Implementation and Experiments of a Distributed SMT Solving Environment

Leyuan Liu*, Weiqiang Kong*, and Akira Fukuda
Graduate School of Information Science and Electrical Engineering
Kyushu University
Fukuoka, Japan
leyuan@f.ait.kyushu-u.ac.jp, weiqiang@qito.kyushu-u.ac.jp, fukuda@ait.kyushu-u.ac.jp

***Abstract*—SMT-based Bounded Model Checking (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 target of this paper is to accelerate SMT solving through distributed computation so as to enhance the performance of SMT-based BMC. To this end, we implement a multi-CPUs & multi-cores based distributed computation environment, to which different SMT solvers can be integrated. In this paper, we describe the implementation details and report our preliminary experimental results.**

Keywords-acceleration, distributed smt solving, mpi, openmp

I. INTRODUCTION

Bounded Model Checking (BMC) is a restricted form of model checking [1] 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 [2] and symbolic-based BMC such as BDD-based [3] SAT-based [4] or SMT-based [5] BMC. It has been reported in [6] that symbolic-based methods perform better than explicit-based methods for verifying general LTL (Linear Temporal Logic) [7] properties. Among the symbolic-based BMC methods, the 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. SMT draws on the most prolific problems in the past century of symbolic logic: the decision problem, completeness and incompleteness of logical theories, and finally complexity theory. The computational complexity of most SMT problems is very high [7], [8]. For all that, it is difficult to accelerate the SMT solving process for the engineers engaged in model checking. One of the most intuitive ways to accelerate SMT solving is through taking advantage of distributed computation and enjoying the power of multi-cores CPU, multi-CPUs, and/or even cloud computing. In this paper, we describe our work on making distributed SMT solving available through utilizing MPI [9] and OpenMP [10]. We describe a multi-CPUs & multi-cores based implementation to which different SMT solvers can be integrated. The preliminary experimental results on a large set of benchmarks demonstrate that SMT solving can be accelerated dramatically as expected.

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 the design and implementation of our proposed server-client style distributed computation environment. Section IV presents the experiments to evaluate the proposed method and discusses the results. Finally, Section V mentions possible extension (application scenarios) of our work and concludes the paper.

II. BACKGROUND

A. Bounded Model Checking

Bounded Model Checking technique was first proposed by Biere et al. in [11]. At the early days, BMC is based on SAT solving [12]. It is commonly acknowledged as a complementary technique to BDD-based symbolic model checking [4]. Recent years, with the development of modern efficient SMT solvers like Z3 [13] and CVC4 [14] 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 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. More formally, BMC can be defined as follows: given a finite state transition system *M* and a temporal property *P*, BMC is to analyze the state space of all possible transitions of *M* whose length is bounded by an integer *k* and to find an execution trace of *M* that violates *P*. A BMC problem in *M* with *P* is formulated in the seminal paper [11] like:

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

where I_0 represents the initial states of system *M*, T_i denotes the transition relation of *M* with step *i*, and $I_0 \wedge \bigwedge_{i=0}^{k-1} T_i$ represents all possible paths from an initial state of the system *M*. The negative form of *P* is a formula used to represent the situation that the property is violated. If there exits an assignment to all the variables used in formula (1) which makes formula (1) evaluate to true, then a counterexample is founded. Otherwise, the property is satisfied by the behaviors of *M* bounded by *k*. Although BMC can also be used for proving correctness (see [11] for more details), BMC is more often used for finding counterexamples rather than correctness proof. The negative form of property is used because the SAT solvers and SMT solvers tend to find the assignments that make the formula being evaluated true.

B. Satisfiability Modulo Theories

Satisfiability Modulo Theories is a research topic that concerns with the satisfiability of formulas with respect to some background theories [15]. 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 achieve significant improving on performance and capability, and thus it becomes 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 [8]. 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 [13]. In analogy with SAT, SMT procedures (whether they are decision procedures or not) are usually referred to as SMT solvers [16]. Actually, the SMT problem emerges in many fields, such as intelligence, formal verification for software and hardware, static program analysis, scheduling and planning.

C. MPI and MPICH

Message Passing Interface (MPI) [9] is a standardized and portable message-passing system. The MPI standard defines the syntax and semantics of a core of library routines useful to wide range of user writing portable message passing programs in different program languages. It is a language-independent communications protocol used to program parallel computers. We choose MPI 2.0 standard in our implementation.

There are several well-tested implementations of MPI, such as LAM/MPI [17], MPICH and Open MPI [10] etc. They are designed for high performance on both massively parallel machines and on workstation clusters. MPICH [18] is a high-performance and widely portable implementation of the MPI standard. It provides an MPI implementation that supports different communication and computation platforms including commodity clusters, high-speed networks and proprietary high-end computing systems. It also provides enable cutting-edge research in MPI through an easy-to-extend modular framework for other derived implementations [18]. We use MPICH2 in our implementation to realize control signals and files transmission among different PCs. In the remaining part of this paper, the MPI and MPICH are all referred to MPICH2.

D. OpenMP

OpenMP (Open Multi-Processing) [19] is an API that supports multi-platform shared memory multiprocessing programming in C, C++, and FORTRAN, on most processor architectures and operating systems, including Solaris, AIX, Mac OS X, and Windows platform [20], [21], [22], [23]. It is developed in 1997 and promoted and supported by a nonprofit organization named OpenMP Architecture Review Board (ARB). The members in ARB are AMD, Fujitsu, HP, IBM, Intel, NEC, and Oracle etc.

OpenMP is different from MPI. MPI is a multi-process mechanism but OpenMP is a multi-thread one. MPI supplies the communication ability to the processes even when the processes are in different PCs. The MPI task manager allocates independent memory to every MPI process. OpenMP achieves parallel by creating multiple threads. The created threads exist in one PC and sharing same memory space.

E. Z3

Z3 is a state-of-the art theorem prover from Microsoft Research [13], [24]. It integrates support for variety of theories and can be used to check the satisfiability of logic formulas over one or more theories. Z3 is a low-level tool so that it is best used as a component of other tools when logical formulas are required to be solved. Z3 supports linear real and integer arithmetic, fixed-size bit-vectors, extensional arrays, uninterpreted functions, and quantifiers. It supplies API for C, C++, .NET, Python etc. Though we choose Z3 as the core solver of our framework and experiments, actually it can be replaced by any other SMT solvers. Z3 is recommended as backend solvers by bounded model checking tools like CBMC [25]. In [26], a portfolio approach to deciding the satisfiability of SMT formulas is proposed, through which the parallel version of Z3 is implemented. In that approach, a sequential solver is parallelized by running multiple solvers and each of them is configured to use different heuristics. Lemmas are shared between different solvers periodically. The input is copied to every core so that there is no need for locking the formulas during solving, and the shared lemmas are the same. It is reported that the parallel Z3 outperforms the sequential Z3 on many benchmarks -- the parallel version beats sequential version by orders of magnitude.

It should be noted that our implementation is different from the Z3's parallel mode. The parallel mode of Z3 is used for solving one SMT file in a parallel way while our method is solving multiple SMT files distributed at the same time.

III. Implementation

A. Overview

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 Fig. 1. All clients are connected to a center server, data is transmitted between server and clients. The server responses the requirement of acquiring file 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 process will be finished until the server has no file to send.

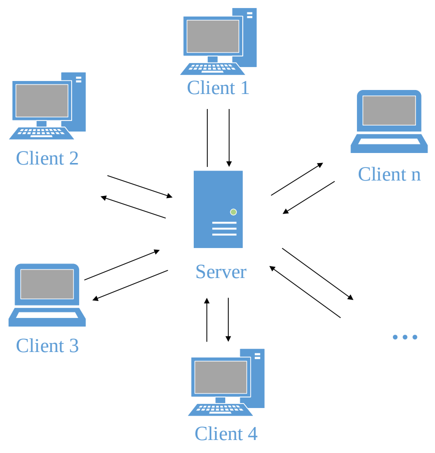

Fig. 1: Network Topology

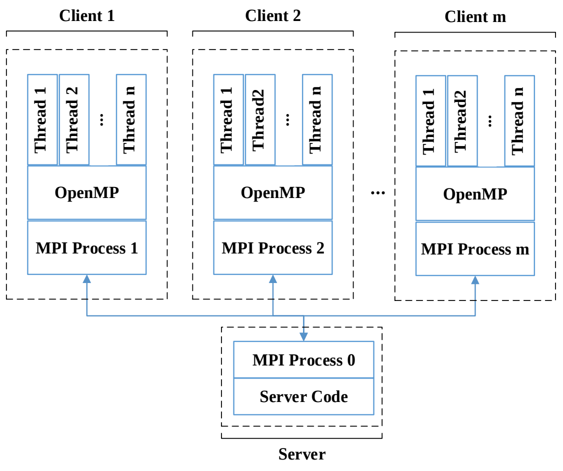

Fig. 2: Architecture of the Distributed Environment

In our implementation, both MPICH and OpenMP are used to gain the purpose of taking best advantage of clients' computing capacity. The architecture of the implementation is shown in Fig. 2. The MPI task managers which is running on different hosts, create severer and clients processes independently. They have the unique names in `int` type, which can be used to distinguish these processes, numbered from `0` to `n`. The MPI task managers set up every process on different hosts assigned by user deterministically from the server. One PC has only one MPICH process running on it. The server and clients need to communicate with each other when the server dispatches workloads to the clients. MPI supplies the message passing mechanism which allows us to send control signals and files among hosts in a network. At this point, we are ready to apply distributed computing. OpenMP is used to perform further parallel SMT solving. In a MPICH process, OpenMP is utilized to create multiple threads which communicate with each other using shared variables. It should be noted that, on each client, the OpenMP threads are not created at the beginning of the process. They are created after the file transmission finished, only for parallel SMT solving. The file receiving and dispatching are completed by MPICH processes. One client creates 4 threads by default running in parallel. Each thread invokes a Z3 SMT solver to solve specific SMT files dispatched to it. Generally speaking, we use MPICH to achieve distributed computing and communication, use OpenMP to achieve parallel SMT solving as well.

B. Server

The server takes charge of responding the requirement from clients and dispatching the SMT files to clients. If there exits enough files, they will be sent to the clients one by one. Every time four files are sent to one client by default. The default value can be changed by user as their wish. The server denoted by the `MPI process_0` in our implementation. The pseudocode of server is shown in Algorithm 1.

Algorithm 1. Pseudocode of the Server

```
Process_0(nTasks){
    Initialization and definition of variables;
    Timer(start);
    Terminate == 0;
    do{
        MPI_Recv (request_type from any client);
        if (require_type is files request){
            search_result=seek_file(process_id);
            if (search_result == no file found)
                MPI_Send( no file founded to process_id);
        }
        else
            terminate++;
    }while (terminate < (nTask-1));
    Timer(stop);
    compute time consuming;
    return 0;
}
```

The server is running in a `do while` loop until there is no `SMT` file to be transferred to the clients. It starts with checking and receiving messages from any clients in the entire MPI communication domain using MPI function `MPI_Recv()`. It is paired with a blocked message sending function `MPI_Send()` invoked by clients at the beginning. The request receiving followed by determining the `request_type` which represents the type the message. If it is a message of requiring `SMT` files from a client, the function `seek_file(process_id)` is invoked. This function explores the local directory of the server, try to find whether there exit `SMT` files can be sent to the client. If so, they will be sent to the client which require files. If not, the server will send a message to notify the client that the server does not have enough files. If the received message is not a file request message, the variable `terminate` will plus one. When terminate bigger than the number of the process, it means that all clients are terminated, the server (`process_0`) will stop loop and terminate. A timer, which is used to calculate the whole parallel solving procedure, is set at the beginning of the solving process and will stop after all processes finished their solving. The control flow graph is shown in Fig. 3.

Algorithm 2. Pseudocode of seek_file(process_id)

```
seek file (process_id) {
    Initialization and definition of variables;
    Construct the path of files to be sent;
    if (no file is founded)
        return 1;
    else{
        MPI_Send(file_founded, to Process_id);
        Acquire the files number: send_counter;
        MPI_Send(send_counter, to Process_id);
        do {
            if (not reach the send counter threshold)
                break;
            Construct the path_of_file;
            f_open(path_of_file);
            Acquire file_size;
            MPI_Send(file_name to Process_id);
            MPI_Send(file_size to Process_id);
            MPI_Send(file to Process_id);
            Delete the file has been sent;
        } while (exit another file to be sent); }
    return 0; }
```

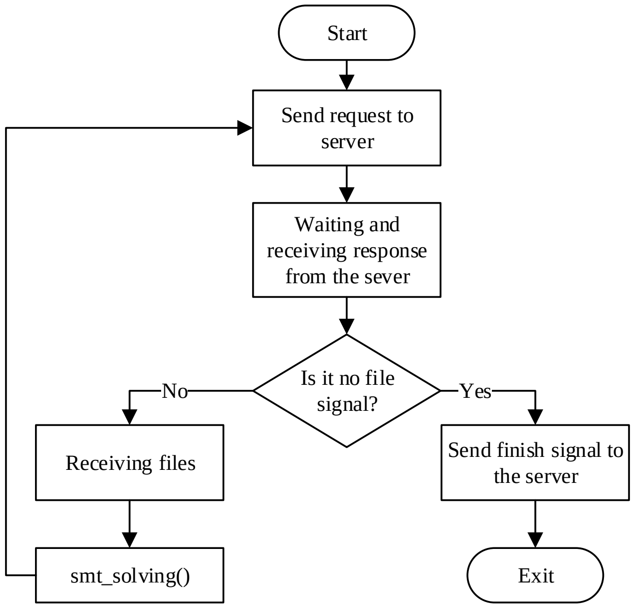

Fig. 3: Control Flow Graph of the Server

The file sending procedure takes place in the function `seek_file()`. The function searches and sends files to the target process (denoted by `process_id`). The pseudocode of `seek_file()` is shown in Algorithm 2. It starts from trying to find at least one file in the directory, if so, the function will inform the client that files are founded (Line 7). Then, it counts how many files can be transferred to the client, `send_counter` in Algorithm 2. Initially, we give the default value 4 to the `send_counter_threshold`. It is the upper bound of the number which counts the number of files sent to the client every time. Since there are more than one clients in the system and we cannot decided the quantity of `SMT` files, it is necessary to determine the number of files to be sent (sometimes it will be less than `send_counter_threshold`). If the number is less than the `send_counter_threshold`, the do while loop (Line 19) will ensure that the file sending procedure will finish within the threshold. Conversely, Line10-Line11 will guarantee that only threshold files will be sent to the client. Secondly, the quantity of files is sent to the client by `MPI_Send()` function from the server (Line 9). At every sending round (Line 10 - Line 20), the file name, file size and the file itself are sent separately in sequence. We use blocked MPI communication mechanism to send files. It means that after the server invokes a `MPI_Send()`, it suspends until the client received the message.

C. Client

The client is implemented by MPI process numbered from 1 to n depending on user requirements. The main process of the client is requiring and receiving `SMT` files to local machines, then invoking Z3 SMT solver to do the verification. After SMT solving, client will report to the server if a counterexample is found. At the beginning, the client require and receive four (or less than four) `SMT` files from server, if there exist such files on server. Then the client verifies these files. We call this process as a solving round. After a solving round is finished, the client will require new files until there is no file could be transferred. In a single solving run, we utilize OpenMP to accomplish parallel SMT solving on a PC to exploit the full computation resources. We suppose that the client in our implementation is equipped a CPU with four cores consequently we set 4 threads using OpenMP in each MPI process as default setting. The pseudocode of client n is shown in Algorithm 3.

The flow of a client is as follows: (1) the client sends file requirement to the server (Line 4), and then waits for the response of the server. If the server has no files to send, this client will receive a `no_file` signal and terminate, otherwise, the client will receive the number of files which will be sent later. (2) Line 10 to Line 17 are the file receiving procedure. The file name, size and the file itself are received respectively. The procedure is a while loop, and will be executed many loops which equal to the number of the files. The buffers to receive the data mentioned above will be clear after a file is received successful. (3) After the all the files are received successful, the client invokes the function `smt_solving()` to do the parallel verification SMT solving procedure. We will describe the parallel solving procedure in the following paragraph. (4) After the SMT solving procedure is finished, the client will check the saved solving results. If a counterexample is found, it will be sent to the server. (5) The client will send request to the server unless the server returns no file founded message to the client. (6) The client sends the `finish_signal` to the server in order to notice that it will be terminated soon. The whole control flow of the client is shown in Fig. 4.

Algorithm 3. Pseudocode of the Client

```
1.  Process_n () {
2.      Initialization and definition of variables;
3.      do {
4.          MPI_Send (request files, to process_0);
5.          MPI_Recv (file_existence_condition from server);
6.          if (no file is founded)
7.              break;
8.          MPI_Recv(file_num from server);
9.          j = 0;
10.         while (j < file_num) {
11.             MPI_Recv(file_name  from server);
12.             MPI_Recv (file_size  from server);
13.             MPI_Recv (file to  from server);
14.             fwrite (file to local HDD);
15.             rename(file);
16.             clear receiving buffers;
17.             j++; }
18.         invoke smt_solving();
19.         if (a counterexample is founded)
20.             report solving result to the server;
21.         clean local files;
22.     }while (there still exists files to be received)
23.     MPI_Send(finish_signal to the server);
24.     return 0;
25. }
```

The parallel SMT solving is accomplished by the function `smt_solving()` which utilize OpenMP to achieve parallel. The pseudocode is shown in Algorithm 4. OpenMP can create specified number of threads which can communicate with each other by sharing variables (the variable is named as signal in Algorithm 4). Line 2 sets the number of thread to be created. In consideration of all our PCs have quad-core CPU, in our implementation, we set the value to four to exploit all the compute resources of a client. The `smt_solving()` function will be invoked after the file receiving procedure. This function will create 4 threads, every thread uses C function `system()` to execute Z3 solving `SMT` files one by one. The verification results are written to four files separately on local hard disk driver.

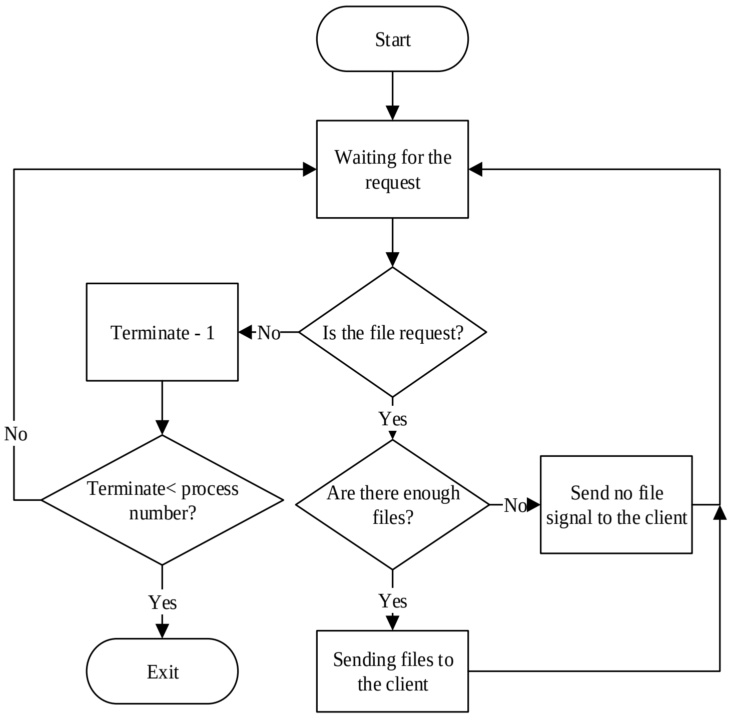

Fig. 4: Control Flow Graph of a Client

Algorithm 4. Pseudocode of the Function smt_solving()

```
1.  smt_solving () {
2.      omp_set_num_threads(m);
3.      omp_init_lock(&lock signal);
4.    #pragma omp parallel {
5.          switch (omp get threadnum()) {
6.          case 0: {
7.                Exploring all files belonging to thread_0 {
8.                  Construct command to invoke Z3;
9.                    system (command);
10.                 goto next file;
11.                 }
12.          }
13.             break;
14.         case 1: {
15.               Exploring all files belonging to thread_1 {
16.                 ... // same as case 01
17.               }
18.         }
19.             break;
20.          ... // case 3 and case 4 are omitted
21.         default: break;
22.         }
23.   }
24.     return 0;
25. }
```

D. Communication Protocol

We describe the communication protocol of the system in this section. In our implementation, there are two types of communication. The first type is controlling signal transmission between the server and clients. It happens in the beginning and terminate stages. The other type is `SMT` file transmission from the server to clients. There is no communication among different clients.

1) Control Signal Transmission: In our program the control signal is an `int` array (denoted by `power[n]`) with two elements. The first element `power[0]` used to indicate the signal's meaning. The value and its corresponding meaning are shown in TABLE. I. The second element is `power[1]` whose value is the source of the signal. The MPICH process ID is used to mark the source of the control signal. `power[0]` = 0 and `power[0]` = 1 can only be generated by a client. Furthermore, `power[0]` = 2 can only be dispatched by the server.

Table I: Meanings of the Signal power[0]

`power[0]`	Meaning
0	Request files from a client or server has files to send.
1	This client will be terminated.
2	There is no file on the server.

The distributed SMT solving system starts from a client sends the file request to the server, and the server search its local directory to determine whether there exist enough files. Then send the result as a control signal to request client. It can be `power[0]` = 0 (files exit) or `power[0]` = 2 (no file). It should be noted that the control signal has two `int` value so that the file request from the client is `power[]` = {0, `process_id`} (value scope of `process_id` is `1` to `n`) and the files exit signal is `power[]` = {0, 0}. If the client receives the `power[]` = {0, 0}, it will turn to file receiving process. Otherwise it will send `power[]` = {1, `process_id`} to server. The control signal protocol is shown in Fig. 5. The dash line of step 2 and step 3 will occur when the server has no files to send. The filled line denoted step 2 and the dash line denoted step 2 do not happen all at once. If the filled line step 2 happens, the client will turn to file receiving process immediately. After a client finishes the SMT solving process, it will request new files, if the server sends the signal in step 2 denoted by the dash line, the client will execute the step 3 to inform the server its termination.

2) File Transmission: The file transmission process is much easier. This process starts from the client receives the file exists signal `power[]` = {0, 0} sending by the server. The file transmission protocol is shown in Fig. 6. The MPI blocked communication mechanism is used, so we simplify the file receiving process. The confirmation after message receiving is eliminated. After step 2, the client knows that how many files will be sent. The sending and receiving processes both run in a loop. Step 3 to step 5, which is boxed by a dashed rectangle, will be repeated until all files have been sent. To make the 3 message sending and receiving in sequence, we use different message tags so that the client can distinguish them and receive them correctly. The file size is sent for establishing receiving buffer dynamically to save memory. The dashed line represented step 6 is an optional step. If a counterexample was found after the parallel SMT solving, the client will report the counterexample to the server. If not, the client will require new file from the server with entering control signal transmission process.

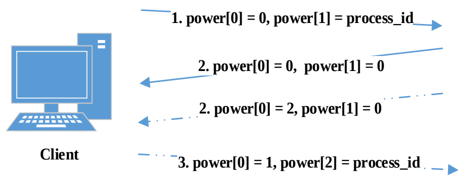
1. power[0] = 0, power[1] = process_id
2. power[0] = 0, power[1] = 0
2. power[0] = 2, power[1] = 0
3. power[0] = 1, power[2] = process_id
Client
Server

Fig. 5: Control Signal Transmission Protocol

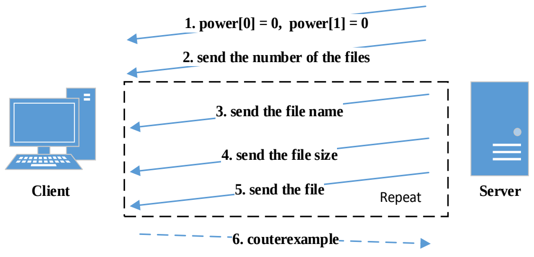

Fig. 6: File Transmission Protocol

IV. EXPERIMENT AND ANALYSIS

To evaluate the efficiency of our distributed SMT solving, we conduct a serial experiments on six groups of benchmarks downloaded from the Satisfiability Modulo Theories Library (SMT-LIB) [27] 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`. Due to space limitations, we will not go into details of those benchmarks here. Please refer to [27] 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 an 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 framework, we use our prototype to solve those benchmarks.

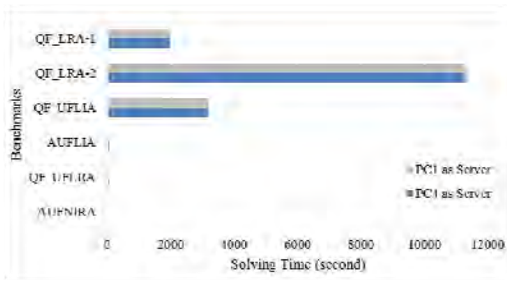

Fig. 7: Influence of the Server's Computation Ability on Solving

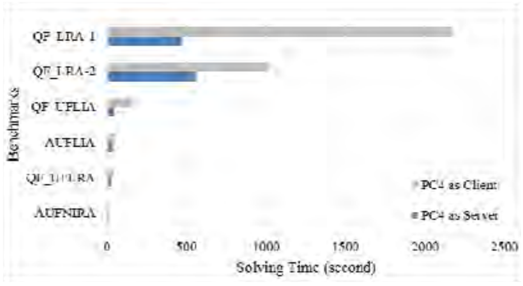

Fig. 8: Influence of the Client's Computation Ability on Solving

In our experiments `PC4` is designated as the server. Actually, the server in our implementation only responses the requests from different clients in a message queue sequentially and transfers `SMT` files. This work is I/O intensive rather than computation intensive. Otherwise, using `PC4` or other faster PCs as a server takes almost the same time consumption. This is proved by our following experiments shown in Fig. 7. Two experiments are conducted to investigate whether the server's computation ability will effect on the final solving results. We use `PC4` as a server and `PC1` as a client firstly, then we use `PC2` as a server, `PC1` as a clients. Six groups of benchmarks are verified in each experiment. In Fig. 7, the gray rectangular denotes the result when `PC4` used as the server. The blue rectangular represents using `PC4` as a client. It can be seen from the figure that, the two rectangular almost denote the same value although the compute ability of the `PC2` and `PC4` are considerable difference. Meanwhile, the target files are smaller than 1MB, the server does not need enormous computing ability.

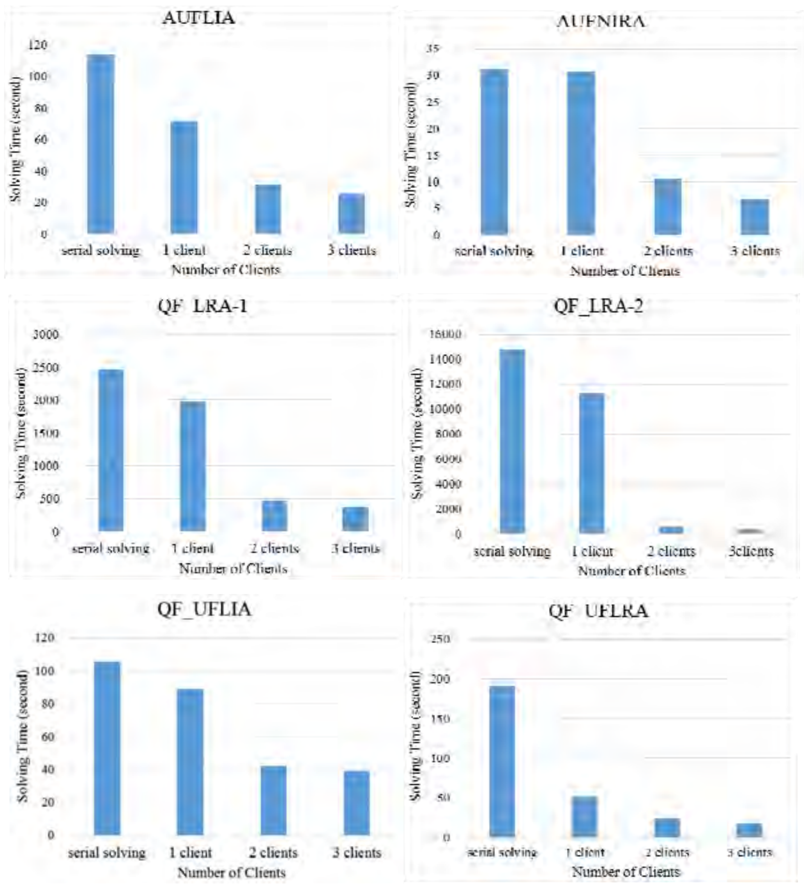

Fig. 9: The Distributed SMT Solving Results

On the other hand, compared with other PCs, using PC4 as a client will slow down the whole solving processing. The workloads (easy ones or complex ones) are dispatched to the client randomly. It is inevitable to assign a complex work to PC4. In that case, the solving process will take more time than using PC4 as a server. We conducted two experiments to preliminary inquire the effect that is taken by the different compute ability of the clients. In Fig. 8, the gray rectangular denotes the solving time when PC4 is used as one of the two clients. The blue rectangular represents the condition that PC4 is a server and PC1 and PC2 are clients. Also the six groups of benchmarks are used. The benchmarks ANFNIRA, QF_UFLRA and AUFLIA consist with a great deal of small size SMT files. In general, these files are easy to solve. Using PC4 as a client, it takes 13s, 35s and 54s to solve ANFNIRA, QF_UFLRA and AUFLIA. It takes 10s, 23s and 31s to solve them when we do not use PC4 as one of the clients. With solving QF_UFLIA, QF_LRA-1 and QF_LRA-2, the gaps are more apparent. When solving QF_LRA-1, using PC4 as one client takes nearly four time longer. Under the random task dispatch strategy, it is a much better alternative to using machines with similar hardware.

We use a single Z3 SMT solver to solve those six groups of benchmarks in a serial way separately. PC1 is used to carry out the serial solving experiments. All the distributed solving experiments use PC4 as the server, the other machines as clients. For every benchmarks, we repeat the solving in the following order: (1) serial solving; (2) one client; (3) two clients; (4) three clients. The results are shown in Fig. 9. Each sub-graph in Fig. 9 represents the 4 solving results on each benchmark. The vertical axis denotes the solving time in terms of seconds. The horizontal axis denotes four types of solving described above. They are serial solving, 1 client, 2 clients, and 3 clients in the order from left to right. Generally speaking, our preliminary distributed SMT solving prototype shows significant improvement on solving different benchmarks especially for some complex SMT problem, e.g. QF_LRA-1 and QF_LRA-2. Furthermore, the improvement seems less noticeable when solving the easier problems than the complex ones, e.g. QF_UFLIA and QF_UFLRA. When we increase the number of clients, the solving time is decreased dramatically before increased to 3 clients. Nonetheless, when we increase the client from 2 to 3, it gets small changes of solving time. The reason may be that when solving complex problems, the total solving time is close to the time solving the most critical single SMT file. Meanwhile, comparing the time delay brought by files and control signal, the time solving easy and small SMT files are small enough. We will investigate that in our future work.

V. Discussion and Conclusion

The point of our work on parallel SMT solving is to make SMT solving distributed and ultimately to accelerate SMT-based BMC. In general, two procedures are involved for this purpose: (1) decomposing a whole BMC problem (formula) into sub-problems (sub-formulas), each of which is relatively smaller in size, easier to solve, and can be solved in parallel. In our previous work, we have done work in this aspect [28], [29]; (2) paralleling SMT-based BMC, as described above in this paper. The classic SMT-based BMC encodes the system and properties into a single formula at specific bound. Restricted by the computation ability of a single PC, the efficiency of SMT-based BMC will decrease sharply. On the basis of decomposition, we can use the parallel SMT solving framework to solve the sub-problem parallel. Obviously, in this way, not only the capacity/scalability, but also the solving speed of model checking can be increased significantly. However, note that this is not to say our framework can increase BMC in all circumstances. Consider a situation where the SMT files in the server are all of small size. This means generally the files are easy to solve. Taking network latency into account, the time consumption here can be much higher than the SMT solving time. The final solving time of the parallel solving with multiple PCs may become longer than serial solving with a single PC.

Another issue we are going to discuss is about load balance. The server sends files to the client, and the client will arrange the work to different threads. For instance, there are four SMT files: $\{f_1, f_2, f_3, f_4\}$ and two threads T_1 and T_2. The solving time of the files f_1 to f_4 are 1s, 2s, 4s and 5s separately. If f_1 and f_2 are assigned to T_1 and the others are assigned to T_2, the solving time is 9s in total. But if f_1 and f_4 assigned to T_1 and the others are assigned to T_2, the solving time is 6s in total. The situation will be worse when the clients have different compute capacity. Considering the worst case that the most complex workload is dispatched to the lowest client, the total solving time depends on the lowest client's compute capacity. How to arrange the workload for different clients impacts directly on the final solving time. In our implementation, we dispatch the SMT files with a random strategy. We will investigate the load balance problem in our future work.

In this paper, we propose an environment to apply distributed SMT solving. The environment is solver independent with a C/S architecture. We use MPI and OpenMP to implement a prototype of the environment, and conduct a series of experiments to evaluate its effectiveness. The experimental results on six groups of benchmarks demonstrate that our environment can offer significant improvement on solving speed. Such an environment can be utilized to accelerate SMT-based bounded model checking for verification of large-scale software and/or hardware designs. We have been extending our previous work on verification of embedded software designs [30] to this proposed environment. The results will be reported in other opportunities.

References

[1] E. M. Clarke, O. Grumberg, and D. Peled, Model checking. The MIT press, 1999.
[2] G. J. Holzmann, "The SPIN model checker: primer and reference manual," 2003.
[3] 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, pp. 142–170, 1992.
[4] A. Biere, A. Cimatti, E. Clarke, O. Strichman, and Y. Zhu, "Bounded Model Checking," Advances in computers, vol. 58, pp. 117–148, May 2003.
[5] 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, pp. 69–83, Nov. 2008.
[6] 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.
[7] L. de Moura and N. Bjørner, "Satisfiability modulo theories: An appetizer," in Formal Methods: Foundations and Applications. Springer, 2009, pp. 23–36.
[8] L. De Moura and N. Bjørner, "Satisfiability modulo theories: introduction and applications," Communications of the ACM, vol. 54, no. 9, pp. 69–77, 2011.
[9] (2013) The message passing interface (mpi) standard. [Online]. Available: http://www.mcs.anl.gov/research/projects/mpi/
[10] (2013) Open mpi: Open source high performance computing. [Online]. Available: http://www.open-mpi.org/
[11] 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, 1999.
[12] C. Barrett, R. Sebastiani, S. Seshia, and C. Tinelli, Handbook of Satisfiability. IOS Press, 2009, vol. 185, ch. 26, pp. 825–885.
[13] 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.
[14] (2013) CVC4: the SMT solver. [Online]. Available: http://cvc4.cs.nyu.edu/web/
[15] A. Biere, Handbook of satisfiability. IOS Press, 2009, vol. 185.
[16] C. Barrett, R. Sebastiani, S. A. Seshia, and C. Tinelli, "Satisfiability modulo theories," in Handbook of Satisfiability. IOS Press, 2009, pp. 825–885.
[17] (2013) Lam/mpi parallel computing. [Online]. Available: http://www.lam-mpi.org/
[18] (2013) Mpich user's guide (version 3.0.4). [Online]. Available: http://www.mpich.org/static/downloads/3.0.4/mpich-3.0.4-userguide.pdf
[19] (2013) Tutorial: Introduction to openmp. [Online]. Available: http://openmp.org/wp/2013/12/tutorial-introduction-to-openmp/
[20] (2013) Openmp 4.0 specifications released. [Online]. Available: http://openmp.org/wp/2013/07/openmp-40/
[21] (2013) Openmp compilers. [Online]. Available: http://openmp.org/wp/openmp-compilers/
[22] A. Silberschatz, P. B. Galvin, and G. Gagne, Operating system concepts. J. Wiley & Sons, 2009.
[23] (2013) Openmp tutorial at supercomputing 2008. [Online]. Available: http://openmp.org/wp/2008/10/openmp-tutorial-atsupercomputing-2008/
[24] (2013) Z3 homepage. [Online]. Available: http://z3.codeplex.com

[25] L. Cordeiro, B. Fischer, and J. Marques-Silva, “SMT-Based Bounded Model Checking for Embedded ANSI-C Software,” IEEE Transactions on Software Engineering, vol. 38, no. 4, pp. 957–974, 2012.
[26] C. M. Wintersteiger, Y. Hamadi, and L. de Moura, “A concurrent portfolio approach to SMT solving,” in Computer Aided Verification. Springer, 2009, pp. 715–720.
[27] (2013) Smt-lib: The satisfiability modulo theories library. [Online]. Available: http://www.smtlib.org/
[28] W. Kong, L. Liu, T. Ando, H. Yatsu, K. Hisazumi, and A. Fukuda, Harnessing smt-based bounded model checking through stateless explicit-state exploration,” in 20th Asia-Pacific Software Engineering Conference (APSEC 2013). IEEE CS, 2013, pp. 355–362.
[29] W. Kong, L. Liu, Y. Yamagata, K. Taguchi, H. Ohsaki, and A. Fukuda, “On Accelerating SMT-based Bounded Model Checking of HSTM Designs.” in 2012 19th Asia-Pacific Software Engineering Conference. IEEE, Jun. 2012, pp. 614–623.
[30] W. Kong, T. Shiraishi, N. Katahira, M. Watanabe, T. Katayama, and A. Fukuda, “An smt-based approach to bounded model checking of designs in state transition matrix,” IEICE Transactions, vol. 94-D, no. 5, pp. 946–957, 2011.