Available online at www.sciencedirect.com
SCIENCE DIRECT®
Electronic Notes in Theoretical Computer Science
ELSEVIER Electronic Notes in Theoretical Computer Science 128 (2005) 19–34
www.elsevier.com/locate/entcs

Load Balancing Parallel Explicit State Model Checking

Rahul Kumar [1] and Eric G. Mercer [2]

Verification and Validation Laboratory
Department of Computer Science
Brigham Young University
Provo, USA

Abstract

This paper first identifies some of the key concerns about the techniques and algorithms developed for parallel model checking; specifically, the inherent problem with load balancing and large queue sizes resultant in a static partition algorithm. This paper then presents a load balancing algorithm to improve the run time performance in distributed model checking, reduce maximum queue size, and reduce the number of states expanded before error discovery. The load balancing algorithm is based on generalized dimension exchange (GDE). This paper presents an empirical analysis of the GDE based load balancing algorithm on three different supercomputing architectures—distributed memory clusters, Networks of Workstations (NOW) and shared memory machines. The analysis shows increased speedup, lower maximum queue sizes and fewer total states explored before error discovery on each of the architectures. Finally, this paper presents a study of the communication overhead incurred by using the load balancing algorithm, which although significant, does not offset performance gains.

Keywords: model checking, parallel, distributed, load balancing, GDE, queue size, error detection, communication

1 Introduction

Explicit state model checking is a methodology to verify properties in a design through reachability analysis. The practical application of model checking, however, is hindered by the state explosion problem [5]. State explosion is a result of enumerating the state space of a concurrent system using interleaving

[1] Email: rahul@cs.byu.edu
[2] Email: egm@cs.byu.edu

doi:10.1016/j.entcs.2004.10.016

semantics where each concurrently enabled transition must be considered separately in any given state. Several techniques exist to address aspects of the state explosion problem. Symmetry and partial order reduction exploit structure and concurrency to reduce the number of states in the reachable state space that must be explored to complete the model checking problem [3][6]. Bit state hashing (supertrace) and hash compaction reduce the cost of storage states in the reachable state space [7][13]. All of these techniques enable the verification of larger problems, but in the end, are restricted to the number of states that can be stored on a single workstation. If the model checking algorithm exhausts resources on the workstation it is running on before completion of the verification problem, then the problem must be altered in some way to reduce the size of its reachable state space until it can fit into the available resources.

The goal of distributed model checking is to combine the memory and computational resources of several processors to enhance state generation and storing capacity. The seminal work in distributed model checking presented by Stern and Dill creates a static partition of the reachable state space during execution [14]. The workload observed as a function of time and communication overhead on each processor depends critically on how the states are partitioned between the verification processes. Several techniques such as caching (sibling and state), partial order reduction, symmetry reduction, different partition functions and dynamic partitioning have been explored in the past to reduce communication overhead and create perfect state distributions [11][10]. Even with the use of the above mentioned techniques, creating a perfect partition for any given problem while maintaining equally loaded processors requires *a priori* knowledge of the state space, which is the very problem we are trying to solve.

This paper presents an empirical study of the seminal static partition algorithm showing the level of load imbalance, regardless of the chosen static partition, that exists between the processes on different supercomputing platforms. The imbalance results in high idle times in several processors, as well as extremely large search queues. The high idle times indicate that many processors are not contributing to state enumeration, and the large search queues lead to premature termination by exhausting memory resources. Furthermore, the imbalance in the partition slows down error discovery since states leading to errors can be buried deep in the search queues. The paper further presents a load balancing algorithm based on generalized dimensional exchange(GDE) to mitigate idle time at the expense of additional communication overhead. Load balancing the state partition algorithm improves speedup in distributed model checking despite the increased communication. In addition, it reduces

maximum queue sizes by up to 10 times, and it reduces the number of states enumerated before error discovery on average. These effects are shown in empirical studies on three different supercomputing architectures: Network of Workstations(NOW), Clusters and Shared memory architectures.

2 Models and Platforms

Empirical analysis of this work is performed to better understand the performance of the static partitioning algorithm and other techniques using parallel Murφ [9]. Testing is done and results are gathered on the following platforms:

(i) IBM 1350 Linux Cluster which has 128 dual processor nodes with high speed interconnect;

(ii) Network of workstations (NOW) with 100 Mbps interconnect; and

(iii) Marylou10 IBM pSeries 690 64 Power4+ processors @ 1.7 GHz 64 GB total memory.

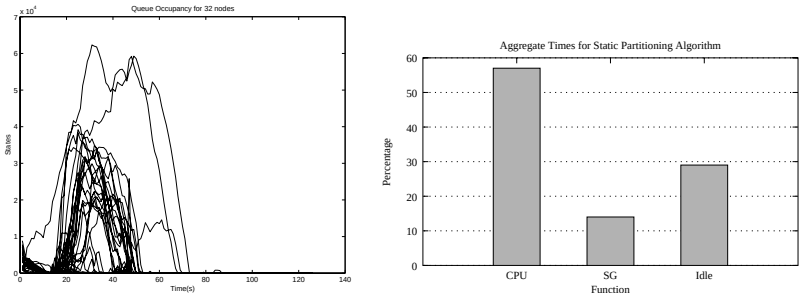
A major part of the testing is performed using the models located at the model database in [1]. These models have been selected because they provide a large, controllable, interesting and diverse set for testing. The selected models are not representative of all types of problems, but they effectively capture our general observations in studying the several problems.

3 Analysis of Distributed Model Checking Techniques

This section will first analyze the static partition algorithm and present the problems encountered when using it. After the analysis, we discuss two existing techniques to solve these problems. Our results indicate that these techniques are not very effective on the models and benchmarks used by us.

3.1 Queue Imbalance and Idle Time

The static partition algorithm enables the verification of larger models at a much faster rate using several processors. Our further investigation of the algorithm supports the conclusion that idle times during distributed verification are high, and there is a high imbalance in the distribution of states across the queues of the participating processors causing premature termination, degraded error discovery and large queue sizes. Figure 1(a) shows the sizes of the queues as sampled at one second intervals for all processors during the entire period of verification using 32 processors on the NOW architecture. The



(a) (b)

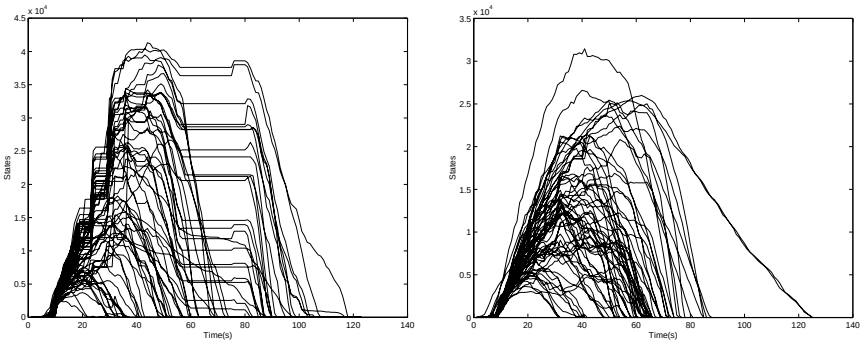
Fig. 1. This figure shows the high queue imbalance through time and time distribution for the major functions using a NOW architecture. (a) Queue sizes for the *jordon* model using 32 processors. (b) Aggregate times for all processors for major functions in verification showing a large idle time.

interval sampling is implemented using the `alarm` and `signal` utilities available on most `UNIX` based systems. The horizontal axis represents time and the vertical axis represents the number of states in the queue of a given processor. The figure shows the imbalance in queue sizes, with many processors having useful work to do only in the early part of the verification. During the latter part of the verification, only 20% of the processors are active. These results are consistent with recent work by Behrmann, [2] where the same imbalance is observed.

Figure 1(b) displays the aggregate percentage of time that has been spent on the major composite functions during verification for the atomix model. The major composite functions are CPU, State generation(SG), and Idling. CPU is time spent performing I/O operations and communication. State generation time is defined to be the time spent processing states from the queue and generating their successors. Idle time is the time spent waiting for new states to be received because the processors search queue is empty. In this state, the verification processor is still processing messages. The states in the messages are discarded however, because they have been previously added to the hash table. The figure demonstrates that after communication, idle time dominates. In fact, almost a third of the total aggregate time is spent idling. Similar results have been observed for other models of varying size and shape.

3.2 Partition Function

Idle time is a direct result of load imbalances. This is seen in experiments with various hash functions and their effect on speedup. We have studied the effects of the partition function on the distribution of states by implementing the partition function using various hash functions including several hash functions in



(a) (b)

Fig. 2. Queue imbalance for the atomix model using 64 processors and two different hash functions from Spin on the NOW architecture.

SPIN [8]. Figure 2(a) shows the queue distribution for 64 queues on the NOW architecture using the single bit forward hash function from SPIN. Although more processors remain active through the verification run when compared to Figure 1(a), a significant number still become idle and remain idle for over half the running time of the verification process. Figure 2(b) shows the same queue distribution using a different bit mask for the same hash function taken from SPIN. The distribution is extremely different even though all other parameters have been maintained for both experiments. Similar results have been noticed for other hash functions such as Jenkins forward hash function and dynamic partitioning algorithms used by us [10]. These results indicate that to create a perfect distribution of the states in the queues across all the processors and to then maintain that distribution through the entire verification process, an *a priori* knowledge of the state space is required. This is the very problem we are trying to solve in the first place.

3.3 State Cache

A technique to improve performance of distributed model checking is the use of state caching to reduce the number of messages and hash lookups. Previous work regarding state caching, to improve performance has been presented in [11][4]. Our analysis indicates the presence of duplicate states in the same message or in different messages being sent to other processors. This is due to the fact that many states in the state space of the model can be reached by different paths and from different states. To avoid this, a block of memory is allocated on each processor to function as a direct mapped state cache. Only states not present in the cache are forwarded to their owning processors.

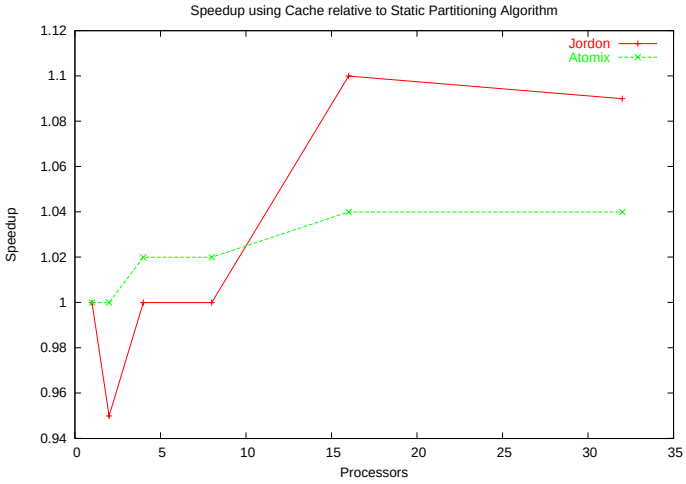


Fig. 3. Speedup relative to the static partition algorithm obtained using a cache of size 1 MB on IBM 1350 Cluster.

Further investigation of the state cache shows that the decrease in the number of messages is not significant once the cache size is over 1 KB in-spite of high occupancy rates. This is due to low transition locality in using a large number of processors in parallel verification. The average speedup that has been observed by using a cache is constant but not very high. Speedups in the region of 1.05 are gained relative to the static partitioning algorithm. Figure 3 shows the speedup achieved using a cache of 1 MB on the IBM 1350 Cluster. The speedup realized due to the cache is offset by time spent looking up states in the direct mapped state cache, as well as time spent inserting states into the cache. This results in the cache not providing a significant improvement in speedup, queue size or error discovery.

4 Methods

We introduce the concept of load balancing the queues to reduce aggregate idle time in distributed model checking. Work in [12] describes a global load balancing technique where all the processors involved in the state-space generation try to achieve a state of perfect equilibrium. In equilibrium, all processors have an equal number of states in the queue in an effort to remove the imbalance. Short periods of state generation are followed by a period where the processors balance the queues with each other by exchanging queue information and extra states in the queues. There are several drawbacks to this method. First, there is too much communication overhead introduced in the parallel model checking problem which itself is very communication intensive.

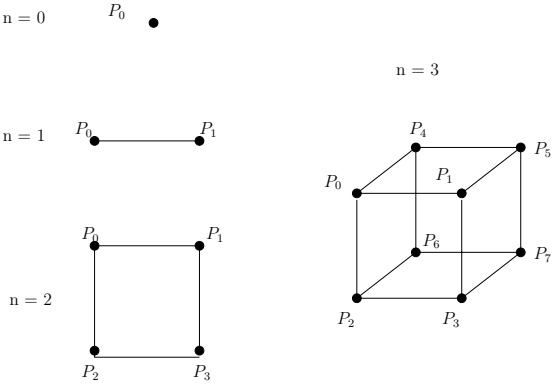


Fig. 4. Hypercube structure for 1 to 8 processors.

This causes the algorithm to not scale if the model or the number of processors is increased significantly. Second, the user/verifier is responsible for specifying the number of iterations after which a load balancing cycle should occur (frequency). If the value selected is very small (high frequency) then the communication overhead is extremely high and the effective speedup is very low. On the other hand, if the value selected is large (low frequency) then effective load balancing does not take place and the parallel verification process behaves in the same manner as the static partition algorithm. Another issue with the global load balancing algorithm is the amount of time incurred waiting for all the processors to synchronize for load balancing. Consider a scenario of 4 processors P_0, P_1, P_2 and P_3. Let us suppose that P_1, P_2 and P_3 are ready for load balancing and have already initiated the load balancing phase. If in this situation P_0 happens to be very heavily loaded inserting states into its queue, P_0 will be unable to enter the load balancing phase at the same time as the other processors, causing the other processors to wait until P_0 has finished processing states/messages and enter the load balancing phase.

We use Generalized Dimensional Exchange(GDE) for load balancing which performs iterative load balancing among the involved processors by having each processor equalize its workload with each dimensional neighbor [15][16]. Dimensional neighbors are calculated based on the topology selected for the processors. Our GDE methodology groups processors in the verification process into a hypercube like structure. Figure 4 shows the structure of a hypercube for 1 through 8 processors as well as the number of *dimensions* that are created in each case. The GDE algorithm setup is similar to the edge coloring problem where no two edges can be assigned the same color; thus, if processors in a hypercube structure are colored with the same rules as the

Algorithm: $GDEBalanceQueues()$
1: /* *Method called by BFS/DFS every* i *iterations on each process* */
2: **for** $AllDimensionalNeighbors$ **do**
3: $n := getNextDimensionalNeighborID()$
4: $q := getLocalQueueSize()$
5: $s := sendQueueSizes(n, q)$
6: return

Fig. 5. GDE load balancing algorithm from sending end.

edge coloring problem, there will be $\lceil log(N) \rceil$ colors that are needed, where N is the number of processors. Each color can then be thought of as a dimension with a subset of the processors belonging to a particular dimension. This network structure enables us to view the processors in an elegant manner and implement algorithms that are more efficient with respect to communication.

In the GDE load balancing scheme, each node balances the workload (number of states in the queue) with it's dimensional neighbors. In a network of 8 processors, processor P_0 balances with processors P_1, P_2 and P_4. During the balancing stage, each processor can choose to balance the workload completely so that each of the two queues have the same amount of work, or they can choose to balance to some other point. The *exchange parameter* is defined as the amount of workload to be shared between the two processors when performing a load balancing operation. Work in [16] proves that the optimal value for creating an equilibrium state in as few iterations as possible is $\frac{1}{2}$. Figures 5 and 6 show the pseudo code for the algorithm. Every i iterations where i is the balance frequency set by the user, the processor sends it's current queue size to it's dimensional neighbors. On the receiving side, once a processor receives a queue size from a dimensional neighbor, the processor executes the algorithm shown in Figure 6. If the workload on the local queue is higher than the workload on the neighbor, then states are sent to the neighbor. If on the other hand the workload is not higher, no action is taken and execution is returned to state generation and communication procedures. Receiving states from other processors happens implicitly and no blocking occurs. For our testing purposes we set the exchange parameter to be one half and the balance frequency to be $\frac{1}{1000}$ states. A balance frequency of $\frac{1}{1000}$ means that load balancing is initiated after 1000 states have been processed from a processor's queue, or if the processor's queue is empty. Since each state in the queue is added only after checking the hash table, transferring queue states directly to other processors queues does not affect the set of visited states; thus, not causing multiple copies of the same state on different processors.

Algorithm: $HandleMessage(M, ID)$
1: /* *If a queue size is received from ID then need to load balance* */
2: /* *Get queue size of neighbor from message M* */
3: $q_n = getQueueSize(M)$
4: $q_l := getLocalQueueSize()$
5: **if** $q_l > q_n$ **then**
6: $sendStates((q_l - q_n)/2, ID)$
7: **else**
8: /* *Receive happens in a non-blocking fashion* */
9: $receiveStates((q_n - q_l)/2, ID)$
10: return

Fig. 6. GDE load balancing algorithm on receiving end.

5 Results

Figure 7 shows the relative speedup of the GDE load balancing scheme relative to the static partitioning algorithm(without cache) for the atomix and jordon models using the IBM 1350 Linux Cluster. Apart from the higher speedup, we can also see that the speedup curves are moving up as the number of processors increases indicating that this algorithm scales to some degree. This is due to the efficient communication patterns created by the N-dimension hypercube of the network topology. Figure 8 shows the speedup of the GDE scheme on the NOW architecture. We can see that the speedup achieved is higher than the speedup achieved on the distributed memory cluster architecture of Figure 7. Figure 9 shows the speedup of the same models and scheme on a shared memory machine as described earlier. The speedup here seems to be less than the speedup achieved on the other two architectures. The general trend seems to be that the slower the interconnect between the processors, the more effective the GDE load balancing scheme; thus, for the shared memory architecture, where the interconnect is the fastest, we can see that the load balancing is the least effective, to the point where it is detrimental. However for the distributed memory architecture, the interconnect is faster than the NOW architecture, but slower than the shared memory architecture: thus, it provides the opportunity to the GDE scheme to increase the performance and efficiency. On the other hand, the NOW architecture provides the slowest interconnect and the best speedup results as seen in Figure 8. The observed behavior and speedup pattern is an area for future research and study.

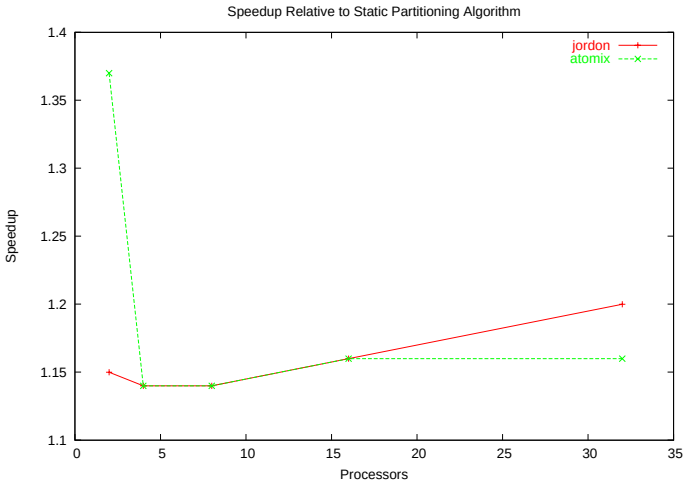


Fig. 7. Speedup for the GDE scheme relative to the static partitioning algorithm on the IBM 1350 Linux Cluster.

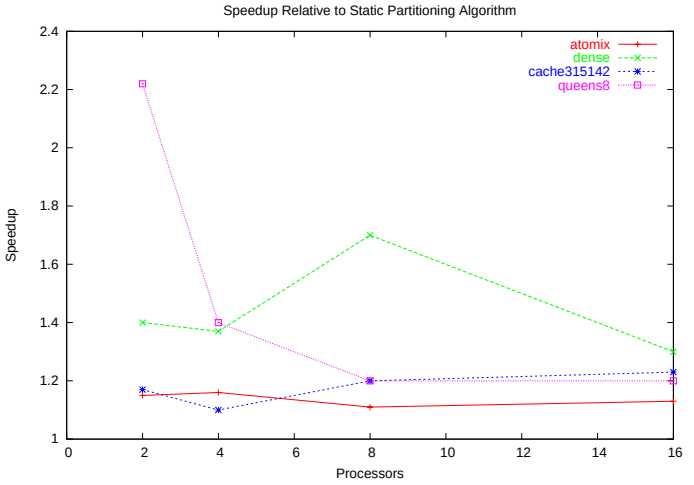


Fig. 8. Speedup for the GDE scheme relative to the static partitioning algorithm on NOW architecture.

Figure 10(a) shows the state of the queues every second after applying the GDE load balancing algorithm when using 32 processors on the IBM Cluster. Comparing this to Figure 1(a), a significant difference is seen in using the GDE load balancing algorithm. More processors have states to expand throughout the verification. Also, the number of snapshots taken in each case indicates that only half the time was needed to verify the model

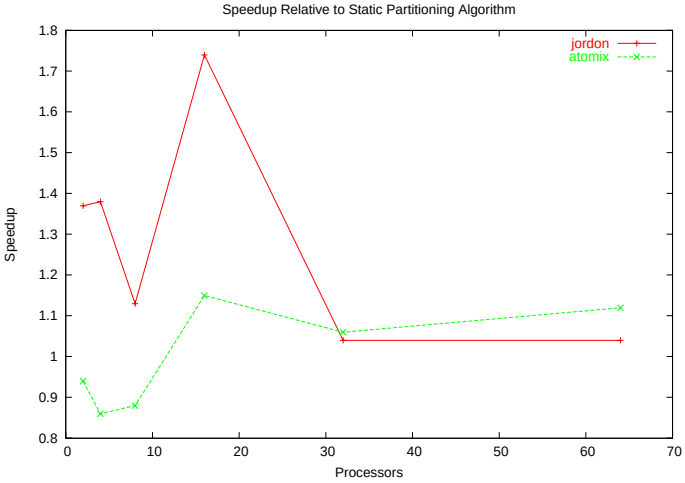
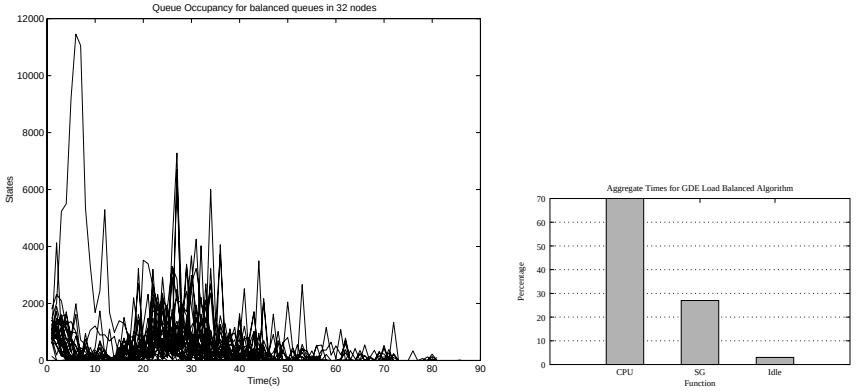


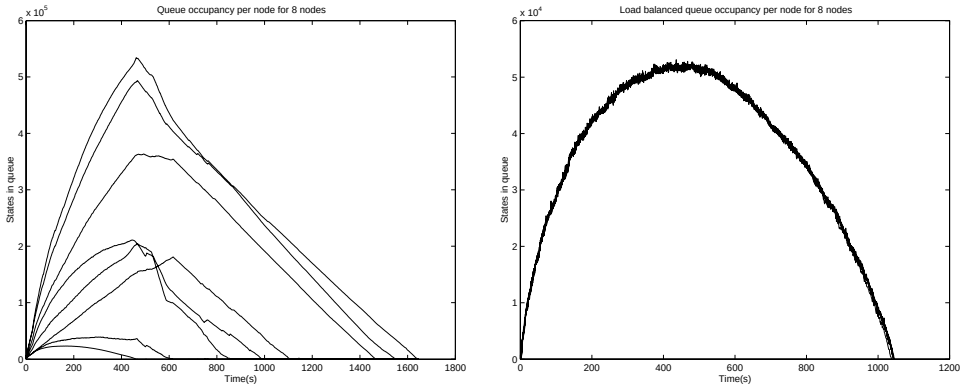
Fig. 9. Speedup for the GDE scheme relative to the static partitioning algorithm on the shared memory architecture.



(a) (b)

Fig. 10. This figure shows the queue imbalance and aggregate times for each function during model checking for the *jordon* model(a) Queue sizes for the jordon model using 32 processors after load balancing showing balanced queues (b)Aggregate times for each function after the GDE load balanced scheme was implemented showing low idle time.

completely. Figure 10(b) shows the percentage of the time spent on the major modules in the parallel model checker. Compared to Figure 1(b) we can see that due to the load balancing, the idle time is reduced from 35% of the total time to almost nothing and the CPU time has been increased due to extra communication to improve performance.



(a) (b)

Fig. 11. Balanced and unbalanced queues for the jordon model on the NOW architectures(a) Queue sizes for the jordon model using 8 processors before load balancing. (b) Queue sizes for the jordon model using 8 processors after load balancing showing highly balanced queues.

Figure 11(a) shows the state of the queues before load balancing and Figure 11(b) shows the state of the queues after load balancing on the NOW architecture. For this particular example a higher balancing frequency was used to achieve a better balance. As can be seen from the figure, using a higher frequency created a near perfect distribution of the states among the participating queues. A speedup of 1.6 was achieved in this case.

The search order of the parallel verification algorithm is modified significantly. Using the static partitioning algorithm, there is some amount of determinism involved since a particular state is guaranteed to be processed on a specific process and each error state or deadlock state is always discovered on the same processor. Since GDE shuffles the states in the queue from one process to another, a state that was originally in a particular queue can be transferred to another queue; thus, it causes error and deadlock states to be discovered on different processors and in a different order when compared to the static partitioning algorithm. The resultant effect is that on average, the shuffling of states causes error states and deadlock states to be discovered earlier than they would have been in the static partition algorithm. Also, if the error state happens to be in a very inconspicuous location, shuffling the states in the queues can help the parallel algorithm find the error earlier. Table 1 shows the results gathered by running the load balancing algorithm and the static partition algorithm on models that have errors using 8 processors. The number of states generated are reported for each algorithm before the error state is discovered. The ratio column is the ratio of the number of states saved in the static partition algorithm to the number of states saved using the GDE

Table 1
Average number of states generated before error state discovered

Model	Static Partition	GDE Algorithm	Ratio
queens8-deadlock	22546	500	45.09
adash1212	17650	8500	2.08
jordon	6600	3860	1.71
queens8-error	1505086	962343	1.56
arbiter4	4416	3300	1.34
sparse-shallow	31640626	25866803	1.22
atomix	2150688	1983550	1.08
two diamonds	2280376	2156913	1.06

algorithm. A clear example of a model containing an inconspicuous error is the *queens8* problem. The *queens8* problem involves placing 8 queens on a chess board in such a manner that no queen is threatening any other queen. The error state is successfully placing all the queens in the described manner. The static partition algorithm performs well compared to the serial algorithm, but the GDE load balanced algorithm outperforms both algorithms. Even in other models we can see that the load balanced algorithm outperforms the static partition algorithm by a significant factor with the worst case scenario of performing only slightly better than the static partition algorithm.

A major challenge with the static partitioning algorithm is the early termination of verification due to lack of space available in the queues of individual processors. In extremely large and dense models with high branching factors, each processed state generates a lot of successors that have to be saved in the queue. If a queue has not been allocated with enough memory to accommodate these states, then the verification process has to be discontinued and verification cannot complete. A high number of states in a queue also occurs due to the high load imbalance in the queues of the processors. A proactive effort is made to keep the queues in a state of equilibrium. The GDE load balancing algorithm, which implicitly causes the queues to be smaller and more manageable; thus, it reduces the strenuous memory requirements of the queue for each processor. Table 2 shows the maximum size of the queue for each algorithm for various models using 8 processors in the verification and the standard deviation within the maximum queue size on each processor for both the static partition and GDE load balanced algorithms. From the table we can see that there is a large difference between the static partition algorithm, and the GDE load balanced algorithm. For the load balanced algorithm, the maximum queue size is almost an order of magnitude smaller than the maximum queue size for the static partition algorithm, and the amount

Table 2
Maximum queue size and standard deviation of maximum queue size

Model	Max Queue Size		Ratio	Standard Deviation	
	Static Partition	GDE		Static Partition	GDE
dense	352226	30824	11.43	139137	575
jordon	511934	54387	9.41	183442	249
two diamonds	389792	48351	8.06	118165	149
sparse-shallow	377491	80416	4.69	117099	855
atomix	239938	62261	3.85	76428	319
queens8	1587876	418589	3.79	235335	2679
cache315142	289803	97492	2.97	99843	579

of memory used for the queue differs by an equal proportion. The standard deviation for the maximum queue sizes for each algorithm is shown in the last two columns of Table 2. For the static partition algorithm, we can see that the standard deviation is very large compared to the maximum queue size. In contrast, the GDE load balanced scheme provides much lower standard deviations indicating that the queues are in an equilibrium state.

Communication overhead is incurred with the use of GDE as a load balancing technique. Table 3 compares the average number of messages sent between any two processors in the verification using the static partition algorithm(with a state cache of 100k) and the optimized load balanced algorithm(with state cache of 100k). From the table we observe that for the models cache, atomix and dense the number of messages sent in the load balanced version is fewer than the messages exchanged in the static partition algorithm. This is due to the state cache. For the other cases the number of messages sent in the load balanced version is greater but within the same order of magnitude. The last column displays the difference as the percentage relative to the higher number of messages sent. We can see that in the worst case, only 10% more messages are sent to achieve a speedup of 1.6 times. This communication overhead is acceptable since it provides us with balanced queues, smaller queues and improves the error detection capabilities of the model checker.

6 Conclusions and Future Work

From the discussion above we can highlight some major points of interest. Parallel model checkers using the static partitioning algorithm have certain inefficiencies due to a variety of factors primarily related to the partitioning function and communication schemes. We have demonstrated that using the static partition algorithm, the queues on each process are highly imbalanced

Table 3
Average number of messages exchanged between two processors

Model	Static Partition	GDE Algorithm	Percentage
atomix	276751.41	254148.7	-8.17
dense	324927.66	320916.58	-1.23
cache315142	350984.19	349259.3	-0.49
jordon	621258.95	632280.92	1.74
sparse shallow	993826.05	1032035.14	3.7
queens8	242680.2	254276.52	4.56
two diamonds	994516.56	1112782.97	10.63

and the effective number of processors during the verification is half of the number of processors that are actually involved in the verification. We have also shown that using a state cache provides a small amount of improvement in terms of speedup over the static partition algorithm.

Using load balancing techniques such as GDE, we have successfully balanced the queues on all processors and reduced the time to verify models in our benchmark suite. Due to the non-deterministic nature of the GDE load balancing algorithm, we have also changed the search order to the degree where error states in our models can be discovered earlier and by exploring a fewer number of states. Using the GDE load balancing algorithm we have also shown that maximum queue sizes have been decreased by an order of magnitude compared to the maximum queue sizes obtained in the static partition algorithm for the models and benchmark suite used by us. We have also shown that the communication overhead does not counteract the usefulness of GDE load balancing when using a state cache.

Future work in this research area would involve creating load balancing schemes that are completely independent of any user input regarding the frequency of load balancing. Processors should be capable of avoiding situations where there is no useful work to do. A more detailed study of the GDE scheme with respect to the amount of load balancing done between a pair of processors is also important. Other dynamic load balancing schemes also provide an interesting field of further research too.

References

[1] VV Lab Model Database. http://vv.cs.byu.edu.

[2] G. Behrmann. A performance study of distributed timed automata reachability analysis. In *Electronic Notes in Theoretical Computer Science*, volume 68. Elsevier, 2002.

[3] E. M. Clarke, T. Filkorn, and S. Jha. Exploiting symmetry in temporal logic model checking.

In *Proceedings of the 5th International Conference on Computer Aided Verification*, pages 450–462. Springer-Verlag, 1993.

[4] C. Demartini, R. Iosif, and R. Sisto. dSPIN: A dynamic extension of SPIN. In *SPIN*, pages 261–276, 1999.

[5] O. Grumberg E. M. Clarke Jr. and D. A. Peled. Model checking. pages 9–11, 2002.

[6] P. Godefroid and P. Wolper. Using partial orders for the efficient verification of deadlock freedom and safety properties. In *Proceedings of the 3rd International Workshop on Computer Aided Verification*, pages 332–342. Springer-Verlag, 1992.

[7] G. J. Holzmann. An analysis of bitstate hashing. In *Proc. 15th Int. Conf on Protocol Specification, Testing, and Verification, INWG/IFIP*, pages 301–314, Warsaw, Poland, 1995. Chapman and Hall.

[8] G. J. Holzmann. The model checker spin. *IEEE Trans. Softw. Eng.*, 23(5):279–295, 1997.

[9] M. Jones, E. Mercer, T. Bao, R. Kumar, and P. Lamborn. Benchmarking explicit state parallel model checkers. In *Electronic Notes in Theoretical Computer Science*, volume 89. Elsevier, 2003.

[10] R. Kumar, M. Jones, J. Lesuer, and E. Mercer. Exploring dynamic partitioning schemes in hopper. Technical Report 3, Verification and Validation Laboratory, Computer Science Department, Brigham Young University, Provo, Utah, September 2003.

[11] F. Lerda and W. Visser. Addressing dynamic issues of program model checking. *Lecture Notes in Computer Science*, 2057:80–100, 2001.

[12] D. M. Nicol and G. Ciardo. Automated parallelization of discrete state-space generation. *Journal of Parallel and Distributed Computing*, 47(2):153–167, 1997.

[13] U. Stern and D. L. Dill. Improved Probabilistic Verification by Hash Compaction. In *Correct Hardware Design and Verification Methods*, volume 987, pages 206–224, Stanford University, USA, 1995. Springer-Verlag.

[14] U. Stern and D. L. Dill. Parallelizing the Murphi verifier. pages 256–278, 1997.

[15] M. H. Willebeek-LeMair and A. P. Reeves. Strategies for dynamic load balancing on highly parallel computers. *IEEE Trans. Parallel Distrib. Syst.*, 4(9):979–993, 1993.

[16] C. Xu and F. Lau. Iterative dynamic load balancing in multicomputers.