Genetic Algorithms

Parallelized

Luisa Rojas

Alexandar Mihaylov

Overview

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 comprised 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.

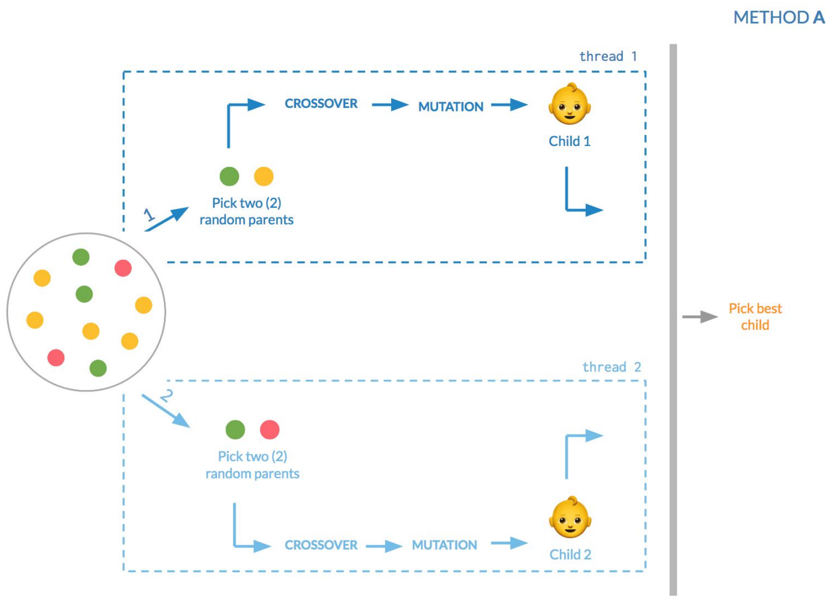
Implementations

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 Python. 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 prototype was to minimize the number of generations required to find the target. The initial implementation was 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 Python code, we implemented it sequentially in both Java and C.

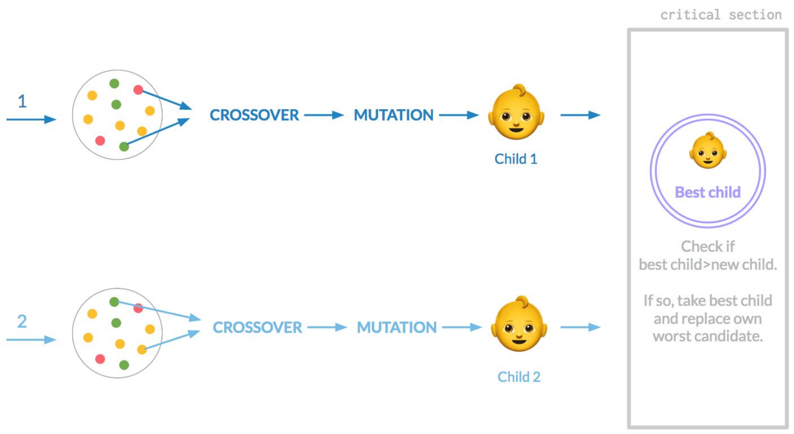
Finally, we parallelized Java using Java Concurrency, and C using POSIX Threads and OpenMP. There are multiple ways in which genetic algorithms can be parallelized; for the purpose of this project, we implemented two: Method A and Method B. Method A was written in both Java and C; however, Method B was only implemented in C due to some unresolved errors we encountered. Below is a general overview of the two methods.

Method B



This method mainly focussed on parallelizing the selection phase of the algorithm. As depicted in the figure above, all the threads would share the same genepool. There are only two threads shown above 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.

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. 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.

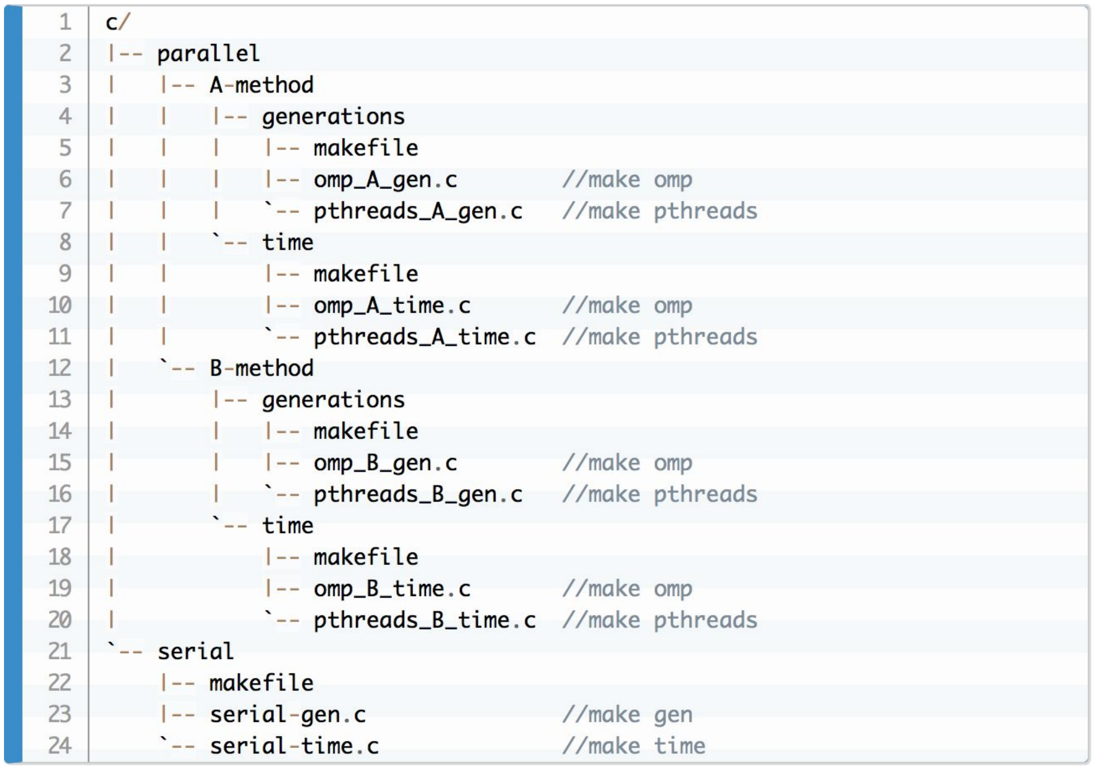
Code

Since there is an overwhelming amount of code, we have split it into three different sections, each corresponding to their respective languages.

Python

The Python code is not as relevant to this project as the others are, given that its primary purpose was to be used as a learning tool. We, however, decided to include it in the submission to highlight the evolution of our learning process.

C



```
c/
|-- parallel
|   |-- A-method
|   |   |-- generations
|   |   |   |-- makefile
|   |   |   |-- omp_A_gen.c         //make omp
|   |   |   `-- pthreads_A_gen.c    //make pthreads
|   |   `-- time
|   |       |-- makefile
|   |       |-- omp_A_time.c        //make omp
|   |       `-- pthreads_A_time.c   //make pthreads
|   `-- B-method
|       |-- generations
|       |   |-- makefile
|       |   |-- omp_B_gen.c         //make omp
|       |   `-- pthreads_B_gen.c    //make pthreads
|       `-- time
|           |-- makefile
|           |-- omp_B_time.c        //make omp
|           `-- pthreads_B_time.c   //make pthreads
`-- serial
    |-- makefile
    |-- serial-gen.c                //make gen
    `-- serial-time.c               //make time
```

The C implementation started off with the serial version which, was adapted from the most optimal serial method we designed in our prototype. We then developed with both Methods **A** and **B**, and implemented two versions of each, one measuring the generations with varying number of threads, and the other measuring the time with varying target length.

OpenMP

OpenMP requires a more abstract API, which made 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.

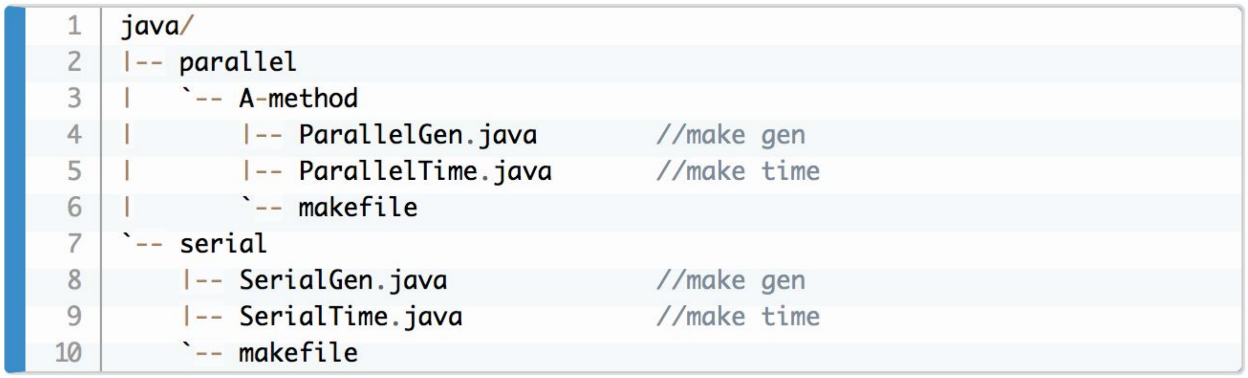
In Method B, a very similar behaviour occurs, except that the initial **`#pragma omp parallel`** directive is placed before the genepool is created. Similarly, a **`best_child`** variable is used, along with a **`#pragma omp critical`** between the threads to determine which thread has the best child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

POSIX Threads

Pthreads are slightly more involved, so it required not only a more intricate set-up, but also more care parallelizing it. In similar fashion to OpenMP's Method A, a thread is created before the selection stage of the algorithm. Some thread and attribute initialization is done beforehand, the the most important one being the **`mutex_best_child`**, which is used as a lock when each thread performs their individual comparisons. Once the threads are created, the **`run_thread()`** function is called. In this function, every thread selects the two parents from the shared genepool and eventually form a child after the mutation step. The critical section of this code starts when **`pthread_mutex_lock(&mutex_best_child)`** is called. It is followed by the comparison to **`best_child`** and the returning of the mutex for use by other threads.

In Method B, a **`run_thread()`** function is similarly called, however, instead the function does quite a bit more. Essentially, steps from generating the gene pool size, to finally reaching the target is done within that function. The one exception is that it also uses the **`mutex_best_child`** mutex in order to perform comparisons and sharing of best child candidates for each iteration of the algorithm.

Java



```
java/
|-- parallel
|   `-- A-method
|       |-- ParallelGen.java        //make gen
|       |-- ParallelTime.java       //make time
|       `-- makefile
`-- serial
    |-- SerialGen.java              //make gen
    |-- SerialTime.java             //make time
    `-- makefile
```

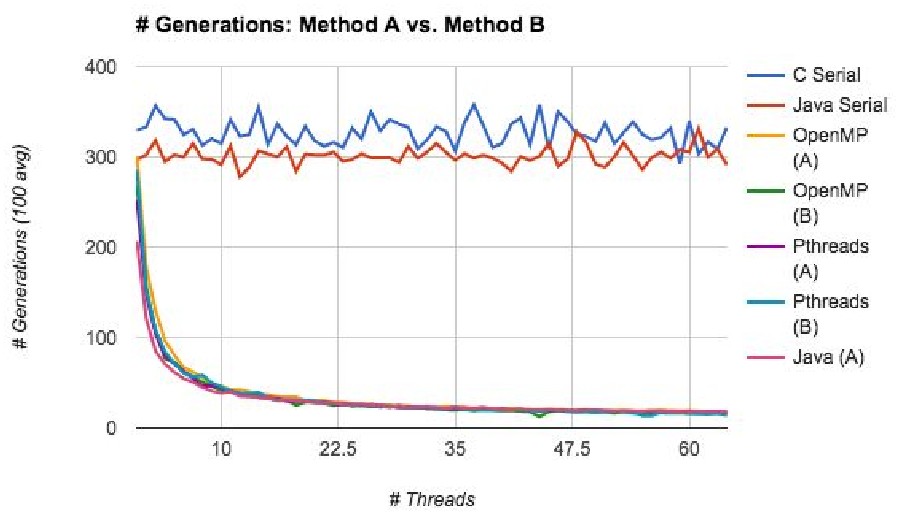
The Java implementation was a only done for Method A and was done similarly to the Pthreads approach. Threads are created in typical Java Concurrency fashion, and within each thread the similar selection, crossover, and mutation take place. Instead of mutexes however, the shared object is **synchronized** to ensure that only one thread can access it at a time during the comparison phase. Two implementations were done in Java, one to produce the **generations vs. threads** data and another for the **time vs. target length** data.

Results

In order to get the most accurate results possible, several executions were done and averaged before being gathered and graphed.

The code was ran on a device with a 2.7 GHz Intel Core i5 processor and 8GBs of RAM.

Generations

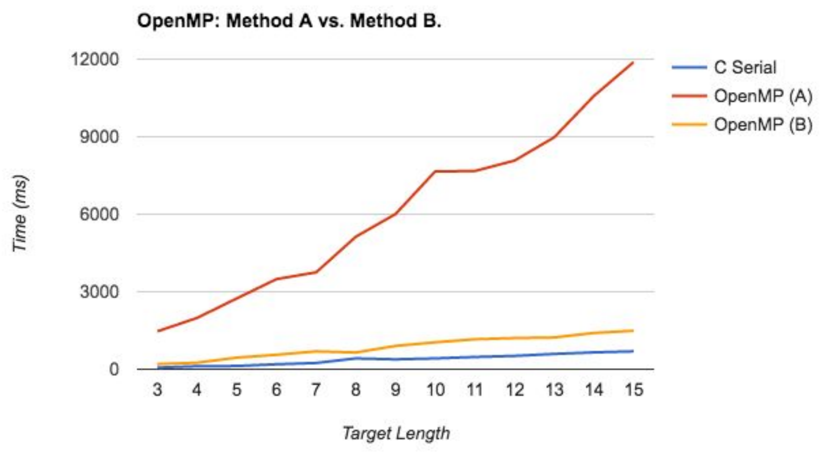


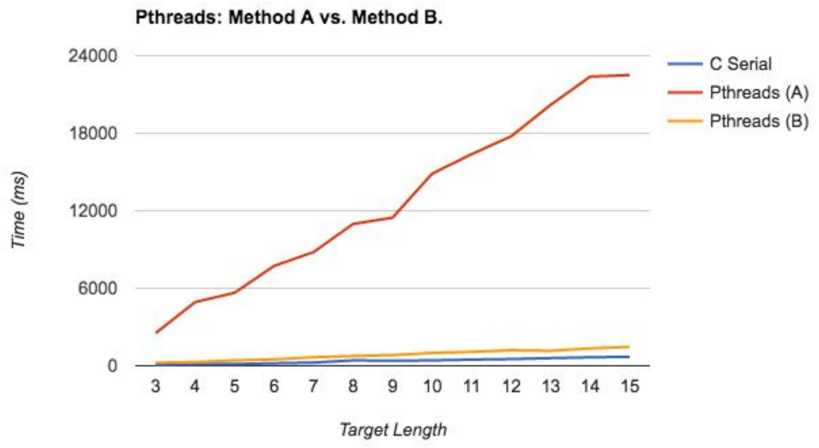
When referring to *# Generations* in our results, we are talking about the **Number of Threads vs. The Number of Generations.** In other words, with a given range of threads {1 , … , 64} the graph above shows how many generations each implementation needed before finding the target with the current provided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

Time





When referring to time in our results, we are talking about the **Time vs. Target Length**, or in other words, how much time is required to find a target given varying target lengths. The target lengths in both graphs above, the target length varied from {3, ... , 15}. When using a target with a DNA of length 15, the algorithms started to show some significant slowdown and it ultimately resulted in being a good upper bound. Both of the graphs above compare the implementation of Method A versus Method B.

Both **`pthreads`** and **`openmp`** are compared in the respective methods mentioned earlier, and there is a distinct winner in both cases. The ultimate goal was to minimize the time taken, and Method B seems to be the most time efficient of the two. It is not quite as low as the serial implementation, but is fairly close to it.

Our theory as to why Method B outperforms Method A is due to the overhead in constantly having to create the threads in each *Selection* phase, which occurs once during every generation. This means that if 1000 generations were required to reach the target, in the omp implementation, for example, would end up calling **`#pragma omp parallel`** 1000 times.

Going back to the OpenMP best practices, we know that this is a pitfall and should be avoided as much as possible. It wasn't until we discovered the flaw following our implementation of Method A that we decided to implement Method B. The modification has proven to be an overall successful enhancement.