Load Balancing Parallel Explicit State Model Checking

Rahul Kumar

Brigham Young University - Provo

Follow this and additional works at: https://scholarsarchive.byu.edu/etd

Part of the Computer Sciences Commons

BYU ScholarsArchive Citation
Kumar, Rahul, "Load Balancing Parallel Explicit State Model Checking" (2004). Theses and Dissertations. 44.
https://scholarsarchive.byu.edu/etd/44

This Thesis is brought to you for free and open access by BYU ScholarsArchive. It has been accepted for inclusion in Theses and Dissertations by an authorized administrator of BYU ScholarsArchive. For more information, please contact scholarsarchive@byu.edu, ellen_amatangelo@byu.edu.
LOAD BALANCING PARALLEL EXPLICIT STATE MODEL CHECKING

by

Rahul Kumar

A thesis submitted to the faculty of
Brigham Young University
in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computer Science
Brigham Young University
August 2004
This thesis has been read by each member of the following graduate committee and by majority vote has been found to be satisfactory.

Date

Eric G. Mercer, Chair

Date

Quinn O. Snell

Date

Bryan S. Morse
As chair of the candidate’s graduate committee, I have read the thesis of Rahul Kumar in its final form and have found that (1) its format, citations, and bibliographical style are consistent and acceptable and fulfill university and department style requirements; (2) its illustrative materials including figures, tables, and charts are in place; and (3) the final manuscript is satisfactory to the graduate committee and is ready for submission to the university library.

Date

Eric G. Mercer
Chair, Graduate Committee

Accepted for the Department

David W. Embley
Graduate Coordinator

Accepted for the College

G. Rex Bryce, Associate Dean
College of Physical and Mathematical Sciences
This research first identifies some of the key concerns about the techniques and algorithms developed for distributed and parallel model checking; specifically, the inherent problem with load balancing and large queue sizes resultant in a static partition algorithm. This research 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 research 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 research presents a study of the communication
overhead incurred by using the load balancing algorithm, which although significant, does not offset performance gains.
ACKNOWLEDGMENTS

Professionally, I would like to thank my committee chair, Eric Mercer, for giving me this opportunity and invaluable guidance over the last two years. I would also like to thank Quinn Snell and the BYU Supercomputing Center for the resources and help they provided during the course of this research.

Personally, I would like to thank my parents, Satish and Sunita, and my sister Arushi for their love, support and advice. Last but not the least, I would like to thank Jennifer for being patient with me, especially during the frustrating times.
# TABLE OF CONTENTS

| Section | Page |
|---------|------|
| **1 Introduction** | 1 |
| 1.1 Thesis Statement | 3 |
| 1.2 Related Work | 3 |
| 1.3 Contributions | 4 |
| 1.4 Thesis Overview | 4 |
| **2 Parallelizing Murϕ** | 7 |
| 2.1 Murϕ | 7 |
| 2.2 Inter-Process Communication | 8 |
| 2.3 Asynchronous Murϕ | 9 |
| 2.4 Synchronous Class Hierarchy | 10 |
| 2.5 Termination Algorithm | 13 |
| **3 Testing Methodology** | 21 |
| 3.1 Test Platforms | 21 |
| 3.2 Models | 23 |
| 3.3 Metrics | 23 |
| **4 Analysis of Distributed Model Checking Techniques** | 27 |
| 4.1 Asynchronous versus Synchronous Murϕ | 27 |
| 4.2 Queue Imbalance | 30 |
| 4.2.1 Partitioning Function | 32 |
| 4.2.2 Dynamic Partition Algorithm | 34 |
| **5 Load Balancing Methods** | 41 |
| 5.1 Global Load Balancing | 41 |
| 5.2 GDE Load Balancing | 43 |
| **6 Results** | 49 |
| 6.1 Global versus GDE Speedup | 49 |
| 6.2 Search Order | 55 |
| 6.3 Queue Sizes | 59 |
| Section                                      | Page |
|----------------------------------------------|------|
| 6.4 Communication Overhead                   | 60   |
| 7 Conclusions and Future Work                | 63   |
| A State Cache                                | 71   |
| B State Buffering                            | 75   |
# LIST OF TABLES

| Table | Description                                                                 | Page |
|-------|-----------------------------------------------------------------------------|------|
| 3.1   | Metrics measured and compared for parallel model checking algorithms        | 25   |
| 4.1   | Standardized performance report for the atomix model on IBM 1350 Cluster.   | 28   |
| 4.2   | Standardized performance report for the jordon model on IBM 1350 Cluster.   | 29   |
| 6.1   | Average number of states generated before error state discovered.            | 59   |
| 6.2   | Maximum queue size and standard deviation of maximum queue size.             | 61   |
| 6.3   | Average number of messages exchanged between a pair of processors on IBM 1350 Cluster. | 61   |
## LIST OF FIGURES

| Figure | Page |
|--------|------|
| 2.1 Architectural overview of parallel static partitioning algorithm. | 11 |
| 2.2 Class hierarchy created to parallelize Murϕ. | 12 |
| 2.3 Pseudo code for termination algorithm for all processors except processor 0. | 14 |
| 2.4 Pseudo code for termination algorithm for processor 0. | 16 |
| 2.5 Figure showing exception to Dijkstra’s token termination algorithm. | 18 |
| 4.1 Speedups achieved for the atomix model using asynchronous and synchronous versions on IBM 1350 Cluster compared to serial Murϕ. | 30 |
| 4.2 Speedups achieved for the jordon model using asynchronous and synchronous versions on IBM 1350 Cluster compared to serial Murϕ. | 31 |
| 4.3 Queue sizes for the jordon model using 32 processors on IBM 1350 Cluster showing extreme imbalance. | 32 |
| 4.4 Aggregate times for composite functions for jordon using 32 processors on IBM 1350 Cluster showing high idle time. | 33 |
| 4.5 Queue sizes for the atomix model for 32 processors using one of SPIN’s partition functions on NOW architecture showing imbalance. | 34 |
| 4.6 Queue sizes for the atomix model for 8 processors using a different partition function on NOW architecture. | 35 |
| 4.7 Splitting pattern of processors in dynamic partition algorithm using 64 processors on NOW. | 36 |
| 4.8 State distribution of hash table using Dynamic Partition Function on IBM 1350 Cluster showing uneven distribution of states. | 37 |
| 4.9 State distribution of hash table using Static Partition Function on IBM 1350 Cluster showing even distribution of states. | 38 |
| 5.1 Pseudo code for global load balancing algorithm. | 42 |
| 5.2 Hypercube structure for 1 to 8 processors. | 43 |
| 5.3 GDE load balancing algorithm from sending end. | 44 |
| 5.4 GDE load balancing algorithm on receiving end. | 45 |
| 6.1 Relative Speedup of the global load balancing versus the traditional static partitioning algorithm on IBM 1350 Cluster. | 50 |
| Figure | Page |
|--------|------|
| 6.2 Speedup for the GDE scheme relative to the traditional static partitioning algorithm on IBM 1350 Cluster | 52 |
| 6.3 Speedup for the GDE scheme relative to the traditional static partitioning algorithm on NOW architecture | 53 |
| 6.4 Speedup for the GDE scheme relative to the traditional static partitioning algorithm on Shared Memory Architecture | 53 |
| 6.5 Queue sizes for the jordon model using 32 processors after load balancing | 54 |
| 6.6 Aggregate times for each function after the GDE load balanced scheme was implemented | 55 |
| 6.7 Queue sizes for the jordon model using 8 processors before load balancing | 56 |
| 6.8 Queue sizes for the jordon model using 8 processors after load balancing | 56 |
| 6.9 Queue sizes using 4 processors before load balancing | 57 |
| 6.10 Queue sizes using 4 processors after load balancing | 57 |
| A.1 Messages sent per processor for 4 processors with varying cache sizes using a NOW | 72 |
| A.2 Messages sent per processor for 8 processors with varying cache sizes using a NOW | 72 |
| A.3 Speedup obtained using a cache of size 1 MB measured on the IBM 1350 Cluster | 73 |
Chapter 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 [1]. State explosion is a result of enumerating the state space of a concurrent system using interleaving 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 [2][3]. Bit state hashing (supertrace) and hash compaction reduce the cost of storing states in the reachable state space [4][5]. 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, 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 compu-
tational resources of several processors to enhance the 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 \[6\]. The workload on each processor observed as a function of time and communication overhead depends critically on how the states are partitioned between the verification processors. Several techniques such as caching (sibling and state), partial order reduction, and different partition functions, including dynamic partitioning, have been explored in the past to reduce communication overhead and create perfect state distributions \[7][8\]. Even with the use of the above mentioned techniques, creating a perfect partition for any given problem while maintaining equally loaded processors requires \textit{a priori} knowledge of the state space, which is the very problem we are trying to solve.

This research 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 processors on different supercomputing platforms. The imbalance results in high idle times in several processors, as well as extremely large search queues. As a result, state generation is not as fast as it could be. 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 are often buried deep in the search queues. The research further presents a load balancing algorithm based on generalized dimensional exchange (GDE) to mitigate idle time in verification processors at the expense of additional communication overhead. Load balancing the state partition algorithm improves speedup in distributed model checking despite the increased communica-
tion. 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: networks of workstations (NOWs), Clusters, and Shared memory architectures.

1.1 Thesis Statement

Load balancing techniques based on the GDE method and state forwarding can be used to extend the parallel model checking process to achieve higher speedup, lower maximum queue sizes, and early error detection; thus, allowing the verification of unverified (larger) models.

1.2 Related Work

The seminal work in parallel and distributed model checking was presented by [6]. Results were presented using a NOW and an IBM SP2. Research on parallel state space generation and model checking exploring new techniques and architectures is presented by Garavel and Barnat in [9][10]. Similar observations of imbalances in the queue causing high idle time in individual processes in the verification are presented by Behrmann in [11]. Behrmann also presents techniques such as state buffering and increasing locality to improve performance. Techniques such as sibling caching and children lookahead to improve the speedup obtained using parallel model checkers are introduced in [7]. Originally, the concept of state caching to reduce the number of hash lookups and take advantage of temporal transition locality within a state space was introduced in [12]. Nicol in [13] also discusses various methods for improving performance and implementing dynamic load balancing using global load balancing schemes. Many serial techniques such as partial order reduction, and symmetry reduction and disk based model checking have also been developed to assist model checking algorithms to verify larger models [2][3][14]. Other approaches such as
symbolic model checking have also been explored by [15]. In spite of the research efforts and methods mentioned, parallel model checking faces many inefficiencies which are addressed during the course of this research.

1.3 Contributions

This research contributes to the parallel and distributed model checking community by introducing a new load balancing algorithm that improves upon the inefficiencies and imbalances of the previous algorithms. The algorithm provides a novel way of balancing the queues on each processor as well as providing a method to reduce the maximum queue size achieved by a single processor to minimize idle time incurred by processors during verification. The new load balancing algorithm enables parallel model checkers to modify the search order significantly; thus, discovering error and deadlock states faster than the previous algorithm by exploring fewer states. Along with the above stated contributions, this research also contributes results to the parallel model checking community on three major supercomputing platforms: distributed memory cluster computing, shared memory architectures and NOWs. Finally this research encourages the development of parallel model checking tools and innovative methods on various architectures.

1.4 Thesis Overview

Chapter 2 discusses the parallelization methodology applied to Murϕ, a serial model checker developed at Stanford University using C++. The tools and libraries used to parallelize Murϕ are discussed in detail as well as the instrumentation of the code to achieve an extensible class hierarchy. Test platforms, models and metrics used during the course of this research are discussed in Chapter 3. Chapter 4 will discuss the research, techniques and methods that have been used to this point in parallel/serial model checkers to achieve higher performance and verify larger models.
Specifically, techniques such as state caching, state buffering and the static partition algorithm are analyzed and empirical results are presented. Load balancing methods are discussed in detail in Chapter 5. Following the methods, effects of load balancing and additional results are presented in Chapter 6. Conclusions and future work are presented in Chapter 7.
Chapter 2

Parallelizing Mur$\varphi$

2.1 Mur$\varphi$

Mur$\varphi$ is an explicit state model checker developed in C++ at Stanford University. Mur$\varphi$ is designed with extensions and future use in mind providing a class hierarchy that is highly modular and reusable. Mur$\varphi$ provides a stable development platform with thoughtful design decisions incorporated, which lend themselves to parallelization. Originally developed for serial algorithms, Mur$\varphi$ is capable of performing multiple search types such as BFS/DFS/Simulation as well as reduction techniques such as hash compaction and symmetry reduction. Each of these features can be controlled using either command line arguments or explicit definitions during compile time. Polymorphism and C++ are used extensively to deal with states, state vectors, transition functions as specified in the model properties, and the various reduction techniques. Mur$\varphi$ also provides a compiler for the Mur$\varphi$ language developed to write models. The Mur$\varphi$ language provides functionality to specify symmetry reduction, partial order reduction and the invariants to check for during the verification process. Once a Mur$\varphi$ model has been created, it is processed by the Mur$\varphi$ compiler to gener-
ate a C++ file, which is then compiled with the model checking part of Murϕ. Murϕ is chosen due to its extensible and flexible nature as described above.

2.2 Inter-Process Communication

Today’s supercomputing world offers multiple paradigms for implementing parallel and distributed applications. Shared memory architectures provide extremely high bandwidth and low latency, while cluster computing provides the ability to allocate tasks and resources in efficient manners. NOWs provide easy access to cheap hardware while retaining a large part of the benefits of supercomputing. To take advantage of all these platforms and paradigms, the code can be structured and developed independently for each paradigm or platform independent standards, and libraries can be utilized to maintain functionality on each platform while maximizing performance. Choosing an approach that utilizes libraries so that the code is platform independent reduces the development time as well as the development cost. In comparison, writing custom code for each platform provides no greater benefits. The Message Passing Interface (MPI) is one such standard that can be used to develop parallel applications [16]. It provides a common interface across several architectures, high performance and ease of use on all architectures as well as maintaining functionality.

MPI has become the de facto standard for parallel/distributed applications, supporting multiple platforms and architectures. Commercial versions provide extremely high performance and diverse functionality on a variety of platforms. Message passing using MPI can be performed using a variety of synchronous, asynchronous, blocking, non-blocking or buffered operations as defined in the MPI specification. The greatest advantage of using MPI is the common interface provided on all architectures used in this research, while providing high bandwidth and ease of use at the same
time. Besides providing message passing capabilities, MPI implementations such as MPICH \cite{17} also provide the technology to start multiple processes on various architectures. Compared to the Parallel Virtual Machine (PVM) interface and libraries, MPI provides appropriate support and functionality for the purposes of this research as well as higher performance \cite{18,19}. For this research, MPICH version 1.2.2 has been selected for parallelizing Mur$\varphi$. MPICH is readily available and has been tested thoroughly due to the various research/user groups using it as their message passing library.

To parallelize Mur$\varphi$ multiple communication paradigms were explored. An asynchronous multi-threaded version using asynchronous sends and receives as well as a synchronous version were developed and tested. The following sections discuss the architecture of each version.

2.3 Asynchronous Mur$\varphi$

The asynchronous communication architecture has two user level threads initiated by Mur$\varphi$ that communicate through shared memory to separately handle the communication and state generation parts of the static partition algorithm. The communication thread is solely responsible for communicating states to other processors and receiving states from other processors. Due to multiple threads accessing common data objects such as the hash table and the queue, mutexes are used to ensure correctness of data and functionality.

Communication occurs using asynchronous sends and receives in the MPI specifications. To receive termination messages and states, the communication thread enters an infinite loop that performs a non-blocking probe. If there are any messages that need to be received, the communication thread performs a receive operation and then processes the received message(s). If the messages include states that need to be
added to the hash table and queue, an attempt to lock the respective mutex is made. The state generation thread also enters an infinite loop to dequeue states from the queue and generate the successors. The hash function is used to calculate ownership of the state. If the state belongs to the generating processor, the state is added to the hash table and queue, otherwise it is scheduled for shipment to the respective owner. A global set of queues is also shared by the two threads on each processor. These queues are used for storing states that need to be sent to the respective owners. Mutexes are also used for these data structures to ensure data correctness. Periodically, if the state buffering limit is reached or the watchdog timer has expired, the states from these queues are sent in an asynchronous fashion and the data structures are reset. Results for the asynchronous architecture are described in Chapter 4.

2.4 Synchronous Class Hierarchy

Figure 2.1 shows the architecture of the distributed static partitioning algorithm as presented in [6]. The major composite functions are State Generation (SG), Waiting Queue (WQ) where states are waiting to be processed by the SG, Passed Queue or hash table (PQ) where explored states are saved for future use and the Send/Receive Thread (SR) used for performing communication. Murφ provides the infrastructure for implementing the SG, WQ and PQ modules. Each module is a class in Murφ providing an interface for performing all necessary operations. The WQ is implemented as a generic queue in Murφ, that contains pointers to states existing in the hash table. The hash table uses a double hashing function to calculate the location of each state. Each of these classes provides an interface to insert new states, dequeue states, enqueue states and check for presence of states. Other functions for gathering data and statistics are also available. The AlgorithmManager class is responsible for executing the algorithm as selected by the user using command line options. The
Figure 2.1: Architectural overview of parallel static partitioning algorithm.

AlgorithmManager class contains the generic BFS/DFS/Simulate methods that execute serial code. The State Manager class is responsible for maintaining the queue and the hash table objects as discussed earlier. Other classes present in Mur$\varphi$ are equally important to the serial execution of the model checker but their discussion is irrelevant to parallelizing Mur$\varphi$.

To parallelize Mur$\varphi$ in a seamless manner, a new class, the MPIStateManager is created which inherits the interface from the StateManager class. Given the correct set of definitions during compile time (to include the MPIStateManager), the user can choose to run the compiled executable in parallel mode using the -parallel command line flag. If this command line argument is used, the StateManager variable is created as a MPIStateManager variable instead of a StateManager variable. This enables Mur$\varphi$ to utilize the methods and functions of the MPIStateManager class which is responsible for handling all communication, distribution of states, and termination. Besides the functionality mentioned, the MPIStateManager class maintains the serial functionality of the StateManager class. The following functions have been modified (by making them virtual):

1. `StateManager::Add`: computes the owner of the state. If the owner is the
current process, the state is added to the local queue otherwise sent to the remote process.

2. `StateManager::Dequeue`: returns a state from the local queue. If the local queue is empty, the function waits until termination is detected or states for dequeuing have been received.

3. `StateManager::QueueisEmpty`: returns false only if termination has been successfully detected. If termination has not been detected and there are no states in the queue, the function calls `StateManager::ReceiveMessages` to detect termination or receive states from other processors.

Other classes responsible for error reporting and trace generation are also taken into account for the parallelization of Murϕ. The `MPIReportManager` class is created as a child class of the `ReportManager` class responsible for generating traces and reporting run time errors during verification. For reporting discovery of error or deadlock states, the `MPIErrorManager` class is created as a child of the `ErrorHandler` class. Each of the child classes, at a very minimum, maintains functionality of the parent class and are only invoked during execution in parallel mode. The `DYNStateManager`
was added to create the dynamic partition algorithm. Figure 2.2 shows the class hierarchy created to parallelize Murϕ. In this manner, multiple child classes can be added to the existing classes to add new behavior to Murϕ.

2.5 Termination Algorithm

Termination detection for distributed/parallel model checking is very important and needs to be implemented in a manner so as to guarantee correct detection every time. Consider a situation where processor $P_0$ sends states to processor $P_r$. If at this point, every processor has an empty queue but there are states in-flight, an incorrect termination algorithm would allow termination of the processors since there is currently no useful work to be performed. This would be incorrect because of the in-flight states between $P_0$ and $P_r$. In such a situation, the termination algorithm needs to detect the states that are in-flight and cause the rest of the processors to wait until the states have been explored by $P_r$. It is possible that the in-flight states contain an error or contain transitions to a portion of unexplored state space that contains error states. For this reason, the termination algorithm used should be capable of accounting for messages that have been exchanged by every processor and determine termination correctly.

To ensure correct termination, a modified Dijkstra’s token termination algorithm is used [20]. Dijkstra’s token termination algorithm assumes that all N processors are organized in a logical ring, such that any message originating at processor $P_o$, is always sent to the next logical processor in the ring $P_{(o+1)\%N}$ until it reaches the destination processor $P_d$. This particular assumption establishes an order in the delivery of messages. In a logical ring, it is not possible that messages from multiple processors to a single destination processor arrive out of order. The pseudo code for Dijkstra’s termination algorithm is shown in Figure 2.3 and Figure 2.4. These
Algorithm: MPIStateManager :: Terminate(ReceivedToken)

1: /∗ Termination initiated only by processor 0 by sending FINISHED token */

2: /∗ Method invoked when token is received from previous logical processor or when idle */

3: 

tokenType := GetTokenColor(ReceivedToken)

4: /∗ Token not forwarded until processor is idle */

5: if myStatus == NOTFINISHED then

6: 

SendToken((MYID + 1) \% N, NOTFINISHED)

7: 

myStatus := FINISHED

8: 

return

9: else

10: 

SendToken((MYID + 1) \% N, tokenType)

11: 

myStatus := FINISHED

12: 

return

13: end if

Figure 2.3: Pseudo code for termination algorithm for all processors except processor 0.
algorithms are only initiated if the processor is idle and has received a token, or processor 0 is initiating termination by sending the \texttt{FINISHED} token. Only processor \( P_0 \) is capable of initiating termination. If \( P_0 \) is idle, the \texttt{FINISHED} token is sent to \( P_1 \). Once a token is received by a processor, it is only processed if the respective processor is idle. If idle and no work has been sent to any other processor since the last token received, a processor forwards the received token unchanged. If on the other hand, work has been sent to other processors by the processor holding the token, the \texttt{NOTFINISHED} token is forwarded to the next logical processor. It should be kept in mind that a token is only processed and forwarded once the processor runs out of useful work to do. In this manner, the token is forwarded in the ring until it reaches \( P_0 \). Once processor \( P_0 \) receives the token, if the \texttt{FINISHED} token is received, the termination signal is broadcast and termination occurs. If on the other hand, the \texttt{NOTFINISHED} token is received by processor \( P_0 \), termination detection is restarted by \( P_0 \) by sending the \texttt{FINISHED} token, if it is still idle or has become idle after work being received and processed.

Now consider a situation as shown in Figure 2.5. Each arrow indicates a time step where a message is sent or received by a processor. Processor \( P_2 \) is currently holding the token and some states, and every other processor is idle. Processor \( P_2 \) sends all the states to \( P_0 \) at time \( T_t \). At this point, processor \( P_2 \) becomes and idle and the \texttt{NOTFINISHED} token is forwarded to processor \( P_0 \) by processor \( P_2 \). After receiving the token, \( P_3 \) then forwards the \texttt{NOTFINISHED} token to \( P_0 \) (\( P_0 \) has not received the work sent by \( P_2 \)). The \texttt{FINISHED} token is sent to \( P_1 \) at time \( T_{t+3} \) by processor \( P_0 \). Since all processors are idle, the \texttt{FINISHED} token is forwarded around the ring and processor \( P_0 \) receives it at time \( T_{t+6} \). Since \( P_0 \) receives the \texttt{FINISHED} signal, the termination signal is broadcast by \( P_0 \) even though there is work that is in-flight between processor \( P_2 \) and
/* Processor 0 performs comparisons upon receipt of token that has been circulated around the logical ring */

**Algorithm:** $\text{MPIStateManager}::\text{Terminate}(\text{ReceivedToken})$

1. $\text{tokenType} := \text{GetTokenType}(\text{ReceivedToken})$
2. If $\text{tokenType} == \text{FINISHED}$ then
3. /* Traditional Dijkstra’s algorithm terminates here */
4. $\text{messagesReceived} := \text{GetMessagesReceived}(\text{ReceivedToken})$
5. $\text{messagesSent} := \text{GetMessagesSent}(\text{ReceivedToken})$
6. If $\text{messagesReceived} == \text{messagesSent}$ then
7. /* Send signal to other processors to terminate */
8. $\text{SendTerminationSignal}()$
9. return
10. Else
11. /* If total messages exchanged are not equal, return without terminating */
12. return
13. End if
14. Else
15. /* Drop token and try to terminate again */
16. return
17. End if

Figure 2.4: Pseudo code for termination algorithm for processor 0.
This is an example of a situation where Dijkstra’s token termination algorithm detects termination incorrectly in an environment other than a logical ring. Incorrect termination is detected because the underlying network structure is not a logical ring as assumed by Dijkstra. It is possible that messages can be sent directly from processor \( P_0 \) to processor \( P_1 \) or \( P_3 \) without going through other processors. Dijkstra’s algorithm on the other hand requires that the processors be connected in a logical ring, where processor \( P_0 \) can only send messages to/through \( P_1 \) and processor \( P_2 \) can only send messages to processor \( P_3 \). In the latter case, the above described situation would have not occurred because the work sent from \( P_2 \) to \( P_0 \) as well as the tokens would have been received by processor \( P_0 \) in order; thus, maintaining correctness of the token termination algorithm.

To rectify the above described situation, the token is modified to contain information about the total number of sent and received messages. Processor 0 always initiates termination by sending the modified token, with the sent and received number of messages fields populated with it’s values. Every time the token is forwarded, the number of sent and received messages fields are updated by the respective processor with it’s local sent and received message counter values. As shown in Figure 2.4, if processor \( P_0 \) receives the FINISHED token, instead of terminating immediately as in the traditional token termination algorithm, the number of sent and received messages are compared. If the comparison is successful (number of sent messages equals the number of received messages), it means that there are no messages in-flight and termination can proceed. If on the other hand, the number of messages sent is greater than the number of messages received, there are still some messages in-flight and termination can not proceed. This simple modification enables the termination algorithm
Figure 2.5: Figure showing exception to Dijkstra’s token termination algorithm.
to correctly count the messages for all processors and ensure correct termination by accounting for in-flight messages.
Chapter 3

Testing Methodology

The following sections will discuss the test platforms and models that have been used to test the algorithms implemented during the course of this research. Section 3.3 will discuss the metrics that have been reported to study the performance of the algorithms.

3.1 Test Platforms

To test the methods, algorithms and techniques utilized during the course of this research, three supercomputing platforms are selected: distributed memory cluster computing, NOW, and shared memory. Each platform presents with it a set of pros and cons. Distributed memory cluster computing provides a very fast interconnect, as well as an independent operating system on each processor. Clusters can provide performance comparable to Symmetrical Multi-Processor (SMP) and shared memory architectures at a lower cost with added support built into the parallel applications. Cluster computing is also better suited if the applications need to be scaled to larger numbers of processors. The NOW architecture on the other hand provides multiple processors processing data at lower speeds and using lower speed interconnects such as Ethernet. The cheapest of all the other mentioned architectures, NOWs can be
useful in situations where the financial resources to build clusters and shared memory machines are not available. The NOW architecture is capable of scaling to a certain degree but is not the best platform to scale on due to the slow interconnects, which increases the communication cost significantly. Shared memory architectures and SMP systems are capable of providing the highest performance in terms of latency and bandwidth. With shared memory architectures, the biggest issue is memory contention and thrashing that is caused by poorly structured software applications. Shared memory architectures are also very expensive and do not scale to a higher number of processors due to financial and physical constraints. The following is a detailed list of the platforms on which empirical testing was performed:

1. IBM 1350 Linux Cluster 256 Pentium Xeon processors @ 2.4GHz 128 X335 compute nodes (2 processors each) 256 GB total memory Linux (2.4 Kernel)

2. Network of workstations (NOW) with 100 Mbps interconnect

3. Marylou10 IBM pSeries 690 64 Power4+ processors @ 1.7 GHz 64 GB total memory

The above set of test platforms has been picked due to multiple reasons. First, each architecture provides its pros and cons as discussed above. This makes the empirical analysis and the study of the behavior of the algorithm very detailed and more complete, since the behavior of the algorithm is directly affected by the type of interconnect, data dependencies and processor speeds used. Second, each of these test platforms are readily accessible to us for testing purposes. Third, these test platforms are the major supercomputing test platforms in use today. By testing and developing on all architectures, the results of the algorithm are more applicable to the research community and likely for future use. Finally, results for distributed and parallel
model checking on shared memory architectures is relatively rare, which provides great incentive to study parallel model checking algorithms on shared memory architectures.

3.2 Models

Testing is performed using the model database developed at the Verification and Validation Laboratory [21]. These models are selected because they provide a controllable and interesting state space. The *atomix* model is a one player game where the atom tries to combine in a specific manner so as to form a molecule. The *jordon* model performs various operations on counters and arrays. It also gives us control of the location and placement of the error state. Other models such as the *queens8* model, based on the N-queens problem, *two diamonds, dense,* and *sparse-shallow* have also been selected for testing due to their large state vector sizes. An increase in the state vector size makes the model checking problem exponentially harder due to the increased memory requirements when verifying the model as well as the larger number of potential states that can exist in the state space of the model. The increased vector size directly affects the number of states that can be saved in the hash table along with the communication overhead. Larger state vector sizes mean that fewer states can be sent in a single TCP/IP message; thus, they require a greater number of messages to be communicated for exchanging work and completing model verification. This selected set of models is not representative of all types of problems, but they effectively capture our general observations in studying several problems.

3.3 Metrics

Algorithm performance can be characterized with basic metrics showing real time, speedup, CPU time, peak memory usage, and communication overhead. This section discusses the metrics used during the course of this research and their applicability.

Time is reported in terms of *real time* to better characterize algorithm behavior.
in the dynamically loaded NOW environment. Real time is the actual time it takes for the algorithm to complete the model checking problem. This can be understood as the time measured on a clock on the wall from when the problem begins to when the solution is complete.

Barr gives a basis for reporting performance on parallel and distributed algorithms in [22] and shows speedup to be a key metric for performance analysis. $S(p)$ is the classical definition of speedup relative to the fastest known serial algorithm [22]. The fastest serial code for model checking is not well defined. As such, it is tempting to revert to a relative speedup given by:

$$RS(p) = \frac{\text{Time to solve with parallel code on a single processor}}{\text{Time to solve with parallel code on } p \text{ processors}}$$

There is danger in relative speedup, however, because it is a self comparison; thus, $S(p)$ is calculated relative to sequential Murφ. Although real($p$) and $S(p)$ give a good indication of performance, they do not fully describe algorithm behavior.

To create a more detailed study, the maximum and minimum real time for each processor is measured individually, as well as the aggregate real time for all the processors. These metrics, provide insight into the behavior of the model checking algorithms.

The $COM(p)$ statistic describes the message passing behavior of the algorithm to show a more complete picture of communication. Specifically, $C = COM(p)$ is a $p \times p$ matrix where each entry $C_{ij}$ is the mean number of states sent from workstation $i$ to workstation $j$ in solving the model checking problem. Although this matrix can be reported in raw form, it can also be visualized as a surface or a stacked bar chart. This research utilizes the $COM(p)$ metric to calculate the communication overhead and study the communication patterns.

To compare the load balanced algorithms to the traditional static partition algo-
| Metric       | Definition                                                                 |
|-------------|--------------------------------------------------------------------------|
| $real(p)$   | mean real time to complete using $p$ processors                           |
| $S(p)$      | $\frac{\text{Mur$\ddot{\text{o}}$ serial code on workstation}}{real(p)}$ |
| $LS(p)$     | $\frac{\text{Time to solve with traditional static partition algorithm on } p \text{ processors}}{\text{Time to solve with load balanced algorithm on } p \text{ processors}}$ |
| $CPU(p)$    | mean CPU time for processor $p$                                           |
| $CPU \min(n)$ | $\min_{i \in n}(CPU(i))$                                           |
| $CPU \max(n)$ | $\max_{i \in n}(CPU(i))$                                           |
| $CPU \text{ agg}(n)$ | $\sum_{i \in n}(CPU(i))$                                           |
| $COM(p)$    | $p \times p$ communication matrix                                        |

Table 3.1: Metrics measured and compared for parallel model checking algorithms

...
Chapter 4

Analysis of Distributed Model Checking Techniques

This chapter analyzes the static partition algorithm and presents the problem of imbalanced queues due to the use of the static partition algorithm. Results for the asynchronous and synchronous implementations discussed in Chapter 2 are presented, after which the static partition algorithm is analyzed using different partition functions as well as a dynamic partition function to show the persistence of the queue imbalance problem, which leads to poor performance.

4.1 Asynchronous versus Synchronous Murϕ

During the presentation of this research we assume that each verification process runs on one dedicated processor; whereas, multiple processors (processes) could exist on a single processing node. Every process generates, receives, queues and distributes states. State ownership is calculated using a predetermined static hash function that is replicated on each processor. The hash function can also be referred to as the partition function due to the partition it creates in the state space. During the course of this research we refer to the hash function as the partition function, to emphasize
Table 4.1: Standardized performance report for the atomix model on IBM 1350 Cluster.

|                  | 2   | 4   | 8   | 16  | 32  | 64  | 96  | 128 |
|------------------|-----|-----|-----|-----|-----|-----|-----|-----|
| **real(n)**      |     |     |     |     |     |     |     |     |
| Asynchronous     | 25:40 | 20:42 | 15:09 | 9:27 | 7:32 | 6:35 | 5:19 | 3:59 |
| Synchronous      | 32:28 | 17:31 | 18:26 | 2:51 | 3:36 | 2:46 | 2:15 | 3:31 |
| **CPU min(n)**   |     |     |     |     |     |     |     |     |
| Asynchronous     | 21:08 | 9:32 | 5:41 | 2:48 | 2:40 | 50  | 25  | 23  |
| Synchronous      | 31:40 | 16:49 | 7:16 | 2:30 | 1:31 | 1:47 | 26  | 12  |
| **CPU max(n)**   |     |     |     |     |     |     |     |     |
| Asynchronous     | 21:23 | 16:17 | 11:47 | 7:23 | 5:36 | 4:51 | 3:50 | 1:40 |
| Synchronous      | 32:04 | 17:20 | 15:34 | 2:34 | 3:13 | 1:04 | 52  | 1:51 |
| **CPU agg(n)**   |     |     |     |     |     |     |     |     |
| Asynchronous     | 42:31 | 53:56 | 1:14:39 | 1:32:49 | 2:09:07 | 3:18:51 | 3:32:24 | 2:01:53 |
| Synchronous      | 1:08:04 | 1:08:35 | 1:15:48 | 41:06 | 1:07:22 | 7:38:06 | 1:01:53 | 3:20:24 |
Table 4.2: Standardized performance report for the jordon model on IBM 1350 Cluster.

|             | 2     | 4     | 8     | 16    | 32    | 64    | 96    | 128   |
|-------------|-------|-------|-------|-------|-------|-------|-------|-------|
| **real(n)** |       |       |       |       |       |       |       |       |
| Asynchronous| 25:44 | 23:39 | 11:36 | 9:03  | 3:53  | 4:08  | 4:26  | 3:47  |
| Synchronous | 25:27 | 22:06 | 7:13  | 3:19  | 2:23  | 1:34  | 1:25  | 1:50  |
| **CPU min(n)** |    |       |       |       |       |       |       |       |
| Asynchronous | 14:26 | 8:42  | 4:19  | 3:29  | 1:18  | 57    | 15    | 24    |
| Synchronous | 22:56 | 15:37 | 4:41  | 2:11  | 1:27  | 24    | 31    | 23    |
| **CPU max(n)** |    |       |       |       |       |       |       |       |
| Asynchronous | 17:32 | 10:48 | 5:34  | 4:32  | 1:51  | 1:44  | 2:31  | 1:31  |
| Synchronous | 23:14 | 19:37 | 6:43  | 3:07  | 2:01  | 41    | 40    | 42    |
| **CPU agg(n)** |    |       |       |       |       |       |       |       |
| Asynchronous | 31:59 | 39:01 | 39:38 | 1:04:45 | 51:06 | 1:23:06 | 2:07:57 | 1:56:19 |
| Synchronous | 46:10 | 1:08:13 | 46:18 | 39:50 | 1:03:12 | 23:24 | 37:46 | 48:01 |

on the dual role the function plays. It should be noted that the partition function has to be consistent and deterministic on each processor to eliminate multiple copies of identical states that have been generated by different processors.

The static partition as discussed earlier, is implemented using asynchronous and synchronous forms of communication. Results for the atomix and jordon models are shown in Table 4.1 and 4.2. Speedup relative to the serial algorithm for each model using the asynchronous and synchronous approaches is shown in Figure 4.1 and 4.2 as observed on the IBM 1350 Cluster. Comparing the asynchronous version to the synchronous version, we can see that the synchronous version outperforms the asynchronous version for both problems as well as for all sets of processors. This large difference in performance is attributed to the more regular and precise communication
patterns that are present in the synchronous version of Murϕ due to a single thread. The single thread has no data dependencies and mutual exclusion issues. For the remainder of the thesis, the static partition algorithm or the traditional algorithm will always refer to the synchronous version of the static partition algorithm.

### 4.2 Queue Imbalance

The speedup achieved by the static partitioning algorithm is a significant improvement over the serial algorithms in Murϕ and enables the verification of larger models. Even though this is the case, the speedups achieved by using the static partition algorithm are dependent on the partition function employed. Further investigation of the algorithm supports the conclusion that idle times during parallel verification are high, and there is a high imbalance in the distribution of states across the queues of the participating processors causing premature termination and degraded error discovery.
Figure 4.2: Speedups achieved for the jordon model using asynchronous and synchronous versions on IBM 1350 Cluster compared to serial Murφ.

Figure 4.3 shows the sizes of the queues sampled at one second intervals for 32 processors throughout the verification on the IBM 1350 Cluster. The interval sampling is implemented using the\texttt{alarm} and \texttt{signal} utilities available on most UNIX based systems. Each processor receives an interrupt every second for printing its unique identification number followed by a queue size. After verification the output is parsed and separated for all the processors. The horizontal axis represents time and the vertical axis represents the number of states in the queue of a given processor. From the figure we can see the high imbalance in the queue sizes for each processor. Verification for this model requires 120 seconds. After the half way mark on the horizontal axis (60 seconds), only three of the processors are active, with the other processors having very little or no useful work to perform. These results are consistent with the observations of Behrmann in [11].
Figure 4.3: Queue sizes for the jordon model using 32 processors on IBM 1350 Cluster showing extreme imbalance.

Figure 4.4 displays the aggregate percentage of time that has been spent on the major composite functions during verification for the jordon 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 processor’s search queue is empty. In this state, the verification processor is still processing messages and discarding states that have been previously visited and stored in the hash table. The figure demonstrates that after communication, idle time dominates with a third of the total aggregate time spent idle.

4.2.1 Partitioning Function

Idle time is a direct result of load imbalances. This is seen in experiments with various partition functions and their effect on speedup. The partitioning function
Figure 4.4: Aggregate times for composite functions for jordon using 32 processors on IBM 1350 Cluster showing high idle time.

plays a key role in creating the state distribution. We have studied the effects of the partition function on the distribution of states by implementing various partition functions including several partition functions in SPIN [23]. Figure 4.5 shows the queue distribution for 64 queues on the NOW architecture using the single bit forward partition function from SPIN. Although more processors remain active through the verification run when compared to Figure 4.3, a significant number still become idle and remain idle for over half the running time of the verification process. Similar results were noticed for different hash functions on the same set of processors. Figure 4.6 shows a similar imbalance in queue distribution using a different bit mask for the hash function taken from SPIN on 8 processors. The distribution is extremely different for all numbers of processors and various hash functions. Similar results have been noticed for other hash functions such as Jenkins forward partition function. 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
Figure 4.5: Queue sizes for the atomix model for 32 processors using one of SPIN’s partition functions on NOW architecture showing imbalance.

is the problem that the load balancing methods presented in Chapter 5 will provide solutions to.

4.2.2 Dynamic Partition Algorithm

To further explore the effects of the partition function, a dynamic hash function can be created which takes into account the irregular state space of the model being verified, and in theory tries to create a perfect partition of the states in the hash table as well as evenly balanced queues across all processors. The primary goal of the dynamic partition function is to verify models in a manner where memory and computational resources are used only if necessary, making the algorithm space efficient. To do so, verification is started on a single processor. The memory threshold is defined as the hash occupancy percentage at which a processor allocates more memory by including another processor in the search. Once the memory threshold is reached on the first processor, a second processor is requested with the same amount of memory
allocated for the hash table as the parent processor. States from the hash table as well as the queue are transferred to the child processor via messages on the underlying interconnect. Before the states have been transferred, the partition function is updated to accommodate the new processor and to create an even partition of states. The new partition function is communicated to the other processors to maintain correctness of the verification and the partition. Splitting and state generation continues until the model has been verified completely or there are no more processors to split to. The current algorithm uses the state vector as the only input to calculate the new hash function. The memory threshold parameter is predefined by the user during the time of compilation. For the results presented in this section, the memory threshold was set at 70%.

Figure 4.6 shows a splitting graph where each node in the graph represents a processor. Initially there is only processor 0, which grows into 64 processors by the time the model has been completely verified. From the figure, we can see that splitting
Figure 4.7: Splitting pattern of processors in dynamic partition algorithm using 64 processors on NOW.

The performance achieved by the algorithm is not comparable to the static partition algorithm discussed above. The state distribution in the hash table achieved is very uneven and imbalanced. Figure 4.8 shows the distribution of states in the hash table on all processors using the dynamic partition algorithm on the IBM 1350 Cluster. The algorithm required 33 processors to completely verify the model. To compare the state distribution of the dynamic partition algorithm to the static par-
Partition algorithm, the static partition algorithm was run using 33 processors and the same amount of memory allocated for the hash table on each processor as the dynamic partition algorithm. Figure 4.8 shows the distribution of the states in the hash table on each processor achieved using the static partition algorithm using the IBM 1350 Cluster. As can be seen from the figures, the distribution is more even for the static partition algorithm compared to the dynamic partition algorithm. An important difference is the maximum number of states saved on a single processor, which can determine the fate of the verification. Using the dynamic partition algorithm, the maximum number of states is at most 1.5 times greater than the maximum number of states on a single processor using the static partitioning algorithm. Apart from the state distribution, the speedup achieved by the static partition is significant compared to the dynamic partition algorithm. This is because the dynamic partition algorithm, does not utilize all 33 processor throughout the verification, instead adding them to the current set of processors as needed by the memory requirements of the model.

Figure 4.8: State distribution of hash table using Dynamic Partition Function on IBM 1350 Cluster showing uneven distribution of states.
Figure 4.9: State distribution of hash table using Static Partition Function on IBM 1350 Cluster showing even distribution of states.

Apart from the performance issues mentioned above, the dynamic partition algorithm does not lend itself very well to the super computing paradigms listed above. The greatest issue being the fact that new processors as well as memory have to be requested on the fly, which is difficult to do with current MPI implementations.

From the above discussion, it can be concluded that the partition function as well as partition algorithms are capable of influencing the workload on each processor, but only to a certain degree. Partition techniques are highly dependent on the number of processors as well as the nature of the state space under investigation. To create a perfect load distribution across all the processors, a static analysis will have to be performed of the state space each time a new model is verified. Performance, which is highly dependent on the workload on each processor is directly influenced by the partition function employed. The focus of this research is to eliminate the dependency on the partition function and enable distributed model checkers to have balanced workloads on each processor to achieve high performance and low idle times. The next chapter discusses load balancing techniques used. Other techniques such as state caching and buffering have also been implemented to improve the load balance.
and performance of distributed model checking. Discussion on these topics can be found in the appendices.
Chapter 5

Load Balancing Methods

5.1 Global Load Balancing

Work presented by Nicol describes a global load balancing technique where all the processors involved in the state generation try to achieve a state of perfect equilibrium [13]. In the equilibrium state, all processors have an equal number of states in the queue in an effort to remove the imbalance. Short periods of computation of states are followed by a period where the processors balance the queues with each other by exchanging queue information and extra states in the queue. Figure 5.1 shows the pseudo-code for the global load balancing algorithm modified to work within the synchronous version of the static partition algorithm.

There are multiple drawbacks to this method. Firstly, there is too much communication overhead introduced in the parallel model checking problem which itself is very communication intensive. This causes the algorithm to not scale as well if the model or the number of processors is increased significantly. Secondly, 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) the communication overhead is extremely high and the effective speedup is very low. On
Algorithm: GlobalLoadBalance()

1: /* Method called by BFS/DFS every i iterations for all N processes */
2: \( q := \text{getLocalQueueSize}() \)
3: /* Send and receive queue sizes from all N processes */
4: \( a := \text{gatherAllQueueSizes}() \)
5: \( \text{root} := 0; \)
6: if \( q > a/N \) then
7: \( i := \text{getDistributionInformation}(\text{root}) \)
8: for \( \text{Allidleprocessors} \) do
9: \( \text{sendStates}((q - s_n)/2, \text{getIdleProcessorID}(i)) \)
10: end for
11: else
12: /* Receive happens in a non-blocking fashion */
13: \( \text{receiveStates}() \)
14: end if
15: return

Figure 5.1: Pseudo code for global load balancing algorithm.
the other hand, if the value selected is large (low frequency) effective load balancing does not take place and the parallel verification processor behaves in the same manner as the traditional algorithm. Another issue with the global load balancing algorithm is the amount of time incurred waiting for all the processors to gather in the load balancing phase for load balancing to occur. 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 overworked and is 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 for $P_0$ which, causes great synchronization effort and delay.

5.2 GDE Load Balancing

This section introduces iterative load balancing in model checkers to balance the states in the queues using generalized dimensional exchange (GDE). The algorithm is first introduced by Willebeek-LeMair and Reeves in [24], and has been discussed
Algorithm: \texttt{GDEBalanceQueues()}

\begin{algorithmic}[1]
\State /* Method called by BFS/DFS every i iterations on each process */
\For {AllDimensionalNeighbors}
\State \texttt{n} := \texttt{getNextDimensionalNeighborID()}
\State \texttt{q} := \texttt{getLocalQueueSize()}
\State \texttt{s} := \texttt{sendQueueSizes(n, q)}
\EndFor
\State \textbf{return}
\end{algorithmic}

Figure 5.3: GDE load balancing algorithm from sending end.

in detail by Xu in [25]. Dimensional exchange methods are capable of taking into account the topology of the processors participating in the verification. The GDE methodology groups processors in the verification process into a hypercube structure. Figure 5.2 shows the structure of a hypercube for 1 through 8 processors. In each case there are \(\lceil \log(N) \rceil\) dimensions that are created where each dimension consists of two processors that have a direct link to each other. This structure is also similar to the graph coloring problem where no two edges are colored with the same color. If applied to a hypercube, no two processors are colored with the same color, thus requiring the problem to have \(\lceil \log(N) \rceil\) different colors. This network structure enables us to view the processors in an elegant manner and implement algorithms that are more communication efficient.

In the GDE load balancing scheme, each node performs load balancing (number of states in the queue) with its 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
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: \( s_n = \text{getQueueSize}() \)
4: \( q := \text{getLocalQueueSize}() \)
5: if \( q > s_n \) then
6: \( \text{sendStates}((q - s_n)/2, ID) \)
7: else
8: /* Receive happens in a non-blocking fashion */
9: \( \text{receiveStates}((s_n - q)/2, ID) \)
10: end if
11: return

Figure 5.4: GDE load balancing algorithm on receiving end.
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 [25] proves that the optimal value for creating an equilibrium state in as few iterations as possible is $\frac{1}{2}$. Figures 5.3 and 5.4 show the pseudo code for the algorithm. Every $i^{th}$ iteration, where $i$ is the balance frequency set by the user, the processor sends its 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 5.4. 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 $\frac{1}{1000}$. 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.

The advantages of using a GDE approach is that it provides a pro-active rather than a re-active scheme when compared to the global load balancing scheme discussed earlier. This feature makes the GDE scheme very applicable to the model checking problem since in parallel model checkers, some processor is bound to go out of balance with the other processors and implicit load balancing is highly effective in such scenarios. Another very important feature of the GDE algorithm is the fact that each processor only balances with its dimensional neighbors thus creating a much more elegant communication scheme compared to the global load balancing algorithm. Also, with the processors structured as a hypercube, network infrastructure topology is
also taken into account thus creating efficient communication patterns. The expected speedup with the GDE load balancing algorithm can be calculated based on two variables. Given \( N \), the number of processors participating in the verification process, and \( E \), the effective number of processors during the verification, the speedup can be calculated using the equation

\[
S = \frac{N}{E}.
\]  

To calculate \( E \), we can assume that one half of the processors are inactive after the half way mark during verification using the static partition algorithm. In such a scenario, the effective number of processors would be

\[
E = \frac{N}{2} + \frac{N}{2} = \frac{3}{4}N.
\]  

If \( E \) has the value of \( \frac{3}{4}N \) as above then the expected speedup is 1.33. The expected speedup thus varies based on the number of processors that are active during the verification and the resulting effective number of processors in the static partition algorithm.
Chapter 6

Results

6.1 Global versus GDE Speedup

Figure 6.1 shows the results that we have obtained after implementing the global load balancing algorithm on the IBM 1350 Linux Cluster. The horizontal axis is the number of processors and the vertical axis is the relative speedup when compared to the traditional static partitioning algorithm without a cache. Results have been displayed for the atomix and jordon models. As can be seen from the figure, the speedup achieved for the atomix model is higher than the speedup achieved for the jordon model. This is because, the atomix model provides a smaller state space compared to the jordon model and is not as communication intensive as the jordon model. With the added load balancing communication overhead the atomix model does not overload the communication infrastructure whereas the jordon model does; thus providing a lower speedup. This phenomenon is also noticed if we increase the number of processors and hence the communication overhead and synchronization effort involved for all processors to perform the load balancing algorithm.
Figure 6.1: Relative Speedup of the global load balancing versus the traditional static partitioning algorithm on IBM 1350 Cluster.
Figure 6.2 shows the 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 obtained, we can also see that the speedup curves are moving up as the number of processors increases, indicating that the GDE algorithm scales better than the global load balancing algorithm for the models used during testing. This is due to the efficient communication patterns created by the $\lceil \log(N) \rceil$-dimension hypercube of the network topology.

Figure 6.3 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 shown in Figure 6.2. For the NOW architecture, other models, larger in state-space and as well as state vector size were also used for testing. The queens8 and dense models, the largest in size, seemed to have achieved the highest speedup on the NOW architecture, indicating the high degree of scalability of the algorithm as the problem size is increased.

Figure 6.4 shows the speedup of the GDE algorithm on the shared memory machine architecture described in Chapter 2. The speedup here is less than the speedup achieved on the other two architectures, with initial slowdowns for the atomix model. The general trend observed in our tests indicates 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. But 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
Figure 6.2: Speedup for the GDE scheme relative to the traditional static partitioning algorithm on IBM 1350 Cluster.

other hand, the NOW architecture provides the slowest interconnect and the best speedup results as seen in Figure 6.3.

To visually observe the effects of the GDE load balancing algorithm, we collect queue size information for queues on all processors using the GDE algorithm and methods described in Chapter 4. Figure 6.5 shows the state of the queues every second after applying the GDE load balancing algorithm when using 32 processors to verify the jordon model. Comparing this to Figure 4.3, a significant difference is achieved by using the GDE load balancing algorithm. All the queues are busy at all times during the verification and the work load seems to be distributed among more processors rather than just being concentrated on a few processors. Also, the number of snapshots taken in each case is very different, with the load balanced version only requiring half the time of the static partitioning algorithm to complete verification.
Figure 6.3: Speedup for the GDE scheme relative to the traditional static partitioning algorithm on NOW architecture.

Figure 6.4: Speedup for the GDE scheme relative to the traditional static partitioning algorithm on Shared Memory Architecture.
Figure 6.5: Queue sizes for the Jordan model using 32 processors after load balancing.

Figure 6.6 shows the percentage of the time spent on the major modules in the parallel model checker. Compared to Figure 4.4 we can see that due to the load balancing, the idle time has been reduced from one third of the total time to almost nothing and the CPU time has been increased due to extra communication to improve performance. The important point to notice here is that there is no time spent waiting for states at all, which shows us that at most times during the verification each processor has some states in the local queue so they perform useful work. These results were gathered on the IBM 1350 Linux Cluster.

Figure 6.7 shows the state of the queues before load balancing using the static partitioning algorithm. Figure 6.8 shows the state of the queues on the individual processors while using the GDE load balancing scheme with a high balance frequency and the exchange parameter set to one half. As can be seen from the figures, using a high balance frequency creates a near perfect distribution of the states among the
Figure 6.6: Aggregate times for each function after the GDE load balanced scheme was implemented.

participating queues. The speedup achieved was also significant in this case. This set of results is gathered on the NOW architecture.

Figure 6.9 shows the state of the queues for 4 processors before applying the load balancing algorithm. Figure 6.10 show the state of the queues on each individual processor while using the GDE load balancing algorithm with a balance frequency that is not very high. The balance frequency is lower than the frequency used in Figures 6.7 and 6.8. As can be seen, the load is well distributed over the processors but not as well distributed as it could have been if a higher frequency had been used. The variation in the load is higher than in the previous set of figures but the general effect can be noticed very clearly.

6.2 Search Order

Using parallel searching techniques and parallel state space generation techniques the search order is modified significantly [26]. Parallel search orders only reflect to
Figure 6.7: Queue sizes for the jordon model using 8 processors before load balancing.

Figure 6.8: Queue sizes for the jordon model using 8 processors after load balancing.
Figure 6.9: Queue sizes using 4 processors before load balancing.

Figure 6.10: Queue sizes using 4 processors after load balancing.
a certain degree, what type of search each processor is performing locally using its structures. From earlier research, it has been found that using a queue like structure on the individual processors causes the parallel search order to resemble a Breadth First Search (BFS) order which has also been found to be more efficient than a Depth First Search (DFS) search for parallel model checking [26].

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 6.1 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 speedup is the ratio of the number of states saved in the GDE 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.

6.3 Queue Sizes

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 pro-active 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 reduces the strenuous memory requirements of the queue for each processor. Table 6.2 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 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 6.2. 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.

6.4 Communication Overhead

Communication overhead is incurred due to the extra communication that takes place when using the load balancing algorithm. To measure the communication overhead, counters are used to keep track of the number of messages that are being sent between processors. Techniques such as state buffering and state caching have been used to reduce the communication overhead and still achieve high performance for both the static partition and the GDE algorithms. Table 6.3 compares the average number of messages sent between any two processors in the verification using the traditional algorithm and the optimized load balanced algorithm.

From the table, we observe that for the models cache, atomix and dense, the
Table 6.2: Maximum queue size and standard deviation of maximum queue size.

| Model              | Max Queue Size | Ratio | Standard Deviation |
|--------------------|----------------|-------|-------------------|
|                    | Traditional    | GDE   | Traditional       | 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 |

Table 6.3: Average number of messages exchanged between a pair of processors on IBM 1350 Cluster.

| Model              | Traditional Algorithm | GDE Algorithm   | Percentage |
|--------------------|-----------------------|-----------------|------------|
| atomix             | 276751.41             | 254148.7        | -8.89      |
| dense              | 324927.66             | 320916.58       | -1.25      |
| 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      |
number of messages sent in the load balanced version is fewer than the messages exchanged in the traditional algorithm. This is due to the state cache that has been implemented and placed to avoid sending the same state multiple times. 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 the in the worst case only 10% extra messages are sent to achieve a speedup of 1.6 times. This communication overhead is acceptable because it provides us with balanced smaller queues and improves the error detection capabilities of the model checker.
Chapter 7

Conclusions and Future Work

From the discussion above we can highlight some major points of interest. Synchronous architectures for parallel model checking algorithms achieve higher speedup and greater performance. Parallel model checkers using the traditional static partitioning algorithm have certain inefficiencies due to a variety of factors primarily related to the partitioning function and communication schemes. This research has demonstrated that using the traditional algorithm, the queues on each process are highly imbalanced, 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 and speedup over the traditional algorithm. Using load balancing techniques such as GDE, we have successfully balanced the queues on all processors and reduced the time to verify models. Due to the non-deterministic nature of the GDE load balancing algorithm, we have also successfully changed the search order to the degree where error states in 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 maxi-
mum queue sizes obtained in the traditional algorithm. We have also shown that the communication overhead does not counteract the usefulness of GDE load balancing.

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.
LIST OF REFERENCES

[1] E. Clarke, O. Grumberg, and D. Peled, “Model checking,” pp. 9–11, 2002.

[2] E. Clarke, T. Filkorn, and S. Jha, “Exploiting symmetry in temporal logic model checking,” in Proceedings of the 5th International Conference on Computer Aided Verification. Springer-Verlag, 1993, pp. 450–462.

[3] 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. Springer-Verlag, 1992, pp. 332–342.

[4] G. Holzmann, “An analysis of bitstate hashing,” in Proceedings of 15th International Conference on Protocol Specification, Testing, and Verification, INWG/IFIP. Warsaw, Poland: Chapman and Hall, 1995, pp. 301–314. [Online]. Available: citeseer.ist.psu.edu/article/holzmann95analysis.html

[5] U. Stern and D. L. Dill, “Improved probabilistic verification by hash compaction,” in Correct Hardware Design and Verification Methods, vol. 987. Stanford University, USA: Springer-Verlag, 1995, pp. 206–224. [Online]. Available: citeseer.ist.psu.edu/stern95improved.html

[6] U. Stern and D. Dill, “Parallelizing the murphi verifier,” pp. 256–278, 1997. [Online]. Available: citeseer.nj.nec.com/article/stern97parallelizing.html
[7] F. Lerda and W. Visser, “Addressing dynamic issues of program model checking,” *Lecture Notes in Computer Science*, vol. 2057, pp. 80–100, 2001. [Online]. Available: citeseer.ist.psu.edu/549890.html

[8] R. Kumar, M. Jones, J. Lesuer, and E. Mercer, “Exploring dynamic partitioning schemes in hopper,” Verification and Validation Laboratory, Tech. Rep. 3, September 2003.

[9] H. Garavel, R. Mateescu, and I. Smarandache, “Parallel state space construction for model checking,” in *Proceedings of the 8th International SPIN workshop on Model Checking of Software*. Springer-Verlag New York, Inc., 2001, pp. 217–234.

[10] J. Barnat, L. Brim, and J. St, “Distributed ltl model checking in spin,” in *Proceedings of the 8th International SPIN workshop on Model Checking of Software*. Springer-Verlag New York, Inc., 2001, pp. 200–216.

[11] G. Behrmann, “A performance study of distributed timed automata reachability analysis,” in *Electronic Notes in Theoretical Computer Science*, vol. 68. Elsevier, 2002.

[12] C. Demartini, R. Iosif, and R. Sisto, “dspin: A dynamic extension of spin,” in *SPIN*, 1999, pp. 261–276. [Online]. Available: citeseer.ist.psu.edu/demartini99dspin.html

[13] D. Nicol and G. Ciardo, “Automated parallelization of discrete state space generation,” *Journal of Parallel and Distributed Computing*, vol. 47, no. 2, pp. 153–167, 1997. [Online]. Available: citeseer.ist.psu.edu/nicol02automated.html
[14] U. Stern and D. L. Dill, “Using magnatic disk instead of main memory in the murphi verifier,” in Computer Aided Verification, 1998, pp. 172–183. [Online]. Available: citeseer.ist.psu.edu/stern98using.html

[15] S. Ben-David, T. Heyman, O. Grumberg, and A. Schuster, “Scalable distributed on-the-fly symbolic model checking,” in Proceedings of the Third International Conference on Formal Methods in Computer-Aided Design. Springer-Verlag, 2000, pp. 390–404.

[16] D. Walker, “The design of a standard message passing interface for distributed memory concurrent computers,” Parallel Computing, vol. 20, no. 4, pp. 657–673, 1994. [Online]. Available: citeseer.ist.psu.edu/walker94design.html

[17] “Mpich home page,” http://www-unix.mcs.anl.gov/mpi/mpich/.

[18] “Pvm home page,” http://www.csm.ornl.gov/pvm/pvm_home.html.

[19] G. Geist, J. Kohla, and P. Papadopoulos, “Pvm and mpi: A comparison of features,” Calculateurs Paralleles, vol. 8, no. 2, pp. 137–150, 1996. [Online]. Available: citeseer.nj.nec.com/geist96pvm.html

[20] “Dijkstra’s token termination,” http://www.cs.panam.edu/~meng/Course/CS6356/Notes/master/node79.html.

[21] “Vv lab model database,” http://vv.cs.byu.edu.

[22] B. Hickman and R. Barr, “Reporting computational experiments with parallel algorithms: Issues, measures and experts’ opinions,” ORSA Journal on Computing, vol. 5, no. 1, 1993. [Online]. Available: citeseer.nj.nec.com/geist96pvm.html
[23] G. J. Holzmann, “The model checker spin,” *IEEE Transactions on Software Engineering*, vol. 23, no. 5, pp. 279–295, 1997.

[24] M. Willebeek-LeMair and A. Reeves, “Strategies for dynamic load balancing on highly parallel computers,” *IEEE Transactions Parallel Distributed Systems*, vol. 4, no. 9, pp. 979–993, 1993.

[25] C. Xu and F. Lau, “Iterative dynamic load balancing in multicomputers.” [Online]. Available: citeseer.ist.psu.edu/xu94iterative.html

[26] G. Behrmann, T. Hune, and F. Vaandrager, “Distributing timed model checking - how the search order matters,” in *Computer Aided Verification*, 2000, pp. 216–231. [Online]. Available: citeseer.ist.psu.edu/behrmann00distributing.html

[27] C. Joubert, “From abstract distributed model checking to concrete implementation,” in *Electronic Notes in Theoretical Computer Science*, vol. 89. Elsevier, 2003.

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

[29] W. Stallings, *Queuing Analysis*. World Wide Web, 2000. [Online]. Available: ftp://shell.shore.net/members/w/s/ws/Support/QueuingAnalysis.pdf

[30] L. Brim, I. Černá, P. Krčál, and R. Pelánek, “Distributed ltl model checking based on negative cycle detection,” *Lecture Notes in Computer Science*, vol. 2245, pp. 96–107, 2001. [Online]. Available: citeseer.nj.nec.com/article/brim01distributed.html
[31] S. Malik, Y. Zhao, C. Madigan, L. Zhang, and M. Moskewicz, “Chaffe: Engineering an efficient sat solver,” in *38th Conference on Design Automation*, 2001, pp. 530–535.

[32] H. Sivaraj and G. Gopalakrishnan, “Random walk based heuristic algorithms for distributed memory modelchecking,” in *Electronic Notes in Theoretical Computer Science*, L. Brim and O. Grumberg, Eds., vol. 89. Elsevier, 2003.
Appendix A

State Cache

Another technique to improve performance of distributed model checkers 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 [7][12]. 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 owned by different processors. To avoid this, a block of memory is allocated on each processor to function as a state cache. Only states not present in the cache are forwarded to their owning processors. The primary purpose of the state cache is to gain extra speedup and performance by reducing the number of messages exchanged between processors.

Figure A.1 shows the average number of messages exchanged between any two processors in the static partitioning algorithm using a cache of varying size for 4 processors on a NOW. Figure A.2 shows the same graph for 8 processors. The number of messages exchanged in each case does decrease as the cache size is increased from
Figure A.1: Messages sent per processor for 4 processors with varying cache sizes using a NOW.

Figure A.2: Messages sent per processor for 8 processors with varying cache sizes using a NOW.
Figure A.3: Speedup obtained using a cache of size 1 MB measured on the IBM 1350 Cluster.

100 bytes to 1 MB. We can see that the decrease in the number of messages is not significant for cache sizes greater than 1 KB. The average speedup that has been observed by using a direct mapped cache is constant but not very high. Average speedups in the region of 1.05 are gained relative to the static partitioning algorithm. Figure A.3 shows the speedup achieved using a cache of 1 MB on the IBM 1350 Cluster. The biggest contribution of using a cache is the reduction in the number of messages that are exchanged between processors. Some speedup is definitely realized with the decrease in the number of messages, but a lot of it is offset by time spent inserting, looking up and deleting states from the state cache.
Appendix B

State Buffering

With the increase in the state vectors for models used in parallel model checking, network traffic and communication patterns play a significant role in the performance of the static partition algorithm. State buffering is a technique that enables processors to reduce the number of messages exchanged between processors. Traditionally, each state is transported as an individual packet on the underlying interconnect. Using this scheme where each packet contains one state, network traffic increases exponentially as the state explosion occurs. One of the methods to avoid high network traffic is to increase the locality of the verification process in an effort to retain ownership on maximum number of states generated by a single processor. If the locality is increased, fewer states need to be exchanged between processors, as a result decreasing the number of messages exchanged on the network. However, to increase locality, one must create an intelligent and dynamic partition function which is capable of taking the nature of the state space as well as the number of processors into account. This, as we have discussed earlier, is as hard a problem as solving the reachability problem (model checking problem) itself.

State buffering is a technique that allows processors to exchange groups of states
in a single network packet [11]. By sending multiple states in one network packet, the cost of creating multiple packets, as well as transporting multiple packets over the network is reduced; thus, it creates a more network friendly algorithm. The number of states to send per packet depends on the size of the state vector for the model being verified, as well as the size of the Maximum Transmission Unit (MTU) for the underlying interconnect. For example, TCP/IP has a MTU of 1500 bytes. For a state vector of 30 bytes, 50 states can be transported in a single packet for any given destination. If $SV$ is the size of the state vector and $B$ the number of states per packet, $B$ should always equal $\lfloor \frac{MTU}{SV} \rfloor$ for maximum network throughput and efficiency. However, this could result in starvation of other processors if states are not being transported because buffer limits have not been met. Another side effect is the delay in transporting the states which could result in outdated information. Taking these factors into account, the buffer limits should be such that minimum starvation occurs in other processors due to any other processor.

For this research, state buffering is implemented and has also shown improvements over the static partition algorithm. The formula mentioned earlier is used to calculate the number of states per packet. However, with the increase in the size of the problems being verified, state buffering has proved to be insufficient for achieving better speedup and balancing queues. Apart from minor speedup improvements in the region of 1.1, state buffering fails to solve the problem of reducing queue sizes, and it does not contribute to modifying the search order in a positive manner for improving degraded error discovery. On the other hand, state buffering causes a minor delay in the communication of states to different processes; thus, creating outdated information and further degrading the error or deadlock state discovery.