

CS 475 Fall 2011 Mock Midterm I Solution

Oct 4 2011

Problem 1: [20 pts]

Parallelize each of the following code fragments using OpenMP pragmas, or explain why it cannot be parallelized.

1a: [10 pts]

```
for (i=1; i<n; i++){
    for (j=20; j<m; j++){
        a[i][j] = foo(a[i-1][j], a[i][j-20]); // foo has no side-effects
    }
}
```

Solution: Technically speaking, this cannot be parallelized just by inserting a pragma [as someone pointed out in class] but needs to be restructured. However, here's the analysis needed to parallelize it.

Note that in this doubly nested loop, each iteration writes to a distinct location (whose value may be read by later iterations). The loop body performs two reads—from $a[i-1][j]$ and $a[i][j-20]$. Because of the first one, the outer loop cannot be parallelized, so our only candidate is the inner one. This is very similar to the problem in discussion, where (it may be useful if you draw this out) there is a chain of dependent computations: $a[i][j-20] \rightarrow a[i][j-20] \rightarrow a[i][j-40] \rightarrow \dots$. However, each adjacent point is part of a separate chain— $[i][j]$ is in a *different* chain than $[i][j-1]$. So let us block out the inner loop into exactly 20 blocks as follows

```

for (i=1; i<n; i++)
    for (jj=20; jj<40; jj++) /* assume m>20 */
        for (j=jj; j<m; j++) /* note that this chain starts at jj */
            a[i][j] = foo(a[i-1][j], a[i][j-20]); // foo has no side-effects

```

Now simply make the jj loop parallel with `omp parallel for` (and make sure that j is private)

1b: [10 pts]

```

for (i=0; i<n; i++){
    for (j=0; j<i; j++){
        x = a[i][j];
        a[i][j] = a[j][i];
        a[j][i] = x;
    }
}

```

Solution: This program computes the transpose of the matrix a and so each instant of the iteration is completely independent. Make the outer loop parallel with `omp parallel for` and make sure that j and x are private.

1c: Quinn claims that in a certain code fragment, the inner j loop can be executed in parallel, so the following (a is an $n \times n$ array, k is an integer, and tmp is a float variable, all appropriately initialized) is a legal parallelization.

```

for (i=0; i<m; i++){
    #pragma omp parallel for
    for (j=0; j<n; j++)
        a[i][j] = MIN(a[i][j], a[i][k]+tmp);
}

```

```
}
```

Explain why this is incorrect [10 pts].

Short answer: Since $0 \leq k < n$, $a[i][k]$ is itself updated (written into) by one of the iterations of the j loop. So all iterations of the loop read this value and one of them writes into it, and there is a data race.

Details: The loop that is parallelized iterates over the i -th row of the matrix, and updates (writes into) each element $a[i][j]$. In order to do this, it reads two values, the previous (i.e., before the loop started) value of $a[i][j]$ itself, and $a[i][k]$ for some k . We don't know the exact value of k , but all we know is that it is between 0 and $n - 1$, so $a[i][k]$ is itself updated by the loop. So, in the sequential program, some iterations of the j loop, namely those where $j < k$, read the value of $a[i][k]$ before this update, and some others.

Give the correct parallelization of the inner loop. [10 pts].

```
#pragma omp parallel
for (i=0; i<m; i++){
    #pragma omp for
    for (j=0; j<k; j++) /* all of these need a[i,k] before it is updated*/
        a[i][j] = MIN(a[i][j], a[i][k]+tmp);
    a[i][k] = MIN(a[i][k], a[i][k]+tmp); /* now update a[i][k]
    #pragma omp for
    for (j=k+1; j<n; j++) /* now read the updated a[i][k]
        a[i][j] = MIN(a[i][j], a[i][k]+tmp);
}
```

Despite the answer to the previous question, are there some conditions when Quinn is correct, or is it *never* legal to parallelize as given above? Justify. [10 pts].

Solution: Consider the loop body when $a[i][k]$ is updated. The two values that the program reads are **both** $a[i][k]$, and the expression is $\text{MIN}(a[i][k], a[i][k] + \text{tmp})$. Because of the semantics of MIN , whenever we can guarantee that tmp is non-negative, then this statement will leave $a[i][k]$ unchanged. So the data race, although it exists, will be benign, and the parallelization will be legal.

Problem 3. Collective Communications: [30+10 pts]

We have a P -processor machine where the processors are arranged in an “octopus-accountant” grid, i.e., each processor can communicate (simultaneously, and bidirectionally) to all four of its neighbors (the boundary processors may have fewer neighbors). The time for a single communication *does not* depend on the message volume, it is just λ , and the arithmetic cost is χ . Assume that the grid is square, $P = p^2$ and that p is even. There is a data packet of size b already distributed on each processor. We want to perform the following computation: Each processor performs an operation `foo` on its packet, producing an answer packet of size b . Then these packets are combined using a special reduction operator, `bar` that takes two packets of size b and produces an answer packet of size b . However, we want the final result to be available at all the processors (i.e., an AllReduce operation). On any processor, the time to do either one of `foo` or `bar` is $b^2\chi$.

A simple algorithm to solve the problem would be (1) perform `foo`; (2) do the reduce operation to processor $\langle 0, 0 \rangle$; (3) who then broadcasts to everyone. Describe the algorithm to do step 2. [5 pts].

Solution: The algorithm has two phases.

Phase I: Every processor participates in a “column-wise reduction:” it gets a partial answer from the processor above it, combines it with its own data using the `bar` operator, and sends the result to the processor below.

Phase II: All the processors in the bottom row participate in a “row-wise reduction:”

get a partial answer from the processor to its right, combine it with its own data (the answer from Phase I) using the bar operator, and send the result to the left.

Derive a formula for the execution times for each of the 3 steps. [2+4+4=10 pts].

- $T_1 = \chi b^2$. All processors apply foo on their data.
- This has two parts. Phase I has $p - 1$ steps and each one takes $\lambda + \chi b^2$ (apply bar and send the result on). Phase II has an identical cost. So, $T_2 = 2(p - 1)(\lambda + \chi b^2)$.
- The third step, broadcast, is similar to the reduction except that there is no need to apply bar at each step, so the χb^2 term can be dropped from the formula above, so $T_3 = 2(p - 1)\lambda$.

Putting all this together and simplifying, the total time is.

$$T = 2(p - 1)(\lambda + \chi b^2) + 2(p - 1)\lambda + \chi b^2 = (4p - 2)\lambda + (2p - 1)b^2\chi \approx 4p\lambda + 2pb^2\chi$$

To improve this algorithm, we choose a different processor $\langle a, b \rangle$ as the destination of the reduction in the step 2. Explain how each step changes and re-derive the formula for the total execution time (it will be a function of a and b). [10 pts].

Solution: In the previous solution, there was nothing special about the processor $\langle 0, 0 \rangle$, we could have used any corner of the grid as the root of the reduce-broadcast tree. So consider the four neighboring processors, $\langle a, b \rangle$, $\langle a + 1, b \rangle$, $\langle a, b + 1 \rangle$, and $\langle a + 1, b + 1 \rangle$. Use them to divide the grid into four quadrants. Do the previous algorithm but use these processors as the root. However, after the reduction step (Step 2), these four combine their answers together (the original algorithm on a 2×2 grid), and this is the value broadcast in step 3.

The time (ignoring for now, the middle step on the 2×2 grid) for each quadrant is the same as above, but the term $2p$, wherever it occurs is replaced by the “half-perimeter” of each quadrant—either $a + b$, $a + p - b$, $p - a + b$, or $p - a + p - b$. The

time for the middle step is $4\lambda + 3b^2\chi$. But since the four quadrants proceed completely independently, they can go in parallel, and the time is

$$T \approx 2(a' + b')\lambda + (a' + b')b^2\chi$$

where $a' = \max(a, n - a)$ and $b' = \max(b, n - b)$

Based on this, what is the fastest algorithm for the problem. [5pts]

Solution: Make $\langle a, b \rangle$ at the center of the grid, so $a' = b' = \frac{p}{2}$. This gives us

$$T \approx 2p\lambda + pb^2\chi$$

which is an improvement by a factor of 2.

d. Extra Credit: The machine is now modified to be a torus, i.e., the connections “wrap around” the edge of the grid—the west border is a neighbor of the east border and the north border is similarly adjacent to the south border. Describe how to further improve the algorithm, and derive a formula for its execution time. [10 pts].

Solution: Nissa had a clever solution, that she will post.