High-Performance Distributed Computation

Hammurabi Mendes and Aidan O'Neill
{hamendes,aioneill}@davidson.edu
Davidson College Davidson, NC 28035 U.S.A.

Overview

We develop a library of functions that performs high-performance computation across many nodes; these functions are particularly suitable to machine learning applications. Our library abstracts the difficulties inherent to performing computationally intensive tasks across many nodes while maintaining sufficient flexibility that developers need not conform to an entirely new paradigm.

Motivation

In the era of big data, the problems that scientists face rely on vast amounts of data that cannot be stored in a single machine. Rather, scientists must use clusters of computers to hold the data requisite to solving their complex problems. This makes challenging problems significantly more difficult, as now, in addition to solving the question at hand, one must worry about communication between machines, communication that both complicates and occasionally slows down programs. We create a library that allows programmers to access memory on other machines as if it were local, removing this layer of complexity.

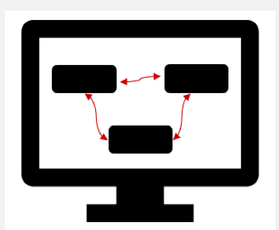

Figure: A sample multithreading environment.

Methods

Multithreading: Multiple agents working within the same processor on the same problem. An example multithreading environment is pictured in the lower left of the poster.

Multiprocessing: Multiple computers working on the same problem. An example multiprocessing environment is pictured below.

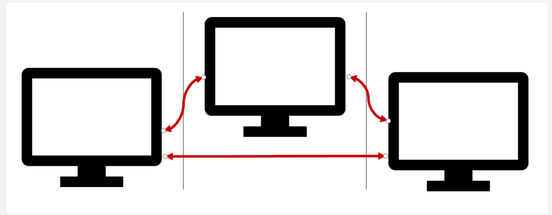

Figure: A sample multiprocessing environment.

We used both the multithreading and multiprocessing programming paradigms to maximize performance.

Benefits:

- **High speeds**
- **Increase in computing power**

Difficulties:

- **Communication faults**
- **Underusing available resources**

Monte Carlo Tree Search

One problem that our system is intended to make easier to distribute is Monte Carlo Tree Search (MCTS). MCTS allows an agent to learn a policy by repeatedly interacting with its environment through "playouts". Playouts begin in the current game state - the root node for our search - and end when the game has completed, i.e. a terminal node has been reached. Based on the outcomes of these playouts, the MCTS algorithm selects the move which appears most beneficial. MCTS has 4 steps.

Selection: We begin from the current game state and select nodes from our search tree until we reach a leaf node.

Expansion: We create a child node from this leaf node.

Simulation: We complete a random playout from the child node.

Backpropagation: We use the result of the simulation to update our search tree.

An example of the first three steps is pictured below.

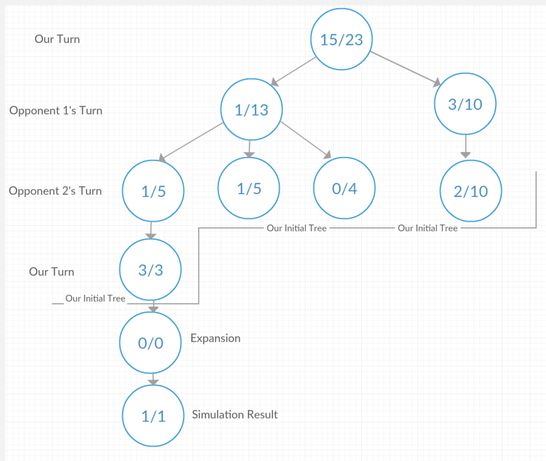

Figure: An example MCTS Tree with three players

Monte Carlo Tree Search (cont.)

Each time we are faced with a choice between children to select, we select the child node that maximizes the formula

$$\underbrace{\frac{Q(v_i)}{N(v_i)}}_{\text{Exploitation}} + \underbrace{c\sqrt{\frac{\log N_v}{N(v_i)}}}_{\text{Exploration}}$$

This gives a good balance between searching relatively unexplored trees and trees with high potential.

Exploitation: the number of wins over the number of times we've visited a child node; it's a measure of the expected utility of a child node.

Exploration: an expression that increases as we explore a subtree

$\mathbf{N_v}$: the number of times we have visited a parent node

$\mathbf{N(v_i)}$: the number of times we have visited the current node

c: Our exploration constant.

Conclusion

Our library of functions works efficiently and precisely while maintaining the required flexibility. We have verified that the library works in a distributed environment. The results are satisfactory to the point that we plan on continuing the work by applying it to an MCTS algorithm informed by two neural networks.