PARALLELIZED GENETIC ALGORITHM

A REPORT ON PACKAGE SUBMITTED BY

ASHWATH P (18PT06)

BALAGURU M (18PT07)

SUBJECT: OPERATING SYSTEMS

DEPARTMENT OF APPLIED MATHEMATICS AND COMPUTATIONAL SCIENCES, PSG COLLEGE OF TECHNOLOGY,COIMBATORE - 641004

TABLE OF CONTENTS

PARALLELIZED GENETIC ALGORITHM

Abstract

The report aims to give the readers some essential information about open multiprogramming and the concepts involved in it. This report will also explain about the tools and technologies used in the implementation of openMP, system calls used in it. Precisely, we will understand the parallel implementation of Genetic Algorithm with openMP and also the performance analysis between parallel and serial implementation.

1.1 Introduction

Openmp is an API for Writing Multithreaded Applications. It is a set of compiler directives and library routines for parallel application programmers. It greatly simplifies writing multi-threaded (MT) programs in Fortran, C and C++. Openmp is designed for multi-processor/core, shared memory machines. Openmp programs accomplish parallelism exclusively through the use of threads. Openmp provides explicit (not automatic) parallelism, offering the programmer full control over parallelization. Now we will discuss the Genetic algorithm in detail.

1.2 Description

1.2.1 Algorithm workflow

Genetic algorithms are structured around the natural process of biological evolution. They are adaptive informed search based algorithms that use a fitness function as its heuristic in order to

find a given target. Each individual at each generation in a population is composed of two parts: its data, which we will refer to as DNA, and a fitness value. The closer the DNA is to the desired target DNA, the better. This is depicted by a lower fitness level, zero being the target's. These are intelligent exploitation of random search provided with historical data to direct the search into the region of better performance in solution space. They are commonly used to generate high-quality solutions for optimization problems and search problems.
Genetic algorithms simulate the process of natural selection which means those species who can adapt to changes in their environment are able to survive and reproduce and go to the next generation. In simple words, they simulate "survival of the fittest" among individuals of consecutive generations for solving a problem. Each generation consists of a population of individuals and each individual represents a point in search space and possible solution. Eachindividual is represented as a string of character/integer/float/bits. This string is analogous to the Chromosome.

Genetic algorithms are based on an analogy with genetic structure and behavior of chromosomes of the population. Following is the foundation of GAs based on this analogy –

1. Individual in population compete for resources and mate
2. Those individuals who are successful (fittest) then mate to create more offspring than others
3. Genes from "fittest" parents propagate throughout the generation, that is sometimes parents create offspring which is better than either parent.
4. Thus each successive generation is more suited for their environment.

1.2.2 Terminologies

The important terminologies of the genetic algorithm are as follows,

1. **Search space -** The population of individuals are maintained within search space. Each individual represents a solution in the search space for a given problem. Each individual is coded as a finite length vector (analogous to chromosome) of components. These variable components are analogous to Genes. Thus a chromosome (individual) is composed of several genes (variable components) as in Figure 1.1

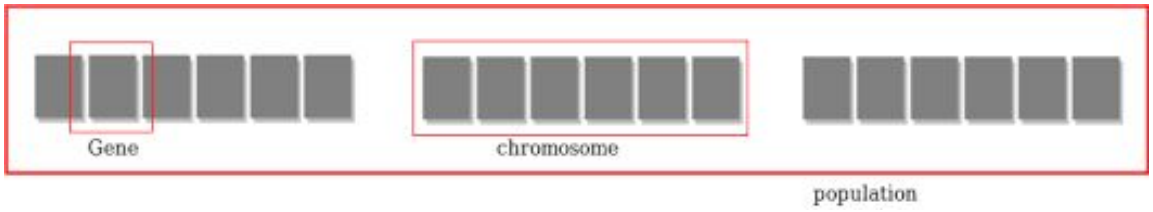

Figure 1.1 Search space of Genetic Algorithm

2. **Selection -** Select the individuals [parents], that will be used to create a new candidate for the next generation. Randomized, but prioritizing the best candidates in the population [genepool].
3. **Crossover -** Combine two parents to form children for the next generation as in Figure 1.2.

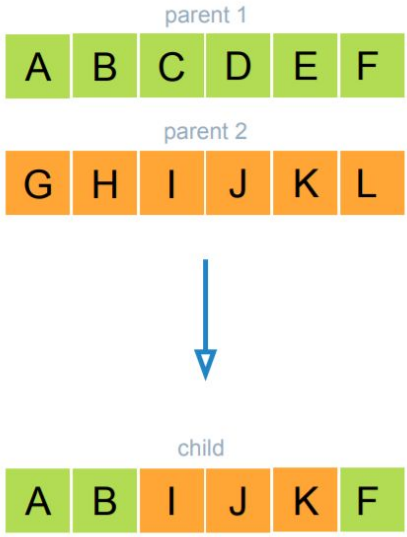

Figure 1.2 Crossover operation in Genetic Algorithm

4. **Mutation -** Apply random changes to the resulting child as in Figure 1.3. This is to prevent the loss of potentially relevant data.

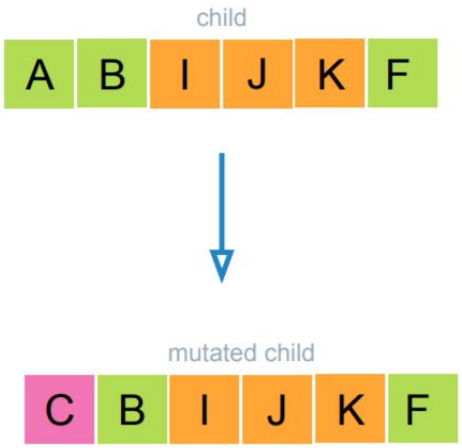

Figure 1.3 Mutation operation in Genetic Algorithm

5. **Fitness score -** A Fitness Score is given to each individual which shows the ability of an individual to "compete". The individual having optimal fitness scores (or near optimal) are sought.

1.3 Serial Algorithm

Genetic algorithms were initially a foreign concept, and as such, we needed to get familiar with them before we could begin parallelizing them. Our approach was to implement a simple genetic algorithm sequentially with a familiar language like C/C++. Once implemented, we looked for ways to modify the algorithm. More specifically, the processes of how the parents are selected, how the crossover is done, and the mutation. The goal of this sequential implementation was to minimize the number of generations required to find the target. The initial implementation took ~3000 generations, using "Hello World" as target, which we were eventually able to bring down to ~1000. Once we were satisfied with the serial C/C++ code, we implemented it parallely using openMP**.** The following is the serial implementation of traditional genetic algorithm,

```
BEGIN /* genetic algorithm */
        randomly generate the initial population
        compute fitness of each individual
        WHILE NOT finished DO
        BEGIN /* produce new generation */
                FOR population_size / 2 DO
                BEGIN /* reproductive cycle */
                        select two individuals from old generation for mating
                                /* biassed in favour of the fitter ones */
                        recombine the two individuals to give two offspring
                        compute fitness of the two offspring
                        insert offspring in new generation
                END
                IF population has converged THEN
                        finished = TRUE
        END
END
```

The Figure 1.4 clearly depicts the workflow of the traditional genetic algorithm as a pictorial view.

Figure 1.4 Traditional Genetic Algorithm

1.4 Parallel Algorithm

There are multiple ways in which genetic algorithms can be parallelized. In this project, we implemented two methods: Method A and Method B. Both methods were implemented in C/C++. Below is a general workflow of the two methods.

1.4.1 Method A

Figure 1.5 Method A

This method mainly focussed on parallelizing the selection phase of the algorithm as depicted in Figure 1.5(the dotted red lines is the parallel portion). As depicted in Figure 1.6, all the threads would share the same genepool. There are only two threads shown in Figure 1.6 for simplicity, but this could be any number of threads in reality. Each thread would independently select two parents from the common genepool and would perform the crossover and mutation steps separately. Once all the threads had created their own version of the child, they would all be compared. The best one would eventually continue down the pipeline of the algorithm as it would sequentially.

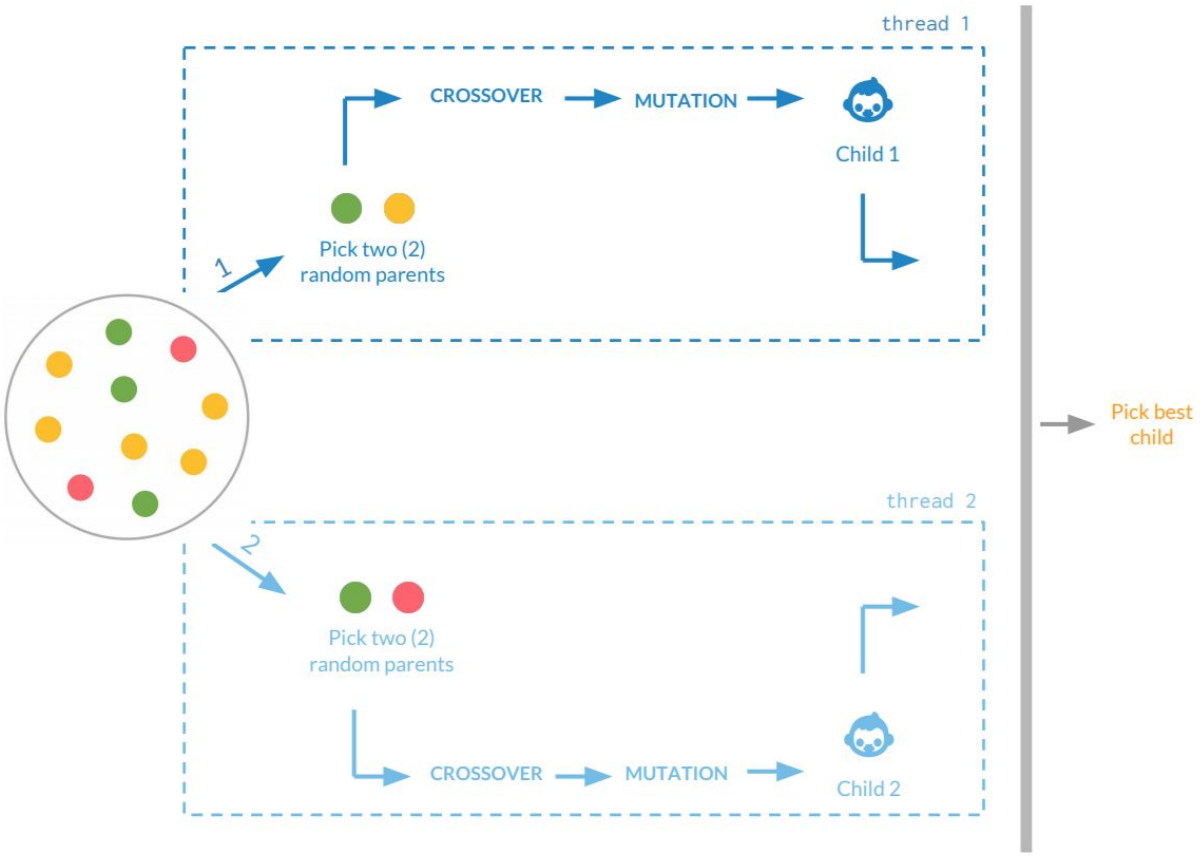

Figure 1.6 Workflow of Parallel Portion in Method A

1.4.2 Method B

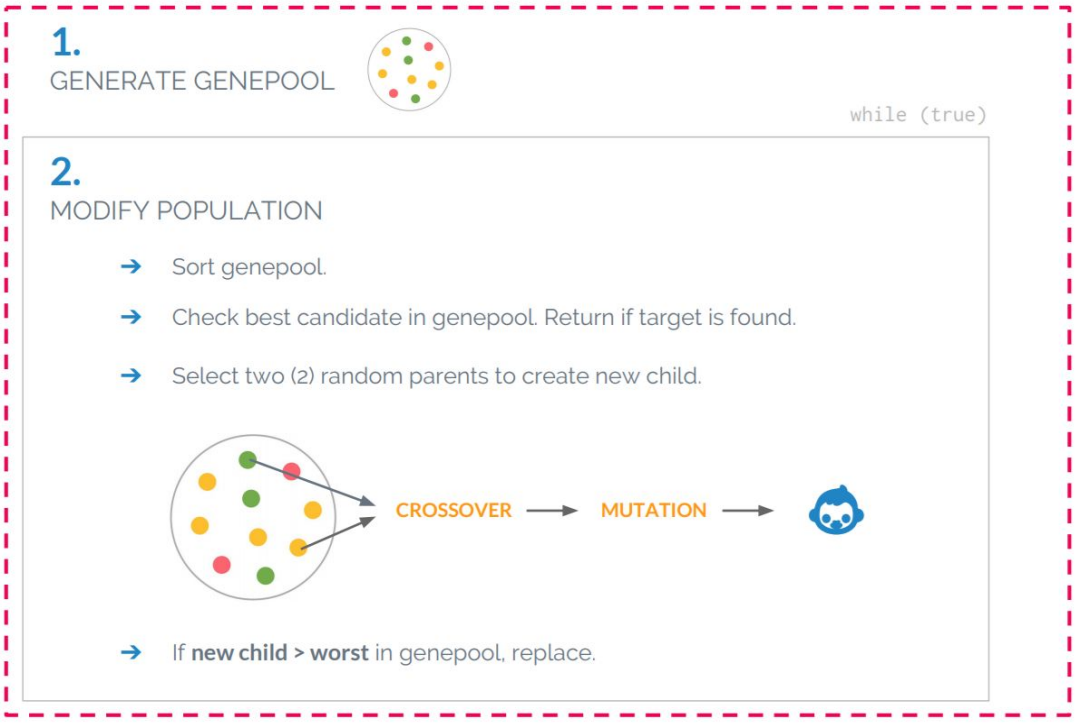

Figure 1.7 Method B

This parallel method approaches the problem in a slightly different manner. Rather than all threads sharing the same genepool, every thread has their own genepool as in Figure 1.7(the dotted red lines is the parallel portion). At the start, each thread would generate their genepool and continue with their individual crossovers and mutations, at which stage each thread would have a child as a result. Once all the threads have created a child, they would compare their own children with the current best child, which is a global shared variable between all threads. This is the key step, not only to decrease the number of generations required, but also to decrease the time elapsed. The reason being is that this approach ensures no threads are being left at disadvantage if they happen to be headed in the wrong direction. The workflow of parallel portion is depicted in Figure 1.8.

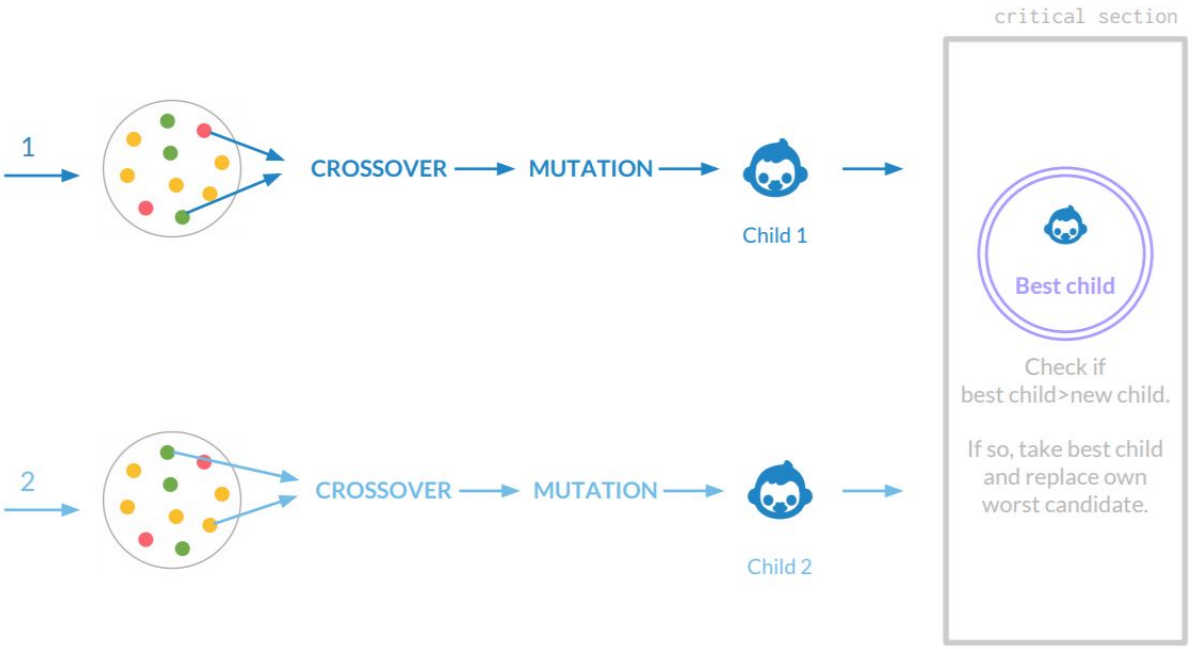

Figure 1.8 Workflow of Parallel Portion in Method B

1.5 System Calls Used

Generally, we have used two system calls to generate random numbers and to get system time. The following are the system calls we have used,

1. **omp_get_wtime() -** returns elapsed wall clock time in seconds.

   Syntax: double omp_get_wtime(void);

2. **omp_get_wtick()** - returns the precision of the timer used by omp_get_wtime.

   Syntax: double omp_get_wtick(void);

3. **rand() -** It is used to generate random numbers.

   Syntax: **int rand(void);**

4. **srand()** - This function sets the starting point for producing a series of pseudo-random

integers.

Syntax: **void srand(unsigned seed):**

1.6 Tools and Technologies

1. Kali Linux 2020.1b

2. GCC 7 compiler

3. OpenMP library

4. Required header files were added

1.7 Implementation

The C implementation started off with the serial version which was adapted from the most optimal serial method we designed in algorithm. We then developed a parallel version with two methods: Method **A** and Method **B** to produce **generations vs. threads** data.

OpenMP requires a more abstract API, which makes for quick parallelization of the code. For Method **A**, a **best_child** candidate is declared before each thread is run. Once each thread has finished generating their child, the **#pragma omp critical** directive creates a critical region in the code when checking if the **best_child** is better than the current child - and if so, it copies it. This is simply a way of determining which of the children generated by the threads is the fittest.

Once the parallelized code completes, the **best_child** is checked against the worst candidate in the population and replaced if better, following the rest of the algorithm.

In Method **B**, a very similar behaviour occurs, except that the initial **#pragma omp parallel** directive is placed before the genepool is created. Similarly, a **best_child** variable is used, along with a **#pragma omp critical** between the threads to determine which thread has the best child in each iteration.

1.8 Results and Discussion

In order to get the most accurate results possible, several executions were done and averaged before being gathered and graphed. The results were based on **the number of Threads vs. the number of Generations.**The code was run on a device with a 1.8 GHz Intel Core i7-8565U processor and 16 GB of RAM.

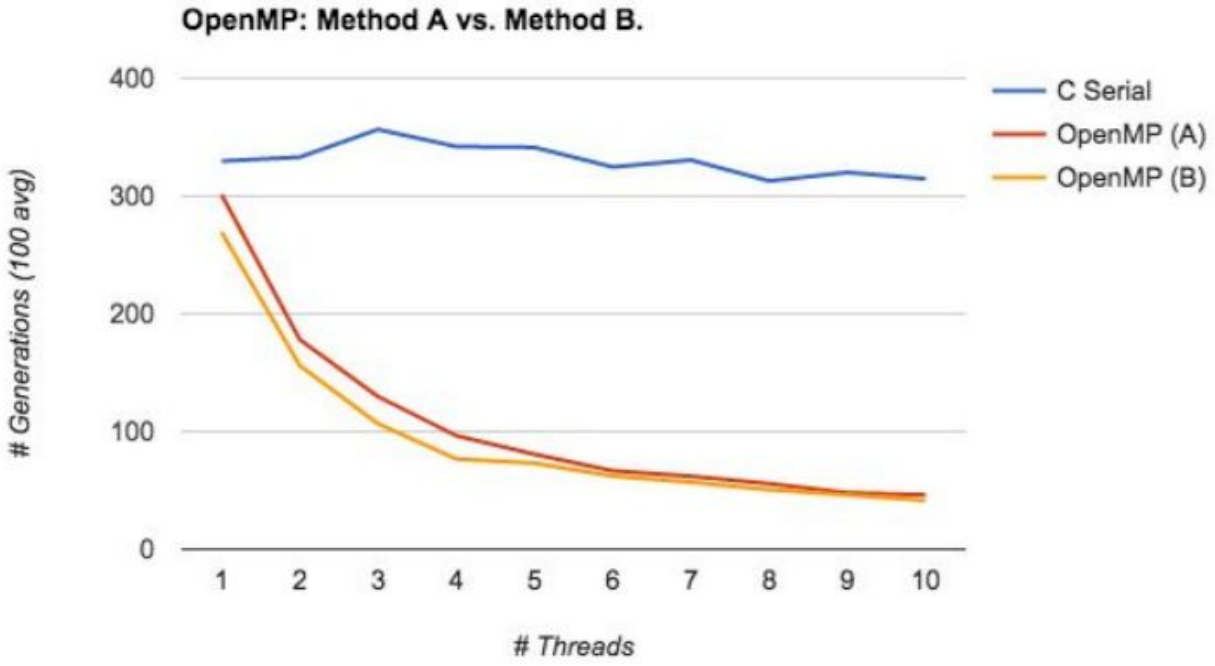

Figure 1.9 Result graph

When referring to **# Generations** in our results, we are talking about **the Number of Threads vs. The Number of Generations**. In other words, with a given range of threads {1 , ... , 64} the graph Figure 1.9 shows how many generations each implementation needed before finding the target with the current provided thread number. There are two takeaways from the graph above:

One is that, as we had initially hypothesized, parallel implementations take significantly less generations compared to their serial counterparts. This is not exactly a revelation, since there are more candidates, crossovers, and sharing of children that ultimately aids in finding the target in less iterations.

The second observation worth noting, is that as the number of threads decreases, the graph takes on an exponentially decreasing form. This just means that regardless of the method or language, as the number of threads increases, the improvement on obtaining less generations begins to fall off. The reason that they are all identical is because the number of generations are independent of the method and language used, and is not as interesting of a fact as it might actually take more time overall.Therefore Method **B** gives a better result than Method **A**.

1.9 Conclusion

These results give us a good reason to use parallelism as we will get better execution times, increase our productivity, reduce costs and reduce the “time-to-market”.The performance of parallel algorithms may be increased significantly with programming techniques oriented to locality and memory issues. However, although parallel programming is hard and error-prone, it is the primary source of performance gain in modern computer systems. Thus, we have successfully demonstrated OpenMP and have gained a deep insight into the usage of OpenMP.

1.10 Bibliography

1.10.1 Books and Research Papers

- OpenMP Common Core: Making OpenMP Simple Again – by Tim Mattson, Helen He, Alice Koniges (2019).
- Yamian Peng, Jianping Zheng, Chunfeng Liu, and Aimin Yang (2011) Research and Application of Parallel Genetic Algorithm, Hebei United University Tangshan 063009Hebei P.R. China, *Springer China.*

1.10.2 Websites

- https://www.openmp.org/wp-content/uploads/omp-hands-on-SC08.pdf
- https://www.rc.fas.harvard.edu/wp-content/uploads/2016/04/Introduction-to-OpenMP.pdf
- http://mat.uab.cat/~alseda/MasterOpt/Beasley93GA1.pdf