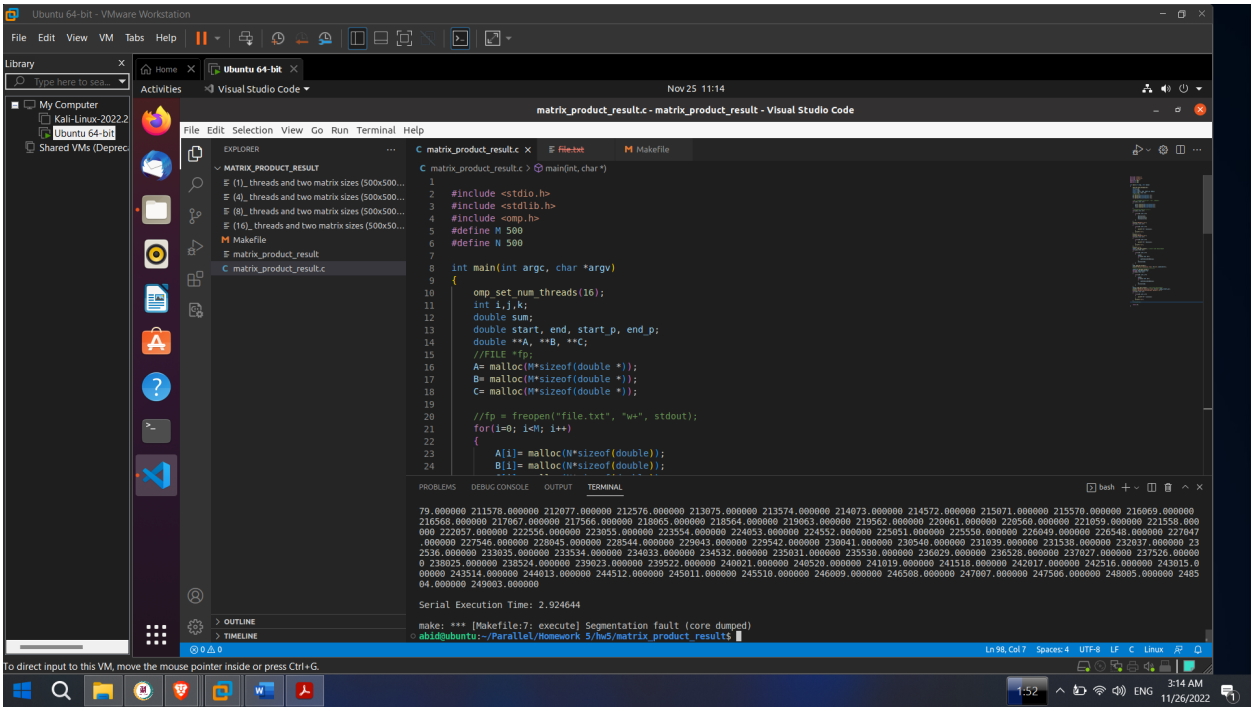


Fig : 16 threads and two matrix sizes 500x500

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

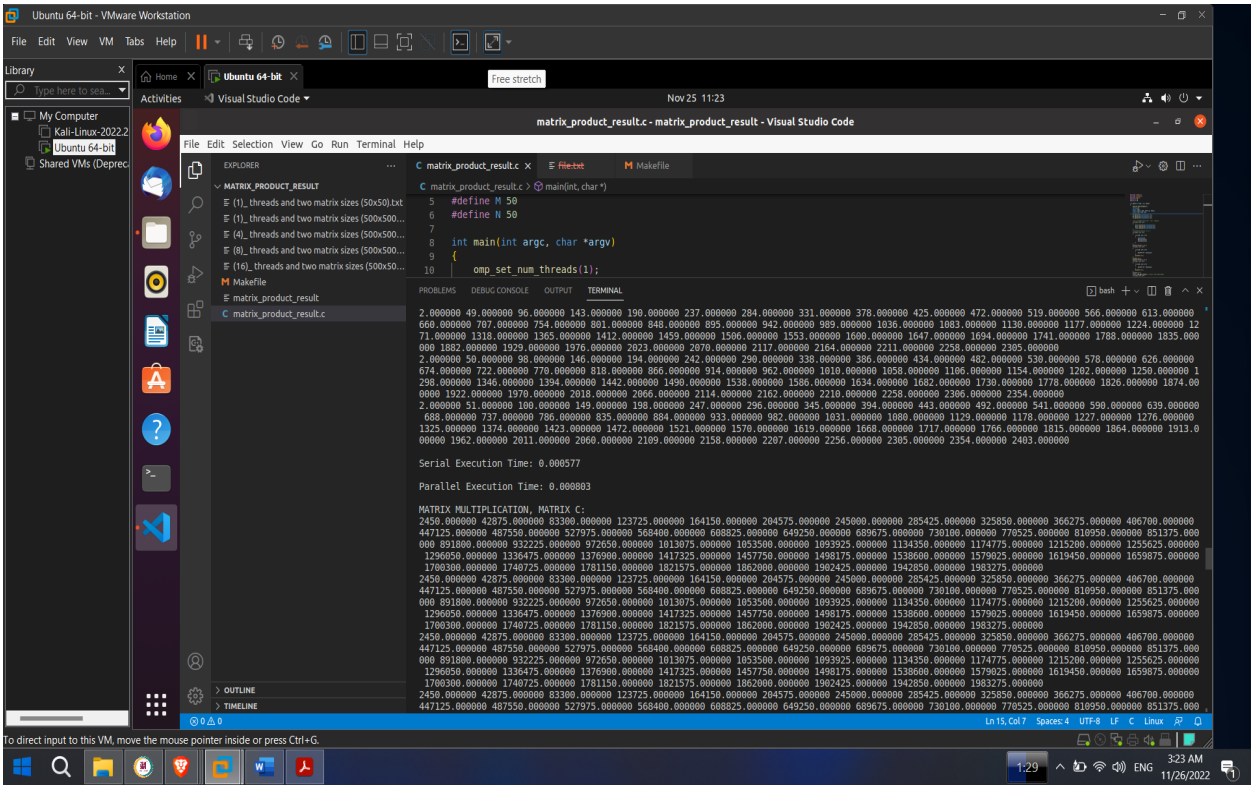


Fig : 1 threads and two matrix sizes 50x50

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.

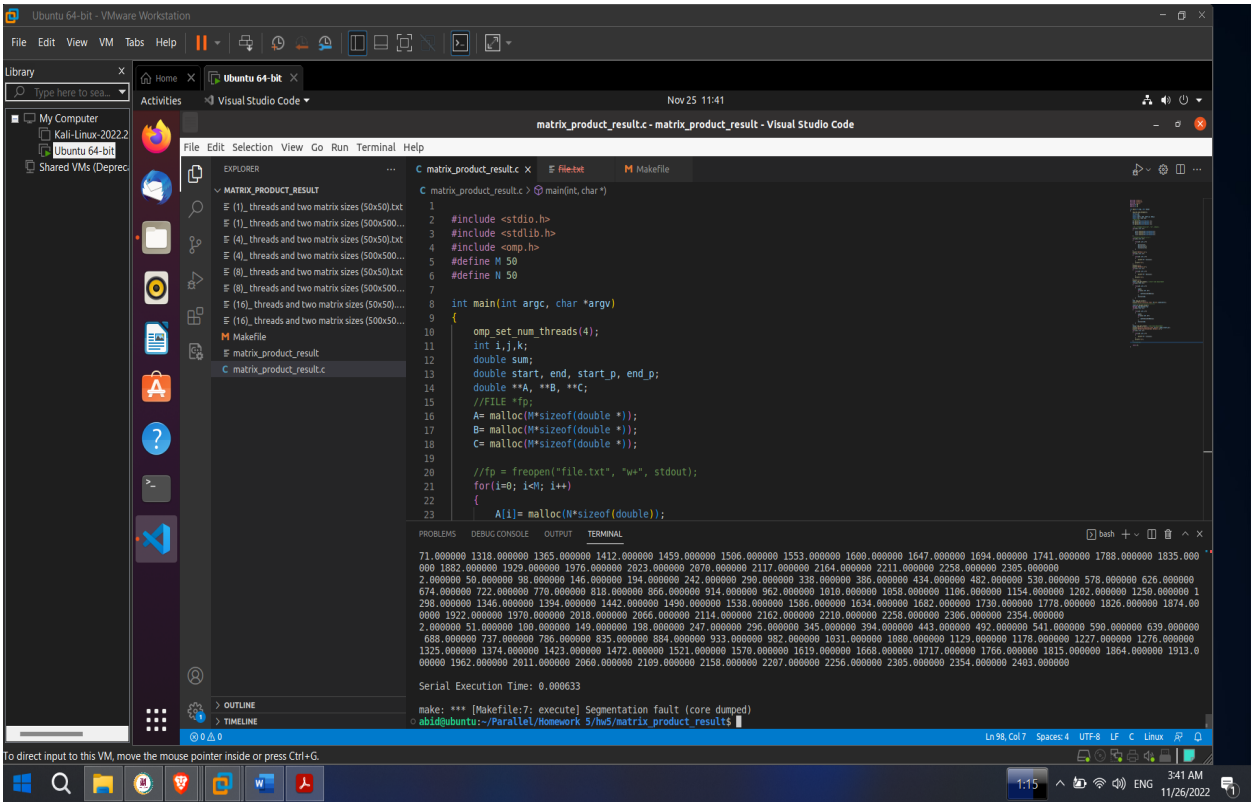


Fig : 4 threads and two matrix sizes 50x50

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.

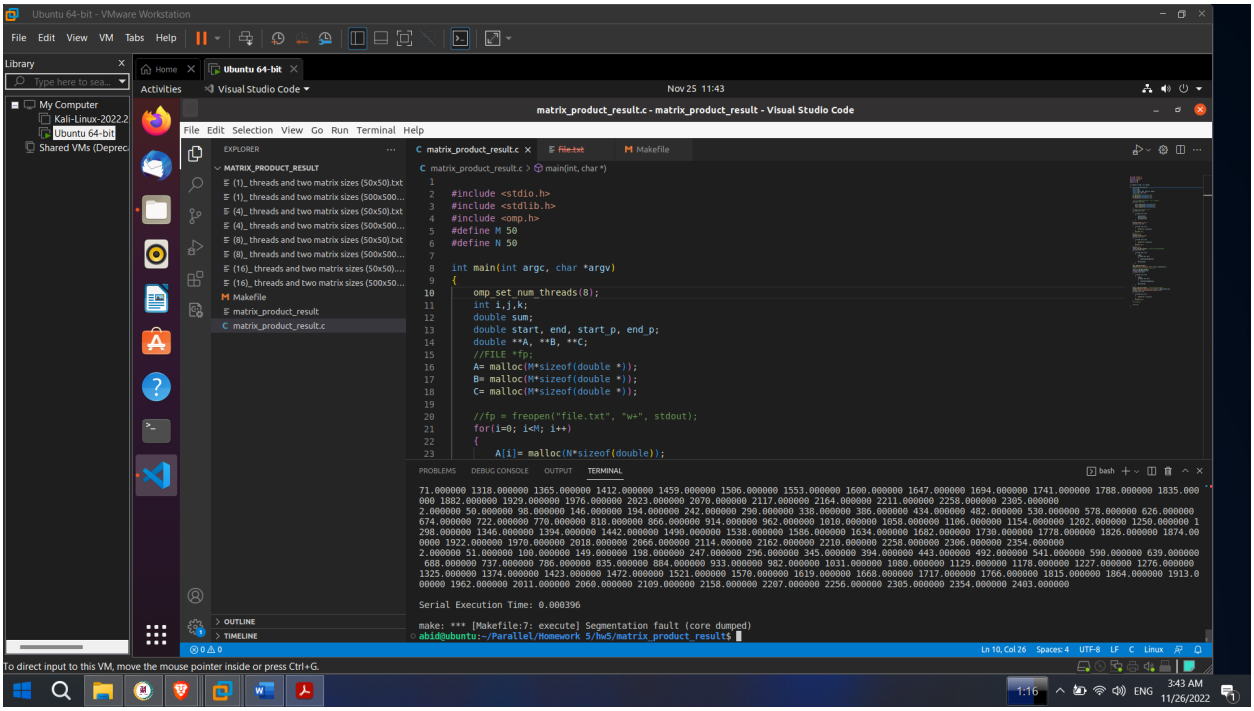


Fig : 8 threads and two matrix sizes 50x50

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.

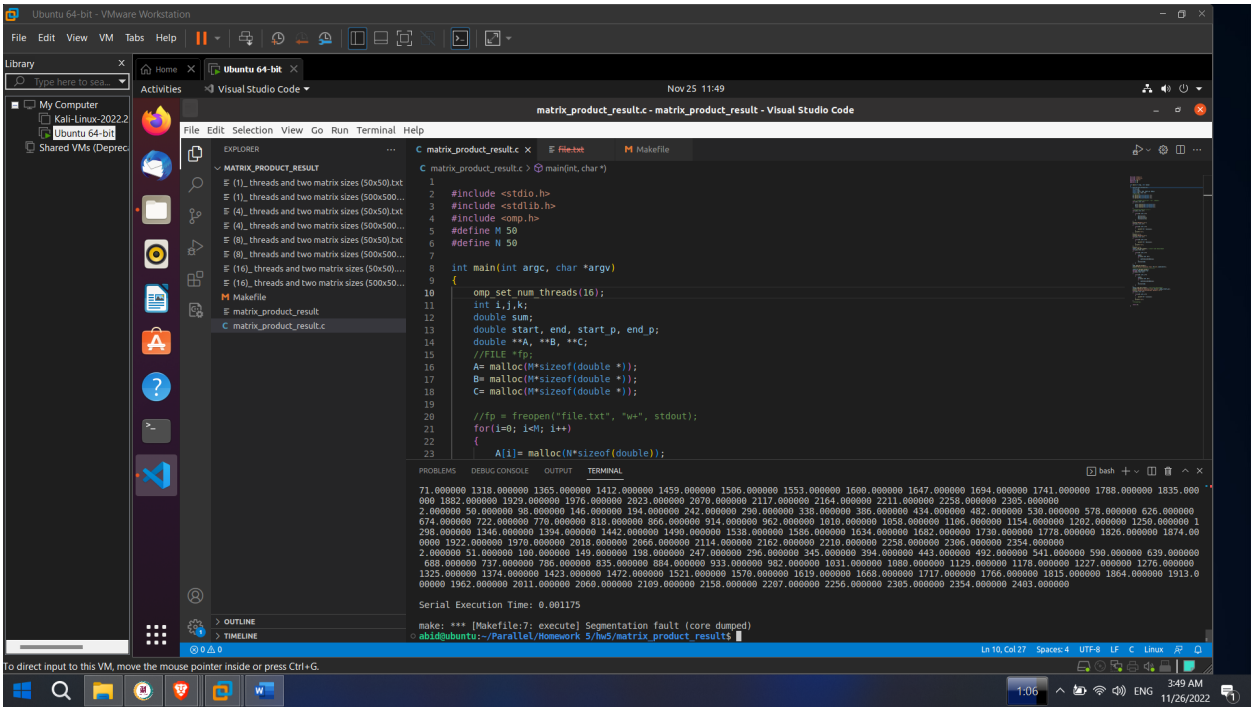


Fig : 16 threads and two matrix sizes 50x50