Distributed Verification of Multi-threaded C++ Programs

Stefan Edelkamp [1,2] Shahid Jabbar [1,3] Damian Sulewski [1,4]

Department of Computer Science
University of Dortmund
Dortmund, Germany

Abstract

Verification of multi-threaded C++ programs poses three major challenges: the large number of states, states with huge sizes, and time intensive expansions of states. This paper presents our efforts in addressing these issues by combining an efficient use of hard disk with the distribution of the state space on several computing nodes. The approach is applicable to clusters and multi-core machines with single or multiple hard disks. We exploit the concept of a *signature* of a state that allows the full state vector to stay on secondary memory. For a distributed exploration of the state space, we report the lessons learned from using different partitioning schemes, including Holzmann and Bosnacki's [21] depth-slicing method, and their effects on blind and directed search algorithms.

Empirical evaluation is done on our experimental C++ verification tool StEAM, which is capable of detecting errors such as segmentation faults, deadlocks, access conflicts, etc. The distributed algorithms are realized through MPI on a cluster of workstations.

Key words: Program Verification, Distributed Model Checking, External Model Checking.

1 Introduction

Model checking [11] is a formal verification method for state based systems, and has been successfully applied in process engineering, hardware design and protocol verification. It traditionally terms the task of verifying a formal model with respect to its specification.

[1] This research is supported by German Research Council (DFG) under projects *directed model checking*.

[2] Email: stefan.edelkamp@cs.uni-dortmund.de
[3] Email: shahid.jabbar@cs.uni-dortmund.de
[4] Email: damian.sulewski@cs.uni-dortmund.de

This is a preliminary version. The final version will be published in Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

Program model checking has its roots in Abstract Interpretation [12] and Data-Flow Analysis [29]. Recent approaches involve the verification of software implementations (rather than checking a formal model); relying on the extension or implementation of architectures capable of interpreting machine code. These architectures include virtual machines [32] and debuggers [28]. Such un-abstracted software model checking does not suffer from the problems of the classical approach: neither the user is burdened with the task of building an error-prone model of the program, nor there is a need to develop a parser that translates the sources into the language of the model checker.

Even with refined exploration techniques, model checking is bounded by the main memory resources. Several memory-limited model checking algorithms have been developed, e.g., [16,20,24], but still memory is the core problem in dealing with large programs. Even with the advent of 64-bit machines, the physical limitation to actually use less than 64 GB of RAM has not disappeared. The use of virtual memory as a remedy to this problem can instead slow down the performance significantly.

Recent work on I/O-efficient model checking algorithms [4,13] minimizes the number of block accesses on disk and help to overcome the limitation in RAM. Extending the techniques as known from the internal world adds significant complications. Efficient external solutions of many problems often require invention of original, novel approaches radically different from those used to solve the same problems sequentially.

Parallel verification [5,30] itself is an emerging field. Several methods for parallel model checking have succeeded in making their way into industrial tools. Performance results on either parallel machines or on clusters of workstations show significant improvements with respect to sequential techniques in run-times along with adequate scalability in the number of processors and in the problem sizes.

Good parallel and I/O-efficient designs have much in common. This paper contributes a combination of I/O-efficient with parallel model checking in the context of the verification of real programs on the object-code level. The paper is structured as follows. First, we introduce the verification of C/C++ programs, and a program model checker that analyzes the object code. Next, we study extensions to the model checker to maintain states on disk together with collapse compression, where different parts of a state are saved separately [14]. We then explain our approach to distributed verification that integrates distributed search with the usage of secondary memory and collapse compression. A major problem in program verification is to deal with the states of huge sizes. Sending such states can create congestion over the network and consume a large amount of processing power of individual computing nodes. This problem is mitigated by exploiting a dual-channel communication framework. Empirical evaluations are presented next followed by a brief survey on the parallel and distributed verification approaches. Finally, we conclude and present some future research avenues.

2 Verification of C++ Programs

StEAM [5] is an experimental model checker for concurrent C++ programs [27]. Users of StEAM are first required to compile their C++ program with `igcc`. The `igcc` compiler is a variant of standard `gcc` compiler except that it translates a C++ source code into a specific object file format similar to the commonly used ELF (Executable and Linking File) format. This object code can be read and executed by the virtual machine and checked by StEAM. The compiler was originally intended to play games over Internet and is powerful enough to have compiled the source code of the famous arcade game DOOM.

In order to facilitate the model checking process, StEAM offers a number of special annotations within the source code that help a user in defining a model checking task. They include:

- BEGINATOMIC/ENDATOMIC to mark a code fragment as an indivisible *atomic* section.
- VASSERT to define an assertion expression.
- RANGE statement that non-deterministically branch on different value assignments for a variable.
- VLOCK/VUNLOCK to lock and unlock variables and resources.

The model checker offers different blind and directed search algorithms, including depth-first search, breadth-first search, best-first search, IDA*, and A* [26]. The types of safety errors addressed by the model checker include: deadlocks, segmentation faults, assertion violations, etc. The model checker branches the execution on threads (either derived from a base class or in form of POSIX `pthreads`) or on variable ranges. For search guidance, it offers a range of state-to-error estimates including the *active process* heuristics to accelerate the search for deadlocks.

A state in StEAM consists of registers, stack frames, global variables and memory pool. The state size grows with every memory allocation in the program and can easily reach to several megabytes. Hence, one of the most important challenges for program model checking on the object code level is the tremendous size of the state vector.

3 External Memory Usage in StEAM

Disk-based algorithms have been proposed for different model checking softwares such as [31] for Murϕ, [2] for Hopper, [13] for liveness checking in SPIN, [18] for NIPS, and most recently [4] for another approach for cycle detection. The general state-expanding algorithm we propose is based on the idea of *mini-states* first coined in [14]. Essentially, a mini-state is a pointer to a full system state residing on the hard disk or in the RAM. A mini-state consists of

[5] http://steam.cs.uni-dortmund.de

ExpandExtern(mini-state ms)
1: State $s \leftarrow$ *Read*(ms) from File
2: **for all** $s' \in$ Expand(s)
3: **for all** $ms' \in H[hash(s')]$
4: **if** (*Read*(ms')=s') **goto** 2
5: allocate mini-state ms''
6: $ms''.pred \leftarrow ms$
7: $ms''.t \leftarrow s.code(s')$
8: *Append* (s') to File

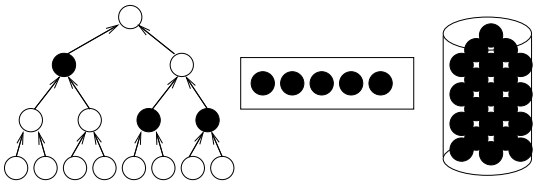

Fig. 1. Externalization of state in a search tree using a cache and an external state pool (right). Filled nodes correspond to states in the cache, while hollow nodes indicate mini-states without any representation in main memory. Pseudo-code (left). H is the hash table that keeps the mini-states but uses the hash function defined on the full state.

the hash value of its corresponding state, a pointer to the state – in the form of a file pointer – its predecessor information to reconstruct the solution path, and the transition used to generate the mini-state. Additional information include its depth and heuristic estimate to the target state that are used to order the states for expansion. All in all, a mini-state has a constant size in contrast to a state that can change its size due to dynamic memory allocation.

Recall that in general state-expanding algorithms, full states are accessed either to be expanded or for duplicate detection. Exploiting the idea of mini-states, we propose to perform the search on a tree skeleton defined on the mini-states, while actual states reside on the secondary memory. A request for expansion now reads the state from the disk based on the file pointer in the corresponding mini-state. Once read, the state is expanded and its children are again saved in the form of mini-states in the internal memory and as full states on the secondary memory.

Duplicate detection relies on a hash-table storing only the mini-states. The hash function, though, is defined on the full state. The advantage of external state representation is that we can restore each state on demand from disk, even if it is not in main memory.

The pseudo-code for external search is given in Figure 1 (left). For a mini-state ms, $ms.t$ denotes the transition (e.g., the sequence of machine instructions), which transformed the predecessor $ms.pred$ into ms. Similarly, for a full state s, $s.code(s')$ denotes the operation which transforms s to its successor state s'. Note, that transitions have a constant-sized representation, which is usually the program counter of a thread running in s. The function *Expand*(s) refers to the expansion of a full state s and results in a list of successors. The hash table H contains the mini-states representatives of all

previously generated states.

With the above-mentioned externalization approach, in the worst case, we perform one I/O operation for every access to a state. To lessen the average number of I/O operations, we associate an internal cache that allows to retrieve and store a small set of states from hard disk. Though this cache seems very much like virtual memory as offered by almost all operating systems, it can be configured to follow the best replacement strategy suited to the search algorithm. The cache principle is illustrated in Figure 1 (right). In all the experiments reported in this paper, we have used the Least-Recently-Used strategy for cache replacements.

We implemented separate caches, one for the data section, one for the binary section, one for the stack contents, and one for the rest of the system state. All of the components can be individually flushed to and read from disk. We will refer this method as *external collapse compression*. For the data and binary section, we incrementally check at construction time, whether a change has occurred; for the stack, we check for redundancies at insertion time. In all three cases, the cache is realized by using an AVL tree sorted by the individual hash values.

To be able to access the individual components of a state, a fourth FIFO cache is realized (not shown in the figure) that contains full-states where the actual contents of sub-parts are replaced by the hash values. If a new state is generated, we first check by a hash comparison if it is new. If a hash conflict is determined, the state is retrieved from the cache (or, if not present, from the hard disk). If the cache exceeds a certain predefined value, all elements that are not yet residing on disk are flushed in a single bulk I/O operation. For interested readers, we recommend [14] for a performance overview of external collapse compression.

4 Distributed Model Checking

Our approach to distributed search for verification of C++ programs integrates external search, collapse compression, and distributed search with state space partitioning. The former two techniques, as described in the previous section, are necessary to deal with the states of large sizes that can exhaust the RAM of a computing node in a matter of minutes. The third technique, distributed search, is used to mitigate the problem of time-expensive successor generation in program verification. To ease the discussion on state space partitioning, we use the term 'owner' (of a state) to identify the node to whom the state belongs to. We assume a disjoint partitioning that leaves no ambiguity in defining the 'owner' of a state. Figure 2 depicts a high-level description of our solution. Each computing node, denoted by P1 and P2 in the figure, contains an internal memory area that holds a hash table for the mini-states and separate caches for the *Stack*, the *Memory Pool*, and the *Data Section* of states. Each of the cache is associated to a file on the local hard disk (to

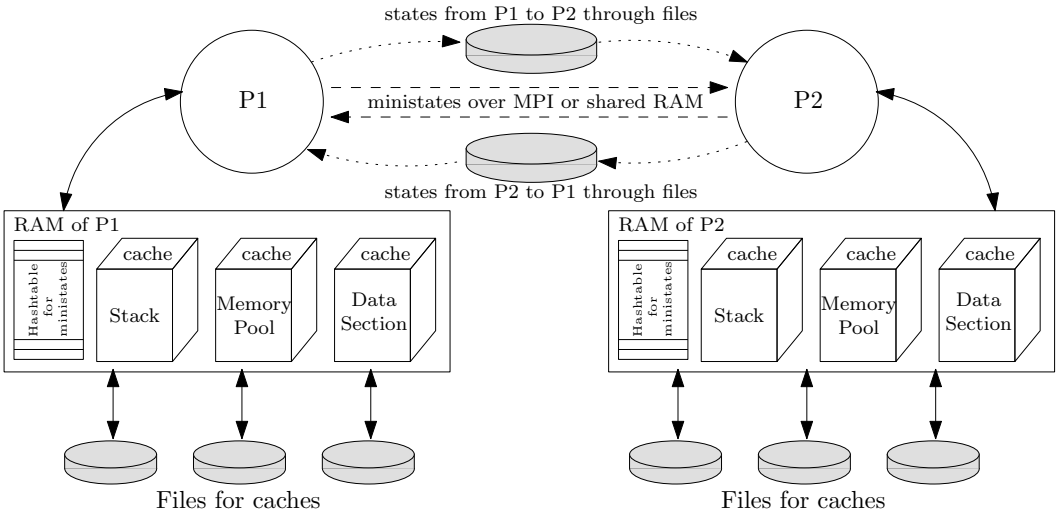

Fig. 2. Dual-channel communication: the mini-states are sent over MPI, while the full states are written to the hard disk. Solid arrows show local memory accesses, dotted arrows show the Network File System traffic, and dashed arrows show the communication over MPI.

avoid network traffic) or to a network disk. The caches along with the files forms the basis for external collapse compression. The mini-state contains the location of the full state on the disk along with the depth, the heuristic value, the hash value, the process ID of the node that generated it, the predecessor pointer on the generator node (for solution reconstruction), and a place holder for special messages.

Unlike other approaches, where a complete state is sent to the other nodes through MPI, we employ a dual-channel communication that involves exploiting both the shared file system (NFS) and the MPI messages. If a newly generated state does not belong to the node that generated it, we first write the full state to the disk and then send only the the mini-state through MPI to the actual owner of the state. The *rationale* behind this is: **(a)** the full state will only be used when it has to be expanded, and hence we can save the time consumed by the MPI thread to receive the state and internal processing for collapse compression; **(b)** for arranging the priority queue *Open*, the only information required is the depth and/or the heuristic value of the state – which is contained in the mini-state.

4.1 The Algorithm

Algorithm 1 depicts the distributed search algorithm. Each process is identified by its process identification number *PID* automatically generated by the MPI. The root (ROOT) triggers the search by inserting the initial state into the priority queue *Open*. It also initializes the set of idle nodes [Lines 1–2]. The endless while loop is split into two parts: handling of messages

Algorithm 1 Distributed search with dual-channel communication.

Input: An initial state: $\mathcal{I}$; Process ID: *PID*; Max messages to deque: MAX_MSG
Output: A reachable error state, if one exists; **true** otherwise
Variables: Priority queue *Open*; Set *idle_nodes*; Bool *idle_reported*; mini-state *ms*

```
1: if PID = ROOT then
2:    idle_nodes ← ∅; insert(Open, I) {Insert the initial state I in Open }
3: else
4:    idle_reported ← true
5: while true do
6:    if MPI.queue ≠ ∅ then
7:       counter ← 0
8:       for all mini-state ms ∈ MPI.queue do
9:          if ms.msg = TERMINATE then
10:            return true
11:         else if ms.msg = IDLE then {A node is idle}
12:            idle_nodes ← idle_nodes ∪ ms.creator
13:         else if ms.msg = BUSY then {A node that was idle is busy now}
14:            idle_nodes ← idle_nodes \ ms.creator
15:         else
16:            insert(Open, ms) {Insert the mini-state in the work queue}
17:         if (PID ≠ ROOT) and (idle_reported =true) then
18:            send (ms.msg ← BUSY) to ROOT {Inform root node is busy}
19:            idle_reported ← false
20:         if counter++ > MAX_MSG break {Break the for loop}
21:      end for
22:   if Open = ∅ then
23:      if (PID = ROOT) and (all nodes are in idle_nodes) then
24:         broadcast (ms.msg ← TERMINATE) to all nodes
25:         return true
26:      else if (PID ≠ ROOT) and (idle_reported = false) then
27:         send (ms.msg ← IDLE) to ROOT
28:         idle_reported = true{Avoid repeated IDLE messages}
29:   else
30:      if (State s ← getNext(Open)) = NULL continue
31:      for all s′ ∈ Expand(s) do
32:         if Errorstate(s′) = true then
33:            broadcast (ms.msg ← TERMINATE) to all nodes
34:            return the error state s′
35:         new_owner ← checkOwner(s′)
36:         if new_owner ≠ PID then
37:            Append s′ to File⟨PID, new_owner⟩ {Transfer through file}
38:            send (s′.mini-state ) to new_owner process
39:         else if checkForDuplicate(s′) = false then
40:            ms.state ← collapseCompress(s′)
41:            insert(Open, s′.mini-state )
42:      end for
43: end while
```

Algorithm 2 getNext: Get next State from *Open* to expand.

```
1: mini-state ms ← getMin(Open)
2: if ms = NULL return NULL
3: if ms.creator ≠ PID then
4:    State s ← Read(ms) from File⟨ms.creator, PID⟩ {Read the state pointed
      by the mini-state from the transfer file}
5:    if checkForDuplicate(s) = true then
6:       return getNext(Open) {Call recursively until a state is found.}
7:    else
8:       ms.state ←collapseCompress(s)
9:       return s
10: else
11:    s ← collapseDecompress(ms.state)
12: return s
```

[Lines 6–28] and expansion [Lines 29–42]. For the sake of clarity, we explain the second part first that deals with the expansion and the distribution of the states. A state is selected from the priority queue through the *getNext* function shown in Algorithm 2. It is a recursive function that selects the best mini-state from *Open*. If the mini-state is not generated by the node, the full state vector is retrieved from the disk and checked for duplicates. In case it has been visited earlier, the function is called recursively. If the state has *not* been visited earlier, it is *collapse compressed* for future references and is returned to the caller. If the state was generated by the process itself, it is *collapse decompressed* and returned to the expand function [Line 31].

Iterating on the successors s', we first check for the safety errors. Upon success we broadcast the termination message to all the nodes and report the encountered error state [Lines 33–34]. The ownership of the newly generated state is queried through the partition function [Line 35] and if the owner does not match the process ID, the successor s' is appended to the appropriate transfer file. File names are composed of two parts: PID of the sender and PID of the receiver; such a scheme avoids any concurrent writes by two different processors. Once the state is flushed, the mini-state representation of s' is sent to the owner via MPI. With this order, it is not possible that the receiver tries to read a state that is not completely written. If the successor's owner matches the current process, the state is checked if it is already visited [Line 39]. In case it is not, the full state is *collapse compressed* (and put into the caches) and its mini-state representation is inserted in the search frontier, *Open*.

The MPI messages are handled in the first part of the algorithm. We distinguish between three types of messages tagged to the mini-state (*ms.msg*):

- TERMINATE: one process has found the error state or the root has detected that all nodes are idle implying that the system is error free.
- IDLE: a process informs the root that its local *Open* queue is empty and it is waiting for work.

- BUSY: a process informs the root that it was idle earlier and has just received a state for expansion.

In order to avoid an infinite behavior while reading the MPI queue [Line 8], we set a limit on the number of mini-states that are extracted in one scan. The limit is set in the `MAX_MSG` input parameter that is compared against the variable `counter` [Line 20]. Similarly, the flag `idle_reported` is used to avoid repeatedly sending the IDLE messages to the root.

All the IDLE messages received by the root are kept in the set `idle_nodes`. If the *Open* queue of the root is empty and all other processes have also reported idle, the root broadcasts a terminate message to all the nodes [Line 24–25] and terminates by returning **true**. Note that, due to network latency the algorithm is not complete. There can still be messages in the channels between the clients at the time when the root decides that all are idle.

4.2 Work Load Distribution

An important step in any distribution paradigm is the choice of a partition function to evenly distribute the search space over the computing nodes. We have experimented with different partition functions.

We tried a hash-based distribution as proposed by Stern and Dill [30] with a linear hash function defined on the full state vector. Such a distribution is effective if, with high probability, the successors of a state expanded at a particular computing node are also mapped to the same node. This results in low communication overhead. In our case, with states of huge sizes, such a partitioning can be costly as computing the hash function is expensive. As one solution to the problem, StEAM offers the option for incremental hashing that relies on the hash difference between the state and its successor only [27].

Another approach we experimented with is partial hashing, based on [25], that truncates the state vector before computing the hash function and restricts to only those parts that are changed least frequently. This function was first proposed in the context of automata-based model checking in SPIN. It considers the states of a single process automaton to decide the ownership of a newly generated state. (An alternative was proposed in [26] for liveness properties where the partitioning was defined on the strongly-connected components of the *never-claim*, i.e., the Büchi automaton of the negated LTL property.) In our setting, we have the flexibility of using the partial function defined on the memory pool or the concatenation of all local stacks.

A recent partitioning method is proposed by Holzmann and Bosnacki [21], a technique that we term here as *depth-slicing*. The rationale behind it is that if a hash-based partition function is used, there is a high probability that the successor states will not be handled by the node that generated them, resulting in a high network overhead. Roughly speaking, the new method horizontally slices the depth-first search tree and assigns each slice to a different node. The situation is depicted to the left of Figure 3.

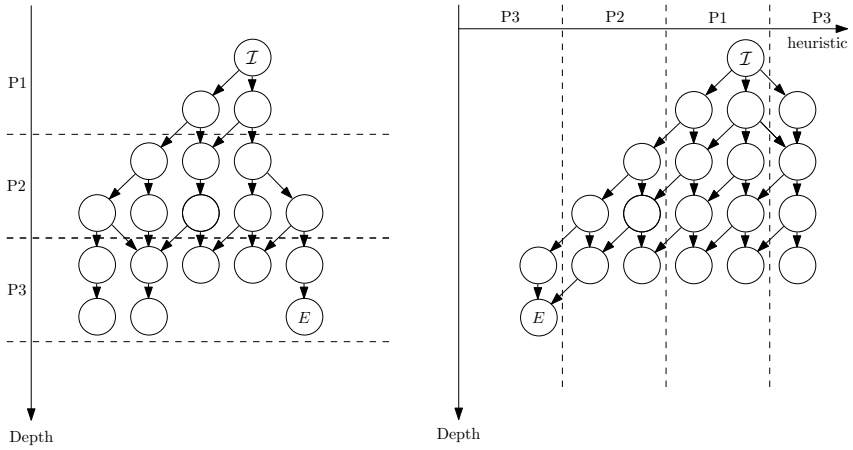

Fig. 3. State space partitioning with depth-slicing (left) and on heuristic values (right). Pi's denote the computing nodes. $\mathcal{I}$ is the initial state and the error state is shown with E.

Walking on the same footsteps, we propose a *vertical* partitioning of the search tree based on the heuristic values. The motivation for load balancing is that the expected distance to the error (used here) is a similar measure compared to the distance to the start state (used in the method above). The core advantage is that it is not only suited to depth-first but any general state expanding strategy including greedy best-first search, A* and breadth-first search. A visualization is given to the right of Figure 3. We do not perform any kind of dynamic load balancing as we do not have any dedicated thread that performs any kind of synchronization.

5 Experiments

We implemented external exploration on top of our tool StEAM. The distributed exploration is realized through MPI. The experiments are performed on a ClusterVision cluster of workstations. There are a total of 224 computing nodes with a total of 464 processors running OpenSuSe 10.0. We used the set of nodes consisting of two AMD Opteron DP 250 (2.4 GHz) processors each, connected by infiniband. Maximum number of parallel processes was 32.

In Figure 4, we show the scaling behavior of our approach on an instance of dining philosophers consisting of 200 philosophers. We employed depth-first search with depth-slicing partitioning by Holzmann and Bosnacki. While solving it on 1 and 2 nodes, the program exceeded the 8 hours time limits set on the cluster. The distribution scaled almost linearly for 4 to 12 nodes. Unfortunately, due to extreme usage at the cluster (around 99%) we were not able to run the experiments for a larger number of nodes.

For the 200 dining philosophers problem, each state was 32KB long. 4 nodes solved the problem with 90GB of external memory consumption in transfer files. A total of 2,256,037 states were generated till a deadlock was

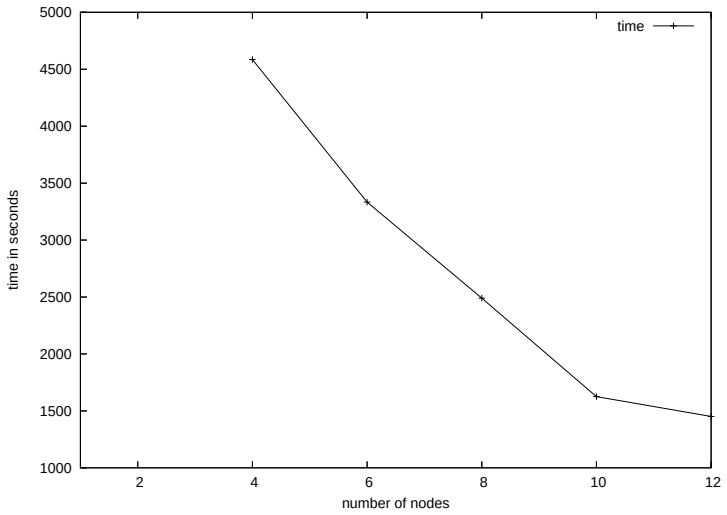

Fig. 4. Times for finding the deadlock in 200 Dining Philosophers with increasing number of computing nodes. Parallel depth-first search with depth-slicing is used. For 1 and 2 nodes the program exceeded 8 hours time bound on the cluster queue.

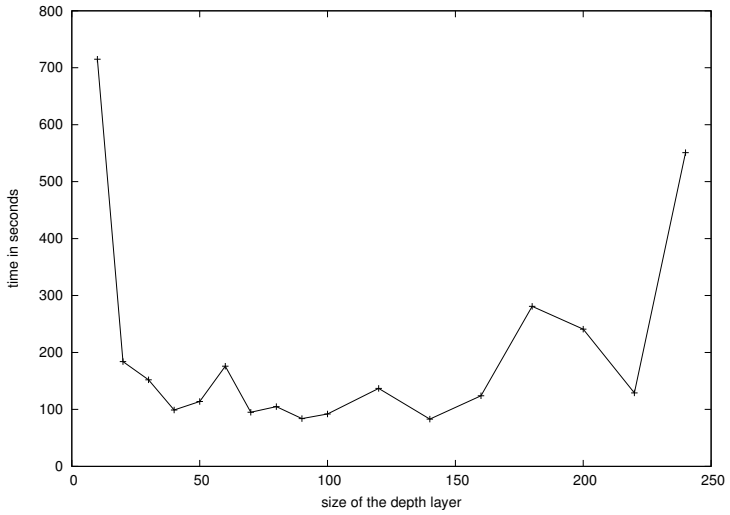

Fig. 5. Effect of slicing depths in Holzmann and Bosnacki algorithm in distributed search for 50 philosophers. Averaged over 6 runs for each depth slice. 2 computing nodes are used with shared NFS.

found. The states were almost uniformly distributed over the 4 nodes. That implies that for 0.51 million states per node, a total of 16GB is needed in the RAM. But due to externalization and collapse compression each node consumed a maximum of 1.5GB of RAM including the 500MB MPI overhead.

Figure 5 shows the behavior of parallel depth-first search with depth-slicing for varying depth sizes. For 50 philosophers, the results are averaged over 6 runs each using 2 nodes. The minima of the "bath-tub" lies at about using 100 layers per partition. The results confirm similar findings by [21].

We have also solved the dinning philosophers instance for **600 philoso-**

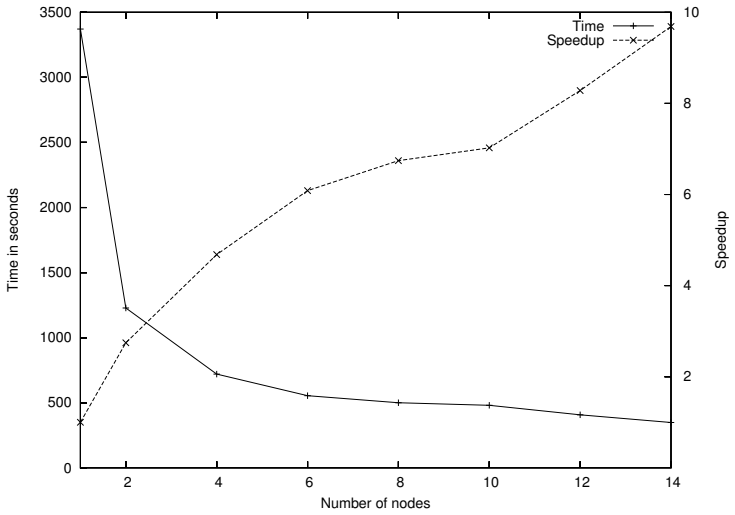

Fig. 6. Absolute times and relative speed-up on a random 15-Puzzle instances. Parallel breadth-first search with partial hash partitioning based on the memory pool is used.

phers with a state size of **97KB**. For 6 computing nodes it took 74 minutes, while consuming 60 GB of hard disk space for transfer files and generating a total of 761K states. Parallel depth-first search with a depth-slicing of 100 layers was used. The deadlock was found at layer 2193. For a smaller number of computing nodes, the time exceeded the bounds on the cluster queue.

The next set of experiments were performed on a n^2-1 sliding tiles puzzle instance taken from AI domain. The problem exemplifies the use of the RANGE statements for the choice of the move operators and contains no threads. The end condition is specified as an assertion violation where all the tiles are at their target location. The graph in Figure 6 shows the scaling behavior on a random $n = 4$ (15-puzzle) instance. The search algorithm used is parallel breadth-first search realized by always selecting the node with the minimum depth value from the *Open* list. The partitioning function was defined only on the memory pool section of the state, since it changed least frequently.

For the last set of experiments, in Figure 7, we took a *verification* problem on an error-free implementation of the Bakery protocol. The concurrent objects, consumers, are modeled by POSIX `pthreads`. We used parallel breadth-first search with partial hashing on the memory pool. The experiments are performed on a 4-processor shared memory architecture, still using the MPI and the local hard disk for communications.

6 Related Work

Conventional model checking techniques have high memory requirements and are computationally hard; they are thus unsuitable for handling real-world systems that exhibit complex behaviors which cannot be captured by simple

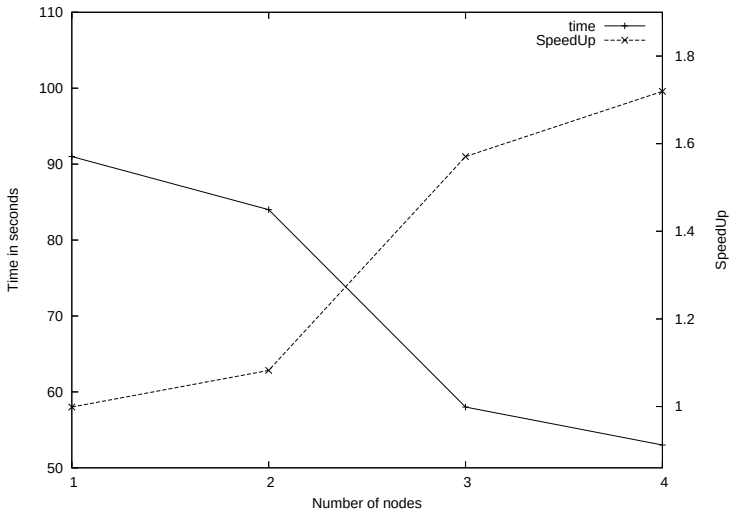

Fig. 7. Bakery with 3 consumers with parallel breadth-first search and using partial hash partitioning based on the concatenation of all local stacks. Experiments are performed on a 4-processor machine with local disk.

models having a small or regular state space. Various authors have proposed ways of solving this problem by distributing the memory requirements over a cluster of workstations. Perhaps one of the first efforts are reported in [1]. Stern and Dill [30] employed a hash-based partitioning scheme to divide the whole state space into multiple computing nodes. The proposed approach was implemented on top of the Murϕ verifier. Lerda and Sisto [25] experimented with a different partition function based on the states of only one process automaton. The rational behind such a function is that a transition usually performs only few local changes in a system. Therefore, with a high probability the successor state might also belong to the current node. Haverkort et al. [19] introduced distributed search for stochastic Petri nets. Distributed verification in μ-calculus is reported by [9] and for CTL* by [22]. There are attempts to also consider symbolic techniques, real-time [7] and SAT-solving [15] in a distributed fashion. Distribution based on partitioning the property automaton is contributed by [26], while [23] extended the external memory search to a distributed search algorithm by exploiting the set-based expansions. Blom et al. [8] proposed a related external collapse compression method for large states utilizing replicated databases instead of the dual-channel communication. A recent approach for distributed model checking based on BDDs is reported by [17].

A wide body of important results on distributed verification for both safety and liveness is contributed by the Paradise lab mostly implemented in Divine environment [6]. A distributed cycle detection algorithm for LTL model checking based on parallel breadth-first search is reported in [3]. Another algorithm by the same group is an extension of 'OWCTY' algorithm for distributed setting [10]. A recent extension contributes an external memory variant of the same algorithm [4].

Recently, with the advent of multi-core machines, the trend is directed towards verification on multi-core machines. Multi-core machines offer the advantage of having negligible overhead for state transfers due to shared memory. Holzmann and Bosnacki [21] present a method for multi-core extension of Spin where the safety analysis is applicable to N-core systems but the fair cycle detection to verify liveness properties is limited to only dual-core.

Unfortunately, in the domain of program model checking, to the authors' best knowledge, no efforts are made to migrate to distributed search.

7 Conclusion

With this work, we have contributed an integrated design for distributed and large scale verification of C++ programs. As the analysis is on the object code no abstraction takes place and the expressivity of concurrent C++ is preserved. The novelty and the algorithmic challenge lies in tackling the states of large sizes. We employed a dual-channel communication that combines MPI and NFS media. It avoids sending the whole state over MPI but a signature of it. The full state vector is flushed to disk in transfer files.

The algorithms we propose support blind and directed parallel model checking for safety properties including parallel depth-first, breadth-first, and best-first search. A new partitioning function based on the estimate value has been introduced that fits well to general state expanding search.

The experimental results are promising. We observe an almost linear speed up in all examples. Future work includes the integration of dynamic load balancing and the evaluation of larger C++ models. Even though we could report the full exploration of sample instances, for infinite state systems, the algorithm can run forever. In future, we also plan to accelerate the I/O operations by a more efficient block flushing of transfer states or using databases [8].

StEAM has been integrated into Eclipse as a plug-in. The error-trail returned can be traced in the original program interactively. We are currently experimenting with data abstraction and slicing to reduce the state space size.

References

[1] S. Aggarwal, R. Alonso, and C. Courcoubetis. Distributed reachability analysis for protocol verification environments. In *Discrete Event Systems: Models and Application*, volume 103 of *Lecture Notes in Control and Information Sciences*, 1987.

[2] T. Bao and M. Jones. Time-efficient model checking with magnetic disks. In *TACAS*, volume 3440 of *LNCS*, pages 526–540. Springer, 2005.

[3] J. Barnat, L. Brim, and J. Chaloupka. Parallel breadth-first search LTL model-checking. In *IEEE International Conference on Automated Software Engineering*, pages 106–115. IEEE Computer Society, 2003.

[4] J. Barnat, L. Brim, and P. Simecek. I/O-efficient accepting cycle detection. In *CAV*, LNCS. Springer, 2007. To appear.

[5] J. Barnat, L. Brim, and I. Černá. Cluster-Based LTL Model Checking of Large Systems. In *Formal Methods for Components and Objects*, volume 4111 of *LNCS*, pages 259–279, 2005.

[6] J. Barnat, L. Brim, I. Černá, P. Moravec, P. Ročkai, and P. Šimeček. Divine – a tool for distributed verification (tool paper). In *CAV*, volume 4144 of *LNCS*, pages 278–281. Springer, 2006.

[7] G. Behrmann, T. S. Hune, and F. W. Vaandrager. Distributed Timed Model Checking – How the Search Order Matters. In *CAV*, volume 1855 of *LNCS*, pages 216–231. Springer, 2000.

[8] S. Blom, B. Lisser, J. van de Pol, and M. Weber. A database approach to distributed state space generation. In *Parallel and Distributed Methods in Verification*, 2007. This volume.

[9] B. Bollig, M. Leucker, and M. Weber. Parallel model checking for the alternation free μ-calculus. In *TACAS*, volume 2031 of *LNCS*, pages 543–558. Springer, 2001.

[10] I. Černá and R. Pelánek. Distributed explicit fair cycle detection (set based approach). In *SPIN*, volume 2648 of *LNCS*, pages 49 – 73. Springer, 2003.

[11] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, 1999.

[12] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *ACM Symposium on Principles of Programming Languages*, pages 238–252. ACM Press, 1977.

[13] S. Edelkamp and S. Jabbar. Large-scale directed model checking LTL. In *SPIN*, volume 3925 of *LNCS*, pages 1–18. Springer, 2006.

[14] S. Edelkamp, S. Jabbar, D. Midzic, D. Rikowski, and D. Sulewski. External program model checking. In *In KI'06 (German Conference on AI) Workshop on New Results in Planning, Scheduling and Design (PuK'06)*, 2006.

[15] H. Garavel, R. Mateescu, and I. Smarandache. Parallel State Space Construction for Model-Checking. In *SPIN*, volume 2057 of *LNCS*, pages 216–234. Springer, 2001.

[16] P. Godefroid. Software model checking: The VeriSoft approach. *Formal Methods in System Design*, 26(2):77–101, 2005.

[17] O. Grumberg, T. Heyman, and A. Schuster. A work-efficient distributed algorithm for reachability analysis. *Formal Methods in System Design*, 29(2):157–175, 2006.

[18] M. Hammer and M. Weber. "To store or not to store" reloaded: Reclaiming memory on demand. In *Formal Methods: Applications and Technology*, volume 4346 of *LNCS*, pages 51–66. Springer, 2006.

[19] B. R. Haverkort, A. Bell, and H. C. Bohnenkamp. On the efficient sequential and distributed generation of very large Markov chains from stochastic Petri nets. In *Workshop on Petri Net and Performance Models*, pages 12–21. IEEE Computer Society Press, 1999.

[20] G. Holzmann. *The SPIN model checker: primer and reference manual*. Addison-Wesley, 2004.

[21] G. Holzmann and D. Bosnacki. The design of a multi-core extension of the Spin Model Checker, 2006. Invited talk at FMCAD.

[22] C. Inggs and H. Barringer. CTL* Model Checking on a Shared Memory Architecture. *Formal Methods in System Design*, 29(2):135–155, 2006.

[23] S. Jabbar and S. Edelkamp. Parallel external directed model checking with linear I/O. In *Verification, Model Checking, and Abstract Interpretation*, volume 3855 of *LNCS*, pages 237–251. Springer, 2006.

[24] C. Jard and Th. Jeron. Bounded-memory algorithms for verification on-the-fly. In *CAV*, volume 575 of *LNCS*, pages 192–202. Springer, 1991.

[25] F. Lerda and R. Sisto. Distributed-memory model checking with SPIN. In *SPIN*, volume 1680 of *LNCS*, pages 22–39. Springer, 1999.

[26] A. Lluch-Lafuente. *Heuristic search in the verification of communication protocols*. PhD thesis, University of Freiburg, Germany, 2003.

[27] T. Mehler. *Challenges and applications of assembly-Level software model checking*. PhD thesis, University of Dortmund, Germany, 2006.

[28] E. Mercer and M. Jones. Model checking machine code with the GNU debugger. In *SPIN*, volume 3639 of *LNCS*, pages 251–265. Springer, 2005.

[29] B. Steffen. Generating data flow analysis algorithms from modal specifications. *Science of Computer Programming*, 21(2):115 – 139, 1993.

[30] U. Stern and D. Dill. Parallelizing the murφ verifier. In *CAV*, volume 1254 of *LNCS*, pages 256–267. Springer, 1997.

[31] U. Stern and D. Dill. Using magnetic disk instead of main memory in the murphi verifier. In *CAV*, volume 1427 of *LNCS*, pages 172–183. Springer, 1998.

[32] W. Visser, K. Havelund, G. Brat, and S. Park. Model checking programs. In *ICSE*, pages 3–12. IEEE Press, 2000.