Homework 2 Solutions

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 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.

Solution:

	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

2. Consider the following serial code, where C and D are all int 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 `x`, and `C` are the same as the given sequential version). You should not add any additional OpenMP directives (aside from the one use of `#pragma omp parallel`), but function calls such as `omp_get_num_threads()` and OpenMP clauses are fine.

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

Solution:

```
k = 0;
x = 999999.0;
int i, j;
#pragma omp parallel private(i,j,k) reduction(min:x)
{
      int thread_id = omp_get_thread_id();
      int i_start = thread_id*N/omp_get_num_threads();
      int i_end = min(N, (thread_id+1)*N/omp_get_num_threads());
      for (i = i_start; i < i_end; i++) {
            k = 4*(i+1);
            for (j = k; j < N; j++) {
                  C[i] += D[j] * 3;
            }
            if (C[i] < x) {
                  x = C[i];
            }
      }
}
```

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`.

```
void *data;
void generate_histogram() {
      int nbins = ...;
      int max_bin_size = 0;
      int *hist = (int *) malloc(sizeof(int)*nbins);
      #pragma omp parallel
      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];
                  }
            }
      }
}
```

a. Please correct the code so that it produces the correct histogram by either adding additional `#pragma` statements or modifying existing ones. You may **not** use the `reduction` clause anywhere.

   Solution:

```
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 critical
            {
                  if (hist[bin] > max_bin_size) {
                        max_bin_size = hist[bin];
                  }
            }
      }
}
```

b. Even though the parallel code is now correct after your modifications from part a), there is no performance improvement over the sequential version. Can you figure out why? Please rewrite the code to improve the parallel performance. (Hint: consider using a second `parallel for` loop)

Solution:

The code has no performance improvement over the sequential version because it uses “omp critical”, which ensures that only one thread can access the section at a time. Since every thread must execute this critical section at every iteration, this code essentially boils down to sequential execution.

Code (reduction in separate loop):

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

Alternative code 1 (reduction in separate loop, use reduction for hist):
Each thread will have its own hist array, which won’t be memory efficient

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

```
    }
}
```

Alternative code 2 (reduction within same loop):

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

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. See below for an example.

Complete the following code that performs this operation using a thread-oriented approach (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`?")

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

```
int num_threads = …;
int N = …; // assume N%num_threads == 0
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

#pragma omp parallel shared(A, B, oddCnt, evenCnt)
{
        /* a) Compute start and end indices for each thread */
        int id = omp_get_thread_num();
        int threads = omp_get_num_threads();
        int start = id*N/threads;
        int end = min((id+1)*N/threads, N);

        /* b) Count how many odd and even numbers there are in each thread's region, update
        oddCnt and evenCnt appropriately */
        for (int i = start; i < end; i++) {
                if (A[i] % 2 != 0) {
                                oddCnt[id]++;
                        } else {
                                evenCnt[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 < threads; i++) {
                                evenCnt[i] += evenCnt[i-1];
                                oddCnt[i] += oddCnt[i-1];
                }
        }
```

/* 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 odd_i = (id > 0) ? oddCnt[id-1] : 0;
        int even_i = oddCnt[threads-1] + (id > 0) ? evenCnt[id-1] : 0;
        for (int i = start; i < end; i++) {
                if (A[i] % 2 != 0) {
                                B[odd_i] = A[i];
                                odd_i++;
                } else {
                                B[even_i] = A[i];
                                even_i++;
                }
        }
}
```