Marcel Englmaier & Michael Sharp

Parallel processing

Project Report

4/22/14

**Description**

For the term project we have decided to work on a genetic algorithm problem. A genetic algorithm is a combination of equations that can be used to show how variations in a population can change over time. The area of GA (genetic algorithms) we are focusing on is how genetic traits of a population change based on the genes that were passed on by the parents.

The main focus of this algorithm focuses on Chromosomes which are made up of genes. Each gene in this case can be considered a trait. The specific type of that trait like blond or brown hair is called alleles and where it is positioned is called its locus. Using these as variables we can plug them into functions that decide how the next population will look. There are three main functions to GA Selection, crossover and mutation. These three function decide which chromosomes and genes to use and how they will be combined.

**Goals**

Our goal is to use MPI to run a program that will test possible outcomes of a certain population using genetic algorithms.

**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**

Our fist pseudo code is listed below

**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. This allowed us to expand on the pseudo giving us a more detailed pseudo code to go off of.

**Outline of the Basic Genetic Algorithm 2**

Setup chromosome parent class

Get target number from user

Create random population

Check if target found

Loop while target not found

Test fitness

Test if target found

Create new population

Create new chromosome class

Loop till population full

Select two parents from old population

Cross over parents

Add to population

End loop

End loop

Count number of generations

End main loop

Assign fitness

Bring in chromosome

Compare to target

Fitness equals how far off from target

Return fitness

Mutate

Bring in chromosome

Check to see if should mutate

Mutate if needed

Return chromosome

Cross over

Bring in to chromosomes

Random choose a cross point

Swap parent values at cross point

Swapped values makes new offspring

Return offspring

Parent selection

Generate random number between 0 and fitness

Multiple number by total fitness to get selection group

Loop through population

If chromosome is great then selection group

Return chromosome

Else Return none

**Program and Method Description**

**Main.c (Adam&Eve)**

The main loop created the first generation by randomly generating 1s and 0s to make up the chromosomes. This loop is run until the population size is met. The parent generation is then given a fitness. This fitness is based off of how close to the target each chromosome is. Next a loop is entered to continuously create children generations until target is found or a max generation is reached. If max generation is reach this means that no matter how long you ran the simulation you probably would not find the target.

**MakeABaby**

This method is called to select parents to create children for the new generation. To do this a minimum fitness is chosen and the program randomly looks for parents to see if one is greater than or equal to the chosen fitness. If parents are found the information is then sent to the cross over method.

**Crossover**

Crossover takes in two parents then randomly chooses a crossover bit. It then swaps the bits at the crossover point. Creating two new children. This is depicted below.



**Mutation**

Mutation takes in the new children and randomly selects a bit from them. It then randomly selects a number if this number matches the mutation rate then the selected bit is flipped.

**AssignFitness**

Assignfitness uses two other methods, BinToDec and ParseBits, to decide the fitness of a chromosome. After calling BinToDec and ParseBits the value of the current chromosome can be compared to the target chromosome. Based off of how close it is to the target decides how high of a fitness it is assigned. BitToDec and ParseBits returns a combination of integers and mathematical operators when combined give the chromosome value. An example is given below.

**0110 1010 0101 1100 0100 1101 0010 1010 0001**

**6        +        5        \*        4         /        2        +       1**

**ParseBits**

ParseBits separates the chromosome into byte to be processed by BinToDec. The way a chromosome is broken up can be seen above.

**BinToDec**

Using the bytes given by ParseBits BinToDec turns every bit into an integer or an operator to be used by assignFitness. How each byte matches up to an integer or operator can be seen below.

**0:         0000**

**1:         0001**

**2:         0010**

**3:         0011**

**4:         0100**

**5:         0101**

**6:         0110**

**7:         0111**

**8:         1000**

**9:         1001**

**+:         1010**

**-:          1011**

**\*:          1100**

**/:          1101**

**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.

**Code and Implementation**

**Sequential Code**

//Authors: Marcel Englmaier & Michael Sharp

#include <stdio.h>

#include <math.h>

#include <time.h>

#include <stdlib.h>

#include <sys/time.h>

#define BITLENGTH 20

#define GENELENGTH 4

#define MAXGENERATIONS 2

#define CROSSOVERRATE 0.7

#define MUTATIONRATE 0.001

#define debug 0

struct subject

{

int bits[BITLENGTH];

float fitness;

};

struct timeval start;

struct timeval end;

float AssignFitness(int [], float);

int BinToDec(int[]);

int ParseBits(int[] , int \*);

void Crossover(int \*, int \*);

void MakeMeABaby(float, double, struct subject \*, int \*);

void Mutate(int \*);

void PrintChromo(int[]);

void PrintGeneSymbol(int);

int main(int argc, char \*argv[])

{

srand(time(NULL));

double populationSize = atoi(argv[1]);

float target = (float)atoi(argv[2]);

printf("Running with:\n Population Size = %.0f\n Bit Length = %d\n", populationSize, BITLENGTH);

long int i = 0;

int j = 0;

struct subject \*sheep = (struct subject\*)malloc(sizeof(struct subject) \* populationSize);

gettimeofday(&start, NULL);

//make the Adam&Eve generation

for(i = 0; i < populationSize; i++)

{

if(debug == 0)

{

printf("subject %d bits = ", i);

}

for(j = 0; j < BITLENGTH; j++)

{

sheep[i].bits[j] = rand() %2;

if(debug == 0)

{

printf("%d", sheep[i].bits[j]);

}

}

sheep[i].fitness = 0.0f;

if(debug == 0)

{

printf(" with fitness %f\n", sheep[i].fitness);

}

}

int howLongThisTook = 0;

int solutionFound = 0; //0 if false, 1 if true

struct subject \*childSpawn = (struct subject\*)malloc(sizeof(struct subject) \* populationSize);

while(solutionFound != 1)

{

float totalFitness = 0.0f;

for(i = 0; i < populationSize; i++)

{

sheep[i].fitness = AssignFitness(sheep[i].bits, target);

if(debug == 0)

{

printf("\n subject %d with fitness %f\n", i, sheep[i].fitness);

}

totalFitness += sheep[i].fitness;

if(debug == 0)

{

printf(" Total Fitness%f\n", totalFitness);

}

}

for(i = 0; i < populationSize; i++)

{

if(sheep[i].fitness == 999.0f)

{

printf("It's been found. It took %d generations to find\n", howLongThisTook);

solutionFound = 1;

break;

}

}

long childSpawnPopulationSize = 0;

int childSpawn1Bits[BITLENGTH];

int childSpawn2Bits[BITLENGTH];

if(solutionFound != 1)

{

printf("Creating New Generation\n");

while(childSpawnPopulationSize < populationSize)

{

MakeMeABaby(totalFitness, populationSize, sheep, childSpawn1Bits);

MakeMeABaby(totalFitness, populationSize, sheep, childSpawn2Bits);

Crossover(childSpawn1Bits, childSpawn2Bits);

Mutate(childSpawn1Bits);

Mutate(childSpawn2Bits);

for(i = 0; i < BITLENGTH; i++)

{

childSpawn[childSpawnPopulationSize].bits[i] = childSpawn1Bits[i];

childSpawn[childSpawnPopulationSize + 1].bits[i] = fuckSpawn2Bits[i];

}

if(debug == 0)

{

printf("childspawn%d: ", childSpawnPopulationSize);

for(i = 0; i < BITLENGTH; i++)

{

printf("%d", childSpawn[childSpawnPopulationSize].bits[i]);

}

printf("\nchildspawn%d: ",childSpawnPopulationSize + 1);

for(i = 0; i < BITLENGTH; i++)

{

printf("%d", childSpawn[childSpawnPopulationSize + 1].bits[i]);

}

printf("\n");

}

childSpawn[childSpawnPopulationSize].fitness = 0.0f;

childSpawn[childSpawnPopulationSize + 1].fitness = 0.0f;

childSpawnPopulationSize += 2;

}

for(i = 0; i < populationSize; i++)

{

sheep[i] = childSpawn[i];

}

howLongThisTook++;

printf("New Generation done! Now on Generation %d\n", howLongThisTook);

}

if(howLongThisTook > MAXGENERATIONS)

{

printf("Did not find a solution in maximum allowable runs\n");

solutionFound = 1;

}

}

gettimeofday(&end, NULL);

int timeran = (((end.tv\_sec - start.tv\_sec) \* 1000000) +(end.tv\_usec - start.tv\_usec));

printf("Completed in %d Nano Seconds\n", timeran);

return 0;

}

int BinToDec(int bits[])

{

int val = 0;

int valueToAdd = 1;

int k = 0;

for(k = GENELENGTH - 1; k >= 0; k--)

{

if(debug == 0)

{

printf("k=%d ", bits[k]);

}

if(bits[k] == 1)

{

val += valueToAdd;

}

valueToAdd \*= 2;

}

return val;

}

int ParseBits(int bits[], int\* buffer)

{

int counterBuffer = 0;

int operator = 1;

int currentGene = 0;

int i = 0;

for(i = 0; i < BITLENGTH; i += GENELENGTH)

{

currentGene = BinToDec(&bits[i]);

if(debug == 0)

{

printf("this gene %d \n", currentGene);

}

if(operator == 1)

{

if((currentGene < 10) || (currentGene > 13))

{

continue;

}

else

{

operator = 0;

buffer[counterBuffer++] = currentGene;

continue;

}

}

else

{

if (currentGene > 9)

{

continue;

}

else

{

operator = 1;

buffer[counterBuffer++] = currentGene;

continue;

}

}

}

for(i = 0; i < counterBuffer; i++)

{

if((buffer[i] == 13) && (buffer[i + 1] == 0))

{

buffer[i] = 10;

}

}

return counterBuffer;

}

float AssignFitness(int bits[], float target)

{

int buffer[(int)(BITLENGTH / GENELENGTH)];

int numberOfElements = ParseBits(bits, buffer);

float result = 0.0f;

int i;

for(i = 0; i < numberOfElements - 1; i += 2)

{

switch (buffer[i])

{

case 10:

result += buffer[i + 1];

break;

case 11:

result -= buffer[i + 1];

break;

case 12:

result \*= buffer[i + 1];

break;

case 13:

result /= buffer[i + 1];

break;

default:

break;

}

}

if(debug == 0)

{

printf("\n result %f \n", result);

}

if(result == (float)target)

{

return 999.0f;

}

else

{

return 1/(float)fabs((double)(target - result));

}

}

void PrintChromo(int bits[])

{

printf("I AM PRINTCHROMO!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!");

int buffer[(int)(BITLENGTH / GENELENGTH)];

int numberOfElements = ParseBits(bits, buffer);

int p;

for(p = 0; p < numberOfElements; p++)

{

PrintGeneSymbol(buffer[p]);

}

}

void PrintGeneSymbol(int val)

{

switch(val)

{

case 10:

printf("+");

break;

case 11:

printf("-");

break;

case 12:

printf("\*");

break;

case 13:

printf("/");

break;

default:

printf("%d", val);

break;

}

}

void Crossover(int \* first, int \* second)

{

double random = (double)rand() / (double)RAND\_MAX;

if(random < CROSSOVERRATE)

{

random = (double)rand() / (double)RAND\_MAX;

int crossOver = (int)(random \* BITLENGTH);

int i = 0;

for(i = crossOver; i < BITLENGTH; i++)

{

int temp = first[i];

first[i] = second[i];

second[i] = temp;

}

}

}

void Mutate(int \* doMe)

{

int i = 0;

for(i = 0; i < BITLENGTH; i++)

{

double random = (double)rand() / (double)RAND\_MAX;

if(random < MUTATIONRATE && doMe[i] == 0)

{

doMe[i] = 1;

}

else if(random < MUTATIONRATE && doMe[i] == 1)

{

doMe[i] = 0;

}

}

}

void MakeMeABaby(float totalFitness, double popSize, struct subject \*test, int \* meh)

{

double random = (double)rand() / (double)(RAND\_MAX - 1);

float slice = (float)(random \* totalFitness);

float currentFitness = 0.0f;

int i = 0;

for(i = 0; i < popSize; i++)

{

currentFitness += test[i].fitness;

if(currentFitness >= slice)

{

int j;

if(debug == 0)

{

printf("\nfit parent found\n");

}

for(j = 0; j < BITLENGTH; j++)

{

if(debug == 0)

{

printf("%d", test[i].bits[j]);

}

meh[j] = test[i].bits[j];

}

if(debug == 0)

{

printf("\n");

}

break;

}

else

{

if(debug == 0)

{

printf(" %dUnfit Parent found", i);

}

}

}

}

**Parallel Code**

//Authors: Marcel Englmaier & Michael Sharp

#include <stdio.h>

#include <math.h>

#include <time.h>

#include <stdlib.h>

#include <sys/time.h>

#include <omp.h>

#define BITLENGTH 20

#define GENELENGTH 4

#define MAXGENERATIONS 2

#define CROSSOVERRATE 0.7

#define MUTATIONRATE 0.001

#define debug 1

struct subject

{

int bits[BITLENGTH];

float fitness;

};

struct timeval start;

struct timeval end;

float AssignFitness(int [], float);

int BinToDec(int[]);

int ParseBits(int[] , int \*);

void Crossover(int \*, int \*);

void MakeMeABaby(float, double, struct subject \*, int \*);

void Mutate(int \*);

void PrintChromo(int[]);

void PrintGeneSymbol(int);

int main(int argc, char \*argv[])

{

srand(time(NULL));

double populationSize = atoi(argv[1]);

float target = (float)atoi(argv[2]);

printf("Running with:\n Population Size = %.0f\n Bit Length = %d\n", populationSize, BITLENGTH);

long int i = 0;

int j = 0;

struct subject \*sheep = (struct subject\*)malloc(sizeof(struct subject) \* populationSize);

gettimeofday(&start, NULL);

//make the Adam&Eve generation

for(i = 0; i < populationSize; i++)

{

if(debug == 0)

{

printf("subject %d bits = ", i);

}

for(j = 0; j < BITLENGTH; j++)

{

sheep[i].bits[j] = rand() %2;

if(debug == 0)

{

printf("%d", sheep[i].bits[j]);

}

}

sheep[i].fitness = 0.0f;

if(debug == 0)

{

printf(" with fitness %f\n", sheep[i].fitness);

}

}

int howLongThisTook = 0;

int solutionFound = 0; //0 if false, 1 if true

struct subject \*childSpawn = (struct subject\*)malloc(sizeof(struct subject) \* populationSize);

while(solutionFound != 1)

{

float totalFitness = 0.0f;

# pragma omp parallel for

for(i = 0; i < populationSize; i++)

{

sheep[i].fitness = AssignFitness(sheep[i].bits, target);

if(debug == 0)

{

printf("\n subject %d with fitness %f\n", i, sheep[i].fitness);

}

totalFitness += sheep[i].fitness;

if(debug == 0)

{

printf(" Total Fitness%f\n", totalFitness);

}

}

# pragma omp parallel for

for(i = 0; i < populationSize; i++)

{

if(sheep[i].fitness == 999.0f)

{

printf("It's been found. It took %d generations to find\n", howLongThischildTook);

# pragma omp atomic

solutionFound = 1;

break;

}

}

long childSpawnPopulationSize = 0;

if(solutionFound != 1)

{

printf("Creating New Generation\n");

# pragma omp parallel for

for(childSpawnPopulationSize = 0; childSpawnPopulationSize < populationSize; childSpawnPopulationSize += 2)

{

int childSpawn1Bits[BITLENGTH];

int childSpawn2Bits[BITLENGTH];

MakeMeABaby(totalFitness, populationSize, sheep, childSpawn1Bits);

MakeMeABaby(totalFitness, populationSize, sheep, childSpawn2Bits);

Crossover(childSpawn1Bits, childSpawn2Bits);

Mutate(childSpawn1Bits);

Mutate(childSpawn2Bits);

for(i = 0; i < BITLENGTH; i++)

{

childSpawn[childSpawnPopulationSize].bits[i] = childSpawn1Bits[i];

childSpawn[childSpawnPopulationSize + 1].bits[i] = childSpawn2Bits[i];

}

if(debug == 0)

{

printf("childspawn%d: ", childSpawnPopulationSize);

for(i = 0; i < BITLENGTH; i++)

{

printf("%d", childSpawn[childSpawnPopulationSize].bits[i]);

}

printf("\nchildspawn%d: ",childSpawnPopulationSize + 1);

for(i = 0; i < BITLENGTH; i++)

{

printf("%d", childSpawn[childSpawnPopulationSize + 1].bits[i]);

}

printf("\n");

}

childSpawn[childSpawnPopulationSize].fitness = 0.0f;

childSpawn[childSpawnPopulationSize + 1].fitness = 0.0f;

}

# pragma omp parallel for

for(i = 0; i < populationSize; i++)

{

sheep[i] = childSpawn[i];

}

howLongThisTook++;

printf("New Generation done! Now on Generation %d\n", howLongThisTook);

}

if(howLongThisTook > MAXGENERATIONS)

{

printf("Did not find a solution in maximum allowable runs\n");

solutionFound = 1;

}

}

gettimeofday(&end, NULL);

int timeran = (((end.tv\_sec - start.tv\_sec) \* 1000000) +(end.tv\_usec - start.tv\_usec));

printf("Completed in %d Nano Seconds\n", timeran);

return 0;

}

int BinToDec(int bits[])

{

int val = 0;

int valueToAdd = 1;

int k = 0;

for(k = GENELENGTH - 1; k >= 0; k--)

{

if(debug == 0)

{

printf("k=%d ", bits[k]);

}

if(bits[k] == 1)

{

val += valueToAdd;

}

valueToAdd \*= 2;

}

return val;

}

int ParseBits(int bits[], int\* buffer)

{

int counterBuffer = 0;

int operator = 1;

int currentGene = 0;

int i = 0;

for(i = 0; i < BITLENGTH; i += GENELENGTH)

{

currentGene = BinToDec(&bits[i]);

if(debug == 0)

{

printf("this gene %d \n", currentGene);

}

if(operator == 1)

{

if((currentGene < 10) || (currentGene > 13))

{

continue;

}

else

{

operator = 0;

buffer[counterBuffer++] = currentGene;

continue;

}

}

else

{

if (currentGene > 9)

{

continue;

}

else

{

operator = 1;

buffer[counterBuffer++] = currentGene;

continue;

}

}

}

for(i = 0; i < counterBuffer; i++)

{

if((buffer[i] == 13) && (buffer[i + 1] == 0))

{

buffer[i] = 10;

}

}

return counterBuffer;

}

float AssignFitness(int bits[], float target)

{

int buffer[(int)(BITLENGTH / GENELENGTH)];

int numberOfElements = ParseBits(bits, buffer);

float result = 0.0f;

int i;

for(i = 0; i < numberOfElements - 1; i += 2)

{

switch (buffer[i])

{

case 10:

result += buffer[i + 1];

break;

case 11:

result -= buffer[i + 1];

break;

case 12:

result \*= buffer[i + 1];

break;

case 13:

result /= buffer[i + 1];

break;

default:

break;

}

}

if(debug == 0)

{

printf("\n result %f \n", result);

}

if(result == (float)target)

{

return 999.0f;

}

else

{

return 1/(float)fabs((double)(target - result));

}

}

void PrintChromo(int bits[])

{

printf("I AM PRINTCHROMO!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!");

int buffer[(int)(BITLENGTH / GENELENGTH)];

int numberOfElements = ParseBits(bits, buffer);

int p;

for(p = 0; p < numberOfElements; p++)

{

PrintGeneSymbol(buffer[p]);

}

}

void PrintGeneSymbol(int val)

{

switch(val)

{

case 10:

printf("+");

break;

case 11:

printf("-");

break;

case 12:

printf("\*");

break;

case 13:

printf("/");

break;

default:

printf("%d", val);

break;

}

}

void Crossover(int \* first, int \* second)

{

double random = (double)rand() / (double)RAND\_MAX;

if(random < CROSSOVERRATE)

{

random = (double)rand() / (double)RAND\_MAX;

int crossOver = (int)(random \* BITLENGTH);

int i = 0;

for(i = crossOver; i < BITLENGTH; i++)

{

int temp = second[i];

first[i] = second[i];

second[i] = temp;

}

}

}

void Mutate(int \* doMe)

{

int i = 0;

for(i = 0; i < BITLENGTH; i++)

{

double random = (double)rand() / (double)RAND\_MAX;

if(random < MUTATIONRATE && doMe[i] == 0)

{

doMe[i] = 1;

}

else if(random < MUTATIONRATE && doMe[i] == 1)

{

doMe[i] = 0;

}

}

}

void MakeMeABaby(float totalFitness, double popSize, struct subject \*test, int \* meh)

{

double random = (double)rand() / (double)(RAND\_MAX - 1);

float slice = (float)(random \* totalFitness);

float currentFitness = 0.0f;

int i = 0;

for(i = 0; i < popSize; i++)

{

currentFitness += test[i].fitness;

if(currentFitness >= slice)

{

int j;

if(debug == 0)

{

printf("\nfit parent found\n");

}

for(j = 0; j < BITLENGTH; j++)

{

if(debug == 0)

{

printf("%d", test[i].bits[j]);

}

meh[j] = test[i].bits[j];

}

if(debug == 0)

{

printf("\n");

}

break;

}

else

{

if(debug == 0)

{

printf(" %dUnfit Parent found", i);

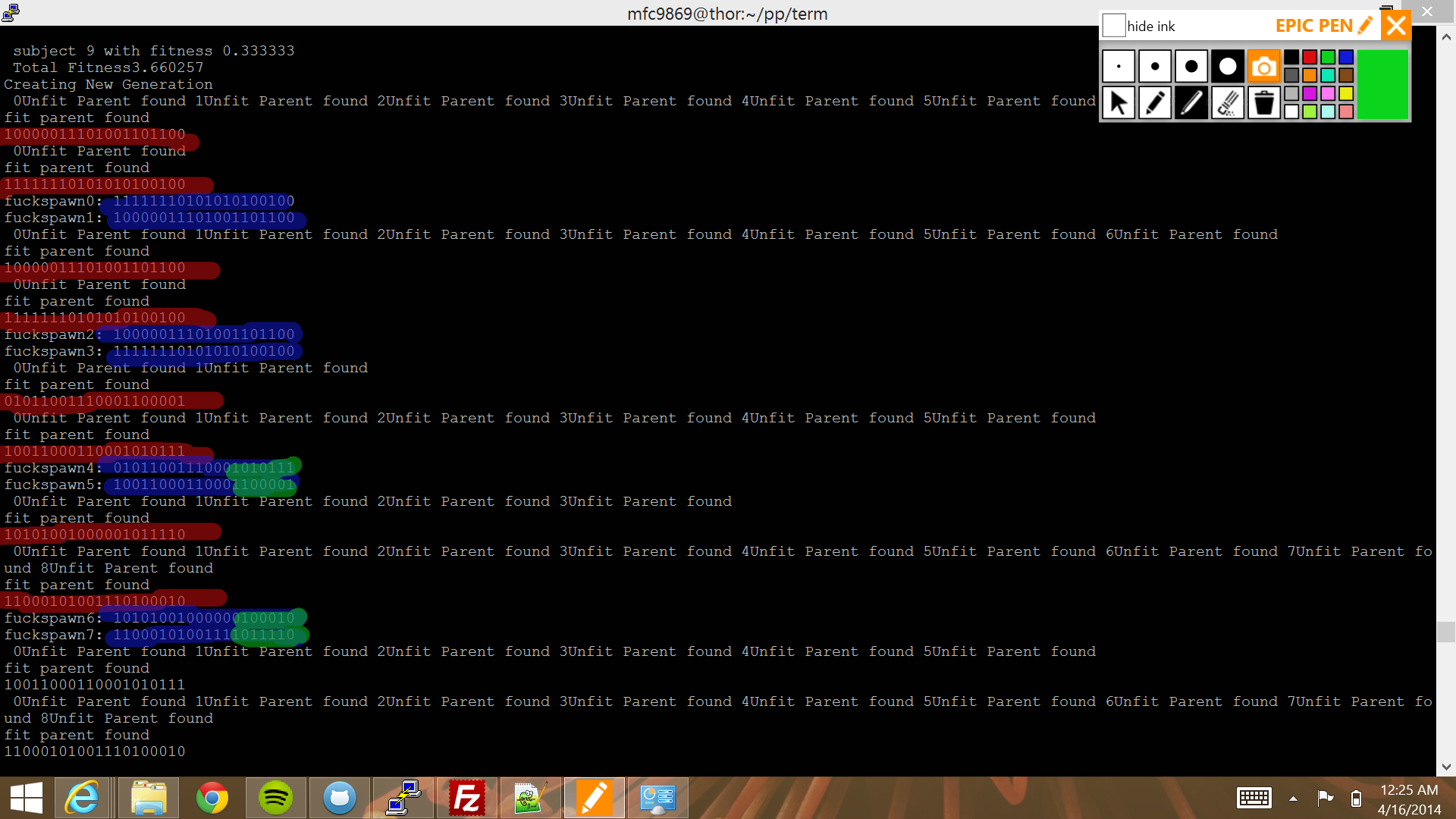
}

}

}

}

**Out Put**

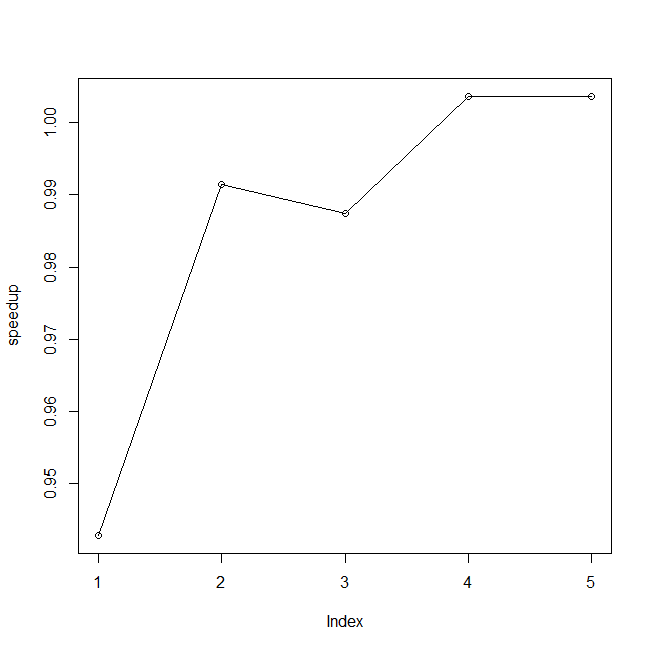
****

The above image shows parents being selected in red and the children in blue. The green is where the crossover is taking place.

**Results**

|  |  |  |
| --- | --- | --- |
| **Subjects** | **Sequentical Nano Seconds** | **Parallel Nano Seconds** |
| **10** | **33** | **35** |
| **100** | **348** | **351** |
| **1k** | **11109** | **11250** |
| **10k** | **977719** | **974163** |
| **100k** | **95578030** | **95233035** |

On average up to 1k, sequential was faster. After that, parallel is faster. Since there was a ton of Overhead and we’re only running on 16 cores on 1 node

****

The above graph displays how the speed of the program was effected by using parallel programing.

**References**

Obitko. Mark (1998). *Introduction to genetic algorithms*. Retrieved from http://www.obitko.com/tutorials/genetic-algorithms/ga-basic-description.php

The University of Kansas Medical Center. (2014). *Determining genetic risk, what is the hardy-weinberg equation?*. Retrieved from http://www2.kumc.edu/genetics/risk/hardy.html