

Parallelizing the NEAT Method using UPC

Russell Feldhausen
Computer Science
Kansas State University
Manhattan, KS 66506
russfeld@ksu.edu
<http://people.cs.ksu.edu/~russfeld>

Abstract

The NeuroEvolution of Augmenting Topologies (NEAT) method^[1] is designed to quickly evolve a neural network that can perform a specific task. This project describes a parallelized version of this method using Unified Parallel C (UPC)^[2] to spread the work across multiple threads and multiple nodes. The algorithms and shared data structures are presented, and the overall performance of the program is analyzed across multiple environments. Finally, the overall success of the project is given along with a short discussion of possible future work.

1. Introduction

A powerful method of building computer algorithms to perform certain tasks is evolutionary computation.^[3] Instead of describing to the program how to perform the task directly, the program learns by slowly changing itself while analyzing its relative success. The versions of the program which are more successful are retained and allowed to continue, and eventually the goal is achieved.

Along with that method, another useful structure for developing computer programs that can learn is neural networks.^[4] A neural network attempts to simulate learning within the human brain, as represented by individual nodes and weighted connections within the network. By adjusting the weight of each connection, the overall output of the network can be adjusted to match a desired output.

The NeuroEvolution of Augmenting Topologies (NEAT) method^[1] combines these two approaches. The method starts with a neural network that is slowly evolved by adding nodes, adding or disabling connections, and adjusting the weights of those connections. In addition, it introduces several novel methods for handling the evolutionary change within the network.

For this project, I chose to parallelize the NEAT method using Unified Parallel C (UPC).^[2] The original implementations of NEAT provided by the authors are single threaded, and are therefore limited by the speed of the computer used. The goal of this project is to explore whether the NEAT method can be effectively parallelized with UPC through the use of the data structures within the partitioned global address space (PGAS) model,^[5] parallel algorithmic design, and performance analysis.

In Section 2, I provide a brief background of the project and tools used. In Section 3, I describe my implementation, and in Section 4 I present my performance data and analysis. Section 5 gives my conclusion and recommendations for future work in this area.

2. Related Work

The idea of using computer programs that can “learn” how to perform a task has been around for several decades. It was notably discussed by Minsky and Papert in *Perceptrons*^[4], but much of the early work was done by White and Rosenblatt.^[6] The general idea of these types of programs, now typically referred to as neural networks, is to create a network of objects called “neurons” that receive input and give outputs, and then adjust the connections and weights between those neurons and through a learning process until the desired goal is achieved.

However, some functions, such as the binary XOR function, cannot be done with a single layered neural network. Therefore, a multilayer perceptron is required for many applications. This was a core point of contention between the two groups of researchers.^[7] Nonetheless, that method for creating intelligent computer programs is still widely used today.

Another technique for building computer programs that learn over time is evolutionary computation.^[3] In this technique, aspects of the computer program, such as the data structures used and parameters, are slowly evolved until they meet the desired criteria.

NeuroEvolution of Augmenting Topologies (NEAT) is a method for creating computer programs that learn first developed by Stanley and Miikkulainen at the University of Texas.^[1] It combines the structure of a neural network with the process of evolutionary computation. In addition, they propose several novel enhancements over earlier efforts to combine the two, most notably the desire to create “species” of similar programs, allowing changes to develop over time in order to find the most useful design.

One interesting use of the NEAT method is to create a neural network capable of playing video games. A YouTube user under the name SethBling did just that,^[8] providing the inspiration for this project. In his video, he describes how he was able to implement the NEAT method in a scripting language that could be connected to an emulation program, allowing the network to “play” the video game until a solution was found.

3. Implementation

The initial implementation of this project was guided by the source code provided by SethBling along with his video.^[9] He provides a very simple implementation of the NEAT method using the LUA scripting language. I also reviewed the source code from the original NEAT implementation provided by Stanley.^[10]

In creating my version of the project, I had to determine how to arrange the data structures within the PGAS model. The original LUA implementation makes heavy use of the variable sized table data structure available in that language, which doesn’t translate well to C. Also, much of the data needed to be shared across all threads and nodes, requiring the use of many UPC data structures and functions.^[11]

In my implementation, I created 3 global arrays: *species*, *genepool*, and *newgenepool*:

- *Species* - This array contains structures defining each species of network currently present in the program as well as array indices into *genepool* for each member of the species. As the networks are slowly evolved, new species may appear while stale species may be removed.
- *Genepool* - This array is generally a read-only structure containing all of the current networks in the global population of networks.

- *Newgenepool* - This array contains newly bred networks as evolution is taking place. It is written to by each thread.

The choice to separate *genepool* and *newgenepool* was made very early on in the design process. Since the global population size will be limited to well below the memory capacity of each node, it makes the most sense to make use of that memory to alleviate the risk of race conditions or errors. By having one array serve as the “parents” of the next generation of “children” being added, it greatly simplified the construction of the program.

The networks themselves are stored as a list of links between numbered nodes along with their weight. Node 0 is the “bias” node that always provides input to the system. The next set of nodes are the inputs, and the outputs follow directly after that. For the test program I used, there are 2 input nodes and 1 output node. Each network is rated based on its relative fitness. In the case of my test program, the fitness is the one more than the square of the number of correct answers it generates. Since there are 4 possible inputs, the maximum fitness is 17.

The main algorithm has two major phases. In the first phase, it removes the bottom half of the networks from each species, as rated by their relative fitness. Then, it uses the remaining half to generate several new networks within each species, allowing cross breeding to occur between remaining members of the same species. When each new network is created, its fitness is measured.

Before the second phase begins, the algorithm reviews the fitness of each network looking for a solution. It also keeps track of the network with the highest fitness in each species, and then removes all networks in the species other than the one with the highest fitness.

In the second phase, the algorithm generates an entire new population of networks using the fittest members of each species as the parents. In this case, crossbreeding is not as prominent, since each species’ representative network can only be bred with itself. Once again, after this is complete, the fitness of each network is analyzed for a solution. The process repeats with phase one until a solution is found or a sufficiently large number of generations has passed to avoid deadlock situations.

4. Performance Analysis

To test the project, I used the simple binary XOR function. It requires at least one hidden node to be generated as part of the evolutionary process in order for a solution to be found, so it helps show that the algorithm is working as intended. The tests were conducted on the Stampede supercomputer at the Texas Advanced Computing Center (TACC). Each computing node on Stampede contains dual 8 core Intel Xeon E5-2680 (Sandy Bridge) processors running at 2.7 GHz, giving a per core performance of 21.6 gigaflops.^[12]

To test the runtime of the program, I ran it using many different combinations of nodes and threads per node. The overall goal is to demonstrate if there was any scaling speedup present in this program, and if so, what the optimal configuration is.

Since the neural network is evolved randomly, the number of generations required can vary widely, affecting the total runtime of the program. To account for this, I repeated each test 24 to 32 times to give a wide range of results. In addition, for each test I also recorded the time taken divided by the number of generations, which should remain relatively constant. Any tests that resulted in a system failure or timed out due to an excessive number of generations were removed from the results.

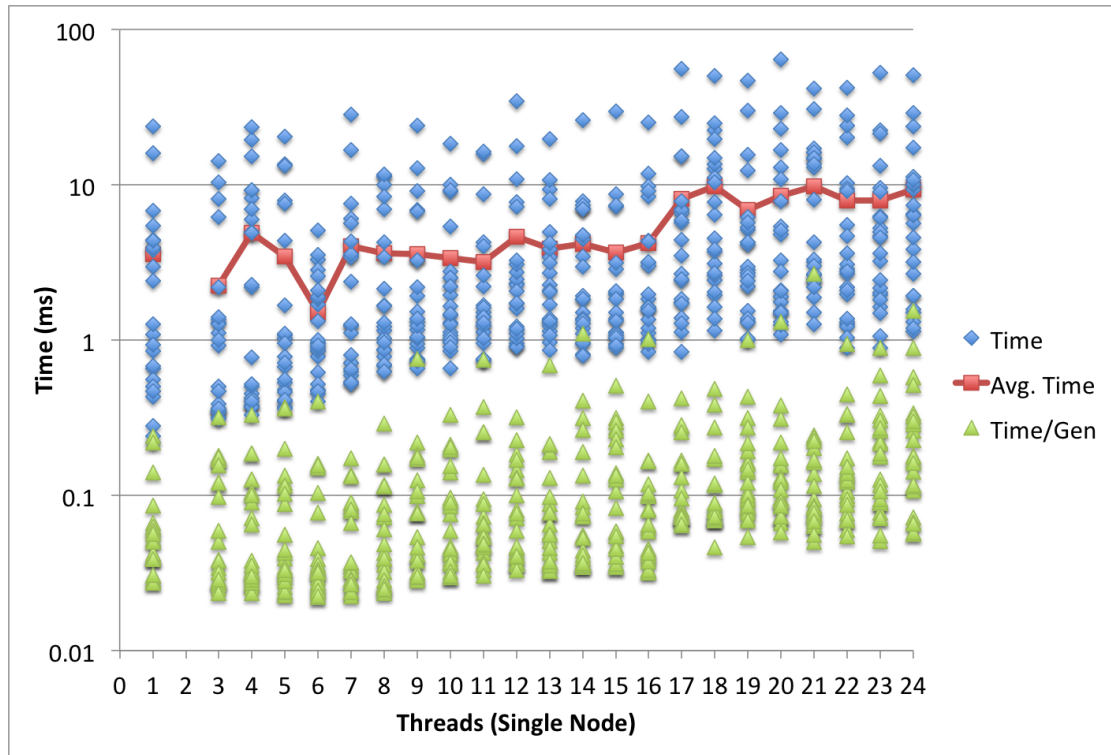


Figure 1: Single Node Scaling

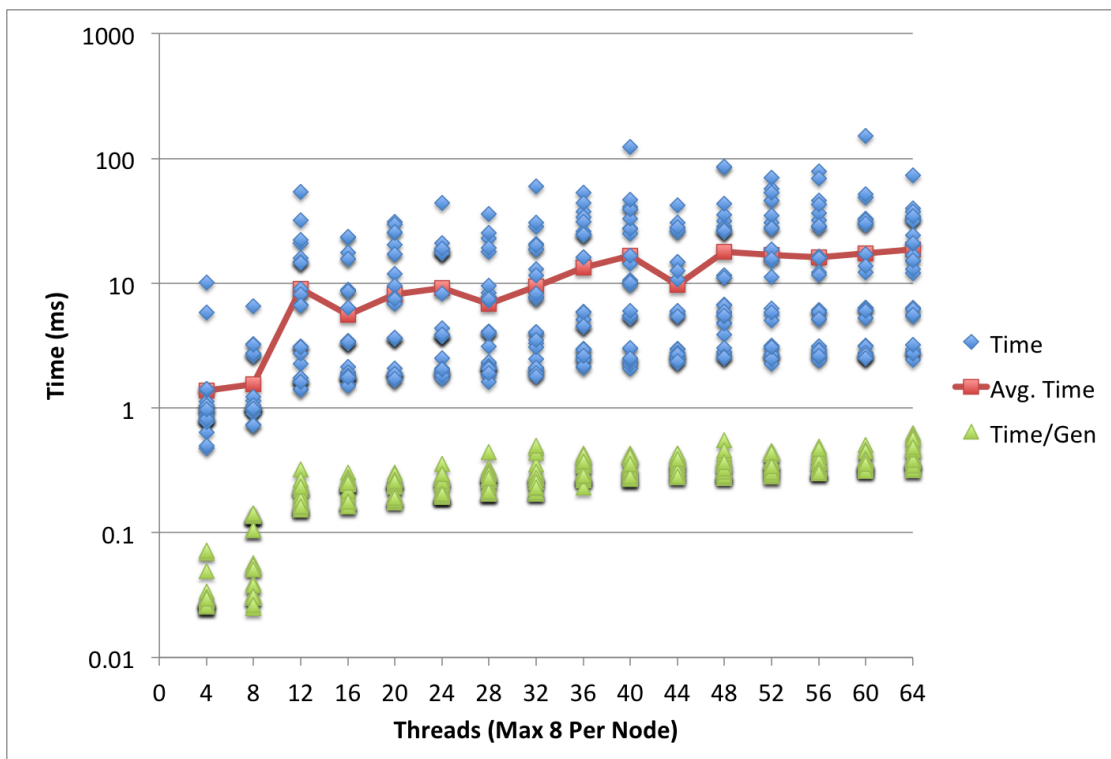


Figure 2: Multi Node Scaling Up to 8 Threads Per Node (8 Nodes Max)

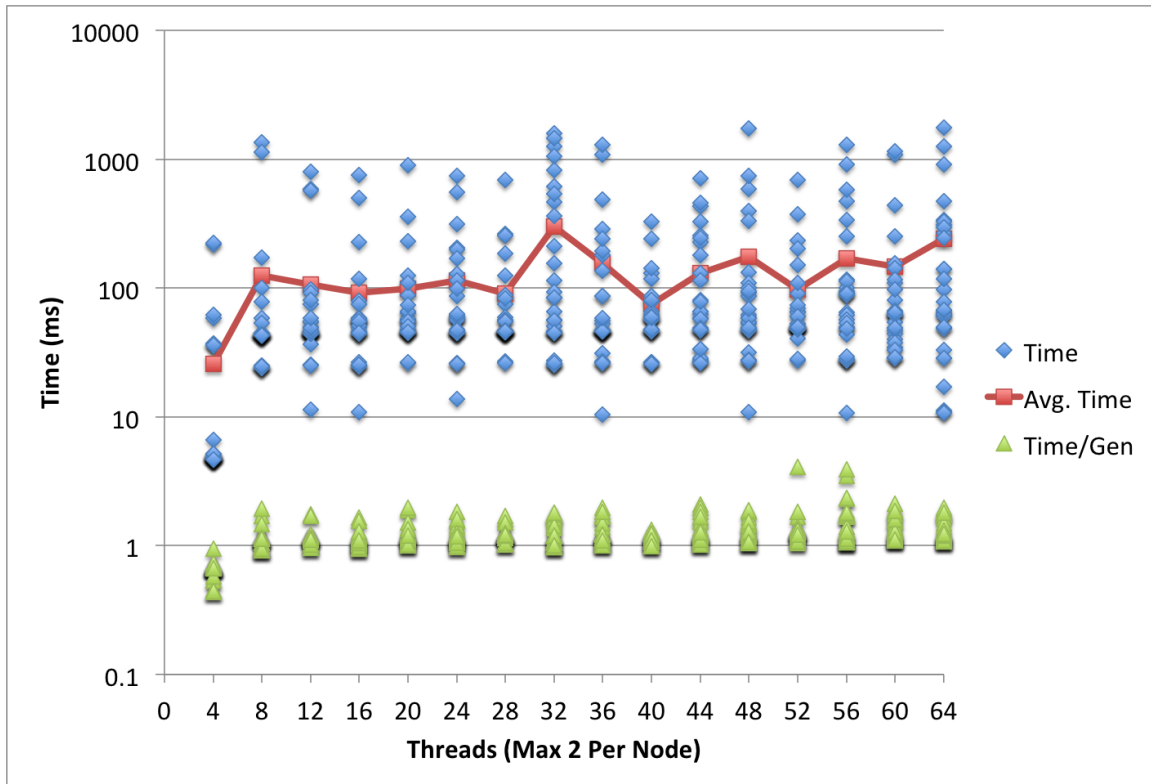


Figure 3: Multi Node Scaling Up to 2 Threads Per Node (32 Nodes Max)

In Figures 1 through 3, there are three sets of data presented. The blue diamonds are the total time taken for individual runs of the program with the given number of threads along the bottom axis. The red squares and red line show the average time of those multiple runs for each number of threads. Finally, the green triangles give the total time divided by the number of generations needed to reach a solution. Note that the time scale is given in milliseconds, but is presented in a logarithmic scale to account for the large variance in the time required for the program to execute.

Unfortunately, throughout my testing, I was not able to show a scaling speedup as the program was spread across multiple nodes. While much of the computation is spread across many threads, the need to stay closely in sync with each other as well as the large amount of data that must be shared across threads does not allow the program to scale well.

My tests do show that as the program scales across multiple nodes the overall runtime of the program does not go up by any appreciable amount. This demonstrates that the overhead added by UPC and all of the shared data structures does not negatively impact the overall runtime of the program by a factor larger than the factor of increased computational power available to the system. So, while running it across multiple systems does not make it find a solution faster, it doesn't make it any slower either.

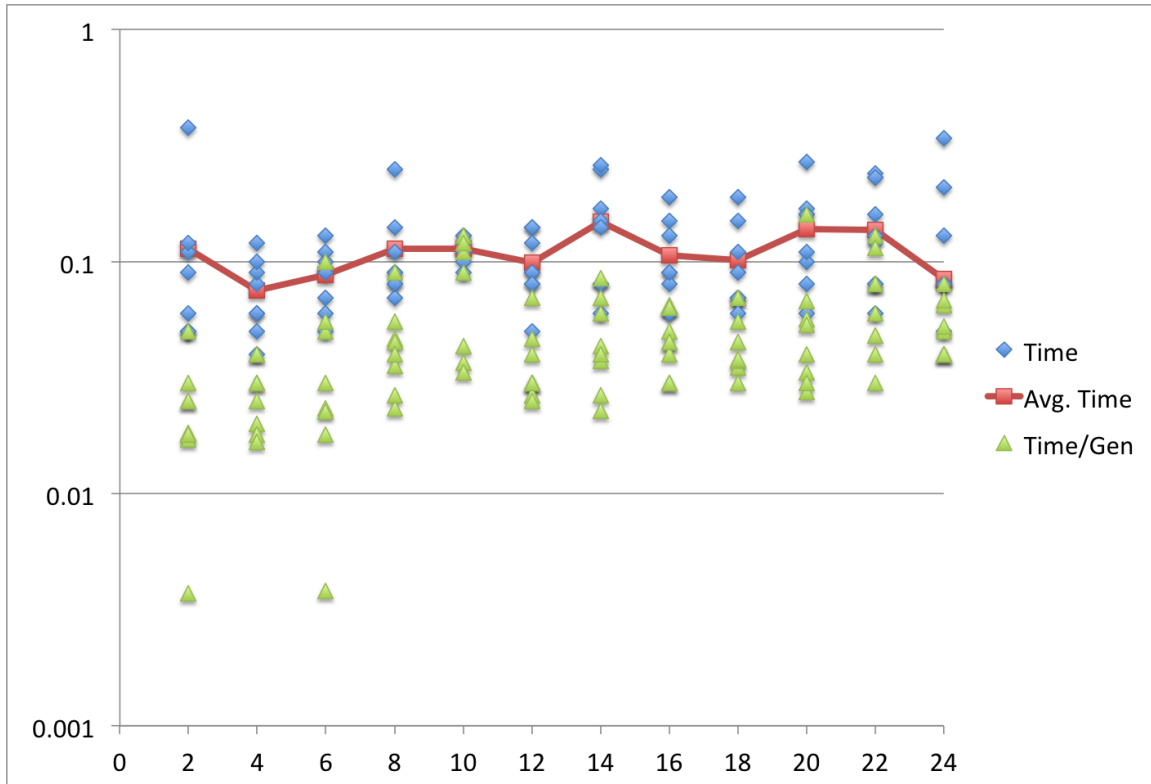


Figure 4: Two Input Binary XOR First Solution, Single Node Scaling

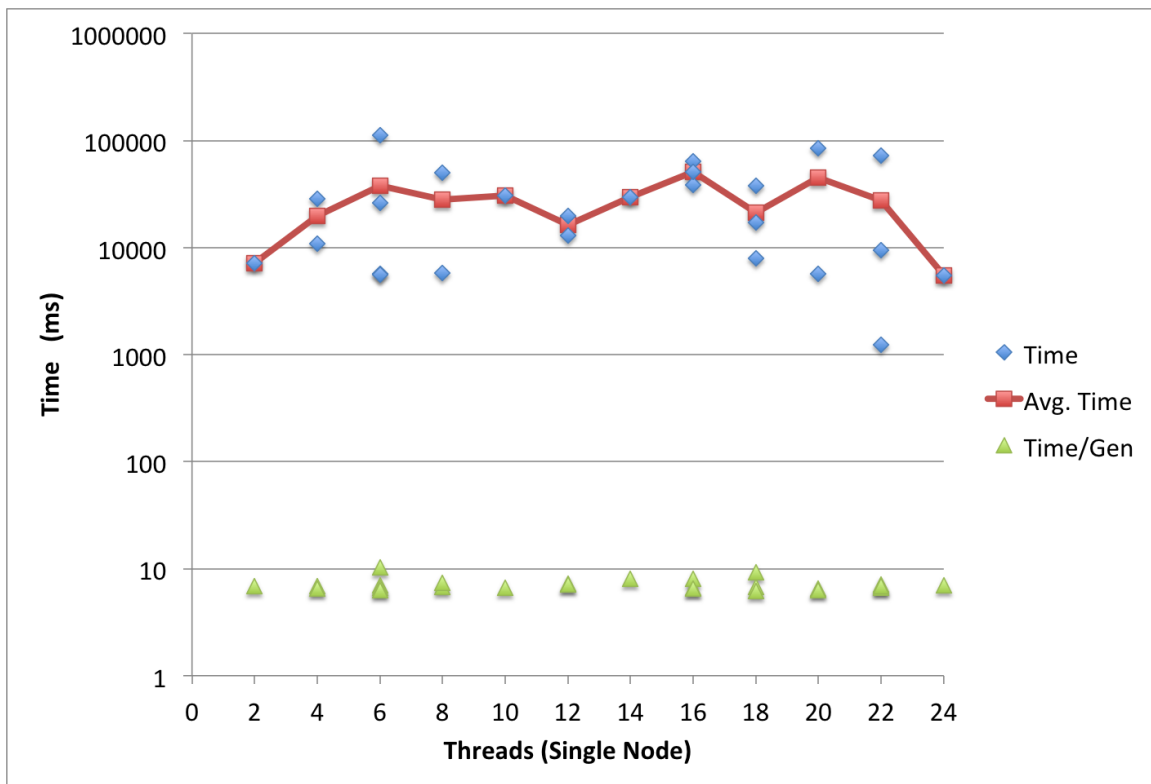


Figure 5: Three Input Binary XOR First Solution, Single Node Scaling

After reviewing these results, I created a couple of additional tests to see if different methods would achieve better scaling results. In these tests, most of the UPC code for sharing data using the PGAS model was removed. Instead, each thread would keep a local population and evolve it independently from the others. However, whenever one thread found a solution, that solution would be presented and all other threads would be stopped. The goal is to achieve faster calculation by simply using multiple threads to reduce the overall number of generations needed on average to find a solution.

For the second test, the same XOR binary function from the first test was used. I ran each test with a specific number of threads 8 times. Then, for the third test, I adapted the function to be a three-input XOR binary function, hoping to get longer runtimes and better results. For this test, I ran each test with a specific number of threads 4 times. As before, the tests where the program didn't give a positive result were excluded in the results. The results of these tests on a single node are presented in Figures 4 and 5.

Unfortunately, as those figures show, I was not able to demonstrate any appreciable scaling speedups across a single node with either test. For the larger problem shown in Figure 5, I did start to see faster times overall as the number of threads increased, even beyond the number of cores on the system. Due to time constraints, I was only able to run the tests 4 times for each number of threads since it could take over 2 minutes per run.

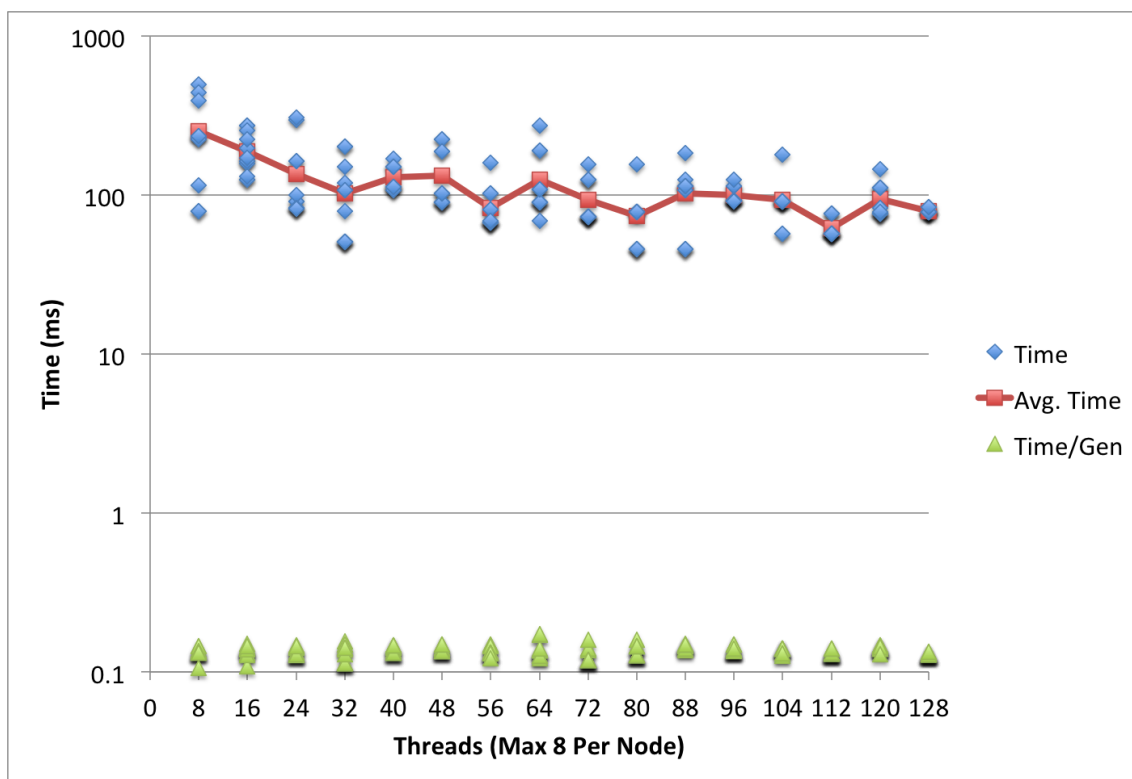


Figure 6: Three Input Binary XOR First Solution, Multi Node Scaling

The final test, shown in Figure 6, is the most interesting. Using the 3 input binary XOR program designed to terminate after any of the threads finds a solution, I ran it on up to 128 threads spread across 16 nodes, with up to 8 threads per node. Each of those tests was repeated 12 times. In this instance, I did see the speed increase I was looking for. On average, with more threads spread across more nodes, a solution was found quicker timewise, and in fewer generations overall, than with fewer threads. This shows that the approach is valid. Unfortunately, the scaling factor is only about 1.2 on average, meaning that each time the number of threads is doubled, the speed is only reduced by that factor.

5. Conclusion and Future Work

Overall, this project was successful in showing that the NEAT framework could be adapted to use UPC to spread the work across multiple threads and multiple nodes. While coding the framework in a language that required mostly static memory allocation for efficiency was difficult, it did yield correct results in the tests performed.

However, through my performance testing I was not able to show any scaling speedups across multiple threads or multiple nodes. The runtimes of the program varied widely due to the nondeterministic nature of evolutionary computing, but on average the runtime of the program was fairly constant or slightly increasing as more nodes or threads were used. However, the average time taken to calculate a single generation held mostly constant across many nodes, demonstrating that my approach did not cause any negative overhead issues as more communication across nodes was needed.

In additional testing, I was able to rework the program to allow each thread to independently work on a local dataset without communication across threads, halting the entire program once one thread found a solution. Again, I did not observe an appreciable increase in the speed of the program as more threads were used on a single node, but spread across multiple nodes I was able to show that the program did terminate faster and using fewer generations on average, meeting my goal.

For the future, I'd like to continue to tweak my implementation of the NEAT framework to be able to solve more problems. I feel that my implementation is still very inefficient, taking many more generations to solve a simple problem than the original creators reported.^[1] Additionally, I plan on continuing to test and tweak the larger three input test to see if I can achieve better results across multiple nodes or with different approaches or seed values generated on each node.

References

- [1] Stanley, Kenneth O., and Risto Miikkulainen. "Evolving Neural Networks through Augmenting Topologies." *Evolutionary Computation* 10.2 (2002): 99-127. Web. <<http://nn.cs.utexas.edu/downloads/papers/stanley.ec02.pdf>>.
- [2] "Berkeley UPC - Unified Parallel C." *Berkeley Unified Parallel C (UPC) Project*. Lawrence Berkeley National Laboratory and University of California, Berkeley, n.d. Web. 24 Apr. 2016. <<http://upc.lbl.gov/>>.
- [3] Fogel, David B. "Revisiting Overlooked Foundations Of Evolutionary Computation: Part I." *Cybernetics and Systems* 41.5 (2010): 343-58. Web. <<http://dx.doi.org/10.1080/01969722.2010.486222>>.
- [4] Hunt, Earl, Marvin Minsky, and Seymour Papert. "Perceptrons." *The American Journal of Psychology* 84.3 (1971): 445.
- [5] UPC Consortium. "UPC Language Specifications." *Berkeley UPC - Unified Parallel C*. Lawrence Berkeley National Laboratory and University of California, Berkeley, n.d. Web. 24 Apr. 2016. <<http://upc.lbl.gov/docs/user/upc-lang-spec-1.3.pdf>>.
- [6] White, B. W., and Frank Rosenblatt. "Principles of Neurodynamics: Perceptrons and the Theory of Brain Mechanisms." *The American Journal of Psychology* 76.4 (1963): 705.
- [7] Olazaran, M. "A Sociological Study of the Official History of the Perceptrons Controversy." *Social Studies of Science* 26.3 (1996): 611-59. Web. 4 May 2016. <<http://www.jstor.org/stable/285702>>.
- [8] SethBling. "MarI/O - Machine Learning for Video Games." *YouTube*. YouTube, 13 June 2015. Web. 04 May 2016. <<https://www.youtube.com/watch?v=qv6UVOQ0F44>>.
- [9] SethBling. "MarI/O Source Code (NEATEvolve.lua)." *Pastebin*. N.p., n.d. Web. 03 May 2016. <<http://pastebin.com/ZZmSNaHX>>.
- [10] Stanley, Kenneth. "NNRG Software - NEAT C Original." *NNRG Software - NEAT C Original*. N.p., n.d. Web. 04 May 2016. <http://nn.cs.utexas.edu/?neat_original>.
- [11] Walsh, Richard. "Parallel Programming with Unified Parallel C (UPC)." *High Performance Computing Center*. The City University of New York, n.d. Web. 24 Apr. 2016. <<http://www.csi.cuny.edu/cunyhpc/pdf/UPC.pdf>>.
- [12] "User Portal." *TACC Stampede User Guide*. Texas Advanced Computing Center, n.d. Web. 24 Apr. 2016. <<https://portal.tacc.utexas.edu/user-guides/stampede>>.