A Performance Study of Distributed Timed Automata Reachability Analysis

Gerd Behrmann

Department of Computer Science, Aalborg University, Denmark

Abstract
We experimentally evaluate an existing distributed reachability algorithm for timed automata on a Linux Beowulf cluster. It is discovered that the algorithm suffers from load balancing problems and a high communication overhead. The load balancing problems are caused by inclusion checking performed between symbolic states unique to the timed automaton reachability algorithm. We propose adding a proportional load balancing controller on top of the algorithm. We evaluate various approaches to reduce communication overhead by increasing locality and reducing the number of messages. Both approaches increase performance but can make load balancing harder and has unwanted side effects that result in an increased workload.

1 Introduction
Interest in parallel and distributed model checking has risen in the last 5 years. Not that it solves the inherent performance problem (the state explosion problem), but the promise of a linear speedup simply by purchasing extra processing units attracts customers and researchers.

Uppaal [3] is a popular model checking tool for dense time timed automata. One of the design goals of the tool is that orthogonal features should be implemented in an orthogonal manner such that competing techniques can be compared. The design of a distributed version of Uppaal [4] which in turn was based on the design of a distributed version of Murϕ[19], is indeed true to this idea and allows the distributed version to utilise almost any of the existing techniques previously implemented in the tool.

The distributed algorithm proposed in [4] was evaluated with very positive results, but mainly on a parallel platform providing very fast and low overhead communication. Experiments on a distributed architecture (a Beowulf cluster) were preliminary and inconclusive. Later experiments on another Beowulf
cluster showed quite poor performance, and even after tuning the implementation, we only got relatively poor speedups as seen in Fig. 1. Closer examination uncovered load balancing problems and a very high communication overhead. We also uncovered that although most options in Uppaal are orthogonal to the distribution, they can have a crucial influence on the performance of the distributed algorithm. Especially the state space reduction techniques of [17] showed to be problematic. On the other hand, a recent change in the data structures[10] showed to have a very positive effect on the distributed version as well.

Fig. 1. The speedup obtained with a unoptimised distributed reachability algorithm for a number of models.

Contributions
We analyse the performance of the distributed version of Uppaal on a 14 node Linux Beowulf cluster. The analysis shows unexpected load balancing problems and a high communication overhead. We contribute results on adding an extra load balancing layer on top of the existing random load balancing previously used in [4,19]. We also evaluate the effect of using alternative distribution functions and buffering communication.

Related Work
The basic idea of the distributed state space exploration algorithm used has been studied in many related areas such as discrete time and continuous time Markov chains, Petri nets, stochastic Petri nets, explicit state space enumeration, etc. [8,1,9,15,16,19] although alternative approaches are emerging[5,12]. In most cases close to linear speedup and very good load balancing is obtained.
Little work on distributed reachability analysis for timed automata has been done. Although very similar to the explicit state space enumeration algorithms mentioned, the classical timed automata reachability algorithm uses symbolic states (not to be confused with work on symbolic model checking, where the transition relation is represented symbolically) which makes the algorithm very sensitive to the exploration order.

Outline

Section 2 summarises the definition of a timed automaton, the symbolic semantics of timed automata, the distributed reachability algorithm for timed automata presented in [4] and introduces the basic definitions and experimental setup used in the rest of the paper. In section 3 we discuss load-balancing issues of the algorithm. In section 4 techniques for reducing communication by increasing locality are presented and in section 5 we discuss the effect of buffering on the performance of the algorithm in general and on the load-balancing techniques presented in particular.

2 Preliminaries

In this section we summaries the basic definition of a timed automaton, the symbolic semantics, the distributed reachability algorithm, and the experimental setup.

Definition 2.1 (Timed Automaton) Let \( C \) be the set of clocks. Let \( B(C) \) be the set of conjunctions over simple conditions on the form \( x \infty c \) and \( x - y \infty c \), where \( x, y \in C \) and \( \infty \in \{<, \leq, =, \geq, >\} \). A timed automaton over \( C \) is a tuple \((L, l_0, E, g, r, I)\), where \( L \) is a set of locations, \( l_0 \in L \) is the initial location, \( E \in L \times L \) is a set of edges, \( g : E \to B(C) \) assigns guards to edges, \( r : E \to 2^C \) assigns clocks to be reset to edges, and \( I : L \to B(C) \) assigns invariants to locations.

Intuitively, a timed automaton is a graph annotated with conditions and resets of non-negative real valued clocks. A clock valuation is a function \( u : C \to \mathbb{R}_{\geq 0} \) from the set of clocks to the non-negative reals. Let \( \mathbb{R}^C \) be the set of all clock valuations. We skip the concrete semantics in favour of an exact finite state abstraction based on convex polyhedra in \( \mathbb{R}^C \) called zones (a zone can be represented by a conjunction in \( B(C) \)). This abstraction leads to the following symbolic semantics.

Definition 2.2 (Symbolic TA Semantics) Let \( Z_0 = \bigwedge_{x,y \in C} x = y \) be the initial zone. The symbolic semantics of a timed automaton \((L, l_0, E, g, r, I)\) over \( C \) is defined as a transition system \((S, s_0, \Rightarrow)\), where \( S = L \times B(C) \) is the set of symbolic states, \( s_0 = (l_0, Z_0 \land I(l_0)) \) is the initial state, \( \Rightarrow = \{(s, u) \in S \times S | \exists e, t : s \Rightarrow t \Rightarrow u\} \) is the transition relation, and:

- \((l, Z) \xRightarrow{\delta} (l, \text{norm}(M, (Z \land I(l)))^1 \land I(l)))\)
• \((l, Z) \Rightarrow (l', r_{e}(g(e) \land Z \land I(l)) \land I(l'))\) if \(e = (l, l') \in E\).

where \(Z^f = \{u + d \mid u \in Z \land d \in \mathbb{R}_{\geq 0}\}\) (the future operation), and \(r_{e}(Z) = \{[r(e) \mapsto 0]u \mid u \in Z\}\). The function \(\text{norm} : \mathbb{N} \times B(C) \rightarrow B(C)\) normalises the clock constraints with respect to the maximum constant \(M\) of the timed automaton.

Notice that a state \((l, Z)\) of the symbolic semantics is actually a set of concrete states \(\{(l, u) \mid u \in Z\}\). The classical representation of a zone is the Difference Bound Matrix (DBM). For further details on timed automata see for instance [2,7]. The symbolic semantics can be extended to cover networks of communicating timed automata (resulting in a location vector to be used instead of a location) and timed automata with data variables (resulting in the addition of a variable vector).

The Algorithm

Given the symbolic semantics it is straightforward to construct the reachability algorithm. The distributed version of this algorithm is shown in Fig. 2 (see also [4,19]). The two main data structures of the algorithm are the \textit{waiting} list and the \textit{passed} list. The former holds all unexplored reachable states and the latter all explored reachable states. States are popped of the waiting list and compared to states in the passed list to see if they have been previously explored. If not, they are added to the passed list and all successors are added to the waiting list.

\[
\begin{align*}
\text{waiting}_A &= \{(l_0, Z_0 \land I(l_0)) \mid h(l_0) = A\} \\
\text{passed}_A &= \emptyset \\
\textbf{while } \neg \text{terminated } \textbf{do} \\
&\quad (l, Z) = \text{waiting}_A.\text{popState}() \\
&\quad \text{if } \forall (l, Y) \in \text{passed}_A : Z \not\subseteq Y \textit{ then} \\
&\quad \quad \text{passed}_A = \text{passed}_A \cup \{(l, Z)\} \\
&\quad \forall (l', Z') : (l, Z) \Rightarrow (l', Z') \textit{ do} \\
&\quad \quad d = h(l', Z') \\
&\quad \quad \text{if } \forall (l', Y') \in \text{waiting}_d : Z' \not\subseteq Y' \textit{ then} \\
&\quad \quad \quad \text{waiting}_d = \text{waiting}_d \cup \{(l', Z')\} \\
&\quad \quad \text{endif} \\
&\quad \text{done} \\
&\text{endif} \\
&\text{done}
\end{align*}
\]

Fig. 2. The distributed timed automaton reachability algorithm parameterised on node \(A\). The waiting list and the passed list is partitioned over the nodes using a function \(h\). States are popped of the local waiting list and added to the local passed list. Successors are mapped to a destination node \(d\).
The passed list and the waiting list are partitioned over the nodes using a distribution function. The distribution function might be a simple hash function. It is crucial to observe that due to the use of symbolic states, looking up states in either the waiting or the passed list involves finding a superset of the state. A hash table is used to quickly find candidate states in the list\[6\]. This is also the reason why the distribution function only depends on the discrete part of a state.

**Definition 2.3 (Node, Distribution function)** A single instance of the algorithm in Fig. 2 is called a node. The set of all nodes is referred to as $\mathbb{N}$. A distribution function is a mapping $h : L \rightarrow \mathbb{N}$ from the set of locations to the set of nodes.

**Definition 2.4 (Generating nodes, Owning node)** The owning node of a state $(l, Z)$ is $h(l)$, where $h$ is the distribution function. A node $A$ is a generating node of a state $(l, Z)$ if there exists $(l', Z')$ s.t. $(l', Z') \Rightarrow (l, Z)$ and $h(l') = A$.

**Termination**

It is well-known that the symbolic semantics results in a finite number of reachable symbolic states. Thus, at some point every generated successor $(l, Z)$ will be included in $\bigcup_{A \in \mathbb{N}} \text{passed}_A$ or more precisely in $\text{passed}_{h(l)}$ for the same reason as in the sequential case. Termination is a matter of detecting when all nodes become idle and no states are in the process of being transmitted. There are well known algorithms for performing distributed termination detection. We use a simplified version of the token based algorithm in [11].

**Transient States**

A common optimisation which applies equally well to the sequential and the distributed algorithm is described in [17]. The idea is that not all states need to be stored in the passed list to ensure termination. We will call such states transient. Transient states tend to reduce the memory consumption of the algorithm. In section 4 we will describe how transient states can increase locality.

**Search Order**

A previous evaluation [4] of the distributed algorithm showed that the distribution could increase the number of generated states due to missed inclusion checks and the non-breadth first search order caused by non-deterministic communication patterns. It was discovered that this effect could be reduced by ordering the states in a waiting list according to distance from the initial state and thus approximating breadth-first search order. The same was found to be true for the experiments performed for this paper and therefore this ordering has been used.
Platform

Our previous experiments were done on a Sun Enterprise 10000 parallel computer equipped with 24 CPUs\cite{4}. The experiments for this paper have been performed on a cluster consisting of 7 dual 733MHz Pentium III machines equipped with 2GB memory each, configured with Linux kernel 2.4.18, and connected by switched Fast Ethernet. It still uses the non-blocking communication primitives of the Message Passing Interface\cite{3}, but a number of MPI related performance issues have been fixed.

Experiments

Experiments were performed using six existing models: The well-known Fischer’s protocol for mutual exclusion with six processes (fischer6); the startup algorithm of the DACAPO \cite{18} protocol (dcapo_sim); a communication protocol (ir) used in B&O audio/video equipment \cite{14}; a power-down protocol (model3) also used in B&O equipment \cite{13}; and a model of a buscoupler (buscoupler3). The DACAPO model is very small (the reachable state space is constructed within a few seconds). The model of the buscoupler is the largest and has a reachable state space of a few million states.

The performance of the distributed algorithm was measured on 1, 2, 4, 6, 8, 10, 12, and 14 nodes. Experiments are referred to by name and the number of nodes, e.g. fischer6×8 for an experiment on 8 nodes. In all experiments the complete reachable state space was generated and the total hash table size of each of the two lists was kept constant in order to avoid that the efficiency of these two data structures depends on the number of nodes (in \cite{4} this was not done and caused the super linear speedup observed). Notice that Fig. 1 was produced with an older version of UPPAAL before the techniques described in this paper were implemented. Since then UPPAAL has become considerably faster and thus the communication overhead has become relatively higher.

3 Balancing

The distributed reachability algorithm uses random load balancing to ensure a uniform workload distribution. This approach worked nicely on parallel machines with fast interconnect \cite{4,19}, but as mentioned in the introduction resulted in very poor results when run on a cluster. Figure 3 shows the load of buscoupler3×2 with the same algorithm used in Fig. 1. In this section we will study why the load is not balanced and how this can be resolved.

Definition 3.1 (Load, Transmission rate, Exploration rate) The load

\footnote{That paper also reported on very preliminary and inconclusive experiments on a small cluster.}

\footnote{We use the LAM/MPI implementation found at \url{http://www.lam-mpi.org}.}
of a node $A$, denoted $\text{load}(A)$, is the length of the waiting list at node $A$, i.e.,

$$\text{load}(A) = |\text{Wait}_A|.$$  

The transmission rate of a node is the rate at which states are transmitted to other nodes. We distinguish between the outgoing and incoming transmission rates. The exploration rate is the rate at which states are popped of the waiting list.

Notice that the waiting list does not have $O(1)$ insertion time. Collisions in the hash table can result in linear time insertion (linear in the load of the node). Collisions are to be expected since several states might share the same location vector and thus hash to the same bucket — after all this is why we did inclusion checking on the waiting list in the first place. Thus the exploration rate depends on the load of the node and the incoming transmission rate.

Apparently, what is happening is the following. Small differences in the load are to be expected due to communication delays and other random effects. If the load on a node $A$ becomes slightly higher compare to node $B$, more time is spent inserting states into the waiting list and thus the exploration rate of $A$ drops. When this happens, the outgoing transmission rate of $A$ drops causing the exploration rate of $B$ to increase, which in turn increases the incoming transmission rate of $A$. Thus a slight difference in the load of $A$ and $B$ causes the difference to increase, resulting in an unstable system where the load of one or more nodes quickly drops to zero. Although the node still receives states from other nodes, having an unbalanced system is bad for several reasons: First, it means that the node is idle some of the time, and second it prevents successful inclusion checking on the waiting list. The latter was proven to be important for good performance\cite{6}. We apply two strategies to solve this problem.

The first is to reduce the effect of small load differences on the exploration rate by merging the hash table in the waiting list with the hash table in the passed list into a single unified hash table. This change was recently
documented in [10]. This tends to reduce the influence of the load on the exploration rate, since the passed list is much bigger than the waiting list. The effect on the balance of the system is positive for most models, although fischer6 still shows signs of being unbalanced, see Fig. 4.\footnote{The load is only shown for a setup with 2 nodes to reduce clutter in the figures. The results are similar when running with all 14 nodes, but much harder to interpret in a small figure.}

Fig. 4. Unifying the hash table of the passed list and the waiting list resolves the load balancing problems for some models (a), but not for others (b).

The second strategy is to add an explicit load balancing scheme on top of the random load balancing. The idea is that as long as the system is balanced random load balancing works fine. The hope is that the explicit load balancer can maintain the balance without causing two much overhead. The load balancer is invoked for each successor. It decides whether to sent the state to its owning node or to redirect it to another node. Redirection has the effect that the state is stored at the wrong node which can reduce efficiency as some states might be explored several times. We will apply a simple proportional controller to decide whether a state should be redirected. The set point of this controller will be the current average load of the system. Notice that it is the node generating a state that redirects it and not the owning node itself. Thus the state is only transfered once. Information about the load of a node is piggybacked with the states.

**Definition 3.2 (Load average, Redirection probability)** The load average is defined as $\text{load}_{\text{avg}} = \frac{1}{|N|} \sum_{A \in N} \text{load}(A)$. The probability that a state is redirected to node $B$ instead of to the owning node $A$ is $P_{A \rightarrow B} = P^1_A \cdot P^2_B$, where:

$$P^1_A = \begin{cases} 0 & \text{if } \text{load}(A) - \text{load}_{\text{avg}} \leq 0 \\ 1 & \text{if } \text{load}(A) - \text{load}_{\text{avg}} \geq c \\ \frac{\text{load}(A) - \text{load}_{\text{avg}}}{c} & \text{otherwise} \end{cases}$$
\[ P_A^2 = \frac{\max(\text{load}_{\text{avg}} - \text{load}(A), 0)}{\sum_{B \in N} \max(\text{load}_{\text{avg}} - \text{load}(B), 0)} \]

\[ P_A^1 \] is the probability that a state owned by node \( A \) is redirected and \( P_B^2 \) is the probability that it is redirected to node \( B \). Notice that \( P_A^1 \) is zero if the load of \( A \) is under the average (we do not take states from underloaded nodes), that \( P_B^2 \) is zero if the load of \( B \) is above the average (we do not redirect states to overloaded nodes), and that \( \sum_{A \in N} P_A^2 = 1 \), hence \( \sum_{B \in N} P_{A \rightarrow B} = P_A^1 \). The value \( c \) determines the aggressiveness of the load balancer. If the load of a node is more than \( c \) states above the average then all states owned by that node will be redirected. For the moment we let \( c = \text{load}_{\text{avg}} \).

Two small additions reduce the overhead of load balancing. The first is the introduction of a dead zone, i.e., if the difference between the actual load and the load average is smaller than some constant, then the state is not redirected. The second is that if the generating node and the owning node of a successor is the same, then the state will not be redirected. The latter tends to reduce the communication overhead but also reduces the aggressiveness of the load balancer.

Experiments have shown that the proportional controller results in the load to be almost perfectly balanced for large systems except \textit{fischer6}. Figure 5(a) shows that the load balancer has difficulties keeping \textit{fischer6} balanced (although it is more balanced than without it), but still results in an improved speedup as seen in Fig. 5(b).

![Figure 5](image)

(a) Load of \textit{fischer6×2}  
(b) Speedup

Fig. 5. The addition of explicit load balancing has a positive effect on the balance of the system. (a) shows the load of \textit{fischer6×2} and the average number of states each node redirects each second, (b) shows the speedup obtained.
4 Locality

The results presented in the previous section are not satisfactory. Speedups obtained are around 50% of linear even though the load is balanced. The problem is overhead caused by the communication between nodes. In this section we evaluate two approaches to reduce the communication overhead by increasing the locality.

![Graph](image)

Fig. 6. The total CPU time used for a given number of nodes divided into either time spent in user space/kernel space (left column) or into time spent for receiving/sending/packing states into buffers/non mpi related operations (right column). Figure (a) shows the time for buscoupler3 with load balancing and figure (b) for fischer6 without load balancing.

Since all communication is asynchronous the verification algorithm is relatively robust towards communication latency. In principle, the only consequences of latency should be that load informations are slightly outdated and that the approximation of breadth first search order is less exact. On the other hand the message passing library, the network stack, data transferred between memory and the network interface, and interrupts triggered by arriving data use CPU cycles that could otherwise be used by the verification algorithm. Figure 6(a) shows the total CPU time used by all nodes for the buscoupler3 system. The CPU time is shown in two columns: the left is divided into time spent in user space and kernel space, the right is divided into time used for sending, receiving, packing data into and out of buffers, and the remaining time (non-mpi). It can be seen that the overhead of communicating between two nodes on the same machine is low compared to communicating between nodes on different machines (compare the columns for 1, 2 and 4 nodes). For 4 nodes and more we see a significant communication overhead, but there is also a significant increase in time spent on the actual verification (non-mpi). The increase seen between 1 and 2 nodes is likely due to two nodes sharing
the same memory bus of the machine. **UPPAAL** is very memory intensive and
sharing the memory bus will cause an overhead. The increase seen between 2
and 4 nodes is likely due to an increased number of interrupts caused by the
communication.

The communication overhead is directly related to the amount of states
transferred. Let \( n = |\mathcal{N}| \) be the number of nodes, \( m \) the number of nodes
located at a single physical machine, and \( S \) be the total number of states
generated. If all machines perform the same amount of work, we expect that each
node generates \( \frac{S}{n} \) states. Assuming that the distribution function distributes
states uniformly, we expect that each node sends \( \frac{S}{n} \) states to any other node
(including itself).

For any given node, there are \( m - 1 \) other nodes located at the same
machine and \( n - m \) nodes at other machines. Let \( t_{local} \) be the overhead of
sending a state to a node located at the same machine, and \( t_{remote} \) to a node at
another machine. We then get the following expression for the communication
overhead:

\[
t_h = n \frac{S}{n^2} (t_{local}(m-1) + t_{remote}(n-m))
\]  

(1)

Figure 6 shows \( t_h + t_v \) (theoretical), where \( t_v \) is the time used for the actual
verification (non-mpi). The two constant \( t_{local} \) and \( t_{remote} \) are computed from
the measured overhead on 2 and 4 nodes. The definition of \( t_h \) assumes that
the overhead of transferring a state is constant which is not necessarily the
case, for instance when the bandwidth requirements are higher that the band-
width available or the load is not balanced so that nodes perform blocking
receives. Figure 6(b) shows the unbalanced verification of **fischer6** and the
time used in blocking receive calls is significant. Consequently, the predicted
communication overhead is less precise. It is interesting to note that the com-
puted overhead tends to be below the actual overhead. This indicates that
it becomes more expensive to sent a state as the number of nodes increases,
either due to the increased load on the network or from overhead in the MPI
implementation (the latter being the more likely explanation).

One way to reduce the amount of states transfered is to choose a distri-
bution function that increases the chance that the generating node is also
the owning node while keeping the balance. In other words, the distribution
function should increase locality.

**Definition 4.1 (Locality)** The locality, \( l \), of a distribution function is the
number of states owned by a generating node relative to the total number of
states generated, \( S \).

In (1) we assume the locality of the distribution function to be \( \frac{1}{n} \). A
good distribution function has a high locality while maintaining that the load
is evenly distributed, i.e. each node explores \( \frac{S}{n} \) nodes. A locality of 1 is
undesirable since it prevents any load balancing. Assuming that all non-local
states are distributed uniformly we get the following expression for the load
overhead:

\[ t(l) = n \frac{1 - \frac{S}{n-1} \ln(n)}{n - 1} \left( t_{local}(m - 1) + t_{remote}(n - m) \right) \]

\[ = \frac{S - L}{n - 1} \left( t_{local}(m - 1) + t_{remote}(n - m) \right) \tag{2} \]

where \( \frac{1 - \frac{S}{n-1} \ln(n)}{n - 1} \) is the number of states each nodes sends to any other node (excluding itself) and \( L \) is the total number of states owned by a generating node. It is easy to see that \( t \left( \frac{1}{n} \right) = t_h \).

In general, it is difficult to construct a distribution function that is guaranteed to have a high locality while maintaining a good load distribution. A good heuristic for input models with a high number of integer variables is to only compute the owning node based on the variable vector. Since not all transitions update the integer variables this tends to increase the chance that the successor is owned by the node generating it. Figure 7(a) shows the resulting locality as a function of number of nodes. Compare this to \( \frac{1}{n} \) locality obtained by hashing on both the location vector and the variable vector. Figure 7(b) shows the CPU time for buscoupler3. Comparing this to Fig. 6(a) shows that the communication overhead is significantly reduced.

Another way to increase locality is by exploring all transient states locally. Transient states are not stored in the passed list anyway, so termination is still guaranteed. Figure 8(a) shows the speedup obtained by only marking committed states as transient. Figure 8(b) uses the technique of [17] to

---

5 The concept of committed locations is an UPPAAL extension to timed automata. Committed locations are used to create atomic sequences of transitions. A state is committed if any of the locations in the state are committed.
increase the number of transient states by marking all non loop entry points as transient. Both approaches increase the locality, but experiments show that using the latter technique actually decreases performance. Not sending transient states to the owning node can cause a significant overhead since these states cannot be coalesced by the waiting list of the owning node anymore. Using the technique of [17] raises the number of transient states to an extent where coalescing performed by the waiting list is more significant than the overhead caused by the communication.

(a) Only committed states are transient.  
(b) All non loop entry points are transient.

Fig. 8. An alternative means of increasing locality is by exploring all transient states locally. Notice that fischer6 has no committed states, hence the locality for this model in figure (a) is $\frac{1}{n}$.

5 Buffering

In the previous section we tried to reduce the amount of communication by reducing the number of states that needed to be transferred between nodes. It is well known that communication overhead can be reduced by putting several states into each message, thereby increasing the message size but reducing the number of messages. In fact, the results in the previous sections where obtained with a buffer size of 8, i.e., each MPI message contained 8 states. In this section we will study the effect of buffering on the load balancing algorithm.

Figure 9 shows the effect of buffering states before sending them. Only the results for fischer6×14 and buscoupler3×14 are shown. In can be seen that the speedup increases as the buffer size is increased up to a certain point at which the speedup decreases again. A size of 20 to 24 states per buffer seems to be optimal.

One might wonder why the performance actually decreases when increasing the buffer size further. There are several explanations. Increasing the buffers
Fig. 9. The speedup obtained increases as more states are buffered and sent in a single message. A buffer size of 20 to 24 states seems to be optimal. The results with and without load balancing are shown. For \textit{buscoupler3} the results of only distributing states based on the variable vector are also shown (the \texttt{bWCapD1} option).}

increases the latency in the system. This in turn makes load information outdated and delays the effect of the load balancing decision. Comparing the load for \textit{buscoupler3}×14 in Fig. 10 when using a buffer size of one and a buffer size of 96 illustrates this point, as the latter is much less balanced and the average number of states redirected is much higher, which in turn increases the number of generated states. Another factor is related to the approximation of breadth first search order. If the latency is increased, then the approximation will be less precise. This in turn might increase the number of symbolic states explored due to fewer successful inclusion checks. And finally, while a state is buffered it cannot be coalesced with other states (which only happens at the owning node), which in turn might increase the number of states explored. The increase in number of generated states is shown in Fig. 11.

6 Conclusion

We have presented a performance analysis of the distributed reachability analysis algorithm for timed automata used in \texttt{Uppaal} on a Beowulf Linux cluster. Experiments have shown load balancing problems caused by non-constant time operations in the exploration algorithm. These balancing problems were shown to be reduced or solved (depending on the input model) by using a unified representation of the passed list and waiting list data structures used in the algorithm, and by adding an extra load balancing layer. Even on a
Fig. 10. Load of buscoupler3×14 using no buffering (a) and a buffer size of 96 states (b). Increasing the buffer size makes the system less balanced which causes a significant overhead.

Fig. 11. The increased latency and unbalance resulting from a large buffer results in an increased number of generated states. The number of states are shown relative to the number of states generated by the sequential version of the algorithm.

balanced system, the communication overhead of MPI over TCP/IP over Fast Ethernet is server. This overhead can be reduced by using alternative distribution functions that only hash on a subset of a state thereby increasing locality in the algorithm. Also, buffered communication is effective at reducing the communication overhead, but at the expense of increased latency which in turn reduces the effectiveness of the load balancing and the search order
heuristic introduced in [4].

For further work we plan to investigate alternatives to the proportional controller used in the load balancer, for instance, using a PI-controller or PID-controller. The communication overhead could be reduced further by using a multi threaded design, such that each physical machine executes several exploration threads instead of several processes. On our cluster, this would effectively reduce the load balancing and communication problems to 7 nodes instead of 14. Finally, alternatives to using MPI over TCP/IP should be evaluated, for instance by accessing the Ethernet devices directly.

References

[1] S. Allmaier, S. Dalibor, and D. Kreische. Parallel graph generation algorithms for shared and distributed memory machines. In *Parallel Computing: Fundamentals, Applications and New Directions, Proceedings of the Conference ParCo’97*, volume 12. Elsevier, Holland 1997.

[2] R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–235, 1994.

[3] Tobias Amnell, Gerd Behrmann, Johan Bengtsson, Pedro R. D’Argenio, Alexandre David, Ansgar Fehnker, Thomas S. Hune, Bertrand Jeannet, Kim Larsen, Oliver Möller, Paul Pettersson, Carsten Weise, , and Wang Yi. Uppaal - now, next, and future. In *MOVEP’2k*, volume 2067 of *Lecture Notes in Computer Science*. Springer-Verlag, 2001.

[4] Gerd Behrmann, Thomas Hune, and Frits Vaandrager. Distributed timed model checking - How the search order matters. In *Proc. of 12th International Conference on Computer Aided Verification*, Lecture Notes in Computer Science, Chicago, Juli 2000. Springer-Verlag.

[5] S. Ben-David, T. Heyman, O. Grumberg, and A. Schuster. Scalable distributed on-the-fly symbolic model checking. In *3rd International Conference on Formal methods in Computer Aided Design (FMCA’00)*, November 2000.

[6] Johan Bengtsson. Reducing memory usage in symbolic state-space exploration for timed systems. Technical Report 2001-009, Uppsala University, Department of Information Technology, May 2001.

[7] Patricia Bouyer, Catherine Dufourd, Emmanuel Fleury, and Antoine Petit. Are timed automata updatable? In *Proceedings of the 12th Int. Conf. on Computer Aided Verification*, volume 1855 of *Lecture Notes in Computer Science*. Springer-Verlag, 2000.

[8] S. Caselli, G. Conte, and P. Marenzoni. Parallel state space exploration for gspn models. In *Application and Theory of Petri Nets*, volume 935 of *Lecture Notes in Computer Science*. Springer-Verlag, 1995.
[9] G. Ciardo, J. Gluckman, and D. Nicol. Distributed state space generation of discrete state stochastic models. *INFORMS Journal on Computing*, 10(1):82–93, 1998.

[10] Alexandre David, Gerd Behrmann, Wang Yi, and Kim G. Larsen. The next generation of Uppaal. Submitted to RTTOOLS 2002.

[11] E. W. Dijkstra and C. S. Scholten. Termination detection for diffusing computations. *Information Processing Letters*, 11(1):1–4, August 1980.

[12] O. Grumberg, T. Heyman, and A. Schuster. Distributed model checking for mu-calculus. In *International Conference on Computer Aided Verification (CAV’01)*, Lecture Notes in Computer Science. Springer-Verlag, July 2001.

[13] K. Havelund, K. Larsen, and A. Skou. Formal verification of a power controller using the real-time model checker UPPAAL. In Joost-Pieter Katoen, editor, *Formal Methods for Real-Time and Probabilistic Systems, 5th International AMAST Workshop, ARTS’99*, volume 1601 of *Lecture Notes in Computer Science*, pages 277–298. Springer-Verlag, 1999.

[14] K. Havelund, A. Skou, K. G. Larsen, and K. Lund. Formal modelling and analysis of an audio/video protocol: An industrial case study using UPPAAL. In *Proc. of the 18th IEEE Real-Time Systems Symposium*, pages 2–13, December 1997. San Francisco, California, USA.

[15] 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 *Proceedings of the 8th International Workshop on Petri Nets and Performance Models PNPM’99*. IEEE Computer Society Press, 1999.

[16] W. J. Knottenbelt and P.G. Harrison. Distributed disk-based solution techniques for large markov models. In *Proceedings of the 3rd International Meeting on the Numerical Solution of Markov Chains NSMC’99*, Spain, September 1999. University of Zaragoza.

[17] Fredrik Larsson, Kim G. Larsen, Paul Pettersson, and Wang Yi. Efficient Verification of Real-Time Systems: Compact Data Structures and State-Space Reduction. In *Proc. of the 18th IEEE Real-Time Systems Symposium*, pages 14–24. IEEE Computer Society Press, December 1997.

[18] H. Lönn and P. Pettersson. Formal verification of a TDMA protocol startup mechanism. In *Proc. of the Pacific Rim Int. Symp. on Fault-Tolerant Systems*, pages 235–242, December 1997.

[19] U. Stern and D. L. Dill. Parallelizing the Murϕ verifier. In Orna Grumberg, editor, *Computer Aided Verification, 9th International Conference*, volume 1254 of *LNCS*, pages 256–67. Springer-Verlag, June 1997. Haifa, Israel, June 22-25.