Quiz 1

Algorithms, Data Structures, Performance Analysis, and Parallelism

Given: In class on Thursday, October 10, 2013

Name:__________________________

CSAIL username:______________

Instructions

• DO NOT open this quiz booklet until you are instructed to do so.

• This quiz booklet contains 19 pages, including this one. You have 80 minutes to earn 100 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 CSAIL username on this coversheet, and write your name 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 are explicitly asked to do so. For these kinds of questions, incorrect answers will be penalized, and so do not guess unless you are reasonably sure.

• Good luck!

<table>
<thead>
<tr>
<th>Question</th>
<th>Parts</th>
<th>Points</th>
<th>Score</th>
<th>Grader</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 Name on Every Page</td>
<td>19</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 True or False</td>
<td>5</td>
<td>10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2 Short Answers</td>
<td>4</td>
<td>16</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3 Loops Gone Wild</td>
<td>7</td>
<td>21</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4 Bit Hacks</td>
<td>5</td>
<td>15</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5 Computation Dags</td>
<td>3</td>
<td>9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6 Parallelism</td>
<td>5</td>
<td>20</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7 Greedy Scheduling</td>
<td>2</td>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td></td>
<td>100</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
6.172 Quiz 1
Name:

1 True or False (5 parts, 10 points)

Indicate whether each statement below is true or false by circling the correct answer. Remember, you need not explain your answers, but since incorrect answers will be penalized, do not guess unless you are reasonably sure.

1.1
Loop unrolling typically results in increased instruction-level parallelism (i.e., the number of instructions that can issue in parallel).

True  False

1.2
Function pointers may hinder the compiler’s ability to build a call graph.

True  False

1.3
Out-of-order processors discard all in-flight instructions upon a cache miss.

True  False
1.4

Consider the following code:

```c
int phooey(double a, double b, double c) {
    if (((a + b) + c < a + (b + c)) {
        return -1;
    } else if (((a + b) + c == a + (b + c)) {
        return 0;
    } else {
        return 1;
    }
}
```

For each of the three possible return values $-1$, $0$, or $+1$, there exist valid inputs to the `phooey` function which cause that value to be returned.

**True**   **False**

1.5

Suppose that the running time of a recursive program satisfies the recurrence

$$T(n) = 2T(n/4) + \Theta(n \log n),$$

where $T(n) = \Theta(1)$ for sufficiently small $n$. Coarsening the base case of the recursion will result in a significant speedup.

**True**   **False**
2 Short Answers (4 parts, 16 points)

Give short answers to the following questions. For multiple-choice problems, circle the letter of
the solution that answers the question. Remember again, since wrong answers will be penalized, do not guess unless you are reasonably sure. (This is your last warning.)

2.1

Why is branch prediction more important to performance on a processor with many pipeline stages than to a processor with few pipeline stages?

2.2

Which x86 architectural features benefit from register renaming by the processor hardware? (Circle all that apply.)

A Branch prediction
B SIMD (single-instruction multiple-data) execution
C Memory prefetching
D Out-of-order execution
E None of the above
2.3

Ben Bitdiddle wants to optimize the following piece of working code:

```c
/* The elements of A and B are known to be nonnegative and not aliased */
/* k is known only at runtime */
static const int N = (1 << 3/zero.noslash);
for (int i = /zero.noslash; i < N; i++) {
    A[i] = k * B[i];
}
```

Using what he learned in this class, he rewrites the code as:

```c
/* The elements of A and B are known to be nonnegative and not aliased */
/* k is known only at runtime */
static const int N = (1 << 3);
A[N - 1] = -1;
int *a = A, *b = B;
while (*a >= /zero.noslash) {
    *(a ++) = k * *(b ++);
}
*a = k * *b;
```

Ben compiles both codes with GCC using optimization -O3. When he runs the two codes, however, Ben finds that his new code is several times slower than his old code. Why?

A  Pointer accesses are slower than array accesses.
B  There are too many cache misses.
C  There are too many branch mispredictions.
D  The compiler can no longer vectorize the loop due to the data-dependent while condition.
E  None of the above.
2.4

The x86 assembly code below is for a function int func(int n), which takes a positive integer n as input. (Hint: Recall that the x86 calling convention uses %rdi/%edi for passing the first argument and %rax/%eax for the return value.)

```assembly
func:  pushq %rbx
     leal -1(%rdi), %ebx ; store rdi-1 into ebx
     movl %ebx, %eax
     cmpl $1, %ebx
     jle .L2
     movl %ebx, %edi
     call func
     addl %ebx, %eax
.L2:  popq %rbx
     ret
```

What does func(n) compute in terms of n?
3 Loops Gone Wild (7 parts, 21 points)

3.1

Consider the following three loops, where a, b, c, and d are length-N arrays of values:

```c
void frob1(int * restrict a, int * restrict b, 
            int * restrict c, int * restrict d, 
            size_t N) {
    for (size_t i = 0; i < N; i++)
        c[i] = a[i] + b[i];
    for (size_t i = 0; i < N; i++)
        d[i] = a[i] * b[i];
}

void frob2(int * restrict a, int * restrict b, 
            int * restrict c, int * restrict d, 
            size_t N) {
    for (size_t i = 0; i < N; i++) {
        c[i] = a[i] + b[i];
        d[i] = a[i] * b[i];
    }
}

typedef struct {
    int val1;
    int val2;
} int_pair ;

void frob3(int_pair * restrict a_b, int_pair * restrict c_d, 
            size_t N) {
    for (size_t i = 0; i < N; i++) {
        c_d[i].val1 = a_b[i].val1 + a_b[i].val2;
        c_d[i].val2 = a_b[i].val1 * a_b[i].val2;
    }
}
```

Suppose that $N = 2^{30}$. What is the order of the performance of frob1, frob2, and frob3 on the cloud machines (i.e., fastest > middle > slowest)?

A frob1 > frob2 > frob3 
B frob1 > frob3 > frob2 
C frob2 > frob3 > frob1 
D frob2 > frob1 > frob3 
E frob3 > frob1 > frob2 
F frob3 > frob2 > frob1
3.2

Consider the following three loops, where \( a, b, c, \) and \( d \) are length-\( N \) arrays of values and \( \text{shuffle} \) is a length-\( N \) array of indexes containing a random permutation (e.g., \( \text{shuffle}([0,1,2,3]) \to [2,3,0,1]) \):

```c
void borf1(int * restrict a, int * restrict b,
           int * restrict c, int * restrict d,
           size_t * restrict shuffle, size_t N) {
    for (size_t i = 0; i < N; i++)
        c[i] = a[shuffle[i]] + b[shuffle[i]];
    for (size_t i = 0; i < N; i++)
        d[i] = a[shuffle[i]] * b[shuffle[i]];
}
```

```c
void borf2(int * restrict a, int * restrict b,
           int * restrict c, int * restrict d,
           size_t * restrict shuffle, size_t N) {
    for (size_t i = 0; i < N; i++) {
        c[i] = a[shuffle[i]] + b[shuffle[i]];
        d[i] = a[shuffle[i]] * b[shuffle[i]];
    }
}
```

```c
typedef struct {
    int val1;
    int val2;
} int_pair;

void borf3(int_pair * restrict a_b, int_pair * restrict c_d,
           size_t * restrict shuffle, size_t N) {
    for (size_t i = 0; i < N; i++) {
        c_d[i].val1 = a_b[shuffle[i]].val1 + a_b[shuffle[i]].val2;
        c_d[i].val2 = a_b[shuffle[i]].val1 * a_b[shuffle[i]].val2;
    }
}
```

Suppose that \( N = 2^{30} \). What is the order of the performance of \( \text{borf1} \), \( \text{borf2} \) and \( \text{borf3} \) on the cloud machines (i.e., fastest > middle > slowest)?

- A \( \text{borf1} > \text{borf3} > \text{borf2} \)
- B \( \text{borf1} > \text{borf2} > \text{borf3} \)
- C \( \text{borf2} > \text{borf1} > \text{borf3} \)
- D \( \text{borf2} > \text{borf3} > \text{borf1} \)
- E \( \text{borf3} > \text{borf1} > \text{borf2} \)
- F \( \text{borf3} > \text{borf2} > \text{borf1} \)
3.3

Given a matrix $M$ in row-major format, we wish to compute the row and column sums (i.e., the sums across the rows and columns).

```c
#define ROWS 2048
#define COLS 3072

void compute_row_sums(int M[ROWS][COLS], int row_sums[ROWS]) {
    for (int i = 0; i < ROWS; i++) {
        row_sums[i] = 0;
        for (int j = 0; j < COLS; j++) {
            row_sums[i] += M[i][j];
        }
    }
}

void compute_column_sums(int M[ROWS][COLS], int column_sums[COLS]) {
    for (int i = 0; i < COLS; i++) {
        column_sums[i] = 0;
        for (int j = 0; j < ROWS; j++) {
            column_sums[i] += M[j][i];
        }
    }
}

void do_stuff(int M[ROWS][COLS]) {
    int row_sums[ROWS];
    int column_sums[COLS];
    compute_row_sums(M, row_sums);
    compute_column_sums(M, column_sums);
    // Does something with the row and column sums.
    ...
}
```

Perform a back-of-the-envelope calculation for the work of a serial execution of `do_stuff` on a $2048 \times 3072$ array. Assume that you have a 2GHz scalar processor (no vector units) which executes one instruction per cycle and that the omitted portion of the function takes negligible time. How long does `do_stuff` take?

A 0.00001–0.001 seconds.
B 0.001–0.1 seconds.
C 0.1–10 seconds.
D 10–1000 seconds.
E None of the above.
3.4
In the code on page 9, the two function calls to compute_row_sums() and compute_column_sums() in lines 26 and 27, respectively, can be spawned off in parallel using cilk_spawn, a cilk_sync can be inserted in line 28, and the code would remain correct.

True  False

3.5
In the code on page 9, we can parallelize the outer loop of compute_row_sums by replacing the for keyword on line 5 with a cilk_for keyword, and the code would remain correct.

True  False

If True: What are the work, span, and parallelism of compute_row_sums() in terms of ROWS and COLS in the modified code?
If False: Explain briefly why this outer for loop cannot be parallelized.
3.6

In the code on page 9, we can parallelize the inner loop of compute_row_sums by replacing the for keyword on line 7 with a cilk_for keyword, and the code would remain correct.

True  False

If True: What are the work, span, and parallelism of compute_row_sums() in terms of ROWS and COLS in the modified code?
If False: Explain briefly why this inner for loop cannot be parallelized.

3.7

The code in compute_row_sums() can be transformed to use vector hardware effectively.

True  False

If True: Explain briefly where and how to vectorize compute_row_sums()?
If False: Explain briefly why the function cannot be vectorized.
4 Bithacks (5 parts, 15 points)

What do each of the following codes do?

4.1

```c
uint64_t bithack_1(uint64_t x) {
    return (x & (x-1)) == 0;
}
```

A Returns the index of the lowest 1-bit in x.
B Returns a 1 in the position of the least-significant 1 in x and a 0 in all other bit positions.
C Returns 1 if x is a power of 2, and 0 otherwise.
D Returns 1 if x is 0 or a power of 2, and 0 otherwise.
E Returns 1 if x is greater than 1, and 0 otherwise.
F None of the above.

4.2

```c
uint64_t bithack_2(uint64_t x) {
    if (x)
        return 1 + bithack_2(x & (x-1));
    else
        return 0;
}
```

A Returns $2^x$.
B Returns $\lceil \log x \rceil$.
C Returns $\lfloor \log x \rfloor$.
D Returns the number of 1’s in x.
E Returns $2^\lceil \log x \rceil$.
F Returns the maximum number of consecutive 1’s in x.
G None of the above.

4.3

```c
uint64_t bithack_3(uint64_t x) {
    return x & -x;
}
```

A Returns a 1 in the position of the least-significant 1 in x and 0 in all other bit positions.
B Returns the absolute value of x.
C Returns $+1$ if $x > 0$, and $-1$ otherwise.
D Returns the index of the least-significant bit in x.
E None of the above.
4.4

```c
uint64_t bithack_4(uint64_t x) {
    x ^= (x >> 32);
    x ^= (x >> 16);
    x ^= (x >> 8);
    x ^= (x >> 4);
    x ^= (x >> 2);
    x ^= (x >> 1);
    return x & 1;
}
```

A. Returns 1 if \(x\) is a power of 2, and 0 otherwise.
B. Returns 0 if \(x\) has a 1 at every power-of-2 bit index, and 1 otherwise.
C. Returns 1 if \(x\) has an odd number of 1's, and 0 otherwise.
D. Returns 0.
E. None of the above.

4.5

```c
uint64_t bithack_5(uint64_t x, uint8_t a) {
    uint64_t y = (0x0101010101010101 * ((uint64_t) a)) ^ x;
    y |= (y >> 4);
    y |= (y >> 2);
    y |= (y >> 1);
    y &= (y >> 32);
    y &= (y >> 16);
    y &= (y >> 8);
    return y & 1;
}
```

A. Returns 1 if the product \(a \cdot x\) has a 1 in its least-significant bit, and 0 otherwise.
B. Returns 1 if the number of aligned bytes in \(x\) that are equal to \(a\) is odd.
C. Returns 0 if \(x\) contains an aligned byte equal to \(a\), and 1 otherwise.
D. Returns 0 if \(x\) is evenly divisible by \(a\), and 1 otherwise.
E. Returns 1 if \(a\) evenly divides any aligned byte in \(x\), and 0 otherwise.
F. None of the above.
5  **Computation Dags (3 parts, 9 points)**

Consider the following dag representing a multithreaded computation, where each circle denotes a serially executing strand that takes unit time to execute:

For each of the problems below, please provide a numerical answer. **Please put a box around your answer.**

5.1  
What is the work of this computation?

5.2  
What is the span of this computation?

5.3  
What is the parallelism of this computation?
6 Parallelism (5 parts, 20 points)

6.1

Five students have implemented recursive Fibonacci programs, where the base case of each program returns 1 if the program input is $n = 0$ or $n = 1$. For $n > 1$, the various students calculate Fibonacci using the code snippets for the recursive cases shown below:

a: 
```
x = fib(n-1);
y = fib(n-2);
```

b: 
```
x = cilk_spawn fib(n-1);
y = cilk_spawn fib(n-2);
cilk_sync;
```

c: 
```
x = fib(n-1);
y = cilk_spawn fib(n-2);
cilk_sync;
```

d: 
```
y = cilk_spawn fib(n-2);
x = fib(n-1);
cilk_sync;
```

e: 
```
x = cilk_spawn fib(n-1);
y = fib(n-2);
cilk_sync;
```

Assume that the overhead of spawning a function is about 10 times the cost of an ordinary function call. Rank these codes in order of the performance you would expect on a 32-core machine (e.g., fastest > second fastest > ⋯ > slowest):

> > > > >

------- ------- ------- ------- ------- -------
The following Cilk function implements a dot product of two length-n vectors in parallel:

```c
double dot_product(double * A, double * B, size_t n) {
    if (n == 1) {
        return *A * *B;
    } else {
        size_t mid = n / 2 ;
        double p1 = cilk_spawn dot_product(A, B, mid);
        double p2 = dot_product(A + mid, B + mid, n - mid);
        cilk_sync;
        return p1 + p2;
    }
}
```

6.2
Write down a recurrence for the work of the `dot_product` function, and argue using the Master Theorem that the work is \( \Theta(n) \).

6.3
Write down a recurrence for the span of the `dot_product` function, and argue using the Master Theorem that the span is \( \Theta(\log n) \).
6.4

A matrix $L$ is \textit{lower triangular} if $L[i][j] = 0$ for $i < j$. The nonzeros of a lower-triangular matrix can be stored compactly in a one-dimensional array by storing row $i$, which contains $i+1$ nonzeros, starting in location $i(i+1)/2$. The following function uses the \texttt{dot_product} code on page 16 to compute the matrix-vector product of a lower triangular matrix $L$ of size $n \times n$ and a vector $X$ of size $n$:

```c
void lt_matrix_vector_product(double *L, double *X,
                              double *B, size_t n) {
    cilk_for (size_t i = 0; i < n; i++) {
        size_t offset = (i * (i + 1)) / 2;
        B[i] = dot_product(L + offset, X, i + 1);
    }
}
```

For the purposes of analysis, assume that the grain size for the \texttt{cilk_for} is 1. What is the parallelism of the \texttt{lt_matrix_vector_product} function?

A $\Theta(n/\lg^2 n)$.
B $\Theta(n)$.
C $\Theta(n^2/\lg^2 n)$.
D $\Theta(n^2/\lg n)$.
E $\Theta(n^2)$.
F None of the above.
6.5

Now consider a different way of computing the matrix-vector product for a lower-triangular matrix:

```c
void new_lt_matrix_vector_product(double *L, double *X, double *B, size_t n) {
    for (size_t i = 0; i < n; i++) {
        size_t offset = (i * (i + 1)) / 2;
        B[i] = cilk_spawn dot_product(L + offset, X, i + 1);
    }
    cilk_sync;
}
```

What is the parallelism of the `new_lt_matrix_vector_product` function?

A. \( \Theta(n/\lg n) \).
B. \( \Theta(n) \).
C. \( \Theta(n^2/\lg^2 n) \).
D. \( \Theta(n/\lg n) \).
E. \( \Theta(n^2) \).
F. None of the above.
7 Greedy Scheduling (2 parts, 8 points)

Ben Bitdiddle measures his deterministic program on 4 and 64 processors of an ideal parallel computer using a greedy scheduler. He obtains running times of $T_4 = 32$ seconds and $T_{64} = 4$ seconds. In lecture, we proved that any greedy scheduler schedules a computation with work $T_1$ and span $T_\infty$ in time $T_P \leq T_1/P + T_\infty$ on a $P$-processor ideal parallel computer. We also showed that $T_P \geq T_1/P$ (Work Law) and $T_P \geq T_\infty$ (Span Law). Based on these formulas, answer the following questions by circling the letter corresponding to the correct answer. You need not explain your answers.

7.1
What is the minimum possible value for the parallelism of the program?

A 24
B 28
C 32
D 40
E 64
F None of the above.

7.2
What is the maximum possible value for the parallelism of the program?

A 28
B 32
C 40
D 64
E 80
F None of the above.