6.172 Quiz

Walker Memorial, 7:30 P.M. – 9:30 P.M. on Tuesday, November 6, 2012

Name: ________________________________

CSAIL username: __________________________

Instructions

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

• This quiz booklet contains 23 pages. You have 120 minutes to earn 120 points.

• This quiz is closed book, but you may use one handwritten, double-sided 8 1/2″ × 11″ crib sheet and a Master Theorem card.

• When the quiz begins, please write your name and username on this coversheet, and write your name on the top of each page, since the pages will be separated for grading.

• If you run out of space for answering a question, use the back side of the same sheet and make a note on the front that you are using the back. Do not use a different sheet for answering a question, since the pages will be separated for grading.

• There will be a 5-minute stretch break about 60% of the way through the exam. No one will be allowed to work on the exam during this short break.

• 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>Name on every page</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Short answer</td>
<td>4</td>
<td>16</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Coding</td>
<td>4</td>
<td>18</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Parallelism</td>
<td>3</td>
<td>14</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Eliminating race bugs</td>
<td>7</td>
<td>28</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Strassen's algorithm</td>
<td>8</td>
<td>43</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td></td>
<td><strong>120</strong></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
1 Short answer (4 parts) [16 points]

For each of the questions below, provide a short answer of at most one or two sentences.

1.1 [4 points]

Explain why semiconductor vendors began circa 2003 to produce multicore chips.

1.2 [4 points]

What invariant on cache states is maintained by an MSI cache-coherence protocol?
1.3 [4 points]
Suppose that the work of a recursive program satisfies the recurrence $T(n) = 4T(n/2) + \Theta(n^3)$. Will coarsening the recursion significantly improve the constants in the running time? Explain why or why not.

1.4 [4 points]
Explain how a debugger, such as gdb, can be used to determine which functions to optimize in a long-running program.
2 Coding (4 parts) [18 points]

Provide an answer for each of the questions below.

2.1 [4 points]

Consider the following funky function:

```c
void funky (double *A, double *B, double c, size_t n) {
    for (size_t i = 0; i < n; ++i) {
        A[i] += c * B[i];
    }
}
```

What might be done to enhance the ability of the compiler to vectorize this code?

2.2 [4 points]

What does the silly function below do?

```c
int silly (int x) {
    int test = (x < 0);
    return (x ^ -test) + test;
}
```

A Returns \(-x\).
B Returns \(x\) with the sign bit flipped.
C Returns the absolute value of \(x\).
D Returns \(-(x+1)\).
E Other: ____________________________
2.3  [4 points]

Suppose that the `pop_count(int v)` function returns the *population count* of `v`, which is the number of 1 bits in the binary representation of `v`. Given a nonzero value `n`, what does the following `giggly` function do?

```c
int giggly(int n) {
    int x = n & (-n);
    return pop_count(x-1);
}
```

A  Returns the index of the most-significant 0 bit in `n`.
B  Returns the index of the least-significant 1 bit in `n`.
C  Returns the number of 0 bits in `n`.
D  Returns the number of 1 bits in `n`.
E  Other: ____________________________

2.4  [6 points]

The code snippet below shows the x86 assembly code for a function `wacky` with the function signature `int wacky(int n)`, which takes as input a positive integer `n`. What does the `wacky` function below do? (Hint: Recall that the x86-64 calling convention uses `%rdi` for passing 1st argument and `%rax` for return value. The instruction `imull` performs integer multiplication.)

```
.globl wacky
.type wacky, @function
wacky:
    pushq %rbp
    movq %rsp, %rbp
    pushq %rbx
    subq $40, %rsp
    movl %edi, -36(%rbp)
    cmpl $1, -36(%rbp)
    jg .L2
    movl -36(%rbp), %eax
    jmp .L3
.L2:
    movl -36(%rbp), %eax
    subl $1, %eax
    movl %eax, %ebx
    movl %eax, %edi
    call wacky
    imull %ebx, %eax
    movl %eax, -20(%rbp)
    movl -20(%rbp), %eax
.L3:
    addq $40, %rsp
    popq %rbx
    leave
    ret
```
3 Parallelism (3 parts) [14 points]

Answer the questions below.

3.1 [4 points]
What is the parallelism of the following code for computing \( \text{fib}(n) \)?

```c
uint64_t fib(uint64_t n) {
  if (n < 2) return n;
  uint64_t x = fib(n - 1);
  uint64_t y = cilk_spawn fib(n - 2);
  cilk_sync;
  return x + y;
}
```

3.2 [3 points]
Give a one-word reason why a deterministic parallel program should generally be preferred over a nondeterministic parallel program.
3.3  [7 points]

A greedy scheduler satisfies the following stronger bound than what we have shown in the class:

\[ T_P \leq \frac{T_1 - T_\infty}{P} + T_\infty. \]

Suppose that a deterministic program is run on an ideal computer that uses a greedy scheduler, and suppose that the program’s runtime is measured when using 4, 10, and 64 processors. With this stronger bound, argue that it is impossible to observe the following running times: \( T_4 = 80 \) seconds, \( T_{10} = 42 \) seconds, and \( T_{64} = 10 \) seconds. (Hint: Apply the Work Law for \( P = 4 \) and the Span Law for \( P = 64 \)).
4 Eliminating race bugs (7 parts) [28 points]

Consider the code snippet below, which performs a computation on an array, where the values in the array are updated as a function of their own values and that of their neighbors:

```c
// global variable indicating whether the computation has converged
bool converged;
// the value is converged when it changes less than the error margin
const double error_margin = 0.001;

extern void init_val(double *V, uint N);
extern double compute(double *mine, double *left, double *right);

int mod(int x, uint n) {
  if (x < 0) x += n;
  return (x % n);
}

void update(double *V, uint N, int i) {
  double my_val = V[i];
  double left_val = V[mod(i-1,N)];
  double right_val = V[mod(i+1,N)];
  double res = compute(my_val, left_val, right_val);
  V[i] += res * 0.5;
  V[mod(i-1,N)] += res * 0.25;
  V[mod(i+1,N)] += res * 0.25;

  double change = fabs(V[i] - my_val) +
                 fabs(V[mod(i-1,N)] - left_val) +
                 fabs(V[mod(i+1,N)] - right_val);
  if (change > error_margin) {
    converged = false;
  }
}

void update_all(double *V, uint N) {
  cilk_for (int i=0; i < N; i++) {
    update(V, N, i); // update V[i]
  }
}

int main(int argc, char *argv[]) {
  ...
  // an array of values to update
  double *V = (double *) malloc( sizeof(double) * N );
  init_val(V, N);
  do {
    converged = true;
    update_all(V, N);
  } while (converged == false);
  ...
  free(V);
  return 0;
}
```
The main function initializes a size-N array of doubles and invokes update_all with the array until all values in the array converge. The function update_all iterates through the array and invokes update for each of the array elements. Within the update function, a value res is computed in line 18 based on \( V[i] \), \( V[\text{mod}(i-1,N)] \), and \( V[\text{mod}(i+1,N)] \). The code in lines 20–22 then updates new values for these array elements based on res.

After updating these elements, the code checks in lines 25–29 whether the new values have changed by too much. If so, the computation has not converged, and the code sets the global converged flag to false. The main routine sets the converged flag to true before each call to update_all in line 46. It then checks for global convergence after each round of updates, invoking update_all repeatedly until every array element in \( V \) converges.

For this relaxation problem, the order in which the array elements are updated does not matter. The update_all function simply performs the update of array elements in parallel using a cilk_for loop. This parallelization contains two determinacy races, however. In this problem, we ask you to examine each of these races more closely and explore ways to mitigate them.

Part I

In Part I, we shall examine the first race, and we ask you to ignore the second race for the time being. That is, pretend that the second race does not exist when answering questions in Part I.

4.1 [4 points]
Explain why the race on the global variable converged is benign and does not affect the correctness of the program.

4.2 [4 points]
Although the race on converged does not adversely affect the correctness of the program, it may have a negative impact on performance due to true sharing. Explain how true sharing might cause poor performance in this instance.
4.3 [4 points]

Suppose that the converged flag is replaced with a reducer of type `cilk::reducer< cilk::op_and<bool>>`, which eliminates the benign race. Explain why this solution should also fix the true-sharing performance bug.

As an alternative to using a reducer, the true-sharing performance bug can be addressed by replacing lines 27–29 with the following code:

```cpp
if (change > error_margin && converged == true) {
    converged = false;
}
```

In other words, in the update function, if the new values have changed by more than `error_margin`, instead of always setting the converged flag to false, the code first checks the value of the converged flag and sets it to false only if it is not set to false already.

4.4 [4 points]

On a computer with `P` processors, what is the maximum number of times in terms of `n` and `P` that the converged flag might now be written during one invocation of the update_all function? Explain how this optimization mitigates the true-sharing performance bug.
Part II

Unlike the first determinacy race, the second determinacy race, which involves the updating of the array elements, is not benign. It may potentially cause incorrect program behavior. Two logically parallel strands may access $V[i]$ in parallel, with one of the accesses being a write (i.e., the updating of $V[i]$ and the reading of $V[i]$ for updating $V[\text{mod}(i-1,N)]$ and $V[\text{mod}(i+1,N)]$).

One possible way to fix the bug caused by this determinacy race is to use fine-grained locking. Suppose that the main routine also allocates an array of `tbb::mutex`, and we modify the `update` function as follows:

```cpp
void locking_update ( double *V, int N, int i, tbb::mutex *locks ) {
    locks[\text{mod}(i-1,N)].lock();
    locks[i].lock();
    locks[\text{mod}(i+1,N)].lock();

    double my_val = V[i];
    double left_val = V[\text{mod}(i-1,N)];
    double right_val = V[\text{mod}(i+1,N)];
    double res = compute(my_val, left_val, right_val);

    V[i] += res * 0.5;
    V[\text{mod}(i-1,N)] += res * 0.25;
    V[\text{mod}(i+1,N)] += res * 0.25;

    // fabs -- returns floating-point absolute value
    double change = fabs(V[i] - my_val) +
                    fabs(V[\text{mod}(i-1,N)] - left_val) + fabs(V[\text{mod}(i+1,N)] - right_val);

    locks[\text{mod}(i+1,N)].unlock();
    locks[i].unlock();
    locks[\text{mod}(i-1,N)].unlock();

    if (change > error_margin) {
        converged = false;
    }
}
```

4.5 [4 points]

Does replacing the call to `update` in line 34 with `locking_update` result in correct code? Explain why or why not.
Let us return to the original code and explore chromatic scheduling. Recall that the idea is to color a conflict graph and process all nodes having the same color in parallel. Chromatic scheduling replaces the function \texttt{update\_all} (lines 32–36) with the following function \texttt{chromatic\_update\_all}:

```c
void chromatic_update_all(double *V, int N) {
    for (int color = 1; color <= num_colors; color++) {
        cilk_for (int i=0; i < N; i+= num_colors) {
            update (V, N, i); // update V[i]
        }
    }
}
```

4.6 [4 points]

Draw the conflict graph for the case when \(N=7\). How many colors are required to color this conflict graph? Color the conflict graph with \texttt{num\_color} colors by labeling each node with an integer from 1 to \texttt{num\_colors} representing its color.

4.7 [4 points]

Continue to assume that the computation uses the \texttt{update} function, but also assume now that \(N\) is evenly divisible by 3. How many colors are required to color the conflict graph? Explain.
5 Strassen’s algorithm (8 parts) [43 points]

Strassen’s algorithm (Strassen, 1969) is a divide-and-conquer algorithm for multiplying square matrices that is asymptotically more efficient than the divide-and-conquer algorithm covered in Lecture 1 and revisited during the term. Assume for convenience that \( n \) is an exact power of 2. Unlike the standard divide-and-conquer algorithm, which performs 8 multiplications of \( n/2 \times n/2 \) submatrices, Strassen’s algorithm performs 7 recursive multiplications of \( n/2 \times n/2 \) submatrices, at the cost of more additions and subtractions of submatrices. Since matrix multiplications are asymptotically more expensive than matrix additions, however, this trade-off results in a better algorithm asymptotically.

We’ll explain Strassen’s algorithm below, and the pseudocode for Strassen’s algorithm is given on the following page. Do not waste time trying to actually understand the algorithm or the pseudocode! You only need to understand what it does, not why it works. The code assumes that the input matrices are stored in row-major order, and the code takes parameters indicating the number of elements per row of each matrix so that submatrices can be properly indexed.

To perform the matrix multiplication \( C = A \cdot B \), Strassen’s algorithm starts by dividing the three matrices into \( n/2 \times n/2 \) submatrices as follows:

\[
C = \begin{pmatrix}
C_{11} & C_{12} \\
C_{21} & C_{22}
\end{pmatrix}, \quad A = \begin{pmatrix}
A_{11} & A_{12} \\
A_{21} & A_{22}
\end{pmatrix}, \quad B = \begin{pmatrix}
B_{11} & B_{12} \\
B_{21} & B_{22}
\end{pmatrix}.
\]

It then computes the following \( n/2 \times n/2 \) submatrix sums and differences:

\[
S_1 = B_{12} - B_{22}, \quad S_6 = B_{11} + B_{22}, \quad S_2 = A_{11} + A_{12}, \quad S_7 = A_{12} - A_{22}, \quad S_3 = A_{21} + A_{22}, \\
S_8 = B_{21} + B_{22}, \quad S_4 = B_{21} - B_{11}, \quad S_9 = A_{11} - A_{21}, \quad S_5 = A_{11} + A_{22}, \quad S_{10} = B_{11} + B_{12}.
\]

Next, the algorithm forms the following \( n/2 \times n/2 \) submatrix products recursively:

\[
P_1 = A_{11} \cdot S_1, \quad P_2 = S_2 \cdot B_{22}, \quad P_3 = S_3 \cdot B_{11}, \quad P_4 = A_{22} \cdot S_4, \\
P_5 = S_5 \cdot S_6, \quad P_6 = S_7 \cdot S_8, \quad P_7 = S_9 \cdot S_{10}.
\]

Finally, it computes the final answer with the following submatrix sums and differences:

\[
C_{11} = P_5 + P_4 - P_2 + P_6, \quad C_{12} = P_1 + P_2, \quad C_{21} = P_3 + P_4, \quad C_{22} = P_5 + P_1 - P_3 - P_7.
\]

Assume that the base case (lines 9–12) takes constant time, and so do the index calculations to compute pointers to submatrices (lines 13–16) and space allocation for temporary matrices (lines 17–20).
void strassen(int n, double * C, int rw_c, double * A, int rw_a, double * B, int rw_b) {
    if (n == 1) { // base case
        ELEM(C,0,0,rw_c) = ELEM(A,0,0,rw_a) * ELEM(B,0,0,rw_b);
        return;
    }
    int half_n = n / 2;
    double *A11 = A, *A12 = &ELEM(A,0,half_n,rw_a);
    double *A21 = &ELEM(A,half_n,0,rw_a), *A22 = &ELEM(A,half_n,0,rw_a);
    ... // compute B11, B12, B21, B22, C11, C12, C21, and C22 similarly
    double *S1,*S2,*S3,*S4,*S5,*S6,*S7,*S8,*S9,*S10;
    double *P1,*P2,*P3,*P4,*P5,*P6,*P7; // n/2 x n/2 temp matrices
    S1 = (double *) malloc( sizeof(double) * half_n * half_n);
    ... // allocate space for all temp matrices
    cilk_for(int i=0; i < half_n; i++) { // compute S1...S10
        cilk_for(int j=0; j < half_n; j++) {
            ELEM(S1,i,j,half_n) = ELEM(B12,i,j,rw_b) - ELEM(B22,i,j,rw_b);
            ELEM(S2,i,j,half_n) = ELEM(A11,i,j,rw_a) + ELEM(A12,i,j,rw_a);
            ELEM(S3,i,j,half_n) = ELEM(A21,i,j,rw_a) + ELEM(A22,i,j,rw_a);
            ELEM(S4,i,j,half_n) = ELEM(B12,i,j,rw_b) - ELEM(B11,i,j,rw_b);
            ELEM(S5,i,j,half_n) = ELEM(A11,i,j,rw_a) + ELEM(A22,i,j,rw_a);
            ELEM(S6,i,j,half_n) = ELEM(B11,i,j,rw_b) + ELEM(B22,i,j,rw_b);
            ELEM(S7,i,j,half_n) = ELEM(A12,i,j,rw_a) - ELEM(A21,i,j,rw_a);
            ELEM(S8,i,j,half_n) = ELEM(B21,i,j,rw_b) + ELEM(B22,i,j,rw_b);
            ELEM(S9,i,j,half_n) = ELEM(A11,i,j,rw_a) - ELEM(A21,i,j,rw_a);
            ELEM(S10,i,j,half_n) = ELEM(B11,i,j,rw_b) + ELEM(B12,i,j,rw_b);
        }
    }
    cilk_spawn strassen(half_n, P1, half_n, A11, rw_a, S1, half_n);
    cilk_spawn strassen(half_n, P2, half_n, S2, half_n, B22, rw_b);
    cilk_spawn strassen(half_n, P3, half_n, S3, half_n, B11, rw_b);
    cilk_spawn strassen(half_n, P4, half_n, S4, half_n);
    cilk_spawn strassen(half_n, P5, half_n, S5, half_n, S6, half_n);
    cilk_spawn strassen(half_n, P6, half_n, S7, half_n, S8, half_n);
    cilk_spawn strassen(half_n, P7, half_n, S9, half_n, S10, half_n);
    cilk_sync;
    // now compute C11...C22 based on P1...P7
    cilk_for(int i=0; i < half_n; i++) {
        cilk_for(int j=0; j < half_n; j++) {
            ELEM(C11,i,j,half_n) = ELEM(P5,i,j,half_n) + ELEM(P4,i,j,half_n)
                - ELEM(P2,i,j,half_n) + ELEM(P6,i,j,half_n);
            ELEM(C12,i,j,half_n) = ELEM(P1,i,j,half_n) + ELEM(P2,i,j,half_n);
            ELEM(C21,i,j,half_n) = ELEM(P3,i,j,half_n) + ELEM(P4,i,j,half_n);
            ELEM(C22,i,j,half_n) = ELEM(P5,i,j,half_n) + ELEM(P1,i,j,half_n)
                - ELEM(P3,i,j,half_n) - ELEM(P7,i,j,half_n);
        }
    }
    ... // free the space for temp matrices
}
5.1 [4 points]

Explain briefly why the work $T_1(n)$ required by Strassen’s algorithm to multiply two matrices of size $n$ satisfies the recurrence

$$T_1(n) = 7T_1(n/2) + \Theta(n^2).$$
5.2 [4 points]

Below is a recursion tree for the recurrence

\[ T_1(n) = 7T_1(n/2) + n^2 , \]

where we’ve simplified the Strassen work recurrence by removing a \( \Theta \), which doesn’t asymptotically affect the answer. Label the height of the recursion tree and the number of leaves. Argue that the solution to the recurrence is \( T_1(n) = \Theta(n^\lg 7) \). (Recall that \( \lg x = \log_2 x \).)
5.3  [6 points]
Explain why the span $T_\infty(n)$ of Strassen’s algorithm satisfies the recurrence

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

Solve this the recurrence.

5.4  [4 points]
What is the parallelism of Strassen’s algorithm in terms of $n$?
Let $Q(n)$ be the number of cache misses incurred by Strassen’s algorithm in an ideal-cache model (fully associative with an optimal or LRU replacement policy, as appropriate) with cache size $M$ and cache-line length $B$, and assume that the cache is tall — $M = \Omega(B^2)$.

5.5 [4 points]

Argue that for $n \gg M$, the number $Q(n)$ of cache misses satisfies the recurrence

$$Q(n) = \begin{cases} \Theta(n^2/B) & \text{if } n^2 < cM, \text{ where } c \leq 1 \text{ is a sufficiently small constant,} \\ 7Q(n/2) + \Theta(n^2/B) & \text{otherwise.} \end{cases}$$

(Hint: For the base case, all the temporary matrices for a subproblem must fit into cache, but there are only a constant number of them.)
5.6  [8 points]

Draw a recursion tree for the recurrence

\[ Q(n) = \begin{cases} \Theta(n^2/B) & \text{if } n^2 < cM, \text{ where } c \leq 1 \text{ is a sufficiently small constant}, \\ \gamma Q(n/2) + n^2/B & \text{otherwise}; \end{cases} \]

labeling the internal nodes and leaves of the tree with the corresponding number of cache misses incurred at them. (Note that we’ve again dropped a \( \Theta \) that doesn’t affect the asymptotics.) Label the height of the recursion tree and the number of leaves. Solve the recurrence, providing a tight asymptotic bound in simple form.
5.7 [5 points]

Argue briefly that the space of a serial execution of Strassen’s algorithm to multiply \( n \times n \) matrices satisfies the recurrence

\[
S_1(n) = S_1(n/2) + \Theta(n^2) .
\]

Solve the recurrence for the space \( S_1 \).
Assume that the cactus stack used by the Cilk runtime system is heap allocated. The busy-leaves property of the Cilk runtime system guarantees that the space \( S_P \) used by Strassen’s algorithm to multiply \( n \times n \) matrices on a system with \( P \) processors satisfies \( S_P = O(Pn^2) \). We shall prove a tighter bound.

On the next page is an asymptotically worst-case recursion tree for the \( P \)-processor space of Strassen’s algorithm for multiplying \( n \times n \) matrices (where we’ve again removed a \( \Theta \) for simplicity). The tree branches 7 ways in parallel at each level until it reaches the first level \( k \) with \( P \) or more nodes, and then each node executes its subcomputation serially. For simplicity, let’s assume that \( P \) is an exact power of 7, and hence there are exactly \( P \) nodes on level \( k \).

5.8 [8 points]

What is the depth of level \( k \)? What is the total amount of space used by all the nodes on level \( k \) in terms of \( P \) and \( n \)? Argue that the total amount of space used by Strassen’s algorithm to multiply \( n \times n \) matrices when run on \( P \) processors is \( O(n^2P^{1-2/\lg 7}) \). (Hint: The algebraic manipulations may be easier if you avail yourself of the identity \( \log_7 P = (\lg P)/\lg 7 \).)