Michael Sharp and Marcel Englmaier

Parallel processing

Preliminary design

3/17/14

**Possible approaches**

When looking at how we wanted to approach this problem we decided we wanted to design the code first to run linearly. Once we had the program running linearly we would look for part thats ran large loops. Since that section of the code is the same line or few lines being executed repetitively, we could break that task up among multiple core and parallelize the program.

Genetic Algorithms can be used in many different ways like finding how long it takes a mutation to become the normal or what a population could look like generations down the road. We choose to focus on the algorithm that calculates how long it takes for a mutation to appear. When running the program after the initial setup the next step is to figure out how the next generation will be created. To decide the next generation a set of parent codes need to be chosen. This process is called selection. There are a few different ways to go about choosing the parent codes: Roulette wheel, ranking, steady state, and Elitism selection. Since we were focusing on possible mutation appearance we chose to go with roulette wheel selection because it selects parents based of their fitness. A codes fitness it how likely it is to be selected to reproduction. This type of selection is the strongest and healthiest will survive concept.

The next portion of the program is to decide how we want our genetic code to be represented. There are a few different ways of doing this and we chose to use binary coding because it allows to generate a code as long as we want and let each chromosome in our data be represented by a one or zero. Another reason we chose binary encoding was because it offered several different ways to cross the parent genes to create the new generation. The four ways to cross the data are single point, two pint, uniform and arithmetic. Single point and double point did not give enough cross over and made the next generation look to similar to the parent generation. Do add a little more variation into the step of creating a new generation we chose to use uniform crossover. This crossover type switched between which parent was providing the code at randomly generated points.

The last step is to create a mutation section. Mutations are alterations in offspring genetic code that have nothing to do with the parents code. The big decision with mutation was basically how much we wanted it to effect it. We choose to give it a low probability and then if it happened it would only change one or two of the binary bits.

**Research description**

When doing research for this project we started off by looking up the basic idea behind genetic algorithms. After looking at the different types we chose which one we wanted to work on and started searching for information pertaining to the type we selected. So we started looking up how encoding, selection, and mutation effected population generation and the different ways to perform each of these tasks. Most of our research was through online articles, data bases, and sites that offered basic tutorials and walk troughs of the flow of genetic algorithms.

**Design Flow**

In the last report we showed the out line of the steps needed to run through a genetic algorithm.

**Outline of the Basic Genetic Algorithm**

1. **[Start]** Generate random population of *n* chromosomes (suitable solutions for the problem)
2. **[Fitness]** Evaluate the fitness *f(x)* of each chromosome *x* in the population
3. **[New population]** Create a new population by repeating following steps until the new population is complete
   1. **[Selection]** Select two parent chromosomes from a population according to their fitness (the better fitness, the bigger chance to be selected)
   2. **[Crossover]** With a crossover probability cross over the parents to form a new offspring (children). If no crossover was performed, offspring is an exact copy of parents.
   3. **[Mutation]** With a mutation probability mutate new offspring at each locus (position in chromosome).
   4. **[Accepting]** Place new offspring in a new population
4. **[Replace]** Use new generated population for a further run of algorithm
5. **[Test]** If the end condition is satisfied, **stop**, and return the best solution in current population
6. **[Loop]** Go to step **2**

Once looking through we figured out that the all the main steps could be parallelized and the New Population set could be parallelized but each of the sub steps out have to be performed within one thread.

**Approaches for testing**

Looking through the program there are a bunch of steps that each perform a small task. Recognizing this we will be able to separate out each task and run tests on each part individually. After we have each task working we can then start writing code for them to share information and work together. At this point we can run tests after each time we add a task to make sure the information is getting passed correctly and is being altered in the way we wanted it to be. Once all tasks have been added in we can then start running simulations of different scenarios to see if we can break the algorithm.