Parallel Game Tree Search

Tsan-sheng Hsu

徐讖昇

tshsu@iis.sinica.edu.tw

http://www.iis.sinica.edu.tw/~tshsu
Abstract

- Use multiprocessor shared-memory or distributed memory machines to search the game tree in parallel.

- Questions:
  - Is it possible to search multiple branches of the game tree at the same time while also getting benefits from the searching window introduced in alpha-beta search?
  - What can be done to parallelize Monte-Carlo based game tree search?

- Tradeoff between overheads and benefits.
  - Communication
  - Computation
  - Synchronization

- Techniques
  - For alpha-beta based search algorithms.
  - Lockless transposition table.
  - For Monte-Carlo based search algorithms.

- Can achieve reasonable speed-up using a moderate number of processors on a shared-memory multiprocessor machine.
Comments on parallelization

- Parallelization can add more computation power, but synchronization introduces overhead and may be difficult to implement.

- Synchronization methods
  - Message passing, such as MPI
  - Shared memory cells
    - Avoid a record becoming inconsistent because one is reading the first item, but the last item is being written.
    - Memory locked before using.
  - It may be efficient to broadcast a message.

- Locking the whole transposition table is definitely too costly.
  - The ability to lock each record.
  - Lockless transposition table technique.

- A global transposition table v.s. distributed transposition tables.
Speed-up (1/2)

- **Speed-up**: the amount of performance improvement gotten in comparison to the amount of hardware you used.
  - Assume the amount of resources, e.g., time, consumed is $T_n$ when you use $n$ processors.
  - Speed-up = $\frac{T_1}{T_n}$ using $n$ processors.
- **Speed-up** is a function of $n$ and can be expressed as $sp(n)$.
  - **Scalability**: whether you can obtain “reasonable” performance gain when $n$ gets larger.
- Choose the “resources” where comparisons are made.
  - The elapsed time.
  - The total number of nodes visited.
  - The scores.
  - …
- Choose the game trees where experiments are performed.
  - Artificial constructed trees with a pre-specified average branching factor and depth.
  - Real game trees.
Speed-up (2/2)

- Three different setups for experiments.
  - Use the sequential algorithm $P_{\text{seq}}$ for the baseline of comparison.
  - Use the best sequential algorithm $P_{\text{best}}$ for the baseline of comparison.
  - Use a 1-processor version of your parallel program $P_{1,\text{par}}$ as the baseline of comparison.
    - It is usually the case that $P_{1,\text{par}}$ is much slower than $P_{\text{best}}$.
    - It is often the case that $P_{1,\text{par}}$ is slower than $P_{\text{seq}}$.
  - Use an optimized sequential version of your parallel program $P_{1,\text{opt}}$ as the baseline of comparison.
    - It is also usually the case that $P_{1,\text{opt}}$ is slower than $P_{\text{best}}$.

- Choose the game trees where experiments are performed.
  - Artificial constructed trees with a pre-specified average branching factor and depth.
  - Real game trees.
Amdahl’s law

- The best you can do about parallelization [G. Amdahl 1967].
- Assume a program needs to execute $T$ instructions and and $x$ of them can be parallelized.
  - Assume you have $n$ processors and an instruction takes a unit of time.
  - Parallel processing time is
    \[ \geq T - x + \frac{x}{n} + O_n \geq T - x. \]
    where $O_n$ is the overhead cost in doing parallelization with $n$ processors.
  - Speed-up is
    \[ \leq \frac{T}{T - x}. \]
- If 20% of the code cannot be parallelized, then your parallel program can be at most 5 times faster no matter how many processors you have.
- Depending on $O_n$, it may not be wise to use too many processors.
Load balancing and speed-up factor

- **Load balancing**
  - The ratio between the amount of the largest work on a PE and the amount of the lightest work on another PE.
  - Good load balancing is a key to have a good speed-up factor.

- **Speed-up factor**: ratio between the parallel version with a given number of processors and the baseline version.

- **Is it possible to achieve super linear speed-up?**
  - Super linear speed-up means you can make the code to run $N$ times faster using less than $N$ times about of hardware.
    - Yes, on badly ordered game trees.
    - Not in real game trees with a reasonable good algorithm.
Super-linear speed-up (1/3)

- Sequential alpha-beta search with a pre-assigned window $[0, 5]$:
  - Visited 13 nodes.
Super-linear speed-up (2/3)

- Parallel alpha-beta search with a pre-assigned window $[0, 5]$ on two processors:
  - P2: visited 5 nodes, and then the root performs a beta cut.
  - P1: being terminated by the root after 5 nodes are visited.

![Game Tree Search Diagram](image-url)
Super-linear speed-up (3/3)

- Total sequential time: visited 13 nodes.
- Total parallel time for 2 processors: visited 6 nodes.
- We have achieved a super-linear speed-up.
Comments on super-linear speed-up (1/2)

- Parallelization can achieve super-linear speed-up only if the solution is not found by enumerating all possibilities.
  - For example: finding an entry of 1 in an array.
- If the solution is found by exhaustively examining all possibilities, then there is no chance of getting a super-linear speed-up.
  - For example: counting the total number of 1’s in an array.
- Overhead in parallelization comes from how much work should each processor “talks” to each other in order to decide the solution.
  - Trivially parallelizable: almost no need to talk to each other.
Why is it possible to obtain a super-linear speed-up in searching a game tree using alpha-beta based algorithm?

- Assume some cut-off happens during the execution.
- Parallel algorithms offer a chance of getting a different “move ordering”.
- It is possible to find a solution faster.

It is also possible to get poor speed-up if the “move ordering” of the parallel version is bad.

- You may perform unnecessary work, e.g., searching a branch that will be cut in the future.

For Monte-Carlo based search algorithm, super-linear speed-up may be obtained by trying out different PV branches at the same time.

- Increase the chance of finding the right branch.
Parallel $\alpha$-$\beta$ search

- Three major approaches: depend on what tasks can be parallelized and the model of parallelism.
  - Principle variation splitting (PV split)
    - Central control or global synchronization model of parallelism.
  - Young Brothers Wait Concept (YBWC)
    - Client-server model of parallelism.
  - Dynamic Tree Splitting (DTS)
    - Peer-to-peer model of parallelism.
Classification of nodes (1/2)

- Classify nodes in a game tree according to [Knuth & Moore 1975].
Classification of nodes (2/2)

- **Type 1 (PV):** principle variation.
  - Nodes in the leftmost branch.
  - PV nodes needs to be searched first to establish a good search bound.
  - After the first child is searched, the rest of its children can be searched in parallel.

- **Type 2 (CUT):** cut nodes.
  - Children of type-1 and type-3 nodes.
  - Because children of a cut node may be cut, it is not wise to perform searches in parallel for children of a cut node.

- **Type 3 (ALL):** all nodes.
  - The first branch of a cut node.
  - All children of an all node need to be explored.
  - It is better to search these children in parallel.
Algorithm $PVS$:
- Execute the first branch to get a PV branch $n_1, n_2, n_3, \ldots, n_d$ where $n_d$ is a leaf node.
- for $i = d - 1$ down to 1 do
  - Update the bound information using information backed-up from $n_{i+1}$
  - for each non-PV branch of $n_i$ do in parallel
  - A processor gets a branch and searches
  - Update the bounds when a branch is done
Comments for PV splitting

- **Comments:**
  - Parallelism is done on type-2 branches of a type-1 node.
  - May not be able to use a large number of processors efficiently.
  - **Load balancing** is not good.
    - The ratio between the amount of the largest work on a PE and the amount of the lightest work on another PE.
  - Synchronization overhead is large.
  - When the first branch is usually not the best branch, then the overhead is huge.
  - Achieve a speed-up of 4.1 for 8 processors and 4.6 for 16 processors [Manohararjah ’01].
    - Poor scalability.
    - Limited speed-up: within 5.
  - **Improvements:**
    - When a processor is idle, it helps out a busy processor by sharing its tasks.
    - Observe some improvements, but not much.
Young brothers wait concept (1/2)

- **Concept:** at each node, when the first branch is explored and a bound is obtained, then all the other branches can be executed in parallel.
  - **Split point:** a node whose value of the first branch is known.
  - **Highest split point** of a tree: a split point whose depth is the least.
  - A processor is assigned and **owns** a subtree rooted at a node.
    - *This processor is the server of this subtree.*
  - An idle processor asks a server for a subtree to search.
    - *This processor is a client of this server.*
Algorithm \textit{YBWC}:

- Let $P_1$ own the root of the game tree and begin to search using alpha-beta pruning until the tree is completely searched.
  - During searching, maintain the split point information.
- While the game tree is not searched completely, do
  
  In parallel for each processor $P_i$ do
  
  - If $P_i$ is idle, it looks for server processors with split points.
  - $P_i$ gets a branch from a highest split point and owns this subtree.
  - $P_i$ begins to search using alpha-beta pruning and maintain the split point information.
  - When a subtree owned by $P_i$ has been searched, returns the information to the server processor where it gets the job from.
  - $P_i$ is idle again.
Comments for YBWC

- Comments:
  - Can utilize many processors.
  - Parallelism is done on almost all nodes.
  - It is possible to use non-shared-memory architectures.
    - For example: distributed memory machines.
    - Speed-up: 137 using 256 processors [Manohararjah ’01].
    - Scalability is moderate.
    - Load balancing is not always good.
  - The cost of splitting a node needs to be calculated to avoid splitting small trees.
Dynamic tree splitting (DTS)

- **Concepts:**
  - Peer-to-peer approach so that no one owns any subtree.
  - The processor who finished last on a split point reports the value to the parent of the split point.
  - More criteria for the selection of split points.
DTS: Classification of nodes

- **D-PV**: a node that has the same alpha and beta values as the root.
- **D-CUT**: a minimizing node with the same beta as the root or a maximizing node with the same alpha as the root.
  - On a **MAX** node,
    - if some branches are searched, then the returned values from the branches may update the lower bound.
    - If the lower bound is highered (updated), then it is possible to visit less nodes.
    - Hence it may not be cost effective to parallelize.
    - Note: It takes time to initialize a new job.
  - On a **MIN** node,
    - if some branches are searched, then the returned values from the branches may update the upper bound.
    - If the upper bound is lowered (updated), then it is possible to visit less nodes.
    - Hence it may not be cost effective to parallelize.
    - Note: It takes time to initialize a new job.
- **D-ALL**: any node that is neither D-PV nor D-CUT.
  - Nothing much is known here.
Split point: confidence

- A confidence factor is associated with each D-CUT and D-ALL node.
  - Means the chance of being a node of the specified type.
- If many moves (up to a limit of 3) have been searched at a D-CUT node, then the confidence that it is a D-CUT node decreases.
- If several moves have been searched at a D-ALL node, then the confidence that it is a D-ALL node increases.
Criteria for a split point:

- The node must be of type D-PV, D-ALL with a high confidence or D-CUT with a low confidence.
- If it is a D-PV node, its first branch must have been searched.
- Set thresholds for confidence factors.
  - A D-ALL node with a high confidence factor remains to be a candidate for a split point.
  - Can also fork a D-ALL node with the highest confidence factor first.
  - A D-CUT node with a low confidence factor may be a split point.

Note:

- Nodes that are higher up in the tree (closer to the root) represent more work.
- You want to fork a branch that are higher up and with a larger confidence factor for D-ALL, or with a smaller confidence factor for D-CUT.
- Use the above information to compute a global priority.
DTS: Algorithm

**Algorithm DTS:**
- Initialize a global job list with the root as the only available job.
- while the job list is not empty do
  - Idle processors look for jobs with the highest priority in the global job list.
  - A working processor maintains its own split point information at the global job list.
  - A working processor updates bounds when a job is finished and then becomes idle.

**Comments:**
- Used by several state-of-the-art chess programs.
- Spend a bit more time to decide whether a node is a split point or not.
  - Takes some time to tune for the best parameters.
- Speed-up factor is very good: 3.7 for 4 processors, 6.6 for 8 processors and 11.1 for 16 processors [Manohararjah ’01].
- Load balancing is good.
- Scalability is reasonable.
Comments: parallel $\alpha$-$\beta$ search

- DTS is currently being used by most Chess-like programs.
- It also takes time to tune the system parameters for DTS to work well.
  - The threshold for confidence factors.
  - Dynamically adjusting of the confidence factors.
Memory issues (1/2)

- During searching, each process needs to maintain the following information.
  - Local data: such as the current depth, current best move.
  - Data that can be used later: such as the hash information.

- Distributed memory model.
  - Maintain each own data in a private memory area.
  - Exchange information when needed.
    - Using message passing to probe a hash entry.
    - Using message passing to return the value of a probe.

- Shared memory model.
  - Maintain each local data in a private memory area.
  - Maintain the re-used information in a global area.
    - Current read is often allowed in the model.
    - Lock the cell when it needs to write.
Memory issues (2/2)

- **Advantage and disadvantage**
  - Distributed memory model.
    - Coding is easy.
    - Slow response time.
  - Shared memory model.
    - Overhead in locking.
    - Fast response time when there is no extensive memory contention.

- **Often used techniques: Lockless transposition tables.**
  - Allow concurrent read.
  - Do not assume writing of an entry is atomic.
Lockless transposition table

- **Scenario**
  - Assume each entry of the transposition table $H$ contains two parts where reading/writing each part is atomic.
    - $\text{Position\_signature}$: 64 bits $\rightarrow H_1$.
    - $\text{Data}$: 64 bits $\rightarrow H_2$.
  - Assume the hash key $\text{hash\_key}$ is the rightmost $h$, say $h = 32$, bits of $\text{Position\_signature}$.

- **To read or write an hash entry given a position $P$, you do the followings.**
  - Compute $\text{Position\_signature}(P)$ and $\text{Data}(P)$.
  - Let $\text{hash\_key}(P)$ be the rightmost $h$ bits of $\text{Position\_signature}(P)$.
  - Read or write $H_1(\text{hash\_key}(P))$.
  - Read or write $H_2(\text{hash\_key}(P))$.

- **Problem**: The hash entry is **corrupted** if
  - $P$ is being visited at the same time by two processes $C_1$ and $C_2$ so that
    - $C_1$ writes $H_1(\text{hash\_key}(P))$.
    - $C_2$ writes $H_2(\text{hash\_key}(P))$. 
Solution

- **Algorithm for writing an entry**
  - **Compute** \( \text{Position\_signature}(P) \) **and** \( \text{Data}(P) \).
  - **Let** \( \text{hash\_key}(P) \) **be the rightmost** \( h \) **bits of** \( \text{Position\_signature}(P) \).
  - **write:** \( H_1(\text{hash\_key}(P)) \leftarrow \text{Position\_signature}(P) \text{ XOR } \text{Data}(P) \).
  - **write:** \( H_2(\text{hash\_key}(P)) \leftarrow \text{Data}(P) \).

- **Algorithm for reading an entry**
  - **Compute** \( \text{Position\_signature}(P) \).
  - **Let** \( \text{hash\_key}(P) \) **be the rightmost** \( h \) **bits of** \( \text{Position\_signature}(P) \).
  - **read:** \( W_1 \leftarrow H_1(\text{hash\_key}(P)) \)
  - **read:** \( W_2 \leftarrow H_2(\text{hash\_key}(P)) \)
  - **reconstruct:** \( W_1 \leftarrow W_1 \text{ XOR } W_2 \)
  - **verify:** **check whether** \( W_1 = \text{Position\_signature}(P) \)
    - **if they equal, then use this entry.**
    - **if they do not equal, then the entry is corrupted.**
Why this works

- \( H_1(\text{hash}\_\text{key}(P)) = \text{Position}\_\text{signature}(P) \oplus \text{Data}(P) \).
- \( H_2(\text{hash}\_\text{key}(P)) = \text{Data}(P) \).
- \( H_1(\text{hash}\_\text{key}(P)) \oplus H_2(\text{hash}\_\text{key}(P)) = \text{Position}\_\text{signature}(P) \).
- If \( H_1(i) \) and \( H_2(i) \) are written by two different processes with \( \text{Data}(P_1) \) and \( \text{Data}(P_2) \), then it will probably not produce the right position signature.

Comments:
- May have errors because of hash collisions.
- It is not too difficult to extend this method to an hash table with more than 2 entries.
Parallel Monte-Carlo tree search

- Leaf parallelization.
- Root parallelization.
- Tree parallelization.
  - With global synchronization.
  - With local synchronization.
MCTS with UCT

- **Algorithm MCTS:**
  - 1: Obtain an initial game tree
  - 2: Repeat the following sequence $N_{total}$ times
    - 2.1: Selection
      - From the root, pick a PV path to a leaf such that each node has best UCB “score” among its siblings
      - May decide to “trust” the score of a node if it is visited more than a threshold number of times.
      - May decide to “prune” a node if its score is too bad now to save time.
    - 2.2: Expansion
      - From a best leaf, expand it by one level.
      - Use some node expansion policy to expand.
    - 2.3: Simulation
      - For the expanded leaves, perform some trials (playouts).
      - May decide to add knowledge into the trials.
    - 2.4: Back propagation
      - Update the “scores” for nodes using a good back propagation policy.

- Pick a child of the root with the best score as your move.
MCTS: example

selection

expansion

simulation

propagation

TCG: Parallel Game Tree Search, 20141225, Tsan-sheng Hsu ©
Leaf parallelization

- **Algorithm PMCTS_{leaf}**:  
  - Select the best leaf and the PV path.  
  - Perform Expansion in sequential.  
  - Perform Simulation, i.e., multiple trials, in parallel on the same leaf.  
  - Perform Back propagation in sequential.

- **Comments**:  
  - Coding is very easy.  
  - Good parallelization for performing a large number of trials.  
  - Can utilize a large number of PE’s.  
  - The best leaf may no longer be the best after only a few more trials.
Root parallelization

- **Algorithm PMCTS\textsubscript{root}:**
  - Duplicate $k$ copies of the current game tree.
  - Perform Monte-Carlo tree simulation on each copy in parallel for a few trials.
  - Combine the copies into one copy by merging statistics on nodes and put the information into the current game tree.

- **Comments:**
  - Coding is easy.
  - Can utilize as many PE\textquotesingle s as available.
  - May need to make sure that each tree does not pick the same best leaf.
  - Need to have a mechanism to properly choose the best leaves among all trees.
    - Avoid duplicated efforts.
Tree parallelization — global synchronization

- **Algorithm PMCTS$_{Tg}$:**
  - Use only one game tree.
  - Perform Selection, Expansion and Simulation in parallel.
    - Different threads may work on different nodes in parallel.
    - Need a mechanism to ensure threads are not working on the same leaf.
  - Use a global lock to make sure the game tree is writable by one thread during the Back propagation phase.

- **Comments:**
  - Speed-up is bad.
Algorithm $\text{PMCTS}_{I_l}$:
- Make every node of the game tree as a global variable.
- Perform Selection, Expansion, Simulation and Back propagation in parallel.
  - Different threads may work on different nodes in parallel.
  - Need a mechanism to ensure threads are not working on the same leaf.
- Use a lock to make sure each node is writable by one thread during Back propagation.

Comments:
- Heavy O.S. overhead.
- Unsure about the scalability.
Problems of parallel Monte-Carlo search

- Each iteration of a Monte-Carlo simulation is a Markov chain process.
  - You need to know the result of the previous trial to decide the current selection.
  - Making trials in parallel has a larger statistical error.
  - May explore the wrong branch if synchronization is done only after a lot of trials.
  - May not have too much parallelism if synchronization is done after only a few trials.

- The cost of synchronization.
  - Shared global variable.
  - Cost of lock and unlock.
  - Memory bandwidth.
  - Network bandwidth.

- The cost of programming.
Parallel Monte-Carlo search: Analysis

- **Amdahl’s law:** assume a program needs to execute $T$ instructions and and $x$ of them can be parallelized. can be parallelized.
  - Assume you have $n$ processors and an instruction take a unit of time.
  - Parallel processing time $\geq T - x + x/n + p_n \geq T - x$ where $p_n$ is the cost for overhead in doing parallelization with $n$ processors.
  - Speed-up $\leq T/(T - x)$.
    - If 20% of the code cannot be parallelized, then your parallel program can be at most 5 times faster no matter how many processors you have.

- Leaf and root parallelization both have a large portion that is not parallelizable.
- Global or local synchronization has a large overhead.
- **Comments**
  - Need a better parallel implementation.
  - Need a better way to deal with the increasing error in doing more samplings.
Concluding remarks

- Need to think about tradeoff between costs in doing parallelism and benefits of saving in searching efforts because of parallelism.
- May need to think how to maintain distributed transposition tables.
- May need to think about the machine architecture.
  - Shared-memory vs. distributed memory.
  - Fine grain or coarse grain.
  - Whether the parallel version is stable or not?
  - Ease of debugging.
  - Ease of coding.


References and further readings (2/2)