كلية علوم الحاسب والمعلومات

King Saud University
Collage of Computer and Information Sciences
Computer Science Department

PROJECT REPORT

Members Names
Faisal Aldhuwayhi – 438102142
Hassan Jaafar – 438105959
Fahad Aldeham – 438100439
Ahmad Almosallam – 438103307
Majed Alkhraiji - 438104708

Table of Contents

INTRODUCTION **2**

HARDWARE SPECIFICATIONS **3**

LIMITATIONS **3**

CODE **4**

CODE EXPLANATION 5

TIME MEASUREMENT **6**

SAMPLE OUTPUTS **7**

DATA GRAPHS **8**

CONCLUSION **10**

APPENDICES **11**

APPENDIX A 11

APPENDIX B 14

APPENDIX C 15

Introduction:

This project discusses the ability to utilize parallel processing to solve the problem of matrix multiplication, especially multiplying two matrices of the squared size of **MxM** of randomly generated numbers. Which is simply consisting of applying multiple dot products between the rows and the columns of the two matrices:

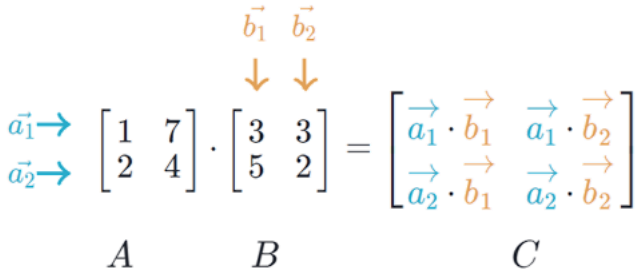


Figure 1

The matrix multiplication is defined as a binary operation that produces a matrix from two matrices. One condition for the operation to be applied, the number of columns in the first matrix must be equal to the number of rows in the second matrix. The resulting matrix, known as the matrix product, has the number of rows of the first and the number of columns of the second matrix. The product of matrices ***A*** and ***B*** is then denoted simply as ***AB*** or ***A · B***.

Matrix multiplication falls under linear algebra, and fortunately, it is a very suitable operation for parallel processing.

Parallel processing is a method in computing of running two or more processing units, by dividing the problem into sub-problems, and each processing unit process a portion of the problem. By doing so, will help reduce the amount of time to run a program. Parallel processing is commonly used to perform complex tasks and computations.

So, the aim of this project is to use the power of parallel processing on applying matrix multiplication in different sizes of matrices and a different number of processing units.

Parallel processing can be utilized using multiple schemes, such as MPI, OpenMP, and more. The project follows the OpenMP scheme, since it uses shared memory as a way of communicating between processing units, which makes it easier for the programmer to use parallel processing. On the other hand, MPI uses message passing to communicate among processing units, which leads to more effort at the programmer side.

Hardware Specifications:

This experiment has been done using the following hardware specifications:

- Intel (R) core (TM) i7-7700HQ @2.80GHz 2.81HGz (7th gen)
- 4 cores
- 8 logical processors
- DDR
- 16GB RAM
- 256KB, 1MB, and 6MB, these are the sizes of L1, L2, and L3 caches, respectively.

Limitations:

Parallel processing does not always reach our expectations, sometimes, it faces some limitations, such as limited computational resources, the size of the problem, the algorithm structure may not be suited for the designated hardware. Also, the nature of the problem could affect the performance of parallel processing negatively.

Code:

OpenMP API has been employed to perform the parallelism to solve the problem. The OpenMP API is a portable, scalable model that gives shared-memory parallel programmers a simple and flexible interface for developing parallel applications on platforms ranging from embedded systems and accelerator devices to multicore systems and shared-memory systems.

The methods used in the program are as follows:

- void r8_mxm(int l, int m, int n): which initializes three matrices a, b, c of the dimensions l, m, n respectively. The aim of this function is to perform the matrix multiplication operation. The implementation of parallelism has been located in this procedure.
- double r8_uniform_01(int *seed): It is called by the previous procedure to generate random numbers to help to fill the matrices.
- int main(int argc, char *argv[]): which is used to execute the whole program.
- **OpenMP Statements Used**:
    1. omp_get_wtime(); to get a value in seconds of the time elapsed from some point. (to get current time in seconds)
    2. omp_set_num_threads(NUM_THREADS); to set the number of threads using.
    3. omp_get_thread_num(); to get the thread number.
    4. omp_get_num_threads(); to get the number of the actual working threads.
    5. omp_get_num_procs(); to get the number of processors.
    6. omp_get_max_threads(); to get the maximum number of threads.

Code Explanation:

In this block code, the core function is r8_mxm(int l, int m, int n), which executes all the parallel code. Firstly, the initialization of the variables and allocation of matrices a, b, and c has been done.

Matrix a is the resulting matrix of multiplication of b and c matrices.

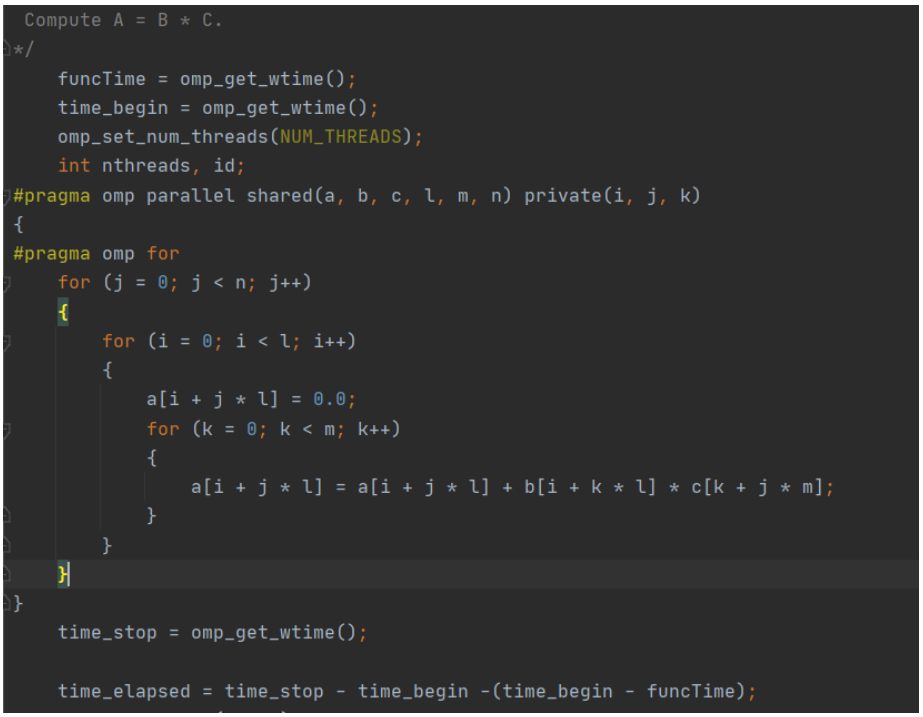
```
  Allocate the matrices.
*/
    a = (double *)malloc( _Size: l * n * sizeof(double));
    b = (double *)malloc( _Size: l * m * sizeof(double));
    c = (double *)malloc( _Size: m * n * sizeof(double));
    /*
  Assign values to the B and C matrices.
*/
    seed = 123456789;
    for (k = 0; k < l * m; k++)
    {
        b[k] = r8_uniform_01(&seed);
    }
    for (k = 0; k < m * n; k++)
    {
        c[k] = r8_uniform_01(&seed);
    }
```

Figure 2

Secondly, where filling the matrices is done. So, what it does is iterating over a matrix and fill out each element with a random number generated by the r8_uniform_01() function. In order to optimize memory allocation, a one-dimensional array is used to store data.

What is the purpose of using a one-dimensional array to store 2D matrices?

The purpose is because computer memory is much more sequential, even if the program perceives a square matrix as a 2D matrix. It is better at sequential access. When it comes to caching, it prefers related data together. In some caching algorithms, the data required for the next operation will be available in the next memory location. Then it can easily load those data to cache before the CPU requests those data. So, it is better to use the 1D approach, since it is faster, where it offers better memory locality, and less allocation and deallocation overhead. requests those data. So, it is better to use the 1D approach, since it is faster, where it offers better memory locality, and less allocation and deallocation overhead.



```
 Compute A = B * C.
*/
    funcTime = omp_get_wtime();
    time_begin = omp_get_wtime();
    omp_set_num_threads(NUM_THREADS);
    int nthreads, id;
#pragma omp parallel shared(a, b, c, l, m, n) private(i, j, k)
{
#pragma omp for
    for (j = 0; j < n; j++)
    {
        for (i = 0; i < l; i++)
        {
            a[i + j * l] = 0.0;
            for (k = 0; k < m; k++)
            {
                a[i + j * l] = a[i + j * l] + b[i + k * l] * c[k + j * m];
            }
        }
    }
}
    time_stop = omp_get_wtime();

    time_elapsed = time_stop - time_begin -(time_begin - funcTime);
```

Figure 3

Here, funcTime, time_begin, and time_stop are used for calculating the execution of time for parallel code. Next, is the start of the parallel region. Notice that, shared and private clauses are employed to specify the shared and private variables among the processing units.

The statement #pragma omp for has been exploited to parallelize the outermost for loop. Using this statement, it delegates portions of the for loop for different processing units.

Time Measurement:

First, the parallel program run-time has been calculated by taking the average of five runs, after removing outlier results. Each of different matrix sizes and different processing units, As shown in Appendix C.

Second, the time of calling the function omp_get_wtime() has been excluded, by calling it twice in a row and subtract that time from the total time. And that captured by the code chunk (time_begin - funcTime).

```
time_elapsed = time_stop - time_begin -(time_begin - funcTime);
```

Figure 4

Sample Outputs:

```
MXM
 C/OpenMP version.

 Matrix multiplication tests.

 Number of processors available = 8
 Max number of threads = 8
 Working number of threads = 1

R8_MXM matrix multiplication timing.
 A(LxN) = B(LxM) * C(MxN).
 L = 2000
 M = 2000
 N = 2000
 Floating point OPS roughly -1179869184
 Elapsed time dT = 38.237000
 Rate = MegaOPS/dT = -30.856740

MXM:
 Normal end of execution.

Process returned 0 (0x0)   execution time : 38.365 s
```

Figure 5

Figure 5 shows the time of serial execution with size of 2000. Which has a time of 38.24 seconds. After calculations, it appears that it has speedup of 1 and efficiency of 1.

```
MXM
 C/OpenMP version.

 Matrix multiplication tests.

 Number of processors available = 8
 Max number of threads = 8
 Working number of threads = 3

R8_MXM matrix multiplication timing.
 A(LxN) = B(LxM) * C(MxN).
 L = 2000
 M = 2000
 N = 2000
 Floating point OPS roughly -1179869184
 Elapsed time dT = 14.676000
 Rate = MegaOPS/dT = -80.394465

MXM:
 Normal end of execution.

Process returned 0 (0x0)   execution time : 14.804 s
```

Figure 6

Figure 6 shows the time of parallel execution of three processing units with the same previous size. Which has a time of 14.68 seconds. After calculations, it appears that it has speedup of 2.60 and efficiency of 0.87.

Looking at the results of both cases, it can be concluded that the parallelism can help to reduce the run-time of the program with even excellent speedup and efficiency.

Data Graphs:

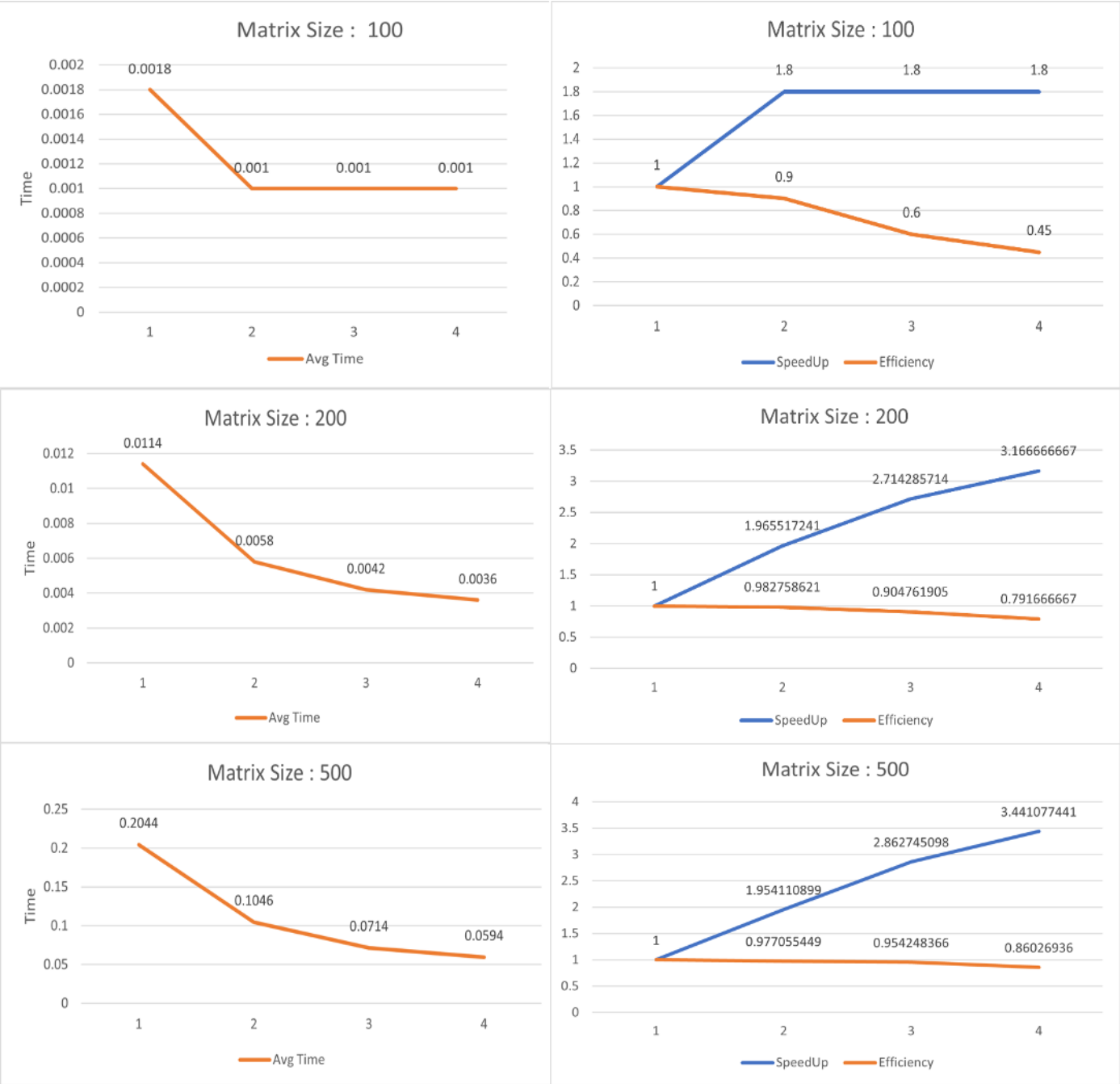


Figure 7

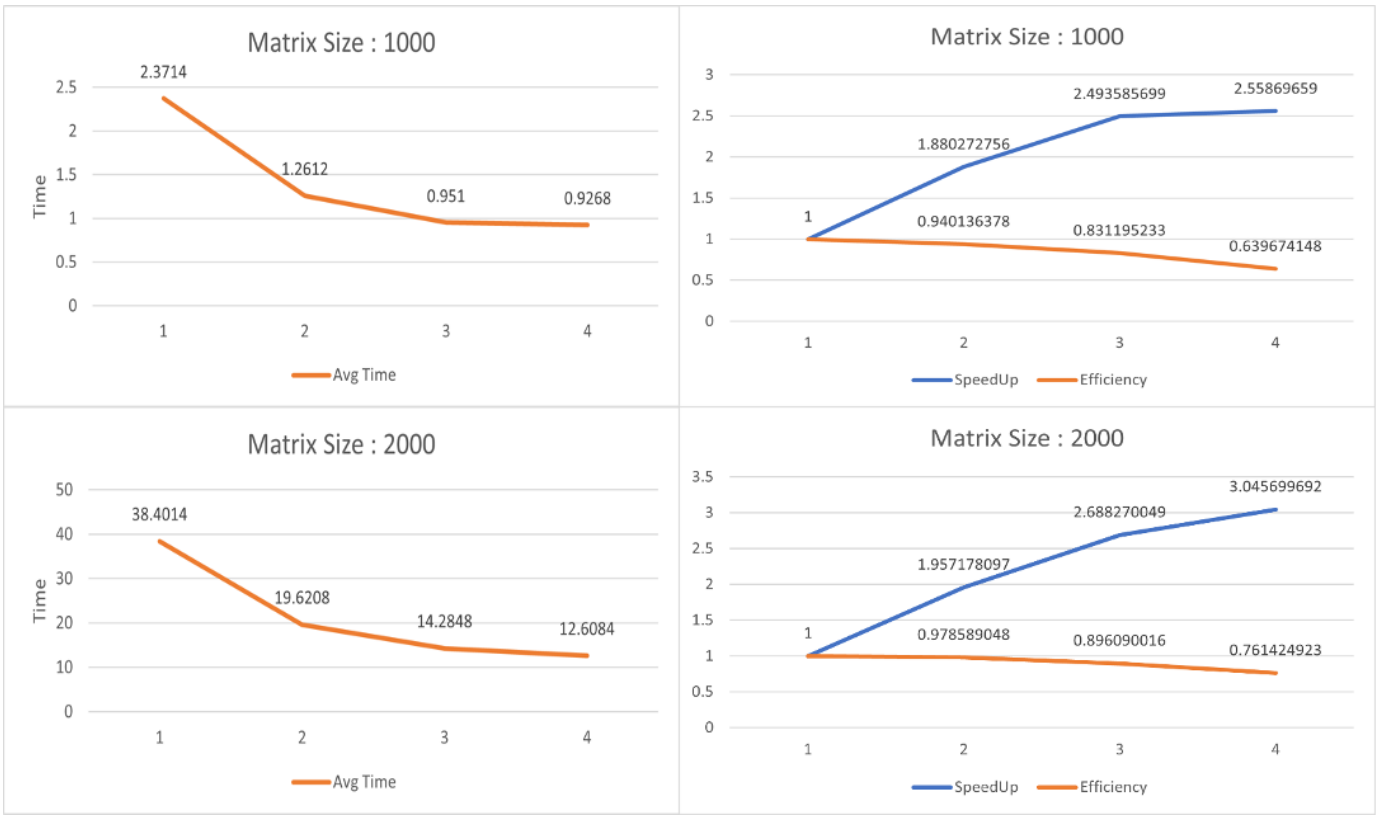


Figure 8

The graphs show that the speedup increases as the number of processing units increases. But looking at the efficiency, it decreases as the number of processing units increases.

Moreover, the average time slows as the number of processing units increases.

And that applied to all the various sizes used in this experiment, except for the matrix of size 100, as it is difficult here to observe the difference between the runs of different processing units. For the reason that with small sizes of a problem, mostly, different processing units will result in time nearly the same.

Conclusion:

After exploring parallel processing on the problem of matrix multiplication, it appears to be that parallel processing can handle complicated problems in less time than serial processing.

In parallel processing, in this problem specifically, the efficiency is reduced as the number of processing units rises. This is due to the lack of utilizing all processing units to their full capability, In the whole run-time of the program.

Potential Domain of Improvement:

The performance of parallel algorithms may be increased significantly with programming techniques oriented to locality and memory issues. For instance, implementing a more friendly cache algorithm, that stores the first matrix as a row-major array, and the second matrix as a column-major, to minimize the jumps between memory locations that have all the related elements together in the memory. This will also reduce the cache misses.

```
#pragma omp parallel for
for(int i=0; i<dimension; i++){
        for(int j=0; j<dimension; j++){
                flatA[i * dimension + j] = matrixA[i][j];
                flatB[j * dimension + i] = matrixB[i][j];
        }
}
```

Figure 9

Also, it is fast to access stack rather than heap memory. But stack has limited memory. The large input memories will be stored in heap memory. For efficient intermediate calculations, the stack with predefined memory allocations can be used.

Appendices:

Appendix A: The Parallel Code

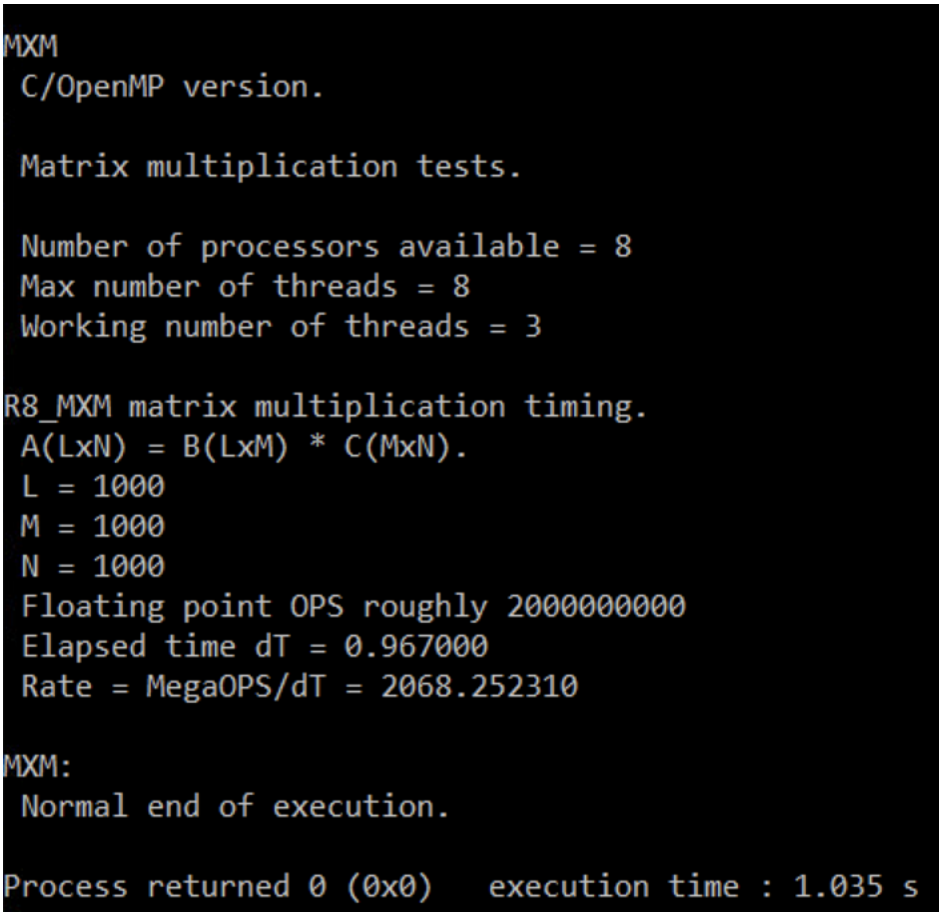
```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <omp.h>
#define NUM_THREADS 3

double r8_uniform_01(int *seed)
{
    int k;
    double r;
    k = *seed / 127773;
    *seed = 16807 * (*seed - k * 127773) - k * 2836;
    if (*seed < 0)
    {
        *seed = *seed + 2147483647;
    }
    r = (double)(*seed) * 4.656612875E-10;
    return r;
}

void r8_mxm(int l, int m, int n)
{
    double *a;
    double *b;
    double *c;
    int i;
    int j;
    int k;
    int ops;
    double rate;
    int seed;
    double time_begin;
    double funcTime;
```

```
  double time_elapsed;
  double time_stop;
  /*
Allocate the matrices.
*/
  a = (double *)malloc(l * n * sizeof(double));
  b = (double *)malloc(l * m * sizeof(double));
  c = (double *)malloc(m * n * sizeof(double));
  /*
Assign values to the B and C matrices.
*/
  seed = 123456789;
  for (k = 0; k < l * m; k++)
  {
    b[k] = r8_uniform_01(&seed);
  }
  for (k = 0; k < m * n; k++)
  {
    c[k] = r8_uniform_01(&seed);
  }
  /*
Compute A = B * C.
*/
  funcTime = omp_get_wtime();
  time_begin = omp_get_wtime();
  omp_set_num_threads(NUM_THREADS);
  int nthreads, id;
#pragma omp parallel shared(a, b, c, l, m, n) private(i, j, k)
{
#pragma omp for
  for (j = 0; j < n; j++)
  {
    for (i = 0; i < l; i++)
    {
      a[i + j * l] = 0.0;
      for (k = 0; k < m; k++)
      {
```

```
                a[i + j * l] = a[i + j * l] + b[i + k * l] * c[k + j * m];
            }
        }
    }

    /*id = omp_get_thread_num();
    if (id == 0)
        nthreads = omp_get_num_threads();*/
}
    time_stop = omp_get_wtime();

    /*printf(" Working number of threads = %d\n", nthreads);*/
    /*
 Report.
*/
    ops = l * n * (2 * m);
    time_elapsed = time_stop - time_begin -(time_begin - funcTime);
    rate = (double)(ops) / time_elapsed / 1000000.0;
    printf("\n");
    printf("R8_MXM matrix multiplication timing.\n");
    printf(" A(LxN) = B(LxM) * C(MxN).\n");
    printf(" L = %d\n", l);
    printf(" M = %d\n", m);
    printf(" N = %d\n", n);
    printf(" Floating point OPS roughly %d\n", ops);
    printf(" Elapsed time dT = %f\n", time_elapsed);
    printf(" Rate = MegaOPS/dT = %f\n", rate);
    free(a);
    free(b);
    free(c);
    return;
}

int main(int argc, char *argv[])
{
    int id;
    int l;
```

```
  int m;
  int n;
  printf("\n");
  printf("MXM\n");
  printf(" C/OpenMP version.\n");
  printf("\n");
  printf(" Matrix multiplication tests.\n");
  printf("\n");
  printf(" Number of processors available = %d\n",
      omp_get_num_procs());
  printf(" Max number of threads = %d\n",
      omp_get_max_threads());
  l = 200;
  m = 200;
  n = 200;
  r8_mxm(l, m, n);

  /*
Terminate.
*/
  printf("\n");
  printf("MXM:\n");
  printf(" Normal end of execution.\n");
  return 0;
}
```

Appendix B: The Sample Output



Appendix C: Tables of the Results of all Runs

Average Time:

RUN#	p	M	Time1	Time2	Time3	Time4	Time5	Avg Time
1	1	100	0.002	0.003	0.001	0.002	0.001	0.0018
2	2	100	0.001	0.001	0.001	0.001	0.001	0.001
3	3	100	0.001	0.001	0.001	0.001	0.001	0.001
4	4	100	0.001	0.001	0.001	0.001	0.001	0.001
5	1	200	0.012	0.011	0.011	0.011	0.012	0.0114
6	2	200	0.007	0.006	0.005	0.006	0.005	0.0058
7	3	200	0.004	0.003	0.005	0.005	0.004	0.0042
8	4	200	0.004	0.003	0.003	0.004	0.004	0.0036
9	1	500	0.208	0.207	0.199	0.202	0.206	0.2044
10	2	500	0.101	0.101	0.105	0.105	0.111	0.1046
11	3	500	0.069	0.07	0.073	0.073	0.072	0.0714
12	4	500	0.065	0.057	0.06	0.063	0.052	0.0594
13	1	1000	2.476	2.272	2.383	2.35	2.376	2.3714
14	2	1000	1.159	1.338	1.321	1.316	1.172	1.2612
15	3	1000	0.822	1.031	0.86	1.064	0.978	0.951
16	4	1000	0.934	0.967	0.982	0.818	0.933	0.9268
17	1	2000	37.861	38.372	38.088	38.651	39.035	38.4014
18	2	2000	19.958	19.176	19.587	19.607	19.776	19.6208
19	3	2000	14.081	14.29	14.909	13.984	14.16	14.2848
20	4	2000	12.849	12.803	12.103	12.45	12.837	12.6084

Measurements and Metrics:

RUN#	p	M	T1	Tp	Speedup	Efficiency

1	1	100	0.0018	0.0018	1	1
2	2	100	0.0018	0.001	1.8	0.9
3	3	100	0.0018	0.001	1.8	0.6
4	4	100	0.0018	0.001	1.8	0.45
5	1	200	0.0114	0.0114	1	1
6	2	200	0.0114	0.0058	1.965517241	0.982758621
7	3	200	0.0114	0.0042	2.714285714	0.904761905
8	4	200	0.0114	0.0036	3.166666667	0.791666667
9	1	500	0.2044	0.2044	1	1
10	2	500	0.2044	0.1046	1.954110899	0.977055449
11	3	500	0.2044	0.0714	2.862745098	0.954248366
12	4	500	0.2044	0.0594	3.441077441	0.86026936
13	1	1000	2.3714	2.3714	1	1
14	2	1000	2.3714	1.2612	1.880272756	0.940136378
15	3	1000	2.3714	0.951	2.493585699	0.831195233
16	4	1000	2.3714	0.9268	2.55869659	0.639674148
17	1	2000	38.4014	38.4014	1	1
18	2	2000	38.4014	19.6208	1.957178097	0.978589048
19	3	2000	38.4014	14.2848	2.688270049	0.896090016
20	4	2000	38.4014	12.6084	3.045699692	0.761424923