High Performance Computing- Fall 2017

Final Project

December 4, 2017

Requirement for the satisfaction of completing
High Performance Computing, Fall 2017

Topic: Optimization of Genetic Algorithms Using OpenMP for Shared Memory Computing

By: Ogor Ifuwe and Kate Schoenberger

A Brief Introduction to Genetic Algorithms

Genetic Algorithm (GA) is a search-based optimization technique, that is based on the principles of Genetics and Evolution. In computer Science, this technique has been adopted for use in finding the optimal or near optimal (maximum or minimum) values of a function to difficult problems which otherwise would take an infinite amount of time to solve.

History and Overview of Genetic Algorithms

It is an established fact that nature has always been a great source of inspiration to all mankind, genetic algorithms are just another proof of this fact. Genetic algorithms are a subset of Evolutionary Computation. They were developed by John Holland, his students and colleagues most notably David E. Goldberg at the University of Michigan in 1975. Since then, there has been a wide spread of the successful use of this concept in optimization.

In genetic algorithms, we have a population of possible solutions to the given problem. These solutions then undergo a selection, crossover and mutation process like in natural genetics for reproducing new offspring. The process is repeated for a set number of generations (iterations). Each candidate solution is assigned a fitness value based on an objective function value. Selection for mating is solely based of the fitness value of the candidates, owing to the principle of Darwinian Theory of Survival of the fittest. As candidates with the highest fitness value are most likely to reproduce individuals with a higher fitness value, this way we keep evolving better individuals or solutions over generations(iterations), until we reach a stopping criterion.

Also, genetic algorithms are sufficiently randomized, however, they perform much better than random local search algorithms (in which we just try various random solutions, keeping track of the best so far), as they take advantage of historical information as well.

Motivation for choosing a Genetic Algorithm

In Computer Science, there is quite a number of problems that are NP-Hard, that is problems that take a very long time to solve even with the most powerful computer systems. Genetic algorithms have experimentally proven to be able to give a good-enough solution fast-enough. Parallelization of genetic algorithms by implementing OpenMP seemed like an interesting project to work on with the hope of helping deliver these solutions in an even shorter amount of time.

For example, the Travelling Salesman Problem, have real-world applications like VLSI Design and path finding. Imagine using a GPS Navigation system that takes longer than expected to compute the optimal path from source to destination. Such delays in real world applications are not acceptable. A good-enough solution which is delivered fast will be required. Also, since Genetic Algorithms quickly converge towards a desired behavior to solve an original problem, with a good enough solution that is not necessarily the optimal solution, they are highly beneficial to Financial Analysts, as a once found optimal solution in unlikely to stay the best one with future development in a specific market space.

Basic Terminologies of Genetic Algorithms

For a good understanding of the discussion of genetic algorithms in detail, a basic familiarity with some Evolutionary terminologies used is required.

- **Population**: Population is the subset of all the possible (encoded) solutions to the given problem. For example, this is basically a representation of the population of human beings in a mating pool, except they are conceptually represented by these candidate solutions.

- **Chromosomes**: A chromosome is one such solution to the given problem which is initialized in the program as chrome.

- **Gene**: A gene is one element position of a chrome in sign bit

Structure of the Genetic Algorithm

The genetic algorithm starts with a randomly generated population called chromosomes represented in a 4*6 matrix array in bits, which are evaluated based on an objective function (y = -x*x + 5), which we used as the Fitness value. See table below for illustration.

S/N	Sb	b5	b4	b3	b2	b1	Fit
1	0	1	0	1	1	0	8
2	1	0	0	1	0	0	6
3	0	0	1	0	1	0	9
4	0	0	0	1	1	0	12

Each row is a chromosome, that consists of 6 genes (represented by columns 1 through 6 in 1 sign bit and 5 other bits), and a fitness value, (Fit, represented by the 7 column) of the function y(x) at the value of the given chrome(x), where the chrome is converted to an integer value from the bits conversion.

Adopting Darwian Theory of Survival of the fittest from Evolution Theory, we select the best candidates from our population by implementing a fitness based selection approach choosing chromes with the highest fitness values using bubble sort. Selecting the best chromes helps ensure we get the optimal value of the function. Following our illustration, our candidate solutions will be as below;

S/N	sb	b5	b4	b3	b2	b1	Fit
4	0	0	0	1	1	0	12
3	0	0	1	0	1	0	9

Crossover was implemented using One Point Crossover, where a fixed point is chosen and the bits of one parent is uniformly exchanged with those of the other parents to form two new children as seen in the figure below, with crossover done starting at b3 to form child 1 and child 2

S/N	sb	b5	b4	b3	b2	b1	Fit
4	0	0	0	1	1	0	12
3	0	0	1	0	1	0	9

S/N	sb	b5	b4	b3	b2	b1	Fit
Child 1	0	0	0	0	1	0	14
Child 2	0	0	1	1	1	0	5

Afterwards, Mutation is done with a low probability in one chrome in the set by simply inverting one of the bits in the chrome.

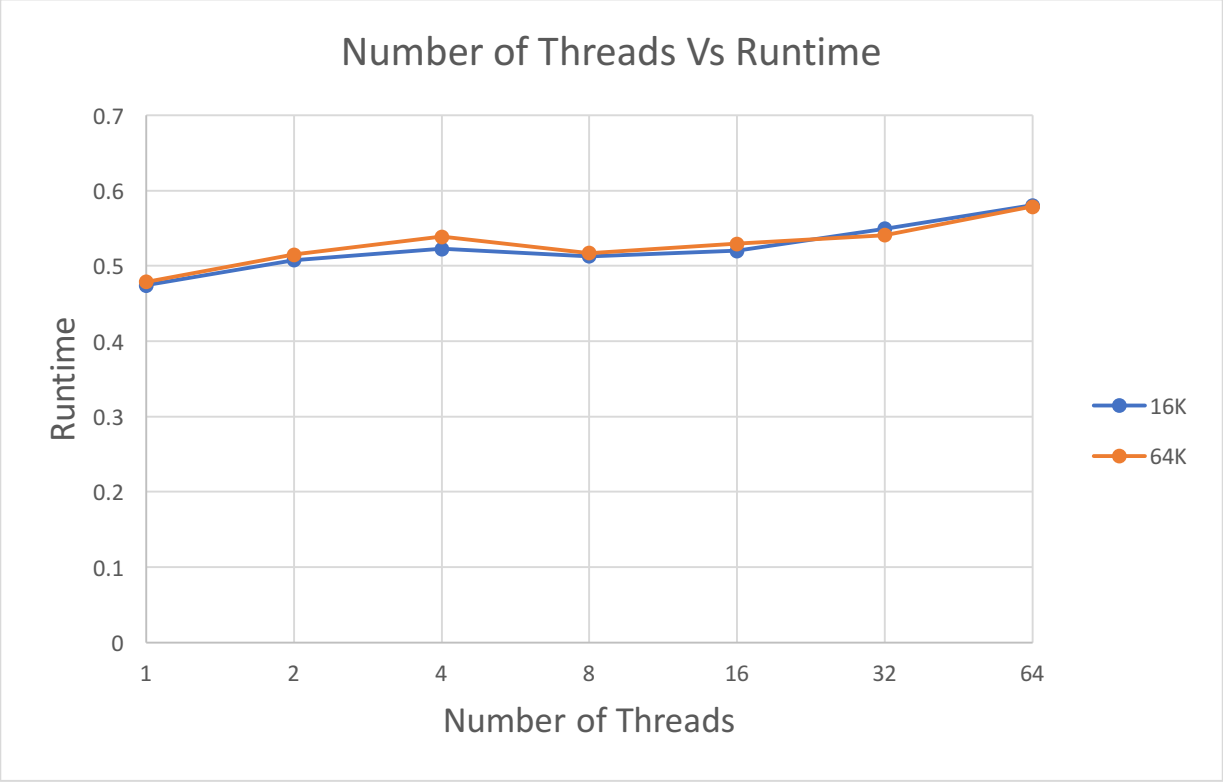
After sufficient iterations, this approach of selection, crossover and mutation recombination leads to candidates that are more and more suited to solve the given problem. The termination condition of the algorithm is usually a maximum number of iterations which is the number of generations considered appropriate depending on the population size and given problem. Computing resource plays a big role here, hence optimization becomes extremely important.

The genetic algorithm procedure can be summarized as follow;

- Initialize a random population of chromosomes
- Evaluate the fitness value of each chromosome in terms of an objective function
- Select chromosomes for recombination
- Perform crossover and mutation
- Evaluate the fitness value of the new children
- Repeat selection, step 3 through 5 until the termination conditioned is satisfied

Performance Metrics and Benchmarks Plots

NUMBER OF THREADS	RUNTIME FOR ITERATION = 16K	RUNTIME FOR ITERATION = 64K
1	0.474285	0.4748773
2	0.507608	0.515033
4	0.52269	0.538681
8	0.512917	0.517113
16	0.520085	0.529441
32	0.54932	0.541095
64	0.58051	0.578875

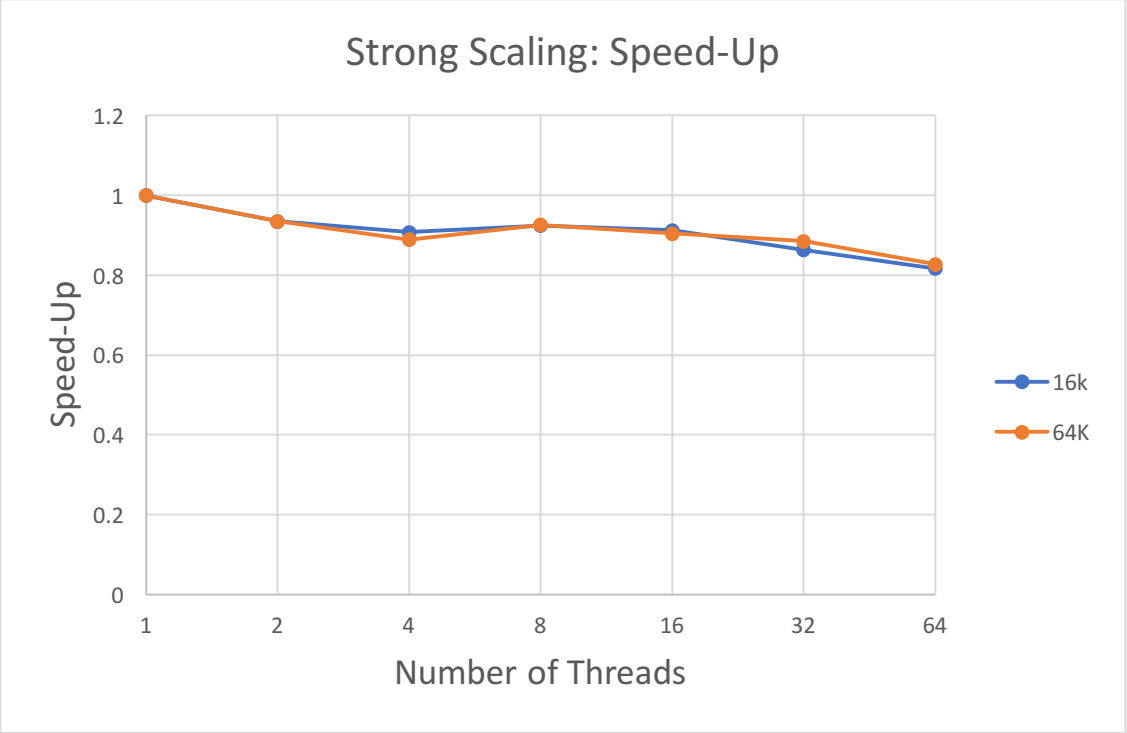


Plotting the runtime values gotten from running the above benchmark on a KNL node on stampede2, we see that the algorithm starts out with a linear performance boost for both 16K and 64K iterations. Using 4 threads, we get

a drop in the performance that reverts towards an upward trend with an increment in the number of threads to 8. This trend continues through 16, 32 and 64 threads respectively. This denotes a slight increase or boost in performance relative to an increase in the number of threads. Also, we can see from the graph that we start with at about 47 percent for one thread and end at about 64 percent.

Strong Scaling: Speed-Up

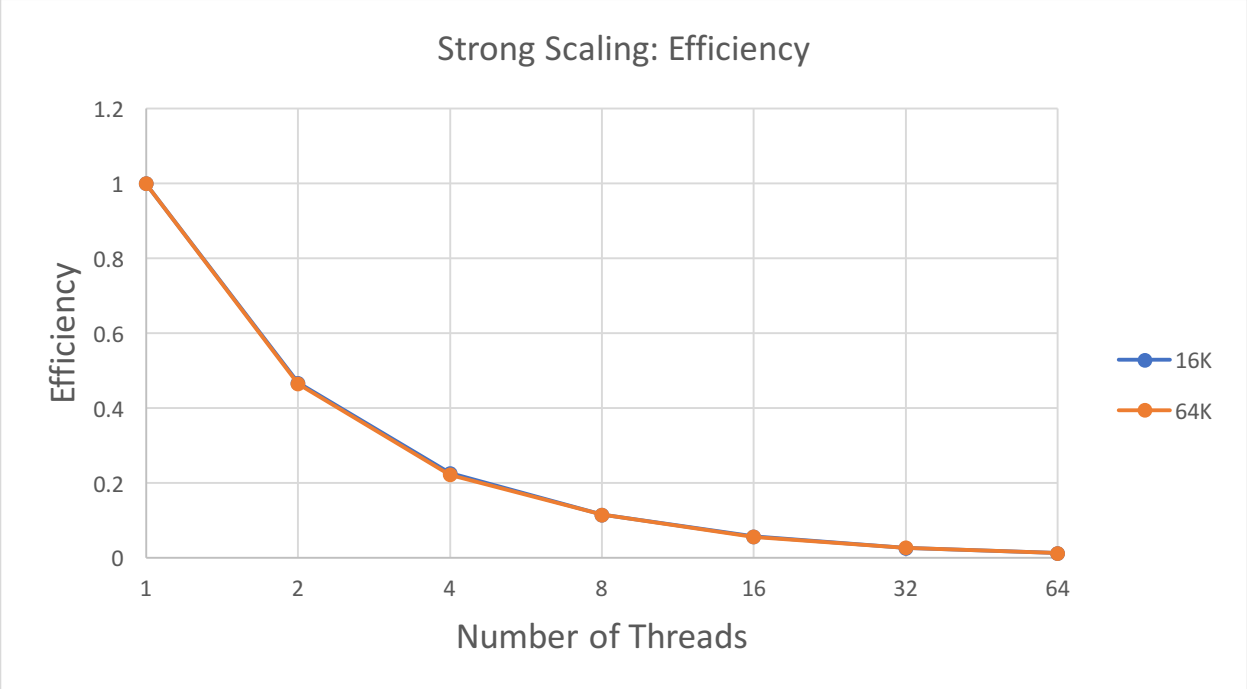
NUMBER OF THREADS	RUNTIME FOR ITERATIONS = 16K	RUNTIME FOR ITERATIONS = 64K
1	1	1
2	0.93435289	0.934352829
4	0.90739253	0.88878761
8	0.92468177	0.9258576
16	0.91193747	0.90429906
32	0.86340384	0.88482244
64	0.81701435	0.82707493



Analyzing the performance of the algorithm in parallel using Speed-Up for Strong scaling, we start at exactly 100 percent and immediately decline to about 93 percent. Incrementing the number of threads to 2 further declines the performance to about 89 percent. We observe a slight boost in speed-up with an increment to 4 threads as we incline to about 93 percent. A further increment in the number of threads, has no relative significance in the performance boost of the algorithm as we see a slight decline through 16, 32 to 64 threads that bring the performance down to about 64 percent for both 16k and 64K.

Strong Scaling: Efficiency

NUMBER OF THREADS	RUNTIME FOR ITERATIONS = 16K	RUNTIME FOR ITERATIONS = 64K
1	1	1
2	0.467176443	0.464798372
4	0.226848131	0.222196903
8	0.115585221	0.115732199
16	0.056996091	0.056518691
32	0.02698137	0.027650701
64	0.012765849	0.012923045



Analyzing the performance of the algorithm in parallel using Efficiency for strong scaling, we start at exactly 100 percent and drop to about 47 percent from 1 thread to 2 for both 16K and 64K. Increasing the number of threads from 2 to 4 does not increase the efficiency, rather falls further to about 22 percent and 12 percent, 6 percent, 3 percent and lastly 1 percent efficiency for 8, 16, 32 and 64 threads respectively.

Summary of Parallelization and Optimization Implemented and Lessons Learned

We implemented parallelization by implementing OpenMP for shared memory computing.
The observation from the effort was even though we expected a significant boost in performance by implementing parallelization, this was not the case for this algorithm as seen from our Strong Scaling analysis with Speed-up and Efficiency. Using Gprof to profile our program, we found out that we were spending 100 percent of the in our selection function. Originally, we were performing the selection function, using bubble sort algorithm, which takes n*n time (i.e relatively computationally expensive). Deciding to refactor the program to use quick sort instead, we experienced a performance boost by a factor of *5 faster, this is because quick sort takes nlogn time.

Parallelization of a sorting algorithm would have taken up a significant amount of time to do, which was not part of this project, we decided to switch to the quick sort algorithm, which reduced our time on Gprof to about 76 percent. Also, we know computers are not great at performing memory based operations like reading and writing data, rather they are great at computing numbers, thus the parallelization of thus parallelization of the bubble sort algorithm might not have given us the expected performance boost since owing to the fact that the threads would not have enough computation to perform and will spend most time communicating with each other and afterwards converging, by so doing increasing the overhead since genetic algorithms are solely based on a selection function that keep track of historical data(fitness value) which it uses to make selection for crossover and mutation.

In addition, we got a significant boost in performance by specifically asking the hardware to optimize the program during runtime using a command Dr. Stone illustrated. This was very helpful as we experienced a runtime that was faster by a factor of *5.

Conclusion

Taking Amdahl's law into account, we can see that from our speed-up and even Efficiency graphs, increasing the number of threads for a fixed amount of work leads to a diminishing return, as such we experienced a continuous decline in performance with a relative continuous increment in the number of threads that would have supposedly given us a performance boost.

Also, we learned that before implementing parallelization, the time it takes to run an algorithm serially should be taken into account and compared against a reasonable predictive time, it will take to achieve parallelization with a likelihood of the performance output expected. Afterwards, a decision can be made as it relates to the gains from the efforts required to make the algorithm/program parallelized. Since the time to achieve a parallelized version is not only the runtime in parallel but also the time it takes to write the code as well.

Final Project's BitBucket Code Repository

References

https://www.sciencedirect.com/science/article/pii/S1877050912002645

https://www.investopedia.com/articles/financial-theory/11/using-genetic-algorithms-forecast-financial-markets.asp

https://www.tutorialspoint.com/genetic_algorithms/

https://www.codeproject.com/KB/cpp/Genetic_Algorithm_in_C.aspx