Project in Language Abstractions for Concurrent and Parallel Programming

Sverrir Thorgeirsson Sander Cox

December 11, 2015

Contents

1 Introduction

We chose to introduce parallelization to a genetic algorithm for solving the subgraph isomorphism problem, in particular the algorithm described by Choi, Yoon and Moon [1]. The subgraph isomorphism problem is about determining whether a given graph "is inside" another graph, i.e. whether there is a bijective mapping from the nodes and edges of the subgraph to nodes and edges of the bigger graph. This is by definition a decision problem, but in multiple applications the problem can be transformed to an optimization problem: how well does a specific subgraph fit into another graph?[1]

A genetic algorithm is in short an algorithm that runs for a number of generations and during each generation performs a number of operations on a set of *chromosomes* (possible solutions for the problem) - for example *crossover* and *mutation* - in order to determine the fitness of every chromosome. Then during each generation it replaces a part of population with chromosomes that it generated. These ideas and terms are borrowed from genetics, hence the name genetic algorithm.

[1]This is for example useful for malware detection [2], after converting computer programs to control flow graphs, and plagiarism detection.

1.1 Algorithm

In a nutshell, the genetic algorithm is described by the authors as follows:

Let two directed graphs be given, with one graph larger than the other. The objective is to find if the smaller graph is a subgraph of the other graph, or at least how close it comes to being a subgraph. Let 100 chromosomes be generated initially, with a chromosome meaning a permutation of the nodes of the the larger graph so that its n first nodes are mapped in order to the nodes of the smaller graph. Then in each generation, twenty chromosomes are generated using a crossover function that has been called *cycle crossover*. With a probability of 0.2, each chromosome is mutated (i.e. altered slightly). Then the fitness of every individual is calculated using a weighted sum of two particular fitness functions; the first one (a previously known fitness function) counts the number of different edges between the two graphs, the second one (a fitness function introduced by the authors) compares the degree of each vertex of the little graph with each corresponding vertex of the chromosome[2]. Then the twenty individuals with the worst fitness are thrown away and replaced by the twenty children generated, at which point the generation ends and the next generation starts with the same procedure. After hundred generations, or after an optimal solution is found, the algorithm terminates and the best solution in the last generation is returned.

1.2 Implementation and parallelization opportunities

The above described an algorithm that was implemented sequentially. We believe that a performance gain could be realized in at least two ways though introducing parallelization:

1. At the start of the algorithm, the fitness of each individual that was generated randomly needs to be calculated. This could easily be done in parallel. A small performance gain could also be achieved by letting the algorithm terminate at this step if an optimal solution is found.

2. The crossover function can generate n children instead of just one. In the next step, the fitness of each child would be calculated, and last, the most fit child would be returned.

After performing the above changes, we can increase the complexity and hence the accuracy of the fitness function without much loss of performance; a poor fitness function is often a significant weakness in genetic algorithms, so this could increase the quality of the algorithm.

In order to introduce this parallelism to the algorithm, we wrote two implementations of the algorithm in two different languages, Python and Erlang, each with one parallelism paradigm in mind: *task-based parallelism* and *actor parallelism* respectively.

[2]The reasoning behind this fitness function is that (in order to have isomorphism) a vertex of the little graph with more outgoing edges cannot be mapped to a vertex in the chromosome with fewer outgoing edges. The same holds for incoming edges. So essentially this fitness function counts the number of edges of the chromosome that have fewer outgoing or incoming edges than the according vertex of the little graph.

2 Use cases

The Python implementation is compiled and run as follows:

```
python Isomorphism.py
```

The Erlang implementation is compiled and run as follows:

```
c(framework).
framework:main().
```

This will run the algorithm for what we consider the default use case; when the larger graph is random 10-regular graph with 100 nodes and the smaller graph is a random 5-regular graph with 10 nodes.

3 Program documentation

For each program version, we implemented the algorithm described in Section 1.1 with the following changes:

1. Instead of testing the algorithm on the usual Erdős–Rényi-type of random graphs, we instead considered random k-regular graphs, i.e. graphs where each node has k randomly selected neighbours (excluding itself).

2. For efficiency purposes, we consider a chromosome to be a permutation of some n nodes of the larger graph where n is the size of the smaller graph. As before, a chromosome represents a particular subgraph x of the larger graph and a mapping between x and the smaller graph.

3. We only used the first fitness function described by the authors; finding how many edges are different in the smaller graph and the subgraph of the larger graph that is being considered, using the smaller graph and the mapping that is encoded by the chromosome.

4. We defined our own crossover function; given two chromosomes, each with size n, we randomly select n nodes from each. Those n nodes give us a new chromosome. Every time the crossover function is called, we generate ten children in this way, then find the fitness of each in parallel and then select the child with the best fitness.

5. Instead of using roulette selection to find two parents from the population, we used another algorithm that is also biased towards individuals with high fitness. Let f be the fitness of the best individual. Then we select randomly (with uniform distribution) an individual x from the population. Let its fitness be called g. If f/g is higher than some random number r (sampled uniformly between 0 and 1, we select x, otherwise we repeat until an individual is found. This means that if the individual with the best fitness is incidentally selected, it is guaranteed to be chosen.

6. We did not implement mutations; they had no significance on the performance of the algorithm.

Each version models the exact same algorithm; we generate the random graphs in the same way, keep all the constants same, etc., which means that the output that we get from each program for each run is similar (the only difference is due to variance; the programs are non-deterministic). By doing this, we are able to compare the programs to each other directly (by seeing which one tends to be faster).

The functions and data structures in both programs are very straightforward; we use lists to keep track of all the individuals in each generation (makes it easy to apply the parallelization ideas in both implementations; in Erlang with pattern matching and in Python with the `pool.map()` function) and we have functions that perform the operations described in the above algorithm, such as a `crossover` function that takes parent chromosomes and the two graphs as arguments and returns a new chromosome.

4 Concurrence abstractions

The two parallel frameworks we used were the actor model for concurrent computation in Erlang and task-based parallelism in Python. For the first implementation, we let concurrent processes find the fitness of a given individual using the graphs of the given problem as reference. When a process has calculated the fitness of the individual in question, two things may happen:

1. The fitness of the individual is zero. This means that a perfect solution was found so continuing searching for solutions becomes irrelevant. Hence the process prints the solution to the console and then the program terminates using `init:stop()`. If, incidentally, multiple chromosomes represent an optimal solution and multiple distinct processes find their fitnesses simultaneously, all those solutions may be printed out in the console before the program terminates.[3]

2. The fitness of an individual is non-zero. Then this result is sent to a supervisor process, which adds it to a list with the corresponding chromosome attached, and when all the processes are done, this list is used in the next step of the algorithm.

For the parallelism in our Python version, we used a `Pool` from the `multiprocessing` library. This library appears to be a relatively recent addition to Python with limited documentation, although we found some information in a blog post by Doug Hellman [3]. Given a list of chromosomes, we can use the pool's `map` function to create a list of tuples, each containing a chromosome and its fitness. As discussed on Hellman's blog, our `pool` takes care of creating jobs out of the elements in the array and passing them to the threads that the `pool` created, presumably just one at a time. Then each thread will apply our fitness function to each element. This will terminate when there are no more jobs to be completed.

Unlike our Erlang implementation, we do not terminate the program immediately if some thread found the fitness of a chromosome to equal zero. Instead,

[3]Note: if there is a need for the program to be contained as a part of another application, this part of the program could be rewritten, but we did not intend that as a use case (both implementations should be a stand-alone application).

we will sort the elements in the resulting array according to their fitnesses and check if fitness of the best chromosome equals zero, in which case that chromosome will be printed out and the algorithm will terminate. This is not a big drawback to the performance compared to the Python implementation (especially considering that optimal solutions are rarely found). On the other hand, it is a larger performance problem to create and work with pools all the time (which has a large overhead); we predicted that since it is cheap in Erlang to create and communicate between processes that our Erlang version would be much faster. (The fitness function is probably just so computationally cheap that it is not worth it to bother with Python's large parallel overhead.) The results in the next section confirmed our suspicions.

5 Performance evaluation

We evaluate the performance of the different implementations by comparing the execution time between the two for the same properties. The time measurements in Python have been done by inserting time stamps in the code with the help of time.clock(). In Erlang the time measurements have been done from the command line with the help of the tc function in the timer module using the command: timer:tc(framework, main, []).

The time measurements were performed on the lab machine `trygger` and therefore 16 threads have been used as a parameter. The following values were attained (see also Figure 1):

Case	Python	Erlang
1	8.77 s	1.70 s
2	12.29 s	2.00 s
3	22.13 s	3.45 s
4	64.63 s	3.49 s
5	98.35 s	8.21 s

The five different cases in question are increasing in complexity, according to the following parameters:

case 1: big_nodes: 50, small_nodes:5, big_density:5, small_density:3
case 2: big_nodes: 50 , small_nodes:5, big_density:10, small_density:3
case 3: big_nodes:100 , small_nodes:10, big_density:10, small_density:5
case 4: big_nodes:200 , small_nodes:20, big_density:10, small_density:5
case 5: big_nodes:200, small_nodes:20, big_density:20, small_density:10

All the other parameters were the same (if applicable) for all cases: generation_no: 100, population_size:100, children_no: 20, child_tries:10 (how many children are generated in the crossover function before one is selected) and processor_no: 16 (not limited in Erlang).

As we see by those results, our suspicions were confirmed; the Erlang implementation appears to be more efficient than the Python implementation for data of any size we tested.

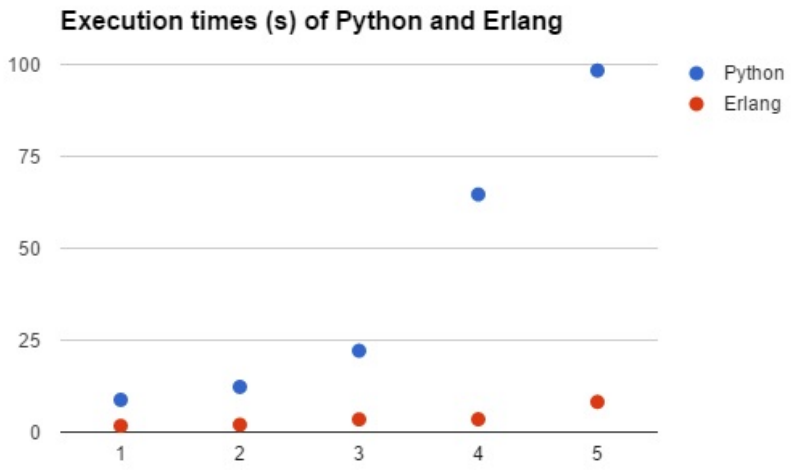

Figure 1:

6 Known shortcomings

Given the default use case (see Section 2) our algorithm will usually return a solution whose fitness is somewhere between 35 to 45. Although we cannot compare this result to anything that we are familiar with (the paper we are following used different types of random graphs), we suspect that these results are not particularly good. The accuracy of the algorithm could most likely be improved by

1. Using the cycle crossover as suggested by Choi et al. However, unlike our crossover function, their crossover function is deterministic given two predetermined parents, so the opportunity for parallelism in the crossover function is less obvious (for example it does not make sense to use that crossover function to generate ten children, find their fitness in parallel and then select the child with the best fitness).

2. Introducing mutations, using a roulette selection algorithm to find parents for the crossover, use the weighted fitness function that was suggested by the authors, etc.

In order to improve the speed of the algorithm, we could introduce more parallelism, such as selecting parents in parallel, or possibly there could be some mechanism for starting work on the next generation in parallel before the previous one finished completely. However, we think that this would probably work better on Erlang than in Python; Python is not know for being speedy (just like our results in the previous section show) and the multiprocessing library seems to come with some relatively high parallel overhead. However, it should be noted here that it was much easier to write the code in Python than Erlang and if the algorithm improvements discussed above were to implemented, it would be

easier to do so in Python (since our Python code is rather modifiable). Possibly one could use the Python version as a prototype before implementing them in Erlang (or maybe in a fast language like C++).

Lastly, another important addition to the implementations that should be made in the future is to allow the user of the program to define the larger graph and the small graph themselves, i.e. that the program could take those graphs as input. Currently, it only considers random graphs whose sizes and degrees are hardcoded constants at the beginning of the programs (which is not good design, but it works for us to see how well the algorithm performs).

References

[1] Choi, Jaeun and Yoon, Yourim and Moon, Byung-Ro, *An Efficient Genetic Algorithm for Subgraph Isomorphism*, Proceedings of the 14th Annual Conference on Genetic and Evolutionary Computation, 2012.

[2] Kim, Keehyung and Moon, Byung-Ro, *Malware Detection Based on Dependency Graph Using Hybrid Genetic Algorithm*, Proceedings of the 12th Annual Conference on Genetic and Evolutionary Computation, 2010.

[3] Doug Hellman, *Communication Between Processes*, Link: https://pymotw.com/2/multiprocessing/communication.html, 2015.