Homework 2

Due: Feb 17th @ 11:59 pm (as a pdf file)

Nathan Nard (nnard2)

1. Consider the following sequence of memory accesses on a quad-core processor with private L1 caches for each core:

```
// Cache state at timestep 0
X = A[0]; // by core 0
// Cache state at timestep 1
Y = A[0] * A[1] ; // by core 1
// Cache state at timestep 2
A[3] = 4 ; // by core 2
// Cache state at timestep 3
Z = A[2] - A[0] ; // by core 3
// Cache state at timestep 4
```

Referring to the MESI protocol, write down the cache line state transitions at each time step (timestep 0 is already provided) as commented above using the table provided below. Assume the cache is initially empty for all cores (hence all line states are invalid), `A` is an array of integers (4 bytes each), the size of a cache line is 16 bytes, and that memory locations `A[0]` to `A[3]` all fit in a single line. State any other assumptions clearly.

	Timestep 0	**Timestep 1**	**Timestep 2**	**Timestep 3**	**Timestep 4**
Core 0	I	E	S	I	I
Core 1	I	I	S	I	I
Core 2	I	I	I	M	S
Core 3	I	I	I	I	S

Assuming:

- Entire array A (ie, the whole cache line) is transferrred when reading and writing to any element in it.
- A core attempting to read the cache line when another has it in the M state is allowed to fetch the modified cache line from that M state core, resulting in both obtaining the S state.

2. Consider the following serial code, where `C` and `D` are integer arrays.

```
k = 0;
x = 999999.0;
int i, j;
for (i = 0; i < N; i++) {
      k += 4;
      for (j = k; j < N; j++) {
            C[i] += D[j] * 3;
      }
      if (C[i] < x) {
            x = C[i];
      }
}
```

Parallelize the outer loop using the `#pragma omp parallel` directive (**not** `parallel for`), ensuring the code is both efficient and correct (i.e. the final values for `i`, `j`, `x`, `k`, `C`, and `D` are the same as the given sequential version). You should not use any additional OpenMP directives (aside from the one use of `#pragma omp parallel`), but function calls such as `omp_get_num_threads()` and adding OpenMP clauses to the `parallel` directive are fine.

(You are allowed to change the code. Note you will encounter multiple issues if you used a simple `#pragma omp parallel` directive without changing the code.)

```
k = 0;
x = 999999.0;
int i, j;

// begin adding your code from here
#pragma omp parallel private(i,j,k)
{
    int id = omp_get_thread_num();
    int p = omp_get_num_threads();

    int sub1 = (N - 4) / 4;
    int mystart1 = (sub1*id)/p;
    int myend1   = (sub1*(id+1))/p;

    int sub2 = N - sub1;
    int mystart2 = (sub2*id)/p + sub1;
    int myend2   = (sub2*(id+1))/p + sub1;


    //continued on next page
```

```
    //account for 0 modulo
    if (id == (p-1)) {
        myend1 = sub1;
        myend2 = N;
    }

    for (i = mystart1; i <myend2; ((i+1)==myend1) ? i=mystart2 : i++){
        k = 4*i+4;
        for (j = k; j < N; j++) {
            C[i] += D[j] * 3;
        }
        if (C[i] < x){
            x = C[i];
        }
    }
}
```

Only the first (N-4)/4 elements in C will be modified, the rest of C will be read and used only to determine some minimum (x). This suggests two different subarrays within C that have different computational loads. The subarray that gets modified at the beginning is smaller, but requires a lot more computation while the rest of the array is only used to check for a minimum, requiring less computation. Thus, the ideal strategy is to divide the range into 2*threadCount subarrays, and each thread will get 1 high computation load subarray and 1 low computation load subarray in order to divide up the computation load evenly. So there are 4 indexes to be calculated for each thread to define the boundaries of these two subarrays that that thread will work on. Along with that method, a private(i,j,k) clause is required to keep i,j,k consistent on the threads and the k+=4 line can be replaced with an algebraic expression dependent on i to remove loop-carried dependence on k.

Here is a printout of my gtests results, with run times, comparing the sequential (vanilla) and parallel versions on an array with size of N = 10000:

```
[==========] Running 2 tests from 1 test case.
[----------] Global test environment set-up.
[----------] 2 tests from hwk2question2
[ RUN      ] hwk2question2.vanilla
Vanilla run time: 16 ms
[       OK ] hwk2question2.vanilla (17 ms)
[ RUN      ] hwk2question2.parallel
Parallel run time: 6 ms
[       OK ] hwk2question2.parallel (7 ms)
[----------] 2 tests from hwk2question2 (24 ms total)

[----------] Global test environment tear-down
[==========] 2 tests from 1 test case ran. (24 ms total)
[  PASSED  ] 2 tests.
```

3. Histogram is a graphical representation of the distribution of data. Data is classified into `nbins` categories/bins. Histogram tells us the number of data items that fall into each of these categories/bins. The following code *attempts* to categorize each of the `n` elements of the array `data` into one of the `nbins` bins/indices of an array `hist`, as well as compute the size of the largest bin, in a parallel fashion using OpenMP. Assume that `f()` returns a number between `0` and `nbins-1`.

a. Please correct the code so that it produces the correct histogram by **only** adding additional `#pragma` statements and/or modifying existing ones (you should not have to create more variables or arrays, or change the non-`pragma` lines in any way). You may **not** use the `reduction` clause anywhere in this part.

```
void *data;
void generate_histogram() {
      int nbins = ...;
      int max_bin_size = 0;
      int *hist = (int *) malloc(sizeof(int)*nbins);
      #pragma omp parallel for
      for (int i = 0; i < n; i++) {
            int bin = f(data[i]);
            hist[bin] += 1;
            #pragma omp critical
            {
                  if (hist[bin] > max_bin_size) {
                        max_bin_size = hist[bin];
                  }
            }
      }
}
```

With the original code, the for loop was being run by each thread, so the generated histogram was a scaled version of the correct histogram:

num_threads * correct_histogram

So to fix this, we can simply add a “for” clause to divide up the work so there’s no repeated calculations.

Part b on the next page.

b. Even though the parallel code is now correct after your modifications from part a), there will not be much performance improvement, if any, over the sequential version. Can you figure out why? Please rewrite the code to improve the parallel performance. You may use the `reduction` clause here. (Hint: consider using a second `parallel for` loop)

The "omp critical" directive is expensive and therefore is bottlenecking our program. There's a more efficient way of accomplishing its purpose. We can extract the if block from the for loop and put it in a second for loop. The if block is computing the max value, so it can be replaced with the max() method. We can also parallelize this second for loop, but we need to include a reduction clause to prevent errors (however, it may be better to not parallelize the second for loop). Once we remove the "omp critical" directive, there are now race conditions on the "hist[bin] += 1" line, so now we need a "omp atomic" directive on that line to prevent the race conditions. This "omp atomic" is much more efficient than the "omp critical" directive. Below is my modified code:

```
void *data;
void generate_histogram() {
      int nbins = ...;
      int max_bin_size = 0;
      int *hist = (int *) malloc(sizeof(int)*nbins);
      #pragma omp parallel for
      for (int i = 0; i < n; i++) {
            int bin = f(data[i]);
            #pragma omp atomic
            hist[bin] += 1;
      }

      #pragma omp parallel for reduction(max:max_bin_size)
       for (int i = 0; i < nbins; i++) {
            max_bin_size = max(hist[i], max_bin_size);
       }

}

//The f I used for my tests
int f(int n) {
    return n/(maxValue/nbins);
}
```

This improves on the code from part a, but if f() is low complexity, ie O(1), then it may not perform better than the sequential version. On my machine, part a's version works a 10000 element array in about 1.934 ms while part b's version works that same array in about 0.854 ms. The sequential version is much faster though, coming in at about 0.032 ms.

4. Consider a sorted array A of N numbers. We want to rearrange it in another array B of N numbers such that 1) all the odd numbers are sorted and appear before any even numbers and 2) all even numbers are sorted. In other words, we want to partition the array into 2 parts, with the first part consisting of odd numbers in sorted order and the second consisting of even numbers in sorted order. For example:

Array A (length 16, colors correspond to regions "owned" by a thread):

1	3	4	8	11	15	21	24	30	37	50	58	60	63	71	75

Array B:

1	3	11	15	21	37	63	71	75	4	8	24	30	50	58	60

Complete the following code that performs this operation using a thread-oriented approach (i.e. using `omp parallel` without worksharing `for` loop), where each thread deals with an approximately equal size region of the input array. (Hint: it is useful to compute the sum of odd numbers in each thread's range of the original array. Then ask "where do my odd numbers go in array B?")

```
int num_threads = omp_get_max_threads();
int n = N; // N is a macro, I defined mine as 10000
int A[N], B[N]; // assume A is already sorted
int oddCnt[num_threads], evenCnt[num_threads]; // assume initially 0
// add any additional declarations here if needed
// note the change from N to n below (can't pass a macro to the shared clause)
#pragma omp parallel shared(A, B, oddCnt, evenCnt, num_threads, n)
{
    /* a) Compute start and end indices for each thread */
    int id = omp_get_thread_num();

    int mystart = (id*n)/num_threads;
    int myend   = ((id+1)*n)/num_threads;

    if (id == num_threads-1) myend = n;


    /* b) Count how many odd and even numbers there are in each thread's region, update
    oddCnt and evenCnt appropriately */

    for (int i = mystart; i < myend; i++){
        if (A[i] % 2 == 0)
            evenCnt[id]++;
        else
            oddCnt[id]++;
    }

    /* c) Compute prefix sums of oddCnt and evenCnt arrays (done by one thread only). */

    #pragma omp barrier
    #pragma omp single
    for (int i = 1; i < num_threads; i++){
        oddCnt[i] += oddCnt[i-1];
        evenCnt[i] += evenCnt[i-1];
    }


// continued on next page
```

```
    /* d) Each thread then iterates through own local region again and, for each odd and
    even number, determine what position in array B it must go to (using oddCnt and
    evenCnt and any additional variables you might have declared) and update B. */

    int evenStart = oddCnt[num_threads-1];
    int evenIndex = (id == 0) ? evenStart : evenCnt[id-1]+evenStart;
    int oddIndex  = (id == 0) ? 0 : oddCnt[id-1];

    for (int i = mystart; i < myend; i++){
        if (A[i] % 2 == 0){
            B[evenIndex] = A[i];
            evenIndex++;
        } else {
            B[oddIndex] = A[i];
            oddIndex++;
        }
    }
}
```

Note: please follow the steps outlined in parts a) through d), any solution that does not will not receive full credit (even if it is a correct approach).

For an array of size 10000 with roughly equal number of odds and evens, my parallel and serial implementations have the following run times:

Parallel run time: 37 µs
Serial run time: 108 µs