UNF UNIVERSITY of NORTH FLORIDA

UNF Digital Commons

UNF Graduate Theses and Dissertations

Student Scholarship

2012

Evaluating Speedup in Parallel Compilers

Deepa V. Komathukattil
University of North Florida

Suggested Citation

Komathukattil, Deepa V., "Evaluating Speedup in Parallel Compilers" (2012). *UNF Graduate Theses and Dissertations*. 417.
https://digitalcommons.unf.edu/etd/417

This Master's Thesis is brought to you for free and open access by the Student Scholarship at UNF Digital Commons. It has been accepted for inclusion in UNF Graduate Theses and Dissertations by an authorized administrator of UNF Digital Commons. For more information, please contact Digital Projects.
© 2012 All Rights Reserved

UNF UNIVERSITY of NORTH FLORIDA

EVALUATING SPEEDUP IN PARALLEL COMPILERS

by

Deepa Komathukattil

A thesis submitted to the
School of Computing
in partial fulfillment of the requirements for the degree of

Master of Science in Computer and Information Sciences

UNIVERSITY OF NORTH FLORIDA
SCHOOL OF COMPUTING

December 2012

Copyright (@) 2012 by Deepa Komathukattil

All rights reserved. Reproduction in whole or in part in any form requires the prior written permission of Deepa Komathukattil or designated representative.

The thesis "Evaluating Speedup in Parallel Compilers" submitted by Deepa Viswanathan Komathukattil in partial fulfillment of the requirements for the degree of Master of Science in Computer and Information Sciences has been

Approved by the thesis committee: Date

______________________________ ________________
Roger Eggen, Ph.D.
Thesis Advisor and Committee Chairperson

______________________________ ________________
Sanjay P. Ahuja, Ph.D.

______________________________ ________________
Behrooz Seyed Abbassi, Ph.D.

Accepted for the School of Computing:

______________________________ ________________
Asai Asaithambi, Ph.D.
Director of the School

Accepted for the College of Computing, Engineering, and Construction:

______________________________ ________________
Mark A. Tumeo, Ph.D.
Dean of the College

Accepted for the University:

______________________________ ________________
Len Roberson, Ph.D.
Dean of the Graduate School

ACKNOWLEDGEMENT

To my husband Kiran and daughter Saachi, thank you for your patience, support and love. To my parents, thank you for challenging me to do better.

Special thanks for inspiration go to my thesis advisor Dr. Roger Eggen. Thank you Dr. Eggen for your expert advice and support; you have been patient and encouraging at all times. I also thank Dr. Sanjay Ahuja and Dr. Behrooz Seyed Abbassi for providing valuable insights and feedback in this research process.

TABLE OF CONTENTS

List of Figures viii

List of Tables x

Abstract xi

Chapter 1: Introduction 1

1.1 Phases of a Compiler 1

1.1.1 Scanner 1

1.1.2 Parser 2

1.1.3 Semantic Analyzer 3

1.1.4 Source Code Optimizer 3

Chapter 2: Literature Review 6

2.1 Parallelizing the Semantic Analysis Phase 6

2.1.1 DKY Avoidance 7

2.1.2 DKY Handling 7

2.1.3 Hybrid Approach 8

2.2 Parallelizing the Lexical Analysis Phase 8

2.3 Parallelizing the Code Generator 9

2.3.1 Master Level 10

2.3.2 Section Level 11

2.3.3 Function Level 11

2.4 Parallelizing a bottom-up parser 11

Chapter 3: Implementation Issues 13

Chapter 4: Parallel Compiler Design 15

4.1 Compiler Structure 15

4.2 Compiler Architecture 18

4.2.1 Master 20

4.2.2 Worker 21

4.3 Design Issues 24

4.4 Host Environment 25

4.4.1 Uranus 26

4.4.2 Atlas 26

Chapter 5: Results 28

5.1 Research Methodology 28

5.2 Test data 29

5.3 Performance Results 32

5.4 Performance Comparison 39

5.5 Statistical Significance 42

5.6 DKY versus No DKY 43

5.7 Contributions 46

Chapter 6: Conclusion 49

Chapter 7: Future Work 51

References 52

Appendix A 54

Vita 64

LIST OF FIGURES

Figure 1: Phases of a Compiler 2

Figure 2: Structure of Parallel Code Generator 10

Figure 3: Structure of a sample program 16

Figure 4: Symbol Table Structure 17

Figure 5:Parallel Compiler Structure 19

Figure 6:Response times for 30 LOC using Atlas 30

Figure 7:Response times for 30 LOC using Atlas RMI 30

Figure 8:Response times for 30 LOC using Uranus... 31

Figure 9:Response times for 50 LOC using Atlas 33

Figure 10:Response times for 50 LOC using Atlas RMI 33

Figure 11:Response times for 50 LOC using Uranus... 34

Figure 12: Response times for 10000 LOC using Atlas 35

Figure 13: Response times for 10000 LOC using Atlas RMI 35

Figure 14: Response times for 10000 LOC using Uranus... 36

Figure 15: Response times for Atlas 37

Figure 16: Response times for Atlas RMI 38

Figure 17: Response times for Uranus... 38

Figure 18: Comparison of response times for 500 LOC 40

Figure 19: Comparison of response times for 5000 LOC 41

Figure 20: DKY versus No DKY for Atlas 44

Figure 21: DKY versus No DKY for Atlas RMI..45

Figure 22: DKY versus No DKY for Uranus 45

Figure 23:Response times for 100 LOC using Atlas..54

Figure 24:Response times for 100 LOC using Atlas RMI..55

Figure 25:Response times for 100 LOC using Uranus..55

Figure 26:Response times for 500 LOC using Atlas..56

Figure 27:Response times for 500 LOC using Atlas RMI..56

Figure 28:Response times for 500 LOC using Uranus..57

Figure 29:Response times for 1000 LOC using Atlas..58

Figure 30: Response times for 1000 LOC using Atlas RMI..58

Figure 31: Response times for 1000 LOC using Uranus..59

Figure 32:Response times for 2000 LOC using Atlas..60

Figure 33: Response times for 2000 LOC using Atlas RMI..60

Figure 34: Response times for 2000 LOC using Uranus..61

Figure 35: Response times for 5000 LOC using Atlas..62

Figure 36: Response times for 5000 LOC using Atlas RMI..62

Figure 37: Response times for 5000 LOC using Uranus..63

LIST OF TABLES

Table 1: Statistical Significance Tests 43

ABSTRACT

Parallel programming is prevalent in every field mainly to speed up computation. Advancements in multiprocessor technology fuel this trend toward parallel programming. However, modern compilers are still largely single threaded and do not take advantage of the machine resources available to them. There has been a lot of work done on compilers that add parallel constructs to the programs they are compiling, enabling programs to exploit parallelism at run time. Auto parallelization of loops by a compiler is one such example. Researchers have done very little work towards parallelizing the compilation process itself.

The research done here focuses on parallel compilers that target computation speedup by parallelizing the process of program compilation during the lexical analysis and semantic analysis phase. Parallelization brings along with it issues like synchronization, concurrency and communication overhead. In the semantic analysis phase, these issues are of particular relevance during the construction of the symbol table. Research done on a concurrent compiler developed at the University of Toronto in 1991 proposed three techniques to address the generation of the symbol table [Seshadri91]. The goal here is to implement a parallel compiler using concepts from those techniques as references. The research done here will augment the work done formerly and measure the performance speedup obtained.

Chapter 1

INTRODUCTION

A compiler is a computer program that translates a program written in one language into an equivalent program written in its target language [Louden97]. The target language can be machine code or intermediate code. Research continues to this day towards generating efficient machine code.

1.1 Phases of a Compiler

Figure 1 illustrates the different phases of a compiler. Every phase in the compiler plays a distinct role. In a compiler implementation, these distinct phases can be coded as different units. Some of these phases are often grouped together. In the parallel compiler developed here the parser, semantic analyzer and the source code optimizer are grouped together. Parallelism was applied to the first four phases of the compiler. This section briefly discusses the first four phases.

1.1.1 Scanner

A scanner also called a lexical analyzer breaks the source code into atomic units of the language called tokens. Keywords and identifiers are examples of tokens. Some compiler

designers start constructing the symbol table in this stage. The scanner invokes the error handler if the characters in the input do not conform to the specified grammar.

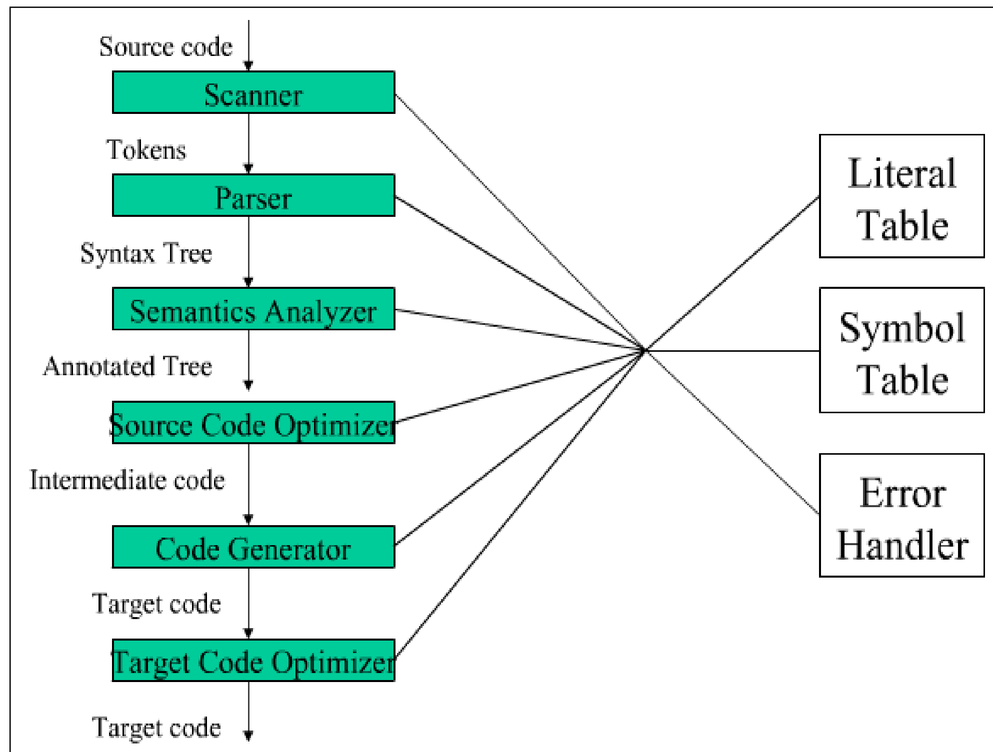

Figure 1: Phases of a Compiler [Louden97]

1.1.2 Parser

The parser performs syntax analysis on the tokens provided by the lexical analysis phase. It verifies that the source code conforms to the syntactic structure defined by the grammar

of the language. Parse tree or syntax tree construction is typically done in this phase based on the rules defined by the grammar.

1.1.3 Semantic Analyzer

The semantic analysis phase verifies that the source code has meaning. For example, this phase can verify that variables have been declared before they are used and can also perform type checking. In this phase, the compiler typically enters information about the data types, scopes and other attributes associated with identifiers into the symbol table. This information guides the semantic analysis phase and subsequent phases of the compiler use this information.

1.1.4 Source Code Optimizer

In this phase, the compiler may include code improvements or optimizations to the source code. In addition, the compiler may also generate intermediate code in this phase. Intermediate code is a form of code representation intermediate between source code and object code [Louden97].

The code generator and target code optimizer generate code for the target computer. The first three phases perform majority of the analysis portion of the compiler. The literal table, symbol table and the error handler interact with some or all of the components. The literal table stores the constants and literals used in the program. The symbol table is a data

structure that contains a record for each identifier. This record stores information about the attributes of the identifier such as name, type, size (for array), return type (for function) etc. Any of the phases of the compiler can invoke the error handler. The first three phases detect a large fraction of the errors.

Compilation of large programs could take a substantial amount of time. With the availability of multi core processors parallel computing is emerging as a prevalent computing paradigm. Modern compilers are now capable of applying optimizations that produce highly efficient code targeted for multiprocessors. The industry focus is largely on producing optimizing compilers that add parallel constructs to the programs, therefore allowing the developers of the code to be oblivious of the underlying machine architecture. However, the compilation process itself is far from optimized. Adding parallel constructs to the program can affect the overall time needed for compilation. Parallelizing the compilation process can help reduce this time.

Compiler code can take advantage of the multiprocessor technology by applying parallelism to the sequential phases of the compiler to speed up compilation time. Although compilation in parallel sounds promising, achieving overall speedup, efficiency and ease of implementation has been an elusive research goal to date.

The present study will enhance the work done in [Seshadri91] by identifying gaps in the techniques described there and addressing those gaps. The techniques used by [Seshadri91] and gaps are discussed in further detail in section 2.1. Although the focus is

on parallelizing the semantic analysis phase, the design used here inherently parallelizes the lexical analysis phase too. Recent work on parallelizing the lexical analysis stage was successfully accomplished with satisfactory performance [Kumar11] and hence the techniques for parallelizing this phase are not discussed here. The parsing technique used by the proposed parallel compiler is top down recursive descent parsing. Applying parallelization techniques to LL(k) and bottom up parsers are beyond the scope of this research. The language used to implement the compiler is Java and parallelism is achieved using Java's concurrent library. The compiler is implemented for two different computer hardware architectures: a shared memory multiprocessor architecture and a Beowulf cluster. The performance metrics obtained for the parallel compiler and the speedup attained with the parallel version as compared to the sequential version of the compiler are discussed here.

Chapter 2

LITERATURE REVIEW

This chapter summarizes recent papers on parallel parsing and compiling. The work of Seshadri and Wortman [Seshadri91] discussed in section 2.1 is chosen as the primary reference in view of the fact that it is also an attempt at trying to solve the same problem; achieving parallelism in the semantic analysis phase. In addition, the authors present a well-structured analysis of the problem at hand.

2.1 Parallelizing the Semantic Analysis Phase

The concurrent compiler developed at the University of Toronto [Seshadri88, Wortman 92] takes the approach of applying parallelism in the semantic analysis phase of compilation. The compiler is built for source languages that require identifiers to be declared before they are referenced and have reserved words that determine program structure. The lexical analysis stage is sequential and was enhanced to recognize structural boundaries and split the source code into blocks for further processing. Rather than splitting up the program into random blocks for parallel processing, the approach taken was to partition data at scope boundaries. A merge operation later combines the object code produced into one program. Some of the major challenges encountered in this approach were the construction of the symbol table and error reporting. The symbol table had to be protected by mutual exclusion mechanisms

to prevent simultaneous writes to the table. This could result in a lot of time spent by a process just waiting to get a lock on the symbol table and consequently slow down processing. Moreover, the symbol table lookup operations for identifiers had to take into account that the tables could be incomplete. Because of concurrent processing, the declaration of an identifier might not be processed before the identifier is used. It is not possible to know at this stage if the declaration does exist and will be processed subsequently. The authors term this scenario as the "doesn't know yet" (DKY) problem. The authors propose the below three strategies for dealing with the DKY problem [Seshadri91].

2.1.1 DKY Avoidance

In this approach, parent scopes are processed before any child scopes resulting in simplified symbol table management. If an identifier declaration is not found while performing a symbol table lookup then it is safe for the compiler to flag it as an error. However, this strategy can affect parallel processing. The amount of parallelism achieved would heavily depend on the structure of the program being compiled.

2.1.2 DKY Handling

In this approach, DKYs are allowed to occur; the process encountering the DKY is suspended until another process resolves the DKY. This complicates and slows down symbol table operations.

2.1.3 Hybrid Approach

Semantic analysis is split into two phases. In the first phase all the declarations in the program are processed and the symbol table is constructed from this information. Construction of the symbol table is complete after this phase. In the second phase, statements of the program are processed. This eliminates any synchronization issues in the second phase and simplifies the compiler algorithm used for parallelism.

Experimental results show that performance of all three approaches was alike. The performance difference between DKY handling and DKY avoidance was small due to significant identifier cross usage between scopes. The hybrid approach did not outperform the other two approaches either. The compiler was built for Modula-2+. Declaration processing in Modula-2+ took more time than statement processing and hence the hybrid approach did not achieve a significant speedup over the other approaches. The speedup obtained was measured as a ratio of execution time of the sequential algorithm to the execution time of the parallel algorithm. The average speedup factor for the above three approaches obtained over a wide variety of source programs was approximately 2.5.

2.2 Parallelizing the Lexical Analysis Phase

Parallelism can be invoked in the lexical analyzer's scanning and tokenizing phases. [Srikanth10] attempts to parallelize tokenization by implementing a prototype of a parallelized lexical analyzer that recognizes tokens of a given language. Aho-Corasick is

used for pattern matching because of its high speed string search capabilities. A block-splitting algorithm is used to split the input into blocks, ensuring that no token is divided over a block boundary. A static block size is first determined based on the input file size. Based on this static block size, the input file is then split into dynamic blocks using the newline character as a delimiter. The blocks are then processed in parallel. Simulated results show that parallelizing the lexical analyzer stage yields substantial improvement in performance over its sequential counterpart.

Kumar *et al.* extended the above work on parallelizing the lexical analyzer [Kumar11]. They too use the Aho-Corasick algorithm for keyword recognition. In order to speedup processing, the source code is first run through a processing element that removes all single line and multiline comments from the source code. Parallel processing of blocks using a dynamic block splitting algorithm is initiated after all comments have been removed. The tokens resulting from each block are written into separate files that are then combined together in the same order that was used to split them. Performance analysis on the parallelized lexical analyzer shows optimized performance with a 50% reduction in execution time as compared to the traditional sequential version.

2.3 Parallelizing the Code Generator

Gross *et al.* explored parallelism in the optimization and code generation phases of compilation [Gross89]. The structure of the programming language used as an input to the compiler consists of a high-level module. This module can contain one or more sections.

Sections in turn can contain one or more functions that constitute a unit of work. These sections can execute independently and hence are good candidates for parallel compilation. The structure of their parallel compiler, as shown in Figure 2, has three hierarchies: master level, section level and function level.

2.3.1 Master Level

The master level corresponds to a module and has exactly one process. The master is aware of the number of sections in the program and hence knows the number of processes it has to spin off.

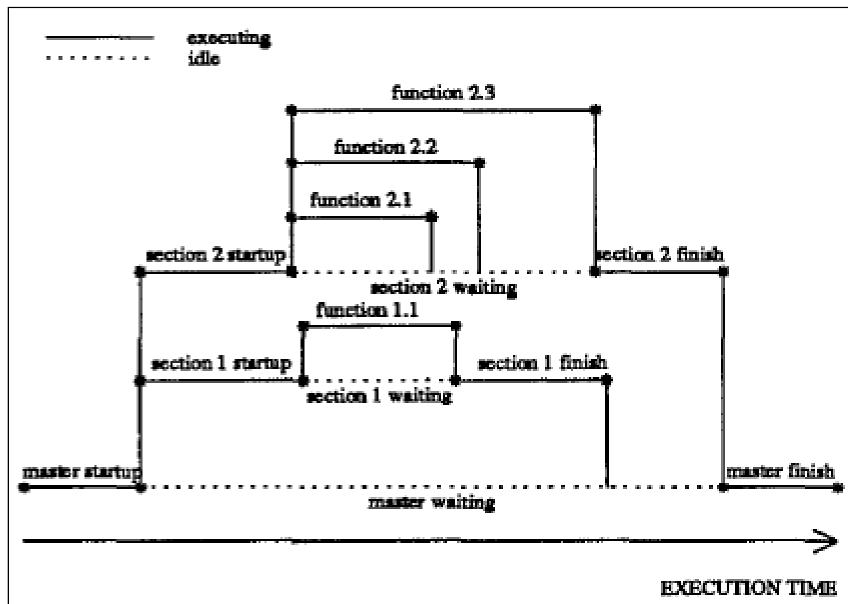

Figure 2: Structure of Parallel Code Generator
[Gross89]

2.3.2 Section Level

Processes that work with the different sections of the module are called section masters. Exactly one section master controls each section. After code generation for all the functions within a section is complete, the section master combines these results.

2.3.3 Function Level

Processes that deal with each function are called function masters. The function master performs the optimization and code generation operations of the compiler.

The master communicates via messages with the section masters under it, as there is no global shared memory involved in the host architecture. Experiments performed by the authors show a speedup factor ranging from three to six over that of the sequential version of their compiler.

2.4 Parallelizing a bottom-up parser

Cohen *et al.* tackled compiler parallelization on a bottom-up parser [Cohen85]. Each processor has its own stack. This stack is used for parsing using the shift-reduce method. A shift consists of pushing the next input token onto the stack. If the reduce operation finds the right hand side of a grammar rule on the top of the stack, it replaces it with its left-hand

side. The grammar used for the parser is a Pascal-like grammar. The input program is split up into n parts where n is the number of available processors. Semicolon or any such equivalent delimiter is used for the split. This reduces nondeterministic situations since a processor will start processing a statement after a semicolon and it would not have to deal with incomplete statements. Each processor performs shift-reduce parsing using its individual stack. If there is not enough information for the process to continue with a reduce operation it places a flag on top of the stack, jumps to the next semicolon and continues with the parsing operation. When a process finishes parsing its portion, it signals its right neighbor for a merge operation and waits for the neighbor to respond. Any flags on the merged stack are dealt with appropriately and eventually the first or the leftmost processor is the only one left to finish the job. Performance results show that the speedup attained was affected due to high times involved with waiting for the merge operation. As a result, the speedup obtained was not as high as predicted by theory.

Chapter 3

IMPLEMENTATION ISSUES

This chapter discusses some of the implementation issues faced when designing a parallel compiler. The traditional serial compilation process does not take advantage of the availability of multiprocessor computers. Parallelizing the different phases in the compiler will allow the compiler to employ more time consuming optimizations thus improving the efficiency of the language. Parallelizing the lexical analysis phase is straightforward. A more difficult aspect of parallel compilation is parallel semantic analysis. The challenge with parallelizing the semantic analysis phase involves splitting up the program into meaningful blocks such that it minimizes the communication required between different processes. Applying an arbitrary static block splitting algorithm complicates the process of merging the states of adjacent parallel processors.

Another design issue with parallelizing the semantic analysis phase is symbol table creation and management. The semantic analysis phase accesses the symbol table frequently to perform additions, deletions and read operations. It is critical for these operations to be efficient and performed in near constant time. As with any kind of parallelism that involves a shared data structure, concurrency and synchronization problems could negate any performance benefit attained.

The traditional compiler algorithms for sequential compilers ensure that the outer scopes are built and that declarations are added to the symbol table before the processing of the inner scopes begin and before these declarations get used. This makes error reporting straightforward because the compiler can flag an error when encountering an identifier not found in the symbol table. A parallel compiler will have to take into account that the outer scope processing might not have completed while the inner scope is processed. The compiler will have to defer error reporting until a future point in time when it knows that all related scopes have definitely been processed. The goal of the present study is to achieve parallelism, while preserving the integrity of the semantic analysis phase so as not to incur incorrect behavior or results.

Chapter 4

PARALLEL COMPILER DESIGN

This chapter begins by discussing the structure of the parallel compiler. Section 4.2 discusses the compiler architecture and its design. Section 4.3 discusses some of the design issues encountered and the strategies used to deal with them.

4.1 Compiler Structure

The grammar chosen for implementing the compiler is a subset of C but this compilation technique can be extended to source languages which:

- Are block structured, with blocks of declarations and statements.
- Require identifier declaration before use.

The source program is divided into multiple parallel units such that each unit can be processed and compiled in parallel. The approach for data partitioning used in the present study is the same as the one used in [Seshadri91]. Function boundaries are used to partition data. In other words, function bodies make up a unit for parallel processing. This approach to partitioning reduces the dependency between processes and can save some expensive communication time between processes. The input is distributed to multiple processors such that each processor processes a function in its entirety. This allows each processor to process the data local to it more efficiently and minimizes the need for exchange of information between processors.

```
int pivot;
int i;
int j;
void quickSort(int numbers[], int arraysize)
{
  sort(numbers, 0, arraysize - 1);
}


void sort(int values[], int left, int right)
{
  i = left;
  j = right;
  pivot = values[left];
  while (left < right)
  {
    while ((values[right] >= pivot) )
      right = right -1;
    if (left != right)
    {
      values[left] = values[right];
      left = left + 1;
    }
    while ((values[left] <= pivot) )
      left = left + 1;
    if (left != right)
    {
      values[right] = values[left];
      right = right -1;
    }
  }
  values[left] = pivot;
  pivot = left;
  left = i;
  right = j;
  if (left < pivot)
    sort(values, left, pivot-1);
  if (right > pivot)
    sort(values, pivot+1, right);
}
```

Figure 3: Structure of a sample program

Figure 3 shows the structure of a sample program and Figure 4 shows how this program would be split into parallel units. Statements defined in functions quickSort and sort can

reference variable declarations pivot, i and j. The outer scope in which these variables are declared serves as the parent scope for all function declarations.

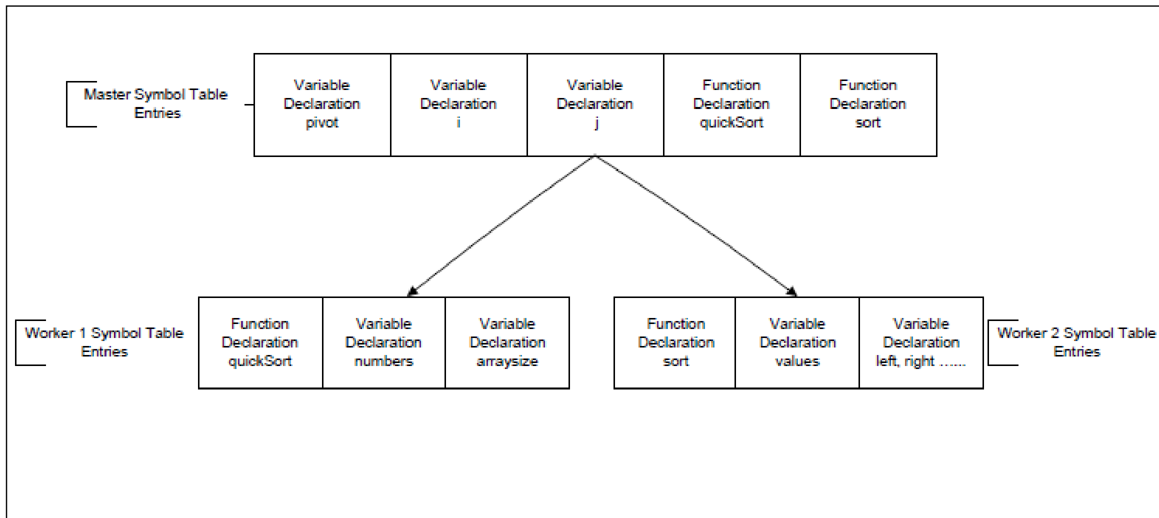

Figure 4: Symbol Table Structure

The program in Figure 3 will result in two additional parallel units being spawned. One unit handles the quicksort function while another unit handles the sort function. The source program is split in the lexical analysis phase. Along with generating tokens, the lexical analyzer also recognizes function boundaries.

4.2 Compiler Architecture

As discussed in section 1.1, the compiler consists of four phases. Non-serial compilation can be readily applied to lexical analysis stage [Kumar11]. Parallelizing the semantic analysis phase poses more of a difficulty due to complexities introduced by symbol table management. A child scope may reference identifiers declared in the parent scope. In sequential compilation, these identifiers will definitely be present in the parent's symbol table since outer scopes will be processed before the inner scopes. With parallel compilation, it is possible that semantic analysis on the scope in which an identifier is declared is not yet complete resulting in incomplete symbol tables. The semantic analysis phase of a parallel compiler has to account for these missing symbol table entries when processing statements that reference identifiers in outer scopes. The semantic analysis phase does not know yet if the identifier is truly declared in one of the outer scopes. Therefore, it cannot decide at this point whether to flag this variable as an error. [Seshadri91] uses the term "doesn't know yet" or DKY to refer to this problem where an identifier has not been found in any parent scope. This section discusses the structure of the parallel compiler along with the technique used to handle incomplete symbol table entries.

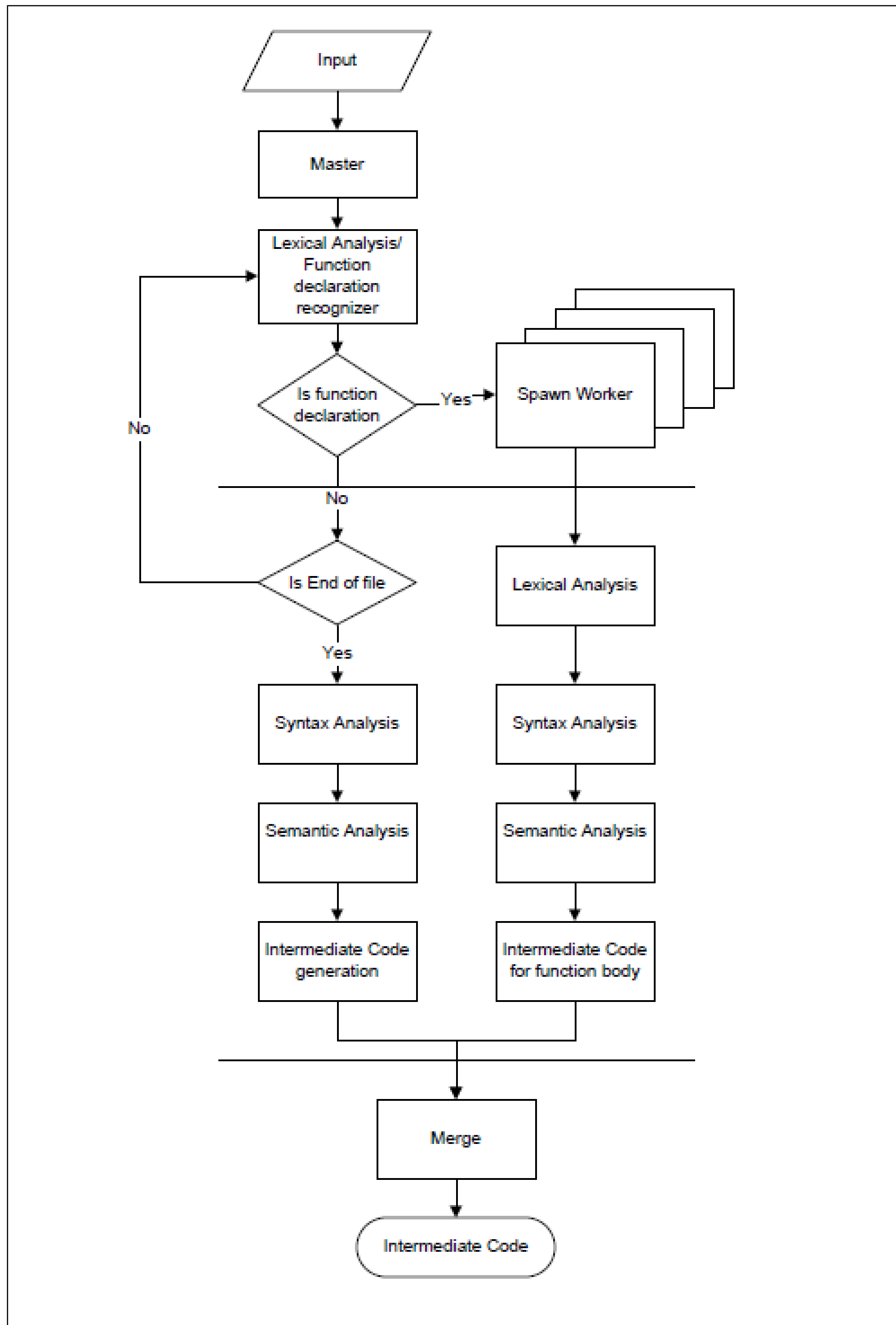

Figure 5: Parallel Compiler Structure

Master - Worker Pattern

The parallel compiler is implemented using the Master-Worker design pattern. A Master-Worker design pattern allows identical computations to be performed in parallel. Figure 5 demonstrates the master-worker pattern and explanation on how the parallel compiler uses this pattern follows.

4.2.1 Master

The main process invoked when processing begins is the master. The master is responsible for lexical, syntax and semantic analysis as well as intermediate code generation for all the global variable and function declarations. The master does not analyze the function body itself. The master first invokes lexical analysis on the input file specified. During the lexical analysis phase, the master looks for any tokens that indicate the start of a function. If the master finds a function declaration, it invokes a worker and passes the file pointer handle that holds the start of the function to the worker. The master then continues with the lexical analysis on the file but does not generate any tokens until it finds the end of the function that it just encountered. The master only generates tokens for any non-function identifiers it encounters.

The process continues with the master invoking a new worker every time it encounters a function declaration. A thread pool dictates how many workers can be active at a time. If the number of functions in a file exceeds the available workers in the pool, the additional workers will be queued. The thread pool matches the number of available processors. A

simple first come first served strategy is used to distribute the tasks over the available workers.

Once the master is done with the lexical analysis stage, it continues with syntax analysis, semantic analysis and intermediate code generation phases with the tokens it had scanned. In the semantic analysis phase, the master builds the symbol table referred to as the master symbol table. The master symbol table will contain all global declarations. In other words, it will contain all scope zero function and variable declarations.

The master waits for all workers to complete their lexical analysis, syntax analysis, semantic analysis and intermediate code generator phases. It then combines the result of processing and validates the outputs from all workers. Figure 5 shows the structure of the compiler with the master or the main thread invoking workers as and when needed.

4.2.2 Worker

The master invokes a worker for every function declaration found in the program. The worker is responsible for lexical, syntax and semantic analysis as well as intermediate code generation for the function body. The worker first invokes lexical analysis on the function body. It stops its lexical analysis when it finds the end of function.

When the worker has enough information about the attributes of the function that it is processing, it sends an update message to the master with this information. Attributes of a

function include its return type, number of arguments and the data type of those arguments. The master updates its symbol table with this information so that the function declaration is available for use by any other workers that make a call to that function. This is the only scenario in which the worker sends an identifier over to the master so that the master can add it to its symbol table.

The tokens generated from the lexical analysis stage are then subject to syntax analysis, semantic analysis and intermediate code generation phases. In the semantic analysis phase, the worker keeps track of any DKY's. When the worker encounters an identifier with a DKY, the worker marks that identifier as a dummy and adds it to a dummy symbol table. As processing continues, the worker starts guessing the attributes related to the dummy identifier. The logic behind the guesswork is to assign values to the identifier that will avoid a compile error at that point in time. For example, consider that the worker comes across a statement as below:

```
SUM = ADD(2,3);
```

Supposing the worker did not find the declaration of the identifier SUM in the master symbol table. It first adds SUM to the dummy symbol table. In order for the above statement to not throw a compile error, the type of SUM needs to be the same as the return type of the function ADD. Two scenarios are possible here: ADD is found in the master symbol table, or ADD also had a DKY. If ADD is found in the master symbol table, the worker assigns the return type of function ADD to the data type of SUM. It stores this information in the dummy symbol table. If ADD had a DKY, the worker cannot deduce any information about the data type of SUM from the statement. In this case, it stores the

fact that SUM and ADD are related by type. The parser can deduce this information from the order of parsing inherent in a recursive descent parser. Information about related identifiers, i.e. identifiers related by type, which have DKY's is stored in a related identifiers list.

For future lookups of the same identifier by the same worker, first the master symbol table will be searched and then the dummy symbol table. If the master did process the identifier by this point in time, the entry from the master symbol table is retrieved. The entry from the dummy symbol table will be retrieved if the master has not processed the identifier yet. The dummy symbol table is local to the worker; the master symbol table is never updated with the information from the dummy symbol table. Once processing of its block of data is complete, the worker hands the intermediate code it generated along with the dummy symbol table back to the master. If a related identifiers list was created during processing, that list is also sent back to the master.

When all workers have finished processing their respective functions the master has all the information necessary in its master symbol table to validate the results from the workers. Validation includes verifying that any identifiers with DKY's are in fact present in the master symbol table. In addition, any information that was guessed by the workers is validated against the entry in the master symbol table. If there is a disparity between the guessed attributes of an identifier and the attributes of the identifier found in the master symbol table the identifier is flagged as an error.

Let us consider the previous statement:

SUM = ADD (2, 3);

Let us assume that SUM is of type integer and return type of ADD is a float.

If SUM had a DKY and ADD did not have a DKY, the worker would have added SUM to the dummy symbol table and assigned float as its type. When the master is validating the results from the workers, it finds a conflict between the declaration for SUM in the dummy symbol table and the master symbol table and reports the conflict as an error. If SUM and ADD both had a DKY, they would be added to the related identifiers list. When examining this related identifiers list, the master would catch the fact that SUM and ADD have different types.

Creating dummy identifiers and guessing their attributes reduces the overhead involved with inter-process communication. This approach drastically minimizes the amount of concurrent writes to the symbol table thereby reducing the number of mutual exclusion locks on the symbol table. Threads running in parallel do not have to wait indefinitely or wait at all. They can continue to run with the amount of information available to them, and delegate the error handling to another thread if they come across identifiers that they cannot find in any scope.

4.3 Design Issues

One of the design issues faced was how to avoid an extra parse of the program that would be needed to split the program into parallel units based on function boundaries. This issue

was solved by adding a little intelligence to the lexical analyzer so that it can recognize the beginning and end of a function. When it encounters a type specifier the lexical analyzer knows that this could either be a variable declaration or a function declaration. In case of a function declaration, the next identifier will be followed by an opening parenthesis. Splitting the program in the lexical analysis phase allowed the compiler to take advantage of parallelism early on.

Another issue to address was the design of the symbol table in order to minimize the time spent on synchronization and locking mechanisms. Having a separate master symbol table and individual worker symbol tables helped solve this issue. The only time the worker needs to send a message to the master to update the master symbol table is when the worker begins processing its function. It sends over the function attributes to the master. Maintaining a separate dummy symbol table per worker for any identifiers with DKY has also helped alleviate the writes on the master symbol table.

4.4 Host Environment

The parallel compiler was run on two different host systems to evaluate which computer architecture would suit the program better. Following are the specifications for the two systems.

4.4.1 Uranus

Uranus is a thirteen-node Beowulf cluster with Gigabit Ethernet network. All nodes are made up of 2.83GHz Intel Xeon processor. On this distributed Uranus cluster, communication between the master and the workers was achieved using message passing. This message passing was implemented using Java's remote method invocation interface [Golub09]. In order to run the tests the workers are first started on the remote nodes. The parallel compiler is then invoked which in turn invokes the master.

4.4.2 Atlas

Atlas is a shared memory multiprocessor machine. It has a Quad Quad-Core Intel Xeon processor with a total of 64 threads running at 2.00 GHz along with 128 GB RAM. In order to take advantage of the shared memory system in Atlas two versions of the program were created. The first version does not use any remote method invocation. The master, at runtime, first creates the workers and then invokes them with tasks. This program will be referred to as the Atlas program. Since the master and workers run on the same Java Virtual Machine (JVM), they can easily take advantage of the shared memory system provided by Atlas.

In the second version, the workers are first started and initialized before the compiler is invoked. The master does not create or initialize the workers. It just calls them with tasks as needed. This is similar to the program developed for Uranus in 4.4.1. It uses Java's

remote method invocation (RMI) for passing messages back and forth between the master and the workers. This program is the Atlas RMI program.

Theoretically, both the versions above have their advantages and disadvantages. In the Atlas version, the workers have the advantage that their copy of the master symbol table is always current since the master symbol table is a shared data structure. This should lead to fewer DKY's. The disadvantage of this program though is that the workers have to go through initialization every single time the program runs. This is because the master creates and initializes the workers. This initialization time adds to the overall response time of the compiler. On the contrary, the response time for the Atlas RMI program will not be dependent on the time it takes to initialize workers. However, since the workers and the master do not run on the same JVM, they communicate with each other using Java's remote method invocation. In this case, the master symbol table becomes distributed.

Chapter 5

RESULTS

Compiling a program, whose size is measured in KLOC (Thousand Lines of Code), can take a significant amount of time on a traditional sequential compiler. Relatively smaller programs might not benefit from parallel compilation. Speedup is measured as the ratio of execution time of the sequential program to the execution time of the parallel program. Ideally, the speedup in compilation time should be n where n is the number of processors involved in compilation. Linear speedup would be the ideal goal, but probably overly optimistic. The overhead associated with communication between multiple processes can prevent linear speedup. The implementation overhead might also contribute to this reduction in performance. In addition, the programming style used in the input program can negatively affect the execution times. Example of this is a program that contains a lot of identifier cross usage between different scopes. The structure of the parallel compiler has been discussed in sections 4.1 and 4.2. This chapter first describes the framework used for testing and then summarizes the performance results collected for the parallel compiler.

5.1 Research Methodology

In order to provide a baseline to measure the effectiveness of the parallel compiler, tests were first run on the sequential version of the compiler. The results of these tests are

compared to results obtained from the tests run on the parallel compiler. The test bed comprises of input programs of different sizes. In addition, programs with different programming styles were also used. This includes programs that have a lot of identifier cross usage between scopes resulting in DKY's.

5.2 Test data

Different kinds of programs and programming styles were used to test the performance of the compiler. The programs used in the test bed are as close to real world examples as possible. It was observed that on both host systems, programs with less than 30 lines of code did not benefit from parallelism. Figure 6, 7 and 8 depict the response time for programs with less than 30 lines of code (LOC). Please note that the sequential response time is represented by the value shown in the graphs when the number of parallel threads is equal to one.

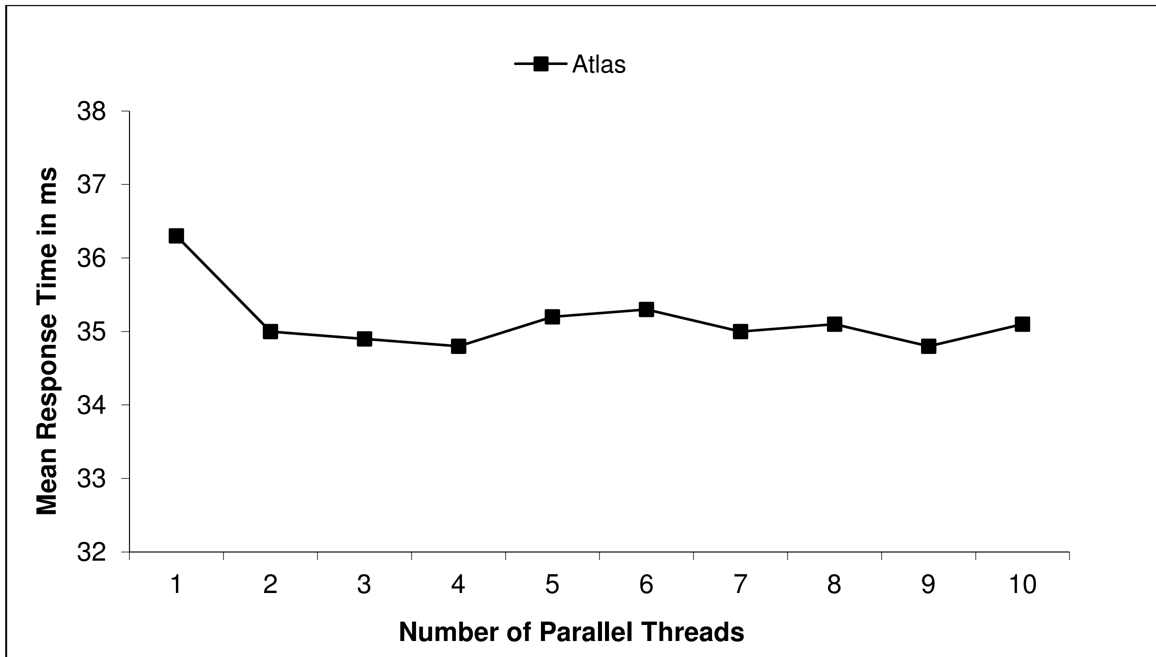

Figure 6: Response times for 30 LOC using Atlas

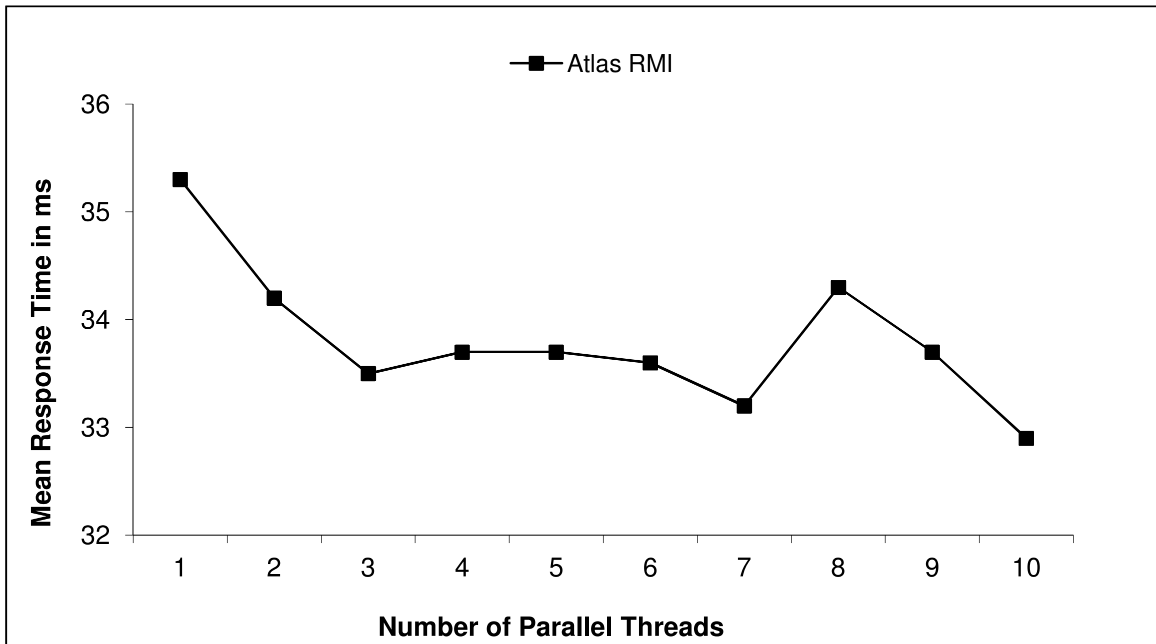

Figure 7: Response times for 30 LOC using Atlas RMI

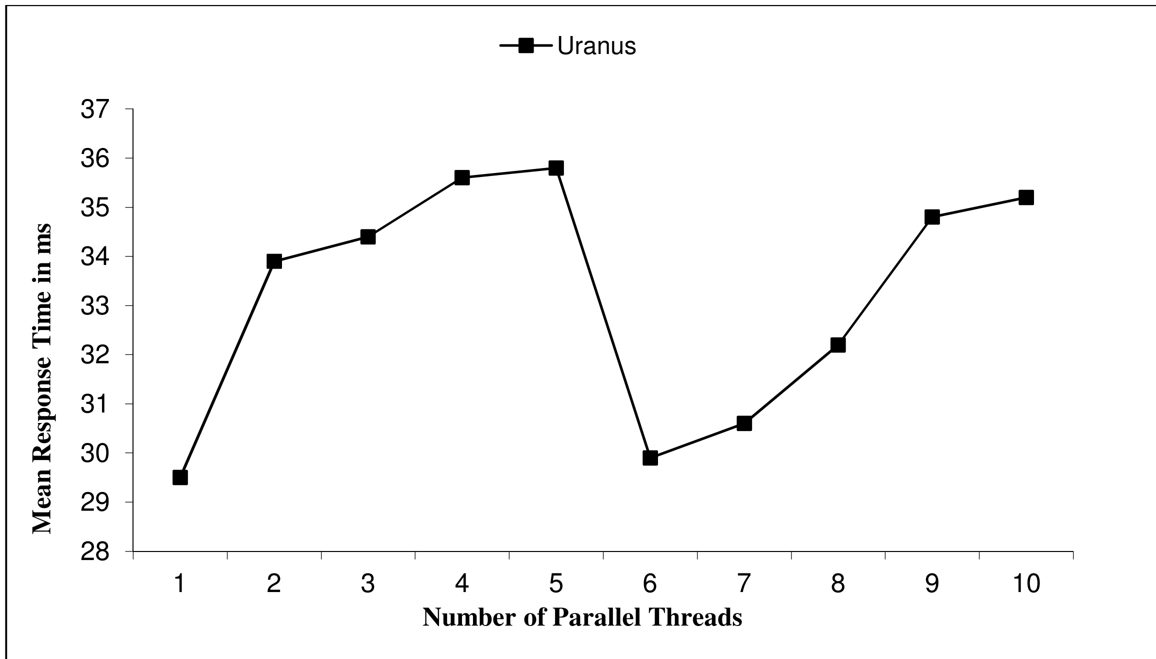

Figure 8: Response times for 30 LOC using Uranus

Figure 6 shows the performance of the parallel compiler on Atlas using a program with 30 lines of code. The mean response time for the sequential compiler is 36ms. This is represented by the value in the graph when the number of parallel threads is equal to one. When two workers are used the mean response time drops to 35ms. As the number of workers is increased, the response time stays at 35ms. The speedup obtained in this case is negligible. Atlas RMI shows a similar trend as can be seen from figure 7. Referring to figure 8, the sequential response time for Uranus is around 29ms. With two workers the mean response time jumps to 34ms showing that distributed communication is costly for small programs.

As can be seen from the above three graphs, the parallel response time is either approximately equal to the sequential response time or slightly more than the sequential response time. Some of the factors that contribute to this increase in response time are:

- The overhead involved with creating parallel threads.
- The overhead involved with message passing.

In the rest of this chapter, only programs with size greater than 30 lines of code are considered.

5.3 Performance Results

In order to evaluate performance, experiments were conducted with programs of varying size. These programs also consisted of functions of varying size. Size of the function itself plays an important role since it defines a unit of work for a parallel thread. The test bed used to measure performance comprised of programs with 50, 100, 500, 1000, 2000, 5000 and 10000 lines of code. All of the input programs used for testing were error free; none of them had syntactic or semantic errors. To capture accurate results each test was run ten times. The response times in the graphs represent the arithmetic mean of the response times from those tests.

Figure 9, 10 and 11 show the response times for a small program with 50 lines of code.

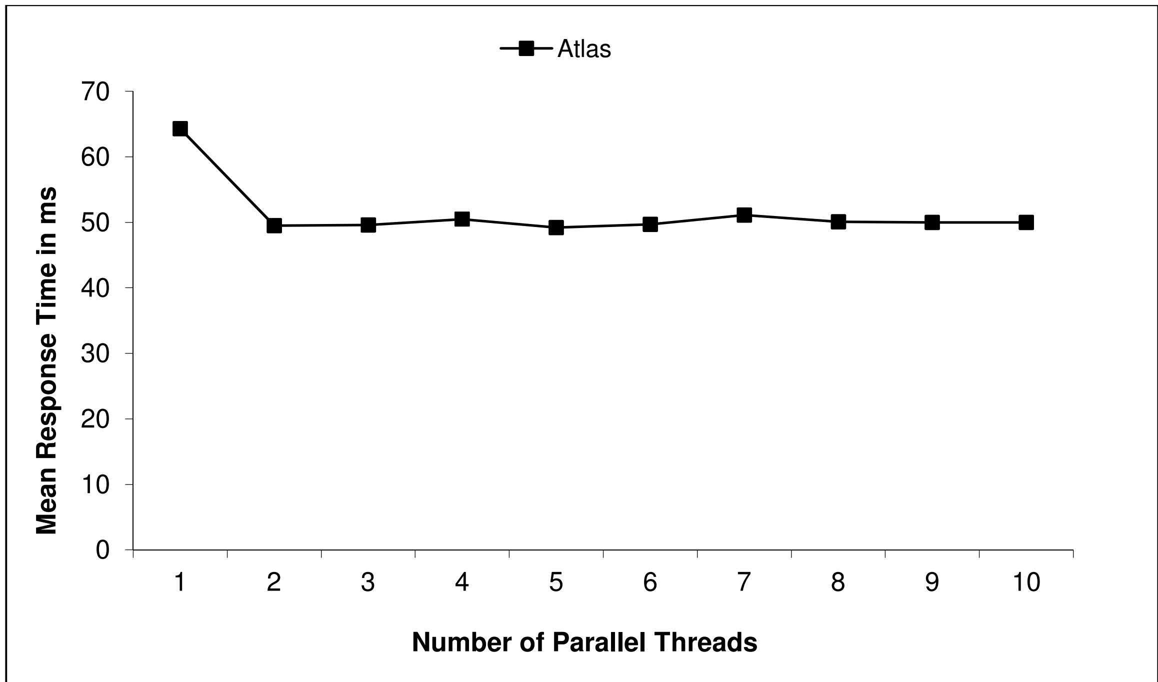

Figure 9: Response times for 50 LOC using Atlas

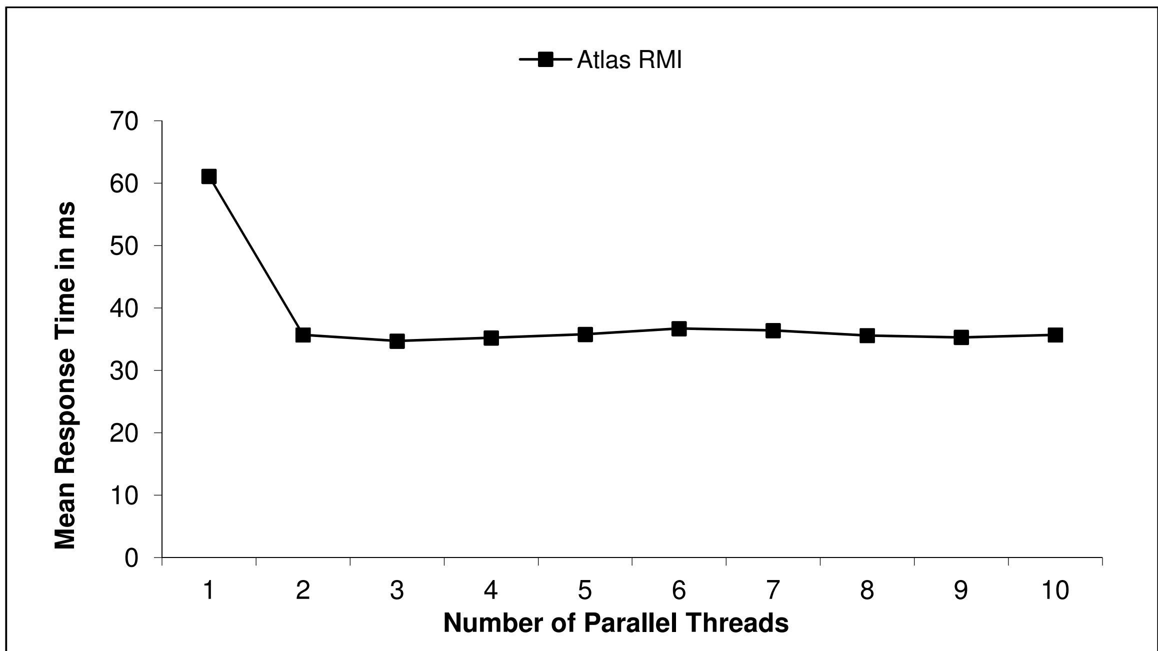

Figure 10: Response times for 50 LOC using Atlas RMI

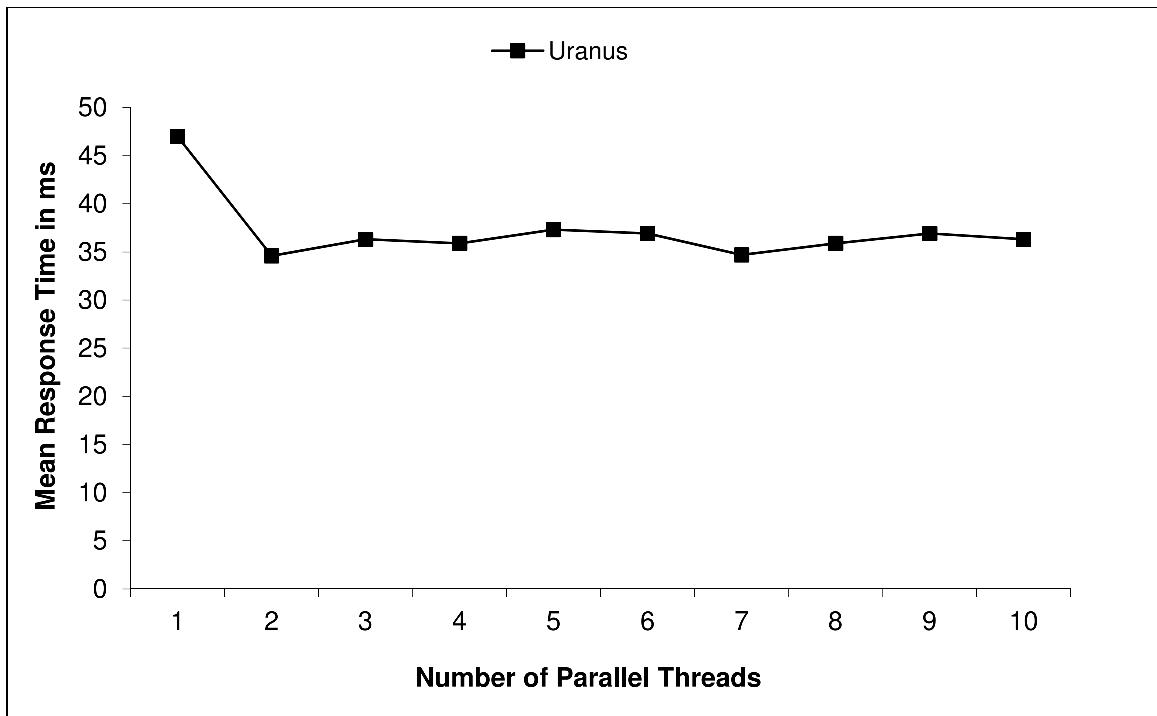

Figure 11: Response times for 50 LOC using Uranus

From figures 9, 10 and 11 it can be seen that for a relatively small program with 50 lines of code, using two threads in parallel reduces the compilation time. Additional parallel threads do not help reduce the compilation time any further.

Figures 12, 13 and 14 show the response time of the parallel compiler for a relatively large program with ten thousand lines of code. Additionally, figures 23 through 37 listed in Appendix A demonstrate the performance of the parallel compiler for inputs of increasing size.

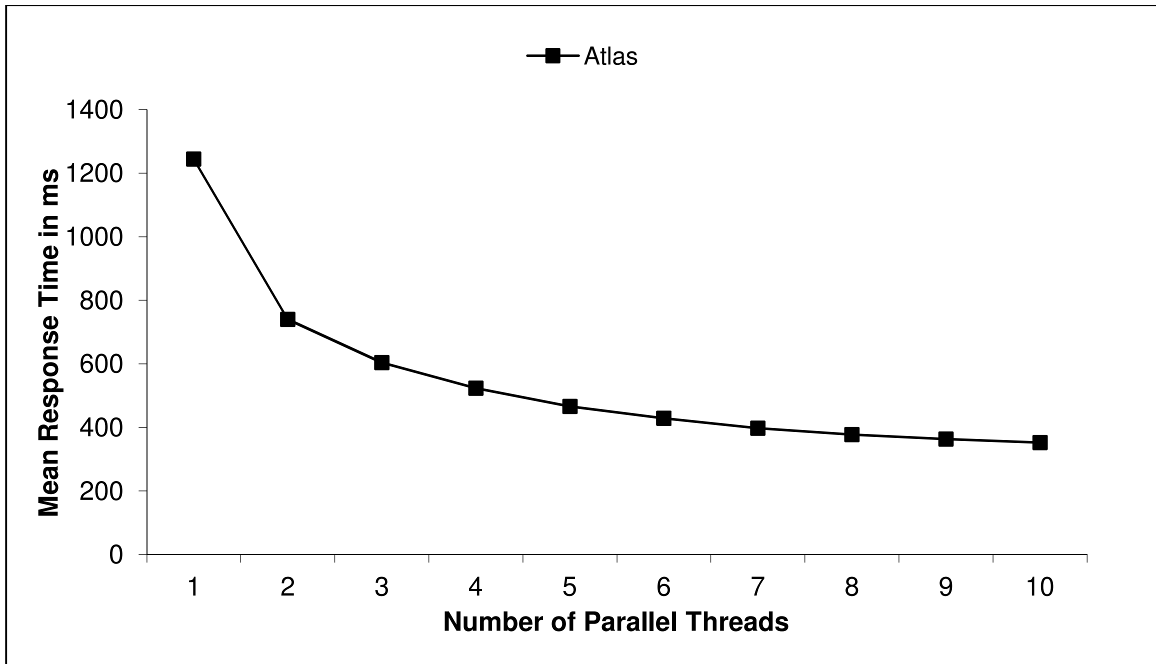

Figure 12: Response times for 10000 LOC using Atlas

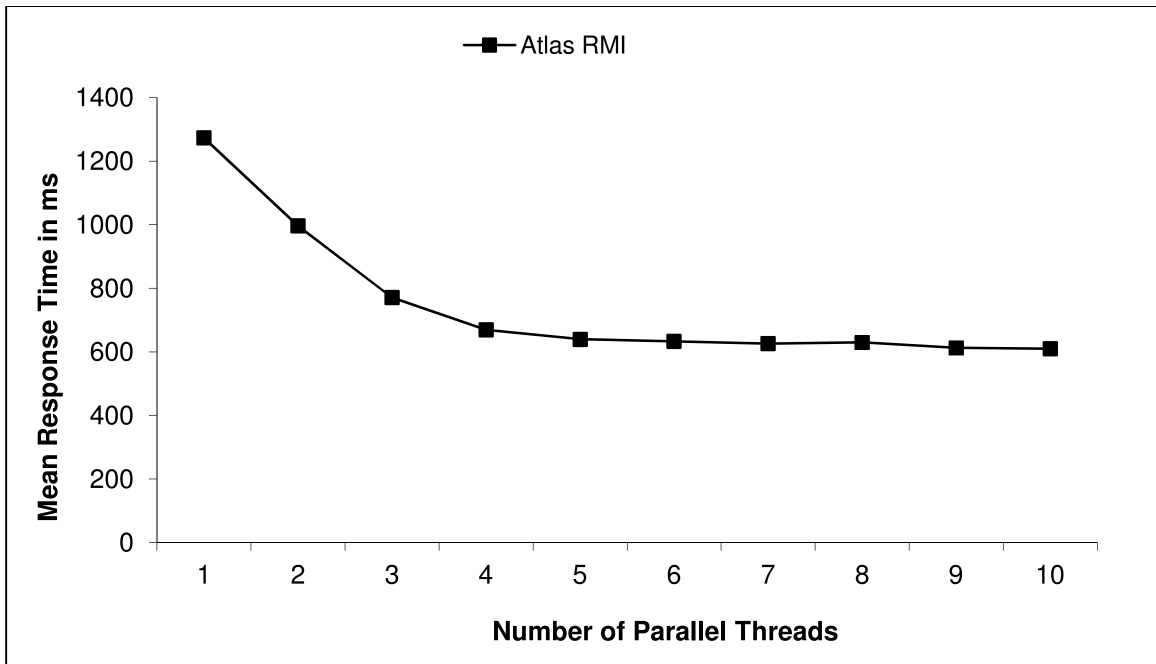

Figure 13: Response times for 10000 LOC using Atlas RMI

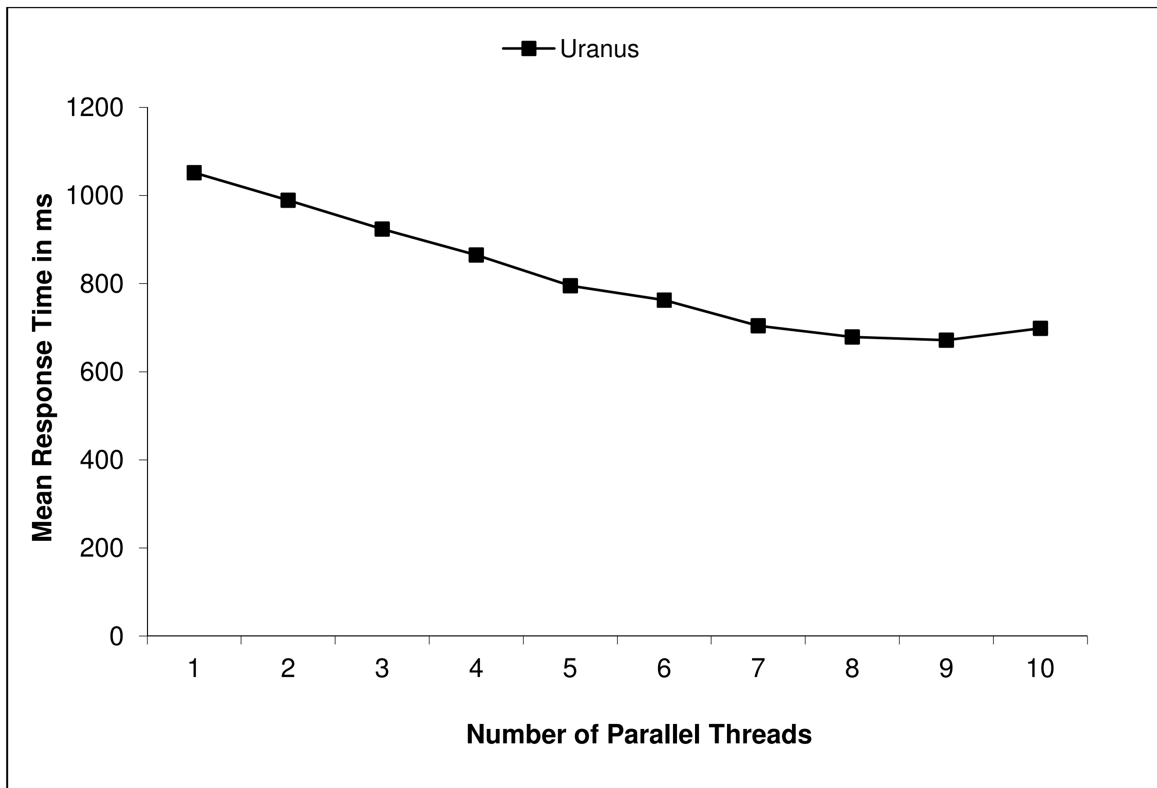

Figure 14: Response times for 10000 LOC using Uranus

The above graphs depict the results obtained for the parallel compiler with all three configurations. The parallel response times are significantly smaller than the sequential response times. Additionally, as can be seen from Appendix A, the speedup obtained increases with the size of the input. In other words, the speedup obtained increases as the number of lines of code in the program increases.

Figures 15 to 17 show performance of the parallel compiler as the size of the input increases. As can been seen from the three graphs, a substantial improvement in

compilation time can be achieved through parallelism. Another observation worth noting is that as the size of the input increases, the resulting increase in the parallel response time is marginal. This is particularly true when the number of parallel processors is around 10.

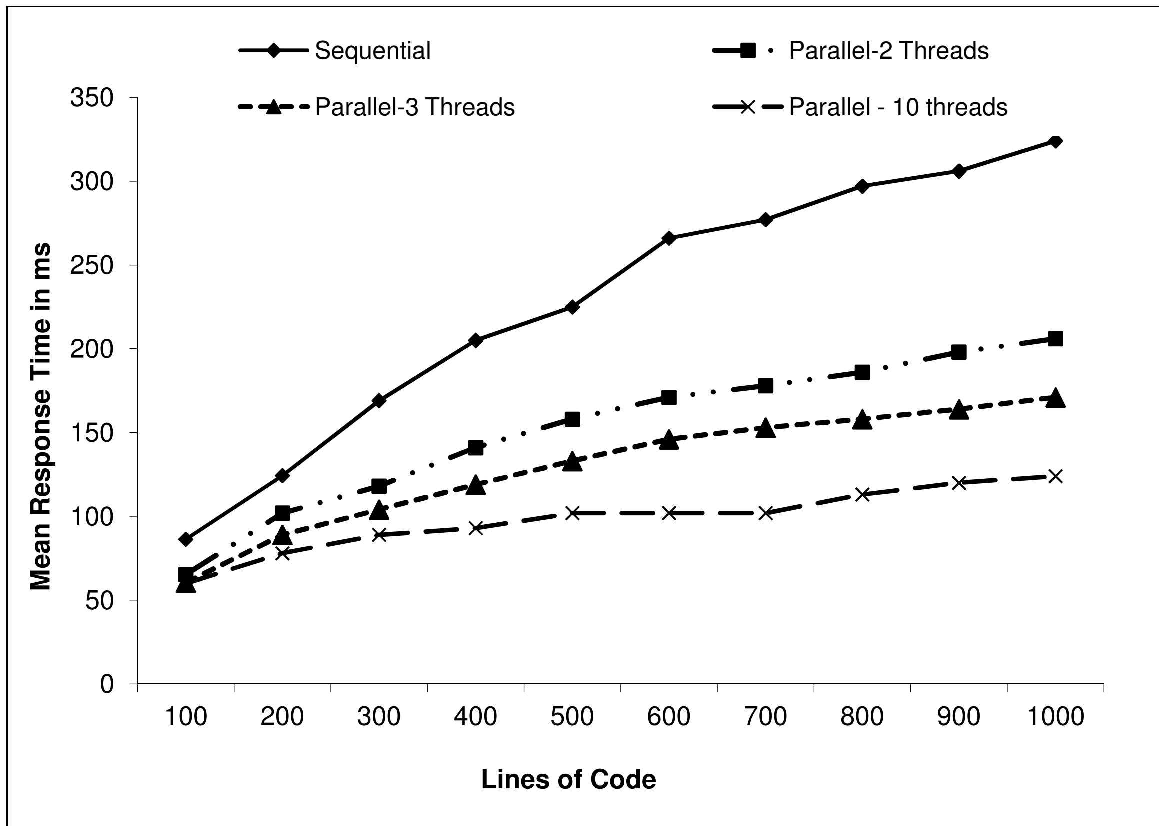

Figure 15: Response times for Atlas

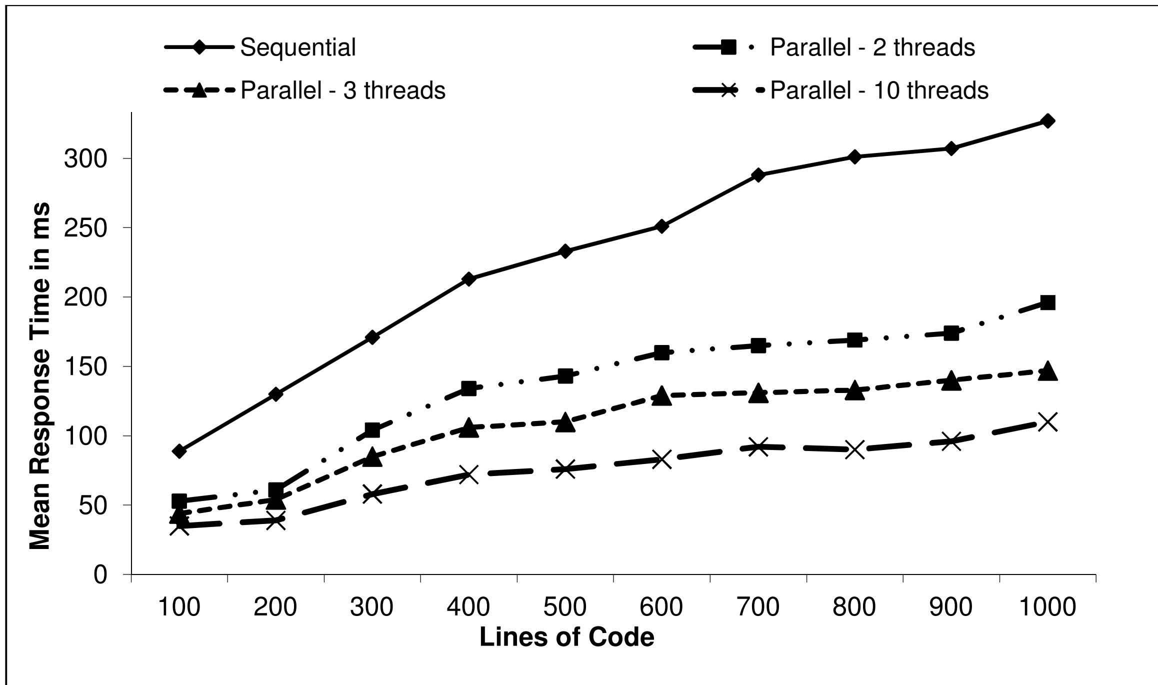

Figure 16: Response times for Atlas RMI

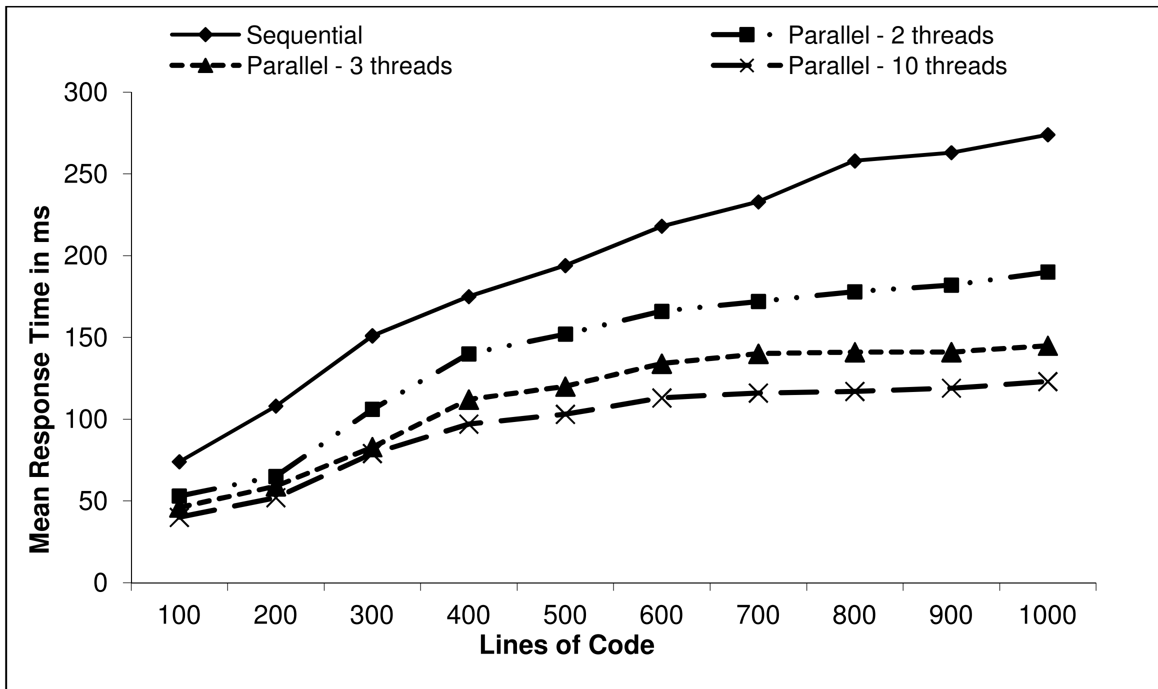

Figure 17: Response times for Uranus

5.4 Performance Comparison

Three interesting observations can be made based on the performance of the parallel compiler.

5.4.1 First observation

For smaller programs, namely programs in the range of 500 lines of code, the Atlas RMI program performs better than the Atlas program. Figure 18 shows a consolidated view of performance of all three programs on an input with 500 lines of code. As discussed in section 5.2.2, the Atlas program does not initialize workers in advance. Instead, workers go through initialization with each run of the program. The time taken to initialize workers in the Atlas program contributes towards the total response time of the program. In case of the Atlas RMI program, workers are initialized in advance. The time taken to initialize and start workers does not contribute towards the overall response time of the program. For smaller programs, this initialization time results in a significant addition to the overall response time and hence the Atlas RMI program performs better than the Atlas program.

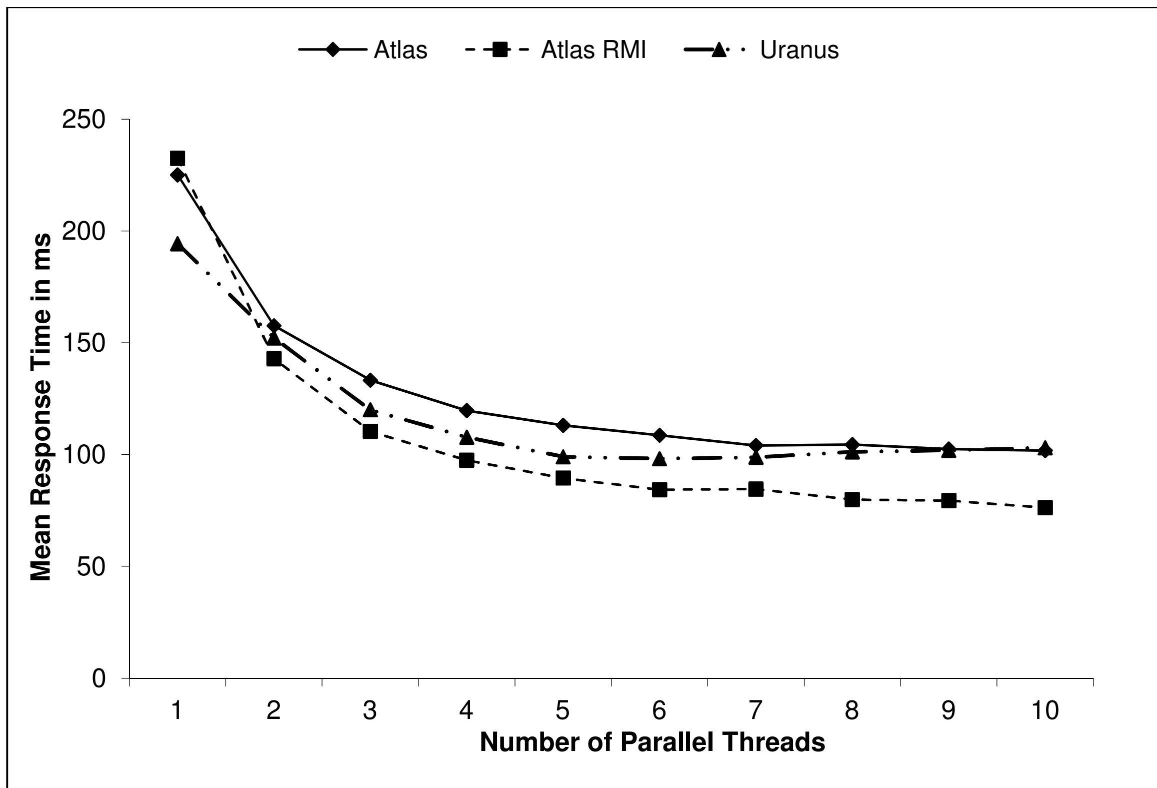

Figure 18: Comparison of response times for 500 LOC

5.4.2 Second observation

In comparison, as the size of the program increases the initialization time of workers is negligible compared to the time spent on remote method invocations. Hence, for larger programs the Atlas program performs better than the Atlas RMI program. For the parallel compiler, it was observed that inputs larger than 1000 lines of code exhibited this behavior. Figure 19 shows a consolidated view of performance of all three programs on an input with 5000 lines of code. The sequential compilation time for the Atlas and Atlas RMI program is around 880 milliseconds. When the program is run with ten threads, this compilation

time is reduced to 235 milliseconds for the Atlas program and 360 milliseconds for the Atlas RMI program.

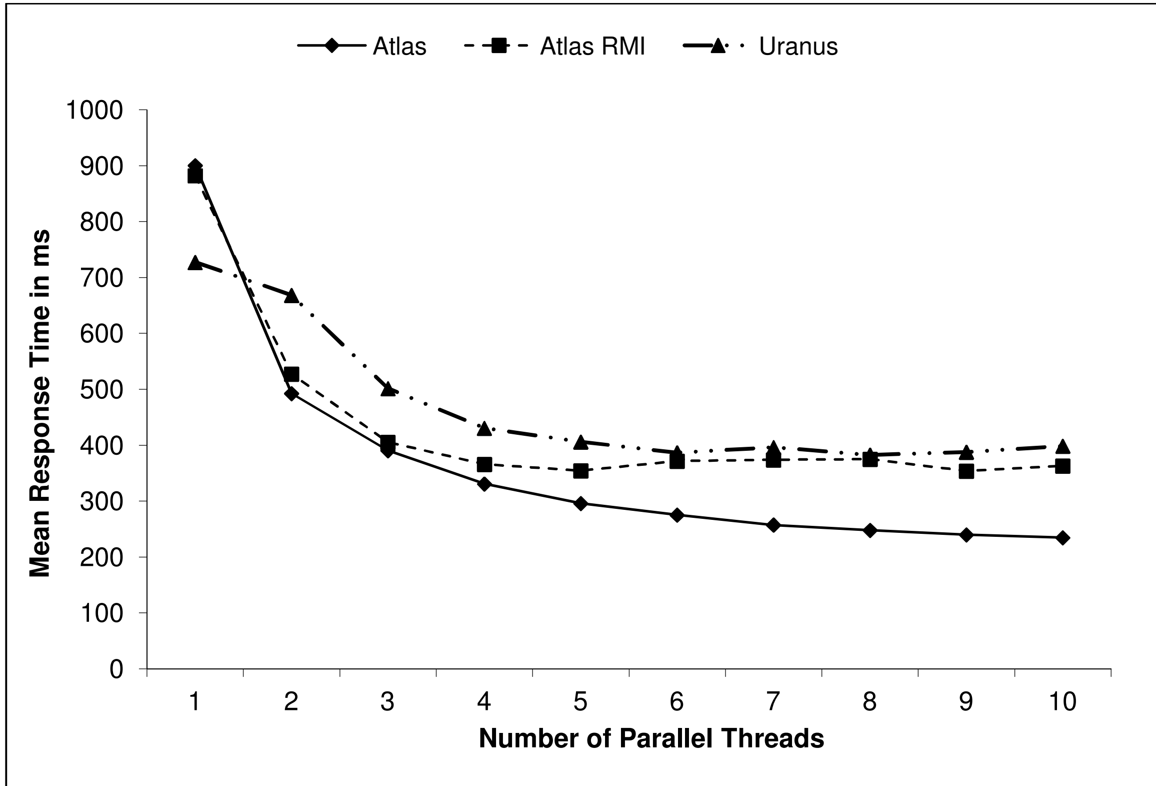

Figure 19: Comparison of response times for 5000 LOC

5.4.3 Third observation

The sequential response time for an input of any given length on Uranus is better than the sequential response time for the Atlas programs. The same cannot be said for the parallel response times. The programs for Uranus and Atlas RMI are the same. Both initialize workers in advance and communication between the master and workers is achieved

through message passing using remote method invocation. However, the speedup obtained using Atlas RMI is marginally better than that obtained using Uranus. This is true for inputs of any given size.

5.5 Statistical Significance

In order to test the statistical significance of the results obtained from the parallel compiler, a paired t test was performed. The significance level chosen was 0.05. The null hypothesis states that the response times obtained from the parallel compiler is not significantly smaller than the response times obtained from the sequential version of the compiler. The alternative hypothesis is that the response times obtained from the parallel compiler is significantly smaller than the sequential version of the compiler.

Tests 1, 2 and 3 in Table 1 below show the results of the T-test. The p-value is less than 0.05 for all three systems: Atlas, Atlas RMI and Uranus. Therefore, we reject the null hypothesis.

Tests 4 through 7 in Table 1 show the response time comparison between various systems with varying lines of code. As can be seen, the p-value is less than 0.05 in all tests demonstrating that the differences in response times are statistically significant.

Tests 8 and 9 are comparisons of response times between Uranus and Atlas RMI. The p-value is greater than 0.05 and hence it cannot be stated that the response times obtained from Atlas RMI are significantly smaller than those obtained from Uranus.

Test No	Response time comparison	p-value
1	Atlas: Sequential versus Parallel (10 threads)	0.00005
2	Atlas RMI: Sequential versus Parallel (10 threads)	0.000005
3	Uranus: Sequential versus Parallel (10 threads)	0.00001
4	Atlas versus Atlas RMI for 500 LOC	0.000084
5	Atlas versus Atlas RMI for 5000 LOC	0.001
6	Uranus versus Atlas for 500 LOC	0.005
7	Uranus versus Atlas for 5000 LOC	0.005
8	Uranus versus Atlas RMI for 500 LOC	0.058
9	Uranus versus Atlas RMI for 5000 LOC	0.115

Table 1: Statistical Significance Tests

5.6 DKY versus No DKY

During execution of the parallel compiler, if any of the parallel threads encounter a DKY situation they guess the attributes of the identifier and move on. These attributes along

with their guess information are added to a list and validated by the master eventually. Depending on the style of programming, there could be a sizeable amount of DKY attributes in this list. The master has to compare this list with the master symbol table to uncover any errors. It is important to evaluate the overhead introduced by this validation. In order to evaluate this overhead, a comparison was made between the response times obtained by an input with minimal or no DKY versus response times obtained from the same input program written such that there would be a lot of DKY identifiers. Figures 20, 21 and 22 demonstrate the results obtained from this comparison. The size of the input used for this comparison was 5000 lines of code. The results show that the overhead introduced by this validation is negligible.

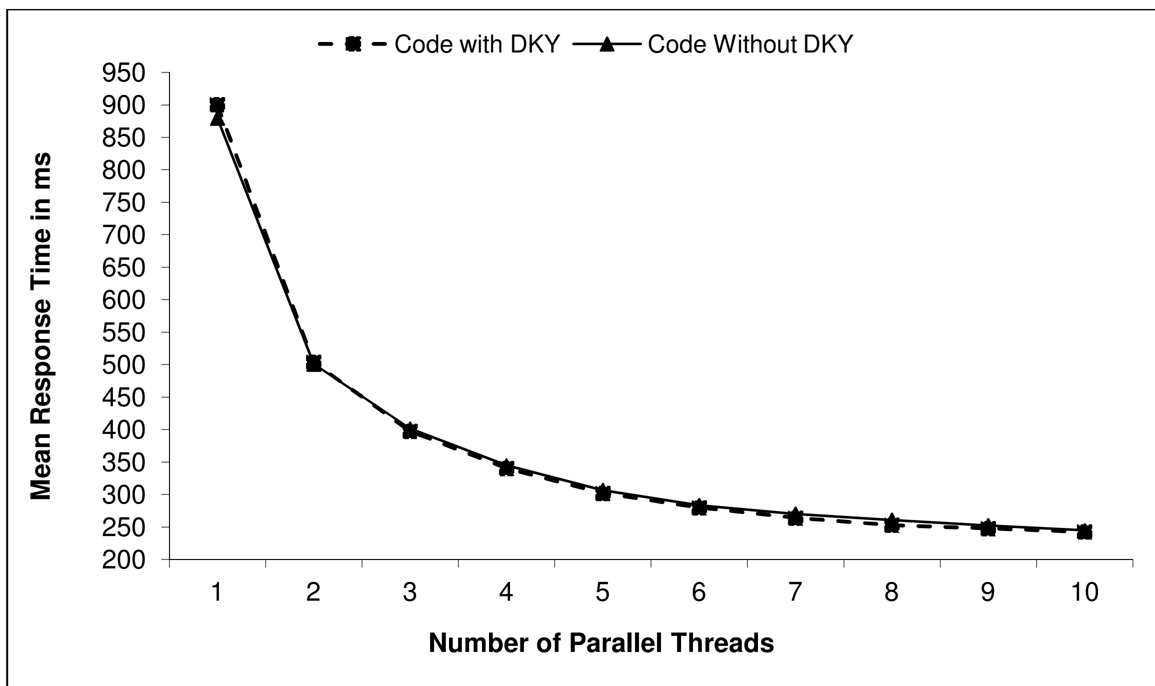

Figure 20: DKY versus No DKY for Atlas

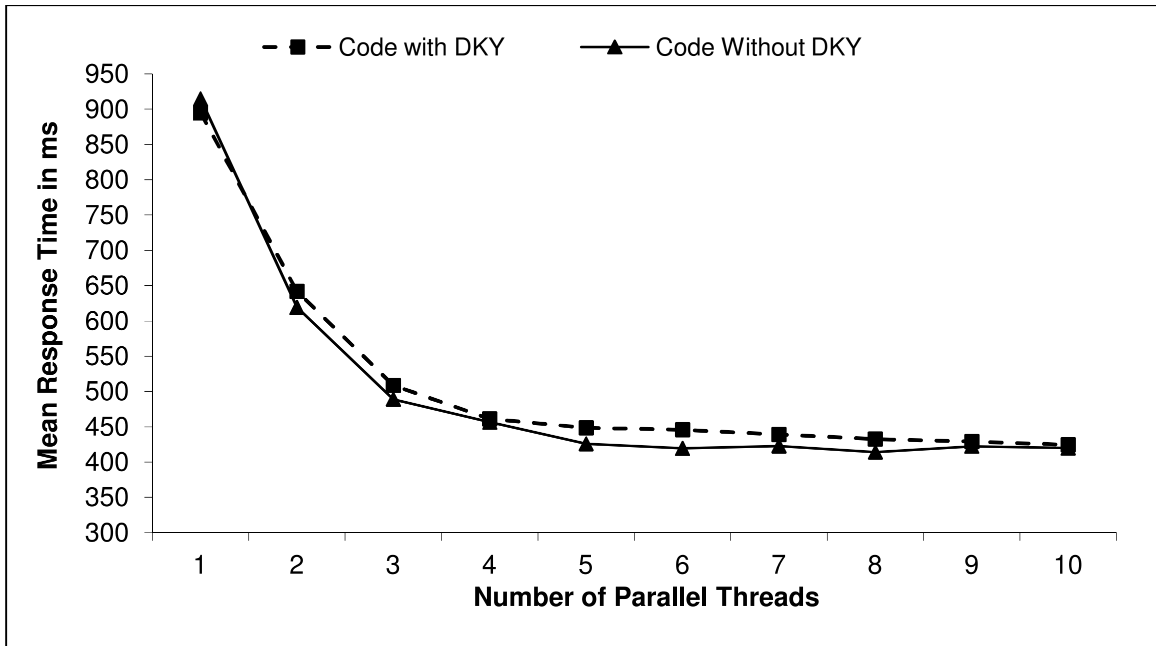

Figure 21: DKY versus No DKY for Atlas RMI

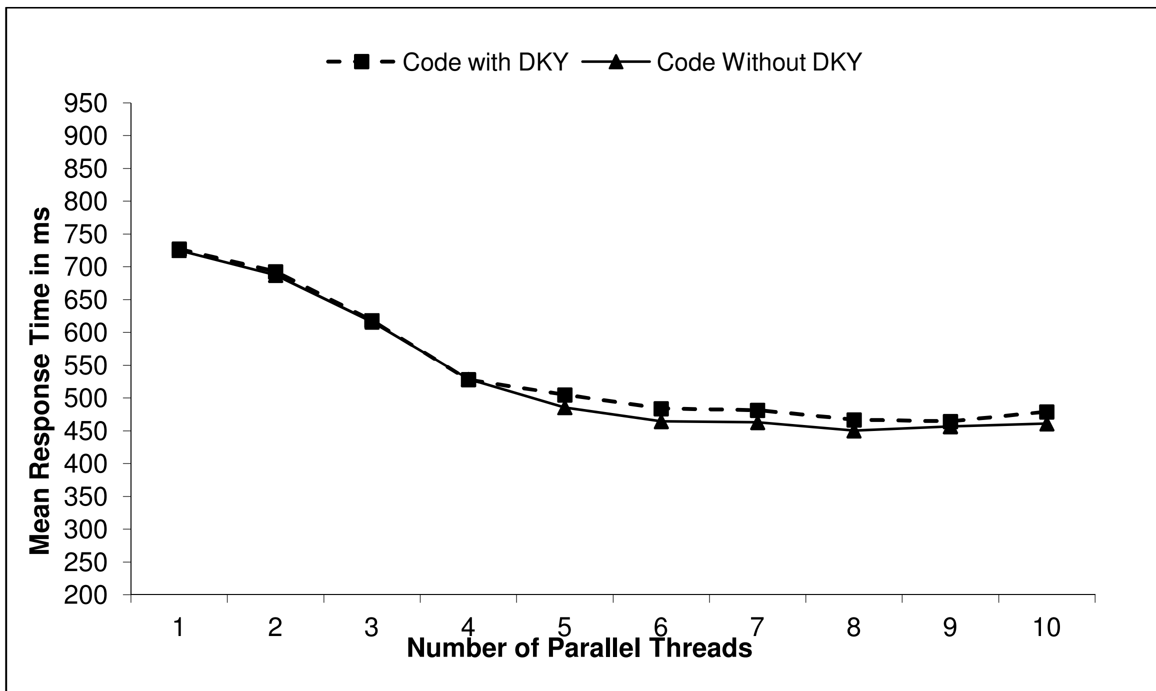

Figure 22: DKY versus No DKY for Uranus

5.7 Contributions

This section summarizes the contributions made by the research done here.

1) This research parallelizes the different phases of the compiler while preserving its correctness. The parallel compiler generates all the same compile errors as a sequential compiler. Comparison of the intermediate code generated by the parallel compiler to the intermediate code generated by the sequential compiler verified the correctness of the compiler.

2) This research compares the implementation of the parallel compiler on two different host environments namely a distributed Beowulf cluster and a shared memory multiprocessor machine. Section 5.4 discusses the results from the comparison.

3) The most significant contribution made here is the strategy of making use of the parsing technique itself to deal with symbol table management. The parallel compiler developed here deduces the attributes of an identifier using the natural order of parsing in a recursive decent parser. This guides the semantic analysis phase eliminating the need for synchronization and locking mechanisms. The overhead introduced by this strategy is negligible as proven in section 5.6. None of the previous works used the parsing algorithm itself to deal with concurrency issues.

Seshadri and Wortman in [Seshadri91] implemented three strategies for dealing with the DKY problem. Their first strategy, DKY Avoidance, involved processing all parent scopes before the child scopes. This however delays processing of the child scopes and affects the amount of parallelism that can be achieved. There is no such delayed processing with the strategy used in the current research. The parent scopes and child scopes are built in parallel. If a child scope references an identifier from a parent scope that has a DKY, it guesses the information it needs and proceeds.

The second strategy in [Seshadri91] is DKY Handling. In this approach, the child scope with a DKY is suspended until another process resolves the DKY. The drawback with this approach is that significant identifier cross usage between scopes will result in a child scope spending a lot of time waiting for another process to resolve the DKY. In the current research, there is no blocking or waiting. In addition, significant identifier cross usage in the program does not affect the speedup obtained. This has been proved in section 5.6, which showed that response times for programs with significant DKY's (resulting from identifier cross usage between scopes) does not significantly differ from the response times of programs with no DKY's.

The last strategy used in [Seshadri91] is the Hybrid approach. This involves two part semantic analysis. The program is parsed once to process all the declarations and then a second parse processes all the statements. Similar to DKY Avoidance this strategy also

affects the amount of parallelism that can be achieved. In addition, it requires an extra parse of the program. The current research parses the program only once exploiting parallelism early on.

C h a p t e r   6

CONCLUSION

Improving the speed of the first four phases of compilation allows the compiler to apply more time-consuming optimizations. Performance tests run on the parallel compiler show that substantial improvements in compilation time can be achieved using concurrency. On the parallel compiler using 10 parallel processors, the speedup achieved was 3 for smaller programs and 3.5 for larger programs. As expected, the implementation overhead prevented liner speedup. The speedup also largely depends on the number of functions in the input and on the size of these functions. For an input program that has many small functions in it, the overhead of delegating each of these functions to the workers proves to be costly. This overhead can decrease the speedup obtained.

The technique used for splitting the program and parallel processing of individual functions neither introduced nor masked any syntax or semantic errors. The strategy of guessing identifiers proved effective and the overhead introduced by it is negligible. The natural order of parsing in a recursive descent parser guides this guess strategy. This strategy avoided the need to use extensive synchronization techniques and locking mechanisms on the symbol table. This strategy addressed the limitations of the DKY Handling approach used in [Seshadri91]. It avoids the need for one process to wait for another to finish. It also saved the time needed for an extra parse of the input. This overcomes the shortcomings of the hybrid approach used in the [Seshadri91]. Section 5.4 discusses the performance of the

parallel compiler on a shared memory multiprocessor versus a thirteen-node Beowulf cluster. The parallel compiler performs best on the shared memory multiprocessor machine.

Chapter 7

FUTURE WORK

The parallel compiler does no computation up front to see if the program is a good candidate for parallelism. Is it worthwhile to parallelize the given input? A reliable estimation of the input workload is needed to answer this question. The size of the program alone is not enough to make this decision. Some of the other factors that can influence this decision are:

- The size of the functions within the input. If the function is small then more time is spent in the parallelization overhead than in doing the actual work. This could negate the performance improvements of a parallel compiler.
- Capacity of the parallel system.

Scheduling of tasks and processor assignment are also good candidates for improvement. Currently tasks are scheduled on different processors using a simple first come first serve strategy. As discussed previously, if the size of the function is small parallel execution could take more time than its sequential counterpart could. A better approach would be to group together all the small functions and process them together using one processor.

REFERENCES

Print Publications:

[Cohen85]
J. Cohen and S. Kolodner, "Estimating the Speedup in Parallel Parsing," Software Engineering, IEEE Transactions on, vol. SE-11, pp. 114-124, 1985.

[Gross89]
T. Gross, A. Sobel, and M. Zolg, “Parallel compilation for a parallel machine,” In Proceedings of the ACM SIGPLAN 1989 Conference on Programming language design and implementation (PLDI '89), ACM, New York, NY, USA, 91-100.

[Kumar11]
P.J.Satish Kumar , M. Rajesh Khanna, H. Shine and S. Arun, “Implementing High Performance Lexical Analyzer using CELL Broadband Engine Processor, ” International Journal of Engineering Science and Technology, vol. 3, pp. 6907-6913, 2011.

[Louden97]
Louden. C. Kenneth, Compiler Construction Principles and Practice, Publication Date: January 24, 1997 , Edition 1.

[Seshadri88]
V. Seshadri, S. Weber, D. B. Wortman, C. P. Yu, and I. Small, “Semantic analysis in a concurrent compiler,” In Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation (PLDI '88), ACM, New York, NY, USA, 233-240.

[Seshadri91]
V. Seshadri and D. B. Wortman, “An investigation into concurrent semantic analysis,” Softw. Pract. Exper. 21, 12 (December 1991), 1323-1348.

[Srikanth10]
G. U. Srikanth, "Parallel lexical analyzer on the cell processor," in Secure Software Integration and Reliability Improvement Companion (SSIRI-C), 2010 Fourth International Conference on, 2010, pp. 28-29.

[Wortman92]
David B. Wortman and Michael D. Junkin, “A concurrent compiler for Modula-2+,” In Proceedings of the ACM SIGPLAN 1992 conference on Programming language design and implementation (PLDI '92), ACM, New York, NY, USA, 68-81.

Electronic Sources:

[Golub09]
M. Golub and D. Jakobović, “An Overview of Distributed Programming Techniques”, http://www.zemris.fer.hr/~yeti/download/MIPRO05.pdf, August 2009, last accessed August 2, 2012.

APPENDIX A

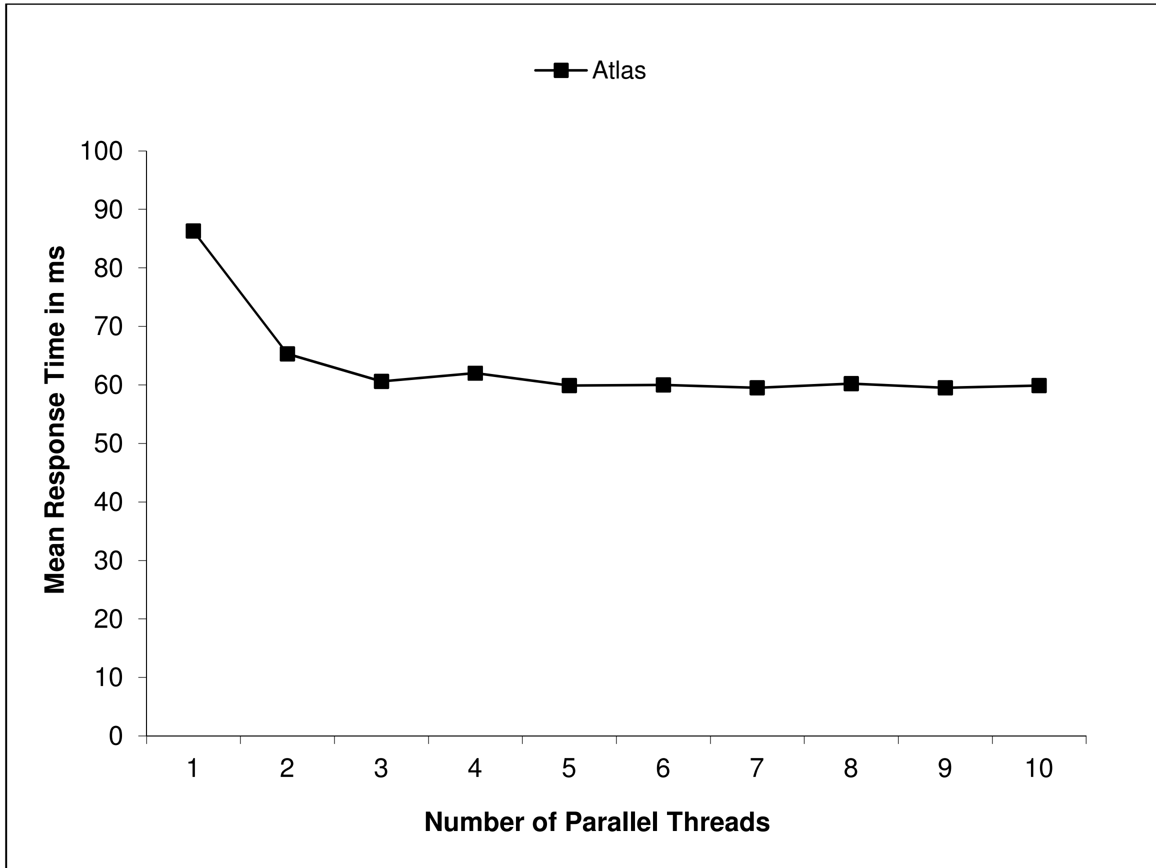

Figure 23: Response times for 100 LOC using Atlas

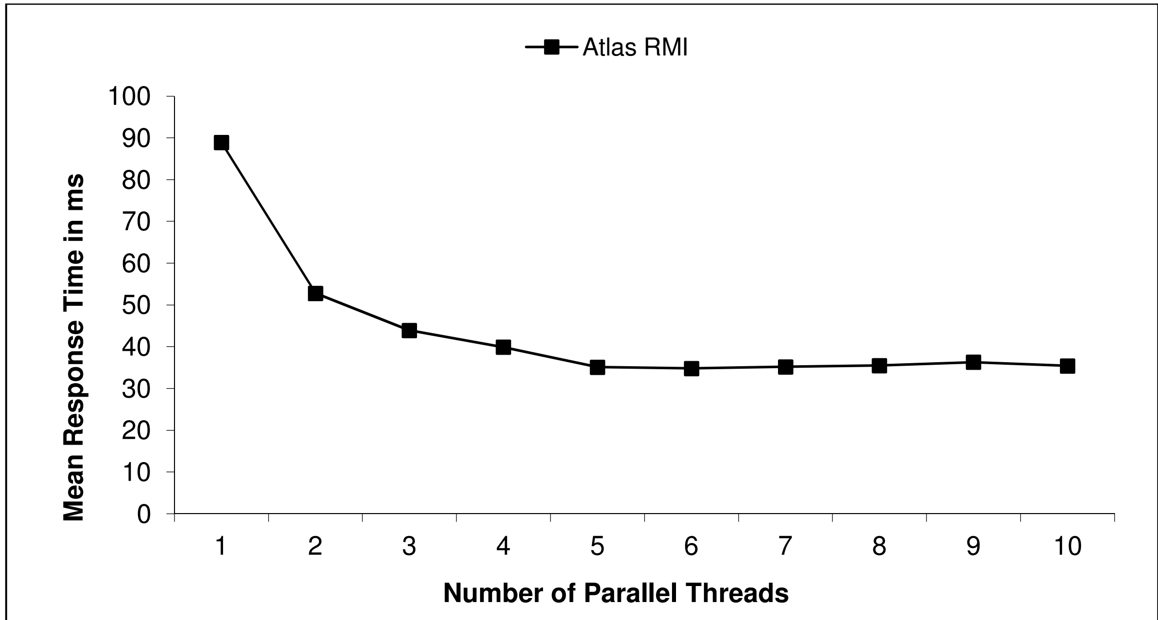

Figure 24: Response times for 100 LOC using Atlas RMI

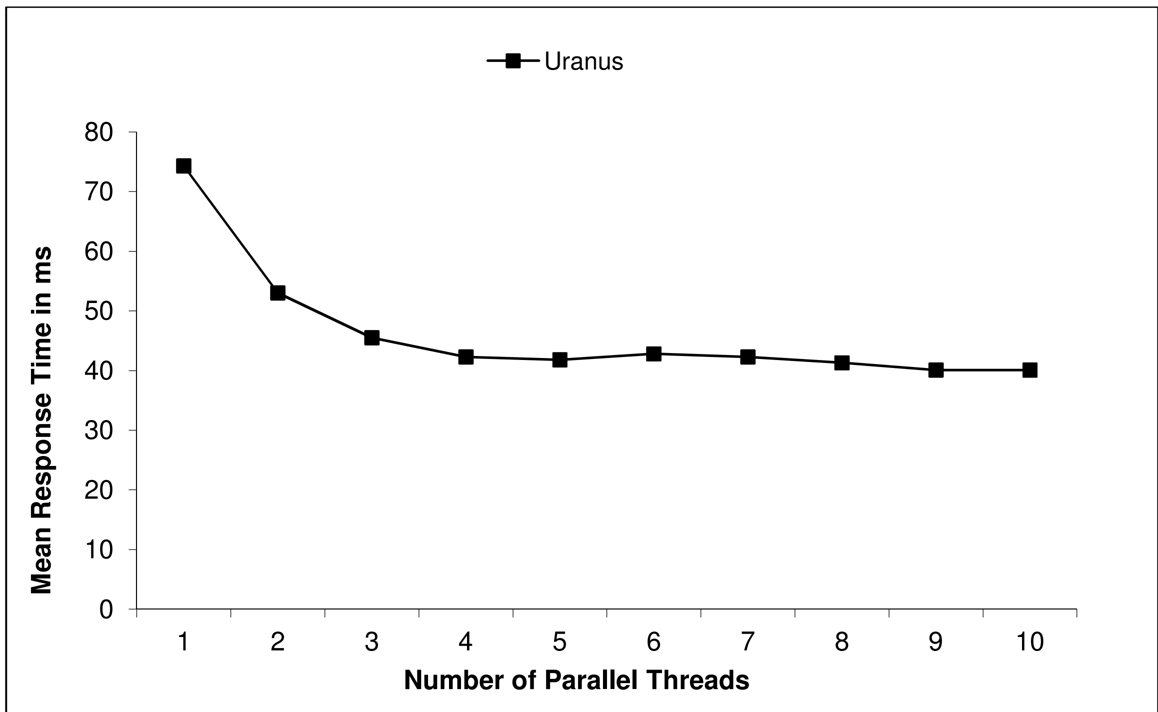

Figure 25: Response times for 100 LOC using Uranus

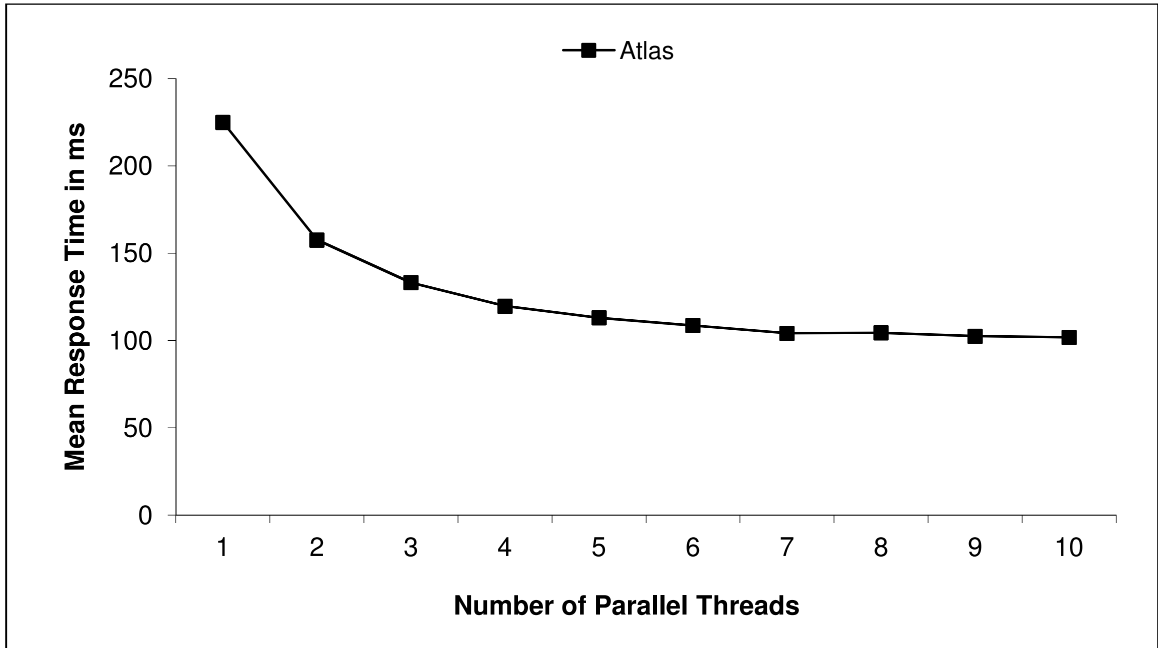

Figure 26: Response times for 500 LOC using Atlas

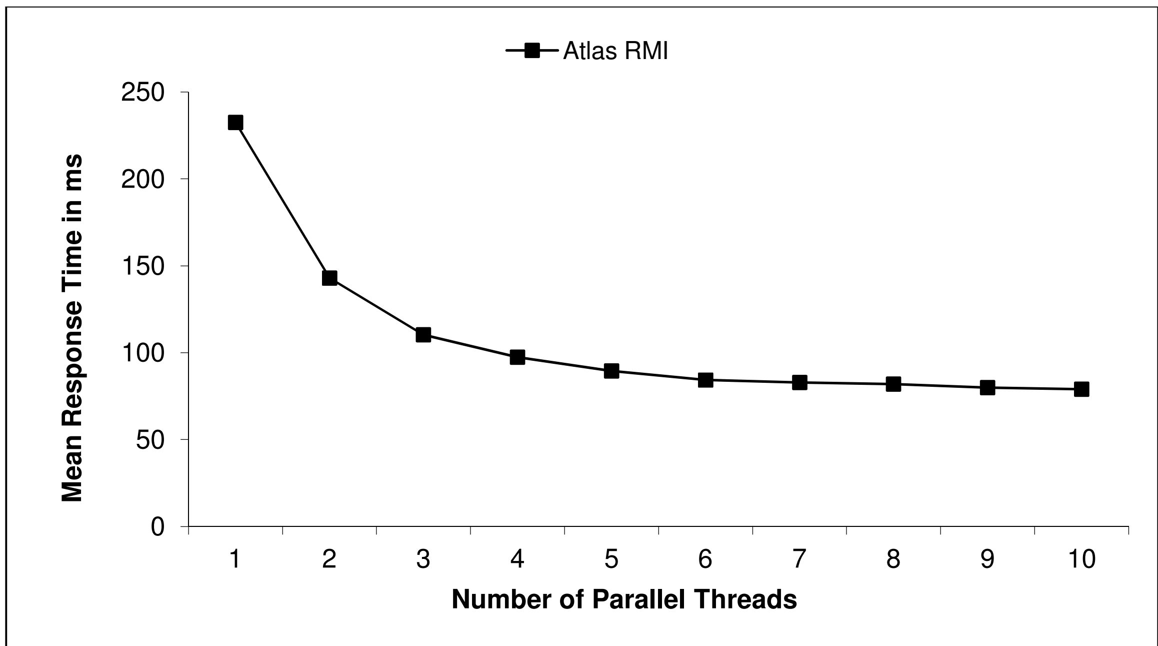

Figure 27: Response times for 500 LOC using Atlas RMI

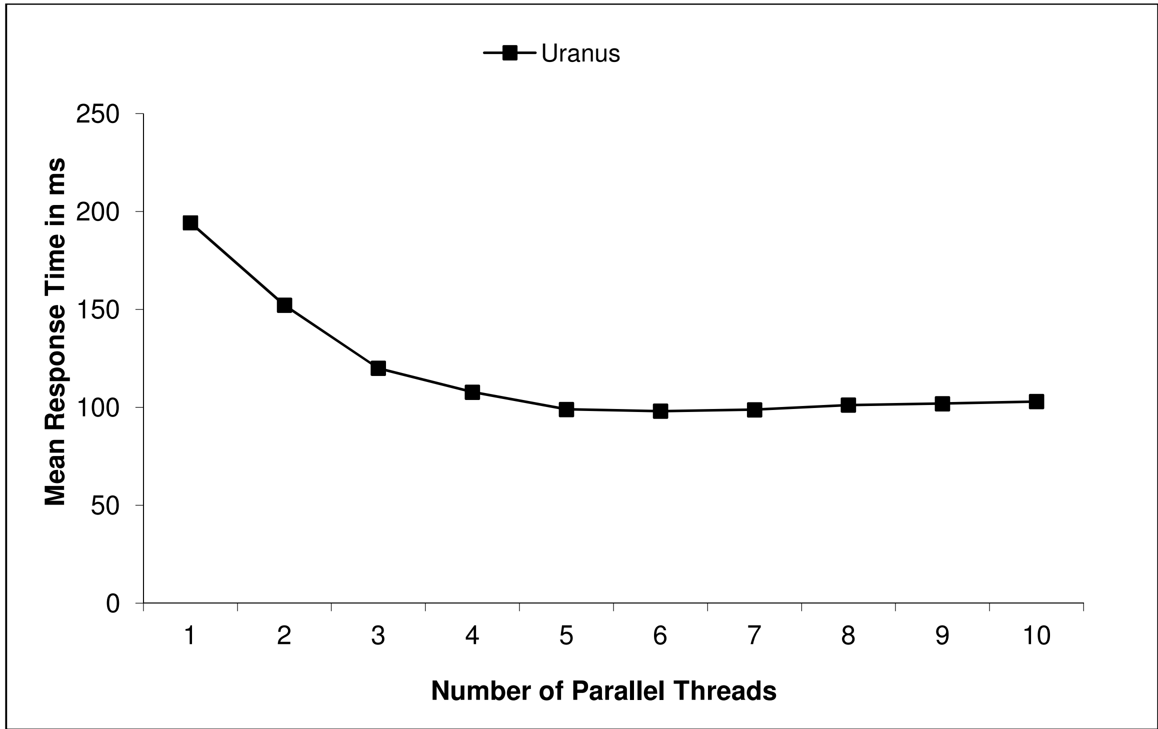

Figure 28: Response times for 500 LOC using Uranus

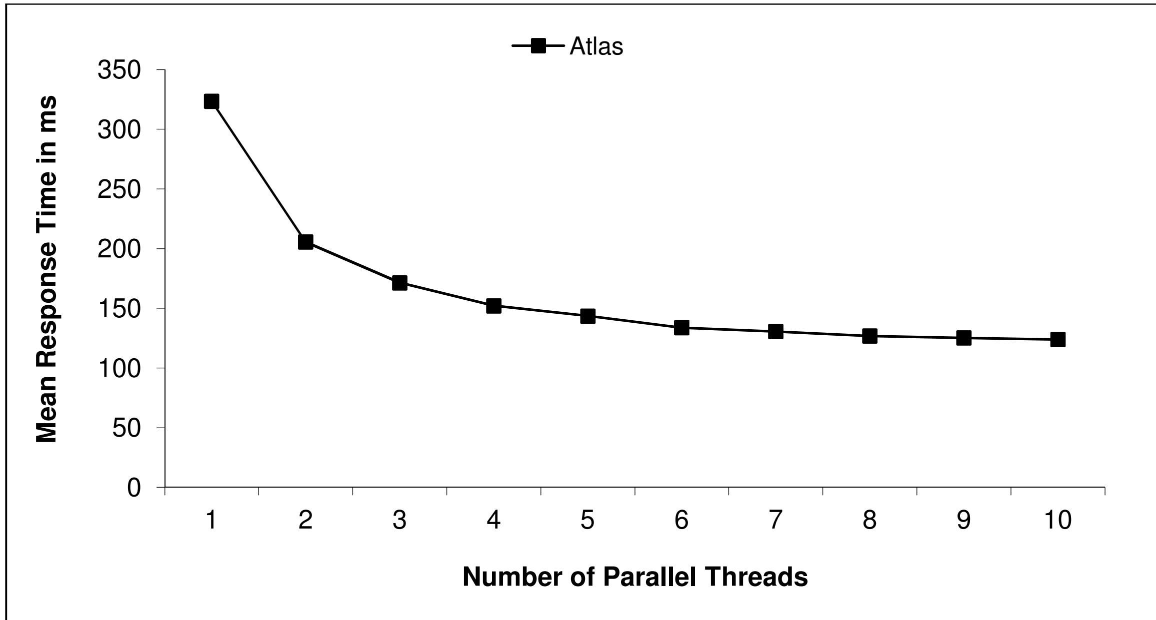

Figure 29: Response times for 1000 LOC using Atlas

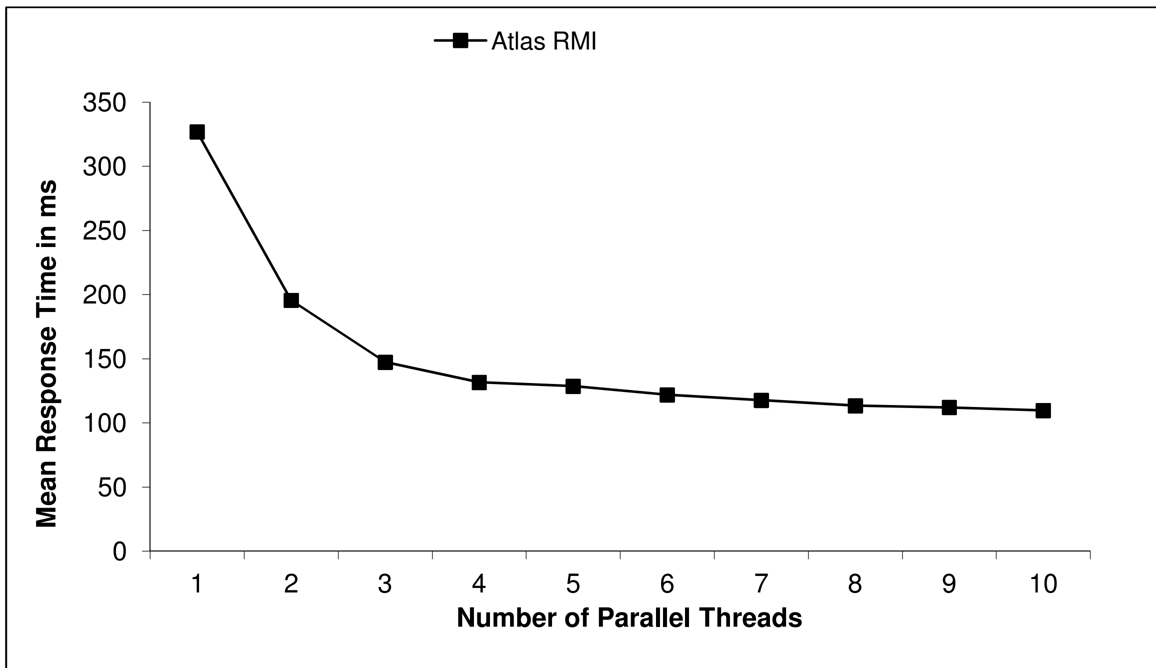

Figure 30: Response times for 1000 LOC using Atlas RMI

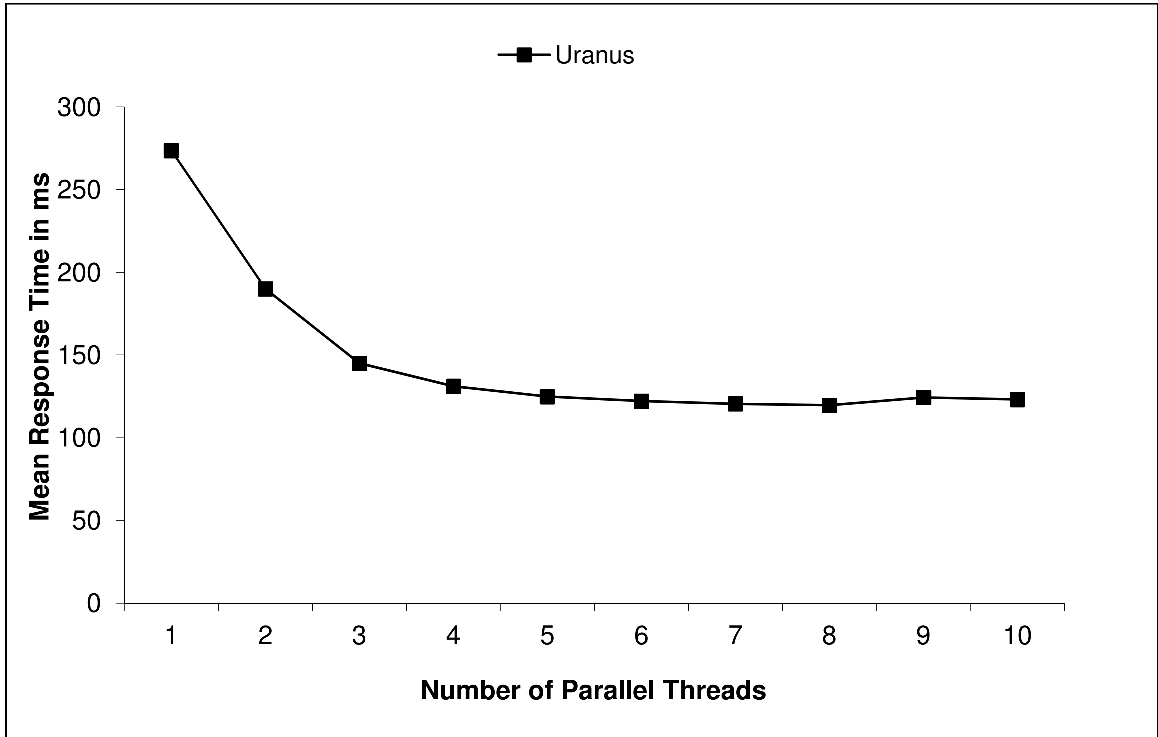

Figure 31: Response times for 1000 LOC using Uranus

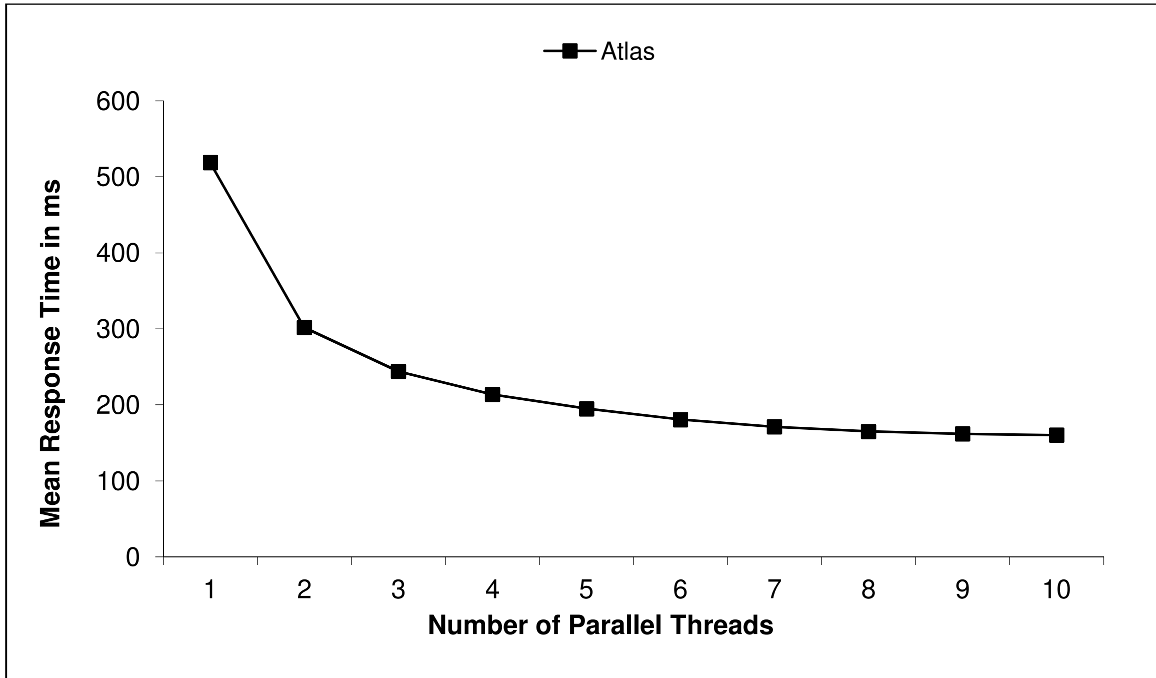

Figure 32: Response times for 2000 LOC using Atlas

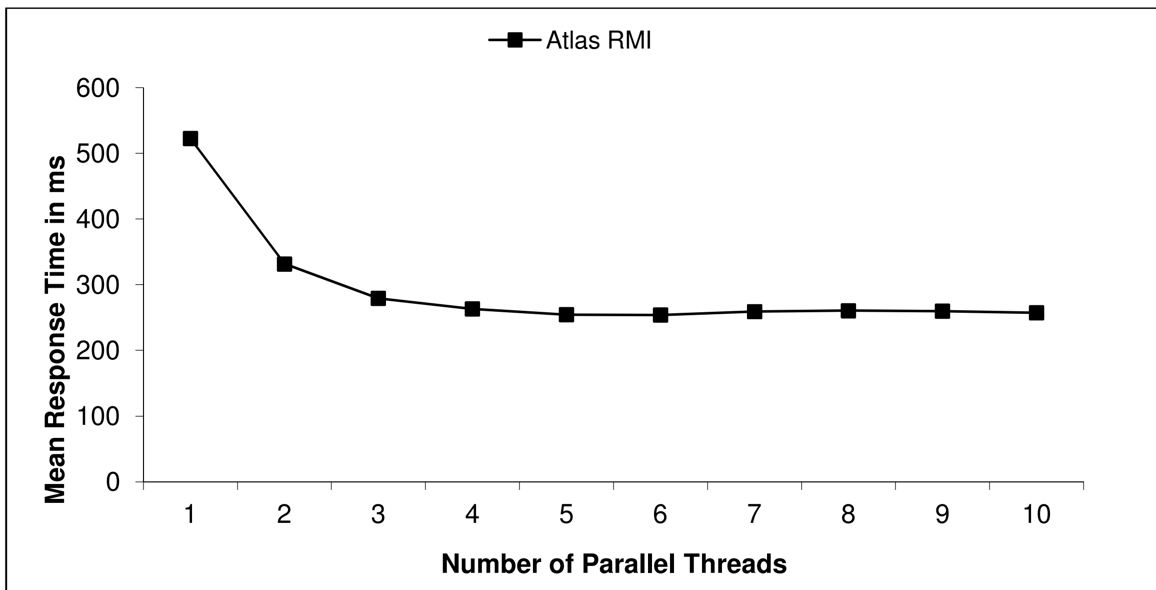

Figure 33: Response times for 2000 LOC using Atlas RMI

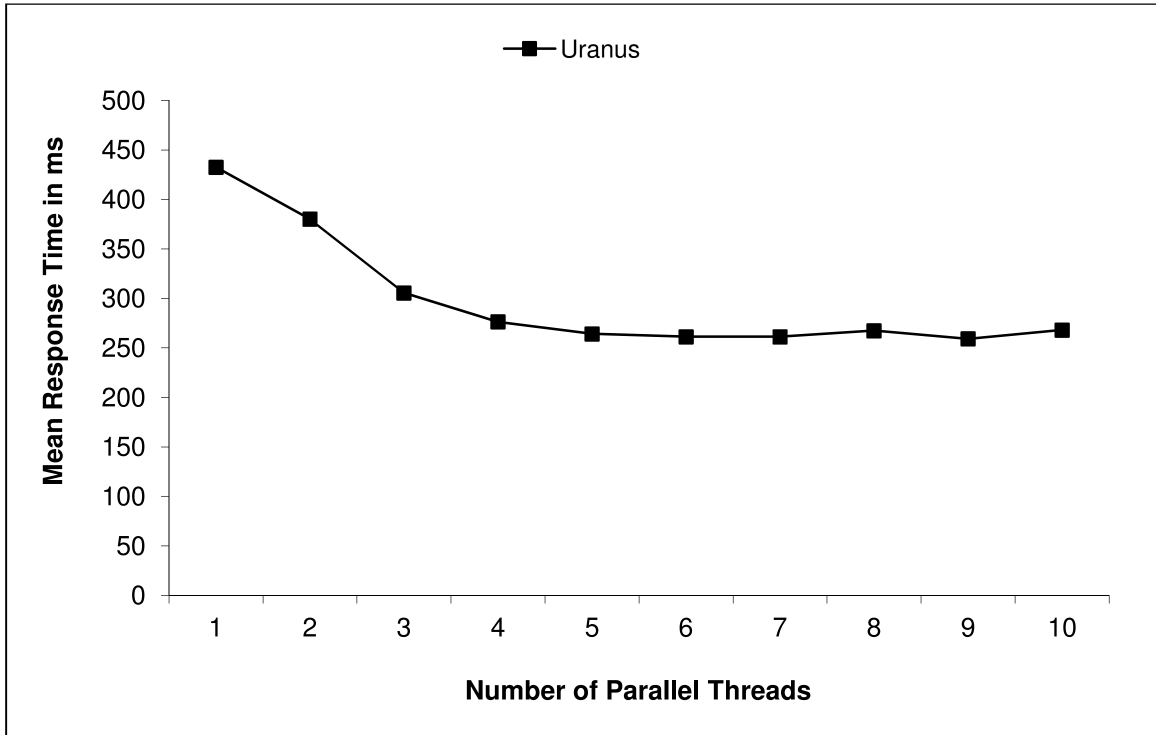

Figure 34: Response times for 2000 LOC using Uranus

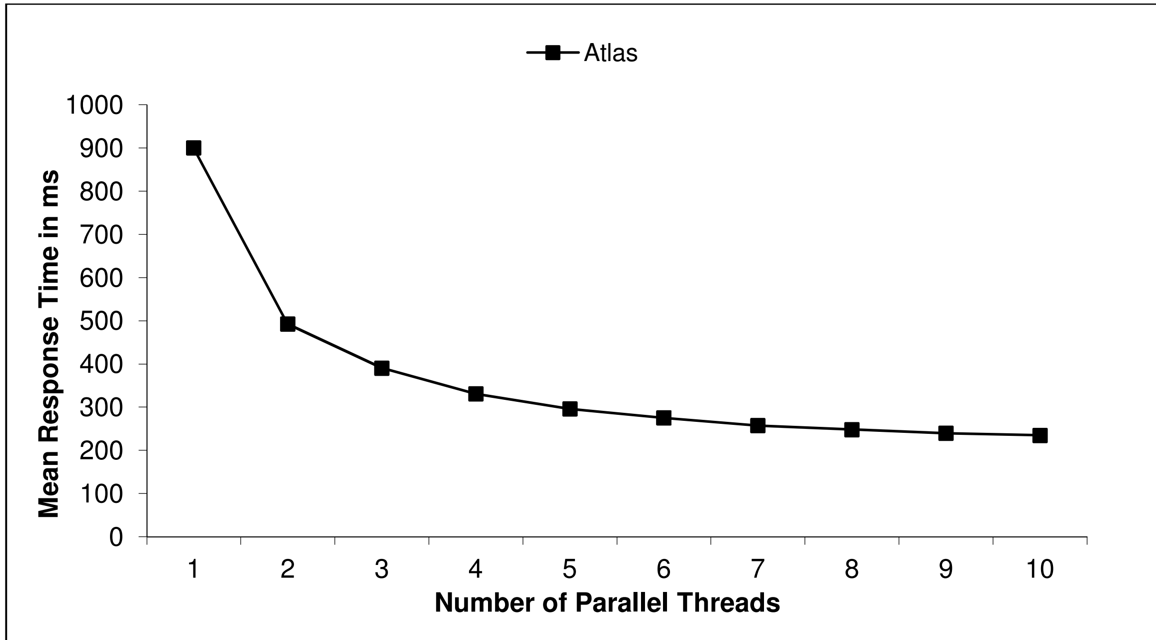

Figure 35: Response times for 5000 LOC using Atlas

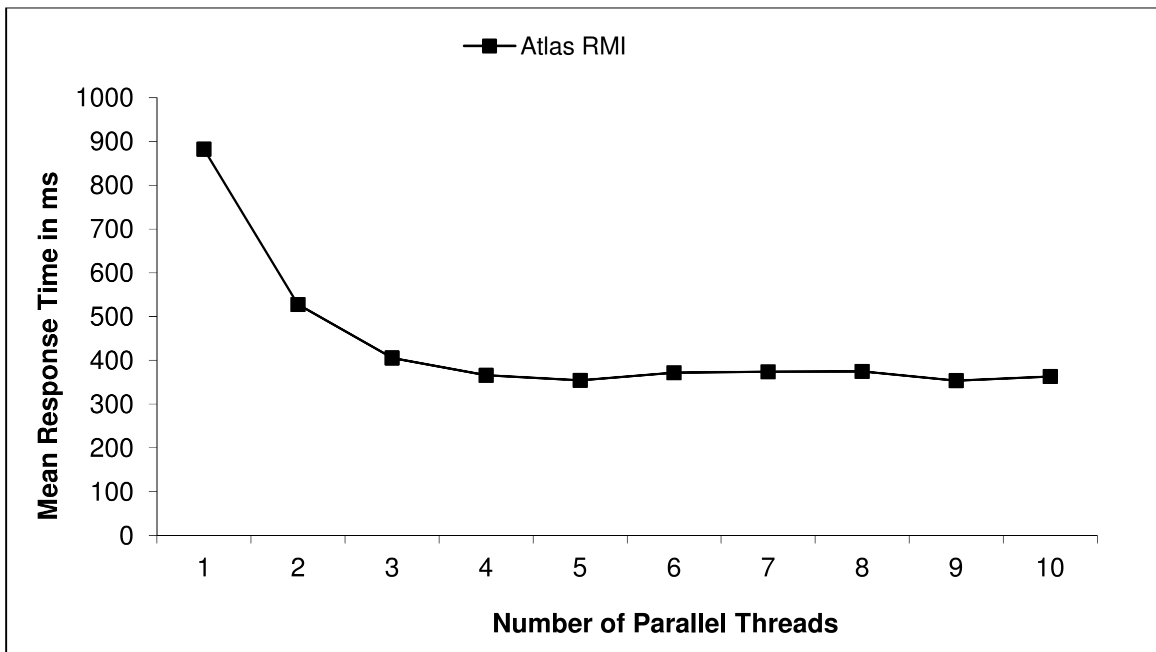

Figure 36: Response times for 5000 LOC using Atlas RMI

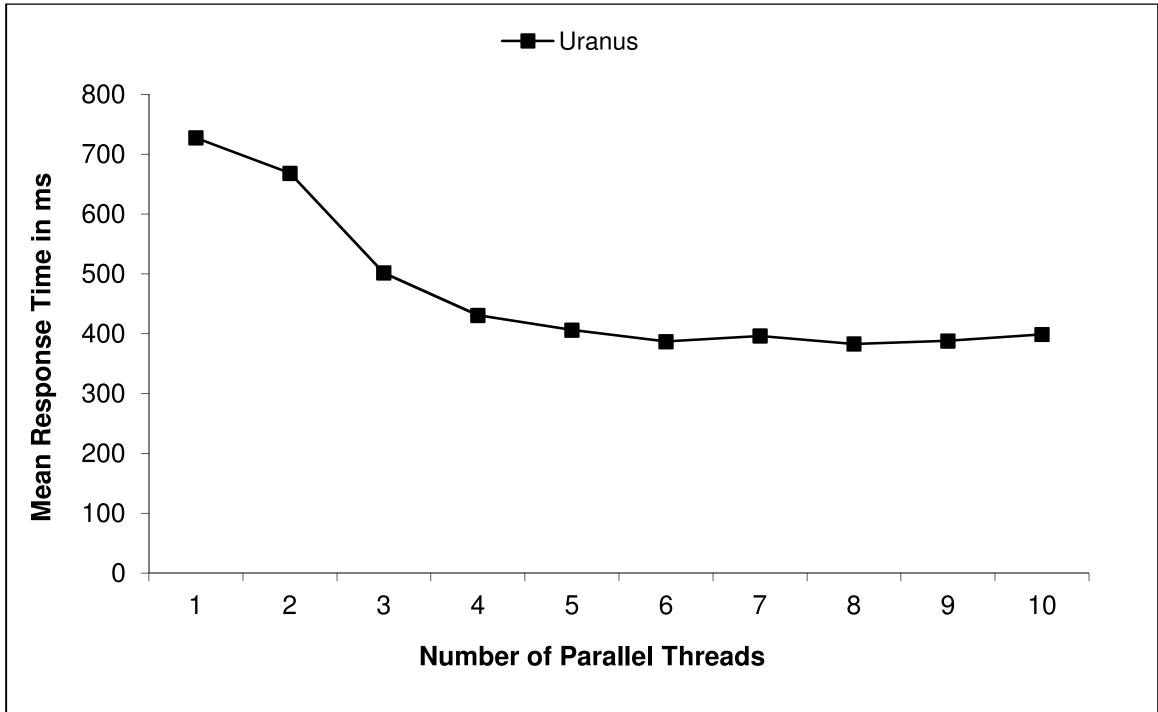

Figure 37: Response times for 5000 LOC using Uranus

VITA

Deepa Viswanathan Komathukattil received the Bachelor of Engineering degree in Electronics Engineering from Pune University, India. She has a Post Graduate Diploma in Advanced Computing from Center for Development of Advanced Computing, Advanced Computing Training School Pune, India. She expects to receive a Masters of Science in Computer and Information Science from the University of North Florida in December 2012.

During 2011 – 2012, she served as the President of Upsilon Pi Epsilon (UPE) chapter at University of North Florida, which is an ACM/IEEE-CS International Honor Society for the Computing and Information disciplines. In 2012, she received a UPE Special Recognition award. She has worked for a number of years in designing and developing enterprise applications. Her research interests are in the area of compilers and natural language processing.

Journal Publications

Ahuja, Sanjay, P., Komathukattil, Deepa., A Survey of the State of Cloud Security, Journal of Network and Communication Technologies (NCT), Volume 1, No. 2, December 2012.