Problem 1

(a)

True. This for loop actually just does the operation `a[i] = b[(i + 1) % N]`. Each iteration doesn't depend on others.

(b)

False. Because of the `if (b[i] == 0) break;` Whether certain iteration will be executed is unknown and the for loop have to be executed sequentially.

(c)

False. Because of the `a[i] += a[i+1]`, the current iteration's `a[i+1]` must be read after the previous iteration's `a[i]` be updated. This could only be done sequentially to ensure its correctness.

(d)

True. This for loop just does the operation of `z[i] = sqrt(x[i] * x[i] + y[i] * y[i])`. Each iteration doesn't depend on others.

(e)

False. It's impossible to predict whether the `func(i)` will do any change to the `a` or the environment that later call to `func` might depend. (i.e. `func` is not pure). So, the for lop have to be executed sequentially.

(f)

True. This for loop increment the second half part by the corresponding element in first half part. Each iteration doesn't depend on others.

Problem 2

```
double tmp, minab;
double minval = INFINITY;

#pragma omp parallel for
for (int i = 0; i < N; i ++) {
    tmp = a[i];
    a[i] = b[i];
    b[i] = tmp;
    #pragma omp critical
    {
        minab = (a[i] > b[i]) ? (b[i]) : (a[i]) ;
        minval = (minval > minab) ? (minab) : (minval);
    }
}
printf("the minimum value if %lf\n", minval);
```

This technically "parallels" the program. But the program is still bottlenecked by the critical section. One better solution is to use reduction:

```
double tmp, minab;
double minval = INFINITY;

#pragma omp parallel for
for (int i = 0; i < N; i ++) {
    tmp = a[i];
    a[i] = b[i];
    b[i] = tmp;
}

#pragma omp parallel for reduction(min:minval)
for (int i = 0; i < N; i ++) {
    minab = (a[i] > b[i]) ? (b[i]) : (a[i]) ;
    minval = (minval > minab) ? (minab) : (minval);
}

printf("the minimum value if %lf\n", minval);
```

This, as I tested on the array of size `100000`, is about 100 times faster than the parallel code that use critical sections. (but it seems for this size, unparalleled code is still better) You could see the `q2.c` in my `hw2` gitlab repo is you want to (I include some correctness testing / time measurement code).

Problem 3

```
array size = 1000000
number of cores = 4
sum is 494825.930000
sum with reduction takes 0.000563 seconds
sum is 494825.930000
sum with atomic takes 0.085948 seconds
sum is 494825.930000
sum with critical takes 0.181362 seconds
```

(1)

the `omp_get_num_procs()` is actually twice of the `number_of_cores` (we could see this in the code) so it is `8`.

(2)

See above

(3)

The reduction one. (Other two methods just make the sum could only be accessed by one thread at one time, so actually only one thread is "working" at the same time. However, the reduction method actually makes multiple thread calculating the sum of one part of the array at the same time, and the main thread sum of the partial sums from each thread in the end, so it's faster).

Problem 4

`#pragma omp parallel for schedule(static, 2)`

Threads	Iterations
Thread 0	0, 1, 8, 9
Thread 1	2, 3, 10, 11
Thread 2	4, 5
Thread 3	6, 7

Time	0	1	2	3	4	5	6	7	8	9	10	11
Thread 0	0	1	1	8	8	9						
Thread 1	2	2	3	3	10	11	11					
Thread 2	4	4	4	5								
Thread 3	6	7	7									

`#pragma omp parallel for schedule(dynamic)`

Threads	Iterations
Thread 0	0, 4, 10
Thread 1	1, 5, 8
Thread 2	2, 6, 9, 11
Thread 3	3, 7

Time	0	1	2	3	4	5	6	7	8	9	10	11
Thread 0	0	4	4	4	10							
Thread 1	1	1	5	8	8							
Thread 2	2	2	6	9	11	11						
Thread 3	3	3	7	7								