Homework 8: Cache-Oblivious Algorithms

Then, answer the writeup questions in this handout and submit an individual writeup. See the following paper for more information on cache-oblivious algorithms:
https://dl.acm.org/citation.cfm?id=2/zero.noslash71383.

For this homework, assume that all matrices are stored in row-major layout.

1 Cache Complexity of Matrix Multiplication

During Lecture 14 we discussed the cache complexity of matrix multiplication of dimension \( n \), with tall cache assumption of size \( M \) and cache line size \( B \). For the naive approach, there were two cases: 1) If \( n > M/B \), then \( \Theta(n^3) \) cache misses occur, and 2) if \( M^{1/2} < n < M/B \), then \( \Theta(n^3/B) \) cache misses occur. For the blocking approach, with block size \( s < M^{1/2} \), \( \Theta(n^3/BM^{1/2}) \) cache misses occur. The cache-oblivious approach achieves the same complexity as the blocking approach without the need of the voodoo parameter \( s \).

Checkoff Item 1: Assume we want to multiply two rectangular matrices: \( m \times n \) with \( n \times r \). Given the same tall cache assumption, please analyze the complexity for one of the following four cases: the two cases for the naive approach \( (n > M/B \) and \( M/r < n < M/B) \), the block approach, and the cache-oblivious approach. You may pick whichever case you want to analyze.

2 Tableau Construction

Consider the tableau-construction problem from Lecture 8. The problem involves filling an \( N \times N \) tableau, where each entry of the tableau is calculated as a function of some of its neighbors. To be specific, the equation to fill an element of the tableau would take the form

\[
A[i][j] = f(A[i-1][j-1], A[i][j-1], A[i-1][j])
\]

where \( f \) is an arbitrary function.
2.1 Iterative Formulation

Consider the code snippet in Figure 1 below.

```c
#define A(i, j) A[N + (i) - (j) - 1]

void tableau(double *A, size_t N) {
    for (size_t i = 1; i < N; i++) {
        for (size_t j = 1; j < N; j++) {
            A(i, j) = f(A(i-1, j-1), A(i, j-1), A(i-1, j));
        }
    }
}
```

**Figure 1:** A simple, iterative loop for filling a tableau.

In this problem, we are only interested in computing the final value of the tableau, stored in \( A(N-1, N-1) \), and hence we really only need \( 2N - 1 \) amount of space during computation. Thus, the algorithm declares \( A \) as an array of size \( 2N - 1 \).

The algorithm initializes the first row and first column of the tableau, and invokes the `tableau` function as shown in Figure 2.

```c
for (size_t i = 0; i < N; i++) {
    A(i, 0) = INIT_VAL;
}
for (size_t j = 0; j < N; j++) {
    A(0, j) = INIT_VAL;
}
tableau(A, N);
res = A(N - 1, N - 1);
```

**Figure 2:** Initializing and calling the iterative `tableau` function.

**Write-up 1:** Explain why \( 2N - 1 \) space is sufficient and how the `tableau` function utilizes the \( 2N - 1 \) space.

Recall the tall cache assumption, which states that \( B^2 < \alpha M \), where \( B \) is the size of the cache line, \( M \) is the size of the cache, and \( \alpha \leq 1 \) is a constant.
Write-up 2: Assuming that an optimal replacement strategy holds and that the cache is tall, give a tight upper bound on the cache complexity $Q(n)$ for each of the following cases using $O$ notation, where $c \leq 1$ is a sufficiently small constant:

1. $n \geq cM$
2. $n < cM$

2.2 Recursive Formulation

Now consider the code snippet for a recursive tableau implementation, as shown in Figure 3. This algorithm similarly uses only $2N - 1$ amount of space, initializes the array $A$, and invokes the `recursive_tableau` function as shown in Figure 4. This recursive algorithm divides the tableau into four quadrants to compute. As discussed in Lecture 8 (slide 88), after the first quadrant is done computing, we can then compute the second and third quadrants in parallel. Parallelizing this way gives us work as $O(n^2)$ and span as $O(n^{\log_3 4})$ with parallelism as $O(n^{2 - \log_3 4})$. We also discussed (slide 92) a more parallel construction that divides up the tableau 9 ways.
```c
for (size_t i = 0; i < N; i++) {
    A(i, 0) = INIT_VAL;
}
for (size_t j = 0; j < N; j++) {
    A(0, j) = INIT_VAL;
}
if (N > 1) {
    recursive_tableau(A, 1, N, 1, N);
}
res = A(N-1, N-1);
```

**Figure 4:** Initializing and calling the `recursive_tableau` function.

**Write-up 3:** Derive the general formula for work and span, assuming a $k^2$-way tableau construction (i.e., the tableau is divided up into $k^2$ pieces of size $n/k \times n/k$).

**Write-up 4:** Answer the following questions assuming that an optimal replacement strategy holds and that the cache is tall.

1. Show the recurrence relation for the cache complexity $Q(n)$ using the 4-way construction of the `recursive_tableau` function.

2. Draw the recursion tree and label the internal nodes and leaves with their cache complexity $Q(n)$. What’s the height of the recursion tree?

3. How many leaves are in the recursion tree?

4. Using the recursion tree and the recurrence relation, derive a simplified expression for $Q(n)$.

**Write-up 5:** Answer the following question assuming that an optimal replacement strategy holds and that the cache is tall. Assuming a $k^2$-way tableau construction, show that if we are “unlucky,” where a subpiece is just slightly above the cache size, then we have $Q(n) = \Theta(n^2k/MB)$. Also show that if we are lucky and this situation does not arise, then we have $Q(n) = \Theta(n^2/MB)$. 
