Towards Real-Time, Volunteer Distributed Computing

Sangho Yi[1], Emmanuel Jeannot[1], Derrick Kondo[1], David P. Anderson[2]
[1]INRIA, France, [2]University of California at Berkeley, USA

Abstract

Many large-scale distributed computing applications demand real-time responses by soft deadlines. To enable such real-time task distribution and execution on the volunteer resources, we previously proposed the design of the real-time volunteer computing platform called RT-BOINC. The system gives low $O(1)$ worst-case execution time for task management operations, such as task scheduling, state transitioning, and validation. In this work, we present a full implementation RT-BOINC, adding new features including deadline timer and parameter-based admission control. We evaluate RT-BOINC at large scale using two real-time applications, namely, the games Go and Chess. The results of our case study show that RT-BOINC provides much better performance than the original BOINC in terms of average and worst-case response time, scalability and efficiency.

1 Introduction

In the latest decade, several large-scale grid computing platforms have been used for tens or hundreds of thousands of parallel applications where they have relatively long enough laxity time [16]. But recent demand for shorter response time has been one of the most prevalent requirements in many-task parallel applications. Some important applications include on-line vehicle routing, real-time digital forensics [6], interactive visualization [13] (possibly with precedence constraints), online audio/video coding [2], and some online strategy games (such as Go [12] or Chess [8]).

Previously, we aimed at enabling the execution of massively parallel, real-time applications on large (on the order of 10,000 nodes) distributed systems, such as volunteer computing platforms. In that work, we addressed one main challenge, namely, the server-side management of hundreds of thousands of tasks for efficient and bounded execution time. We worked on developing a prototype of the real-time volunteer computing platform called RT-BOINC [18]. We focused on how to provide low worst-case execution time for each operation and the server daemon process.

In this work, we present a full implementation of RT-BOINC, adding essential features including parameter-based admission control and deadline timers. We evaluate RT-BOINC using two applications with tight time-constraints, in particular, the games of Go and Chess. The deployment of these applications are enabled by the new aforementioned features of RT-BOINC. We conduct large-scale experiments with those two applications and evaluate RT-BOINC in terms of the average and the worst-case response time, efficiency and scalability. From the case studies, we show that RT-BOINC outperforms the original BOINC in all aspects while meeting scalability and real-time constraints given by the applications.

The remainder of this paper is organized as follows. Section 2 describes related work in large-scale grid computing. Section 3 presents the original BOINC, and Section 4 presents the design and internal structures of RT-BOINC in brief, how to admit requests, how to implement, and some remaining issues. Section 5 evaluates performance of RT-BOINC and BOINC with two practical case studies. Section 6 finally presents conclusions of this work.

2 Related Work

Large-scale computing systems using volunteers, such as XtremWeb and Condor, are tailored for maximizing task throughput, not minimizing latency on the order of seconds. This has been one big limitation on utilizing volunteer computing for a bunch of real-time, interactive parallel applications. For instance, in [16], Silberstein et al. proposed GridBot, which provides efficient execution of *bags-of-tasks* on heterogeneous collections of computing platforms including grid, volunteer, and cluster computing environments virtualized as a single computing host. While the system uses a hierarchical task management system, the system cannot provide the task-level guarantees of execution time.

Scalability and performance can be extended by adapting hierarchical systems, but they still often do not provide any guarantee of performance in terms of worst-case execution time. For instance, in [10], Kacsuk et al. proposed a SZTAKI desktop grid, which is a hierarchical system developed on top of BOINC server structures. They modified the original BOINC server to have multiple levels of workunit

distribution. By doing this, SZTAKI can reduce load on the primary BOINC server by using the second and third-level BOINC servers. But, each level of BOINC servers still has the same characteristics of the original BOINC, which performance is not guaranteed.

Possible, but an expensive solution is using the dedicated supercomputers. They can run real-time tasks with guaranteed performance, but computing with volunteers could be an low-cost alternative if it could support real-time guarantees. In the domain of very complex games, especially Chess games, Deep Blue [8] was the first machine defeat the human world champion in 1996. IBM developed a dedicated server system for Deep Blue, and the server achieved about 11.38 GFLOPS on the LINPACK benchmark. Since 2006, several researchers in the world have been developed MoGo, which is a software to find the next move in the game of Go. They adapted monte-carlo-based algorithms, and now, they are near the professional Go players in the 9×9 small board (with certain parameter settings) based on the cluster computing machines [12].

Recently, Wu et al. presented a volunteer computing-based grid environment for Connect6 application [17], where the game "Connect6" has very high complexity similar to Chess and the game of Go. They suggested an on-demand computing by using a "push" model on communication between the main server and hosts, because a "pull" model-based systems (such as BOINC) should wait for requests from the hosts to distribute jobs. However, they did not describe how to provide low-latency with guaranteed performance. There was no comprehensive evaluation that showing the effect of using a "push" model rather using a "pull" model according to the request rate, the size of jobs, and several other possible parameters.

Several related works also deal with reliability issues of volunteer computing resources [5]. These works use predictive mechanisms to indicate nodes' reliability for scheduling purposes. As such, they address important but orthogonal issues compared to this work. Here we focus on real-time server-side task management, and these mechanisms can be combined with other mechanisms that ensure reliability across volatile resources.

3 The Original BOINC

3.1 Internal Structure of BOINC

Figure 1 shows the internal structures of the original BOINC server platform and flow of work distribution and reporting. BOINC server consists of two parts, the main database storage, and server daemon processes.

Current BOINC projects are now working with tens or hundreds of thousands of volunteers in the world. To manage such large amount of volunteers in a server, BOINC

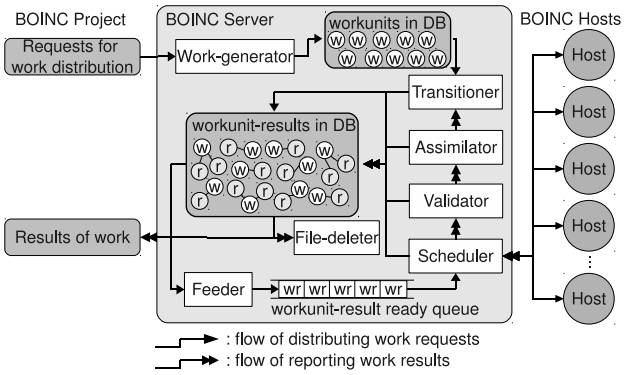

Figure 1. Internal structures of BOINC server

works in a pipelined manner. When the *project-manager* sends work to the BOINC server, the *work-generator* creates several *workunits* on the main database. Then, *transitioner* makes *workunit-results* pairs on the database. The pairs are fed to the *scheduler* by the *feeder*, and they are distributed to multiple BOINC hosts. When each host returns its result, the *scheduler* reports it to the *validator*. When the validation is completed, the data will be processed and finalized by the *assimilator*. Finally, the *project-manager* gets the aggregated results from the BOINC server.

3.2 Characteristics

BOINC is geared towards large-scale and long-term computation. The execution time of each workunit is relatively long enough, so that the BOINC server performs a relatively small amount of work distribution and reporting at the same time. Existing BOINC projects handle about $1 \sim 50$ workunits per second [1, 7].

However, when computing highly-interactive and short-term tasks with deadlines, the BOINC server must perform a relatively large number of transactions per period to guarantee the worst-case performance. In Fig. 1, most of the daemon processes read/write the main database. This means that the execution time of each daemon process depends on the number of records n in the database storing application, host information, workunit, and result data, for example. MySQL in particular has $O(\log n) \sim O(n^2)$ time complexity [15]. We previously have found that the daemon processes have at least linear, and up to polynomial complexity, thus, this makes it hard to provide relatively low *worst-case execution time* compared with the *average execution time* for all data-related operations and processes.

3.3 Challenges to Support Low-Latency, Real-Time Performance

If we assume that the number of volunteer hosts is about $10,000$, each move should be calculated within *20 seconds*,

each worker program on host-side consumes *5 seconds*, and the communication delay is around *5 seconds*, then the expected number of transactions between hosts and the server is at least $10,000/(20-5-5) = 1,000$ per second. This means that the server should finish each transaction less than $1\ ms$. If the applications need guaranteed real-time execution, the worst-case execution time on the server-side should be less than that amount of time for each transaction. To provide such a *low bounded execution time*, the internal server structures should be designed to limit the performance gap between the *average* and the *worst-case* execution time.

In addition, the server should be able to control *admission* of each request to provide the guaranteed execution within given laxity to deadline. To do this, the admission controller also needs to know the execution time (or, time consumption) on both server and client-sides including network communication delay. Thus some practical case studies are necessary to understand or expect the time consumption on the volunteer computing networks which is hard to expect with mathematical analysis.

4 RT-BOINC Internals

In the recent work, we proposed the real-time volunteer computing platform, namely RT-BOINC, for real-time large-scale computing on top of grids, and volunteered resources [18]. We presented the internal in-memory data structures which is mainly for providing O(1) complexity on retrieving data records including workunit, record, host, and user entries. In this section, we focus on new features and some implementation issues on RT-BOINC.

4.1 Assumptions

We have the two main assumptions in this work. First, we assumed that the volunteer hosts are highly available with the small fraction of time, because the RT-BOINC works with very short-term, interactive applications which works normally less than 30 seconds. In the recent work [9], we have found that the availability of each host may hardly be changed during very small time duration. Some machines can be in unavailable situations infrequently, then we can circumvent their deadline-missing problems by using a deadline clock timer on a server, and the server may not utilize the unavailable machines in the next round of task execution. Second, we assumed that the worst-case network communication delay can be measured as a constant value. Note that, in theory, the network communication delay varies over the volunteered resources, and in practice, it is hard to be bounded with a constant value. We will focus on relieving the above assumptions in our future work (see Sec 4.7).

4.2 Overall Structure of RT-BOINC

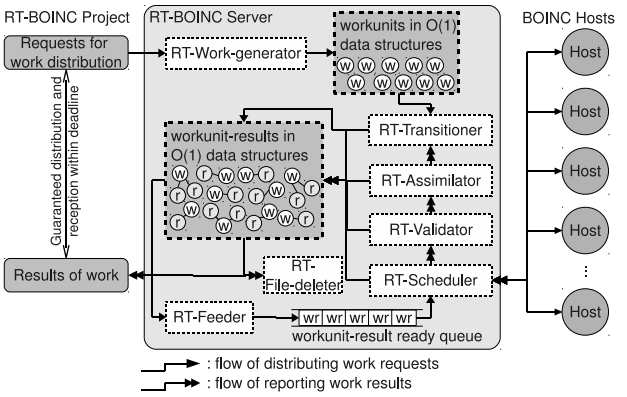

Figure 2. Internal structures of RT-BOINC

The main goal of RT-BOINC is to provide massively parallel computation for highly interactive, short-term real-time tasks with deadlines. RT-BOINC was designed to provide guaranteed real-time performance for distributing work and reporting their results in the BOINC server. To do this, we modified several components of the original BOINC server (see Fig. 2), and added new data structures and interfaces for retrieving them. The major difference between the original BOINC and RT-BOINC comes from the management of data records. RT-BOINC does not use the central database, and uses instead only *in-memory data structures* shared among daemon processes. We also modified the internal processing structures of the server daemon processes to reduce their complexity.

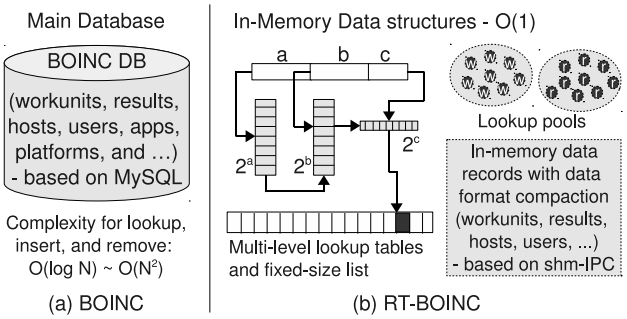

Figure 3. Comparison of data management of BOINC and RT-BOINC

Figure 3 shows a comparison of data record management between the original BOINC and RT-BOINC. The original BOINC uses MySQL as the main DBMS, and this widens the gap between the *average* and the *worst-case* execution time for reasons discussed in Section 3.2. In RT-BOINC, we replaced the database with *in-memory data structures*,

which provide $O(1)$ lookup, insertion, and deletion of data records. The data structures are shared by several daemon processes via *shared memory IPC*.

Exclusion of the main database leads to the faster performance on the server-side daemon processes. For example, the *scheduler* process works every coming *sched_request* messages from clients. When a request comes, the *scheduler* performs *matchmaking* which is finding the best amount of workunit for the requested client. At every request, the *scheduler* needs to access the main data records (the database in case of BOINC) and it may happen n-times per second where n is the number of clients and when each client requests the server per second. Thus, the performance improvement of each data retrieving operation can lead to significant improvement of each server daemon process.

4.3 Worst-Case Performance vs. Memory Space Overhead

The dedicated in-memory data structures have non-negligible memory space overhead on the server platform. For example, if the RT-BOINC server works with the three-level lookup tables with the same setting described in [18], and communicates with $10,000$ clients, the in-memory data structures require 421MB, and the whole memory space requirement of the server becomes roughly 1.09GB. In that setting, the previous results showed that the RT-BOINC is roughly $1,000$ times better than the original BOINC in terms of the worst-cast execution time for each operation [18].

When we have $100,000$ clients, the whole memory requirement becomes roughly $7 \sim 8$GB, but this is still reasonable with the current common-off-the-shelf 64-bit machines. In addition, we can adjust the depth of the O(1) data structures according to the given parameters of the target applications. If the target application has relatively bigger laxity time to deadline, the server manager can reduce the depth of the multi-level lookup tables to have less memory space usage with higher bounded execution time.

Note that, the bounded execution time can vary on the different machines, and different settings on RT-BOINC data structures. Previously, we have provided detailed experimental results in [18] which represent the average and the bounded execution time of each operation and each process with the given target system environment as a reference.

4.4 Server Processes for RT-BOINC

As we mentioned, each daemon process in BOINC has at-least linear time complexity for handling data records such as workunits and their results. To reduce complexity by orders of magnitude, we modified the internal structures of the server processes. We replaced all the BOINC database-related code with $O(1)$ lookup, insertion, replace, and deletion code. We removed unnecessary loops and redundant code from the remaining parts of the server processes.

4.5 Parameter-based Admission Control for Real-time Response

RT-BOINC server works with several daemon processes, and they share processing cores and the data records in main memory during the run-time. Thus, to guarantee the worst-case performance, we need to control admissions for every work distribution request. To do this, we need to use the measured bounded execution time for each process, and need to know the flow of work distribution process.

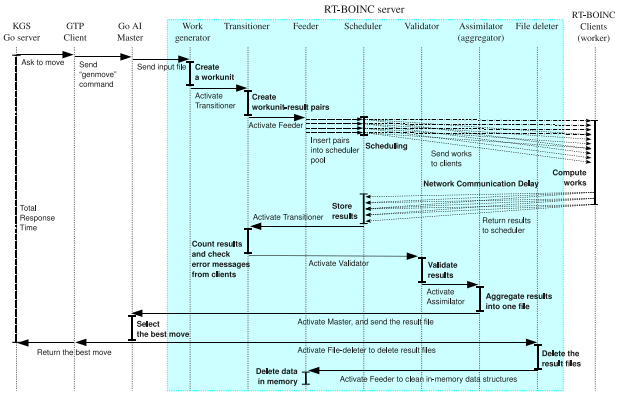

Figure 4. A work distribution process on the RT-BOINC (the game of Go example)

Figure 4 shows an example, the game of Go application's work distribution process when using RT-BOINC. Each daemon works with just one CPU core except for the *scheduler*. In both the original BOINC and the RT-BOINC, the *scheduler* is not actually a daemon process, but a program instance initiated by clients' request messages. Therefore, the server may have multiple *scheduler* instances at the same time, and they may consume multiple CPU cores to handle a lot of requests from clients. In the server, one simple program code controls admissions of work requests based on the measured worst-case execution time for each process and the given parameters from the request itself (e.g. the parameters include the degree of parallelism, the time requirement of a work, and the amount of time to deadline). The following Eq. (1) presents how to calculate the expected worst-case response time for the given request r and the number of cores in the server-side platform.

$$T(r, n_c) = w_c(r, n_c) + \sum_{k \in S_d} w_k(r) + w_{tr}(r) + \frac{2 \cdot n_p \cdot w_{sc}(r)}{n_c} \quad (1)$$

Here, r denotes a request for task distribution and processing over volunteered resources. The request r has several parameters, such as the work requirement time of a task for each volunteer host, and the deadline time of the entire task distribution and processing. The number of host clients to be used, which is the degree of parallelism for the computation, is also a parameter given by the request r. The sizes of the input and the output data are the possible constraints because they affect the network communication delay. In this equation, the n_c is the number of available cores on a server. Using the basic parameters we defined the $w_k(r)$, which is the worst-case execution time of a k^{th} server process with given request, and $w_c(r, n_c)$ is the worst-case network communication delay between a server and host clients with given request and the server-side parameter. The S_d is the set of all server processes including the daemon processes, and the *master* process which works with the target application. Finally, the $T(r, n_c)$ is the expected worst-case task completion time.

In this equation, the worst-case execution time for the *transitioner*, $w_{tr}(r)$ is added twice because it works twice in the distribution process (see Fig. 4). Also, the execution time for the *scheduler* $w_{sc}(r)$ is multiplied by the two times of the number of hosts and divided by the number of available CPU cores, because it is executed by job requests and responses from the hosts, and multiple instances can be run in parallel. Using the equation, the server system expects the worst-case response time of a given task request, and then the value $T(r, n_c)$ is compared with the deadline to decide whether to admit the request or not. In this way, the server system controls the admission of the task requests.

4.6 Implementation

We implemented RT-BOINC on top of the BOINC server source code[1]. RT-BOINC has 99% backward compatbility with BOINC. The full source code and sample test applications of RT-BOINC (including Chess and the game of Go) are available at the following website: `http://rt-boinc.sourceforge.net/`. The current RT-BOINC used in this paper is the version released in Nov. 2010.

4.6.1 Data Structures and Interfaces

We implemented the data structures using *shared memory* IPC among several daemon processes. The current implementation of RT-BOINC supports up to 64K active hosts, which is reasonable based on the size of most BOINC projects [3]. To provide $O(1)$ lookup, insertion, and deletion operations on the data structures, we used three-level lookup tables and fixed-size list structures as we have presented in [18]. We used a 4-bit lookup table for each level, thus each lookup table has $2^4 = 16$ fields. The current implementation allows the number of data records for each table up to 50K, and the total memory usage is 3.76GB for 50K available records.

[1]We used the *server_stable* version of BOINC released in Nov. 2009.

4.6.2 Application-specific Code Development

When we need to create a project, we should modify a few parts of the RT-BOINC to support the target BOINC application. For instance, if we perform a Game of Go AI parallelization project, we need to know the input and output format of the client-side application. Then we need to modify the *work-generator*, *validator*, and the *assimilator* because they are dependent to the client-side application. In the previous example in Fig. 4, we also need to develop a *master* daemon process, which reads the aggregated result and decides the best move. The rest parts of RT-BOINC such as *feeder*, *transitioner*, *file-deleter*, and *scheduler* are still independent to the project settings and the target applications.

4.6.3 Deadline Timer

To circumvent possible unavailability, deadline missing, or non-responding clients, we added a new deadline timer on the RT-BOINC. The deadline timer is initiated by the *work-generator* when creating a workunit, and the timer triggers the *validator* when it has expired. Then the *validator* performs validation with subset of the available (successful) results, and the in-progress (non-received) results are requested to be ignored and discarded.

4.7 Remaining Issues

We have some remaining issues and possible work in the future to improve the current RT-BOINC in terms of both theoretical and practical point of views.

4.7.1 Dynamic In-memory Data Structures

The current RT-BOINC works with *shared-memory-segment* over several server daemon processes. When the RT-BOINC requires more available data entries, the *shared-memory-segment* should be re-allocated with bigger size, and copied from the previous to the newly allocated memory. If this happens at run-time, the server will not work during certain amount of time to enlarge the *shared-memory-segment*.

4.7.2 Scalability / Extensibility

The major challenge on existing BOINC-based projects comes from the single server-based architecture. The server is a single point of failure, and it has limited capacity and network bandwidth. SZTAKI's method [10] is an easy way to distribute the server-side load to several hierarchical servers. By forming multilevel RT-BOINC servers with tree structure, we can extend both the scalability and extensibility. However it has some trade-off relationship between scalability and latency, because servers on each level may have additional latency to distribute, manage, and aggregate multiple results.

4.7.3 Communication between the Server and Hosts

Currently working distributed computing platforms including BOINC and XtremWeb work with *pull* model between the server and hosts. This means the worker clients on volunteers *pull* tasks from the server. But for the short-term tasks, especially for deadlined tasks, the pull model may incur a little bit higher overhead on the server. Here is an example. When a chess player submits a task to RT-BOINC server to get the best move, the server will create workunit and prepare scheduling. The clients should *pull* the tasks as soon as possible to maximize the laxity time. Then, there is a trade-off. If clients request for tasks more frequently, the server needs to consume more time on handling requests. Otherwise, we may have more delay on delivering tasks from the server to clients.

Table 1. Comparison of push and pull model

Comparison	Pull	Push
Number of active connections	low	high
Memory space overhead	low	high
Number of communication packets	high	low
Number of scheduler invocations	high	low

Push model is an alternative way to communicate between the server and clients. This allows the server to *push* tasks to clients, and thus there is less delivering delay between them. It seems to be an ideal solution for short-term, interactive tasks, but it still has problems such as higher number active connections on server-side, and higher memory usage to manage connections. Many existing projects have more volunteers than the maximum available number of TCP/IP connections on a physical server. Therefore, we need to have a yet-another way of communication between the server and clients which is neither *pull* nor *push* model.

4.7.4 Latency-aware, Availability-aware Scheduling

In RT-BOINC, the major challenge comes from reducing latency and increasing laxity time. This means, we need to reduce the physical communication delay between server and clients. Latency-aware scheduling is a way to reduce the delay, and we can expect the worst-case communication delay. By using communication delay for each client as a major factor of scheduling, the server can distribute tasks to appropriate clients. Measuring the communication delay may have negligible overhead, and the latency-aware scheduling may reduce the total makespan time significantly when distribution of the communication delay has a high variance.

In addition, we need to consider the availability of each client. Availability-aware scheduling can reduce the number of non-responding clients within deadline. BOINC currently has the factor *error rate* to handle such unavailability, but it works only for statistical long-term expectation.

5 Performance Evaluation: Case Studies

In this section, we evaluate performance using two practical case studies on both BOINC and RT-BOINC and show several results in many aspects.

5.1 Environment Settings

Table 2. Specification of the server platform

Platform detail	Description
Processor	2.0GHz (8 threads) Intel Xeon E5504 (2 dual-quads)
Main memory	8×4GB (1066MHz) dual-channel DDR3
Secondary storage	1 TB disk (7.2K RPM)
Network interface	1 Giga-bit Ethernet
Operating system	Ubuntu 9.10 (64-bit) kernel version 2.6.31-19
Web and database	Apache and MySQL released in Aug. 2010
RT-BOINC detail	**Description**
Lookup tables	Three-levels tables first, second: 4 bits, third: 8 bits
Number of records	50K for each table in-memory data structure
Volunteer resources	Grid'5000 hosts (64-bit) grenoble, nancy, rennes, and sophia sites (40 $\sim$ 800 cores)

Table 2 shows the hardware and software specification of the base server platform. We used a general-purpose, off-the-shelf server system to measure practical performance of RT-BOINC without paying significant cost for the hardware platform. Volunteers were chosen using GRID'5000 sites [4].

5.2 Case Study 1: Game of Go (Baduk)

The game of Go, also known as Baduk, is a well-known, very complex game. In fact, numerical estimates show

that the number of possible status the game of Go is much greater than the number of atoms in the known universe. The full size board of Go is 19×19, but the small board which has 9×9 still has high complexity, and the best professional players are still much stronger than the best Go AI mechanism which is based on several cluster machines [12].

In the first case study, we focus on parallelizing monte-carlo-based Go AI based on RT-BOINC. Table 3 shows the parameters and their settings for the game of Go case study.

Table 3. Parameter setup for the game of Go

Parameter	Setting
Board size	9×9
The game of Go server	KGS Go Server
Protocol between players	Go Text Protocol
Player's AI engine	Fuego 0.4.1
Opponent's AI engine	GNU Go 3.8
Computation time for each worker	5 seconds
Deadline for each move	25 seconds

We used the small 9×9 board, and we made experiments on both BOINC and RT-BOINC. We used a "Fuego" for the player (on BOINC and RT-BOINC side), which is a well-known open-source monte-carlo Go engine (`http://fuego.sourceforge.net/`). Also, we used a "GNU Go" for the opponent, which is known to be 1st strongest open-source Go engine without using parallelism (`http://www.gnu.org/software/gnugo/`).

We used a KGS (`http://www.gokgs.com`) as the main server, and used GTP (Go Text Protocol) between two players. We made 10 games for each spot, and measured the average and the worst-case response time for making one move.

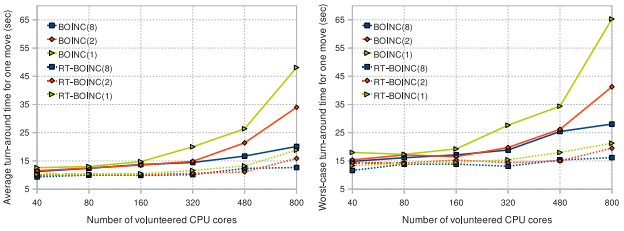

Figure 5. The time for deciding one move according to the amount of volunteer cores (left: average, right: worst-case)

Figure 5 represents the average and the worst-case response time for deciding one move according to different amount of volunteer cores. We made experiments with different cores on the server system to find out the performance difference based on the computational capacity of the main server. Here, BOINC(8) means that the BOINC server utilizes 8 cores on the server, thus, BOINC(2) and BOINC(1) use less cores on executing the server-side loads. As we observe in Fig. 5, BOINC-based Go AI has much longer response time when the number of volunteer cores are greater than 160. This means, the some execution flow on the BOINC server is saturated with 160 volunteer cores because of handling so much requests from the volunteers. However, RT-BOINC does not have significant difference up to 800 cores.

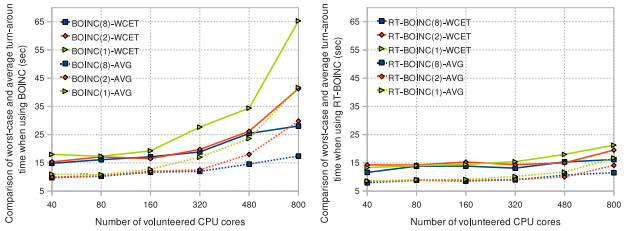

Figure 6. Comparison of the worst-case and the average response time (on the game of Go) according to the amount of volunteer cores (left: BOINC, right: RT-BOINC)

Figure 6 shows the comparison between the worst-case and the average time for making one move on the game of Go. In case of BOINC, there is a huge, $10 \sim 25$ seconds difference between the average and the worst-case, while RT-BOINC has less than 5 seconds.

5.3 Case Study 2: Chess

To play chess using RT-BOINC, the general idea we used is to distribute the current position to each workers, then ask each of them to do several random moves and use a chess engine to analyze the position reached after these random moves. Once this analysis is done, the worker returns: the sequence of random moves, the best possible move from the intermediate position and the score. When the server has received the answers, it builds a tree based on the results from the workers and determines the best move.

Stockfish: A UCI Chess Engine UCI (Universal Chess interface) is a protocol to communicate with chess engines. It provides ways to configure the engine, to define the position to search and to set several parameters (max depth to search, max time to search, etc.). It then returns what it estimates to be the best moves along with a score for each of them.

Among the available UCI chess engines we have used the "stockfish" chess engine, an open-source engine ported

on many systems (Windows, Linux, and Mac OS). It ranks second on CCRL (Computer Chess Rating Lists) website[2], and is evaluated with an elo of $3,218$ for long games at December 2010 (the best professional players on the world are around $2,800$ elo).

A Randomized Distributed Chess Engine Most chess engine algorithms are based on the min-max algorithm with alpha-beta pruning [11]. However, these algorithms require a global knowledge of the search state. Hence, they are not suitable for a distributed environment where communication between workers is difficult (if not impossible) to implement. Therefore, we need that each worker performs its computation independently. Moreover, for performance question, it is better if each client receives the same input. To fulfill all these constraints we have designed a randomized algorithm that works as follows.

- ◇ When it is time for the workers to compute the best moves the server generates a single input file with: the move number; the current position; the moves played since the start; the maximum allocated time for search; n: a number of random moves to be played before searching.
- ◇ When a worker receives such an input file it plays n moves at random starting from the current position and reach an intermediate position.
- ◇ When a worker has reached its intermediate position it start searching this position using a local chess engine (*e.g.* stockfish).
- ◇ When the allocated time for search (including the random moves) has expired, the engine returns what it estimates is the best move with its score. The workers then returns to the server an output file with: the move number; the starting position; the random moves; the intermediate position; the estimated best move with its score.
- ◇ The server receives the different answers from the workers. It aggregates the different random moves and the estimated best move of the different workers to form a tree. This tree has a depth of $n + 1$ (The n random moves plus the best move found at the intermediate position). Then it runs the min-max algorithm on this tree to determine what is the best sequence of moves among the ones explored by the different workers. Using this sequence it determines what is the best move to play from the current position.

[2] `http://www.computerchess.org.uk/ccrl/4040/rating_list_pure.html`

Uniform Random Search The key point of the above algorithm is the n random moves done by each worker. If we have a lot of workers and n is not too large it is expected that when we aggregate their answers the formed tree will cover all possible move combinations and hence the decision taken by the server will be the same than if it would have generated all possible intermediate positions at depth n from the current position and explore each of them using the chess engine. However, if the workers are not enough, or the value of n is too large then some sequences of move can be missed and the aggregation can be erroneous.

If the moves are chosen uniformly among all possible moves then the number of workers required to reach all possible intermediate positions is similar to the coupon collector's problem [14]: "how many sample trials are required to collect m coupons with replacement?". Analysis shows that for large m the number of trials is lower than $m \ln m + 10m$ in 99.99% of the cases([14] section 5.4.1).

For a typical chess position, there is about 30 legal moves. Hence $m = 30^n$. If $n = 2$ we have about 900 possible intermediate positions and we need around $15,000$ workers to be sure that in 99.99% of the cases all these positions are generated and explored by the workers.

Biased Random Search Given a position not all moves are equivalent and hence choosing one move should not be done uniformly. Indeed, it can be very easy to see that some moves are very bad (i.e. one put his queen in chess without obvious counter play) or very good (move leading to a quick mate). Hence, we should biased the random choice towards these good moves while keeping a chance that every moves can potentially be chosen.

To rapidly estimate which are the "good" moves, given the current position, the engine explores this position to a fixed shallow depth (in our case depth 5, as it takes some milliseconds). Then the engine ranks all the moves according to their score after this exploration.

We have done the following experiment to determine the quality of the ranking of moves after an exploration at depth 5. We have taken 1407 positions from the Spassky-Fischer games of the 1972 world championship match (these are all the played position excluding the one on the stockfish opening book). Most of these positions are tight positions and potentially very hard to analyze (among the 21 games we had 11 draws). The engine analyzes each position for 20 minutes (in general to a depth greater than 20) and returns a so-called "best move". During the search, we record the rank of this "best move" when the search reached depth 5. The result is plotted as an empirical cumulative distributed function (ECDF) in Fig. 7. For a given value on the x-axis, we give the proportion of the "best moves" that have a rank less or equal to this value when the search reached depth 5.

The result shows that 47.5% of the "best moves" are

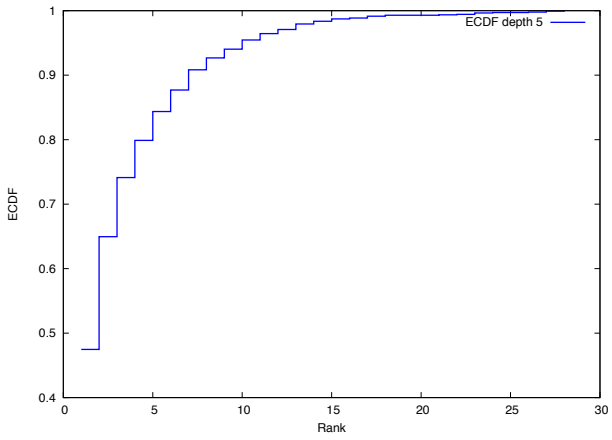

Figure 7. Empirical cumulative distributed function of the rank of the best chess move when search reaches depth 5

ranked 1 at depth 5, 17.5% of the "best moves" are ranked 2 at depth 5, ...and 0.07% are ranked 28 at depth 5.

Thanks to this experiment we have biased the search as follows: 1) Enumerate each possible moves. 2) Rank them according to the score computed when the search is limited to depth 5. 3) A move is chosen according to the empirical law shown in Fig. 7 (*i.e.* move ranked number one 1 is chosen with probability 0.475 moved ranked number 2 is chosen with probability 0.175 etc.). The advantage of such a biased search is that a good move is more likely to be chosen than a bad one, we provide redundancy of good position for the exploration of the next level and last, one need less node to explore the good part of the tree.

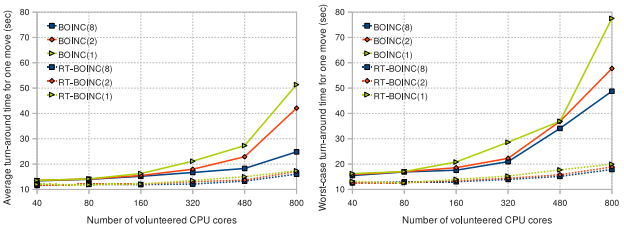

Figure 8. The time for deciding one move (on Chess) according to the amount of volunteer cores (left: average, right: worst-case)

Figure 8 shows the response time for one move on Chess according to the different amount of volunteers. Similar to the results in the game of Go, BOINC requires higher response time on larger number of volunteer cores. Especially, the input and output file for the Chess AI is a little bit larger than Go, thus it takes a little bit more time to distribute and aggregate them.

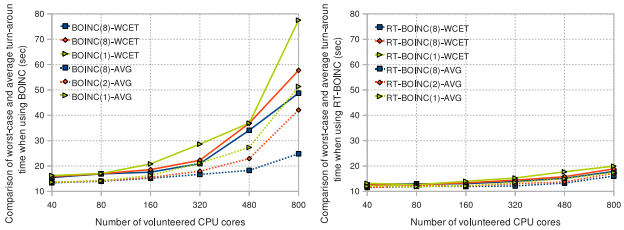

Figure 9. Comparison of the worst-case and the average time (on Chess) according to the amount of volunteer cores (left: BOINC, right: RT-BOINC)

Figure 9 presents the difference between the average and the worst-case when playing Chess games on different amount of volunteers. As we have expected, RT-BOINC has less difference, while BOINC has significant difference among both the volunteer cores and the server-sides cores.

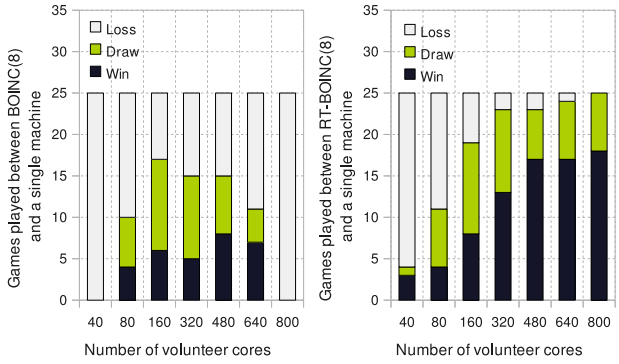

Figure 10. Results of Chess games (25 games per each) (left: BOINC, right: RT-BOINC)

Figures 10 and 11 shows the results of Chess games in several aspects. We made 25 games per each setting, and we observe that BOINC is not suitable for the Chess AI parallelization because there is deadline misses if there is high number of volunteers. Even when using low numbers of cores, there is a difference between RT-BOINC and BOINC. The difference comes from the fact that BOINC consumes more time to make each move, where the same amount of time is consumed by the opponent at the next turn. This is the main reason why RT-BOINC is still better than BOINC even though BOINC does not miss the deadline.

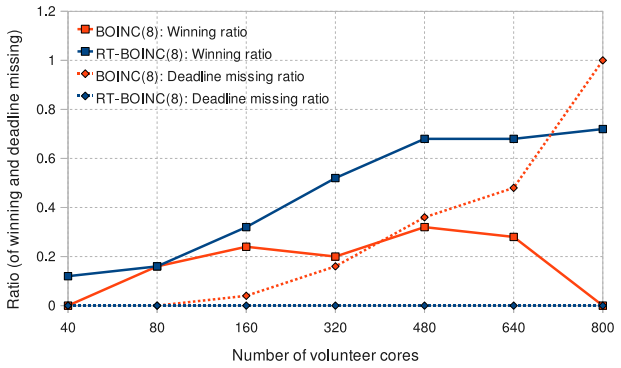

Figure 11. Winning, and deadline missing ratio on Chess games (25 games per each)

6 Conclusions

In this work, we presented a full implementation of RT-BOINC, with additional features such as deadline timer and admission controlling mechanism for providing guaranteed real-time performance on the work distribution and managing process. We evaluated RT-BOINC and the original BOINC using two applications with tight time constraints and high-parallelism, in particular, the games of Go and Chess. We conducted various experiments on the different scales with those two applications and the two server platforms in terms of both the average and bounded response time, scalability and efficiency. From the case studies, we showed that RT-BOINC outperforms BOINC in terms of both the average, the worst-case response time, deadline miss ratio, and scalability, while showing appropriate performance for the games of Go and Chess applications.

Acknowledgments

This research was supported by supported the ALEAE project (INRIA ARC), and the ANR Clouds@home project (contract ANR-09-JCJC-0056-01). Experiments presented in this paper were carried out using the Grid'5000 experimental testbed, being developed under the INRIA ALADDIN development action with support from CNRS, RENATER and several Universities as well as other funding bodies (see https://www.grid5000.fr).

References

[1] D. P. Anderson. *Talk at Condor Week, Madison, WI,* `http://boinc.berkeley.edu/talks/condor_boinc_06.ppt`, April 2006.

[2] Animoto Productions. `http://animoto.com/`.

[3] BOINC Statistics. `http://boincstats.com/stats/`.

[4] R. Bolze, F. Cappello, E. Caron, M. Daydé, F. Desprez, E. Jeannot, Y. Jégou, S. Lanteri, J. Leduc, N. Melab, G. Mornet, R. Namyst, P. Primet, B. Quetier, O. Richard, E.-G. Talbi, and I. Touche. Grid'5000: A large scale and highly reconfigurable experimental grid testbed. *Int. J. High Perform. Comput. Appl.*, 20:481–494, November 2006.

[5] K. Budati, J. Sonnek, A. Chandra, and J. Weissman. Ridge: combining reliability and performance in open grid platforms. In *HPDC*, pages 55–64, 2007.

[6] Capsicum Group: Digital Forensics. `http://www.capsicumgroup.com/contentpages/services/digital-forensics.html`.

[7] Catalog of BOINC Powered Projects - Unofficial BOINC Wiki. `http://www.boincwiki.info/Catalog_of_BOINC_Powered_Projects`.

[8] Deep Blue (chess computer). `http://en.wikipedia.org/wiki/Deep_Blue_(chess_computer)`.

[9] B. Javadi, D. Kondo, J. Vincent, and D. Anderson. Mining for Availability Models in Large-Scale Distributed Systems: A Case Study of SETI@home. In *17th IEEE/ACM International Symposium on Modelling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*, September 2009.

[10] P. Kacsuk, A. C. Marosi, J. Kovacs, Z. Balaton, G. Gombs, G. Vida, and A. Kornafeld. Sztaki desktop grid: A hierarchical desktop grid system. In *Proceedings of the Cracow Grid Workshop 2006*, Cracow (Poland), 2006.

[11] D. E. Knuth and R. W. Moore. An analysis of alpha-beta pruning. *Artificial Intelligence*, 6(4):293 – 326, 1975.

[12] C.-S. Lee, M.-H. Wang, G. Chaslot, J.-B. Hoock, A. Rimmel, O. Teytaud, S.-R. Tsai, S.-C. Hsu, and T.-P. Hong. The Computational Intelligence of MoGo Revealed in Taiwan's Computer Go Tournaments. *IEEE Transactions on Computational Intelligence and AI in games*, 2009.

[13] J. Lopez, M. Aeschlimann, P. Dinda, L. Kallivokas, B. Lowekamp, and D. O'Hallaron. Preliminary Report on the Design of a Framework for Distributed Visualization. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'99)*, pages 1833–1839, June 1999.

[14] M. Mitzenmacher and E. Upfal. *Probability and Computing: Randomized Algorithms and Probabilistic Analysis*. Cambridge University Press, New York, NY, USA, 2005.

[15] MySQL: Developer Zone. `http://dev.mysql.com/`.

[16] M. Silberstein, A. Sharov, D. Geiger, and A. Schuster. GridBot: Execution of Bags of Tasks in Multiple Grids. In *SC'09: Proceedings of the 2009 ACM/IEEE conference on Supercomputing*, New York, NY, USA, 2009. ACM.

[17] I.-C. Wu, C. Chen, P.-H. Lin, G.-C. Huang, L.-P. Chen, D.-J. Sun, Y.-C. Chan, and H.-Y. Tsou. A Volunteer-Computing-Based Grid Environment for Connect6 Applications. In *12th IEEE International Conference on Computational Science and Engineering (CSE'09)*, pages 110–117, August 2009.

[18] S. Yi, D. Kondo, and D. P. Anderson. Toward Real-time, Many-Task Applications on Large Distributed Systems. In *European Conference on Parallel and Distributed Computing (Euro-Par)*, August 2010.