Parallel and Distributed computing: Lab Assignment Two

Mustapha Tidoo Yussif

February 17Th, 2020

1 Introduction

The main purpose of this assignment is to learn to use shared memory programming libraries, Pthread and OpenMP, for transposing large square matrix in memory. This text explains four implementations: diagonal threading algorithm implementation, Naive matrix transposition with OpenMP, block oriented transposition of matrix with both pthreads and OpenMP.

1.1 Pthreads Implementations

This section lists and explains the pthreads implementations.

1.1.1 A Diagonal-Threading Algorithm implementation

I used the SPMD design pattern to implement the diagonal-threading algorithm. I used two nested loops to implement this where the outer loop goes through the diagonal elements and the inner loop swaps the row elements at each diagoal element to the corresponding column element. Based on the number of threads, the outer loop interations are shared among the threas. I used formulae to find the range of iterations each thread is going to execute given its id/rank. The formulae are

$$start = thread\_id * \frac{num\_iterations}{num\_threads}$$

$$end = (thread\_id + 1) * \frac{num\_iterations}{num\_threads} - 1$$

For example if the number of threads are 4, and there are 8 iterations to handle, the following shows the breakdown of the range of iterations to be handled by each thread.
Thread = 0, start = 0, end = 1
Thread = 1, start = 2, end = 3
Thread = 2, start =4 , end = 5,
Thread = 3, start = 6, end = 7.

Algorithm 1: A Diagonal-Threading Algorithm implementation

```
Input: matrix, N

Extract parameters for threads.
 $start \leftarrow data- > start$;
$end \leftarrow data- > end$;
$mat \leftarrow data- > matrix$;
$rank \leftarrow data- > thredId$;
$matSize \leftarrow data- > start$;

for ( $i = start$; $i < end$; $i = i + 1$ ) {
    for ( $j = i + 1$; $i < N$; $j = j + 1$ ) {
        swap(matrix, i, j);
    }
}
```

1.1.2 A Block-Oriented Transposition.

This implementation is suitable for larger matrices. The matrix is seen as a big matrix that contains other blocks of matrices within it. This routine solves the problem in chunks and with many swaps. I used nested loops to go through the blocks as well as the element in each blocks. The outer loop iterates up to the number of blocks in the matrix. In the outer loop, I have nested inter loops that interchange A[i][j] and A[j][i]. Basically, the inner nested loops work like the naive matrix transposition alogorithm. I then added the parallel and worksharing for directives of OpenMP to automatically parallelize the code. In pthreads however, I had to use the SPMD pattern by manually sharing the blocks among the available threads.

1.2 OpenMP Implementations

Below are the implementations of the matrix transposition algorithms using OpenMP, an Application Program Interface (API) for developing shared memory parallel applications.

1.2.1 A Naive-OpenMP Algorithm

This algorithm is the OpenMP threading version of the simple matrix transpose algorithm. The simple transpose matrix algorithm uses simple nested two loops to swap elements at index i, and j. For example, the values $A[i][j]$ will be swapped with $A[j][i]$. The Open parallel and worksharing for directive are then added to allow the algorithm handle by multiple threads. This implementation is done in such a way that the diagonal elements will not be swapped to reduce unnecessary swappings. The pseudocode of the routine is shown below:

Algorithm 2: A Block-Oriented Transposition.

```
Input: matrix, N

if $N\%blocksize\ !{=}0$ then
    Matrix must be a multiple of blocksize and exit;
end

for ( $block = 0;\ i < N;\ block = block + BLOCK\_SIZE$ ) {
    for ( $i = block;\ i < block + BLOCK\_SIZE;\ i = i + 1$ ) {
        for ( $j = i + 1;\ i < block + BLOCK\_SIZE;\ j = j + 1$ ) {
            swap(matrix, i, j);
        }
    }
    for ( $i = block + BLOCK\_SIZE;\ i < N;\ i = i + 1$ ) {
        for ( $j = block;\ i < block + BLOCK\_SIZE;\ j = j + 1$ ) {
            swap(matrix, i, j);
        }
    }
}
for ( $i = block;\ i < N;\ i = i + 1$ ) {
    for ( $j = i + 1;\ i < N;\ j = j + 1$ ) {
        swap(matrix, i, j);
    }
}
```

Algorithm 3: A Naive-OpenMP Algorithm

```
Input: matrix, N
#pragma omp parallel for nowait
for ( $i = 0;\ i < N;\ i = i + 1$ ) {
    for ( $j = i + 1;\ i < N;\ j = j + 1$ ) {
        swap(matrix, i, j);
    }
}
```

1.2.2 A Block-Oriented OpenMP Transposition.

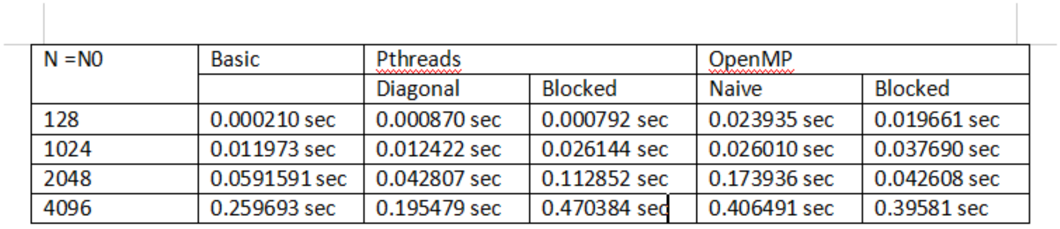
Algorithm 4: A Block-Oriented Transposition.

```
Input: matrix, N

if N%blocksize !=0 then
    Matrix must be a multiple of blocksize and exit;
end

#pragma omp parallel for private(i, j, block) schedule(static,
 CHUNK$_SIZE$)
for ( $block = 0;\ i < N;\ block = block + BLOCK\_SIZE$ ) {
    for ( $i = block;\ i < block + BLOCK\_SIZE;\ i = i + 1$ ) {
        for ( $j = i + 1;\ i < block + BLOCK\_SIZE;\ j = j + 1$ ) {
            swap(matrix, i, j);
        }
    }
    for ( $i = block + BLOCK\_SIZE;\ i < N;\ i = i + 1$ ) {
        for ( $j = block;\ i < block + BLOCK\_SIZE;\ j = j + 1$ ) {
            swap(matrix, i, j);
        }
    }
}
for ( $i = block;\ i < N;\ i = i + 1$ ) {
    for ( $j = i + 1;\ i < N;\ j = j + 1$ ) {
        swap(matrix, i, j);
    }
}
```

1.3 Comparative table of the performance of the algorithms



N =N0	Basic	Pthreads		OpenMP	
		Diagonal	Blocked	Naive	Blocked
128	0.000210 sec	0.000870 sec	0.000792 sec	0.023935 sec	0.019661 sec
1024	0.011973 sec	0.012422 sec	0.026144 sec	0.026010 sec	0.037690 sec
2048	0.0591591 sec	0.042807 sec	0.112852 sec	0.173936 sec	0.042608 sec
4096	0.259693 sec	0.195479 sec	0.470384 sec	0.406491 sec	0.39581 sec