USING THE MPPA ARCHITECTURE FOR UCT PARALLELIZATION

Aline Hufschmitt
LIASD - University of Paris 8, France
alinehuf@ai.univ-paris8.fr

Jean Mehat
LIASD - University of Paris 8, France
jm@ai.univ-paris8.fr

Jean-Noël Vittaut
LIASD - University of Paris 8, France
jnv@ai.univ-paris8.fr

ABSTRACT

We present here a study of the use of a *Multi-Purpose Processor Array* (MPPA) architecture for the parallelization of the UCT algorithm applied to the field of *General Game Playing*. We evaluate the constraints imposed by this architecture and show that the only parallelization of UCT proposed in the literature that is feasible on MPPA is a *leaf parallelization*. We show that the MPPA provides good scalability when increasing the size of the communications, which is useful when using synchronous communications to send large sets of game initial positions to be processed. We consider two approaches for the calculation of the playouts: the distributed computing of a playout on each cluster and the calculation of several playouts per cluster; we show that the second approach gives better results. Finally, we describe experiments concerning the thread management and present a surprising result: it is more efficient to revive threads than keep them alive and try to communicate with them.

KEYWORDS

Multi-Purpose Processor Array, Parallelization, UCT, MCTS, General Game Playing

1. INTRODUCTION

General Game Playing (GGP) is a branch of Artificial Intelligence with the aim of achieving versatile programs capable of playing any game without human intervention. These programs must be able to analyze the rules of an unknown game, to understand the goals, to discover the sequences of moves leading to victory and to play with expertise.

To find out what move to select at each stage of the game, different techniques of tree search have been developed to explore every position of the game and the different branches of possibilities which depend on the selected moves. The root of the tree represents the initial state of the game and the goal is to reach one of the leaves corresponding to the end of the game the score of which is the highest possible for the player.

Currently the algorithm used by most players is the *Upper Confidence bound applied to Trees* (UCT), a member of the *Monte Carlo Tree Search* (MCTS) algorithms family [Browne et al., 2012]. The number of game simulations (playouts) done by UCT determines the quality of the evaluation of a game tree node [Kocsis and Szepesvári, 2006]. Therefore, various approaches of UCT parallelization have been proposed to increase the number of playouts made in a given time.

Former GGP players used Prolog to interpret the language used to describe the rules, i.e. to determine the current state of the game, the list of legal moves or the scores in terminal states. These calculations are slow [Björnsson and Schiffel, 2013] but recently, the use of *Propositional Networks* (propnets) has allowed an important improvement in playout computation speed.

Different parallelization techniques of MCTS algorithms have been explored so far on machines with multi-core and/or multi-threads CPU used alone or networked to provide more computing power. Some of these techniques are applied to *General Game Playing* [Méhat and Cazenave, 2011a; Finnsson, 2012] but none, to our knowledge, realizes this parallelism on a player using a *propnet*. The important acceleration in playout computation brings new conditions for parallelization of UCT because communication and synchronization times become significant.

In this article, we study the parallelization of UCT using a *propnet* to interpret game rules on a *Multi-Purpose Processor Array* (MPPA), a new architecture marketed since 2013, created by the Kalray Company (Essonne, France) and dedicated to many-core processing. This very recent architecture has barely been tested on practical applications, thus we have chosen to explore the possibilities it can offer and evaluate its limitations.

Section 2 describes the different parallelization techniques proposed in the literature. Section 3 presents the MPPA architecture. Section 4 presents the feasible parallelization on MPPA. Section 5 evaluates the scalability enabled by this architecture for communications of varying size. Section 6 establishes the best approach to compute the playouts and section 7 compares different ways of managing threads inside clusters.

2. PARALLELIZATION APPROACHES OF UCT

Tree parallelization [Gelly et al., 2006] involves several processes in the construction of the game tree in a single shared memory. The use of a global mutex for the whole tree causes a bottleneck, so various improvements consisting in the use of local mutexes [Chaslot et al., 2008; Chaslot, 2010] or a lock free algorithm [Enzenberger and Müller, 2010] have been proposed.

On a distributed memory system, *root parallelization* [Cazenave and Jouandeau, 2007] consists in developing several distinct UCT trees from a position of the game. The different evaluations are collected after some time and combined to choose the best move according to different strategies (Best, Sum, Sum10, Rave, majority vote) [Méhat and Cazenave, 2011b; Soejima et al., 2010]. An improvement consists in synchronizing the evaluations of the top nodes at regular intervals during search [Gelly et al., 2008].

With *leaf parallelization* [Cazenave and Jouandeau, 2007], a single tree is constructed by a master process sending the positions to be explored to slave processes calculating the playouts. To minimize communication costs, Finnsson [2012] has proposed to realize multiple playouts per position, but some unworthy positions are then over-exploited. Chaslot et al. [2008] have proposed to stop a group of playouts that does not look promising to search again from another position but this approach does not provide good scalability. Cazenave and Jouandeau [2008] have suggested the use of asynchronous communications, which provides better scalability.

The *UCT-Treesplit* algorithm [Graf et al., 2011; Schaefers et al., 2011] combines the advantages of parallelization on distributed memory with the construction of a single tree. The machine on which the node is stored is selected using a hash key associated with the node. One drawback is the importance of communications necessary to perform a descent in the tree, i.e. select the start of a playout, or to update the evaluation of nodes. Yoshizoe et al. [2011] have proposed to solve this problem by performing a depth-first search.

3. MPPA ARCHITECTURE

The MPPA-256 chip is a multi-core processor composed of 256 processing cores (PC) organized in a grid of 16 clusters connected through a high-speed *Network-on-Chip* (NoC). It is the first member of the MPPA MANYCORE family, the others reaching up to 1024 processors in a single silicon chip.

The MPPA-256 is a MIMD architecture with a distributed memory accessible only locally, i.e. each cluster is encapsulated, only accessing its own memory, and can execute code independently of other clusters.

Each cluster contains 16 PC, a shared memory of 2MB and a system core using a specific operating system (NodeOS). This system core supervises the scheduling, execution of tasks and data transfers while the 16 PC are dedicated to application code.

Four I/O interfaces allow communications between the host machine and clusters for two of them and between clusters and the Ethernet network for the two others, but with the current version of the middleware we

only have access to one of the I/O interfaces: the software tools designed for the MPPA and grouped under the name of MPPA ACCESSCORE are currently under development. This I/O interface has a quad core SMP processor with a 4GB DDR3 memory and a PCI Gen3 interface for communications with the host. Different connectors are available to implement synchronous or asynchronous communications using a simple buffer or queues.

A technical report Jouandeau [2013] states that "the computing capabilities of Intel i7 3820 processor with 8 cores and a MPPA processor with 256 cores are close". The estimate is based on the performance observed for a single core on solving the spin glass problem and multiplied by the number of cores, which implies parallel algorithms with zero communication times.

The limitation a cluster's memory to 2MB is a major constraint in the development of applications for the MPPA. In our case, we see in section 4 that it reduces the choices of possible parallelization approaches. Even if the MPPA-256 card seems to be limited in its performance, we must concede that it is only the first member of the MPPA MANYCORE family. A *Coolidge* processor with 1024 cores is expected in 2015. In addition, several MPPA-256 cards can be used together in the same host machine to increase computing capacity. Therefore, the MPPA architecture presents possibilities for the evolution of computing power that deserve investigation.

4. PARALLELIZATION OF UCT ON MPPA

Our experiments were carried out on a server equipped with an Intel Core I7 at 3.6GHz running Linux OS and a PCIe Application Board AB01 equipped with a chip MPPA-256 [Kalray, 2013]. Our program is based on Jean Noël Vittaut's *LeJoueur*, written in C++.

The limited size of memory inside the clusters does not allow the construction of a UCT tree. Therefore neither a *tree parallelization* inside each individual cluster nor a *root parallelization* nor a *UCT-Treesplit* can be performed on a MPPA. The only remaining choice is a *leaf parallelization*. Unfortunately, this technique is less effective than *root* or *tree parallelizations* because it suffers from limitations caused by the master process and communications [Soejima et al., 2010]. In addition to this, memory limitations imply a coding as concise as possible so that the *propnet* can fit into the cluster memory. For the most complex games, like Hex, the size of the *propnet* or the size of the positions to be processed exceeds the available space or connectors capacity.

For all the experiments presented in this article we used synchronous communications. This allows us to have benchmark results that we can later compare to the ones using asynchronous communications. Sets of game positions are sent to the MPPA from which playouts are computed. The I/O interface waits for the results from all the clusters before sending them back to the host. Each transmission of a position set, calculation of the playouts and recovery of the results (the scores) is referred to as a *run*.

5. SCALING FOR VARIABLE SIZE OF COMMUNICATIONS

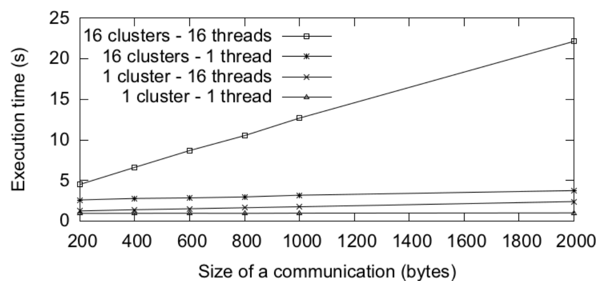
In GGP, the size of position descriptions varies significantly from one game description to another in a ratio from one to ten: we need communications of about 200 to 2000 bytes to send a position to the MPPA. In the case of synchronous communications, if we want to make one playout per thread (i.e. per PC), the size of a communication is the size of a position multiplied by the number of threads.

To evaluate the scalability of the MPPA we varied the size of messages sent between the host and the I/O node and between the I/O node and the clusters from 200 to 2000 bytes. To prevent the variable time required to calculate the playouts from disturbing measurements, no playout was actually calculated and an empty result was returned immediately. Each execution of the program performs 1000 runs. We measured performance as the time necessary to execute these 1000 runs for 1 to 16 clusters with 1 to 16 threads per cluster.

Figure 1 presents the experimental results. With one thread on one cluster, we can see that the running time is almost constant while the size of the communications increases tenfold. With 16 threads on one cluster, we can see an additional time corresponding to the time required to start the threads and the time necessary to distribute the data among threads. The curve corresponding to 16 clusters with one thread each shows

the time required for the I/O node to distribute data: the connector is set to the destination cluster before sending each part of the data.

Figure 1. Evolution of the execution time depending on the size of communications (1000 runs)



Scaling is then constrained by two aspects: first, when the I/O node receives a set of positions it needs time to distribute them among the clusters and second, the thread start time inside a cluster is not negligible. Scaling is only slightly hindered when considering these two aspects separately but when we combine them (for 16 clusters and 16 threads per cluster), we find that it significantly degrades it. However, we see that scaling remains satisfactory since the execution time increases almost fivefold when the message size increases tenfold.

6. PLAYOUT CALCULATION APPROACHES

In the restrictive conditions of a *leaf parallelization*, the choice remains of the playout calculation approach. The *propnet* is represented by a logic circuit with different layers of logic gates. Each layer is represented by a set of rules, which can be evaluated in any order. We present here two approaches for the calculation of playouts with a *propnet*. In the first one, we distributed the calculation of each rule layer among the different threads of a cluster and then processed a single playout per cluster. In the second we proceeded to the calculation of a complete playout on each thread of a cluster, obtaining N playouts per cluster.

We conducted these experiments on three games: *Tictactoe* which has short playouts (between 5 and 9 moves) and a small quick to evaluate *propnet*, *EightPuzzle* which has a small *propnet* too but playouts up to 60 moves and *Breakthrough* which has a *propnet* which is longer to evaluate. Each execution of the program performs 10000 runs. We measured the performance based on the number of playouts carried out per second.

Figure 2 presents the performance obtained on the game of *Tictactoe* for 1, 2, 4, 8, 16 clusters with 1, 2, 4, 8, 16 threads per cluster and demonstrates the significant cost in synchronization generated by the division of playout computation. Each layer of the *propnet* circuit depends on its predecessor, so the threads must synchronize. The calculation of each part of a playout is considerably speeded up by the use of a *propnet*. The distributed computing of a playout cannot therefore provide any benefit considering the overhead introduced by the synchronization barrier between layers.

Figure 3 presents the results for the three games with 1 to 16 clusters and 1 to 16 threads per cluster. It shows that the performance scales well when a complete playout is computed on each PC.

Results for the game of *Tictactoe* show a progression from ≈450 playouts/s for one cluster with one thread to ≈77700 playouts/s for 16 clusters with 16 threads, i.e. a speed increase of 170 for a 256 PC machine.

For the game *EightPuzzle*, the speed increase is, at best, 133 (for 14 threads per cluster) with 224 PC. The performance does not scale well after 14 threads. We explain this result by the constant length of playouts: playing randomly, the solution has little chance of being found and playouts are stopped after 60 moves by the stepper. Therefore, all the scores are sent to the I/O node at the same time by all clusters. The size of the sent messages increases with the number of threads and the communications get slower. A similar deceleration hindering scalability can be observed in *Tictactoe* by forcing the players to completely fill the grid for each game: playout length is then set to 9 moves.

Scaling is better for *Breakthrough*. Computation time is longer, therefore communication and synchronization times are lower in comparison. We can observe a speed increase of 155 for a 256 PC machine.

Figure 2. Evolution of the performance for the game of *Tictactoe* i.e. number of playouts per second (for 10000 runs) with calculation of a playout distributed on N threads

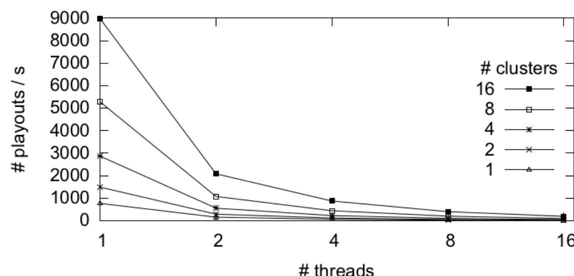
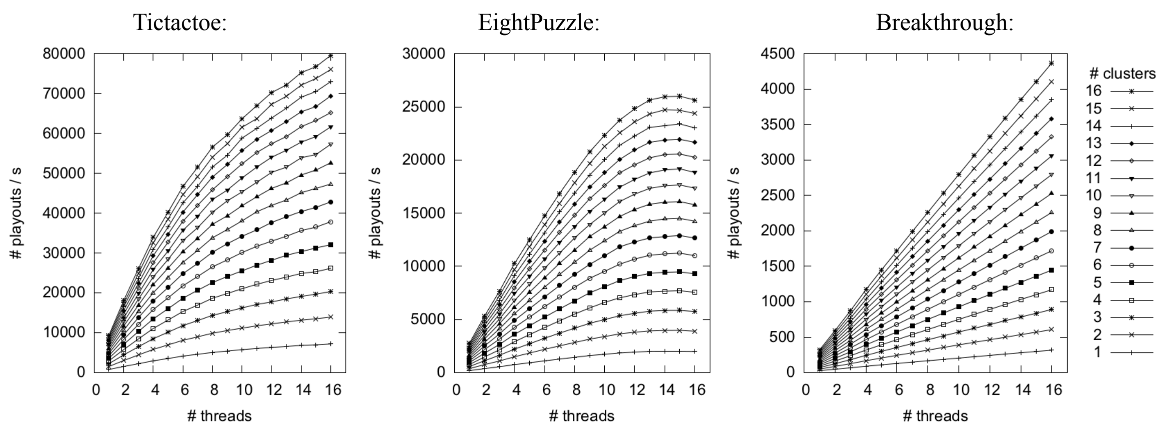


Figure 3. Evolution of the performance i.e. number of playouts per second (for 10000 runs) with the calculation of one playout per thread



7. THREAD MANAGEMENT

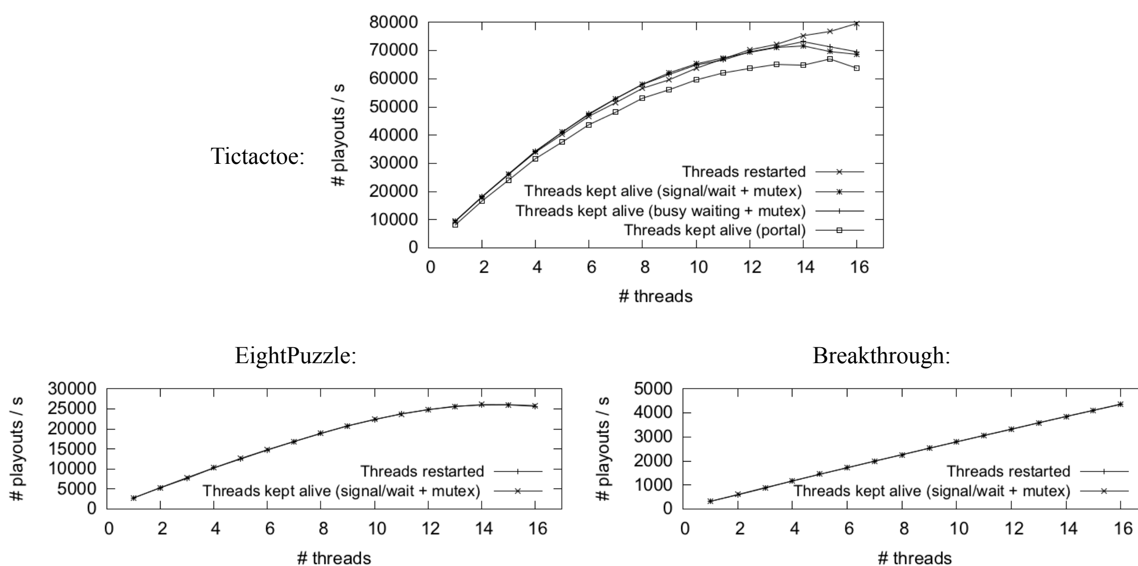
In the previous experiment threads were restarted for each set of playouts. Another strategy is to keep the threads alive waiting for playout requests. To test different approaches for communications between a cluster and its threads, we take the second experiment of the previous section and change the management of threads. Each cluster receives a set of positions it distributes to the different threads. The tests are performed on the different games by varying the number of clusters and threads per cluster; 10000 runs are executed every time.

In the first part of the experiment we use the Posix functions `pthread_cond_wait` and `pthread_cond_signal` to communicate with the threads. In the second part, we replace the Posix functions by a busy wait. In the third part we use portal connectors which are part of the MPPA ACCESCORE tools and uses the NoC to create communication paths between the main thread of the cluster and the other threads. This third approach avoids the use of mutexes.

Results are displayed in figure 4 for 16 clusters and 1 to 16 threads per cluster and compared with results obtained with the second experiment of the previous section where threads were restarted. For the games *EightPuzzle* and *Breakthrough*, we compare only the restart of the threads with the use of the Posix functions `pthread_cond_wait` and `pthread_cond_signal`.

With the game of *Tictactoe*, the use of the portal is less effective regardless of the number of threads used, and the different approaches keeping threads alive does not scale well above 14 threads per cluster. We note with surprise that the technique consisting in restarting threads for each request of calculation provides a better scalability and gives the best results. The results obtained on the games *EightPuzzle* and *Breakthrough* do not show such a marked difference since the curves are overlaid, but confirm that keeping threads alive provides no benefit.

Figure 4. Evolution of the performance i.e. number of playouts per second (10000 runs, 16 clusters) with different thread management modes



This can be explained by the use of a shared memory inside each cluster and the fact that all the synchronization primitives end on a spinlock or equivalent. When a thread has finished its work, it waits for a signal by polling in the shared memory. Therefore, if the other threads have not finished the calculation of their playout, they are slowed down since they also need to access the shared memory. The main process running on PC0 and responsible for the communications with the I/O node is also slowed down. The more threads finish their task, the greater the execution of another process is slowed. On the contrary, when a thread uses `pthread_exit`, the PC running this thread is placed in an idle state, so it does not disturb the work of other PCs. Halting and relaunching the threads is then more efficient.

CONCLUSION

In this paper we have studied the capabilities offered by a *Multi Purpose Processor Array* (MPPA) architecture for the parallelization of the UCT algorithm in the field of *General Game Playing*. The limitation of the memory inside the cluster to a size of 2MB is the major constraint. Among various parallelization techniques described in the literature the only applicable one, with this limited memory space, is *leaf parallelization*.

We have demonstrated that the MPPA provides good scalability when increasing the size of communications, giving good results when using synchronous communications and sending large sets of game initial positions to be processed in a single run.

We have considered two approaches for the calculation of playouts. The distributed computing of a playout on the different PC of a cluster causes an important synchronization overhead. The calculation of a complete playout per PC gives better results.

We were able to establish that on MPPA it is more efficient to restart threads for each calculation request. All synchronization primitives on the MPPA end on a spinlock. Therefore, threads kept alive slow down the working ones because of memory access competition.

It would be possible to reduce the weight of communications by making several playouts from each sent position, but carrying out several playouts from the same position may be less beneficial for the UCT exploration than starting from different positions [Chaslot et al., 2008].

One may think that the use of asynchronous communications would improve these results but unfortunately the first experiments we have conducted have yielded disastrous results. This comes from middleware problems that our experiments have revealed. These problems should be fixed by Kalray in the next release. The use of asynchronous communications will therefore be the subject of future experiments.

We should also compare our results with what could be achieved with a GPU using a framework like Cuda on the same problem. The SIMT architecture requires the use of synchronous operations but this can be an advantage to distribute the calculation of each layer of the *propnet* without the need for a specific synchronization barrier. Moreover, the large quantity of memory shared by computing units allows the implementation of the different parallelization techniques presented.

Future works also include the test of new approaches for UCT parallelization or modifications of UCT. For example, the creation of mini-UCT-trees in cluster memory can save communication costs. The SHOT alternative to UCT [Cazenave, 2015] offers interesting perspectives as it scales well in addition to using less memory and it can be efficiently parallelized.

REFERENCES

Björnsson,Y.and Schiffel, S. 2013. Comparison of gdl reasoners.In *Proceedings of the IJCAI-13 Workshop on General Game Playing (GIGA'13)*, pp. 55–62.

Browne, C. B., et al. 2012. A survey of monte carlo tree search methods. *Computational Intelligence and AI in Games, IEEE Transactions on*, 4(1):1–43.

Cazenave, T. 2015. Sequential Halving Applied to Trees. IEEE Trans. Comput. Intellig. and AI in Games 7(1): 102-105

Cazenave, T. and Jouandeau, N. 2007. On the parallelization of UCT. In *Proceedings of the Computer Games Workshop*, pp. 93–101.

Cazenave, T. and Jouandeau, N. 2008. A parallel monte-carlo tree search algorithm. In van den Herik, H. J., Xu, X., Ma, Z., and Winands, M. H. M., editors, *Computers and Games*, volume 5131 of *Lecture Notes in Computer Science*, pp. 72–80. Springer.

Chaslot, G. 2010. *Monte-Carlo Tree Search*. PhD thesis, Universiteit Maastricht.

Chaslot, G., Winands, M. H. M., and van den Herik, H. J. 2008. Parallel monte-carlo tree search. In *Computers and Games*, pp. 60–71.

Enzenberger, M. and Müller, M. 2010. A lock-free multithreaded monte-carlo tree search algorithm. In *Proceedings of the 12th International Conference on Advances in Computer Games*, ACG'09, pp. 14–20, Berlin, Heidelberg. Springer-Verlag.

Finnsson, H. 2012. *Simulation-Based General Game Playing*. Doctor of philosophy, School of Computer Science, Reykjav´ık University.

Gelly, S., et al. 2008. The parallelization of monte-carlo planning - parallelization of mc-planning. In Filipe, J., Andrade-Cetto, J., and Ferrier, J.-L., editors, *ICINCO-ICSO*, pp. 244–249. INSTICC Press.

Gelly, S., et al. 2006. Modification of uct with patterns in monte-carlo go. Technical Report 6062, Inria.

Graf, T., et al. 2011 .Parallel monte-carlo tree search for hpc systems. In *Proceedings of the 17th International Conference on Parallel Processing - Volume Part II*, Euro-Par'11, pp. 365–376, Berlin, Heidelberg. Springer-Verlag.

Jouandeau, N. 2013. Intel versus mppa. Technical report, LIASD Universit Paris8.

Kalray 2013. *MPPA ACCESSCORE 1.0.1 - POSIX Programming Reference Manual - KETD-325 W08*. Kalray SA.

Kocsis, L. and Szepesvári, C. 2006. Bandit based monte-carlo planning. In *Proceedings of the 17th European Conference on Machine Learning*, ECML'06, pp. 282–293, Berlin, Heidelberg. Springer-Verlag.

Méhat, J. and Cazenave, T. 2011a. A parallel general game player. *KI*, 25(1):43– 47.

Méhat, J. and Cazenave, T. 2011b. Tree parallelization of ary on a cluster. In *GIGA 2011, IJCAI 2011*, Barcelona.

Schaefers, L., Platzner, M., and Lorenz, U. 2011. Uct-treesplit - parallel mcts on distributed memory. In *MCTS Workshop*, Freiburg, Germany.

Soejima, Y., Kishimoto, A., and Watanabe, O. 2010. Evaluating root parallelization in go. *IEEE Trans. Comput. Intellig. and AI in Games*, 2(4):278–287.

Yoshizoe, K., et al. 2011. Scalable distributed monte-carlo tree search. In *SOCS*.