CME 323: Distributed Algorithms and Optimization

Instructor: Reza Zadeh (rezab@stanford.edu)
TA: Alex Yang (yangzj@stanford.edu)
HW#1 – Due Thursday April 18 (on Gradescope)

1. The Karatsuba algorithm multiplies two integers x and y. Assuming each has n bits where n is a power of 2, it does this by splitting the bits of each integer into two halves, each of size $n/2$. For any integer x we will refer to the low order bits as x_l and the high order as x_h. The algorithm computes the result as follows:[1]

```
function KM($x, y, n$)
    if $n = 1$ then
        return $x \times y$
    else
        $a \leftarrow \text{km}(x_l, y_l, n/2)$
        $b \leftarrow \text{km}(x_h, y_h, n/2)$
        $c \leftarrow \text{km}(x_l + x_h, y_l + y_h, n/2)$
        $d \leftarrow c - a - b$
        return $(b \cdot 2^n + d \cdot 2^{n/2} + a)$
    end if
end function
```

Note that multiplying by 2^k can be done just by shifting the bits over k positions.

(a.) Assuming addition, subtraction, and shifting take $O(n)$ work and $O(n)$ depth what is the work and depth of **km**?

(b.) Assuming addition, subtraction, and shifting take $O(n)$ work and $O(\log n)$ depth what is the work and depth of **km**?

2. Suppose a square matrix is divided into blocks:

$$M = \begin{bmatrix} A & B \\ C & D \end{bmatrix}$$

where all the blocks are the same size. The *Schur complement* of block D of M is $S = A - BD^{-1}C$. The inverse of the matrix M can then be expressed as:

$$M^{-1} = \begin{bmatrix} S^{-1} & S^{-1}BD^{-1} \\ -D^{-1}CS^{-1} & D^{-1} + D^{-1}CS^{-1}BD^{-1} \end{bmatrix}$$

This basically defines a recursive algorithm for inverting a matrix which makes two recursive calls (to calculate D^{-1} and S^{-1}), several calls to matrix multiply, and one each to elementwise add and subtrace two matrices. Assuming that matrix multiply has work $O(n^3)$ and depth $O(\log n)$ what is the work and depth of this inversion algorithm?

[1] If you have seen this before, you might have thought of it as a sequential algorithm, but actually it is a parallel algorithm, since in particular, the three recursive calls to **km** can be made in parallel.

3. Describe a divide-and-conquer algorithm for merging two sorted arrays of lengths n into a sorted array of length $2n$. It needs to run in $O(n)$ work and $O(\log^2 n)$ depth. You can write the pseudocode for your algorithm so that it looks like your favorite sequential language (C, Java, Matlab, ...), but with an indication of which loops or function calls happen in parallel. For example, use `parallel for` for a parallel for loop, and something like:

```
parallel {
    foo(x,y)
    bar(x,y)
}
```

to indicate that `foo` and `bar` are called in parallel. You should prove correctness at the level expected in an algorithms class (e.g. CME305 or CS161).

4. Given a sequence of n real numbers $\boldsymbol{s} = (s_1, ..., s_n)$, the maximum contiguous subsequence sum problem is to find a contiguous subsequence of $\boldsymbol{s}$ such that its sum is maximal, i.e.

$$F(\boldsymbol{s}) = \max_{1 \le i \le j \le n} \sum_{k=i}^{j} s_k$$

   (a) Consider the following algorithm running in parallel manner, what is the work and depth of it? (Hint: be aware of the depth and work of max and sum)

```
function MaxContiguousSubsequenceSum($\boldsymbol{s}[1 \dots n]$)
    for $1 \le i \le j \le n$ do
        Compute $SubSum(i,j) = \sum_{k=i}^{j} s_k$
    end for
    return max($\{SubSum(i,j)\}_{1 \le i \le j \le n}$)
end function
```

   (b) Give an algorithm to solve this problem that runs in O(n log n) work and O($\log^2$n) depth. Give a short proof about its work and depth. (Hint: Divide and Conquer)

5. In this problem, we'll look at how fast the maximum of a set of n elements can be computed when allowing for concurrent writes. In particular we allow the arbitrary write rule for "combining" (i.e. if there are a set of parallel writes to a location, one of them wins). Show that this can be done in $O(\log \log n)$ depth and $O(n)$ work.

   (a) Describe an algorithm for maximum that takes $O(n^2)$ work and $O(1)$ depth (using concurrent writes).

   (b) Use this to develop an algorithm with $O(n)$ work and $O(\log \log n)$ depth. Hint: use divide and conquer, but with a branching factor greater than 2.

6. **Interval Scheduling Problem** Given a set of n tasks with start time and finish time for each task, $T = \{(s_1, f_1), (s_2, f_2), \dots, (s_n, f_n)\}$, we want to find the largest subset $S \subseteq \{1, 2, ..., n\}$ such that for any pair i, j in S, the intervals (s_i, f_i) and (s_j, f_j) do

not overlap. Here we assume $s_i < f_i$ holds for any task, which means every task takes positive time to finish. Design a greedy algorithm to solve this problem, and prove that the resulting schedule is optimal.

7. **Solving Linear Systems**

Lower Triangular Systems Consider the task of solving the linear system $Ax = b$ where we assume A is lower triangular. A popular method for solving $Ax = b$ is *forward substitution*. The forward substitution algorithm can be represented as the following series of serial updates:

```
$x_1 \leftarrow b_1/a_{11}$
for $i = 2, \ldots, n$ do
    $x_i \leftarrow \left(b_i - \sum_{j=1}^{i-1} l_{ij}x_j\right)/a_{ii}$
end for
```

(a) What is the computation complexity of the forward substitution algorithm?

The parallel forward substitution algorithm operates by parallelizing the serial forward substitution algorithm. Note that the y_j updates can all be executed in parallel.

```
$x_1 \leftarrow b_1/a_{11}$
for $i = 2, \ldots, n$ do
    $x_i \leftarrow (b_i - y_i)/a_{ii}$
    for $j = i + 1, \ldots, n$ do
        $y_j \leftarrow a_{j1}x_1 + \ldots + a_{ji}x_i$
    end for
end for
```

(b) Construct the DAG representing the parallel forward substitution algorithm. What is the depth of the DAG?

Tridiagonal Systems We now consider solving the system $Ax = b$ where A is tridiagonal. Explicitly, $a_{ij} = 0$ if $|i - j| \geq 2$. Note that this is equivalent to solving the following system of linear equations:

$$\begin{aligned} g_1x_1 + h_1x_2 &= b_1 \\ f_ix_{i-1} + g_ix_i + h_ix_{i+1} &= b_i, \quad i = 2, \ldots, n-1 \\ f_nx_{n-1} + g_nx_n &= b_n \end{aligned}$$

where g_i are the diagonal elements of A, f_i the entries below the diagonal, and h_i the entries above the diagonal. The idea behind *even-odd reductions* is to recursively reduce the above system to one of half the size. Explicitly, if none of the diagonal entries are zero, we can solve for each x_i in terms of x_{i-1} and x_{i+1}. If we do this for all odd i, and substitute the expression back in, we obtain a system on just the even indexed variables.

(a) Using the above system of equations, derive a tridiagonal system of equations on just the even indexed variables.

(b) What is the computational complexity of computing the coefficients of the reduced system?

The above procedure can be recursively applied until the problem is reduced to a single equation. Then we work backwards to solve for the value of the eliminated variables.

(c) What is the computational complexity of solving for the eliminated variables?

(d) Construct the DAG representing this algorithm.

(e) What is the runtime of the even odd reduction algorithm on $O(n)$ processors?

Givens Rotations Givens Rotations are used to zero out the subdiagonal entries of the matrix A one at a time. Crucially, a Givens rotation only affects two rows of the matrix. We will use this fact to derive a parallel implementation of the Givens rotation algorithm. Specifically, if two successive Givens rotations affect disjoint sets of rows, then they can be computed in parallel.

(a) When n rows are available, what is the maximum number of Givens rotations we can apply simultaneously?

Implementing the Givens rotations in parallel ultimately comes down to deriving a schedule of the entries to eliminate at a particular step. We consider two functions $T(j,k)$ and $S(j,k)$ where $T(j,k)$ represents the iteration in which the jkth entry is eliminated, and j and $S(j,k)$ are the rows the Givens rotation operates on. To simulateneously implement the Givens rotations, we require that $T(j,k)$ and $S(j,k)$ satisfy:

- If $T(j,k) = T(j',k')$ and $(j,k) \neq (j',k')$ then $\{j, S(j,k)\} \cap \{j', S(j',k')\} = \emptyset$.
- If $T(j,k) = t$ and $S(j,k) = i$, then $T(j,l) < t$ and $T(i,l) < t$ for all $l < k$.

(b) Prove that the schedule given by

$$T(j,k) = n - j + 2k - 1$$
$$S(j,k) = j - 1$$

satisfies the above conditions.

(c) What is the maximum number of stages required by this schedule?