Computer Science 133, Homework 2

Michael Wu
UID: 404751542

February 4th, 2019

Problem 1

a) There is a read after write dependency between statements S_1 and S_2. There is also a write after write dependency between statements S_1 and S_2. Inside the loop, S_2 has all three types of dependencies with itself: read after write, write after read, and write after write.

b)

1. Loop Permutation - This can be applied to the i and j loops, but not the k loop since the c array is set to zero outside the k loop.
2. Loop Distribution - This can be applied by splitting the initialization of the c array into a separate doubly nested loop. Then the k loop could be permuted.
3. Loop Fusion - This cannot be applied as there is only one loop structure.
4. Loop Shifting - This can be applied to any of the loops, but would not result in any benefits since each iteration performs the same operation.
5. Loop Unrolling - This can be applied to the innermost loop. It would lead to speed increases due to pipelining.
6. Loop Strip-Mining - This can be applied to any loop. At higher dimensions this is equivalent to tiling.

7. Loop Unroll-and-Jam - This can be applied to any of the loops, as long as the zero assignments are placed prior to the k loop.

8. Loop Tiling - This can be applied to the i and j loops, but the zero assignment means that tiling k outside of the i and j loops would not work.

9. Loop Parallelization - This would work for the i and j loops since they cause assignments to unique locations in the c array, but parallelizing the k loop could lead to race conditions.

10. Loop Vectorization - This would require first splitting the initialization out, then permuting the arrays so that the j index is the innermost loop. Then the j loop could be vectorized.

Problem 2

```
void histogram (int* a, int* h) {
#pragma omp parallel
    {
        int* local = (int*) malloc(m*sizeof(int));
        for (int i=0; i<m; i++) {
            local[i] = 0;
        }
#pragma omp for
        for (int i=0; i<N; i++) {
            local[a[i]]++;
        }
#pragma omp critical
        for (int i=0; i<m; i++) {
            h[i] += local[i];
        }
        free(local);
    }
}
```

There is a possibility of a race condition if two threads try to update one bucket of the histogram at the same time. Then they might overwrite one

another and accidentally cause an increment to be lost. I handle this by having each thread make a local array of size m and then collecting the local arrays into the histogram inside a critical section. This way the threads do not overwrite each other.

Problem 3

```
#include <omp.h>
#include <math.h>
int main()
{
    int n = 100000;
    double dx = 1.0/(double) n;
    double result = 0;
#pragma omp parallel for num_threads(16) reduction(+:result)
    for (int i=0; i<n; i++) {
        double x = i*dx;
        result += dx * sqrt(x) / (1 + x * x * x);
    }
}
```

In my implementation I dealt with race conditions by using the reduction operator. I could also have each thread store its own accumulator variable and then collect them together with a critical section. This would have the effect of manually performing the reduction and perform about the same as the current implementation. Alternatively, I could lock the result variable while adding to it inside the loop. This would lead to horrible performance as threads would constantly be waiting to add to the result. Clearly my implementation is better. Note that since an analytical solution exists to this problem, it would be simpler to use that as it would be constant time. My implementation violates the "start with the fastest optimal algorithm" principle.

Problem 4

I expect Alice to finish first. This is because by assigning the shortest jobs last, Alice can achieve better load balancing than Bob. For example assume

that there are two processors. There is one very long job and many very short jobs. When Alice schedules her jobs, one processor completes the very long job while the other handles the very short jobs. When Bob schedules his jobs, both processors have approximately half of the short jobs until the very end, when one processor must handle the long job while the other one sits idle. In this case, Alice's implementation has less idle time than Bob's, so her implementation should finish first.

Problem 5

a) It seems like the default scheduling method is static with the iterations being divided into blocks equal to the number of threads. Since there are 4 threads, the first thread will execute iterations 0 to 2, the second 3 to 5, the third 6 to 8, and the last 9 to 11. Then the time taken will be dependent on the last thread and the program will complete in $10 + 11 + 12 = 33$ minutes.

b) The first thread will execute iterations 0 and 1, taking 3 minutes. The second thread will execute iterations 2 and 3, taking 7 minutes. The third thread will execute iterations 4 and 5, taking 11 minutes. The fourth thread will execute iterations 6 and 7, taking 15 minutes. The first thread will be free before the others and execute iterations 8 and 9, taking 19 minutes. The second thread will then be free next and execute iterations 10 and 11, taking 23 minutes. Then the longest running time is caused by thread two, which takes a total of 30 minutes to complete.

c) Let's assume that in guided scheduling the chunk size will be rounded up. Our initial chunk size starts at 3, so the first thread will execute iterations 0 to 2, taking 6 minutes. There are 9 iterations left, so our chunk size remains at three. The second thread will execute iterations 3 to 5, taking 15 minutes. There are 6 iterations left so our chunk size is 2. The third thread will execute iterations 6 and 7, taking 15 minutes. Our chunk size is now 1. The fourth thread will execute iteration 8, taking 9 minutes. Then when thread one is free it will execute iteration 9, taking 10 minutes. Then when thread four is free it will execute iteration 10, taking 11 minutes. Either thread two or three will then execute iteration 11, taking 12 minutes. Then the total time to complete is $15 + 12 = 27$ minutes.