In-place Multithreaded Square Matrix Transpose

Sanele Ndlovu-716411
Knowledge Dzumba - 813137
School of Electrical and Information Engineering, University of the Witwatersrand, Johannesburg
ELEN4020A:Data Intensive Computing in Data Science

1 Introduction

This report presents the results of a programming laboratory work on shared memory programming using OpenMP and Pthread libraries. The purpose of the laboratory work was to explore the functionality of these libraries for simple scientific applications of manipulating very large matrices which are maintained in memory. The matrix operation of focus was in-place transposition, achieved through three different parallel algorithms. The performance of these algorithms was evaluated by observing the time taken to perform the transposition and benchmarking it against the performance of a basic serial algorithm. The sections that follow provide pseucodes for the different algorithms used as well as a brief description of how the algorithms work and a comparative table of performance for the different algorithms implemented using the two different shared programming libraries.

2 Basic Non-threaded algorithm

Algorithm 1 computes the transpose of the input matrix A by first creating a secondary matrix B and then interchanging the rows and columns of elements of A and assigning them to the matrix B. Matrix B is then returned as the transposed version of A. In this algorithm, the original matrix and its transposed version are stored in different memory locations since a new memory allocation had to be made for the returned matrix B.

Algorithm 1: Basic matrix transpose

Input: n x n matrix A
Output: n x n matrix B

```
1 n = A.rows
2 Let B be a newly created n x n matrix
3 for i = 0 to n - 1 do
4 |   for j = 0 to n - 1 do
5 |   |   B[i][j] = A[j][i]
6 |   end
7 end
8 return B
```

3 OpenMP Naive Threaded Algorithm

Algorithm 2 Computes the transpose of a matrix A in-place, which means that the same memory locations that stored the input matrix are used to store the transposed matrix without an allocation for a new matrix. The nested **for** loops in the algorithm iterate through the main diagonal of the matrix, swapping the elements *A[i][j]* with the elements *A[j][i]*. The result of this operation is a transposition of the actual matrix A. Using the OpenMP directive **pragma omp parallel for**, the nested loops are automatically made to be executed by multiple threads if the algorithm is compiled as an OpenMP program. This is because the **pragma omp parallel for** directive causes multiple threads to be forked at the point of execution, with the multiple threads dividing the workload of the loops.

Algorithm 2: Naive OpenMp matrix transpose

```
Input: n x n matrix A
Output: n x n matrix B
for i = 1 to n - 2 do in parallel
    for j = 1 + 1 to n - 1 do in parallel
        swap(A[i][j], A[j][i])
    end
end
return A
```

4 OpenMP Diagonal Threaded Algorithm

Algorithm 3 computes the transpose of a matrix *A* in-place. For each diagonal element *A[i][i]*, the algorithm generates 4 threads that swap the elements on the right of the diagonal element with the corresponding elements below the diagonal element. Thread 0 starts by swapping the elements immediately right and below the diagonal element, thread 1 swaps the elements 2 index positions further from the index position, thread 2 swaps elements 3 index positions and thread 3 swaps elements that are 4 index positions from the diagonal index. When thread 0 is done swapping its elements, it continues to swap the elements 5 positions from the index position (that is, you add $numberOfThreads$ to the previous index to get next position swapped by this thread). The other threads follow the same pattern until the row and column for the i^{th} diagonal element are completely interchanged. A consequence of this algorithm is that the size of the array should be divisible by the number of threads used, which in the case of the lab worked well because all the matrix sizes were divisible by 4. The outer **for** loop ensures that the swapping operation is performed for all diagonal elements(its loop variable determines the diagonal element at which the threads are operating) inner loop performs the actual swapping of elements.

Algorithm 3: OpenMP diagonal threaded matrix transpose

```
Input: n x n matrix A
Output: n x n matrix B
for i = 1 to n do
    start ↩ threadNumber + i + 1
    increament ↩ numberOfThreads
    for j = start to n do in parallel
        swap(A[index][j],A[j][index])
        j += increament
    end
end
return A
```

5 Pthread Diagonal Thread Algorithm

Algorithm 4 performs matrix transposition in place, in a similar manner to that of **Algorithm 3**, the difference being that this algorithm is parallelized using Pthreads instead of OpenMP. While threads are forked through the **pragma omp parallel** for **Algorithm 3**, the creation and behaviour of threads is hard coded for this algorithm.

Algorithm 4: Pthread diagonal threaded matrix transpose

Input: *n* x *n* matrix *A*
Output: *n* x *n* matrix *B*
for *i = 1 to n* **do**
  $start \hookleftarrow threadNumber + i + 1$
  $increament \hookleftarrow numberOfThreads$
  for j = $start$ *to n* **do in parallel**
    swap(A[index][j],A[j][index])
    $j \mathrel{+}= increament$
  end
end
return A

6 Algorithm Performance

The algorithms mentioned above were compiled on a Lenovo Ideapad 320, core i5 laptop with a 4GB RAM and 4 2.5GHz processor cores. For each algorithm, an $N_0 \times N_1$ square matrix was used as the input to be transposed for varying values of N_0 as indicated in TABLE 1. It was required that the input matrix sizes be up to 4096, but to test the scalability of the algorithms for larger inputs, the algorithms were tested with input sizes up to 16384.

TABLE 1: Timing results for parallel libraries

$N_0 = N_1$	Basic	Pthreads	OpenMP	
		Diagonal	Naive	Diagonal
128	0.000271	0.024574	0.02591	0.003667
1024	0.011464	0.060502	0.00551	0.012012
2048	0.035667	0.119387	0.020867	0.042493
4096	0.155627	0.271844	0.079695	0.119827
8192	0.712869	0.816841	0.376895	0.408006
16384	no terminate	4.90293	3.96019	2.4639