ECE496 Final Report

Distributed Minimax

Project No. 307

Authors:

Steven Blenkinsop (123456789)

Jonathan Gray (998198588)

Supervisor:

Prof. Jorg Liebeherr

Administrator:

John Taglione

Submitted March 19, 2015

1 Executive Summary

2 Group Highlights and Individual Contributions

3 Table of Contents

4 Background and Motivation

Some of the earliest software written for programmable computers sought to enable them to play abstract strategy games such as chess and checkers [1]. In the late 1940s, artificial intelligence pioneers like Claude Shannon, John von Neumann, and Alan Turing independently developed game-playing algorithms that relied on the concept of minimaxing, an optimal algorithm for game-playing which searches for the most advantageous move assuming the opponent plays perfectly [2].

Parallelization of general-purpose computation on a graphics processing unit (GPU) has been popular for the last 15 years, but GPU frameworks and hardware remain complicated and expensive compared to central processing unit (CPU) cluster technology [3]. Recently, researchers have developed algorithms to parallelize the minimax algorithm on GPUs and on CPU clusters [4][5]. Given the availability and low cost of commodity server hardware and cloud computing platforms, and the relative ease with which commodity-grade processors can be replaced or added to a cluster, CPU clusters are often preferable to GPUs in achieving scalable parallelization of general-purpose computing [6]. For these reasons, we have focused on CPU clusters for this project.

This project aimed to produce a software framework for distributing the execution of the minimax algorithm over an arbitrarily-sized cluster of computers. The framework allows an end-user to describe the game tree and provide heuristics for evaluating the state of a game that has not yet terminated, while abstracting away the mechanics of the distributed nature of the computation. It also provides a simple interface to add, remove, and replace computers participating in the minimax search.

The primary motivation for this project was the group members interest in game artificial intelligence and distributed systems.

5 Project Goal

The goal of this project was to create a software framework with which a user can distribute the evaluation of a user-defined perfect-information zero-sum sequential game using the minimax algorithm over an arbitrary number of computers.

6 Project Requirements

6.1 Functional Requirements

The design shall:

1. correctly implement the minimax algorithm for user-defined perfect-information zero-sum sequential games
2. allow for distributing evaluation over a number of nodes between 2 and 10[1]; and
3. allow the user to set a time limit (at least 1 second) on the computation

6.2 Constraints

The resulting software must be executable on all of:

1. Mac OS 10.9 or above
2. Windows 8.1 or above
3. Ubuntu 14.04 or above

6.3 Objectives

The resulting software should:

1. evaluate more game states per unit of time compared to an equivalent implementation of the minimax algorithm running on single computer
2. efficiently utilize processors of different speeds

7 Final Design

Section 7.1 provides an overview of the final design and the interactions between the various modules.

Following that, Section 7.2 provides a more detailed description about the major design components and processes described in the system overview.

[1] TODO

7.1 System-Level Overview

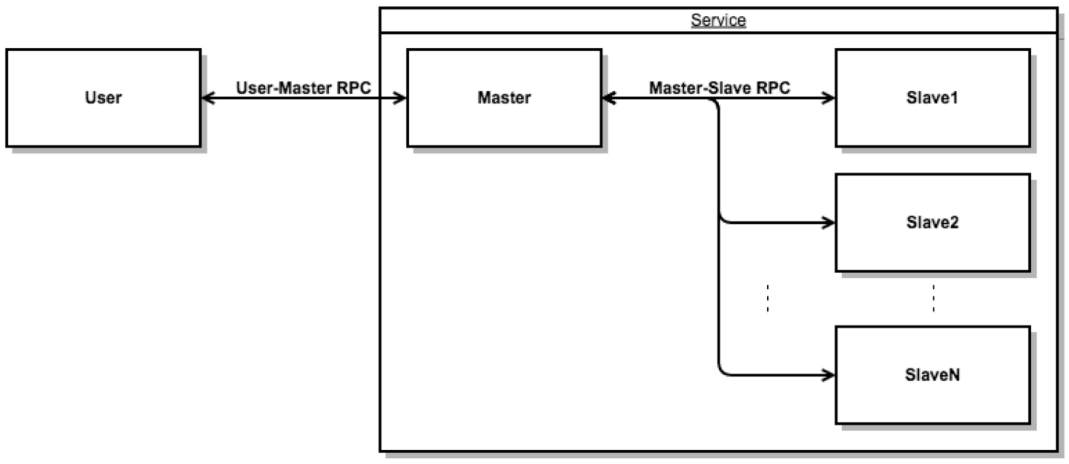

Figure 1: Component diagram for the distributed minimax service showing the separation of *user*, *master*, and *slave* processes

A single *master* process and a number of identical *slave* processes communicate with each other to provide a network *service* to a *user* process. As illustrated in Figure 1, the *user* and the *slaves* communicate only with the *master* (over the *User-Master RPC* and *Master-Slave RPC*, respectively) and not with each other.

On receiving a request from the *user* over the *User-Master RPC*, the *master* creates a *job pool* by breaking down the Minimax algorithm into discrete jobs that can be performed concurrently, and distributes these jobs to the slaves over the *Master-Slave RPC*. The *slaves* execute their jobs and return the results of their analyses to the *master*. The *master* consolidates these results into a single *best move* suggestion and returns this to the *user*.

7.2 Module-Level Descriptions

This section gives a formal description of the major modules referenced in the previous section. Each module's inputs and outputs are listed along with a brief description of the module's purpose. The final design of each module is also described, and areas for potential improvement are noted in some cases.

7.2.1 User-Master RPC

Module Inputs:

- Serialized game state
- Time limit

Module Outputs:

- Suggestion for best move

Module Description:

This module represents network interaction between the *user* and the *master*.

The user makes a request to the master with a serialized game state and a maximum time allowed. The Master node is expected to confer with the slaves and respond within the time limit with its suggestion for a move to be played.

Final Design:

The open-source *Protobuf*[2] library was used to describe the message formats and generate RPC code. For an exact specification of the RPC, see Section 13.1 in Appendix C.

7.2.2 Job Pool Creation

Module Inputs:

- Game state to be analyzed
- A set of slaves that are available to do work

Module Outputs:

- A set of jobs that can be worked independently and concurrently

Module Description:

This module is a component of the *master*.

Given a game state, this module generates a set of jobs that can be worked independently. The number of jobs created will depend on the number of slaves available to do work. In particular, it should produce at least as many jobs as there are slaves, so no slave is left with no work to do.

[2]TODO

Final Design:

A *job* in the final design is a game state that is a descendent[3] of the game state to be analyzed. This module will examine the direct descendents of the viven game state, then each of their direct descendents, etc., until the number of descendents is greater than the number of slaves. When that condition is met, these game states are returned as jobs.

7.2.3 Job Allocation

Module Inputs:

- A set of jobs that can be worked independently and concurrently
- A set of *slaves* that are available to do work

Module Outputs:

- Exactly one *slave* assigned to each job

Module Description:

This module is a component of the *master*.

Given a set of jobs and a set of *slaves*, this module decides which *slaves* will work on which jobs.

Final Design:

The final design assigns jobs to slaves in a round-robin fashion. For example, if there are 10 jobs and 7 slaves, then the first three slaves will be assigned two jobs each, and the other four slaves will be assigned only one job. This method ensures that the *number* of jobs assigned to each slave is fairly even, but is agnostic about *which* jobs are assigned to which slave.

This is a major area for improvement. Given more time, this module should attempt to measure the relative importance of jobs and the effectiveness of slaves at performing work, and assign more important jobs to more effective slaves.

[3]The "children" of a game state are the game states that can be reached by making a single move. For example, an empty tic-tac-toe board has nine children, representing a piece played in each of the nine empty spots. The "descendents" of a game state are its children, its children's children, etc.

7.2.4 Master-Slave RPC

Module Inputs:

- A set of jobs
- Time Limit

Module Outputs:

- Result of analysis for each job

Module Description:

This module represents network interaction between the *master* and the *slaves*.

The master delivers to each slave a set of jobs to be performed and a maximum time allowed. The slaves return their results to the Master within the allowed time.

This module deals primarily with serialization and communication with the master node, while the *Slave Work* module below deals primarily with performing analysis.

Final Design:

The open-source *Protobuf*[4] library was used to describe the message formats and generate RPC code. For an exact specification of the RPC, see Section 13.1 in Appendix C.

7.2.5 Slave Work

Module Inputs:

- A set of jobs
- Time Limit

Module Outputs:

- Result of analysis for each job

[4]TODO

Module Description:

This module performs executes the jobs that were allocated by the *master*. This involves running the Minimax algorithm on subsets of the game tree. The results of the jobs are returned to the *Master-Slave RPC* module to be returned to the *master*.

7.2.6 Result Consolidation

Module Inputs:

- Results of analysis for each job

Module Outputs:

- Suggestion for best move

Module Description:

This module combines the results of the analyses that the *slaves* performed to produce a single best move suggestion to return to the user.

8 Testing and Verification

9 Summary and Conclusions

10 References

11 Appendix A: Gantt Chart History

12 Appendix B: Original Validation and Acceptance Tests

13 Appendix C: Code Samples

This appendix includes raw samples of code from the project to serve as further information for interested developers, and as proof of work for ECE496 administrators.

13.1 Protobuf Specifications

User-Master RPC

```
message DoWorkRequest {
    required bytes state = 1;
    required uint64 timeLimitMillis = 2;
}

message DoWorkResponse {
    required bytes move = 1;  // suggestion for next state
}

service UserService {
    rpc DoWork(DoWorkRequest) returns (DoWorkResponse);
}
```

Master-Slave RPC

```
message GetWorkRequest {
    message Result {
        required bytes state = 1;
        required int32 value = 2;  // should match Value type in game/
        required int64 numStatesAnalyzed = 3;
    }
    // optionally return the results of a previous workload
    repeated Result result = 1;
}

message GetWorkResponse {
```

```
    repeated bytes state = 1;
    required uint64 timeLimitMillis = 2;
}

service SlaveService {
    rpc GetWork(GetWorkRequest) returns (GetWorkResponse);
}
```