```c
#include <mpi.h>
#include <stdlib.h>
#include <stdio.h>

#define N 1000

int a[N];
int rank, size;

int maxloc(void) {
    int i, index, gmax, maxindex;
    int max = -1;

    /* find local maximum */
    for (i = 0; i < N; i ++) {
        if (a[i] > max) {
            index = i;
            max = a[i]
        }
    }

    /* find global maximum */
    MPI_Allreduce(&max, &gmax, 1, MPI_INT, MPI_MAX, MPI_COMM_WORLD);

    if (gmax == max) {
        index = N * rank + index;
    } else {
        index = N * size
    }

    /* find index of first occurence of global maximum */
    MPI_Allreduce(&index, &maxindex, 1, MPI_INT, MPI_MIN, MPI_COMM_WORLD);

    if (rank == 0) {
        return maxindex;
    } else {
        return -1;
    }
}

int main() {
    int result;
    MPI_Init(NULL, NULL);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    ... /* initialize a */

    result = maxloc();
    if (rank == 0)
        printf("%d\n", result);
    MPI_Finalize();
}
```

Problem 2

(1)

(c), since we want to **distribute** the data to all the p processes (i.e. `set[i]` to `process[i]`), the `MPI_scatter` does what we want

(2)

(b), `MPI_Wait` will block us from doing anything until the specified operation is finished, while the `MPI_Test` will immediate returns and tells us whether the specified operation is finished or not. We probably don't want our program blocked, because in that case we could do other computation. So `MPI_Test` is preferred.

(3)

(c), see https://www.mpi-forum.org/docs/mpi-1.1/mpi-11-html/node64.html.

(4)

(b), see https://www.mpi-forum.org/docs/mpi-1.1/mpi-11-html/node75.html#Node75

Problem 3

(1)

We use

```
MPI_Allreduce(&count, &global_count, 1, MPI_INT, MPI_SUM, MPI_COMM_WORLD);
```

The `count` is the number of element current rank has, it will be sent to other process. The `global_count` is the integer we will receive, representing the total number of element in the array. The `1, MPI_INT` means that the content in the buffer `&count` (which is actually a pointer to a integer, but it could be thought as a buffer) needs to be interpreted as integers, and only the first integer in the buffer shall be sent to other process (which essentially just mean that `count` will be sent to other processes). `MPI_SUM` means that the `MPI_Allreduce` will perform summation operation, that is, the final `global_count` will be the sum of all the `count` received from each processes. We assume that all the processes use the default `MPI_COMM_WORLD` communicator.

(2)

We use

```
MPI_Scan(&count, &start_index, 1, MPI_INT, MPI_SUM, MPI_COMM_WORLD)
```

The `count` is the number of element current rank has, it will be sent to other process. The `start_index` is the integer we will receive, representing the the array index which its elements start. The `1, MPI_INT` means that the content in the buffer `&count` (which is actually a pointer to a integer, but it could be thought as a buffer) needs to be interpreted as integers, and only the first integer in the buffer shall be sent to other process (which essentially just mean that `count` will be sent to other processes). We see that in the end process rank `i` will have `start_index` equal to the summation of `count` from Process `i`, `i - 1` ... `1`, which is what we want.

(3)

We first calculate a `local_sum`, it is the sum of all the elements that current process has. Assume that the data in the array has type `MPI_CUSTOM_TYPE`. Then, we use

```
MPI_Allreduce(&local_sum, &global_sum, 1, MPI_CUSTOM_TYPE, MPI_SUM,
MPI_COMM_WORLD)
```

we will get the `global_sum`, which is the sum of all elements in the array.

we could get what rank our process is by using

```
MPI_Comm_rank(MPI_COMM_WORLD, &rank)
```

Then, we check if we `rank == 0`, if it's true, we `printf` the `global_sum / global_count`, where `global_count` is defined in (1)

(4)

Note that the $N^{\text{th}}/2$ is the element in the array with index `global_count / 2 - 1`.

First, each process will have its `start_index` from (2), and the `global_count` from (1). Again, assume that the data in the array has type `MPI_CUSTOM_TYPE`. We create a `MPI_Request req`.

We first get the rank of our current process (using `MPI_Comm_rank` just like what we did in (2))

then if the process satisfies the condition that `start_index <= global_count / 2 - 1` and that `start_index + count >= global_count / 2` (That's, the process has the element with index `global_count / 2 - 1` in the array) and that `rank != 0`, we will use

```
MPI_Send(&local_array[global_count / 2 - 1 - start_index], 1, MPI_CUSTOM_TYPE, 0,
0, MPI_COMM_WORLD, MPI_STATUS_IGNORE)
```

where the `local_array` is the part of array that stored in current process, the `global_count / 2 - 1 - start_index` is just the corresponding index of the $N^{\text{th}}/2$ element in the current `local_array`. Since we send to Process 0, our fourth parameter should be `0`, which mean that we will send the data into the Process 0. The tag here (fifth parameter) is `0`.

If `rank == 0`, that means we are in Process 0 and we should receive the data:

```
MPI_IRecv(&target, 1, MPI_CUSTOM_TYPE, MPI_ANY_SOURCE, 0, MPI_COMM_WORLD, &req)
```

the `target` is the $N^{\text{th}}/2$ element, and the `MPI_ANY_SOURCE` means we really don't care which rank send us this data. The tag here (fifth parameter) is `0`.

Then in the end we could wait for the receive to finish

```
MPI_Wait(&req, MPI_STATUS_IGNORE)
```

Notice the order of each MPI call: (it should roughly looks like something down below)

```
if (rank == 0) {
    MPI_Irecv( ... )
}

if ( ... ) {
    MPI_Send( ... )
}

MPI_Wait( ... )
```

This order, together with the `MPI_Irecv`, makes sure that if Process 0 sending data to itself, it will not create a dead-lock.

We could then use `printf` to print the `target`.

(5)

Again, assume that the data in the array has type `MPI_CUSTOM_TYPE`. The process 0 first need to send the value `v` to all other processes:

```
MPI_Bcast(&v, 1, MPI_CUSTOM_TYPE, 0, MPI_COMM_WORLD)
```

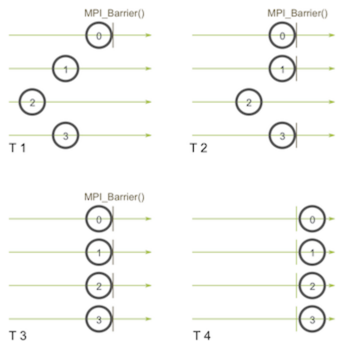
The fourth parameter (root) is `0` since we are sending data from Process 0. Then we will find the number of elements in each process that is less than or equal to the `v`. This process is easy and not included here. Call this number `le_count` ("less than or equal to count"). Then we will run:

```
MPI_Reduce(&le_count, &global_le_count, 1, MPI_CUSTOM_TYPE, MPI_SUM, 0,
MPI_COMM_WORLD)
```

We use `MPI_SUM` because we want total number of elements that is less than or equal to v, which is the summation of the number of elements that is less than or equal to v in each process. The fifth parameter (the destination process) should be 0, since we are going to send to Process 0 to print it. The `global_le_count` is the total number of elements in the array which are less than or equal to the specified value v. Then we could just do `printf` to print the `global_le_count`.

Problem 4

This is possible if Rank 0 reach the barrier first, and then the rank 1, and in the last, the rank 2. To be more specific, the `MPI_Barrier` will block all the process from continuing until they all reach the barrier. Something like this:



(Source: https://mpitutorial.com/tutorials/mpi-broadcast-and-collective-communication/). Then, in this case, say the program start at $t = 0s$, then at t_0, Rank 0 reach the code `barrier_before = get_time()` and record the time t_0. The Rank 0 is stuck at barrier to wait for other processes. Then at $t_0 + 3.5s$, the Rank 1 reach the code `barrier_before = get_time()` and record $t_1 = t_0 + 3.5s$. The Rank 1 is then also stuck at barrier to wait for other processes. Finally, at time $t_0 + 3.998s$, the Rank 2 reach the code `barrier_before = get_time()` and record $t_2 = t_0 + 3.998s$. The barrier then take some time to process it, and then Rank 0, 1, 2, are allowed to record the time t_3 at $t_3 = t_0 + 4s$. We see that therefore $\Delta t_0 = t_3 - t_0 = 4s$ and $\Delta t_1 = t_3 - t_1 = 0.5s$, $\Delta t_2 = t_3 - t_2 = 0.002s = 2ms$. This is roughly what happens for these three Ranks, and this explains why they record different elapsed time.