High Performance Computing Software Assignment 2 - Parallel Genetic Algorithm

Robert Cronin 12313002

January 20, 2017

1 Number of Ones - Serial

The initial question establishes the genetic algorithm in `genetic.h`. This header file is then reused throughout the rest of the code. It provides functions to set up an initial population, crossover, mutate, form a new generation etc.

Chromosomes are stored as an array of integers where each integer is effectively representing a binary string. Each integer in the array can represent a binary string up to length 30. A new integer is added to the array for every group of 30 needed. 30 was used instead of 32 to avoid messing around with unsigned ints and potential overflow issues by working at the limit.

For the number of 1s in a chromosome problem we see that the overall fitness converges towards the length of each chromosome*pop_size. Figure 1 has been normalised based on this value and we see it converges towards 1 indicating our population is "evolving" as expected on a survival of the fittest basis.

2 Prisoner's Dilemma - Serial

Next we adapt this code for Prisoner's Dilemma(PD). The same `genetic.h` file is used, meaning we just have to create a function to calculate fitness. The optimal result in terms of combined payoffs is (Cooperate, Cooperate) where they each get a payoff of 3 meaning a total payoff of 6.

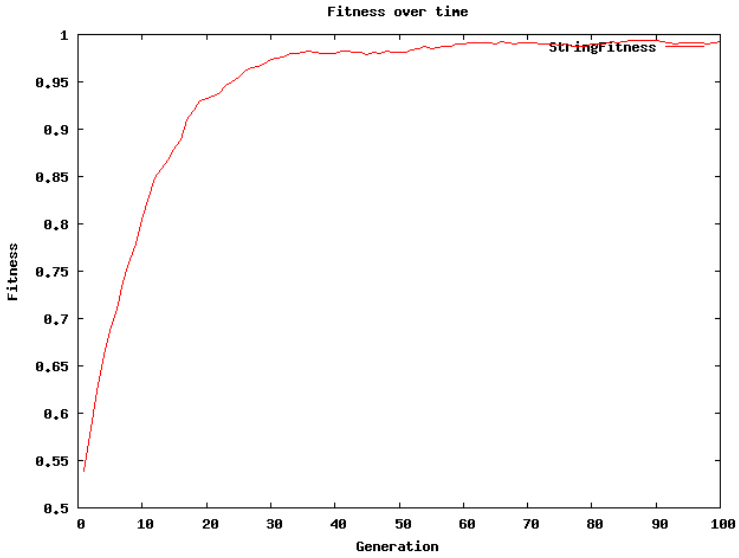

Figure 1: String of Ones Fitness - crossover 0.9, mutate 0.01, popsize 100

2.1 Convergence

Therefore if the program runs correctly we would expect the total fitness to converge to

$$(numberofgames) * (popsize - 1) * (popsize) * 3$$

So we normalise our values to this value again and we see in Figure 2 that it does converge to 1 and hence this value.

The shape of the graph is interesting however. The overall fitness initially tends to decrease before converging up to the maximum. This is most likely due to the nature of the game. In a one off playing of this game (Defect, Defect) is the Nash equilibrium whereby no player has any incentive to deviate from this strategy (they will receive a lower payoff by switching to Cooperate).

Looking at the payoffs though it is obvious that both players would be better off at (Cooperate, Cooperate) with a payoff of 3 each instead of 1. However at this point both player's have an incentive to deviate to Defect and receive a payoff of 5. As stated in an one off game (Cooperate, Cooperate) is thus unlikely to happen due to human's greedy nature.

Since we are playing a repeated game though the player's will learn (through evolution) that if they both choose to cooperate they will both be better off in the long run. So the dip in the graph is due to them exploring the "greedy" option before they see that those who cooperate start

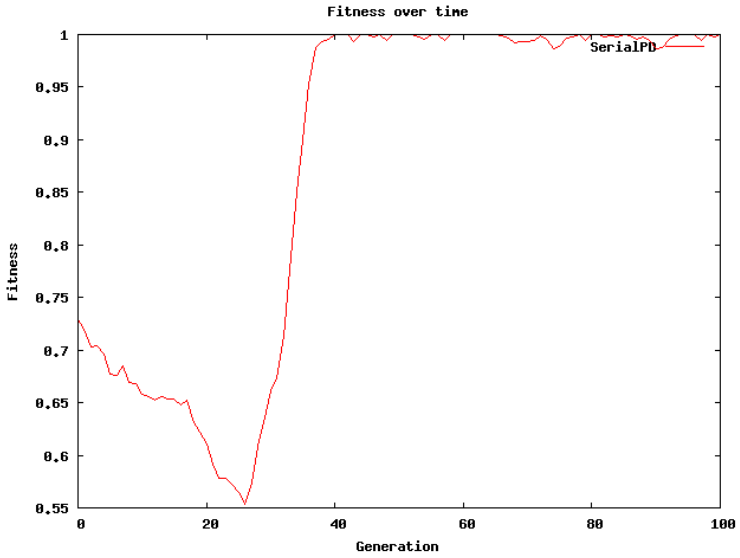

Figure 2: Serial Prisoner's Dilemma Fitness - crossover 0.9, mutate 0.01, popsize 200

receiving a higher payoff, after which the graph converges to the optimal solution.

In practice as humans reach the last few games, greediness becomes a factor again as the "long term" benefits are no longer in play and they stand to make a quick buck. Of course this code doesn't account for such behaviour and assumes the game will go on indefinitely.

2.2 Improvements

To calculate the fitness we have every player play everybody else a set number of times and then the sum of payoffs is used as fitness. This means the fitness calculation algorithm squares quadratically with popsize. A population size of 500 for 100 generations playing 100 games each time takes over 20s to run. However since the fitness calculations heavily dominates the compute time we can try implement some parallelisation for this step.

3 Prisoner's Dilemma - Parallel

The nature of the fitness function is "embarrassingly parallel" as we just need to run the same function over and over. To evaluate fitness everybody has to play everybody. The master sends a copy of the current population to

all the slaves at the beginning of each fitness calculation. He then sends all values of i in $[0, popsize)$ individually to each slave, the slave runs all games between i and $[i+1, popsize)$, reports back when finished and receives a new i value and so on until all values of i have been calculated by someone.

Then the master sends a finished signal telling all slaves to send their updated fitness values which are collated and summed by the master.

This naturally gives a similar graph to the serial version as the code is the same. However, popsize of 500, for 100 generations playing 100 games now only takes 8.4s on 4 processes or 3.7s on 16. It is not a direct speed up based on number of slaves but it definitely cuts the wall time significantly.

4 Prisoner's Dilemma - Island Model

The Island model is another technique for running genetic algorithms in parallel. Instead of Master-Slave, each process is their own "island" with their own population. Every specified number of generations a "migration" takes place whereby a specified number of "citizens" migrate from one island to another to promote genetic diversity, to try and find the global maximum.

Since the problem scales quadratically by popsize, splitting the population up across islands speeds up each generation calculation significantly but each island's results may be of lower quality due to smaller sample.

The same inputs as before now run in 2.2s on 4 processes and as seen in Figure 3 no quality appears to actually be lost as it converges at roughly the same rate but in far less computer time.

4.1 Different Rates

Now that we have found our quickest model we can use it to see how crossover and mutate rates affect our results. The default values are crossover=0.9 and mutate=0.01. We see in Figure 4 how changing one or both of these affect our results.

Setting crossover to 1 or mutate to 0 leads to faster convergence. However the risk with these is that the population may get stuck at a local maximum and won't be able to evolve out of it. So the slower convergence is a price we must pay to reduce the chance of local maxes.

Increasing the mutate rate to 0.1 causes much slower convergence as the mutation has more power to drag the population away from its target when it gets close.

Likewise if we set both to 0.5 we get terrible convergence as the rates are seriously hampering the evolution that is trying to take place.

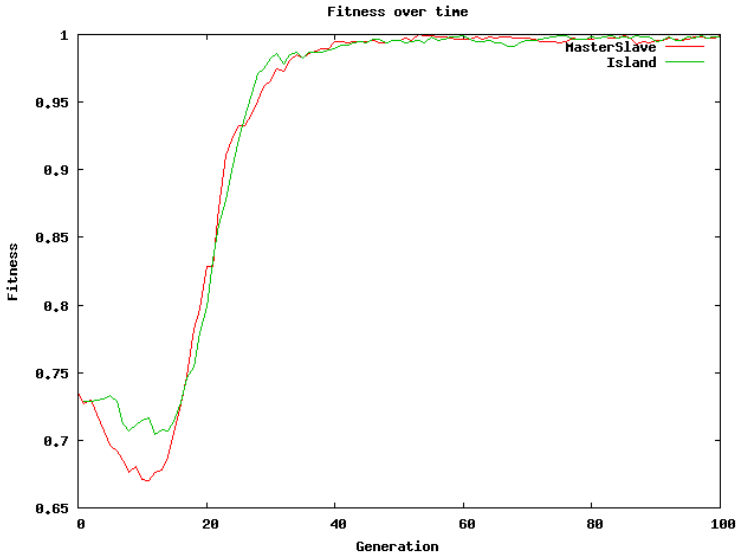

Figure 3: Parallel Prisoner's Dilemma Fitness - crossover 0.9, mutate 0.01, popsize 1000

5 Rock Paper Scissors

To model this game there is only 3 strategies (one for each option). Our old functions therefore need to be tweaked (as 3 isn't a power of 2). These tweaks are done in `rpsgenetic.h` where everything is essentially the same apart from this.

The fitness for each rock, each scissor and each paper are identical and can be calculated by counting the number of each and multiplying by relative payoffs times relative strategies. For example, all rocks have a fitness of $(no\_rocks - 1) * 2 + no\_scissors * 10$.

This can be done extremely quickly relative to other fitness functions. Since it is only a one off calculation as opposed to before where we had to repeatedly do the same calculations, there is no reason to write this in parallel and so it was written in serial.

The results in Figure 5 graph the fitness per strategy over time. As expected there is no clear winner and all 3 strategies constantly go up and down over time. If rock did well last generation it means more rocks will spawn next generation meaning scissors will do a lot better next time and so on causing this drastic cycles. The expected value for each game is between 2 and 4. 4 in the case of roughly equal amounts of each strategy and 2 in the generations when one strategy briefly dominates (i.e. majority are draws). This is reflected in the outputted expected total fitness range.

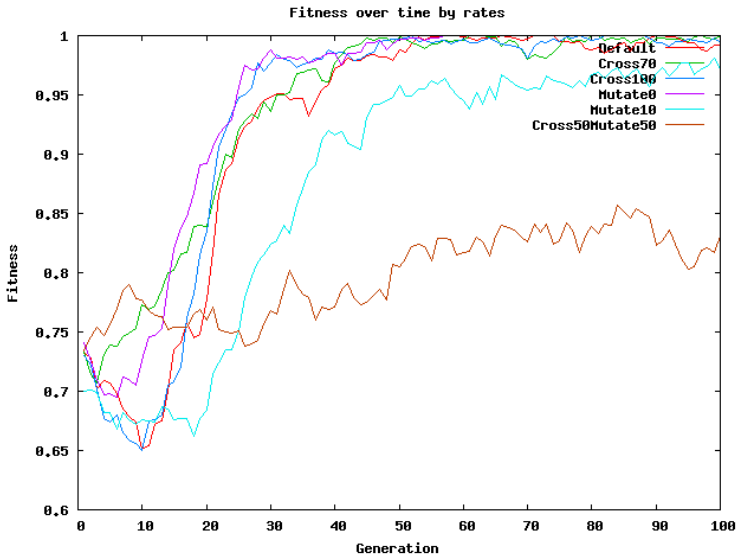

Figure 4: Prisoner's Dilemma Fitness - default crossover 0.9, mutate 0.01 (Labels show differences), popsize 400

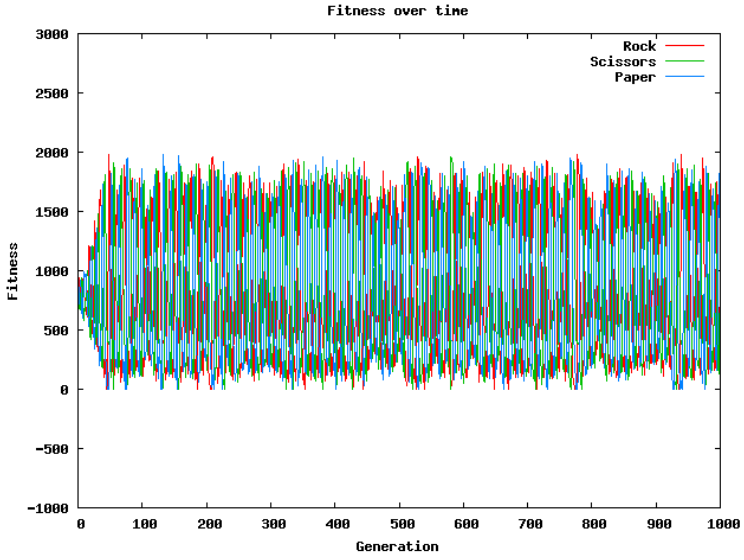

Figure 5: Rock Paper Scissors Fitness - crossover 0.9, mutate 0.01, popsize 200