6.172 Quiz 1

Adriano Hernandez

TOTAL POINTS

73 / 80

QUESTION 1

Question 1. 12 pts

1.1 2 / 2

✓ **+ 2 pts Correct (false)**

+ 0 pts Incorrect

1.2 2 / 2

✓ **+ 2 pts Correct (false)**

+ 0 pts Incorrect

1.3 0 / 2

+ 2 pts Correct (false)

✓ **+ 0 pts Incorrect**

1.4 2 / 2

✓ **+ 2 pts Correct (true)**

+ 0 pts Incorrect

1.5 2 / 2

✓ **+ 2 pts Correct (true)**

+ 0 pts Incorrect

1.6 2 / 2

✓ **+ 2 pts Correct (true)**

+ 0 pts Incorrect

QUESTION 2

Question 2 24 pts

2.1 4 / 4

✓ **+ 4 pts Correct (B)**

+ 0 pts Incorrect

2.2 4 / 4

✓ **+ 4 pts Correct (B)**

+ 0 pts Incorrect

2.3 4 / 4

✓ **+ 4 pts Correct (B)**

+ 0 pts Incorrect

2.4 4 / 4

✓ **+ 4 pts Correct (B)**

+ 0 pts Incorrect

2.5 4 / 4

✓ **+ 4 pts Correct (D)**

+ 0 pts Incorrect

2.6 0 / 4

+ 4 pts Correct (C)

✓ **+ 0 pts Incorrect**

QUESTION 3

Question 3 44 pts

3.1 12 / 12

✓ **+ 2 pts a = 1**

✓ **+ 2 pts b = 2**

✓ **+ 2 pts c = 1**

✓ **+ 2 pts k = 0**

✓ **+ 2 pts c = 1 (second one)**

✓ **+ 2 pts k = 0 (second one)**

3.2 14 / 14

✓ **+ 2 pts a = 1**

✓ **+ 2 pts b = 2**

✓ **+ 2 pts c = 0**

✓ **+ 2 pts k = 1**

✓ **+ 2 pts c = 0 (second one)**

✓ **+ 2 pts k = 2 (second one)**

✓ + 2 pts n/lg^2 n

3.3 12 / 12

✓ + 2 pts **E** = `0`

✓ + 2 pts **F** = `0`

✓ + 2 pts **A** = `1`

✓ + 2 pts **B** = `-1`

✓ + 2 pts **C** = `scratch`

✓ + 2 pts **D** = `scratch[i]`

3.4 2.5 / 3

✓ + 2.5 pts **$$\Theta(W(n) + n)$$**

+ 3 pts $$\Theta(W(n))$$

+ 0 pts Incorrect.

- 1 pts Incorrectly assumed W(n) = $$\Theta(n)$$

3.5 2.5 / 3

✓ + 2.5 pts **$$\Theta(S(n) + \log n)$$**

+ 3 pts $$\Theta(S(n))$$

+ 0 pts Incorrect.

Quiz 1

- DO NOT OPEN this quiz booklet until you are instructed to do so. This quiz is closed book, but you may use one handwritten, double-sided 8 1/2″ × 11″ crib sheet. Please put all personal items unrelated to the exam on the floor.
- Please read these instructions carefully. This quiz booklet contains 12 pages, including this one, but excluding 3 pages of appendix (an x86 assembly guide, an LLVM IR guide, and the master theorem) and 1 scratch paper. You have 80 minutes to earn 80 points.
- Please write your name and Kerberos on this cover sheet in the space provided. When the quiz begins, please additionally write your name or Kerberos at the top of each page, since the pages will be separated during grading.
- Good luck!

Name: Adriano Hernandez Kerberos: adrianoh

Problem	Title	Parts	Points	Score	Grader
1	True or False	6	12		
2	Multiple Choice	6	24		
3	Parallel Algorithms	5	44		
	Total		**80**		

1 True or False (6 parts, 12 points)

Circle the correct answer for each statement below. You need not explain your answers.

1.1 Suppose that a program `pow.c` contains the statement `x8 = x*x*x*x*x*x*x*x;`, where `x` has a floating-point type. When compiling with `clang -O3 pow.c`, the compiler might optimize the computation of `x8` by computing `x2 = x*x; x4 = x2*x2; x8 = x4*x4;` (written here in C for convenience).

True **False** Not associative.

1.2 In lecture we saw that using pthreads to parallelize recursive Fibonacci leads to only a 1.5× speedup with 2 cores because of the overhead of thread creation.

True **False** It was unbalanced (i.e $F_{n-2} < F_{n-2}$) by that ratio.

1.3 The Cilk statement sequence `cilk_spawn f(); cilk_spawn g(); cilk_sync;` causes `f()` and `g()` to execute in parallel.

True **False**

1.4 If an ostensibly deterministic Cilk program contains multiple races when run on a particular input, Cilksan guarantees to report at least one race.

True **False** It's supposedly "proven" to catch races. Would be nice to see the text.

1.5 For a greedy scheduler operating on a deterministic program, an incomplete step is guaranteed to reduce the span of the yet-to-be computed trace of the program.

True **False** Complete → P
inc. → < P

1.6 When using the `time` shell command, it is possible for "user" time to be greater than "real" time for a program execution.

True **False** User → Σ over cores

2 Multiple Choice (6 parts, 24 points)

Circle the letter corresponding to the most appropriate answer. Circling multiple letters will receive zero points.

2.1 Assume that the function `int popcount(uint32_t a)` counts how many 1's are set in the binary representation of a. Given the implementation of function `baz()` below, what does `baz(x)` do?

```
01 bool baz(uint32_t a) {
02     a ^= a >> 16;
03     a ^= a >> 8;
04     a ^= a >> 4;
05     a ^= a >> 2;
06     a ^= a >> 1;
07     return (a & 1) == 0;
08 }
```

- A Returns true when and only when `popcount(x)` is odd
- B Returns true when and only when `popcount(x)` is even
- C Returns true when and only when the XOR of all the bits of `x` is 1
- D Returns true when and only when $a = 2^N - 1$ for some integer N
- E None of the above

2.2 Suppose that we represent a 32×32 bit matrix with a length-32 array `A` of 32-bit unsigned integers, where the `i`th row of the matrix is `A[i]`. Suppose also that the utility function `GET_BIT(x, j)` returns the `j`th bit of 32-bit unsigned integer `x`. Similarly, `SET_BIT(&x, j, v)` sets the `j`th bit of the 32-bit unsigned integer `x` to the least-significant bit of the integer `v`.

Many cryptographic computations involve bit-matrix multiplication, where scalar sum and product are performed over 1-bit values modulo 2. Interestingly, over Boolean values, scalar multiplication is equivalent to an AND, and addition is equivalent to an XOR. The function `multiply_base()` in lines 9–19 of the code below implements bit-matrix multiplication without this optimization.

Because you're not happy with the performance of this code, and inspired by Project 1 and Homework 3, you decide to optimize using word parallelism and your knowledge of Boolean arithmetic. Your efforts lead to function `multiply_base_new()` in lines 21–30, but you are not quite done.

```
09 void multiply_base(uint32_t *A, uint32_t *B, uint32_t *C) {
10     for (int i = 0; i < 32; i++) {
11         for (int j = 0; j < 32; j++) {
12             uint32_t sum = 0;
13             for (int k = 0; k < 32; k++) {
14                 sum += GET_BIT(A[i], k) * GET_BIT(B[k], j);
15             }
16             SET_BIT(&C[i], j, sum);
17         }
18     }
19 }
20
21 void multiply_base_new(uint32_t *A, uint32_t *B, uint32_t *C) {
22     for (int i = 0; i < 32; i++) {
23         C[i] = 0;
24         for (int j = 0; j < 32; j++) {
25             uint32_t Brow = B[j];
26             uint32_t Aij = GET_BIT(A[i], j); // 0 or 1
27             ____________ // <== what goes here?
28         }
29     }
30 }
```

LSB is [redacted] modulo 2 sum

Which of the following statements can serve as the missing code in line 27?

A `C[i] ^= Aij & Brow;`

B `C[i] ^= -Aij & Brow;` → all zeroes or all 1's (i.e. negative vectors / two's complement representation)

C ~~`C[i] ^= ~Aij & Brow;`~~

D ~~`C[i] ^= ~Aij && Brow;`~~

E None of the above

2.3 Nuria Numbercruncher invents the ***length-tagged array*** data structure, which she writes in lines 31–34 of the C code below. She uses this data structure for a function which squares all elements in an array. Initially, she writes this function as `square_array_inc()` in lines 36–40. She finds the performance of `square_array_inc()` disappointing, however, even with `-O3` enabled. She tries a new approach with `square_array_dec()` in lines 42–46 and finds this implementation is much faster! What's going on?

```
31 struct array {
32     int *elements;
33     int length;
34 };
35
36 void square_array_inc(struct array *arr) {
37     for (int i = 0; i < arr->length; i++) {
38         arr->elements[i] *= arr->elements[i];
39     }
40 }
41
42 void square_array_dec(struct array *arr) {
43     for (int i = arr->length - 1; i >= 0; i--) {
44         arr->elements[i] *= arr->elements[i];
45     }
46 }
```

on the int length?

restrict?

why? no sense!

A Iterating backwards has better cache locality

B The compiler was only able to vectorize the backwards iteration

why?

C Using `i >= 0` allows for better branch prediction same either way!

D Loading elements of `arr->elements[i]` triggers the prefetcher only when iterating backwards maybe?!

E None of the above

2.4 How could the forward-iteration implementation of `square_array_inc()` be made similarly fast?

tell it that length is not in elements

A Index in reverse order using `arr->elements[arr->length - 1 - i]`

B Precompute the loop trip count `arr->length` before the loop

so it did this by accident previously?

C Change the type of the loop index to `unsigned int`

D Use the loop unroll `#pragma` maybe yes/no?

E None of the above

2.5 Consider the function `void operator(uint32_t *a, uint32_t *b, uint32_t *c)` whose LLVM IR is shown here:

```
47 define dso_local void @operator(
48 i32* nocapture readonly %0,
49 i32* nocapture readonly %1,
50 i32* nocapture %2
51 ) local_unnamed_addr #0 {
52   %4 = load i32, i32* %0, align 4
53   %5 = load i32, i32* %1, align 4
54   %6 = shl i32 %5, 2
55   %7 = add i32 %6, 5
56   %8 = mul i32 %7, %4
57   store i32 %8, i32* %2, align 4
58   ret void
59 }
```

Which of the following is a potential C implementation of `operator()`?

A `*c = 4*(*a)*(*b) + 20*(*b)`

B `*c = 4*(*a)*(*b) + 5*(*b)`

C `*c = 4*(*b)*(*c) + 5*(*a)`

D `*c = 5*(*a) + 4*(*a)*(*b)`

E None of the above

2.6 You have defined a function which takes in two length-16384 arrays of 32-bit integers as arguments, and you find the following snippet near the entry point of the generated assembly for the function. What does it do?

(A cheat sheet for x86 assembly language is provided at the end of this booklet.)

```
60 # %bb.0:
61          leaq    65536(%rsi), %rax
62          cmpq    %rdi, %rax
63          jbe     .LBB0_2
64 # %bb.1:
65          leaq    65536(%rdi), %rax
66          cmpq    %rsi, %rax
67          jbe     .LBB0_2
68 # %bb.4:
69          xorl    %eax, %eax
70 .LBB0_2:
71          ...
```

load effective address

rdi + 65536 ≥ rsi
OR
rsi + 65536 ≥ rdi

if jump

jbe same as jle

according to x86 manual

rdi rsi

(A) If there is not enough space to make duplicates of the argument arrays, clear eax

(B) If the first values in each array are less than 65536, clear eax

(C) If the arrays overlap, clear eax

(D) If the first and last values of the two arrays are different, clear eax

(E) None of the above

E

max uint_16

if they DON'T overlap, clear eax

rdi ≤

if [(rsi+len) ≥ rdi] or [(rdi+len) ≥ rsi]
skip

3 Parallel Algorithms (5 parts, 44 points)

Unimpressed by the pseudocode for parallel prefix sum in homework 5, Theodore Threadripper writes his own Cilk implementation of an "in-place" prefix-sum algorithm. The function `prefix_sum()` shown below takes three arguments: an array `A`, the length `len` of `A`, and a parameter `r` which starts at 2 and doubles for each recursive call until `r` is larger than `len`. The function modifies `A` such that for all $i = 0, 1, \ldots, \texttt{len} - 1$, the sum $\texttt{A[0]} + \texttt{A[1]} + \cdots + \texttt{A[i]}$ is stored at `A[i]` after `prefix_sum()` has run.

```
72 void prefix_sum(int *A, int len, int r) {
73   // The initial call is prefix_sum(A, len, 2)
74
75   if (r > len) return;
76   cilk_for (int i = r-1; i < len; i += r) {
77     A[i] += A[i-r/2];
78   }
79   prefix_sum(A, len, 2*r);
80   cilk_for (int i = r-1; i < len-r/2; i += r) {
81     A[i+r/2] += A[i];
82   }
83 }
```

double stride = halve strides

$n + n + \frac{n}{2} + \frac{n}{2} + \frac{n}{4} + \frac{n}{4} + \cdots$

$\rightarrow \Theta(n)$

3.1 A work recurrence for `prefix_sum()` on an input of length n can be expressed in the form

$$T_1(n) = aT_1(n/b) + \Theta(n^c \log^k n) .$$

run $\frac{n}{r}$ times

Fill in the value of each constant below.

$a =$ 1

$b =$ 2

$c =$ 1

$k =$ 0

The solution to the work recurrence for $T_1(n)$ can be expressed in the form

$$\Theta(n^c \lg^k(n)) .$$

Fill in the value of each constant below.

$c =$ 1

$k =$ 0

3.2 A span recurrence for `prefix_sum()` on an input of length n can be expressed in the form

$$T_\infty(n) = aT_\infty(n/b) + \Theta(n^c \log^k n) .$$

Fill in the value of each constant below.

$a = 1$

$b = 2$

$c = 0$

$k = 1$ (actually n/n but ok)

The solution to the span recurrence for $T_\infty(n)$ can be expressed in the form

$$\Theta(n^c \lg^k(n)) .$$

Fill in the value of each constant below.

$\log_b a = \log_2 1 = 0$

$n^0 \log^1(n) = \log^1(n)$

$+$

$\log^2(n)$

$c = 0$

$k = 2$

What is the asymptotic parallelism of `prefix_sum()`?

Parallelism = Work/Span = $T_1/T_\infty = \frac{n}{\log^2(n)}$

3.3 A string of parentheses is ***well formed*** if every open parenthesis is eventually followed by a close parenthesis, and the number of open parentheses equals the number of close parentheses. Below are examples of well-formed and malformed strings of parentheses:

unique (TA told me)

- malformed: `( ( ) ( ( ) ( ) )`
- malformed: `) (`
- well formed: `( ( ( ) ( ) ( ( ) ) ) )`

~~)) ((~~ well formed (by your definition)

Assume that `void bb_prefix_sum(int *A, int n)` is an efficient black-box implementation of in-place prefix sum over a length-`n` array of integers, `A`. We shall use `bb_prefix_sum()` to construct a parallel algorithm for detecting whether a string of parentheses is well formed.

The next page shows the skeleton of a Cilk function, `bool wf_paren(char *P_in, int n)`, which returns true if a length-`n` input string `P_in` of parentheses is well-formed, and false otherwise. The implementation of `wf_paren()` is missing six expressions, marked `____<X>____` in the code. For each letter `<X>` in the missing-expression blocks, please write that letter next to the expression that should go into the respective part of the code. (It may be that multiple letters associate with the same expression.)

```
84 void bb_prefix_sum(int *A, int n);
85
86 bool wf_paren(char *P_in, int n) {
87   // Returns true iff P_in is well formed.
88   // P_in is a string of parentheses (and no other characters).
89
90   int scratch[n];
91   int flags[n];
92
93   cilk_for (int i = 0; i < n; i++) {
94       if (P_in[i] == '(')
95             scratch[i] = ____A____;
96       else
97             scratch[i] = ____B____;
98   }
99
100  bb_prefix_sum(____C____, n);
101
102  cilk_for (int i = 0; i < n; i++) {
103      flags[i] = 0;
104      if (____D____ < 0) {
105          flags[i] = 1;
106      }
107  }
108
109  bb_prefix_sum(flags, n);
110
111  return flags[n-1] == ____E____ && scratch[n-1] == ____F____;
112 }
```

dyck path

1

-1

scratch

nice

scratch[i]

failed

0

(i.e. not like some big #

0 ↑ no bad loc

↑ ends ok

i

E, F 0

A 1

2

B -1

_____ scratch[i] ^ scratch[i-1]

C scratch

_____ scratch + n - 1

D scratch[i]

_____ flags[i] + flags[i+1]

_____ P_in

_____ scratch[i] & 1

_____ *scratch

_____ scratch[n-1]

_____ flags

_____ flags[i] ^ flags[i+1]

_____ P_in[i] == ')'

Assume that the work of the black-box prefix-sum algorithm on an input of size n is $W(n)$ and that the span is $S(n)$.

3.4 Express the asymptotic work of `wf_paren()` in simple terms.

$\Theta(n) + W(n) + \Theta(n) + W(n) + \Theta(1)$

load dyck path — prefix sum dyck path — load flags — prefix sum flags — return check

$\Theta(n) + W(n)$

It does a dyck path check. Dyck paths are paths of length $2n$ of 1 \$ -1 \$ s.t. at each point $\Sigma \geq 0$ \$ total $\Sigma = 0$. It's bijective to valid parens.

3.5 Express the asymptotic span of `wf_paren()` in simple terms.

$\Theta(\log(n)) + S(n) + \Theta(\log(n)) + S(n) + \Theta(1)$

same identities →

$\Theta(\log(n)) + S(n)$