All of the executables can be built by typing make in the respective directory.
The executables will be named “prog”.
For task 2 and 3 the program expects arguments to be passed to it like so:
./prog pop_size num_generations num_iterations crossover_rate mutation_rate
For crossover and mutation rate you should provide a float between 0 and 1, such as 0.01 for a 10% rate.
An example call is: ./prog 1000 10 5 0.6 0.001

Task 1:

For the results see the *fitness.eps* graph. As can be seen my implementation quickly achieves maximum fitness.

For crossover I took two chromosomes and swapped the upper and lower halves. I tried several other approaches, such as swapping at a random point, but no other crossover method gave me good results.

I used this method for implementing a roulette wheel selection:
https://en.wikipedia.org/wiki/Fitness_proportionate_selection

My implementation works as long as the chromosome size is a multiple of 32, so it can span multiple ints.

Task 2:

The results of the most recent two games are stored in a 4 bit string. A cooperate (“C”) result is stored as 0, and a defect (“D”) result is stored as 1. So if player 1 defected and player 2 cooperated in the last two games the string would be 0101, or CDCD.

To determine the moves in the first two games I used the first two bits of the chromosome. It is encoded as follows: 00 = always cooperate, 01 == always defect, 10 and 11 = random choice.

I encoded the general strategy as follows: each chromosome has bits that determine the “reaction” to the previous two games. A previous game state of CCCC uses bit 2 to determine the move, while CCCD uses bit 3, and so on. If the respective bit is set to 1 the player will defect, and if it is set to 0 they will cooperate.

Task 3:

Everything is similar to what was described for task 2.
The only change of note is how I divided the work between multiple MPI nodes.

The naive approach is to have a for loop that looks like so:

```
for(i = proc_start_index; i < proc_end_index; i++){
        for(n = 0; n < pop_size; n++){
```

```
            play(i, n);
        }
}
```

In this approach each strategy plays every other strategy, but because of how the work is being divided strategies will play each other more than once. This means that when a game is played we can only record the result for one of the strategies, and we have to wait until the next time these strategies play each other to record the result for the second strategy.

I came up with an algorithm that plays each strategy against every other strategy exactly one time, and distributes the workload fairly well amongst the processes. To show this consider this example: we have a list of players like this

0	1	2	3	4

With two processes the following happens:

Process 1	**Process 2**
T0: Play 0 against 1	T0: Play 0 against 2
T1: Play 0 against 3	T1: Play 0 against 4
T2: Play 1 against 2	T2: Play 1 against 3
T3: Play 1 against 4	T3: *Nothing to do*

and so on.

Each process plays each strategy against strategies who are certain distances away. These distances are based on the process' rank and the number of other worker processes. In this algorithm when a game is played the result is recorded for both players. This means no redundant work is done.

With this algorithm each process has a partial result for the fitness of each player. In the above example we will need to sum the results that each process has for each player. This was done using MPI_Reduce, with an MPI_SUM operation to collect the total fitness.

The workload is not perfectly evenly distributed between processes, but as the population size grows it comes very close to being evenly distributed. Testing with various population and thread sizes did not show any problems with the population size not being a multiple of the number of processes.

Using the parameters shown below, the serial version took 200 seconds while my parallel version took 43 seconds. The parallel version had 6 processes, of which 5 were workers/slaves. This is quite good scaling.

Population size = 10000, number of generations = 10, number of PD rounds = 20, crossover rate = 0.6, mutation rate = 0.001

MPI_Bcast was used to send the chromosome data from the root process to the other processes. This was useful since the same data needed to be sent to each processes, and removed the need for me to use a for loop with an MPI_Send inside.

The overall fitness fluctuates in both my parallel and serial versions. In the number of 1s in a string problem each string can approach peak fitness without affecting other strings. However in the PD game chromosomes can only become fitter at the expense of others. Not all chromosomes can become perfectly fit. Consider how defecting is statistically the best choice. If all chromosomes always defected it would no longer be a good choice, as the payoff will always be 1 instead of the maximum of 5.