Parallelizing the Murφ Verifier*

Ulrich Stern and David L. Dill

Computer Science Department
Stanford University
Stanford, CA 94305

{uli, dill}@cs.stanford.edu

Abstract

With the use of state and memory reduction techniques in verification by explicit state enumeration, runtime becomes a major limiting factor. We describe a parallel version of the explicit state enumeration verifier Murφ for distributed memory multiprocessors and networks of workstations using the message passing paradigm. In experiments with three complex cache coherence protocols, parallel Murφ shows close to linear speedups, which are largely insensitive to communication latency and bandwidth. There is some slowdown with increasing communication overhead, for which a simple yet relatively accurate approximation formula is given. Techniques to reduce overhead and required bandwidth and to allow heterogeneity and dynamically changing load in the parallel machine are discussed, which we expect will allow good speedups when using conventional networks of workstations.

1 Introduction

Complex protocols are often verified by examining all reachable protocol states from a set of possible start states. This reachability analysis can be done using two different methods: the states can be explicitly enumerated by storing them individually in a table, or a symbolic method can be used, such as representing the reachable state space with a binary decision diagram (BDD) [3]. Both methods have application domains in which they outperform the other; explicit state enumeration has worked better for the types of industrial protocols examined in our group [11].

There have been two approaches to improve explicit state enumeration. First, state reduction methods have been developed that aim at reducing the size of the reachability graph while ensuring that protocol errors will still be detected. Examples are exploiting symmetries, utilizing reversible rules, and employing repetition constructors [13]. These methods directly tackle the main problem in reachability analysis – the very large number of reachable states of most protocols. The second approach aims at reducing the amount of memory needed to perform the reachability analysis. Examples are bitstate hashing [9], hash compaction [28, 22], and using magnetic disk instead of main memory to store the state space [21].

*A preliminary version of this paper was published in the 9th International Conference on Computer Aided Verification, 1997 [23].

In this paper, we explore a third approach to improve explicit state enumeration: parallel processing. With the use of state and memory reduction techniques, runtime becomes a major limiting factor [28, 22]. For example, when verifying complex protocols with the Murφ verifier [7] using symmetry reduction in combination with hash compaction, a single verification run that does not expose new errors typically takes several days. Norris Ip [13] even reported a runtime of 13.9 hours for only 38 269 states when using symmetry reduction in combination with reversible rules and repetition constructor reductions.

We describe a parallel version of the Murφ verifier for distributed memory multiprocessors and networks of workstations using the message passing paradigm. Parallel Murφ was originally developed on a network of workstations (NOW) at UC Berkeley (SPARC20s connected via Myrinet) using generic active messages [6] as the message passing layer; later it was ported with little effort to an SP2 at IBM Watson.

In parallel Murφ, the state table, which stores all reached protocol states, is partitioned over the nodes of the parallel machine. Thus, the table can be larger than on a single node. Each node maintains a work queue of unexplored states. When a node generates a new state, the "owning" node for this state is calculated with a hash function and the state is sent to this node; this policy implements randomized load balancing. Upon receipt of a state descriptor, a node first checks if the state has been reached before (with the local part of the state table). If the state is new, it is inserted in the state table and the local work queue. Special algorithms for termination detection and error trace generation have to be employed in this distributed setting. We also show analytically that the state space is typically very evenly distributed over the nodes.

We measured the speedup of parallel Murφ when verifying three complex cache coherence protocols: SCI [12], DASH [17], and FLASH [16]. On a 63-node SP2 the speedup was 44.2 for SCI and 53.7 for DASH, while we obtained speedups of 26.6 for SCI, 27.8 for DASH, and 29.4 for FLASH on a 32-node NOW in Berkeley. Thus, the algorithm achieves close to linear speedup. In addition, experiments performed at Berkeley [18] show that the runtime of parallel Murφ is largely insensitive to increased communication latency and reduced bandwidth. There is some sensitivity to communication overhead, however. We give a simple formula for the expected runtime on a parallel machine as a function of the communication overhead. We show empirically that the formula accurately predicts parallel speedup.

Aggarwal, Alonso, and Courcoubetis [1] also presented a distributed reachability algorithm. Their algorithm seems more complicated than ours and has not been implemented. In addition, the correctness of their termination detection relies on timing assumptions that may be difficult to guarantee. One potential advantage of their method is that it might be usable under dynamically changing load conditions on a network of heterogeneous workstations. We propose an extension of our algorithm, however, that also allows heterogeneous systems with dynamically changing load and, at the same time, reduces the communication volume typically by one or two orders of magnitude.

Kumar and Vemuri [15] proposed and implemented a distributed algorithm to check the equivalence of two finite state machines by essentially performing a reachability analysis of the product machine. Their algorithm synchronizes after each breadth-first level and does not overlap communication and computation. Although

the examples they present require only infrequent communication with very small messages, the reported speedups are worse than the ones reported here. In addition, their algorithm seems to have a high overhead, since it is only faster than a sequential one when running on four nodes. In contrast, parallel Murφ running on one node is as fast as the most recent version (3.0) of sequential Murφ, for which the runtime was optimized. In fact, parallel Murφ is based on this version of sequential Murφ, which contains symmetry reduction and hash compaction.

There have also been some efforts to parallelize BDD-based verification methods. Stornetta and Brewer [25, 24] and Ranjan et al. [19] have presented distributed memory BDD algorithms. Both algorithms only achieve speedups in comparison to sequential versions if the sequential versions run out of memory and are forced to do swapping, but they enable the use of the total memory of the parallel machine. Kimura and Clarke [14] presented BDD algorithms for a shared memory machine and reported a speedup of roughly 10 on 15 nodes, while efficiently using the total available memory.

This paper is organized as follows. Section 2 provides background on active messages. The algorithm for parallel explicit state enumeration is described in detail in Section 3. Results running the algorithm are reported in Section 4. Sections 5 and 6 show how the speedup of the algorithm can be predicted and how the algorithm can be tailored towards a conventional network of workstations. Finally, Section 7 gives some concluding remarks.

2 Active Messages

Active messages [26] have been introduced to reduce the communication costs in message passing, and can be thought of as a fast message passing library. In contrast to a message in conventional message passing, an active message also contains the address of a procedure, called *handler*, that will be called on the destination node after the arrival of the message. For example, when a state descriptor s is to be sent to some node i in parallel Murφ, the active message "RECEIVE(s)" will be sent to i, indicating that the handler RECEIVE() should be called on node i with the state descriptor s as argument.

When sending an active message, the sender does not wait for the message to arrive at the receiver but continues immediately. Upon arrival of the message, the receiver's current stream of control is not interrupted. Instead, the receiver has to periodically call poll(), which, in turn, calls all handlers for the active messages that have arrived since the last call to poll(). We will say that a message is *received* after the corresponding handler has returned. The use of handlers and polling enables efficient implementations of the active message scheme.

All nodes (asynchronously) execute the same program when using active messages. (A newer version of active messages, however, allows communication between differing programs.) Each node is assigned a unique node number from $\{0, \ldots, N-1\}$, where N denotes the number of nodes in the parallel machine. To implement a "master" node with special responsibilities such as startstate generation, an if statement can be used with the condition that the node number is 0. The barrier() command synchronizes all the nodes running the parallel program by waiting until every node has reached a barrier.

3 Parallel Explicit State Enumeration

We now give the basic algorithm for parallel explicit state enumeration, followed by the algorithms for termination detection and error trace generation. Finally, we analyze the employed randomized load balancing scheme.

3.1 The Basic Algorithm

The basic algorithm that runs on each node of the parallel machine is given in Figure 1 and described in this paragraph. Note that the global variables are local to each node, i.e., each node maintains its own local state table T, for example. The state enumeration is started by calling SEARCH() on each node. The master node generates the startstates and distributes them by calling SEND(). In the SEND() procedure, the state is first canonicalized (for symmetry reduction) and then sent to the owning node, whose node number is calculated by a hash function $h()$. The handler RECEIVE() checks an incoming state against the local state table and potentially inserts it into the state table and queue. The search loop dequeues a state, generates its successors, and sends them to the owning nodes. Note that this loop calls poll() to execute the handlers for newly arrived messages. The search loop is exited as soon as termination is detected. We now look at the termination detection algorithm in more detail.

3.2 Termination Detection

The parallel search has terminated when the following two conditions hold: there are no more messages in progress (i.e., sent but not received), and there are no more states in the queues Q. (Here and in the remainder of the paper, "message" refers to a message holding a state, not to a control message used, say, in the termination detection algorithm.) Note that the condition that there are no more states in the queues Q also implies that no state is currently being expanded, since states being expanded are removed from the queue only after their expansion.

Figure 2 shows the termination detection algorithm used. When the master has been idle for longer than a certain threshold value, it queries each node about the numbers of messages it has sent and received (counted in the variables *Sent* and *Received*) and about the number of states in its queue. The master then sums the answers to these queries, yielding

$$\sum_{i=0}^{N-1} Sent_i - Received_i + |Q_i| \ , \tag{1}$$

where i denotes the node number.

Immediately before a node answers the query, it disallows the sending of messages by setting *StopSend* to true. Thus, if $\sum_{i=0}^{N-1} Sent_i - Received_i = 0$, the master obtained a snapshot of the system in which all sent messages had been received. If, in addition, $\sum_{i=0}^{N-1} |Q_i| = 0$, then no more messages can be generated after the snapshot was taken. Hence, the parallel search has terminated if the sum (1) equals zero.

While this termination detection algorithm is easy both to implement and prove correct, it has the disadvantage that the state enumeration is stopped (by disallowing

```
var      // global variables, but local to each node
  T: hash table;      // state table
  Q: FIFO queue;      // state queue
  StopSend: boolean;      // for termination detection
  Work, Sent, Received: integer;

SEARCH()      // main routine
begin
  T := ∅; Q := ∅;      // initialization
  StopSend := false; Sent := 0; Received := 0;
  barrier();
  if I am the master then      // master generates startstates
    for each startstate s_0 do
      SEND(s_0);
    end
  do      // search loop
    if Q ≠ ∅ then begin
      s := top(Q);
      for all s' ∈ successors(s) do
        SEND(s');
      end
      Q := Q − {s};
    end
    poll();
  while not TERMINATED();
end

SEND(s: state)      // send state s to "random" node h(s)
begin
  s_c := canonicalize(s);      // symmetry reduction
  while StopSend do      // wait for StopSend = false
    poll();      // (for termination detection)
  end
  Sent++;
  send active message RECEIVE(s_c) to node h(s_c);
end

RECEIVE(s: state)      // receive state (active message handler)
begin
  Received++;
  if s ∉ T then begin
    insert s in T;
    insert s in Q;
  end
end
```

Figure 1: Parallel Explicit State Enumeration

```
TERMINATED(): boolean
begin
  if I am the master then      // master initiates termination check
    if idletime exceeds threshold then begin
      Work := 0;
      send active message REPORTCOUNTERS() to all nodes;
      wait for all replies (i.e., calls to SUMCOUNTERS());
      if Work > 0 then      // continue search
        send active message CONTINUE() to all nodes;
      else      // terminate search
        notify all nodes of termination (details omitted);
    end
  return termination status;
end

// active message handlers
REPORTCOUNTERS()      // report counter values
begin
  StopSend := true;
  send active message SUMCOUNTERS(Sent − Received + |Q|) to master;
end
SUMCOUNTERS(w: integer)      // master sums counter values
begin
  Work := Work + w;
end
CONTINUE()      // continue with search
begin
  StopSend := false;
end
```

Figure 2: Termination Detection

the sending of messages) when nodes are queried. The resulting runtime overhead of the termination detection algorithm, however, can be made negligibly small by setting the threshold value for the master's idle time to, say, 0.1s, which is roughly the time to generate 100 states on a single node when verifying complex protocols. Then, the algorithm will rarely be invoked if the search has not yet terminated, but will still be invoked soon after termination has occurred.

3.3 Error Trace Generation

The information required for error trace generation is stored in a file. More precisely, each node writes a record to a local file for each state it finds to be new after checking the (local) state table. The record written consists of the new state's compressed value and of the node number and record position in the file of the new state's predecessor.

When a protocol error is discovered by some node, a leader election algorithm is started among all the nodes that have just discovered an error. Only the error found by the elected leader will be reported, and the elected leader is responsible for gathering the information to generate the error trace from the distributed files.

3.4 Randomized Load Balancing

We now examine how well the hash function $h()$ balances the state space over the individual state tables. Consider a particular node and assume that for each state the probability that it is sent to this node is $1/N$, i.e., that the hash function distributes states uniformly. (Universal hashing [4], which is used in Murφ, distributes at least as well as uniformly.) Let n denote the number of reachable states and Y the random variable describing the number of states sent to our node, which has the expected value $\bar{Y} = n/N$. For large n, Y is distributed according to a normal distribution, by the central limit theorem. To bound the probability that the relative error of Y compared to $\bar{Y}$ is larger than a certain constant r, we use the fact that for every $x > 0$, $1 - \Phi(x) < \phi(x)/x$, where $\Phi(x)$ and $\phi(x) = e^{-x^2/2}/\sqrt{2\pi}$ denote the standard normal distribution and density functions [8]. Using basic calculations one obtains

$$\Pr(|Y/\bar{Y} - 1| > r) < 2\,\phi(z)/z \ , \quad \text{where } z = r\,\sqrt{n/(N-1)} \ .$$

For example, when n=10^8 and N=32, the probability that the relative error exceeds r=0.1% is smaller than 8.85%, and the probability that the relative error exceeds r=0.5% is smaller than $2.73 \cdot 10^{-19}$. Generally, if the number of reachable states n is large and the number of nodes N is not too large, the state space will be distributed very evenly over the nodes.

4 Results

Figures 3 and 4 show the measured speedup of parallel Murφ on a 63-node SP2 at IBM Watson and on a 32-node UltraSPARC/Myrinet NOW at Berkeley, for instances of the SCI, DASH, and FLASH protocols. Some parameters of these instances are shown in Table 1. The protocols were scaled to both provide interesting

Table 1: Example protocols

protocol	reachable states	successors generated	bytes/ state	diameter	single-node runtime NOW	single-node runtime SP2
SCI	1179 942	2973 536	124	46	717s	2804s
DASH	254 937	2646 647	532	64	1287s	5204s
FLASH	1021 464	4556 496	136	45	2477s	n/a

data and yet prevent the process of running the examples from becoming too time-consuming. The speedup graphs illustrate that the Murφ verifier can be parallelized quite efficiently.

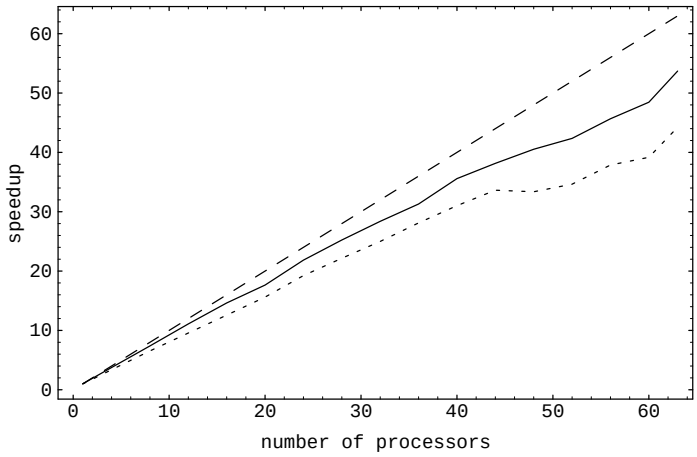

Figure 3: Speedups for the SCI (dotted) and DASH (solid) protocols, calculated from the average runtime over two runs on an SP2, in comparison to linear speedup (dashed)

5 Estimating the Speedup

Rich Martin et al. [18] have performed a study on the Berkeley NOW of the impact of communication performance on several parallel applications including Murφ. They characterized communication performance based on the LogGP model [5, 2] with four parameters: latency L, overhead o, gap g, and time-per-byte G. The latency is the delay in communicating a small message, the overhead is the average time consumed in sending or receiving a message, the gap is the minimum time between two messages, i.e., the reciprocal message bandwidth, and the time-per-byte is the reciprocal bulk transfer bandwidth. For example, when a small message is sent from one node to another, the total time for this message is at least $2o + L$, which corresponds to the time consumed in the send and receive routines and the communication delay.

For their study, Rich Martin et al. modified the communication layer of the NOW so that each of the four parameters could be slowed down independently, starting from the following values of the unmodified communication layer: o=3.5μs, g=7.0μs, L=5.5μs, and G=38MB/s. Parallel Murφ, when verifying the SCI exam-

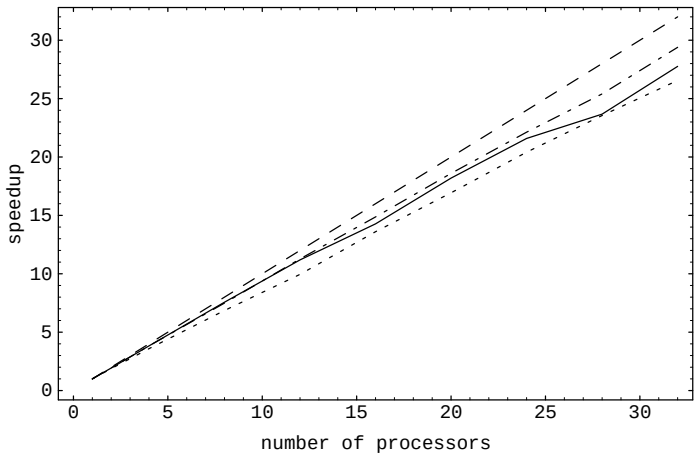

Figure 4: Speedups for the SCI (dotted), DASH (solid), and FLASH (dashed and dotted) protocols, calculated from the average runtime over five runs on the Berkeley NOW, in comparison to linear speedup (dashed)

ple, demonstrated only negligible slowdown when either increasing latency or gap by up to 100μs or when reducing the bulk transfer bandwidth to 1MB/s. The insensitivity to latency can be explained by observing that if there are enough states in the state queues, all latency is overlapped with computation. Parallel Murφ is not sensitive to increased gap since it does not send messages in bursts. Finally, in this example the bandwidth requirement per node is smaller than 1MB/s (roughly 0.5MB/s).

The runtime of parallel Murφ did, however, demonstrate some dependency on the overhead. We now derive an approximation formula for the runtime as a function of the overhead. Assume that each node sends m/N messages, where m denotes the total number of messages sent. With high probability, a fraction of close to $(N-1)/N$ of these messages will be sent to nodes other than the sender, each resulting in an overhead of $4o$, which stems from the sending and receiving of the message and its (automatically generated) reply message. Assuming linear speedup if there were no overhead, we approximate the runtime t_N on N nodes as

$$t_N = 4o\, m\, (N-1)/N^2 + t_1/N \ , \tag{2}$$

where t_1 denotes the runtime on a single node. Table 2 shows that (2) quite accurately predicts the measured runtimes for a range of different overhead values and numbers of nodes. Note that the numbers of messages sent are (slightly) smaller than the number of generated successors ($2.974 \cdot 10^6$), which is due to a small cache of recently sent states. Also note that the effect of this cache becomes negligible when the number of nodes increases.

6 Improvements of the Basic Algorithm

6.1 Message Aggregation

By packing several states into one message, one can reduce the overhead per state. This well-known technique basically trades excess parallelism for communication

Table 2: Measured ($t_{N,\mathrm{m}}$) and predicted ($t_{N,\mathrm{p}}$) runtimes for the SCI protocol (in seconds) when varying the overhead. Measurements are averaged over five runs.

	messages	added overhead [μs]									
	sent m	0		25		50		100		200	
N	[million]	$t_{N,\mathrm{m}}$	$t_{N,\mathrm{p}}$	$t_{N,\mathrm{m}}$	$t_{N,\mathrm{p}}$	$t_{N,\mathrm{m}}$	$t_{N,\mathrm{p}}$	$t_{N,\mathrm{m}}$	$t_{N,\mathrm{p}}$	$t_{N,\mathrm{m}}$	$t_{N,\mathrm{p}}$
1	1.983	705.8									
2	2.534	372.2	361.8	432.1	425.1	496.1	488.5	617.4	615.2	891.7	868.6
16	2.697	52.7	46.3	66.5	62.1	81.6	77.9	114.1	109.5	177.2	172.7
32	2.706	26.2	23.2	34.6	31.4	43.1	39.6	59.4	56.0	92.8	88.7

performance. As shown in Figure 5, each of our three sample protocols provides a high degree of parallelism as measured by the number of states in each level of a breadth-first search, which is what enabled the efficient parallelization in the first place.

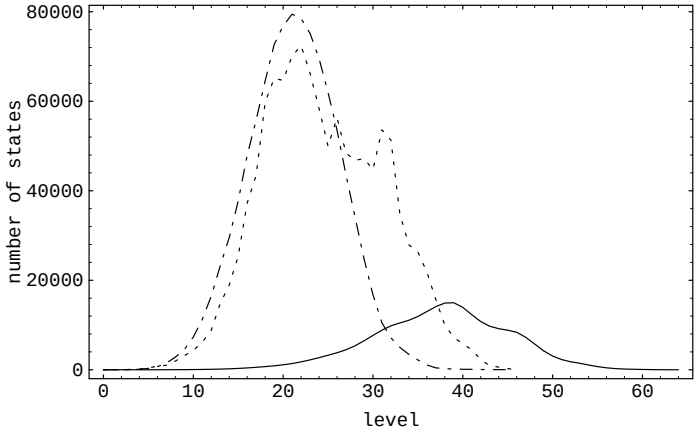

Figure 5: Number of states in each breadth-first level for the SCI (dotted), DASH (solid), and FLASH (dashed and dotted) protocols

Table 3 shows the effect of message aggregation on runtime t_N and on the number m of messages sent for the unmodified NOW and with an additional overhead of 100μs. An overhead of 100μs is typical for message passing libraries based on TCP/IP. In both cases, the number of messages sent is strongly reduced by message aggregation. The implemented scheme packs 10 states into each message given that there are more than 20 states in the local queue. Optimizing these values or even making the message size vary with the number of states in the queue has not been tried. While the reduction in runtime is small in the case of the unmodified NOW (as expected), in the high overhead case approximately a factor of two reduction is achieved, approaching the runtime on the unmodified NOW.

6.2 Aiming for a Conventional NOW

The bandwidth requirement of parallel Murφ becomes a problem on a conventional NOW, such as a collection of heterogeneous workstations connected via Ethernet. For example, when verifying the DASH protocol, each node requires a bandwidth

Table 3: Measured ($t_{N,\mathrm{m}}$) and predicted ($t_{N,\mathrm{p}}$) runtimes and messages sent m (in million) for the SCI protocol when using message aggregation in comparison to the basic scheme. Measurements are averaged over six runs.

	added overhead [μs]									
	0					100				
	basic scheme		aggregation			basic scheme		aggregation		
N	$t_{N,\mathrm{m}}$	m	$t_{N,\mathrm{m}}$	m	$t_{N,\mathrm{p}}$	$t_{N,\mathrm{m}}$	m	$t_{N,\mathrm{m}}$	m	$t_{N,\mathrm{p}}$
16	52.3s	2.692	47.9s	.334	43.9s	114.1s	2.692	54.8s	.315	51.3s
32	25.7s	2.708	25.9s	.400	22.0s	58.9s	2.703	29.8s	.350	26.2s

of roughly 0.5MB/s, which would make an implementation even on top of switched Ethernet (where each node gets 10Mb/s for itself) difficult. In addition, the algorithm presented here works optimally only if for each node the state table size is proportional to the node's speed. This restriction allows only limited heterogeneity. Also, randomized load balancing performs poorly if the load on the nodes changes dynamically, since the hash function is fixed.

The algorithm can, however, be adapted to the situation of a slow network, heterogeneity, and dynamically changing load. Instead of sending a full state s, each node only sends a (hash) signature $c(s)$ to the node $h(s)$, which, in turn, returns a bit indicating whether the state had been reached before, but does not expand the state. (Note that expanding would be impossible in any case, since only signatures are sent.) Similarly as in [22], it can be shown that this scheme will typically enable a reduction in the bandwidth requirement of one or two orders of magnitude at the cost of a small probability that the verifier incorrectly claims that an erroneous protocol is correct. Note that in the new scheme each node generates its work (new states to explore) by itself, which has the effect that a fast node "needing" much work will also generate much work for itself. Thus, the scheme is well suited for situations of dynamically changing load. To provide each node with initial work and to "restart" a node that runs out of states, a load balancing protocol similar to the one described in [27] can be employed. In addition, heterogeneous systems are allowed since the tabulation of states is now independent of their expansion.

7 Conclusion

Runtime has been becoming a major bottleneck in verification by explicit state enumeration. This paper demonstrates that the explicit state verifier Murφ can be parallelized quite efficiently. The resulting algorithm is shown to run with close to linear speedup on a wide range of distributed memory multiprocessors and networks of workstations. In addition, a formula is derived that quite accurately predicts the speedup of parallel Murφ as a function of the communication performance. Since the state table is partitioned over the parallel machine, the algorithm also allows the verification of larger protocols.

The methods used to parallelize Murφ could also be used for other explicit state verification tools like SPIN [10]. The architectures for which parallel Murφ was developed – distributed memory multiprocessors and networks of workstations – are becoming more common. Techniques to reduce overhead and required bandwidth

and to allow heterogeneity and dynamically changing load in the parallel machine are discussed. They are expected to allow good speedups when using conventional networks of workstations.

The algorithm presented is compatible with the two newer state reduction techniques in Murφ [13], reversible rules and repetition constructors, which were not yet available in the public release. It is also compatible with the latest version of hash compaction [22]; details about this combination can be found in [20].

The most recent version of sequential Murφ, on which the parallel version is based, does not support the checking of temporal properties because of difficulties combining this checking with symmetry reduction. Thus, we did not put high priority on parallelizing the verification of temporal properties. This seems to be an interesting area for future research, however.

Acknowledgments

We are grateful to the Berkeley active message group and all the people at IBM Watson that assisted with the SP2 port. In particular, Joefon Jann and Yarsun Hsu from IBM Watson helped in many ways. We would especially like to thank Rich Martin from Berkeley for making his modified communication layer available to us and for his invaluable assistance on active message issues.

This work was supported in part by the Defense Advanced Research Projects Agency through NASA contract NAG-2-891 and a scholarship to the first author from the German Academic Exchange Service (DAAD-Doktorandenstipendium HSP-II). The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency, NASA, or the US Government.

References

[1] S. Aggarwal, R. Alonso, and C. Courcoubetis. Distributed reachability analysis for protocol verification environments. In *Discrete Event Systems: Models and Applications. IIASA Conference*, pages 40–56, 1987.

[2] A. Alexandrov, M. F. Ionescu, K. E. Schauser, and C. Scheiman. LogGP: incorporating long messages into the LogP model – one step closer towards a realistic model for parallel computation. In *7th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 95–105, 1995.

[3] J. R. Burch, E. M. Clarke, K. L. McMillan, and D. L. Dill. Sequential circuit verification using symbolic model checking. In *27th ACM/IEEE Design Automation Conference*, pages 46–51, 1990.

[4] J. L. Carter and M. N. Wegman. Universal classes of hash functions. *Journal of Computer and System Sciences*, 18(2):143–54, 1979.

[5] D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. von Eicken. LogP: towards a realistic model of parallel computation. In *4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 1–12, 1993.

[6] D. Culler, K. Keeton, C. Krumbein, L. T. Liu, A. Mainwaring, R. Martin, S. Rodrigues, K. Wright, and C. Yoshikawa. *Generic Active Message Interface Specification.* UC Berkeley, 1995. Version 1.1.

[7] D. L. Dill. The Murφ verification system. In *Computer Aided Verification. 8th International Conference*, pages 390–3, 1996.

[8] W. Feller. *An Introduction to Probability Theory and Its Applications*, volume 1. John Wiley & Sons, 3rd edition, 1968.

[9] G. J. Holzmann. On limits and possibilities of automated protocol analysis. In *Protocol Specification, Testing, and Verification. 7th International Conference*, pages 339–44, 1987.

[10] G. J. Holzmann and D. Peled. The state of SPIN. In *Computer Aided Verification. 8th International Conference*, pages 385–9, 1996.

[11] A. J. Hu. *Techniques for Efficient Formal Verification Using Binary Decision Diagrams.* PhD thesis, Stanford University, 1995.

[12] *IEEE Std 1596-1992, IEEE Standard for Scalable Coherent Interface (SCI).*

[13] C. N. Ip. *State Reduction Methods for Automatic Formal Verification.* PhD thesis, Stanford University, 1996.

[14] S. Kimura and E. M. Clarke. A parallel algorithm for constructing binary decision diagrams. In *IEEE International Conference on Computer Design*, pages 220–3, 1990.

[15] N. Kumar and R. Vemuri. Finite state machine verification on MIMD machines. In *European Design Automation Conference*, pages 514–20, 1992.

[16] J. Kuskin, D. Ofelt, M. Heinrich, J. Heinlein, R. Simoni, K. Gharachorloo, J. Chapin, D. Nakahira, J. Baxter, M. Horowitz, A. Gupta, M. Rosenblum, and J. Hennessy. The Stanford FLASH multiprocessor. In *21st Annual International Symposium on Computer Architecture*, pages 302–13, 1994.

[17] D. Lenoski, J. Laudon, K. Gharachorloo, W.-D. Weber, A. Gupta, J. Hennessy, M. Horowitz, and M. S. Lam. The Stanford DASH multiprocessor. *Computer*, 25(3):63–79, 1992.

[18] R. P. Martin, A. M. Vahdat, D. E. Culler, and T. E. Anderson. Effects of communication latency, overhead, and bandwidth in a cluster architecture. In *24th Annual International Symposium on Computer Architecture*, 1997.

[19] R. K. Ranjan, J. V. Sanghavi, R. K. Brayton, and A. Sangiovanni-Vincentelli. Binary decision diagrams on network of workstations. In *International Conference on Computer Design*, pages 358–64, 1996.

[20] U. Stern. *Algorithmic Techniques in Verification by Explicit State Enumeration.* PhD thesis, Technical University of Munich, 1997.

[21] U. Stern and D. L. Dill. Using magnetic disk instead of main memory in the Murφ verifier. Submitted for publication.

[22] U. Stern and D. L. Dill. A new scheme for memory-efficient probabilistic verification. In *Joint International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols, and Protocol Specification, Testing, and Verification*, pages 333–48, 1996.

[23] U. Stern and D. L. Dill. Parallelizing the Murφ verifier. In *Computer Aided Verification. 9th International Conference*, pages 256–67, 1997.

[24] T. Stornetta. Implementation of an efficient parallel BDD package. Master's thesis, UC Santa Barbara, 1995.

[25] T. Stornetta and F. Brewer. Implementation of an efficient parallel BDD package. In *33rd Design Automation Conference*, pages 641–4, 1996.

[26] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active messages: a mechanism for integrated communication and computation. In *19th Annual International Symposium on Computer Architecture*, pages 256–66, 1992.

[27] C.-P. Wen. A distributed task queue for load balancing on the CM5. Unpublished paper written in Katherine Yelick's group at UC Berkeley.

[28] P. Wolper and D. Leroy. Reliable hashing without collision detection. In *Computer Aided Verification. 5th International Conference*, pages 59–70, 1993.