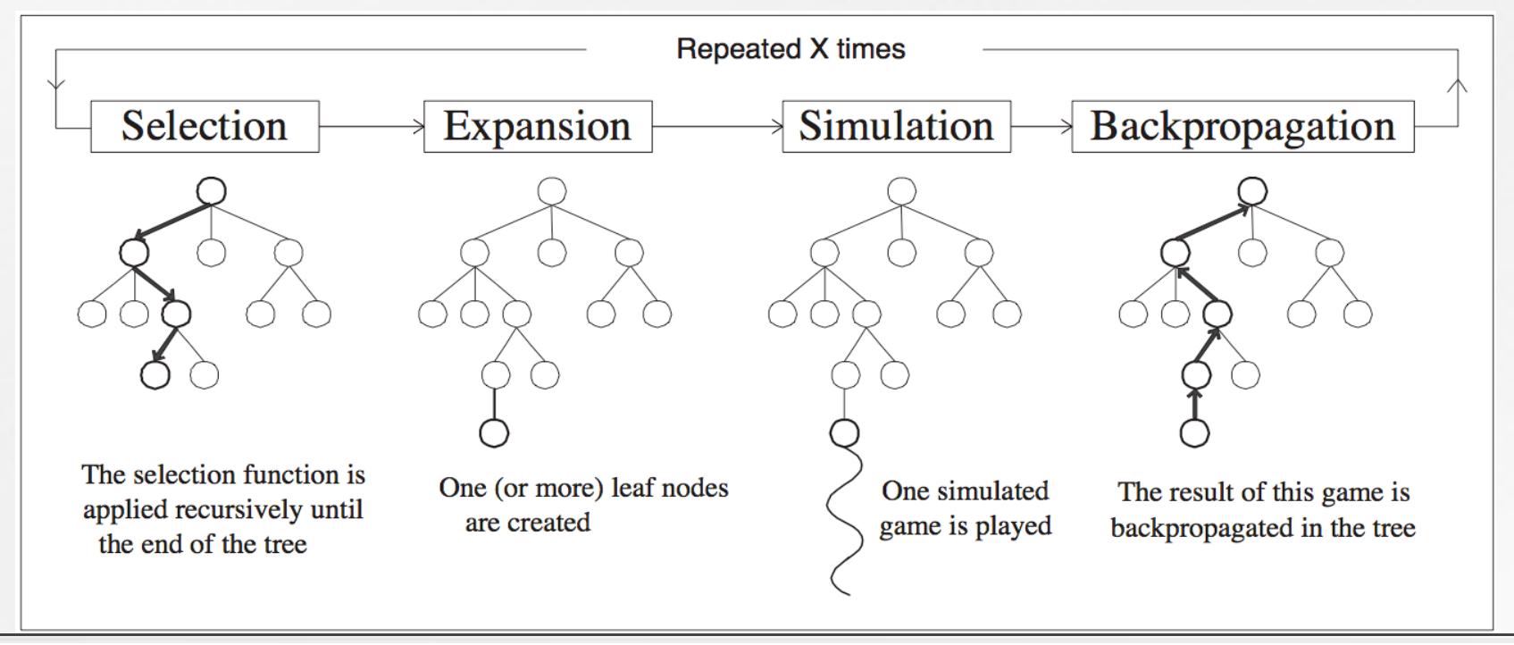
# Root and leaf parallelization: Adapting Monte Carlo tree search for a cluster

Robert Dygert[1], Travis Mandel[2], Jens Mache[3], Peter Drake[3]
[1] University at Buffalo, Buffalo, NY; [2] Carnegie Mellon University, Pittsburgh, PA; [3] Lewis & Clark College, Portland, OR

## Monte Carlo Tree Search

Monte carlo tree search allows Go playing programs to evaluate positions without the need for a static board evaluation function. MCTS builds a tree where the root is the current board position and each child node represents a potential move.

Games are randomly simulated from the leaves of the tree. After the simulation is completed the winner is calculated. Using this information the win and run count is updated in the tree. This allows us to decide whether to run more simulations from the current node or move to another leaf node. After a leaf node has had enough simulations, it is expanded so that following moves can be individually analyzed.


Repeated X times
Selection
Expansion
Simulation
Backpropagation
The selection function is applied recursively until the end of the tree
One (or more) leaf nodes are created
One simulated game is played
The result of this game is backpropagated in the tree

## Parallelization

Parallelization allows for performance gains by using multiple computers at once. Typical computing is done in a sequential fashion, where each command is executed after the next. However, this is often slow; we would rather run multiple computer for the same amount of time to get get more results, since MCTS improves with more computational power.

Some previous work has focused on parallelizing MCTS on a shared-memory, also known as a multi-core, computer. This means that two programs can be running separately on the same computer, as each uses a single core. We call these programs "threads". Even though a computer has multiple cores, it usually has a single, unified RAM. This means that each thread needs to share the same block of memory. The good news is that this makes communication between threads very easy, as one thread can just change a memory location and the other can see it changed. The bad news is that this severely limits the number of processors one can use.

We, however, use a Java-based non-MPI client-server model.

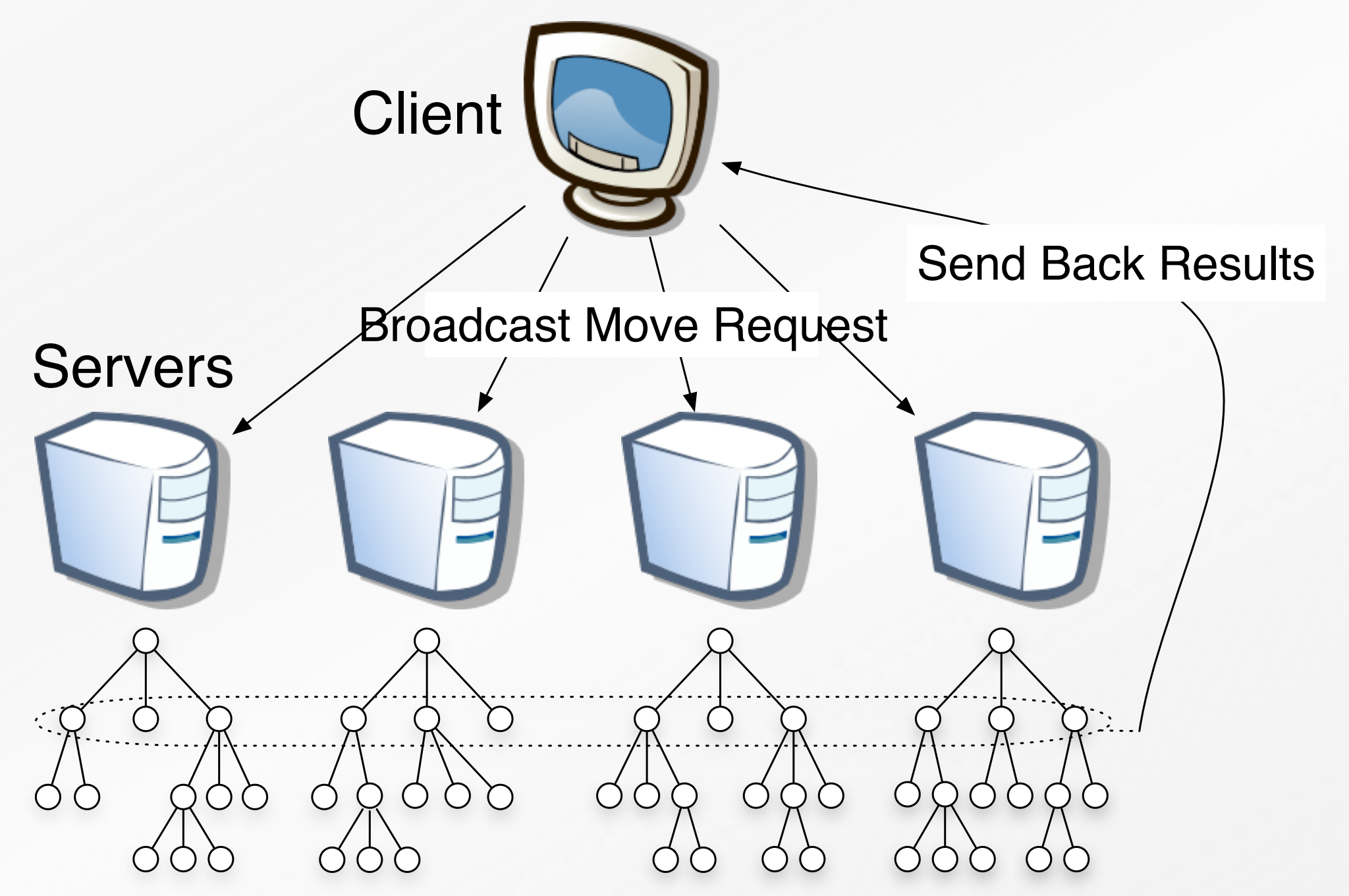
In the client-server model we have one computer, called the client, which is commanded by a user to solve a specific problem, and n other computers, called servers, which wait to receive information from the client. The client divides the problem into small parts that can be computed independently of each other. For each part, it sends a message to one or more servers giving it the information it needs to compute that subproblem. Each server receives a message and begins to work on the subproblem. When the problem is finished, the server replies back to the client with the solution and is ready to receive another subproblem. In this way the client is able to combine the results of the subproblems to solve the larger problem.

Parallelization can often induce many slowdowns into a given program, called overhead. Here are three of the most common:

- Communication:
  - the packets have to be created
  - travel through the network to reach the recipient
  - checked for corruption.
- Wasted Computation:
  - Many problems are time-sensitive.
  - Late information is useless.
- Idle Time:
  - Computers should always be working for maximum efficiency.
  - Coordination of tasks to keep all servers busy is a challenging problem

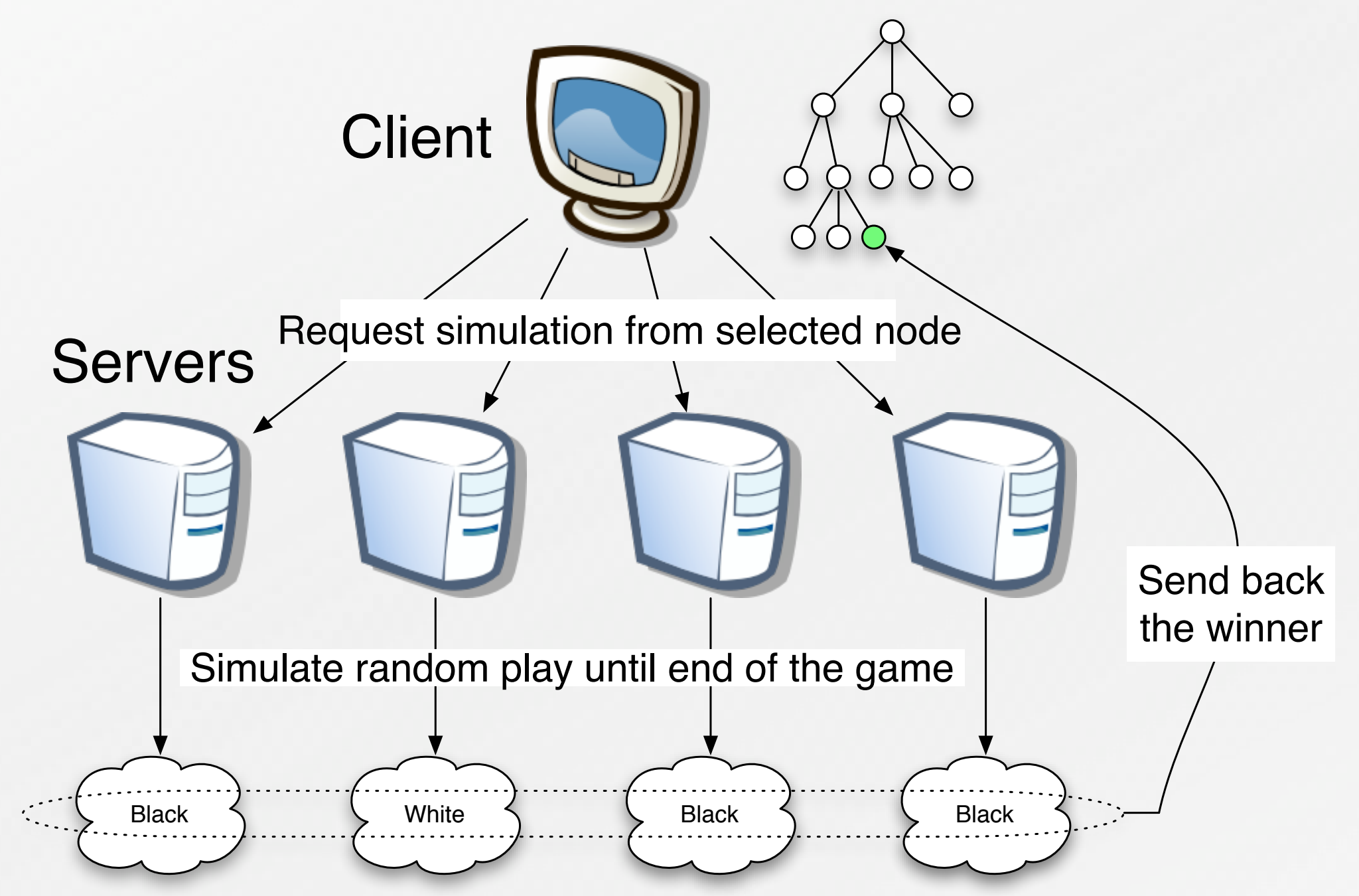
## Parallel MCTS Algorithms

### Root Parallelization



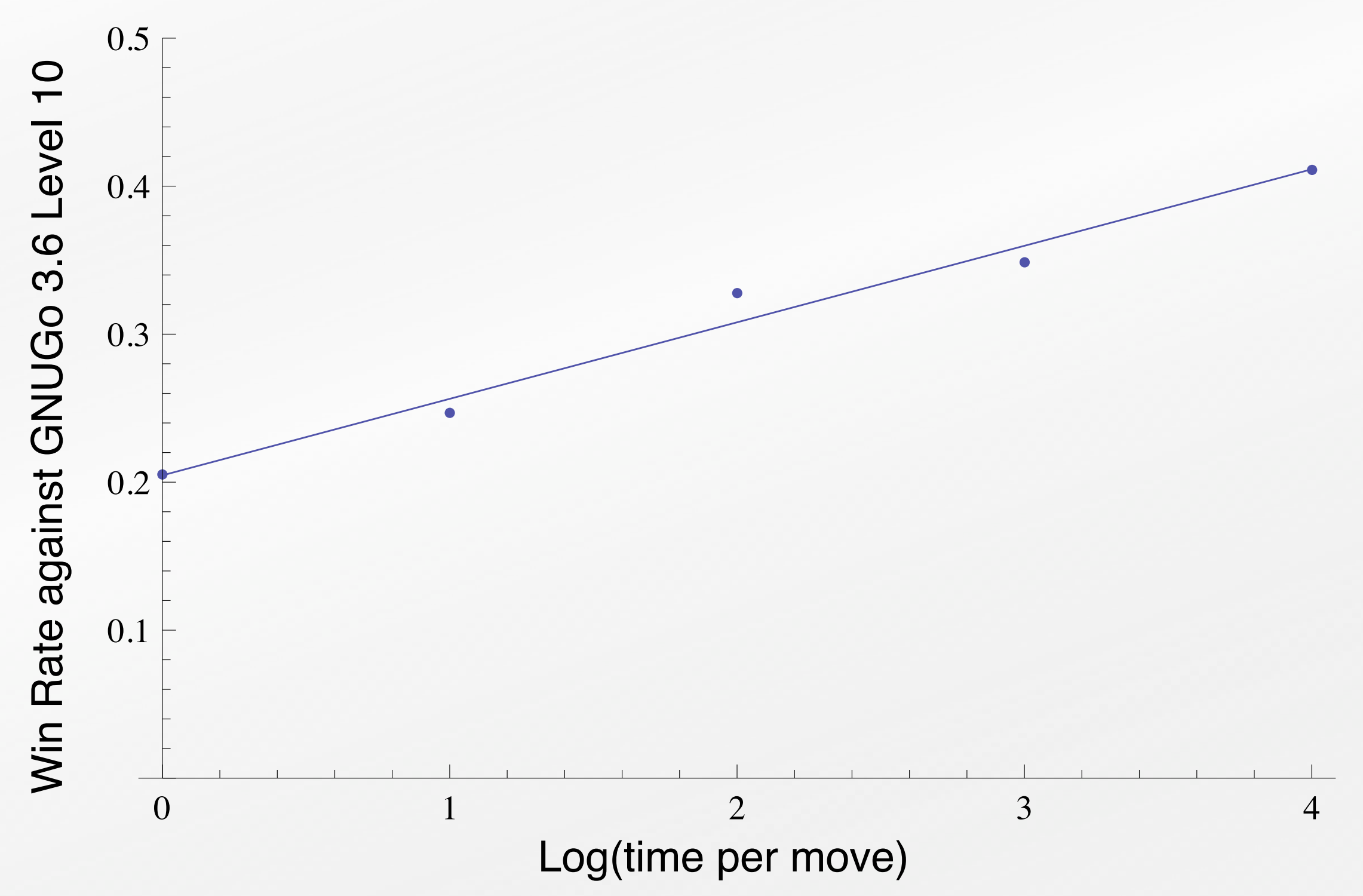
With root parallelization the client is rather simple, and the servers are more complex. The client only requests the scores for each of the next possible moves and aggregates this data, whereas the servers handle all the steps of the MCTS algorithm. Each server performs MCTS independently, creating its own tree, and sending back results when time is almost up.

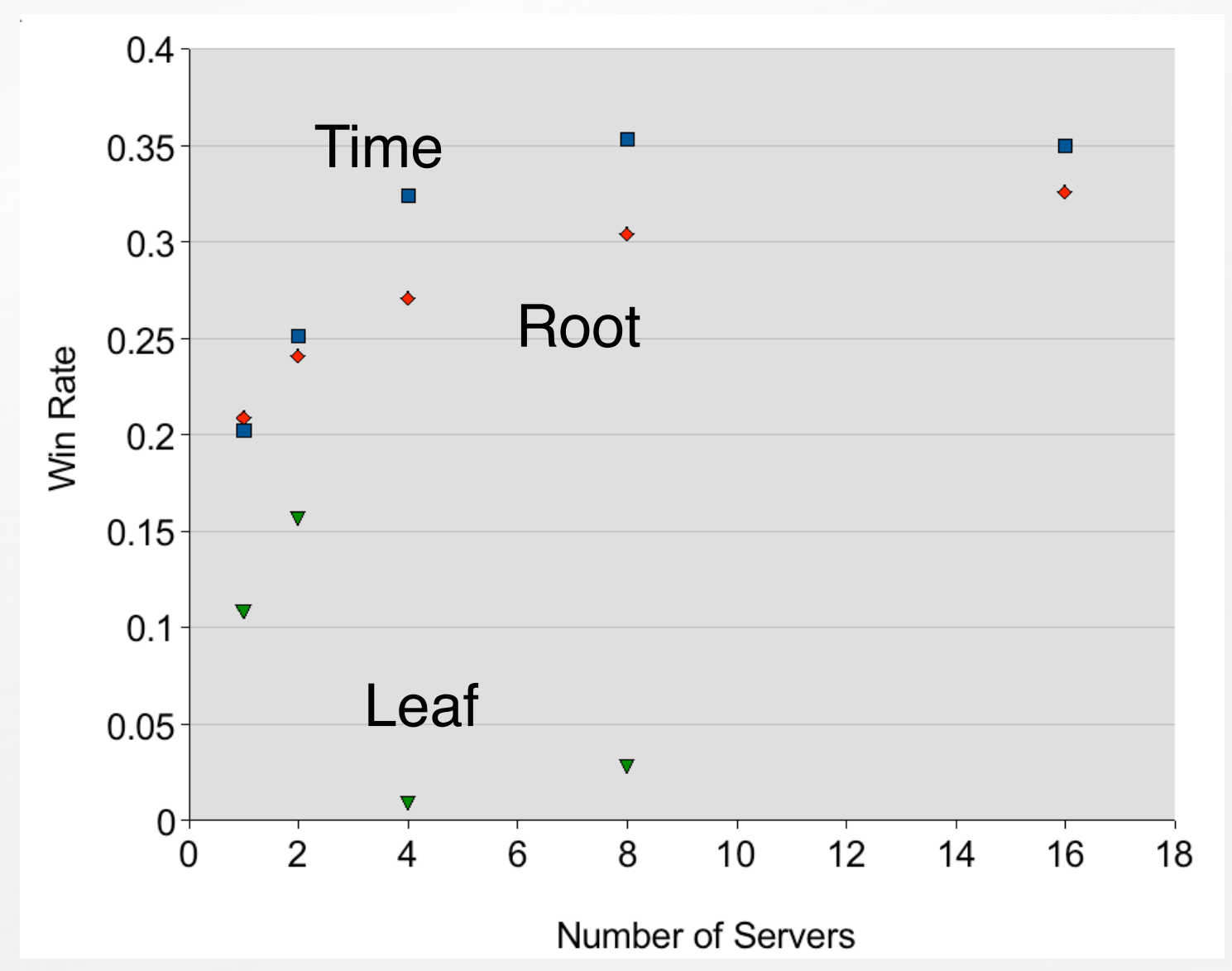
### Leaf Parallelization


Client
Servers
Request simulation from selected node
Send back the winner
Simulate random play until end of the game
Black
White
Black
Black

With leaf parallelization, the servers are simple, while the client is more complex.The client must maintain the tree and perform the selection, expansion, and backpropogation steps of the MCTS algorithm. The server only takes the node selected by the client, simulates random games, and reports the winning color. The client then incorporates the information into the tree. In the basic version of leaf parallelization the client waits for all of the servers to respond before it continues with the MCTS algorithm and selects a new node on which to run trials.

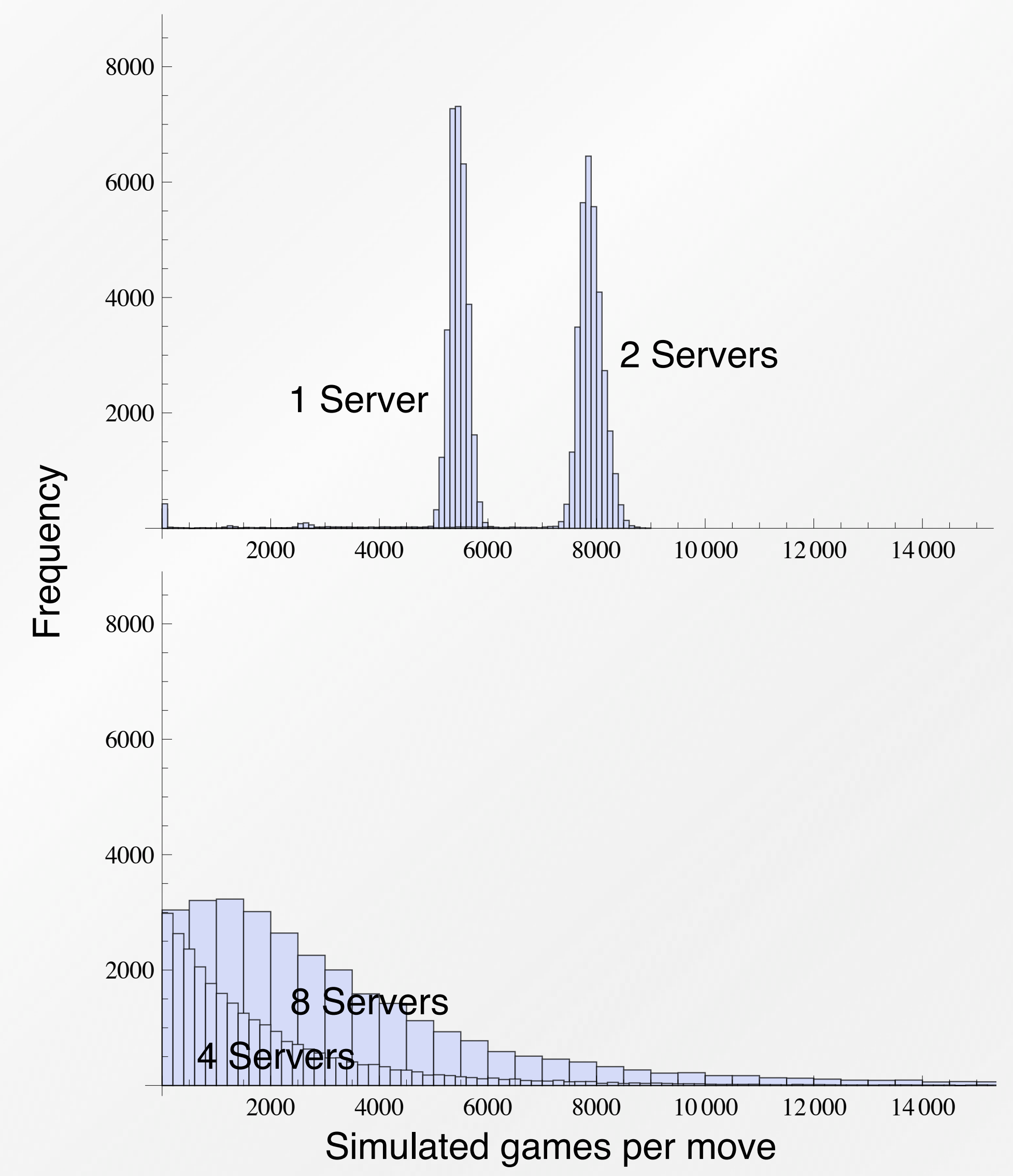
## Results





Root parallelizes well, given that we are able to get the same percent of wins against GNUGo with 16 servers as with running the sequential version for 4 times the length. Leaf, however, does not scale well, given that we see a sharp decrease in win rate from 4 to 8 servers.

## Results



As the number of servers increases from 1 to 2, the number of simulated games increases leading to better play. With more than 2 servers, the distribution of playouts is biased towards zero, which resulted in poor play due to insufficient information.

The reason for this is that as the number of servers increases, the probability that a single server takes a very long time increases. So since we wait for the longest server, we have to wait a much longer time every time we send a playout request, causing fewer successful playouts.

## Future Work

Investigating Leaf No-Wait, a new version of leaf wait that does not wait for every server to reply but rather takes one reply off a queue, broadcasts another request, and then selects another move.

Adapting Tree Parallelization, one of the best shared-memory parallelizations, to a cluster. The key part of this method is that each processor works on a different part of the tree, but the difficult part is that this method requires multiple processors accessing the tree at the same time.

**Further Information**
tmandel@andrew.cmu.edu
meep116@gmail.com

**Acknowledgements**
Willamette Valley REU-RET
National Science Foundation
Lewis & Clark College

## References

1) Cazenave, T. and Jouandeau, N.: On the Parallelization of UCT, Proceedings of the Computer Games Workshop 2007 (CGW 2007) (van den Herik, J., Uiterwijk, J., Winands, M. and Schadd, M., eds.), MICC Technical Report Series, No. 07-06, Universiteit Maastricht, pp.93–101 (2007).

2) Chaslot, G. M., Winands, M. H. and van den Herik, J. H.: Parallel Monte-Carlo Tree Search, Proceedings of the 6th International Conference on Computer and Games (2008).