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

```c
// 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
// 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:**

<table>
<thead>
<tr>
<th></th>
<th>Timestep 0</th>
<th>Timestep 1</th>
<th>Timestep 2</th>
<th>Timestep 3</th>
<th>Timestep 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Core 0</td>
<td>I</td>
<td>E</td>
<td>S</td>
<td>I</td>
<td>I</td>
</tr>
<tr>
<td>Core 1</td>
<td>I</td>
<td>I</td>
<td>S</td>
<td>I</td>
<td>I</td>
</tr>
<tr>
<td>Core 2</td>
<td>I</td>
<td>I</td>
<td>I</td>
<td>M</td>
<td>S</td>
</tr>
<tr>
<td>Core 3</td>
<td>I</td>
<td>I</td>
<td>I</td>
<td>I</td>
<td>S</td>
</tr>
</tbody>
</table>
2. Consider the following serial code, where A, B, C, and D are all int arrays:

```c
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. Note you would encounter multiple issues if you used a simple `#pragma omp parallel` directive without changing the code. You should not add any additional OpenMP directives or clauses, but function calls such as `omp_get_num_threads()` are fine.

**Solution:**
```c
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 \( \text{data} \) into one of the \( \text{nbins} \) bins/indices of an array \( \text{hist} \), 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 \( \text{nbins}-1 \).

```c
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:**
```c
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):**
```c
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 bin = 0; bin < nbins; bin++) {
        if (hist[bin] > max_bin_size) {
            max_bin_size = hist[bin];
        }
    }
}
```

**Alternative code (reduction within same loop):**
```c
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:
a. All the odd numbers are sorted and appear before any even numbers.
b. 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.

Write code for carrying out this operation, using a thread-oriented approach (using 

\texttt{omp parallel}, without worksharing \texttt{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 \( B \)?”)

\textbf{Solution:}

\textbf{General algorithm:}

1. Compute for each thread, the number of even and odd numbers owned by preceding threads (similar to prefix sum thread-oriented approach), also compute the total number of odd numbers.
   a. \( \text{prefix \_ sum \_ odd}[i] = \) total number of odd numbers “owned” by threads 0..i
   b. \( \text{prefix \_ sum \_ even}[i] = \) total number of even numbers “owned” by threads 0..i
2. Save the total number of odd numbers as the starting index for the “even partition”. The starting index for the “odd partition” is trivially 0.
3. Go through each thread’s subarray, keeping track of whether the current element is the 1st/2nd/3rd/etc odd/even number seen so far.
   a. Place the ith odd number at the \((i + \text{prefix \_ sum \_ odd[curr \_ thread]})\) position in \( B \)
   b. Place the ith even number at the \((i + \text{prefix \_ sum \_ even[curr \_ thread]} + \text{total \_ odd \_ nums})\) position in \( B \).

Here is an example implementation. This is not the only solution.

\begin{verbatim}
int num_threads = omp_get_num_threads();
int* prefix_sum_odd = (int*)malloc(num_threads*\text{sizeof}(int));
int* prefix_sum_even = (int*)malloc(num_threads*\text{sizeof}(int));
#pragma omp parallel shared(prefix_sum_odd,prefix_sum_even,A,B)
{
    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);

    // each thread counts how many evens and odds it owns, as well as // global count of odds
    prefix_sum_odd[id] = 0;
    prefix_sum_even[id] = 0;
    for (int i = start; i < end; i++) {
        if (A[i] % 2 != 0) {
            prefix_sum_odd[id]++;
        } else {
            prefix_sum_even[id]++;
        }
    }
}
\end{verbatim}
#pragma omp barrier

// one thread computes the prefix counts of evens and odds
#pragma omp single
{
    for (int i = 1; i < threads; i++) {
        prefix_sum_even[i] += prefix_sum_even[i-1];
        prefix_sum_odd[i] += prefix_sum_odd[i-1];
    }
}

// starting locations in B for each thread’s even and odd numbers
int odd_i = (id > 0) ? prefix_sum_odd[id-1] : 0;
int even_i = prefix_sum_odd[threads-1] +
             (id > 0) ? prefix_sum_even[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++;
    }
}
}