ABSTRACT
We examine the distributed breadth-first enumeration of a state space that is partitioned using a static partition function. Two of the key drawbacks of this approach are the high communication overhead and the excessive growth of the queues that hold states received from other nodes, ultimately resulting in memory exhaustion that causes premature termination of the distributed enumeration algorithm. This paper investigates a number of strategies to reduce communication times and queue lengths. Some aspects of the strategies are novel, including a local search technique, while others are standard techniques for reducing memory usage and time for both serial and parallel state enumeration algorithms. The queue reduction strategies effectively reduce the queue lengths and save memory, while the standard techniques outperform the other strategies in their effectiveness to reduce communication overhead.

Categories and Subject Descriptors
D.2.4 [Software Engineering]: Software/Program Verification—Model checking; C.2.4 [Computer-Communication Networks]: Distributed Systems—Distributed applications

General Terms
Formal Methods, Distributed computation

Keywords
Distributed model checking, Breadth-first search

1. INTRODUCTION
Model checking is both memory- and time-intensive. Distributing the work over a cluster of networked machines is an attractive proposition, but poses a significant challenge: the high cost of communication between different nodes of the cluster often offsets any gains in the execution time. Furthermore, a significant portion of the available memory may have to be devoted to communication queues that store states waiting to be processed.

In this paper we focus on explicit-state, on-the-fly state enumeration by breadth-first search (BFS) instead of full-blown model checking of general properties. (Even when considering only safety properties, additional complications — such as the problem of recognizing recurring states and premature termination — may arise; we ignore these issues for now.) The state space is partitioned among the nodes of the cluster, and a function maps each state to its owner — the node that is responsible for its exploration. For a particular node \( i \), a state \( s \) is called a local state if \( \text{owner}(s) = i \), and is called a foreign state if \( \text{owner}(s) \neq i \).

We investigate four techniques that address the communication cost and queue problems: (A) reorganization of the communication queues; (B) local search of several generations; (C) compressing communicated states to improve throughput; and (D) buffering messages to avoid network latency. As far as we know, only technique B is entirely novel. The other approaches have certainly been invented and reinvented by many others, but, unfortunately, it is almost impossible to determine their provenance with accuracy.

1.1 Related Work
It is important to mention two approaches that we do not address directly: locality of states, and load balancing. The first approach, locality of states, refers to techniques that reduce the number of cross-transitions, i.e., the states that are sent between nodes. This can also be characterized as maximizing the number of instances where a state is handled by the same node that handles its parents.

Sometimes only a part of the state contents is used for the calculation of the state owner, as in the algorithm developed by Lerda and Sisto [20], and several partitioning strategies implemented for distributed model checkers for timed automata [3, 4]. This is based on the observation that the effect of many transitions is limited to a small part of the state, such as the local variables of a particular process. Since each process is responsible for only a fraction of the transitions in the system, cross-transitions are few if they are defined as only the transitions in one of the fractions.

Barnat et al. [2] employ a partition strategy where all states in the same maximal strongly-connected component are assigned to the same node. In another approach, Espada et al. approximate the state space using abstract interpreta-
tion so that states of the same abstract cycle are assigned to the same node, also resulting in fewer cross-transitions. In the distributed timed model checker Zeus, communication is reduced by a push/poll scheme where a node requesting a foreign state will receive, in addition to the foreign state, all regions it needs from the state's owner [5, 6]. It will, in turn, send all regions that the same remote node requires.

The second approach, load balancing, aims to ensure that all the nodes are busy all the time. Not only do we waste computational resources if some nodes are waiting idly while other nodes are frequently busy, but a load imbalance may mean that the memory of one node is exhausted while other nodes have plenty of free memory.

If nodes are explored on their owner nodes, the state partitioning function has a major influence on load balance. Since there is often a trade-off between locality and load balance, some partitioning strategies ignore cross-transitions completely and rather aim for an evenly distributed state space. In the implementation of the Murë verifier, for example, states are allocated to nodes using a randomized partition function — in other words, a hash function — that assigns an approximately equal number of states to each node [22].

Such predetermined partitioning functions are called static. Of course, even when we use a static partitioning function that distributes the state space evenly among the nodes, nodes could still be idle while waiting for busy nodes to reach states in their partitions, because the order in which states are visited depends on variable factors such as the structure of the state graph and network latency. Dynamic partitioning functions, in contrast to static partitioning functions, try to balance the load by assigning states to nodes on-the-fly, or to move them from one node to another. In some cases this process is carried out continuously, while in other cases the states are repartitioned at regular intervals or at certain synchronization points [1, 14, 21].

Algorithms for load balancing can also be based on queue lengths rather than on the state partitions. When nodes process their states at a slower rate than that at which they arrive, the queues (which store either all new arrivals or only unexplored states) may grow very long and exhaust the available memory.

In the distributed UPPAAL algorithm, states that belong to such a "slow" node whose queues are growing too long, are redirected to other nodes [3]. This strategy, however, may cause the same state to be explored and stored on more than one node. In Kumar and Mercer’s algorithm, arriving states are first checked against the store, so that the queues store only unexplored states. The nodes send the lengths of their queues to each of their closest neighbours at iterative steps [19]; and if, during the exchange with a neighbour, a node finds that the neighbour has a shorter queue it sends some of its states directly to that queue. States are therefore stored on their owners, but might be explored on another node.

1.2 Paper Outline

The basic framework of our distributed state enumeration algorithm is described in Section 2. The four sections that follow deal with queue-reduction techniques (Section 3), a local search technique that, in effect, "steals" work from other nodes (Section 4), a technique based on state compression (Section 5), and message buffering (Section 6). An experimental analysis is presented in Section 7, and we conclude with a discussion of the work in Section 8.

2. DISTRIBUTED BFS ALGORITHM

As mentioned in the introduction, our framework implements what is essentially a breadth-first search of a state graph that is generated on-the-fly. It forms the basis of an automata-theoretic LTL model checker, but here we focus only on state generation and the associated communication and queue problems, and we omit the full cycle-detection algorithm.

Each state of the state graph is a fixed-length vector of values that represent the program counters and local and global variables of a set of concurrent processes. The underlying modelling language, ESML, is similar to Promela [17], but employs only rendezvous communication, and therefore the state vector does not need to store any information about interprocess message queues. (Further details of ESML and our models are discussed in Section 7.)

We assume that the cluster is fully connected, so that any node can send messages to any other. We also assume that no messages are ever lost, although the order of messages in transit are not necessarily preserved. The nodes communicate in a peer-to-peer fashion. A small fraction of the messages exchanged relates to initialization and termination, but the majority of messages simply contain states that are sent from node to node. Every node runs two threads, a worker and a receiver. The worker removes states one-by-one from the worker queue and, for each state, generates its successors and sends them to the receivers. The receiver accepts states from the workers in the cluster, and places them in the receiver queue. States are added to the tail of both queues and removed from the head of the worker queue. Once the worker queue has been exhausted, it is refilled from the receiver queue by swapping the queues around. When all of the queues are empty and no more states are in transit, the system is ready to terminate; this is detected using a slight variant of Dijkstra’s token-passing algorithm [9]. In other words, this is a relatively straightforward distributed BFS algorithm, but, since states may be received out of order, the state space is not necessarily explored in pure breadth-first order.

The state graph is statically partitioned among the N nodes of the cluster by means of a function owner(s) = h(s) mod N, that maps state s to a node number 0…N−1. Here h(·) is an approximately uniform hash function. On node i, a state s is called local if owner(s) = i, and foreign if owner(s) ≠ i.

Each node maintains two important data structures: a state store where the local states of the node are stored permanently, and a send cache where the foreign states that are sent to other nodes are stored temporarily. Both structures are implemented as hash tables. Once a state has been inserted in the state store, it is never removed again. (Although technically possible, we ignore this scenario here because it is not relevant to our measurements.) The size of the send cache, on the other hand, is bounded from above. Once this bound is reached, old states are randomly replaced by new ones.

3. TECHNIQUE A: QUEUE MANAGEMENT
Although the framework architecture may appear simple, it already raises a number of questions. The receiver and worker threads share a common address space, so how is simultaneous access to the data structures governed? Who is responsible for adding states to the store? Exactly when are new states added to the store? We start with the following naive queueing strategy:

**Q**: The receiver appends incoming states to its queue. The worker removes one state at a time from its queue and tries to insert it in the store. If already present, the state is ignored. Otherwise, the state is stored immediately, and explored. Its local children that are not present in the store are added to the worker queue; foreign children are immediately sent to the appropriate foreign receiver.

This approach presents a serious memory bottleneck: the receiver queues grow long and a significant portion of the memory of each node is devoted to storing the queue contents. One problem is that the receiver queue may contain many duplicate entries, and that the same states may also be present (more than once) in the worker queue. A natural way to attack this is the store-aware queueing strategy:

**QS**: The receiver checks each incoming state against the store. New states are only appended to the receiver queue if they are not already present in the store.

Another way to accomplish the same goal is the limit queueing strategy:

**QL**: The receiver queue is limited to contain at most \( L \) states, initialized to a user-defined value \( L_0 \). When a new state arrives, the receiver appends it to its queue, unless the queue is full. In the latter case, the entire queue is scanned, and any state already in the store, is removed from the queue. The limit is adjusted: \( L := f \times L \), where \( f \) is a user-defined growth factor. (We use \( f = 1.1 \) in our implementation.) When the receiver and worker queues are swapped, the limit is reset: \( L := L_0 \).

It makes sense to combine the QL strategy (that purges the receiver queue of store-present states) with the QS strategy: an incoming state may be present in the worker queue but not yet in the store. It will be added to the receiver queue but may be purged after the worker has processed it. We refer to the combined strategy as QSL. A variation on QL is the extended limit queueing strategy:

**QE**: Similar to QL, except that after each scan of the receiver queue, the last state is marked. In the next scan, the states in front of the marked state are not scanned again. So, only “fresh” states are candidates for removal, even though some of the older states may now appear in the store.

As in the case of QL, the QE strategy can be combined with QS; we refer to the result as QSE. All of the strategies described so far influence the behaviour of the receiver thread, but the worker thread is not affected. None of the strategies modify the store, and therefore they do not require synchronized access.

The last approach we consider is the handle queueing strategy. It is similar to an idea described by David et al. [7], and affects the behaviour of both the worker and receiver:

**QH**: Before a state is added to either the receiver or worker queue, it is first inserted in the store. The queues do not store the actual state vectors, but merely pointers to the position of the states in the store. Such a pointer uniquely identifies a state, and can be used throughout the node as a handle.

Under the QH strategy, both the receiver and worker can modify the store contents, and consequently access to the store needs to be synchronized with some mutual exclusion mechanism. In our implementation we use the built-in POSIX lightweight thread mutex mechanism.

There are many other possibilities we have not mentioned. For example, the worker may insert the local successors of a state in the receiver queue, instead of appending them to its own worker queue. However, in terms of memory consumption it does not really matter where a particular state is queued, and because of limited space we are not able to explore all ideas.

4. TECHNIQUE B: LOCAL SEARCH

An optimal organization of the communication queues is important, but in some sense it is only addressing a symptom of the problem, and not the underlying cause. If the queues grow long, it is not because the workers are slow to process their queues, but for exactly the opposite reason: state generation is efficient and the workers generate many states quickly which they then send off to other nodes. It would be much better if the workers performed more work locally and communicated less. This is the same issue addressed by improving the locality of states through more sophisticated static or dynamic partitioning.

Local search offers another solution to this problem. Instead of exploring only one generation of successors as depicted in Figure 1(a), a worker is allowed to explore several generations. The exact number is a user-defined constant \( \text{depth} \), as shown in Figure 1(b). When \( \text{depth} = 1 \), the two algorithms perform identically.

During this “deeper” exploration, local search encounters both local and foreign states, as before. The important difference is that the foreign states are not sent to their owners, but explored locally. This is, in effect, a mild form of dynamic repartitioning. Communication is reduced because the locally-explored foreign states are never transmitted across the network.

Local search is illustrated in Figure 2. On the left is the original state graph, where every state is numbered and labeled by its owner, either node \( A \) or \( B \). The next three columns show how state exploration is distributed between \( A \) and \( B \) for depths 1, 2, and 3. Vertical arrows are used when a child state is generated and horizontal arrows when a state is sent to another node. Dashed horizontal arrows indicate that a state is sent from one node to the other, while solid horizontal arrows are used for states that remain with their owner.

When \( \text{depth} = 2 \), state 2 is explored by node \( A \), even though it properly belongs to node \( B \). Thus the state is investigated without any communication overhead. The case of \( \text{depth} = 3 \) highlights both a further advantage and a disadvantage of local search. Only one state is sent between the nodes (as opposed to two states for \( \text{depth} = 1, 2 \)) this is the intended side effect of local search. Moreover, the ra-
WORKER()
1 while not workingQueue.isEmpty():
2 s ← workingQueue.removeFirst()
3 EXPLORECHILDREN(s)

EXPLORECHILDREN(s)
4 if store.contains(s): return
5 store.insert(s)
6 for s’ in s.children():
7 send s’ to owner(s’)

LSWORKER()
1 while not workingQueue.isEmpty():
2 s ← workingQueue.removeFirst()
3 LSEXPLORECHILDREN(s, depth)

LSEXPLORECHILDREN(s, d)
4 if store.contains(s): return
5 store.insert(s)
6 for s’ in s.children():
7 if d > 1: LSEXPLORECHILDREN(s’, d - 1)
8 else: send s’ to owner(s’)

Figure 1: Distributed state exploration (a) without and (b) with local search

<table>
<thead>
<tr>
<th>Node A</th>
<th>Node B</th>
<th>Node A</th>
<th>Node B</th>
<th>Node A</th>
<th>Node B</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1_A</td>
<td>1_A</td>
<td>1_A</td>
<td>1_A</td>
<td>1_A</td>
</tr>
<tr>
<td>2</td>
<td>2_B</td>
<td>2_B</td>
<td>3</td>
<td>3_B</td>
<td>2_B</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>3_B</td>
<td>4</td>
<td>4_B</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td>3</td>
<td>3_B</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td>4</td>
<td>4_B</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>5_A</td>
<td>5_A</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>depth = 1</th>
<th>depth = 2</th>
<th>depth = 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>in store</td>
<td>in store</td>
<td>in store</td>
</tr>
</tbody>
</table>

Figure 2: Local search can be both advantageous and disadvantageous: for the state graph on the left, states 1, 2, and 3 will be explored redundantly when depth = 3, but when depth = 2, state 2 is never sent to its owner.
tio between explored states and transmitted states has been increased. On the other hand, states 1, 2, and 3 are now explored twice: once on A and once on B.

For small values of depth, the improvement in communication is not significant. When the value of depth is large, we might expect an increase in the amount of redundant work. There is also the danger that nodes may have to idly wait for others when the computation time is too large. In practice, there is a trade-off between too small and too large values for depth; the size and nature of the model plays an important role in the optimal depth of the local search.

The correctness of state enumeration with local search is guaranteed if all states are reached. This is so, because the only difference between exploring a foreign state and a local state locally, is that the foreign state is not placed in the store, which may lead to redundant visits. Checking of safety properties and deadlocks remain unchanged from the case in which local search is not used.

4.1 The Cache/Store/Ignore Option

Figure 3 depicts a local search that reaches local states $L$ and $L'$, and foreign states $F$ and $F'$. States $L$ and $F$ are generated when depth is reached and the local search ends. States $L$ and $F$ are generated before depth is reached and can be handled as follows: $L$ states can be

1. immediately placed in the worker queue and not explored further (for the moment),
2. explored but not marked in the store,
3. explored and marked, or
4. explored and cached in a special cache (the search cache) dedicated to local searches

$F$ states, on the other hand, can be

5. explored and cached in the sent cache or
6. explored and cached in the search cache.

It is difficult to predict the difference between implementing option 1 and ignoring it. Figure 4(a) depicts a scenario where it is better to implement 1, and place option 1 and ignoring it. Figure 4(b) depicts a scenario where it is better to continue exploring $L$ states instead of implementing 1. To remain consistent with the algorithm pseudo-code, depth counts down towards 1 in both cases. Furthermore, all local states are labeled with an $L$ and all foreign states with an $F$.

The state graph in Figure 4(a) is explored with depth = 2. If option 1 is implemented and state 2 is reached, it is queued. Then when state 2 is removed from the queue, a new local search is started in which state 3 is explored locally. If state 2 was explored immediately, state 3 would have been sent to its owner node.

The example in Figure 4(b) shows how the saving of send operations can result in more redundant work. The graph is explored with depth = 3 and when option 1 is not implemented, five foreign states are sent to their owning nodes. With option 1, on the other hand, these foreign states are explored locally in local searches that start from the three local states. The use of option 1 forces all local states found during a local search to start their own searches and this increases the number of foreign states explored locally.

Options 2 and 3 — whether local states are marked or not — are more significant. The advantage of marking states is clearly that it may avoid future redundant work (should the marked state be revisited), even though the marking may waste time and space. Not marking states has exactly the opposite effect: it may save time and space, but it makes revisits more expensive.

Options 1, 2, and 3 only apply to local states, because foreign states are stored in their owners' stores and worker queues. However, both local and foreign states can be cached. Local states are cached in a special cache for local searches (called the search cache, see option 4) so that they do not fill up the sent cache, which may increase communication (fewer duplicate sends are avoided when the sent cache is full). Caching local states in this second cache can save memory if the local state is encountered again before it is replaced in the cache. When replaced, a record of the state will not exist.

Foreign states can be cached in either cache. If in the cache for sent states (option 5), duplicate sends are avoided should the same foreign state be encountered in a local search and also as a “final” state. On the other hand, caching foreign states in the search cache (option 6) frees up space in the first cache for only sent states, because it is possible that many foreign states are only encountered during local searches and never as “final” states.

5. TECHNIQUE C: STATE COMPACTON

The cost of relaying a state across the network is influenced by many factors, including the physical bandwidth, network contention, and operating system overhead. One way of avoiding, if not reducing, these costs is to simply transmit fewer bytes per state.

General-purpose compression algorithms are not widely used to compress the relatively short states that occur in model checking. There are alternative ways of reducing the size of individual states [16, 18, 23]. We employ state compaction [11], which is both simple and efficient. Our modelling language ESML allows for upper and lower bounds on variables in a natural way, and bounds on program counters are easy to calculate. State vectors are then encoded as multiple-radix numbers, where each component of the state vector acts as a single digit. In this way the state generator can allocate the minimum number of bits needed to represent each state. Because the technique is runtime efficient, it is used throughout the system: in the store, caches, and queues, and thus it reduces not only the communication time but also the memory requirements of the entire system. Moreover, state compaction is entirely decoupled from the behaviour of the receiver and the worker, and from any higher-level algorithms we choose to implement.

6. TECHNIQUE D: BUFFERED COMMUNICATION

Another facet of communication cost is latency — the overhead associated with each message, regardless of its size. This can be addressed by transmitting more states per message, as suggested by Stern and Dill [22]. Each node maintains one outgoing buffer for each of the other nodes. Foreign
Figure 3: Handling local and foreign states

Figure 4: Depiction of two small state graphs during a local search: in (a) depth = 2, and states 3 and 5 are “final” and sent to their owners, while in (b) depth = 3, and 5 foreign states and 1 local states are “final”. However, when option 1 is used, only state 5 is “final” in (a), and no states are “final” in (b).

Table 1: Models used in the experiments

<table>
<thead>
<tr>
<th>Model</th>
<th>Description</th>
<th>n</th>
<th>States (millions)</th>
<th>Mean branching factor</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADn</td>
<td>Concurrent adding puzzle with values ≤ 100n</td>
<td>8</td>
<td>13.5</td>
<td>1.5</td>
</tr>
<tr>
<td>BAn</td>
<td>Bakery mutual exclusion algorithm for n processes</td>
<td>7</td>
<td>29.0</td>
<td>3.5</td>
</tr>
<tr>
<td>DnP</td>
<td>Dining philosophers problem for n philosophers</td>
<td>15</td>
<td>14.3</td>
<td>10.0</td>
</tr>
<tr>
<td>ELn</td>
<td>Elevator controller for n floors</td>
<td>3</td>
<td>7.7</td>
<td>7.2</td>
</tr>
<tr>
<td>FLn</td>
<td>Fischer’s mutual exclusion algorithm for n processes</td>
<td>6</td>
<td>2.9</td>
<td>4.3</td>
</tr>
<tr>
<td>HAn</td>
<td>Towers of Hanoi puzzle for n discs</td>
<td>15</td>
<td>14.3</td>
<td>3.0</td>
</tr>
<tr>
<td>LFn</td>
<td>Leader election based on filters for n processes</td>
<td>7</td>
<td>23.6</td>
<td>3.5</td>
</tr>
<tr>
<td>LPn</td>
<td>Lamport’s mutual exclusion algorithm for n processes</td>
<td>7</td>
<td>38.7</td>
<td>4.2</td>
</tr>
<tr>
<td>SZn</td>
<td>Szymanski’s mutual exclusion algorithm for n processes</td>
<td>4</td>
<td>2.3</td>
<td>3.7</td>
</tr>
</tbody>
</table>
states destined for a particular node are not sent immediately, but placed in the corresponding buffer. The size of the buffers is a user-defined constant \( B \). As soon as \( B \) states have accumulated in a buffer, they are transmitted together in a single message. To avoid potential deadlocks, a node also flushes its buffers (i.e., transmits their contents) when it becomes idle (i.e., when its receiver and worker queues become empty).

A drawback of this scheme is that states may have to be copied in and out of buffers, but this is ameliorated somewhat through the use of state compaction. Furthermore, each message must now include information about the number of states it contains, but this overhead is negligible, and the implementation is only slightly more complicated than for one-state messages. Little additional memory is required for the buffers, unless \( B \) is extremely large.

### 7. EXPERIMENTAL ANALYSIS

The experiments that follow were conducted on a Beowulf cluster with nine machines connected via a T1 network. Each machine has 1GB of memory and two Intel 3.06GHz Xeon processors. All of the results were obtained by assigning one “virtual” computational node to one physical machine.

The parameterized verification models used for the experiments are shown in Table 1. For each model, the table includes a specific parameter value and the corresponding number of states and the mean branching factor (i.e., number of transitions per state). The models are written in ESML [12], a strongly-typed modelling language inspired by Dijkstra’s Guarded Command Language [8], CSP [15], and Joyce [13]. We use ESML because it was specifically designed for modelling complex data structures (before such features were available in Promela). All communication in ESML is synchronous and unbuffered.

The framework described in Section 2 is implemented in C and uses Unix sockets for inter-node communication. The receiver and worker are both implemented with lightweight POSIX threads and uses the associated wait-and-signal mechanisms for coordinating access to shared data structures.

Running times are measured in two ways:

- **CPU time**: the total processor time used by the framework, and
- **Real time**: the elapsed time from the start of a run to when it finishes.

Both these times are important, and either one may be greater than the other. The CPU time reflects the actual work performed during a run, while the real time includes the communication cost (including network delays), the operating system overhead, and so forth. Of course, it also includes the processor times, but all these contributing costs may overlap. It is a bad sign if the real time is greater than the CPU time; this means that nodes are frequently idle, and that communication costs dominate the running time. For a cluster of \( N \) nodes, ideal speedup is obtained when \( \text{CPU time} = N \times \text{real time} \).

#### 7.1 Queue Management

The effect of the queueing strategies on the queue memory consumption and CPU times is shown in Table 2 for three models, LP6, BA5, and AD8. Memory and time consumption are shown in kilobytes (2^{31} bytes) and seconds, respectively, in the QN column; the other columns contain ratios with respect to this baseline. The initial limit value \( L_0 \) (for QL, QE, QSL, and QSE) is determined experimentally, which admittedly is not often possible in practice. We use \( L_0 = 100,000 \) for LP6 and BA5, and \( L_0 = 80,000 \) for AD8.

All of the strategies reduce the queue memory requirements, but — unsurprisingly — QH is the clear winner. When it comes to reducing the number of states in the queues (not shown in the table), QH also performs best: it reduces the queue lengths by 28% and 33% for LP6 and BA5, respectively. For these first two models, time consumption is not dramatically reduced: from less than 2% (QH/LP6) to about 17% (QN/BA5), and savings do not point to a particular strategy.

In the case of AD8 the impact of all of the strategies is noticeably weaker. Less memory is saved, and the running times are clearly much longer. We believe that this is due to AD8’s small mean branching factor of 1.5, compared to 3.6 for LP6 and 3.4 for BA5. Less branching leads to fewer cross-transitions and shorter queues. Models like AD8 with short queues require little reduction. The QH strategy is still effective (in this case, reducing the queue memory requirements by 94.3%), but proportionally less memory is needed for storing the queues in the first place, and the runtime penalty that the reduction strategies incur is high.

#### 7.2 Local Search

Table 3 presents two measures of the performance of local search for depths 1, 3, 5, and 7:

- The first measure is the real time: this is shown in seconds in the \( \text{depth} = 1 \) column, and as ratios, relative to this time, in the other columns.
- The second measure is the redundant work factor: the number of explored states at a particular depth, relative to the number of explored states at \( \text{depth} = 1 \). It appears in columns labeled “rwf”.

In most cases, the real time decreases for \( \text{depth} = 3 \), but then increases again for greater depths. There are two exceptions to this trend: LP6 and HA15. These results can be explained by looking at the ratio of the real time to CPU
Table 4: The table depicts that in instances where nodes are idle frequently, local search contributes to faster speedups in running times (seconds).

<table>
<thead>
<tr>
<th>Model</th>
<th>real time/CPU time</th>
<th>real time speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>LF5</td>
<td>31.50</td>
<td>8.00</td>
</tr>
<tr>
<td>HA15</td>
<td>55.07</td>
<td>4.41</td>
</tr>
<tr>
<td>BA5</td>
<td>7.72</td>
<td>1.49</td>
</tr>
<tr>
<td>EL3</td>
<td>5.83</td>
<td>1.46</td>
</tr>
<tr>
<td>FI6</td>
<td>1.19</td>
<td>0.67</td>
</tr>
</tbody>
</table>

time at $depth = 1$, as shown in Table 4. The last column of this table shows the real time when $depth = 1$ relative to the fastest real time measured for any depth. We can interpret the real time/CPU time ratio as an indicator of how idle the workers are when local search is not in use. If the ratio is large, local search has a greater chance of reducing the real times: rather than waiting for communication, workers can carry on with exploring new states. When the ratio is small, there is less scope for such an improvement. As the local search depth grows, so does the CPU time and the number of states that are explored unnecessarily. This redundancy only refers to those states that are explored on many different nodes. There is very little that can be done about this phenomenon. At very large depths, the extra CPU time consumed by redundant work will outweigh the time saved by not sending foreign states to their rightful owners. This means, for a particular model, there is a point beyond which the redundant-work/communication-reduction tradeoff no longer produces any improvement. Of course, it is practically impossible to determine this point without considerable effort.

Another factor behind the increase in CPU times is the repetition of visiting the same foreign state on the same node. As mentioned in Section 4.1, the state cache can help to combat this problem, but it can be difficult to determine the optimal size of such a cache. It must clearly be large enough to accommodate as many foreign states as possible. For example, in the case of the FI6 model, memory is exhausted and state exploration fails to complete if the cache is too small; the receiver queues contain too many duplicate states. Since redundant work increases the CPU times, the local search store options that will be the most efficient are those that eliminate as many duplicate states as possible. The options we look at are:

1. local states are stored while foreign states are cached with the sent states, i.e., in the sent cache,
2. local states are queued while foreign states are cached in the sent cache,
3. local states are not stored while foreign states are cached in the sent cache,
4. local states are stored while foreign states are cached in a cache separate from the sent states, i.e., in the search cache,
5. local states are stored while foreign states are not stored, and
6. local states and foreign states are not stored.

The model LP6 is examined for options 1 - 4 in the top half of Table 5 where for each depth the real times and redundant work factors are shown. Option 4 is repeated three times. In 4a both the sent cache and search cache can hold 2 million states, in 4b the sent cache holds 1.5 million and the search cache 0.5 million, and in 4c vice versa.

For all of the options the time and redundant work grow rapidly, but the instances where the largest number of redundancies and repetitions are avoided — options 1, 2, and 4c — times are the fastest. Since option 4c depicts improvement behaviour over 4a and 4b, having foreign states in one cache helps reduce more state redundancies.

Local searches with options 5 or 6 increase the redundant work considerably. To examine this, a smaller model is selected, BA4, consisting of only 150 000 states. Table 5 illustrates clearly how in each case the redundant work is vastly increased with increasing depth and the real times are affected as a result. When BA4 is run with option 1, 11.7 times the original state space is explored while the redundant work factor in option 6 climbs to 1922.7!

7.3 State Compaction

Table 6 shows the impact of state compaction — in combination with local search — for two models. The numbers in the table are the real times in seconds, except for the lines labeled ratio which gives the ratio of the time with state compaction to the time without. The state sizes of BA5 and HA15 are reduced from 196 and 320 bytes to 16 and 36, respectively. Without exception, state compaction improved the running time of all of our models.

Unfortunately, state compaction and local search do not appear to work well together. The case of HA15 provides one explanation for this. For $depth = 1$, the ratio of the real time to the CPU time is 55.0 without state compaction, and 0.77 with state compaction. In the former case, communication costs dominate, and nodes are idle most of the time. In this scenario, we can increase the local search depth to take advantage of this. On the other hand, with state compaction the communication costs are reduced to such an extent, that it does not make much sense to employ local search.

7.4 Buffered Communication

Table 7 depicts the real times with and without buffering of messages for the naive (QN) and handle (QH) queueing strategies. All of these experiments also make use of state compaction. The columns labeled “msgs” give the number of messages sent (in millions), and the columns labeled “B” gives those buffer sizes that produced the smallest real times.

It is clear that buffered communication can reduce the running time in many cases, but it is also true that the improvement is not overwhelming. In the case of HA15 the deterioration in the real time for QN, is partly explained by the small queue lengths of this model. Consequently, the buffers are never filled completely, and for all practical purposes buffered communication is not fully in effect.

8. CONCLUSIONS

We have presented four techniques to reduce either the communication times or the lengths of receiver queues. The queue-reduction strategies are generally successful in reducing the queue lengths and thus also the queue memory re-
Table 3: The influence of local search on real times (seconds) and the redundant work factor (rwf)

<table>
<thead>
<tr>
<th>Model</th>
<th>depth = 1</th>
<th>depth = 3</th>
<th>depth = 5</th>
<th>depth = 7</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>time $= t_0$</td>
<td>r wf</td>
<td>time/$t_0$</td>
<td>r wf</td>
</tr>
<tr>
<td>F16</td>
<td>195.67</td>
<td>1.00</td>
<td>0.69</td>
<td>3.63</td>
</tr>
<tr>
<td>LF5</td>
<td>200.00</td>
<td>1.00</td>
<td>0.43</td>
<td>3.22</td>
</tr>
<tr>
<td>AD6</td>
<td>297.62</td>
<td>1.00</td>
<td>0.46</td>
<td>1.29</td>
</tr>
<tr>
<td>EL3</td>
<td>311.68</td>
<td>1.00</td>
<td>0.66</td>
<td>3.29</td>
</tr>
<tr>
<td>LP6</td>
<td>355.65</td>
<td>1.00</td>
<td>0.76</td>
<td>3.50</td>
</tr>
<tr>
<td>BA5</td>
<td>381.12</td>
<td>1.00</td>
<td>0.67</td>
<td>2.83</td>
</tr>
<tr>
<td>HA15</td>
<td>9834.61</td>
<td>1.00</td>
<td>0.48</td>
<td>2.88</td>
</tr>
</tbody>
</table>

Table 5: The effect of local search strategies on real times (seconds) and the redundant work factor (rwf)

<table>
<thead>
<tr>
<th>Model</th>
<th>Strategy</th>
<th>depth = 3</th>
<th>depth = 5</th>
<th>depth = 7</th>
<th>depth = 9</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>time $= t_0$</td>
<td>r wf</td>
<td>time/$t_0$</td>
<td>r wf</td>
</tr>
<tr>
<td>LP6</td>
<td>1</td>
<td>270.50</td>
<td>3.50</td>
<td>415.62</td>
<td>7.56</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>292.90</td>
<td>4.02</td>
<td>542.32</td>
<td>9.16</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>322.57</td>
<td>4.65</td>
<td>772.63</td>
<td>13.48</td>
</tr>
<tr>
<td></td>
<td>4a</td>
<td>459.63</td>
<td>4.10</td>
<td>949.33</td>
<td>10.64</td>
</tr>
<tr>
<td></td>
<td>4b</td>
<td>434.03</td>
<td>5.00</td>
<td>884.37</td>
<td>13.23</td>
</tr>
<tr>
<td></td>
<td>4c</td>
<td>481.79</td>
<td>4.08</td>
<td>927.81</td>
<td>8.92</td>
</tr>
<tr>
<td>BA4</td>
<td>5</td>
<td>9.33</td>
<td>3.86</td>
<td>8.46</td>
<td>17.03</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>14.37</td>
<td>8.12</td>
<td>12.09</td>
<td>51.17</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>153.78</td>
<td>50.46</td>
<td>121.45</td>
<td>223.24</td>
</tr>
</tbody>
</table>

Table 6: Real times (seconds) with and without state compaction for various local search depths

<table>
<thead>
<tr>
<th>Model</th>
<th>depth = 1</th>
<th>depth = 3</th>
<th>depth = 5</th>
<th>depth = 7</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>time</td>
<td>r wf</td>
<td>time/$t_0$</td>
<td>r wf</td>
</tr>
<tr>
<td>BA5</td>
<td>381.12</td>
<td>0.11</td>
<td>88.98</td>
<td>0.51</td>
</tr>
<tr>
<td>BA5+SC</td>
<td>41.51</td>
<td>0.35</td>
<td>119.99</td>
<td>0.51</td>
</tr>
<tr>
<td>HA15</td>
<td>9834.61</td>
<td>0.01</td>
<td>168.56</td>
<td>0.06</td>
</tr>
<tr>
<td>HA15+SC</td>
<td>130.31</td>
<td>0.04</td>
<td>191.65</td>
<td>0.08</td>
</tr>
</tbody>
</table>

Table 7: The effect of buffered communication on real times (seconds) and the number of messages sent (millions) in combination with naive queueing strategy (QN) and the handle queueing strategy (QH)

<table>
<thead>
<tr>
<th>Model</th>
<th>No BC</th>
<th>QN</th>
<th>QH</th>
<th>BC+QN</th>
<th>BC+QH</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>time</td>
<td>time</td>
<td>msgs</td>
<td>time</td>
<td>time</td>
</tr>
<tr>
<td>LF7</td>
<td>196.24</td>
<td>142.36</td>
<td>51.41</td>
<td>60.42</td>
<td>51.41</td>
</tr>
<tr>
<td>BA7</td>
<td>218.44</td>
<td>172.02</td>
<td>59.49</td>
<td>60.42</td>
<td>59.49</td>
</tr>
<tr>
<td>LP7</td>
<td>363.68</td>
<td>272.95</td>
<td>99.49</td>
<td>100.29</td>
<td>99.49</td>
</tr>
<tr>
<td>HA15</td>
<td>153.78</td>
<td>79.60</td>
<td>27.29</td>
<td>27.29</td>
<td>27.29</td>
</tr>
</tbody>
</table>
requirements, with the handle queueing strategy QH outperforming the others. The CPU times are unaffected in the main, except in the case of models that might not require the use of these strategies in the first place. However, our communication-reduction strategy, local search, does not work as well as we expected. The problem is that local search increases the amount of redundant work so much, that it negates the time that it saves on communication.

The use of buffered communication and especially state compaction is surprisingly successful. Neither technique incurs much overhead, but nevertheless the resultant savings are significant, and the number of messages transmitted is reduced. In both cases, little extra strain is put on the processors while fewer bytes and messages are sent over the network.

9. REFERENCES


