# CS434: Introduction to Parallel and Distributed Computing

Ndze'dzenyuy, Lemfon K.

8th March 2021

**Lab 2**

# 1 Explaining the algorithms and presenting the pseudo-code

## 1.1 Using the naive approach

Using the naive approach, we iterate through the elements of the matrix that are not on the leading diagonal and swap the elements at row $i$ column $j$ for that row $j$ column $i$. However, we must be careful that swapping is done only once, as double swapping will leave the matrix unchanged at the end. This explains why we only traverse the lower triangle of the matrix, and exploit the square nature of the matrix to swap elements appropriately. The pseudo-code for this naive approach is presented below.

```
1 ALGORITHM: transpose-basic(int ** mat, int dim)
2 for i ← 0 to dim do
3 |   for j ← 0 to i do
4 |   |   swap mat[i][j] and mat[j][i]
5 |   end
6 end
```

**Algorithm 1:** Transposing the matrix using a naive approach

## 1.2 Using OpenMP and the naive approach

Now we apply some multi-threading to the naive approach described above. We do so mainly by adding an OpenMP directive that makes the loops run in parallel. The interesting decision made here is with regards to the number of threads used. Since it can be argued that each iteration of the outer loop is independent of the other, we use $N_o threads, where N_o is the dimension of the matrix$.

```
1 ALGORITHM: naive-openmp(int ** mat, int dim)
2 insert an OpenMP directive to make the loops run in dim threads.
3 for i ← 0 to dim do
4 |   for j ← 0 to i do
5 |   |   swap mat[i][j] and mat[j][i]
6 |   end
7 end
```

**Algorithm 2:** Transposing the matrix using a naive approach and OpenMP

## 1.3 Using OpenMP and the diagonal threads approach

The diagonal thread approach realises that we could simply iterate along the leading diagonal, using as many threads as the dimension of the matrix and letting each thread exchange rows and columns appropriately. To do this, we iterate through each row. However, for each row, we only iterate from the element after the leading diagonal up till the the end of the row and swap this element with the element that is currently in the position it will occupy in the matrix. The threads can run without communicating to each other, and so we use a simple OpenMP directive to achieve parallelism.

```
1 ALGORITHM: diag-openmp(int ** mat, int dim)
2 insert an OpenMP directive to make the loops run in dim threads.
3 for i ← 0 to dim do
4 |   for j ← i to dim do
5 |   |   swap mat[i][j] and mat[j][i]
6 |   end
7 end
```

**Algorithm 3:** Transposing the matrix using the threaded approach and OpenMP

## 1.4 Using PThreads and the diagonal threads approach

The diagonal thread approach realises that we could simply iterate along the leading diagonal, using as many threads as the dimension of the matrix and letting each thread exchange rows and columns appropriately. To do this, we iterate through each row. However, for each row, we only iterate from

the element after the leading diagonal up till the the end of the row and swap this element with the element that is currently in the position it will occupy in the matrix. We create *dim* PThreads, and assign each thread a rank or a number between 0 and $dim - 1$. These numbers will help us to assign specific threads to carry out the iteration at specific positions of the leading diagonal. To achieve this, we use two functions. The first being the function that is run by every pthread, and the second function that creates the pthreads and runs them to transpose the matrix. The pseudo-code for the two functions is presented below.

```
1 ALGORITHM: pth-diag(void*rank)
2 i ← castranktoalong
3 for j ← i to dim do
4 |  swap mat[i][j] and mat[j][i]
5 end
```

**Algorithm 4:** The transposition function that will be run by each pthread

```
1 ALGORITHM: diag-pthreads(int ** mat, int dim)
2 threads = [thread0, thread1, thread2 ... thread(dim − 1)]
3 for t ← 0 to dim do
4 |  create a thread that will run the pth_diag function and give it a rank of t
5 end
6 for t ← 0 to dim do
7 |  run the dim threads and join them
8 end
```

**Algorithm 5:** Transposing the matrix using the threaded approach and PThreads

## 1.5 Using a main function to call the functions appropriately

After defining these functions, we then proceed to call them in a main function. The main is structured to take command line arguments that indicate the dimension of the matrix and whether or not the user wants the matrix to be printed out in the console. For small dimensions, the matrix can be printed out nicely in the conosle and this can help in testing the correctness of our algorithms. However, thesame cannot be said for dimensions as large as 2048. Once the user enters the dimension, we generate a square matrix with random numbers ranging betwene 0 and the specified dimension. For each of the approaches used to transpose the matrix, we start a clock before calling the associated function and end the clock once the transposition is complete. We then use the start and end of the clock to calculate the execution time for the function. Because the transposition is being done in place, it is important to transpose the matrix an additional time to make sure that we have the original matrix.

## 2 Comparing the run times

After runnining the main several times with the dimension arguments as being 128, 1024, 2048, 4096, we etablish the run times to be as presented in the table below. The time is measures in seconds, and is presented in 6 decimal places for reasons of comparison.

| $N_o = N_1$ | Basic | PThreads Diagonal | OpenMP | |
|---|---|---|---|---|
| | | | Naive | Diagonal |
| 128 | 0.000147 | 0.015245 | 0.026458 | 0.016363 |
| 1024 | 0.004931 | 0.048507 | 0.068722 | 0.036478 |
| 2048 | 0.019403 | 0.164025 | 0.151904 | 0.166329 |
| 4096 | 0.102819 | 0.467405 | 0.440250 | 1.087429 |

Looking at the table, we see that for most of the values of $N_0$, the basic solutions seems to be the most efficient. As the value of $N_0$ increases, the naive soultion with OpenMP seems to be performing better than that with diagonal threads. It also seems that the naive approach with OpenMP also gets more efficient that the diagonal approach using pthreads as the value of $N_0$ increases.

## 3 Instructions on how to run the code

Visit and clone the repository at https://github.com/WybeTuring/PDC-Labs.git. Navigate to the branch named lab2. Ensure that you are in the folder that contains the file lab2.c and then open the command line.

To compile: gcc -g -Wall -fopenmp -o lab2 lab2.c -lpthread
To run: ./lab2 <size of the matrix> <1 to print transpose, 0 not to print>