Available online at www.sciencedirect.com

ScienceDirect

Procedia Computer Science 119 (2017) 22–29

Procedia Computer Science

www.elsevier.com/locate/procedia

6th International Young Scientists Conference in HPC and Simulation, YSC 2017, 1-3 November 2017, Kotka, Finland

A distributed system for conducting chess games in parallel

Aleksander Rydzewski[a], Paweł Czarnul[a,*]

[a] *Faculty of Electronics, Telecommunications and Informatics, Gdansk University of Technology, Narutowicza 11/12, 80-233, Poland*

Abstract

This paper proposes a distributed and scalable cloud based system designed to play chess games in parallel. Games can be played between chess engines alone or between clusters created by combined chess engines. The system has a built-in mechanism that compares engines, based on Elo ranking which finally presents the strength of each tested approach. If an approach needs more computational power, the design of the system allows it to scale. The system was designed using a loosely coupled architecture approach and the master-slave pattern. It works under Unix or MacOS operating systems. In order to split chess engine processing between every CPU in the system the Akka technology with the Scala language was used while the other part was written in Java. We tested many free chess engines connected to the system by the UCI protocol supported by the proposed system. CloudAMQP is an implementation of Advanced Message Queue Protocol and was used as a message-oriented middleware. This layer was created to split games between every available processing node connected to the system. This element also contributes to greater fault tolerance. We present results of games played between many available chess engines.

Peer-review under responsibility of the scientific committee of the 6th International Young Scientist conference in HPC and Simulation

Keywords: chess engine, Universal Chess Interface protocol, Elo ranking, cloud computing, loosely coupled architecture

1. Introduction

The first expert system was created in late 1960s. The idea behind it was to create a machine which would be an expert in a special branch of science and would advice people. In contemporary history of computer game playing, we should note several important wins of machines against humans. Example of those wins is the AlphaGo engine for the Go game, which in November 2015 won with European master in this game [16, 13]. Regarding chess, Deep Blue won with Garry Kasparov [14] and recent engines have matured to the degree to be able to win with top tier human players. A great example is also Komodo which won matches against chess masters in 2015 [11]. Each engine is tuned with weights and today typically uses a multithreaded implementation to make use of multi-core CPUs.

In this work, we propose a system that allows to make use of several nodes in a cloud and demonstrate how it can be used for playing tournament games in parallel between particular engines or even combine answers from several

* Corresponding author. Tel.: +48-58-3471288
E-mail address: pczarnul@eti.pg.edu.pl

1877-0509 © 2018 The Authors. Published by Elsevier B.V.
Peer-review under responsibility of the scientific committee of the 6th International Young Scientist conference in HPC and Simulation
10.1016/j.procs.2017.11.156

engines run in parallel to produce a move for a given position. In order to analyze a position on a chessboard by several engines, we can send a proper command to the system and it will distribute analysis to every free node available in the computing environment. We present several uses of the proposed system, both for parallelization of a tournament among chess engines to benchmark engine strengths and experiments in which responses from cluster engines are integrated when playing against individual engines.

2. Related work

Many chess engines today incorporate parallelization among cores of a CPU or CPUs available within a single node. Most of works related to chess in distributed systems focused on performance improvement of a chess engine or algorithm through distribution among several nodes. In ChessBrain [9] the main idea was, similarly to the SETI project [1], to use computing power of distributed machines to play a game of chess. Many players would create a network cluster by connecting to the SuperNode. The master would send the details of calculations which need to be performed by the nodes. After computations have been finished, the results are sent back to the master. ChessBrain II [7] proposed an improved architecture with network hubs – ClusterNodes which manage networks of PeerNodes. This allows to reduce the load of the main server. Such a multi-layered approach was later used in other volunteer computing projects, such as Comcute [5]. Paper [2] presents implementation of a distributed chess engine using the NodeJS framework and Heroku. It distributes Jobs in a queue and handles processing using many nodes. MongoDB is used. Paper [18] demonstrates the real-time volunteer computing platform called RT-BOINC evaluated for chess playing distributed across up to 800 volunteer cores and demonstrating increasing performance. Paper [8] adopts PV Splitting on a multi computer distributed system with an algorithm for CPU availability prediction and demonstrates increase of search speed.

In terms of game management, cutechess-cli [15] is a lightweight interface designed for connecting two engines into a multi-game battle against each other. It is a very useful tool if we want to manage games between two engines in order to obtain their relative strengths. It is written in C++ using the Qt framework.

Several rankings for chess have been conducted. These include the CCRL ranking [4] with engines run on various numbers of cores, including free and open source engines and the possibility to filter out engines by options. Chess Engines Grand Tournament [10] allows engines working in the Unix system but considers fewer options. FastGM [17] uses the cutechess-cli interface and collects many details about chess engines. For example, it contains documents on scaling to a larger number of threads. The rankings generally show similar engine orders.

3. Motivations

Examining existing solutions revealed lack of several interesting features that would be desirable in a single chess game playing management system and which are addressed by the system proposed in this work. These include:

- Clustering – an option to integrate decisions of particular engines. The proposed system allows several engines with a particular move integration and decision function to act as a single player.
- Scalability in terms of tournament play i.e. among several games and chess engines. Design of the new system assumed that everything should be scaled. The proposed system is even able to play several games on one machine, if resources allow to do it given imposed settings. The implemented solution is also able to split several games among available nodes.

As all chess rankings are based on the Elo algorithm [6], the best option to measure strength of new clustering methods was intuitively using a ranking based on Elo.

4. Proposed solution

The architecture of the proposed system is shown in Figure 1. It is composed of several components, with the following three being the most important in the context of this work:

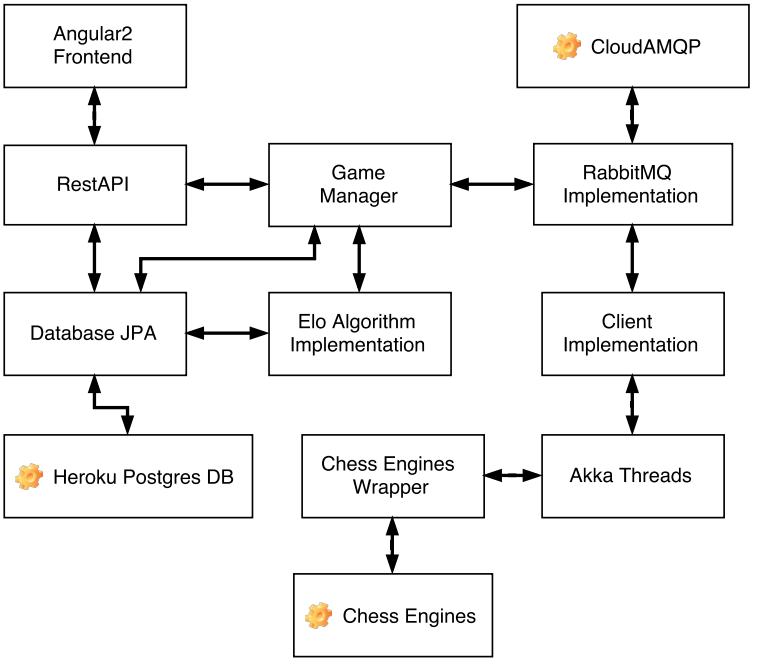

Fig. 1. General overview of system architecture

- Chess management – a module responsible for management of games on slave nodes.
- Message middleware – this layer of the system was built on CloudAMQP [3]. This message broker allows not only communication among elements in a loosely coupled architecture but also stores game details and releases these to slaves in a controlled manner. If any slave encounters an error then the message is taken by another one.
- Chess actor environment - This module uses the Akka technology [12] (JVM actor toolkit) to treat chess engines wrapped in UCI protocol implementation as actors. Actors will always works in parallel approach if this is possible for them.

4.1. Chess management

Game Manager is a module responsible for calculating new Elo ratings based on the results from games played by the slaves. The second task of this component is saving a new Elo value and data about votes taken in the game to the database. The third function of this particular element is to combine all necessary parameters which are needed to create a new game and insert these together to a new json message. This message is later sent to the end of an AMQP queue and waits for its own turn to initialize a new game.

4.2. Message middleware

The AMQP queue is a transportation layer (Figure 2), which in the proposed system constitutes a link between the master and the slaves. It usually contains more messages than the number of slaves. Each of the messages in this queue contains relevant information about a new game, depending on context:

- Type and value for a new game – for example ‘timeout 3000‘ means every move must take at most three seconds,
- Array of engine names – all engine names are stored in an array, firstly for the first cluster than for the second,
- Number of engines in each cluster – two numbers which denote the number of engines in each cluster,
- Elo value – Elo value of each chess engine,
- Result – result of the played game - 1 if the first cluster wins, 2 if the second, 0 if was a draw, -1 if one of engines wanted to make a move which is not allowed on a chessboard,

- Vote stats – numbers of decisions in the game taken by each of engines.

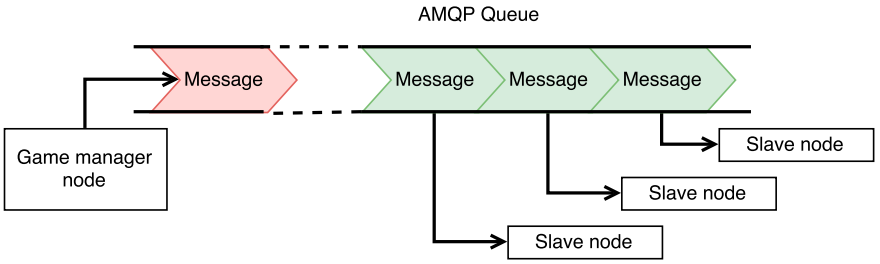

Fig. 2. An AMQP queue in the context of the proposed system

Each of the messages is taken by an available actor, and the game is started according to the information contained in the directive. After the game, two elements are added to the message. Those are ‘result‘ and ‘vote stats‘ as described above. The return message has the same information but is returned through a special queue which has a special identifier. This return queue was created at the moment of receiving the message from the master. If an unexpected event happens to the slave which has taken one of the messages, for example it has lost a connection or power, the message is automatically taken by another node. If all nodes are down the queue will still store all elements for the next few days.

4.3. Chess actor environment

Each of the slave nodes is composed of three elements. The first one is the AMQP receiver and it is responsible for taking the message from the defined queue and sending it back after the game. After checking all parameters, the message is transferred to the module, which initializes Akka actor's environment (Figure 3). Each of the actors is a different thread which can be handled side-by-side with other ones.

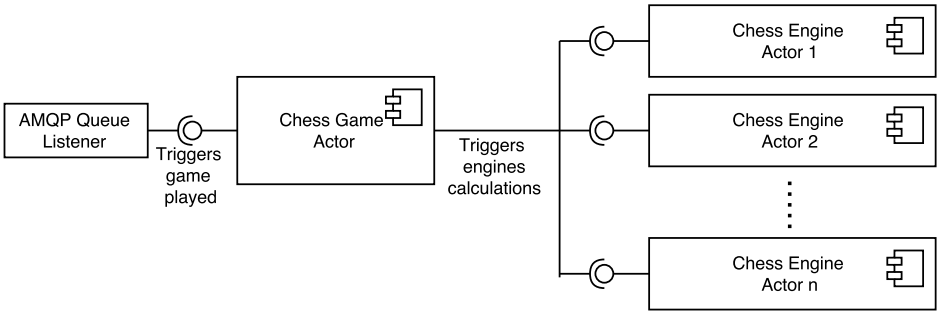

Fig. 3. Akka module components scheme

After proper initialization, the main actor called supervisor is run. It defines a new chessboard, elements of the game and initializes actors which are responsible for handling the engine calculations. Later, in each turn during the game, engine actors, which are responsible for the current player, turn on the component which initializes a chess engine and can communicate with it via the UCI protocol. After receiving the result, the engine actor terminates the engine responsible for calculation, waits until the CPU will recover all resources and passes the results to the supervisor. After recovering the data from engines which are responsible for the first player, the second player actors run engines to calculate moves which belong to this side. This process happens until the end of the game.

5. Experiments and Results

Three experiments were performed using the proposed system. The first one was to obtain relative engines' strengths using the Elo algorithm, as described in Section 5.3. In the next two tests, described in Section 5.4, we used clusters composed of a few engines. Each of those engine combinations was used with three different ways of voting for the final move. The first one (called EloVote) used engines' Elo values to vote on an move to make. If two weaker engines voted for a move different than the strongest engine in a cluster then this vote would be taken in the game. In the second approach (EloDist) a move was selected randomly out of those proposed by the engines with probabilities proportional to Elo values of the engines. Therefore strong engines were selected more often than weaker ones. In the third approach a move was selected at random out of those proposed by the engines.

The goal of the second test was to check the strengths of a chess engine cluster with selected voting methods. The aim was also to narrow down the number of the considered solutions for third test to the most interesting ones.

The third test was to check performance of chess clusters against individual chess engines.

5.1. Testbed environments

A cloud architecture needs a provider which can supply the system with necessary computation units. In this case this role was filled by Scaleway. For setting the initial chess engines ranking, an initial setting with ten machines was used with the following parameters: 4 virtual computation units, 4 GBs RAM, 200Mbit/s Ethernet connections. The environment for subsequent experiments used seventeen machines with the following parameters: 6 virtual computation units, 8 GBs RAM, 200Mbit/s Ethernet connections. The change was made due to memory size needed by a four chess engine cluster. Defined heap size required for these experiments was 5GBs.

5.2. Chess engines

A few requirements were imposed for chess engines used in the system. The first one was to be able to run in a UNIX environment and properly support the UCI interface. Some engines ignored the timeout rule and analyzed until the 'stop' command was sent.

5.3. Benchmarking chess engine strengths

During the first test several move timeouts and depth limits were used in order to benchmark the engines. Parameters in depth games were designed to check a shallow decision tree. They were subsequently 3, 5, 7 and 9 depths. Timed games had a greater spread with 3, 6, 9 or even 20 seconds per move. All time rankings revealed the same engine orders among the engines tested, i.e. from the best one: Stockfish 8 x64, Gull 3, Komodo 8 x64, Fire 2.2, Senpai 1.0, Cheng 4.39, Protector 1.6.0, Greko 2016, Ruy 1.0.6, Cinnamon 2.0, Fruit 2.2.1, Ethereal 8.13. We have noted that our ranking order is very similar to the one of the same engines in the CCLR ranking [4]. As a side note, shallow depth games showed different results than timed games.

5.4. Clustering engines with voting

The next two tests were to find performances of engines versus clusters. Tested engines were split into 3 groups: weak, average and strong. Six clusters were created. Due to the variety of the time parameters, clusters could have different combinations of engines, outlined below:

- Four strong/best - four best engines,
- Four average - four average engines,
- Four weak - four worst engines,
- Black sheep - three best engines and one average,
- Weaklings with leader - three worst engines with the best engine,
- 2Best2Best - two best time engines with 2 best depth engines.

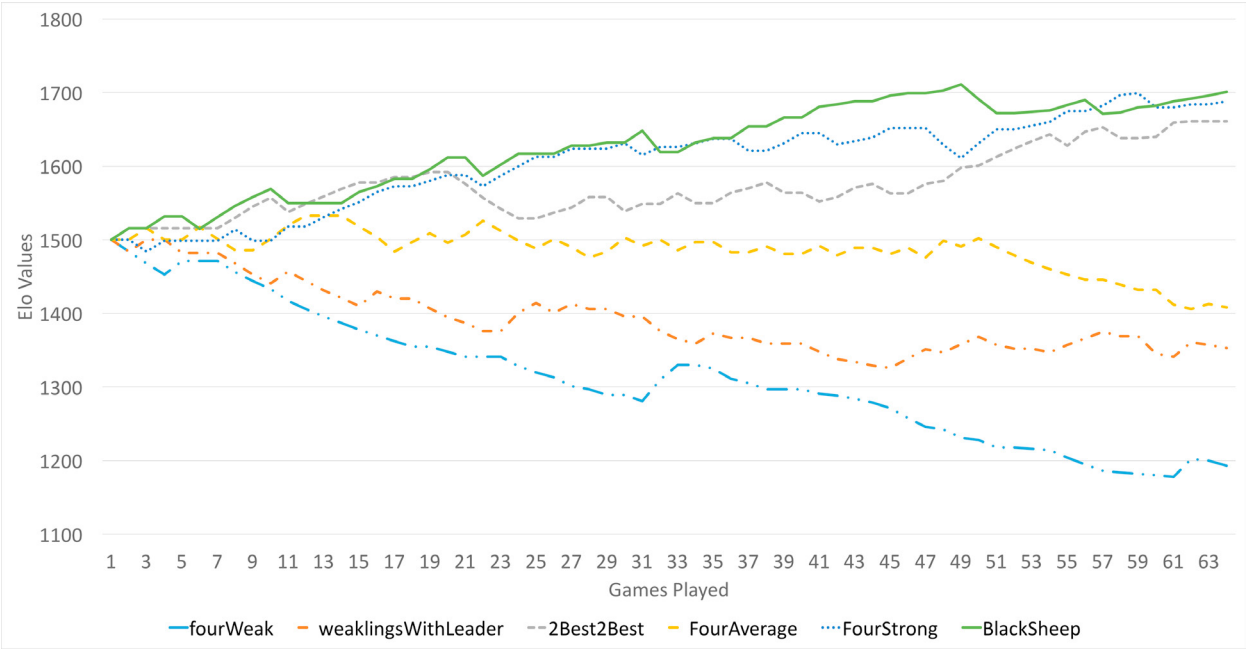

Fig. 4. Battle between clusters

The first test was to create a ranking using the same algorithm as in the first ranking to measure the order of the clusters. All of the tests resulted in the same ranking order. The strongest clusters are Black sheep and Four strong. The third position was taken by 2Best2Best. Later, much weaker was Four average, Weaklings with leader and the last was Four weak. Results of the initial phase of the tournament are shown in Figure 4.

As the best clusters in this ranking were Black sheep and Four strong then those engines were taken to phase three which was to measure how strong is engine cluster is versus a single engine.

The last test was to check the relation between clusters (chosen in the second phase) and engines included in this cluster. Clusters also work in EloDist, EloVote and Random approach. Cases based on random value were only around the average of engine rankings used in cluster. Best results were seen in EloVote cluster games and the results of this scheme are shown in Figure 5 (Elo values of engines are presented, those of clusters omitted).

As we can notice from Figure 5, the cluster is still worse than the best engine used in those clusters. However, we can also notice that the cluster can win against the best of engines.

Due to this interesting result, some additional research was conducted. In this research, two thousand games were played between Stockfish and two most promising clusters. Results have confirmed the initial findings that the cluster also wins quite often in a long-term game but generally it still is not as good as the engine itself. Elo rating changes of Stockfish are presented in Figure 6 and Figure 7. We noticed that in the long term play best engines were outvoted in 24% of all moves. Furthermore, next games were played between two Stockfish engines. We could notice low likelihood of winning in this type of game with default settings and 3 seconds per move. This might indicate that clusters could be used in upgrading original engines into new methods of play if the context i.e. board position or the phase of the game is taken into account during voting.

6. Conclusion and Future Work

In this paper key elements of a distributed system designed for playing chess games in parallel and results of some research performed with this particular system were presented. The system allows parallel processing both among separate nodes and within nodes and was deployed in a cloud. It allowed to benchmark several free chess engines and also clusters of engines with various voting algorithms. During research it was determined that a cluster usually plays weaker or equal compared to the best engine using voting or a randomized approach considering Elo

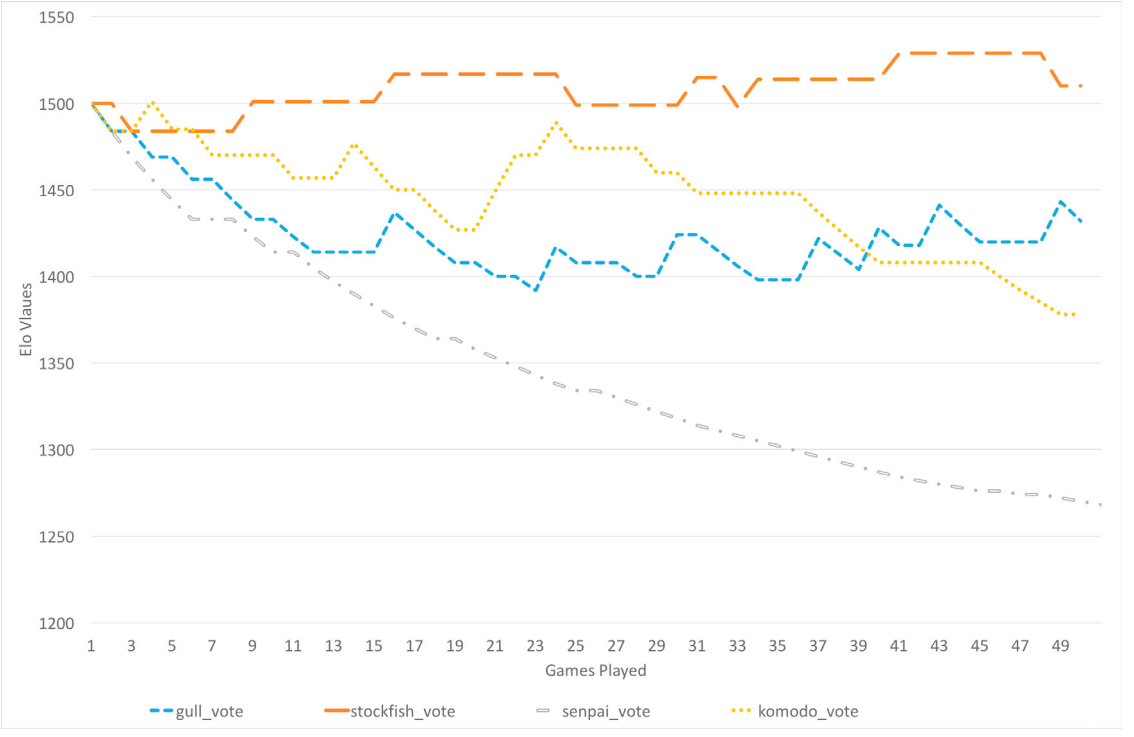

Fig. 5. Engines versus EloVote approach in Four strong

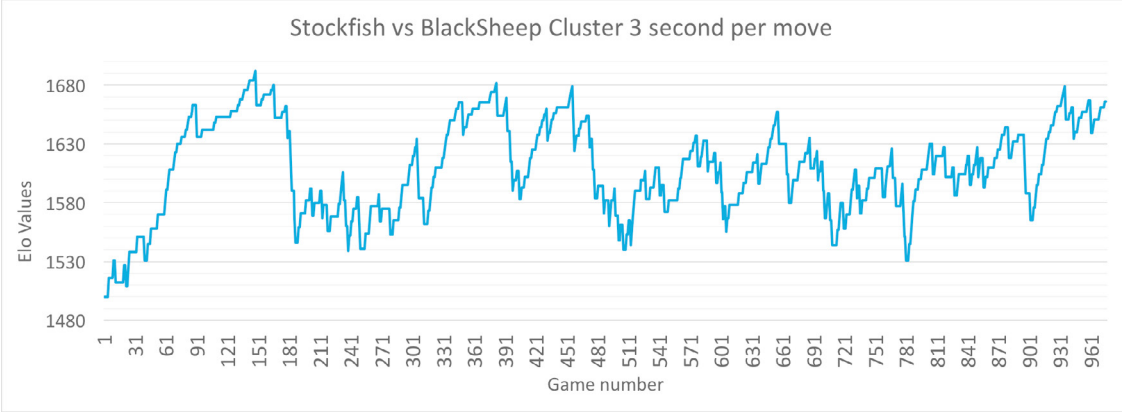

Fig. 6. Stockfish engine versus Black sheep cluster

strength. However, it was also observed at the same time that a cluster can win games against an engine much more frequently than the same engine playing against itself. This behavior possibly suggests that a chess expert system can be connected to identify positions in which the cluster might be better and possibly offer a better move depending on the context i.e. the position on the board or game phase.

Future work will explore the possibility to engage a neural network which will assign chess engines depending on the side of the board (Black or White) and the phase of a chess game or a position of the chessboard. Moreover, every chess piece could have a different value in a cluster than the one taken by the engine itself. Another possibility to use the proposed system would be to allow parallelization through running several instances of one engine to potentially increase performance through parallelization, similarly to the multithreaded parallelization within a node.

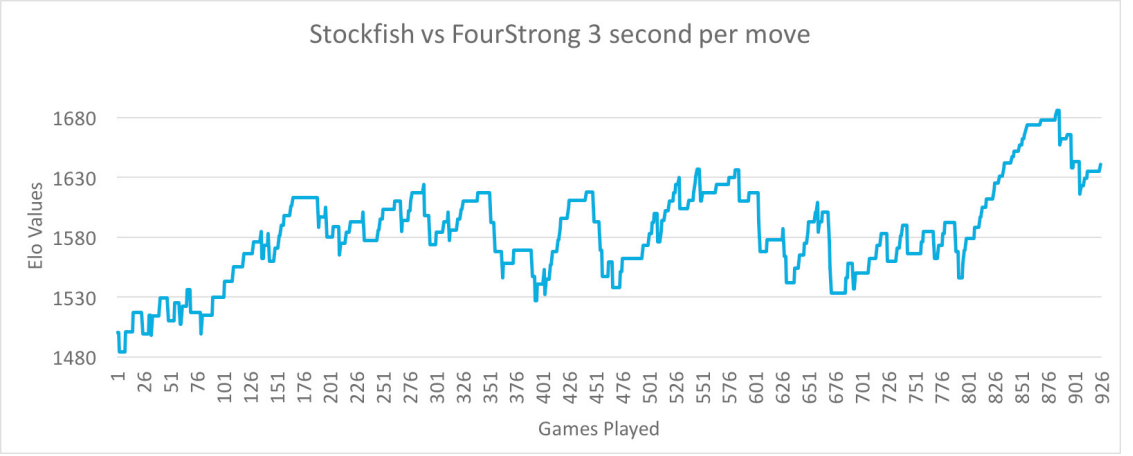

Fig. 7. Stockfish engine versus Four strong cluster

References

[1] Anderson, D.P., Cobb, J., Korpela, E., Lebofsky, M., Werthimer, D., 2002. Seti@home: An experiment in public-resource computing. Commun. ACM 45, 56–61. URL: http://doi.acm.org/10.1145/581571.581573, doi:10.1145/581571.581573.

[2] Badhrinathan, G., Agarwal, A., Kumar, R.A., 2012. Implementation of distributed chess engine using paas, in: 2012 International Conference on Cloud Computing Technologies, Applications and Management (ICCCTAM), pp. 38–42. doi:10.1109/ICCCTAM.2012.6488068.

[3] CloudAMQP, 2017. CloudAMQP Specification. [online]. https://www.cloudamqp.com/docs/.

[4] Computer Chess, 2017. Cclr ranking. [online]. http://www.computerchess.org.uk/.

[5] Czarnul, P., Kuchta, J., Matuszek, M., 2014. Parallel Computations in the Volunteer–Based Comcute System. Springer Berlin Heidelberg, Berlin, Heidelberg. pp. 261–271. URL: https://doi.org/10.1007/978-3-642-55224-3_25, doi:10.1007/978-3-642-55224-3_25.

[6] Elo, A., 2008. The Rating of Chess Players, Past and Present. Ishi Press.

[7] Frayn, C., Justiniano, C., 2007. The ChessBrain Project — Massively Distributed Chess Tree Search. Springer Berlin Heidelberg, Berlin, Heidelberg. pp. 91–115. URL: https://doi.org/10.1007/978-3-540-72705-7_5, doi:10.1007/978-3-540-72705-7_5.

[8] Hasan, K.S., 2011. A distributed chess playing software system model using dynamic cpu availability prediction. Software Engineering Research and Practice (SERP-11), WorldComp.

[9] Justiniano, C., 2003. Chessbrain: a linux-based distributed computing experiment. [online]. http://www.linuxjournal.com/article/6929.

[10] van Kempen, H., 2017. Cegt ranking. [online]. http://www.cegt.net/.

[11] Lefler, M., 2015. Komodo handicap matches. [online]. https://komodochess.com/store/pages.php?cmsid=17.

[12] Lightbend Inc., 2017. Akka webpage. [online]. http://akka.io/.

[13] Mackenzie, D., 2016. Why this weeks man-versus-machine go match doesnt matter. Science .

[14] McPhee, M., 2015. Deep blue, ibm's supercomputer, defeats chess champion garry kasparov in 1997. [online]. http://www.nydailynews.com/news/world/kasparov-deep-blues-losingchess-champ-rooke-article-1.762264.

[15] Pihlajisto, I., Jonsson, A., 2017. Cutechess-cli project page. [online]. https://github.com/cutechess/cutechess.

[16] Silver, D., Huang, A., Maddison, C.J., Guez, A., Sifre, L., van den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., Dieleman, S., Grewe, D., Nham, J., Kalchbrenner, N., Sutskever, I., Lillicrap, T., Leach, M., Kavukcuoglu, K., Graepel, T., Hassabis, D., 2016. Mastering the game of Go with deep neural networks and tree search. Nature 529, 484–489. doi:10.1038/nature16961.

[17] Strangmller, A., 2017. Fastgm ranking. [online]. http://fastgm.de/.

[18] Yi, S., Jeannot, E., Kondo, D., Anderson, D.P., 2011. Towards real-time, volunteer distributed computing, in: Proceedings of the 2011 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, IEEE Computer Society, Washington, DC, USA. pp. 154–163. URL: http://dx.doi.org/10.1109/CCGrid.2011.54, doi:10.1109/CCGrid.2011.54.