Parallel Monte-Carlo Tree Search with Simulation Servers

HIDEKI KATO[†1,†2] and IKUO TAKEUCHI[†1]

Recently Monte-Carlo tree search is boosting the performance of computer Go playing programs. A novel parallel Monte-Carlo tree search algorithm is proposed. A tree searcher runs on a client computer and multiple Monte-Carlo simulation servers run on other computers on a network. The client broadcasts a position to be simulated to every server, which then simulates a game from the position to the end and sends the result (win or loss) back to the client. The statistical information in the search tree is updated by the client according to the result. This architecture allows servers on-the-fly connection or disconnection. Experimental results using four quad-core Linux computers on a private Gigabit Ethernet LAN show its performance scales well.

1. Introduction

Monte-Carlo tree search (MCTS) is a powerful tree search algorithm that can be applied to trial-based planning tasks including the game of Go [⋆1].

It is empirically proved that the performance of MCTS scales well against the number of simulations to select an optimal move in computer Go. In addition, developing efficient parallel MCTS (PMCTS) algorithms is important to improve the performance because single processor's performance may not be expected to increase as used to.

This paper is organized as follows. Section 2 introduces previous related works, Section 3 explains PMCTS algorithms, Section 4 proposes our algorithm, Section 5 describes our experiments, Section 6 discusses the results of the experiments, and Section 7 provides conclusion and describes future research.

Note: Our Go playing program, Fudo Go, which features 3 x 3 local patterns in MC simulations and RAVE with UCT, was used for the experiments and was 12th out of 18 and 7th (shared) out of 13 participants for 9 x 9 and 19 x 19 Go, respectively, at 13th International Computer Games Championship Beijing. The source code is available at http://www.gggo.jp/.

†1 The Department of Creative Informatics, The Graduate School of Information Science and Technology, The University of Tokyo

†2 Fixstars Corporation

⋆1 http://senseis.xmp.net/, for more info.

2. Related Work

S. Gelly et al.[6] first introduced shared-tree symmetrical multi-thread PMCTS for shared memory computers. T. Cazenave et al.[1] proposed and evaluated three master-slave style PMCTS algorithms on an MPI cluster of 16 Intel Pentium-4 computers connected via a 100-Megabit Ethernet LAN. G. Chaslot et al.[2] evaluated shared-tree and two, leaf- and root-parallelization, of the three methods proposed by T. Cazenave et al.[1] on 16-core IBM Power5 shared-memory computers. S. Gelly et al.[5] describes their shared-tree PMCTS[6] in detail and proposes a PMCTS algorithm for MPI clusters. Each shared-memory multiprocessor node, on which shared-tree PMCTS runs, of an MPI cluster periodically exchanges and merges statistical information in the tree.

Another related work, D. Dailey et al.[3],[4] showed that double the number of simulations for a move increases ELO rating [⋆2] by 50 to 100 (100 ELO roughly corresponds to one stone handicap). This clearly shows that developing efficient PMCTS algorithms is important to improve the performance of any MCTS Go programs as well as good algorithms or heuristics.

3. Parallel Monte-Carlo Tree Search

3.1 Monte-Carlo Tree Search

We give a brief description of MCTS algorithm first (Fig. 1).

⋆2 http://en.wikipedia.org/wiki/Elo_rating, for example.

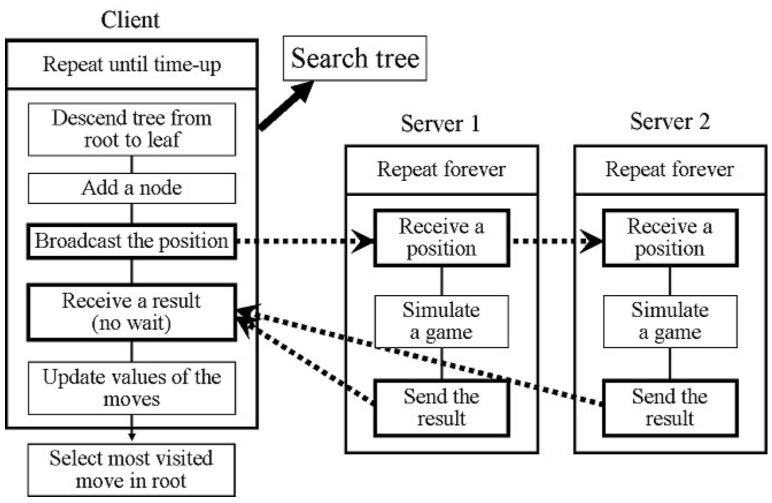

Fig. 3 Proposed PMCTS algorithm. The difference from Fig. 1 is that the simulation part is separated.

A typical MCTS consists of four steps: *descend tree*, *extend tree*, *evaluate a position by MC simulations* and *update values of the moves*.

3.2 PMCTS algorithms

Currently two parallelisms of MCTS have been developed.

One is a fine-grain parallelism, which runs multiple MC simulations simultaneously. A straightforward extension of a sequential implementation of MCTS algorithm, shared-tree symmetrical multi-thread PMCTS (**Fig. 2**) has been used for almost all MCTS programs. This is first introduced by S. Gelly et al.[6] where multiple threaded sequential MCTS programs share the search tree and execute MC simulations in parallel.

Another approach, client-server model is first introduced by us[7], which is based on a master-slave style PMCTS and can be thought as an improvement of at-the-leaves PMCTS proposed by T. Cazenave et al.[1] This is described in Section 4.

The other is a coarse-grain parallelism. MoGo[5] features this for a MPI cluster where each node periodically exchanges and merges the statistical information in the tree, together with the shared-tree PMCTS for each node. This can be thought as a kind of root-parallelization in 2) or single-run PMCTS in 1). We discuss this parallelism no more in this paper, though this can also be applied to our algorithm.

4. Proposed Algorithm

Some notes on terminology: we call our

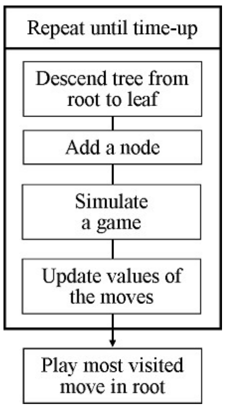
Repeat until time-up
Descend tree from root to leaf
Add a node
Simulate a game
Update values of the moves
Play most visited move in root

Fig. 1 A typical sequential Monte-Carlo tree search.

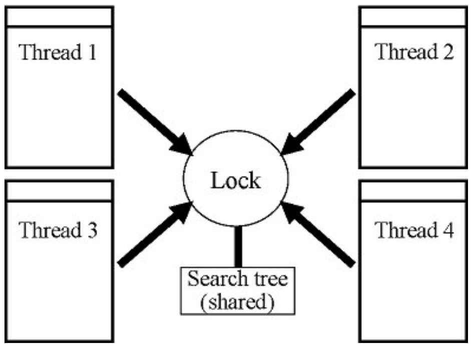

Fig. 2 A typical shared-tree symmetrical multi-thread parallel Monte-Carlo tree search. 4 threads. Each thread corresponds to the outlined box in Fig. 1.

```
GENERATEMOVE($pos$)
 1  $root \leftarrow$ ADDNODE($pos, node$)
 2  repeat
 3      $node \leftarrow$ DESCENDTREE($root$)
 4      $pos \leftarrow node.position$
 5      $move \leftarrow$ SELECTMOVE($pos$)
 6      if $move.visits \geq MIN\_VISITS$
 7        then
 8          $pos \leftarrow$ PLAY($move, pos$)
 9          $node \leftarrow$ ADDNODE($pos, node$)
10          $move \leftarrow$ SELECTMOVE($pos$)
11      $packet.move \leftarrow move$
12      $packet.pos \leftarrow pos$
13      $packet.node \leftarrow node$
14      BROADCASTPOSITION($packet$)
15      $packet \leftarrow$ RECEIVERESULT($NO\_WAIT$)
16      if $packet.received$
17        then
18          UPDATE($packet.score, packet.node$)
19    until TIMELEFT() $\leq 0$
20  return MOSTVISITEDMOVE($root$)
```

Fig. 4 Pseudo code for the client.

```
MCSERVER()
1  repeat
2      $packet \leftarrow$ RECIEVEPOSITION($WAIT$)
3      $move \leftarrow packet.move$
4      $pos \leftarrow packet.position$
5      $packet.score \leftarrow$ DOSIMULATION($move, pos$)
6      SENDBACK($packet$)
7    until $FOREVER$
```

Fig. 5 Pseudo code for the server.

algorithm "client-server" style PMCTS and use "client" and "simulation servers" for the "tree searchers" and the "MC simulators", respectively, to emphasize our algorithm is designed for loosely coupled PC clusters over moderate speed networks, possibly including the Internet. Also, as current system consists of one client and multiple servers, some readers may think it is strange but please note our system allows multiple clients which may co-operate for a game or work independently for multiple games.

Our PMCTS algorithm is sketched in **Fig. 3**. **Figure 4** and **Fig. 5** are the pseudo codes of our implementation.

The differences from MCTS (Fig. 1) are just broadcasting the positions to be simulated and receiving the results without wait, i.e., if at least one result arrives the statistical information in the tree is updated and new iteration starts. At-the-leaves PMCTS is reported being worse by T. Cazenave et al.[1]. Later, G. Chaslot et al.[2] pointed out the major reason is to wait until all threads finish simulations. So, this is not our case.

Following subsections discuss some design issues.

4.1 Communication Time over Networks

The major problem implementing fine-grain PMCTS on computer clusters is longer communication time over network. One solution is to use high-speed network interface devices such as InfiniBand, though they are very expensive now. **Table 2** shows *round trip time* (RTT) over Gigabit Ethernet (see Section 5.2 for detail). Note that 150 μs is longer than a typical simulation time on 9 x 9 board. RTT is virtually increases simulation time and results in decreasing simulation speed. One of our objects is to evaluate these effects.

4.2 Peer-to-Peer vs. Broadcasting

The theoretical bandwidth of Gigabit Ethernet is about 100 MB/s. Assuming the number of servers is 100 and the size of a packet is 2 kB, which is a typical packet size for a position on 19 x 19, up to 500 packets can be sent a second using peer-to-peer communications. So, we have to broadcast a position to be simulated otherwise upto only 500 games can be simulated a second.

UDP/IP is said less reliable than TCP/IP[9]. Losing some packets, however, is not a big problem for our implementation but decreases performance only a little. This allows servers on-the-fly connection or disconnection and makes the system fault tolerant. Using UDP/IP also gives some advantages over TCP/IP, shorter communication time, for example.

Hence, we choose broadcasting for sending positions to be simulated and the results are sent back with peer-to-peer communications using UDP/IP.

4.3 Use of Other Cores

The client is not parallelized but some modern processors feature multiple cores. Using other cores for simulation servers should improve total performance. The

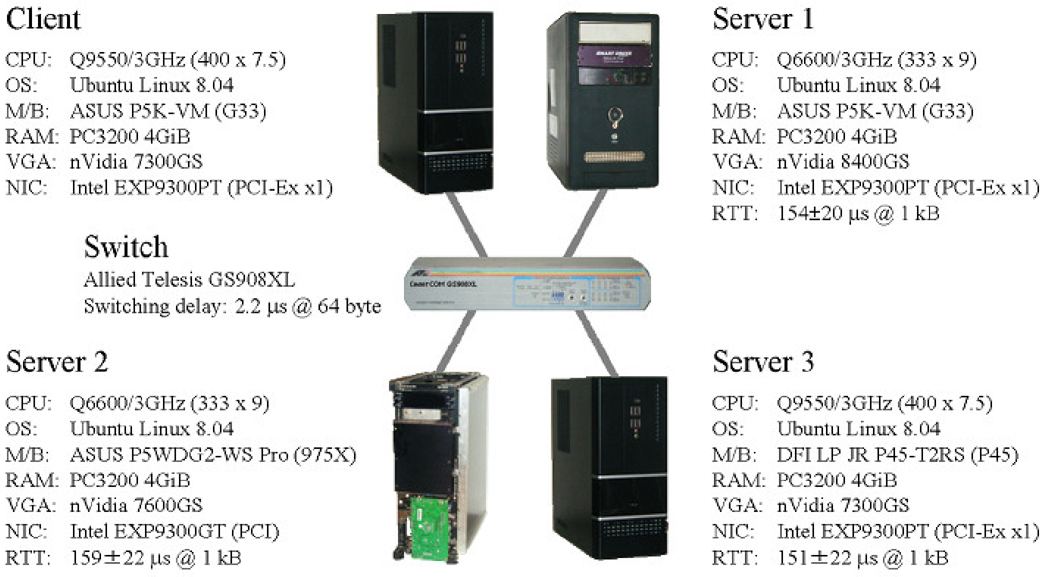

Fig. 6 Experimental system.

choice is *internal* or *external*.

Internal servers are implemented using POSIX threads and communicate with the client via FIFO queues on shared memory. *External* servers are independent Linux processes and communicate through a local loop-back network.

Table 1 shows the performances with the use of the cores. Although the differences of the winning rates are so small that they can be statistical errors, using all other cores for internal servers is likely the best choice.

5. Experiments

5.1 Experimental System

The test bed (**Fig. 6**) used for our experiments consists of four handmade Ubuntu Linux PCs connected via a Gigabit switching hub, Allied Telesis CenterCOM GS908XL.

Table 1 Winning rate with changing the use of cores in a quad-core processor. One core is dedicated for the client. Other cores are used for internal and external servers. "Int" is the number of *internal* servers. "Ext" is the number of *external* servers. "WR" is the winning rate against GNU GO 3.7.11 level 0. The numbers after winning rates are standard deviations.

Int	Ext	WR
1	2	44.4±2.2% (-39±16 ELO)
2	1	46.2±2.2% (-27±16 ELO)
3	0	47.4±2.2% (-18±16 ELO)

Every PC has an Intel Core 2 Quad processor, Q9550 or Q6600, runs at 3 GHz and an Intel Gigabit Ethernet (GbE) network interface card (NIC), EXPI9300PT or GT. The MTU of every NIC is set to 9000 or *jumbo packet* is enabled.

5.2 Round Trip Time

Round trip time is a common measure of the time to communicate via a network. Table 2 shows RTTs from the client to each server.

5.3 Results

Figure 7 shows the performance of our implementation on two board sizes, 9 x 9 and 13 x 13. All results are the winning rate against

Table 2 Round trip time from client to each server on a private Gigabit Ethernet LAN. Average of a thousand 1 kB packets. The numbers after times are standard deviations. Ping command is used.

Server	Round trip time
PC1	154±21μs
PC2	159±22μs
PC3	151±22μs

Table 3 Combinations of the servers used in Fig. 7. "Int" and "Ext" are the numbers of internal and external servers, respectively.

Label	Int	Ext
1 server	1	0
1 pc	3	0
4 pc	3	12

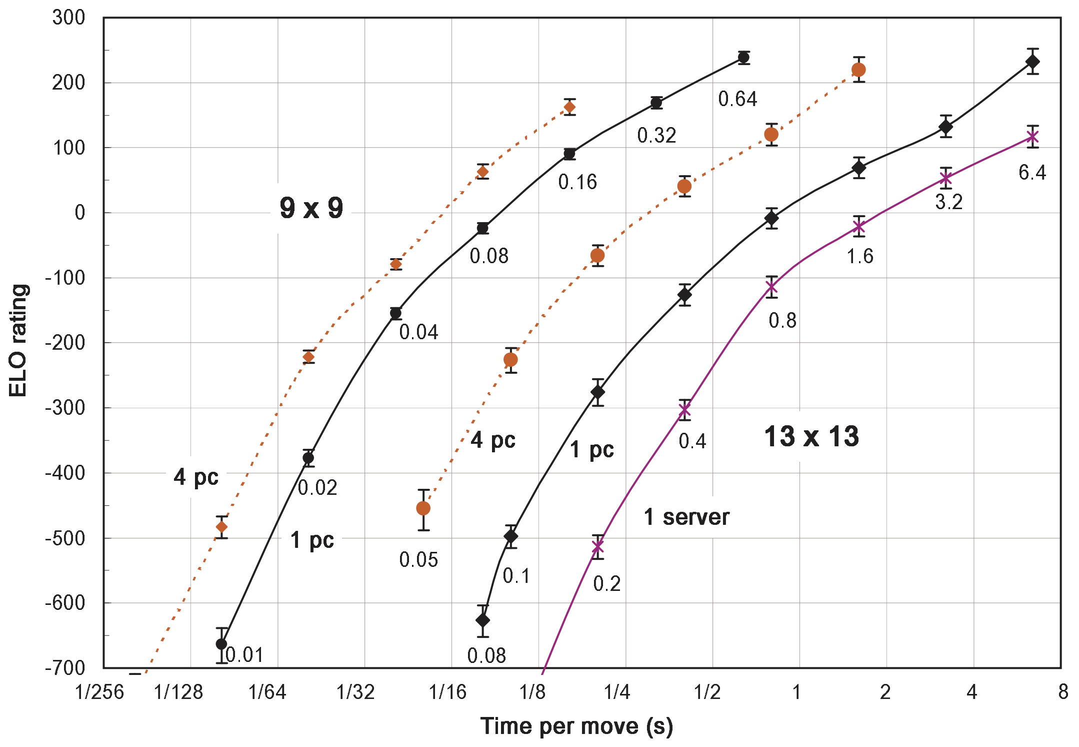

Fig. 7 ELO rating against GNU GO 3.7.11 level 0 with time.

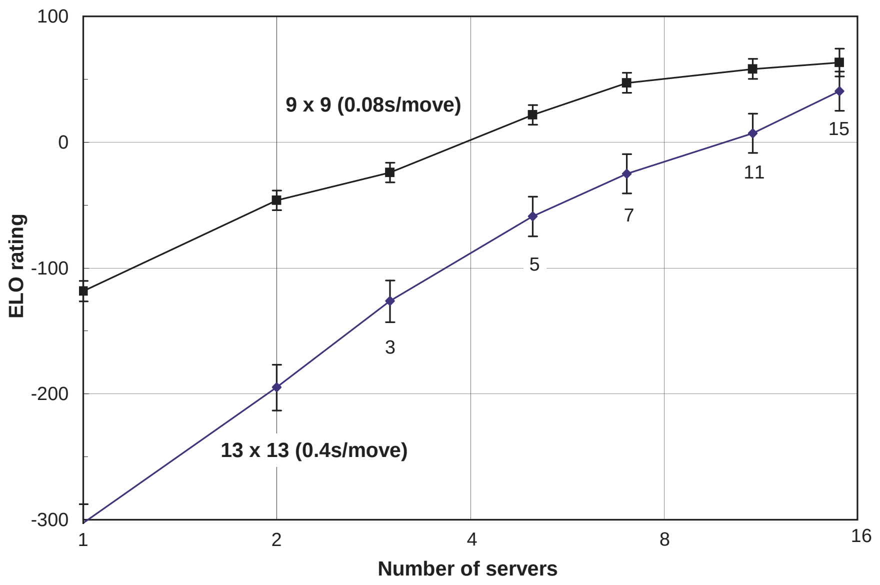

Fig. 8 ELO rating against GNU GO 3.7.11 level 0 with number of servers.

GNU GO 3.7.11 level 0. **Table 3** shows the combinations of the servers in Fig. 7. Each winning rate in Fig. 7 is an average of 2000 and 500 games for 9 x 9 and 13 x 13, respectively.

6. Discussion

Although all results of our experiments are shown in Fig. 7, other figures from different viewpoints would be useful to analyse them.

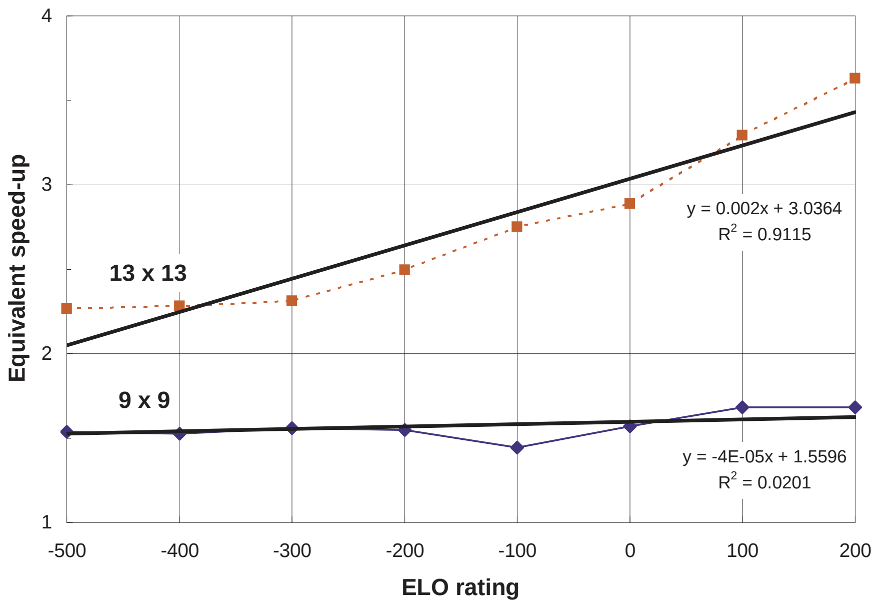

Fig. 9 Speed-up ratio by 4 pc with the winning rate against GNU GO 3.7.11 level 0.

6.1 Scalability

Figure 8 shows the scalability with the number of servers on both 9 x 9 and 13 x 13 boards.

Scalability on 9 x 9 board seems saturating beyond 7 servers and is well on 13 x 13 at least upto 15 servers. It can be expected that our algorithm scales as well on larger boards.

6.2 Speed-up

With no doubt, the major object of parallelizing is the acceleration of execution. The effect of our parallelization way could be measured by the improvement of the winning rate but the speed-up factor would be better.

We use an *equivalent speed-up*[⋆1] instead of the number of simulations done a second, which is commonly used on shared-memory systems, because the benefit from each simulation varies time to time on a network environment.

Let the winning rate of 4 computers is p with the given time for a move is t_4 and one computer has the same winning rate p if the time for a move is t_1, the equivalent speed-up for 4 vs. 1 computer is given by t_1/t_4. The speed-up factors in **Fig. 9** are taken from Fig. 7 with manual interpolation.

For the speed-up, we use the number of computers instead of the servers because our algorithm uses one core for the client and hence it is unfair to use the number of servers to compare overall performance with other benchmarks.

Figure 9 shows the equivalent speed-up for 4 computers. For 9 x 9, the speed-up factor is almost 1.55, quite fewer than 4, because the time used for the parallel part, MC simulation, is close to the time for sequential part (cf. Amdahl's law[⋆2]). In other words, there is not enough parallelism for 9 x 9 in our implementation.

In contrast, the speed-up factor increases according to the winning rate, i.e. the time for a move, on 13 x 13. This is somewhat strange and needs more experiments.

7. Conclusion and Future Work

A novel PMCTS architecture using multiple MC simulation servers was described.

A tree searcher runs on a client computer and multiple MC simulators run on other server computers on a network. The client broadcasts a position to the servers, which

⋆1 Our "equivalent speed-up" is the same as "strength-speed-up" in 2).

⋆2 http://en.wikipedia.org/wiki/Amdahls_law, for detail then send the results of the simulated game from the position back to the client. The statistical information in the tree is updated by the client according to the results. This architecture allows servers on-the-fly connection or disconnection.

We already have implemented the server on Sony Playstation3 and will report the performance and scalability on a heterogeneous PC cluster.

Acknowledgments We would like to express our thanks to Dr. Sylvain Gelly, for his suggestion to separate the tree searcher and the simulator. We also thank to the reviewers for their comments that helped a lot to improve the first version of this paper.

References

1) Cazenave, T. and Jouandeau, N.: On the Parallelization of UCT, *Proceedings of the Computer Games Workshop 2007 (CGW 2007)* (van den Herik, J., Uiterwijk, J., Winands, M. and Schadd, M., eds.), MICC Technical Report Series, No. 07-06, Universiteit Maastricht, pp.93–101 (2007).

2) Chaslot, G. M., Winands, M. H. and van den Herik, J. H.: Parallel Monte-Carlo Tree Search, *Proceedings of the 6th International Conference on Computer and Games* (2008).

3) Dailey, D. and volanteers: 13x13 Scalability study, Website (2008). http://cgos.boardspace.net/study/13/index.html.

4) Dailey, D. and volanteers: 9x9 Scalability study, Website (2008). http://cgos.boardspace.net/study/.

5) Gelly, S., Hoock, J.-B., Rimmel, A., Teytaud, O. and Kalemkarian, Y.: The Parallelization of Monte-Carlo Planning, *ICINCO* (2008).

6) Gelly, S., Wang, Y., Munos, R. and Teytaud, O.: Modification of UCT with Patterns in Monte-Carlo Go, Technical Report 6062, INRIA, France (2006).

7) Kato, H. and Takeuchi, I.: A Study on Implementing Parallel MC/UCT Algorithm, *Proceedings of 12th Game Programming Workshop 2007*, IPSJ (2007). http://www.gggo.jp/. (In Japanese).

8) Kocsis, L. and Szepesvári, C.: Bandit based Monte-Carlo Planning, *Machine Learning: ECML 2006* (Fürnkranz, J., Scheffer, T. and Spiliopoulou, M., eds.), Lecture Notes in Computer Science, Vol.4212, Springer, pp.282–293 (2006).

9) Stevens, W. R.: *UNIX Network Programming, Volume 1: Networking APIs: Sockets and XTI, 2nd Edition*, Prentice Hall PTR, Upper Saddle River, NJ, USA (1998).