

An Investigation of Parallel Approaches in Creating a Computer Backgammon Player

by:

Eddie Scholtz

May 14, 2007

Department of Electrical Engineering and Computer Science

Massachusetts Institute of Technology
Cambridge, MA 02139

Abstract

The goal of this project was to implement a strong backgammon player using a parallel approach on the Cell Broadband Engine processor. This involved researching existing backgammon programming techniques, examining how they could be adapted to the Cell architecture, and implementing them. One of the hurdles of this project was the lack of a well documented, lightweight C backgammon framework. This was overcome by implementing such a framework from the ground up. Included in the framework is a neural network training method using temporal difference (TD) learning. Although the resulting player did not fare well against pubeval, one of the strongest linear neural network players, tremendous improvement was seen when using a 1-ply look-ahead.

1 Introduction

During the course of this project I have completed a backgammon implementation from the ground up. This included researching the techniques used to create a strong backgammon player such as training board evaluation functions and improving move choices through expectiminimax look-ahead. A linear neural network was trained to evaluate the quality of the board for a given player using temporal difference learning. The network showed promising results when combined with a 1-ply expectiminimax look-ahead aimed at minimizing the maximum expected loss. Both of these techniques were implemented without the use of existing AI or backgammon code. The source code has been released under an open source license and should provide a great springboard for future backgammon projects.

2 Background

One of the first strong backgammon programs was BKG 9.8. It was created by Hans Berliner at Carnegie Mellon University in the late 1970s. Berliner used a static evaluation function to analyze boards and determine the best move. The evaluation function was a linear function, a weighted sum of board features. The function was created using expert knowledge and the weights were tuned through trial and error. It is also important to note that the weights were adjusted by the program as the game progressed in order to account for changing importance of board features. BKG 9.8 was strong enough to defeat the reigning world champion, Luigi Villa. However, post game analysis indicated that BKG 9.8 made mistakes at several points in the game and was lucky in that Villa did not capitalize on them.

The next major advance in backgammon AI, implemented by Gerald Tesauro, used a form of reinforcement learning called temporal difference learning. This method uses a neural network to learn the board evaluation function. The advantage of this approach is that the system can learn by playing against itself rather than using an existing dataset (a large number of boards for which you already know the best move for a given state). Also, using a neural network with hidden layers allows it to approximate nonlinear functions. This means that various combinations of board features can be taken into account. Temporal difference learning has been used to produce master level programs. However, producing this level of play requires learning over hundreds of thousands of games of self play.

The final technique used by backgammon programs is look-ahead. Look-ahead is difficult because of the uncertainty of future dice rolls. As a result, the search tree has a huge branching factor (estimated to be ~400 - compared to ~10 for checkers and ~35-40 for chess). This uncertainty requires using an extension of minimax method of decision making called expectiminimax. Most existing backgammon programs are only capable of searching 3 or 4 turns ahead and generally apply some sort of search pruning.

3 Implementation and Results

Since there are no high-quality existing backgammon frameworks written in C, it was necessary to first create one. This involved implementing the rules of the game (including checking that a move is legal), finding all of the valid move sequences for a given turn, displaying the board, and collecting input. In order to create a computer player, it is necessary to have a way of evaluating how good a given board is for a given player. Knowing the current state of the game and all possible move sequences for the current turn, the board evaluation function can be used to choose the best move. The first evaluation function I used was *pubeval*, a benchmark evaluator created by Gerald Tesauro that plays at the intermediate level. *Pubeval* consists of two single-layer neural networks with 122 input nodes and 1 output node. Each neural network is used for a different game state: racing or contact.

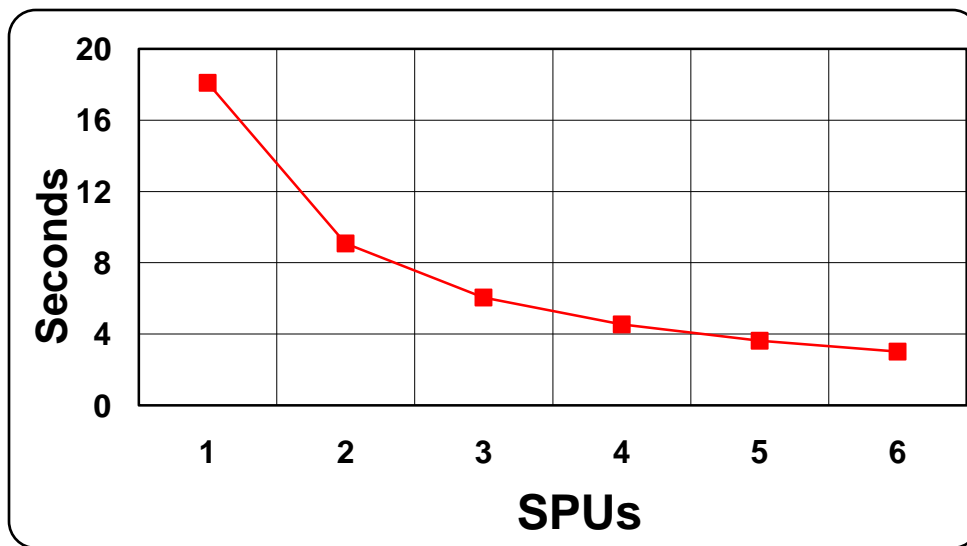


Figure 1: The time it takes to evaluate one million boards (non-simd) as a function of the number of SPUs used. The speedup achieved was very close to linear.

The next goal was to try to improve the player by implementing an expectiminimax look-ahead. The first attempt used a three step method: (1) Create the move tree. (2) Evaluate the leaf nodes. (3) Moving up the tree, determine the best move. Step 2 was easily to parallelize

using the 6 SPEs and produced good scaling (Figure 1). Converting the board evaluation code to utilize SIMD instructions improved performance by about 32% (Figure 2). This number is much less than expected because a good deal of time was spent compressing and uncompressing the boards before they were sent to the SPU. The compression code was not easily convertible to utilize SIMD instructions.

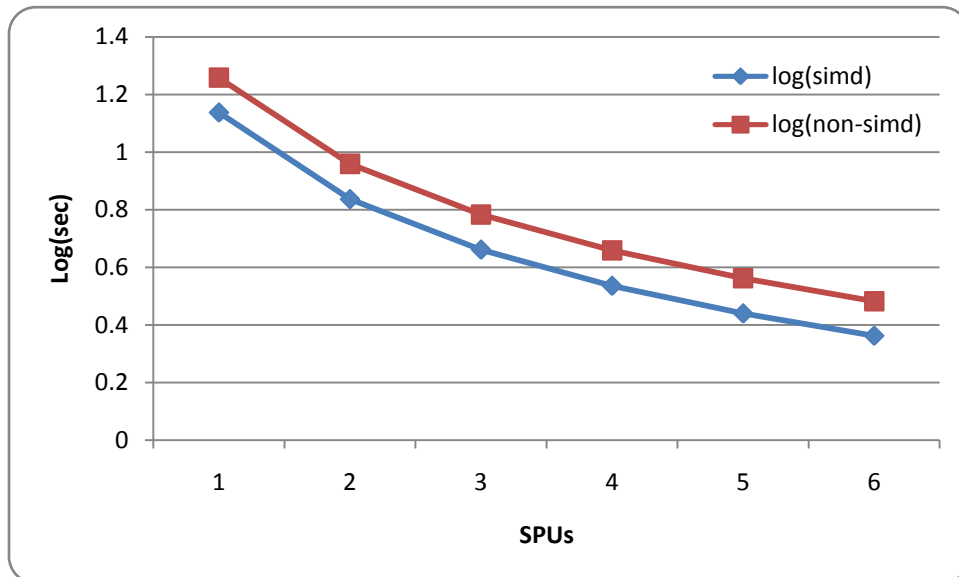


Figure 2: Converting the board evaluation function to utilize SIMD instructions resulted in an approximate 32% speedup.

Unfortunately, steps 1 and 3 of the expectiminimax method were difficult to parallelize and performed poorly in terms of both time and space when executed serially. The reason steps 1 and 3 were necessary is that pubeval is not zero-sum which greatly complicates things. At this point it was decided that the best approach would be to switch to a zero-sum evaluator. A zero-sum evaluator would greatly reduce the memory overhead of the expectiminimax search, which would be simplified to: (1) PPU finds the N possible move sequences for the given turn. (2) PPU sends N/6 move sequences to each SPU. (3) SPU performs depth-first expectiminimax search and returns an equity for each of the N/6 move sequences it receives.

At this point the two options were to create a zero-sum evaluator or find an existing one. GNU Backgammon appeared to be a good source for the evaluator. The project is open source, written in C, and known to play very well. However, the code was poorly documented, difficult to integrate, and was not structured to support SIMD instructions. Unfortunately, a good deal of time was wasted trying to integrate the GNU Backgammon board evaluator before the decision was made to create a new board evaluator.

Temporal Difference (TD) Learning was the method I used to train the board evaluation function. This method was developed by Richard Sutton and successfully implemented in the backgammon domain by Gerald Tesauro to produce a strong player. The interesting aspect of this approach is that the evaluator can improve by playing against itself, without any existing knowledge of the game. The neural network I used for training consisted of 198 input nodes, one output node, and no hidden layers. For each of the 24 backgammon board positions, there were 8 input nodes (4 for each player). The first input node indicated if the player had one piece at the position, the second input indicated two pieces, the third three pieces, and the fourth the number of pieces greater than three. Also included as input were two nodes to indicate the player whose turn it was, two nodes to indicate how many pieces each player had on the bar, and two nodes to indicate how many pieces each player had on the bear-off. The single output node multiplied each input by a weight and summed the results. The sum was passed through the sigmoid function: $1 / (1 + e^{-x})$, which produced a value between zero and one. If the output was closer to zero, it would indicate a good board position for player A and if the output was closer to one, it would indicate a good board position for player B.

The key to training the neural network is figuring out the best way to update the weights. Temporal difference learning uses a sequential method which is based on successive predictions.

$$\Delta w_t = \alpha (P_{t+1} - P_t) \sum_{k=1}^t \lambda^{t-k} \nabla_w P_k$$

In this equation, w_t is the weight vector that specifies the network and P_t is the network output for the input vector x_t (x_t specifies the current board state). $\nabla_w P_k$ is the gradient of the network output with respect to the weights. For the linear case, $\nabla_w P_k$ is equal to x_t . The learning rate is specified by α , which was set to 0.1 as suggested by Tesauro. Finally, λ is the temporal difference parameter. It determines the degree to which successive predictions influence the change in weight. This value was set to 0.7. The final reward signal is based on which player wins the game. If player A wins it is set to zero and if player B wins it is set to one.

At the start of training, the weights were randomly initialized with values $[-0.5, 0.5]$.

Training took place over the course of tens of thousands of self-play games. In order to determine if the training was successful, the network was pitted against a player that randomly chose moves. After the network was trained for 30,000 games it could beat a random player 74% of the time (Table 1). After 60,000 games, the network won 90% of games versus the random player. It is important to note that even a player choosing random moves will win occasionally due to the stochastic nature of the game. For example, pubeval, an intermediate level player, wins 90% of games against a random player.

Player A	Player B	$\frac{\text{Games won by A}}{\text{Total Games}}$
nn30k	random	.74
nn60k	random	.90
pubeval	random	.90
nn30k	pubeval	.23
nn60k	pubeval	.25
nn30k	nn30k	.50
nn60k	nn60k	.60

Table 1: The results of pitting two computer players against each other for 1,000 games. The evaluation functions trained for this project begin with nn followed by the number of games they were trained for. All evaluators in this table were 0-ply (no look-ahead).

Note that even though the networks trained for this project performed well against a random player they performed poorly against pubeval. The important point is that they were zero-sum so they could be used to test the value provided by using look-ahead. (Another point of interest is that nn60k managed to win 60% of games when playing against itself. The cause of this requires further investigation.)

In backgammon, one turn by one player is called a ply. Using no look-ahead, meaning only considering the possible moves for this turn, is considered 0-ply. Considering the possible moves for the current turn as well as the possible moves the opponent can commit next turn is considered 1-ply look-ahead. Even a 1-ply look-ahead is very expensive. If we assume that there are usually 20 legal moves for the current turn (although there can be several hundred legal moves if the player rolls doubles and has a wide spread of checkers), there are 21 possible dice combinations that the opponent can roll, and 20 legal moves for each dice combo that the opponent rolled, then the expectiminimax look-ahead needs to evaluate 8,400 boards. In addition, the overhead of finding all of the legal moves for a given turn is expensive as well.

In order to determine if the look-ahead was worthwhile to implement in parallel, it was necessary to first implement it sequentially and evaluate the game play improvement. The findings, summarized in Table 2, show that significant game play improvement was achieved by a 1-ply look-ahead. The 1-ply look-ahead almost doubles the number of games one by the 0-ply look-ahead when playing pubeval.

Player A	Player B	$\frac{\text{Games won by A}}{\text{Total Games}}$
nn30k 1-ply	nn30k 0-ply	.80
nn30k 1-ply	pubeval	.42
nn30k 0-ply	pubeval	.23

Table 2: The results of 1000 games played between two computer players. 1-ply look-ahead showed significant improvement over the 0-ply evaluator.

4 Future Work

Although this project was aimed at creating a strong backgammon player using the Cell processor, most of the effort was spent building a backgammon framework and the components necessary to begin parallelizing the backgammon player. This should provide a good starting point as I continue working on the project this summer. In addition, other developers working in the backgammon domain should be greatly benefited since the code was released as open source.

A good first step would be getting the look-ahead running on the SPEs. This should provide the performance necessary to do a 2-ply or 3-ply look-ahead. Given the increase in game play performance when moving from a 0-ply to 1-ply look-ahead, a 2-ply or 3-ply look-ahead could potentially perform at the level of pubeval or better.

The second goal should be to improve the board evaluation function. Tesauro has shown that this can easily be done by adding a hidden layer to the neural network, thus making it non-linear. Adding such a layer greatly increases the computational complexity of the neural network, potentially making it a good application for Cell.

A third potential application of Cell is in implementing Monte-Carlo search to pick the best move. The Monte-Carlo method works by randomly sampling future game states as opposed to the expectiminimax search which samples all of the possible game states for the next few turns. For each of the 20 or so possible moves sequences for the current turn, Monte-Carlo commits the move sequence and then plays out the rest of the game thousands of times using random dice rolls. The move sequence which results in the greatest number of games won for the current player is chosen to be the best move. Galperin and Tesauro found good results using this method. Their implementation used 32 computer nodes, each of which was capable of about

100 MFlops. One of their limitations was in the quality of the evaluation function they could use. The Cell processor is capable of a theoretical 256 GFlops. One Cell processor or several networked together could be used to test the Monte-Carlo approach with more complex evaluation functions that Galperin and Tesauro were not able to use.

Acknowledgements

Portions of the backgammon source code were developed by Mike Fitzgerald during IAP 2007. Specifically, he developed a graphical board display using X.

References

Galperin, Gregory and Tesauro, Gerald. On-line Policy Improvement using Monte-Carlo Search. *Advances in Neural Information Processing Systems*. 1996.

Sutton, Richard. Learning to Predict by the Methods of Temporal Differences. *Machine Learning*, 3, 9-44 (1988).

Tesauro, Gerald. Practical Issues in Temporal Difference Learning. *Machine Learning*, 8, 257-277 (1992).

Tesauro, Gerald. Temporal Difference Learning and TD-Gammon. *Communications of the ACM*, March 1995.