Parallel Naive MCTS for Go AI Report

Thai Flowers, Special Topics in Computer Science Fall 2017

Overview

This document outlines the development process, development goals, and performance of a simplistic naive artificial intelligence for playing the board game “Go”. In particular utilizing a parallel variant of the monte carlo tree search with upper confidence bound (MCTS with UCT) method we explored in a previous class but this time for a game with much more complicated tactics, and this time in C. These terms will be explained in the background material. The simplest possible parallelization method delivered an approximate doubling in number per games in the given time interval, and more complicated parallelization resulted in a very slight performance gain in addition to that. Project structure and goals will explain why weak scaling was chosen and the parallelization methods considered. The development process will explain the methods attempted, and the intermediary performance increases. Finally, the performance analysis of the current codebase is self explanatory.

Background Material

Go is an ancient chinese game of strategy with incredibly simple rules that takes minutes to learn, but with tactics so deep it takes a lifetime to truly master. Despite being an additive game, the huge 19x19 game board gives hundreds of possible moves per turn, even well into the midgame, which makes the state space for the game truly enormous. Additionally the difference between a good move and a bad move can be a single space on the board, a difference a human learns to identify visually and instinctively.

From this it can be understood that it has historically been difficult for computers to compete with human players, but in recent years (esp. 2015-2017) new developments in AI techniques have enabled Go software to surpass professional players. The current most powerful Go software AlphaGo was developed by Alphabet Inc (the parent company of Google), and at it’s core is an advanced form of MCTS aided by a pair of convolution neural networks that perform computer vision based analysis.

The software developed for this project uses the most simplistic version of monte carlo tree search in common use: MCTS with upper confidence bound (or UCT for short). It is a heuristic of min-max evaluation, and for some games can actually converge to the min-max result given enough time. It constructs a tree of game states, with the initial game state given to it at the root. Each iteration of MCTS expands the state tree by one leaf, and back-propagates results back to the root of random play based on the new leaf. After any iteration the tree can be queried for the best move so far seen. More information on MCTS can be found at [1], including a detailed description of the four main phases of MCTS.

Project Structure and Goals

The AI constructed in this project is exceedingly simple and does not handle incorrect input well. It was intended as a backend component of a more user friendly system that generates the input sent to the AI, and sends the output into an appropriate interface (in particular implementing the Go Text Protocol). Likewise the goal of being an AI humans can potentially play against naturally came with the requirement that it is guaranteed to run for a short time, for example 10 seconds or less. MCTS lends itself well to this since it can be terminated after any iteration. The bulk of the program therefore constitutes a while loop that calls the MCTS algorithm until the time limit is exceeded.

The goal for the parallelism and analysis section of this project was to increase the average number of games simulated in the fixed time window. That is we focused on weak scaling.

As can be seen in [2] there are four main methods for parallelizing MCTS. Using the paper as guidance we decided to explore the two tree parallelism methods, for the following primary reasons: 1) Our initial concept using MPI is analogous to the single mutex version, 2) according to the paper the multiple mutex version chooses moves approximately as strong as root parallelism, and 3) root parallelism is embarrassingly parallel and thus forbidden for use in this project. Additionally, an attempt was made to use intrinsic functions / vector operations to speed up methods underpinning game simulation, but the results were horrendous.

Development Process

Initial Plan

The initial plan for this project was to use MPI to implement an analog to a the thread-pool design pattern found in some programming languages (Java was our particular inspiration). The primary process, process 0, was to hold the tree, and in a loop perform descents, pick a move, send it to an available process, then receive the results from whatever process has completed its MCTS iteration and perform back-propagation. The idea was that the simulations would be slow enough that the completions would be staggered, and (given enough processes in the pool) process 0 would always have a process ready and waiting when it reached the reply stage. That is process 0 would always be busy, and locks/synchronization would not be necessary since only it would have access to the tree

However, during development of the serial version it was obvious that the lightweight (mostly random/completely naive) playouts were far too fast for that architecture, that even in shared memory lock contention would be a bottleneck. Since network communication is even slower, then obviously shared memory would be preferred.

Actual Development

The serial version of the go_ai program was developed using Kjeld Petersen's binary matrix method for verifying move correctness and enforcing game play rules, and it can be found at [3]. All binmatrix methods, play_move and bensons_pass_alive are derived from his work, and the binmatrix methods were written in a pseudo object oriented style. Similarly, estimate score is a simple implementation of Bouzy's 5/21 algorithm as originally published in [4] and found in the Gnu Go documentation [5]. The only other code borrowed from another source is Brian Kernighan's bit counting algorithm which can be found in "The C Programming Language 2nd Edition" and many other locations, and is named count_set_bits in our codebase.

As noted before, initial serial development revealed that shared memory was best approach for this project. Using openmp a simple global mutex (critical section secured) version was produced in a matter of minutes and a handful of new lines of code. However, the difference in performance was negligible; on the order of 12-20k simulations serial and 14-24k parallel (with 32 threads). Some experimentation revealed that a dynamic array was the bottleneck, as it was being reallocated and copied over every 10 simulations, but the simulation was running thousands of simulations. Setting the growth rate to 80 resulted in serial code running on the order of 12-20k and parallel on the order of 33-36k (with 32 threads).

Next an attempt was made to use Intel intrinsics to speed up the many short for loops in the binmatrix methods. After much difficulty to get intrinsics working on gpu2, including switching to gcc44 as our compiler, the impact was beyond underwhelming. Vectorizing only four of the binmatrix methods more than quartered the number of completed games. An accurate analysis was not performed as the negative impact was exceedingly obvious, so the line of thought was completely abandoned. The code is still in the codebase but must be enabled by defining the symbol "VECTOR".

The last avenue explored was the fine grained locking, aka local mutex, version of tree parallelism. That is each tree node has its own lock. Also following the work in [2], we tried a simple version of virtual loss, which lessens lock contention by encouraging threads following the same tree descent to diverge as early as possible. This method only yielded a 2k increase compared to the global mutex version. Attempts to use atomic pragmas in place of locking for modifications to tree node members during back-propagation increased this performance boost to 5k or more, but lead to occasional segfaults (around 1 out of 10 runs).

Key Files in Project

1. go.c -- Contains the binmatrix methods and our implementation of play_move, bensens_pass_alive and several binmatrix printing methods. Grouped together because of shared constants and we prefer cluttered source files over cluttered source directories.
2. main.c -- The fine grained locking openmp implementation of MCTS.
3. main_omp.c -- A copy of main storing the global mutex openmp version of the program.
4. main_serial.c -- As above but for the serial version of the program.

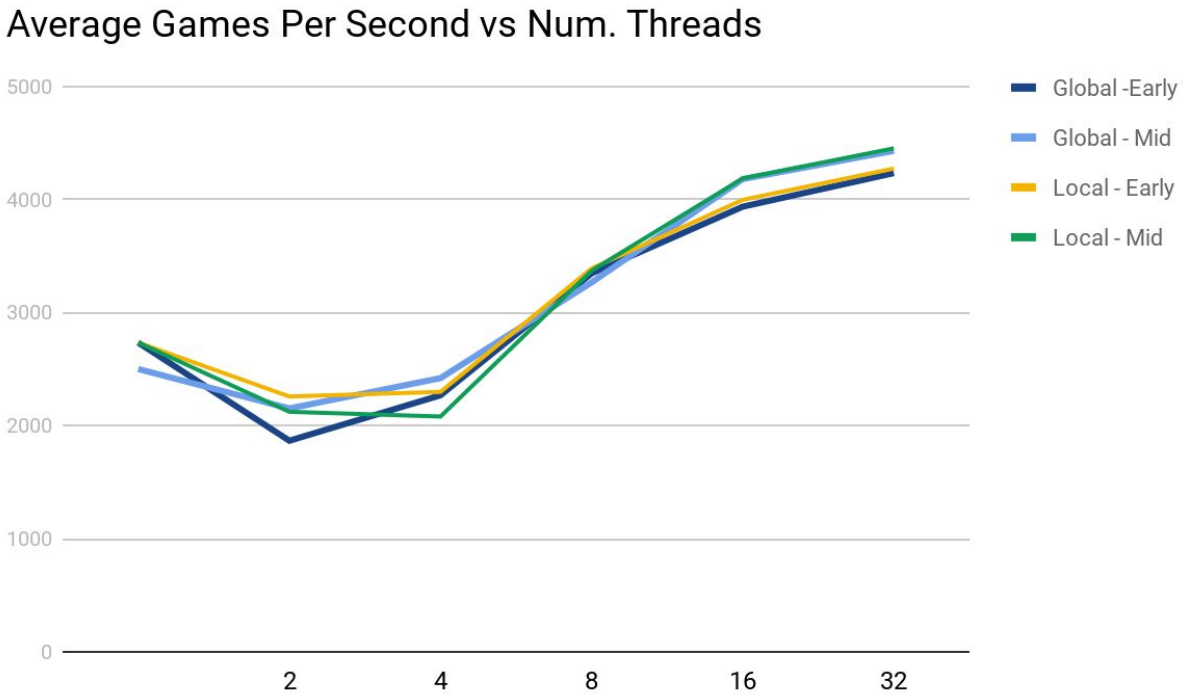
5. go.h -- Header for go.c. Also includes key constants and types.
6. links.txt -- Web address of several resources used/referenced.
7. notes.txt -- A round log of ideas and development process.
8. Various board.txt files, input to main program used during testing.

Performance Analysis of Current Codebase

In the following, each version of the program and associated input/settings was run for 10 seconds five times. The resulting number of simulations run averaged together, and the games per second measures are based on this average. Each version was tested on linprog4, as it is somewhat faster than gpu2 and bridges is overkill for our purposes.

Early game state gathered from [6] and is found in file early_board.txt. Mid game state gathered from [7] and is found in file mid_board.txt.

Program	Settings / Threads	Avg. # games Early game	Avg. games/sec Early game	Avg. # games Mid game	Avg. games/sec Mid game
Serial	Default	27356.4	2735.60	25015.6	2501.56
Global mutex	2	18674.6	1867.46	107653	2153.06
Global mutex	4	22694.4	2269.44	24210.6	2421.06
Global mutex	8	33476.0	3347.60	32678.4	3267.84
Global mutex	16	39374.6	3937.46	41814.0	4181.4
Global mutex	32	42348.2	4234.82	44315.4	4431.54
Local mutex	2	22586.4	2258.64	21232.6	2123.26
Local mutex	4	22997.0	2299.70	20814.8	2081.48
Local mutex	8	33913.6	3391.36	33714.8	3371.48
Local mutex	16	39979.6	3997.96	41874.8	4187.48
Local mutex	32	42747.4	4274.74	44535.4	4453.54
Serial	Vector ops, on gpu2	4762.2	476.20		



Firstly the most striking result is that using vector operations are dramatically slower than would be expected. We posit that the root cause is using unaligned input data, and cost of cache coherence being much higher than any performance boost from any vectorization of the many short for loops at the core of the binmatrix methods.

The graph above makes it clear that this program is far from maximally efficient: it is nowhere close to the optimal weak scaling curve (the line y=x). However, with enough cores then the algorithm is much more effective than the serial version. Even at 32 cores the diminishing returns have not resulted in an absolute maximum on the graph.

Also it is readily apparent that the local mutex is, in general, only around 2000 games better than the global mutex version, and with 32 cores using the mid game data the performance boost is negligible.

One last remark, the performance penalty for using mutexes with a low number of cores is much lower for local mutexes than for global mutexes. However, in every examined case it was worse than serial.

Concluding Remarks

Possible Improvements

A. Modify the core program to accept current game score.
B. Write a wrapper script that implements the Go Text Protocol so that go_ai can be used to play real games.

C. Bensen's Pass alive and random play produces many zero score games, it could be helpful to ignore such games.
D. Heavier playout/less naive playouts, such as the convolution networks used by Alpha Go
E. Attempt to use read-copy-update or similar techniques to implement lockless MCTS traversal/updates.
F. Build a version using OpenMP and MPI utilizing dual parallelism techniques for example run root parallelism across nodes, and tree parallelism on each node, or run tree parallelism across all nodes (original MPI thread-pool design) and run leaf parallelism on each node.
G. Using read-write locks in the multiple mutex tree parallelism version.

Summary

From this project it can be concluded that tree parallelism with local mutexes is only slightly more effective than using a single global mutex. This project can also be used as a jumping off point for much more interesting and robust AI and parallelism techniques. Which is good news, since even though computer go can now beat grand masters, the base techniques explored here are worth study for application for other games (including real time games), and go AI efficiency can still be improved (AlphaGo and similar aren't at peak power on consumer devices). To bring this to a close, if we take nothing else away from this project, we learned that openmp can work miracles on certain codebases, and that vector operations/intrinsic functions should either be left to the experts, or only used after reading the manual.

Resources

1. http://jeffbradberry.com/posts/2015/09/intro-to-monte-carlo-tree-search/
2. Guillaume M.J-B. Chaslot, Mark H.M. Winands, and H. Jaap van den Herik: Parallel Monte-Carlo Tree Search, Computers and Games: 6th International Conference, 2008
3. https://senseis.xmp.net/?BinMatrix
4. B. Bouzy, Les ensembles flous au jeu de Go Actes des Rencontres Françaises sur la Logique Floue et ses Applications LFA-95, Paris, France (1995), pp. 334-340
5. http://sporadic.stanford.edu/bump/gnugo/gnugo_14.html
6. https://senseis.xmp.net/?BouzyMap
7. https://senseis.xmp.net/?MiddleGameExercise3