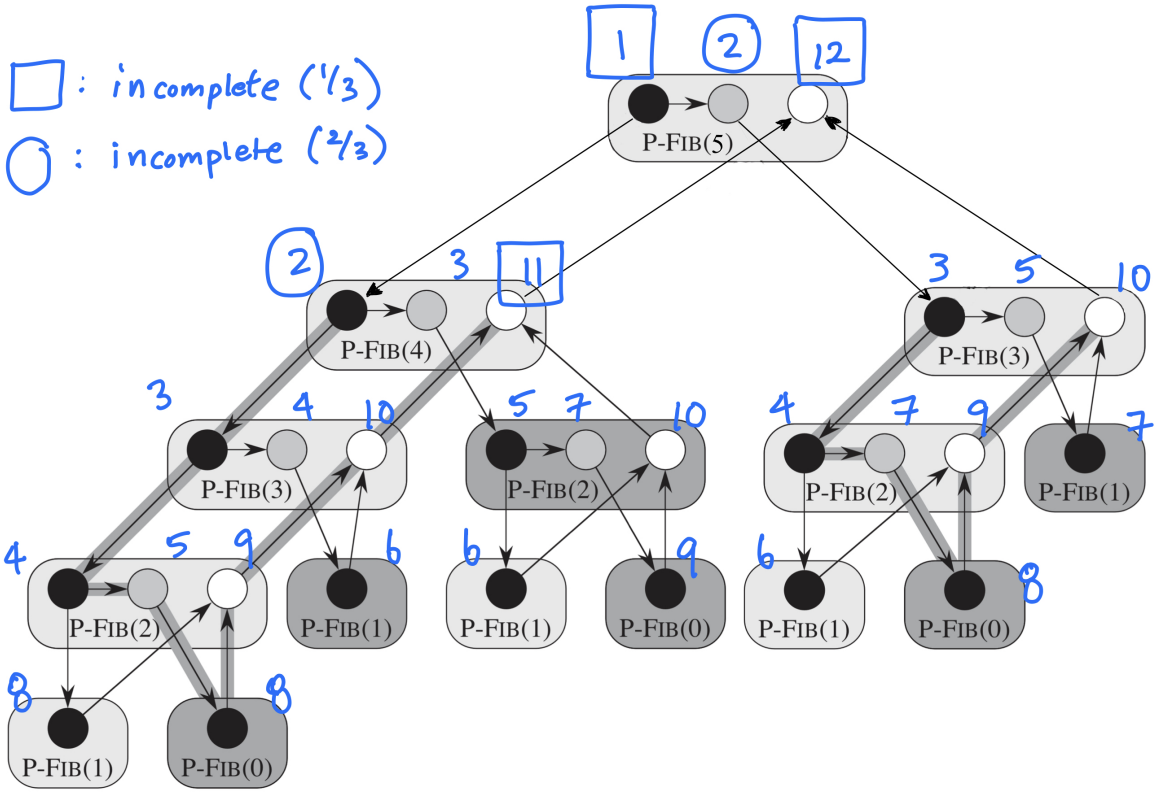
CS 255: Design and Analysis of Algorithms **SJSU Spring 2019**
Professor Christopher Pollett March 4, 2019

Homework 2: Parallel Algorithms

Problem 1

CLRS 27.1-2. Draw the computation dag that results from executing P-FIB(5). Assuming that each strand in the computation takes unit time, what are the work, span, and parallelism of the computation? Show how to schedule the dag on 3 processors using greedy scheduling by labeling each strand with the time step in which it is executed.

Solution: Scheduling the dag on 3 processors using a greedy DFS rule,



- Work (T_1) is the total number of strands: $T_1 = 29$
- Span (T_∞) is the longest (critical) path from the initial strand to the final strand: $T_\infty = 10$
- Parallelism is the ratio of work to span: $T_1/T_\infty = 29/10 = 2.9$
- There are 8 complete (3 strands), 1 incomplete (2 strands), and 3 incomplete (1 strand) steps $= 29 = T_1$.

Problem 2

CLRS 27.1-6. Give a multithreaded algorithm to multiply an $n \times n$ matrix by an n-vector that achieves $\Theta(n^2/\lg n)$ parallelism while maintaining $\Theta(n^2)$ work.

Solution: Consider the MAT-VEC procedure given in the textbook (Appendix), which has work $T_1 = \Theta(n^2)$ and span $T_1 = \Theta(n)$. To improve the parallelism of MAT-VEC to $\Theta(n^2/\lg n)$, we investigate the dominating linear term from the inner loop that is not parallelized. We are initially unable to parallelize the inner loop due to potential race conditions, but to avoid races totally, we parallelize by computation of whole rows, as opposed one term at a time. In other words, we expand the matrix into row vectors; given a matrix $A \in M_n(\mathbb{C})$ and vector $x \in \mathbb{C}_n$, the i^{th} element of the resultant vector $y = Ax$ is the inner product of the i^{th} row of A with x. We can implement this by slightly modifying MAT-VEC with a divide-and-conquer subroutine using nested parallelism similar to MAT-VEC-MAIN-LOOP (Appendix) from the textbook,

Algorithm 1: P-MAT-VEC(A, x)

input : Matrix A, vector x
output: vector y: outcome of matrix-vector multiplication of A and x

```
1 n = A.rows
2 let y be a new vector of length n
3 parallel for i = 1 to n do
4     y_i = 0
5 parallel for i = 1 to n do
6     y_i = P-MAT-VEC-LOOP(A, x, i, 1, n)
7 return y
```

Algorithm 2: P-MAT-VEC-LOOP(A, x, i, j, j')

input : Matrix A, vector x, index i, index j, index j'
output: The inner product of i^{th} row of matrix A with vector x from index j to j'

```
1 if j == j' then
2     return A_ij × x_j
3 else
4     mid = ⌊(j + j')/2⌋
5     leftHalf = spawn P-MAT-VEC-LOOP(A, x, i, j, mid)
6     rightHalf = P-MAT-VEC-LOOP(A, x, i, mid + 1, j')
7     sync
8     return lowerHalf + upperHalf
```

The work necessary for P-MAT-VEC is found by analyzing the recursive calls in P-MAT-VEC-LOOP as they are the dominating term. Given that P-MAT-VEC-LOOP recursively calls two subproblems of size $n/2$, calculating the base case takes constant time, and combining their results takes constant time:

$$T_1(n) = 2T_1(n/2) + \Theta(1)$$

By the Master theorem, the work associated with P-MAT-VEC is $T_1(n) = \Theta(n^2)$, as expected.

The span of P-MAT-VEC can be found by considering the recurrence tree of P-MAT-VEC-LOOP. It is a complete binary tree with depth $\lg n$, thus the span is $T_\infty = \Theta(\lg n)$. Therefore, the parallelism of P-MAT-VEC is $T_1/T_\infty = \Theta(n^2/\lg n)$ while maintaining $\Theta(n^2)$ work.

Problem 3

CLRS 27.2-5. Give pseudocode for an efficient multithreaded algorithm that transposes an $n \times n$ matrix in place by using divide-and-conquer to divide the matrix recursively into four $n/2 \times n/2$ submatrices. Analyze your algorithm.

Solution: Consider the transpose of a block matrix expansion of some arbitrary square matrix $A \in M_n$,

$$A^T = \left[\begin{array}{c|c} A_{11} & A_{12} \\ \hline A_{21} & A_{22} \end{array}\right]^T = \left[\begin{array}{c|c} A_{11}^T & A_{21} \\ \hline A_{12} & A_{22}^T \end{array}\right]$$

We may solve for A^T by partitioning A into four submatrices of size $n/2 \times n/2$. Then, we swap block A_{12} with A_{21}, recursively call two subproblems A_{11}^T and A_{22}^T on submatrices of size $n/2 \times n/2$, and combine the results. The transpose of a single element matrix is itself. High level pseudocode is presented below:

Algorithm 3: P-MAT-TRANSPOSE(A)

input : Matrix A
output: The transpose of A

```
1  n = A.rows
2  if n == 1 then
3      return A
4  else
5      partition A into four n/2 × n/2 submatrices: A_11, A_12, A_21, and A_22
6      // swap A_12 and A_21
7      A'_12 = A_21
8      A'_21 = A_12
9      // recursively call P-MAT-TRANSPOSE on subproblems.
10     A'_11 = spawn P-MAT-TRANSPOSE A_11
11     A'_22 = P-MAT-TRANSPOSE A_22
12     sync
13     A' = combine A'_11, A'_12, A'_21, and A'_22
14     return A'
```

Analysis: The work T_1 of P-MAT-TRANSPOSE in terms of the number of elements in the matrix $E = n^2$ is given by the recurrence $T_1(E) = 2T_1(E/4) + \Theta(E)$ as each recursive call to P-MAT-TRANSPOSE is called on a subproblem of size $n/2 \times n/2$ with $E/4$ elements and each swap procedure must traverse at least $E/4 = \Theta(E)$ elements. By the master theorem, the work $T_1 = \Theta(E) = \Theta(n^2)$.

The span of P-MAT-TRANSPOSE is given by the maximum span of the swap procedure and the recursive calls to P-MAT-TRANSPOSE. Each swap procedure has a span of $\Theta(\lg(n^2)) = \Theta(\lg n)$ using infinite processors on n^2 elements and the recursive calls obey the recurrence $T_\infty(E) = T_\infty(E/4) + \Theta(1)$ because combining takes constant time. By the Master theorem, the span of the recursive calls is also $\Theta(\lg n)$. Therefore, because the span of the swap procedure and the recursive calls is asymptotically equal, the overall span of P-MAT-TRANSPOSE is $T_\infty = \Theta(\lg n)$. The parallelism of P-MAT-TRANSPOSE is $T_1/T_\infty = \Theta(n^2/\lg n)$.

Problem 4

Let $A = (A_{ij})$ represent the adjacency matrix of an n node graph. I.e. a_{ij} is 1 if there is an edge from i to j and 0 otherwise. Let $A_{trans} = (a'_{ij})$ be the adjacency matrix of the transitive closure of A. That is, $a'_{ij} = 1$ if there is a path from i to j in A and 0 otherwise. Design a multithreaded algorithm which computes A_{trans} with work $O(n^4)$ and span $O(\log^3(n))$.

Solution: Main Idea: The transitive closure of a graph given its adjacency matrix A can be found by taking n^{th} power of A, i.e. $A_{trans} = A^n$. This is equivalent to finding the existence of paths of length at most n from i to j in A. To do exponentiation, we call P-Matrix-Multiply-Recursive (Appendix) recursively on pairs of matrices of size $n/2$. These subproblems are independent and can be solved in parallel. For example, at $n = 8, A^8 = A^4 \times A^4 = (A^2 \times A^2)(A^2 \times A^2) = (A^1 \times A^1)(A^1 \times A^1)(A^1 \times A^1)(A^1 \times A^1)$.

Algorithm 4: P-Matrix-Transitive-Closure(A)

input : Adjacency Matrix A
output: Transitive Closure A_{trans}

```
1 return P-Matrix-Exponentiation(A, 1, n)
```

Algorithm 5: P-Matrix-Exponentiation(A, i, i')

input : Matrix $A \in M_n$, index i, index i'
output: A to the power of $i' - i + 1$

```
1  Let C ∈ M_n
2  if i == i' then
3      return A
4  if i' - i == 1 then
5      return P-Matrix-Multiply-Recursive(C, A, A)
6  mid = ⌊(i + i')/2⌋
7  leftHalf = spawn P-Matrix-Exponentiation(A, i, mid)
8  rightHalf = P-Matrix-Exponentiation(A, mid + 1, i')
9  sync
10 return P-Matrix-Multiply-Recursive(C, leftHalf, rightHalf)
```

Proof of correctness: Matrix multiplcation works and we multiply A at the base level n times $\implies A^n$.

Running Time and Analysis: The work $M_1(n)$ of P-Matrix-Multiply-Recursive is found by serialization. Partitioning takes constant time, and 8 recursive calls are made on $n/2 \times n/2$ matrices. Combining the results of the subproblems is equal to the number of terms in the resultant matrix $n \times n = n^2$, or $\Theta(n^2)$ time. The recurrence for $M_1(n)$ is therefore $M_1(n) = 8M_1(n/2) + \Theta(n^2) = \Theta(n^3)$ by the Master theorem. The span M_∞ of P-Matrix-Multiply-Recursive is determined by the span of any recursive call (as all non base-case subproblems have the same span) and the cost of the parallel for loops after synchronization which is $\Theta(\lg n)$. Therefore, the recurrence of the span of P-Matrix-Multiply-Recursive is $M_\infty(n) = M\infty(n/2) + \Theta(\lg n) = \Theta(\lg^2(n))$.

The work $T_1(n)$ of P-Matrix-Transitive-Closure is equal to the work $E_1(n)$ of P-Matrix-Exponentiation. Serializing P-Matrix-Exponentiation would yield n subproblems, i.e. n calls to P-Matrix-Multiply-Recursive, and work of $E_1(n) = \Theta(nM_1(n))$. Therefore the total work is $T_1(n) = E_1(n) = \Theta(nM_1(n)) = \Theta(n^4)$, as desired. The span $T_\infty(n)$ of P-Matrix-Transitive-Closure is equal to the span $E_\infty(n)$ of P-Matrix-Exponentiation. The span of P-Matrix-Exponentiation on n subproblems is $E_\infty(n) = \Theta((\lg n)M_\infty(n))$. Therefore, the total span is $T_\infty = E_\infty = \Theta((\lg n)M_\infty(n)) = \Theta(\lg^3(n))$.

Problem 5

Suppose we have an array $A[1]$ to $A[n]$ each entry of which has a positive integer. Devise a CREW PRAM algorithm that sets each of these $A[i]$'s to the value of the maximum among $A[i]$'s in $O(\log n)$ steps using at most n processors.

Solution: We sort the array using BoxSort in $O(\log n)$ steps. Once the array is sorted, we access the value of the maximum and store it in some variable x in $O(1)$ time. Then, setting each $A[i]$ for $i \in 1 \dots n$ to x with at most n processors takes another $O(\log n)$ steps. Therefore, the overall procedure takes $O(\log n)$ steps.

Pseudocode:

Algorithm 6: SET-MAX(A)

input : Array A
output: Array A with all elements set to the maximum value in A

```
1 n = A.rows
2 BOX-SORT-SUB(A, n, 1, n)
3 x = A[n − 1]
4 parallel for i = 1 to n do
5 |  A[i] = x
6 return A
```

Algorithm 7: BOX-SORT-SUB(A, n, i, i')

```
1 if i' − i <= log n then
2 |  LOGSORT(A[i ... i']) ;                          // trivial for n processors on a PRAM
3 else
4 |  S ← Pick k = √n elements at random and sort with n processors ;          // splitters
5 |  Use the elements of S as "splitters" in A[i ... i'] to create k + 1 subproblems
6 |  for each subproblem A[i, S_1], A[S_1, S_2], ... A[S_k, i'] do
7 |  |  BOX-SORT-SUB(A, n, ...)
```

Coding Portion

For the coding part of this homework I would like you to write two parallel programs in Java: ThreadOuter.java and JoclOuter.java. These programs should compute the outer product of two column vectors $\vec{v}$ and $\vec{w}$, i.e. $\vec{v}\vec{w}^T$. The first program makes use of Java Threads and the second makes use of JOCL jar file that provides Java bindings to OpenCL. Both of these programs should compute the norm of a vector of integers that comes from a file. I would like you to code both of your programs so that their spans are $O(\log n)$. ThreadOuter.java will be compiled from the command line via:

```
java ThreadOuter.java
```

It can use either classic Java Threads or the Fork Join/Parallel Array frameworks in java.util.concurrent. We didn't talk about the latter so you'll be on our own to lean if you want to use those. To run your program I will then type:

```
java ThreadOuter filename_with_vector_data
```

On this input, you program should read in the contents of `filename_with_vector_data`, which should consist of lines with two comma separated integers followed by a new line character/line, make two vectors $\vec{x}$ and $\vec{y}$ from these, and computes their outer product. Finally, it should output the resulting matrix, entries in a row comma separated, rows delimited by a newline. For example, I might have the file `my_vectors.txt` with contents:

```
1, -1
2, 1
3, 4
10, 5
```

On this input, it should output:

```
-1, 1, 4, 5
-2, 2, 8, 10
-3, 3, 12, 15
-10, 10, 40, 50
```

Your JoclOuter.java program should do exactly the same thing, but use JOCL rather than Java Threads. I.e., I'll compile it with:

```
java JoclOuter.java
```

You can assume I have set up the classpath to find the JoCL jar file. Then I'll run your program with a line like:

```
java JoclOuter filename_with_vector_data
```

For each program you should add a mechanism of your choice to time just the portion of the code in which parallel processing is done (not reading in the file, you probably want to be using System.nanoTime()). I want you to do experiments with both programs varying the length of the vector you compute the outer of. Look up the number of cores the machine you are experimenting on has, and the number of GPU shader processors it has. If you plot time versus the length of vectors you compute the outer products of, does it match what you'd expect in each case if Brent's Theorem were an equality rather than an inequality? Write up your experiments also in Hw2.pdf. This timing should be turned off by default.

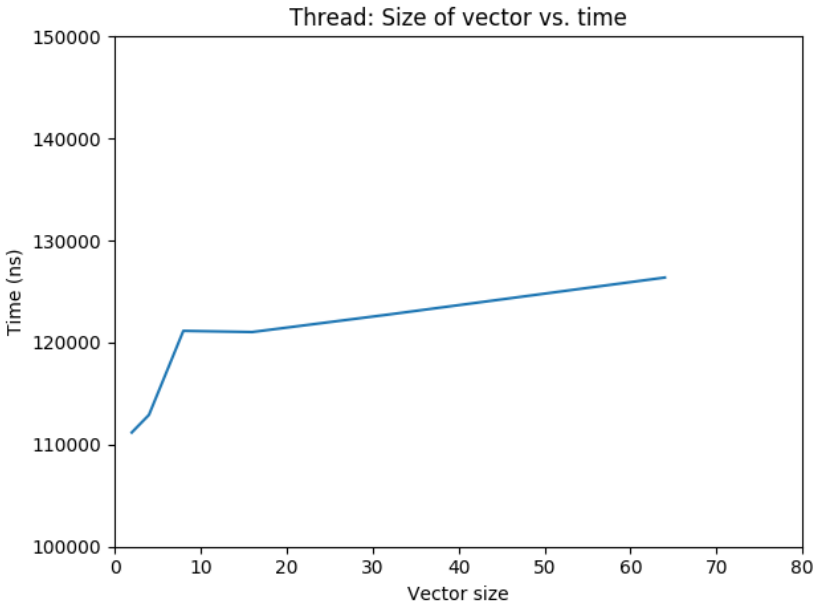
The above concludes the description of the required homework. I am also willing to give 1 bonus point if you recode your JOCL program in Vulkan and if you show me your Vulkan code working.

Solution:

Using Java Thread,

```
Josephs-Mac-Pro:hw2 joseph$ javac ThreadOuter.java
Josephs-Mac-Pro:hw2 joseph$ java ThreadOuter my_vectors.txt
-1,1,4,5
-2,2,8,10
-3,3,12,15
-10,10,40,50
Josephs-Mac-Pro:hw2 joseph$ cat my_vectors.txt
1,-1
2,1
3,4
10,5
```

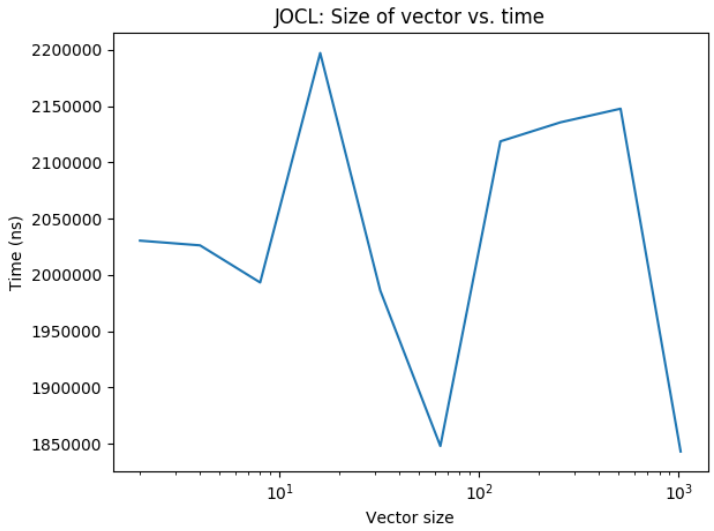
Vector size	Time (ns)
2	111190
4	112913
8	121152
16	121036
32	122767
64	126379



Using Java OpenCL,

```
Josephs-Mac-Pro:hw2 joseph$ javac -classpath "./jocl-2.0.1.jar" JoclOuter.java
Note: JoclOuter.java uses or overrides a deprecated API.
Note: Recompile with -Xlint:deprecation for details.
Josephs-Mac-Pro:hw2 joseph$ java -classpath ".:./jocl-2.0.1.jar" JoclOuter my_vectors.txt
-1,1,4,5
-2,2,8,10
-3,3,12,15
-10,10,40,50
Josephs-Mac-Pro:hw2 joseph$ cat my_vectors.txt
1,-1
2,1
3,4
10,5
```

Vector size	Time (ns)
2	2030586
4	2026400
8	1993398
16	2197324
32	1986226
64	1848095
128	2118836
256	2135733
512	2147843
1024	1843220



We calculate the outer product of two vectors using Java Threads and Java OpenCL (JOCL). This was done in span $T_\infty = \Theta(\log n)$. Coding and experiments were conducted on a 2015 Macbook Pro with a 2.8 GHz Quad-core Intel Core i7 processor. There is a Built-In Intel Iris Pro GPU as well as a PCIe AMD Radeon R9 M370X GPU. The AMD Radeon R9 is running 640 cores with 640 shading units.

The experiments were timed using Java's `System.nanoTime()` function. Vectors of varying length were randomly generated with elements ranging from 1 to 1000. Brent's Theorem is given by:

$$T_P \leq \frac{T_1}{P} + T_\infty$$

The amount of work scales with the input size, $T_1 = \Theta(n)$, and the span $T_\infty = \Theta(\log n)$. Therefore, Brent's theorem becomes $T_P \leq \Theta(n) + \Theta(\log n)$. Assuming equality, $T_P = \Theta(n)$. The experimental data from both Threads and JOCL are roughly constant and therefore satisfy Brent's Theorem with respect to T_∞ in this scenario. However this performance will decrease when the size of the input is greater than the number of processors P available, i.e. when T_1/P dominates T_∞.

Appendix

Algorithm 8: MAT-VEC(A, x)

input : Matrix A, vector x
output: vector y: outcome of matrix-vector multiplication of A and x

```
n ← A.rows
let y be a new vector of length n
parallel for i = 1 to n do
    y_i = 0
parallel for i = 1 to n do
    for j = 1 to n do
        y_i = y_i + a_ij x_j
return y
```

Algorithm 9: MAT-VEC-MAIN-LOOP(A, x, y, n, i, i')

```
if i == i' then
    for j = 1 to n do
        y_i = y_i + a_ij x_j
else
    mid = ⌊(i + i')/2⌋
    spawn MAT-VEC-MAIN-LOOP(A, x, y, n, i, mid)
    MAT-VEC-MAIN-LOOP(A, x, y, n, mid + 1, i')
    sync
```

Algorithm 10: P-MATRIX-MULTIPLY-RECURSIVE(C, A, B)

```
n = A.rows
if n==1 then
    c_11 = a_11 × b_11
else
    let T be a new n × n matrix
    partition A, B, C, and T into n/2 × n/2 submatrices
      A_11, A_12, A_21, A_22; B_11, B_12, B_21, B_22; C_11, C_12, C_21, C_22; and T_11, T_12, T_21, T_22; respectively
    spawn P-MATRIX-MULTIPLY-RECURSIVE(C_11, A_11, B_11)
    spawn P-MATRIX-MULTIPLY-RECURSIVE(C_12, A_11, B_12)
    spawn P-MATRIX-MULTIPLY-RECURSIVE(C_21, A_21, B_11)
    spawn P-MATRIX-MULTIPLY-RECURSIVE(C_22, A_21, B_12)
    spawn P-MATRIX-MULTIPLY-RECURSIVE(T_11, A_12, B_21)
    spawn P-MATRIX-MULTIPLY-RECURSIVE(T_12, A_12, B_22)
    spawn P-MATRIX-MULTIPLY-RECURSIVE(T_21, A_22, B_21)
    P-MATRIX-MULTIPLY-RECURSIVE(T_22, A_22, B_22)
    sync
    parallel for i = 1 to n do
        parallel for j = 1 to n do
            c_ij = c_ij + t_ij
```