CS255 HW2
Michael Nguyen

1. Suppose that we spawn P-FIB(n-2) in line 4 of P-FIB, rather than calling it as is done in the code. What is the impact on the asymptotic work, span, and parallelism?

It seems there is no impact on asymptotic work. Normally, calling line 4 keeps it on the main thread while the spawned subroutine, its child on line 3, runs in parallel. However, spawning a subroutine in line 4 will force the program to wait on line 5 when it is waiting for both subroutines to complete and synchronize together. Span does not seem affected if the subroutine of line 4 will be the same as if it was called normally. Because the work and span are not changed, the parallelism is also not changed.

2. Consider the following multithreaded pseudocode for transposing an *n x n* matrix *A* in place:

```
P-TRANSPOSE(A)
1       n = A.rows
2       parallel for j = 2 to n
3               parallel for i = 1 to j - 1
4                       exchange a_ij with a_ji
```

Analyze the work, span, and parallelism of the algorithm.

The serialized work has 2 nested for-loops, which gives $O(n^2)$ complexity. The span can be obtained from the exchange function in the inner loop of P-TRANSPOSE. Since it's running in parallel, it is a span of $\Theta(\log n)$ instead of $\Theta(n)$. Parallelism of the whole algorithm is $\frac{O(n^2)}{\Theta(log\ n)}$.

3. Give pseudocode for an efficient multithreaded algorithm that multiplies a *p x q* matrix by a *q x r* matrix. Your algorithm should be highly parallel even if any of p, q, and r are 1. Analyze your algorithm.

```
P-MATRIX-MULTIPLY(A, B)
1       p = A.rows
2       q = A.cols
3       r = B.cols
4       let C be a new p x r matrix
5               for k = 1 to q
6                       P-MATRIX-MULTIPLY-LOOP(A, C, p, r, k, q)
```

```
7       return C

P-MATRIX-MULTIPLY-LOOP(A, C, i, j, k, k')
1       if k == k'
2               for i = 1 to p
3                       for j = 1 to r
4                               C_ij = 0
5                               C_ij = C_ij + A_ik · B_kj
6       else mid = [(k+k')/2]
7               spawn P-MATRIX-MULTIPLY-LOOP(A, C, i, j, k, mid)
8               P-MATRIX-MULTIPLY-LOOP(A, C, i, j, mid+1, k')
9               sync
```

The serialized work has 3 nested for loops, giving $\Theta(n^3)$ complexity. The total span given is $\Theta(\log n) + \Theta(\log^2 n) = \Theta(\log^2 n)$. Parallelism is $\frac{\Theta(n^3)}{\Theta(log^2 n)}$.

4. Suppose we have an array of n many n×n matrices $A_1, A_2, ...A_i..., A_n$. Design a multithreaded algorithm which computes $\prod_{i=1}^{n} A_i$ with work at most $O(n^4)$ and span $O(\log^3 n)$.

```
N-MATRIX-SUM(array, n)
1       n = array.length
2       let M be the resulting matrix and N be an nxn matrix
3       for x = 1 to n
4               if x+1 < n
5                       spawn B = N-MATRIX-SUM-LOOP(array[x+1], N, n, x, n)
6               A = N-MATRIX-SUM-LOOP(array[x], N, n, x, n)
7               sync
8               MATRIX-ADD(A, B, 0, n)
9               MATRIX-ADD(M, A, 0, n)
10              x=x+2
11      return M

N-MATRIX-SUM-LOOP(A, N, n, k, k')
1       if k == k'
2               for i = 1 to n
3                       for j = 1 to n
```

```
4                               N_ij = 0
5                               N_ij = N_ij + A_ik
6       else mid = [(k+k')/2]
7               spawn N-MATRIX-SUM-LOOP(A, N, i, j, k, mid)
8               N-MATRIX-SUM-LOOP(A, N, i, j, mid+1, k')
9               sync
10      return N

MATRIX-ADD(M, N, i', j')
1       if (i' == j')
2               for i = 1 to n
3                       for j = 1 to n
4                               M_ij = M_ij + N_ij
5       else mid = [(i'+j')/2]
6               spawn MATRIX-ADD(M, N, i', mid)
7               MATRIX-ADD(M, N, mid+1, j')
8               sync
```

The serialized work has 3 nested for loops, giving $\Theta(n^3)$ complexity. The total span given is $\Theta(\log n) + \Theta(\log^2 n) + \Theta(\log^2 n) + \Theta(\log^3 n) = \Theta(\log^3 n)$. Parallelism is $\frac{\Theta(n^3)}{\Theta(log^3 n)}$.

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

Assuming we know the value for n:

```
1       Load i in acc0
2       acc1 = reg[i]
3       reg[0] = 0
4       Add acc1 to reg[0]
5       reg[i] = floor (reg[0] / n)
6       Halt
```

Coding part

Look up the number of cores the machine you are experimenting on has, and the number of GPU shader processors it has.

Computer: MacBook Air (2018), 2 cores, 1 GPU processor

If you plot time versus the length of vectors you compute the norms of, does it match what you'd expect in each case if Brent's Theorem were an equality rather than an inequality?

To review Brent's Theorem:

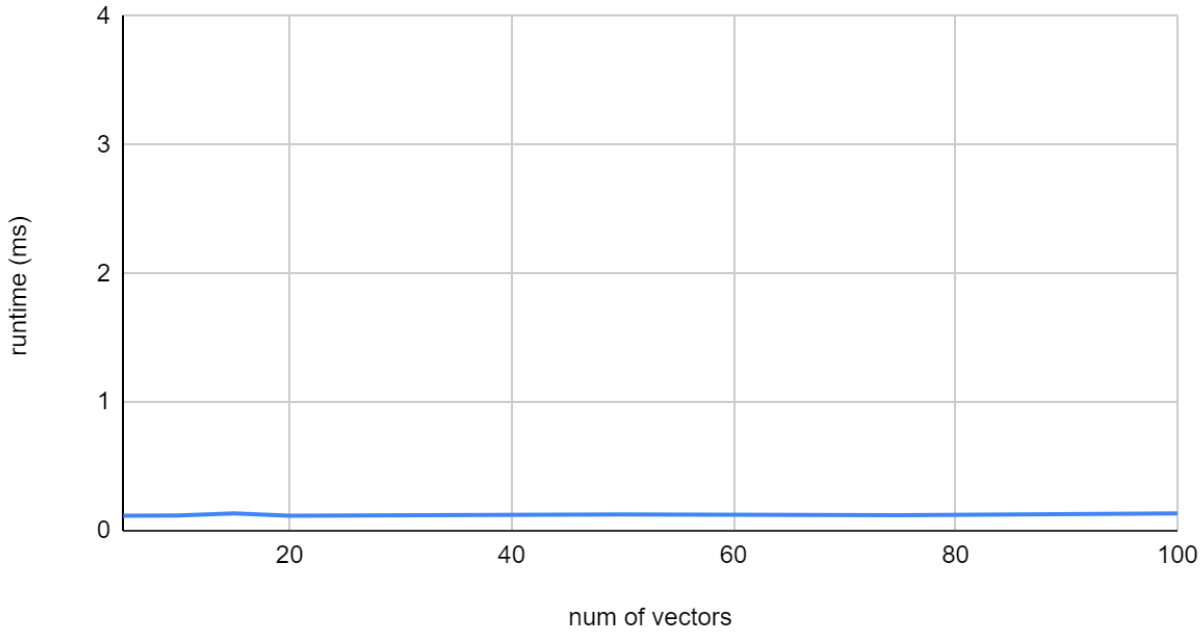
On an ideal parallel computer with P processors, a greedy scheduler executes a multithreaded computation with work T_1 and span T_∞ in time:

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

Let's say our algorithms have a span of O(log n).

If a greedy scheduler is a centralized scheduler that assigns as many strands to processors as possible in each step, then the work is able to be parallelized, in this case T_1 = O(log n). If Brent's theorem was an equality, then T_p = O(log n) for P many processors, since we can only hope for the best running time to be achieved in T_∞. Our experiments show that in the graphs below for n many vectors on both algorithms, the running time still remained the same no matter what.

ThreadMaxProd



JOCLMaxProd