The Thesis committee for Karthik Srinivasa Murthy
Certifies that this is the approved version of the following thesis

A Proposed Memory Consistency Model For Chapel

APPROVED BY

SUPERVISING COMMITTEE:

Calvin Lin, Supervisor

Brad Chamberlain

A Proposed Memory Consistency Model For Chapel

by

Karthik Srinivasa Murthy, B.E.

THESIS

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

MASTER OF ARTS

THE UNIVERSITY OF TEXAS AT AUSTIN

August 2010

Dedicated to people, cs and math.

Acknowledgments

"It is the dream of every graduate student to find in his advisor a true mentor" - Lorenzo Alvisi, 1996. I have found mine. My advisor, Dr. Calvin Lin, has been my mentor and my friend for the past 3 years. He has helped me at every step of my stay in UT Austin. He has introduced me to the wonderful world of parallel systems and this thesis would not have been possible without his guidance. I believe my best work at UT Austin came as a TA for CS380P in the spring of 2010. Dr. Lin's confidence in me helped me get and do a good job as the TA. I am eternally thankful to him.

I also thank Dr. Brad Chamberlain, who co-supervised my master's thesis. Dr. Brad Chamberlain is a wonderful guide. His questions and directions have given structure to this thesis. He has treated me as a peer during our discussions and has helped me enjoy working on this thesis.

I also thank Dr. Lorenzo Alvisi, my graduate advisor, and the CS department at UT Austin for these three wonderful years.

I thank my family and friends for their support during this journey of mine.

A Proposed Memory Consistency Model For Chapel

Karthik Srinivasa Murthy, M.A.
The University of Texas at Austin, 2010

Supervisor: Calvin Lin

A memory consistency model for a language defines the order of memory operations performed by each thread in a parallel execution. Such a constraint is necessary to prevent the compiler and hardware optimizations from reordering certain memory operations, since such reordering might lead to unintuitive results. In this thesis, we propose a memory consistency model for Chapel, a parallel programming language from Cray Inc.

Our memory model for Chapel is based on the idea of multiresolution and aims to provide a migration path from a program that is easy to reason about to a program that has better performance efficiency. Our model allows a programmer to write a parallel program with sequential consistency semantics, and then migrate to a performance-oriented version by increasingly changing different parts of the program to follow relaxed semantics. Sequential semantics helps in reasoning about the correctness of the parallel program and is provided by the strict sequential consistency model in our proposed memory model. The performance-oriented versions can be obtained either by using the

compiler sequential consistency model, which maintains the sequential semantics, or by the relaxed consistency model, which maintains consistency only at global synchronization points. Our proposed memory model for Chapel thus combines strict sequential consistency model, compiler sequential consistency model and relaxed consistency model.

We analyze the performance of the three consistency models by implementing three applications: Barnes-Hut, FFT and Random-Access in Chapel, and the hybrid model of MPI and Pthread. We conclude the following:

- The strict sequential consistency model is the best model to determine algorithmic errors in the applications, though it leads to the worst performance.
- The relaxed consistency model gives the best performance among the three models, but relies on the programmer to enforce synchronization correctly.
- The performance of the compiler sequential model depends on accuracy of the dependence analysis performed by the compiler.
- The relative performance of the consistency models across Chapel and the hybrid programming model of MPI and Pthread are the same. This shows that our model is not tightly bound to Chapel and can be applied on other programming models/languages.

Table of Contents

Acknowledgments **v**

Abstract **vi**

List of Tables **xi**

List of Figures **xii**

Chapter 1. Introduction **1**
1.1 Our solution 5
1.2 Performance Analysis 6
1.3 Overview of Chapters 7

Chapter 2. Related Work **8**
2.1 Hardware Consistency Model: IA64 10
2.1.1 Description 10
2.1.2 Comparison with the Chapel model 12
2.2 Unified Parallel C 12
2.2.1 Description 13
2.2.2 Comparison with the Chapel model 14
2.2.3 Comparison with Chapel model via Examples 15
2.3 Java 18
2.3.1 Description 18
2.3.2 Comparison with the proposed Chapel model 20
2.4 Summary 20

Chapter 3. Proposed Memory Consistency Model in Chapel 23
3.1 Background . . . 24
3.1.1 Sync variables . . . 24
3.1.2 Fences . . . 26
3.2 Strict Sequential Consistency Model . . . 26
3.2.1 Description . . . 26
3.2.2 Analysis of the Strict Memory Model . . . 27
3.2.3 Implementation of *strict_seq* construct . . . 29
3.3 Compiler Sequential Consistency Model . . . 30
3.3.1 Description . . . 30
3.3.2 Analysis of the Compiler Sequential Consistency Model 31
3.3.3 Implementation of Delay Set Analysis . . . 35
3.4 Relaxed Consistency Model . . . 37
3.4.1 Description . . . 37
3.4.2 Analysis of the Relaxed Memory Model . . . 38
3.4.3 Implementation of Relaxed Memory Model . . . 38
3.5 Interaction of memory models . . . 38
3.6 Summary . . . 39

Chapter 4. Performance Analysis of Memory Model 41
4.1 Barnes-Hut . . . 42
4.1.1 Implementation . . . 43
4.1.2 Strict Sequential Consistency . . . 43
4.1.3 Compiler Sequential Consistency . . . 44
4.1.4 Analysis of Performance . . . 46
4.2 FFT . . . 48
4.2.1 Implementation . . . 49
4.2.2 Strict Sequential Consistency . . . 50
4.2.3 Compiler Sequential Consistency . . . 51
4.2.4 Analysis of Performance . . . 53
4.3 Random Access . . . 56
4.3.1 Implementation . . . 56

4.3.2 Strict Sequential 56
4.3.3 Compiler Sequential 57
4.3.4 Performance Analysis 57
4.4 Summary 59

Chapter 5. Conclusion **63**

Chapter 6. Future Work **65**

Appendix **66**

Appendix 1. Chapel Statements in Strict_Seq Block **67**

Index **69**

Bibliography **70**

Vita **73**

List of Tables

2.1 Outstanding write operations (example taken from the Intel manual) 11
2.2 Chapel memory model Vs Hardware memory model 12
2.3 Chapel memory model Vs UPC memory model 15
2.4 Example 1 in UPC 16
2.5 Example 1 in Chapel 16
2.6 Example 2 in UPC 17
2.7 Example 2 in Chapel 18
2.8 Causal loops 19
2.9 Chapel memory model Vs Java memory model 22

3.1 Proposed memory models at a glance. 40

List of Figures

3.1 Example 1 after Delay Set Analysis 33
3.2 Dekker's Algorithm after Delay Set Analysis 34
3.3 Example 1 with a mixed cycle 37

4.1 Application: Barnes-Hut. Performance of the Chapel Implementations. 47
4.2 Application: Barnes-Hut. Performance of the MPI Implementations. 48
4.3 Application: FFT. Performance of the Chapel Implementations. 54
4.4 Application: FFT. Performance of the MPI Implementations. . 55
4.5 Application: RA. Performance of the Chapel Implementations. 58
4.6 Application: RA. Number of errors noted under relaxed consistency. 60
4.7 Application: RA. Performance of the MPI Implementations. . 61
4.8 Spectrum of Applications based on degree to which data dependences are statically determinate. 62

Chapter 1

Introduction

The advent of multicore processors has led to the rise of new parallel programming languages and models. Hardware and compiler optimizations, which help sequential languages obtain good performance efficiency, can help parallel languages to achieve the same. However, in a parallel scenario, with multiple accesses to memory occurring concurrently and complex causal relations among actions of distinct parallel threads, these optimizations can lead to inconsistent executions[1] producing unintuitive results. As parallel language developers, we believe that the answer is a good memory consistency model which defines how a program interacts with memory.

To illustrate the need for a consistency model, consider the simplified form of Dekker's algorithm for mutual exclusion shown in Listing 1.1. In this simplified version, T1 and T2 are threads executing the critical section *'critical_sec'* based on the state of shared variables *flag1* and *flag2*. In a sequentially consistent execution, T1 enters *critical_sec* only after raising *flag1* and only if the condition (*flag2==0*) holds true. Similarly, T2 enters *critical_sec* only

[1]inconsistent executions of a program are executions whose output does not match the output of any sequentially consistent execution of the program. A sequential consistent execution is an execution which follows Lamport's definition [8] of sequential consistency.

after raising *flag2* and only if the condition (*flag1==0*) holds true. Thus, a sequentially consistent version guarantees mutual exclusion of *critical_sec*.

```
1  Initially flag1=flag2=0.
2        T1                    T2
3  flag1=1;              flag2=1;
4  if(flag2==0){         if(flag1==0){
5    critical_sec          critical_sec
6  }                     }
```

Listing 1.1: Dekker's algorithm for mutual exclusion.

However, an optimizing compiler could prefetch the values of *flag2* and *flag1* in thread 1 and 2 respectively, violating sequential consistency. This leads to both T1 and T2 entering the *critical_sec*, violating mutual exclusion. The compiler is free to do this, as there is no data dependency between lines *3* and *4* of the code.

We can avoid such violations of mutual exclusion by enforcing a sequential memory consistency model, which orders execution according to Lamport's definition [8] of sequential consistency:
the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program.

On the other hand, the sequential consistency model might lead to poor performing parallel code, because it does not allow any optimizations that reorder code. An alternative is to use a relaxed consistency model, which

provides consistency at global synchronization points. When a relaxed model is enforced on the code in Listing 1.1, it will still allow the compiler optimization to reorder instructions, unless there is a global synchronization point such as a *barrier* introduced by the programmer between lines *3* and *4* of the code. The relaxed model places the responsibility of introducing the synchronization primitives needed for correctness on the programmer.

A memory consistency model thus helps a developer to reason about and control the output of a parallel execution.

There have been several memory models proposed in the last two decades. However, no memory model has been universally accepted. We feel that the main reasons are as follows:

- No memory model proposed to date has a solution for each of the three players in a parallel system, namely: the programmer, the compiler and the hardware.
- Most of the memory models express their consistency models in terms of low-level loads and stores, which is very difficult for programmers to reason about. We believe that these models should be defined in terms of programming language statements.

In this thesis, we propose a memory consistency model for Chapel. Chapel is a parallel programming language from Cray Inc. and is one of

the languages competing in the DARPA HPCS program. While our memory model can be be applied to any parallel programming model, we have chosen Chapel to showcase our model for the following reasons:

- One of the aims of Chapel is to provide high productivity (performance, programmability, portability and robustness). We believe that Chapel has delivered on the programmability front and is striving to deliver on the performance front. Since the Chapel team is in the process of implementing compiler optimizations, we feel that it is the ideal time to apply our proposed memory consistency model.

- As a multiresolution language, Chapel supports a good "separation of concerns" [6]. Our memory model is built on this idea and provides a migration path from a program that is easy to reason about to a program that has good performance efficiency. Our model allows a programmer to write a parallel program with sequential consistency semantics, which can then be migrated to a more performance-oriented version without excessive rewriting of code.

- Any memory model needs the support of good synchronization constructs. The Chapel language provides synchronization variables which provide full/empty semantics and are expressive enough to capture the needs of any memory model.

1.1 Our solution

Built on the idea of multiresolution, our memory model allows a programmer to write a parallel program with sequential consistency semantics, and then migrate to an advanced more performance-oriented version by increasingly relaxing the consistency of different parts of the program. Moreover, the performance-oriented version can also be obtained with the help of the compiler. Our memory model is therefore a combination of the following three types of consistency models:

- **Strict Sequential Consistency for the cautious programmer.**
  A cautious programmer favors the usage of sequential semantics in reasoning about the correctness of parallel code. For such a programmer, we provide the Strict Sequential Consistency model. A block of Chapel code executing under strict sequential semantics follows Lamport's definition [8] of sequential consistency, while the rest of the code follows either compiler sequential consistency or relaxed consistency.

- **Compiler Sequential Consistency for the trusting programmer**
  A trusting programmer delegates the responsibility of avoiding inconsistent executions (from code snippets as in Listing 1.1) to the compiler. We introduce the Compiler Sequential consistency model for such programmers. In this model, the compiler provides sequential consistency in the parallel program by introducing *fence* operations at appropriate places in the program to break race conditions. *Fences* prevent compil-

er/hardware re-ordering of instructions across the point in the parallel program where they are inserted. *Fences* ensure that all outstanding memory operations are complete. Shasha and Snir's Delay Set Analysis [13] is used to identify the points where *fences* should be inserted.

- **Relaxed consistency for the advanced programmer.**
  An advanced programmer wants to exploit compiler optimizations to get maximum performance efficiency. The programmer retains the responsibility of avoiding inconsistent executions by enforcing constraints via the synchronization constructs provided by the language. For such a programmer, we introduce the relaxed consistency model which provides consistency at global synchronization points.

1.2 Performance Analysis

To evaluate the performance implications of the three consistency models, we implement three applications: Barnes-Hut, FFT, Random-Access in Chapel and a hybrid MPI and Pthreads model. We conclude the following:

- Strict Sequential is the best model to determine algorithmic errors in the applications, though it leads to the worst performance. Programmers should use the strict sequential consistent version as a reference version to compare against other versions.

- Relaxed consistency gives the best performance among the three models, but it relies on the programmer to enforce synchronization correctly.

- The performance of the compiler sequential model depends on accuracy of the compiler's dependence analysis (which consists of reference analysis, synchronization analysis, may-happen-in-parallel (MHP) analysis). A conservative analysis leads to the introduction of more *fence* operations, in the extreme case bringing down the performance of this model to a strict sequentially consistent version. An accurate analysis leads to a smaller number of *fence* operations, which in the best case yields performance equivalent to that of the relaxed consistency version. In both cases the compiler sequential version does not compromise on the sequential consistency guarantee.

- The relative performance of the consistency models across Chapel and across the hybrid programming model of MPI and Pthread are the same. This shows that irrespective of the programming model, the developer pays the same performance overhead when moving among the different consistency models. This also shows that our model is not tightly bound to Chapel and can be applied on other programming models/languages.

1.3 Overview of Chapters

Chapter 2 provides an overview of the memory models of popular programming languages. Chapter 3 describes the proposed memory model in detail in the context of Chapel language. Chapter 4 analyses the performance of the proposed memory model. Chapter 5 concludes the thesis with our findings. Chapter 6 suggests future work.

Chapter 2

Related Work

Over the past two decades, a number of memory models have been proposed. The basic idea that unites all of these models is the aim to provide coherence among parallel threads of execution and to enforce ordering constraints in the same thread. Providing coherence means ensuring all the threads observe the writes to a memory location in the same order and at the same time. Enforcing ordering constraint means respecting the data and flow dependencies in the thread of execution. These memory models, however, differ based on the following factors:

- **Points in the program where coherence is provided.** If the model maintains coherence at each point in the program then the model provides sequential consistency If the model provides coherence only at synchronization points then this form of weakened sequential consistency is called relaxed consistency.

  The relaxed consistency performs better when compared to the sequential consistency because it allows all possible compiler and hardware optimizations but, it rests the responsibility of enforcing the necessary synchronization to prevent inconsistent executions (as discussed in the

case of Dekkars algorithm in Chapter 1) on the programmer. This factor captures one of the most important trade-offs in memory models i.e. between performance and correctness.

- **Who enforces the order of memory operations.** There are three players in this category: the hardware, the compiler, and the programmer via the use of synchronization constructs such as locks. We believe a memory model which supports the idea of multiresolution must have a solution to each of these categories but, no memory model till date has provided such a solution.

- **Granularity of the memory operations.** The granularity at which the semantics of a memory consistency model is defined varies from low-level load and store operations to programming language statements. We believe the right level of abstraction to be programming language statements, because it is difficult to reason about the consistency semantics of a program in terms of low-level load and store operations.

- **Data dependency between memory operations.** Some memory models enforce ordering constraints on independent memory operations, while other memory models enforce the constraint on data dependent memory operations. We believe it is useful to enforce constraints on all memory operations to improve the comprehensibility of the program.

- **Memory operations are differentiated as strict/relaxed.** Some languages allow the programmer to differentiate shared variables as strict

or relaxed. The memory model enforces sequential consistency among operations involving strict variables and relaxed consistency among operations involving relaxed variables. Such models cannot be applied to all programming models and also lead to complicated semantics when blocks of code contain both strict and relaxed variables.

In this chapter we discuss the following memory models:

- The IA64 hardware Memory Consistency Model
- The UPC Memory Consistency Model
- The Java Memory Consistency Model

2.1 Hardware Consistency Model: IA64

In this section, we discuss the IA64 memory consistency model.

2.1.1 Description

The IA64 memory model is a weak (relaxed) consistency model [3]. The model allows hardware optimizations such as write buffering with bypassing to reorder memory operations. These optimizations are applied to hide memory latency because accessing main memory is a costly operation. While these optimizations result in faster executions, they can cause inconsistent executions unless the programmer has used the synchronization constructs correctly.

To illustrate the effect of write buffering with bypassing optimization, consider the assembly code in Table 2.1 (which has been taken from the Intel IA64 manual [3]). This optimization allows execution to continue in the presence of outstanding write operations. Reads to a location whose write is outstanding gives the latest value to the processor which issued the outstanding write, whereas another processor might obtain a stale value.

Processor 0	Processor 1
mov [_x], 1	mov [_y], 1
mov r1, [_y]	mov r2, [_x]

Table 2.1: Outstanding write operations (example taken from the Intel manual)

```
[_x], [_y] are memory locations of x and y.
r1 and r2 are registers.
Intuitive Output: r1==1 and r2==1 or
          r1==1 and r2==0 or
          r1==0 and r2==1.
Unintuitive Output: r1==0 and r2==0.
```

The result (r1==0 and r2==0) is not consistent with the output of any sequential execution of the code in Table 2.1. Such code snippets lead to violations of mutual exclusion of code, when code written in a higher level language is translated to code snippet in Table 2.1. However, IA64 also exposes memory fence operations to help the programmer control such reorderings. The memory fence operations prevent the movement of operations across the point where they are inserted in the program thus helping in enforcing sequential consistency.

2.1.2 Comparison with the Chapel model

The table 2.2 gives a comparison between the proposed Chapel memory model and the IA64 memory model.

IA64 Memory Model	Proposed Chapel Memory Model
Model is described in terms of low level loads and stores.	Model is described in terms of high level language statements.
Memory Model is weak. It allows for reordering of memory operations.	Both strict and weak models are provided. The relaxed model is similar to the weak model.
Programmer uses memory *fence* operations to control reordering of operations and establish sequential consistency.	In the strict sequential model, the programmer does not need memory fences to enforce sequential consistency. In the compiler sequential model, the compiler (not the programmer) uses memory fences to enforce sequential consistency. In the relaxed model, programmer uses Chapel's *sync* constructs to enforce sequential consistency.

Table 2.2: Chapel memory model Vs Hardware memory model

2.2 Unified Parallel C

In this section, we describe the memory model of the UPC language.

2.2.1 Description

UPC language is an extension of the C language for parallel computing, which provides an SPMD (single program multiple data) programming model. UPC is a PGAS (partitioned global address space) language, which means that a shared variable in the UPC program can be read and written by any processor but is present in the physical memory of one processor.
UPC provides the programmer with two types of shared variables: *shared* and *shared strict* [2]. The *shared* variables are also called the relaxed variables. UPC's memory model ([2]) can be summarized as:

- Model enforces a $<\_strict$ [2] ordering across all threads. The <_strict is a partial order which enforces an ordering constraint between every pair of *shared strict* variable operations of all the threads.
- Model enforces a $<\_t$ [2] ordering in each thread. This <_t ordering is a total order which enforces an ordering constraint in each thread between every pair of operations where each operation consists of:
  - All the write operations(*shared*, *shared strict*, non-shared) in the program.
  - All the *shared strict* read operations in the program.
  - All the write/read operations(*shared*, *shared strict*, non-shared) in the thread.

- The ordering constraint mentioned in the previous two points is obtained by respecting the data and flow dependencies in the input program. In UPC, the ordering constraint is described via a *precedes* ([2]) relationship. A formal definition as stated in "Formal UPC Memory Consistency Semantics" section of UPC manual [2] is as follows:
  a formal definition for the Precedes(m1,m2) partial order, a predicate which inspects two memory operations in the execution trace that were issued by the same thread and returns true if and only if m1 is required to precede m2, according to the sequential abstract machine semantics of [ISO/IEC00 Sec. 5.1.2.3], applied to the given thread.

- the Model also provides for synchronization operations such as
  - *upc_fence* which prevents the movement of any memory operations across it.
  - *upc_lock* acts as an acquire barrier (prevents the movement of operations from after to before *upc_lock*).
  - *upc_unlock* acts as an release barrier (prevents the movement of operations from before to after *upc_unlock*).

2.2.2 Comparison with the Chapel model

The table 2.3 gives a comparison between the proposed Chapel memory model and the UPC memory model.

UPC Memory Model	Proposed Chapel Memory Model
Memory Model provides both strict and relaxed models. The programmer can obtain sequential consistency by using only *shared strict* variables. The programmer can obtain relaxed memory consistency by using only *shared* variables.	Model provides both strict(strict sequential) and relaxed (relaxed consistency) models.
Model is described in terms of low level loads and stores. It is difficult for the programmer to identify whether variables need to be *shared strict* or *shared*.	Model is described in terms of high level programming language statements. Changing strict and relaxed models involves a removal of the (*strict_seq*) construct or a command line parameter.
Absence of a compiler option translates to more effort by the programmer in enforcing memory barrier operations.	A convenient compiler option for programmers who want performance and sequential consistency but without getting their hands dirty.

Table 2.3: Chapel memory model Vs UPC memory model

2.2.3 Comparison with Chapel model via Examples

The UPC model is very similar to the proposed Chapel model. The different consistency requirements of any program can be achieved by both the proposed Chapel and UPC memory model.

We show some example pieces of code where the Chapel code can achieve the same consistency as the UPC model by using the proposed memory model. *Note: Examples are from the UPC Specification 1.2.*

- Example 1. All the variables are initialized to 0.

T0	T1
RR(y,1)	RR(x,2)
RW(x,2)	RW(y,1)

Table 2.4: Example 1 in UPC

Note:RR(x,1) - relaxed read of shared variable x yielded the value 1, RW(x,1) - relaxed write of value 1 to shared variable x.

In this example, Thread T0 reads the value of y as 1, which indicates that both the operations of T1 were completed. But, T1 reads the value of x as 2, which means that T0 should have completed both of its operations. Such a behavior is possible only when T0 and T1 execute their second operations first. This reordering was allowed because these were operations containing only relaxed variables.

Such a behavior can be obtained in Chapel by writing code as follows.

T0	T1
temp1 = y	temp2 = x
x = 2	y = 1
writeln(temp1)	writeln(temp2)

Table 2.5: Example 1 in Chapel

```
All variables are initialized to 0.
Possible Output: temp1=1 and temp2=2.
```

In the absence of a strict sequential or compiler sequential model, Chapel

compiler allows reordering of statements providing the relaxed semantics of UPC.

- Example 2. All variables are initialized to 0.

T0	T1
WR(y,1)	WR(x,0)
WW(x,2)	WW(y,1)

Table 2.6: Example 2 in UPC

Note:WR(x,1) - strict read of shared variable x yielded value 1,

WW(x,1) - strict write of value 1 to shared variable x.

In this example, Thread T0 reads the value of y as 1 which means that both the operations of T1 were completed. T1 reads the value of x as 0 which means that T0 did not complete its write of x. Such a behavior is consistent with a sequential execution of program. The execution order is depicted in Listing 2.1.

```
        T1 :WR( x , 0 )
             |
2       T1 :WW( y , 1 )
             |
4       T0 :WR( y , 1 )
             |
6       T0 :WW( x , 2 )
```

Listing 2.1: Order of execution of statements in T0 and T1.

Such a behavior can be obtained in Chapel by writing code as follows.

T0	T1
strict_seq {temp1 = y x = 2 writeln(temp1)}	strict_seq {temp2 = x y = 1 writeln(temp2)}

Table 2.7: Example 2 in Chapel

```
All variables are initialized to 0.\\
Possible Output: temp1=1 and temp2=0.\\
```

This output of the Chapel code is consistent with a sequential execution of the program code.

2.3 Java

In this section, we discuss the Java memory model.

2.3.1 Description

Java is an object oriented programming language. One of the popular uses of the language is in writing multithreaded code. Java's memory model has the following characteristics:

- A correctly synchronized program exhibits sequential consistency [9]. This principle states that if the programmer has enforced synchronization correctly then the execution of such a program will be consistent with the output of any sequentially consistent execution of the program.

- An incorrectly synchronized program does not violate security guarantees [9].

  This principle states that in the absence of proper synchronization, the execution shall not violate the security guarantee of the program.

  Note: All variables are 0 initially, r1 and r2 are register variables. *The below example is taken from [9].*

T0	r1=x	y = r1
T1	r2 =y	x = r2

Table 2.8: Causal loops

To illustrate this principle, consider the example in Table 2.8.
In the example, if an aggressive compiler predicts (optimization:*value prediction*) that the value of x is 42 in T0, then this causes r1 in T0 to get 42, y in T0 then gets the value of 42, r2 in T1 then gets 42. Then x in T1 gets 42, thus justifying the prediction of x as 42. This causal cycle is presented in Listing 2.2. Such pieces of code can also generate random object references leading to violation of security. Java avoids such scenarios with the principle:
"Early execution of an action does not result in an undesirable causal cycle if its occurance is not dependant on a read returning a value from a data race." [9] When this principle is applied in the above scenario, it prevents the reading of 42 for x.

```
T0:predict x as 42
     |
T0:r1=x, r1->42
     |
T0:y=r1, y->42
     |
T1:r2=y, r2->42
                |
T1:x=r2, x->42
                |
T0:x is 42!
```

Listing 2.2: Order of execution of statements in Table 2.8 in the absence of memory model.

- Java provides strong synchronization constructs like *volatile* variables, *synchronize* constructs which act as memory barriers and prevent the movement of operations from moving across them.

2.3.2 Comparison with the proposed Chapel model

The table 2.9 gives a comparison between the proposed Chapel memory model and the Java memory model.

2.4 Summary

In this chapter, we have shown that the Chapel memory model is capable of providing the same semantics as other memory models. We have also described scenarios where the Chapel model provides a better comprehensibility (by comparing proposed model with the hardware consistency model). By

providing a combination of three memory consistency models, we believe that the proposed model has delivered a multiresolution design which is lacking in the other memory models.

Java Memory Model	Chapel Memory Model
Model is described in terms of low level loads and stores.	Model is described in terms of high level language statements.
Memory Model is weak. It allows for reordering of memory operations.	Both strict and weak models are provided. The relaxed model is similar to the weak model.
Programmer uses volatile variables and synchronization constructs to establish sequential consistency.	The strict sequential model provides sequential consistency. In the compiler sequential model, the compiler and not the programmer uses memory fences to enforce sequential consistency. In the relaxed model, programmer uses Chapel's *sync* constructs to enforce sequential consistency.
Causal cycles are prevented via program analysis,which detect data races and preventing early execution of operations which lead to such causal loops.	In strict sequential consistency, such causal loops do not occur. These causal loops occur when program order is violated and in strict sequential consistency, program order is not violated. Compiler sequential consistency inserts fences between operations to prevent undesirable effects of such causal loops. Relaxed sequential consistency is prone to such causal loops if the programmer does not use synchronization constructs appropriately.

Table 2.9: Chapel memory model Vs Java memory model

Chapter 3

Proposed Memory Consistency Model in Chapel

As a multiresolution language, the main aim of Chapel is to support a good *"separation of concerns"* [6]. Our memory model for Chapel is based on this idea and aims to provide a migration path from a program that is easy to reason about to a program that has better performance efficiency. Our model allows a programmer to write a parallel program with sequential consistency semantics and then migrate the program to a more performance-oriented version by increasingly changing different parts of the program to follow relaxed semantics. Sequential semantics helps in reasoning about the correctness of the parallel program and is provided by the strict sequential consistency model in our proposed memory model. The advanced performance-oriented versions can be obtained either by using the compiler sequential consistency model, which maintains the sequential semantics, or by the relaxed consistency model, which maintains consistency only at global synchronization points. Our proposed memory model for Chapel thus combines strict sequential consistency model, compiler sequential consistency model and relaxed consistency model.

Our model also solves the issue raised by Padua et al [10], who state that *"It is not known what consistency models best suit the needs of the programmer, the compiler, and the hardware simultaneously"*. We solve this issue by providing a memory model with a solution for each of the three categories: the programmer, the compiler and the hardware. The strict sequential model suits the programmer who wants to reason about the parallel implementation like sequential code. The compiler sequential model relies on the compiler to provide sequential consistency along with performance efficiency. The relaxed consistency model gives the hardware and the compiler complete freedom to perform optimizations (which may violate sequential consistency). The relaxed consistency model is used to obtain maximum performance efficiency and is preferred by advanced programmers.

In this chapter, we first present background material needed to understand our memory models, followed by a discussion of each of the proposed consistency models in the context of the Chapel language.

3.1 Background

In this section we present background material needed to understand our memory models and their implementations.

3.1.1 Sync variables

The Chapel language provides synchronization constructs called *sync* variables. These variables provide full/empty semantics. A *full* state indicates

that the *sync* variable contains a value to be read. When the variable is in a *full* state, an attempt to write a value blocks the thread of execution until the value has been read, which atomically changes the state of the variable to the *empty* state. An *empty* state indicates that the *sync* variable does not contain a value to be read. When the variable is in an *empty* state, an attempt to read blocks the thread of execution until a value has been written, which atomically changes the state of the variable to the *full* state. There are several versions of the write and the read functions available on these *sync* variables. The functions important to us are as follows:

- **writeEF()** (**write** when **E**mpty, leave **F**ull)

  The *writeEF()* operation blocks until the variable is empty, writes a value, and leaves the variable in a full state. The writing of the value and the changing of the state are performed in one atomic step. If multiple threads are executing the function simultaneously, only one thread will successfully complete the function, while the other threads block until the variable is empty.

- **readFE()** (**read** when **F**ull, leave **E**mpty)

  The *readFE()* operation blocks until the variable is full, returns the *value* present in the variable and leaves the variable in the empty state. The reading of the value and the changing of the state are performed in one atomic step. If multiple threads are executing the function simultaneously, only one thread will successfully complete the function, while the other threads block until the variable is full.

3.1.2 Fences

Fences are memory barrier operations available via the C language function __sync_synchronize(). They serve two purposes. One, they ensure the completion of all outstanding memory operations. Two, they restrict compiler/hardware optimizations from moving code across them. Thus they help in enforcing sequential consistency. Usage of *fences* degrades performance because they are costly operations to implement, and because they restrict compiler/hardware optimizations. Hence, they must be used minimally.

3.2 Strict Sequential Consistency Model

We introduce the strict sequential consistency model to satisfy three purposes. One, application developers find it convenient to have a version of the application enforced with sequential semantics against which they can evaluate their advanced more performance-oriented versions. Two, developers use sequential semantics to prevent hardware consistency models (refer to Section 2.1) from allowing inconsistent executions. Three, sequential semantics can be used to debug algorithmic errors in parallel sections of code.

3.2.1 Description

The strict sequential consistency model provides Lamport's definition [8] of sequential consistency:
the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual

processor appear in this sequence in the order specified by its program.

Currently, Chapel does not have constructs that can specify that a block of code be executed with strict sequential consistency. We propose that a new construct *"strict_seq"* be introduced into Chapel for this purpose. The syntax for the *strict_seq* block should be:

strict_seq {

    //code

}

Because the *strict_seq* construct is a block-level construct, it provides flexibility in the use of sequential semantics, on different parallel sections of code and at different times, to debug algorithmic errors.

The rest of the code (i.e. the code not in the *strict_seq* block) will be enforced with compiler sequential consistency, or the relaxed consistency model, as chosen via a compiler flag (refer to 3.3.1).

3.2.2 Analysis of the Strict Memory Model

The rules of the strict sequential model are as follows:

1. Every Chapel statement present in the *strict_seq* block is guaranteed to be executed atomically and in program order. For a detailed description of this rule, please refer to Appendix 1.

2. Nesting of *strict_seq* blocks has the same effect as if the whole block were inside one *strict_seq*.

3. The effects of the *strict_seq* construct are static i.e. the effects are not inherited across function boundaries, providing the programmer with greater control in ensuring that the strictness is not propagated to unintended blocks of code.

While other memory models (refer to Chapter 2) describe their consistency models in terms of low level reads and writes, we have raised the level of abstraction to programming language statements. We believe that this is the right approach, because it is difficult for the programmer to reason about the consistency semantics of a program via low level reads and writes. Our approach provides a more strict form of sequential consistency (hence the name 'strict sequential') that may yield worse performance efficiency than other sequential consistency models. In particular, the strict sequential model does not allow for interleaving of reads and writes of any *statements*. In Listing 3.2 we see that the execution of $a{=}b$ in thread 1, and $c{=}d$ in thread 2 can be interleaved in the case of non-strict sequential consistency models, while the Strict Sequential consistency does not allow such interleaving. Other memory models allow interleaving of reads and writes when there is no violation of sequential consistency.

T1	T2
a=b	c=d

Listing 3.1: Sequential consistency in other models vs Strict sequential (code).

```
Only two Strict Sequential executions are possible:
   Execution 1          Execution 2
        a=b                  c=d
         |                    |
        c=d                  a=b

4 non-strict but sequentially consistent executions
are possible. Two of them are given below:
   Execution 1          Execution 2
        read b               read d
         |                    |
        write a              read b
         |                    |
        read d               write a
         |                    |
        write c              write c
```

Listing 3.2: Sequential consistency in other models vs Strict sequential (analysis).

3.2.3 Implementation of *strict_seq* construct

Since the Chapel compiler does not have a strict_seq construct, this section refers to a possible implementation that the Chapel compiler can follow to implement the construct. We have followed the implementation below for providing strict semantics while evaluating the strict sequential model.

We ensure atomicity of *statements* (as specified in the previous section) via the use of *sync* (refer to Section 3.1.1) variables. We create a unique sync variable named *lck$* for each *strict_seq* block. Each parallel thread executes a *statement* present in the *strict_seq* block in the following order:

lck$.writeEF(1)

statement

fence (refer to Section 3.1.2)

lck$.readFE()

As specified in Section 3.1.1, the *lck$.writeEF(1)* operation is an atomic operation and only one parallel thread will be successful in completing the operation, while the other threads are blocked waiting for the successful thread to perform *lck$.readFE()* operation. Thus, the sync operations provide a locking and unlocking mechanism, allowing for an atomic execution of the *statement*. By virtue of the semantics of the sync variables and each thread doing a *lck$.writeEF(1)* before and a *lck$.readFE()* after the execution of a *statement*, there is no possibility of a deadlock.

3.3 Compiler Sequential Consistency Model

The compiler sequential consistency model leverages the compiler to enforce sequential consistency.

3.3.1 Description

In this model, *fences* 3.1.2 are used to enforce sequential consistency. Identifying *fence* points in the parallel code is difficult because of possible inter-thread and intra-thread data dependencies. If we delegate the responsibility of introducing these *fences* to a beginner/intermediate developer, it might lead to an aggressive or insufficient set of *fences*. An aggressive set of *fences*

degrades performance because *fences* are costly operations and because they restrict compiler/hardware optimizations. On the other hand, an insufficient set might not provide sequential consistency.

A solution to the problem of identifying *fence* points has been proposed by Shasha et al [13], who propose an analysis called *delay set analysis*, that finds the minimal set of *fence* operations (also called as *delays*) to enforce sequential consistency. Padua et al [10] utilize the work of Shasha et al to build a compiler phase that performs delay set analysis (refer to Section 3.3.3). The Compiler Sequential Model follows the work of Padua et al and Shasha et al.

The delay set analysis is very effective when it is performed at the program level and not at the granularity of blocks of code. This is because the amount of information obtained by dependence analysis at lower abstraction would be highly conservative. This would lead to a conflict edge (refer to Section 3.3.3) being established between independent statements, leading to unnecessary fences. Therefore, we expose this model via the command line option *'-compiler_sequential'* in Chapel.

3.3.2 Analysis of the Compiler Sequential Consistency Model

In this section, we illustrate how the compiler sequential consistency model enforces sequential consistency in different scenarios.

- Example 1

```
          Initially A=B=0, register1 = -1.
2 ------------------------------------------------------
           T1                      T2
4 ------------------------------------------------------
          A=1                   if(B==1)
6          .....                  register1 = A
          if(A==1)              ....
8           B=1                 ....
------------------------------------------------------
10 Intuitive output:  register1 = -1 or 1.
   Unintuitive output: register1 = 0.
```

Listing 3.3: Example 1

Consider the code in Listing 3.3: A, B are shared variables between threads T1 and T2. An optimizing compiler might look at T1 and via constant propagation decide that the *if* condition is unnecessary. This might result in *B=1* in line 8 to move above the assignment to A in T1 (this move is possible as there is no data dependency between the assignments). This results in register1 in T2 obtaining the value 0 as shown in Listing 3.4.

```
          T1:B=1
2              |
          T2: if(B==1)
4              |
          T2: register1 = A //register1 gets 0.
6              |
          T1:A=1
```

Listing 3.4: Order of execution of statements in T1 and T2 after constant propagation.

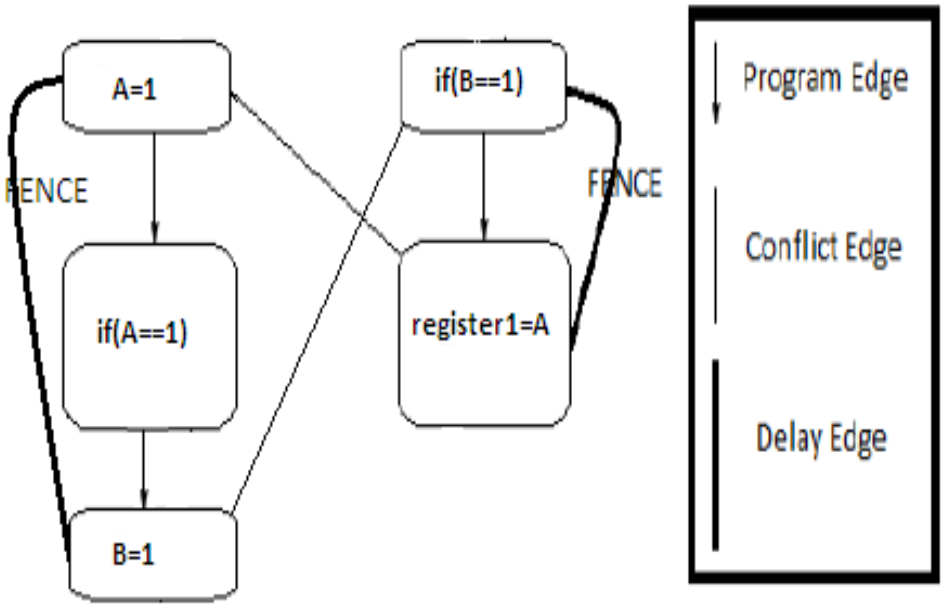

Figure 3.1: Example 1 after Delay Set Analysis

In the compiler sequential consistency model, the compiler performs delay set analysis (which is explained in Section 3.3.3) and determines that *delays* have to be introduced between *A=1* and *B=1* in T1 and between *if(B==1)* and *register1=A* in T2. The *delay* in T1 ensures that *B=1* is executed only after *A=1*. This *delay* in T1 ensures that if T2 enters the *if* body, the value of variable A will be 1 preventing the unintuitive output of register1 obtaining the value 0. The *delays* are shown in Figure 3.1.

- Example 2: Dekker's Algorithm

  In Chapter 1, we learn that the execution of Dekker's algorithm with-

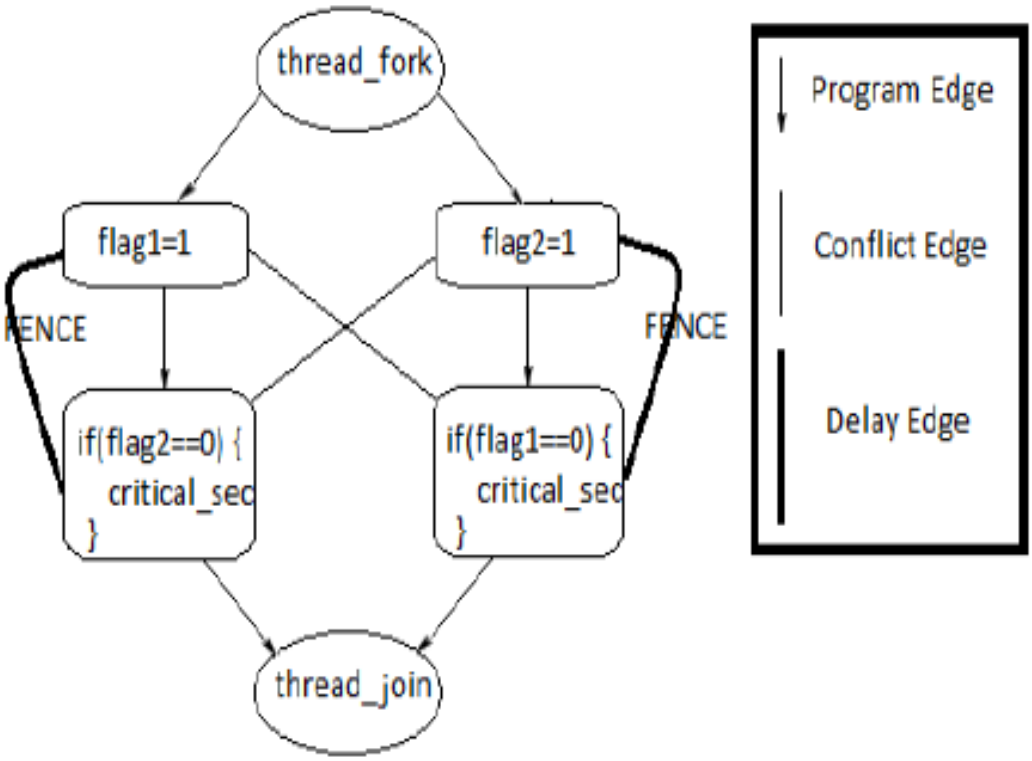

Figure 3.2: Dekker's Algorithm after Delay Set Analysis

out sequential consistency leads to a violation of mutual exclusion of the critical section. Here, we show how the compiler sequential consistency model prevents such a violation. In the compiler sequential consistency model, the compiler performs delay set analysis and determines that *delays* have to be introduced between *flag1=1* and *if(flag2==0)* in thread 1 and between *flag2=1* and *if(flag1==0)* in thread 2. The *fence* in thread

1 prevents the assignment of *flag1* from moving below the *if* condition present in thread 1. Similarly in thread 2, the assignment of *flag2* is prevented from moving below the *if* condition (such executions occur when compiler optimizations such as speculation and branch prediction execute the *if* condition and the body of the *if* construct even before the assignment of the *flag* variables has been completed). Thus, the *fences* ensure mutual exclusion of critical section.

3.3.3 Implementation of Delay Set Analysis

In this section, we outline the implementation of the delay set analysis and the introduction of *delays*. For complete details, we refer the readers to Shasha et al [13] and Padua et al [10].

1. The compiler initially creates a graph where the nodes are the statements present in the parallel code. The edges in the graph are of two types, program edges and conflict edges. Program edges are directed and represent the program order in which the two statements linked by the program edge have to be executed. Conflict edges are between statements in different threads, where at least one of the statements writes to a location accessed by the other statement and where the statements can happen in parallel. For example, in Figure 3.1 we see a program edge between *A=1* and *if(A==1)* in T1. This program edge signifies that the programmer wants the assignment to A to precede the check of A.

2. To identify the conflict edges, the compiler performs alias analysis, escape analysis [12] and MHP (May Happen in Parallel) analysis [11].

   Alias analysis is used to determine which variables in the parallel code refer to the same location. This helps in establishing a conflict edge between statements where one statement is modifying a location that the other statement is accessing via an alias. Escape analysis is used to identify if a location or its alias is accessed by more than one thread. MHP analysis helps in determining the sets of statements that can execute in parallel.

   For example, in Figure 3.1 we see a conflict edge between *B=1* in T1 and *if(B==1)* in T2. This is because both the statements are accessing the same memory location B.

3. Once the graph is created, the compiler identifies the minimal mixed cycles present in it. Mixed cycles are cycles formed by the combination of program edges and conflict edges. Minimality of these mixed cycles means that any other mixed cycle in graph is formed from one or more of these minimal mixed cycles.

   The mixed cycle in Figure 3.1 is shown in Figure 3.3.

4. The set of program edges present in these mixed cycles form the delay set. The compiler then introduces a *delay* between each of the statements linked by these program edges. We can see this in Figure 3.1 and Figure 3.2.

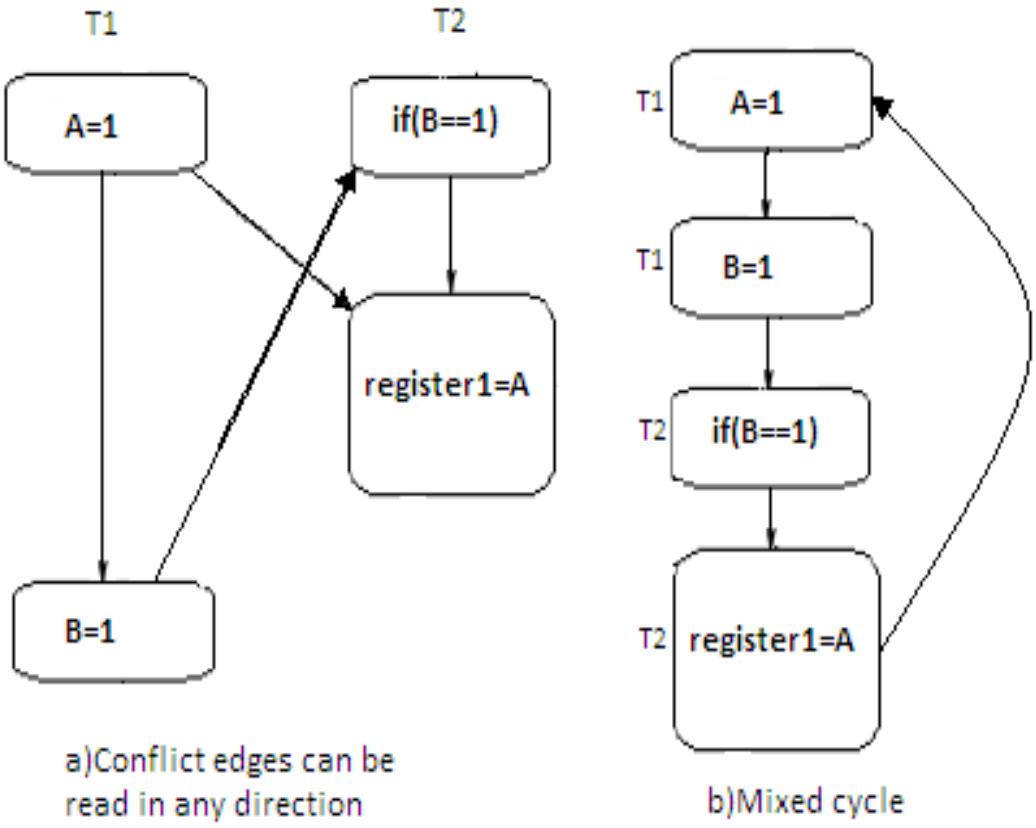

Figure 3.3: Example 1 with a mixed cycle

3.4 Relaxed Consistency Model

3.4.1 Description

In the relaxed consistency model, the responsibility of enforcing sequential consistency semantics is left to the programmer. The programmer can enforce sequential semantics via the usage of synchronization constructs such as *sync* variables present in the Chapel language. These *sync* variables provide full empty/semantics and can be used in various scenarios such as

to implement mutual exclusion of critical sections (as we have used in Section 3.2.3). The Chapel compiler guarantees that there will be no movement of memory operations across the use of the *sync* variables and that all the functions exposed on these variables will be executed atomically.

3.4.2 Analysis of the Relaxed Memory Model

Without the enforcement of sequential analysis, the compiler and the hardware are free to apply optimizations leading to performance efficiency than other consistency models. If the programmer does not do a good job in enforcing the required synchronization constraints, then the cost paid is unintuitive outputs due to inconsistent executions (refer to the example of Dekker's algorithm in Section 3.3.2).

3.4.3 Implementation of Relaxed Memory Model

The Chapel compiler currently provides the relaxed consistency semantics by default.

3.5 Interaction of memory models

Currently code snippets as in Listing 3.5 are not allowed. In Listing 3.5, we see that one of the processes is following sequential semantics, while the other processes are following relaxed semantics. Such interactions among the memory models are not allowed. We plan to define the semantics of such interactions in the future.

```
coforall loc in Locales {
        if(here.id == 0) {
                strict_seq{
                ... //strict semantics
                }
        } else {
                ...//relaxed semantics
        }
}
```

Listing 3.5: Interaction of different memory models are currently disallowed.

3.6 Summary

In this chapter, we have discussed the details of our memory model for Chapel, which combines strict sequential consistency model, compiler sequential consistency model and relaxed consistency model. In the table 3.1, we present an overview of the proposed memory models.

	Strict Sequential	Compiler Sequential	Relaxed Consistency
Theme	Code is enforced with Lamport's definition of sequential consistency.	Compiler enforces sequential consistency semantics by introducing memory barriers in the program.	Consistency only at global synchronization points (as defined by the programmer).
Purpose	Comprehensibility of parallel program.	Performance-oriented version of the parallel program with sequential semantics.	Performance oriented versions of the parallel program (may sacrifice consistency for performance).
Specification	strict_seq{} construct.	command line flag "-compiler_sequential".	default.
Implementation	usage of *sync* variables (not exposed to the programmer).	memory barriers are introduced via 'delay set analysis' [13], [10].	programmer identifies and defines global synchronization points via 'sync' variables.
Changes to Chapel	expose a new construct 'strict_seq' in the language.	changes in the compiler to do delay set analysis.	required synchronization constructs are present in the language.

Table 3.1: Proposed memory models at a glance.

Chapter 4

Performance Analysis of Memory Model

In this chapter, we analyze the memory models from the perspective of performance by implementing the following applications:

- Barnes-Hut algorithm for solving n-body problem
- Radix-4 FFT (Fast Fourier Transform)
- Random Access (one of the High Performance Computing benchmarks)

We choose these programs because they represent a spectrum of applications, the spectrum being based on the degree to which the data dependencies in the program are statically determinate. Random Access represents one end of the spectrum with the least degree of determinacy while, Barnes-Hut represents the other end of the spectrum with a high degree of determinacy. We believe that this degree is inversely proportional to the likelihood of memory inconsistencies. Hence, we believe that this spectrum is appropriate for evaluating our memory models.

Our performance analysis compares the performance of three versions of each application, where each version uses a different memory model (Strict

Sequential/Compiler Sequential/Relaxed Consistency). We also examine the performance effect of the proposed models on programs written in a hybrid programming model that uses MPI and Pthreads, which shows that the performance gap between relaxed consistency and strict sequential consistency is not specific to the Chapel language.

In the following sections, we discuss the performance of the models in the context of the above applications.

4.1 Barnes-Hut

The n-body problem models the interaction of a set of bodies on each other. In this application, we model the effects of the gravitational force on the n-body system. Each body has a weight and initial position. Every body exerts a gravitational force on every other body in the system which causes each body to move. A simple approach to solving this n-body problem is to compute the force between every two bodies. This approach takes $O(n * n)$ time. The Barnes-Hut algorithm reduces this complexity to $O(n * log(n))$ time by approximating the force exerted by a group of sufficiently far bodies to be the force exerted by a single virtual body acting at the center of mass of the group of bodies [5], [14], [17].

4.1.1 Implementation

We have followed the implementation provided by Warren et al [17]. The algorithm consists of two phases. In the first phase a quad tree is constructed, the quad tree represents the spatial distribution of the n-bodies. Each internal node in the tree represents the virtual body at the center of mass of the bodies in the subtree rooted at the internal node. The second phase is the force calculation phase. The force experienced by each body is calculated by using MAC (multipole acceptance criteria) [17]. The velocity and new position of the body is then calculated using a Leapfrog-Verlet integrator.

4.1.2 Strict Sequential Consistency

We develop a strict sequentially consistent version of the Barnes-Hut application by enclosing the code, in Listing 4.1, within the *strict_seq* block. (A section of the code which is heavily used and having intricate data interactions is selected to be the *strict_seq* block, because sequential execution of such sections of code will help in reasoning about the behavior of the program. As the *strict_seq* construct has not yet been implemented in Chapel, enclosing the code within the *strict_seq* block translates to marking the block with comments). The code in Listing 4.1 represents the core of the Barnes-Hut algorithm. It involves constructing the quad tree and computing the displacement of the bodies.

```
//-------------strict_seq block begins-------------
 coforall loc in Locales do on loc {
//construction of quad tree
constructTree();
//each process is responsible for computing
// force and displacement experienced
// by a subset of bodies
beginIndex = (here.id * (n/numLocales)) + 1;
endIndex = ((here.id+1) * (n/numLocales));
forall i in beginIndex..endIndex do {
        //compute the total force experienced by body i.
        compute_interaction(root(loc.id), i);
        //move the body.
        move_body(i);
}
}
//-----------------strict_seq block ends-------------
```

Listing 4.1: The *strict_seq* block in Barnes-Hut.

4.1.3 Compiler Sequential Consistency

In this section, we develop a compiler sequential consistency version of the Barnes-Hut application. As discussed in Section 3.3, we need to perform delay set analysis and introduce fences at places as determined by the analysis (we do manual delay set analysis as the Chapel compiler does not yet have a phase which does the delay set analysis). To understand the results of delay set analysis, we convert the code in Listing 4.1 to a simpler form as shown in the below steps.

1. Listing 4.1 to Listing 4.2

   Each process reads the positions of all the bodies to construct the tree.

   It then computes the force and displacement of a subset of bodies.

```
coforall loc in Locales do on loc {
constructTree();
//in construct each process reads the
//positions of all the nbodies.
//Let us consider two such bodies x and y.

forall i in beginIndex..endIndex do {
        compute_interaction(root(loc.id), i);
        move_body(i);
        }
}
```

Listing 4.2: Delay set analysis of Barnes-Hut Algorithm, Step:1.

2. Listing 4.2 to Listing 4.3

   In the Listing 4.3, we replace the *coforall* loop with two processes, T1 and T2. We also replace the force computation step and moving the n-body step with writes to the body's position.

```
                    T1              T2
constructTree     read y          read x
                  read x          read y
move_body         write x         write y
*conflict: read x in T2 and write x in T1
*conflict: read y in T1 and write y in T2
```

Listing 4.3: Delay set analysis of Barnes-Hut Algorithm, Step:2.

3. Barnes-Hut Implementation with fences.

   Applying delay set analysis (refer to Section 3.3.3), fences are introduced as shown in Listing 4.4.

```
                        T1              T2
constructTree       read y          read x
                    read x          read y
                  ————FENCE————————
move_body           write x         write y
                  ————FENCE————————
```

Listing 4.4: Delay set analysis of Barnes-Hut Algorithm, Step:3.

4.1.4 Analysis of Performance

Graph in Figure 4.1 illustrates the following points.

- The compiler sequential model is as efficient as relaxed consistency, because the *forall* loop in Listing 4.3 is unaffected by the delay set analysis. The delay set analysis introduced *fences* before and after this *forall* loop. Each iteration of the *forall* is independent, and the parallelism present here is exploited to the same amount by both the compiler sequential and the relaxed consistency models. We consider this a win for the compiler sequential model because along with giving comparable performance to the relaxed model, the compiler sequential model gives the guarantee that none of the processes reads a stale position value of the bodies in a new timestep.

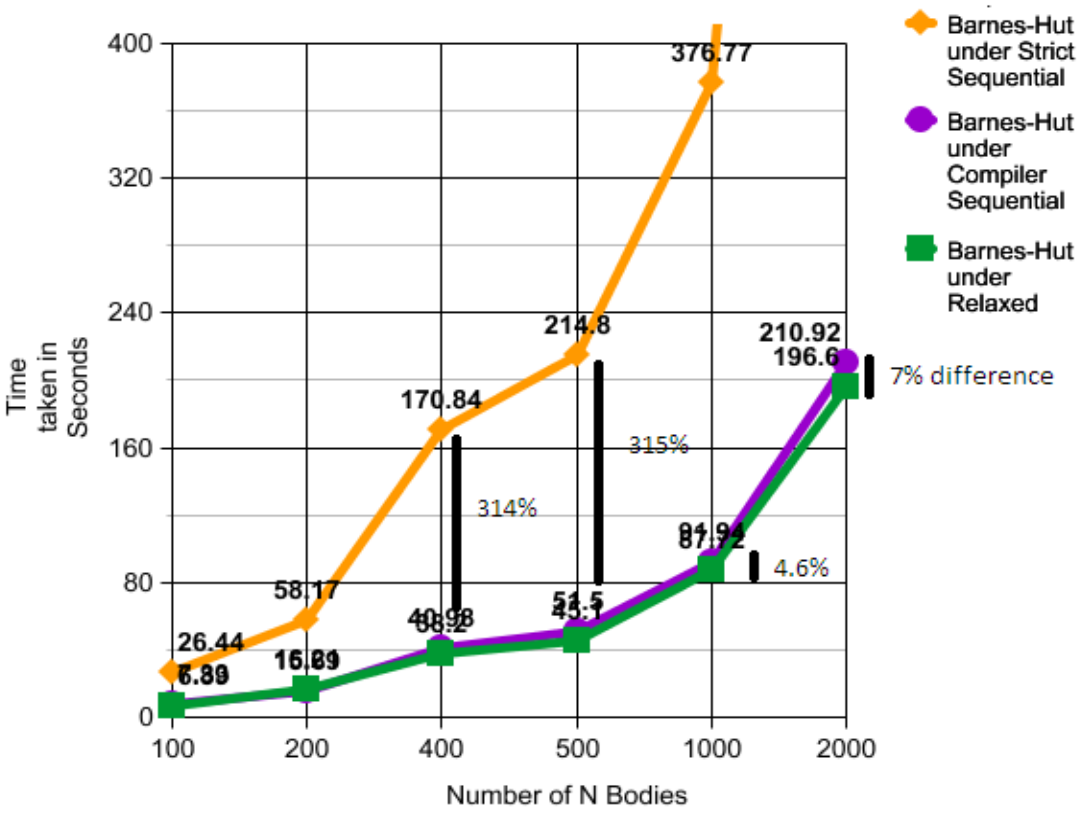

Figure 4.1: Application: Barnes-Hut. Performance of the Chapel Implementations.

- As expected, the strict sequential version shows poor performance because of the presence of locks and memory barriers after each *statement* in the *strict_seq* block. However, it does give the guarantee that none of the processes reads a stale position value of the bodies in a new timestep.

- The hybrid programming model also follows the same trend (refer to Figure 4.2).

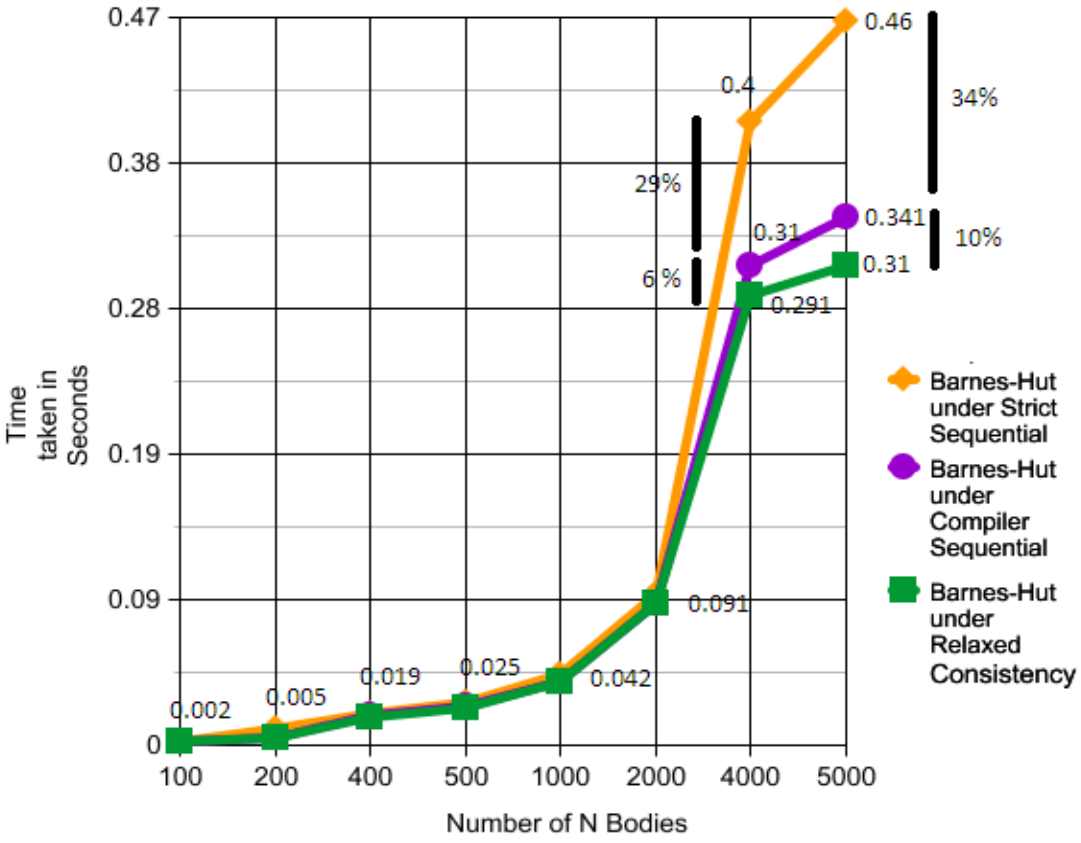

Figure 4.2: Application: Barnes-Hut. Performance of the MPI Implementations.

4.2 FFT

The DFT (discrete fourier transform) of a set of n complex values(x_n) is a set of values X_n such that (source: [1]):

$X_k = \sum_{n=0}^{N-1} x(n) W_N^{kn}, 0 \leq k \leq N-1$

$W_N = e^{-j*2*\pi/N}$

One of the algorithms to solve the DFT problem is FFT. FFT is fast because it

exploits symmetry and periodicity in the computation of the X values. A radix-4 fft problem translates the computation of each X_n value into the following formula (source: [1]):

$X_k = \sum_{n=0}^{N/4-1} x(n)W_N^{kn} + W_N^{Nk/4} \sum_{n=0}^{N/4-1} x(n + (N/4))W_N^{kn}$
$W_N^{Nk/2} \sum_{n=0}^{N/4-1} x(n + (N/2))W_N^{kn} + W_N^{3Nk/4} \sum_{n=0}^{N/4-1} x(n + (3N/4))W_N^{kn}$

4.2.1 Implementation

```
def butterfly(wk1, wk2, wk3, X:[0..3]) {
//X[0..3] is a slice of the complex
//whose DFT values have to be
//computed.
//wk1,wk2,wk3 are multipliers of
//complex type.
  var x0 = X(0) + X(1),
      x1 = X(0) - X(1),
      x2 = X(2) + X(3),
      x3rot = (X(2) - X(3))*1.0i;

  X(0) = x0 + x2;
  // compute the butterfly in-place on X
  x0 -= x2;
  X(2) = wk2 * x0;
  x0 = x1 + x3rot;
  X(1) = wk1 * x0;
  x0 = x1 - x3rot;
  X(3) = wk3 * x0;
}
```

Listing 4.5: Butterfly function in Chapel implementation of FFT.

We have used the FFT implementation provided with the Chapel installation. The principle that an 8-point DFT can be computed by solving two 4-point

DFTs and a 4-point DFT can be solved by computing two 2-point DFTs has been used in the FFT implementation. Computing the 2-point DFT's involves the application of the radix-4 butterfly operation. In the Listing 4.5, we show the radix-4 *butterfly* function in Chapel. The implementation of FFT consists of two phases. In the first phase, the x_n values are block distributed among the processors and then the *butterfly* operation is applied on the input complex values. In the second phase, the x_n values are block distributed among the processors and then, the *butterfly* operation is applied on the complex values output from the first phase. For details regarding the implementation, we direct the user to the FFT example present in the Chapel installation.

4.2.2 Strict Sequential Consistency

We develop the strict sequential consistent version of the FFT implementation by marking the piece of code in Listing 4.7 as a *strict_seq* block. The code in Listing 4.7 depicts the application of the butterfly operation on the input complex values.

```
//iterate over each of the banks of butterflies
//-------------strict_seq ---------------------
forall (bankStart, twidIndex) in (ADom by 2*span, 0..) {
      var wk2 = W(twidIndex),
      wk1 = W(2*twidIndex),
      wk3 = (wk1.re - 2 * wk2.im * wk1.im,
            2 * wk2.im * wk1.re - wk1.im):complexType;
```

Listing 4.6: Part of the DFFT function in Chapel implementation of FFT.

```
//iterating over the lower bank of butterflies.
      forall lo in bankStart..#str do
        on ADom.dist.ind2loc(lo) do
          local butterfly(wk1, wk2, wk3,
                          A.localSlice(lo..by str #radix));

      wk1 = W(2*twidIndex+1);
      wk3 = (wk1.re - 2 * wk2.re * wk1.im,
             2 * wk2.re * wk1.re - wk1.im):complexType;
      wk2 *= 1.0i;

//iterating over the high bank of butterflies.
      forall lo in bankStart+span..#str do
        on ADom.dist.ind2loc(lo) do
          local butterfly(wk1, wk2, wk3,
           A.localSlice(lo.. by str #radix));
    }
//-------------strict_seq ---------------------
```

Listing 4.7: Part of the DFFT function in Chapel implementation of FFT.

4.2.3 Compiler Sequential Consistency

We have developed two versions of the FFT application providing compiler sequential consistency, but differing in the amount of semantic information used while performing the delay set analysis. The semantics of the *forall* loop states that each iteration is independent of other iterations, which makes the iterations of the *forall* loop in line 10 (and similarly line 21) in Listing 4.7 independent of one another. If this information is used in the delay set analysis, then no *fence* is introduced into the *forall* loops; otherwise delay set analysis introduces a fence. Both versions are presented below:

```
forall (bankStart, twidIndex) in (ADom by 2*span, 0..) {
      var wk2 = W(twidIndex),
      wk1 = W(2*twidIndex),
      wk3 = (wk1.re - 2 * wk2.im * wk1.im,
            2 * wk2.im * wk1.re - wk1.im):complexType;

//iterating over the lower bank of butterflies.
      forall lo in bankStart..#str do
        on ADom.dist.ind2loc(lo) do
          local butterfly(wk1, wk2, wk3,
                          A.localSlice(lo..by str #radix));

      wk1 = W(2*twidIndex+1);
      wk3 = (wk1.re - 2 * wk2.re * wk1.im,
             2 * wk2.re * wk1.re - wk1.im):complexType;
      wk2 *= 1.0i;

//iterating over the high bank of butterflies.
      forall lo in bankStart+span..#str do
        on ADom.dist.ind2loc(lo) do
          local butterfly(wk1, wk2, wk3,
           A.localSlice(lo.. by str #radix));

----------FENCE----------
    }
```

Listing 4.8: Compiler Sequential version of FFT considering the semantics of *forall*.

```
forall (bankStart, twidIndex) in (ADom by 2*span, 0..) {
      var wk2 = W(twidIndex),
      wk1 = W(2*twidIndex),
```

Listing 4.9: Compiler Sequential version of FFT NOT considering the semantics of *forall*.

```
            wk3 = (wk1.re - 2 * wk2.im * wk1.im,
          2 * wk2.im * wk1.re - wk1.im):complexType;
//iterating over the lower bank of butterflies.
      forall lo in bankStart..#str do
        on ADom.dist.ind2loc(lo) do {
                butterfly(wk1, wk2, wk3,
                      A.localSlice(lo..by str #radix));
        ----------FENCE----------
        }

      wk1 = W(2*twidIndex+1);
      wk3 = (wk1.re - 2 * wk2.re * wk1.im,
             2 * wk2.re * wk1.re - wk1.im):complexType;
      wk2 *= 1.0i;

//iterating over the high bank of butterflies.
      forall lo in bankStart+span..#str do
        on ADom.dist.ind2loc(lo) do {
            butterfly(wk1, wk2, wk3,
         A.localSlice(lo.. by str #radix));
        ----------FENCE----------
        }
    }
```

Listing 4.10: Compiler Sequential version of FFT NOT considering the semantics of *forall*.

4.2.4 Analysis of Performance

Graph in Figure 4.3 illustrates the following points:

- The compiler sequential version that uses more semantic information performs on par with the relaxed version, illustrating that a rigorous

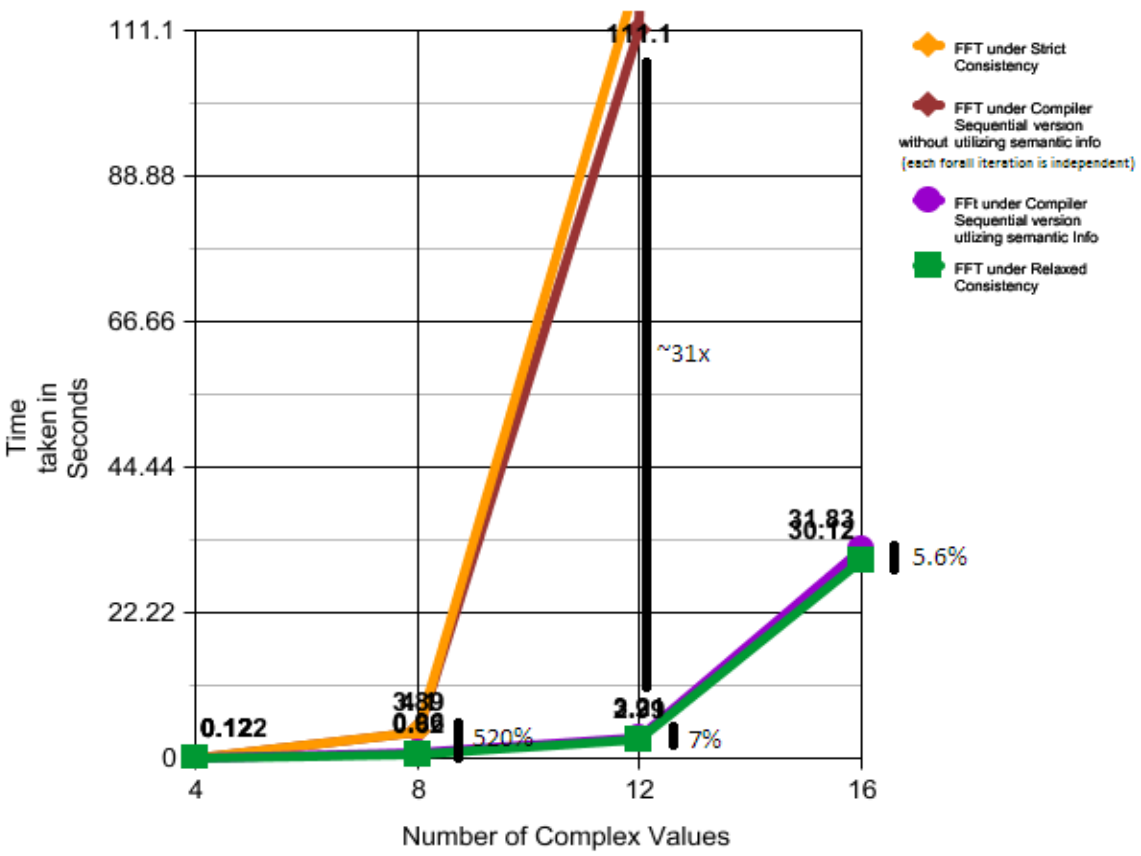

Figure 4.3: Application: FFT. Performance of the Chapel Implementations.

compiler analysis (using available semantic information) can give good performance.

- The compiler sequential version that uses less semantic information performs on par with the sequential version, showing the disadvantage of a conservative compiler analysis.

- We do not have two versions of the hybrid program because there are no

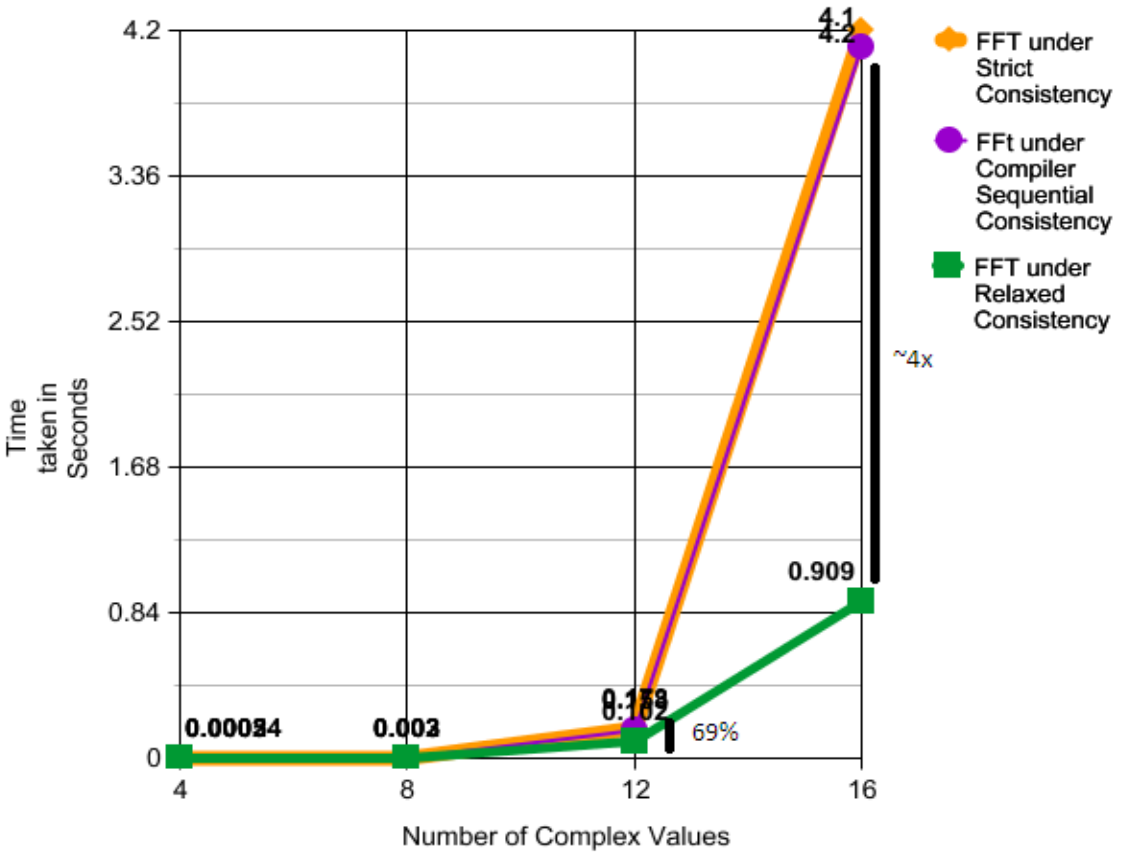

Figure 4.4: Application: FFT. Performance of the MPI Implementations.

special constructs which provide useful semantic information. However, we see the same performance trend in the hybrid model (refer to Figure 4.4) for the different memory models.

4.3 Random Access

Random Access is one of the applications present in the high performance computing benchmark. It involves updating data values in a table distributed among the processing nodes present in the parallel system.

4.3.1 Implementation

Note: We utilize the RA implementation present in the Chapel Examples directory. The implementation of RA is straightforward. A request for an update is sent to the process which owns the data value. The process then updates the data value. An interesting scenario is when multiple updates to the same memory location are being done. As the Chapel example does not use *locks* (via *sync* variables), we face possible data corruptions when such updates occur. Below we present how our memory models handle such situations.

4.3.2 Strict Sequential

```
//-------------strict_seq-----------------
//RAStream generates the stream of updates.
 forall (_, r) in (Updates, RAStream()) do
   //identify the location where data values is present.
    on TableDist.ind2loc(r & indexMask) do {
      const myR = r;
      local {//update value in Table T
        T(myR & indexMask) ^= myR;
      }
   }
//-------------strict_seq-----------------
```

Listing 4.11: *strict_seq* block in RA.

We develop the strict sequential consistent version of the RA implementation by marking the piece of code in Listing 4.11 as *strict_seq*. The code in Listing 4.11 handles the updates to the distributed table.

4.3.3 Compiler Sequential

We apply the delay set analysis to create a compiler sequential version. The delay set analysis is not able to detect which updates are independent and hence enforces atomicity of every update.

```
//RAStream generates the stream of updates
//to be performed.
 forall (_, r) in (Updates, RAStream()) do
   //identify the location where data values is present.
    on TableDist.ind2loc(r & indexMask) do {
      const myR = r;
      local {
       //update value in Table T
        T(myR & indexMask) ^= myR;
        ------FENCE----------
      }
   }
```

Listing 4.12: Fences in RA.

4.3.4 Performance Analysis

We note the following points:

- Compiler and Strict sequential models lose out on performance, because they effectively serialize the updates.

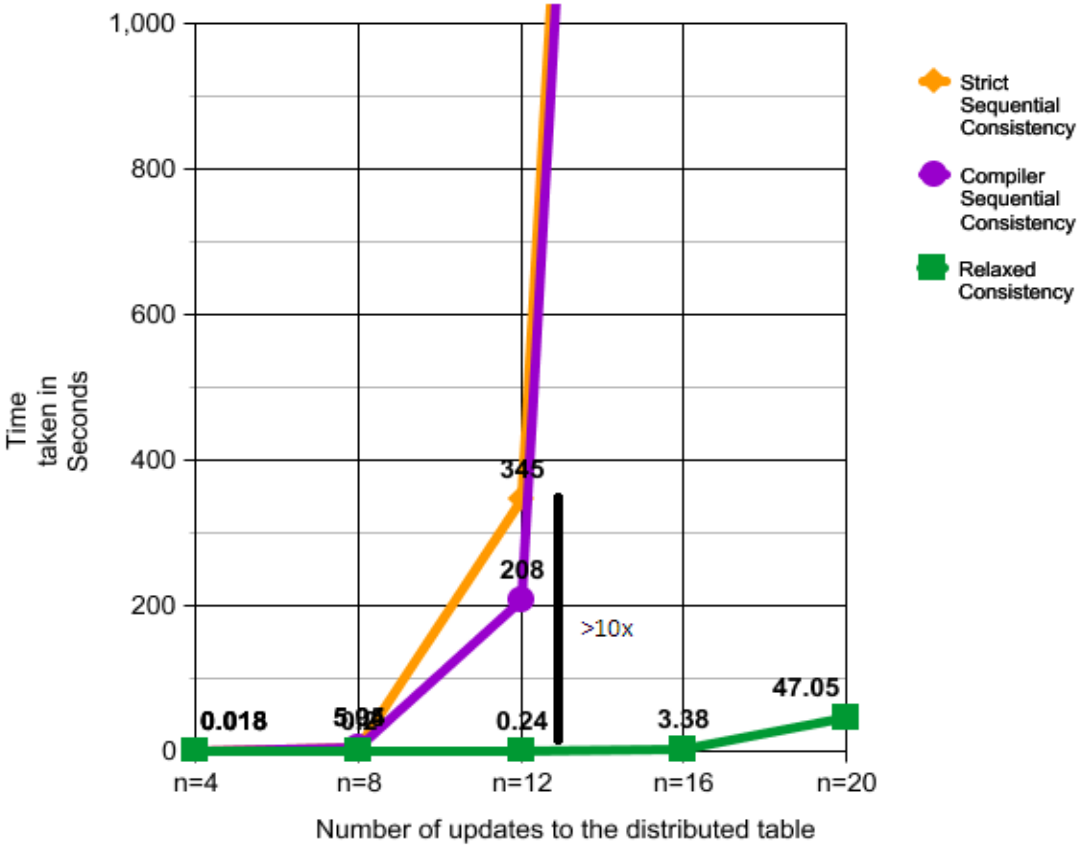

Figure 4.5: Application: RA. Performance of the Chapel Implementations.

- Relaxed sequential gets best performance but cannot provide the guarantee that the table is not corrupted. The data corruption occurs because the model allows multiple updates to the same location to happen without atomicity. This is illustrated in Figure 4.6 (errors here refer to data corruptions).

- As noted from Figure 4.7, the hybrid model follows the same performance

trend as the Chapel implementations of RA.

4.4 Summary

Figure 4.8 summarizes the results from our experiments. Relaxed consistency provides the best performance efficiency while strict sequential consistency provides the least. The performance of the compiler sequential model depends on the degree to which the data dependences are statically determinate. In the Barnes-Hut application, the calculation of force and subsequent movement of n-bodies are data parallel, which can be determined statically leading to compiler sequential version performing on par with the relaxed consistency version. For applications such as FFT, the performance obtained from the compiler sequential version is dependent on the degree of semantic information used in the delay set analysis. In applications such as RA, it is not possible to determine statically whether the updates are independent or not. Hence the compiler sequential version is unable to obtain performance on par with the relaxed consistency version.

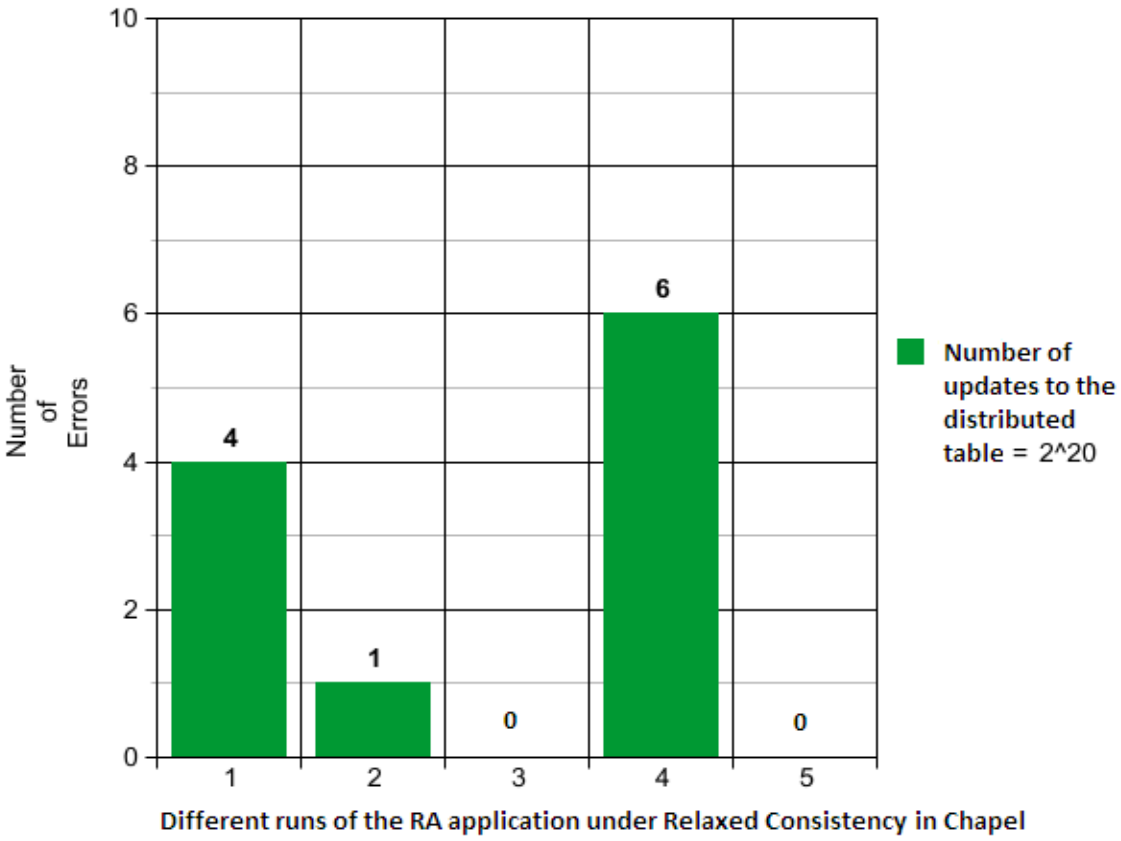

Figure 4.6: Application: RA. Number of errors noted under relaxed consistency.

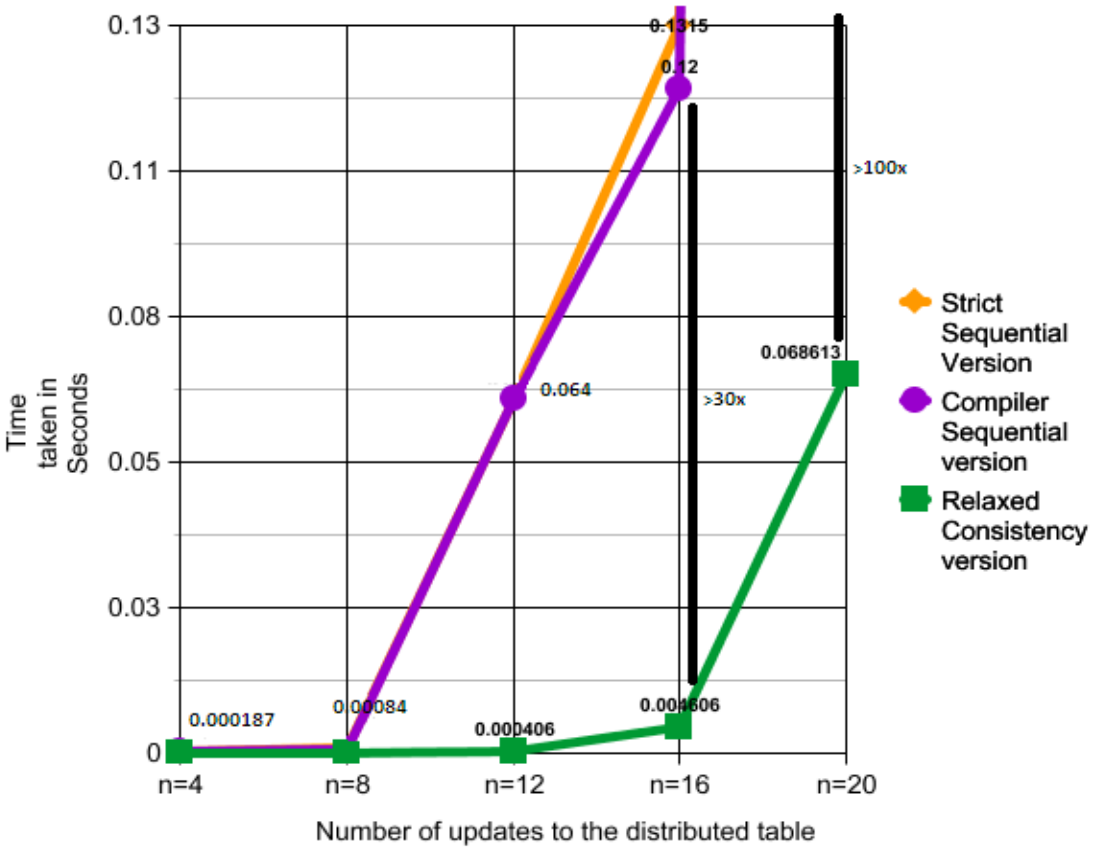

Figure 4.7: Application: RA. Performance of the MPI Implementations.

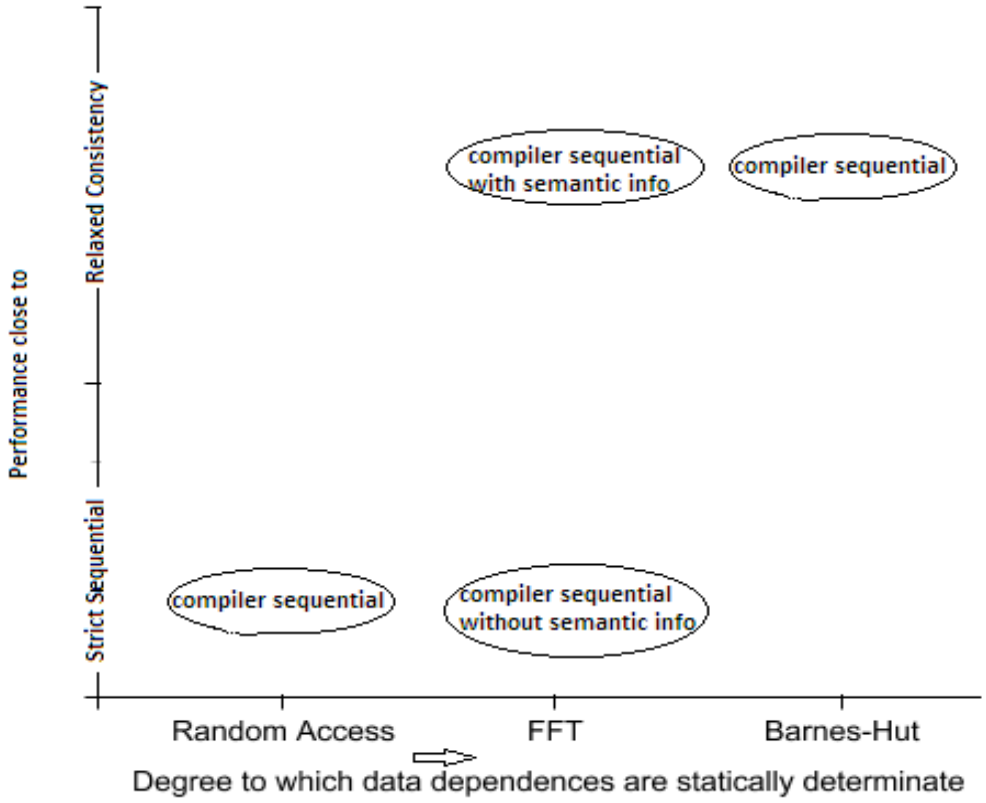

Figure 4.8: Spectrum of Applications based on degree to which data dependences are statically determinate.

Chapter 5

Conclusion

Chapel is a multiresolution language, the main aim of Chapel is to support a good *"separation of concerns"* [6]. We have proposed a memory model for Chapel which is based on this idea. Our memory model provides a migration path from a program that is easy to reason about to a program that has better performance efficiency. Our memory consistency model combines the following three types of models:

- Strict sequential consistency model for the cautious Programmer.
- Compiler sequential Consistency model for the trusting Programmer.
- Relaxed consistency model for the advanced Programmer.

To evaluate the memory models, we have implemented three applications: Barnes-Hut, FFT, Random-Access in Chapel, and the hybrid model of MPI and Pthread. Our experiments have shown that:

- Strict Sequential is the best model to determine algorithmic errors in the applications, though its performance is the worst among the three models.

- Relaxed consistency gives the best performance among the three models. However, it does not provide a guarantee that inconsistent executions will be avoided.

- The performance of the compiler sequential model depends on accuracy of the data analysis (consisting of reference analysis, synchronization analysis, MHP analysis) performed by the compiler. A conservative analysis leads to the introduction of more number of *fence* operations, bringing down the performance of the model to a strict sequential consistent version. An accurate analysis leads to lesser number of *fence* operations. This leads to performance equivalent to a relaxed consistency model. In both cases, the compiler sequential version does not compromise on the sequential consistency guarantee.

- The relative performance of the consistency models in Chapel, and in MPI and Pthread are the same, showing that irrespective of the programming model, the developer pays the same performance overhead when moving among the different consistency models.

From our results, we believe that we have developed a generic memory model based on the idea of multiresolution that can be applied to any parallel programming model.

Chapter 6

Future Work

We plan to extend our work in several directions.

- We want to build a compiler phase into the Chapel Compiler which performs delay set analysis (currently we are manually doing delay set analysis). As part of this work, we want to explore, how the compiler can utilize the semantic information in a program to provide good performance irrespective of the application.
- We want to provide feedback to the developer regarding the placement of the fences, so that the developer can reduce the number of fences by reducing the number of shared memory interactions.
- We want to allow interactions of different memory models i.e. the same code block is executed with strict semantics by one thread of execution while another thread executes it with relaxed semantics.

Appendix

Appendix 1

Chapel Statements in Strict_Seq Block

In this appendix, we will be presenting the rules enforced on the *statements* [1] present in a block of code declared with the *strict_seq* construct.

1. A *statement* is guaranteed to be executed atomically and in program order.

2. Item 1 applies to the following statements:
   expression-statement
   assignment-statement
   swap-statement
   return-statement
   yield-statement

3. The *statements* below are parallel constructs which introduce parallelism into the Chapel program.
   forall-statement
   cobegin-statement

[1] (*statement* refers to the *statement* as defined in Chapel grammar. Please refer to Chapel Specification 0.795 for more details)

coforall-statement

begin-statement

These parallel constructs create multiple parallel threads of execution. Each *statement* present in the body will be executed in program order. They will be executed atomically with respect to *statements* in all the parallel threads of execution.

4. The *statements* below are sequential constructs which do not introduce parallelism. Each *statement* present in the body of the below constructs will be executed atomically and in program order.

   conditional-statement

   select-statement

   while-do-statement

   do-while-statement

   for-statement

   param-for-statement

5. The below *statements* do not contain any executable memory operations. Hence, strictness does not apply for them.

   empty-statement

   all-declaration-statement

   label-statement

   break-statement

   continue-statement

Index

Abstract, vi
Acknowledgments, v
Appendix
    Chapel Statements in Strict_Seq Block, 67
Appendix, 66

Bibliography, 71

Conclusion, 63

Dedication, iv

Future Work, 65

Introduction, 1

Performance Analysis of Memory Model, 41

Related Work, 8

Bibliography

[1] Fast fourier transform. http://www.cmlab.csie.ntu.edu.tw/cml/dsp/training/coding/transform/fft.html.

[2] Formal upc memory consistency semantics, upc specification 1.2. http://upc.lbl.gov/docs/user/upc_spec_1.2.pdf.

[3] Ia64 volume 2: System architecture, revision 2.3. http://www.intel.com/Assets/PDF/manual/245318.pdf.

[4] Memory consistency (itanium64). http://www.gelato.unsw.edu.au/IA64wiki/MemoryConsistency.

[5] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, and Katherine A. Yelick. The landscape of parallel computing research: A view from berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006.

[6] Bradford L. Chamberlain. Multiresolution languages for portable yet efficient parallel programming. October 2007.

[7] Donald K. Knuth. *The TEXbook*. Addison-Wesley, 1984.

[8] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, (28), 1979.

[9] Jeremy Manson, William Pugh, and Sarita V. Adve. The java memory model. In *POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 378–391, New York, NY, USA, 2005. ACM.

[10] Zehra Sura Chi-Leung Wong Xing Fang Jaejin Lee Samuel P. Midkiff and David Padua. Automatic implementation of programming language consistency models. 2481/2005:172–187, 2005.

[11] Gleb Naumovich, George S. Avrunin, and Lori A. Clarke. An efficient algorithm for computing mhp information for concurrent java programs. pages 338–354, 1999.

[12] Alexandru Salcianu and Martin Rinard. Pointer and escape analysis for multithreaded programs. pages 12–23, 2001.

[13] Dennis Shasha and Marc Snir. Efficient and correct execution of parallel programs that share memory. *ACM Trans. Program. Lang. Syst.*, 10(2):282–312, 1988.

[14] J. Singh, C. Colt, T. Totsuka, A. Gupta, and J. Hennessy. Load balancing and data locality in hierarchical n body methods. *Journal of Parallel and Distributed Computing*, 1994.

[15] Herb Sutter. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's Journal*, March 2005.

[16] The Chapel team. Chapel language specification 0.795, April 2010.

[17] M. S. Warren and J. K. Salmon. Astrophysical n-body simulations using hierarchical tree data structures. In *Supercomputing '92: Proceedings of the 1992 ACM/IEEE conference on Supercomputing*, pages 570–576, Los Alamitos, CA, USA, 1992. IEEE Computer Society Press.

Vita

Karthik Srinivasa Murthy was born in Bangalore, India on October 8th, 1983. He was welcomed to this world by his wonderful family consisting of Sharada, his mother; Srinivasa, his dad; Padma, his aunt; Hyma and Indu, his sisters. He completed his high school with distinction in 1999. He achieved a top 100 rank in the entrance exams and joined R. V. C. E for his undergraduate degree in Computer Science. He obtained his bachelor's degree in computer science in 2005 and worked for two years at National Instruments. He left NI to pursue his masters at The University of Texas at Austin. He completed 3 amazing years at UT Austin. On august 15th 2010, he is going to embark on a new adventure as a doctoral student at Rice University, Houston. He loves to dream, and his favorite dream is to see himself become a professor.

Permanent address: 334, 12th B Cross, WCR, Mahalakshmipuram,
Bangalore, India

This thesis was typeset with LaTeX[†] by the author.

[†]LaTeX is a document preparation system developed by Leslie Lamport as a special version of Donald Knuth's TeX Program.