Distributed and Parallel Systems — Due on March, 21, 2022

Assignment 2

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. [The doubly-logarithmic Paradigm 20 points] We saw in class how, on a COMMON CRCW PRAM machine, one can find the maximum value of n integer numbers in time $T(n,p) = \Theta(1)$ for $p = n^2$ processors. Let us call this algorithm SUPER-FAST-MAX. The efficiency of that algorithm is $E(n,p) = 1/n$, Thus, SUPER-FAST-MAX is only interesting in practice when n is much smaller than p, namely $p \in \Omega(n^2)$. In order to make this algorithm more useful in practice, one can use it as a *base-case* of another algorithm solving the same problem in time $T(n,p) = \Theta(\log_2 \log_2 n)$ for $p = n$ processors. We describe this other algorithm, that we shall call FAST-MAX.

For simplicity assume that $n = 2^{2^h}$ for some integer $h > 0$. This means that $h = \log_2 \log_2 n$ holds. Assume that the input data $M[1 \cdots n]$ is partitioned into $\sqrt{n}$ segments $A_1, A_2, \ldots, A_{\sqrt{n}}$, corresponding to the sub-arrays $M[1 \cdots \sqrt{n}]$, $M[\sqrt{n}+1 \cdots 2\sqrt{n}]$, $\ldots$, $M[n - sqrtn - 1, \cdots, n]$, respectively. The algorithm FAST-MAX proceeds as follows.

(1) recursively find the maximum value for each sub-array A_i, for $1 \leq i \leq \sqrt{n}$, denoting those values $m_1, \ldots, m_{\sqrt{n}}$,

(2) call SUPER-FAST-MAX to find the maximum value of $m_1, \ldots, m_{\sqrt{n}}$.

Question 1. [5 points] Prove that the algorithm FAST-MAX can be executed on a COMMON CRCW PRAM machine using $p = n$ processors.

From now on, we assume $p = n$. For simplicity, we denote by $T(n)$ (resp. $W(n)$) the time $T(n, p)$ (resp. the work $W(n, p)$) of FAST-MAX on a COMMON CRCW PRAM machine.

Question 1=2. [5 points] Prove that we have:

$$T(n) \leq T(\sqrt{n}) + c_1 \quad \text{and} \quad W(n) \leq \sqrt{n}\, W(\sqrt{n}) + c_2 n.$$

for some positive constants c_1 and c_2.

Question 3. [5 points] Deduce that the following estimates

$$T(n) \in O(\log_2 \log_2 n) \quad \text{and} \quad W(n) \in O(n \log_2 \log_2 n)$$

What is the efficiency $E(n, n)$ of FAST-MAX?

Question 4. [5 points] Adapt FAST-MAX to work with $p \in \Theta(\sqrt{n})$ processors still using a COMMON CRCW PRAM machine. What are the running time, work and efficiency of the new algorithm.

PROBBLEM 2. [Prefix sum: 15 points]

The prefix sum is a fundamental operation in computer science with many basic applications. Given a vector $\vec{x} = (x_1, x_2, \ldots, x_n)$, the *prefix sum* of $\vec{x}$ is the vector $\vec{y} = (y_1, y_2, \ldots, y_n)$ such that $y_i = \sum_{j=1}^{j=i} x_j$ for $1 \leq j \leq n$. For instance, the prefix sum of $\vec{x} = (1, 2, 3, 4, 5, 6, 7, 8)$ is $\vec{y} = (1, 3, 6, 10, 15, 21, 28, 36)$. Hence, a `Julia` implementation of the above specification would be:

```
function prefixSum(x)
   n = length(x)
   y = fill(x[1],n)
   for i=2:n
      y[i] = y[i-1] + x[i]
   end
   y
end
```

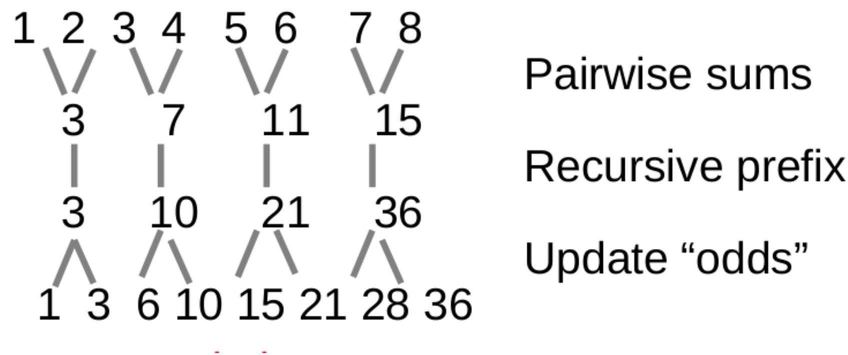
The i-th iteration of the loop is not at all decoupled from the $(i-1)$-th iteration. Hence, it looks like it is impossible to parallelize prefix sum computations, right?

In fact, there is a parallel counterpart (and even several ones) of the above algorithm. In this problem, we propose to analyze a divide and conquer algorithm strategy for computing prefix sums in a parallel fashion.

Consider again the input vector $x[1], x[2], \ldots, x[n]$ where n is a power of 2. This divide and conquer strategy works as follows:

(1) if $n = 2$ then return $(x[1], x[1] + x[2])$;
(2) for all even k's between 2 and n, replace the pair $(x[k-1], x[k])$ with $(x[k], x[k-1]+x[k])$;
(3) make a recursive call on $x[2], x[4], \ldots, x[n]$;
(4) for all even k's between 2 and n, compute $x[k-1] = x[k] - x[k-1]$;
(5) return x.

The above picture illustrates this algorithm with $n = 8$.



Question 1. [3 points] Write down the recurrence relation satisfied by the work $T_1(n)$ of the algorithm described above. Then, using that recurrence relation and the Master Theorem, give an asymptotic estimate of $T_1(n)$.

Question 2. [5 points] Explain in plain words how this algorithm can be parallelized in the fork-join model, that is, using `Cilk`. Writing pseudo-code is recommended, but writing real code is not required.

Question 3. [3 points] Write down the recurrence relation satisfied by the span $T_\infty(n)$ of your algorithm. Then, using that recurrence relation and the Master Theorem, give an asymptotic estimate of $T_\infty(n)$.

Question 4. [4 points] Discuss what would be challenging in the implementation on multi-core processors.

PROBBLEM 3. [Using prefix sums for parallel quick-select: 15 points] We discuss in this problem an application of the prefix sum operation discussed in Problem refpb:2: the parallel *quick-select*. Given an array A of integer numbers and an integer k, where $0leqk < |A|$ holds, the quick-select operation asks for the k-th smallest element in A. Naively, we may first sort

the sequence A, then select the k-th element; this requires $O(n\log_2 n)$ work. In fact, it is possible to perform quick-select with work $\Theta(n)$ and span $O\log_2^2 n)$ in the fork-join model.

The key is a sub-algorithm, that we call SPLIT, with the following specifications

Input: an array A of n integer numbers and a *pivot element* $\min(A) \leq p \leq \max(A)$,

Output: two arrays L and R containing the elements in A that are less than (or greater than or equal to) p.

Pseudo-code for SPLIT follows:

(1) Compute the *left indicator array* $I_L[0, \ldots, n1]$ of size n so that $I[i]$ is equal to 1, if $A[i] < b$ holds, and 0 otherwise

(2) Deduce the *right indicator array* $I_R[0, \ldots, n1]$ of size n so that $I[i]$ is equal to 1, if $A[i] \geq b$ holds, and 0 otherwise

(3) Apply prefix sum to I_L, storing the result in an array P_L.

(4) Create the array L of size $n - P_L[n-1]$

(5) Apply prefix sum to I_R, storing the result in an array P_R.

(6) Create the array R of size $n - P_R[n-1]$

(7) for $i \in \{1, 2, \ldots, n\}$ do

  (7.1) if $A[i] < b$ then $L[P_L[i]] := A[i]$ else $R[P_R[i]] := A[i]$

(8) return (L, R)

Question 1. [5 points] Prove that SPLIT has work $\Theta(n)$ and span $O\log_2 n)$ in the fork-join model.

Question 2. [5 points] Propose an algorithm performing Quick-Select that uses SPLIT as a sub-algorithm. Your algorithm can be stated in the same style of pseudo-code as SPLIT

Question 3. [5 points] Prove that the algorithm proposed in Question 2 has work $\Theta(n)$ and span $O\log_2^2 n)$ in the fork-join model.

PROBBLEM 4. [Effective parallel matrix inversion 50 points]

Let A be a square matrix of order $n \geq 2$, where n is assumed to be a power of 2. The goal of this problem is to write an efficient program for computing the inverse of A. To do so, we will rely on a classical technique, the called Blockwise inversion.

We will start by analyzing the work and the critical path of this algorithm in the fork-join model. We assume that we have two parallel sub-algorithms at our disposal:

- $\mathrm{ADD(C, T, n)}$ computing the sum of two square matrices C and T (C is replaced by $C + T$) of order n in work $A_1(n) = \Theta(n^2)$ and with critical path $A_\infty(n) = \Theta(\log(n))$;
- $\mathrm{MULT(C, A, B, n)}$ computing the product of two square matrices A and B (C is replaced by AB) of order n in work $M_1(n) = \Theta(n^3)$ and with critical path $M_\infty(n) = \Theta(\log^2(n))$;

As noted in class $\mathrm{MULT(C, A, B, n)}$ would not be what we would use in an actual computer program, say on multi-core processors.

We review below the principle of lockwise inversion. We partition A into four square blocks of order $n/2$:

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \quad (1)$$

Let O and I be the zero and identity matrices of order $n/2$, respectively. We assume that the upper right block A_{11} is an invertible matrix. Then we define $S = A_{22} - A_{21}A_{11}^{-1}A_{12}$ (called the *Schur complement*). We assume that S is also invertible. Then we have:

$$A^{-1} = \begin{bmatrix} I & -A_{11}^{-1}A_{12} \\ O & I \end{bmatrix} \begin{bmatrix} A_{11}^{-1} & O \\ O & S^{-1} \end{bmatrix} \begin{bmatrix} I & O \\ -A_{21}A_{11}^{-1} & I \end{bmatrix} \tag{2}$$

Let us denote by U, D, L the above three matrices, respectively.

Question 1. [5 points] Write a `Cilk`-like program computing the inverse of A based on the above formula. We will **assume** that at each recursive call the upper right block (namely $A_1 11$) and the Schur complement are invertible. You shall try to make the critical path as small as possible.

Question 2. [5 points] Estimate the work $I_1(n)$ and the critical path $I_\infty(n)$ of your algorithm.

We shall write a `Cilk` program for computing the inverse of A, based on blockwise inversion. Towards that goal and based on our experience with matrix multiplication, we consider modifying the above algorithm so as to avoid intermediate allocation of extra memory storage, say at each recursive call.

Question 3. [5 points] Propose a multithreaded algorithm, in the form of a `Cilk`-like program, computing the inverse of A based on blockwise inversion, so that no intermediate allocation of extra memory storage is needed. To this end, this algorithm will take the following matrices as input arguments:

- the input matrix A of order n,
- the output matrix I (that is, A^{-1}) of order n,
- one *work-space* matrix W of order n,
- one (or more) *work-space* matrix C of order $n/2$.

Since the work-space matrices are allocated once for all, they are not considered as intermediate allocation of extra memory storage. Recursive calls should make use of blocks of those work-space matrices. Before the matrix I is fully computed, blocks of I can also be used as work-space to store intermediate expressions. The matrix A can also be overwritten and used as work-space.

Question 4. [5 points] Analyze the work and span of the algorithm presented in Question 3.

Question 5. [30 points] Realize a `CIlk` implementation of the algorithm proposed at Question 3. For the tests, use dense matrices:

1. with randomly generated coefficients of type *float*, as well as
2. the *Hilbert matrix* $H_n = (\frac{1}{i+j-1}, 1 \leq i, j \leq n)$.

Note that, with probability 1, a dense randomly generated matrix is invertible. As for the Hilbert matrix H_n, it can be shown to be invertible for every positive n.

You must provide two types of tests with your code:

- **correctness tests:** a couple examples with $n = 4$ for which your code verifies that $A \cdot B = I_n$ (or close to that due to truncation errors), where B is the computed inverse and I_n is the identity matrix of order n.
- **performance tests:** tests for which n takes successive powers of 2, namely $4, 8, 16, 32, 64, 128, 256, 512, 1024$ considering both for A randomly generated matrices and Hilbert matrices, collecting both parallel running times and serial running times. Moreover, for $n = 1024$, you should use `CilkScale` to produce the same type of performance plots as in the collection of `OpenCilk` examples used in class.