Distributed and Parallel Systems — Due on Friday, 18, 2022

Assignment 1

CS4402B / CS9635B — *University of Western Ontario*

Submission instructions.

Format: The answers to the problem questions should be typed:

- source programs must be accompanied with input test files and,
- in the case of `Cilk` or `C+11` code, a `Makefile` (for compiling and running) is required,
- in the case of `Julia`, code with comments must be gathered in a `Jupyter` notebook, and
- for algorithms or complexity analyzes, LaTeX is highly recommended.

A PDF file (no other format allowed) should gather all the answers to non-programming questions. All the files (the PDF, the source programs, the input test files and Makefiles) should be archived using the UNIX command `tar`.

Submission: The assignment should submitted through the OWL website of the class.

Collaboration. You are expected to do this assignment *on your own* without assistance from anyone else in the class. However, you can use literature and if you do so, briefly list your references in the assignment. Be careful! You might find on the web solutions to our problems which are not appropriate. For instance, because the parallelism model is different. So please, avoid those traps and work out the solutions by yourself. You should not hesitate to contact the instructor or the TA if you have any questions regarding this assignment. We will be more than happy to help.

Marking. This assignment will be marked out of 100. A 10 % bonus will be given if your paper is clearly organized, the answers are precise and concise, the typography and the language are in good order. Messy assignments (unclear statements, lack of correctness in the reasoning, many typographical and language mistakes) may yield a 10 % malus.

PROBBLEM 1. [Parallelizing Strassen multiplication: 25 points] Multiplying dense matrices can be achieved by various algorithms, see the Wikipedia page Matrix multiplication algorithm. We saw in class how to efficiently implement the cubic (or plain) matrix multiplication algorithm using a divide-and-conquer process. There exist various matrix multiplication algorithms with a sub-cubic algebraic complexity, the most famous one is due to Volker Strassen and is described in the Wikipedia page Strassen algorithm

You will find at this page `Julia` code for a serial implementation of Strassen algorithm:

http://rosettacode.org/wiki/Strassen%27s_algorithm#Julia

Note that two algorithms are implemented there. One using a blocking strategy (and dynamic padding) and one using a direct recursive approach. The latter is clearly not as optimized as the former. Indeed, the latter allocates more auxiliary memory. Also, its threshold between the plain multiplication or Strassen algorithm is clearly too low.

Question 1. [5 points] Estimate the amount of auxiliary memory used by each of those `Julia` codes of Strassen algorithm. **HINT:** For the recursive one, one can proceed as follows. Let $M(n)$ be the amount of auxiliary memory in words for a call to the function `Strassen` on input matrices of order `n`. Determine the recurrence relation satisfied by $M(n)$ and deduce the value of $M(n)$ from that equation.

Question 1. [5 points] Compare experimentally the two implementation of Strassen algorithm: the one using dynamic padding and the recursive one. For this, collect running times for randomly generated matrices A and B (as we did in class for the plain multiplication), for n taking successive powers of 2, namely $4, 8, 16, 32, 64, 128, 256, 512, 1024$. The goal is to determine which one performs better. You are welcome to optimize this recursive approach, for instance, by using a larger threshold between the base case and the case where recursive calls are made. One can also try to reduce the amount of auxiliary memory.

Question 3. [10 points] Using multi-threading or multi-processing, make a parallel Julia version of the Strassen multiplication code that you have determined to be best performing at Question 1. For the tests, use matrices with randomly generated coefficients, in the way we did in class. You must provide two types of tests with your code:

- **correctness tests:** a couple examples with $n = 4$ for which your code verifies that $A * B$ is indeed what your parallel implementation computes.
- **performance tests:** tests for which n takes successive powers of 2, namely $4, 8, 16, 32, 64, 128, 256, 512, 1024$ and the matrices A and B are randomly generated as we did in class for the plain multiplication.

Question 4. [5 points] Compare experimentally the performance of your parallel implementation of Strassen multiplication against the parallel version of the plain multiplication studied with multi-processing (or the one we did as home work using shared arrays ,and possibly multi-threading). Use the same performance tests as in Question 3. Here, we should report not only the running times of each parallel implementation but also its speedup ratio w.r.t. each serial counterpart. These running times and speedup ratios should be measured using 4 workers, thus on a multi-core processor with at least 4 physical cores.

PROBBLEM 2. [Euclidean Traveling Salesperson Problem: 25 points] We consider the famous *Traveling Salesperson Problem (TSP)*, see the Wikipedia page Traveling Salesperson Problem. The TSP asks the following question: Given a list of cities and the distances between each pair of cities connected by a road, what is the shortest possible route that visits each city exactly once and returns to the origin city?" One variant of the TSP asks the simpler question: Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city exactly once and returns to the origin city?" In that second problem, it is assumed that every city is connected to every other by a road. In other words, the graph of those connecting roads is complete. One approximate algorithm solving that simpler TSP is described in Section 7.4 of those lecture notes:

http://www.cs.cmu.edu/afs/cs/academic/class/15210-s15/www/lectures/dandc-notes.pdf

We will call this algorithm `eTSP`, for *Euclidean Traveling Salesperson Problem.*

The article Polynomial time approximation schemes for Euclidean traveling salesman and other geometric problems by Sanjeev Arora gives the details of that approximate algorithm. The above mentioned lecture notes give enough details about `eTSP` except may be for the subroutine `splitLongestDim`, so we do it here.

For a group $\mathcal{P}$ of points in the plane (where each point M is known by its coordinates (x_M, y_m), we call *enclosing box*, the rectangle with smallest area (with edges parallel to the (Cartesian) coordinate axes) containing all points of $\mathcal{P}$. This rectangle is fully determined by two points $A(x_A, y_A)$ and $B(x_B, y_B)$, where A (resp. B) is the North-West (South-East) corner of $\mathcal{P}$. Hence, we assume $x_A \leq x_B$ and $y_A \geq y_B$. In fact, x_A (resp. y_B) is the minimum value of the x_M (resp. y_M) for $M \in \mathcal{P}$. Similarly, y_A (resp. x_B) is the maximum value of the y_M (resp. x_M) for $M \in \mathcal{P}$. The *spread along* x of $\mathcal{P}$ is $x_B - x_A$ and the *spread along* y of $\mathcal{P}$ is $y_A - x_B$. If $x_B - x_A \geq y_A - x_B$, `splitLongestDim` splits $\mathcal{P}$ vertically otherwise `splitLongestDim` splits $\mathcal{P}$ horizontally. To be precise, let us assume $x_B - x_A \geq y_A - x_B$. Then, we find a value m so that roughly half of the points in $\mathcal{P}$ have their x-coordinate satisfying $x < m$ and the rest of the points in $\mathcal{P}$ have their x-coordinate satisfying $m \leq x$.

Question 1. [5 points] Give an upper bound estimate (as sharp as possible) for the number of cache misses incurred by `eTSP` for an input collection of points of size n (each coefficient of each point is a floating point number occupying a single word) and an ideal cache of size Z with L words per cache line.

Question 2. [5 points] Estimate the work $W(n)$ and the span $S(n)$ of `eTSP` in the fork-join model.

Question 3. [5 points] Implement `eTSP` in `Julia`. We note that `Julia` implementation of algorithms for `TSP` are available at this place:

https://ericphanson.github.io/TravelingSalesmanExact.jl/dev/

Question 4. [5 points] Using multi-threading and shared arrays, make a parallel Julia version of `eTSP` This implies determining threshold experimentally between serial and parallel execution. You must provide two types of tests with your code:

- **correctness tests:** a couple examples with $n = 8$ for which your code is "verified" against `Julia`'s pre-existing implementation of `TSP`. Of course, our approximate algorithm for `eTSP` may not compute the exact solution given by `TSP` (applied to the complete graph of all cities). But it should not be too far from it either. By that we mean that the length of the route computed by the approximate algorithm for `eTSP` should be in the same order of magnitude that the route computed by the exact algorithm for `TSP`. In fact, theoretically, the former should be within a factor of 2 of the latter.
- **performance tests:** tests for which n takes successive multiples of 2, namely $20, 22, 24, \ldots, 64$. For each value of n, the test used for the serial and parallel approximate `eTSP` codes should be the same. Therefore, for each value of n, the

test should be generated once for all and used whenever a problem of size n is needed. Generating a test of size n means generating n points with x and y integer coordinates ranging -100 and 100.

Question 5. [5 points] Compare experimentally the performance of your parallel implementation of `eTSP` against its serial counterpart. Use the same performance tests as in Question 4. Here, we should report not only the running times of each parallel implementation but also the speedup ratio between the two. These running times and speedup ratio should be measured using 4 workers, thus on a multi-core processor with at least 4 physical cores.

PROBBLEM 3. [3D transpose: 25 points] Consider a three-dimensional array `A` implemented in the C programming language. Hence `A` can be seen as a pile of two-dimensional arrays `A[1 ; ; ]`, `A[2 ; ; ]`, ..., using `Julia` notations. For convenience, we refer to these 3 dimensions as z (the altitude index), x (the row index) and y (the column index). We denote that the dimension sizes of `A` by m, n, p respectively for x, y, z. Because in C all arrays are one-dimensional arrays, we need to decide on a storage layout for `A` and we choose by altitude, then row, then column. We denote this storage layout as $z > x > y$. Similarly, one can define 5 other storage layouts $z > y > x$, $x > z > y$, $y > z > x$, $x > y > z$ and $y > x > z$. The `3D Transpose Problem (3DTP)` asks for an algorithm converting `A` to another one-dimensional array `B` when changing the storage layout from $z > x > y$ to another storage layout.

The following article discusses `3DTP`:

https://booksc.eu/book/49184328/409d17

You will find `Julia` functions for multi-dimensional array permutations at:

https://docs.julialang.org/en/v1/base/arrays/

Question 1. [10 points] Design an algorithm for `3DTP` for the input C array `A` when the target storage layout is

- $z > y > x$
- $y > x > z$

The intention is, of course, to minimize cache-misses.

Question 2. [5 points] Give an upper bound estimate (as sharp as possible) for the number of cache misses incurred by `3DTP` for the input C array `A` and an ideal cache of size Z with L words per cache line, when the target storage layout is

- $z > y > x$
- $y > x > z$

Are your algorithms optimal in the point of view of cache complexity?

Question 3. [5 points] Realize a C (or C++) implementation of your algorithms as well as an implementation of their naive counterparts.

Question 4. [5 points] Compare experimentally the performance of your algorithms against their naive counterparts. You are free to design your experimentation.

PROBBLEM 4. [Pipelined factorization:25 points] Given a square matrix A of order n. QR decomposition computes an orthogonal matrix Q and an upper triangular matrix R such that $A = QR$. This factorization has numerous applications. One way of computing a QR decomposition of A is through the so-called *Householder reflections*, see the Wikipedia page

https://en.wikipedia.org/wiki/QR_decomposition#Using_Householder_reflections

A `Julia` implementation can be found here:

https://gist.github.com/eamartin/8118181

One can observe that the body of the for-loop between Line 10 and Line 26 has three parts:

Stage 1: between Lines 11 and 12, `M` is updated with the current of value of `R`,
Stage 2: between Lines 17 and 19, `R` is updated with the current of value of `M`,
Stage 3: between Lines 22 and 24, `Q` is updated with the current of value of `M`.

As a result, the updates of `R` and `Q`, that is, Stage 2 and Stage 3 can be done concurrently. In Questions 1 and 2, we take advantage of this observation using respectively a pipeline approach and multi-processing (or multi-threading).

Question 1. [15 points] Realize a `Julia` implementation of the QR decomposition (based on Householder reflections and the above mentioned `Julia` code) by means of the producer-consumer scheme. This implies refactoring this code: defining auxiliary functions and using `Julia`'s channels. Compare experimentally the performance of serial and pipelined code for randomly generated square matrices A with floating point number coefficients and order n taking successive powers of 2, namely $4, 8, 16, 32, 64, 128, 256, 512, 1024$.

Question 2. [10 points] Realize a `Julia` implementation of the QR decomposition (based on Householder reflections and the above mentioned `Julia` code) by means of multi-processing (or multi-threading). This implies refactoring this code: defining auxiliary functions and using `Julia`'s constructs for multi-processing (or multi-threading). Compare experimentally the performance of serial and multi-processed (or multi-threaded). for randomly generated square matrices A with floating point number coefficients and order n taking successive powers of 2, namely $4, 8, 16, 32, 64, 128, 256, 512, 1024$.