Laboratory Exercise 1: Multidimensional Array Multiplication

ELEN4020: Data Intensive Computing in Data Science

Authors:
Diana Munyaradzi (1034382) | Morongwa Thobejane (1432370) | Michael Asamoah-Bekoe (1353083)

School of Electrical & Information Engineering, University of the Witwatersrand, Private Bag 3, 2050, Johannesburg, South Africa

I. Introduction

Pthread and OpenMP are application programming interfaces (API) that allow a program to control multiple different flows of work that overlap in time [1]. Pthreads are low-level and they also allow the user to control the thread management [1]. OpenMP is high-level, more portable and allows for the division of work across multiple threads easy [1]. The aim of this lab is to implement an algorithm that computes the matrix multiplication of two 3D matrices using the Pthread library and OpenMP.

II. 3D Array Multiplication Algorithm

The 3D Array Multiplication algorithm is implemented by considering a 3D array of dimension NxNxN as N 2D arrays with the dimension NxN. This made the algorithm easier to implement, as the 2D multiplication algorithm is executed on each of the layers of the corresponding arrays. This concept is shown in the figure below.

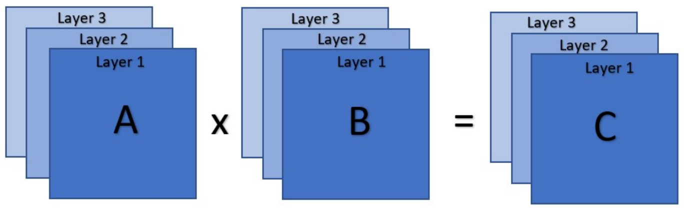

Fig. 1: 3D Multiplication Concept

A. PThreads

This algorithm was implemented with the use of PThreads where, the *ThreadDataIndex* struct is defined. The structure of threads contains the thread ID, pointers to the matrices that the the threads have access to and the matrix size. The number of threads to be used are defined by the user. A global variable named *nxt_operation_index* is defined and used to allow a thread to know which operation to do next. Each thread is assigned to an operation (which is the multiplication of two elements) based on the operation index. A PThread Mutex Lock is created to ensure that the threads wait for the the thread that is currently accessing data to complete its operation before the next operation is executed. This is done to make sure that the results are accurate as each thread will work independently from one another. The pseudocode for this algorithm is presented in Algorithm 5 of the Appendix.

B. OpenMP

Algorithm 10 and Algorithm 111, in the appendix, show the algorithms that are used to implement parallelisation using OpenMP for the 2D and 3D matrix multiplication, respectively. A static schedule clause was chosen to implement this using a pragma directive. Therefore, when the parallel for loop is entered, OpenMP automatically assigns each thread the iterations it will run. The scheduling clause also takes into account which variables are shared by all the threads and which are only used by the specific thread assigned to an iteration.

III. Results

Figure 2 and Figure 3, in the Appendix show the execution time for 2D and 3D matrix multiplication, specific to the number of threads used for the execution. The results were conducted on an 8th Generation 4 core i7 Dell Latitude 7490 laptop.

A. OpenMP

For OpenMP, when N=10, the execution time decreases slightly with an increase in the number of the threads from 1 to 2 threads. It increases slightly when 3 or 6 threads are used. A similar pattern can be observed when N=20, with the exception that the optimum execution time can be seen when 5 threads are used instead of 6. When N=30, the execution time when using 1, 2 or 3 threads is approximately the same and increasing the threads to 4 and 5 slightly increases the execution time. However, a large increase in execution time can be observed when 6 are used and a slight decrease can be observed when 8 threads are used. The decreases in execution time can be attributed to effective scaling by the parallelisation procedure. The increases in execution time can be attributed to the use of static scheduling instead of dynamic scheduling. Although static scheduling minimises the chances of memory conflicts occurring, it does not take into the length of the iterations when distributing the iterations to the threads. As a result, threads with an equal number of iterations will not necessarily take the same time to complete the iteration in theory but all threads will only be able to commence their next iteration once all other threads have completed theirs. A similar trend can be seen in the 2D Algorithm Results shown in Figure 3.

B. PThreads

For PThreads, the execution time increased slightly when the number of threads increased from 1 to 2. It increases again when 3 to 7 threads are used after dropping at 3 and drops slightly when 8 threads are used. For N = 20, the execution time keeps an increasing trend as the number of threads used increase from 1 to 4, 5 to 6 and from 7 -8. When N=30, the execution time decreases when the number of threads used increases from 1 to 4 and it keeps decreasing slightly drastically when the number of threads used increase from 5 to 6 and from 7 to 8. The decrease in the execution time maybe caused by the use of locks and shared-resources (or synchronization) because the threads have to wait for each other [2]. Using more threads also reduces the execution time. These results are presented in Figure 2. A similar trend can be seen in the 2D Algorithm Results shown in Figure 3.

From the results, it can be seen that PThreads have a longer execution time than OpenMP for both 2D and 3D Matrix Multiplication Algorithms. OpenMP is faster as it is a higher level API that is designed to automatically create, schedule and join threads while PThreads require the programmer to do these processes manually [1]. In the code implemented for this laboratory, algorithms using OpenMP were not placed in functions due to erroneous results while the Algorithms using functions were all placed in function, this may also be a factor that increasing the execution time for the PThreads.

IV. Conclusion

The report presents the detailed explanation of 2D and 3D matrices multiplication algorithms implemented in C++ programming language. These algorithms are implemented using two threading libraries called PThreads and OpenMP. The two algorithms were timed and the results were analyse, leading to the discovery that the algorithms using OpenMP has a faster execution time than the algorithms using PThreads.

References

[1] B. Barney, L. - OpenMP, Livermore National Laboratory. In: Smithsonian.com, https://computing.llnl.gov/tutorials/openMP/. [Accessed 25 Feb 2020]

[2] Z. Smith, Project 2017 - How do I... Use threading to increase performance in C? (Part 1), https://www.techrepublic.com/blog/how-do-i/how-do-i-use-threading-to-increase-performance-in-c-part-1/, [Accessed 25 Feb 2020]

Appendix

Figures

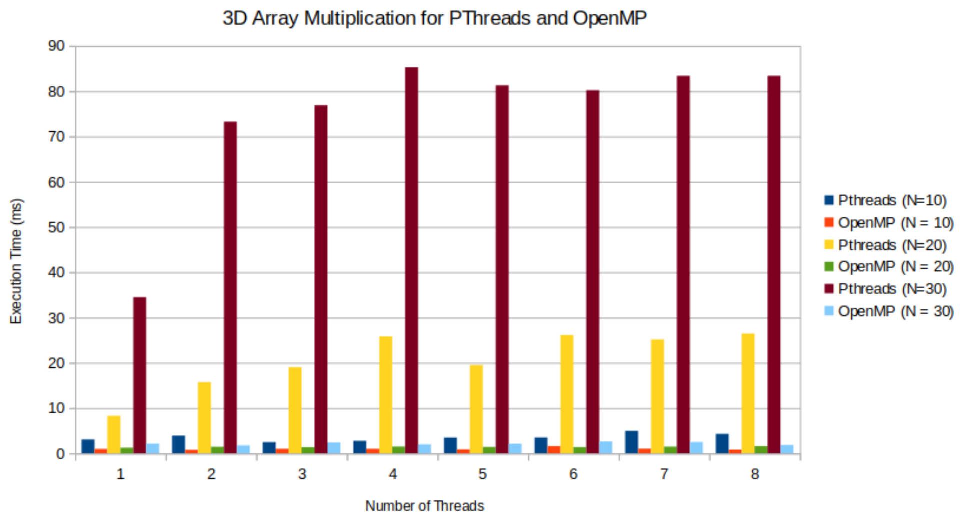

Fig. 2: Execution Time vs Number of Threads for 3D Matrix Multiplication

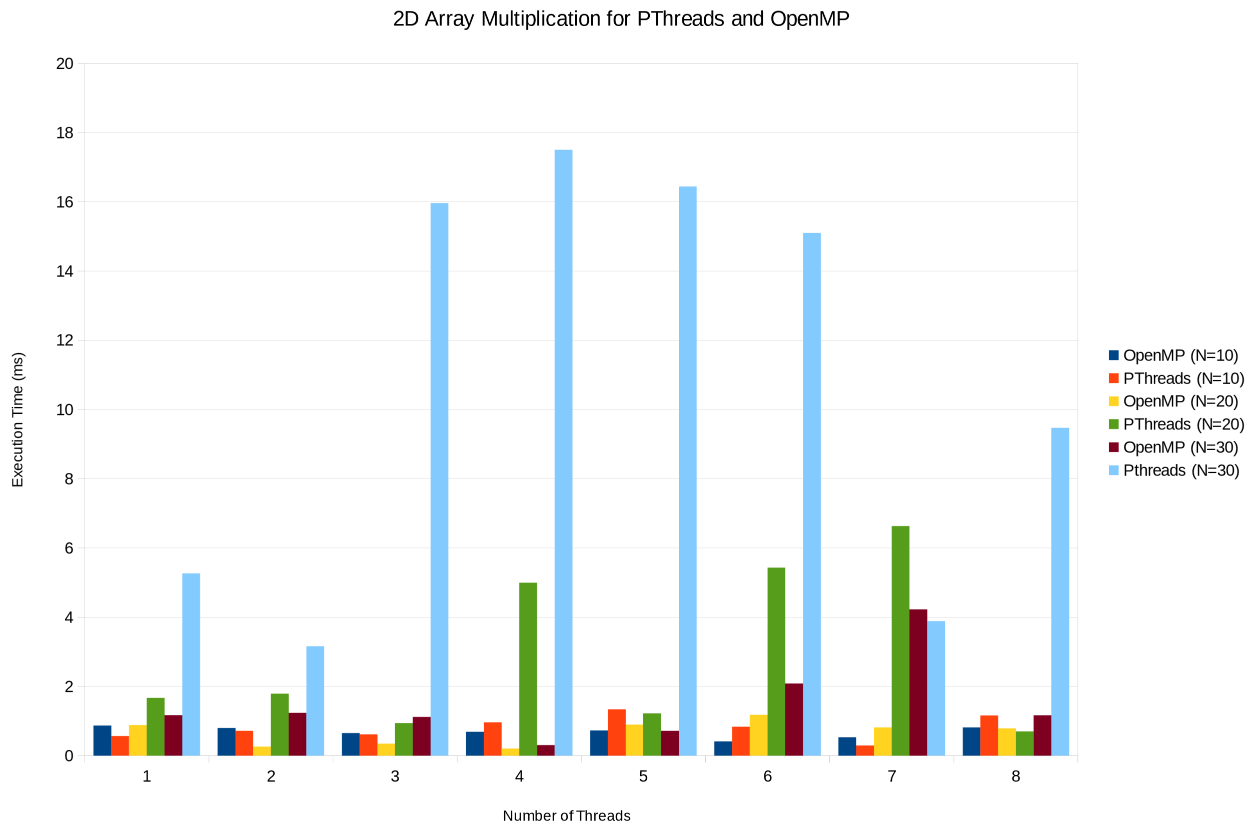

Fig. 3: Execution Time vs Number of Threads for 2D Matrix Multiplication

Appendix

Pseudocode

Appendix

A. GenerateMatrix

Algorithm 1: Dynamically Generate and Populate Matrix with Random Values

```
Result: Pointer, pointing to the first element of the matrix
Input   : Matrix Length (N)
1 dimension = N*N
2 seed = rand()
3 srand(seed)
4 matrix = (int* )calloc(dimension, sizeof(int))
5 *element_ptr = matrix
6 for m=0 to dimension do
7     *element_ptr = rand()
8     element_ptr++
9 return matrix
```

B. AllocateMatrix

Algorithm 2: Dynamically Generate a Matrix

```
Result: Pointer, pointing to the first element of the matrix
Input   : Matrix Length (N)
1 dimension = N*N*N
2 result = (int*)calloc(dimension, sizeof(int))
3 return result
```

C. getElementPosition2D

Algorithm 3: Get the Corresponding Element Position of the 2D Matrix from the 1D Matrix of Elements

```
Result: Integer containing position of element in 1D matrix
Input   : Matrix Containing the Elements Position (row and column)
Input   : Length of the Matrix (N)
1 row = coords[0]
2 column = coords[1]
3 return $(row * N) + column$
```

D. getElement

Algorithm 4: Get the Element in the 2D matrix

```
  Result: Element
  Input  : Pointer to the first element of the matrix
  Input  : 1D Matrix containing indexes
  Input  : Length of Matrix (N)
1 return $*(matrix\_ptr + getElementLocation2D(index, N))$
```

E. ElementMultiplier

Algorithm 5: Element Multiplication using PThreads

```
   Result: Void Pointer
1  Create global variable nxt_operation_index
   Input   : Void pointer
2  ThreadDataIndex* thread_data = (ThreadDataIndex*)arg
3  N = thread_data− >N
4  Operations = N*N*N
5  dimension = N*N
6  while true do
7      m = thread_data− >operation_index
8      i = m/dimension
9      j = m%N
10     k = (m/N)%N
11     indexA[2] = {i, k}
12     indexB[2] = {k, j}
13     elementA = getElement(thread_data− >matrixA, indexA, N)
14     elementB = getElement(thread_data− >matrixB, indexB, N)
15     indexC[2] = {i, j}
16     total = thread$_d$$ata\text{-¿}matrixC + getElementPosition2D(indexC, N)$
17     *total+=(elementA*elementB)
18     pthread_mutex_lock(updateIndexLock)
19     Mutex Lock
20     if nxt_operation_index  Operations then
21         thread$_d$$ata\text{-¿}operation_index =$
           $\_nxt\_operation\_index + +nxt\_operation\_index$;
22     else
23         thread_data− >operation_index = Operations
24     Mutex Unlock
25     Mutex Lock Updated
26     if thread_data− >operation_index == Operations then
27         break
28     exit pthread
29 return void pointer
```

F. rank2TensorMultPThread

Algorithm 6: 2D Matrix Multiplication Algorithm

```
   Result: Updating the values of the resultant 2D Matrix
1  Define the number of threads (num_threads)
2  Create global variable nxt_operation_index
   Input  : Pointer to matrix A's first element
   Input  : Pointer to matrix B's first element
   Input  : Pointer to matrix C's first element
   Input  : Matrix Length (N)
3  Declare rc as an integer
4  pthread_t threads[num_threads]
5  ThreadDataIndex threads_data[num_threads]
6  nxt_operation_index = (int)num_threads;
7  for i = 0 to num_threads do
8      threads_data[i].ID = i
9      threads_data[i].operation$_i$$ndex = i$
10     threads_data[i].matrixA = matrixA
11     threads_data[i].matrixB = matrixB
12     threads_data[i].matrixC = matrixC
13     threads_data[i].N = N
14     rc = pthread_create(&threads[i], NULL, ElementMultiplier,
        &threads_data[i])
15     if rc is true then
16         Display ”ERROR creating thread
17         Program terminates
18     for j = 0 to num_threads do
19         pthread_join(threads[j],NULL)
20 return
```

G. rank3TensorMultPThread

Algorithm 7: 3D Matrix Multiplication Algorithm

```
   Result: Updating the values of the resultant 3D Matrix
1  startPos = 0
2  elements2D = N*N
3  for i = 0 to N do
4      rank2TensorMult(matrixA + startPos, matrixB + startPos, matrixC +
        startPos, N)
5      startPos+= elements2D
```

H. Display2DMatrix

Algorithm 8: Display the 2D Matrix

```
Result: Output the Resultant 2D Matrix
Input  : Pointer, pointing to the first element of Resultant Matrix C
Input  : Length of the Matrix (N)
1  dimension = N*N
2  for  i = 0 to dimension do
3      if i mod Length is equal to zero then
4          output a newline
5      Dereference pointer
6      Increment pointer by one
7  return
```

I. Display3DMatrix

Algorithm 9: Display the 3D Matrix

```
Result: Display the Resulting 3D Matrix
Input  : Pointer, pointing to the first element of a Matrix
Input  : Matrix Length N
1  dimension = N*N*N
2  if matrix pointer is not equal to NULL then
3      for  i = 0 to dimension do
4          if i mod N is equal to zero then
5              Display ”Layer:”
6          if i mod N is equal to zero then
7              Dereference pointer
8              Increment pointer by one
9  else
10     Display ”The Matrix is empty”
11 return
```

Algorithm 10: Multiply 2 2D matrices using OpenMP

```
Result: Integer determining whether the program was successfully executed
N = 10
matA2D[N][N] = {}
matB2D[N][N] = {}
matC2D[N][N] = {}
seed = random number between 0 and 100
srand(seed)
for i=0 to N do
    for j=0 to N do
        matA2D[i][j] = seeded random number between 0 and N
        matB2D[i][j] = seeded random number between 0 and N
    end
end
Begin parallelisation using pragma directive and a static schedule clause
for i=0 to N do
    for j=0 to N do
        for k=0 to N do
            matC2D[i][j] = matC2D[i][j] + matA2D[i][k]*matB2D[k][j];
        end
    end
end
Append schedule clause to “for” loop’s pragma directive
for i=0 to N do
    for j=0 to N do
        Display the element in the matC2D[i][j]
    end
    Output a newline
end
return
```

Algorithm 11: Multiply 2 3D matrices using OpenMP

```
Result: Integer determining whether the program was successfully executed
N = 10
matA3D[N][N][N] = {}
matB3D[N][N][N] = {}
matC3D[N][N][N] = {}
seed = random number between 0 and 100
srand(seed)
for i=0 to N do
    for j=0 to N do
        for k=0 to N do
            matA3D[i][j][k] = seeded random number between 0 and N;
            matB3D[i][j][k] = seeded random number between 0 and N;
        end
    end
end
Begin parallelisation using pragma directive and a static schedule clause
for i=0 to N do
    for j=0 to N do
        for k=0 to N do
            for m=0 to N do
                matC3D[i][j][k] = matC3D[i][j][k] + matA3D[i][j][m]*matB3D[i][m][k];
            end
        end
    end
end
Append schedule clause to “for” loop’s pragma directive
for i=0 to N do
    for j=0 to N do
        for k=0 to N do
            Display the element in the matC3D[i][j][k]
        end
        Output a new line
    end
    Output a new line
end
return
```