Software Tools for Technology Transfer manuscript No.
(will be inserted by the editor)

Parallel *and* Distributed Model Checking in Eddy[⋆]

I. Melatti, R. Palmer, G. Sawaya, Y. Yang, R. M. Kirby, G. Gopalakrishnan

School of Computing, University of Utah

Received: date / Revised version: date

Abstract. Model checking of safety properties can be scaled up by pooling the CPU and memory resources of multiple computers. As compute clusters containing 100s of nodes, with each node realized using multi-core (e.g., 2) CPUs will be widespread, a model checker based on the *parallel* (shared memory) and *distributed* (message passing) paradigms will more efficiently use the hardware resources. Such a model checker can be designed by having each node employ *two* shared memory threads that run on the (typically) two CPUs of a node, with one thread responsible for state generation, and the other for efficient communication, including (i) performing overlapped asynchronous message passing, and (ii) aggregating the states to be sent into larger chunks in order to improve communication network utilization. We present the design details of such a novel model checking architecture called *Eddy*. We describe the design rationale, details of how the threads interact and yield control, exchange messages, as well as detect termination. We have realized an instance of this architecture for the Murphi modeling language. Called Eddy_Murphi, we report its performance over the number of nodes as well as communication parameters such as those controlling state aggregation. Nearly linear reduction of compute time with increasing number of nodes is observed. Our thread task partition is done in such a way that it is modular, easy to port across different modeling languages, and easy to tune across a variety of platforms.

1 Introduction

This paper studies the following question:

> Given that shared memory programming will be supported by multicore chips (multi-CPU shared memory processors) programmed using light-weight threads, and given that such shared memory processors will be interconnected by high bandwidth message passing networks, how best to design a safety model checker that is (i) efficient for such hardware platforms, and (ii) is modular to permit multiple implementations for different modeling languages?

The importance of this question stems from many facts. First of all, basic finite-state model checking must continue to scale for large-scale debugging. Multiple CPUs per node are best exploited by multi-threaded code running on the nodes; the question, however, is how to organize the threads for high efficiency and modularity, especially given that thread programming is error-prone. Moreover, most parallel versions of safety model checkers employ hash tables distributed across the nodes, with new states possibly sent across the interconnect to be looked up in these tables (as was done since the very first model checkers of this kind, namely Stern and Dill [1] and Lerda and Sisto [2]); we do not deviate from this decision.

What we explore in this paper is whether, by specializing the threads running within each node to specific tasks, (i) the state generation efficiency can be kept high, (ii) communication of states across the interconnect can be performed efficiently, and (iii) the overall code remains simple and modular to be trustworthy.

We have developed a parallel *and* distributed model checking architecture called Eddy that meets the above objectives. A specific model checker following this architecture, called Eddy_Murphi (for the Murphi [3] modeling language) has been developed and released. To the best of our knowledge, such a model checker has previously not been discussed in the literature. There are a wide array of choices available in deciding how to go

[⋆] Supported in part by NSF award CNS-0509379 and SRC Contract 2005-TJ-1318

about designing such a model checker. The decisions involved are how to allocate the CPUs of each compute node to support state generation, hash-table lookup, coalescing states into bigger *lines* before shipment, overlapped computation and communication, and handling distributed termination. Many of these choices may not achieve high performance, and may lead to tricky code. We are placing a great deal of importance on achieving simple and maintainable code, allowing the model checker to be easily re-targeted for a different modeling language, and even make the model checker self calibrating over a wide range of hardware platforms.

While much remains to be explored as well as implemented, Eddy_Murphi has realized many of the essential aspects of the Eddy architecture. In particular, Eddy_Murphi employs shared memory CPU threads in each node running POSIX PThreads [4,5] code, with the nodes communicating using the Message Passing Interface (MPI, [6]). It dramatically reduces the time taken to model check several non-trivial Murphi models, including cache coherence protocols.

We have also: (i) ported Eddy_Murphi to work using a Win32 porting of PThreads [7] as well as Microsoft Compute Cluster Server 2003 [8]; (ii) created Eddy_SPIN, a *preliminary* distributed model checker for Promela[1]. Both Eddy_SPIN and Eddy_Murphi are based on the same architecture: while the state generation ("worker") thread more or less executes the reachability computation aspects of the standard sequential SPIN or Murphi, the communication threads are *organized in an identical manner.*

In the rest of the paper, we will focus on the internal organization of Eddy_Murphi, the impact of its performance over the number of nodes as well as communication parameters such as those controlling state aggregation, as well as scalability results from a catalog of benchmarks. Since we do not have the ability to compare "apples to apples" with other existing model checkers, our contributions fall in the following categories. (i) We provide a detailed description of the algorithms used in Eddy_Murphi. (ii) We report the performance of Eddy_Murphi across a wide spectrum of examples. In one case, Eddy_Murphi model-checked a very huge protocol in 9 hours using 60 nodes when sequential Murphi had not enough memory resources to verify it and a disk-based sequential Murphi [10][2] did not finish even after a week. (iii) In [11], we provide extensive experimental results, the full sources of Eddy_Murphi, as well as a Promela verification model that explicates the detailed organization of its thread and message passing code.

[1] Eddy_SPIN was based on a refactored implementation of SPIN [9] which did not exhibit the scalability advantages reported here for Eddy_Murphi owing to its very high overheads; this will be corrected in our next implementation.

[2] This version of Murphi is able to limit the performance slowdown due to disk usage to an average factor of 3.

The rest of this paper is organized as follows. Section 1.1 presents specific design considerations that lead to the selection of a natural architecture and implementation for Eddy. Section 2 presents the algorithm used by Eddy. Section 3 has our experimental results. Section 4 concludes.

Related Work: Parallel and distributed model checking has been a topic of growing interest, with a special conference series (PDMC) devoted to the topic. An exhaustive literature survey is beyond the scope of this paper. Many distributed model-checkers based on message passing have been developed for Murphi and SPIN. Distributed BDD-based verification tools have been widely studied (e.g., [12]). In [13], a multithreaded SAT solver is described. The idea of coalescing states into larger messages for better network utilization in the context of model checking was pointed out in [14]. Previous parallel Murphi versions has been devised by Stern and Dill [15], Sivaraj and Gopalakrishnan [16], and Kumar and Mercer [17]. As said earlier, a parallel and distributed framework for safety model checking similar to Eddy is believed to be new.

1.1 Design Considerations for Eddy

Our main goal is to have the two threads used in Eddy run without too many synchronizations. This increases the intra node parallelism. Furthermore, if thread-binding to CPUs is available (depending on the underlying OS), then context-switching overhead can also be reduced. Hence, we design our two threads to have complementary tasks, thus maximizing the parallelism between them. One thread will be responsible for state generation, hash table lookup and error analysis, while the other one will handle the communication part, i.e. receiving and sending messages. We also give to this latter thread the task to group up states to be communicated in a big coalesced chunk of memory called a *line*. We experimentally show that this is far more efficient than suffering the overhead of sending individual states across.

Terminology: A *Nondeterministic Finite State System* (shortened *NFSS* in the following) $\mathcal{S}$ is a 4-tuple $(S, I, \mathcal{A}, \texttt{next})$, where S is a finite set of states, $I \subseteq S$ is the set of the initial states, $\mathcal{A}$ is a finite set of labels and $\texttt{next} : S \to 2^{S \times \mathcal{A}}$ is a function taking a state s as argument and returning a set $\texttt{next}(s)$ of pairs $(t, a) \in S \times \mathcal{A}$. Given an NFSS $\mathcal{S} = (S, I, \mathcal{A}, \texttt{next})$ and a property ϕ defined on states (i.e., $\phi : S \to \{true, false\}$), we want to verify if ϕ holds on all the states of $\mathcal{S}$ (i.e., for all $s \in S$, $\phi(s)$ holds). The algorithm in Figure 1 is what Murphi essentially implements[3]. We seek to parallelize this algorithm based on a number of established as well as new ideas. Our objective is to support distributed hash tables as in contemporary works. This assigns each

[3] This rather straightforward algorithm is included in this paper to help contrast our distributed model checker.

```
FIFO_Queue Q = ∅; /* BF consumption queue */
HashTable T = ∅; /* for visited states */

/* Returns true iff $\phi$ holds in all the reachable states */
bool BFS(NFSS $\mathcal{S}$, SafetyProperty $\phi$) {
 let $\mathcal{S} = (S, I, \mathcal{A}, \mathrm{next})$;
 /* is there an initial state which is an error state? */
 foreach s in $I$ {
  if (!IfNotVisitedCheckEnqueue(s, $\phi$))
   /* s is an error state and $\mathcal{S}$ does not satisfy $\phi$ */
   return false;
 }
 while (Q $\neq$ ∅) { /* main BFS loop */
  s = Dequeue(Q);
  foreach (s_next, a) in next(s) { /* s is expanded */
   if (!IfNotVisitedCheckEnqueue(s_next, $\phi$)) return false;
 } /* foreach */ } /* while */
 return true; /* error not found, $\mathcal{S}$ satisfies $\phi$ */
} /* BFS() */

/* Returns false if s is an error state (i.e. does not satisfy $\phi$), true otherwise */
bool IfNotVisitedCheckEnqueue(s, SafetyProperty $\phi$) {
 if (s is not in T) {
  if (!$\phi$(s)) return false;
  HashInsert(T, s);
  Enqueue(Q, s);
 }
 return true;
} /* IfNotVisitedCheckEnqueue() */
```

Fig. 1. Explicit Breadth–First Search

state s to a home node `owner`(s) determined by a surjective partitioning function `owner` that maps state vectors to node numbers lying in the range $\{1 \dots N\}$. Kumar and Mercer [17] study the effect of partitioning on load balancing—an important consideration in parallel model checking. We consider the selection of partition functions to be orthogonal to our work.

Given all this, the state generation rate and the communication demands of a parallel safety model checker very much depends on many factors. The amount of work performed to generate the successor states of a given state is a critical consideration. In Murphi, for instance, each "rule" is a $\langle guard, action \rangle$ pair, with guards and actions being typically coarse-grained. Often, the guards and actions span several pages of code, often involving procedures and functions. In other modeling languages such as Promela and Zing [18], the amount of work to generate the successors of a given state can vary greatly. After gaining sufficient understanding, we hope to have a user-assisted calibration feature for all model checkers constructed following the Eddy architecture.

In the rest of this paper, we assess results from our preliminary implementation. In Section 2 we discuss and describe our algorithm, by giving the pseudocode (Sections 2.1 and 2.2), the rationale behind it (Section 2.3) and an informal discussion of its correctness (Section 2.4). In Section 3 we give and discuss the experimental results we obtained with Eddy_Murphi. Section 4 concludes the paper, giving some future research guidelines.

2 A New Algorithm for Parallel Model Checking

We present the Eddy_Murphi algorithms in Section 2.1, after a brief overview of the MPI and PThread functions used.

MPI functions employed in Eddy_Murphi

MPI (Message Passage Interface, [19,20,6]) is a message-passing library specification, designed to ease the use of message passing by end users, library writers, and tool developers. It is in use in over 60% of the world's supercomputers and clusters. We now present a simplified description of the semantics of certain MPI functions used in our algorithm descriptions (we also take the liberty to simplify the names of these functions somewhat).

– `MPI_Isend(obj, dest_node, msg_label)` sends `obj` to `dest_node`, and the message is labeled `msg_label`.

Note that this operation is non-blocking (the 'I' stands for *immediate*), i.e. it does not wait for the corresponding receive. Here, `obj` is an object of any type, `dest_node` is a node of the computing network, `msg_label` is the label message (chosen between *state*, *termination*, *termination probing*, *termination probing answer* and *continue*). The following always holds:

  - if `msg_label` is *state*, then `obj` is a set of states;
  - if `msg_label` is *termination probing* or *continue*, then `obj` is a dummy variable, since no further information is required;
  - if `msg_label` is *termination probing answer*, then `obj` is a structure containing an integer and a boolean, used in the termination probing procedure;
  - if `msg_label` is *termination*, then `obj` is a boolean value (to be assigned to the global variable `result`).

- `MPI_Iprobe(src_node, msg_label)` returns `true` if there is a message sent by the `src_node` node with the label `msg_label` for the current node. Otherwise, `false` is returned. As the 'I' suggests, also this call is non-blocking. If `src_node` is `ANY_SOURCE` instead of a specific node, then only the message label is checked.
- `MPI_Recv(src_node, msg_label)` returns the message sent by the `src_node` node to the current one with the label `msg_label`. We will call this function only after a successful call to `MPI_Iprobe`, thus we are always sure that a `MPI_Isend` had previously sent something to the current node with the given `msg_label`. Again, if `src_node` is `ANY_SOURCE`, then the current node is retrieving the message without checking which node is the sender (only the message label is checked).
- `MPI_Test(obj)` returns true iff `obj` has been successfully sent, i.e. if the sending has been completed. Note that this is necessary because we are using `MPI_Isend`, that performs an asynchronous sending operation. We will call this function only for test sending completion for states.
- `MPI_MyRank()` returns the rank or identifier of the node. We suppose that each node is numbered starting from 0 and that the node with rank 0 is the root. We also use function `IAmRoot()` as a shortcut for the test `MPI_MyRank() == 0`.
- `MPI_NumProcs()` returns the number of nodes in the MPI universe, i.e. the number of nodes partecipating to the parallel computation.

Finally, with `#MPI_Isend(msg_label)` (resp., `#MPI_Recv(msg_label)`), we denote the number of `MPI_Isend` (resp. `MPI_Recv`) performed with the message label `msg_label`. Note that here `msg_label` is always *state*, i.e. we count only the sending operations regarding sets of states.

PThread functions employed in Eddy_Murphi

POSIX PThread [4,5] is a standardized programming interface for threads usage. In our model checker we use the following functions. Note that, w.r.t. the PThread standard, we again change the function interface to make their usage clearer:

- `pthread_create(f)` creates a new thread. Namely, the thread that calls this function continues its execution, whilst a new thread is started which executes the function `f`.
- `pthread_exit()` terminates the thread which calls it.
- `pthread_join()` called by the "main" thread (i.e. the one having called `pthread_create`), suspends the execution of this thread until the other one terminates (because of a `pthread_exit()`), unless it is already terminated.
- `pthread_yield()` Forces the calling thread to relinquish use of its processor.

Moreover, PThread also provides a mechanism to suspend the execution of a thread till when another thread wakes it up. For the sake of simplicity, in the following we will use this mechanism with an informal description.

Finally, there are suitable PThread functions for the mutual exclusion, to be used when accessing to variables which are shared between the threads.

2.1 Eddy_Murphi Algorithms

In Figg. 3, 4, 5, 6, 7 and 8, we show how the breadth-first (BF) visit of Figure 1 is modified in our parallel approach. The most important variables and data structures used by the parallel algorithm are shown in Figure 2. Since we use a SPMD (Single Program Multiple Data) paradigm, *the code listed is executed on all the nodes of the computational network*. Moreover, each node of the computation network has its own copy of the variables in Figure 2. These variables are shared by the two threads in each node.

The *worker thread* is described in Figg. 3 and 4, and the *communication thread* in Figg. 5, 6, 7 and 8.

Note that some of the data structures we use are read and written by both the threads, e.g. the queues `Q` and `CommQueue`. In our implementation, these accesses are protected with the mutual exclusion mechanism provided by the PThread library. However, for the sake of simplicity, we do not detail this protection in the description of our threads.

The worker thread is somewhat similar to the standard BF visit of Figure 1, but with important changes. One is that only the computation root node generates the start states. However, the most important change is in the handling of the local consumption queue `Q`.

```
/* local data (each node has its own
   copy of this) */
FIFO_Queue Q = ∅;
HashTable T = ∅;
bool Terminate = false;
bool result = true;
bool may_send = true;
FIFO_Queue_lines CommQueue[NumNodes] = ∅;
```

Fig. 2. Variables for each node

```
bool ParBFS(NFSS $\mathcal{S}$, SafetyProperty $\phi$) {
 pthread_create(CommThread);
 /* only root node generates initial
    states */
 if (IAmRoot()) {
  foreach s in $I$ {
   if (!CheckState(s)) {
    Terminate = true;
    break;
 } } }
 /* main loop */
 while (!ParTerminate()) {
  (s, checked) = Dequeue(Q);
  if (!checked) {
   /* s was sent by some other node */
   if (s in T) /* s already visited */
    continue; /* dequeue next state */
   else
    HashInsert(T, s);
  }
  foreach (s_next, a) in next(s) {
   if (!CheckState(s_next)) {
    Terminate = true;
    break;
 } } }
 Terminate = true;
 pthread_join();
 return result;
} /* ParBFS() */
```

Fig. 3. Worker thread (1)

In fact, whenever a new state s is generated, and s turns out not to be an error state, then a states distribution function (called `owner()` in Figure 4) determines if s belongs to the current node or not. In the first case, the current node inserts s in `Q` as well as in the local hash table, unless it was already visited, as it happens in a standalone BF. In the second case, s will be sent to the node `owner`(s) owing it; `owner`(s) will eventually then explore s upon receiving it.

However, in order to avoid too many messages between nodes, we use a queuing mechanism that allows to group as many states as possible in a unique message. To this aim, the worker thread enqueues s in a *communication queue* (`CommQueue` in Figg. 2, 4, 5 and 6). Then, the communication thread will eventually dequeue s from `CommQueue` and send it to `owner`(s). The details of this queuing mechanism will be explained in Section 2.2.

```
/* false if error state found */
bool CheckState(state s) {
 owner_rank = owner(s);
 if (owner_rank == MPI_MyRank()) {
  /* this node owns s */
  if (s is not in T) {
   if (!$\phi$(s)) {
    result = false; return false;
   }
   HashInsert(T, s);
   Enqueue(Q, (s, true));
  } /* otherwise, s already visited */
 } else { /* this node does not own s */
  if (!$\phi$(s)) {
   result = false; return false;
  }
  Enqueue_line(CommQueue[owner_rank],s);
 }
 return true;
} /* CheckState() */

/* true if computation is over */
bool ParTerminate() {
 if (Terminate) return true;
 if (Q $\neq \emptyset$) return false;
 if (!Terminate) sleep;
 if (Terminate) return true;
 return false; /* new states are in Q */
} /* ParTerminate() */
```

Fig. 4. Worker thread (2)

Note that only the worker thread can dequeue states from the local BF consumption queue `Q`. On the other hand, the enqueuing of states in `Q` is performed both by the worker thread (see function `CheckState()` in Figure 4) and the communication thread. This latter case happens as a result of receiving states from some other node (see function `ReceiveStates()` in Figure 6). Since the states received from other nodes could be both new or already visited, the worker thread performs a check after having dequeued a state received from another node. To distinguish between local generated states (already checked for being new or not) and received states (on which the check has to be performed), `Q` stores pairs (state, boolean) instead of states.

As for the communication thread, it consists of an endless loop essentially trying to receive and send messages. As stated earlier, there are five types of messages, each carrying:

– states; this kind of messages can be exchanged by every couple of nodes, where the sender is the node generating the states and the receiver is the node

```
/* Communication thread */
CommThread() {
 while (true) { /* endless loop, may be broken by ProcMess */
  ProcMess(); /* if a termination message is received, terminates the computation */
  if (Terminate) /* may be set by the worker thread (φ does not hold) */
   End(true);
  if (may_send)
   DoSends(); /* try to send what is now on the communication queues */
  Free_lines(CommQueue); /* tests sending completion */
  if (IAmRoot())
   TermProbing(); /* handles termination probing */
} } /* CommThread() */

/* Processes all incoming messages */
ProcMess() {
 if (MPI_Iprobe(ANY_SOURCE, termination)) {
  /* some other non-root node found an error, or the root decided that the search
     has to be terminated */
  result = MPI_Recv(ANY_SOURCE, termination);
  End(false);
 }
 if (MPI_Iprobe(ANY_SOURCE, state))
  ReceiveStates();
 if (!IAmRoot() && MPI_Iprobe(root, termination probing))
  ReceiveTermProb();
 if (!IAmRoot() && MPI_Iprobe(root, continue))
  may_send = true;
} /* ProcessMessages() */
```

Fig. 5. Communication thread (1)

owning the states. More details on the sending of this kind of messages are in Section 2.2.

– termination probings; here, we adapt the procedure employed with Active Messages in [1], of which we recall the basic ideas. Namely, in order to state that the global computation is over, all the nodes in the computing network have to be inactive (i.e. the local BF consumption queue is empty and nothing can be sent and received, see function `SmthngToDo` in Figure 7) and all the messages which have been sent by some node must have been received by some other node. In order to have this latter condition to hold, we must check if $M = \sum_{0 \leq i < N}(S_i - R_i) = 0$, being S_i (resp., R_i) the number of state messages sent (resp., received) by node i and N the number of nodes in the computing network.
Thus, when the root node is inactive, it queries each other node i for its $S_i - R_i$. Note that all of the non-root nodes are not allowed to send any other message after having answered to a query from the root, thus the number S_i is frozen. Moreover, each node also communicates if it is active or not (in the sense described above). As for the root node, it waits for all the answers and computes M, then it decides for the termination. Namely, if the result is 0 and all the nodes are inactive, the parallel computation is indeed over. Otherwise, the parallel computation must go on, and the root sends a message to all the non-root nodes to re-enable the sending of state messages.
Summing up, there are three types of messages for the termination probing, each carrying:
  – termination probing requests; these type of messages can be only sent by the root to the other nodes;
  – termination probing answers; these type of messages are used to reply to the root termination probing requests, thus they are sent by the non-root nodes to the root;
  – continuation notifications; these type of messages are sent by the root to the other nodes when the parallel computation is not over.
– termination; messages of this kind are always broadcasted by one node to all the others. Namely, the source can be either the root node (when the termination probing is successfully terminated) or any node. In the first case, all the reachable states have been globally visited, and the system is correct w.r.t. the invariant property ϕ we want to verify. In the second case, there is an error state somewhere (i.e. a state s such that $\phi(s) = 0$), and the termination message will be sent by the node which has discov-

```
/* Processes incoming state messages */
ReceiveStates() {
 S = MPI_Recv(ANY_SOURCE, state);
 foreach state s in S
  Enqueue(Q, (s, false));
 /* here Q might be empty because of thread scheduling */
 if (worker sleeping && Q ≠ ∅)
  wake the worker thread up; /* wake up and work */
} /* ReceiveStates() */

/* Try to send what it is now in CommQueue */
DoSends() {
 foreach computing node n different from MPI_MyRank() {
  while (lines_ready(CommQueue[n])) {
   S = Dequeue_line(CommQueue[n]);
   MPI_Isend(S, n, state);
} } } /* DoSends() */

/* Shuts down CommThread() */
End(bool broadcast){
 if (broadcast) { /* terminate all the other nodes */
  foreach computing node n
   MPI_Isend(result, n, termination);
 }
 Terminate = true; /* also the worker thread will terminate */
 if (worker sleeping)
  wake the worker thread up; /* wake up and terminate */
 pthread_exit();
} /* End() */
```

Fig. 6. Communication thread (2)

ered it (note that it could be also the root node, and that more than one error state could be discovered at the same time by different nodes).

2.2 The Communication Queue Mechanism

A more detailed description is needed for the communication queue handling (i.e. `CommQueue` in Figg. 2, 4, 5 and 6). The purpose of this data structure is to avoid sending each state separately: on the contrary, it allows to group up as many states as reasonable, thus reducing the communication overhead. Of course, grouping is possible only if the destination is the same, thus there is a communication queue for every possible destination node[4].

Differently from `Q`, which is a traditional FIFO queue (storing pairs (state, boolean)), each communication queue is organized as an array of arrays of states. We will refer to each array of states as a *line*, thus our parallel algorithm depends on two parameters:

`NumLines` the number of lines used;

`LineSize` the number of states for each line.

In Figg. 4, 5 and 6, there are four functions accessing `CommQueue`. In order to explain how they work, we have to say that at every execution time there is only one *active* line (i.e. the line on which the states are currently added), while the other lines status can be:

waiting to be sent these lines already contain all the `LineSize` states they are allowed to, and they are waiting to be sent;

currently being sent also these lines are filled up, but they have already been passed to `MPI_Isend`; however, the sending operation is still not terminated. Following the MPI standard specification, the contents of these lines cannot by accessed until the sending operation has been successfully completed;

waiting to be active these lines contain no states, or have already been successfully sent, so their content can be overwritten with new states.

Thus, three line index lists are maintained, one for each of these line types; we will call the former list `WTBS`, the second one `CBS` and the latter `WTBA`. Initially, the first line is the active one, `WTBA` contains all the other `NumLines` − 1 lines and `WTBS` and `CBS` are empty.

[4] Indeed, our implementation uses `NumNodes` − 1 communication queues per node, while in Figure 2 `NumNodes` queues are declared. This allows to simplify our pseudocode.

```
/* Called by root only, possibly starts the termination probing */
TermProbing() {
 if (SmthngToDo())
  return;
 /* send all the requests */
 foreach computing node n different from root
  MPI_Isend(dummy, n, termination probing);
 num_answers = 0;
 total_messages = 0;
 at_least_one_working = false;
 /* collect all the answers */
 while (num_answers < MPI_NumProcs - 1) {
  if (MPI_Iprobe(ANY_SOURCE, termination probing answer)) {
   answer = MPI_Recv(ANY_SOURCE, termination probing answer);
   if (answer.still_working)
    at_least_one_working = true;
   total_messages = total_messages + answer.messages;
   num_answers = num_answers + 1;
 } }
 /* all answers received, is the computation over? */
 total_messages = total_messages + #MPI_Isend(state) - #MPI_Recv(state);
 if (at_least_one_working || total_messages != 0) {
  /* no, the computation is not over; let the other nodes re-enable state sending */
  foreach computing node n different from root
   MPI_Isend(dummy, n, continue);
 }
 else /* yes, the computation is over */
  End(true);
} /* TermProbing() */
```

Fig. 7. Communication thread: functions for termination (1)

```
/* Called by non-root nodes only; processes incoming termination probing messages */
ReceiveTermProb() {
  dummy = MPI_Recv(root, termination probing);
  answer.still_working = SmthngToDo();
  answer.messages = #MPI_Isend(state) - #MPI_Recv(state);
  MPI_Isend(answer, root, termination probing answer);
  may_send = false;
} /* ReceiveTermProb() */

/* True iff something can be done locally */
bool SmthngToDo() {
 if (worker sleeping) {
  Try DoSends(), then ProcMess();
  return (true iff at least one operation has been performed);
 }
 return true;
} } /* SmthngToDo() */
```

Fig. 8. Communication thread: functions for termination (2)

```
/* Returns a line that can be sent to
   the owner node */
line Dequeue_line(FIFO_Queue_line Q) {
 if (Q.WTBS $\neq \emptyset$) {
  ret = head(Q.WTBS);
  Q.WTBS = tail(Q.WTBS);
  Q.CBS = Q.CBS $\cup$ ret;
  return ret;
 }
 else if (worker sleeping)
  return Q.active_line;
 else
  return NULL;
} /* Dequeue_line() */

/* Checks for sending completion */
Free_lines(FIFO_Queue_lines Qs) {
 foreach computing node n {
  if (n != MPI_MyRank()) {
   foreach line ln in Qs[n].CBS {
    if (MPI_Test(ln)) {
     Q.WTBA = Q.WTBA $\cup$ ln;
     /* here, length(ln) == 0 */
     remove ln from Q.CBS;
} } } } } /* Free_lines() */
```

Fig. 10. Communication queue handling (2)

We are now ready to give the semantics of the four functions manipulating `commQueue`:

`Enqueue_line(CommQueue, state)` called by the worker thread, adds `state` at the end of the active line of `CommQueue`. It also handles the active line filling, by properly modifying `WTBS` and `WTBA`.

`Dequeue_line(CommQueue)` called by the communication thread, returns the first line ready to be sent in `CommQueue`, and properly modifies `WTBS` and `CBS`. If there are no ready lines, and the worker thread is sleeping, then the active line is returned.

`lines_ready(CommQueue)` returns true if `Dequeue_line` returns (a line with) at least one state.

`Free_lines(CommQueues)` calls `MPI_Test` on all the lines currently being sent (no matter which queue they belong). Those lines passing the test are moved to the `WTBA` list.

A more detailed pseudocode describing these function can be found in Figg. 9 and 10.

Summing up, the evolution of a line status is shown in Figure 11, where we use the list acronyms to denote the status of the lines that are stored in them. As for the events causing the status transitions, if l is the line under analysis then the following holds:

1. is triggered when a call to `Enqueue_line` fills up the active line and l is the first of the `WTBA` list;
2. is triggered when a call to `Enqueue_line` fills up the active line (which coincides with l)

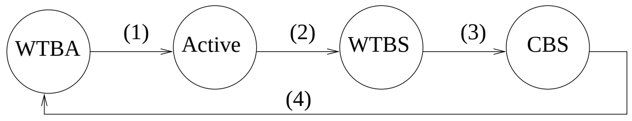


Fig. 11. Evolution of a line status

3. is triggered when a call to `Dequeue_line` returns l;
4. is triggered when a call to `Free_line` finds l to be entirely sent.

Finally, note that the initial state of the automaton in Figure 11 is `Active` for the first line in the lines array, and `WTBA` for all the others.

2.3 Algorithm Rationale

In parallel algorithms for model checking proposed to date, nodes alternate between state generation, state sending, and state receiving. With only one thread available, providing maximal overlap between these activities requires the use of non-blocking MPI communications amidst the rather intricate state generation steps of a model checker. This can render the code brittle, non-portable, and ultimately inadequately concurrent. In contrast, in our design, state generation and communication are in two threads which, on an increasing number of hardware platforms, map onto multi-core CPUs. Through the use of threading and the lines queues, we minimize the time that a worker spends in a waiting state. The threading itself allows the worker not to be kept waiting for communication handling. In fact, there are only two other events that cause the worker thread to wait:

– When the consumption queue is empty (function `ParTerminate` in Figure 4); in this case, the worker thread enters a sleeping status, waiting for some other node to send some new states, or for termination. However, the wait for new states to be processed could be extended if the communication threads keep sending small lines (i.e., containing too few states) to the other nodes. It should be clear that it is more convenient to send as many states as possible in one shot. To achieve this, it is sufficient to set `LineSize` to an adequately high number. Note however that setting this parameter to a too high number may cause a delay in the sending of the states, thus causing other nodes to be idle.
– When there are no available lines in `WTBA` of the communication queue for some node; thus, all the lines are in `WTBS` or `CBS` (in this case, the worker loops in the `while(true)` statement of function `Enqueue_line` in Figure 9). In this case, after a given number of attempts, the worker thread yields to the communication thread, so that some line becomes available earlier. Note that at each iteration the worker also

```
/* Puts s in the active line, and handles filling */
Enqueue_line(FIFO_Queue_line Q, state s) {
 while (true) { /* will be broken when there is an active line */
  if (Q.active_line is defined) {
   Q.active_line = Q.active_line ∪ s;
   if (length(Q.active_line) == LineSize) {
    Q.WTBS = Q.WTBS ∪ Q.active_line;
    if (Q.WTBA == ∅) undefine Q.active_line;
    else {
     Q.active_line = head(Q.WTBA);
     Q.WTBA = tail(Q.WTBA);
     Clear(Q.active_line); /* now, length(Q.active_line) == 0 */
   } }
   break; /* exits while(true) */
  } /* end if (Q.active_line is defined) */
  if (Terminate)
   break; /* exits while(true) */
  if (too much iterations without an active line found)
   pthread_yield(); /* yields to the communication thread */
} } } /* Enqueue_line() */

/* Can something be sent? */
bool lines_ready() {
 if (Dequeue_line can return at least one state)
  return true;
 else
  return false;
} /* lines_ready() */
```

Fig. 9. Communication queue handling (1)

checks if `Terminate` has been set as a result of receiving a termination message (without this check, deadlocks are possible if a termination message is received when the worker is inside `Enqueue_lines`). This problem can be mitigated by properly choosing the number of lines and their length. If there are too few lines, then the worker thread will often be stopped in a waiting status when trying to submit states to the communication queues. Thus, the parameter `NumLines` should be as high as possible.

However, `NumLines` and `LineSize` cannot be set indefinitely high, since they are memory consuming: e.g., if 10 bytes are needed to represent a state in a given model to be verified, then having 1024 lines each with 1024 states on a 50-nodes computation will result in about 500MB RAM memory requirement for each node. This will reduce the space for hash table and consumption queue, so affecting the worker thread performances.

Fortunately, we will show that 1024 or 512 states are a good value for `LineSize`, whilst `NumLines` can be much smaller, e.g. 8 or 16. *In fact, the number of lines merely needs to be large enough to allow overlap of the two threads.*

2.4 Algorithm Correctness

In this Section we give the intuition of why our algorithm is correct, by informally showing that:

- it is not affected by deadlock neither between threads nor between nodes;
- it terminates only when the parallel computation is finished, i.e. when there are no more states to be explored;
- there are no critical runs when the threads access on the shared variables.

As already stated in Section 2.1, we avoid critical runs on shared variables between threads by using proper (and correct) mutual exclusion mechanisms, i.e. the ones provided by the PThread library.

As for the remaining correctness issues, we discuss them individually:

Thread deadlock freeness The two threads running in parallel on each node have to synchronize in only two points, i.e.

1. when a new state has to be enqueued on a line, but there are no available (active) lines. In this case, the worker thread has to wait till the communication thread has freed some lines by sending them;

2. when there are no states in the BF consumption queue, thus the worker thread has to wait for the communication thread to receive some states and put them in the queue.

However, the communication thread never waits for the worker thread[5], thus, for point 1, eventually some line is sent and the worker may continue, and for point 2, eventually some state or a termination message is received (or termination is stated, if the current node is the root).

Nodes deadlock freeness The correctness about this issue comes directly from the fact that each node essentially behaves (as a black-box) as in the algorithm in [1], exploring its states and sending across the states owning to someone else.

Termination We use the termination algorithm in [1], which is known to be correct.

3 Experimental Results

To assess the feasibility of our approach, we implemented our parallel algorithm within the model checker Murphi [21]. We will call the resulting verifier Eddy_Murphi [11].

We use Eddy_Murphi to run different kinds of experiments. All the experiments we run are computed as an average over at least two runs, and were repeated until an acceptable standard deviation was reached (all details provided at [11]).

Initially, we tune the communication parameters, i.e. the number of lines (`NumLines`), and the size of each line (`LineSize`). To do this, we use the protocol sci [15], available within the standard Murphi distribution, modifying its parameters in a way such that it has now a fairly high number of states (approx. 2.7×10^6). We then run different verifications on sci, changing the values for `NumLines` and `LineSize`; these values, as already said in Sect. 2.3, are chosen to be low for `NumLines` and high for `LineSize`; we also change the number of nodes. The results are in Table 1, where **NL** stand for `NumLines`, **LS** for `LineSize` and **Time** % is the ratio between the execution time for Eddy_Murphi and the execution time for standard Murphi. In Table 1, we report only the four best configurations for our parameters, ordered by decreasing time. It is clear that the best results are obtained with 1024 states for each line, and with a number of lines between 8 and 32. To keep memory occupation small enough, we choose 8 lines with 1024 states each.

[5] Indeed, one can think that the communication thread has to wait for the worker thread when a state received from some other node has to be enqueued, but the BF queue is full. However, we use known efficient mechanisms to swap the BF queue on disk, thus this is unlikely to happen, and we did not consider this case in our pseudocode. In our implementation, it is an error for a node not to be able to store all the states in the queue, and the parallel computation is killed.

Next, we use these parameters values to compare the performances of Eddy_Murphi with (standard) Murphi. In these experiments we use five protocols from the Murphi distribution, in order to be able to compare the performances of Eddy_Murphi vs Murphi. These protocols have been chosen in such a way that their number of states is high enough to make the use of parallel model checker meaningful; indeed, they all have between 10^6 and 10^8 states.

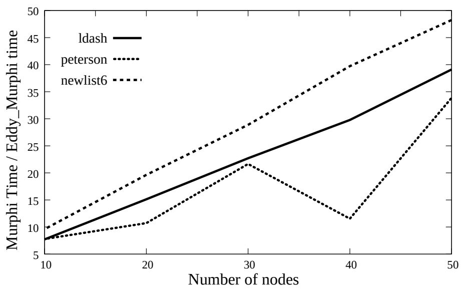
The results are in Figure 12, where we graph the speedup obtained by Eddy_Murphi w.r.t. Murphi (the inverse of Table 1, i.e. $\frac{\text{Murphi time}}{\text{Eddy\_Murphi time}}$) as a function of the number of compute nodes. Figure 12 shows that we obtain a nearly linear speedup on almost all the examples, and that on all examples we are considerably faster than standalone Murphi. Moreover, note that the protocol `peterson` is the only one not showing a linear speedup: running the verification on 40 nodes is worse than on 30. However, this is due to the particular state partition function we use (i.e. the implementation we chose for function `owner` in Figure 4): on this protocol, for 30 nodes we have that each node owns about $\frac{n}{30}$ states, but this does not hold for 40 nodes. Here, we do not address state partition functions performances, since this is an orthogonal problem to our work.

Note that a previous parallel version of Murphi was already developed [22]. We could not re-run the parallel Murphi implementation of [22] because it was developed for the Berkeley NOW hardware which is unavailable. However, when using an MPI porting (reported by [16]), we do not observe the speedup mentioned in [22], and it is always much slower than standard Murphi. This is probably due to the fact that now CPUs are faster, and that the clusters network used in [22] are optimized for message passing, which is not the case with MPI, that privileges the portability. Parallel Murphi implementations were also reported by [17], but we were not able to obtain a reliable version of this code.

Finally, we present a very large protocol whose verification is not feasible on a standalone machine. This is the case of the FLASH protocol [23] with 5 processors and 2 data values as parameters. This protocol has more than 3×10^9 states, and its verification with standard Murphi would require a huge amount of RAM memory (assuming 40 bits for each state in hash compaction, we would need 15 GB of RAM for the hash table only), as well as an unacceptable computational time. On the other hand, by using a disk version of Murphi [10], the computation lasts more than 1 week (we do not know the exact amount of time, but a projection based on the first part of the verification leads to a probable execution time of 3 weeks). However, we successfully completed the verification of this protocol with Eddy_Murphi on 100 nodes in approximately 10 hours on a parallel cluster with 256 nodes (512 CPUs) interconnected with Myrinet and GigE.

40 Nodes			20 Nodes			10 Nodes		
NL	**LS**	**Time %**	**NL**	**LS**	**Time %**	**NL**	**LS**	**Time %**
32	1024	0.023984	32	1024	0.046594	16	1024	0.106446
16	1024	0.023989	2	1024	0.046677	32	1024	0.106805
8	1024	0.024058	16	1024	0.046717	8	1024	0.106833
2	1024	0.024136	8	1024	0.046884	1	512	0.107657

Table 1. Experimental results for the parameter tuning, carried out on a Linux cluster located at the University of Utah with 128 nodes each with two 2.4 Ghz Intel Xeon processors. Each node has two GB of PC2100 SDR SDRAM. The nodes are connected with a GB Ethernet network with MPI latency $< 25\mu s$ and MPI bandwidth of 100 MB/s



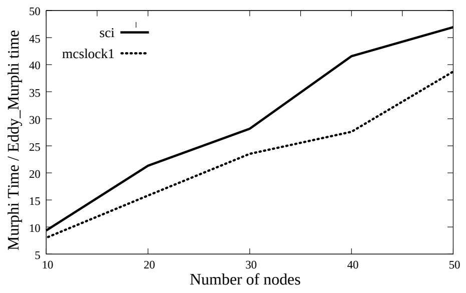


Fig. 12. Experimental results for performances comparison with standard Murphi, carried out on the same cluster of Table 1

4 Conclusions and Future Works

We have developed a novel algorithm and an associated framework for shared memory *and* distributed memory model checking of safety properties, called "Eddy." This is the first such model checker that we are aware of. Eddy meets many goals that we had originally set forth. One important goal was to ensure a clean separation of concerns between *next-state generation* and *communication* during distributed model checking. This, in turn, has several advantages. One advantage is that it makes the code easier to understand, validate, and modify. It also helps make the model checking framework more generic by allowing us to replace the next-state generation logic (e.g., switch over from, say, Murphi to SPIN or Zing) without changing the communication management part very much. Another advantage is the increased concurrency possible when the next-state generation and communication management activities are run as two separate threads. Last but not least, the two threads running per node of Eddy can exploit the two separate CPUs of dual-core CPUs that will become widely available soon. These threads will then have lower or no context-switch overheads, and also utilize the cache memories of the CPUs much more effectively. Eddy optimizes communication in several ways: (i) by not sending individual states, but rather much more bulky units that collect several states before shipment, the interconnection utilization vastly improves. (ii) by performing multiple asynchronous *send*s in an overlapped manner, the overall throughput improves.

Our experiments confirm that the Eddy algorithm is quite robust and scales extremely well on a wide variety of nodes as well as communication parameters such as those controlling state aggregation. In particular, large instances of the Stanford FLASH protocol that cannot be verified through sequential model checking on powerful uniprocessors can now be verified quite fast using multiple nodes. The measurements reported in this paper indicate the actual speed-ups obtained as well as the impact of line sizes and the number of lines on performance.

As part of future work, we hope to combine other optimizations with Eddy. Some of the ideas under consideration are: (i) the use of other ways to record visited states per node, including disk-based algorithms [10], and the use of minimal automata [24], (ii) the use of thread-pools if multiple CPUs are available per node (e.g. hyper-threaded multi-cores), and (iii) self-calibrating versions of Eddy that set its communication thread parameters based on the measured network characteristics.

Finally, it would be interesting to provide some mathematic formulas and/or estimations of the impact of the Eddy parameters on the performances. Apart from the length and the number of the lines of states (determined experimentally in Section 3), this would be interesting also for the number of unsuccessfull attempts the worker thread does to enqueue a state on a line, before yielding the communication thread (Section 2.3).

Acknowledgments

We are grateful to anonymous reviewers for truly helpful comments to improve a previous version of this paper.

References

1. U. Stern and D. Dill. Parallelizing the Murϕ verifier. *Formal Methods in System Design*, 18(2):117–129, 2001. (Journal version of their CAV 1997 paper).
2. F. Lerda and R. Sisto. Distributed-memory model checking with SPIN. In *Proc. of SPIN 1999*, volume 1680 of *LNCS*. Springer, 1999.
3. D. L. Dill. The mur*phi* verification system. In *Proc. of CAV 1996*, volume 1102 of *Lecture Notes in Computer Science*, pages 390–393. Springer, 1996.
4. D. R. Butenhof. *Programming with POSIX Threads.* Addison-Wesley, 1997.
5. POSIX PThreads http://www.llnl.gov/computing/tutorials/pthreads/.
6. MPI tutorial http://www-unix.mcs.anl.gov/mpi/tutorial/gropp/talk.html.
7. PThreads Win32 home page http://sourceware.org/pthreads-win32/.
8. MCCS 2003 http://www.microsoft.com/windowsserver2003/ccs/overview.mspx.
9. R. Palmer and G. Gopalakrishnan. Refactoring spin for safety. Technical report, University of Utah, July 2005.
10. G. Della Penna, B. Intrigila, I. Melatti, E. Tronci, and M. Venturini Zilli. Exploiting transition locality in automatic verification of finite state concurrent systems. *Software Tools for Technology Transfer*, 6(4):320–341, 2004.
11. Eddy_Murphi distribution http://www.cs.utah.edu/formal_verification/software/murphi/eddy_murphi
12. O. Grumberg, T. Heyman, N. Ifergan, and A. Schuster. Achieving speedups in distributed symbolic reachability analysis through asynchronous computation. In *Proc. of CHARME 2005*, volume 3725 of *Lecture Notes in Computer Science*, pages 129–145. Springer, 2005.
13. Y. Feldman, N. Dershowitz, and Z. Hanna. Parallel multithreaded satisfiability solver: Design and implementation. In *Proc. of PDMC 2004*, volume 128 issue 3 of *Electronic Notes in Theoretical Computer Science*, pages 75–90. Elsevier, 2005.
14. R. Kumar and E. Mercer. Scalable distributed model checking: Experiences, lessons, and expectations. In *Proc. of PDMC 2003*, volume 89 issue 1 of *Electronic Notes in Theoretical Computer Science*, page 3. Elsevier, 2003.
15. U. Stern and D. L. Dill. Automatic verification of the sci cache coherence protocol. In *Proc. of CHARME 1995*, volume 987 of *Lecture Notes in Computer Science*, pages 21–34. Springer, 1995.
16. H. Sivaraj and G. Gopalakrishnan. Random walk based heuristic algorithms for distributed memory model checking. In *Proc. of PDMC 2003*, volume 89 issue 1 of *Electronic Notes in Theoretical Computer Science*, pages 51–67. Elsevier, 2003.
17. R. Kumar and E. Mercer. Load balancing parallel explicit state model checking. In *Proc. of PDMC 2004*, volume 128 issue 3 of *Electronic Notes in Theoretical Computer Science*, pages 19–34. Elsevier, 2004.
18. T. Andrews, S. Qadeer, S. K. Rajamani, J. Rehof, and Y. Xie. Zing: A model checker for concurrent software. Technical report, Microsoft Research, 2004.
19. W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface.* MIT Press, 1999.
20. MPI official specification http://www.mpi-forum.org/docs/docs.html.
21. Murphi distribution http://sprout.stanford.edu/dill/murphi.html.
22. U. Stern and D. Dill. Parallelizing the murφ verifier. In Orna Grumberg, editor, *Proc. of CAV 1997*, volume 1254 of *Lecture Notes in Computer Science*, pages 256–278. Springer, 1997.
23. J. Kuskin and D. Ofelt et al. The Stanford FLASH multiprocessor. In *Proc. of SIGARCH 1994*, pages 302–313, May 1994.
24. G.J. Holzmann and A. Puri. A minimized automaton representation of reachable states. *Software Tools for Technology Transfer*, 3(1):270–278, 1998.