5-1-2013

A Domain Specific Language Based Approach for Generating Deadlock-Free Parallel Load Scheduling Protocols for Distributed Systems

Pooja

A domain specific language based approach for generating deadlock-free parallel load scheduling protocols for distributed systems

By

Pooja Adhikari

A Dissertation
Submitted to the Faculty of
Mississippi State University
in Partial Fulfillment of the Requirements
for the Degree of Doctor of Philosophy
in Computer Science
in the Department of Computer Science and Engineering

Mississippi State, Mississippi

May 2013

A domain specific language based approach for generating deadlock-free parallel load scheduling protocols for distributed systems

By

Pooja Adhikari

Approved:

Edward Allen Luke Associate Professor of Computer Science and Engineering (Major Professor)	Ioana Banicescu Professor of Computer Science and Engineering (Committee Member)
Eric Hansen Associate Professor of Computer Science and Engineering (Committee Member)	Edward B. Allen Associate Professor and Graduate Coordinator of Computer Science and Engineering (Committee Member)
Sarah A. Rajala Professor and Dean of Bagley College of Engineering	

Name: Pooja Adhikari

Date of Degree: May 10, 2013

Institution: Mississippi State University

Major Field: Computer Science

Major Professor: Dr. Edward Allen Luke

Title of Study: A domain specific language based approach for generating deadlock-free parallel load scheduling protocols for distributed systems

Pages of Study: 179

Candidate for Degree of Doctor of Philosophy

In this dissertation, the concept of using domain specific language to develop error-free parallel asynchronous load scheduling protocols for distributed systems is studied. The motivation of this study is rooted in addressing the high cost of verifying parallel asynchronous load scheduling protocols. Asynchronous parallel applications are prone to subtle bugs such as deadlocks and race conditions due to the possibility of non-determinism. Due to this non-deterministic behavior, traditional testing methods are less effective at finding software faults. One approach that can eliminate these software bugs is to employ model checking techniques that can verify that non-determinism will not cause software faults in parallel programs. Unfortunately, model checking requires the development of a verification model of a program in a separate verification language which can be an error-prone procedure and may not properly represent the semantics of the original system. The model checking approach can provide true positive result if the semantics of an implementation code and a verification model is represented under a single framework

such that the verification model closely represents the implementation and the automation of a verification process is natural. In this dissertation, a domain specific language based verification framework is developed to design parallel load scheduling protocols and automatically verify their behavioral properties through model checking. A specification language, LBDSL, is introduced that facilitates the development of parallel load scheduling protocols. The LBDSL verification framework uses model checking techniques to verify the asynchronous behavior of the protocol. It allows the same protocol specification to be used for verification and the code generation. The support to automatic verification during protocol development reduces the verification cost post development. The applicability of LBDSL verification framework is illustrated by performing case study on three different types of load scheduling protocols. The study shows that the LBDSL based verification approach removes the need of debugging for deadlocks and race bugs which has potential to significantly lower software development costs.

Key words: Asynchronous parallel applications, Load Scheduling Protocols, Domain specific language, Model checking

DEDICATION

To my parents

ACKNOWLEDGEMENTS

I would like to thank my advisor Dr. Edward A. Luke for his direction and guidance to complete this work. It would not have been possible without his encouragement and support throughout my tenure at Mississippi State University. I am also very thankful to Dr. Ioana Banicescu for her support and suggestions in my journey as a PhD student. I would also like to thank Dr. Edward Allen and Dr. Eric Hansen for their comments and suggestions for improving my dissertation. I would like to take this opportunity to show my sincere gratitude to the Department of computer science and engineering and High Performance Computing Collaborotary (HPC) for financially supporting my graduate study. This work would have been very difficult to complete without the high performance computing infrastructure at HPC. I show my gratitude to the support staff of HPC.

I would like to thank my husband Nischal Dahal for being there for me and holding me on my downs and always believing in me. My parents have always been a source of motivation throughout my life. I am very lucky to have been showered by such love and care. I show my respect to my parents. I am very blessed to have my best friend, my sister Pragya around me to share moments of life. I would also like to thank my in-laws for their moral support in my journey.

Finally, I would like to thank every teacher, friends and relatives who has helped me reach this milestone in my career.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF CODELETS

CHAPTER 1

INTRODUCTION

Ensuring that parallel programs are free of bugs due to the possibility of non-determinism is a challenging task. It is challenging because re-running a non-deterministic program under seemingly identical conditions can produce different results including unwanted behaviors such as deadlocks and race bugs, which are rarely reproducible [33]. Note that sometimes non-determinism can be eliminated from parallel programs to reduce these undesirable side effects. However, some domains such as asynchronous load scheduling protocols, rely on non-deterministic execution in order to achieve high performance.

Load scheduling protocols are an important part of parallel and distributed applications. In many high performance computing scenarios, tasks and processors may be heterogeneous, causing variability in task completion time. This variability can cause uneven workload and significant inefficiency due to idle processors. In order to maintain high efficiency and to maximize CPU utilization, the workload has to be evenly scheduled among available processors. Dynamic load scheduling is generally adopted for this purpose in order to adjust task assignment among processors based upon their current load distribution. Task adjustment is done via task migration to other processors, and the decision for task migration is made at run time. In this scenario, a certain protocol needs to be

followed by communicating processes to execute an intended load scheduling. Communication protocols define a set of rules for task migration during load scheduling process. The effectiveness of a load scheduling process largely depends upon the effectiveness of the communication protocol. An efficient communication protocol will reduce an average response time and communication overhead that occurs during load scheduling. An asynchronous communication protocol is generally used with load scheduling for message passing systems.

A message passing system consists of multiple processing nodes, each with its own exclusive address space. Each processing nodes can either be a single processor or a shared address space multiprocessor. Interaction between the processes on different processing nodes is accomplished exclusively by sending and receiving messages. Message passing programs are often written using asynchronous paradigms. Asynchronous programs are characterized by the absence of a known bound on relative processors speeds or message transfer times. Such programs are therefore harder to reason about and can have a non-deterministic behavior due to race conditions. Due to the non-deterministic behavior, traditional testing methods are less effective at finding software faults such as deadlocks, which are examples of common catastrophic bugs seen in this mode of programming.

A deadlock is an event that occurs when processes lock resources [17]. It is possible that a process may lock more than one resource at a time. In this scenario, a dependency loop is formed whereby each process is unable to reserve the needed resources due to a cycle of requests that can never be satisfied. Occurrence of deadlocks in an asynchronous system may depend upon a specific interleaving of communicating events, and on subtle

timing between messaging and computing events. Due to such non-determinism, deadlocks are difficult to detect using traditional testing techniques such as test-case simulation. Also, the complexity of these systems usually makes complete verification prohibitively costly.

A trivial example of deadlock is shown in Figure 1.1 where two processes are sending a message to each other. Depending on the implementation of the underlying send protocol, processes may remain blocked, each waiting for the message to be received by the destination processor.

Process 0	**Process 1**
Send (to :1)	Send (to:0)
Receive (from:1)	Receive (from:0)

Figure 1.1

A trivial deadlock scenario in message passing systems

Scheduling and load balancing are two key techniques used in parallel computing systems to maximize efficiency where task running time variability would otherwise cause processors to become idle and underutilized. A number of distributed load-balancing schemes [7, 64] have been developed and such schemes are proven effective in balancing workload distributions to obtain an optimal performance. Deadlock-free load scheduling protocols are therefore very important.

Various research efforts are conducted for diagnosing deadlocks in parallel applications, especially for parallel applications written using the MPI [2] paradigm. Deadlock

detection using timeouts is an approach that has been implemented in tools such as Marmot [49] and MPI-CHECK [54]. However, this method is prone to giving false positive result for applications that have long computation phases. Another approach to deadlock detection is through the analysis of execution traces of the parallel application. One disadvantage of these tools is that they cannot precisely characterize the operations that constitute deadlocks with wild card receiving calls.

1.1 Objective of the dissertation

One approach that can eliminate deadlocks is to employ model checking [23] techniques. This technique performs an exhaustive exploration of all possible interleavings of execution of a system and does a complete verification in principle. However, this approach requires a separate verification model of a candidate system that is written using specific verification languages [23]. Verification is then performed in the model instead of the original system. The development of a verification model of the program is a tedious and error prone procedure. Following are the limitations of applying model-based verification technique with full confidence.

- Lack of programming expressiveness for common programming structures such as function calls, structures, and arrays in existing protocol verification languages, making a verification model not a conservative representation of its original system.
- Lack of confidence in verification result, since the original system is not used for verification purpose. Original system may still contain bugs even if the verification model is verified as error-free.
- Need to keep verification model up-to-date with modifications of protocol specification.
- Need to have an in depth knowledge about verification methodologies and its corresponding languages.

Model based formal verification is useful if the semantics of an implementation code and a verification model are represented under a single framework, such that the verification model closely represents the implementation, and that the automation of a verification process is natural.

In this dissertation the limitations of model-based verification are addressed with a new language, called Load Balancing Domain Specific Language (LBDSL), that integrates the implementation and verification of asynchronous load scheduling protocols under a single framework tuned to the development of parallel asynchronous protocols for message passing systems. By limiting the scope to a rather narrow domain, it is intended that such language would help reveal the non-deterministic bugs present in the system, which otherwise would be hidden and would reveal themselves only after the production is completed for large systems. The intention is not to claim that this is the only approach that will lower software development costs in terms of lowering verification costs. The only intention is to demonstrate that this is one of the approaches that can be used in the efficient development of parallel load scheduling protocols (PLSP). It can be one step towards developing a robust verification framework for PLSP.

1.2 Approach to solve the problem stated

The domain specific language (DSL) approach used in this thesis is not a new concept. Some of the research efforts that use a DSL concept to obtain a verified application have used in Promela++[11], Teapot[19] and Rebecca[72]. However, its applicability to verify the parallel applications, especially PLSP, has not been explored yet. Simply stated, a do-

main specific language is a mini-language that is designed to solve problems in a particular domain. This approach has been suggested as a means to develop reliable software [39]. It can be viewed as a programming abstraction that contains notation supporting a particular application domain and is based upon the relevant concepts and features of an application domain. Behaviors of a specific domain are projected in its specification framework. Figure 1.2 provides a general concept about domain specific language.

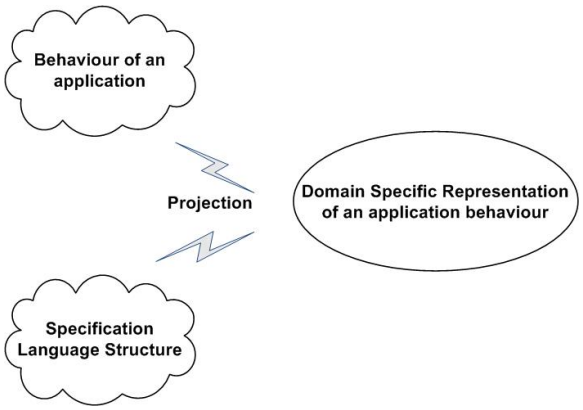

Figure 1.2

Concept of a Domain Specific Language

This dissertation shows, through case studies, that the components of LBDSL help simplify the understanding and identification of the essential parts of PLSP. LBDSL allows viewing and differentiating the relevant and irrelevant components of a protocol in terms of detecting deadlocks and race conditions, and offers guidance towards developing a strong verification model. The implementation of a PLSP specification in LBDSL also helps to remove the unwanted computational details that have no effect in determining the

correctness of the protocol, and can complicate the verification process if present in the verification model.

The design of the LBDSL language is done in an iterative fashion and is based upon the case studies conducted. The first case study is the implementation of a probe-based centralized load scheduling protocol. This case study helps to identify the essential components of the LBDSL language to properly represent the domain of PLSP. Two further case studies, multithreaded-load scheduling protocol and hierarchical load scheduling protocol, helped to realize new features to be added to the language.

The design of the LBDSL provides notations for representing processes, message communicated between the processes during task distribution, and the medium of communication between the processes. It also provides a way of representing the verification irrelevent features of PLSP as an opaque component such that they have no effect in the verification process. The language structure also supports the automatic generation of both an executable code in a desired high level language and a verification model in a specific verification language. The model checking approach is used as a verification back-end for the LBDSL framework. By combining model checking concept with a domain specific language approach, it is hoped that the overall productivity for developing a verified and robust PLSP will be greatly improved.

Therefore, this dissertation shows that the associative cost required to generate a separate model, independent of protocol implementation, can be decreased if the semantics of the protocol are embedded into a language abstraction. The embedded information about the logical structure of the protocol can facilitate an automatic extraction of a verification

model. The process of protocol development and verification is then possible in a single step. It also shows that the components of the LBDSL facilitate identifying the essential features of a protocol, and therefore have the potential benefit of helping the ddesign of parallel load scheduling protocols more productively.

The major contributions of this dissertation can be summarized as followings:

- The concept of using a domain specific language to be used in the design of an error-free parallel load scheduling protocols for distributed systems is studied.
- A new domain specific language is presented that unites twin goals of protocol verification and error-free protocol construction under a single framework.
- The practical implementation of this approach is demonstrated by developing three different types of load scheduling protocols in the LBDSL language.

1.3 Summary

It should be clear that the motivation of this work is not to develop a safety critical system, but to develop a mechanism which can be used to reduce the cost associated with the development of an error-free parallel system. In this regard, the LBDSL verification framework is different than other methods dedicated to develop a safety critical system, such as B-method [63]. B-method provides tools and a process of developing verified software through a process of formal specification and refinement. The LBDSL is not addressing safety critical applications where software development costs are secondary to correctness, but rather reduces the overall cost of developing high performance software. The LBDSL language framework integrates the implementation and verification of parallel asynchronous load scheduling protocols. It reduces the complexity of testing and debug-

ging protocols for subtle deadlock and race bugs which has the potential to significantly lower the software development costs.

In the following chapter, an overview of existing debugging and verification techniques for parallel applications is provided, and their limitations that motivates the need for a protocol construction framework is highlighted. This chapter also justifies the reason for which a language-based approach is appropriate. In the following chapters, an introduction is provided to the LBDSL language and techniques used by the LBDSL verification framework to compile the LBDSL program. The usability of LBDSL is then demonstrated by implementing three types of dynamic load scheduling protocols, namely: the probe-based centralized load scheduling protocol, the multi-threaded centralized load scheduling protocol, and the multithreaded hierarchical load scheduling protocol. Threats to the validity to these case studies are analyzed next followed by conclusions and future work.

CHAPTER 2

LITERATURE REVIEW

In many high performance computing scenarios, work load has to be evenly scheduled among available processors to maintain high efficiency and to maximize CPU utilization. Dynamic load scheduling is generally adopted to adjust task allocation based upon the processors' current load distribution. In this scenario, asynchronous communication protocols are generally utilized to coordinate collecting current load status information and to direct the migration of load between processors.

Non-determinism introduced by the protocol asynchrony is a main source of catastrophic bugs such as deadlocks and race conditions in load scheduling protocols. System non-determinism makes it very difficult to verify that an asynchronous load scheduling algorithm is error free. It is difficult to verify that the protocols satisfy the safety requirements, such as freedom from deadlocks and race bugs, and the termination at valid end states. Traditional approaches such as test case simulation, are tedious and error prone. They tend to produce false negative result due to their incapability of capturing all possible executions of an asynchronous application. A number of research efforts have been carried out to develop verification methods for generating error-free asynchronous application.

This chapter will start by discussing the parallel load scheduling protocols being used in high performance computing. Then, various research efforts in developing the verification tools and techniques for parallel applications will be discussed.

2.1 Load Scheduling in Distributed Systems

The interest on distributed computing has grown considerably in recent years. A distributed system [79] consists of independent processing nodes linked through a communication network and appears to the users as a single system. Processing nodes communicate with each other in order to achieve a common goal such as minimization of execution time and maximization of resource utilization. The processing nodes in a distributed system may be heterogeneous in terms of processing power and work load. In addition, tasks assigned to these processors may widely vary in their computational complexity. This heterogeneity in processing nodes and tasks may lead to unbalanced load in the system. Therefore, the important issue in such systems is the development of the effective techniques to distribute the parallel load to the processing nodes efficiently in order to achieve the desired goal. The method of load distribution among the processing nodes is generally termed as load scheduling. In a distributed system, a load scheduling algorithm determines the processing node responsible for executing tasks. Several research efforts are found in the literature that demonstrate the usefulness of load scheduling algorithms. This section discusses some research efforts that are used for load scheduling.

2.1.1 Static Load Scheduling

In static scheduling[68], assignment of tasks to processors is done before program execution. This type of algorithm assumes that the information regarding the processing resources and task execution time is already known at the compile time. These algorithms are non-preemptive which means the task is always executed in the processor to which it is assigned. The goal of static load scheduling is to achieve efficient execution time while minimizing the inter-processor communication. This type of algorithm is useful if the complexity of workload is known prior to execution and does not vary during runtime such that initially balanced partition gives the optimal result. Chaco[38], METIS[10] and SCOTCH[61] are some examples of static partitioning tools that support static load scheduling.

2.1.2 Dynamic Load Scheduling

Dynamic load scheduling does the redistribution of tasks among the processors during execution. This redistribution is performed by transferring the load from heavily loaded processors to lightly load processors to utilize the available resource. Dynamic load scheduling is useful for applications where processor workloads vary during runtime. Optimal performance is achieved in such applications by dynamically distributing the workload among processors. Dynamic load scheduling algorithms are used in many applications in high performance computing, ranging from adaptive mesh refinement to contact detection algorithms[13]. PLUM and Charm++ are the example of some tools that supports dynamic load scheduling.

2.1.2.1 PLUM

PLUM[59] is used to balance the imbalance caused by mesh adaption among processors in a distributed system. This tool dynamically balances the load among the processors with a global view. PLUM framework performs Mesh adaption, remapping, repartitioning and processor assignment. These actions are executed rapidly and efficiently to reduce significant overheads during numerical simulation. Its data redistribution model predicts the remapping cost and determines whether or not the gain from balanced workload distribution offsets the cost of data movement. This framework has proved to be an effective dynamic load balancing tool and demonstrates the significant performance improvement even when the number of processors is large.

2.1.2.2 Charm++

CHARM++[47] is a portable and concurrent object-oriented system based on C++ in which work loads are dynamically created during its execution. The implementation of dynamic load scheduling algorithms in CHARM++ manages irregular parallel computations as a result of dynamically created work loads. Charm++ implements the following two types of dynamic load scheduling algorithms: centralized and hierarchical. In centralized load scheduling algorithm, one of the processors is chosen to be scheduler. Other processors send newly generated tasks to the scheduler. The scheduler buffers new tasks in a prioritized queue and assigns them to worker processors. Worker processors periodically update their load information to the scheduler by either periodically sending a message or by piggybacking a message along with a new task request. The scheduler uses load

scheduling policies to maintain the work load of each processor within a range of allowable load. The main drawback to this approach is the excessive memory requirement and bottleneck for the central scheduler as it has to store the tasks sent by all worker processors.

In a hierarchical load scheduling algorithm, CHARM++ groups a number of processors into clusters. Each cluster consists of its own scheduler. The processors in each clusters sends the task and load information to its corresponding scheduler. Load schedulers are responsible to distribute load and maintain an allowable load among the processors within a cluster. The schedulers of all clusters communicate with each other to balance load and priorities among them to prevent task imbalance in clusters. Schedulers exchange their load information and migrate tasks to other processors (or clusters) based upon the information exchanged. If all schedulers are balanced, they exchange a fixed number of highly prioritized tasks. This mechanism is called prioritized load balancing. If the schedulers are not balanced, then the scheduler with a heavier load sends the tasks to the scheduler with lighter load. This mechanism is called task-load balancing.

2.1.2.3 LBtool

LBtool[20] is a general-purpose dynamic load balancing tool that is developed at Mississippi State University. Applications with computationally intensive parallel loops use this tool to perform load balancing. This tool is based on the MPI library and is suitable for those applications that use the distributed pool of independent tasks. LBtool allows applications to dynamically schedule the execution of chunks of loop iterates and directs the transfer of data between the processes. This uses the dynamic load scheduling policies

discussed above to compute the chunk size during redistribution. LBtool can be used to parallelize sequential applications with parallel loops or to implement load scheduling in an existing parallel application.

A dynamic load scheduling algorithm is composed of two important factors: load scheduling policies and communication protocols. Load scheduling policies are used by load scheduling algorithms to balance the load among the processors and minimize the overhead in computing the schedule of iterations. Load scheduling policies can be static or dynamic. Lets assume there are N independent iterations to be scheduled among P processors. Static load scheduling policies schedules N/P iterations to be performed by each processor. However, there are different variations of dynamic load scheduling policies such as fixed sized chunking [50], guided self scheduling [62], factoring [40], fractiling and adaptive weighted factoring[18].

The type of dynamic load scheduling algorithm is determined by where the policies are executed. In a centralized algorithm, these policies are executed by the central scheduler. In a distributed algorithm, all processors in the system execute these policies. Processors frequently communicate with their neighbors to exchange their load information. Centralized load scheduling algorithms suffer from scalability problems, especially in machines with a relatively small amount of memory. On the other hand, distributed load scheduling algorithms tends to yield poor load balance on large machines due to the incomplete information of the system. Hierarchical load scheduling algorithms is the balance between these two extremes where the processors are divided into different autonomous groups, which are organized in a hierarchy, decentralizing the load balancing process.

Other forms of load scheduling approaches have also been discussed in the literature. A gradient load scheduling algorithm is one of them, in which task movement from heavily loaded processors is directed by a pressure gradient that is established by the task requests from nearby idle processors [53]. The balanced system is achieved by successful task migration to requesting processors. Similarly, a drafting communication protocol for dynamic task migration is another form of load scheduling approach [58]. A three level system, namely heavy, normal, and light is used to categorize the processors based upon their work load. Processors communicate only with a group of processors termed as candidate processors. A lightly loaded processor requests a heavily loaded processor to send the bid for task migration. Tasks are migrated to the lightly loaded processor after it receives a select message from that processor. Multi-threaded load scheduling is an important approach being discussed in literature [80]. This approach allows multiple processes to exist within the context of a single processor.

Communication protocols have an important role in dynamic load scheduling[74]. The decision of task re-distribution is done at run time and is based on the current loading information of processors. The effectiveness of a dynamic load scheduling hinges on the effectiveness of the communication protocol. Various communication protocols have been proposed for dynamic task migration. There are basically two types of communication for message passing systems: synchronous and asynchronous [44]. A synchronous communication requires the sender and receiver to wait for each other to transfer a message. In this protocol, the sender is not allowed to contribute unless the receiver acknowledges the receipt of the message. Synchronous communication contributes to undesired and un-

controlled waiting and even deadlock. Asynchronous communication delivers a message from sender to receiver, without waiting for the receiver. An important advantage of asynchronous communication is that it allows overlapping of computation with communication since a processor does not need to wait for the completion of a communication. Asynchronous communication protocols are desired to be error-free as their performance has a great impact on load scheduling process. Different testing and verification techniques that are being used to obtain an error-free communication protocols are discussed in the following chapter.

2.2 Debugging Techniques for Parallel Asynchronous Systems

This section reviews the research efforts in developing the deadlock detection tools that facilitate the debugging and verification of parallel applications.

2.2.1 Debugging Tools for MPI Applications

Message passing interface (MPI)[2, 3] is a commonly used standard used in the development of parallel applications in high performance computing. Unfortunately, the richness of the MPI standard and its inherent non-deterministic behavior has made it difficult for programmers to use it efficiently and correctly. Numerous research efforts have been conducted to develop debugging tools and mechanisms for MPI applications such as, MPI-Check[54], MPIDD[31], UMPIRE[78], MARMOT[49], and MUST[66]. Following subsection gives a brief overview about these debugging tools.

2.2.1.1 MPI-Check

MPI-Check performs by changing the MPI program where MPI calls are replaced with modified arguments. The arguments provide information about the line number where the MPI call has been made and the information is stored in a database. Therefore, the debugging process in MPI-Check occurs in two phases: program instrumentation and execution of instrumented program under the control of MPI-Check. MPI-Check performs deadlock detection by creating dependency graphs from calls made for point to point and collective communication. It reports a potential deadlock using timeouts where the dependency graph is not resolved in a user specified time. Deadlock-detection using timeouts is prone to false positive result for applications with long computation phases

2.2.1.2 MPIDD

MPIDD performs deadlock detection by analyzing the execution trace of parallel program. It has a central manager that traps MPI calls using PMPI. The central manager runs as another MPI process. The trapped information trace is sent to this process using MPI calls. The central manager then creates the dependency graphs of execution trace and performs a depth first search for deadlocks in the dependency graph. The major disadvantage of this tool is that it cannot precisely characterize the operations that constitute deadlocks with wild card receiving calls.

2.2.1.3 UMPIRE

UMPIRE uses timeout mechanism and dependency graphs for deadlock detection. It performs deadlock detection by analyzing the execution trace of a parallel program. It can detect deadlocks caused by blocked calls and deadlocks involving spin loops over non-blocking completion calls. Similar to MPIDD, it has a central manager to trap MPI calls using PMPI. Unlike MPIDD, it runs as a separate process and communicate with other processes using shared memory. Similar to MPIDD, it also cannot precisely detect the deadlocks related to wild card receives.

2.2.1.4 MARMOT and MUST

Similar to MPI-Check, MARMOT also uses timeout mechanism to detect deadlocks in MPI programs. It traps the communication calls using the MPI profiling interface. However, a dependency graph is not created using this tool. Similar to MPI-Check, this tool is also prone to giving false positive results for applications with long computation phases.

MUST is a MPI runtime error detection tool that combine the capabilities of MARMOT and UMPIRE. It uses Profiling MPI for the underlying infrastructure along with the set of fine grain modules that implement MPI checks. It can execute correctness checks either in an application process or in extra processes to offload the analysis. It resolves the scalability issues of UMPIRE and MARMOT.

2.2.2 Deadlock Detection of Parallel Multi-Threaded Applications

Nondeterminator [29] is an on-the-fly race condition detection tool, designed to debug parallel programs written in a CILK [15] language. Cilk is a general purpose programming language designed for multithreaded parallel computing. Nondetermintor for CILK programs is aimed at locating determinacy races. Determinacy races occur in parallel applications due to race conditions, making a program behave non-deterministically. The original Nondeterminator was a serial program that was used to debug a parallel multithreaded application. Parallelization over the serial approach has been done on the latest version of its race detector. The debugging process of nondeterminator starts by instrumenting each read and write statement in a user's program. This allows the debugging CILK compiler to perform determinacy-race checking at runtime. An execution of an instrumented CILK program occurs in depth first fashion, generating a directed acyclic graph (DAG) with nodes representing the computation and edges representing the processing threads. The compiler of CILK performs race-checking action on the generated DAG when read, write, and parallel control statements are executed. A debugging tool keeps track of a series-parallel relationship between the threads to detect race conditions on DAGs, giving an approximation of the existence of data-race conditions. Tracking a series-parallel relationship is a process of determining the logical operation of current thread (series or parallel) with certain previously executed threads. Nondeterminator typically verifies that a program is race-free and produces same end behavior for a given input data. However, it does not verify that a program is free of race conditions for those input data sets that are not used in the debugging process.

2.3 Deadlock Detection using Formal Verification

Formal verification methods [14, 9] hinge on the use of mathematical logic for verifying the correctness of a program. It views a program as a mathematical object with well defined behavior. While the debugging tools described in the last section explores some of the possible behaviors of system, the formal verification approach explores all possible behaviors of a system and gives confidence that the system is completely verified in principle. There are mainly two approaches to formal verification: proof-based approach and model-based approach.

2.3.1 Proof-Based Formal Verification

Theorem proving belongs to the category of proof-based approachs. The basic ingredients of the theorem proving method are axioms, theorems, and inference rules. In theorem proving, the system under verification is modeled as a set of logic formulae τ called axioms. Property specifications that the system should satisfy are represented as theorems ϕ. Inference rules are then applied to find a proof that $\tau \vdash \phi$. The inference rules represent the verification path. PVS[60] is an example of a theorem proving tool. PVS provides an environment for writing specifications of a system and developing proofs. It has been used to verify several parallel and distributed protocols such as verification of protocol designed for a dynamically scaled aircraft [30]. The theorem proving technique has the ability to model parametric systems or infinite state systems. However, this method is not fully automatic. It requires a considerable amount of user intervention and technical expertise to create the specification for establishing a logical deduction. The cost of using this tech-

nology in industry is very high. There are many success stories where a complex system has been verified by theorem provers, but industrial acceptance of this technique is still minimal. Unlike model checking methods, this method lacks the generation of a counter example. When a proof attempt fails, this technique fails to clarify whether the system is not correct or the theorem has not correctly modeled the system.

2.3.2 Model-Based Formal Verification (Model Checking)

In early 1980's Clarke and Emerson proposed a model checking method for automatic verification of finite state concurrent systems[23]. The model checking technique performs an exhaustive depth first search on the finite state model of a given system, and automatically verifies whether the model meets the given specification. This technique is used in the verification of hardware or software systems. Generally, the specification contains the safety requirements such as absence of deadlocks and race conditions.

Model checking algorithms can only be applied in a finite state system[21]. Properties to be verified against the system are expressed in temporal logics [67]. Model checking is based on temporal logic which states that a particular property is not statically true or false in all the states of a model rather it can be true in some states of the model and false in others. System properties expressed in temporal logics change their truth value along with the system transition from state to state. Such a transition system is represented by a model μ and the properties are represented by a formula ϕ. An efficient and flexible search procedure is used to find correct temporal patterns in finite state graphs of concurrent systems. The search process enumerates all reachable states and possible transitions of the mathe-

matical model to verify whether or not μ satisfies ϕ. This process always terminates with a yes or no answer if sufficient memory is available. This method has some advantages over the theorem proving process. It requires no proof deduction and is easier to implement. It has the ability to generate a counter example in case the verification fails, which helps not only to show that the system contains bugs, but also to trace the source of the bugs.

Different model checking tools are available to facilitate model checking. These tools take the abstract model of the system written in a verification language as input and the property specifications represented in temporal logic. Spin [36], MURPHI [25], SMV [23] and BLAST [12] are the examples of some of popular model checking tools.

2.3.2.1 Spin

Spin is a general model checking tool that facilitates the design and verification of asynchronous systems [65]. Promela is the input language of this tool. The verification model of the system is written in Promela specification language and is used by Spin to prove the correctness of process interaction in an automated fashion. Promela supports the modeling of asynchronous distributed algorithms as non-deterministic automata. Properties to be verified against the verification model are expressed as Linear Temporal Logic formulas. During the verification, these are negated and converted into Buchi automata which are done automatically by Spin.

Spin uses the explicit state space enumeration mechanism. It explores all possible execution states of the model and checks the specified properties against them. In addition to model checking, it can also operate as a simulator, demonstrating one possible execution

path of the system. It uses mechanisms such as partial order reduction, state compression, and bit state hashing to reduce the number of states to be explored, and thus speeding up the verification process. By default, Spin can verify following safety properties: freedom from deadlock, proper termination of processes at valid end state and complete transfer of messages to correct recipients.

2.3.2.2 MURPHI

MURPHI is another model checking tool used in Microprocessor industry for verifying the cache coherence protocols. Its input language is also called Murphi. The main building blocks of this language are global variables and guarded commands in a `guard->action` notation. Similar to Spin, MURPHI also use an explicit state space enumeration mechanism. The state exploration is performed as a depth first search or breadth first search of the state space. Murphi was originally developed by Professor David Dills group at Stanford. Many versions of Murphi have since been developed by the same group and other research group. An MPI based distributed implementation of Murphi called Eddy_Murphi [57] is one such example that can do the parallel and distributed model checking.

2.3.2.3 SMV

The SMV [23] model checker is mostly used for the verification of hardware systems. Similar to Spin and MURPHI, it is also an explicit state model checker, but it uses ordered binary decision diagram(OBDD)[16] based symbolic model checking approach to reduce state space required to store enumerated states. A binary decision diagram is a data struc-

ture that is used to represent a boolean function. BDDs are generally used as a compressed representation of sets or relations in order to reduce the state space requirement during formal verification.

2.3.2.4 BLAST

BLAST [12] is another model checking tool that is used for the verification of software applications. It is based on the principle of counter example guided software abstraction known as CEGAR refinement. The CEGAR refinement process automates the model extraction process using a top-down stepwise abstract refinement approach. BLAST does not support the non-deterministic programs.

2.4 State Explosion Problem in Model Checking

Model checking methods seem to be an ideal choice for verifying asynchronous communication protocols; however, the state explosion problem makes it difficult to employ it for the complete verification of a system. The state explosion problem states that the global states of a concurrent system with many processes can be enormous. This problem is mainly observed in the systems having many components that interact with each other during the execution. In the model checking process, the system to be verified is represented as a kripke structure, a type of non deterministic finite state machine. An example of a kripke structure of a coffee vending machine is given in Figure 2.1

In this example, vertices represent states of a system, edges represent transition functions between states, and the label of vertices represents a set of atomic propositions that states have to satisfy. For a system consisting of n non-interacting processes each with k

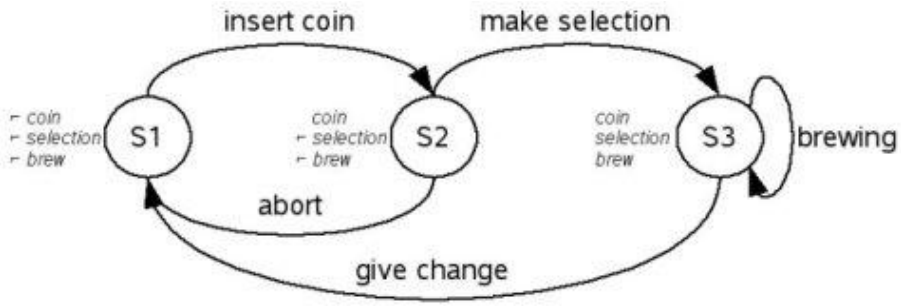

Figure 2.1

Kripke Structure of Coffee Vending Machine

local states, a possible number of reachable states is k^n. For example, a state transition system with n bit counters have 2^n states [22]. Complete verification of an asynchronous system using a model checking method is therefore expensive. A properly abstracted verification model should not suffer from state explosion problem. An abstract representation of a system can either be a model with small state space which can be fully verified, or a model with large state space, which cannot be verified under available system resources. Therefore, it is very important to have an in-depth knowledge of an application to design its verification model for efficient model checking. In addition to domain knowledge, it is also required to have very good concept knowledge about model checking tools and an understanding of the limitation of model checking process. Generally, the system to be verified is written in high level language like C++ and Java but the specification language of model checkers have a very low level of expressiveness compared to commonly programming idioms. Lack of expressiveness of specification language makes it hard to represent the complex semantics of a system in an abstract representation. Model extraction is also a manual process for classic model checkers such as Spin. Manual model extraction is error-

prone and expensive for large systems. The verification result of the abstract model can produce false positive results because the model may not be a proper representation of the real system. Manual model extraction also leads to an iterative process of updating both model and implementation with changes in system specification. The necessity of a model checking process, requiring the models in specific input language of model checkers, is another limitation of the model checking approach.

2.5 Automatic Model Generation in Finite State Verification

The model checking approach can be more attractive if the manual model generation step can be completely eliminated. There has been research on automated model extraction approach. The main objective of automatic model extraction process is to automate generation of a verification model. In this approach, a model extractor is designed that can generate a verification model from given specification and verify the system at the same time. A system specification can be in some specific languages or in high level languages such C++ or Java. An algorithm to automatically generate a Promela model from the system specification written in C using two parsers exists, where the first parser extracts the control structure of the system and the command boundaries [37]. Second parser builds an abstract model of the system with the help of a manually created lookup table and a model template. The lookup table contains source code in C on the left and its corresponding abstract representation in Promela on the right. The model template contains required data declarations and an outline of the required process declarations. The outline is filled later with detailed behavior specifications while extracting a model. This approach automates

the process of model generation covering 75% of code leaving only 25% of code to be written manually [35]. Feaver[34] is another tool that applies similar kind of approach to auto generate a verification model directly from source code and verify them. It is a model extractor for C or C++ programs. Automated model extraction process has also been successful for programs written in Java. Bandera [24] and Java pathfinder [32] are some model generation tools that works for Java programs.

2.6 Model checking of MPI Applications

Finite state verification of MPI applications has already been studied in literature [5]. The Verified Software Laboratory(VSL) at the University of Delaware conducts research on verification of MPI applications using model checking [9], [69], [70]. Similarly, Formal verification group at the University of Utah conducts research on formal verification of MPI applications at code level. The following subsection elaborates the on-going research in detail.

2.6.1 Representation of MPI primitives in Promela structure

Addressing the issues related to modeling of MPI semantics in a Promela structure is an important research topic and is considered by the VSL[71]. Only a subset of MPI primitives that includes blocking point to point communications, excluding the wildcard receive primitives such as MPI_ANY_SOURCE and MPI_ANY_TAG, are considered for this study. MPI_Send is a blocking send primitive and does not return until the message transfer operation is complete and the send buffer can be re-used. To model this behavior in Promela for channel size greater than zero, a blocking statement is introduced until the

Codelet 2.1

MPI_Send in Promela structure

```
1    inline MPI_Send(schan, msg){
2      schan!msg;
3      if ::1 ->empty(schan) ::1 fi
4    }
```

channel is empty. Let schan be the channel and msg be the data, then standard MPI_Send can be modeled in Promela as showm in Codelet 2.1.

Here, the second statement represents the send operation and the third statement represents the blocking statements. Third statement is true until the channel is empty. Similarly, MPI_Recv and MPI_Sendrecv can be modeled. Like general applications, verification of MPI applications can also suffer from the state explosion problem. Although the Spin model checker implements state reducing algorithms to address this problem, it is too generalized and ineffective for the domain of MPI applications. The complexity of the semantics of MPI programs makes state reduction algorithms applied in Spin ineffective. Additional work has to be done to limit the state space. Different theorems have been deduced which makes the model checking process more tractable for asynchronous systems. These theorems also help in pinpointing potential deadlock situations.

2.6.2 Techniques to address state explosion problem for MPI applications

The first theorem states that, “For any standard MPI point to point communication without wildcard receive, the model is said to be deadlock free if it is synchronously deadlock-free”. This theorem addresses the problem of large state space of asynchronous

systems, compared to their synchronous counterparts. It is one way of applying abstraction to asynchronous MPI applications. Similarly, introduction of a barrier to the system may affect the verification process. Another theorem states that, “If a model having a barrier is deadlock-free, then the original system is also deadlock-free ”. Applications of these theorems during the modeling phase helps reduce the state space of MPI applications and finite state verification technique can be applied effectively.

Theorems to effectively model wild card free MPI applications have also been introduced [9]. In these theorems, introduction of barriers to reduce the state space and modeling of barriers in an abstract way are discussed. Channel size can also increase the number of states visited caused by the varying number of pending messages in the channel. This problem can be addressed by modeling the channel depth in such a way that the number of pending messages never exceeds a limit.

2.6.3 Representation of MPI Wild- Card receives and Non-blocking communication in Promela

The MPI application domain is further expanded to explore the verification of halting properties in MPI programs that have wildcard-receives and MPI programs that executes non-blocking operations [69]. A parallel system is said to be halted if all processors of the system have stopped executing either due to normal termination or due to deadlock condition. Freedom from deadlock is one of the halting properties. Theorems about wild card free communications are extended to cover non-blocking wildcard communications. The Urgent algorithm is introduced that deals with MPI wildcard receive MPI_ANY_SOURCE by moving between synchronous and buffering mode search for commutative executions

in the state space. The Urgent algorithm is similar to a partial order reduction. It is claimed to have drastically reduced the number of states explored even when partial order reduction methods are applied and improved the model checking process. This algorithm does not require explicitly modeling all the possible executions of MPI application.

2.6.4 MPI-SPIN

In order to make modeling of MPI processes more effective the general version of Spin is extended to a new library called MPI-Spin [70]. The Model checking process verifies the model instead of the actual system. The Verification model of the MPI programs need to be written in Promela. However, Promela does not contain the programming expressiveness required for MPI applications and does not contain notions about MPI primitives. A deep expertise on both the semantics of MPI primitives and the shortcomings of Promela language is required to design an effective verification model of MPI application in the Promela language. The main purpose of MPI-Spin is to ease the design of the verification model for MPI applications. It ensures correctness of MPI primitives representation in Promela semantics. It provides many commonly used MPI functions, constants, and types including those used for non-blocking point to point communication. By default, this extension of Spin verifies some generic properties as follows:

- Freedom from deadlock.
- Two incomplete requests do not exists whose buffers intersect non-trivially.
- Total number of outstanding requests never exceeds a specified bound.
- When MPI_finalize is executed there are no request objects allocated for and there are no buffered messages destined for the calling processes.
- Size of the incoming messages is never greater than the size of the received buffer.

2.6.5 ISP

Insitu Partial Order (ISP) is another tool developed for formal verification of MPI applications [76]. This tool is developed at Formal Verification Lab at University of Utah. ISP is similar to Spin, as it also verifies the safety properties of state space representation of the application. However, it does not require a separate verification model to perform verification. It performs code level verification, meaning it verifies all relevant interleaving of a MPI application by replaying the actual program code. ISP has been used to successfully verify a MPI application having up to 14,000 lines of code for deadlocks and assertion violations.

2.7 Domain Specific Languages

There are basically two types of languages for designing a system: generic languages and domain specific languages [77]. Generic languages provide a framework to write a general solution for many problems, but the solution may be sub-optimal. On the other hand, a domain specific language (DSL) provides a specific framework that provides a better solution for a smaller set of problems.

A DSL is a programming abstraction that provides notation supporting a particular application domain and is based upon structures and features of the application domain. The concept of the DSL has been also used for the automatic verification of parallel and distributed applications. A new language is designed to support the development of error-free applications. As discussed, the major drawback of the model based formal verification method is the lack of expressiveness and inconsistencies in the behavior of a resulting

model with respect to its original system. And therefore, there is always a lack of confidence in the verification result obtained from the model checking process. A DSL based finite state verification approach can address this problem in terms of providing a sufficient expressive power to make the specification task reasonably straightforward. The concept of domain specific language has been used to ease model checking complex applications. Promela++, Teapot and Rebeca are some example of such domain specific languages.

2.7.1 Promela++

Promela++ [11] is a language based approach designed to construct an error-free communication protocol. Communication protocols considered in this work are based upon layered protocol stacks. The layered protocol stacks approach divides the protocol into distinct layers and each layer performs some unique function independently of the other layers. The layering concept is similar to that of TCP/IP or OSI communication protocols. The Promela++ is an extension to Promela which allows the protocol developer to specify the control flow of protocol layer in C like language. Promela++ compiler is designed to convert the Promela++ code into Promela. The safety properties are specified in the same way as they are specified in Promela. This approach does not guarantee the absolute correctness of the program; however, it is used to trace major logical errors. Generation of C code from the Promela++ is done with the help of event handlers for each protocol layer. Each event handler constitutes a co-routine that asynchronously processes the events. These co-routines generate appropriate corresponding C code from Promela++

specification. Promela++ has two major deficiencies. It does not support explicit memory allocation primitives like malloc() and it does not have any support for timers.

2.7.2 Teapot

Teapot [19] is another domain specific language designed to construct verified cache coherence protocols. Cache coherence can be defined as the consistency of data stored in local caches in shared resources. It is an important issue in parallel and distributed systems, in which local replicas of shared data are created to improve scalability and performance. A cache coherence protocol is an algorithm that manages the required consistency of the cached copies. Teapot has two major functionalities: first, it translates the teapot protocol into executable C code and second, it generates an input code for the model checker Murphi. Murphi then detects violations of the specifications in the protocol. A teapot program consists of a set of states with each state specifying a set of message types and the action to be taken in response to a particular message. A distinct feature of teapot is to structure the control flow of protocol using continuation, rather than a flat state machine. Using continuation, it is not required to complete all the actions of one state before moving to another state. The program execution can move back and forth between states by suspending the action of one state and then moving the program control to another state and executing its corresponding actions and later resuming the actions of suspended state. The overall system of Teapot consists of the following: a teapot compiler which translates the protocol written in the Teapot language into either C or Murphi based upon desired functionality. The Teapot compiler consists of a C backend and a Murphi backend. C support routines

and Murphi support routines are also required when converting to respective C code and Murphi code.

2.7.3 Rebeca

Rebeca[73] is an actors[6] based modeling language with a formal foundation whose objective is to bridge the gap between formal verification approaches and real applications. It can be considered as a reference model for concurrent computation, based on an operational interpretation of an actor model. Object oriented concurrent systems can be designed using this modeling language. The main advantage of using this modeling language is twofold. First, it allows an appropriate and efficient way for modeling concurrent and distributed systems. Second, it allows verifying such systems for correctness using formal verification. Formal verification using Rebeca is supported by a set of verification tools. Initially, Rebeca model translation tools were provided to translate rebeca code into a specification language of model checkers such as Spin and NuSMV and verify the resulting model. Since 2005, it is supported by a direct model checker based on Modere[43]. Modere uses modular verification and translation techniques to reduce the state space of given system. This makes it possible to verify complicated reactive systems. Modere also uses the general state space reduction techniques such as partial order reduction and symmetry reduction.

2.8 Summary

This chapter covers different types of communication protocols that are used in load scheduling algorithms and verification techniques to obtain error-free load scheduling pro-

tocols. Centralized and distributed are two broad categories of load scheduling protocols. Distributed schemes tend to be more efficient than centralized schemes because of sharing of the responsibility of managing load among many processors. Other schemes, such as gradient scheme and prioritized scheme, are variants of distributed scheme. Asynchronous communication protocols applied in load scheduling techniques are prone to subtle bugs such as deadlocks and race conditions. Traditional testing mechanisms such as exhaustive test case simulation are also of limited help. Non-deterministic behavior shown by asynchronous systems makes it hard for the traditional methods to track all possible errors. Hence, the generation of an efficient and correct communication protocol is a challenging task.

Communication protocols can be synchronous, loosely synchronous or asynchronous. Deadlock detection is easier in synchronous communication than asynchronous. Asynchronous applications show the non-deterministic behavior and generate different results in different executions even with the same set of input data. MPI is the most common paradigm for writing parallel applications. There are also different debugging tools for MPI applications. Some of these debugging tools require modification of MPI programs and some require one extra processor to debug MPI applications. MPI debuggers also help in debugging for general bugs found in MPI applications but cannot guarantee to cover all possible interleaving of the application. Formal methods, a proven verification mechanism for hardware and software, are also being also used for the verification of asynchronous load scheduling protocols. This approach uses formal methods of mathematics for verification of applications and provides a wide range of coverage on executions. Two approaches

of formal methods as mentioned in section 2.3 are: proof-based and model-based. Proof-based approach uses inference rules and deduction logic for verification which requires technical expertise for proof deduction and makes verification process harder especially for large systems. Model checking is a simpler approach and is used in analyzing the correctness of concurrent reactive systems. Various model checking tools are available to facilitate the model checking process. The model checking process works on a finite state model of the application system. Applying model checking techniques requires the original system to be scaled down to an abstract representation of the model. Since the model checking is done on the abstracted model, the generated model should contain all possible behaviors of the original system. Different model checkers have different specification languages. A designer of the model should have expertise in the original system, model checking tools, and also should be aware of the limitations of model checking process. The need to generate an abstract model of the underlying system is a limitation for the model checking process because creating a verification model manually is both difficult and error prone. Inefficient model extraction also leads to a state explosion problem which is a major obstacle for implementing model based formal verification in asynchronous systems. Various research efforts to address the state explosion problem and an efficient model extraction are also discussed in this chapter. Abstraction is the most commonly used approach, where the size of the state transition graph is reduced by either eliminating the variables irrelevant to properties of interest or by mapping actual values of the system with a small set of abstract values. The resulting model will have a small state space which is feasible for model checking. Other methods, such as partial order reduction or symmetry, are used along with

the abstraction mechanism. Model checkers need to understand the semantic knowledge of the system to efficiently implement reduction techniques. For example, partial order reduction is ineffective while verifying MPI applications using Spin because Spin cannot handle the complex semantics of the MPI paradigm. A lot of manual intervention is required while designing an abstract model although the verification process is automated. Ongoing research of how to model a system to make Spin handle the semantics of MPI is covered in section 2.5. Model checking MPI applications are important because parallel systems in high performance computing are commonly written using the MPI paradigm. In any case, human errors (due to manual model extraction) can result in false negative result in the verification process. This may increase the cost of verification, and therefore reduce the usefulness of model checking. The requirement of manual effort in abstracting the design also hampers the use of model checking of actual systems. Different techniques are proposed in the literature to automatically generate an abstract model from the given specifications.

Tools such as Bandera and Javapathfinder have been very successful in automatically generating a model from Java specifications while minimizing manual intervention. Although these automated model generation tools reduce the model extraction cost, it is not completely automated and requires a great deal of human expertise. Successful verification of the application requires the model to be a conservative representation of original system. The model-based verification cannot guarantee that a model is a proper conservative representation of an actual system and therefore verification result cannot be completely relied upon. Implementation languages are not designed for verification. The extracted models

are either prohibitively expensive to verify or the models need significant culling in the extraction process, thus limiting the confidence in the extracted representation. Model based formal verification is useful if the semantics of an implementation code and a verification model is represented under a single framework. The verification model should closely represent the implementation making the automation of a verification process natural.

The domain specific approach discussed in section 2.7 discusses the works done to bridge the semantic gap between implementation languages like C, Java, and verification languages like Promela. Promela++, a domain specific language and an extension of Promela, is the best example to use in describing this approach. Designing a new language abstraction for a specific domain, which in this context are asynchronous communication protocol, allows a sufficient expressive power to make the protocol specification relatively easy for the protocol designer. In the case of protocol verification, Promela++, Teapot and Rebeca, discussed in section 2.7, allows an efficient C code generation from the language abstraction and also allows the verification of protocol correctness. Domain specific approaches correctly address semantic gap between actual system and the model extracted in the model based formal verification approach. Single program written in certain domain specific language can be compiled to efficient C code and can be used for the automatic verification of protocol correctness against property specifications. An important advantage of the DSL approach is that it reduces the cost of model extraction and also increases the effectiveness of the model checking process. The model extracted is a conservative representation of underlying implementation because of the embedded semantic knowledge of the system in the DSL.

CHAPTER 3

LOAD BALANCING DOMAIN SPECIFIC LANGUAGE(LBDSL)

Domain specific languages (DSLs) are application oriented, special purpose language that provide notions and constructs tailored towards a particular application domain. Asynchronous load scheduling protocols are considered as a problem domain for the design of Load Balancing Domain Specific Language (LBDSL). These protocols suffer from more complex non-deterministic behavior, which is difficult and expensive to debug. Often times, apparently correct protocols turns out to have subtle problems that only reveal themselves on large systems where failures are difficult to unwind. Such late discovery of software faults makes the development of robust and sophisticated load scheduling protocols more costly than one might expect. In an approach where the implementation of load scheduling protocol is verified to be free of deadlocks and race conditions, its development cost should be significantly reduced. However, there is no such framework that combines the necessary components such as ease of programming, modularity in protocol specification, and finite state verification.

In this chapter, a domain specific framework is proposed to ease the development and verification of load scheduling protocol for distributed systems. The structure of LBDSL is inspired from the actor model [6], a mathematical model of concurrent computation that

treats “actors”as a universal primitive of concurrent digital computation. This chapter is organized as follows. In section 3.1, an introduction to load scheduling in distributed systems is provided. In section 3.2, design requirements for a new domain specific language are discussed. In section 3.3, a description about the actor model is provided. Section 3.4 introduces the main components of the LBDSL language. In section 3.5, syntactic definition of LBDSL components is provided. Finally section 3.6 concludes this chapter.

3.1 Load scheduling in Distributed Systems

A distributed system consists of multiple autonomous processing units that communicate by message passing through a computer network. Each processor has their own memory and processing power to execute a task. They interact with each other to solve a common problem. A problem assigned to a distributed system is divided into many independent tasks and is assigned to one or more processors of the system. However, the processing power these processing units may be heterogeneous, executing the assigned task in a different speed. Similarly, tasks assigned to them may also be heterogeneous in terms of their computational complexity, some taking more time to execute than others. The heterogeneity in processing power and heterogeneity in load lead to an unbalanced system resulting to a distributed system with poor efficiency. Load scheduling is a process of improving the performance of parallel and distributed systems through a redistribution of load among the processing elements, in order to maximize the resource utilization and minimize the execution time [8]. There are two broad categories of load scheduling algorithms: static and dynamic [42].

3.1.1 Static Load Scheduling

In static load scheduling, assignment of tasks to processors is done before program execution begins. Information about the task execution times and processing resources is determined at the compile time. A task assigned to a processor is never re-distributed to another processor at run time. Static algorithms are non-preemptive. They never initiate a context switch from a running process to another process. Examples of such algorithms are round robin algorithms [48], randomized algorithms [1] and central manager algorithm [8].

3.1.2 Dynamic Load Scheduling

In dynamic load scheduling, distribution of tasks to processors takes place during run time[41]. Current workload of processors is taken into consideration for maximum utilization of CPU time. Task is redistributed when a load imbalance is detected in the system. Dynamic load scheduling may be carried out by a central authority [52] or may be distributed among the processing elements [47]. This dissertation is focused to the development of dynamic load scheduling protocols. An implementation of both centralized and distributed dynamic load scheduling protocols are discussed in [47].

In a centralized load scheduling, one of the processors is chosen to be the scheduler. Other processors send newly generated tasks to the scheduler. Scheduler buffers new tasks in a priority queue and assigns them to the worker processors. Worker processors periodically update their load information to the scheduler either sending the message separately

or by piggybacking it along with a new task request. The scheduler uses the load scheduling policies to maintain the work load of each processor within a range of allowable load.

In a distributed load scheduling, processors are grouped in clusters. Each cluster consists of its own scheduler. Processors in each cluster send the task and load information to its corresponding scheduler. Load schedulers are responsible to distribute load and maintain an allowable load among the processors within a cluster. The schedulers of all clusters communicate with each other to balance load and priorities among them to prevent task imbalance in clusters. Schedulers exchange their load information and migrate tasks to other processors (or clusters) based upon the information exchanged. Since the decision of task re-distribution is done at the run time and is based on the current load information of processors, effectiveness of a dynamic load scheduling hinges on the efficient and error-free communication protocol [75].

Processors in a distributed system can communicate either synchronously or asynchronously [45]. A synchronous communication requires both the sender and receiver to wait for each other to transfer a message. Asynchronous communication does not require such synchronization. A message transfer is completed from sender to receiver without waiting for the receiver to be ready. The advantage of asynchronous communication is that sender and receiver can overlap their computation because they do not wait for each other.

The main purpose of this dissertation is to provide a language-based framework for construction of deadlock-free asynchronous load scheduling protocols for distributed systems. The following section discusses the design requirements for the new domain specific language.

3.2 Design Requirements

The main objective of a domain specific language is to build the semantics of a problem domain into a mini-language so that software developers can easily design a software for that domain using the language. Object identification and abstraction are the most important steps in the development of a specification language, where nouns and verbs to describe a domain are identified.

The domain of dynamic load scheduling protocol is defined by two factors: a load scheduling policy and a communication protocol. Load scheduling policy schedules the execution of iterates in chunks with variable sizes. At any instant, it computes the size of chunks and identifies a proper recipient process during task distribution. A dynamic load scheduling policy uses current state information for task distribution. On the other hand, a communication protocol defines the communication pattern during load scheduling. It determines the process of interaction between system components to achieve the objectives defined by the load scheudling policy. Therefore, a load scheduling policy selects where a load should be distributed whereas a load scheduling protocol directs the transportation of the load to a processor identified by the policy.

In a centralized load scheduling protocol, a central scheduler executes the policies and manages load scheduling. Worker processors coordinate with the central scheduler to obtain a task for execution. In a distributed load scheduling protocol, the role of the scheduler is distributed among many processors. The processors coordinate with each other to determine the chunk size at any particular instant. A third type is the hierarchical load scheduling protocol, which lies in between centralized and distributed load scheduling protocol.

In this type, a tree of schedulers exists where leaves are the workers. Each scheduler in a tree controls the sub-domain. Information exchange and decision making occurs along the scheduler tree. In summary, communication protocol defines the type of load scheduling protocol.

An asynchronous load scheduing protocol allows overlapping of communication between the processors with the execution of load scheduling policies or work computation. Such type of communications rely on non-deterministic execution in order to achieve high performance. Unfortunately, non-determinism introduced by asynchronous protocols can lead to catastrophic bugs such as deadlocks and race conditions which are difficult to detect. A load balancing domain specific language (LBDSL) is designed to ease the generation of deadlock-free asynchronous load scheduling protocols. Therefore, LBDSL framework should facilitate an easy representation of asynchronous protocol specification in its structure. The single representation should be sufficient to generate an implementation code in C++ (a high level language) and a verification model in PROMELA specification language. The verification model can be then verified using SPIN model checker for possible deadlocks and race conditions. Since the goal of LBDSL is to support automatic verification of the protocol specification, the design of LBDSL should provide a mechanism to identify the verification relevant components during protocol specification. For example, the result of load scheduling policies does not play any role in the correctness of load scheduling protocols. In this thesis, deadlock and race-condition free protocols are termed as correct protocols. Correctness of a protocol is independent of a particular choice of chunk size and recipient process computed by the load scheduling policies. It is, therefore, not re-

quired to represent the load scheduling policies in the verification model. However, the rules of communication between the processes during task distribution play an important role to determine the protocol is deadlock free and should be considered during protocol verification.

The design of LBDSL should provide notations for representing processes, message communicated between the processes during task distribution and the medium of communication between the processes. It should also provide a way of representing the load scheduling policies as an opaque component, such that they have no effect in the verification process. It should also allow the modular specification of load scheduling protocol so that the components can be used by creating their instance. The language structure should support the automatic generation of both an executable code and a verification model. The single representation of protocol specification in LBDSL should be sufficient to automatically generate an implementation code in a desired high level language and a verification model in a specific verification language.

The design of LBDSL is inspired from the actor model[6]. A summary about the actor model is provided in the next section.

3.3 Actor Models

The actor model is a mathematical model of concurrent computation that treats "actors"as the universal primitives of concurrent digital computation. Similar to objects in an object oriented system, the actor model adopts the philosophy that everything is an actor. However object-oriented software is executed sequentially while the actor model is inher-

ently concurrent. Each actor has an address and a behavior associated with it. An actor can influence the actions of another actor only by message passing. The recipient actor can act to an incoming message by the following three ways:

- Send a finite number of messages to other actors.
- Create a finite number of new actors.
- Determine the behavior to execute for the next message it receives.

Actors execute these actions in a concurrent and non deterministic manner. Two messages sent by an actor can arrive in any order. An actor can send messages only to actors whose addresses it knows. An actor can obtain the address of another actor either in a message it receives or the addresses of actors it creates. Therefore, when an actor sends a message to another actor, it also includes its address in the message.

The actor model can be summarized by having an inherent concurrency in computation among actors, dynamic generation of new actors, inclusion of actor address in messages and interaction between the actors through asynchronous message passing.

3.4 Introduction to LBDSL

LBDSL is an actor-based language and is considered as a platform for developing load scheduling protocols for distributed systems. Formal verification approaches are used to ensure the correctness of protocols developed using this language. The main components of LBDSL language are: Processes, chunk of iterates, message types and communicating structure. In addition, LBDSL also provides commonly used programming constructs such as constants and variables, conditional and loop statements, and basic mathematical

operations. The idea is to provide a language-based infrastructure to easily represent a load scheduling protocol specification in a correct way and then generate a verified implementation code from it. The LBDSL provides a mechanism for embedding C++ code in LBDSL programs. This allows integration of load scheduling polices and communication protocol in a single framework. Role of each LBDSL components are described in the following subsections.

3.4.1 Processes

LBDSL is similar to the actor model which consists of independent active objects known as processes. The processes communicate by exclusively sending and receiving of message types in a non-deterministic manner. The processes are composed of process variables, a set of states and transition functions. Any number of processes can be instantiated. During protocol execution, each process must be mapped to a processor of a distributed system. Multiple processes can be mapped to a single processor in a multithreaded load scheduling protocol that supports preemption.

Any number of process variables can be declared with their scope local to the process. The declared variables must be initialized before they are used. After variable initialization, processes begin their execution from the start state. Every state but the end state consists of actions to execute in that state and a transition function to next state. States of LBDSL processes are labeled. State labels should be declared before they are used by the processes. The declaration is global and any process can use the declared state label. However, it is

not necessary that every process should use all state labels. In other words, a process can use only a subset of declared state labels.

Actions associated with each state of a process can be message transmission from other processes or can be execution of load assigned to that process. Processes always make transitions to the next state defined by the transition function. A process can be at only one state at a time and the state is termed as current-state. It makes a transition to a new state when triggered by a receive event or a change in the value of a process variable. State definition of a process is local to a process and is invisible to other processes. End state of a process defines the terminating condition for the process execution. A process always terminates at the end state when the terminating conditions are satisfied.

Every process in LBDSL should be associated with a processtype. This value is mainly useful for message communication in a multi-threaded architecture. Suppose a source processor(Say, I) is executing two processes(Say P and Q). Processor address of both of these processes will be the same. Suppose another processor(Say, J) is sending a message to process P of processor I. J will, then, use the processtype of P to ensure that the message will be received by a process P, not Q. Processtype is used by LBDSL communication to ensure message is received by a correct process.

LBDSL also allows to define a special type of process in order to initialize the system enviroment before executing load scheduling. Unlike regular processes, this process does not contain states and transition function. It is not necessary to use this process while implementing every load scheduling protocol. However, if this process is defined, it will be executed by all the processors in the system before executing load scheduling protocol.

3.4.2 Chunk of Iterates

Communication between the processes performing the load scheduling is always associated with sending and receiving of independent workloads or iterates. A group of iterates in a iterate space is defined as a chunk. A chunk is an abstraction of the dynamic partitioning of the iterate space. There are different ways to define a chunk either defining a starting point and an ending point, or by defining starting point and number of iterates from the starting point. In LBDSL, a chunk is defined by the starting point of iterates in a iterate space and total number of iterates from that starting point. Besides this information, sometimes it is required to communicate estimated execution time of iterates, address of the destination processor, estimated size of iterates and the action associated with iterates. All this information defines the meta data of the chunk being communicated.

3.4.3 Messagetypes

Load scheduling protocol assumes that the pool of tasks is distributed among processors. Different types of messages need to be communicated based upon the degree of imbalance in the system. In one scenario, when the processes are homogeneous, it is sufficient to inform workers about their current workload by sending the meta data of a chunk. In another scenario, when processes are heterogeneous and workload assigned to one process need to be redistributed to another process, actual chunk is also required to be communicated along with the meta data. These different types of messages to be communicated during load scheduling process are represented as messagetypes in LBDSL.

There will be two message types for the scenario described above. Messagetype defined for the first scenario is responsible for handling the communication of chunk metadata. Messagetype defined for the second scenario is responsible for handling the communication of both metadata and the actual chunk. Load information, either only metadata of chunk or both metadata and the actual chunk, should be serialized before sending to another process. Upon receiving the message, destination process deserializes the load information. The mechanism of message serialization and deserialization must be defined in a message type definition. Messagetypes of LBDSL are, therefore, handlers used by processes to communicate different types of load information to other processes.

A message type definition takes meta data of chunk as its input and performs the necessary action. Every messagetype definition should always associate with two tasks: serialization of message before sending and deserialization of message after receiving. Besides these, any number of tasks related to the communication of load information can be associated with a message type.

3.4.4 Communication Structures

Communication between the processes is carried out exclusively by sending and receiving of messagetypes through directional process queues. In order to model asynchronous message passing, the capacity of a process queue is greater than 0. Each process is associated with one process queue. A message instance to be communicated is defined by a messagetype name, source process, destination process, processtype of destination process and metadata of workload associated with the messagetype. Whenever a communication

is invoked, messagetype specified in a message instance associates itself with its definition and prepares the load information based upon the type of communication. For example, if the communication is related to sending of load information, then serialization of the load information is performed. On the other hand, if the communication is related to receiving of load information, then deserialization of the load information received is performed. A process always makes transition to a new state after receiving a message. A transition to a new state is guarded by the action defined under metadata received. It has to be noted that the action defined in the metadata of workload is actually the state label. Besides sending and receiving, a process can just check for messages without actually receiving them. This checking operation can be either a blocking operation or a non-blocking operation. A blocking operation will not return until it senses there is some message for its owner process. Whereas a non-blocking operation just tests for message and resume its task if there is no message for its owner process. If there is a message, the checking operation can return the source and size of the expected message. This concept is useful when the recipient process is receiving variable sized message and cannot anticipate the size of message it will be receiving. In this case, size of the message need to be known beforehand the actual communication to allocate the enough space for the buffer holding the received message. Communication structures use processtype to ensure that messages has reached to the proper destination.

3.4.5 Variables, variable types, constants and control structures

LBDSL supports the basic data types such as int, bool, double and char. Like structures in C++, LBDSL also allows one to define user defined data types. A user defined data type is a group of data elements under one name. These data elements are known as members of the defined data type and can have different data types.

LBDSL also introduces the concept of variabletypes. The idea of variable types is to allow the distinction in the usage of variables during their definition. Various variabletypes defined in LBDSL are as follows:

- Compute variabletype: These variabletypes play a role in the computation of load and execution of load scheduling policy. Variables defined as compute variabletype can only be used in the embedded code. These variabletypes can be associated with any basic data types such as integer, boolean, and double that are supported by C++ and also can also be associated with user defined data types.
- Convey variabletype: These variabletypes play a role in the communication process. These variabletypes can be associated with integer and boolean data types and user defined data types.
- Decision variabletype: These variabletypes are used in the decision making process within the conditional and loop statements. These variable types can be associated with either integer type or boolean type.
- Storage variabletype: These variabletypes represent arrays and buffers. These can also be associated with any basic data types supported by C++ as well as user defined data types.

All variabletypes but the storage variabletypes should be initialized during variable declaration. LBDSL also supports the definition of constant values. Values defined as constant can be associated with only the basic data types allowed in LBDSL.

LBDSL supports the concept of conditional programming and loop programming. Conditional programming is a feature of LBDSL that executes different computations or

actions depending upon whether a programmer-specified boolean condition evaluates to true or false. This type of programming selectively alters the control flow of the program based on the given conditions. Control flow refers to an order in which individual statements or actions are executed or evaluated. An example of conditional programming is the `If-Else` statement in C++.

Similarly, loop programming is a feature of LBDSL that executes a sequence of statements which is specified once but which may be carried out several times in succession. Total number of executions is controled by specifying the number of iterations.

Lastly, LBDSL also supports two basic mathematical operations: addition and subtraction.

3.4.6 Code Embed

As mentioned in the section 3.2, a load scheduling protocol executes the load scheduling policies, computes the workload and provides rules of communication between the processes. The role of load scheduling policies is to compute the chunk size and determine the recipient process. Such computations are not relevant in terms of protocol correctness and need not be specified in terms of LBDSL. Similarly, computation of workload is not relevant in determining that a protocol is free of deadlocks and race conditions. LBDSL allows abstracting out computations such as, calculation of a chunksize, allocation of a buffer, computation of workload that do not affect the rule of communications as a black box. Such computation details are embedded in their original C++ code in LBDSL program. Embedded code does not contribute to the verification of the protocol.

In the above section, an introduction to various components of LBDSL is presented. The following section describes how these components are structured in LBDSL.

3.5 Syntax of LBDSL components

The structrue of the LBDSL components models the communication structure of a load scheduling protocol. Communication structure is separated from the routines executing load scheduling policies and other computations such as, address allocation and iterate execution. Routines for such computations are embedded as a block of C++ code in the LBDSL program. The embedded code is not checked for correctness by the validator. This technique to delineate the communication structure from computation is used in the language translation process. The result of language translation is a verification model in Promela and a complete implementation code in C++.

A simple foreman-worker example is provided to describe the syntactic structure of LBDSL components. In this example, a worker process starts the communication by sending a message to the scheduler. It terminates after the communication is complete. The scheduler process starts by waiting for the message from worker. The scheduler terminates itself after receiving the message.

In the following subsections, the syntactic structure of LBDSL components is provided. Note that every statement in an LBDSL program is defined as a rule and is preceded by "`$`" sign. Statements preceded by this sign are LBDSL-specific components and are processed to generate corresponding code in C++ and Promela language by the LBDSL language translator.

3.5.1 Structure for defining Processes

The structure of a process construct is divided into three parts: header, body and footer.

The header of a process construct begins with a keyword `Begin_Process` followed by a unique process name and input arguments. Each process must have a distinct name and can take any number of input arguments. Line 1 of Codelet 3.1 and line 1 of Codelet 3.2 are the examples of declaring a process header. Processes defined in these examples have `Foreman` and `Worker` as their process name and take `myRank` as an input argument. Here, `myRank` is the address allocated to the corresponding processes. This input argument is a convey variabletype identified by keyword `conveyV`. Footer of a process construct is represented by the keyword `End_Process`.

The body of a process construct is composed of process variables, states and transition functions. Lines 2-23 in Codelet 3.1 represents the body of the process `Foreman`. Similarly, lines 2-17 in Codelet 3.2 represents body of the process `Worker`.

Lines 2-5 of Codelet 3.1 declares the process variables. Some of them are convey variabletypes which will play a role in the communication process. The rests of the variables are decision variabletypes and are used by the conditional statements. Similarly, in lines 2-4 of a Codelet 3.2, process variables of different variabletypes are declared. Values of the process variables that are associated with the basic data types are initialized to the initial values except for user-defined data types.

States are defined after the declaration of process variables. Every state of a process is labeled. State label declaration is represented by the keyword `Enumerate_State`. Any

Codelet 3.1

Process definition in LBDSL

```
1  $Begin_Process Foreman(convey<int> myRank)
2   $datatype conveyV<Chunk> newChunk;
3   $datatype conveyV<MessageInfo> mInfo;
4   $datatype conveyV<int> (msgSrc, -1);
5   $datatype decisionV<int> (gotWork, 1);
6
7
8   $Start_State GET:
9     $Poll_Wait(ANY_SOURCE, SCHD, &mInfo);
10    $Update(msgSrc, mInfo.Source);
11    $SetState(RETRIEVE);
12
13  $Next_State RETRIEVE:
14                $ReceiveMessage(ChunkMetaData,myRank,
15                  msgSrc,SCHD,&newChunk);
16    $SetState(newChunk.chunkAction);
17
18  $Next_State WORK:
19    $Update(gotWork,0);
20    $SetState(TERMINATE);
21
22  $End_State TERMINATE:
23    $ConditionToTerminate(gotWork==0);
24 $End_Process
```

Codelet 3.2

Process definition in LBDSL

```
1  $Begin_Process Worker(convey<int> myRank)
2    $datatype conveyV<Chunk> newChunk;
3    $datatype conveyV<int> (msgSrc, 0);
4    $datatype decisionV<int> (gotWork, 1);
5
6    $Start_State REQUEST:
7      $GetChunk(newChunk,0,10,0,0,WORK);
8      $SendMessage(ChunkMetaData,myRank,
9                              foreman,SCHD,&newChunk);
10     $SetState(WORK_COMPLETE);
11
12   $Next_State WORK_COMPLETE:
13     $Update(gotWork,0);
14     $SetState(TERMINATE);
15
16   $End_State TERMINATE:
17     $ConditionToTerminate(gotWork==0);
18 $End_Process
```

Codelet 3.3

State label enumeration in LBDSL

```
1  $Enumerate_State = {
2          GET, RETREIEVE, REQUEST,
3          WORK, WORK_COMPLETE,
4          TERMINATE
5  };
```

number of unique state labels can be declared. Global declaration of state labels is given in Codelet 3.3. In this example, six state labels are declared.

Definition of each state consists of a set of tasks to be executed. The scope of a state definition is local to the process.

Processes begin their execution from the state identified by the keyword `Start_State` followed by the state label. Other states follow the `Start_State` and are defined by the keyword `Next_State`. Keyword `End_State` defines the terminating state. `End_State` also defines the terminating condition of the process execution. Terminating conditions are defined using decision variabletype using keyword `ConditionToTerminate`.

Two processtypes are defined namely, `SCHD` and `CLNT`. They are assigend to constant value 0 and 1 respectively.

In Codelet 3.1, process `foreman` begins the load scheduling process at `Start_State` with the state label `GET`. In this state, process `Foreman` waits for a message. Upon receiving the message, it makes transition to a new state defined by the transition function represented by the keyword `SetState`. Process in state `RETRIEVE` receives the message and set the received label as a next state. In state `WORK`, it updates its terminating condi-

tion to true. Notice that every state has a transition function except for the `End_State`. `End_State` defines the terminating condition using `ConditionToTerminate` construct.

Worker process executes the similar behavior. In state `REQUEST`, it request for task from foreman and moves to state `WORK_COMPLETE`. In state `WORK_COMPLETE` it sets the terminating condition to false, moves to the `End_State` nad terminates.

3.5.2 Defined structures in LBDSL

LBDSL provides two predefined structures namely, `ChunkInfo` and `MessageInfo` which are discussed in this chapter. Users can only use these structures but cannot modify them.

3.5.2.1 Structure `Chunk`

As mentioned in the SubSection 3.4.2, processes in a load scheduling protocol communicate the meta data about chunk of iterates. LBDSL provides a built-in structure called `Chunk` to allow users easily represent such information in LBDSL language. The structure of `Chunk` is given in Codelet 3.4.

This is a user defined data type and is defined using keyword `newType`. Members of the `Chunk` structure are defined as follows:

- `chunkStart`: This member can hold an integer value and represents the chunk starting point in a iterate space.
- `chunkSize`: This member can hold an integer value and represents the number of iterates in a chunk from `chunkStart`.
- `chunkAction`: This member can hold an integer value and represents the action associated with this chunk. Value associated with this member is a state label. A

Codelet 3.4

Structure `Chunk`

```
1 newType Chunk{
2   int chunkStart;
3   int chunkSize;
4   int chunkParam1;
5   double chunkParam2;
6   int chunkAction;
7 };
```

process upon receiving the messagetype, makes transition to the state with the state label received as `chunkAction`.

- `chunkparam1`: This member can hold an integer value. This member can be used to convey information such as size of iterates or address of a process depending upon the situation.
- `chunkParam2`: This member can hold a floating point value. This member can be used to convey information such as estimated execution time of the chunk, and ratio of workload depending upon the situation.

An instance of `Chunk` structure should be declared to define the chunk information. LBDSL provides a construct `GetChunk` to assign values to an instance of `Chunk` structure. `GetChunk` takes name of the `Chunk` instance and values to define the meta data of iterates as its input arguments, and outputs an enumerated `Chunk` instance. An example of applying this construct is given in line 7 of Codelet 3.2. In this example, `newChunk` is an instance of `Chunk` and rest of the arguments will enumerate the members of `newChunk`.

3.5.2.2 Structure `MessageInfo`

Similar to `Chunk` structure, LBDSL defines another defined structure, `MessageInfo`, to represent specific attributes of the message being exchanged. `MessageInfo` is defined

Codelet 3.5

Structure `MessageInfo`

```
1 newType MessageInfo{
2   int Source;
3   int Tag;
4 };
```

by `Source` and `Tag` as its members. Both of these members can hold integer values. As mentioned in section 3.4.3, a process can just check for the messages without actually receiving it. Member `Source` of this structure will return the source of the message to be received. And, member `Tag` of this structure is used the by communication structure to check whether a process is allowed to receive this message. Syntactically, MessageInfo is described as shown in Codelet 3.5.

A new instance of `MessageInfo` should be declared to access the values. Information about message source can be then obtained by accessing the value of `Source` from that instance. Similarly, information about message tag can be obtained by accessing the value of `Tag` from that instance. Like `Chunk` structure, this is a builtin structure in LBDSL. An example of using this structure is given in line 3 in Codelet 3.1. In this example. an instance of `MessageInfo` is declared as `mInfo`. In line 10, its member `Source` is accessed from an instance of `mInfo` which holds the address of source process.

3.5.2.3 Built-In handlers

LBDSL also provides some builtin handlers that can be used as utilities during protocol specification. Following seven handlers are defined in LBDSL.

- `GetMessageLength`: This handler takes an instance of `MessageInfo` as input and returns the length of a message represented by that instance.
- `GetComputeSize`: This handler takes the total number of unexecuted chunksize assigned by the scheduler as input. Worker processes use this handler to compute a size of chunk to compute at any instant.
- `GetSubDomainSize`: This handler is required during hierarchical load scheduling protocol implementation. A local scheduler uses this handler to get information the size of its sub-domain. This handler takes as input the address of local scheduler and the maximum allowed size of a subdomain to compute the subdomain size.
- `GetLocalSchedularAddress`: This handler is also required during hierarchical load scheduling protocol implementation. A worker process uses this handler to determine the address of its local scheduler. This handler takes as input the address of worker process, and the maximum allowed size of a subdomain to compute the address of local scheduler.
- `GetNumGroups`: This handler is also required during hierarchical load scheduling protocol implementation. This handler takes as input total number of processors in the system and maximum allowed subdomain size specified by the user. It returns the total number of subdomains in the system.

3.5.3 Structure for defining Messagetypes

Similar to process definition, messagetype definition also consists of three parts: header, body and footer. Header of a messagetype begins with keyword `Begin_Message` followed by a distinct message name and input arguments. Messagetype definition takes an instance of Chunk data type as its input argument. Footer of a messagetype is represented by the keyword `End_Message`.

Body of a messagetype is composed of defining variables and modules for performing serialization and deserialization of chunk information. Structure of `ChunkMetaData` messagetype is shown in Codelet 3.6. The defined messagetype consist of two modules: `sendmessage` and `receivemessage`. Each module begins with a keyword

Codelet 3.6

Messaegtype definition in LBDSL

```
$Begin_Message ChunkMetaData(convey<Chunk> *MetaData)
  $datatype conveyV<int> (pSize,0);
  $datatype storeV<unsigned char> (buf);

  $Begin_Module SendMessage
    $reSize(buf, pSize);
    $packMetaData(*MetaData, buf, pSize);
  $End_Module

  $Begin_Module ReceiveMessage
    $Update(pSize, *MetaData.chunkSize);
    $reSize(buf, pSize);
    $unPackMetaData(buf, pSize, *MetaData);
  $End_Module
$End_Message
```

`Begin_Module` followed by module name and end of the module definition is identified by the keyword `End_Module`.

In this example, first module performs the serialization of metadata information and is defined by keyword `packMetaData` that performs serialization. But before this module is called, a buffer should be prepared to hold the serialized data. Keyword `reSize` is used to allocate the buffer space. The input arguments for `packMetaData` are: an instance of chunk, destination buffer which holds the serialized data and the size of the metadata parameters.

In the same example, the second module performs the deserialization of the received message. Similar to serialization, buffer size should be allocated before it receives the serialized data. In this example, expected size of message to be received is already known

and is assigned in `chunksize` member of `Metadata` instance. Construct `resize` uses this value to allocate the buffer space. Data received in this buffer is then deserialized and enumerated in the instance of Chunk Structure. The input arguments of data deserialization construct takes the following parameters as input arguments: buffer holding the serialized data, size of the buffer and an instance of chunk to which the resulting deserialized data are enumerated.

The definition of a messagetype is responsible for serialization and deserialization of both meta data of chunk and actual iterates. Similar to metadata, serialization and deserialization of actual iterates is abstracted by constructs `packLoad` and `unPackLoad`. The input arguments of `packLoad` constructs are chunk starting point, number of iterates from starting point to be transferred, the iterate space, destination buffer, size of destination buffer and pointer returning the address of serialized buffer as its input arguments. The input arguments of `unPackLoad` construct are same but deserialization is done in the value obtained in the buffer and enumerated starting from `chunkStart` up to the `chunkSize`.

3.5.4 Communication Structures

LBDSL provides communication operations `SendMessage` and `ReceiveMessage` for exchanging messages between processes. Sending a message in LBDSL is represented by the keyword `SendMessage` followed by input arguments: messagetype to be communicated, address of the source process, address of the destination process, processtype and an enumerated instance of the `Chunk` structure. Process `foreman` is sending

messagetype ChunkMetaData associated with chunk instance newChunk to process msgSrc with processtype CLNT using SendMessage in line 8 of Codelet 3.2.

The structure for receiving a message is similar to that of sending a message. Receiving of a message is abstracted by the keyword ReceiveMessage which takes the following input arguments: messagetype to be received, address of the destination process, address of the source process, processtype and an instance of structure Chunk which holds the received chunk information. Process Foreman receives a messagetype ChunkMetaData that is associated with chunk information newChunk from process msgSrc of processtype SCHD in line 14 in Codelet 3.1.

In addition to the structures for sending and receiving of messages, communication structures for waiting and testing for messages are also defined in LBDSL. A process can wait for the message from other processes using Poll_Wait function, without actually retrieving the message. An example is illustrated in line 9 of Codelet 3.1. This is a blocking function and a foreman process uses this to wait for a message from any workers. Upon receipt, this function returns the information about the message source and action associated with the message in an instance of MessageInfo. Similarly a process can just check for messages using Poll_Test. Poll_Test is a nonblocking test for message. The arguments of Poll_Test contains extra boolean parameter in addition to that of Poll_Wait to notify whether the process is receiving message.

LBDSL also supports wildcard receives such as ANY_SOURCE and ANY_TAG. If the source of message is set to ANY_SOURCE, then the process can receive from any processes in the system. Similarly, if the action is set to ANY_TAG, then the process can receive any

kind of message from other processes. The use of these wildcard receives is shown in line 9 in Codelet 3.1.

3.5.5 Constants, Variables, Conditional and Loop Statements

LBDSL has a rich set of constructs supporting basic programming idioms. It provides common programming constructs such as constants, variables, control statements and mathematical operations. It also provides constructs for defining new data types. These basic structures help to bring the specification language close to C++ programming model and Promela verification language, making the translation process more obvious.

A constant is declared as follows:

```
$datatype constant<data-type>(cname, cvalue)
```

Here, `datatype constant` is a keyword for declaring a constant, whereas the values of `data-type`, `cname` and `cvalue` are controlled by the user. Any basic data types supported by LBDSL can be specified under `data-type` filed. `cname` can be any combination of letters and numbers. `cvalue` can be either an alphabetical letter or a numerical value. An example of constant value declaration is given in Codelet 3.7. In this example, constants `NProcessors`, `NWorkers` and `foreman` are declared whose value is set to `N`, `N` and `0` respectively. All these constants are defined to hold integer type values. Also two constant values `SCHD` and `CLNT` are defined to use their values as processtypes during communication.

Codelet 3.7

An example of constant value declaration in LBDSL

```
1  $datatype constant<int> (NProcessors, N);
2  $datatype constant<int> (NWorkers, N);
3  $datatype constant<int> (foreman, 0);
4  $datatype constant<int> (SCHD, 0);
5  $datatype constant<int> (CLNT,1);
```

LBDSL categorizes variables into four types based upon their usage in the protocol execution, as mentioned in section 3.4.5. To support this concept, LBDSL defines four variabletype names as follows:

- `computeV`: It represents the compute variables. Variables defined as computeV are used inside the embedded code where computation routines are defined.
- `conveyV`: It represents the convey variables. Variables defined as convey are used in the message instance during process communication.
- `decisionV`: It represents the decision variables. Variables defined as decisionV are used to define boolean conditions for conditional statements and loop statements. Also, they are used to define the terminating condition at the end state of a process definition.
- `storeV`: It represents the store variables. Variables defined as storeV are used to define arrays and vectors ina LBDSL program.

LBDSL also supports user defined data type. User defined data type can be used only as a convey type variable. A user defined data type is declared using the keyword `newType`. Its structure is similar to `Structs` in C++. The members of this structure should be one of the following datatypes: int, bool or double. The members are accessed by using instance of that datatype followed by `.membername`, similar to accessing values from structure datatype in C++. Codelet 3.4 is an example of user defined datatype. Although

this datatype is already defined within LBDSL, this example gives the idea about how to define a userdefined datatype in LBDSL.

A variable is declared as follows:

```
$datatype variabletype<data-type> (vname, vvalue)
```

Here, `datatype` is a keyword. A user needs to specify one of four variabletypes in `variabletype` field, followed by the basic data type associated with this variable type, variable name and initial value of variable.

Lines 2-5 in Codelet 3.1 is an example of variable declaration. Note that all the variables are initialized to a value except for user defined datatypes in lines 2 and 3, because they are not associated with basic datatypes. Variables `gotWork` and `numIdle` are defined as type `decisionV` to indicate they will take part in the decision process in the control structure.

Variable values can be updated whenever a new value needs to be assigned by updating a new value to an old value. Another way to update a value of a variable is by using mathematical operations in the original value. LBDSL provides a keyword `Update` to reassign a new value to the process variables. In line 13 of Codelet 3.1, value of a variable `msgSrc` is updated to the value of `msgSrc.Source`. Construct `Update` also supports addition, substraction, multiplication, division and modulo mathematical operations.

Conditional statements in LBDSL are deterministic. Operation of conditional statements is similar to If statement in C++. As in C++, a second condition executes if and only if the first guarded statement returns zero. Every conditional statement starts with a keyword `Begin_If` and ends with a keyword `End_If`. Loop statement in LBDSL is

similar to for statement in C++. An iterator value executes the statements inside the looping statements from an initial value to a final value. Loop statement in LBDSL starts with the keyword `Begin_Loop` and ends with the keyword `End_Loop`. The header of a loop statement is as follows:

`Begin_Loop(<iterator-name>:<initial-value> To <end-value>`

Variable `iterator-name` is a decisionV variable and is initialized to a value at the beginning of loop execution. Statements specified with the loop statement will continue to execute until the `iterator-value` reached to the `end-value`. LBDSL can decide whether to increase or decrease the `iterator-value` by comparing whether `initial-value` is greater than or smaller than the `end-value`.

3.5.6 Structure for embedding C++ code

As mentioned in the introduction of LBDSL, computations of load scheduling policies, allocation of buffer, execution of computation indicated by the iterates are embedded in their original C++ syntax inside LBDSL structure. LBDSL provides two ways to allow such embedding.

First, it allows to define routines related to executing load scheduling polices or policies like computations as a passive rule. Basic policy like information includes allocating buffer, computing buffer size and other computational details that do not require being in the verification model. Passive rules are treated as a black box and do not contribute to the verification of the protocol. However there are some restrictions related to defining such passive rules. A passive rule should include only those computational details that are

required for protocol implementation and whose values will not affect the correctness of the protocol. It cannot include any LBDSL structures or communication structures inside it.

Similar to the structure of process and messagetype, the structure of a passive rule is also divided into three structures: header, body and footer. Header of a passive rule begins with the keyword `Begin_Rule_P` followed by passive rule name and input arguments. Body of a passive rule consists of C++ code related to computational and policy like information details. Footer of a passive rule is defined by the keyword `End_Rule_P`

Another method to include policy like information in the LBDSL program is by embedding C++ code. Embedded C++ code should be bounded by keywords `Begin_Embed` and `End_Embed`. However, variables used inside the embedded C++ code should be declared as type `computeV` before using them.

3.6 Summary

The LBDSL is a platform for developing asynchronous load scheduling protocols for high performance computing. The main objective of LBDSL language is to facilitate the development of an asynchronous load scheduling protocols that are correct and efficient while reducing the cost of the development. LBDSL provides domain-specific notions to guide protocol construction in a correct order. It also provides common programming idioms that protocol developers are used to. The design of LBDSL is based on the concept of finite state machine and composed of process, initial variables, set of states and transition functions. Processes in LBDSL communicate with each other using messagetypes.

LBDSL provides the communication structure that facilitate the messagetype communication. LBDSL uses the concept of the dynamic partition of iterate space and each partition is called chunk. Messagetypes of LBDSL are responsible for communicating the metadata of iterate chunks. LBDSL allows embedding of protocol specification that do not play role in protocol correctness in their original format.

LBDSL specific constructs are identified by "$" symbol. Language translator for LBDSL uses the LBDSL specific constructs to support auto translation of a LBDSL program into a complete implementation code and a verification model. Verification model is generated in Promela specification language and SPIN model checker is used to perform finite state verification on the protocol logic to detect errors. In summary, LBDSL language supports the cost-efficient generation of error-free load scheduling protocols for distributed systems having distributed iterate space.

CHAPTER 4

VERIFICATION FRAMEWORK FOR LBDSL

This dissertation provides a mechanism that supports the construction of deadlock-free asynchronous load scheduling protocols using the domain specific language approach. A verification framework is designed to support the protocol development in LBDSL language discussed in last chapter. This chapter elaborates the components of this verification framework in detail.

4.1 The Verification Framework

The LBDSL verification framework takes as input the load scheduling protocol specified in the LBDSL language. It exploits the delineation mechanism provided by the LBDSL to generate a verification model automatically from a high-level specification to which protocol validation techniques may be applied to detect deadlocks. The verification framework takes advantage of an existing finite state verification tool, the Spin model checker, to provide protocol verification functionality.

The main components of the LBDSL verification framework as shown in Figure 4.1 consists of a specification language, language translation mechanism and model checking back-end.

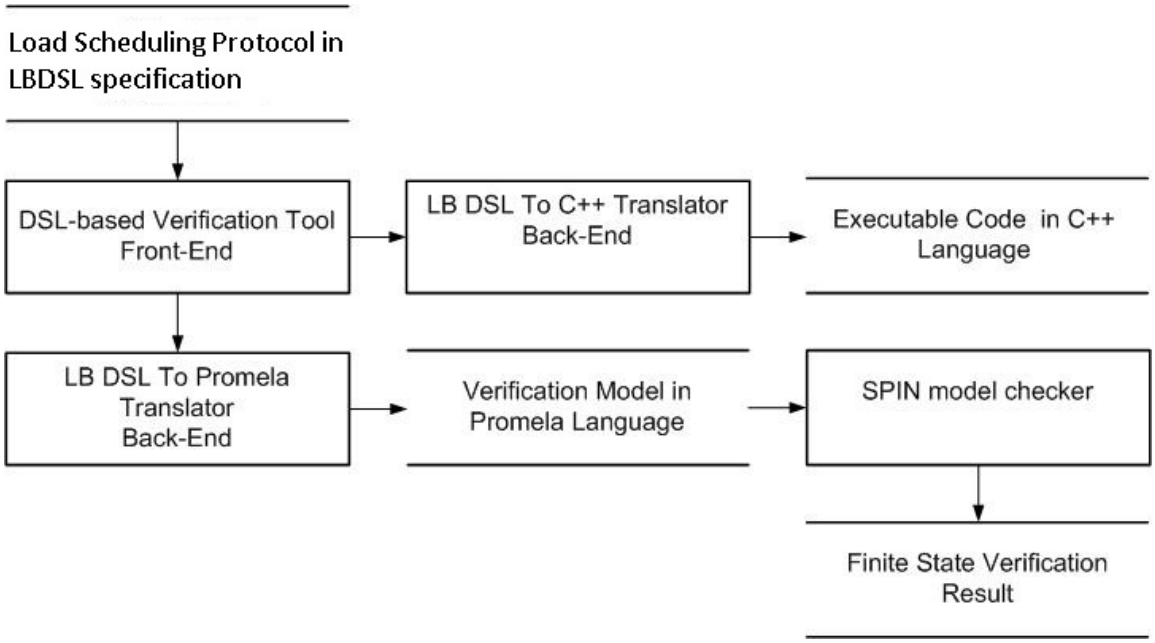

Figure 4.1

Verification framework to support LBDSL language

The verification framework takes as input the protocol specification written in LBDSL language. The single LBDSL representation of a protocol is then sufficient to automatically generate an implementation code in the C++ language and a verification model in the Promela language. Model checking is applied to the verification model to detect deadlocks and race conditions in the protocol structure. At the end of the verification process, an executable code and a verification result are obtained as two products from a single protocol specification. The extracted verification model will be a conservative representation of underlying implementation because of the embedded semantic knowledge of the system in the domain specific language. This approach can remove the cost of maintaining a separate verification model.

Instead of directly coding a verification model in Promela language, an indirect model generation approach is preferred because of the following reasons:

- A properly designed domain specific language overcomes the need for expertise in the protocol validation language and verification technique for protocol developers.
- The new approach allows the protocol developer to simultaneously specify the protocol and its verification model.
- The new approach avoids the exhaustive dual work of maintaining implementation code and verification model specifications along with the changes in protocol specifications.

The spin model checker is chosen as a verification tool for the LBDSL verification framework because it is a general tool for verifying the correctness of software models for distributed and concurrent systems. Since this research focuses on the verification of communication protocol for distributed systems, Spin is the ideal choice. Also, the closeness in structure between Promela and C++ adds one more reason to choose Spin over other verification tools. It is important to briefly discuss about the structure of Promela language before discussing the language translation mechanism, as the verification model will be in this language. Following section provides an overview of the Promela language.

4.2 The Promela Language

Promela is the specification language for the Spin model checker. A Promela program models the control structure of a protocol and checks for correctness with respect to the provided specifications in terms of linear temporal logic. The control structure of a protocol basically represents the interactions between various protocol components while computational details are left unspecified. Communication between two processes in Promela is

represented as writing to and reading from a FIFO channel without representing the implementation details on how those messages will be stored in the channel. This reduces the state space of the system to be represented by just focusing on those features that requires verification.

4.2.1 Components of Promela Language

Processes, message channels and variables are the main components of the Promela specification language. Processes are used to represent the independent objects in a system. Processes are the finite state machines that communicate with each other via message channels. Channels in Promela are represented by the complex data type `chan`. Variables are used to store the states of the process. Variables can be local or global. Promela supports the following basic data types for variables: `bit`, `bool`, `byte`, `short` and `int`. It also allows declaration of arrays and user defined data types, similar to structs in C++.

Processes are declared using the keyword `proctype`. Processes are similar to procedures in C++. Multiple instances of a process can be executed. Each instance is assigned a unique process-id by Promela run time system. Processes execution can be synchronous or asynchronous. Execution of the concurrent processes interleave with each other. A process can be executed using a keyword `run`. A simple process declaration in Promela is as follows:

```
proctype test (int arg);.
```

Message channels model the communication between concurrent processes. Message channels are declared using the keyword `chan`. An example of message channel declaration is as follows:

```
chan comm=[SIZE] of {int};
```

Here, comm defines the name of the channel. SIZE defines total number of messages that the channel can hold at any instant and `int` represents the type of variable a channel can hold. If the SIZE is set to zero, communication is synchronous, otherwise it is asynchronous. Channels are treated as FIFO queues. Sending and receiving of messages along these channels is represented by two symbols respectively: ! and ?. For example, a message is sent to the channel by executing

```
comm!  value;
```

which means, channel `comm` is sending value of type integer through it. Similarly, a message is received from the channel by executing

```
comm?  value1;
```

which means channel `comm` is receiving value1 of type integer from it.

4.2.2 Control Flow in Promela

Promela provides three mechanisms for control flow namely, case selection, repetition and unconditional jumps. Case selection is non-deterministic and is written as:

```
if ::(condition0)-> option 1 ::(condition1)-> option 2 fi
```

In the above statement, condition0 and condition1 forms a boolean guarded statement. Corresponding options gets executed whose guarded statements evaluates to true. However, the case selection process is non-deterministic. If both guarded states are true at any instance, then only one condition is chosen randomly for execution.

Repetition is used to specify the repeated execution of certain statements and is represented by the keyword `do`. An example of repeated statement is as follows:

```
do ::(condition0)->option1 ::else->break; od
```

In the above example, option1 is executed until the condition `condition0` returns true otherwise else is executed which breaks the repetition using keyword `break`.

4.2.3 Deadlock detection of a communication protocol in Promela

The Promela language provides various ways for specifying correctness conditions. Assertion is one way to denote a correctness condition that must be satisfied when the system being modeled reaches a given state. It is written as `assert(condt)` where `condt` is the boolean statement that should be satisfied. Promela assertions have the same semantics as assert statements in C.

Similarly, deadlock detection is enabled by allowing users to specify legal end states. Legal end states are labeled by the keyword `end` followed by zero or more characters for example, `end`, `end1`, `end_of_this` are all valid end state labels. A legal end state can either be a state that is reached when the protocol terminates or, in case of non-terminating protocols, a state that is reached infinitely often. Unexpected end states either denote a deadlock or an error condition that is caused by an incomplete protocol specification.

4.2.4 Limitations of Promela

Promela is very well suited for the protocol verification and provides rich set of mechanism for specifying the correctness conditions. However, the sole purpose of this language is to check logical errors in protocol specification and is not a general purpose programming language. The language does not support the programming constructs such as procedure calls. Also, the Promela language doesnot have a language compiler to generate a C++ code out of its specification. The protocol designer has to rewrite a validated Promela specification in C++. This approach not only requires the duplication of effort but is also a source of potential errors. A combination of all these factors has contributed to the restricted use of Promela as a protocol programming language. LBDSL verification framework utilizes the advantages of Promela language to design a protocol programming language that supports automatic verification.

4.3 The Language Translator

The verification framework provides a translation mechanism from LBDSL to implementation code in C++ and verification model in Promela. LBDSL language translator is built using Yacc. Yacc [46] is a parser generator developed by Stephen Johnson for Unix operating system. Yacc generates a parser based on the analytical grammar written in a notation similar to BNF. Yacc requires an external lexical analyzer to parse a file. Lexical analyzing tool Lex is commonly used along with Yacc to build a language compiler. Lex reads the input file and splits it into tokens. Yacc uses these tokens to generate the hierarchical structure of the program.

The LBDSL language translator uses the delineation rules to automate the translation process. The two products of the translation process are an implementation code in the C++ language and a verification model in the Promela language. Rules preceded by the `\$"` sign are converted into both C++ language and Promela verification language. The embedded code is not translated to Promela and is not the part of verification process. It is kept as is in the implementation code. LBDSL to C++ compiler ensures semantic consistency with the verification model. Embedded code, however, can introduce software errors. Embedded sections should be tested for syntactic and logical errors before including in the protocol specification. Figure 4.2 is an example of the compilation sequence of a program in LBDSL.

The pattern file in this diagram contains the tokens defined for constructs of LBDSL language. Lex will read the patterns and generate a C code for a lexical analyzer. The lexical analyzer then matches the statements of LBDSL program, based on the pattern file and converts each string in an LBDSL file into tokens. Tokens are the numerical representation of those strings to simplify parsing.

The grammar file in this diagram defines the correct syntax of LBDSL components. Yacc reads the grammar file and generates a C code for a syntax analyzer. The syntax analyzer will use the grammar rules to analyze tokens from the lexical analyzer and creates a syntax tree for the LBDSL program. Associated with grammar rules are processing steps to generate C++ program and Promela program that corresponds to the syntax tree.

In the given example, LBDSL's construct `Enumerate_State` is processed to generate its corresponding version in C++ and Promela format. Every rule proceeded by `\$"`

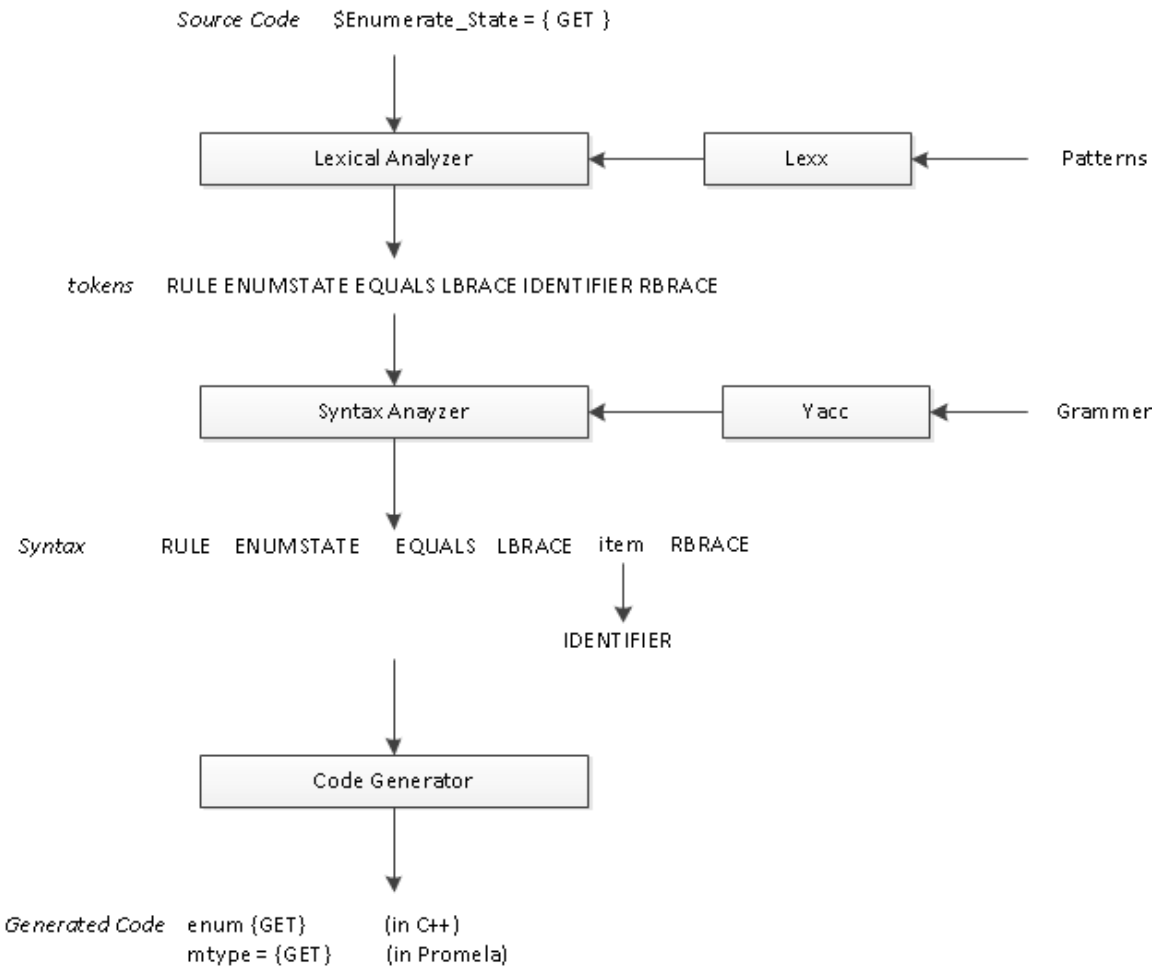

Figure 4.2

Process to build a compiler using Lex and Yacc

sign represents a special LBDSL component that represents a particular action executed during load scheduling. The meaning of these rules and how they should be represented in each language format is also defined in the grammar file. Using this information, implementation code in C++ and verification model in Promela is generated at the end.

4.3.1 Model Checking Backend

The model checking back-end of LBDSL's verification framework performs a finite state verification of the generated Promela model. Spin model checker is used as a verification tool for this purpose. Figure 4.3 provides the general idea about structure of model checking process.

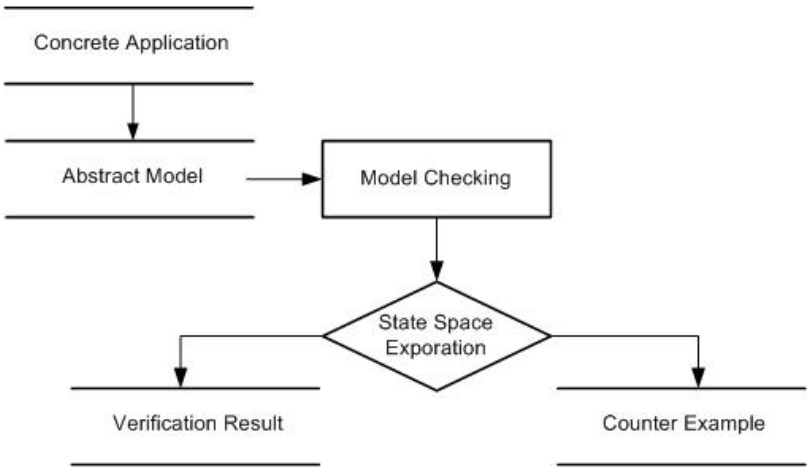

Figure 4.3

Finite State Verification

The Spin model checker converts the Promela specification into its finite state representation. Spin does a state space exploration of the resulting finite automata to check for

any violations of user-defined properties. It searches every interleaving of a resulting verification model. Spin can by default verify that the system is free of deadlocks, that the system always terminates at a valid end state, and that no dangling messages are present at the end of the execution. In case of error-detection, Spin will produce a counter example that allows the developer to trace the path of error.

4.4 Summary

This section summarizes the framework to support the development of load scheduling protocol using LBDSL language. The framework uses Lex and Yacc to build the compiler for the LBDSL program generating the corresponding C++ file and Promela file as the outputs. The generated verification model in Promela language is checked by Spin model checker for deadlocks and race conditions. Therefore, the end result of this verification framework is a C++ file and the output of the finite state verification.

The three case studies performed using the LBDSL program and its verification framework are discussed in next three chapters.

CHAPTER 5

CASE STUDY: IMPLEMENTATION OF PROBE-BASED CENTRALIZED LOAD SCHEDULING PROTOCOL

Load Balancing Domain Specific Language (LBDSL) facilitates the development of deadlock-free and race-condition-free load scheduling protocols for distributed systems. The usability of LBDSL is demonstrated by implementing three types of dynamic load scheduling protocols.

Implementation of a centralized probe-based load scheduling protocol for message passing systems is the first illustration of the applicability of LBDSL[4]. A message passing system consists of P processing nodes, each with its own exclusive address space[51]. Each processing node can either be a single process or a shared address space multiprocessor. Interaction between the processing nodes is accomplished exclusively by sending and receiving of messages. Message passing programs are often written using asynchronous paradigms. Asynchronous programs are characterized by the absence of a known bound on relative processors speed or message transfer times. Implementation of an asynchronous probe-based centralized load scheduling protocol is described in this chapter.

5.1 The protocol

Because of the absence of a centralized memory in a message passing system, the centralized load scheduling algorithm is designed to schedule a pool of iterates that is distributed to all processors in the system. One of the processors executes the role of a scheduler and manages task distribution. The scheduler maintains a chunk table in order to keep track of work chunks executed by each worker. A worker communicates with the scheduler to obtain the task information for execution.

The general architecture of a centralized load scheduling protocol is given in Figure 5.1.

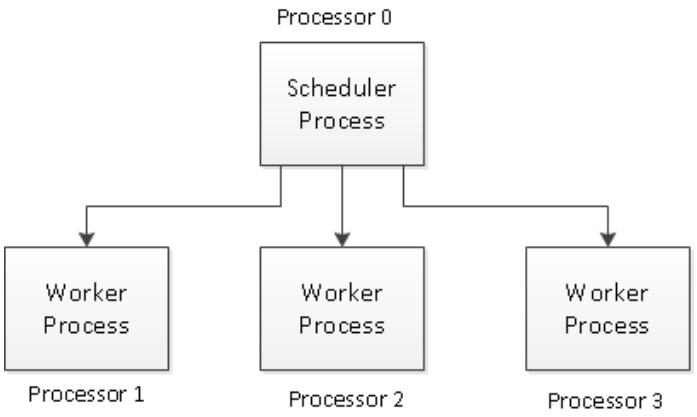

Figure 5.1

Architecture of a centralized load scheduling protocol

A load scheduling protocol starts by a scheduler computing a chunk size using user specified load scheduling policies. One of the following four types of load scheduling policies is executed by the scheduler to compute a chunk size at any instant:

- (N)o (L)oad (B)alancing (NLB): In NLB, no load scheduling policies are applied. Available iterates are divided equally among the workers. The chunk size assigned

to each worker will be N/w where, N represents the total number of iterates in a system and w represents the total workers in a system. Workers will execute what they are assigned without coordinating with other workers.

- (Fixed) (S)ized (C)hunking: $(\sqrt{2Nh})/(\sigma P\sqrt{logP})^{2/3}$ is the algorithm for computing chunk size in FSC. where h is the overhead time and represents standard deviation of independent loop execution times. This algorithm is mainly suitable for homogeneous and equally loaded processors. This algorithm is believed to achieve optimal performance if the required parameters are known.
- (Fac)toring: In FAC, tasks are scheduled in batches of P equal-sized chunks and the total number of iterations per batch is a fixed ratio of those remaining tasks.
- (G)uided (S)elf (S)cheduling: In GSS, chunk size is computed as $remaining/P$, where $remaining$ is the remaining number of independent tasks to solve. Initially this policy generates a large chunk size which decreases gradually with the decreasing number of remaining tasks.

Scheduler computes an initial chunk size using load scheduling policy specified and sends the chunk metadata to workers. A chunk metadata contains information about the starting point of iterates in a iterate space and total number of iterates from that starting point. Besides this information, sometimes it is required to communicate estimated execution time of iterates, address of the destination processor, estimated size of iterates and the action associated with iterates. All this information defines the metadata of chunk being communicated.

In the mean time, workers will be waiting for task information from the scheduler. Upon receiving the information, they start to execute the assigned tasks. A worker will request a new set of tasks once currently assigned tasks are complete. The scheduler which is constantly probing for messages, receives the worker's request, computes a new chunk size and sends the new metadata to the requesting worker. The scheduler takes into consideration current task execution time to compute the chunk size for next the iteration.

The described scenario is shown in Figure 5.2. The distributed work queue logic has the advantage of communicating only metadata with the scheduler; eliminating the need of actual iterates movement during task computation and hence saving a lot of communication overhead.

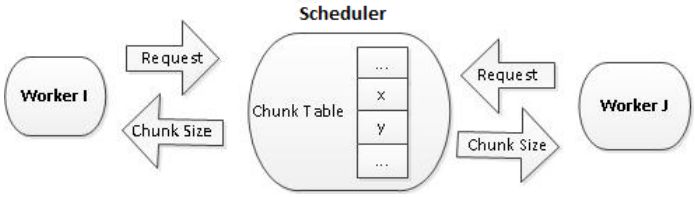

Figure 5.2

Load Scheduling

A dedicated processor for the scheduler may not be necessary since a chunk size computation involves executing a load scheduling policy which involves simple arithmetic. The scheduler, therefore, also participates in executing iterates and doubles as a worker. The scheduler uses a probing mechanism to switch between the role of a scheduler and worker. As a worker, it will also receive its share of chunk to compute. The worker-self of a scheduler does not work on the entire chunk size. At any instant, it will execute iterates at the rate of a size (Say, tSize) that is less than or equal to the chunk size. After executing each tSize, it will probe for messages from workers. If there is a message, it will switch to its role of a scheduler, responds to the request, and switch back to its role of a worker.

In order to allow task sharing, workers also do not compute the entire chunk size at once. It computes iterates at the rate of a size(Say, tSize) at any time. Computed value of

tSize should be less than or equal to chunk size. After computing each tSize, a worker will check for new messages from either scheduler or workers. Task sharing between the workers form the second part of centralized load scheduling and is called remote scheduling. Remote scheduling is depicted in Figure 5.3.

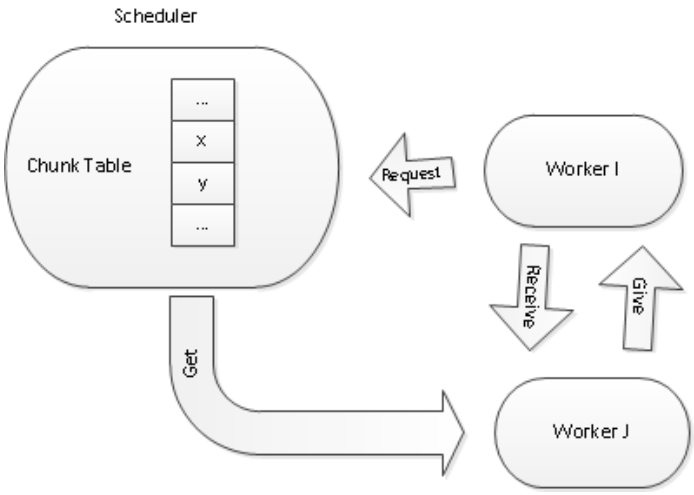

Figure 5.3

Remote Scheduling

Here, worker I has completed executing its share of iterates and requests for new task from the scheduler. Scheduler, based upon its chunk table, determines a worker (Say, worker J) for task sharing. Policy for worker selection for task sharing can be chunk execution time, by considering the worker that takes a longest execution time be the slower and eligible for task sharing. The selection factor can also be the worker having largest unexpected chunks to compute. Scheduler also determines the chunk size to be shared from worker J. Worker J which is constantly checking for messages, receives the metadata

from scheduler for task sharing. Worker J prepares the requested chunk size and sends the chunk information to worker I. Worker J continues to execute its own task. Worker I retrieves the remote chunk information, executes the task and returns the result to worker J. Worker I also sends request to the scheduler for new task. Worker J, upon receiving the shared chunk result, updates its executed task pool and resumes to its own task.

In the course of responding to the worker's request, the scheduler may find a situation when there aren't any unfinished iterates in the system. In this scenario, scheduler will send a request to terminate to the worker. The scheduler terminates after all workers in the system have terminated.

The complete probe-based centralized load scheduling protocol is shown in Figure 5.4.

5.2 The protocol in LBDSL language

A program in LBDSL requires declaring state labels, defining message types and processes. LBDSL components specifying the centralized probe-based load balancing protocol are now presented. Terms chunk and workload are used interchangeably throughout this chapter. These terms stand for the group of iterates in a task pool.

5.2.1 State Label Declaration

Processes in LBDSL program is composed of finite number of states and transition relations that trigger the movement between states. Each state are labeled. State labels are declared globally before they are used. State label declaration for centralized load scheduling protocol is shown in Codelet 5.1. Fourteen distinct state labels are declared.

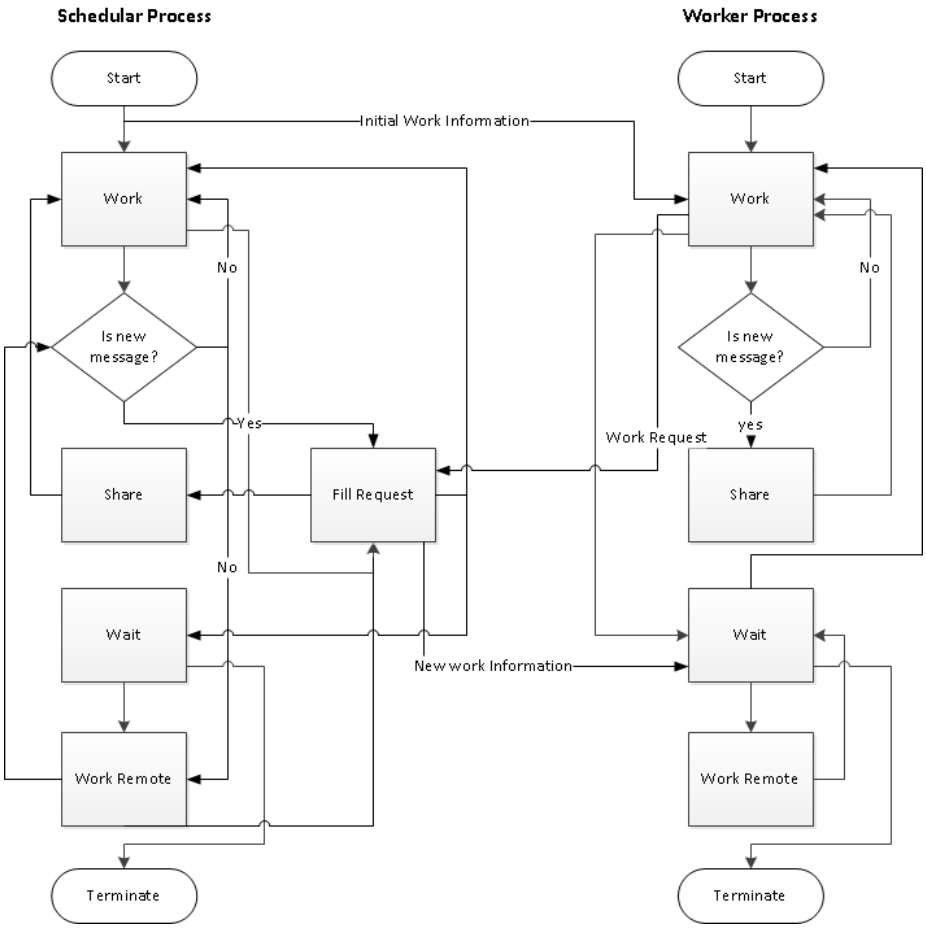

Figure 5.4

Probe-based Centralized Load Scheduling Protocol

Codelet 5.1

State Declaration

```
$Enumerate_State = {
    WAIT4_MSG, TEST4_MSG,
    WORK_LOCAL, EXEC_LOCAL_PART,
    RETRIEVE_MSG, FILL_REQUEST,
    WORK_REMOTE, EXEC_REMOTE_PART,
    WORK_COMPLETE, EXEC_REMOTE_WHOLE,
    SEND_INPUT, RECV_OUTPUT,
    FILL_LOCAL_REQUEST, TERMINATE
};
```

Note that all the processes in LBDSL can use these state labels. However, each state within a process should have a unique state label.

5.2.2 Constants Declaration

Total number of processors and process instantiation to be mapped to the processors in a system is declared. If the number of processors is equal to the number of processes, one to one mapping of processors to the process instantiation is done. If total process instantiation is more than number of processors, more than one process will be mapped to a single processor, forming a case of multithreaded system.

For the centralized probe-based load scheduling protocol, the total number of processors is set to N. Also, total number of process instantiation is set to N. It means that there will be one to one mapping between the processors in the system and the process instantiation. Total number of worker processes is declared as a constant value `NWorkers`. Note that the number of worker process instantiation is less than number of processors, meaning

Codelet 5.2

Constants for Centralized Load Scheduling Protocol

```
1  $datatype constant<int> NProcessors N
2  $datatype constant<int> NProcesses N
3  $datatype constant<int> NWorkers  N-1
4
5  $datatype constant<int> SCHD  0
6  $datatype constant<int> CLNT    1
```

not all the processors are workers. One processor will execute the scheduler process and rest of the processors will execute the worker process.

Declaration of these constant values are given in Codelet 5.2

Every process is assigned a constant value as a processtype. This protocol has two distinct processes: scheduler and worker, and hence, following two constant values are defined: `SCHD` and `CLNT`. `SCHD` is set to `0` and represents the scheduler process. Similarly, `CLNT` is set to `1` and represents the worker process. These values are used during message communication.

5.2.3 Passive Rules

Protocol specification in LBDSL allows defining some passive rules that are not required during message communication, but play an important role during workload computation and preparing the messages for communication. This LBDSL program defines four such passive rules which are now briefly discussed:

1. `GetChunkPackSize` : This rule computes the size of LBDSL's defined data type `Chunk`. The computed value is required during serialization and deserialization of chunk metadata during communication.

2. `Generate_inputPackSize`: This rule computes the size of iterates that are scheduled to be shared to another process. The computed value is required for the serialization and deserialization of iterates being shared.

3. `Generate_outputPackSize`: This rule computes the size of workload that is executed remotely. The computed value is required for the serialization and deserialization of computed iterates to return it back to the iterate owner.

4. `GetChunkSize`: This rule is used by the scheduler to compute a chunk size based on user specified load scheduling policies.

5.2.4 Messagetypes Definition

Processes in a LBDSL program communicate by sending and receiving of messagetypes. As mentioned in the previous chapter, messagetypes in LBDSL program represents the different types of messages that will be communicated during load scheduling. Messages can either be the chunk metadata or the iterates itself. A messagetype definition provides a mechanism to serialize a message before sending it to another process; and to deserialize a message after receiving it from another process.

Following three messagetypes are defined in this protocol:

5.2.4.1 Messagetype `ChunkInformation`

The definition of `ChunkInformation` is given in Codelet 5.3.

Messagetype `ChunkInformation` handles the communication of chunk metadata. This messagetype definition takes an instance of data type `Chunk` as its input. `Chunk` is a predefined data type in LBDSL and is used to define the metadata of work chunk in a LBDSL program. Members of `Chunk` data type allows defining the starting point for

Codelet 5.3

MessageType ChunkInformation

```
1  $Begin_Message ChunkInformation(conveyV<Chunk> *MetaData)
2      $datatype computeV<int> (pSize,0);
3      $datatype storeV<unsigned char> (buf,0);
4
5      $Begin_Module SendMessage
6           $GetChunkPackSize(&pSize);
7           $reSize(buf,pSize);
8           $packMetaData(*MetaData,buf,pSize);
9      $End_Module
10
11     $Begin_Module ReceiveMessage
12          $Update(pSize,*MetaData.chunkSize);
13          $reSize(buf,pSize);
14          $unPackMetaData(buf,pSize,*MetaData);
15     $End_Module
16 $End_Message
```

chunk and size of chunk in the pool of iterates, estimated execution time of chunk in the previous iteration, action associated with the messagetype and chunk destination.

Definition of this messagetype starts with declaring variables required for message serialization and deserialization. Variable `pSize` is defined as compute variable type as it will not take part in messagetype communication. Variable `buf` stores the serialized data that needs to be communicated and hence is defined as a store variable type.

This messagetype defines following two modules:

- Module `SendMessage`:
  In this module, actions necessary for serialization of metadata specified in the instance `Metadata` is described. Passive rule `GetChunkPackSize` is used to compute the value `pSize`; that is the size of `Chunk` data type. Store type variable `buf` is then re-sized to `pSize`. Then, LBDSL builtin handler `packMetaData` is used to pack the chunk metadata information. This module takes following values as input: the address of `Chunk` instance, the store variabletype `buf` to store the serialized

message and the computed pack size `pSize`. After the execution of this module, a serialized data is generated that is ready to be communicated to another process.

- Module `ReceiveMessage`:
  In this module, actions necessary for deserialization of metadata specified in the instance `Metadata` is described. First of all, expected size of the message to be received is retrieved from `chunkSize`, a member of data type `Chunk`, and is stored to compute variable type `pSize`. Store type variable `buf` is resized to `pSize`. Then, LBDSL inbuilt handler `unPackMetaData` is used to deserialize the metadata information. This module takes following values as input: the store variabletype `buf`, expected pack size `pSize` and the address of `Chunk` instance. Serialized data received is deserialized and stored in the instance of `Chunk`. After the execution of this module, received data is ready to be used by the process.

This messagetype is used by the processes when they have to communicate the metadata information about the workload.

5.2.4.2 Messagetype `ChunkShared`

The definition of `ChunkShared` is given in Codelet 5.4

This messagetype handles the communication of chunks, in addition to the chunk metadata. It takes an instance of data type `Chunk` as its input. This instance contains information required to communicate the iterates between processes.

Similar to messagetype `ChunkInformation`, definition of this messagetype starts with declaring variables required to assist message serialization and deserialization.

This messagetype defines following two modules:

- Module `SendMessage`:
  In this module, actions necessary for serialization of metadata of chunk specified in the instance `Metadata` and the iterate chunk itself is described. Passive rule `Generate_inputPackSize` is executed that computes the total size of the workload in a compute variable type `pSize`. LBDSL's inbuilt handler `packMetaData` is used to pack the chunk metadata information. LBDSL inbuilt handler `packLoad` is used to handle serialization of iterates. This handler takes following values as its argument: starting point of the iterate chunk in task pool, total number of iterates

Codelet 5.4

MessageType ChunkShared

```
$Begin_Message ChunkShared(conveyV<Chunk> *MetaData)
  $datatype computeV<int> (pSize, 0);
  $datatype storeV<unsigned char> (buf, 0);

  $Begin_Module SendMessage
    $GetChunkPackSize(&pSize);
    $Generate_inputPackSize(*MetaData.chunkStart,
      *MetaData.chunkSize,&pSize);
    $reSize(buf, pSize);
    $Update(*MetaData.chunkParam1,pSize);
    $packMetaData(*MetaData,buf,pSize);
    $packLoad(*MetaData.chunkStart,
      *MetaData.chunkSize,inputs,buf,pSize);
  $End_Module

  $Begin_Module ReceiveMessage
    $Update(pSize,*MetaData.chunkSize);
    $reSize(buf,pSize);
    $unPackLoad(0,*MetaData.chunkSize,
      inputs,buf,pSize);
  $End_Module
$End_Message
```

that define the workload, name of the task pool, name of the store variabletype that will hold the serialized data and the size of this variable type. After the execution of `packMetaData` and `packLoad`, a serialized data is generated that is ready to be communicated to another process.

- Module `ReceiveMessage`:
  In this module, actions necessary for the deserialization of chunk specified in the instance `Metadata` is described. LBDSL's inbuilt handler `unPackLoad` is used to handle the deserialization of iterate chunks received in the serialized form. This handler takes following values as its argument: starting point of the received workload (which is zero, as a process is receiving a remote data that does not belongs to its task pool), total number of iterates that define the workload, name of the task pool, name of the store variabletype that holds the serialized data and the size of this variable type. This handler deserializes the information stored in `buf` and enumerates the total number of chunkSize received in the specified task pool.

After the execution of handle `unPackLoad`, received data is ready to be used by the process. This messagetype is used by the processes to share their workload to another process.

5.2.4.3 MessageType `RemoteChunkResult`

The definition of `RemoteChunkResult` is given in Codelet 5.5

Structure of this message type is similar to message type `ChunkShared`. This messagetype is used to communicate the result of the shared workload along with the shared workload execution time between processes. It takes an instance of data type `Chunk` as its input. This instance contains information required to communicate the shared chunk result between the processes.

Similar to messagetypes `ChunkInformation` and `ChunkShared`, definition of this messagetype starts with declaring variables required to assist message serialization and deserialization. This messagetype defines following two modules:

Codelet 5.5

MessageType RemoteChunkResult

```
$Begin_Message RemoteChunkResult(conveyV<Chunk> *MetaData)
  $datatype computeV<int> (pSize,0);
  $datatype storeV<unsigned char> (buf,0);

  $Begin_Module SendMessage
    $GetChunkPackSize(&pSize);
    $Generate_outputPackSize(0,*MetaData.chunkSize,
      &pSize);
    $reSize(buf,pSize);
    $Update(*MetaData.chunkParam1,pSize);
    $packMetaData(*MetaData,buf,pSize);
    $packLoad(0,*MetaData.chunkSize,outputs,
      buf,pSize);
  $End_Module

  $Begin_Module ReceiveMessage
    $Update(pSize,*MetaData.chunkSize);
    $reSize(buf,pSize);
    $unPackLoad(*MetaData.chunkStart,*MetaData.chunkSize,
      outputs,buf,pSize);
  $End_Module
$End_Message
```

- Module `SendMessage`:
  In this module, actions necessary for serialization of metadata of chunk specified in the instance `Metadata`, and the iterate chunk itself is defined. Passive rule `Generate_outputPackSize` is executed that computes the total size of the executed workload in compute variabletype `pSize`. `packMetaData` is used to serialize the chunk metadata information. Similarly, LBDSLs inbuilt handler `packLoad` is executed to handle serialization of actual workload. After the execution of handles `packMetaData` and `packLoad`, a serialized data is generated that is ready to be communicated to another process.

- Module `ReceiveMessage`:
  In this module, actions necessary for deserialization of chunk specified in the instance `Metadata` is defined. LBDSL's inbuilt handler `unPackLoad` is executed to handle deserialization of iterates received in the serialized form. This handler will deserialize the information stored in `buf` and enumerates the total number of chunkSize received in the specified task pool.

After the execution of handle `unPackLoad`, received data is ready to be used by the process.

5.2.5 Process Declaration

Task scheduling and task execution are two distinct jobs to be performed in a centralized load scheduling protocol. Two processes, Scheduler and Worker, are defined for this purpose. Following subsections describe the implementation of these processes.

5.2.5.1 Worker Process

Worker process executes work assigned by the scheduler as well as to perform task sharing. Total number of worker process instantiation is defined by a constant value `Nworkers`. Each instantiated worker process is associated with a processorid of a processor it is mapped to.

Codelet 5.6

Definition for WAIT4_MSG state

```
1 $Start_State  WAIT4_MSG:
2    $Poll_Wait(ANY_SOURCE,CLNT,&mInfo);
3    $SaveState(WAIT4_MSG);
4    $SetState(RETRIEVE_MSG);
```

First of all, different variables are defined that are required for specifying the worker process. Instances of LBDSL inbuilt data types `Chunk` and `MessageInfo` are among these variable declarations.

Following the variable declaration, states and transition functions for worker process are defined.

Worker process begins its execution from `Start_State` labeled as `WAIT4_MSG`. Definition of this state is given in Codelet 5.6.

In this state, worker process waits for a message from other processes using module `Poll_Wait`. Arguments of this module specifies that worker can receive message from any processes that are sending to this process address with processtype `CLNT`. It will make a transition to the next state labeled as `RETRIEVE_MSG` shown in Codelet 5.7 after sensing a message.

In this state, worker gets the information about the message source from the instance of `MessageInfo`. Module `GetMessageLength` is evoked that will return the estimated message size to be received. Module `GetChunk` is executed which will enumerate the variable `newChunk`. More specifically, obtained message length and state label

Codelet 5.7

Definition for RETRIEVE_MSG state

```
1 $Next_State  RETRIEVE_MSG:
2    $Update(msgSrc, mInfo.Source);
3    $GetMessageLength(mInfo,&msgLen);
4    $GetChunk(newChunk,0, msgLen, msgSrc,
5        0.0,RETRIEVE_MSG);
6    $ReceiveMessage(ChunkInformation, myRank,
7        msgSrc,CLNT,&newChunk);
8    $Update(chunkDest, newChunk.chunkParam1);
9    $SetState(newChunk.chunkAction);
```

`RETREIVE_MSG` that is associated with the messagetype to be received are assigned to members `chunkSize` and `chunkAction` of variable `newChunk`.

Worker receives the messagetype `ChunkInformation` using `ReceiveMessage`. Module `ReceiveMessage` takes following parameters as input: the name of messagetype to be received (`ChunkInformation`), address of process receiving this messagetype (`myRank`), address of the source process (`msgSrc`), processtype associated with the receiving process (`CLNT`) and an address of the instance of data type `Chunk` which will be enumerated during this communication (`newChunk`). Member `chunkAction` of the variable `newChunk` defines the state a worker process should make transition after this communication is completed.

Note that, when a module `SendMessage` executes, LBDSL evokes `SendMessage` module defined in the messagetype definition associated with this communication. Similarly, when a module `ReceiveMessage` executes, LBDSL evokes `ReceiveMessage` module defined in messagetype associated with the communication.

Codelet 5.8

Definition for WORK_LOCAL state

```
1 $Next_State WORK_LOCAL:
2   $Update(localStart, newChunk.chunkStart);
3   $Update(localSize, newChunk.chunkSize);
4   $Begin_Compute
5     // Embedded C++ code to set enviroment
6   $End_Compute
7   $SetState(EXEC_LOCAL_PART);
```

If the next state is `WORK_LOCAL`, worker process sets the necessary environment for the iterate execution and moves to the state `EXEC_LOCAL_PART`. The definition of these states is shown in Codelets 5.8 and 5.9.

In the state `EXEC_LOCAL_PART`, worker calculates the value `tSize` that is less than or equal to the `chunkSize` using the module `GetComputeSize`. Worker will compute the assigned iterates at the rate of one `tSize` at a time. Worker will constantly check for new messages it may receive, after executing every `tSize` of iterates. Checking for new message is done in state `TEST4_MSG` using module `Poll_Test` as shown in Codelet 5.10.

Testing for messages is an asynchronous step. If a worker finds out that there is a message, it will respond to the new message. Otherwise it will continue executing the task. When the value of `tSize` becomes equal to the value `localSize` indicating the last sub chunk, worker sends a request to the scheduler for a new task. After finished executing the last sub chunk worker moves to `WAIT4_MSG` state waiting for the new task.

Codelet 5.9

Definition for EXEC_LOCAL_PART state

```
$Next_State EXEC_LOCAL_PART:
  $Begin_If(localSize == tSize)->
     $GetChunk(newChunk,0,0,myRank,0.0,FILL_REQUEST);
     $SendMessage(ChunkInformation,myRank,foreman,
       SCHD,&newChunk);
  $End_If

  // Embedded C++ code for local iterate computation

  $Update('+',localStart,tSize);
  $Update('-',localSize,tSize);

  $Begin_If(localSize>0)->
    $SaveState(EXEC_LOCAL_PART);
    $SetState(TEST4_MSG);
  $Else->
    $SetState(WAIT4_MSG);
  $End_If
```

Codelet 5.10

Definition for TEST4_MSG state

```
$Next_State TEST4_MSG:
  $Poll_Test(ANY_SOURCE,CLNT,&mInfo,&isMsg);
  $Begin_If(isMsg==1)->
    $SetState(RETRIEVE_MSG);
  $Else->
    $SetState(SavedState);
  $End_If
```

Codelet 5.11

Definition for SEND_INPUT state

```
$Next_State SEND_INPUT:
  $Update(newChunk.chunkAction,WORK_REMOTE);
  $Begin_If(chunkDest==foreman)->
      $SendMessage(ChunkShared,myRank,chunkDest,
        SCHD,&newChunk);
  $Else
      $SendMessage(ChunkShared,myRank,chunkDest,
       CLNT,&newChunk);
  $End_If
  $Update('+',returns,1);
  $SetState(SavedState);
```

If a worker receives a message for sharing its workload, worker process will move to state `SEND_INPUT` shown in Codelet 5.11.

In this state, it will prepare the chunk metadata with the information about task to be shared using LBDSL's inbuilt module `GetChunk`. Worker process then executes module `SendMessage` to send message type `ChunkShared`. Module `SendMessage` takes following parameters as input: the name of messagetype to send(`ChunkShared`), address of the process sending this messagetype(`myRank`), address of the process receiving this messagetype(`msgSrc`), processtype associated with the receiving process(`CLNT` or `SCHD`) and an address of the instance of data type `Chunk` which contains the information to be communicated through this messagetype. Recipient worker to this message type will move to the state `WORK_REMOTE`, receives the shared iterates and set the environment for the remote task execution. Execution of remote task is then performed in the state `EXEC_REMOTE_WHOLE`. Definition of these states is shown in Codelets 5.12 and 5.13.

Codelet 5.12

Definition for WORK_REMOTE state

```
1 $Next_State WORK_REMOTE:
2     $ReceiveMessage(ChunkShared, myRank, msgSrc,
3       CLNT,&newChunk);
4     $Update(remoteStart, newChunk.chunkStart);
5     $Update(remoteSize, newChunk.chunkSize);
6     $Update(saveSize, remoteSize);
7     $Update(saveSrc, msgSrc);
8     //Embedded C++ code to set enviroment
9     $SetState(EXEC_REMOTE_WHOLE);
```

Since the worker is executing the remote task, it will compute the whole remote size at once. Upon completion, worker will send the message type `RemoteChunkResult` associating the action `RECV_OUTPUT` to the owner of the chunk. Worker will also send the new task request to the scheduler.

If the worker receives the message having `WORK_COMPLETE` as `chunkAction`, scheduler is requesting it for termination. In a `WORK_COMPLETE` state, it will set the terminating condition to true which in this case is done by assigning boolean value `gotWork` to zero. Worker process finally moves to the end state (`End_State`) and terminates. End state also defines the condition to terminate which should be evaluated to true, to ensure a process has reached to the valid end state. For a worker process to be in valid end state, it should complete executing iterates it is been assigned to and should have sent the result of remote task, if it was assigned any. Definition of the `End_State` for worker process is shown in Codelete 5.14

Codelet 5.13

Definition for EXEC_REMOTE_WHOLE state

```
$Next_State  EXEC_REMOTE_WHOLE:
    //Embedded code for remote iterate computation
    $GetChunk(newChunk,0,saveSize,0,currentChunkTime,
        RECV_OUTPUT);

    $Begin_If(saveSrc==foreman)->
        $SendMessage(ChunkShared,myRank,chunkDest,
            SCHD,&newChunk);
    $Else->
        $SendMessage(ChunkShared,myRank,chunkDest,
            CLNT,&newChunk);
    $End_If

    $GetChunk(newChunk,0,0,saveSrc,0.0,FILL_REQUEST);
    $SendMessage(ChunkInformation,myRank,foreMan,
        SCHD,&newChunk);
    $Update(remoteSize, 0);
    $SetState(WAIT4_MSG);
```

Codelet 5.14

Definition for TERMINATE state

```
$End_State TERMINATE:
    $ConditionToTerminate((gotWork==0)&&(returns==0)
        &&(remoteSize==0));
```

5.2.5.2 Scheduler Process

The main objective of a scheduler process is to schedule iterates among the workers with an attempt to minimize the total completion time. Scheduler also doubles as a worker and performs task execution. By default, process-id of a scheduler process is zero. However, it can take any numerical value as process-id. Scheduler process starts by computing the initial chunksize, based upon the load scheduling policies specified, and assigning the initial chunk metadata to all workers including itself. Process then begins its execution from the start state `TEST4_MSG`. Structure of this state is similar to the worker process. A scheduler is performing a non-blocking check for messages in this state. If there is no pending messages, scheduler will move to the `WORK_LOCAL` state. In this state, it will set the necessary environment for iterate execution and moves to `EXEC_LOCAL_PART` state. Structure of state `WORK_LOCAL` is similar to worker. However the structure of state `EXEC_LOCAL_PART` is slightly different and is shown in Codelet 5.15.

While in state `EXEC_LOCAL_PART`, scheduler will compute the assigned chunks at the rate of `tSize` that is less than or equal to `chunkSize`. After computing each `tSize`, it will probe for messages from workers from `TEST4_MSG` state.

If there is any response to the probe, the response is taken into action by saving the current state. Otherwise, the scheduler continues executing the task, until it runs out of tasks. The worker-self of a scheduler will then request for new task to its scheduler self.

While in `TEST4_MSG` state, if the scheduler senses for a new message, it will move to `RETRIEVE_STATE` and retrieves the message source and action associated with the message from the instance of `MessageInfo`. Using these information, it will receive

Codelet 5.15

Definition for EXEC_LOCAL_PART state

```
1  $Next_State EXEC_LOCAL_PART:
2      $GetLocalSize(localSize,tSize,sendRequest,probeFreq);
3      //Embedded C++ code for iterate computation
4      $Update('+',localStart,tSize);
5      $Update('-',localSize,tSize);
6      $Begin_If(localSize>0)->
7          $SaveState(EXEC_LOCAL_PART);
8          $SetState(TEST4_MSG);
9      $Else->
10         $SaveState(WAIT4_MSG);
11         $Update(msgSrc,myRank);
12         $Update(chunkDest,myRank);
13         $SetState(FILL_LOCAL_REQUEST);
14     $End_If
```

message type `ChunkInformation` using module `ReceiveMessage` and moves to the state associated with the message. `Processtype` associated with this communication is `SCHD`, since it is scheduler process receiving the message.

If the retrieved message is related to sharing its workload, scheduler moves to the state `SEND_INPUT` which is similar to worker process. In this state, it will prepare the chunk metadata with the task share information and executes `SendMessage` to communicate message type `ChunkShared`. This message type is associated with action `WORK_REMOTE`.

The worker-self of a scheduler can also receive commands from its scheduler self to work in remote data. In this case, scheduler moves to `WORK_REMOTE` state, receives the remote data and sets the environment for the remote task execution, and executes remote

task in EXEC_REMOTE_PART state. Structure of state WORK_REMOTE is similar to worker process. However, structure of state EXEC_REMOTE_PART is different and is shown in Codelet 5.16.

Like executing its own task, scheduler executes the remote task by executing each tSize at any instance. After computing each tSize, it will probe for messages in a TEST4_MSG state. However, if it is done computing all the allocated remotesize, it will request for new task from its scheduler-self. It will also send the message type RemoteChunkResult to the chunk owner.

There are two states defined in the scheduler that will perform the task allocation. State FILL_LOCAL_REQUEST is responsible for assigning tasks to worker-self of the scheduler process and state FILL_REQUEST is responsible for assigning task to rest of the workers. The structure of the first state is given in Codelet 5.17 and second is given in Codelet 5.18.

To assign a new task to requesting workers, the scheduler looks for unexecuted iterates in its chunk table. If the task pool for requesting worker has unexecuted iterates, the scheduler will compute a new chunk size and send the chunk information to that worker. In case of its worker-self, it will set a new chunk size and moves to the working mode.

If the requesting worker is done executing all of its assigned iterates, scheduler will check whether or not the worker has already shared its work. If it has, scheduler will make itself not eligible for executing remote task; otherwise the worker will be eligible for executing remote task. If the worker is eligible, scheduler will compute a worker for tasks sharing, and send the worker's information and remote chunk size to the requesting

Codelet 5.16

Definition for EXEC_REMOTE_PART state

```
$Next_State EXEC_REMOTE_PART:
    $GetComputeSize(remoteSize,tSize,MIN_PROBE_FREQ,0);
    //Embedded C++ code for remote iterate computation
    $Update('+',remoteStart,tSize);
    $Update('-',remoteSize,tSize);

    $Begin_If (remoteSize>0)->
        $SetState(TEST4_MSG);
        $SaveState(EXEC_REMOTE_PART);
    $Else->
        $GetChunk(newChunk,0,saveSize,0,0.0,RECV_OUTPUT);
        $SendMessage(RemoteChunkResult,myRank,
            saveSrc,CLNT,&newChunk);
        $Update('-',incoming,1);
        $Update(msgSrc,myRank);
        $Update(chunkDest,saveSrc);
        $SaveState(WAIT4_MSG);
        $SetState(FILL_LOCAL_REQUEST);
    $End_If
```

Codelet 5.17

Definition for FILL_LOCAL_REQUEST state

```
$Next_State FILL_LOCAL_REQUEST:
  $Begin_If(yMap[2*myRank+1]>0)->
     $Update(tSource,myRank);
  $Else_If(InputSent[myRank]>1)->
     $Update(tSource,-1);
  $Else->
     //Embedded C++ code identifying heavy processor
  $End_If

  $Begin_If(tSource == -1)->
     $SetState(WAIT4_MSG);
  $Else_If(tSource==myRank)->
     $GetChunkSize(i,minChunkSize,tSource,&yMap,
    &tStart,&tSize,&batchSize,&batchRem);
     $GetChunk(newChunk,chunkStart,chunkSize,0,
        0.0,WORK_LOCAL);
    $SetState(WORK_LOCAL);
  $Else->
    $GetChunk(newChunk,tStart,tSize,myRank,0.0,
           SEND_INPUT);
    $SendMessage(ChunkInformation,myRank,tSource,
       CLNT,&newChunk);
  $End_If
```

Codelet 5.18

Definition for FILL_REQUEST state

```
$Next_State FILL_REQUEST:
  //Identify the processor for task scheduling

  $Begin_If(tSource == -1)->
     $Update('+',numIdle,1);
   $GetChunk(newChunk,0,0,0,0.0,QUIT);
   $SendMessage(ChunkInformation,myRank,msgSrc,
         CLNT,&newChunk);
   $Begin_If(numIdle==Nprocessors-1)->
      $SetState(WORK_COMPLETE);
   $Else->
      $SetState(SavedState);
     $End_If
  $Else_If(tSource==msgSrc)->
     $GetChunk(newChunk,tStart,tSize,0,
         0.0,WORK_LOCAL);
   $SendMessage(ChunkInformation,myRank,tSource,
      CLNT,&newChunk);
   $SetState(TEST4_MSG);
  $Else->
     $Begin_If(tSource != foreMan)->
        $GetChunk(newChunk,tStart,tSize,msgSrc,
          0.0,SEND_INPUT);
        $SendMessage(ChunkInformation,myRank,tSource,
       CLNT,&newChunk);
      $SetState(TEST4_MSG);
     $Else->
      $GetChunk(newChunk,chunkStart,chunkSize,chunkDest,
          0.0,SEND_INPUT);
      $SetState(SEND_INPUT);
   $End_If
  $End_If
```

Codelet 5.19

Definition for TERMINATE state

```
1    $End_State TERMINATE:
2        $ConditionToTerminate((gotWork==0)&&(incoming==0)
3            &&(remoteSize==0));
```

worker. In the case when there is no iterates left to execute and all workers are done with their task, scheduler will send terminate message to the requesting worker.

In the case of its worker-self requesting, the scheduler will moves to `WORK_COMPLETE` state and update the boolean value representing completion of its local work to true. If all the processors have terminated it will move to the end state (`End_State`) and terminate; otherwise it will move to waiting mode.

End state also defines the condition to terminate which should be evaluated to true, to ensure a process has reached the valid end state. For a scheduler process to be in a valid end state, it should complete executing iterates it is been assigned to and should have sent the result of remote task, if it was assigned any. Definition of the `End_State` for scheduler process is shown in Codelet 5.19

Scheduler process will terminate when its worker-self has completed executing the assigned task and all workers are already terminated.

5.3 Verification Result

The main purpose of specifying the protocol semantics in LBDSL is to track for deadlocks and race conditions, if there are any. The protocol specified in the last section is

the verified protocol which is free of deadlocks and race conditions. However, the initial protocol specification was not the same as the final version. In the first specification, the starting state for the scheduler was set to `WORK_LOCAL` instead of `TEST4_MSG`. The Spin verifier associated with LBDSL framework found a deadlock situation after reaching to the depth of 5980. While a process is at state `WORK_LOCAL` it cannot not receive any requests from a client process. At the same time, client process will be waiting for this execution to complete in order to receive a new workload information from the scheduler. The scheduler, on the other hand, determines this process as being heavily loaded and requests task sharing. However, this communication could not be completed because the scheduler first has to receive the message from this worker. This situation was overcome by the scheduler by first receiving the message and then only executing its own task.

Because of the non-deterministic behavior of the protocol, the initially defined protocol did not crash at every execution. It was impossible to track this deadlock situation using the traditional testing methods. Only finite state verification could trace every possible execution and determine if there is any situation that leads to a deadlock situation.

5.4 Summary

This chapter describes the implementation of centralized probe-based load scheduling protocol in LBDSL. The main drawback of the centralized load scheduling protocol is that the central scheduler can become bottleneck, since workers need to communicate with the scheduler in order to obtain the task. Similarly, at the end of each execution, all the results need to be collected at this scheduler.

Probing for messages by the scheduler is another disadvantage of this protocol. It is done to perform the role of scheduler and worker by a single processor. The scheduler need to constantly probe for the messages and had to switch its role from scheduler to worker. In the next section we redefine this algorithm using a multithreaded approach. Two individual threads are defined for the scheduler process. One thread is completely dedicated to executing the scheduling action. Another thread is responsible for executing the task of a working process. The detail discussion about this form of load scheduling problem is discussed in next section.

CHAPTER 6

CASE STUDY: IMPLEMENTATION OF MULTI-THREADED CENTRALIZED LOAD SCHEDULING PROTOCOL

Implementation of a multithreaded centralized load scheduling protocol(MCLSP) for message passing systems is the second illustration of the applicability of LBDSL. A brief introduction to multithreaded system in provided in the first section. In the second section, the protocol for a multithreaded centralized load scheduling protocol is described. In the third section, implementation of the protocol in LBDSL language is presented. Finally the verification result concludes the chapter.

6.1 Multi-Threaded Architecture

Multithreading can be defined as an ability of an operating system to allow multiple threads to execute within a context of a single processor. A thread can be defined as a light-weight process that contains a sequence of programmed instructions that can be executed independently[27].

Processors in a distributed system have local memory and communicate with each other by sending and receiving messages. This communication paradigm fits well when processors have a single thread of execution. This paradigm can suffer a performance penalty through high context switching and being prone to deadlocks especially when a single pro-

cessor has to execute two different tasks. For example, in a probe-based centralized load scheduling protocol, the scheduler is executing two different tasks. It is switching between its role of a scheduler and worker. Asynchronous communication was used to support overlapping between the communication between the processors with the computation of its worker-self. Non-determinism introduced by this asynchrony can lead to a deadlock situation.

In comparision to a single-threaded architecture, the multithreaded paradigm provides finer granularity of concurrency by making virtual processes independent of processors and allowing the overlap of communication and computation[56]. In a multithreaded architecture, multiple processes are executing within a single processor. These processes share the same address space. A thread-safe system ensures that the shared data do not get corrupted during the execution of threads.

To allow the clean separation between the scheduler and worker processes, a multithreaded centralized load scheduling protocol(MCLSP) is implemented as a second case study. The complete description of the protocol is now presented.

6.2 The protocol for Multithreaded architecture

MCLSP is a modification of a probe-based centralized load scheduling protocol implemented in last chapter. In MCLSP, scheduler and worker processes are executed into two different threads in one processor. The task of a scheduler process is now simplified as it has to only perform task scheduling and distribution. Its worker-self is now executed in a separate process. Similar to the probe-based protocol, the scheduler process in this

protocol also maintains a chunk table in order to keep track of work chunks executed by each worker.

The general architecture of a multithreaded centralized load scheduling protocol is given in Figure 6.1.

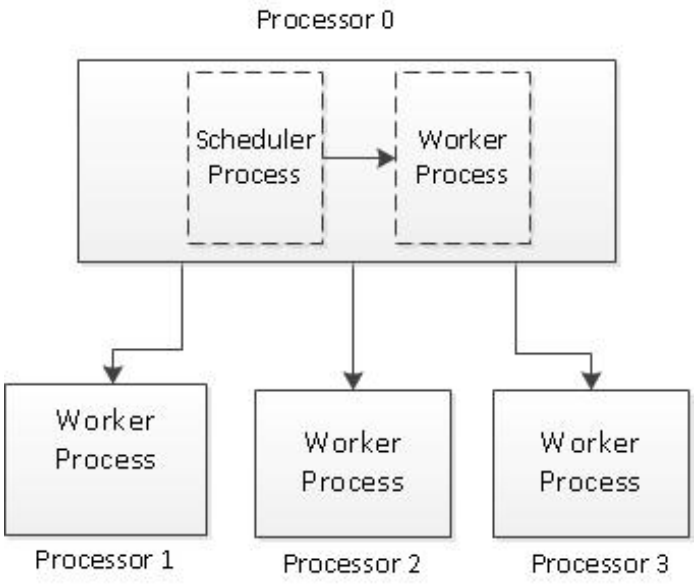

Figure 6.1

Architecture of a multithreaded-centralized load scheduling protocol

Every worker process, including that in processor 0 communicates with the scheduler via message passing to obtain the task information for execution.

The scheduler computes an initial chunk size using a specified load scheduling policy and sends the chunk metadata to workers. A chunk metadata contains information about the starting point of iterates in a iterate space and total number of iterates from that starting point. Besides these information, sometimes it is required to communicate estimated exe-

cution time of iterates, address of the destination processor, estimated size of iterates and the action associated with iterates. All this information defines the metadata of a chunk being communicated. In the mean time, workers will be waiting for task information from the scheduler. Upon receiving the information, they start to execute the assigned task. Workers will request new set of tasks from the scheduler once the currently assigned task is complete.

The scheduler upon receiving the request, responds by either sending local work information or by requesting to share the work of another worker. In order to allow task sharing, workers do not compute the entire chunk size at once. It computes iterates at the rate of a size(say, tSize) at any time. Computed value of tSize should be less than or equal to chunk size. After computing each tSize, worker will check for new messages from either scheduler or workers. The worker process will respond to the message if there are any, otherwise it will continue executing the assigned task. The protocol for the worker in this case is same as that of probe-based load scheduling protocol described in the last section.

In the course of responding to worker's request, the scheduler may find a situation when there aren't any unfinished iterates in the system. In this scenario, scheduler will send a request to terminate to the worker. The scheduler terminates once all workers in the system have terminated.

The complete protocol is given in Figure 6.2.

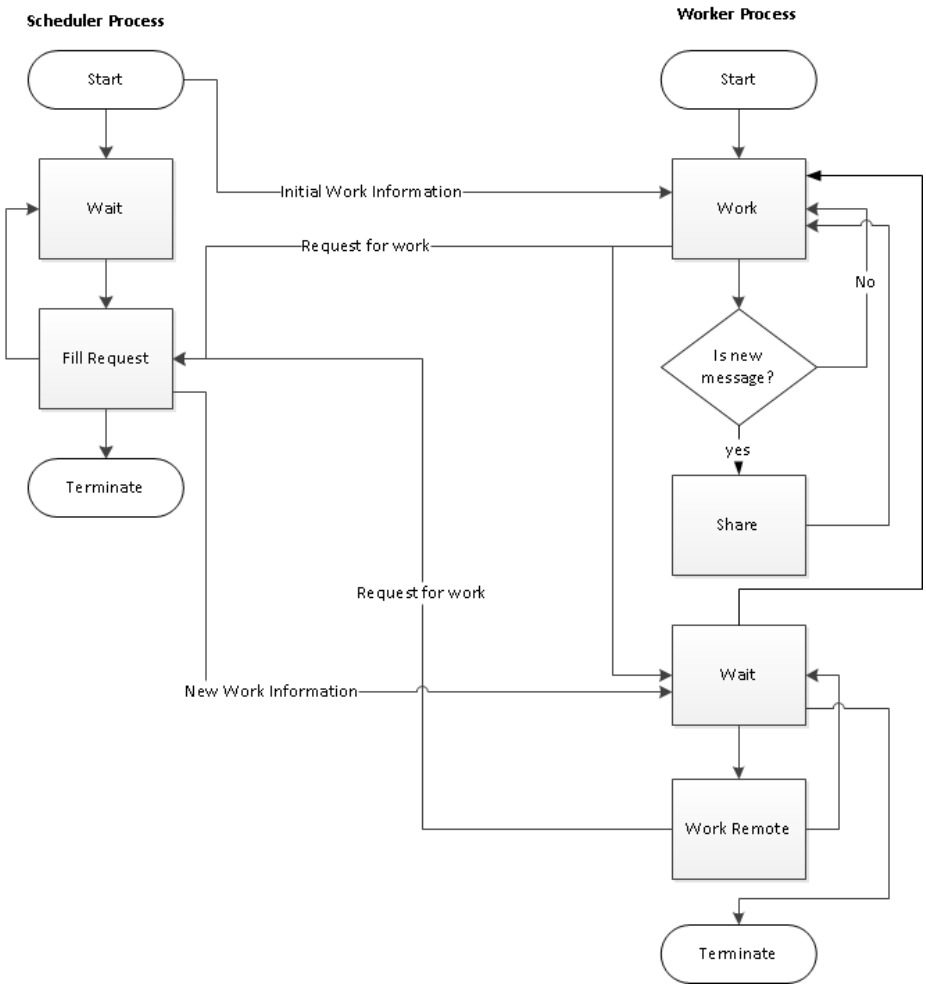

Figure 6.2

Multithreaded Load Scheduling

Codelet 6.1

State Declaration for

```
1  $Enumerate_State = {
2          WAIT4_MSG, TEST4_MSG, WORK_LOCAL,
3          EXEC_LOCAL_PART, RETRIEVE_MSG,
4          FILL_REQUEST, WORK_REMOTE,
5          WORK_COMPLETE, EXEC_REMOTE_WHOLE,
6          SEND_INPUT, RECV_OUTPUT, TERMINATE
7  };
```

6.3 The protocol in LBDSL language

A program in LBDSL requires declaring state labels, defining messagetypes and processes. LBDSL components specifying the multithreaded centralized load balancing protocol are now presented. Terms chunk and workload are used interchangeably throughout this chapter. These terms stand for the group of iterates in a task pool.

6.3.1 State Label Declaration

Processes in LBDSL program is composed of finite number of states and transition relations that trigger the movement between states. Process states are labeled. State labels are declared globally before they are used. The state label declaration for MCLSP is shown in Codelet 6.1. Twelve distinct state labels are declared. Note that all the processes in LBDSL can use these state labels. However, each state within a process should have a unique state label.

6.3.2 Constants Declaration

A LBDSL program requires declaring the number of processors in the system and the number of processes instantiated to be mapped to the processors. If the number of processors is equal to the number of processes, one to one mapping of processors to the process instantiation is done. If total process instantiation is more total number of processors, more than one process will be mapped to a single processor, forming a case of a multithreaded system.

For MCLSP, the total number of processors is set to N. Similarly, total number of processes is set to N+1. It means that more than one processes should be mapped to a single processor. Processor 0 will be mapped to both scheduler and worker process. The rest of the processors will execute a worker process.

Every process is assigned a constant value as its processtype. This protocol has two distinct processes: scheduler and worker, and hence, following two constant values are defined : `SCHD` and `CLNT`. `SCHD` is set to `0` and represents the processtype for scheduler. Similarly, `CLNT` is set to `1` and represents the processtype for worker. These values are used during message communication.

Declaration of all these constant values is given in Codelet 6.2

6.3.3 Passive Rules

Protocol specification in LBDSL requires defining some passive rules, that are not required during message communication, but play an important role during workload com-

Codelet 6.2

Constants for Centralized Load Scheduling Protocol

```
1  $datatype constant<int> NProcessors N
2  $datatype constant<int> NProcesses N+1
3  $datatype constant<int> NWorkers N
4
5  $datatype constant<int> SCHD 0
6  $datatype constant<int> CLNT 1
```

putation and preparing the message for communication. This LBDSL program reuses the same passive rules that are defined in probe-based centralized load scheduling protocol.

6.3.4 Messagetypes Definition

Three messagetypes defined in the first case study are resused for this case study. `ChunkInformation` handles the communication of chunk metadata. `ChunkShared` handles the communication of unexecuted chunks and chunk metadata. Communication of executed shared chunk and shared workload execution time is handled by the messagetype `RemoteChunkResult`.

6.3.5 Process Declaration

Task scheduling and task execution are two distinct jobs to be performed in MCLSP. Two processes, Scheduler and Worker, are defined for this purpose. Following subsections describe the implementation of these processes in LBDSL.

6.3.5.1 Worker Process

The Worker process executes the work assigned by the scheduler, as well as performs task sharing. Total number of worker process instantiation is defined by a constant value `Nworkers`. Each instantiated worker process is associated with a processor-id of a processor it is mapped to.

Structure of the worker process for MCLSP is same as the probe-based centralized load scheduling protocol.

First of all, different variables are defined that are required for specifying the worker process. Instances of LBDSL inbuilt data types `Chunk` and `MessageInfo` are among these variable declarations. Following the variable declaration, states and transition functions for worker process are defined.

A Worker process begins its execution from `Start_State` labeled as `WAIT4_MSG`. In this state, worker process waits for a message from other processes using module `Poll_Wait`. Arguments of this module specifies that worker can receive message from any processes that are sending to this process address with processtype `CLNT`. It makes a transition to a new state labeled as `RETRIEVE_MSG` after sensing a message.

In this state, the worker gets the information about the message source from an instance of `MessageInfo`. Module `GetMessageLength` is used to get the estimated message size to be received. Module `GetChunk` is used to enumerate the variable `newChunk`. More specifically, obtained message length and state label `RETREIVE_MSG` that is associated with the messagetype to be received are assigned to members `chunkSize` and `chunkAction` of variable `newChunk`.

Worker receives the messagetype ChunkInformation using communication module ReceiveMessage. Module ReceiveMessage takes following parameters as input: the name of messagetype to be received (ChunkInformation), address of process receiving this messagetype (myRank), address of the source process (msgSrc), processtype associated with the receiving process (CLNT) and an instance of datatype Chunk which will be enumerated during this communication. After this communication is completed, member chunkAction of variable newChunk will contain the state label of the next state that this process should make transition to.

Note that, when a communication module ReceiveMessage executed within a state, LBDSL evokes the ReceiveMessage module defined in the messagetype associated with the communication. Similarly, when a communication module SendMessage is executed within a state, LBDSL evokes the SendMessage module defined in the messagetype definition associated with this communication.

If the worker makes a transtion to state labeled as WORK_LOCAL, it sets the necessary environment for the iterate execution and moves to the state labeled as EXEC_LOCAL_PART.

In the state EXEC_LOCAL_PART, the worker calculates the value tSize that is less than or equal to the chunkSize using the module GetComputeSize. The Worker will compute the assigned iterates at the rate of one tSize at a time. The Worker will constantly check for new messages it may receive, after executing every tSize of iterates. Checking for a new message is done in state TEST4_MSG using module Poll_Test .

This is an asynchronous step. If a worker finds out that there is a message, it will respond to the new message. Otherwise it will continue executing the task. When the

value of tSize becomes equal to the value localSize indicating the last sub chunk, the worker sends a request to the scheduler for a new task. After it finishes executing the last sub chunk the worker moves to WAIT4_MSG state waiting for the new task.

If a worker receives a message that asks for sharing its workload, the worker process will move to state SEND_INPUT. In this state, it will prepare the chunk metadata with the information about task to be shared using LBDSL's inbuilt module GetChunk. The worker process then executes module SendMessage to send messagetype ChunkShared. Module SendMessage takes the following parameters as input: the name of messagetype to send (ChunkShared), address of the process sending this messagetype (myRank), address of the process receiving this messagetype (msgSrc), processtype associated with the receiving process (CLNT) and an address of the instance of data type Chunk which contains the information to be communicated through this messagetype. The Recipient worker to this message type will move to the state WORK_REMOTE, receives the shared iterates and set the environment for the remote task execution. Execution of the remote task is then performed in the state EXEC_REMOTE_WHOLE.

Since the worker is executing the remote task, it will compute the whole remotesize at once. Upon completion, worker will send the message type RemoteChunkResult associating the action RECV_OUTPUT to the owner of the chunk. The Worker will also send the new task request to the scheduler.

If the worker receives the message having WORK_COMPLETE as chunkAction, it means that scheduler is requesting it to terminate. In a WORK_COMPLETE state, the worker will set the terminating condition to true which in this case is done by assigning boolean

value gotWork to zero. Worker process finally moves to the end state (End_State) and terminates. End state also defines the condition to terminate which should be evaluated to true, to ensure a process has reached to the valid end state. For a worker process to be in valid end state, it should complete executing iterates it is been assigned to and should have sent the result of remote task, if it was assigned any.

6.3.5.2 Scheduler Process

The main objective of the scheduler process is to schedule iterates among the workers with an attempt to minimize the total completion time. Processid of the scheduler process is set to the rank of a processor it is mapped to. The Scheduler process starts by computing the initial chunksize, based upon the load scheduling policies specified, and assigning the initial chunk metadata to all workers. The Scheduler process then begins its execution from the start state (Start_State) labeled as WAIT4_MSG. The structure of this state is similar to the worker process except for SCHD is set as the processtype.

It will make a transition to the next state labeled as RETRIEVE_MSG after sensing a message. In this state, the scheduler retrieves the information about the message source from an instance of MessageInfo. Module GetMessageLength is evoked that will return the estimated message size to be received. Module GetChunk is executed which will enumerate the variable newChunk. More specifically, the obtained message length and state label RETREIVE_MSG that is associated with the messagetype to be received are assigned to members chunkSize and chunkAction of variable newChunk.

The scheduler's only state label it can receive is `FILL_REQUEST`. In this state, scheduler assigns a new task to the requesting worker. The structure of the state `FILL_REQUEST` is given in Codelet 6.3.

To assign a new task to the requesting worker, the scheduler looks for unexecuted iterates in its chunk table. If the task pool for the requesting worker has unexecuted iterates, the scheduler will compute a new chunksize and send the chunk information to that worker.

If the requesting worker is done executing all of its assigned iterates, the scheduler will check whether or not the worker has already shared its work. If it has, the scheduler will make the worker eligible for executing a remote task. Otherwise, the scheduler will compute a worker for task sharing, and send the worker's information and remote chunk size to the requesting worker. In a situation, when there are no unexecuted iterates and all of the workers are done with their tasks, the scheduler will send the terminate message to the requesting worker.

If all the processors have terminated it will move to the end state (`End_State`) and terminate. Otherwise it will move to waiting mode.

The End state also defines the condition to terminate which should be evaluated to true, to ensure a process has reached to the valid end state. For a scheduler process to be in the valid end state, it should have sent terminate messages to all the workers in the system and the terminating conditions should be true. Definition of the `End_State` for scheduler process is shown in Codelet 6.4

The Scheduler process will terminate after all workers are terminated.

Codelet 6.3

Definition for FILL_REQUEST state

```
1     $Next_State FILL_REQUEST:
2     $Begin_If(yMap[2*msgSrc+1]>0)->
3       $Update(tSource, msgSrc);
4     $Else_If(InputSent[msgSrc]>1)->
5       $Update(tSource, -1);
6     $Else->
7       //Embedded C++ code identifying heavy processor
8     $End_If
9
10  $Begin_If(tSource == -1)->
11          $Update('+',numIdle,1);
12     $GetChunk(newChunk,0,0,0,0.0,QUIT);
13     $SendMessage(ChunkInformation,myRank,msgSrc,
14        CLNT,&newChunk);
15
16     $Begin_If(numIdle==Nprocessors-1)->
17             $Update(gotWork,0);
18        $SetState(TERMINATE);
19     $Else->
20        $SetState(WAIT4_MSG);
21     $End_If
22
23  $Else_If(tSource==msgSrc)->
24          $GetChunk(newChunk, tStart, tSize, 0,
25                  0.0, WORK_LOCAL);
26     $SendMessage(ChunkInformation, myRank, tSource,
27                  CLNT, &newChunk);
28     $SetState(WAIT4_MSG);
29
30  $Else->
31          $GetChunk(newChunk, tStart, tSize, msgSrc,
32                 0.0, SEND_INPUT);
33          $SendMessage(ChunkInformation, myRank, tSource,
34                CLNT, &newChunk);
35     $SetState(WAIT4_MSG);
36        $End_If
37    $End_If
```

Codelet 6.4

Definition for TERMINATE state

```
1    $End_State TERMINATE:
2        $ConditionToTerminate(gotWork==0);
```

6.4 Verification Result

Multithreaded centralized load scheduling protocol is similar to the probe-based load scheduling protocol except for the scheduler process whose only task is to schedule the iterates. Since a deadlock-free probe-based load scheduling protocol specification was already generated, verification process did not find any deadlocks or race-conditions in this protocol. The new protocol is then integrated to the Loci library[55]. The integrated library is used to execute an application that requires load scheduling. The resulting efficiency of application is increased by 4% when executed in 4 and 8 number of processors in comparison to probe-based load scheduling protocol.

6.5 Summary

This chapter describes the implementation of a multi-threaded probe-based load scheduling protocol in LBDSL. The protocol consist of a centralized scheduler to perform task scheduling. All the processors in the system execute the worker process. A processor that is mapped to both scheduler and worker processes executes two thread's one for each of them. Processes within a processor also communicates via message passing to ensure a thread safe system is built.

The main drawback of this protocol is that the central scheduler can become a bottleneck, since workers need to communicate with the scheduler in order to obtain the tasks. Multithreaded hierarchical load scheduling protocol is the third case study that overcomes the bottleneck of centralized load scheduling protocol, decentralizing the load scheduling process.

CHAPTER 7

CASE STUDY: IMPLEMENTATION OF MULTI-THREADED HIERARCHICAL LOAD SCHEDULING PROTOCOL

Implementation of a multithreaded hierarchical load scheduling protocol for message passing systems is the third illustration of the applicability of LBDSL. A brief introduction of hierarchical load scheduling protocols and their benefit over centralized load scheduling protocol are discussed in first section. The second section describes the multithreaded hierarchical load scheduling protocol considered for the case study. Implementation of the protocol in LBDSL language is presented in the third section. This chapter concludes by discussing the verification result of the protocol implemented.

7.1 Hierarchical Load Scheduling

Centralized load scheduling is a simplest form of load scheduling protocol where load balancing decisions are made on a specific processor as a scheduler. The decisions are based upon the load execution time of a processor in its previous iteration. Since the global information about processors work load is readily available on the central scheduler, load scheduling decisions will be a best choice. This protocol has proved to work well for a few thousands of processors. However, this algorithm faces scalability issues, especially in machines with a small amount of memory. Also, the scheduler can become a bottleneck

for achieving higher performance as all the processors have to communicate with the single processor to get load information. The solution to this problem can be a fully distributed load scheduling protocol, where each processor in the system exchanges the load information, decentralizing the load scheduling task. However, the processors will not have the global load information, and this tends to lead to poor load balancing, especially in big systems.

A hierarchical load scheduling protocol inherits the good qualities of both centralized and distributed load scheduling protocol. The best part of centralized load scheduling is the concept of having global information, while deciding for task scheduling. The best part of distributed load scheduling is to decentralize the role of the scheduler such that every processor of the system does not have to communicate with one scheduler, increasing the speed of the scheduling process.

The basic idea behind a hierarchical load scheduling is to divide the processors into independent autonomous groups and to organize the groups in a hierarchy[81]. Each autonomous group consists of one scheduler that manages load scheduling across all processors in its sub tree. This approach decentralizes the role of a scheduler. Local schedulers of each sub domain form the next level of hierarchy and communicates with each other to balance the load across all the groups. One of the local schedulers is assigned as a root that maintains the global information by communicating with the local schedulers. Root scheduler is responsible for balancing load across all the subdomains. Hierarchical load scheduling protocol can reduce the time and memory required for load balancing because

the size of each sub domain is smaller than the entire number of processors. Workers in each sub domain need to communicate with its scheduler to get the load information.

A hierarchical load scheduling algorithm is more advanced than the centralized load scheduling algorithm. A general idea about how the hierarchical load scheduling algorithm operates is discussed above in this section. The next section describes the hierarchical load scheduling implemented as the third case study.

7.2 The protocol

A hierarchical load scheduling algorithm maintains a hierarchy of schedulers so that the load scheduling task will be decentralized across a group of local schedulers. There can be any number of hierarchy levels, with the processors on the bottom of the hierarchy performing task execution and processors in different levels of hierarchy performing task scheduling for its sub domain.

Hierarchical load scheduling protocol implemented for this case study has two levels of hierarchy. Initially, total processors in the system are divided into different sub domains, each of size less than or equal to L. Processor with lowest ranking in each sub domain is assigned as the scheduler for that sub domain. The schedulers of all sub domains form the second level of hierarchy. Processor with rank 0 executes the role of a global scheduler. Note that a processor with rank 0 has to perform three tasks: global scheduling, local scheduling and task execution. Similarly, every processor that performs local scheduling is also executing a task. This protocol also uses a multithreading approach in order to allow a processor to execute multiple roles. For example, processor 0 will execute three

processes global scheduling, local scheduling and task execution. Each process is executed in a separate thread. A multithreading approach will simplify the protocol, minimizing the need for asynchronous communication and reducing frequency to deadlock.

The global scheduler maintains a chunk table in order to keep track of work assigned to each sub-domain. Similarly, local scheduler keeps track of work executed by each worker in its sub domain. The general architecture of multithreaded-hierarchical load scheduling protocol is shown in Figure 7.1. The Terms chunk and workload are used interchangeably throughout this chapter. These terms stand for the group of iterates in a task pool.

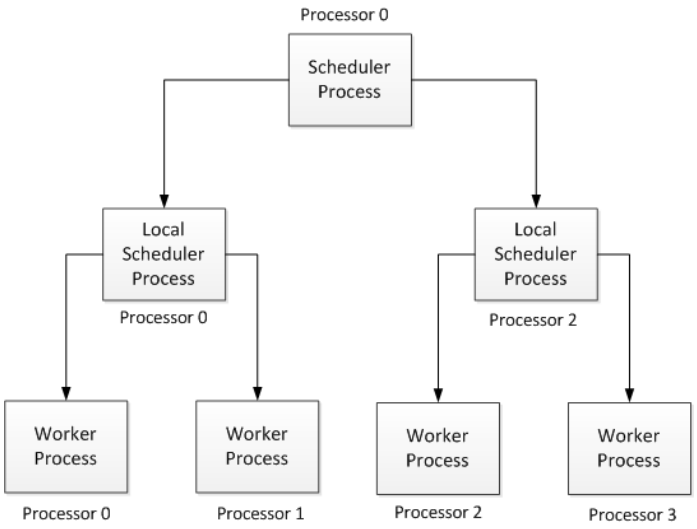

Figure 7.1

Architecture of a multithreaded-hierarchical load scheduling protocol

The hierarchical load scheduling protocol starts by global scheduler assigning the meta-data of workbatch to each local scheduler. Here, workbatch represents a group of iterates that belongs to the workers of a particular subdomain. To facilitate load scheduling, global

scheduler does not assign the complete task to local schedulers, such that a subdomain performing slowly can share its unexecuted workbatch to another subdomain.

The global scheduler is only sending the metadata of workbatch. It is because the load scheduling protocol assumes the workload is distributed among the workers. Local schedulers upon receiving the metadata, each computes a chunksize using user specified load scheduling policies for each worker in its domain. While computing chunksize, local scheduler takes into consideration the remaining unexecuted workload from the assigned batch. Local scheduler sends the chunk metadata to all the workers that belong to its sub domain. A chunk metadata contains information about the starting point of iterates in a iterate space and total number of iterates from that starting point. Besides this information, sometimes it is required to communicate estimated execution time of iterates, address of the destination processor, estimated size of iterates and the action associated with iterates. All this information defines the metadata of chunk being communicated.

In the mean time, workers will be waiting for task information from their local scheduler. Upon receiving the information, they start to execute the assigned task. Protocol for the worker process is same as described in the probe-based centralized load scheduling protocol and multithreaded-centralized load scheduling protocol. However, workers are allowed to communicate only to other workers in its sub domain and its local scheduler in this case.

In the course of responding to the worker's request, the local scheduler may find a situation when there aren't any unfinished iterates in the assigned batch for its sub domain. In this scenario, it will send a request for a new batch of work to the global scheduler.

The global scheduler assigns a new batch of work to the local scheduler if there are any. Otherwise, the global scheduler will send the message to terminate to the local scheduler.

The local scheduler will send either work metadata or the request to terminate to its workers based upon the information it has received from the global scheduler. The local scheduler terminates after all workers in its sub domain have terminated. The global scheduler terminates after all the local schedulers have terminated.

The complete hierarchical load scheduling protocol is shown in Figure 7.2.

7.3 The protocol in LBDSL language

A program in LBDSL requires declaring state labels, defining message types and processes. LBDSL components specifying the multithreaded hierarchical load scheduling protocol(MHLSP) are now presented.

7.3.1 State Label Declaration

Processes in LBDSL program is composed of finite number of states and transition relations that trigger the movement between states. Each state is labeled. State labels are declared globally before they are used. State label declaration for MHLSP is shown in Codelet 7.1. Sixteen distinct state labels are declared. Note that all the processes in LBDSL can use these state labels. However, each state within a process should have a unique state label.

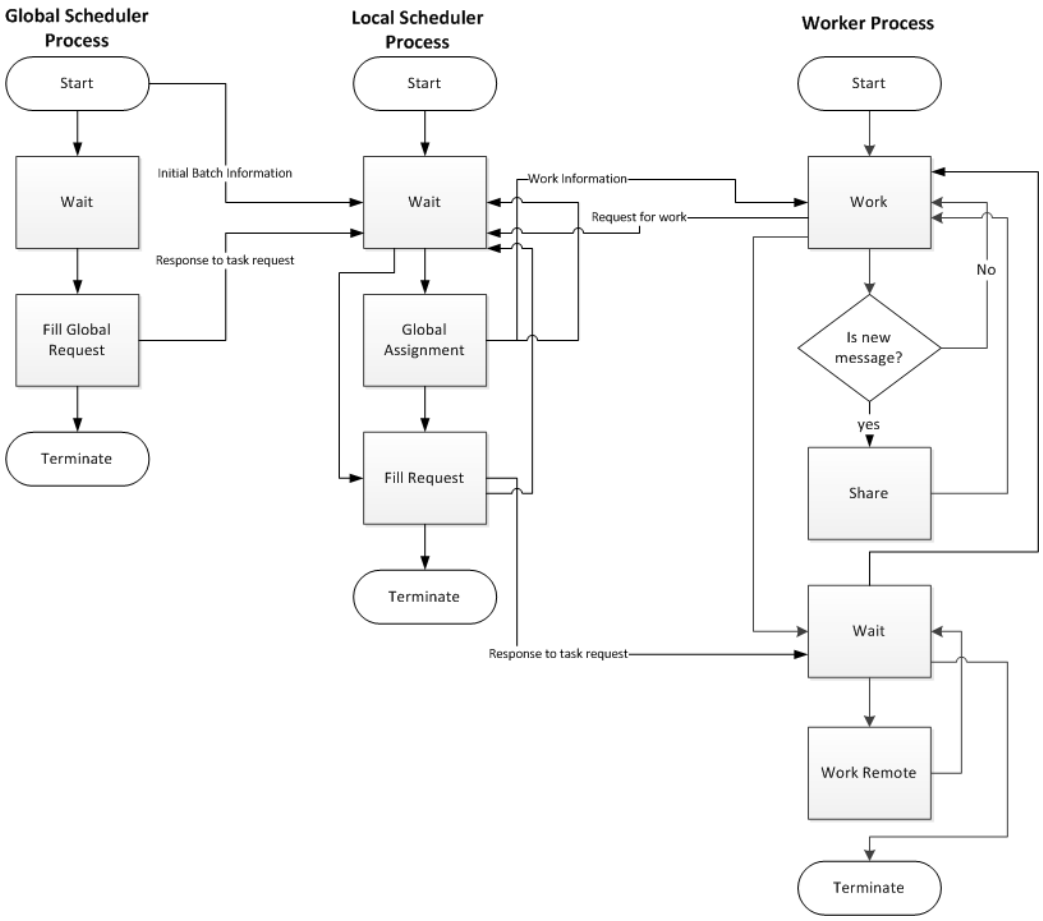

Figure 7.2

Hierarchical load scheduling protocol

Codelet 7.1

State Declaration

```
$Enumerate_State ={
        WAIT4_MSG, TEST4_MSG, RETRIEVE_MSG,
        WORK_LOCAL, WORK_REMOTE, SEND_INPUT,
        RECV_OUTPUT, FILL_REQUEST, TERMINATE,
        EXEC_LOCAL_PART, EXEC_REMOTE_WHOLE,
        WORK_COMPLETE, GLOBAL_ASSIGNMENT,
        GLOBAL_SHARE, FILL_GLOBAL_REQUEST,
        WORK_COMPLETE_SCHEDULER
    };
```

7.3.2 Constants Declaration

A LBDSL program requires declaring the number of processors in the system and the number of processes instantiated to be mapped to the processors. If the number of processors is equal to the number of processes, one to one mapping of processors to the process instantiation is done. If total process instantiation is more than number of processors, more than one process will be mapped to a single processor, forming a case of a multithreaded system.

For the hierarchical multithreaded load scheduling protocol, the total number of processors is set to N. Similarly, total number of processes is set to M. Number of processes should always be greater than or equal to number of processors. Here, it is not equal to N meaning there are more processes than processors in a system, making the protocol eligible for a multithreaded structure. Also, number of workers `NWorkers` is equal to number of processors in the system, meaning, all processors will execute this process. Similarly, the maximum size of a sub domain is defined by the constant value `LeafSize`.

Every process in LBDSL is associated to a constant value as its processtype. The following three processtypes are defined: `GSCHD`, `LSCHD` and `CLNT`, each representing processes, globalscheduler, localscheduler and worker respectively. `GSCHD` is set to `2`, `LSCHD` is set to `0` and `CLNT` is set to `1`. These values are used during message communication between processes.

Declaration of these constant values are given in Codelet 7.2

Codelet 7.2

Constants for multithreaded hierarchical load scheduling protocol

```
1  $datatype constant<int> NProcessors N
2  $datatype constant<int> NProcesses M
3  $datatype constant<int> NWorkers N
4  $datatype constant<int> LeafSize L
5
6  $datatype constant<int> GSCHD  2
7  $datatype constant<int> LSCHD  0
8  $datatype constant<int> CLNT   1
```

7.3.3 Passive Rules

Protocol specification in LBDSL requires defining some passive rules, that are not required during message communication, but play an important role during workload computation and preparing the message for communication. This LBDSL program reuses the same passive rules that are defined in probe-based centralized load scheduling protocol.

7.3.4 Messagetypes Definition

Three messagetypes that are defined in the first case study are resused for this case study. Messagetype `ChunkInformation` is defined to handle the communication of chunk metadata. Messagetype `ChunkShared` is defined to handle the communication of unexecuted chunks and chunk metadata. Messagetype `RemoteChunkResult` is defined to handle the communication of executed shared chunk and shared workload execution time.

7.3.5 Process Declaration

Global scheduling of tasks, task scheduling within a subdomain and task execution are three distinct jobs to be performed in a hierarchical load scheduling protocol. Three processes, GlobalScheduler, LocalScheduler and Worker, are defined for this purpose. Following subsections describe the implementation of these processes.

7.3.5.1 Worker Process

A Worker process executes the work assigned by the local scheduler, as well as performs task sharing between workers within the same sub domain. The total number of Worker process instantiations is defined by a constant value `Nworkers`. Each instantiated Worker process is associated with a processor-id of a processor it is mapped to.

A worker process starts by determining the address of its local scheduler using an builtin module `GetLocalSchedularAddress`. This module uses the process address and the maximum allowed size of sub domain to compute the address of the local scheduler. Since, processors are assigned to the role of local scheduler at run time, processors as workers also need to determine their local scheduler at run time. After this step, States and transition functions for Worker process are defined.

The load scheduling protocol of the Worker process for MHLSP is same as the centralized load scheduling protocol as described in chapters 5 and 6.

7.3.5.2 LocalScheduler Process

The main objective of a LocalScheduler process is to schedule a batch of work load it receives from the global scheduler across the workers in its sub domain, with an attempt to minimize the total completion time. Process-id of a local scheduler process is set to the rank of a processor it is mapped to.

Process LocalScheduler starts by computing the size of its subdomain using builtin module `GetSubDomainSize`. This module takes the process address and maximum allowed size of sub domain to determine its sub domain size. LocalScheduler needs this information to manage the task pool that belongs to its sub domain.

LocalScheduler waits for the task information for its workers from the GlobalScheduler in the start state (`Start_State` labeled as `WAIT4_MSG`. The structure of this state is same as that of a worker except for it has `LSCHD` as a processtype in handler `Poll_Wait`. LocalScheduler will move to the state `RETRIEVE_MSG` when it senses a message. The structure of this state is also similar to the Worker process except for it uses processtype `LSCHD`. In this state, LocalScheduler may get the information about the ratio of the total size of the task pool it needs to work on if the message is from GlobalScheduler or it may also receive the request for a task from workers.

If the message is from GlobalScheduler, LocalScheduler moves to the state labeled as `GLOBAL_ASSIGNMENT` where it computes a batch size for all Workers in its subdomain. It then computes an initial chunksize for each worker and sends the initial chunk information to all the workers in its subdomain. The structure of this state is given in Codelet 7.3.

Codelet 7.3

Definition for GLOBAL_ASSIGNMENT state

```
1
2  $Next_State Global_Assignment:
3    //Embedded C++ Code to compute batchsize
4
5    $Begin_Loop(tSource:myRank To sizeSubDomain)
6      $Update(inputSent[tSource],0);
7      $GetChunkSize(i, minChunkSize, tSource, &yMap,
8          &chunkStart, &chunkSize, &batchSize, &batchRem);
9      $GetChunk(newChunk,chunkStart,chunkSize,tSource,
10           0.0,WORK_LOCAL);
11     $SendMessage(ChunkInformation,myRank,tSource,
12         CLNT,&newChunk);
13   $End_Loop
14
15   $SetState(WAIT4_MSG);
```

If the message is from the workers requesting work, the LocalScheduler moves to the state labeled as `FILL_REQUEST`. State `FILL_REQUEST` is responsible for assigning tasks to workers. The structure of this state is given in Codelet 7.4.

To assign a new task to the requesting worker, local scheduler looks for unexecuted iterates in its chunk table. If the task pool for requesting worker has unexecuted iterates, local scheduler will compute a new chunk size and send the chunk information to that worker.

If the requesting worker is done executing all of its assigned iterates, the local scheduler will check whether or not the worker has already shared its work. If it has, the local scheduler will make the requesting worker not eligible for executing remote task. Otherwise, the local scheduler will compute a worker for tasks sharing, and send the worker's information

Codelet 7.4

Definition for FILL_REQUEST state

```
$Next_State FILL_REQUEST:
  $Begin_If(msgSrc!=newChunk.chunkDest)->
    $Update(i,newChunk.chunkDest);
    $Update('-',inputSent[i-myRank],1);
  $End_If

  $Begin_If(yMap[msgSrc] > 0)->
     $Update(tSource,msgSrc);
  $Else_If(InputSent[msgSrc]> 1)->
     $Update(tSource,-1);
  $Else->
   // Embedded code identifying processor for task sharing
 $End_If

  $Begin_If(tSource == -1)->
    $Update('+',numIdle,1);

    $Begin_If(numIdle==sizeSubDomain)
      $GetChunk(newChunk,0,sizeSubDomain,msgSrc,
            0,FILL_GLOBAL_REQUEST);
      $SendMessage(ChunkInformation,myRank,foreMan,
            GSCHD,newChunk);
      $Update(numIdle,0);
    $End_If
      $SetState(WAIT4_MSG);
  $Else_If(tSource==msgSrc)->
    $GetChunk(newChunk,chunkStart,chunkSize,0,
            0,WORK_LOCAL);
    $SendMessage(ChunkInformation,myRank,tSource,
            CLNT,newChunk);
    $SetState(WAIT4_MSG);
  $Else->
    $GetChunk(newChunk,chunkStart,chunkSize,msgSrc,
             0.0,SEND_INPUT);
    $SendMessage(ChunkInformation,myRank,tSource,
             CLNT,newChunk);
    $SetState(WAIT4_MSG);
  $End_If
```

Codelet 7.5

Definition for WORK_COMPLETE_SCHEDULER state

```
$Next_State WORK_COMPLETE_SCHEDULAR:
  $Begin_Loop(i:myRank To sizeSubDomain)
    $Update(mappedSource, myRank+i);
    $GetChunk(newChunk,0,0,msgSrc,0.0,WORK_COMPLETE);
    $SendMessage(ChunkInformation,myRank,mappedSource,
        CLNT,&newChunk);
  $End_Loop
  $Update(gotWork,0);
  $SetState(TERMINATE);
```

and remote chunk size to the requesting worker. In case when there is no iterates left to execute and all workers are done with their task, LocalScheduler will request for new work batch to the global scheduler.

It may receive a new batch of tasks, in which case it again goes to state labeled as `GLOBAL_ASSIGNMENT` and distributes the task to its workers. It may also receive the request to terminate from the global scheduler. In this case, local scheduler will move to state labeled as `WORK_COMPLETE_SCHEDULER` where it sends the terminate message to its workers, sets the condition to terminate as true and moves to the end state. Structure for states `WORK_COMPLETE_SCHEDULER` and `TERMINATE` is given in Codelets 7.5 and 7.6.

The End state also defines the condition to terminate which should be evaluated to true, to ensure a process has reached to the valid end state. For a scheduler process to be in a valid end state, it should have sent terminate messages to all the workers in the system.

Codelet 7.6

Definition for TERMINATE state

```
1  $End_State TERMINATE:
2    $ConditionToTerminate(gotWork==0);
```

7.3.5.3 GlobalScheduler Process

The main objective of a GlobalScheduler process is to schedule workload across all the sub domains in the system, with an attempt to minimize the total completion time and balance the workload in the system. The Process-id of a GlobalScheduler process is set to the rank of a processor it is mapped to. GlobalScheduler starts by computing the initial batchsize to assign to each subdomain, based upon the user specified batchsize ratio. It then assigns the metadata about the current batch size to all local schedulers in the system using messagetype `ChunkInformation`. Batchsize ratio is assigned to the member `chunkparam2` of datatype `Chunk`. It is assigned to this specific member because the ratio can be a floating point value and, as mentioned before in the chapter about LBDSL structure, the data type of member `chunkParam2` is also floating point.

After sending the initial batch information to all the local schedulers, GlobalScheduler waits for messages from the local schedulers in state `WAIT4_MSG`. The structure of this state is similar to that of LocalScheduler and Worker except for the handler `POLL_WAIT` uses the process type `GSCHD`. When it senses a message, it moves to state `RETRIEVE_MSG`, where it receives the address of a local scheduler requesting for the information about unscheduled task. Global scheduler will then move to the state `FILL_GLOBAL_REQUEST`.

In this state, global scheduler will send the new batch size information if there are any unscheduled batches of work remained in the task pool. Otherwise it will send the message to terminate to the local scheduler. It will terminate when it is done sending the terminate message to all the local schedulers in the system by moving to the end state `TERMINATE`. Structure of both of these states is given in Codelets 7.7 and 7.8.

The terminating condition for this process is when the value of a variable `gotWork` is set to false meaning there are no unscheduled work in the system.

7.3.5.4 Initialize Process

Initialize process computes the number of workers for each sub-domain and identifies the local scheduler of those sub-domains. This process is executed by all the processors in the system before the execution of hierarchical load scheduling protocol.

7.4 Verfication Result

The specification of hierarchical load scheduling protocol in LBDSL language is executed with the verification framework for deadlocks. The protocol specified in the last section is a verified protocol which is free of deadlocks and race conditions. However, the initial protocol specification was not the same as the final version. In the first specification, within a sub-domain, when a worker has already given its task and hasn't received the executed task, it was still eligible for task sharing. But the specification provided flexibility that leads to a race condition situation where a process tries to receive the request to task sharing and result of already shared task. To avoid this situation a worker is restricted from being eligible for sending its next share of task, if it is currently sharing a task and hasnt

Codelet 7.7

Definition for FILL_GLOBAL_REQUEST state

```
$Next_State FILL_GLOBAL_REQUEST:
  $Update(sizeSubDomain,newChunk.chunkParam1);
  $GetGroupMap(msgSrc,sizeSubDomain,&groupMap);

  $Begin_If(numBatch[groupMap]< totalBatch)->
    $GetChunk(newChunk,0,0,0,1,GLOBAL_ASSIGNMENT);
    $SendMessage(ChunkInformation,myRank,msgSrc,
        LSCHD,&newChunk);
    $Update(numBatch[groupMap],numBatch[groupMap]+1);
  $Else->
    $GetChunk(newChunk,0,0,0,0,WORK_COMPLETE_SCHEDULER);
    $SendMessage(ChunkInformation,myRank,msgSrc,
        LSHD,&newChunk);
    $Update(numIdle, numIdle + 1);

    $Begin_If(numIdle==numGroups)->
      $Update(gotWork,0);
      $SetState(TERMINATE);
    $Else->
      $SetState(WAIT4_MSG);
    $End_If
  $End_If
```

Codelet 7.8

Definition for TERMINATE state

```
$End_State TERMINATE:
  $ConditionToTerminate(gotWork==0);
```

yet received the result. These details are implemented in state `FILL_REQUEST` within process LocalScheduler.

A Race condition was also detected in the protocol specified. The problem is prominent in processor 0 which is executing all three processes. The problem occurs when a GlobalScheduler and Worker are sending message to the LocalScheduler. Both of these message races with each other to be received by the LocalScheduler and the correct message is not received in some executions. To avoid this situation, task sharing is disabled for worker 0.

Because of the non-deterministic behavior of the protocol, the initially defined protocol did not crash at every execution. It was impossible to track this race situation using the traditional testing methods. Only finite state verification could trace every possible execution and determine if there is any situation that leads to a race situation.

7.5 Summary

The hierarchical load scheduling protocol implemented here still does not provide the load sharing functionality among the sub-domains. The Load sharing feature among the sub-domains provides a functionality where a complete workload is scheduled in different batches by the global scheduler among the local schedulers. If one sub-domain is done executing its own work, GlobalScheduler requests it to share the workload of another sub-domain. However, task sharing has been enabled among the workers within a sub-domain. For implementing task sharing among sub-domains, a new message type is required that handles the communication of a batch of work that belongs to one sub-domain, to another

sub-domain. The implementation of task sharing among the sub-domains has been left as a future work.

This case study implements a hierarchical load scheduling protocol. The verification result showed a problem related to the original implementation. Based on the output, the structure of LBDSL program was improved. The protocol described here is the final deadlock free and race condition free hierarchical load scheduling protocol.

CHAPTER 8

THREATS TO VALIDITY

It is very important to consider the threats to validity in order to judge the quality of research based on case studies. Threats to validity can be defined as conditions, other than that has been considered during research, that could be responsible for the outcomes concluded [28]. Research conclusions that are based on case studies are prone to a multitude of possible threats. A discussion about the validity threats can help to analyze the research fairly and mitigate these threats in future research. This chapter provides a discussion of the validity threats on this research work.

Case studies conducted in this dissertation help to validate the hypothesis that a domain specific language based verification approach is a cost efficient way to generate deadlock-free load scheduling protocols for message passing systems. Threats to validity for the conclusions derived from these case studies are discussed in this chapter.

8.1 Internal Validity

Internal validity refers to the characteristic of a study design. It is concerned with whether the correct conclusion was drawn from the research conducted. In this research, the internal validity is related to whether or not the conclusion that the application of domain specific language (DSL) based approach for protocol specification generates a dead-

lock free load scheduling protocol. Our research does not pose this threat. It is because the DSL based verification framework uses the protocol specification written in LBDSL and automatically generates a verification model in the Promela language which is further verified by the Spin model checker. Had there been further manual intervention during verification model generation, there would have definitely been a threat to internal validity. However, the conclusions are drawn solely from the outcomes obtained from the verification framework and therefore this threat to validity does not exist.

Misuse of code embed features of LBDSL is also a threat to internal validity. If the crucial features of the communication semantics that can cause deadlocks are inserted inside LBDSL program using code embed features, then the delineation mechanism of the verification framework does not include them in the verification model. These semantics, therefore, do not take part in verification process and the final verification result will be a false positive result. LBDSL program, however, does not allow one to embed the communication semantics inside the code embed feature. Messages to be communicated must be specified as messagetypes and must use communication structures to communicate these message types, hence avoiding the possible threat to internal validity.

8.2 External Validity

One problem inherent in the research based on case studies is external validity[26]. It means, the results obtained from conducting particular type of case studies may or may not apply to another development project. External validity is concerned with whether we can generalize the results outside the scope of our study. This dissertation does posses this

threat. Although this dissertation is investigating the applicability of a DSL to devise a cost-efficient verification methodology for parallel load scheduling protocols, only three types of case studies are conducted. These case studies demonstrate the implementation of load scheduling protocols for message passing systems where the task pool has been distributed among the processors. However, there are multitude of ways to design load scheduling protocols. For example, the third case study is one way to design a hierarchical load scheduling protocol but this is not the only way. Therefore, it requires many case studies to determine the solid structure of LBDSL that is able to model any kind of load scheduling protocol for distributed systems. The objective of this work is to provide evidence that will help to determine whether or not the hypothesis that a domain specific approach allows a cost efficient way to generate deadlock free load scheduling protocols over traditional systems. Although, this dissertation poses the external validity threat, it certainly establishes the evidence for the hypothesis stated.

8.3 Construct Validity

Construct validity focuses on the relation between the theory behind the experiment and the observed outcomes. In other words, it addresses whether we are testing what we intended to test. This research is not prone to this threat. This research proposed that the verification of asynchronous load scheduling protocols can be made more automatic, robust and cost effective, if the semantics of such protocols are properly encoded into a domain specific language. We claimed that the proposed approach would reduce time and cost of producing an error-free asynchronous load scheduling protocols within a supported

domain and an implementation of a load scheduling protocol, within a narrow domain, that are directly verified would be obtained.

In this research, a specification language having constructs that reflects the domain of asynchronous load scheduling protocols for message passing systems is developed. Three different types of load scheduling protocols are represented in this specification language. With LBDSL supported verification framework, the protocol specification is automatically verified to be free of deadlocks using finite state verification approach. The verification is only on the communication structure. Therefore this implementation cannot determine other types of compute bugs or memory errors. An implementation code in C++ is also generated as one of the output of this verification framework. Therefore, the experimental set up correlates with the hypothesis proposed and hence construct validity is not a threat.

8.4 Summary

This chapter discusses the possible threats to validity to the conclusions based on the case studies. While the conclusions derived are not prone to internal validity and construct validity errors, it is definitely prone to external validity. This validity threat indicates that there is not one way to design a protocol and the experiment conducted in a certain controlled environment may not be applicable to other cases. However, this research definitely concludes that a domain specific approach can provide a cost efficient way for designing deadlock free load scheduling protocols for distributed systems.

CHAPTER 9

CONCLUSION AND FUTURE WORK

Deadlock-free parallel load scheduling protocols (PLSP) have a very important role in high performance computing. However PLSP cannot be completely verified for deadlocks using existing parallel debugging techniques because of the non-determinism introduced due to asynchrony between process communications. Verification result may be false positive and the parallel applications may still contain subtle software faults.

Finite state verification can guarantee the complete verification of a parallel load scheduling protocols, in comparison to the debugging techniques that evaluate only some of the total execution paths, especially in asynchronous protocols. However the problem with finite state verification is that it is difficult to guarantee that verification model is a proper conservative representation of an actual system, and therefore verification results cannot be completely relied upon. Implementation languages were not designed for verification, and therefore the extracted models are either prohibitively expensive to verify, or the models need significant culling in the extraction process limiting the confidence in the extracted representation. Model based formal verification is useful if the semantics of an implementation code and a verification model is represented under a single framework such that the

verification model closely represents the implementation and the automation of a verification process is natural.

An assembly of protocol implementation and automatic extraction of verification model is possible using a domain specific language. The primary motivation of this dissertation is to reduce the cost of writing an error-free parallel load scheduling protocols for distributed systems with the help of language abstraction. The proposed approach does not claim to be the best approach for generating verified communication protocols; however, it will certainly improve the current state of art.

A specification language, LBDSL, is introduced that facilitates the development of deadlock free asynchronous load scheduling protocols. The components of LBDSL language closely represent the targeted domain. The verification framework for LBDSL uses the model checking techniques to verify the asynchronous behavior of the protocol. It allows the same protocol specification to be used for verification and the code generation. The communication structure in the protocol, denoted by the delineation symbol, is extracted from LBDSL specification and fed to the SPIN model checker for verification against safety properties. Marking the communication semantics between processes as verification relevant directs the auto translation process. LBDSL, therefore, overcomes the need of expertise of the protocol validation language and verification techniques for protocol developers, which would otherwise add to the verification costs. Using the LBDSL verification framework, the single protocol representation is sufficient to generate an implementation code and a verification result. It also avoids the exhaustive dual work of maintaining implementation code and verification model specifications along with the changes

in protocol specifications. The main objective of saving manpower and time for testing and debugging subtle deadlock and race bugs is fulfilled using the LBDSL.

The benefit of using LBDSL for developing PLSP is three fold. First, the newly developed programming interface clearly shows the important components required to build a clean and sophisticated PLSP. The building blocks of this language are Processes, Chunk structure, Messagetypes and Communication structure which are also the essential components of a PLSP. Processors in a distributed message passing system communicate by explicit sending and receiving of messages. Different types of messages will be exchanged in this process. Messages may only include the metadata about load or may contain the actual load information. To execute load scheduling, some processors execute the task of load scheduling and some processors execute the task of load execution. Using LBDSL, different processes can be created for each task. Similarly, different message types can be created based upon types of information exchanged during load scheduling. Similarly, Chunk structure allows defining the metadata of load that need to be exchanged during load scheduling. Communication structures can be used to initiate the sending and receiving of messages. In summary, the structures ease the PLSP development process.

PLSP has an important feature of Code Embedding. A PLSP is composed of load scheduling policies and a communication protocol. Deadlock is mainly caused due to the nature of the communication protocol. The size of work load to be computed and the particular process that is chosen during communication is irrelevant in terms of deadlock detection. In other words, a PLSP should be deadlock free, irrespective of the particular choice of these values. In PLSP, components that play role during communication and the

communication structure are distinguished from the components that execute load scheduling policies or execute load computation. The latter can be embedded in their original C++ code using Code Embed feature. This delineation mechanism is used during verification model generation by the language translator, such that the resulting model will have relatively small size and only focuses on the communication structure which can introduce deadlocks. Components that are embedded in LBDSL program do not appear in the verification model and do not play role in verification process. This feature not only reduces the size of resulting verification model but also simplifies the verification process. This is the second benefit of using the LBDSL framework.

Lastly, Spin model checker is used as the verification back end for LBDSL framework which verifies the resulting verification model written in Promela. To perform verification, Spin generates the finite state representation of the verification model and performs the depth first search of all possible execution paths from the start state. It checks for possible deadlocks and race conditions by exploring all possible inter leavings of an asynchronous system and does a complete verification. Support to automatic verification during protocol development is the third benefit of using LBDSL framework.

Implementation of three different PLSPs namely, Probe based centralized, multithreaded, and hierarchical demonstrates the applicability of LBDSL in a practical field. These experiments also help to identify the essential components of LBDSL language. However one must note that the LBDSL language is not a fully functional language. It means, the results obtained from conducting particular type of case studies may or may not apply to another development project. Although this dissertation is investigating the applicability of

a DSL to devise a cost-efficient verification methodology for parallel load scheduling protocols, only three types of case studies are conducted. These case studies demonstrate the implementation of load scheduling protocols for message passing systems where the task pool has been distributed among the processors. However, there are many ways to design load scheduling protocols. It requires many case studies to determine the solid structure of LBDSL that is able to model any kind of load scheduling protocol for distributed systems. Objective of this work is to provide evidence that will help to determine whether or not the hypothesis that a domain specific approach allows a cost efficient way to generate deadlock free load scheduling protocols over traditional systems. Implemented case study definitely made a case showing that the LBDSL based verification approach removes the need for debugging for deadlocks and race bugs which has potential to significantly lower software development costs.

There are various things that need to be done to make LBDSL framework more perfect. First of all, various state space reduction methods can be included during verification model generation process such that efficient verification model is generated that can verify large systems without worrying about the state space problem. Secondly, the current version of the LBDSL verification framework only verifies against the deadlocks and race conditions. If the framework is able to support the verification of other correctness features of PLSP specified in Linear Temporal Logic (LTL) specifications, LBDSL framework will be more generic in terms of supporting the protocol verification. Besides deadlocks and race conditions, correctness features of various PLSP vary. Protocols may need to verify some specific state is always executed or a particular property is always satisfied. The

support to verify any type of properties specified in temporal logics is also left as a future work of this dissertation.

REFERENCES

[1] "Static randomized incremental algorithms," *Towards Dynamic Randomized Algorithms in Computational Geometry*, vol. 758 of *Lecture Notes in Computer Science*, Springer Berlin Heidelberg, 1993, pp. 22–34.

[2] "Message Passing Interface Forum. MPI :A Message-Passing Interface standard," 1995, http://www.mpi-forum.org/docs/.

[3] "Message Passing Interface Forum. MPI-2: Extensions to the Message-Passing Interface.," 1997, http://www.mpi-forum.org/docs/.

[4] P. Adhikari and E. Luke, "Verificaition of Parallel Asynchronous Load Scheduling Protocols using Domain Specific Language Approach," *Proceedings of 21th International Conference on Software and Data Engineering*, Los Angeles, CA, 2012.

[5] P. Adhikari, E. A. Luke, and E. B. Allen, "Verification of a Loop Scheduling Protocol using Finite State Verification," *Proceedings of the 22nd International Conference on Parallel and Distributed Computing and Communication Systems*, 2009.

[6] G. Agha and P. Thati, "An Algebraic Theory of Actors and Its Application to a Simple Object-Based Language," *From Object-Orientation to Formal Methods*, vol. 2635 of *Lecture Notes in Computer Science*, Springer Berlin, Heidelberg, 2004, pp. 26–57.

[7] E. E. Ajaltouni, A. Boukerche, and Z. Ming, "An Efficient Dynamic Load Balancing Scheme for Distributed Simulations on a Grid Infrastructure," *12th IEEE/ACM International Symposium on Distributed Simulation and Real-Time Applications*, october 2008, pp. 61 –68.

[8] G. R. Andrews, D. P. Dobkin, and P. J. Downey, "Distributed allocation with pools of servers," *Proceedings of the first ACM SIGACT-SIGOPS symposium on Principles of distributed computing*, New York, NY, USA, 1982, PODC '82, pp. 73–83, ACM.

[9] G. S. Avrunin, S. F. Siegel, and A. R. Siegel, "Finite-state verification for high performance computing," *Proceedings of the second international workshop on Software engineering for high performance computing system applications*, New York, NY, USA, 2005, pp. 68–72, ACM.

[10] K. Barker, A. N. Chernikov, N. Chrisochoides, and K. Pingali, "A Load Balancing Framework for Adaptive and Asynchronous Applications," *IEEE Transaction on Parallel and Distributed Systems*, vol. 15, no. 2, Feb. 2004, pp. 183–192.

[11] A. Basu, *A language-based approach to protocol construction*, doctoral dissertation, Cornell University, Ithaca, NY, USA, 1998.

[12] D. Beyer, T. A. Henzinger, R. Jhala, and R. Majumdar, "The software model checker Blast: Applications to software engineering," *International Journal of Software Tools for Technology Transfer*, vol. 9, no. 5, 2007, pp. 505–525.

[13] M. Bhandarkar, L. V. Kal, E. de Sturler, and J. Hoeflinger, "Adaptive Load Balancing for MPI Programs," *Proceedings of the International Conference on Computational Science-Part II*, London, UK, 2001, pp. 108–117, Springer-Verlag.

[14] P. Bjesse, "What is formal verification?," *SIGDA Newsletter*, vol. 35, no. 24, 2005, p. 1.

[15] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, "Cilk: an efficient multithreaded runtime system," *SIGPLAN Notice*, vol. 30, no. 8, 1995, pp. 207–216.

[16] R. E. Bryant, "Binary decision diagrams and beyond: enabling technologies for formal verification," *Proceedings of the 1995 IEEE/ACM international conference on Computer-aided design*, Washington, DC, USA, 1995, pp. 236–243, IEEE Computer Society.

[17] J. Brzezinski, J.-M. Helary, and M. Raynal, "Deadlocks in distributed systems: request models and definitions," *Proceedings of the Fifth IEEE Computer Society Workshop on Future Trends of Distributed Computing Systems*, 28-30 1995, pp. 186 –193.

[18] R. L. Carino and I. Banicescu, "Dynamic load balancing with adaptive factoring methods in scientific applications," *The Journal of Supercomputing*, vol. 44, no. 1, 2008, pp. 41–63.

[19] S. Chandra, B. Richards, and J. R. Larus, "Teapot: A Domain-Specific Language for Writing Cache Coherence Protocols," *IEEE Transaction on Software Engineering*, vol. 25, no. 3, 1999, pp. 317–333.

[20] R. Chaube, I. Banicescu, and R. Cario, "Parallel implementations of three scientific applications using LB_migrate," *IEEE International Parallel and Distributed Processing Symposium*, 2008, pp. 1–8.

[21] J. Chen, H. Zhou, and S. D. Bruda, "Combining Model Checking and Testing for Software Analysis," *Proceedings of the 2008 International Conference on Computer Science and Software Engineering*, Washington, DC, USA, 2008, pp. 206–209, IEEE Computer Society.

[22] E. M. Clarke, E. A. Emerson, and J. Sifakis, "Model checking: algorithmic verification and debugging," *Communications of the ACM*, vol. 52, no. 11, 2009, pp. 74–84.

[23] E. M. Clarke, O. Grumberg, and D. A. Peled, *Model Checking*, MIT Press, 1999.

[24] J. C. Corbet, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Păsăreanu, Robby, and H. Zheng, "Bandera: extracting finite-state models from Java source code," *ICSE '00: Proceedings of the 22nd international conference on Software engineering*, New York, NY, USA, 2000, pp. 439–448, ACM.

[25] D. L. Dill, A. J. Drexler, A. J. Hu, and C. H. Yang, "Protocol Verification as a Hardware Design Aid," *Proceedings of the 1991 IEEE International Conference on Computer Design on VLSI in Computer & Processors*, Washington, DC, USA, 1992, pp. 522–525, IEEE Computer Society.

[26] T. T. Dinh-Trong and J. M. Bieman, "The FreeBSD Project: A Replication Case Study of Open Source Development," *IEEE Transaction on Software Engineering*, vol. 31, no. 6, June 2005, pp. 481–494.

[27] J. Emer, M. D. Hill, Y. N. Patt, J. J. Yi, D. Chiou, and R. Sendag, "Single-threaded vs. multithreaded: Where should we focus?," *IEEE Micro*, vol. 27, no. 6, 2007, pp. 14–24.

[28] R. Feldt and A. Magazinius, "Validity Threats in Empirical Software Engineering Research - An Initial Survey," *Proceedings of the Software Engineering and Knowledge Engineering Conference*, Redwood City, San Fransisco Bay, CA, USA, 2010, pp. 374–379.

[29] M. Feng and C. E. Leiserson, "Efficient detection of determinacy races in Cilk programs," *Proceedings of the ninth annual ACM symposium on Parallel algorithms and architectures*, New York, NY, USA, 1997, pp. 1–11, ACM.

[30] A. E. Goodloe and C. A. Muoz, "Compositional Verification of a Communication Protocol for a Remotely Operated Vehicle," *Formal Methods for Industrial Critical Systems*, vol. 5825, 2009, pp. 86–101.

[31] W. Haque, "Concurrent deadlock detection in parallel programs," *International Journal of Computers and Applications*, vol. 28, no. 1, 2006, pp. 19–25.

[32] K. Havelund and T. Pressburger, "Model checking JAVA programs using JAVA PathFinder," *International Journal on Software Tools for Technology Transfer*, vol. 2, no. 4, 2000, pp. 366– 381.

[33] R. C. Holt, "Some Deadlock Properties of Computer Systems," *ACM Computing Surveys*, vol. 4, no. 3, Sept. 1972, pp. 179–196.

[34] G. Holzmann, , G. J. Holzmann, and M. H. Smith, "Automating Software Feature Verification," *Bell Labs Technical Journal*, vol. 5, 2000, pp. 72–87.

[35] G. J. Holzmann, "Logic Verification of ANSI-C Code with SPIN," *Proceedings of the 7th International SPIN Workshop on SPIN Model Checking and Software Verification*, London, UK, 2000, pp. 131–147, Springer-Verlag.

[36] G. J. Holzmann, *The SPIN Model Checker: Primer and Reference Manual*, Addison-Wesley, 2003.

[37] G. J. Holzmann and M. H. Smith, "A practical method for verifying event-driven software," *ICSE '99: Proceedings of the 21st international conference on Software engineering*, New York, NY, USA, 1999, pp. 597–607, ACM.

[38] Y. F. Hu and R. J. Blake, ," *Progress in computer research*, F. Columbus, ed., Nova Science Publishers, Inc., Commack, NY, USA, 2001, chapter Load balancing for unstructured mesh applications, pp. 117–148.

[39] P. Hudak, "Modular domain specific languages and tools," *Proceedings of the 5th International Conference on Software Reuse*, june 1998, pp. 134 –142.

[40] S. F. Hummel, E. Schonberg, and L. E. Flynn, "Factoring: a method for scheduling parallel loops," *Communications of the ACM*, vol. 35, no. 8, 1992, pp. 90–101.

[41] R. C. I. Banicescu, "Addressing the stochastic nature of scientific computations via dynamic loop scheduling," *Electronic Transactions on Numerical Analysis*, vol. 21, 2005, pp. 66–80.

[42] S. Iqbal and G. F. Carey, "Performance analysis of dynamic load balancing algorithms with variable number of processors," *Journal of Parallel and Distributed Computing*, vol. 65, no. 8, Aug. 2005, pp. 934–948.

[43] M. M. Jaghoori, A. Movaghar, and M. Sirjani, "Modere: the model-checking engine of Rebeca," *Proceedings of the 2006 ACM symposium on Applied computing*, New York, NY, USA, 2006, pp. 1810–1815, ACM.

[44] E. Johnsen and O. Owe, "An asynchronous communication model for distributed concurrent objects," *Proceedings of the Second International Conference on Software Engineering and Formal Methods*, September 2004, pp. 188 – 197.

[45] E. Johnsen and O. Owe, "An asynchronous communication model for distributed concurrent objects," *Proceedings of the Second International Conference on Software Engineering and Formal Methods*, September 2004, pp. 188 – 197.

[46] S. C. Johnson, *Yacc: Yet another compiler-compiler*, Bell Laboratories, 1975.

[47] L. V. Kale and S. Krishnan, "CHARM++: a portable concurrent object oriented system based on C++," *ACM SIGPLAN Notices*, vol. 28, no. 10, Oct. 1993, pp. 91–108.

[48] P. Konghong and M. Hamdi, "Distro: a distributed static round-robin scheduling algorithm for bufferless Clos-Network switches," *IEEE Global Telecommunications Conference*, nov. 2002, vol. 3, pp. 2298 – 2302.

[49] B. Krammer, M. S. Mller, and M. M. Resch, "MPI Application Development Using the Analysis Tool MARMOT," *International Conference on Computational Science*, vol. 3038 of *Lecture Notes in Computer Science*, Springer Berlin, Heidelberg, 2004, pp. 464–471.

[50] C. P. Kruskal and A. Weiss, "Allocating Independent Subtasks on Parallel Processors," *IEEE Transaction on Software Engineering*, vol. 11, no. 10, 1985, pp. 1001–1016.

[51] V. Kumar, A. Grama, A. Gupta, and G. Karypis, *Introduction to parallel computing: design and analysis of algorithms*, Benjamin-Cummings Publishing Co., Inc., Redwood City, CA, USA, 1994.

[52] W. Leinberger, G. Karypis, V. Kumar, R. Biswas, and R. Biswas, "Load balancing across near-homogeneous multiresource servers," *In 9th Heterogeneous Computing Workshop*, 2000, pp. 60–71.

[53] F. C. Lin and R. M. Keller, "The Gradient Model Load Balancing Method," *IEEE Transaction on Software Engineering*, vol. 13, no. 1, 1987, pp. 32–38.

[54] R. Luecke, Y. Zou, J. Coyle, J. Hoekstra, and M. Kraeva, "Deadlock detection in MPI programs," *Concurrency and Computation: Practice and Experience*, vol. 14, no. 11, 2002, pp. 911–932.

[55] E. A. Luke, *A rule-based specification system for computational fluid dynamics*, doctoral dissertation, Mississippi State, MS, USA, 1999.

[56] T. Marsland, Y. Gao, and F. Lau, *A study of software multithreading in distributed systems*, Citeseer, 1995.

[57] I. Melatti, R. Palmer, G. Sawaya, Y. Yang, R. M. Kirby, and G. Gopalakrishnan, "Parallel and distributed model checking in Eddy," *Inernational Journal on Software Tools Technology Transfer*, vol. 11, no. 1, 2009, pp. 13–25.

[58] L. M. Ni, C.-W. Xu, and T. B. Gendreau, "A Distributed Drafting Algorithm for Load Balancing," *IEEE Transaction on Software Engineering*, vol. 11, no. 10, 1985, pp. 1153–1161.

[59] L. Oliker and R. Biswas, "PLUM : Parallel Load Balancing for Adaptive Unstructured Meshes," *Journal of Parallel and Distributed Computing*, vol. 52, no. 2, 1998, pp. 150 – 177.

[60] S. Owre, J. M. Rushby, and N. Shankar, "PVS: A prototype verification system," *Automated DeductionCADE-11*, 1992, pp. 748–752.

[61] F. Pellegrini and J. Roman, "Sparse Matrix Ordering with SCOTCH," *Proceedings of the International Conference and Exhibition on High-Performance Computing and Networking*, London, UK, UK, 1997, HPCN Europe '97, pp. 370–378, Springer-Verlag.

[62] C. D. Polychronopoulos and D. J. Kuck, "Guided self-scheduling: A practical scheduling scheme for parallel supercomputers," *IEEE Transaction on Computers*, vol. 36, no. 12, 1987, pp. 1425–1439.

[63] M. Poppleton and R. Banach, "Requirements validation by lifting retrenchments in B," *Proceedings of 9th IEEE International Conference on Engineering Complex Computer Systems*, 2004, pp. 87–96.

[64] X. Qin, H. Jiang, A. Manzanares, X. Ruan, and S. Yin, "Communication-Aware Load Balancing for Parallel Applications on Clusters," *IEEE Transactions on Computers*, vol. 59, no. 1, January 2010, pp. 42 –52.

[65] J. L. Quiroz-Fabian, M. Aguilar-Cornejo, G. Román-Alonso, and M. A. Castro-García, "Model Checking for Integrating Dynamic Load Distribution into Parallel Applications," *Proceedings of the 2008 Mexican International Conference on Computer Science*, Washington, DC, USA, 2008, pp. 221–231, IEEE Computer Society.

[66] B. Schaeli and R. D. Hersch, "Dynamic testing of flow graph based parallel applications," *Proceedings of the 6th workshop on Parallel and distributed systems*, New York, NY, USA, 2008, pp. 1–10, ACM.

[67] P. Schnoebelen, "The Complexity of Temporal Logic Model Checking," 2002.

[68] S. Sharma, S. Singh, and M. Sharma, "Performance Analysis of Load Balancing Algorithms," *World Academy of Science, Engineering and Technology*, vol. 38, 2008.

[69] S. F. Siegel, "Efficient verification of halting properties for MPI programs with wildcard receives," *Proceedings on the 6th International Conference on Verification, Model Checking, and Abstract Interpretation*, 2005, vol. 3385 of *Lecture Notes in Computer Science*, pp. 413–429.

[70] S. F. Siegel, "Using MPI-Spin to Model Check MPI Programs with Nonblocking Communication," 2006.

[71] S. F. Siegel and G. S. Avrunin, "Verification of MPI-Based Software for Scientific Computation," *Model Checking Software: 11th International SPIN Workshop*. 2004, pp. 286–303, Springer-Verlag.

[72] M. Sirjani, "Rebeca: theory, applications, and tools," *Proceedings of the 5th international conference on Formal methods for components and objects*, Berlin, Heidelberg, 2007, pp. 102–126, Springer-Verlag.

[73] M. Sirjani, A. Movaghar, A. Shali, and F. S. de Boer, "Modeling and Verification of Reactive Systems using Rebeca," *Fundamenta Informaticae*, vol. 63, no. 4, January 2004, pp. 385 –410.

[74] T. Suen and J. Wong, "Efficient task migration algorithm for distributed systems," *IEEE Transactions on Parallel and Distributed Systems,*, vol. 3, no. 4, jul 1992, pp. 488 –499.

[75] T. Y. Suen and J. K. Wong, "Efficient Task Migration Algorithm for Distributed Systems," *IEEE Transaction on Parallel and Distributed Systems*, vol. 3, no. 4, 1992, pp. 488–499.

[76] S. S. Vakkalanka, S. Sharma, G. Gopalakrishnan, and R. M. Kirby, "ISP: a tool for model checking MPI programs," *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, New York, NY, USA, 2008, pp. 285–286, ACM.

[77] A. van Deursen, P. Klint, and J. Visser, "Domain-specific languages: an annotated bibliography," *ACM SIGPLAN Notices*, vol. 35, no. 6, 2000, pp. 26–36.

[78] J. S. Vetter and B. R. de Supinski, "Dynamic software testing of MPI applications with umpire," *Proceedings of the 2000 ACM/IEEE conference on Supercomputing*. 2000, IEEE Computer Society.

[79] D. W. Walker, "The design of a standard message passing interface for distributed memory concurrent computers," *Parallel Computing*, vol. 20, no. 4, 1994, pp. 657 – 673.

[80] B.-Y. Zhang, Z.-Y. Mo, G.-W. Yang, and W.-M. Zheng, "Dynamic Load-Balancing and High Performance Communication in Jcluster," *IEEE International Parallel and Distributed Processing Symposium*, march 2007, pp. 1 –7.

[81] G. Zheng, E. Meneses, A. Bhatele, and L. V. Kale, "Hierarchical load balancing for Charm++ applications on large supercomputers," *39th IEEE International Conference on Parallel Processing Workshops*, 2010, pp. 436–444.

APPENDIX A

LEX SYMBOL FILE

A.1 Lexical Rules of the LBDSL Language

```
1  "$"                    return RULE;
2  "datatype"             return SYSTEMDEFINEDTYPE;
3  "constant"             return DEFINE;
4  "<"                    return LEFTARROW;
5  "int"                  return INT;
6  "float"                return FLOAT;
7  "double"               return DOUBLE;
8  "unsigned char"        return LONGCHAR;
9  ">"                    return RIGHTARROW;
10 "Enumerate_State"      return ENUMSTATE;
11 "="                    return EQUALS;
12 "{"                    return LBRACE;
13 "}"                    return RBRACE;
14 ","                    return COMMA;
15 ";"                    return SEMICOLON;
16 "."                    return DOT;
17 "*"                    return MULTIPLY;
18 "/"                    return DIVIDE;
19 "["                    return LSBRACE;
20 "]"                    return RSBRACE;
21 "+"                    return PLUS;
22 "-"                    return MINUS;
23 "=="                   return ISEQUAL;
24 ">="                   return GREATEROREQUAL;
25 "<="                   return LESSOREQUAL;
26 "!="                   return NOTEQUAL;
27 "&&"                   return AND;
28 "&"                    return POINTER;
29 "||"                   return OR;
30 "("                    return SPARAN;
31 ")"                    return RPARAN;
32 "->"                   return IMPLIES;
33 ":"                    return SINGLECOLON;
34 "::"                   return DOUBLECOLON;
35 "\""                   return DOUBLEQUOTE;
36 "'"                    return QUOTE;
37 "&"                    return BITAND;
38 "_"                    return UNDERSCORE;
39 "Handles"              return HANDLES;
40 "variable"             return VAR;
```

```
41 "buffer"              return BUFFER;
42 "Begin_Rule_P"        return BEGINPASSIVERULE;
43 "End_Rule_P"          return ENDPASSIVERULE;
44 "Begin_Message"       return BEGINMESSAGE;
45 "End_Message"         return ENDMESSAGE;
46 "Begin_Typedef"       return BEGINACTIVERULE;
47 "End_Typedef"         return ENDACTIVERULE;
48 "Typedef"             return TYPEDEF;
49 "IP"                  return INPUT;
50 "OP"                  return OUTPUT;
51 "array"               return AARRAY;
52 "Assign"              return VALUEASSIGN;
53 "Execute"             return EXECUTEMODULE;
54 "packMetaData"        return MESSAGEPACK;
55 "packLoad"            return LOADPACK;
56 "unPackMetaData"      return MESSAGEUNPACK;
57 "unPackLoad"          return LOADUNPACK;
58 "Send"                return MESSAGESEND;
59 "Receive"             return MESSAGERECV;
60 "SendMessage"         return SENDMESSAGE;
61 "ReceiveMessage"      return RECEIVEMESSAGE;
62 "pointer"             return POINTER;
63 "AssignOutput"        return POINTERRETURN;
64 "Begin_Process"       return BEGINACTOR;
65 "End_Process"         return ENDACTOR;
66 "SetState"            return SETSTATE;
67 "SaveState"           return SAVEACTION;
68 "SavedState"          return LASTACTION;
69 "Begin_Execute"       return BEGINWHILE;
70 "End_Execute"         return ENDWHILE;
71 "Until"               return UNTIL;
72 "Start_State"         return STARTSTATE;
73 "End_State"           return ENDSTATE;
74 "Next_State"          return NEXTSTATE;
75 "Poll_Wait"          return BLOCKPROBE;
76 "Poll_Test"          return NBLOCKPROBE;
77 "GetMessageLength"    return GETMSGLENGTH;
78 "Operation"           return OPERATION;
79 "Begin_If"            return BEGINIF;
80 "Else_If"             return ELSEIF;
81 "Else"                return ELSE;
82 "End_If"              return ENDIF;
83 "Begin_Loop"          return BEGINFOR;
```

```
84  "End_Loop"                  return ENDFOR;
85  "To"                        return TO;
86  "Update"                    return UPDATE;
87  "conveyV"                   return TRANSFER;
88  "computeV"                  return COMPUTE;
89  "storeV"                    return STORE;
90  "decisionV"                 return DECISION;
91  "GetChunk"                  return GETCHUNK;
92  "Begin_Compute"             return BEGINCOMPUTE;
93  "End_Compute"               return ENDCOMPUTE;
94  "GetLocalSize"              return GETLOCALSIZE;
95  "Begin_Module"              return BEGINMODULE;
96  "End_Module"                return ENDMODULE;
97  "reSize"                    return RESIZE;
98  "ConditionToTerminate"          return TERMINATECONDITION;
99  [*&]*[a-zA-Z]+[0-9_a-zA-Z]*
100                 {
101                 char *res=new char[strlen(yytext+1)];
102                 strcpy(res, yytext);
103                 yylval.sval = res;
104                 return IDENTIFIER;
105                 }
106
107 [-]*[0-9]+[.0-9]*
108                 {
109                 char *res=new char[strlen(yytext+1)];
110                 strcpy(res, yytext);
111                 yylval.sval = res;
112                 return SINTEGER;
113                 }
114
115 [ \t\n]+        ;
```

APPENDIX B

YACC RULE FILE

B.1 Analytical Grammer for the LBDSL Language

```
1
2  Program: body;
3
4  body:
5      | body body_lines;
6
7  body_lines:
8        declaration | enumeration
9      | embeddedcode | Handlers
10     | userdefineddatatype
11     | assignment | Actors   ;
12
13 assignment:
14       RULE VALUEASSIGN SPARAN IDENTIFIER COMMA
15       value RPARAN SEMICOLON
16     | RULE UPDATE SPARAN IDENTIFIER LSBRACE value
17       RSBRACE COMMA value RPARAN SEMICOLON
18     | RULE UPDATE SPARAN value COMMA value RPARAN
19       SEMICOLON
20     | RULE UPDATE SPARAN QUOTE operator QUOTE COMMA
21       value COMMA value RPARAN SEMICOLON
22     | RULE RESIZE SPARAN IDENTIFIER COMMA value
23       RPARAN SEMICOLON    ;
24
25
26 declaration:
27       RULE SYSTEMDEFINEDTYPE DEFINE LEFTARROW datatype
28       RIGHTARROW IDENTIFIER value
29     | RULE SYSTEMDEFINEDTYPE STORE LEFTARROW datatype
30       RIGHTARROW SPARAN IDENTIFIER COMMA value RPARAN
31       SEMICOLON
32     | RULE SYSTEMDEFINEDTYPE TRANSFER LEFTARROW
33       datatype RIGHTARROW SPARAN IDENTIFIER COMMA value
34       RPARAN SEMICOLON
35     | RULE SYSTEMDEFINEDTYPE DECISION LEFTARROW
36       datatype RIGHTARROW SPARAN IDENTIFIER COMMA value
37       RPARAN SEMICOLON
38     | RULE SYSTEMDEFINEDTYPE COMPUTE LEFTARROW datatype
39       RIGHTARROW SPARAN IDENTIFIER COMMA value RPARAN
40       SEMICOLON
```

```
41      | RULE SYSTEMDEFINEDTYPE TRANSFER LEFTARROW datatype
42        RIGHTARROW IDENTIFIER SEMICOLON
43      | RULE SYSTEMDEFINEDTYPE COMPUTE LEFTARROW datatype
44        RIGHTARROW IDENTIFIER  SEMICOLON
45      | RULE SYSTEMDEFINEDTYPE STORE LEFTARROW datatype
46        RIGHTARROW IDENTIFIER SEMICOLON
47      | RULE SYSTEMDEFINEDTYPE STORE LEFTARROW datatype
48        RIGHTARROW IDENTIFIER LSBRACE value RSBRACE
49        SEMICOLON ;
50
51  datatype:
52        INT | FLOAT | DOUBLE | IDENTIFIER
53      | SINTEGER | DECISION | LONGCHAR ;
54
55  value:
56        datatype | value DOT value
57      | value IMPLIES value
58      | IDENTIFIER LSBRACE datatype operator
59        datatype RSBRACE
60      | IDENTIFIER LSBRACE datatype operator datatype
61        operator datatype RSBRACE
62      | IDENTIFIER LSBRACE RSBRACE
63      | IDENTIFIER LSBRACE IDENTIFIER RSBRACE ;
64
65  enumeration:
66        RULE ENUMSTATE EQUALS LBRACE item RBRACE
67        SEMICOLON ;
68
69  item:
70        IDENTIFIER | IDENTIFIER COMMA item ;
71
72  embeddedcode:
73        RULE BEGINCOMPUTE | embed | RULE ENDCOMPUTE  ;
74
75  embed:
76    | embed embedded_code ;
77
78  embedded_code:
79       datatype | value | punctuation
80    | brackets | comparision | operator ;
81
82  punctuation:
83        SEMICOLON | COMMA
```

```
84      | EQUALS    | DOT
85      | IMPLIES   | DOUBLECOLON
86      | SINGLECOLON | UNDERSCORE
87      | DOUBLEQUOTE   ;
88
89  brackets:
90        SPARAN | RPARAN
91      | LBRACE | RBRACE
92      | LEFTARROW | RIGHTARROW
93      | LSBRACE | RSBRACE   ;
94
95  comparision:
96        AND | OR | GREATEROREQUAL
97    | LESSOREQUAL | NOTEQUAL
98    | ISEQUAL | LEFTARROW
99    | RIGHTARROW | BITAND ;
100
101 operator:
102     PLUS | MINUS | MULTIPLY | DIVIDE ;
103
104 Handlers:
105     passive_handlers | messagetypes ;
106
107 passive_handlers:
108     handle_head handle_body handle_tail ;
109
110 handle_head:
111     RULE BEGINPASSIVERULE IDENTIFIER SPARAN
112     argument RPARAN ;
113
114 argument:
115   | COMPUTE LEFTARROW datatype RIGHTARROW
116     IDENTIFIER
117   | TRANSFER LEFTARROW datatype RIGHTARROW
118     IDENTIFIER
119   | STORE LEFTARROW datatype RIGHTARROW
120     IDENTIFIER
121   | COMPUTE LEFTARROW datatype RIGHTARROW
122     IDENTIFIER ending  argument
123   | TRANSFER LEFTARROW datatype RIGHTARROW
124     IDENTIFIER ending  argument
125   | STORE LEFTARROW datatype RIGHTARROW
126     IDENTIFIER ending  argument
```

```
127     | value | value COMMA argument   ;
128
129 handle_tail:
130       RULE ENDPASSIVERULE ;
131
132 ending:
133       COMMA | SEMICOLON ;
134
135 handle_body:
136     | embeddedcode   ;
137
138 userdefineddatatype:
139       RULE BEGINACTIVERULE TYPEDEF IDENTIFIER
140       msg_type_body RULE ENDACTIVERULE ;
141
142 msg_type_body:
143       SYSTEMDEFINEDTYPE VAR LEFTARROW datatype
144       RIGHTARROW IDENTIFIER
145     | SYSTEMDEFINEDTYPE VAR LEFTARROW datatype
146       RIGHTARROW IDENTIFIER SEMICOLON msg_type_body ;
147
148 messagetypes:
149       messagetype_head | messagetype_body
150     | messagetype_tail ;
151
152 messagetype_head:
153       RULE BEGINMESSAGE IDENTIFIER SPARAN
154       argument RPARAN ;
155
156 messagetype_tail:
157       RULE ENDMESSAGE ;
158
159 messagetype_body:
160       declaration | module ;
161
162 module:
163       RULE BEGINMODULE SENDMESSAGE
164     | RULE BEGINMODULE RECEIVEMESSAGE
165     | RULE ENDMODULE | modulebody ;
166
167 modulebody:
168       assignment | embeddedcode | executemodule
169     | packmessage | messagesend | unpackmessage
```

```
170     | messagerecv | pointerassignment
171     | mathematical_operations
172     | BEGINCOMPUTE
173     | ENDCOMPUTE | ifelse ;
174
175
176 executemodule:
177       RULE IDENTIFIER SPARAN argument RPARAN SEMICOLON   ;
178
179 packmessage:
180       RULE MESSAGEPACK SPARAN value COMMA value COMMA
181       value COMMA value RPARAN SEMICOLON
182     | RULE LOADPACK SPARAN value COMMA  value COMMA
183       value COMMA value COMMA value COMMA
184       value RPARAN SEMICOLON   ;
185
186 unpackmessage:
187       RULE MESSAGEUNPACK SPARAN value COMMA value COMMA
188       value COMMA value RPARAN SEMICOLON
189     |  RULE LOADUNPACK SPARAN value COMMA
190       value COMMA value COMMA value COMMA
191       value COMMA value RPARAN SEMICOLON   ;
192
193 messagesend:
194       RULE MESSAGESEND SPARAN IDENTIFIER COMMA
195       IDENTIFIER COMMA IDENTIFIER COMMA
196       IDENTIFIER COMMA IDENTIFIER RPARAN SEMICOLON ;
197
198 messagerecv:
199       RULE MESSAGERECV SPARAN IDENTIFIER COMMA
200       IDENTIFIER COMMA IDENTIFIER COMMA
201       IDENTIFIER COMMA IDENTIFIER COMMA
202       IDENTIFIER RPARAN SEMICOLON   ;
203
204 pointerassignment:
205       RULE POINTERRETURN SPARAN value COMMA
206       datatype LSBRACE SINTEGER RSBRACE
207       RPARAN SEMICOLON ;
208
209 Actors:
210       actorhead | actorbody | actortail ;
211
212 actorhead:
```

```
213       RULE BEGINACTOR IDENTIFIER SPARAN argument
214       RPARAN ;
215
216 actortail:
217       RULE ENDACTOR ;
218
219 actorbody:
220       messagetype_body | initializestate
221     | whileloop | ifelse
222     | communication_operations
223     | messagetypes | for_loop
224     | chunkoperations
225     | RULE TERMINATECONDITION condition SEMICOLON ;
226
227 initializestate:
228       RULE SETSTATE SPARAN LASTACTION RPARAN SEMICOLON
229     | RULE SETSTATE SPARAN value RPARAN SEMICOLON
230     | RULE SAVEACTION SPARAN value RPARAN SEMICOLON ;
231
232
233 whileloop:
234       RULE BEGINWHILE SINGLECOLON UNTIL
235     | condition | comparision
236     | RULE ENDWHILE ;
237
238 condition:
239       SPARAN value comparision value RPARAN
240     | SPARAN value comparision value RPARAN
241       comparision condition  ;
242
243 ifelse:
244       RULE STARTSTATE IDENTIFIER SINGLECOLON
245     | RULE ENDSTATE IDENTIFIER SINGLECOLON
246     | RULE NEXTSTATE IDENTIFIER SINGLECOLON
247     | RULE BEGINIF | condition
248     | comparision  | IMPLIES
249     | RULE ENDIF   | RULE ELSE
250     | RULE ELSEIF ;
251
252 for_loop:
253       RULE BEGINFOR SPARAN value SINGLECOLON
254       value TO value RPARAN
255     | RULE ENDFOR ;
```

```
256
257 communication_operations:
258       blockprobe | generatemessagelength
259     | nblockprobe ;
260
261 blockprobe:
262     RULE BLOCKPROBE  SPARAN value COMMA
263     value COMMA value RPARAN SEMICOLON ;
264
265 nblockprobe:
266     RULE NBLOCKPROBE SPARAN value COMMA
267     value COMMA value COMMA value RPARAN SEMICOLON;
268
269 generatemessagelength:
270     RULE GETMSGLENGTH SPARAN value COMMA
271     value RPARAN SEMICOLON;
272
273 messagetypes:
274     RULE SENDMESSAGE SPARAN value COMMA
275     value COMMA value COMMA value COMMA value
276     RPARAN SEMICOLON
277   | RULE RECEIVEMESSAGE SPARAN value COMMA
278     value COMMA value COMMA value COMMA value
279     RPARAN SEMICOLON ;
280
281 mathematical_operations:
282     RULE OPERATION SPARAN operation RPARAN SEMICOLON
283   | RULE OPERATION SPARAN operation RPARAN ;
284
285 operation:
286     value COMMA value COMMA QUOTE operator QUOTE
287     COMMA value ;
288
289 chunkoperations:
290     RULE GETCHUNK SPARAN value COMMA value COMMA
291     value COMMA value COMMA value COMMA value
292     RPARAN SEMICOLON
293   | RULE GETLOCALSIZE SPARAN value COMMA value
294     COMMA value COMMA value RPARAN SEMICOLON ;
```