# Parallel and Distributed Computing: Homework 5

## Due May 1, 2020, in zip format on Canvas.

The main goal of this exercise is to experimentally measure the performance of the parallelizations of a data transfer process using different versions of fine grain and coarse grain muti-threading.

Consider the problem of in-memory transposition of a matrix $A$. In a single processor, the pseudo-code to in-memory transpose the matrix would be as follows:

```
For i = 0, n − 1, do:
    For j = i + 1, n − 1, do:
            temp=a_{i,j}
            a_{i,j} = a_{j,i}
            a_{j,i} = temp
        endfor
    endfor
endfor
```

Note that this code scans the matrix by rows and exchanges symmetric elements with respect to the main diagonal focusing on individual elements of the upper triangular section of the matrix $A$, excluding the main diagonal.

One way of parallelizing this process of transposing $A$ in a muti-threaded multi-core environment is to implement the matrix $A$ as a shared data structure that can be accessed by different threads. Threads may have different levels of coarseness. In a light, or **fine grain** version, each thread would access the matrix to read one element of the upper triangular $A$ and perform the exchange of that element with its main diagonal symmetric element. Elements of $A$ would be read by each thread in some sequence following a coordinated indexing scheme to guarantee each element of $A$ is accessed once and only once by only one thread.

Threads can be made heavy, or **coarse grain**, by allowing each thread to do exchanges of several elements of the upper triangular $A$ each time the thread is executed. Each thread is then in charge of exchanging/transposing an $O(n)$ number of elements of $A$ during each execution. This coarser grain approach balances the workload among threads as each thread deals with the same $O(n)$ number of elements at a time.

For the purposes of experimentation, consider a square matrix $A$ of $n \times n$ elements, where $2 \leq n \leq 10,000$ is an integer argument received from the command line. The matrix is stored in row major form containing distinct integers in the natural order from 0 to $n^2 - 1$ (the matrix elements are sorted on each row in increasing order from left to right and every row contains elements larger than the largest element of the previous row). For example, if `n=10,000`, $A$ contains $10,000 \times 10,000$ elements, each element being a distinct number from 0 to $10^8 - 1$. Each row contains $10^4$ distinct, successive, ordered elements and every row contains elements larger than the largest element of the previous row.

1. Create a C program for execution in a single processor computer to transpose the matrix $A$. The program will receive the size $n$ of the matrix $A$ as a command line parameter. Characterize this program"s execution time for a matrix of size $n = 8, 100, 1000,$ and $10,000$.

2. Generate a fine grain multi threaded version of a C program to transpose the matrix $A$ such that each fine grain thread deals with the exchange of a single element of $A$. The matrix $A$ is a shared data structure accessible by the threads.

   To create this multi-threaded version of the transposition program, it is suggested to use a synchronized shared counter accessed by the threads. A thread requests exclusive access to the counter, reads the value of the counter and increments it by one before relinquishing the counter. The thread then computes the indexes $i$ and $j$ corresponding to the element the thread will transpose:

   $$i = \lfloor \frac{counter}{n} \rfloor \qquad j = remainder(\frac{counter}{n})$$

   where $\lfloor \rfloor$ is the floor of the expression, the rounded down integer part of $\frac{counter}{n}$. The thread then should check that the indexes $i$ and $j$ correspond to an element of the upper triangular section of $A$ and transpose the element if indeed $j > i$. If the element is not on the upper triangular of $A$, the thread does nothing and goes back to request its next element from the counter.

   Your program should accept on the command line the parameters $n$ for the size of the matrix, and $t$ for the number of threads.

   Characterize this program's execution time for 2, 4, 8, 16, 32, and 64 threads and $n = 8, 100, 1000,$ and $10,000$.

3. Generate a coarse grain multi threaded version to transpose the matrix $A$ such that each coarse grain thread deals with the exchange of a number of elements of $A$ at a time, say *delta*. The matrix $A$ is a shared data structure accessible by the threads.

   As in the previous multithreaded version, it is suggested to use a shared synchronized counter that is now incremented by a value of *delta* instead of a value of 1. After reading the value of the counter, incrementing it by a value of *delta* and relinquishing it, the thread should scan all elements $a_{ij}$ with the indexes $i$ and $j$ generated by successive values of the read counter and for a number of elements *delta*.

   In this case, your program should accept on the command line the parameters $n$ for the size of the matrix, $t$ for the number of threads, and *delta* to define the granularity of each thread.

   Characterize this program's execution time for 2, 4, 8, 16, 32, and 64 threads, $n = 10,000$, and $delta = 10$ to 150 in increments of 10. How does this version compare to the previous light threaded one when $delta = 1$?

4. Graph the speed up with respect to the single processor version for each of the number of threads in the fine grain version and the coarse grain version. Generate superimposed curves, one for each matrix size.

For the coarse grain version of the multi-threaded version of the program, provide the graphs as above for the single value of *delta* showing the best improvement in performance.

5. For verification purposes, have your program print on **standard output** each integer element of the resulting transposed matrix in row major order, **one** element per line, only for $n = 8$ and with **no** additional information or text.

Submit a zip file named `x-y.zip`, where `x` and `y` are your UCInetIDs (the username used to login in Canvas), containing 4 files:

- The well commented C source code called `uniproctranspose.c`, `finetranspose.c` and `coarsetranspose.c`.
- A digitally produced report with the graphs and a discussion of the results (recommended length of 6 pages) called `report.pdf`.