An Improvement on Acceleration of Distributed SMT Solving

Leyuan Liu, Weiqiang Kong, Takahiro Ando, Hirokazu Yatsu, and Akira Fukuda

Graduate School of Information Science and Electrical Engineering
Kyushu University, Japan
Email: leyuan@f.ati.kyushu-u.ac.jp, weiqiang@qito.kyushu-u.ac.jp,
{ando.takahiro, hirokazu.yatsu, fukuda}@ait.kyushu-u.ac.jp

***Abstract*—Satisfiability Modulo Theories-based Bounded Model Checking consists of two primary tasks: (1) encoding a bounded model checking problem into a propositional formula that represents the problem, and (2) using a SMT solver to solve the formula. Solving the formula (namely, SMT solving) involves computation-intensive processes and is thus time-consuming. The target of this paper is to improve the distributed SMT solving techniques we previously proposed in [1] for further enhancing the effectiveness of SMT-based BMC. To this end, we improve the file dispatching scheme and reform the communication protocols in our previous work. In this paper, we describe the amelioration details and give a series of experiments to show the effectiveness of our improvement. Experimental results show that the improved implementation outperform the previous one. In addition to solving 8 groups of benchmarks by increasing the number of clients, we also make a preliminary experiment on increasing Central Processing Unit cores to investigate the influence.**

Keywords–Satisfiability Modulo Theories; Distributed Solving; Acceleration; MPI; OpenMP.

I. Introduction

Bounded Model Checking (BMC) is a restricted form of model checking [2] that analyzes if a desired property hold in bounded execution/behaviors of a system. In a nutshell, BMC can be explicit-state based BMC such as the methods described in [3] and symbolic-based BMC such as Binary Decision Diagram (BDD)-based [4], Boolean Satisfiability (SAT)-based [5] or SMT-based [6] BMC. It has been reported in [7] that symbolic-based methods perform better than explicit-based methods for verifying general Linear Temporal Logic (LTL) [2] properties. Among the symbolic-based BMC methods, the Satisfiability Modulo Theories (SMT)-based method is more expressible (thanks to its rich background theories) and is able to generate more compact formulas, and therefore, is more and more adopted by researchers and engineers.

SMT-based BMC consists of two primary tasks: (1) encoding a bounded model checking problem into a propositional formula that represents the problem, and (2) using a SMT solver to solve the formula, that is, finding a set of variable assignments that makes the formula true. Solving the formula (namely, SMT solving) involves computation-intensive processes and is thus time-consuming. Furthermore, as the model-checking bound increases, the encoded formulas become larger in size and harder to solve. The computational complexity of most SMT problems is very high [8], [9]. For all that, it is difficult to accelerate the SMT solving procedure for the engineers engaged in model checking. We have conducted [1] an implementation of using distributed computation and utilizing the power of multi-cores Central Processing Unit (CPU), multi-CPUs, and/or even cloud computing, to accelerate SMT solving. Although a series of experiments has shown the effectiveness of our implementation on increasing the solving efficiency, there still exist shortcomings which prevent the distributed solving to take advantage of CPU cores as much as possible. For example, the communication protocols could be reformed in order to reduce the unnecessary usage of network communication. In this paper, we describe our work on improving the distributed SMT solving by changing the file dispatching schemes that consider work load balance, and by reforming the communication protocols. We change file dispatching from coarse-grained to fine-grained, which can help in increasing the usage of CPU cores. Some unnecessary steps in the communication protocols are removed or merged. We have discussed the effectiveness of our improvement theoretically. We repeat the experiments conducted in our previous work [1] to make comparisons between these two schemes. We also conduct an experiment by increasing the CPU cores used in parallel SMT solving to investigate the influence in a microscopic view. The experimental results demonstrate the feasibility and efficiency of our improved implementation. However, we have also found, for a given target benchmark, that increasing CPU cores involved in computing will not always increase the solving speed.

The rest of this paper is structured as follows. Section II provides necessary preliminary knowledge and a brief introduction of the tools and techniques that are used in our work. Section III describes our previous work about distributed SMT solving. Section IV shows our methods used to improve the distributed SMT solving. Section V presents the experiments to evaluate the improvement and discusses the results. Finally, Section VI mentions possible extension (application scenarios) of our work and concludes the paper.

II. Preliminary Knowledge

A. Bounded Model Checking

BMC was first proposed by Biere et al. in [10]. At the early days, BMC is based on SAT solving [11]. It is commonly acknowledged as an complementary technique to BDD based symbolic model checking [5]. Recent years, with the development of modern efficient SMT solvers like Z3 [12] and CVC4 [13] etc., there is a trend to use SMT solvers instead of SAT solvers in BMC for better expressiveness. The basic idea of BMC is to search for counterexamples (i.e., design bugs) in transitions (state space) whose length is restricted by

 ISBN: 978-1-61208-339-1

an integer bound k. If no bug is found, then k is increased by one and the procedure repeats until either a counterexample is found or the pre-defined upper bound is reached.

B. Satisfiability Modulo Theories

SMT is a research topic that concerns with the satisfiability of formulas with respect to some background theories [14]. The development of SMT can be traced back to early work in the late 1970s and early 1980s. In the past two decades, SMT solvers have been well researched in both academic and industry, and achieved significant improving on performance and capability. Therefore, it has become possible to use SMT solver in BMC problem solving.

SMT is an extension of propositional satisfiability (SAT), which is the most well-known constraint-satisfaction problem [9]. SMT generalizes boolean satisfiability (SAT) by adding equality reasoning, arithmetic, fixed-size bit-vectors, arrays, quantifiers, and other useful first-order theories. An SMT solver is a tool for deciding the satisfiability (or dually the validity) of formulas in these theories [12]. In analogy with SAT, SMT procedures (whether they are decision procedures or not) are usually referred to as SMT solvers [15].

$$BMC(M,P,k) = I_0 \wedge \bigwedge_{i=0}^{k-1} T_i \wedge (\neg P) \quad (1)$$

In BMC, states and transitions among them are encoded to logic formulas like (1). Then the encoded formula is sent to a SMT solver. Solving the formula (namely, SMT solving) involves computation-intensive processes and is thus time-consuming. Furthermore, as the model-checking bound increases, the encoded formulas become larger in size and harder to solve. In certain circumstances, the time in solving the formula may be unacceptably long. Therefore, an acceleration is needed for this procedure.

III. Previous Work

A. Overview

We have done some preliminary work in [1] on accelerating SMT solving procedure by using Message Passing Interface (MPI) [16], [17] and Open Multi-Processing (OpenMP) [18]. Our attempt is distributed SMT solving. MPI is used to implement distributed computing (i.e., multi-CPUs) and OpenMP is used for multi-cores parallel computing. As mentioned in Section I, there are two primary tasks in SMT-based BMC. Acceleration techniques applicable to either task can increase the whole solving efficiency. In our implementation, we choose to accelerating the second task – SMT solving procedure.

We have implemented distributed SMT solving in C language, using Z3 SMT solver for satisfiability verification. The system has a Client/Server (C/S) architecture. The topology of the network is shown in Figure 1. All clients are connected to a center server. Data is transmitted between server and clients. The server responds to requests for acquiring files from clients. If there exist enough SMT files, then the files will be sent to the target client. The SMT solving procedure happens on the clients after receiving SMT files from the server. OpenMP is used to create multiple threads, each thread invokes a Z3 SMT solver to solve specific SMT files. The solving procedure will be finished until the server has no file to send.

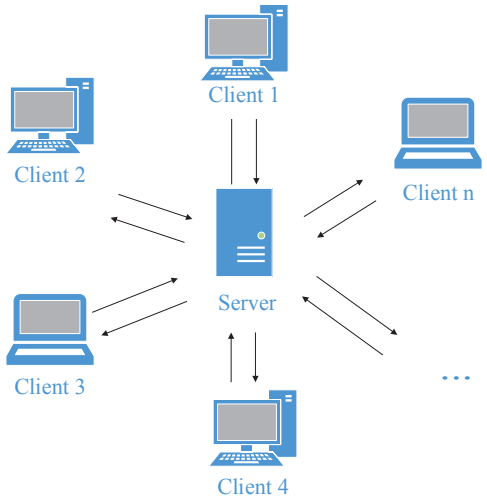

Figure 1: Network Topology

We conducted a series of experiments on six groups of benchmarks downloaded from the Satisfiability Modulo Theories Library (SMT-LIB) [19] that conform to version 2.0 of the SMT-LIB format. The benchmarks are `AUFNIRA`, `QF_UFLRA`, `AUFLIA`, `QF_UFLIA`, `QF_LRA-1`, and `QF_LRA-2`. The results shown in Table I and II are exciting. In Table I, the four benchmarks are easy problems, which means that SMT files can be solved in a short time. In addition, benchmarks in Table II are time consuming problems. The second column of the two tables shows the runtime which is obtained by applying sequential SMT solving. The third to fifth columns show the time of distributed SMT solving. The number of PCs (client) connected to the server is increased by one each time from 1 to 3 clients. The results in Table I show that the distributed solving strategy are proved effective for easy problems. We can obtain more than three times faster than serial solving in most benchmarks. The results, which are shown in Table II, are more positive when hard problems are considered. In the best case, the solving speed was raised by 36 times (3 clients are connected) comparing to the serial solving.

Obviously, in this way, not only the capacity/scalability, but also the solving speed of bounded model checking can be increased significantly. However, our strategy can not increase solving speed in all circumstances. When the problems to be solved are all small and easy solved, the efficiency boost is very limited. In the worst case, the speed only increased by two times even 3 clients are used.

TABLE I: MEASUREMENTS OF EASY PROBLEMS (SECOND)

Benchmarks	Serial	1 Client	2 Clients	3 Clients
QF_UFLRA	191.10	52.36	23.81	17.41
AUFNIRA	31.30	30.64	10.55	6.65
AUFLIA	105.80	88.87	41.90	50.48
QF_UFLIA	114.52	72.58	31.65	25.84

 ISBN: 978-1-61208-339-1

TABLE II: MEASUREMENTS OF HARD PROBLEMS (SECOND)

Benchmarks	Serial	1 Client	2 Clients	3 Clients
QF_LRA1	2465.78	1976.71	470.93	366.55
QF_LRA2	14796.03	11265.31	558.64	409.28

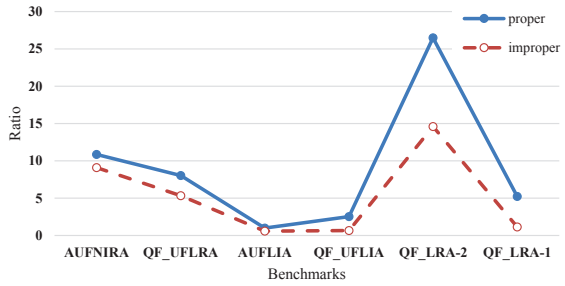

Figure 2: Comparison of Proper/Improper Task Dispatch

TABLE III: MEANING OF THE SIGNAL `power[0]`

`power[0]`	Meanings
`0`	Request files from a client or server has files to be send.
1	This client will be terminated.
2	No file in the server.

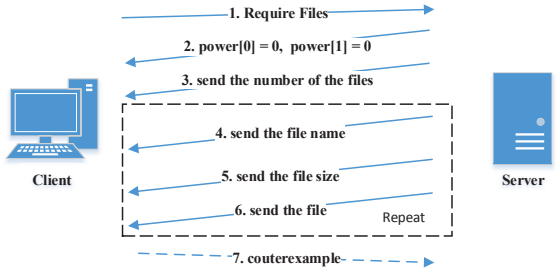

Figure 3: File Transferring Protocol

B. *Shortcoming*

Although our attempt gains significant improvement on solving performance, there are still some shortcomings of our previous work. The first is load balance, which is the core problems in the development of distributed model checker [20]. It means that our previous file distribution strategy is inefficient for scenarios where the hard problems or the combination of the easy and hard problems are considered. By *hard problems*, we mean problems that consume, e.g., 600 seconds or more per file in our experiment. The easy problems often take less than 1 second per file. In our previous work, the files are sent to clients by group. That means 4 files as a group are sent to a client after one file request. The server chooses files randomly. In other words, a client may receive easy problems as well as hard problems. For instance, there are four tasks named `Task1`, `Task2`, `Task3` and `Task4`. `Task4` is a hard problem and takes more time to solve. In a client, the 4 tasks are solved in parallel on different CPU cores. After `Task1` - `Task3` are finished, `Task4` is still being handled. In this case, 3 CPU cores are idle and no new tasks is assigned to them until `Task4` is finished. The best case is that all files have the same solving hardness. The more different the computing divergences are, the longer the total solving time is. A primitive experiment has been done to demonstrate this shortcoming. The result is shown in Figure 2. The two lines denote time-improvement ratio of distributed solving to serial solving. The blue solid line denotes the results where the workload is dispatched evenly to all clients (called *proper* case) while the red dash line denotes uneven dispatching (called *improper* case). It is clear that the proper dispatching gains higher improvement ratio than the improper case. It should be noted that this experiment is a trivial one just for demonstrating the importance of task dispatch. To summarize, an improper task dispatching can slow down the whole solving procedure.

The second shortcoming is that there are some unnecessary communication between the server and clients during file transfer. In our previously proposed file transferring protocol, which is shown in Figure 3, when we try to transfer one file to a client, a 3-time communication is needed (step 4 to step 6). Actually it is unnecessary to send two messages in step 3 and step 4. If we did so, we have to spend more time on establishing connections. In Table I (recall that all tasks in this table are easy ones that can be solved less than 1 second), when we increase the number of clients, the improvement are limited. One of the reasons is that the delays brought by establishing connections and the data transmission overwhelm the superiority gained by distributed computing. In addition, not only the file transferring stage but also other unnecessary communication between the server and clients could be reduced. In Figure 3, the array `power[]` is used to send controlling signals. The first element `power[0]` stores the signal's type. The second element `power[1]` is used to indicate the source of a message. The values and the meaning represented by the value are shown in Table III.

IV. Improvement on Previous Distributed SMT Solving

The purpose of our distributed SMT solving is increasing the solving performance by leveraging computing resources of multiple computers as much as possible. In Section III, we have discussed two main shortcomings in our previous work. These shortcomings prevent our distributed implementation from further enhancing the performance of the distributed solving. We try to improve the utilization of computing resources in the following two aspects.

A. *Fine-grained Dispatching Scheme*

The first considered aspect is changing file dispatching grain from coarse-grain to fine-grain. Our previous file dispatching scheme is coarse-grained. That means the minimum unit to assign workload is client which realized by one MPI process even 4 threads running in it. The client (process) sends request to the server and receives returned files. After it receives files from the server, the client dispatches files to different threads where the SMT solving is done in a parallel way. This procedure is shown in Figure 4. In this figure, the part boxed by a dash rectangles denotes a client connected to the server. The whole procedure starts from sending file requirement from a client to the server for the first time. If

 ISBN: 978-1-61208-339-1

there exits at least one file on the server, they will be sent to the client. Then the control flow enters the parallel solving stage. After the parallel solving, all four threads need to synchronize with each other before a new require-receiving round. The synchronization shown in Figure 4 means that threads which have finished their tasks in current round have to wait for other threads which are still in solving to finish their tasks. After the synchronization, the client will request new files again and the procedure will be repeated. The synchronization is needed because the file is dispatched to a client not thread. From the macroscopic view, the work load is dispatched to a client on process-level and the synchronization of threads will cause the shortcoming we discussed in the above section.

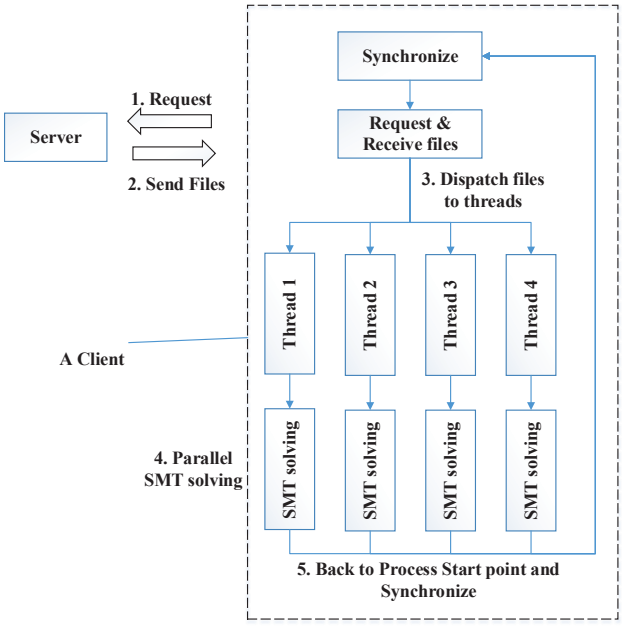

Figure 4: Previous File Dispatching Scheme

We change the dispatching scheme described above so that the synchronization between threads is removed after finishing one solving round. The improved scheme is shown in Figure 5. In our new scheme, the client starts from sending initial request to the server and receiving the first file set. It should be noted that this requesting-receiving round is executed only once. Then the received files are solved by 4 threads respectively in parallel. After that, 4 threads (in one client) will send file requests to the server separately when they need new files. The following receiving procedure is conducted by these threads also. If new files were received, threads will enter parallel SMT solving procedure again until a counterexample is founded or no files in the server. All four threads conduct the same procedures so that we admitted the detail of Thread 2 to Thread 3 in Figure 5. At this point, no synchronization is needed. Threads in a client are more independent than the previous scheme. The function of requiring and receiving files is implemented in thread level. Each thread can obtain new tasks from the server by itself. It is no longer necessary to wait for other threads to finish their solving tasks.

In our implementation with the new designed fine-grained dispatching scheme, the architecture is still C/S. The server runs in a loop to receive the messages sent by the clients

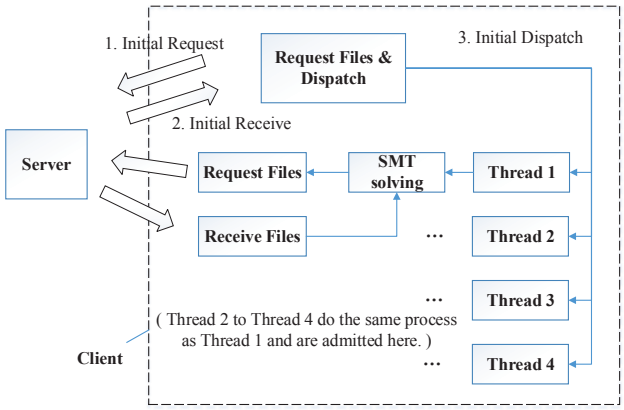

Figure 5: New File Dispatching Scheme

Algorithm 1. Newly Designed Client Procedure

```
1.  Process_n (int process_id) {
2.    Initialization and definition of variables;
3.    MPI_Send (request files, to process_0);
4.    MPI_Recv (file_existence_condition from server);
5.    if (no file is founded)
6.      return 0;
7.    j = 0;
8.    while (j < file_num) {
9.      MPI_Recv(file_information from server);
10.     MPI_Recv(file from server);
11.     fwrite (file to local HDD);
12.     rename(file);
13.     Clear receiving buffers;
14.     j++;
15.   }
16.   invoke parallel_solving(char *working_path);
17.   clean local files;
18.   MPI_Send(finish_signal to server);
19.   return 0;
20. }
```

and prepare files for them. So we only expand the length of the control message `power[]` without any other changes. A major change comes up on the client side so we present it here in Algorithm 1. The argument `process_id` is an unique `int` number to distinguish a process which is running as a client. At the beginning, the client sends a request to the server (Line 3) and receives the file existence condition. The function `MPI_Send()` is a message sending function supplied by MPI [16]. It is a basic blocking message send operation. Routine returns only after the application buffer in the sending task is free for reuse. The function `MPI_Recv()`, which also is a MPI supplied function, receives a message and block until the requested data is available in the application buffer in the receiving task. The two functions must be used in as a pair. Otherwise, a dead block will happen. If file exists, an initial receiving and dispatching procedure will be done (Line 8 to Line 15). The initial dispatching is done by a client in our design because at the beginning, all CPU cores are idle, there is no need to consider the load balance problem at that time. The parallel SMT solving will take place by invoking the function `parallel_solving()` with the parameter `char *working_path` which indicates the local path where the target files are saved in.

The parallel solving part is also changed. It is done inside each client. We use OpenMP [18] to create 4 threads to leverage computing capacity of clients as much as possible. As we mentioned above every thread in a client now has the

 ISBN: 978-1-61208-339-1

Algorithm 2. Newly designed Function parallel_solving()

```
1. parallel_solving (char *working_path) {
2.   omp_set_num_threads(m);
3. #pragma omp parallel
4.   {
5.       switch (omp_get_threadnum()) {
6.       case 0:
7.       {
8.         Exploring all files in working_path {
9.           invoke Z3 to solve;
10.          goto next file;
11.        }
12.        do{
13.            Require and Receive files from the server;
14.            if (no file is founded)
15.              break;
16.            receive all files in this round;
17.            Exploring all files in working_path_t0 {
18.              invoke Z3 to solve;
19.              goto next file }
20.            delete solved files in working_path_t0;
21.        } while(server_has_files)
22.      }
23.        break;
24.    ... // Case 1 to case 3 are mostly like case 0 and omitted here
25.      }
26.  }
27.  return 0;
28. }
```

ability to request files from the server if necessary. Meanwhile, the files sent by the server will be transferred to the thread to achieve our fine-grained dispatching. We show the procedure of function `parallel_solving`() in Algorithm 2. In each client, firstly, a thread resolves the files dispatched in the initial step of the client. Then it enters a `do {...} while`() loop to request and receive files until there is no file in the server. After one receiving round, which means that a thread has received a set of files from the server, the received files are solved by Z3 SMT solver. The number of files in a set is a variable and its value is set to 2 by default. The argument `working_path_t0` is generated from the argument `working_path` to indicate its own working path. Thread 0 to Thread 3 are running in parallel and we omit the pseudo-code of thread 1 to thread 3 in Algorithm 2. If a counterexample is found in a solving round, the thread will report to the server. This activity of a counterexample finding and reporting is the same as our previous one so that we omit it in this algorithm.

In practical implementation, the new scheme might be less effective for the case which the solved problems are all easy problem or easy problem dominated (most of files are easy problems). In such case, the solving time of a single file is short enough. In our previous scheme, the main time consuming is the synchronization of threads. Even if the previous scheme is used, the synchronization time is a short duration. It will not effect the total solving time using the synchronization or not. However, by using the new dispatching scheme, we expect to get better performance for solving hard problems or combined problems under the server's random file dispatching strategy.

B. Communication Reduction

The second aspect to improve solving efficiency is by reducing unnecessary communications between the server and clients. In Figure 3, we can emerge step 2 with step 3 firstly. We expand the control signal `power[]`, which is an array with two elements, to 3 elements. For instance, if the `power[0]` = `0` (means there exists files on the server), `power[2]` will be the number of the files while `power[1]` denotes the source of the message. If not, `power[2]` will set to 0 and `power[0]` = `2`. In the file sending stage, before sending file data, the server will informs the file names and sizes, which are used by the client to create a receiving buffer dynamically. We use a `struct`, which consists of the name and size of the file to be sent in the future, and the structure could be sent by using `MPI_send()` function once. The step 4 and step 5 in Figure 3 can be merged by using the new data structure.

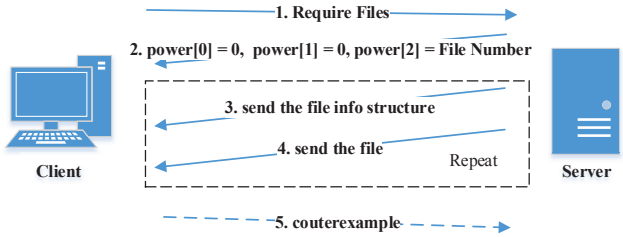

Figure 6: New File Transferring Protocol

The reformed protocol is shown in Figure 6. The new File transferring protocol needs one communication to send the control information and two communications before sending one single file, while the previous protocol needs two and three communications, respectively. The first merging brings an expansion of the array size from 2 to 3. In C language, one `int` element consumes 2 Byte memory. For a modern PC and Ethernet, 2 Byte is not an issue. In the second merging we use a `struct` to store the file name and size. Comparing with sending these information separately, it consumes more bandwidth for one time sending. Even so, the increased bandwidth overhead is noting to a 100M/1000M Ethernet.

$$T_{pre} = n * (\bar{s} + t_{name} + t_{size}) + (t_{control} + t_{number}) * n/number \quad (2)$$

$$T_{new} = n * (\bar{s} + t_{structure}) + t_{control} * n/number \quad (3)$$

However, not every case can gain positive effect on promoting solving performance. If the target problems are all hard problems the communication overhead is not obvious comparing with the solving time. But for easy problems, the situation is opposite. The solving time of a easy problem is much more shorter than establishing communication and transfer control messages. The constitution of previous distributed SMT solving time T_{pre} is shown in 2. n denotes the total number of files to be solved, $\bar{s}$ is the average solving of a single file. t_{name} and t_{size} represent the time of establishing communication and sending file's name and size respectively. $t_{control} + t_{number}$ are consumption of sending control signal and the number of files. *number* denotes files which will be sent by the server over one requirement of a client. After the reduction, the time consumption constitution T_{new} is shown in 3. In some cases, the average solving time $\bar{s}$ is no match for establishing the connection between the server and client. So reducing the time denoted by t_x can increase the performance significantly. t_x presents the time consummation of t_{name}, $t_{control}$ and t_{number}.

V. Experiments and Analysis

To evaluate the efficiency of our improved distributed SMT solving implementation, we conduct a serial experiments on

 ISBN: 978-1-61208-339-1

six groups of benchmarks downloaded from the Satisfiability Modulo Theories Library (SMT-LIB) [19] that conform to version 2.0 of the SMT-LIB format. The benchmarks are same as our previous work which are `AUFNIRA`, `QF_UFLRA`, `AUFLIA`, `QF_UFLIA`, `QF_LRA-1`, and `QF_LRA-2`. We will not go into the details of those benchmarks, which are basically not relevant to the topic of this paper. Please refer to [19] for the meaning of those benchmarks. We deploy the program of the above described algorithms in four PCs running Windows 7 Enterprise Edition with MPICH 2.0 installed. One of the PCs is used as the server and the others are as clients. The hardware of the PCs are as follows: PC1 has a quad-core Intel Xeon CPU (2.66GHz) with 8GB RAM; PC2 and PC3 have a quad-core Intel i7 CPU (2.7GHz) with 8GB RAM; PC4 has a Intel Core2 Duo dual-core CPU (1.8GHz) with 2GB RAM. All the four PCs are connected to 100MB Ethernet. To evaluate the effective of our improvement, we use our prototype to solve those benchmarks and compare the results of our previous work. As we mentioned in the previous section, the four benchmarks, which are `AUFNIRA`, `QF_UFLRA`, `AUFLIA` and `QF_UFLIA`, are easy problems and the benchmarks `QF_LRA-1` and `QF_LRA-2` are hard problem. We design some new experiments beside the previous one. `Combined-1` is a combination of easy problems with hard problems and `Combined-2` is a set of 3000 easy problems. We use our previous algorithm and the new algorithms on solving these benchmarks respectively to prove the effectiveness of the latter.

Firstly, we conduct same experiments using the improved implementation for benchmarks which are used in our previous experiments [1]. We perform a serial solving experiment for each benchmarks using PC1. The SMT files are solved one after another in a serial way. Then the clients are connected to the server one by one and the same benchmarks are solved. PC4 is used as the server. Secondly we perform the same procedure mentioned above on new benchmarks `Combined-1` and `Combined-2`. The results are shown in Figure 7. The vertical rectangular marked as grey denotes the solving time of serial solving. The blue vertical rectangular presents results using previous algorithm. The red vertical rectangular denotes the solving time by using our new algorithm with two improvements. It is obvious that our improvements are useful on accelerating our previous distributed SMT solving implementation, especially for combined benchmarks. In Figure 7(e) and 7(f) when we add clients to 2 and 3, the improvement seems elusive. The reason is that these two benchmarks contain one or more hard problems which take nearly 300 seconds to be solved. In other words, the limit of distributed solving time is about 300 seconds no matter how many clients are connected.

Our new improved architecture give us the ability to control the usage of CPU cores more precisely. This means that we can add threads involved in parallel solving procedure one by one, in an easier way than before. We conduct experiments with `Easy Benchmarks` and `Combined Benchmarks` to investigate the influence by increasing of CPU cores. We use PC4 as the server and other PC as clients. At first one client is connected, but only one CPU core is used, the second time the number of the CPU cores is increased to 2. Four cores will be used on each PC, after one PC reached the max value of used CPU cores, new client will be connected. We increase

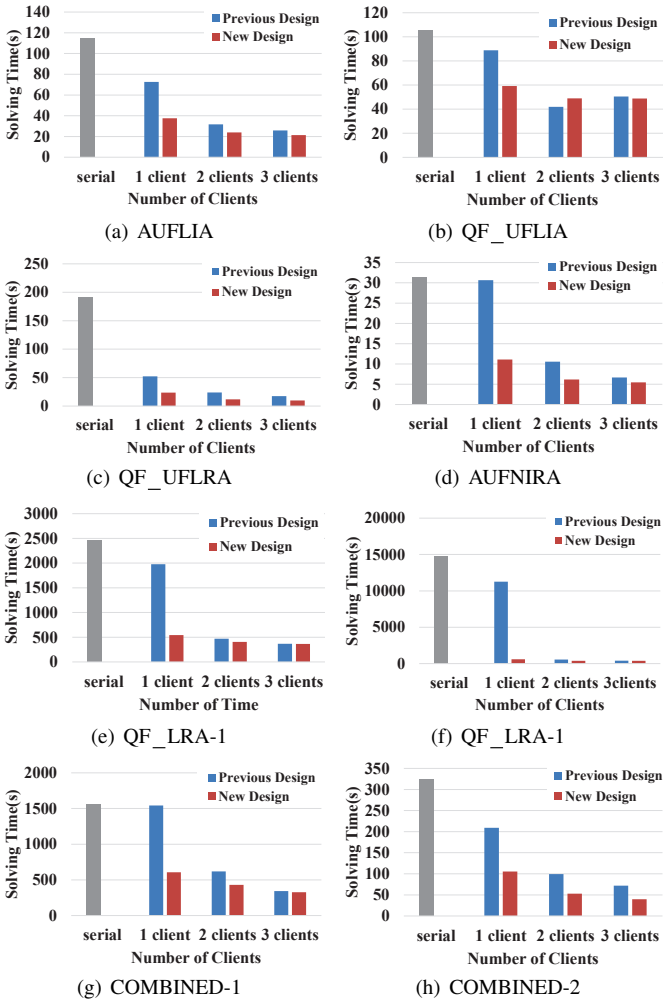

Figure 7: The Distributed SMT solving Results

the CPU cores by one each time and repeat this procedure with two benchmarks respectively. The results are shown in Figure 8 and Figure 9. The results show that the curve of solving speed decrease sharply until the fourth CPU cores are involved. After that, the curve becomes flat. We have mentioned the possible reason in the paragraph above. For solving the `Combined Benchmarks`, the solving time of the hardest single problem is a limit of distributed SMT solving. For `Easy Benchmarks`, due to the CPU cores are on different PCs which are distributed in networks, the more CPU cores involved, the more communication will take place. Considering the time consumption resolving single easy problem and the overhead taken by network communication, the whole communication time consumption will be the predominant factor. In other words, if the solving target is determined, the whole solving time consumption could not be decreased always by simply adding more clients.

VI. Conclusion

In this paper, we first described our previous work on accelerating SMT solving using a distributed computation architecture, and discussed its shortcomings. To tackle those shortcomings, we proposed the fine-grained dispatching scheme and communication reduction methods. A series of experiments

 ISBN: 978-1-61208-339-1

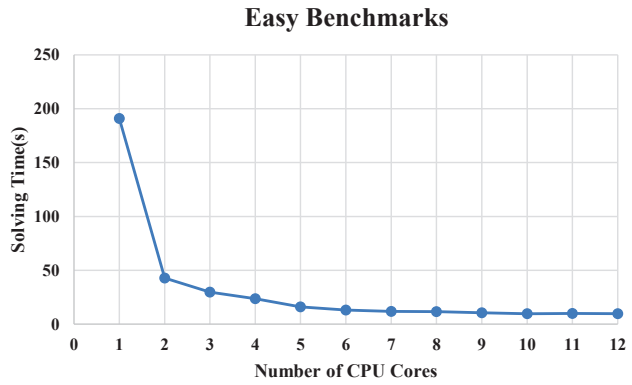

Figure 8: Influence by increasing CPU cores on easy benchmarks

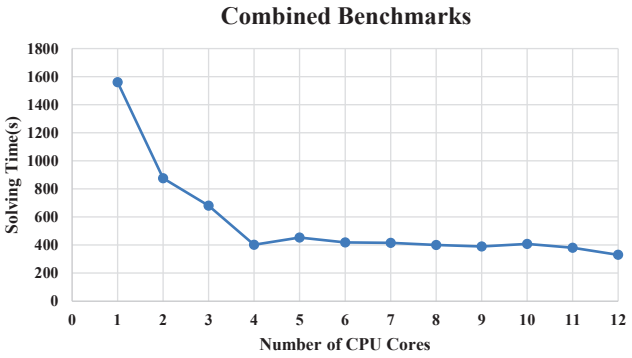

Figure 9: Influence by increasing CPU cores on combined benchmarks

were carried out to demonstrate the feasibility and efficiency of our improved techniques. Discussions and analysis are raised after the experiments.

Regarding future work, in addition to the techniques methods proposed in this paper, there are other methods that can be used for improving the efficiency of distributed SMT solving. The methods proposed in this paper are only for the client side. However, we can actually further improve the efficiency from the server side as well. In our current implementation, the number of requests from clients is four times higher than before, which may make the server get stuck. The server responds to the clients' requests in a serial way while parallel I/O can be used to give the server an ability to respond to various requests at the same time. Currently, the server randomly chooses files to send to the clients without considering the computation ability of different clients. Another possible idea is that the sever could use other optimized file choosing strategy, e.g., by the size of files, so as to avoid dispatch hard problems to weak clients. We will investigate those possibilities in the future.

References

[1] L. Liu, W. Kong, and A. Fukuda, "Implementation and Experiments of a Distributed SMT Solving Environment," International Journal on Computer Science and Engineering, vol. 6, 2014, pp. 80–90, ISSN: 0975-3397.

[2] E. M. Clarke, O. Grumberg, and D. Peled, Eds., Model Checking. The MIT Press, 1999, ISBN: 978-0-262-03270-4.

[3] G. J. Holzmann, Ed., The SPIN Model Checker: Primer and Reference Manual. ADDISON WESLEY Publishing Company Incorporated, 2003, ISBN: 978-0-321-77371-5.

[4] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L.-J. Hwang, "Symbolic model checking: 10^{20} states and beyond," Information and Computation, vol. 98, no. 2, 1992, pp. 142–170.

[5] A. Biere, A. Cimatti, E. M. Clarke, O. Strichman, and Y. Zhu, "Bounded Model Checking," Advances in computers, vol. 58, May 2003, pp. 117–148.

[6] A. Armando, J. Mantovani, and L. Platania, "Bounded model checking of software using SMT solvers instead of SAT solvers," International Journal on Software Tools for Technology Transfer, vol. 11, no. 1, Nov. 2008, pp. 69–83.

[7] N. Amla, R. Kurshan, K. L. McMillan, and R. Medel, "Experimental analysis of different techniques for bounded model checking," in Tools and Algorithms for the Construction and Analysis of Systems. Springer, 2003, pp. 34–48.

[8] L. de Moura and N. Bjørner, "Satisfiability modulo theories: An appetizer," in Formal Methods: Foundations and Applications. Springer, 2009, pp. 23–36.

[9] L. De Moura and N. Bjørner, "Satisfiability modulo theories: introduction and applications," Communications of the ACM, vol. 54, no. 9, 2011, pp. 69–77.

[10] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu, "Symbolic Model Checking without BDDs," in In Proc. of the Workshop on Tools and Algorithms for the Constrction and Analysis of Systems (TACAS'99). Springer Berlin Heidelberg, 1999, pp. 193–207.

[11] C. Barrett, R. Sebastiani, S. Seshia, and C. Tinelli, Handbook of Satisfiability. IOS Press, 2009, vol. 185, ch. 26, pp. 825–885.

[12] L. de Moura and N. Bjørner, "Z3: An efficient SMT solver," in Tools and Algorithms for the Construction and Analysis of Systems. Springer, 2008, pp. 337–340.

[13] "CVC4: the SMT Solver," 2014, URL: http://cvc4.cs.nyu.edu/web/ [accessed: 2014-01-18].

[14] A. Biere, Handbook of satisfiability. IOS Press, 2009, vol. 185.

[15] C. Barrett, R. Sebastiani, S. A. Seshia, and C. Tinelli, "Satisfiability modulo theories," in Handbook of Satisfiability. IOS Press, 2009, pp. 825–885.

[16] "The Message Passing Interface (MPI) Standard," 2014, URL: http://www.mcs.anl.gov/research/projects/mpi/ [accessed: 2014-01-02].

[17] "MPICH User's Guide (Version 3.0.4)," 2014, URL: http://www.mpich.org/static/downloads/3.0.4/mpich-3.0.4-userguide.pdf [accessed: 2014-01-02].

[18] "Open MPI: Open Source High Performance Computing," 2014, URL: http://openmp.org/wp/2013/12/tutorial-introduction-to-openmp/ [accessed: 2014-01-02].

[19] "SMT-LIB: The Satisfiability Modulo Theories Library," 2013, URL: http://www.smtlib.org/ [accessed: 2013-12-10].

[20] G. J. Holzmann and D. Bosnacki, "The Design of A Multi-core Extension of the SPIN Model Checker," IEEE Trans on Software Engineering, vol. 33, no. 10, 2007, pp. 659–674.

 ISBN: 978-1-61208-339-1