Practice Quiz 2

Name: ______________________________

Instructions

- DO NOT open this quiz booklet until you are instructed to do so.
- This quiz booklet contains 17 pages, including this one. You have 80 minutes to earn 80 points.
- This quiz is closed book, but you may use one handwritten, double-sided 8 1/2″ × 11″ crib sheet and the Master Method card handed out in lecture.
- When the quiz begins, please write your name and Kerberos on this coversheet, and write your name or Kerberos on the top of each page, since the pages may be separated for grading.
- Some of the questions are true/false, and some are multiple choice. You need not explain these answers unless you wish to receive partial credit if your answer is wrong. For these kinds of questions, **since incorrect answers will be penalized, do not guess unless you are reasonably sure**.
- Good luck!

Number	**Question**	**Parts**	**Points**	**Score**	**Grader**
0	Name on Every Page	17	1		
1	True or False	9	18		
2	Heap space for Cilk programs	3	7		
3	Concurrent free lists	2	8		
4	Parallel Strassen	5	15		
5	Space usage for Strassen	6	18		
6	Cache-oblivious Strassen	3	13		
	Total		80		

1 True or false (9 parts, 18 points)

Incorrect answers will be penalized, so do not guess unless you are reasonably sure. You need not justify your answer, unless you want to leave open the possibility of receiving partial credit if your answer is wrong. Comments will have no impact on a correct answer.

1.1

There are generally fewer conflict misses in a set-associative cache than a direct-mapped cache.

True **False**

1.2

If there is a TLB miss, then the corresponding cache line is not already in cache.

True **False**

1.3

When dealing with multiple pages, a new block should be allocated from the free list for the emptiest page to incur the fewest TLB misses.

True **False**

1.4

It is generally more important for a memory allocator to quickly allocate small blocks than to quickly allocate large blocks.

True **False**

1.5

Streaming writes allow for higher memory bandwidth utilization by bypassing cache with the additional cost of higher latency.

True **False**

1.6

On `x86`, sequentially consistent behavior can always be achieved through the use of memory fences.

True **False**

1.7

If a program that runs on 2 threads protects its critical sections using Peterson's algorithm, then it is free of determinacy races.

True **False**

1.8

If a program that runs on 2 threads protects its critical sections using Peterson's algorithm, then it is free of data races.

True **False**

1.9

Ignoring the effects of true and false sharing, on `x86`, a compare-and-swap operation is just as fast as an equivalent comparison and assignment.

True **False**

2 Heap space for Cilk programs (3 parts, 7 points)

Consider the following parallel Cilk code.

```
1 void foo(int depth, size_t X) {
2   if (depth < 1) return;
3   cilk_spawn foo(depth-1, X);
4   void *ptr = malloc(X);
5   cilk_sync;
6   free(ptr);
7 }
```

Assume that the stack is large enough such that for all inputs of `depth` the code does not overflow the stack. Furthermore, assume that `foo` is run using an optimal allocator that achieves perfect utilization.

2.1

Suppose that `foo` is executed using just 1 worker. Argue that the space needed to satisfy the calls to `malloc` from `foo` is `X`.

2.2

Now suppose that `foo` is executed using 2 workers. In terms of `depth` and `X`, what is the maximum amount of space needed to satisfy the calls to `malloc` from `foo`? (Hint: Consider an execution of `foo` in which one worker steals line 4 from the other worker in every invocation of `foo`.)

2.3

In lecture, we saw a theorem stating that a P-worker execution of any Cilk program uses at most P times the space used in a serial execution of that Cilk program. Why does the theorem not apply to `foo`, or does it?

3 Concurrent free lists (2 parts, 8 points)

Ben Bitdiddle is parallelizing a program that uses a free list to handle the allocation of objects of a fixed size `X`. Ben's serial code uses a global free-list data structure of type `FreeList_t`, defined as follows:

```
typedef struct Node_t {
  /* A */
  struct Node_t *next;
} Node_t;

typedef struct {
  /* B */
  Node_t *head;
} FreeList_t;

void push(FreeList_t *fl, Node_t *node) {
  /* C */
  node->next = fl->head;
  /* D */
  fl->head = node;
  /* E */
}

Node_t *pop(FreeList_t *fl) {
  /* F */
  Node_t *node = fl->head;
  /* G */
  if (NULL != node) {
    /* H */
    fl->head = node->next;
    /* I */
  }
  /* J */
  return node;
}
```

Ben considers parallelizing his free list by implementing thread-local free lists. Each thread would only `push` or `pop` storage from its local free list. If the thread's local free list is empty, then the thread simply calls the system's `malloc`.

3.1

Describe a performance problem that Ben might see with thread-local free lists that can cause Ben's parallel program to use much more space than his original serial program.

Ben decides to modify his serial free-list code into a global free list, with `push` and `pop` operations synchronized using mutex locks. Ben wants to add as few instructions as possible to his free-list code to implement a correct and efficient global free list. Ben's free list code contains convenient labels `A` through `J` at lines where Ben might insert a new statement.

3.2

Write an `X` in each table cell below where Ben should insert the statement in the corresponding column at the line specified by the corresponding row. Each row should contain at most one `X`.

	`mutex_t L;`	`lock(fl->L);`	`unlock(fl->L);`	`lock(node->L);`	`unlock(node->L);`
A					
B					
C					
D					
E					
F					
G					
H					
I					
J					

4 Parallel Strassen (5 parts, 15 points)

Recall the (cache-oblivious) divide-and-conquer matrix multiplication algorithm from lecture and from Homework 8. Given two $n \times n$ matrices A and B, where n is a power of 2, the algorithm computes the product $C = A \cdot B$ in $\Theta(n^3)$ work as follows. First, the algorithm partitions the matrices A, B, and C into four quadrants:

$$A = \begin{bmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{bmatrix}, B = \begin{bmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{bmatrix}, C = \begin{bmatrix} C_{1,1} & C_{1,2} \\ C_{2,1} & C_{2,2} \end{bmatrix}$$

The algorithm then computes the quadrants of C using eight recursive multiplications:

$$\begin{aligned} C_{1,1} &= A_{1,1} \cdot B_{1,1} + A_{1,2} \cdot B_{2,1} \\ C_{1,2} &= A_{1,1} \cdot B_{1,2} + A_{1,2} \cdot B_{2,2} \\ C_{2,1} &= A_{2,1} \cdot B_{1,1} + A_{2,2} \cdot B_{2,1} \\ C_{2,2} &= A_{2,1} \cdot B_{1,2} + A_{2,2} \cdot B_{2,2} \ . \end{aligned}$$

In 1969 Volker Strassen invented an asymptotically faster matrix-multiplication algorithm and showed that the n^3 matrix multiplication algorithm was not optimal. Strassen's algorithm partitions the matrices into quadrants as before, but then recursively computes **seven** intermediate matrix products: (Do not waste time verifying that these formulae are correct.)

$$\begin{aligned} M_1 &= (A_{1,1} + A_{2,2}) \cdot (B_{1,1} + B_{2,2}) \\ M_2 &= (A_{2,1} + A_{2,2}) \cdot B_{1,1} \\ M_3 &= A_{1,1} \cdot (B_{1,2} - B_{2,2}) \\ M_4 &= A_{2,2} \cdot (B_{2,1} - B_{1,1}) \\ M_5 &= (A_{1,1} + A_{1,2}) \cdot B_{2,2} \\ M_6 &= (A_{2,1} - A_{1,1}) \cdot (B_{1,1} + B_{1,2}) \\ M_7 &= (A_{1,2} - A_{2,2}) \cdot (B_{2,1} + B_{2,2}) \ . \end{aligned}$$

The quadrants of the C matrix can be computed in terms of $M_1, M_2, \ldots, M_7$ as follows:

$$\begin{aligned} C_{1,1} &= M_1 + M_4 - M_5 + M_7 \\ C_{1,2} &= M_3 + M_5 \\ C_{2,1} &= M_2 + M_4 \\ C_{2,2} &= M_1 - M_2 + M_3 + M_6 \ . \end{aligned}$$

The next page presents pseudocode for a parallel implementation of Strassen's algorithm for $n \times n$ matrices, where n is an even power of 2. Do not waste time trying to understand how the code works. You only need to understand its parallel structure.

```
// Pseudocode for a parallel version of Strassen's algorithm to
// multiply two square matrices A, B. The matrices have side length n
// in each recursive call, where n is an even power of 2.
// Assume that A11...A22, B11...B22, C11...C22 are quadrants of A, B, C.
strassen(A, B, n) {
  // Coarsened base case
  if (n < BASE_CASE_SIZE) return base_case(A, B, n);

  // Compute the quadrants A11, A12, etc. for A, B, and C.
  // Create 8 temporary matrices of size n^2/4.
  Temp[8];
  for (int i = 0; i < 8; ++i) Temp[i] = malloc(n * n / 4);

  //////// Block 1
  cilk_spawn { Temp[0] = A21 + A22;
               Temp[1] = strassen(Temp[0], B11, n/2) }; // M2
  cilk_spawn { Temp[2] = B12 - B22;
               Temp[3] = strassen(A11, Temp[2], n/2) }; // M3
  cilk_spawn { Temp[4] = B21 - B11;
               Temp[5] = strassen(A22, Temp[4], n/2) }; // M4
  Temp[6] = A11 + A12;
  Temp[7] = strassen(Temp[6], B22, n/2);                // M5
  cilk_sync; // End block 1

  //////// Block 2
  cilk_spawn { C11 = Temp[5] - Temp[7] }; // C11 = M4 - M5
  cilk_spawn { C12 = Temp[3] + Temp[7] }; // C12 = M3 + M5
  cilk_spawn { C21 = Temp[1] + Temp[5] }; // C21 = M2 + M4
  C22 = Temp[3] - Temp[1];                // C22 = M3 - M2
  cilk_sync; // End block 2

  //////// Block 3
  cilk_spawn { Temp[0] = A11 + A22;
               Temp[1] = B11 + B22;
               Temp[2] = strassen(Temp[0], Temp[1], n/2) }; // M1
  cilk_spawn { Temp[3] = A12 - A22;
               Temp[4] = B21 + B22;
               C11 += strassen(Temp[3], Temp[4], n/2) };    // C11 += M7
  Temp[5] = A21 - A11;
  Temp[6] = B11 + B12;
  C22 += strassen(Temp[5], Temp[6], n/2) ;                  // C22 += M6
  cilk_sync; // End block 3

  //////// Block 4
  cilk_spawn{ C11 += Temp[2] }; // C11 += M1
  C22 += Temp[2];               // C22 += M1
  cilk_sync; // End block 4

  for (int i = 0; i < 8; ++i) free(Temp[i]);
  return C;
}
```

We will first analyze the work and span of this Strassen code. Assume that this Strassen code uses a routine for adding or subtracting two $n \times n$ matrices in $\Theta(n^2)$ work and $\Theta(\lg n)$ span, such as in line 52.

4.1

Explain why the recurrence for the work $T_1(n)$ of the parallel Strassen code satisfies

$$T_1(n) = 7T_1(n/2) + \Theta(n^2). \tag{1}$$

4.2

The solution to the work recurrence (1) is

$$T_1(n) = \Theta(n^a \lg^b n)$$

for some values a and b. Select the correct values for a and b from the choices below.

A $a = 2$, $b = 0$

B $a = 2$, $b = 1$

C $a = \lg 7$, $b = 0$

D $a = \lg 7$, $b = 1$

E $a = 3$, $b = 0$

F None of the above

4.3

The recurrence for the span $T_\infty(n)$ of this parallel Strassen code is composed from the spans of four blocks in the code: Block 1 (lines 51–60), Block 2 (lines 62–67), Block 3 (lines 69–79), and Block 4 (lines 81–84). For each block, write an `X` in the table below to identify the span of that block.

	$\Theta(\lg n)$	$T_\infty(n/2) + \Theta(\lg n)$	$3T_\infty(n/2) + \Theta(\lg n)$	$4T_\infty(n/2) + \Theta(\lg n)$
Block 1				
Block 2				
Block 3				
Block 4				

4.4

Describe how the spans of those four blocks compose to make the recurrence for the span of this parallel Strassen code equal to

$$T_\infty(n) = 2T_\infty(n/2) + \Theta(\lg n). \tag{2}$$

4.5

The solution to the span recurrence (2) is

$$T_\infty(n) = \Theta(n^a \lg^b n)$$

for some values a and b. Select the correct values for a and b from the choices below.

A $a = 0, b = 1$

B $a = 0, b = 2$

C $a = 1, b = 0$

D $a = 1, b = 1$

E $a = 2, b = 0$

F $a = 2, b = 1$

G $a = \lg 7, b = 0$

H $a = \lg 7, b = 1$

I None of the above

5 Space usage for Strassen (6 parts, 18 points)

This question studies the space usage during an execution of the parallel Strassen code described in Question 4. As lines 48–49 in the pseudocode show, this parallel Strassen code uses $\Theta(n^2)$ temporary space per recursive call.

5.1

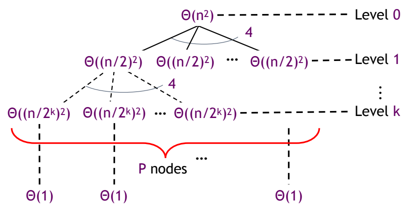
Describe why the total amount of temporary space $S_1(n)$ used by a 1-worker execution of this parallel Strassen program satisfies the following recurrence:

$$S_1(n) = S_1(n/2) + \Theta(n^2). \tag{3}$$

5.2

Solve the recurrence (3), and write your answer using Θ-notation.

Although the Cilk scheduler provides a weak bound on the space used by a P-worker execution of this parallel Strassen code, we can prove a stronger bound by considering the worst-case recursion tree for this code. It turns out that this worst-case recursion tree for space usage is as illustrated below:


Θ(n²)
Level 0
4
Θ((n/2)²) Θ((n/2)²) ⋯ Θ((n/2)²)
Level 1
4
Θ((n/2ᵏ)²) Θ((n/2ᵏ)²) ⋯ Θ((n/2ᵏ)²)
Level k
P nodes
Θ(1) Θ(1) Θ(1)

This recursion tree recursively branches 4 ways until it reaches the first level k that contains P nodes. Each worker then serially executes the computation under a distinct level-k node, which the recursion tree models with 1-way branching after level k.

5.3

Why is the branching factor 4 for the top part of this recursion tree?

5.4

Argue that the total space usage in the top part of the recursion tree, in all levels $i \le k$, is $\Theta(n^2 \log_4 P)$.

5.5

Argue that the total space usage in the bottom part of the recursion tree, in all levels $i > k$, is $\Theta(n^2)$.

5.6

What is the total worst-case space used by an execution of parallel Strassen? Express your answer in Θ-notation.

6 Cache-oblivious Strassen (3 parts, 13 points)

This question examines the cache-efficiency of Strassen's algorithm for matrix multiplication. For this question, we will consider a *serial* implementation of Strassen's algorithm, such as the serial elision of the parallel implementation described in Question 4. For this problem, assume an ideal cache model (fully associative with an optimal or LRU replacement policy, as appropriate) with cache size $\mathcal{M}$ and cache-line length $\mathcal{B}$. Assume that the cache is tall (i.e. $\mathcal{M} = \omega(\mathcal{B}^2)$). Furthermore, assume that Strassen's algorithm is run using an optimal allocator that achieves perfect utilization. Assume that the input matrix is stored in row-major order, and also assume that the routine for adding or subtracting two m-by-m submatrices is $\Theta(m^2/\mathcal{B})$.

The recurrence for the worst-case number $Q(n)$ of cache misses incurred by Strassen's algorithm when multiplying two $n \times n$ matrices is given by

$$Q(n) = \begin{cases} \Theta(n^2/B) & \text{if } 0 \le n < c\sqrt{\mathcal{M}} \text{ for sufficiently small constant } c \le 1; \\ 7Q(n/2) + \Theta(n^2/B) & \text{otherwise.} \end{cases} \tag{4}$$

6.1

Why is $Q(n) = 7Q(n/2) + \Theta(n^2/B)$ for n sufficiently large?

6.2

Why is $Q(n) = \Theta(n^2/B)$ for $n < c\sqrt{\mathcal{M}}$?

6.3

Sketch the recursion tree for the recurrence for (4). Compute the height and the number of leaves of the recursion tree, and label the internal nodes and leaves of the tree with the corresponding number of cache misses incurred at them. Solve the recurrence, providing a tight asymptotic bound in simple form.

Sketch of recursion tree for $Q(n)$:

Height =

Number of leaves =

Misses per leaf =

$Q(n) =$