Lab2:Matrix Transposition using parallel computing

Gift Langa (1043882) *, Pierre Nayituriki (1045397) †,
Zanele Malinga (0706885N) ‡, Mokhulwane Nchabeleng (1117206) §
March 15, 2018
ELEN4020: Data Intensive Computing
School of Electrical and Information Engineering, University of the Witwatersrand, Johannesburg 2050, South Africa

I. Introduction

This paper presents the implemantion and the testing of a parallel algorithm for transposing an N by N matrix using the C language. The already existing algorithm for transposing a square matrix has the computational complexity of O(n^2). Parallel implementation of this algorithm enhences the performance of this algorithm to complexity of O(n). In this this lab the same algorithm is implemented using twice with two different C supported libraries i.e the OMP library and the PThread library. The OMP library offers more flexibility and simplicity since no explicit instances of a thread model is required. The user just define a a block of code and the desired number of threads require parallelization then Omp takes care of the rest. Unlike with the Omp library, the PThread library is low level threading API which require explicit construction and management of threads object which results in a higher code implementation complexity than the other library. The two algorithm were implemented by the group and tested for different matrix sizes and threads. The results discussion is presented in section 3 which follow the algorithm implemention section.

II. Algorithm Implementation

The traditional algorithm for transposing an N by N matrix traverses through the upper/lower triangle using two nested loop and sumultenously swaping the visited element with its corresponding pair in the lower/upper triangle according to transposing Mathematical rule. This algorithm is presented by the pseudocode below:

```
for n = 0 to N - 2
    for m = n + 1 to N - 1
        swap A(n,m) with A(m,n)
```

Fig. 1. general O(n2) matrix transposition pseudocode

Omp Parallel implementation The above algorithm was modified as according to the Omp pseudocode presented below. The function accept three parameters: Matrix size, number of threads used and the Matrix in a pointer form.

A. Parallelism using OpenMP

The above algorithm was modified as according to the Omp pseudocode presented below. The function accept three paramaters: Matrix size, number of threads used and the Matrix in a pointer form.

```
Algorithm1: omp_TransposeMatrix ( sizeN,nThreads,Matrix_ptr)
begin:
    set_num_threads_used (nThreads)
    begin Parallel Section:
        OmpFor j ←0 to sizeN-1 do
            For i ←j+1 to sizeN-1 do
                location1 ←j + i * sizeN
                location2 ←i + j * sizeN
                ptrSwapElement (location1, location2)
            EndFor
        EndOmpFor
    EndpParallel
EndAlgorithm
```

Fig. 2. Omp pseudo-code

Omp divides the outer loop according to the number of threads used (although the division of the loop iterations is not always equal distributed) and then each thread execute the inner loop in a parallal fashion. The shifting and reallocation of matrix (which will result in poor performance) is avoided by using Pointers based design. Using the fact that the 2D array is stored in a sequential manner row major, the J and I counters together with the Matrix pointer are used to map the actual memory locations of the element pairs to be swapped as shown in the pseudocode above. A pointer based swapping function is then used to swap the values of this elements.

B. Parallelism using pThreading

The pThread implementation is quite complicated due to its low level design and objects, hence a detailed flow chart of the implementation is provided below in Figure 3. In simple term this approach align the tread from the top-left corner of the matrix diagonally going downwards of which each thread perform all the transposition in its column and row. Although this algorithm do work, its how ever not the optimal design because it means that thread in the lower row has more work to do than the threads assigned to upper rows.

III. Results

The algorithms were tested with different input sizes of 128,1024 and 8192 and for each matrix size a different number of threads were using to in order to observe and try to understating the performance of the implemented algorithms. The following table of results represents the amount of time taken to thread different size matrices using different number of threads.

TABLE I
COMPUTATION TIME(S) OF THE OMP AND PTHREAD ALGORITHMS

size	**128**		**1024**		**8192**	
Threads	**Omp**	**pThread**	**Omp**	**pThread**	**Omp**	**pThread**
4	0.001660	0.000000	0.010004	0.000001	0.858893	0.000002
8	0.004328	0.000000	0.010735	0.000000	0.776556	0.000000
16	0.001123	0.000001	0.007966	0.000002	0.624264	0.000002
32	0.002106	0.000002	0.008700	0.000003	0.615456	0.000004
64	0.004393	0.000005	0.010097	0.000005	0.607575	0.000008
128	0.008617	0.000010	0.014308	0.000011	0.589871	0.000015

From this results in the table is clearly observable that the performance of the Omp has poor performance compared to the pThread design. It was also expected to have better perform when using many threads, which however turn out to be otherwise according to the results in table. It is believed that the cause of this trend could be due to the fact that Omp has to do the thread management task such auto partition of the loop iterations. The factor which could have resulted in this trend could be because of the of how the algorithms are design (i.e. transposing row and column) which in physical memory are not usually located near each when using row major representation of 2D array. Also from the data its observable that the computational time fluctuate a lots which could probably b as the results of other process that the processor is executing. The Omp implementation show the expected computation trend for the on size of 8192 which is a linear decrease in the computational time as the number of threads increase.

IV. Conclusion

OpenMP provide an easy way to implement parallel programs than pThread although it doesn't really provides one with a better understanding of how threads work. Test results are presented in the previous section which shows that the pThread algorithm performs better than the OpenMP program. The computation time increase with the increasing number of threads which is the opposite of what's expected, the actual reason why the program is behave like this is not known at the moment. The developers of this program are currently researching and investigating on this unexpected behaviour to gain a better understanding of the working of threads of which is later going to be used later to improve the program and perform more testing.

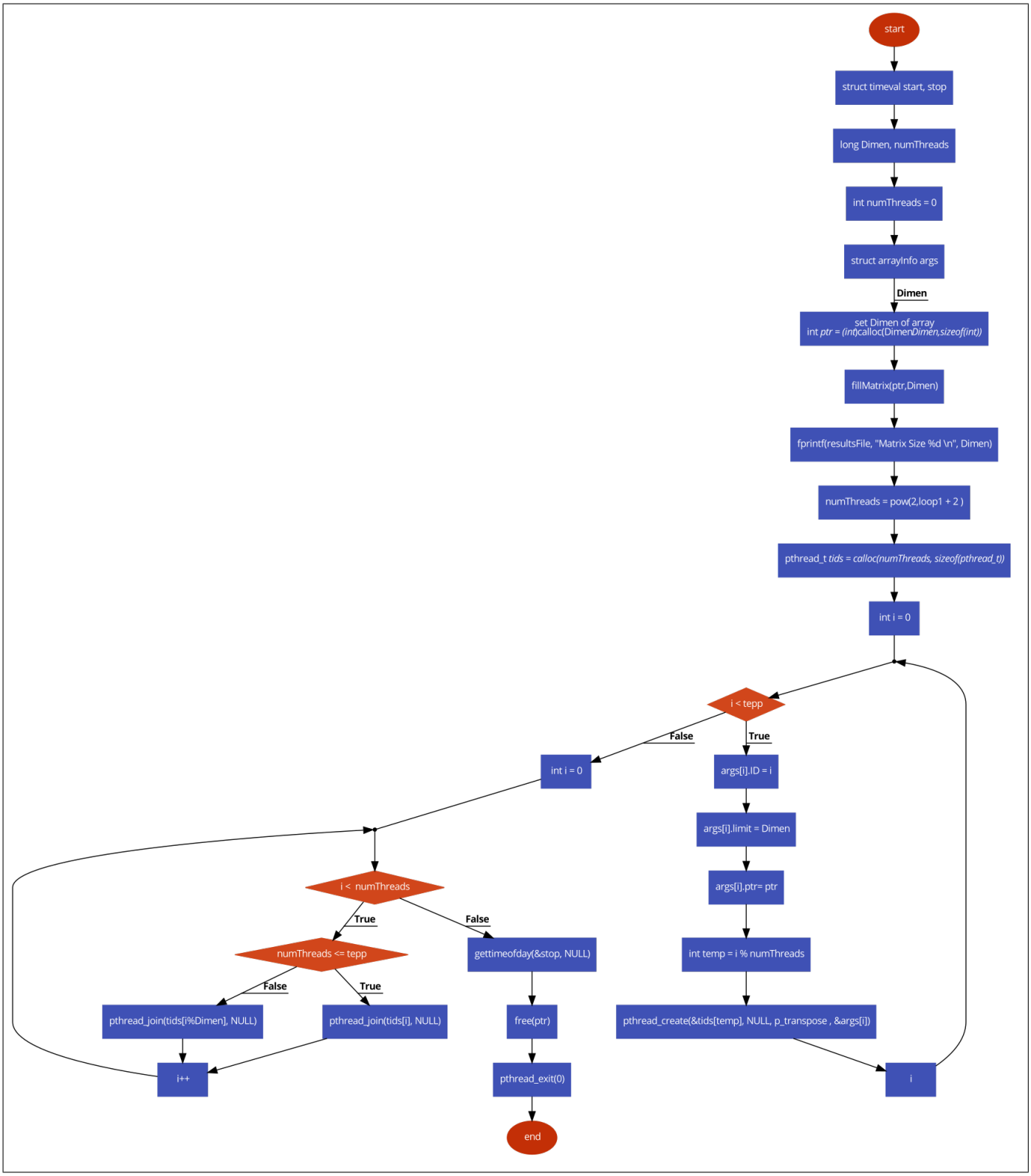

Fig. 3. Flow diagram indicating the pThread code implementation.