Giovanni Matthew DiSalvo
May 14, 2024
Professor Siewert
CSCI 551 Numerical and Parallel Programming

Final Program Report

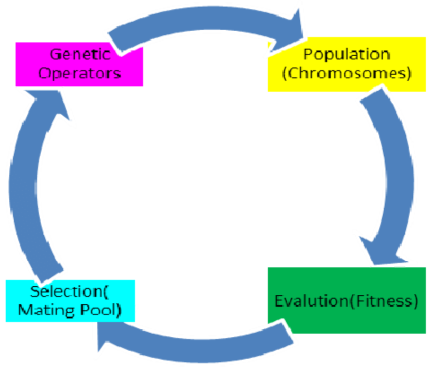
Explanation of my Algorithm

Hello Professor Siewert! Hope this report finds you well, as I begin today I will be discussing my final parallel program, which was based on the genetic algorithm which I attempt to speed up using OpenMP (I tried pthreads too!). My genetic algorithm's goal was to find a target string. It does so by evolving a population of strings (which in my case was always 100) which are constructed by genes, which are just the possible characters that could occupy a given index in a string. Initially we start with 100 random gnomes constructed with random genes in our population which are ranked by their fitness level.

The reason we can calculate the fitness level is because our algorithm is given the target string beforehand. To calculate the fitness level of a gnome, we simply subtract correct incorrect characters - correct characters. For instance fitness level of the following:

Target: Elephant **Gnome:** 4lephunt **Fitness Score:** 2
Target: Elephant **Gnome:** Elephant **Fitness Score:** 0

The purpose of the fitness score is to deem how accurate the gnome is in comparison to the target. The goal of the algorithm is to mutate the initial gnomes and following generations of gnomes, enough to correctly form the target string (get fitness score of 0). The algorithm will then rank the generation of gnomes and sort them based on their fitness score. My algorithm then allows the top 10% of the most fit (the ten with the lowest fitness scores of the 100 population) gnomes into the next generation, where the gnomes are then left to mutate/mate to fill the remaining 90 slots of the population. Mutation is done by stealing characters from two randomly selected parents, in which my program has a 45% chance to steal a character from parent 1, 45% for parent 2, and a 10% chance that we just completely get a random gene to steal a character from. We do this until we hit our target size, and we continue this loop until we eventually reach our target string. We continue to use numerical methods such as the fitness evaluation, selection, and mutation which are all different forms of string manipulation and transformations.



Here is an image I found online that puts it in really simple terms, but demonstrates the cycle and loop we go through. (credit: https://www.researchgate.net/figure/Genetic-Algorithm-Cycle_fig1_311519179)

Why?

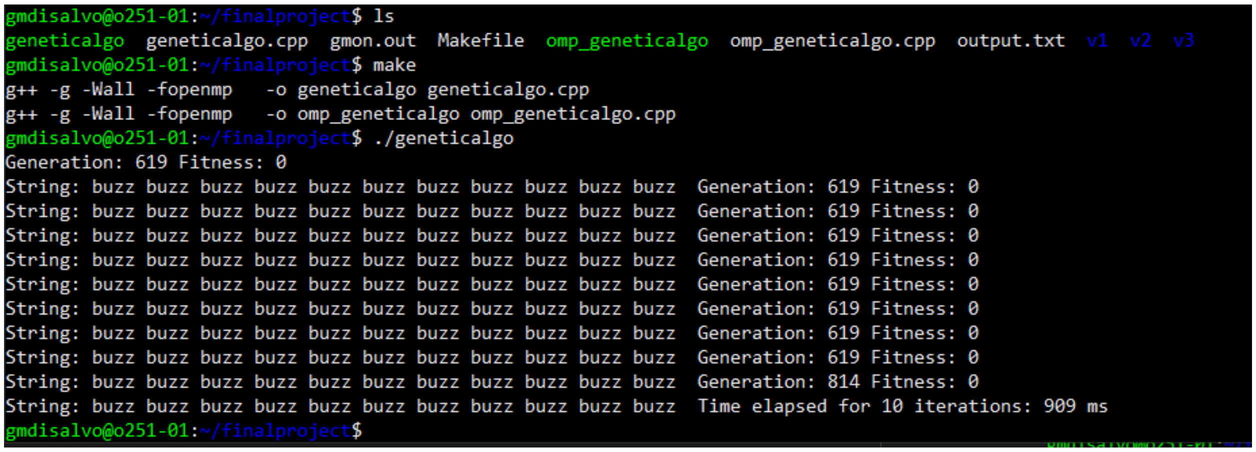
The algorithm is important in terms of versatility because it can be used in search problems that not only are non-linear but those that are non-differentiable or discontinuous functions. This makes them applicable in fields such as finance, biology, and of course computer science. On a more usable scale, these algorithms can be used for applications such as drug discovery/development, disease diagnosis/prognosis, genetic analysis which can give us insight on cancer and cardiovascular diseases.

Sequential

The sequential code is a modified version of the code used on Geeks for Geeks website (https://www.geeksforgeeks.org/genetic-algorithms/). For the sequential version, I got rid of a lot of the warnings, added a posix clock timers, tuned some of the percentages, added our posix clock_gettime functions, more comments, made it iterate over the algorithm 10 times to get more computation time in our program, changed output, and changed the target string dependent on my tests.

Build and Run:

Target for the program during the runs: “buzz buzz buzz buzz buzz buzz buzz buzz buzz buzz buzz”



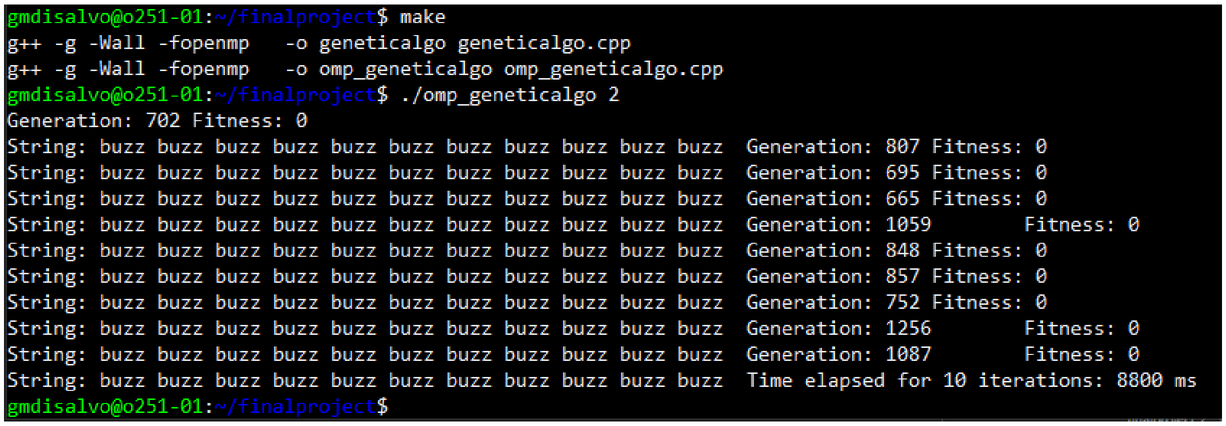
```
gmdisalvo@o251-01:~/finalproject$ ls
geneticalgo  geneticalgo.cpp  gmon.out  Makefile  omp_geneticalgo  omp_geneticalgo.cpp  output.txt  v1  v2  v3
gmdisalvo@o251-01:~/finalproject$ make
g++ -g -Wall -fopenmp   -o geneticalgo geneticalgo.cpp
g++ -g -Wall -fopenmp   -o omp_geneticalgo omp_geneticalgo.cpp
gmdisalvo@o251-01:~/finalproject$ ./geneticalgo
Generation: 619 Fitness: 0
String: buzz buzz buzz buzz buzz buzz buzz buzz buzz buzz buzz  Generation: 619 Fitness: 0
String: buzz buzz buzz buzz buzz buzz buzz buzz buzz buzz buzz  Generation: 619 Fitness: 0
String: buzz buzz buzz buzz buzz buzz buzz buzz buzz buzz buzz  Generation: 619 Fitness: 0
String: buzz buzz buzz buzz buzz buzz buzz buzz buzz buzz buzz  Generation: 619 Fitness: 0
String: buzz buzz buzz buzz buzz buzz buzz buzz buzz buzz buzz  Generation: 619 Fitness: 0
String: buzz buzz buzz buzz buzz buzz buzz buzz buzz buzz buzz  Generation: 619 Fitness: 0
String: buzz buzz buzz buzz buzz buzz buzz buzz buzz buzz buzz  Generation: 619 Fitness: 0
String: buzz buzz buzz buzz buzz buzz buzz buzz buzz buzz buzz  Generation: 619 Fitness: 0
String: buzz buzz buzz buzz buzz buzz buzz buzz buzz buzz buzz  Generation: 814 Fitness: 0
String: buzz buzz buzz buzz buzz buzz buzz buzz buzz buzz buzz  Time elapsed for 10 iterations: 909 ms
gmdisalvo@o251-01:~/finalproject$
```

The field following “String:" is the string that is deemed the most fit at the end of the algorithm, which is going to match our target string if we are successful, and our accuracy remains great. The field following “Generation:” is our generation that iteration of the algorithm ended before it reached a fitness score of 0. The last field is just our fitness score and our time for all 10 iterations is at the end.

Average run time over three runs: 909 + 970 + 951 / 3 = 943.3 ms

<u>Parallel</u>

Here I experienced a ton of slowdown, which I get further into in my video. Although here it is building and running. I could only get it working with 2 and 4 threads. The parameter for the program is just the number of threads.



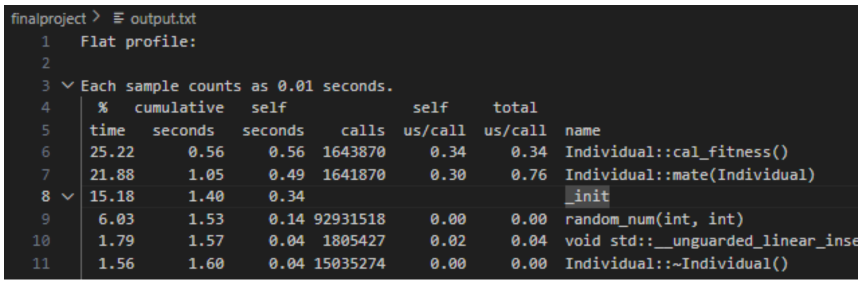
```
gmdisalvo@o251-01:~/finalproject$ make
g++ -g -Wall -fopenmp   -o geneticalgo geneticalgo.cpp
g++ -g -Wall -fopenmp   -o omp_geneticalgo omp_geneticalgo.cpp
gmdisalvo@o251-01:~/finalproject$ ./omp_geneticalgo 2
Generation: 702 Fitness: 0
String: buzz buzz buzz buzz buzz buzz buzz buzz buzz buzz buzz  Generation: 807 Fitness: 0
String: buzz buzz buzz buzz buzz buzz buzz buzz buzz buzz buzz  Generation: 695 Fitness: 0
String: buzz buzz buzz buzz buzz buzz buzz buzz buzz buzz buzz  Generation: 665 Fitness: 0
String: buzz buzz buzz buzz buzz buzz buzz buzz buzz buzz buzz  Generation: 1059        Fitness: 0
String: buzz buzz buzz buzz buzz buzz buzz buzz buzz buzz buzz  Generation: 848 Fitness: 0
String: buzz buzz buzz buzz buzz buzz buzz buzz buzz buzz buzz  Generation: 857 Fitness: 0
String: buzz buzz buzz buzz buzz buzz buzz buzz buzz buzz buzz  Generation: 752 Fitness: 0
String: buzz buzz buzz buzz buzz buzz buzz buzz buzz buzz buzz  Generation: 1256        Fitness: 0
String: buzz buzz buzz buzz buzz buzz buzz buzz buzz buzz buzz  Generation: 1087        Fitness: 0
String: buzz buzz buzz buzz buzz buzz buzz buzz buzz buzz buzz  Time elapsed for 10 iterations: 8800 ms
gmdisalvo@o251-01:~/finalproject$
```

Average run time over three runs (2 threads): 8800 + 7734 + 8391 / 3 = 8308.3 ms

Average run time over three runs (4 threads): 7572 + 7713 + 7642 / 3 = 7642.3 ms

<u>Speed up analysis</u>

Here I used gprof to find the heavy hitting functions on my sequential code.



```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  us/call  us/call  name
 25.22      0.56     0.56  1643870     0.34     0.34  Individual::cal_fitness()
 21.88      1.05     0.49  1641870     0.30     0.76  Individual::mate(Individual)
 15.18      1.40     0.34                             _init
  6.03      1.53     0.14 92931518     0.00     0.00  random_num(int, int)
  1.79      1.57     0.04  1805427     0.02     0.04  void std::__unguarded_linear_inse
  1.56      1.60     0.04 15035274     0.00     0.00  Individual::~Individual()
```

The two functions I made parallel in my final version were cal_fitness and mate so our program should now be 25.22 + 21.88 = 47.1% parallel, while our sequential is 52.9%.

P = 0.47 S = 4

Ideal Speed Up (Amdahl's Law): S = 1 / ((1 - 0.47) + (0.47 / 4)) = 1.54x

The speed up we can expect is 1.54x

Although we know this isn't true for me since I couldn't mine fully functioning as my program experienced much more slowdown.

Actual Speed Up: S = 943.3 / 7642.3 = 0.12x

The speed down that my OpenMP version was DRAMATIC and we can tell that something is incredibly wrong, we are experiencing a tremendous slow down. We aren't near anywhere our theoretical speed up should be due to my implementation.

I know I could've made this embarrassingly parallel by setting up a synthetic workload and make it a benchmark but just looping the algorithm sequentially and using OpenMP, but I really wanted to tackle the problem of making the algorithm itself faster.