UNIVERSITY OF WITWATERSRAND

ELEN4020

DATA INTENSIVE COMPUTING IN DATA SCIENCE

Laboratory Exercise 2

Authors:
Kishan Narotam - 717 931
Jesal Chana - 603 177

Authors:
Syed Hussain - 600 524
Yusuf Ally - 604 973

9th March 2018

I. Problem Description

A square matrix, defined as $A[N_0][N_1]$, must be created with various dimensions, where $N_0 = N_1 = 128, 1024, 8192$. The matrix could be an integer which is 4 bytes or a short floating point which too is 4 bytes. The elements of the matrix must be generated using the formula presented in equation 1:

$$A\langle i, j\rangle = i * N_j + j \tag{1}$$

The transpose of these matrices must be computed without wasting memory by creating a copy of the original matrix, but rather modify the original matrix to become the transposed matrix. Since the size of the matrix is so large, PThread and OpenMP programming must be performed on the matrix. The number of threads that must be used are $4, 8, 16, 128$ for each value of N_0. The time taken to perform the transposition of each matrix must be recorded.

II. Background

A. PThread

A thread is a set of instructions that are independent and scheduled to run by the operating system. A POSIX Thread (PThread) however is defined as a standardized programming interface for UNIX systems, i.e. an execution model independent of any programming language. It could also be defined as a set of *C* language programming types, functions, procedures and constants that are implemented with the library header file, `pthread.h` [1].

PThreads are also regarded as low-level application programming interface (API) when managing threads. It allows the user to have as refined control as possible over the threads as well as multiple exclusion objects [2]. Since the Pthreads are such low level, it is fairly limited to the language used.

B. OpenMP

Open Multi-Processing (OpenMP) is an API that is used explicitly to direct multi-threading and shared memory parallelism. It is comprised of three primary API components:

- Compiler Directives
- Runtime Library Routines
- Environment Variables[3].

OpenMP is a much higher level API, and portable allowing the user to utilize this API in different languages such as C, C++ and Fortran. It is more easily scaled than PThreads as it has the ability to divide the work across multiple threads with ease [2], [3].

For this type of problem, the program that utilizes OpenMP should reveal faster times than the program that utilizes PThreads.

C. Amdahl's Law

Amdahl's Law is a key concept that is used to understand in parallel computing. It is used to predict the maximum speedup in latency for a program processing using multiple processors [4].

In parallel computing, the law states that if P is the program that can be made parallel, i.e. proportion of execution time, and given that $1 - P$ is the remaining proportion, i.e. the proportion that is still serial, the maximum speedup in latency using N number of processors can be defined as equation 2.

$$S_{latency} = \frac{1}{(1-P) + \frac{P}{N}} \tag{2}$$

The speedup is limited by the total time needed for the serial proportion of the program. For example is a program need 10 hours using a single processor, and 9 hours can be parallelized, and 1 cannot, the maximum speedup is limited to $10\times$ using equation 2 [4].

III. Function Description

A. Transpose method

The input matrix `array` is transposed by swapping element $A[i, j]$ with $A[j, i]$. Note only elements in the upper triangle of the array are transposed. For a square matrix of width N, the number of upper triangle elements Δ is given by

$$\Delta = \frac{N^2 - N}{2} \tag{3}$$

A second array, `utArray`, is populated with only the indices of upper triangle elements of `array`. In this way, only Δ elements need to be traversed in order to do the transpose. `utArray` is therefore a 1D array of size Δ. As `utArray` is traversed, it's corresponding 2D coordinate in `array` is obtained using

$$row = i \ / \ N \tag{4}$$

and

$$column = i \ \% \ N \tag{5}$$

Where i is the index of a particular element. For example a 4x4 matrix given as

$$Array = \begin{bmatrix} 0 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \\ 8 & 9 & 10 & 11 \\ 12 & 13 & 14 & 15 \end{bmatrix}$$

will produce an upper triangle matrix

$$utArray = \begin{bmatrix} 1 & 2 & 3 & 6 & 7 & 11 \end{bmatrix}$$

Example : choose element '6' in `utArray`; $i = 6$.

$\rightarrow row = i \ / \ N = 6 \ / \ 4 = 1$ (integer division)
$\rightarrow column = i \ \% \ N = 6 \ \% \ 4 = 2$ (modulo division)

So at coordinate (1,2) in `array`, the value is 6 as expected. This is then swapped with the element at (2,1) in `array` - which is 9. This process is repeated for every element in `utArray`.

B. *Multithreading approach*

For optimal performance, it is desirable to have the workload divided equally between all threads. In this case it means having each thread to the same amount of transposes. For this, we calculate α, which is the number of elements in `utArray` to traverse per thread.

$$\alpha = \frac{\Delta}{Threads} \tag{6}$$

So for our previous 4×4 matrix, using 2 threads ($Threads = 2$),

$$\alpha = \frac{\Delta}{Threads} = \frac{6}{2} = 3 \tag{7}$$

This means thread 0 transposes the first three elements in `utArray`, and thread 1 transposes the next three elements.

$$Array = \begin{bmatrix} 0 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \\ 8 & 9 & 10 & 11 \\ 12 & 13 & 14 & 15 \end{bmatrix}$$

then

$$utArray = \begin{bmatrix} 1 & 2 & 3 & 6 & 7 & 11 \end{bmatrix}$$

Here the yellow highlighted elements are processed by thread 0, and the green highlighted elements are processed by thread 1.

In order for the program to be as scalable as possible, the user has the ability to enter the size of the square matrix and the number of threads to be used. For OpenMP, dynamic teams are disabled which is equivalent to setting the the environmental variable to `false`. This allows the user to specify and define the desired number of threads that must be used [5].

C. `transposeBlock`

This is the function that executes the transpose of each upper triangle element. It takes as arguments the starting point index for each thread, `sp`, and the number of elements per thread, `alpha`. It then takes the index value from `utArray` (using `utArr_ptr`) and gets the corresponding row (`r`) and column (`c`) values in `Array`. Using this row and column value, the element is then transposed.

IV. PSEUDO CODE

The pseudo code for the entire program is seen in Appendix A. Although written in one file, the pseudo code is done in such a way that each method/function is split into their own algorithms. Algorithm 1 in Appendix A is the defined `struct` for the thread. Algorithm 2 in Appendix A is the function to transpose the matrix. Algorithms 3, 4 and 5 are all within the `main` class. Algorithms 4 and 5 were commented out during testing so that each individual time could be recorded for the results presented in Table I.

V. FINAL CODE AND OUTPUT

The final programs were coded in *C* on an *Ubuntu* system and compiled using a custom makefile. Table I shows the times recorded for each of the square matrices defined in Section I. Each matrix was tested three times and the average time taken to transpose the matrix using no threading, PThread and OpenMP was recorded. The time is recorded in milliseconds and is tested on a machine with the following specifications.

A. *Testing Machine*

An ASUS N550JV high performance laptop was used for testing. The machine has an *Intel(R) Core(TM) i7-4700HQ CPU* with $12GB$ of RAM installed with a 64-bit operating system installed. The operating system installed is *Windows 10 Pro* however for testing, a virtual machine was installed with the *Ubuntu 16.04.3* 64-bit installed. $5GB$ of the total RAM is dedicated to the virtual machine. The following code was run in the terminal to determine the total number of threads in a shared memory space:

```
cat /proc/sys/kernel/threads-max
```

The output of this command was 38255.

TABLE I
TABLE SHOWING TIME TAKEN TO TRANSPOSE THE MATRICES USING NO THREADING, PTHREAD AND OPENMP

N0 = N1	128	1024	8192
No threading			
	0.03833	6.552	1439.045
PThread			
4	0.211	0.118	0.163667
8	0.370667	0.191667	0.254667
16	0.459333	0.339667	0.332
64	1.390667	1.45333	1.622667
128	4.793667	19.79	571.2143
OpenMP			
4	0.273333	16.899	2735.027
8	0.413333	17.605	2740.109
16	0.752333	19.877	2729.007
64	2.405	19.732	2750.453
128	4.99866	21.541	2749.371

Figure 3 in Appendix B shows the time recorded in milliseconds for each test run and the average of each set of tests are calculated. Figures 4, 5 and 6 in Appendix B show the plotted time for the various matrices and the different thread counts for both PThread and OpenMP.

Theoretically, OpenMP should be faster than PThreads however as can be seen from the results presented in Table I and the entirety of Appendix B PThread have shown to be a better method of parallel programming. For the matrix of size 128, due to the small size of the matrix, as the number of threads increased, so did the time taken to transpose the matrix, as can be seen in Figure 4 in Appendix B.

When $N_0 = N_1 = 1024$, the PThread yielded better times than OpenMP, however after 8 threads were allocated, the time taken to transpose the matrix increased drastically, as can be seen in Figure 5 in Appendix B.

For the matrix where $N_0 = N_1 = 8192$, the PThread yielded the best results. As the number of threads increased the time decreased, however after 64 threads, the time began to increase, however after using 128 threads, the time was still significantly less than when no threading was done. This can be seen in Figure 6 in Appendix B.

VI. DIFFERENT METHODS

A. Splitting the Matrix

Since parallel computing can be used, another method that can be used to transpose a matrix would be to split the matrix into smaller matrices and giving each thread a smaller matrix to transpose. Figure 1 gives an example an 8×8 matrix that is split into four smaller matrices and 4 threads would be applied to this. Each thread handles a smaller 4×4 matrix and processes the matrix and calculates the transpose.

Although an efficient method, this would be poor on the memory. The reason for this is that an index would have to be created in order to track which elements have already been transposed. This is a trade off of this method, where efficiency is improved however memory is wasted.

$$\left[\begin{array}{cccc|cccc}
a[0][0] & a[0][1] & a[0][2] & a[0][3] & a[0][4] & a[0][5] & a[0][6] & a[0][7] \\
a[1][0] & a[1][1] & a[1][2] & a[1][3] & a[1][4] & a[1][5] & a[1][6] & a[1][7] \\
a[2][0] & a[2][1] & a[2][2] & a[2][3] & a[2][4] & a[2][5] & a[2][6] & a[2][7] \\
a[3][0] & a[3][1] & a[3][2] & a[3][3] & a[3][4] & a[3][5] & a[3][6] & a[3][7] \\
\hline
a[4][0] & a[4][1] & a[4][2] & a[4][3] & a[4][4] & a[4][5] & a[4][6] & a[4][7] \\
a[5][0] & a[5][1] & a[5][2] & a[5][3] & a[5][4] & a[5][5] & a[5][6] & a[5][7] \\
a[6][0] & a[6][1] & a[6][2] & a[6][3] & a[6][4] & a[6][5] & a[6][6] & a[6][7] \\
a[7][0] & a[7][1] & a[7][2] & a[7][3] & a[7][4] & a[7][5] & a[7][6] & a[7][7]
\end{array}\right]$$

Fig. 1. An 8×8 matrix that is split into smaller 4×4 matrices

B. Diagonal

This method uses the diagonals of the matrix. Figure 2 shows how threads would be assigned. The red diagonal is the main diagonal and the threads must work around this diagonal. A single thread will be assigned to one of the diagonals highlighted in blue. These two diagonals will be swapped. Another thread will be assigned to the green diagonals and once again will be swapped, and so on.

This method is efficient however could be harsh on memory as an index will be needed in order to determine which diagonals have been swapped. This method would be not be as efficient as other methods as the first thread would be assigned to the longest diagonals and the later threads would have less elements to process. Since each thread will not be given the same amount of data to process and some threads would be completed before others.

VII. CONCLUSION

Parallel programming was explored throughout this report using PThread and OpenMP. A method for transposing a matrix was implemented using the upper triangle of the matrix and swapping the elements with its relevant counterparts in the lower triangle of the matrix. It was seen that PThread was the better option for parallel programming although OpenMP theoretically should have been better. The results yielded were not as anticipated, however the implementation was as efficient as possible and scalable. The coding was done in C on an $Ubuntu$ operating system and was executed using a makefile in the command line prompt.

$$\begin{bmatrix}
a[0][0] & a[0][1] & a[0][2] & a[0][3] & a[0][4] & a[0][5] & a[0][6] & a[0][7] \\
a[1][0] & a[1][1] & a[1][2] & a[1][3] & a[1][4] & a[1][5] & a[1][6] & a[1][7] \\
a[2][0] & a[2][1] & a[2][2] & a[2][3] & a[2][4] & a[2][5] & a[2][6] & a[2][7] \\
a[3][0] & a[3][1] & a[3][2] & a[3][3] & a[3][4] & a[3][5] & a[3][6] & a[3][7] \\
a[4][0] & a[4][1] & a[4][2] & a[4][3] & a[4][4] & a[4][5] & a[4][6] & a[4][7] \\
a[5][0] & a[5][1] & a[5][2] & a[5][3] & a[5][4] & a[5][5] & a[5][6] & a[5][7] \\
a[6][0] & a[6][1] & a[6][2] & a[6][3] & a[6][4] & a[6][5] & a[6][6] & a[6][7] \\
a[7][0] & a[7][1] & a[7][2] & a[7][3] & a[7][4] & a[7][5] & a[7][6] & a[7][7]
\end{bmatrix}$$

Fig. 2. An 8×8 matrix that swaps diagonals

REFERENCES

[1] Barney, B; Livermore, L; *POSIX Threads Programming*; https://computing.llnl.gov/tutorials/pthreads/; Last Accessed: 01/03/2018

[2] Ball, M; *c - Pthreads vs. OpenMP - Stack Overflow*; https://stackoverflow.com/questions/3949901/pthreads-vs-openmp; Last Accessed: 01/03/2018

[3] Barney, B; Livermore, L; *OpenMP*; https://computing.llnl.gov/tutorials/openMP/; Last Accessed: 01/03/2018

[4] Unknown; *What is Amdahl's Law? - Definition from Techopedia*; https://www.techopedia.com/definition/17035/amdahls-law; Last Accessed: 02/03/2018

[5] Iliev, H; *c++ - OpenMP set_num_threads() is not working - Stack Overflow*; https://stackoverflow.com/questions/11095309/openmp-set-num-threads-is-not-working/11096742#11096742; Last Accessed: 04/03/2018

APPENDIX A
PSEUDO CODE

Algorithm 1 Struct `thread_data`

int $thread\_id$
refToInteger arr_ptr
refToInteger $utArr\_ptr$
int sp
int $Alpha$
int $arraySize$

Algorithm 2 `transposeBlock` Function

Input: refToVoid $threadarg$

refToStruct my_data
$my\_data \leftarrow$ refToStruct $threadarg$
int $temp$

for $row = my\_data$ refTo sp to my_data refTo $+my\_data$ refTo $Alpha$ **do**
  $r \leftarrow my\_data$ refTo $utArr\_ptr[row]$ / my_data refTo $arraySize$
  $c \leftarrow my\_data$ refTo $utArr\_ptr[row]$ % my_data refTo $arraySize$
  $temp \leftarrow my\_data$ refTo $arr\_ptr[r * my\_data$ refTo $arraySize + c]$
  my_data refTo $arr\_ptr[c * my\_data$ refTo $arraySize + r] \leftarrow temp$
end for

$pthread\_exit(NULL)$

Algorithm 3 `main` Function

Input: $option$ and $size$

int $size$
int $temp$
char $option[20]$
double $time\_spent$

Display "Enter the library you want to use (No, PThread, OpenMP): "
$option \leftarrow$ user input
Display "Enter matrix size : "
$size \leftarrow$ user input

$num\_elements \leftarrow size * size$
$delta \leftarrow (num\_elements - size)/2$
refToInteger $array$
$array \leftarrow$ refToInteger $malloc(sizeof($refToInteger $num\_elements))$

if $array = NULL$ **then**
  Display "malloc failed"
  $exit(1)$
end if

$utIndex \leftarrow 0$
for $diag = 0$ to $size - 1$ **do**
  for $col = diag + 1$ to $size$ **do**
    $utArray[utIndex] \leftarrow size * diag + col$
    $utIndex + +$
  end for
end for

Display "$time\_spent$ ms"

Algorithm 4 Normal Transposition method without threading within the `main` Function

if $option =$ "No" **then**
  Begin $clock$

  for $d = 0$ to $size - 1$ **do**
    for $col = d + 1$ to $size$ **do**
      $temp \leftarrow array[d * size + col]$
      $array[d * size + col] \leftarrow array[col * size + d]$
      $array[col * size + d] \leftarrow temp$
    end for
  end for

  End $clock$
  $time\_spent \leftarrow$(double)$(end - begin) * 1000/CLOCKS\_PER\_SEC$
end if

Algorithm 5 PThread Transposition method within the `main` Function

Input: $Threads$

if $option =$ "PThread" **then**

  int $Threads$
  Display "Enter number of threads : "
  $Threads \leftarrow$ user input
  $alpha \leftarrow delta/Threads$
  $pthread\_t$:
    begin
      $threads[Threads]$
    end
  thread_data $td[Threads]$

  Begin $clock$

  for $i = 0$ to $delta$ with increments of $i+ = alpha$ **do**
    Display "main() : creating thread" $i/alpha$

    $td[i/alpha]$ refTo $thread\_id \leftarrow i/alpha$
    $td[i/alpha]$ refTo $sp \leftarrow i$
    $td[i/alpha]$ refTo $Alpha \leftarrow alpha$
    $td[i/alpha]$ refTo $arr\_ptr \leftarrow array$
    $td[i/alpha]$ refTo $utArr\_ptr \leftarrow utArray$
    $td[i/alpha]$ refTo $arraySize \leftarrow size$
    $pthread\_create$:
      begin
        $threads[i/alpha]$
        $NULL$
        $transposeBlock$
        refToVoid $td[i/alpha]$
      end
  end for

  End $clock$
  $time\_spent \leftarrow$(double)$(end - begin) * 1000/CLOCKS\_PER\_SEC$
end if

Algorithm 6 OpenMP Transposition method within the `main` Function

Input: $threads$

if $option$ = "OpenMP" **then**

int r
int c
int $threads$
Display "Enter number of threads : "

Begin $clock$

$omp\_set\_dynamic(0)$
$omp\_set\_num\_threads(threads)$

$pragma\ omp\ parallel$:
$pragma\ omp\ for$:

for $i = 0$ to $delta$ **do**
$r \leftarrow utArray[i]/size$
$c \leftarrow utArray[i]\%size$

$temp \leftarrow array[r * size + c]$
$array[r * size + c] \leftarrow array[c * size + r]$
$array[c * size + r] \leftarrow temp$
end for

End $clock$
$time\_spent \leftarrow$(double)$(end - begin) * 1000/CLOCKS\_PER\_SEC$
end if

APPENDIX B
RESULTS

Number of Thread	Pthread Run 1	Pthread Run 2	Pthread Run 3	Pthread Average	OpenMP Run 1	OpenMP Run 2	OpenMP Run 3	OpenMP Average
				128				
0	0.038	0.039	0.038	0.038333	0.038	0.039	0.038	0.038333
4	0.253	0.117	0.263	0.211	0.268	0.28	0.272	0.273333
8	0.557	0.366	0.189	0.370667	0.364	0.374	0.502	0.413333
16	0.444	0.603	0.331	0.459333	0.721	0.837	0.699	0.752333
64	1.496	1.356	1.32	1.390667	2.407	2.241	2.567	2.405
128	4.751	4.66	4.97	4.793667	5.078	4.476	5.442	4.998667
				1024				
0	7.321	6.513	5.822	6.552	7.321	6.513	5.822	6.552
4	0.119	0.117	0.118	0.118	16.224	17.222	17.251	16.899
8	0.193	0.193	0.189	0.191667	18.686	15.96	18.169	17.605
16	0.346	0.342	0.331	0.339667	21.391	17.623	20.617	19.877
64	1.479	1.548	1.333	1.453333	18.918	22.489	17.789	19.732
128	20.809	19.507	19.054	19.79	22.415	21.044	21.164	21.541
				8192				
0	1438.224	1455.841	1423.071	1439.045	1438.224	1455.841	1423.071	1439.045
4	0.117	0.121	0.253	0.163667	2692.089	2758.751	2754.241	2735.027
8	0.381	0.188	0.195	0.254667	2716.532	2729.243	2774.552	2740.109
16	0.331	0.337	0.328	0.332	2702.464	2768.123	2716.434	2729.007
64	1.795	1.503	1.57	1.622667	2714.906	2770.522	2765.932	2750.453
128	490.847	516.387	706.409	571.2143	2749.76	2742.924	2755.43	2749.371

Fig. 3. Recorded times for each test on the program

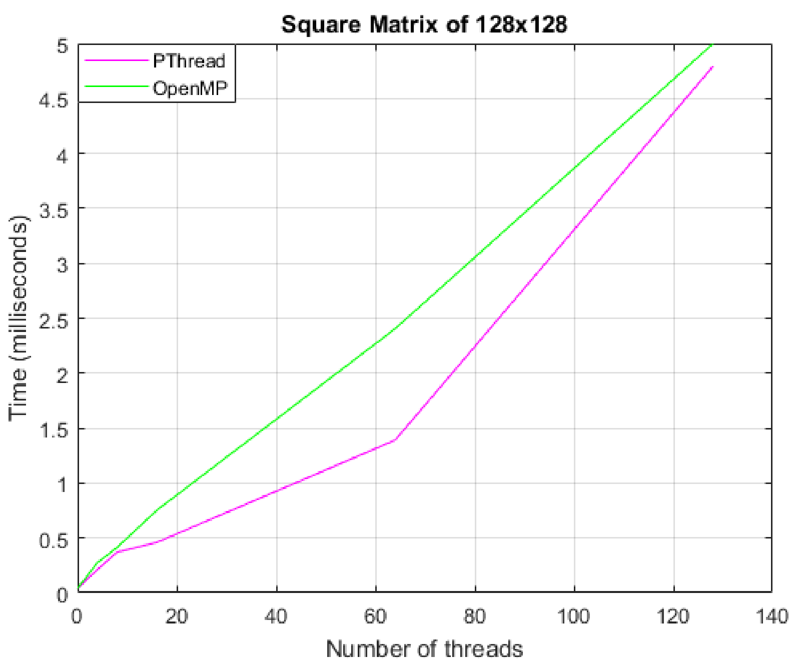

Fig. 4. Plotted average time for each thread for the square matrix where $N_0 = N_1 = 128$

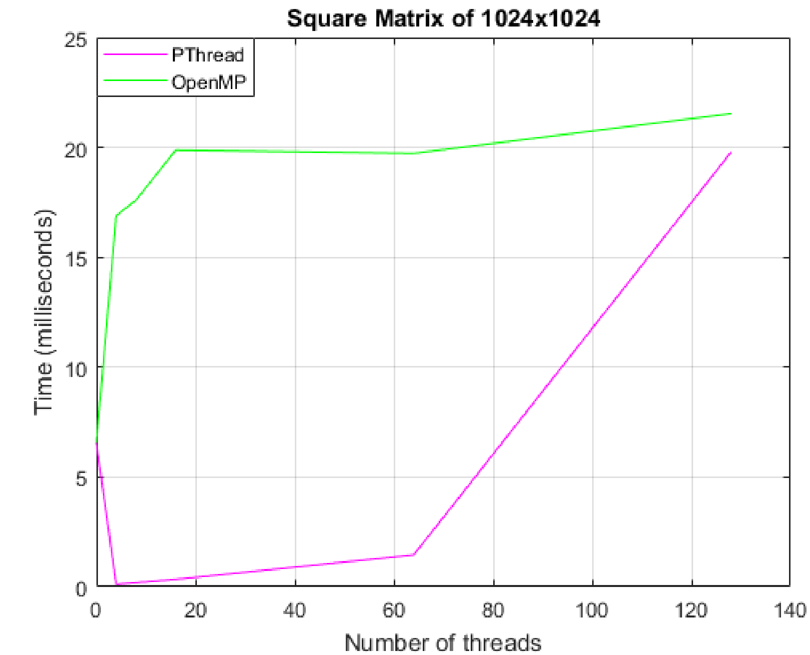

Fig. 5. Plotted average time for each thread for the square matrix where $N_0 = N_1 = 1024$

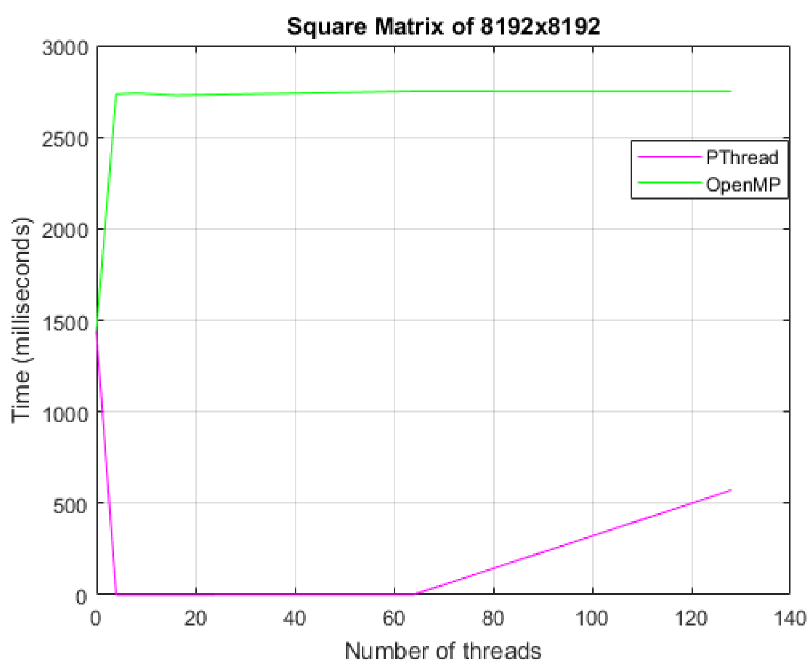

Fig. 6. Plotted average time for each thread for the square matrix where $N_0 = N_1 = 8192$