

Parallel Genetic Programming

Abiyaz Chowdhury Allen Kim

May 21, 2018

Abstract

Genetic programming is a technique in the field of genetic algorithms that evolve computer programs themselves. Programs are encoded in some computational representation that are then run through the standard genetic operations (selection, crossover, mutation). However, the computational cost to train genetic programs grows rapidly as the complexity of the problem increases. We aim to implement a highly parallel version of genetic programming that will scale efficiently.

1 Introduction

Genetic algorithms have primarily seen use in optimizing numerical parameters. However, genetic programming has not seen as much mainstream use in the past due to its severe computational costs. Genetic programming aims to evolve the computer program themselves over time rather than various function parameters [1]. By having programs themselves be the population, genetic programming ultimately aims to have computer-generated programs that show distinct characteristics from human written programs and ideally, efficient as well.

The main problem with genetic programming is the fact that in practice, it is only able to solve small problems. The sheer number of possible programs that exist makes it difficult to even create simple programs. However, with the growth in modern computing power and the use of parallelism, genetic programming is beginning to seem more practical.

2 Prior Work

One main topic in prior work in genetic programming deals with how to represent the programs themselves structurally. The simplest idea was to use a forest of parse trees since compilers are used to dealing with such structures for programs, but they tend to be much slower to analyze. Other data structures such as directed acyclic graphs [2] were later proposed to allow for greater efficiency, but we focused on parallel distributed genetic programming (PDGP) [3].

In PDGP, programs are encoded as grids in which each node can be non-terminal or terminal. Non-terminal nodes represent different functions that take in other nodes as arguments while terminal nodes represent the input to the program. There are many variations to the grid, but we chose one of the simplest representations. Akin to a feed forward neural network, we considered grids that had connective edge between adjacent layers. Thus, our programs would propagate values upward from bottom to top through a series of nodes. This was still an efficient representation as nodes can be reused by the layers above, but was also simple enough to understand and implement. With the identity node, any grid could be represented in this way given sufficient rows and columns.

Regarding the parallel implementations of genetic programming, most prior work splits up the population among the processors and evaluate fitness in parallel [4]. We aimed to implement these core ideas in our program as well.

3 Overview

For our project, we implemented a parallelized version of PDGP, which was built to solve arbitrary combinatorial circuit problems. We allow the use of AND, OR, NOT, NAND, NOR, and I (identity) as our non-terminals. We also use x_1 to x_n for our input. Given a function we want to learn, we use our program to find circuits that can solve it. We test our program on solving XOR as well as solving the even-3 parity problem.

Regarding parallelization, we experimented with parallelizing different aspects of the project. We found that for small grids, parallelizing the tasks related to computing on a single grid was not efficient. Thus, most of our parallelization was in the genetic programming itself, which we discuss in the implementation section.

4 Description of Implementation

4.1 Random Graph Generation

To initialize the graph, we assume that we are given a set width and height for our grid. Then, we randomly fill the grid with non-terminal and terminal nodes randomly. We do place certain constraints, however. The bottom row of the grid are always restricted to be terminals while the output node at the top is always restricted to be non-terminal. This is to ensure that every grid corresponds to a valid program. By default, each node is inactive, but after initializing, we mark every node connected to the output as active. An example is shown in Figure 1.

For our random number generator, we use a variant of XOR shifting pseudo random number generators, which run much faster than the rand function in the standard library. This is important to optimize the speed in which our populations are generated.

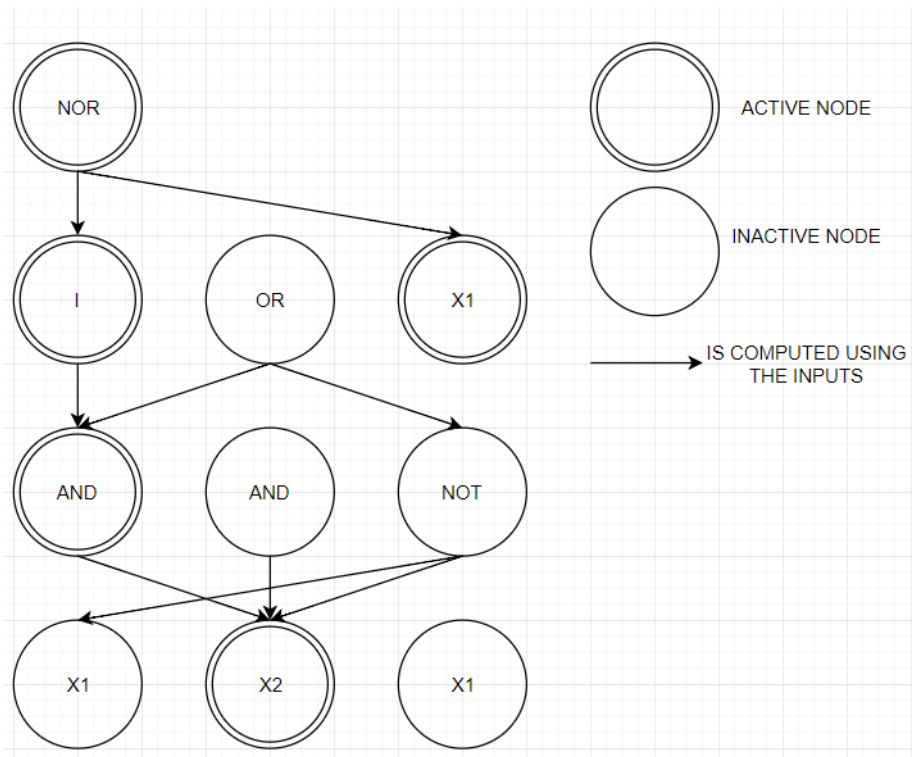


Figure 1: Example random grid

For larger grids, it may potentially be faster to parallelize the generation of the population, but for our case, the grids were small enough that benefits were not seen in adding more processors.

4.2 Evaluation and Fitness

For our evaluation, we compute the values for the output bottom up in the grid. At each node, we consider only the active nodes and cache the temporary value for the output as we move up. For fitness, we use the Hamming distance from the truth table. Since we chose to maximize fitness for our population, we subtract the Hamming distance from the total possible outputs, which is exponential in the input.

4.3 Genetic Operators

4.3.1 Crossover

For crossover, there are many variations discussed in Poli's work on PDGP, but we chose to implement the one that seemed most effective and simplest to implement: sub-sub graph active-active node. In this version of crossover, we first randomly select an active node from the one graph and a random active node from the second graph. Then, we copy the subgraph rooted at the first node over to the second node. We prune off any nodes that may extend beyond the graph as well. An example of this is shown in Figures 2 and 3.

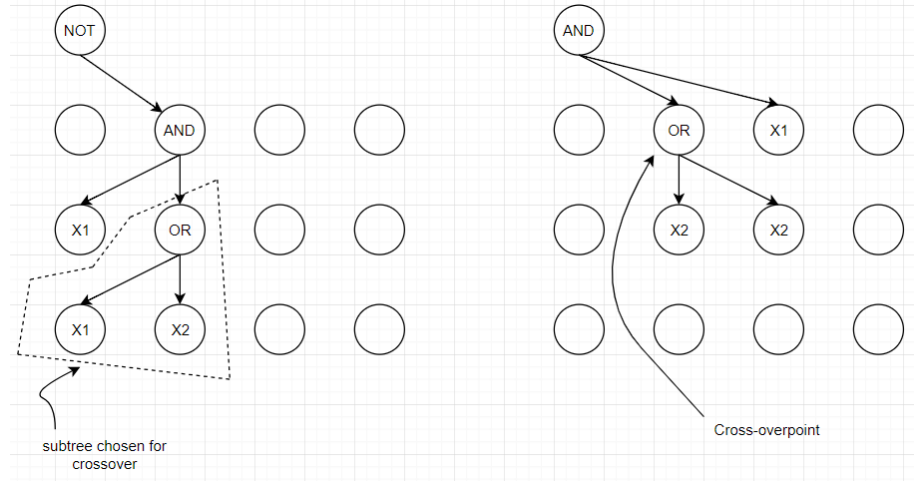


Figure 2: Before Crossover

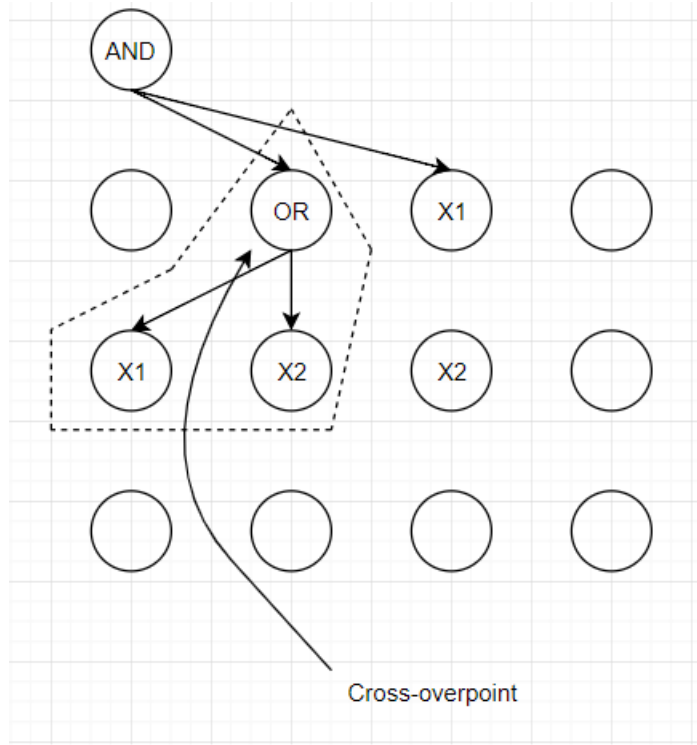


Figure 3: After Crossover

4.3.2 Mutation

We consider three forms of mutations: global, link, and node. The first two are discussed in Polis original work, but the third form was not used. We also implemented it to see if it had any positive effect.

- Global mutation injects a random subgraph at a random active node in the candidate graph. We implemented this by conducting crossover with a randomly generated graph.
- Link mutation randomly changes an edge in the grid to point to a random node below it.
- Node mutation randomly changes a node in the grid to become terminal or non-terminal and then, randomly assigns a new function or value to the node.

4.3.3 Selection

For selection, we use tournament selection where the size of tournament is a parameter to the program. In tournament selection, we are given a number

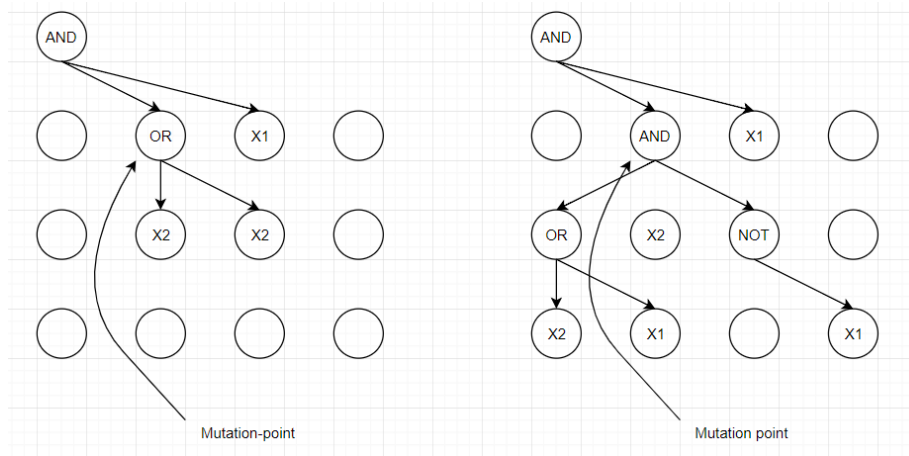


Figure 4: Global Mutation

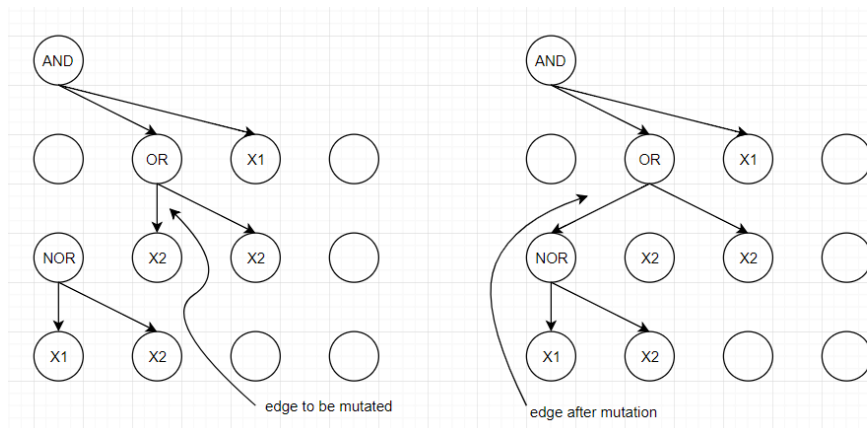


Figure 5: Link Mutation

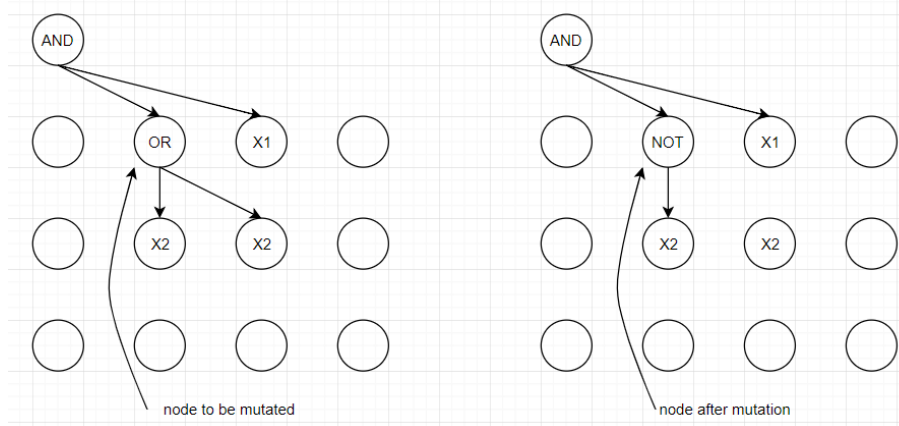


Figure 6: Node Mutation

of participants, say k , and we randomly sample k candidates from the current population. The one with the highest fitness from this set is selected. By varying the tournament size, one can control the selection pressure. If the tournament size is small, then candidates with lower fitness are more likely to survive whereas if the tournament gets larger, then there are more and more possibilities for stronger candidates to go through.

4.4 Run of a single generation

For a single generation, we run the following steps:

1. Run selection twice to find candidates for both parents.
2. With some probability, conduct crossover to create a child grid.
3. With some probability, run global mutation on the child.
4. With some probability, run link mutation on the child.
5. With some probability, run node mutation on the child.
6. Add the child to the new population and repeat steps 1 to 5 until population is full.

4.5 Multi-start Evaluation

Since genetic programming is not guaranteed to converge to the global best, we also employ a simple technique to try and initialize and run our programs with different seeds. We then take the best fit across all of these programs. This is easily parallelizable since each genetic program can be run independently from one another.

5 Analysis

We now conduct a running time analysis of our genetic program.

To generalize, we use the following variables in our program: x for the number of inputs, b for the number of bits per input, $m \times n$ grid size, α for the largest arity out of the non-terminals, S for population size, k for tournament size, G for number of generations, and T for the number of multi-starts.

$$\begin{aligned}
\text{eval} &= O(mnA) \\
\text{fitness} &= O(mnA2^{bx}) \\
\text{generate-graph} &= O(mnA) \\
\text{evaluate-fitness} &= O(SmnA2^{bx}) \\
\text{initialize-pop} &= O(SmnA + SmnA2^{bx}) \\
\text{crossover} &= O(mnA) \\
\text{global-mutation} &= O(mnA) \\
\text{link-mutation} &= O(mnA) \\
\text{node-mutation} &= O(mnA) \\
\text{selection} &= O(k) \\
\text{next-gen} &= O(S(2k + 4mnA) + SmnA2^{bx}) \\
\text{run} &= O(G(S(2k + 4mnA) + SmnA2^{bx})) \\
\text{run-ensemble} &= O(T(G(S(2k + 4mnA) + SmnA2^{bx})))
\end{aligned}$$

We now provide a brief description of each function:

$$\begin{aligned}
\text{eval} &= \text{evaluate output given input values for a single grid} \\
\text{fitness} &= \text{compute Hamming distance fitness of a grid} \\
\text{generate-graph} &= \text{initialize a random grid} \\
\text{evaluate-fitness} &= \text{evaluate fitness for each member of population} \\
\text{initialize-pop} &= \text{generate grids for each individual in population size} \\
\text{crossover} &= \text{crossover as described prior} \\
\text{global-mutation} &= \text{global mutation as described prior} \\
\text{link-mutation} &= \text{link mutation as described prior} \\
\text{node-mutation} &= \text{node mutation as described prior} \\
\text{selection} &= \text{selection as described prior} \\
\text{next-gen} &= \text{single run as described prior} \\
\text{run} &= \text{runs next gen for certain number of generations} \\
\text{run-ensemble} &= \text{run multi-start number of instances}
\end{aligned}$$

Simplifying run-ensemble , since k is upper bounded by S , we get that it takes

$$T_1 = O(TGSmnA2^{bx})$$

With parallelism, we get that

$$T_\infty = O(GmA \log T \log S)$$

The logs follow from the fact that finding the maximum over an array takes logarithmic time, which happens when finding the best candidate. Thus, we get

$$T_p = O\left(T_\infty + \frac{T_1}{p}\right) = O\left(GmA \log T \log S + \frac{TGSmnA2^{bx}}{p}\right)$$

Most of these variables are quite small, but the exponential on the number of inputs can blow up if we want to handle larger inputs. However, the main reason for this exponential is due to the fitness function. In general, the fitness function need not be exponential, so this analysis is true for our truth table fitness, but a more general expression would encapsulate the exponential term as an oracle F . In this case, we get the following new analysis:

$$\begin{aligned} \text{eval} &= O(mnA) \\ \text{fitness} &= F \\ \text{generate-graph} &= O(mnA) \\ \text{evaluate-fitness} &= O(SF) \\ \text{initialize-pop} &= O(SmnA + SF) \\ \text{crossover} &= O(mnA) \\ \text{global-mutation} &= O(mnA) \\ \text{link-mutation} &= O(mnA) \\ \text{node-mutation} &= O(mnA) \\ \text{selection} &= O(k) \\ \text{next-gen} &= O(S(2k + 4mnA) + SF) \\ \text{run} &= O(G(S(2k + 4mnA) + SF)) \\ \text{run-ensemble} &= O(T(G(S(2k + 4mnA) + SF))) \end{aligned}$$

Now, we get

$$T_1 = O(2TGSk + 4TGSmnA + TGSF) = O(TGS(k + mnA + F))$$

and

$$T_\infty = O(G \log T \log S(\log k + mA + F))$$

Thus, we get

$$\begin{aligned} T_p &= O\left(T_\infty + \frac{T_1}{p}\right) \\ &= O\left(G \log T \log S(\log k + mA + F) + \frac{TGS(k + mnA + F)}{p}\right) \end{aligned}$$

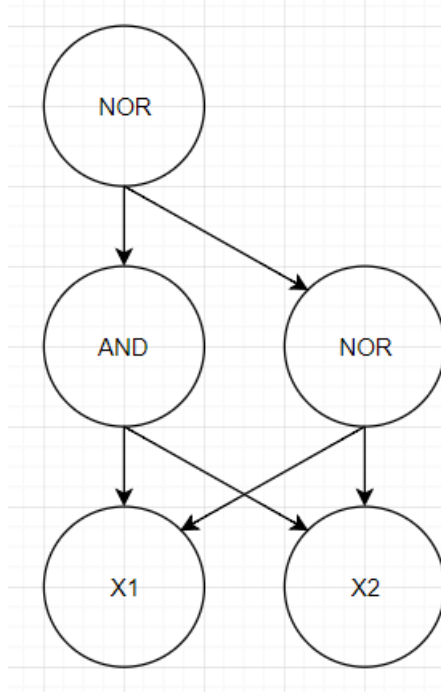


Figure 7: Example of grid that computes XOR

6 Results

Our program successfully discovers new ways to compute XOR with every run. In Figure 7, we show one example of a grid that computes XOR. This was run on a 3 by 3 grid with a population size of 200 and with 20 generations. The tournament size was 7, and the crossover probability was 0.7. Also, the probability for global, link, and node mutations were all set to be 0.25. In Figure 8, we show the scalability of the XOR program.

Our program also successfully computed even-3 parity with different programs every run. In Figure 9, we show an example of a grid that computes even-3 parity.

In Figure 10, we show the scalability of the even-3 parity program. This was run on a 4 by 2 grid with a population size of 2000 and with 50 generations. The tournament size was 30, and the crossover probability was 0.7. Also, the probability for global, link, and node mutations were all set to be 0.05. Our program successfully discovers new ways to compute even-3 parity with every run.

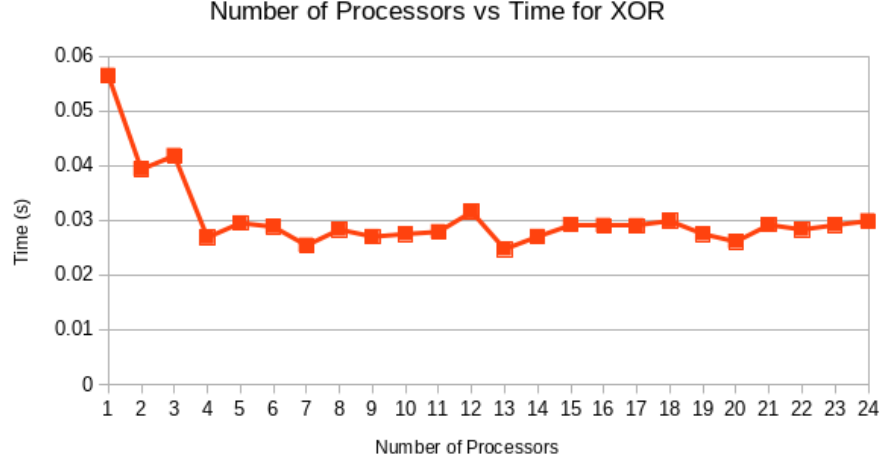


Figure 8: Scalability of XOR

7 Future Work

Currently, our approach is restricted to be a feed-forward grid, and as such, we cannot expect to solve any problems that go beyond the scope of a general combinatorial circuit. Ideally, we want to compute general functions. One simple extension would be to compute symbolic regressions, in which our non-terminals would be common mathematical operators, and we would try to deduce what function maps a given set of training points. Another nice expansion would be to use assembly instructions as our non-terminals and have our input nodes accept some string encoding to our program.

We also could try experimenting with different crossover operations. At the moment, we have one specific kind of crossover that truncates the subgraph if it is too large to fit in the other parent's grid. However, there are other variants that may handle these cases differently such as wrapping around the grid instead.

Additionally, another way to generalize our approach would be to use meta-genetic programming [5]. Currently, we hard-code in the parameters to our programs for each example such as the crossover and mutation probabilities or tournament size. In theory, these parameters can also be evolved with our programs. However, it is not clear that this would add much benefit, but can be something to experiment with.

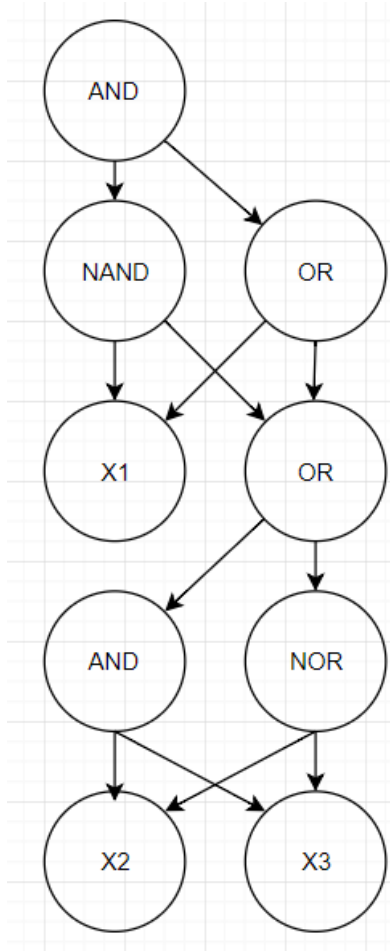


Figure 9: Example of grid that computes even-3 parity

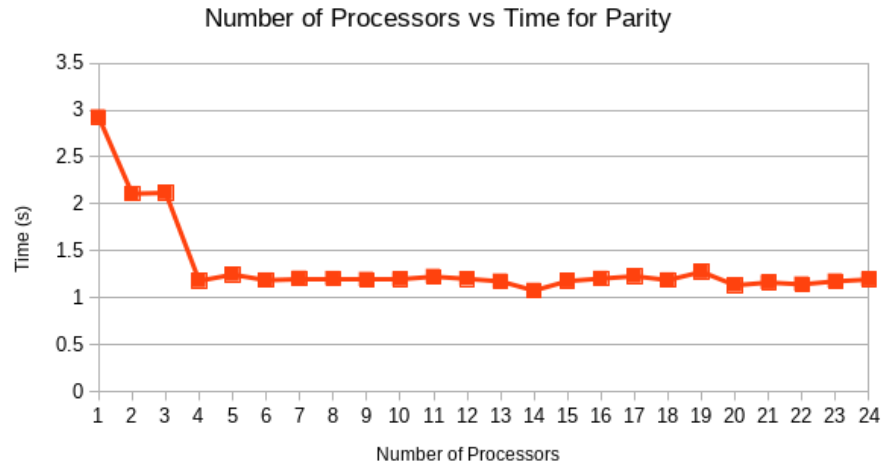


Figure 10: Scalability of Even-3 Parity

References

- [1] McPhee, Nicholas Freitag, Riccardo Poli, and William B. Langdon. Field guide to genetic programming. (2008).
- [2] S.Handley. On the use of a directed acyclic graph to represent a population of computer programs. In Proceedings of the 1994 IEEE World Congress on Computational Intelligence, pages 154-159, Orlando, Florida, USA, 27-29 June 1994. IEEE Press.
- [3] Poli, Riccardo. Parallel distributed genetic programming. University of Birmingham, Cognitive Science Research Centre, 1996.
- [4] Kilian Stoffel and Lee Spector. High-performance, parallel, stack-based genetic programming. In John R. Koza, David E. Goldberg, David B. Fogel, and Rick L. Riolo, editors, Genetic Programming 1996: Proceedings of the First Annual Conference, page 224, Stanford University, CA, USA, 28-31 July 1996. MIT Press.
- [5] B. Edmonds, Meta-genetic programming: Co-evolving the operators of variation, CPM Report No.: 98-32. Centre for Policy Modelling, Manchester Metropolitan University. <http://www.cpm.mmu.ac.uk/cpmrep32.html>, 1998.