# Introduction to Parallel and Distributed Computing:

# Lab Assignment 2

Kweku Andoh Yamoa(71712022)

March 10, 2021

# 1 Introduction

The main purpose of this laboratory exercise was to refresh and learn the use of C and then explore, learn and apply shared memory programming libraries, i.e., Pthread and OpenMP, for simple scientific applications of manipulating very large matrices that are maintained in memory.

# 2 Overview

To achieve the outcomes of the assignment matrix transposition was the challenge utilised in exploring shared memory libraries. Matrix transposition was done in-place using the basic(brute force) version of the algorithm. After, parallelisation was added to see how shared memory operations optimised memory. Finally, another version of the algorithm was implemented with the use of shared memory libraries. This other algorithm was called Diagonal-Threading.

# 3 Basic Algorithm

The basic algorithm uses two nested loops to at index i, and j. For example, the element at A[i][j] will be swapped with the element A[j][i].

### 3.1 Pseudocode

**Algorithm 1:** transposeBasic(matrix, size)

```
Brief:  Transposes a given N x N matrix inplace
Input:matrix, size(where size = N)
Output:  Transpose of the given matrix i.e matrix^T
```

**for** $i \leftarrow 0$ **to** *size* **do**
  **for** $j \leftarrow i + 1$ **to** *size* **do**
    swap(matrix,i,j);
  **end**
**end**

## 4 Pthreads Diagonal

This section lists and explains diagonal threading using pthreads for matrix transposition.

### 4.1 Overview

I used the SPMD(Single Program Multiple Data) design pattern to implement the diagonal-threading algorithm. Two nested loops were used 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 elements. The outer loop iterations are shared among the threads based on the number of threads. To do this, I used formulae to find the range of iterations each thread is going to execute given its id. Formulae:

$$start = (threadId) * \frac{matrix_size}{numThreads}; \qquad end = (threadId + 1) * \frac{matrix_size}{numThreads}$$

## 4.2 Pseudocode

**Algorithm 2:** diagoalPthreads(matrix, size)

```
Brief:  Transposes a given N x N matrix inplace using pthreads for
 diagoal threading
Input:matrix, size(where size = N)
Output:  Transpose of the given matrix i.e matrix^T
```

threadId ← getThreadID()

\\compute start and end

start ← $(threadId) * \frac{matrix_size}{numThreads}$

end ← $(threadId + 1) * \frac{matrix_size}{numThreads}$

**for** $i$ ←*start* **to** *end* **do**

  **for** $j \leftarrow i + 1$ **to** *size* **do**

    swap(matrix,i,j);

  **end**

**end**

# 5 OpenMP

This section talks about naive OpenMP matrix transposition and diagonal threading transposition using OpenMP

## 5.1 Naive OpenMP

### 5.1.1 Overview

The naive OpenMP threading automatically parallelize the code by inserting #pragma before the for loops. This when compiled ensures that the same basic algorithm is now run in parallel.

### 5.1.2 Pseudocode

**Algorithm 3:** naiveOMPTranspose(matrix, size)

```
Brief:  Transposes a given N x N matrix inplace using openmp for
 parallelisation
Input:matrix, size(where size = N)
Output:  Transpose of the given matrix i.e matrix^T

nThreads
#pragma omp parallel{
  id ← omp_get_num_threads()
  if id == 0 then
  |   nThreads ← omp_get_num_thread()
  |
end
  #pragma omp for nowait
  for i ←0 to size do
  |   for j ← i + 1 to size do
  |   |   swap(matrix,i,j);
  |  end
end
}
```

## 5.2 Diagonal Threading OpenMP

This section gives an overview of diagonal threading using openmp.

### 5.2.1 Overview

Diagonal threading using OpenMP is similar to the approach explained for Pthreads Diagonal(See Section 4). The only distinct difference is for the omp implementation, the loops are parallelised using the # pragma omp for call.

### 5.2.2 Pseudocode

**Algorithm 4:** diagonalOMPTranspose(matrix, size)

```
Brief:  Transposes a given N x N matrix inplace using openmp for
 diagoal threading
Input:matrix, size(where size = N)
Output:  Transpose of the given matrix i.e matrix^T

nThreads
omp_set_dynamic(0)    \\Explicitly disable dynamic teams
omp_set_num_threads(4)    \\Use 4 threads for all consecutive parallel
 regions
#pragma omp parallel{
   id ← omp_get_num_threads()
   nThreads ← omp_get_num_thread()
   \\compute start and end
   start ← (id) * matrix_size / nThreads
   end ← (id + 1) * matrix_size / nThreads
   for i ←start to end do
   |    for j ← i + 1 to size do
   |    |    swap(matrix,i,j);
   |    end
   end
}
```

# 6 Comparative Table of Performance

The table below shows the performance of the various algorithms as implemented in the C programming language.

| $N_0$=$N_1$ | Basic | Pthreads Diagonal | OpenMP | |
|---|---|---|---|---|
| | | | Naive | Diagonal |
| 128 | 0.000355 s | 0.002989 s | 0.094637 s | 0.003679 s |
| 1024 | 0.008534 s | 0.053530 s | 0.079286 s | 0.006984 s |
| 2048 | 0.023156 s | 0.132865 s | 0.064469 s | 0.052527 s |
| 4096 | 0.099668 s | 0.210901 s | 0.24091 s | 0.159302 s |

# 7 Comments on Running Code

To run the code successfully for all the sizes of N, make sure that variable *matrix_size* is changed dynamically to the defined sizes N,N1,N2 and N3.