Dynamic Loop Nesting in Shared Memory Programming

Orestis Agathokleous

August 18, 2011

MSc in High Performance Computing
The University of Edinburgh
Year of Presentation: 2011
Abstract

Regions of nested loops are a common feature of High Performance Computing (HPC) codes. In shared memory programming models, such as OpenMP, these structures are the most common source of parallelism. Parallelising these structures requires programmers to make a static decision on how parallelism should be applied. However, depending on the parameters of the problem as well as the nature of the code, a static choice cannot exploit any runtime characteristics of the execution. Changes to the iterations of the loop which is chosen to be parallelised might limit the amount of processors that can be utilised.

This work investigates the possibility of applying code transformations before the compilation of the code in order to enable a dynamic choice to be made at runtime on how parallelisation should be applied on the nested loops. We have developed a source to source compiler to perform the transformations automatically. We use a directive based approach, similar to OpenMP, where programmers are required to specify how the loops of the region can be parallelised. Our runtime library however is responsible for making the decisions dynamically during the execution of the code.
# Contents

1 Introduction

2 Background theory
   2.1 OpenMP Programming Model
   2.1.1 Execution Model
   2.1.2 Compiler Directives
   2.1.3 Runtime Library
   2.2 Parallelisation Strategies For Nested Loops
   2.2.1 Outermost loop
   2.2.2 Inner loop
   2.2.3 Nested
   2.2.4 Loop collapsing
   2.3 Source-To-Source Compilers
   2.4 The Lua Programming Language
   2.5 The Lpeg Parsing Library
   2.5.1 Construction of Parsers

3 Design And Implementation Of Compiler
   3.1 High Level Overview
   3.2 Custom Directives
   3.3 Code Transformations Of Loop Regions
   3.3.1 Loop Analysis
   3.3.2 Using OpenMP If Clause
   3.3.3 Using Code duplication
   3.4 Code Design
   3.4.1 Module base.Class
   3.4.2 Module base.Driver
   3.4.3 Class base.AstNode
   3.4.4 Class base.Grammar
   3.4.5 Class language.c.C99
   3.4.6 Class language.c.PreOMP
   3.4.7 Class Container language.c.C99.Ast
   3.4.8 Class language.c.C99.CodeGen
   3.5 Driver
   3.6 Testing
<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>Runtime Library</td>
<td>4.1 Decision Based On Heuristics</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4.2 Decision Based On Heuristics With Profiling</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4.3 Code Design</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4.3.1 Public Interface</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4.3.2 Private Interface</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4.4 Testing</td>
</tr>
<tr>
<td>5</td>
<td>Experimental Design</td>
<td>5.1 The System</td>
</tr>
<tr>
<td></td>
<td></td>
<td>5.2 The Test Codes</td>
</tr>
<tr>
<td></td>
<td></td>
<td>5.2.1 Synthetic Benchmark Code</td>
</tr>
<tr>
<td></td>
<td></td>
<td>5.2.2 Computational Fluid Dynamics Extract Code</td>
</tr>
<tr>
<td></td>
<td></td>
<td>5.3 Compiler Flags</td>
</tr>
<tr>
<td></td>
<td></td>
<td>5.4 Job Submission And Collection Of Results</td>
</tr>
<tr>
<td>6</td>
<td>Synthetic Benchmark</td>
<td>6.1 Parameters</td>
</tr>
<tr>
<td></td>
<td></td>
<td>6.2 Results</td>
</tr>
<tr>
<td></td>
<td></td>
<td>6.3 Discussion</td>
</tr>
<tr>
<td>7</td>
<td>CFD Extract Code Benchmark</td>
<td>7.1 Comparison Of Code Generation Modes</td>
</tr>
<tr>
<td></td>
<td></td>
<td>7.1.1 Parameters</td>
</tr>
<tr>
<td></td>
<td></td>
<td>7.1.2 Results</td>
</tr>
<tr>
<td></td>
<td></td>
<td>7.1.3 Discussion</td>
</tr>
<tr>
<td></td>
<td></td>
<td>7.2 Comparison Of Decision Functions</td>
</tr>
<tr>
<td></td>
<td></td>
<td>7.2.1 Parameters</td>
</tr>
<tr>
<td></td>
<td></td>
<td>7.2.2 Test Case: 4 Blocks, 4 Harmonics (2 n_harms)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>7.2.3 Test Case: 4 Blocks, 8 Harmonics (4 n_harms)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>7.2.4 Test Case: 8 Blocks, 4 Harmonics (2 n_harms)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>7.2.5 Discussion</td>
</tr>
<tr>
<td>8</td>
<td>Relaxed Version Of Decision Function With Profiling</td>
<td>8.1 Results</td>
</tr>
<tr>
<td></td>
<td></td>
<td>8.1.1 Test Case: 4 Blocks, 4 Harmonics (2 n_harms)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>8.1.2 Test Case: 8 Blocks, 4 Harmonics (2 n_harms)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>8.1.3 Test Case: 15 Blocks, 14 Harmonics (7 n_harms)</td>
</tr>
<tr>
<td>9</td>
<td>Conclusions and Future Work</td>
<td>9.1 Conclusions</td>
</tr>
<tr>
<td></td>
<td></td>
<td>9.2 Future Work</td>
</tr>
<tr>
<td>A</td>
<td>Example Outputs From Source-To-Source Compiler</td>
<td>A.1 Using The If-Clause mode</td>
</tr>
</tbody>
</table>

**ii**
# List of Tables

2.1 OpenMP conformance rules for loop statements ........................................ 7
2.2 Alternative strategies for parallelising nested loop regions .................... 8
2.3 Lpeg functions for creating patterns ....................................................... 13
2.4 Lpeg operator for pattern repetition ....................................................... 14
2.5 Lpeg operators for pattern composition ................................................... 15
2.6 Lpeg operators for syntactic predicates .................................................. 16
2.7 Summary of useful captures with Lpeg ..................................................... 17
3.1 Extracted loop information for calculating the number of its iterations .... 26
3.2 base.Class Module fields ................................................................. 30
3.3 base.Driver Module fields .................................................................... 30
3.4 base.AstNode Class fields ................................................................. 30
3.5 base.Grammar Class functions .............................................................. 31
3.6 language.c.C99 Class fields ............................................................... 32
3.7 language.c.PreOMP Class fields ........................................................... 33
3.8 language.c.C99.Ast Class Container fields ............................................. 33
3.9 language.c.C99.CodeGen Class fields ....................................................... 34
3.10 Driver Command Line Options ............................................................. 34
6.1 Execution time required by the delay function for various iterations .... 54
6.2 Execution time required by the delay function for various iterations .... 54
7.1 CFD Extract Code Parameters For The If Clause ................................... 59
7.2 Overheads of GCC’s nested parallelism .................................................... 67
7.3 CFD Extract Code parameters when comparing the two decision functions 67
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Fork-Join parallel execution model with four threads</td>
<td>5</td>
</tr>
<tr>
<td>3.1</td>
<td>Compilation process with the source-to-source compiler being involved</td>
<td>24</td>
</tr>
<tr>
<td>3.2</td>
<td>Example of using the if clause to parallelise (a) the outer and (b) the</td>
<td>26</td>
</tr>
<tr>
<td></td>
<td>inner loop of two nested loops with two threads</td>
<td></td>
</tr>
<tr>
<td>3.3</td>
<td>Representation of a nested loop region with code duplication</td>
<td>28</td>
</tr>
<tr>
<td>3.4</td>
<td>UML diagram of the source-to-source compiler components</td>
<td>29</td>
</tr>
<tr>
<td>4.1</td>
<td>Example of the Heuristics With Profiling Decider with three loops</td>
<td>39</td>
</tr>
<tr>
<td>6.1</td>
<td>Synthetic Benchmark With Outer Loop Delay : 0s</td>
<td>55</td>
</tr>
<tr>
<td>6.2</td>
<td>Synthetic Benchmark With Outer Loop Delay : 0.022s</td>
<td>55</td>
</tr>
<tr>
<td>6.3</td>
<td>Synthetic Benchmark With Outer Loop Delay : 0.041s</td>
<td>56</td>
</tr>
<tr>
<td>6.4</td>
<td>Synthetic Benchmark With Outer Loop Delay : 0.079s</td>
<td>56</td>
</tr>
<tr>
<td>6.5</td>
<td>Synthetic Benchmark With Outer Loop Delay : 0.15s</td>
<td>57</td>
</tr>
<tr>
<td>7.1</td>
<td>Comparison of code generation modes - 4 blocks, 4 harmonics, 1 repetition,</td>
<td>60</td>
</tr>
<tr>
<td></td>
<td>0 alterations</td>
<td></td>
</tr>
<tr>
<td>7.2</td>
<td>Comparison of code generation modes - 4 blocks, 4 harmonics, 1 repetition,</td>
<td>61</td>
</tr>
<tr>
<td></td>
<td>1 alteration</td>
<td></td>
</tr>
<tr>
<td>7.3</td>
<td>Comparison of code generation modes - 4 blocks, 4 harmonics, 1 repetition,</td>
<td>61</td>
</tr>
<tr>
<td></td>
<td>2 alterations</td>
<td></td>
</tr>
<tr>
<td>7.4</td>
<td>Comparison of code generation modes - 4 blocks, 4 harmonics, 5 repetitions,</td>
<td>62</td>
</tr>
<tr>
<td></td>
<td>0 alterations</td>
<td></td>
</tr>
<tr>
<td>7.5</td>
<td>Comparison of code generation modes - 4 blocks, 4 harmonics, 5 repetitions,</td>
<td>63</td>
</tr>
<tr>
<td></td>
<td>1 alteration</td>
<td></td>
</tr>
<tr>
<td>7.6</td>
<td>Comparison of code generation modes - 4 blocks, 4 harmonics, 5 repetitions,</td>
<td>63</td>
</tr>
<tr>
<td></td>
<td>2 alterations</td>
<td></td>
</tr>
<tr>
<td>7.7</td>
<td>Speedup of code generation modes - 4 blocks, 4 harmonics, 1 repetition of</td>
<td>64</td>
</tr>
<tr>
<td></td>
<td>core calculations</td>
<td></td>
</tr>
<tr>
<td>7.8</td>
<td>Speedup of code generation modes - 4 blocks, 4 harmonics, 5 repetitions of</td>
<td>65</td>
</tr>
<tr>
<td></td>
<td>core calculations</td>
<td></td>
</tr>
<tr>
<td>7.9</td>
<td>Comparison of decision functions - 4 blocks, 4 harmonics, 1 repetition,</td>
<td>68</td>
</tr>
<tr>
<td></td>
<td>0 alterations</td>
<td></td>
</tr>
<tr>
<td>7.10</td>
<td>Comparison of decision functions - 4 blocks, 4 harmonics, 1 repetition,</td>
<td>69</td>
</tr>
<tr>
<td></td>
<td>1 alteration</td>
<td></td>
</tr>
</tbody>
</table>
7.11 Comparison of decision functions - 4 blocks, 4 harmonics, 1 repetition, 2 alterations ................................................................. 69
7.12 Comparison of decision functions - 4 blocks, 4 harmonics, 5 repetitions, 0 alterations ............................................................. 71
7.13 Comparison of decision functions - 4 blocks, 4 harmonics, 5 repetitions, 1 alteration ............................................................... 71
7.14 Comparison of decision functions - 4 blocks, 4 harmonics, 5 repetitions, 2 alterations ............................................................... 72
7.15 Speedup of decision functions - 4 blocks, 4 harmonics, 1 repetition of core calculations ......................................................... 73
7.16 Speedup of decision functions - 4 blocks, 4 harmonics, 5 repetition of core calculations ......................................................... 74
7.17 Comparison of decision functions - 4 blocks, 8 harmonics, 1 repetition, 0 alterations ................................................................. 75
7.18 Comparison of decision functions - 4 blocks, 8 harmonics, 1 repetition, 1 alteration ................................................................. 76
7.19 Comparison of decision functions - 4 blocks, 8 harmonics, 1 repetition, 2 alterations ................................................................. 76
7.20 Comparison of decision functions - 4 blocks, 8 harmonics, 5 repetitions, 0 alterations ................................................................. 77
7.21 Comparison of decision functions - 4 blocks, 8 harmonics, 5 repetitions, 1 alteration ................................................................. 78
7.22 Comparison of decision functions - 4 blocks, 8 harmonics, 5 repetitions, 2 alterations ................................................................. 78
7.23 Speedup of decision functions - 4 blocks, 8 harmonics, 1 repetition of core calculations ......................................................... 79
7.24 Speedup of decision functions - 4 blocks, 8 harmonics, 5 repetition of core calculations ......................................................... 80
7.25 Comparison of decision functions - 8 blocks, 4 harmonics, 1 repetition, 0 alteration ................................................................. 81
7.26 Comparison of decision functions - 8 blocks, 4 harmonics, 1 repetition, 1 alteration ................................................................. 82
7.27 Comparison of decision functions - 8 blocks, 4 harmonics, 1 repetition, 2 alterations ................................................................. 82
7.28 Comparison of decision functions - 8 blocks, 4 harmonics, 1 repetition, 4 alterations ................................................................. 83
7.29 Comparison of decision functions - 8 blocks, 4 harmonics, 5 repetitions, 0 alterations ................................................................. 84
7.30 Comparison of decision functions - 8 blocks, 4 harmonics, 5 repetitions, 1 alteration ................................................................. 84
7.31 Comparison of decision functions - 8 blocks, 4 harmonics, 5 repetitions, 2 alterations ................................................................. 85
7.32 Comparison of decision functions - 8 blocks, 4 harmonics, 5 repetitions, 4 alterations ................................................................. 85
7.33 Speedup of decision functions - 8 blocks, 4 harmonics, 1 repetition of core calculations ........................................ 86
7.34 Speedup of decision functions - 8 blocks, 4 harmonics, 5 repetition of core calculations ........................................ 87

8.1 Comparison of decision functions with the relaxed version of the profiling decision function - 4 blocks, 4 harmonics, 1 repetition, 0 alterations 90
8.2 Comparison of decision functions with the relaxed version of the profiling decision function - 4 blocks, 4 harmonics, 1 repetition, 1 alteration 91
8.3 Comparison of decision functions with the relaxed version of the profiling decision function - 4 blocks, 4 harmonics, 1 repetition, 2 alterations 91
8.4 Comparison of decision functions with the relaxed version of the profiling decision function - 8 blocks, 4 harmonics, 5 repetition, 0 alterations 92
8.5 Comparison of decision functions with the relaxed version of the profiling decision function - 8 blocks, 4 harmonics, 5 repetition, 1 alteration 93
8.6 Comparison of decision functions with the relaxed version of the profiling decision function - 8 blocks, 4 harmonics, 5 repetition, 2 alterations 93
8.7 Comparison of decision functions with the relaxed version of the profiling decision function - 8 blocks, 4 harmonics, 5 repetition, 4 alterations 94
8.8 Comparison of decision functions with the relaxed version of the profiling decision function - 15 blocks, 7 harmonics, 1 repetition, 0 alterations 95

D.1 Comparison of decision functions - 4 blocks, 4 harmonics, 5 repetition, 0 alterations .................................................. 107
D.2 Comparison of decision functions with the relaxed version of the profiling decision function - 4 blocks, 4 harmonics, 5 repetition, 1 alteration 108
D.3 Comparison of decision functions with the relaxed version of the profiling decision function - 4 blocks, 4 harmonics, 5 repetition, 2 alterations 108
D.4 Comparison of decision functions with the relaxed version of the profiling decision function - 8 blocks, 4 harmonics, 1 repetition, 0 alterations 109
D.5 Comparison of decision functions with the relaxed version of the profiling decision function - 8 blocks, 4 harmonics, 1 repetition, 1 alterations 109
D.6 Comparison of decision functions with the relaxed version of the profiling decision function - 8 blocks, 4 harmonics, 1 repetition, 2 alterations 110
D.7 Comparison of decision functions with the relaxed version of the profiling decision function - 8 blocks, 4 harmonics, 1 repetition, 4 alterations 110
# Listings

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Compiler directive for Parallel construct</td>
<td>5</td>
</tr>
<tr>
<td>2.2</td>
<td>Compiler directive for Loop construct</td>
<td>6</td>
</tr>
<tr>
<td>2.3</td>
<td>Routine omp_in_parallel()</td>
<td>7</td>
</tr>
<tr>
<td>2.4</td>
<td>Routine omp_get_num_threads()</td>
<td>7</td>
</tr>
<tr>
<td>2.5</td>
<td>Routine omp_get_thread_num()</td>
<td>7</td>
</tr>
<tr>
<td>2.6</td>
<td>Routine omp_get_wtime()</td>
<td>7</td>
</tr>
<tr>
<td>2.7</td>
<td>Outer loop parallelisation of a nested loop region</td>
<td>9</td>
</tr>
<tr>
<td>2.8</td>
<td>Inner loop parallelisation of a nested loop region</td>
<td>10</td>
</tr>
<tr>
<td>2.9</td>
<td>Nested loop parallelisation of a nested loop region</td>
<td>10</td>
</tr>
<tr>
<td>2.10</td>
<td>Parallelisation of a nested loop region with loop collapsing</td>
<td>11</td>
</tr>
<tr>
<td>2.11</td>
<td>Example of creating patterns with Lpeg</td>
<td>14</td>
</tr>
<tr>
<td>2.12</td>
<td>Example of pattern repetition with Lpeg</td>
<td>14</td>
</tr>
<tr>
<td>2.13</td>
<td>Example of pattern composition with Lpeg</td>
<td>15</td>
</tr>
<tr>
<td>2.14</td>
<td>Example of syntactic predicates with Lpeg</td>
<td>16</td>
</tr>
<tr>
<td>2.15</td>
<td>Example of parser for evaluating arithmetic expressions</td>
<td>19</td>
</tr>
<tr>
<td>2.16</td>
<td>Extract from the C99 language specification for the statement rule</td>
<td>20</td>
</tr>
<tr>
<td>2.17</td>
<td>Example of writing a grammar to match statements with Lpeg</td>
<td>21</td>
</tr>
<tr>
<td>3.1</td>
<td>A nested loop region with preomp</td>
<td>25</td>
</tr>
<tr>
<td>3.2</td>
<td>An example of a C for loop</td>
<td>25</td>
</tr>
<tr>
<td>3.3</td>
<td>Usage of the driver</td>
<td>34</td>
</tr>
<tr>
<td>4.1</td>
<td>Pseudo-code for decision based on heuristics</td>
<td>37</td>
</tr>
<tr>
<td>4.2</td>
<td>Pseudo-code for decision based on heuristics with profiling</td>
<td>38</td>
</tr>
<tr>
<td>4.3</td>
<td>Pseudo-code for function before the execution of a serial loop</td>
<td>40</td>
</tr>
<tr>
<td>4.4</td>
<td>Pseudo-code for function after the execution of a serial loop</td>
<td>41</td>
</tr>
<tr>
<td>4.5</td>
<td>Pseudo-code for function before the execution of a parallel loop</td>
<td>43</td>
</tr>
<tr>
<td>4.6</td>
<td>Pseudo-code for function after the execution of a parallel loop</td>
<td>44</td>
</tr>
<tr>
<td>5.1</td>
<td>Synthetic Benchmark Code</td>
<td>49</td>
</tr>
<tr>
<td>5.2</td>
<td>CFD extract code</td>
<td>50</td>
</tr>
<tr>
<td>B.1</td>
<td>Two nested loops with work in-between each loop</td>
<td>102</td>
</tr>
<tr>
<td>C.1</td>
<td>Testing the if clause and code duplication modes</td>
<td>105</td>
</tr>
</tbody>
</table>
Acknowledgements

I would like to thank my supervisor Adrian Jackson for his support and guidance during this project. His friendly attitude was a real source of motivation for the completion of this project.

I would also like to give my appreciation to Dr. Judy Hardy and Dr. Fiona Reid for their understanding and moral support.

Furthermore, I would like to give my thanks to all of the friends I have made during this MSc. course.

Last but not least, I would like to thank my family for making these studies possible.
Chapter 1

Introduction

HPC codes and in particular scientific codes, require parallel execution in order to achieve a large amount of performance increase. Depending on the underlying parallel platform which is used, programmers use different programming models in order to achieve parallel execution. In distributed memory systems, the message passing programming model is the most commonly used approach for applying parallelism in the codes. In shared memory systems however, an attractive choice for parallel programming is with OpenMP[16].

The parallelisation of codes with OpenMP is often achieved with loop parallelisation. As long as the iterations of a loop are independent, they can be distributed to the available processors of the system in order to execute them in parallel. A programmer is required to specify a loop that can be parallelised by placing compiler directives before the loop, resolving any dependency issues between the iterations beforehand.

HPC codes often consist of regions with nested loops of multiple levels. In order to parallelise these regions, a choice must be made on how parallelism should be applied on the loops. Even though OpenMP supports a variety of strategies for parallelising nested loops, only a single one can be used to parallelise the code. A static choice however, cannot exploit any runtime characteristics during the execution of the program. Changes in the input parameters of the executable which affect the iterations of the parallel loops may render the decision suboptimal. In addition to this, the iterations of a loop can change at runtime due to the nature of the code. A common feature of HPC codes is to organise the data into hierarchies, for example blocks of multi-dimensional arrays. Depending on the problem, the blocks can have different shapes and sizes. These parameters affect the loops that are responsible for accessing these data. In some situations, a static decision has the potential to impose a limitation on the amount of processors that can be used for the parallel execution of the loops. With the current trend of chip manufacturers to increase the number of cores in the processors in each generation, a more dynamic approach must be considered for taking such decisions.

The aim of this dissertation is to investigate various strategies that can be applied at runtime in order to make a dynamic decision on how to parallelise a region with nested
loops. We focus on shared memory parallel programming models, OpenMP in particular, and we try to automatically perform modifications to the original code before compilation in order to enable the code to make these decisions dynamically at runtime. Specifically, we are investigating the possibility of having multiple versions of a loop within a region of nested loops in order to make a dynamic choice on whether a loop should execute sequentially or in parallel.

Code duplication is considered as a bad programming practice since it makes the code hard to maintain. However, it can be done automatically right before the compilation of the code. By exposing a simple interface to the programmers with compiler directives, similar to the case of OpenMP, a source to source compiler can be used to pre-process the code and perform the necessary transformations. The opposite to this would have required the users to apply the modifications on the original code manually. This task however is error-prone and ultimately removes all the good qualities of a code such as its readability, ease of debugging and maintainability. For this reason, we have chosen to develop a source to source compiler in order to automatically perform this task and remove this responsibility from the users. Furthermore, this approach provides the users the choice of enabling or disabling our functionality with minimum effort without altering the structure of the original source code. The only requirement is to include or omit the compiler directives which are exposed by our compiler.

OpenMP already provides a way of forcing a parallel region to execute sequentially with the use of the \textit{if clause} in the OpenMP directive. However, programmers are still required to manually write code which makes the decision. In addition to code duplication, we also investigate the use of the \textit{if clause}, by producing the code which is responsible for the decision automatically.

The decision considers the number of iterations of a loop in order to chose a parallelisation strategy that makes best use of the available processors. Other authors [12] have already taken a similar approach by modifying the OpenMP runtime library in order to make these decisions dynamically. However, applying this logic in the OpenMP runtime library limits the implementation to a specific compiler. In this work, we are investigating the possibility of transferring the logic in user code in order to maintain the portability of our solution.

In addition to simple heuristics, we also explore the idea of a profile-based approach at runtime in order to detect the best possible parallelisation strategy with time measurements. A heuristics based approach alone cannot capture any information on the amount of the actual computations when making a decision on parallelising a loop. With this approach, we are interested in finding situations where using less threads on an outer loop might provide a better execution time. The idea of an auto tuning code is not new itself, as it has already been proposed by other compiler-related researches [13] [7] for producing optimised code. We also use this idea by applying a similar logic at runtime.

The organisation of this report is organised in the following Chapters:

Chapter 2 provides the background theory of all the concepts which are used in this report, including information on the technologies used.
In Chapter 3, the design and implementation of the source to source compiler is described. The capabilities of the compiler and the design choices which were made are explained.

Chapter 4 provides detailed information about the runtime library which accompanies the source to source compiler. Each decision function is explained in detailed, followed by an overview of the design of the library.

Chapter 5 contains information about the experimental design which was used during the conduction of the experiments. A brief overview of the system and the benchmark codes is given.

Chapter 6 and 7 present the results of the benchmarks of the two codes. For each graph there are comments and discussion on the observed behaviour.

Chapter 8 provides an alternative version of the decision function with profiling, and the results from the benchmarks used by this version.

Finally, Chapter 9 summarises our conclusions from the project, as well as some ideas that can be applied for future enhancements.
Chapter 2

Background theory

2.1 OpenMP Programming Model

OpenMP [16] is an industry standard for writing parallel programs on shared memory systems. Now at version 3.1, OpenMP has already become a popular choice in the HPC community.

What makes OpenMP an attractive choice to programmers, as opposed to alternatives such as the Posix Threads standard, is the approach it takes for specifying the parallel logic within a code. Unlike Posix Threads, which works entirely on library calls and forces the programmer to write code with parallel semantics in mind, OpenMP operates on compiler directives. The programmer provides a description of how the code should be parallelised, which the compiler then uses to transform the original code into the parallel version. By providing this higher level of abstraction, OpenMP codes tend to be easier to develop, debug and maintain. Moreover, with OpenMP it is very easy to develop the parallel version of a serial code without any major modifications. The same cannot be argued for Posix Threads.

The remainder of this section provides an overview of the OpenMP programming model and a basic introduction to the most commonly used compiler directives. It is out of the scope of this dissertation to provide a full coverage of the OpenMP standard.

2.1.1 Execution Model

At the beginning of a code, the program starts its execution with a single active thread. The thread executes instructions in a sequential fashion, until a Parallel construct is found. At that point, the thread creates a thread team by spawning a number of threads and becomes the Master of that team, that is, the initial thread is also a member of the team. Each one of the threads starts executing the portion of the code which is enclosed by the Parallel construct. In OpenMP lingua, the enclosed code is referred to as a parallel region.
Each thread executes the code of the parallel region individually. In normal circumstances, the threads will only get synchronised at the end of the parallel region by an implicit barrier. However, compiler directives are available for the programmer in order to force synchronisation explicitly within the parallel region. When all of the threads of the team reach the implicit barrier, the Master thread continues its execution sequentially, until a new Parallel construct is found.

The model of execution described above is known as the Fork-Join model of parallel execution. The Fork phase of the model refers to the creation of threads by the master thread, whereas the Join phase refers to the synchronisation of the threads at the end of the parallel region. A visual depiction of the model is presented in figure 2.1.

![Fork-Join parallel execution model with four threads](image)

Figure 2.1: Fork-Join parallel execution model with four threads

### 2.1.2 Compiler Directives

#### Parallel Construct

The fundamental construct of OpenMP is the Parallel construct, which is responsible for starting the parallel execution. The syntax of the associated compiler directive is presented in listing 2.1. Upon reaching such a construct, the execution of the program follows the Fork-Join model described in section 2.1.1 of this dissertation.

```c
#pragma omp parallel [[clause][,][clause] ...] new_line
{
    /* statements */
}
```

Listing 2.1: Compiler directive for Parallel construct
A number of optional clauses can be specified in the line of the directive. These clauses are mainly used to specify the data sharing attributes of the lexical symbols which appear in the parallel region. Apart from specifying data sharing attributes, the directive also supports the \texttt{if(scalar-expression)} clause.

The purpose of the \texttt{if} clause is to determine at runtime whether the code enclosed in the parallel region should execute sequentially or in parallel. When the scalar expression of the clause evaluates to \texttt{0}, the region is executed sequentially. Any other value will result in parallel execution. However, a new parallel region is always created in either case. The presence of the \texttt{if} clause only affects the number of threads that get assigned to the parallel region. When a sequential execution is determined, the thread team of the parallel region consists of a single thread, the master.

As discussed in section 2.1.1, a thread that creates a parallel region is also a member of that region. When a new parallel region is created by an external thread and its determined that sequential execution should be applied, then the external thread is the only member of the new parallel region.

\textbf{Loop Construct}

Within a parallel region, various programming language constructs can expose additional parallelisation opportunities. In order to take advantage of this feature, OpenMP provides additional directives to divide the work of such constructs to the threads of the associated team.

The most common worksharing construct is the \textit{Loop Construct} as shown in listing 2.2. Its purpose is to divide the iterations of the following loop to the threads which are members of the active team. A variety of \textit{clauses} are available in order to choose the scheduling scheme of the iterations to the members of the thread team. OpenMP allows the combination of the compiler directive for the \textit{Loop Construct} and \textit{Parallel construct} into a single line. By using this format, the parallel execution is initiated right before the loop, followed by the distribution of the loop iterations to the threads of the newly created team.

\begin{verbatim}
#pragma omp parallel for [[clause][,]...] new_line
for (init-expression; cond-expression; step-expression)
{
    /* statements */
}
\end{verbatim}

Listing 2.2: Compiler directive for Loop construct

OpenMP expects the \textit{for statement} to comply with a number of conformance rules. These restrictions affect the format of the three expressions which appear in the header of the \textit{for statement}. By placing such restrictions, it is possible for a compiler which supports the OpenMP language to analyse a parallel loop in order to calculate the number of iterations of the loop as well as the iteration space of the loop variable. The
calculations are needed in order to correctly divide and schedule the iterations of the loop to the threads. The most common cases for each expression are shown in table 2.1.

<table>
<thead>
<tr>
<th>Initialisation</th>
<th>Condition</th>
<th>Step</th>
</tr>
</thead>
<tbody>
<tr>
<td>var = expression</td>
<td>var &lt; expression</td>
<td>var++</td>
</tr>
<tr>
<td>type var = expression</td>
<td>var &lt;= expression</td>
<td>++var</td>
</tr>
<tr>
<td></td>
<td>var &gt; expression</td>
<td>var−</td>
</tr>
<tr>
<td></td>
<td>var &gt;= expression</td>
<td>−var</td>
</tr>
<tr>
<td>expression &lt; var</td>
<td>var += expression</td>
<td>var += expression</td>
</tr>
<tr>
<td>expression &lt;= var</td>
<td>var −= expression</td>
<td>var −= expression</td>
</tr>
<tr>
<td>expression &gt; var</td>
<td>var = var + expression</td>
<td>var = var + expression</td>
</tr>
<tr>
<td>expression &gt;= var</td>
<td>var = var - expression</td>
<td>var = var - expression</td>
</tr>
</tbody>
</table>

Table 2.1: OpenMP conformance rules for loop statements

### 2.1.3 Runtime Library

In combination with compiler directives, the OpenMP specification defines an Application Programming Interface (API) which provides a number of routines that can be used within a parallel code. The functionality which is provided by these routines allows the code to read and modify the values of various internal parameters of the OpenMP library, as well as the use of data protection mechanisms and timing facilities. An overview of the relevant routines which are used in this work is given below.

```c
int omp_in_parallel (void)
```

Listing 2.3: Routine omp_in_parallel()

```c
int omp_get_num_threads (void)
```

Listing 2.4: Routine omp_get_num_threads()

```c
int omp_get_thread_num (void)
```

Listing 2.5: Routine omp_get_thread_num()

```c
double omp_get_wtime (void)
```

Listing 2.6: Routine omp_get_wtime()

The `omp_in_parallel()` routine is responsible for identifying whether the call to the routine is enclosed by an active parallel region. If this condition does not apply, the routine returns a zero value which in the C programming language denotes a false expression. In contrast to this, if the call is enclosed within a parallel region, a non-zero value is returned.
The `omp_get_num_threads()` routine returns the number of threads that take part in the parallel region which encloses the function call. Outside of a parallel region, the routine returns a value of one.

The `omp_get_thread_num()` routine returns the thread number which is assigned to the calling thread of a specific thread team. The returned value of zero denotes the thread number of the master thread.

The `omp_get_wtime()` routine is used for timing purposes, and returns the elapsed time in seconds from some time in the past. The reference time is undefined, however timing of a code region can be accomplished by using a pair of calls to the routine one to mark the time before the profiling region and another one to mark the time after the execution of that region. The difference of the produced values can be used to calculate the amount of time spent in the region of the code.

### 2.2 Parallelisation Strategies For Nested Loops

HPC codes, and particularly scientific codes, deal with numerical computations based on mathematical formulas. These formulas are often expressed in the form of nested loops, where a set of computations is applied to a large amount of data (generally stored in arrays) and parallelisation can be applied to each loop individually. The arrays often consist of multiple dimensions and the access on the data is achieved with the presence of nested loops. Furthermore it is not uncommon that the arrangement of the data is done in multiple hierarchies, most commonly in blocks with multi-dimensional arrays, where additional loops are require in order to traverse all the data. When such code is presented, a choice must be made on which loop level to parallelise (where the parallelisation should occur). [12]. A summary of the available strategies is presented in table 2.2.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Outermost Loop</td>
<td>Parallelisation of the outermost loop</td>
</tr>
<tr>
<td>Inner Loop</td>
<td>Parallelisation of one of the inner loops</td>
</tr>
<tr>
<td>Nested</td>
<td>Parallelisation of multiple loops with nested parallel regions</td>
</tr>
<tr>
<td>Loop Collapsing</td>
<td>Collapsing the loops into a single big loop</td>
</tr>
</tbody>
</table>

Table 2.2: Alternative strategies for parallelising nested loop regions

#### 2.2.1 Outermost loop

The most commonly used approach is to parallelise the outermost loop of a nested loop region, as shown in listing 2.7. Using this strategy, the iterations of the loop are distributed to the members of the thread team. The threads operate in parallel by
executing the portion of iterations they are assigned to them individually. The nested loops of the parallel region are executed in a sequential manner.

Parallelising the outermost loop is often a good choice, as it minimises the overheads which are produced from the parallel logic of the code. Such overheads are the result of the initialisation of the parallel region, the scheduling of the iterations to the threads, as well as the synchronisation which takes place by the implicit barrier at the end of the Parallel construct. Furthermore, the work of each iteration of the outer loop is coarse enough, which reduces the likelihood of the parallel overheads having a negative impact on the execution time. A more extensive work on the overheads of various OpenMP directives can be found in [10] [14].

```c
#pragma omp parallel for private(j)
for (i = 0; i < I; i++) {
    for (j = 0; j < J; j++) {
        work();
    }
}
```

Listing 2.7: Outer loop parallelisation of a nested loop region

Despite all the advantages of the Outermost Loop parallelisation strategy, there is a subtle drawback in this choice. The maximum amount of available parallelism is limited by the number of iterations of that loop. Considering the example code in listing 2.7, it is only possible to have \( I \) tasks being executed in parallel. This also restricts the number of threads the code can utilise upon its execution.

### 2.2.2 Inner loop

The Inner Loop parallelisation strategy is similar to the outermost loop strategy. As the name implies, one of the inner loops of the region is chosen to be parallelised which potentially exposes a greater amount of parallelism due to the number of its iterations. This allows more threads to be used, which can result in reduction of the overall execution time.

However, this approach can only be beneficial if the iteration’s amount of work of the chosen loop remains coarse enough to compensate for the overheads of the parallel logic. As shown in listing 2.8, the parallel region is created and terminated for each iteration of the outer loop. What is more, parallelising a loop of a deeper level will make matters even worse; The parallel overheads appear a lot more times, whereas the amount of work of each iteration becomes more fine.

Another problem of this strategy is when the loops are not perfectly nested. In the situation when there are computations in-between the loops, parallelising a loop of a deeper level will result in sequential execution of that work. Depending on the amount of the execution time which is now serialised, this approach has the potential to increase the
execution time of the code.

```
for (i = 0; i < I; i++) {
    #pragma omp parallel for shared(i)
    for (j = 0; j < J; j++) {
        work();
    }
}
```

Listing 2.8: Inner loop parallelisation of a nested loop region

### 2.2.3 Nested

The Nested parallelisation strategy exploits the fact that more than one loops can be executed in parallel. By opening multiple nested parallel regions at different levels of loops, as presented in listing 2.9, more threads can be utilised during the parallel execution of the code.

Unlike the Outermost Loop and the Inner Loop approaches, which can only utilise as many threads as the iterations of the loop with the biggest number of iterations, this strategy can exploit even more parallelisation opportunities. Other studies have shown that nested parallelism can give good results on systems with a large number of processors.[17][8].

```
#pragma omp parallel for private(j)
for (i = 0; i < I; i++) {
    #pragma omp parallel for shared(i)
    for (j = 0; j < J; j++) {
        work();
    }
}
```

Listing 2.9: Nested loop parallelisation of a nested loop region

### 2.2.4 Loop collapsing

The loop collapsing strategy takes a different approach for exposing additional parallelism within nested loop regions. By performing code transformations, multiple nested loops are combined, or collapsed, into a single loop. The newly created loop has a large amount of iterations, which can be distributed to the threads. In order to achieve this task, a compiler must perform a good analysis on the code in order to ensure that the semantics of the nested loop region are still preserved in the newly created loop.

As of version 3.0, OpenMP supports loop collapsing by using the COLLAPSE clause in the Loop Construct, requiring the programmer to provide the number of loop levels.
to collapse. It is worth mentioning that ideal collapsing of loops into a single loop may not be possible. In spite of this, when ideal collapsing is possible, it can produce better results than both the inner loop and nested loop strategies, since the parallel overheads are minimal.

```cpp
#pragma omp parallel for collapse(3)
  for (i = 0; i < I; i++) {
    for (j = 0; j < J; j++) {
      for (k = 0; k < K; k++) {
        work();
      }
    }
  }
```

Listing 2.10: Parallelisation of a nested loop region with loop collapsing

### 2.3 Source-To-Source Compilers

Source-to-Source compilers are a type of compilers that accept code for a High Level Programming Language as input and output code for another High Level Programming Language. Unlike conventional compilers, their output is not an executable binary file. A conventional compiler must be used in order to convert the output code into an executable.

The most common reason for developing a source-to-source compiler instead of a conventional one is portability. By targeting a widely supported language such as C as an output language, one can benefit from all the available compilers of the language and the platforms they support. Furthermore, the construction of an optimising compiler that outputs machine code is a very complex and time-consuming process. Therefore, source to source compilers are able to re-use all of the work that has been undertaken into the development and testing of an existing compiler.

Another use of source-to-source compilers is to extend an already existing programming language. In this case, the input language is a superset of the output language. The compiler is used to translate, or rewrite the extensions of the input language into semantics of the output language. From this point of view, a source to source compiler acts as a preprocessor to the underlying target language.

Despite the advantages of using a source-to-source compiler instead of a conventional one, there is one major limitation. Any optimisations that are potentially carried out by the compiler are limited by the semantics of the target language; After all, the code will be eventually translated into the output language. In contrast to this, conventional compilers can exploit all of the available optimisation opportunities. Once the input code is parsed, the compiler is free to break out of the semantics of the input language. This limitation of source to source compilers may have a negative impact on the performance of the output code.
2.4 The Lua Programming Language

Lua [3] is a small, interpreted programming language designed to be used as a scripting language. The main design goals of Lua are to be an extension and extensible language. The former refers to Lua’s ability to be embedded and interact with C and C++ codes, providing scripting capabilities to programs. The latter refers to the ability of extending the core library of the language with the use of C and C++ languages. For these reasons, the language is very lightweight with small overheads.

As with most popular interpreted languages such as Python and Ruby, Lua has a dynamic type system. Unlike statically typed systems such as C, C++ and Java, variables in a Lua code do not have a type. They are arbitrary references to objects in memory and can point to any type of values. The number of core types which are provided by the language is really small, however one can create arbitrary data-structures by using Lua’s associative arrays.

Unlike most programming languages, Lua does not restrict the programmer into a specific programming paradigm. In contrast, Lua provides a small set of well thought out features along with "syntactic sugar" elements within the syntax of the language, which make it possible to develop codes following a variety of programming paradigms. For example, although Lua does not support object-oriented programming constructs by default, it is very easy to create a well-featured class system with Lua code [4].

This work uses Lua, along with the Lpeg [2] parsing library for the construction of the Source-to-Source compiler described in section 3 of this dissertation. The choice has been made in order to reduce the development time of the compiler and remain within the available timing constraints of the project, since the language is an excellent choice for Rapid Application Development (RAD) and prototyping of ideas.

2.5 The Lpeg Parsing Library

Lpeg [2] is a pattern-matching library for the Lua programming language. It is based on the work on Parsing Expression Grammars (PEG)[9], which can be viewed as a formalised way of describing top-down parsers. Despite the fact that Lpeg is an independent library and not a part of the core Lua programming language, it is very well supported as it is developed by the creators of the language.

What distinguishes Lpeg from more popular parser construction tools such as ANTLR and YACC/Bison is that the grammar of the target language is defined in plain Lua code. The construction of a parser with these tools requires the programmer to define a grammar by using some sort of representation, normally based on the BNF notation. The representation is then transformed by the tool into source code of a target programming language, with the programmer having little or no influence on the structure and properties of the generated code. With Lpeg, rules, or patterns in Lpeg’s lingua, become first
class values within the language. Each rule of the grammar is essentially a Lua value and can be stored in a normal Lua variable. This gives great flexibility of how the code is laid out which allows the programmer to organise the code as he sees fit.

Another feature of Lpeg, which is inherited from PEG is that there is no need to transform the input text into tokens. The library provides facilities to describe a grammar down to the character level by allowing the use of ideas inherited from regular expressions within the definition of patterns. As a result, a grammar defined with Lpeg is self contained.

2.5.1 Construction of Parsers

The Lpeg library supports two methods of defining the rules of a grammar. The first method, which is provided by the `re` module, accepts a grammar definition as a string. Although the syntax of the input grammar is simple and closely resembles regular expressions for the repetition of rules, the method is not flexible as the grammar cannot be modified after its definition. The second method is the use of normal Lua code with function calls provided by the Lpeg library for the creation and composition of patterns. This section of the dissertation focuses on the second method, which is also the one used for the development of our compiler.

Pattern Creation

There are three basic functions provided by Lpeg which can be used in order to create patterns for matching text, as shown in table 2.3. With the exception of the `lpeg.P()` function, which behaves differently based on the type of the provided argument, the functions are straightforward.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>lpeg.P(string)</code></td>
<td>Generates a pattern that matches the provided string</td>
</tr>
<tr>
<td><code>lpeg.P(n)</code></td>
<td>Generates a pattern that matches exactly n characters</td>
</tr>
<tr>
<td><code>lpeg.P(boolean)</code></td>
<td>Generates a pattern that always succeeds or fails</td>
</tr>
<tr>
<td><code>lpeg.S(string)</code></td>
<td>Generates a pattern that matches one of the characters from the set</td>
</tr>
<tr>
<td><code>lpeg.R(&quot;xy&quot;)</code></td>
<td>Generates a pattern that matches any character between x and y</td>
</tr>
</tbody>
</table>

Table 2.3: Lpeg functions for creating patterns

Similar to regular expression, one can use these functions to match strings, characters from a set or characters within a range. If we consider the example in listing 2.11, we can observe three things:

- The example is a normal Lua code
- `int`, `digit` and `char` are normal Lua variables which can be used to match the string "int", a digit or an ASCII character respectively.
• Lpeg does not need the use of a lexer as patterns can be defined down to the character level.

```lua
require("lpeg")
int = lpeg.P("int")
digit = lpeg.R("09")
char = lpeg.R("az", "AZ")
```

Listing 2.11: Example of creating patterns with Lpeg

**Pattern Repetition**

Lpeg allows to specify repetition of patterns as shown in table 2.4. This is accomplished by overloading the Lua ^ operator on values which are patterns. By using this operator, a new pattern can be created which can match at most \( n \), or at least \( n \) repetitions of another pattern. If we extend the previous example code in listing 2.11 we can create patterns to match words and positive numbers as shown in listing 2.12.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pattern^ n</td>
<td>Match at least ( n ) repetitions of pattern</td>
</tr>
<tr>
<td>pattern^ -n</td>
<td>Match at most ( n ) repetitions of pattern</td>
</tr>
</tbody>
</table>

Table 2.4: Lpeg operator for pattern repetition

```lua
require("lpeg")
digit = lpeg.R("09")
char = lpeg.R("az", "AZ")
number = digit^1
word = char^1
```

Listing 2.12: Example of pattern repetition with Lpeg

**Pattern Composition**

New patterns can be created by composing two or more existing patterns in order to create more complex but more powerful patterns. This is possible with the use of the '*' and '+' operators. The former is used in order to match a series of patterns, and the latter to make a choice between patterns (table 2.5).

Using these two operators, the definition of a pattern which matches identifiers is shown in listing 2.13. The example code defines an identifier so that it must begin with either an underscore or a single ASCII character, followed by zero or more repetitions of a combination of underscores, ASCII characters and digits.
The composition operators are the traditional *multiplication* and *addition* operators. Therefore, their precedence also reflects their mathematical meaning. Similar to arithmetic operations, patterns can be grouped together in order to enforce precedence, if this is necessary. It can be observed from the example that repetitions, with the ^ operator, can be applied directly on these groups, without the need to define a separate pattern beforehand.

An important feature of PEG and the Lpeg library is that choices are ordered. In the identifier example, char is only tried if underscore fails, whereas digit is only tried if underscore, then char, fails. This is a very powerful feature to have when the target is a complete programming languages, as it leads to a grammar without ambiguities. For example, a common use case of ordered choices is to resolve the precedence rules of arithmetic operations, by carefully selecting the order of the appropriate rules when defining the grammar. In contrast to this, a grammar defined in YACC/Bison must also specify hints to the tool, which describe in what order the operations must be evaluated.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pattern1 * pattern2</td>
<td>Match pattern1 followed by pattern2</td>
</tr>
<tr>
<td>pattern1 + pattern2</td>
<td>Match pattern1 or pattern2</td>
</tr>
</tbody>
</table>

Table 2.5: Lpeg operators for pattern composition

```plaintext
1  require("lpeg")
2
digit = lpeg.R("0-9")
3  char = lpeg.R("az", "AZ")
4  underscore = lpeg.P("_")
5
6  identifier = ( underscore
7      + char ) * ( underscore
8      + char
9      + digit )^0
```

Listing 2.13: Example of pattern composition with Lpeg

**Syntactic Predicates**

Syntactic predicates are a feature of Lpeg and PEG in general which serve as a mechanism to achieve *look-ahead* within the grammar. Look-ahead is generally used in order to make additional decisions on whether a pattern successfully matches or fails, regardless of what the definition of the pattern describes, based on what input follows next. In this point of view, syntactic predicates provide a way to specify simple logic within a pattern. Table 2.6 presents the operators for syntactic predicates. The '-' operation can be used for *negation* in order to specify a case where a pattern is successful only if another pattern fails. In contrast, the '#' operator is used for *positive* look-ahead, where
the matching of a pattern succeeds if another pattern succeeds as well. An important feature of these operators is that they do not consume any input. They merely perform tests.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pattern1 - pattern2</td>
<td>pattern1 is successful if pattern2 matches</td>
</tr>
<tr>
<td>-pattern</td>
<td>pattern must not match</td>
</tr>
<tr>
<td>#pattern</td>
<td>pattern must match</td>
</tr>
</tbody>
</table>

Table 2.6: Lpeg operators for syntactic predicates

If we considering the identifier example in listing 2.13, the pattern for the identifier cannot be used as part of a programming language’s grammar, since it will also match restricted keywords of the language. In order to prevent this behaviour, the pattern can be defined as shown in listing 2.14. This way, an identifier succeeds only if the input text in question is not a keyword.

```lua
1 require("lpeg")
2 3 digit = lpeg.R("0 9")
4 char = lpeg.R("az", "AZ")
5 underscore = lpeg.P("_")
6 7 keyword = lpeg.P("for")
8     + lpeg.P("while")
9 10 identifier = (−keyword) * ( underscore
11     + char ) * ( underscore
12     + char
13     + digit )^0
```

Listing 2.14: Example of syntactic predicates with Lpeg

Semantic Actions And Captures

Lpeg uses the term *Capture* to describe information that is captured from a successful match of a pattern, as well as actions to apply on the captured information. If we compare Captures with concepts from traditional parser generation tools, a capture in Lpeg is both a *semantic value* produced by a pattern as well as a *semantic action* which is applied on it.

The definition for captures in a pattern can be done in a number of ways. First, as part of the pattern, using functions provided by the Lpeg library, or as a normal Lua function which is associated with the pattern. A summary of some useful Captures is presented in table 2.7.

The most basic form to produce a capture of a pattern is to make a function call to the lpeg.C() function, providing the pattern as argument. In the same way that function
<table>
<thead>
<tr>
<th>Operation</th>
<th>Produced values</th>
</tr>
</thead>
<tbody>
<tr>
<td>lpeg.C(pattern)</td>
<td>The part of the input text matched by the pattern or any additional captures produced by the pattern</td>
</tr>
<tr>
<td>lpeg.Carg(n)</td>
<td>The n’th extra argument passed to the lpeg.match function</td>
</tr>
<tr>
<td>lpeg.Cp()</td>
<td>The current position (byte offset) of the input text during matching.</td>
</tr>
<tr>
<td>pattern / string</td>
<td>The produced capture of the pattern is the supplied string</td>
</tr>
<tr>
<td>pattern / function</td>
<td>The return values of the function when it is called with the captures produced by the pattern.</td>
</tr>
</tbody>
</table>

Table 2.7: Summary of useful captures with Lpeg

calls are used to define captures, this function can be used as part of the definition of the capture as well. This is possible because in the user’s point of view, a capture creates a new pattern. Internally, Lpeg knows that there is also an action involved when the pattern matches a text.

Using the Lua ‘/’ operator on a pattern can either produce a constant string capture as the semantic value of the pattern, or attach a normal Lua function on the pattern in order to perform some actions on the values which are captured by it. In the former form, despite of what text the pattern matches, the supplied string is always generated as the captured value of that pattern. In the latter form, when used with a Lua function, the captured values of the pattern are passed as arguments to the function. The function can then modify or ignore the supplied values as well as generate additional values. The final captured values of the pattern will become the values returned by the function.

The lpeg.Carg() and lpeg.Cp() functions are less common, however their functionality is of great importance when developing complex parsers for programming languages. The first function can be used to make external values accessible to the parser. In conventional compilers, before the parsing of an input text is started, a lot of data-structures are initialised which are needed by the parser. A common design choice of developers is to make these structures globally accessible, in order to reduce the complexity of the code for the parser. However, a major drawback of this approach is that the parser is non-reentrant. Only one input text can be parsed at a time, and the data-structures must get re-initialised before a new input is parsed. The lpeg.Carg() function allows to make external values accessible to the parser. The function can be used as part of the definition of a pattern, producing the provided external value as part of the pattern’s semantic values. This provides an elegant solution to the re-entrance problem. Finally, the lpeg.Cp() function is useful for error reporting, as it produces the position within the input text during the matching phase.

An example of a simple parser which evaluates an arithmetic expression of additions and subtractions of numbers is presented in listing 2.15. A simple twist to the example is that the final value of the expression is multiplied by a user provided number.

An important feature of the parser is the way it handles spaces between a number and the
addition and subtraction operators. In conventional compilers, the tokenisation phase of the input text takes care of spaces. Since Lpeg does not use a tokeniser, this must be resolved within the patterns. For this reason, we define a ‘space’ pattern which matches zero or more repetitions of a space or a tab character. This pattern is then used in the definition of the ‘plus’, ‘minus’ and ‘number’ patterns in order to match trailing spaces. However, we do not want to include any spaces in the captured values of these patterns. Therefore, we use the `lpeg.C()` function to capture only digits when matching a number, and a constant capture of ‘+’ and ‘-’ when matching ‘plus’ and ‘minus’ respectively.

An expression is then defined as a number, followed by zero or more repetitions of an addition or subtraction operator and a number. The pattern also produces the first extra argument passed to the `lpeg.match()` function. In order to actually evaluate the input expression, we use a function capture by attaching the `evaluate()` Lua function onto the pattern of the expression. Upon successfully matching an expression, the function will be called with the captures of the ‘expression’ pattern as arguments.

The `evaluate()` function always receives at least two arguments; The produced value from `lpeg.Carg(1)`, which is stored in the `multiply_by` variable, and the first number of the ‘expression’ pattern, stored in the `first_number` variable. It can also receive an arbitrary number of extra arguments, of addition or subtraction operators and numbers. Therefore, the extra arguments are converted into a table (`extra_args`), and the sum is initialised as the first number of the expression. The table is then iterated in steps of two, adding or subtracting values as necessary to the sum during each iteration. Finally the product of the evaluated expression and the externally supplied number is returned as the final capture of the expression pattern.

In order to perform matching of an input expression, a call to the `lpeg.match()` function is made. The first argument is the ‘expression’ pattern which is followed by the input string of the expression. The third argument of the function specifies the position within the input string to begin matching. Finally the value of ‘10’ is the first extra argument, which is made accessible to the parser. The captures of the ‘expression’ pattern, which is the product calculated by the evaluate function, will be the returned value of the `lpeg.match()` function.

Although the functionality of the example is simple, it demonstrates the expressiveness and the ease of use of Lpeg and Lua for developing parsers. Moreover, the same ideas in the example are also used for the development of the Source to Source compiler, to be discussed.
require ("lpeg")

--- Shortcuts to lpeg functions
Carg = lpeg . Carg
C  = lpeg . C
S  = lpeg . S
P  = lpeg . P
R  = lpeg . R

function evaluate (multiply_by , first_number , ...)  
  local extra_args = {...}
  local sum = first_number  
  for i = 1 , #extra_args , 2 do  
    if args [ i ] == "+" then  
      sum = sum + tonumber (extra_args [ i + 1 ])  
    else  
      sum = sum − tonumber (extra_args [ i + 1 ])  
    end  
  end  
  return multiply_by * sum  
end

space = S ( "\t" ) ^ 0  
plus  = (P ("+" ) * space ) / "+
minus = (P("−") * space ) / "−"
digit = R ("09")
number = C ( digit ^ 1 ) * space
equation = Carg (1) * number * (( plus  
                  + minus ) * number) ^ 0 / evaluate  
product = lpeg . match (equation , "1+8−4" , 1 , 10)
print("Product = ", product)

Listing 2.15: Example of parser for evaluating arithmetic expressions

Arrangement Of Patterns Into Grammars

The main design goal of Lpeg is to match patterns. Unlike other parser construction
tools, which can only be used to parse a text based on a complete grammar, Lpeg was
designed to be used as an alternative to regular expression, when a more powerful parser
is needed. For this reason, the core element of Lpeg is a pattern and not a grammar. If
we consider the example of the identifier pattern in listing 2.13, the pattern can be used
directly to parse an input text. In tools such as ANTLR, identifier would be a rule within
a larger grammar, rendering the rule impossible to be used directly.

For small grammars with a very simple syntax this is not a problem. Patterns can be de-
dered incrementally and stored in a separate variable at each step. New patterns then can
be composed by already defined patterns, until the final pattern is generated, which can
be viewed as the complete grammar. In the identifier example (listing 2.13), the pattern
for the identifier can only be created after the definition of digit, char and underscore.
However, incremental definition of the grammar of a complete programming language
is impossible.

A common feature of the grammar of programming languages is that a lot of rules of
the grammar have recursive references; A rule may reference itself, or two rules may
reference each other. If we consider the definition of a statement in the C programming
language (listing 2.16), a statement is composed by various types of statements, one of
which can be a selection statement. However, a selection statement can also contain
additional statements. Defining these rules incrementally is impossible, since each one
needs the other to be defined beforehand.

```
statement :
  labeled−statement
  compound−statement
  expression−statement
  selection−statement
  iteration−statement
  jump−statement

selection−statement :
  if ( expression ) statement
  if ( expression ) statement else statement
  switch ( expression ) statement
```

Listing 2.16: Extract from the C99 language specification for the statement rule

Lpeg provides the ability to arrange the patterns in a Lua table, in order to define gram-
mars with recursive references. Each key, or index, of that table is a string which serves
as the name of a rule, and the associated value is a normal pattern. A pattern of a rule
can reference another rule, by specifying its name as an argument to the lpeg.V(string)
function. For example, one can define the rules for matching a statement as shown in
listing 2.17.

The definition of rules in Lua tables effectively creates Open Grammars. Any refer-
ences to rules with the lpeg.V() function are not resolved immediately. They just server
as place-holders, or open references. In order for the grammar to become suitable for
parsing, two more steps must be taken:

- Specification an entry point rule of the grammar, from which parsing of any input
  will begin.
- Conversion of the grammar into a pattern

The entry point rule of a grammar can be defined by setting the index key _I_ to the table
which contains the grammar. The value is a normal pattern, and can either be a single
reference to another rule, or a composition of other rules. Once all of the rules of the
grammar are defined in the table of the grammar, it can be converted into a pattern with
a call to the \texttt{lpeg.P}. The function will resolve all of the open references to rules and generate the final pattern which is ready to be used with the \texttt{lpeg.match()} function.

\begin{lstlisting}[language=lua]
require("lpeg")

−− Shortcuts to \texttt{lpeg} functions
V = lpeg.V
P = lpeg.P

−− Create a table for the grammar
C99 = {}

−− Statement rule
C99["statement"] = V("labeled_statement")
+ V("compound_statement")
+ V("expression_statement")
+ V("selection_statement")
+ V("iteration_statement")
+ V("jump_statement")

−− Selection Statement rule
C99["selection_statement"]
= P("IF") * P("(") * V("expression") * P(")")
* ( P("ELSE") * V("statement") )^−1
+ P("SWITCH") * P("(") * V("expression") * P(")")
* V("statement")

−− Definition of the rest of the rules follows after this point ...

−− Definition of the entry-point rule.
In C it is the 'translation_unit' rule
C99[1] = V("translation_unit")

FinalisedGrammar = P(C99)
\end{lstlisting}

Listing 2.17: Example of writing a grammar to match statements with \texttt{Lpeg}
Chapter 3

Design And Implementation Of Compiler

As discussed in section 2.2 of this dissertation, there are a number of strategies available which can be used in order to parallelise a region of nested loops, each one with its strengths and limitations. However, parallelising a code with one of these strategies does not take into consideration any runtime characteristics during the execution of the program - mainly the amount of parallelism which is exposed by the number of iterations of each loop and the number of available threads.

In order to make a dynamic decision on the parallelisation strategy to be used, we have developed a source-to-source compiler, accompanied by a small runtime library, which is able to produce code which can exploit the runtime information mentioned above. Our implementation is currently limited to parallelising a single loop of a nested loop region, taking advantage only the Outermost and Inner loop strategies. The remainder of this chapter focuses on the design and implementation details of the developed source-to-source compiler.

3.1 High Level Overview

The source-to-source compiler acts as a preprocessor to a C code which can contain OpenMP directives, as well as our own pragma extensions, discussed in section 3.2. The compiler parses the code, and creates an internal representation of the code in the form of an Abstract Syntax Tree (AST). The regions of the input code that contain our pragma directives are translated into semantics of the C programming language and OpenMP directives during the parse phase, and appropriate nodes for these regions are placed in the AST. The created AST is then translated back to C code which now only contains OpenMP directives.

The choice to use OpenMP as the underlying target platform for the parallel logic of the code was made in order to keep our compiler simple and remain within the time
frame of this project. The development of the full OpenMP standard in terms of Posix Threads is out of the scope of our work and it's a topic for a dissertation on its own. Furthermore, we are focusing in the idea of a dynamic choice on whether a loop should execute sequentially or in parallel. The development of a parallel platform along with our solution would have introduced another variable that must be taken under consideration; Whether or not the idea of a dynamic choice in user code is useful as well as whether or not our parallel platform was efficient enough. By eliminating the latter, we are focusing on the idea of dynamic choice itself. Consequently, this choice makes the generated code of our compiler portable to all the existing compilers that support the OpenMP programming model.

One concern that was raised during the development of our compiler was how to handle preprocessor directives. A C code, even in its simplest form, must first be preprocessed by the C Preprocessor. This step is needed in order to include any library headers used by the code, expand the preprocessor definitions and macros, as well as to remove regions of code that reside in regions of disabled preprocessor definitions. Although the C Language Specification provides the grammar of the C Preprocessor, compilers handle these directives differently, often providing their own extensions. We solve this problem by accepting only preprocessed code in order to maintain the flexibility and portability of our compiler.

The full compilation cycle of a code with our extensions is depicted in figure 3.1. The input code is first passed through the C Preprocessor which handles all the preprocessor-related parts. The preprocessed code is ready to be used by our source-to-source compiler, which handles our custom directives. The final code, containing only C code with OpenMP directives can be used as an input to a C compiler in order to produce the final executable.

### 3.2 Custom Directives

Similar to the work of [12] we exploit the idea of using directives in order to specify the available parallelism within a region of nested loops. However, instead of operating directly on OpenMP directives, which begin with `#pragma omp`, we have chosen to define a custom directive for a number of reasons:

- We want to make an explicit distinction between the parallel loops defined with OpenMP and the parallel loops defined with our custom directive since their execution has different semantics.
- We want to provide the ability for a code to keep the semantics of OpenMP in parts where our functionality is not needed.
- We want to provide additional clauses in our directive which are not defined by the OpenMP standard.
Figure 3.1: Compilation process with the source-to-source compiler being involved

As such, our directive begins with `#pragma preomp`. A loop that is preceded by a `#pragma preomp for` directive is considered by our compiler as a suitable candidate for applying parallelisation. When such a loop is found, the compiler performs the necessary code transformations so that a decision can be made at runtime whether the loop should run sequentially or in parallel. If we consider the example code in listing 3.1, although it looks like standard OpenMP code, the semantics are very different.

Since we use OpenMP as the target platform in order to apply parallelism on the code, our directive supports the full grammar definition of the OpenMP Loop Construct. Furthermore we also extend the grammar to support an additional clause, the `parallel_threshold(expression)` clause. The clause is optional, and when it is not present the compiler will assume a default value of 1.0, which will be given as argument to the decision function. If the threshold clause is present, then the provided expression will be used as an argument instead. This value is passed as an argument to a function of our runtime library, discussed in chapter 4.
3.3 Code Transformations Of Loop Regions

When a loop with a preceding `#pragma preomp` directive is found, the compiler performs a number of transformations on the AST in order to enable the choice of parallel or sequential execution of the loop at runtime. In addition to this, a simple analysis of the loop is performed in order to facilitate the computation of a loop’s iterations during the making of the decision.

3.3.1 Loop Analysis

Loop analysis is necessary in order to extract the required information for calculating the iterations of the loop at runtime, right before its execution. OpenMP already places a restriction on the format of a for loop, discussed in section 2.1.2, and the compiler is able to handle all the mentioned cases. The basic equation for calculating the iterations of a loop is as follows:

\[
\text{Iterations} = \frac{(\text{loop}_\text{end}) - (\text{loop}_\text{start})}{\text{loop}_\text{increment}}
\] (3.1)

If we consider the example code in listing 3.2, then the value of `loop_start` is 0, the value of `loop_end` is the `n` variable, and the `loop_increment` is 1. More complicated loops need adjustment to these parameters. For example, if the `loop_end` value is inclusive in the condition of the loop, with `’<=’` or `’>=’`, the dividend of the equation must be adjusted. Therefore, our compiler is able to extract the information exhibited in table 3.1. This information, along with the value of the parallel threshold previously discussed, is used as an argument for the functions of the runtime library. It is worth mentioning that the actual evaluation of the above expressions is performed at runtime in order to capture any changes in the loop’s iterations.

```c
for (i = 0; i < n; i++) {
    work();
}
```

Listing 3.2: An example of a C for loop
### Table 3.1: Extracted loop information for calculating the number of its iterations

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>loop start value</td>
<td>The expression used for the initial value of the loop’s variable</td>
</tr>
<tr>
<td>loop end value</td>
<td>The expression used for the last value of the loop’s variable</td>
</tr>
<tr>
<td>loop increment value</td>
<td>The expression used to specify the amount of change of the loop’s variable during each iteration.</td>
</tr>
<tr>
<td>loop end inclusiveness</td>
<td>Whether or not the end value is included in the loop’s variable</td>
</tr>
<tr>
<td>subtraction of increment</td>
<td>Whether or not the increment value is subtracted or added to the loop’s variable.</td>
</tr>
</tbody>
</table>

#### 3.3.2 Using OpenMP If Clause

The first method uses the already provided *if clause* of the OpenMP Parallel Construct. A custom directive is translated to an OpenMP Parallel For directive, attached with an *if* clause in order to decide whether to execute the loop in parallel or not. An example output of this transformation is exhibited in Appendix A.1. The expression of the *if* clause consists of a call to a decision function of our runtime library. The decision function takes the *evaluated* expressions of the loop’s information in order to make a decision.

The main idea behind this approach is to utilise the functionality which is already provided by OpenMP as much as possible, automating only the necessary parts for the decision making. In addition to this, the same code is used for both the serial and parallel execution of each loop.

A major drawback of this approach, however, is that a parallel region will be created regardless of whether a loop is parallelised or not. Considering the example in fig-
ure 3.2, parallelising the outer loop of two nested loops with two threads will result in three parallel regions. Each thread of the outer region will create a new parallel region and become its master. In the case of the inner loop being parallelised, two parallel regions are created. For nested regions with a larger number of loops this method has the potential to produce excessive parallel overheads.

3.3.3 Using Code duplication

This approach uses code duplication in order to provide a separate serial and parallel version of each loop. The idea is to avoid the creation of unnecessary parallel regions when a loop is executed in a sequential fashion. When a loop is preceded by a `#pragma` `openmp for` directive, the loop is duplicated and wrapped in a normal if-else statement. The expression of the if statement consists of a call to a decision function provided by our runtime library and the information extracted from the analysis of the loop is provided as arguments.

From the example in Appendix A.2 it is observed that the output code can get very big. If we number the loops according to their level starting from level one for the outermost loop, the appearances of each loop can be calculated with the equation 3.2. The total number of loops is then shown in equation 3.3

\[
\text{Loop Appearances} = \text{level}^2 \quad (3.2)
\]

\[
\text{Total Number Of Loops} = \sum_{\text{level}=1}^{\text{number of levels}} \text{level}^2 \quad (3.3)
\]

Each loop which is duplicated must also appear in both versions of the outer loop. This is made more clear if the structure of the code is represented as a graph, shown in figure 3.3. There are two versions of the outer loop for the serial and parallel execution. The inner loop however has four versions in total. This results in a bigger execution file, which might have a negative impact on the cache memory of the system. However, this approach provides the additional flexibility of including code before and after each version of the loop, a functionality that is not supported by the previous method.
Another advantage of this approach is that the serial versions of a loop will be inlined within the created function of the enclosing loop which has an OpenMP directive. If we consider the graphical representation of the code, each arrow denoting a parallel loop will result in a function. The function of the outer loop, however, will contain an inlined version of the inner loop for its serial execution, as well as a call to the function of its parallel version. In contrast to the if clause approach discussed previously, where each loop is transformed into a separate function when the executable is created, this method provides additional flexibility for the compiler in order to apply more optimisations on the loops.

### 3.4 Code Design

Our primary goal during the design of the compiler was to provide a high level of flexibility, extensibility and re-usability. As such, instead of developing a monolithic systems with components being tightly coupled together, we have implemented a set of libraries which form a small framework for parser construction. Furthermore, we have chosen to implement the libraries as Classes in order to take advantage of inheritance between related Classes when such relation is needed. Similar to conventional compilers, a driver then uses these components to implement the business logic of the executable.

The compiler is composed by two categories of functionality which serve as namespaces to the implemented Classes and Modules. The base category contains base Classes and Modules which are used as the building blocks for constructing grammars
and drivers. The language category contains Classes that are used during the parsing of a programming language. The relationship of between the components is shown in figure 3.4. The remainder of this section describes the Modules and Classes of the two namespaces.

Figure 3.4: UML diagram of the source-to-source compiler components

### 3.4.1 Module base.Class

The module contains a single function, named `Class(super)`. This function provides the functionality for creating Classes within our framework. In other words, it implements the Class System used by the compiler. Single inheritance is allowed by providing the super Class as an argument to the function. When the function finishes its execution, a Lua table is returned, which represents the new Class. This table is callable and can be used as a function in order to create objects.

The table of the Class is constructed in such a way that a number of methods can be implemented by the Class for specialised functionality. The method `__init()`, when implemented by the Class serves as the constructor of the instances of the Class. Similar to this, the method `__class_init()` can be implemented in order to execute code when this Class is inherited by another Class.
### Table 3.2: base.Class Module fields

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>base.Class(super)</td>
<td>Function</td>
<td>Returns a table which can be used as a Class. Super is optional and can be used for inheritance by providing the table of the Super Class</td>
</tr>
</tbody>
</table>

#### 3.4.2 Module base.Driver

The module provides functionality which helps the development of drivers. The functions which are provided can be used to read the contents of a file, write text to an output file and handle command line parameters. The functions for each operation are highlighted in table 3.3

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Driver.read_file(filename)</td>
<td>Function</td>
<td>Reads the content from filename and returns it as string representation</td>
</tr>
<tr>
<td>Driver.write_file(filename, content)</td>
<td>Function</td>
<td>Writes content to filename.</td>
</tr>
<tr>
<td>Driver.getopts(args, optstr)</td>
<td>Function</td>
<td>Handles command line options found in the args table based on the description provided by the optstr string. The returned value is a table which can be indexed by the character of each argument.</td>
</tr>
</tbody>
</table>

Table 3.3: base.Driver Module fields

#### 3.4.3 Class base.AstNode

Base Class which is used by the Classes of each AST node for the construction of the AST. The Class exposes a single method, named `dump()` which prints, or `dumps` in compiler lingua, the generated AST starting from that node. The function is available in order to facilitate testing as it can show the AST of a code in its complete form as it is stored in memory.

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AstNode:dump()</td>
<td>Method</td>
<td>Recursively prints the generated AST of a node. The method must be used on objects.</td>
</tr>
</tbody>
</table>

Table 3.4: base.AstNode Class fields
3.4.4 Class base.Grammar

Base Class which is used by grammars. The Class implements a single function, \texttt{copy(class)} which copies the rules and actions of a Class to another Class. It is used by the \texttt{__class_init()} function of each Class that implements a grammar, in order to allow inheritance between grammars. The function expects each grammar to be implemented so that the rules of the grammar reside in the \texttt{rules} table of the Class, and the actions for each rule to be in the \texttt{actions} table.

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Grammar:copy(class)</td>
<td>Method</td>
<td>Copies the rules and actions of a grammar Class into a new Class.</td>
</tr>
</tbody>
</table>

Table 3.5: base.Grammar Class functions

3.4.5 Class language.c.C99

The Class implements the full grammar of the C Programming Language according to the C99 Specification. Our implementation of the grammar is based on the work of [6] which provide an implementation of the C99 grammar specification for their tool. We have chosen to use this grammar as a basis of our implementation with the Lpeg parsing library, since the authors state that their implementation was successfully tested on a number of preprocessed open-source projects.

Each rule of the language resides in the rules table of the Class and the associated semantic actions in the actions table. The two tables can be indexed by the name of a rule in order to access the associated pattern of that rule and its semantic action. For example, the pattern which describes the parsing of a 'statement' is located in \texttt{rules["statement"]} and the associate semantic action, which is implemented as a function capture, is located in \texttt{actions["statement"]}.

Unlike the examples previously presented in section 2.5.1, the captures of each rule are not directly applied on the pattern of a rule. This choice was made in order to allow a derived class to modify either the pattern or the semantic action of a rule separately. The \texttt{parse(config)} method, which is used by instanced of the Class, combines the patterns with their semantic action upon its execution. The method acts as a wrapper to the Lpeg.match() function and finalises the grammar based on the options of the provided \texttt{config} table.

The parse function expects the provided config table to have at least the \texttt{config["subject"]} field set, which contains the input source code to be parsed as a string. In addition to this, the table may contain the optional fields of \texttt{config["position"]}, \texttt{config["entry"]} and \texttt{config["typedefs"]}. These options can be used to to specify the initial offset to begin parsing from, the entry point rule and any externally defined type definitions the parser
must know about which are not defined in the input source code. When the function
finishes its execution, the generated AST of the source code is returned.

Unlike the 'position' and 'entry' fields, which are mainly present for development and
testing purposes, the 'typedefs' field is of great importance. Type definitions in the C
language are a very well known ambiguity of the language. They can be seen as a
way to extend the language dynamically at run-time. Therefore, the grammar of the
language alone cannot be used to parse an input source code. The parser is required
to keep track of type definitions during parsing, in order to correctly parse the code.
Since our compiler only implements the C99 specification, it might not be able to parse
code that is generated by the preprocessor of a conventional compiler. The reason for
this is that each compiler provides its own extensions to the C99 specification. Even if
the user code does comply with the C99 specification, the included header files in the
code might not. One way to overcome this problem is to separate the code that needs to
be handled by our compiler with code that possibly includes unsupported header files.
However, if the code uses type definitions which are located in one of the unsupported
header files, our compiler will still not be to parse the code. In this case, a driver can
read a list of type definitions from the command-line, and provide this information to
the parse() method.

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>C99.rules</td>
<td>Table</td>
<td>Table which contains the rules of the grammar</td>
</tr>
<tr>
<td>C99.actions</td>
<td>Table</td>
<td>Table which contains the semantic actions of each rule</td>
</tr>
<tr>
<td>C99:parse(config)</td>
<td>Method</td>
<td>Method which is used by objects to begin parsing. It returns the generated AST on completion.</td>
</tr>
</tbody>
</table>

Table 3.6: language.c.C99 Class fields

3.4.6 Class language.c.PreOMP

The Class is used for parsing OpenMP directives as well as our custom directive, dis-
cussed in section 3.2. It subclasses the C99 Class, therefore it contains the full imple-
mentation of the C99 grammar as well. OpenMP directives are placed on the generated
AST as normal nodes. In contrast to this, custom directives are processed and translated
in terms of C and OpenMP AST nodes. This Class acts as an extension to the C99 class.
As such, it merely extends the C99 grammar by implementing some additional rules in
the grammar.

In order to support the code transformations during parse time and avoid the need of
an additional pass on the AST, we have decided to extend the statement rule of the C
grammar. Further to the standard iteration constructs of the language, it also accepts a
for loop preceded by our custom directive. This proved to be a good design choice as
nesting of duplicated loops is handled automatically by the grammar. Given a region
of nested loops, code duplication will begin from the innermost loop, constructing the
necessary AST nodes. These nodes will be then passed to the semantic action of the loop one level above, which will also perform code duplication of its loop for serial and parallel execution, including the received nodes in both versions. The same operation is performed during the *if clause* code generation mode. However, no code duplication is performed. The compiler merely attaches an if clause to the OpenMP directive of each loop, starting from the innermost loop level. The AST nodes of the loop with the directive are passed as arguments to the semantic action of the outer loop.

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PreOMP.rules</td>
<td>Table</td>
<td>Table which contains the rules of the grammar</td>
</tr>
<tr>
<td>PreOMP:actions</td>
<td>Table</td>
<td>Table which contains the semantic actions of each rule</td>
</tr>
<tr>
<td>PreOMP:parse(config)</td>
<td>Method</td>
<td>Method which is used by objects to begin parsing. It returns the generated AST on completion.</td>
</tr>
</tbody>
</table>

Table 3.7: language.c.PreOMP Class fields

### 3.4.7 Class Container language.c.C99.Ast

This Module serves as a container to Classes which are used as the nodes of the AST. It is used by both the C99 and PreOMP Classes, which require this functionality during the parsing of an input code.

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ast</td>
<td>Table</td>
<td>Table which contains the classes for constructing the AST</td>
</tr>
</tbody>
</table>

Table 3.8: language.c.C99.Ast Class Container fields

### 3.4.8 Class language.c.C99.CodeGen

The Class servers as the back-end of our compiler. It exposes a single method, named *generate(node)*, which receives as argument an object of one of the AST Classes. The method will traverse the node and translate back to C code, returning it as a string upon its completion.

Internally, the module implements functions which translate AST nodes back to C code individually. For example, there is a function that can generate a 'for' statement from an object of the Ast.For Class. Similarly, another one exists for the translation of a 'while' statement. The *generate(node)* method uses the *Visitor Design Pattern*[15], in order to find the appropriate code generation function of a node based on its Class.
<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CodeGen.generate(node)</td>
<td>Method</td>
<td>Generates the C code of the provided AST node. The code is returned as a string representation.</td>
</tr>
</tbody>
</table>

Table 3.9: language.c.C99.CodeGen Class fields

### 3.5 Driver

The driver of the compiler acts as the main executable. It uses the components discussed previously in order to create objects of the classes, read the command line arguments provided by the user and parse the input code in question producing the final output. Since the code is written in the Lua Programming Language, the driver must be executed as a script to the Lua interpreter. The usage of the driver is simple. It accepts a number of options, listed in table 3.10, and an input as the last argument.

```lua
lua preomp [options] input
```

Listing 3.3: Usage of the driver

<table>
<thead>
<tr>
<th>Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-d</td>
<td>Prints the generated Abstract Syntax Tree</td>
</tr>
<tr>
<td>-h</td>
<td>Prints the usage message</td>
</tr>
<tr>
<td>-m genmode</td>
<td>Instructs the parser to generate code according to a specific mode</td>
</tr>
<tr>
<td>-o file</td>
<td>Writes output to file</td>
</tr>
<tr>
<td>-t identifier</td>
<td>Instruct the parser to consider 'identifier' as a typedef</td>
</tr>
</tbody>
</table>

Input

<table>
<thead>
<tr>
<th>Description</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>filename</td>
<td>Reads content from filename</td>
</tr>
<tr>
<td>-</td>
<td>Reads content from stdin</td>
</tr>
</tbody>
</table>

Output

<table>
<thead>
<tr>
<th>Description</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>filename</td>
<td>Writes the generated code to filename</td>
</tr>
<tr>
<td>-</td>
<td>Prints the generated code to stdout</td>
</tr>
</tbody>
</table>

Code Generation Mode

<table>
<thead>
<tr>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>duplicate</td>
</tr>
<tr>
<td>duplicate_prof</td>
</tr>
<tr>
<td>if_clause</td>
</tr>
</tbody>
</table>

Table 3.10: Driver Command Line Options

The options are self explanatory. They handle the input and output modes of the driver. As mentioned in section 3.4.5, if a type definition is needed in order to parse the input code, it can be specified with the `-t` flag. Unlike all the other flags, this one can appear multiple times for specifying different type definition. The code generation modes are associated with the decision functions of the runtime library, discussed in chapter 4.
3.6 Testing

In order to test the correctness of the compiler, we used various methods to ensure that each part of the system worked as expected.

The parsing phase was tested with the use of the \textit{-d} flag of the driver which outputs the full representation of the generated AST in a readable format. This feature of the compiler proved to be a valuable addition during the development of the PreOMP Class. Unlike the C99 Class which constructs the AST automatically from the semantic actions of the matched rules of the grammar, the code transformations in the PreOMP Class are performed manually. In order to transform a loop which is preceded by custom directives in terms of C AST nodes, new objects of the nodes needed to be created by user code. Having the flag already available in the compiler was very useful since we could test the correct form of the created AST nodes by comparing it with the generated output of similar input codes.

The C99 grammar, along with the PreOMP extensions and the back-end code generation engine were tested with codes of varying complexity. We have manually checked the final code of the compiler for each input and verified that the generated C code was indeed correct. As a last step, we compiled the output code with a real C compiler and made sure that the produced result after the execution was the expected one.
Chapter 4

Runtime Library

The runtime library implements the logic for deciding which version of a loop to choose right before its execution. As discussed previously, our compiler only performs code transformations on the original code, including code for function calls to the decision functions of this library. A code that has been preprocessed by the source-to-source compiler must then be linked with our runtime library. The remainder of this section focuses on the functionality of this library.

4.1 Decision Based On Heuristics

This method uses heuristics based on information which are collected at runtime in order to decide whether a loop should execute sequentially or in parallel. The idea of this approach is to look for the first loop that has enough iterations to utilise all of the available threads. Before the execution of a loop, the decider checks whether a loop of an outer level is already running in parallel. If this condition is met, then the loop is serialised. In the case that no outer loop is running in parallel the number of iterations of the loop is calculated and it is divided with the available number of threads. If this results in a value that is greater than or equal to a specified threshold, then the parallel version of a loop is chosen, otherwise the loop is serialised. As discussed in section 3.2, the default value of the threshold is 1 which requires each thread to get at-least one iteration in order to choose the parallel version of the loop.

The calculations of the iterations is based on the parameters of the loop which are extracted by the source to source compiler and are provided as arguments to the decision function. In the case that the original code of the loop uses variables for its boundaries, any change in their value will also be captured by the decision function during the calculation. This design allows constant monitoring of any changes in the iterations of the loops which also results in dynamic adaptation of the parallelisation strategy during the execution of the program.

The algorithm is very simple and with minimum overheads. Moreover, there is no need
to maintain any state for the loops. However, the logic which is used by the function is based on optimism. It only considers the amount of parallelism exposed by the loop regardless of whether the amount of work of the loop is big enough to justify any overheads. The pseudo code for this decision function is shown in listing 4.1.

```c
1 heuristics (loop information, threshold)
2 {
3   decision = serial version
4   if (!omp_in_parallel())
5     {
6       iterations = calculate_iterations(loop information)
7       if (iterations / number of threads >= threshold)
8         {
9           decision = parallel version
10         }
11     }
12   return decision
13 }
```

Listing 4.1: Pseudo-code for decision based on heuristics

4.2 Decision Based On Heuristics With Profiling

This approach is a more elaborate version of the Heuristics decider. In the same manner as the heuristics decider, it uses the same information extracted by the source to source compiler in order to determine whether the loop should be parallelised or not. However, if a loop does not meet the conditions, then the function falls back to profiling mode in order to decide which version of the loop, serial or parallel, to choose from based on timings.

The pseudo code for this approach is shown in listing 4.2. At the beginning of the function, each thread calculates the number of iterations that is going to execute and saves the information into a data structure which serves as the descriptor of the loop. Then the function proceeds in the same manner as the heuristics decider. The first time a loop is executed, the heuristics decider determines if the loop should be parallelised. If the conditions are not met, the sequential version of the loop is chosen and profiling is enabled for this loop. At the next execution of the loop, the evaluation of the heuristics is still performed. If the conditions are still not met (for example there where no changes in the iterations of the loop), the loop is now parallelised since there is only timing for the serial version. Consecutive executions of the loop will first check the heuristics conditions by always calculating the number of iterations of the loop and dividing it with the number of threads, falling back to profiling mode if the condition is not satisfied. However, the function will detect that timings for both versions are available. If both versions of the loop have performed the same amount of work, the timings are
considered valid, and the fastest version is chosen as the final decision. In contrast to this, if the amount of work is not the same the timings get invalidated, and profiling is re-initiated.

heuristics_with_profiling ( loop information , threshold )
{
    decision = serial version
    iterations = calculate_iterations ( loop information )
    id = get_thread_num ()
    loop_level = get_loop_level (id)
    state = loop_state[loop_level]

    save_iterations (state , id , iterations )

    if ( !omp_in_parallel ( ) )
    {
        if ( iterations / number of threads >= threshold )
        {
            decision = parallel version
        }
        else
        {
            if ( have_serial_time ( state ) && have_parallel_time ( state ) ) {
                if ( serial work != parallel work ) {
                    reset_times ( state )
                }
            }
            if ( !have_serial_time ( state ) )
                decision = serial version
            else if ( !have_parallel_time ( state )
                decision = parallel version
            else
                decision = min ( serial time , parallel time )
            enable_profiling ( state )
        }
    }

    return decision
}

Listing 4.2: Pseudo-code for decision based on heuristics with profiling

Figure 4.1 demonstrates an example of three nested loops, where only the inner-most loop has enough iterations to satisfy the Heuristics conditions. When Loop 1 is reached, the serial version is chosen and profiling for this version is started. During the first execution of Loop 2 (i = 0), heuristics will also fail, causing it to choose the serial version as well. Profiling is also started for this loop. The third loop can be parallelised therefore during the serial execution of Loop 2 (j = 0 to 2), Loop 3 will be executed in parallel. When Loop 2 finishes its execution, the execution time for its serial execution is saved, and Loop 1 will proceed to its second iteration (i = 1). The decider of Loop 2
will still fail on the heuristics condition. However, it will detect that the serial version of the loop has already been profiled and so it will choose the parallel version of the loop in order to profile it as well. Parallelising Loop 2 causes Loop 3 to run sequentially. When the parallel execution of Loop 2 is finished, the time for the parallel execution will be saved in the state of the loop and Loop 1 will continue at its third iteration ($i = 2$). The decider for Loop 2 fails once more, falling back to profiling mode once again. In this case, it is detected that both versions of a loop have already been profiled. The fastest one will be chosen as the final decision.

We can observe from the figure that in order to actually utilise the functionality of the profiler, a loop must be nested within another loop. In the case of Loop 1, although profiling is performed for the serial version, the parallel version is never executed since it is the outer-most loop. Moreover, the best time of the two versions will be chosen after the second execution of a loop. Therefore, the outer loop must have at least three iterations. For Loop 3, profiling will never be performed as it has enough iterations to satisfy the conditions of the heuristics decider.

During the profiling of the serial path of an external loop, the measured timing also includes the slow versions of the internal loops which need profiling. In ideal conditions, profiling needs to be made for each execution path individually in order to measure the correct time. However, this leads to a tree-based decision. Since this operation is performed at runtime during the execution of the code, the allocation of more iterations for
the profiling would become really costly for the execution time, as more work would be serialised in order to take measurements for each execution path. In order to minimise the impact of profiling on the execution time, we have decided to take a less accurate approach which has the potential of making the wrong decision. It should be noted however that a wrong decision does not affect the results of the code.

In order to support this functionality, the following additional logic was required:

- A way to keep track of what loop level a thread operates in.
- A way to start and stop the profiling of each version of a loop
- A way to count the amount of work performed by each version of a loop.

The above requirements are met by including additional function calls to the generated output code of the source to source compiler before and after each loop version, serial or parallel, when this decision function is used. Since this distinction of the loops is necessary, the decision method was only implemented for the code duplication mode of the compiler.

**Entering A Serial Path**

A function call is placed before the execution of the serial version of each loop. The purpose of this function is to keep track of the loop level by incrementing an internal counter, as well as to mark the starting time for the loop in the case where the decision function has decided to enabled profiling. The pseudo code of the function is shown in listing 4.3.

This function may be called within an outer loop that is already executing in parallel. Therefore, each thread needs to keep track of its own counter for the loop level. Internally, the runtime library defines this data structure for each thread. When profiling is enabled for the loop, we are certain that no outer loop is running in parallel, as exhibited in listing 4.2. In this case, timing will only be performed by the master thread, which is the only one active. We have chosen to use the timing functions which are provided by the OpenMP API, since they have good resolution.

```
1  enter_serial_path ()

2 [ 
3     id =omp_get_thread_num ( )
4     increase_loop_level ( id )
5     loop_level = get_loop_level ( id )
6     state = loop_state [ loop_level ]
7     if (profilling_is_enabled ( state ) ) { 
8         mark_start_of_serial_time ( state )
9     }
10 ]
```

Listing 4.3: Pseudo-code for function before the execution of a serial loop
Exiting A Serial Path

A function call is also placed at the end of the serial version of a loop in order to complement the one that is called before its execution, as shown in listing 4.4. The very first functionality of the function is to mark the ending time of the loop’s execution when profiling is enabled. This is performed by the master thread since profiling mode is enabled only if all of the outer loops run sequentially.

Each thread that executes this function first checks whether or not any amount of work has been produced by the execution of any internal loops. If no work has been reported, this denotes that the current loop is the innermost in the loop region. In this case, the number of iterations that have been executed for this loop is used as the amount of work performed by the sequential execution of the loop. The computed value is saved in the state of the loop, which is used by the decision function when the comparison of the timings is performed. If the loop is not the outermost one in the nested region, this value is also saved in the state of the loop of the outer level.

```c
exit_serial_path ()
{
    id = omp_get_thread_num ()
    loop_level = get_loop_level (id)
    state = loop_state [loop_level]

    if (profiling_is_enabled(state)) {
        mark_end_of_serial_time (state)
    }

    work = get_work_produced_by_internal_loops (state, id)

    if (work == 0) {
        /* We are the innermost loop */
        work = get_number_of_iterations (state, id)
    }

    if (profiling_is_enabled(state)) {
        set_serial_work (state, work)
        disable_profiling (state)
    }

    if (loop_level > 0) {
        outer_level = loop_level - 1
        add_to_internal_work (loop_state[outer_level], id, work)
    }

    decrease_loop_level (id)
}
```

Listing 4.4: Pseudo-code for function after the execution of a serial loop

As stated in section 4.2, the timings of both versions of a loop are only considered
to be valid if both versions have executed the same amount of computations (work). This information is based on the number of times the body of the innermost loop in the region has been executed from the current loop downwards. Each time one of the loops finishes its execution, this information is saved in the loop’s state along with the measured timing, if profiling is performed for the loop. Furthermore, the information is also reported to the state of the immediate outer loop (one level up). This way, there is a constant flow of information on the amount of times the code in the body of the innermost loop has been executed. Obviously, this approach cannot capture any computations which might vary in-between the loops. But given the limited amount of time we had available for the project, our compiler merely acts as a preprocessor to the code; No source code analysis is currently being performed. Therefore, we only consider the structure of the loops preceded by our custom compiler directives. In listing 4.4, the mentioned behaviour is presented by the following functionalities:

- **get_work_produced_by_internal_loops**: Provides the number of times the body of the innermost loop has been executed from this loop level downwards.
- **set_serial_work**: Saves this value to the state of the current loop which will be used by the decision function to make sure that both timings have executed the same amount of computations.
- **add_to_internal_work**: adds this value to the state of the outer loop.

**Entering A Parallel Path**

In the same manner as with the case of executing the serial version of a loop, a function call is also placed before the execution of the parallel version. The functionality is similar to the function which is called before the serial version of the loop, however there is one additional detail that needs to be taken care of.

In order for the code to reach to a parallel version of a loop during its execution all of the outer loops must be executed sequentially. Therefore, up to this point only the master thread is active and the internal data structure for the counter of the loop level is only valid for this particular thread. The master thread must therefore update the value of the loop level for each thread that is going to be created at the execution of the parallel loop. Listing 4.5 exhibits this behaviour.
Listing 4.5: Pseudo-code for function before the execution of a parallel loop

**Exiting A Parallel Path**

Similar to the function call which is placed after the execution of a sequential loop, this also happens at the end of each parallel loop in order to perform a similar operation. The function marks the ending time of the execution which is stored in the descriptor of the loop. A slight difference appears in the way that the function computes the work produced by any internal loops. During the execution of the loop all of the threads are executing the loop in parallel, potentially reporting the amount of work they are producing. Thus, the amount of work of all threads is summed in order to compute the final value. The logic after this point remains the same as with the exit function of the sequential loops. If no work is reported by internal loops, the amount of iterations of the current loop is used as the value for the work instead. It is then saved in the descriptor of the loop if profiling mode is enabled and is reported to the loop of the outer level, if any exists.
exit_parallel_path ()
{
    loop_level = get_loop_level(master)
    state = loop_state[loop_level]

    if (profiling_is_enabled(state)) {
        mark_end_of_parallel_time(state)
    }

    for thread in number_of_threads {
        work += get_workproduced_by_internal_loops(state, thread)
    }

    if (work == 0) {
        /* We are the inner most loop */
        work = get_number_of_iterations(state, master)
    }

    if (profiling_is_enabled(state)) {
        set_parallel_work(state, work)
        disable_profiling(state)
    }

    if (loop_level > 0) {
        outer_level = loop_level - 1
        add_to_internal_work(loop_state[outer_level], master, work)
    }

    decrease_loop_level(master)
}

Listing 4.6: Pseudo-code for function after the execution of a parallel loop

4.3 Code Design

During the development of the runtime library, a number of considerations were raised with regard to the visibility of the data structures used by its functions. Exposing any variables relevant to the library in the user code, after the preprocessing phase of our source-to-source compiler, would require the compiler to alter the data sharing clauses of the OpenMP directives. Moreover, this would also require the compiler to perform additional modifications on the code in cases where the nested loop region was split into various functions. As such, we have decided to hide the data structures concerning the state of the library within the library itself.

A code that uses the runtime library must include the preomp.h header file, which exposes the interface of the provided functions. Out of all these functions, only the preomp_init( void) function is required to be added by the users in the original source code.
in order to initialise the library. All the other functions are placed *automatically* in the code which is produced by the source-to-source compiler.

### 4.3.1 Public Interface

This section of the report describes the available functions of the runtime library.

```c
void preomp_init(void)
```

The `preomp_init(void)` function is used for initialising the internal data structures of the runtime library. A program that needs to use the library must first call the function before a nested loop region is reached.

```c
void preomp_enter_serial_path(void)
void preomp_exit_serial_path(void)
```

The two functions are placed before and after the execution of a serial loop when profiling is used by the decision function. They serve the purpose of timing the execution of the loop, as well as counting the amount of work performed within the loop.

```c
void preomp_enter_parallel_path(void)
void preomp_exit_parallel_path(void)
```

Similar to the serial version of a loop, these functions are placed before and after the execution of its parallel version. In the same manner as the previous two functions, they perform the timing and counting of the amount of work of the loop.

```c
int preomp_parallelise_loop_heuristics_single(
    long lstart,
    long lend,
    long incr,
    int inclusive,
    int sub,
    double parallel_threshold)
```

```c
int preomp_parallelise_loop_heuristics_profiler_single(
    long lstart,
    long lend,
    long incr,
    int inclusive,
    int sub,
    double parallel_threshold)
```

These functions implement the decision functionality of the runtime library. The parameters they receive include the information of the loop which is extracted by the source-to-source compiler, as well as the parallel threshold value when it is specified as part of the `preomp directive`. 

45
4.3.2 Private Interface

The private interface of the library serves the purpose of defining the data structures for the state of the loops, as well as helper functions which are used by the ones of the public interface. The private interface is divided into three separate header files depending on the functionality they provide. The header files are then used by the main code of the library in order to create variables of the data structures and incorporate them into the main functions of the public interface.

Timer Structure

The file `preomp_timer_p.h` defines the data structure which represents a timer. This data structure is used when profiling mode is incorporated into the decision function. Accessing a variable of this structure is performed with the use of `preprocessor macro definitions` which are defined in the file as well. This choice was made in order to maintain the readability and extensibility of the code, since macros look similar to function calls, but also ensure that within the library itself the code is inlined.

Loop State Structure

The state of each loop is defined in an independent structure located in the file `preomp_loop_state_p.h`. It uses the data structure of the timer in order to incorporate the provided functionality for each loop descriptor. Similar to a timer, access to variables of this type is also performed with `preprocessor macro definitions`.

Library Configuration Options

We have chosen to use statically allocated memory in order to specify the maximum number of threads the library can handle, as well as the maximum number of loops a nested loop region can have. The choice was made in order to avoid the use of pointers which generally prevent the compiler from performing certain kinds of optimisations. Since most of the library’s functions are used in regions of nested loops, any reduction in the overheads of the library is beneficial.

However, we do provide the ability to change these parameters during the compilation of the library. The file `preomp_config_p.h` includes theses values as preprocessor definitions, which can be altered by either using compiler flags or by modifying their values in the file directly.

Internal Helper Functions
The two functions are used internally by the library in order to facilitate with the calculation of the available number of threads, as well as the calculation of the iterations for each loop. The function `preomp_calculate_num_available_threads()` is called by the `preomp_init()` function in order to calculate the number of requested threads when the application is executed. The `preomp_calculate_loop_num_iters()` function is called by each decision functions for the calculation of the number of iterations for the loop in question. Both functions use the INLINE preprocessor definition which can be defined by various compilers in order to force inlining of their code.

### 4.4 Testing

The library includes code which is placed in various parts of its functions in order to produce information on the progress of the implemented algorithms. We have used this information to verify that the decision functions produce the expected behaviour by executing a number of testing codes having loops with a small number of iterations. The debugging information can be enabled or disabled during the compilation of the library by defining or undefining the DEBUG preprocessor definition in the compilation command respectively.

Moreover, the library includes additional code in order to perform boundary checks when accessing the internal data structures of the library. The relevant codes can be enabled or disabled as needed with the use of the SAFETY_CHECKS preprocessor definition. We have used these guards during the development of the library to ensure that memory accesses are performed within the correct boundaries.
Chapter 5

Experimental Design

5.1 The System

For the purposes of this project we used Ness [5], a parallel machine which is operated by EPCC in order to support local projects.

The system is composed by two parts, a front-end for development and job submission and a back-end for job execution. The management of the two parts is handled by the Sun Grid Engine which allows submission of jobs from the front-end that must be executed on the back-end nodes in isolation.

The back-end part of the system is composed by two X4600 Shared Memory nodes. The central processing unit (CPU) of each node is an AMD Opteron processor of 16 processing cores which execute instructions at a clock frequency of 2.6GHz. Each core has 64K of first level cache memory (L1) used for data as well as 64K L1 cache memory for the instructions of an executable. In addition to the L1 cache memory, there is also 1 MB of second level unified cache (L2) available to each core individually which is used for both data and instructions. Moreover, each core can access up to 2GB of main memory.

5.2 The Test Codes

5.2.1 Synthetic Benchmark Code

A synthetic benchmark code was developed for the conduction of some experiments in order to simulate situations where nested loops are not perfectly nested. The code is very simple and it allows the alteration of the amount of work between two nested loops. Listing 5.1 shows the main part of the code.

The code is comprised of a nested loop region of three levels. The outermost loop
always executes sequentially and controls the total amount of iterations of the benchmark whereas the two inner loops are parallelised. The delay function merely performs a computational delay. Depending on the value of the received argument, the delay time of the function varies. Since this functionality is already present in the EPCC’s OpenMP Micro-benchmark Suit [1], we have decided to re-use the code of the delay function located in the delay.c file of the suit.

```c
for (i = 0; i < num_iters; i++) {
    for (j = 0; j < outer_iters; i++) {
        delay(outer_delayreps);
        for (k = 0; k < inner_iters; k++) {
            delay(inner_delayreps);
        }
    }
}
```

Listing 5.1: Synthetic Benchmark Code

**Supported Compiler Flags**

We have placed preprocessor directives before the two nested loops in order to provide a choice of the parallelisation strategy to be used for the produced code at compile time. When the code is compiled with the compiler flag `-DOMP_LOOP=1`, only the loop of level 2 is parallelised by placing an OpenMP directive directly above the loop. Similarly, when the compiler flag `-DOMP_LOOP=2` is used, the loop at level 3 is parallelised. Lastly, the compiler flag `-DWITH_PREOMP` places our custom directives to the loops of both levels in order to make the code suitable for being processed by our source-to-source compiler.

**Timings**

The execution time of the benchmark is measured with the use of the `omp_get_wtime()` function which is a part of the OpenMP runtime library. We measure the time of the whole nested region, placing a call before and after the execution of the `num_iters` loop.

### 5.2.2 Computational Fluid Dynamics Extract Code

The benchmark code is an extract of a finite-volume cell-centred structured Navier-Stokes code for undertaking Computational Fluid Dynamics (CFD) simulation. It is a structured mesh multigrid code which works with multiblock grids, and includes a range of CFD solvers including: steady state, time-domain dual time-stepping, frequency-domain harmonic balance, and time-domain Runge-Kutta. The code has the following general pattern for the computations within the code:
for (iter = 0; iter < n_iters; iter++) {
    for (block = 0; i < n_blocks; i++) {
        for (harm = 0; h < 2 * n_harms; h++) {
            for (j_cell = 0; j_cell < n_cell_j; j_cell++) {
                for (i_cell = 0; i_cell < n_cell_i; i_cell++) {
                    for (repeat = 0; n_repeats < repeat++;) {
                        /* Perform Computations */
                    }
                }
            }
        }
    }
}

Listing 5.2: CFD extract code

The loop parameters are as follows:

- **n_iters**: The total amount of iterations of the execution
- **n_blocks**: The number of blocks in the grid
- **n_harms**: The number of harmonics being used
- **n_cell_j**: The j’th dimension of the current block
- **n_cell_i**: The i’th dimension of the current block
- **n_repeats**: The amount of repetitions of each core calculation, which affects the execution time of each iteration.

Given the code can use different methods, as previously outlined, the range of these loops can vary. For instance when performing a time domain simulation the *n_harms* loop has a single iteration. However, when performing a harmonic balance simulation it can have a range of values, generally between 2 and 16. Furthermore, it is not uncommon to run large simulations with a single block, or a small number of blocks, meaning that the *n_blocks* loop has a very small number of iterations. Finally, each block can have different values for its dimensions.

The innermost *n_repeats* loop is an extension to the code which is used in order to vary the execution time which is required by each code calculation. This was placed in order to study the effect that has on different parallelisation strategies.

**Supported Compiler Flags**

Starting from the *n_blocks* loop, we have placed preprocessor directive before each loop in order to parallelise each one individually. When the preprocessor flag -DOMP_LOOP=1 is used during the compilation, only the *n_blocks* loop is parallelised with the appropriate OpenMP directive. In a similar fashion, the flags -DOMP_LOOP=2 to 4 can be used
to parallelise one of the remaining loops. Finally, when the flag -DWITH_PREOMP is specified, all of the mentioned loops are parallelised with our custom directives in order to be preprocessed by our source-to-source compiler. It should be noted that the outermost loop always executes sequentially.

Timings

Similar to the case of the synthetic benchmark code, the timing is performed with the use of the `omp_get_wtime()` function. Furthermore, the whole execution time of the nested loop region is measured by placing a function call of the timing function before and after the execution of the `n_iters` loop.

5.3 Compiler Flags

Portland Group Compiler

The main compiler which was used during this project is the Portland Group Compiler version 10.0.0. All of the codes running the benchmarks, as well as the runtime library of our source to source compiler used the following compilation flags:

- **-O4**: This flag enables serial optimisations of level four.
- **-c99**: This flag uses the C99 Programming Language Standard for compiling the input source code.
- **-mp**: This flag enables support for OpenMP pragma directives

Further to the above flags, the runtime library was compiled without any debugging code. The relevant compiler flags are:

- **-UDEBUG**: This flag disables the DEBUG preprocessor definition which prevents the runtime library for producing debugging information during the execution of a code.
- **-USAFETY_CHECKS**: This flag disables the SAFETY_CHECKS preprocessor definition which disables boundary checks in the operations of the runtime library.

GNU C Compiler

The GCC compiler was used for conducting the experiments with the if clause code generation mode. The Portland Group Compiler on Ness does not allow a thread team of a nested parallel region to have more than one threads when an outer region is serialised with the if clause. According to the OpenMP specification however, the if clause only
affects the number of threads that get assigned to a particular parallel region, not the thread teams of its nested regions.

The compiler flags which were used with this compiler are the following:

- **-O3:** This flag enables serial optimisations of level three.
- **-std=c99:** This flag uses the C99 Programming Language Standard for compiling.
- **-fopenmp:** This flag enables support for OpenMP pragma directives the input source code.

In the same manner as with the Portland Group Compiler, the runtime library was also compiled without any debugging flags.

### 5.4 Job Submission And Collection Of Results

We have used a batch script to submit the benchmarks for execution on the back-end nodes of the system. The configuration of Ness places a restriction on the amount of threads that can be requested by each job. Therefore, our benchmarks are limited to 16 threads. Despite this, in the test cases where less than 16 threads were used, our batch script still allocated all of the 16 cores in order to ensure exclusive access to the back-end node. This way we have limited the impact of other running jobs on our results. Moreover, each benchmark was executed 3 times, and the worst timing is considered, since this is the limiting factor for the execution time.
Chapter 6

Synthetic Benchmark

6.1 Parameters

The synthetic benchmark was used in order to test our implementations. When the inner loop is chosen to be parallelised, the computations between the outer loop and the inner loop are serialised. Depending on the amount of the work of the inner loop, as well as the amount of the work that is now serialised, this might introduce an increase in the execution time.

If we consider the example code in listing 5.1, the execution time of the code of the two internal nested loops when only the outer loop is parallelised with a certain amount of threads \(outer\_threads\) can be calculated as shown in equation 6.1. \(T_{pOuter}\) is the execution time when parallelising the outer loop, \(T_{outer\_work}\) is the time needed for the work in-between the loops and \(T_{inner\_work}\) is the time needed for the amount of work within the innermost loop.

\[
T_{pOuter} = \frac{outer\_iters \times (T_{outer\_work} + (inner\_iters \times T_{inner\_work}))}{outer\_threads}
\]  

(6.1)

In a similar fashion, when parallelising the inner loop using \(inner\_threads\), the execution time of the loops is shown in equation 6.2

\[
T_{pInner} = outer\_iters \times (T_{outer\_work} + \frac{inner\_iters \times T_{inner\_work}}{inner\_threads})
\]  

(6.2)

If we want to have a reduction in the overall execution time by parallelising the inner loop, the constraint in equation 6.3 must be satisfied. Solving the inequation in terms of \(T_{outer\_work}\) we can get the maximum allowed threshold of the execution time for the work of the outer loop as shown in equation 6.4. The full steps to reproduce the equation are presented in Appendix B. It is worth mentioning that this model is an ideal...
performance model, where the work is evenly distributed to the threads. In reality, the
time of $T_{\text{outer\_work}}$ might be affected by the presence of parallel overheads.

$$T_{p_{\text{Inner}}} < T_{p_{\text{Outer}}}$$  \hspace{1cm} (6.3)

$$T_{\text{outer\_work}} < \frac{\text{inner\_iters} \cdot T_{\text{inner\_work}} \cdot \left(\frac{1}{\text{outer\_threads}} - \frac{1}{\text{inner\_threads}}\right)}{1 - \frac{1}{\text{outer\_threads}}}$$  \hspace{1cm} (6.4)

In order to test our hypothesis, we measured the amount of time which is required by
the delay function for various values, as shown in table 6.1. Then, we have chosen 8
iterations for the outer parallel loop, and 16 iterations for the inner loop. The parameters
allow up to 8 threads to be used for the outer loop and up to 16 threads for the inner
loop respectively. Furthermore, the value of 24800000 iterations was chosen as the
amount of work for the inner loop, which takes approximately 0.041 seconds, since it is
big enough to overshadow any parallel overheads. Solving the equation 6.4 with these
parameters we have the constraint of $T_{\text{outer\_work}} < 0.0468$ seconds approximately.

<table>
<thead>
<tr>
<th>Delay Iterations</th>
<th>Approximate Time In Seconds</th>
</tr>
</thead>
<tbody>
<tr>
<td>12400000</td>
<td>0.022</td>
</tr>
<tr>
<td>24800000</td>
<td>0.041</td>
</tr>
<tr>
<td>49600000</td>
<td>0.079</td>
</tr>
<tr>
<td>99200000</td>
<td>0.15</td>
</tr>
</tbody>
</table>

Table 6.1: Execution time required by the delay function for various iterations

We then executed the synthetic benchmark code (listing 5.1) with the parameters shown
in table 6.2 using the previously mentioned values for the delay repetitions of the outer
loop. We have measured the execution time of each loop when it is parallelised man-
ually and use it as a reference to compare the two decision functions with the code
duplication mode of our source-to-source compiler.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>num_iters</td>
<td>100</td>
<td>Total number of iterations for the simulation</td>
</tr>
<tr>
<td>outer_iters</td>
<td>8</td>
<td>Iterations of the outer parallel loop</td>
</tr>
<tr>
<td>inner_iters</td>
<td>16</td>
<td>Iterations of the inner parallel loop</td>
</tr>
<tr>
<td>inner_delayreps</td>
<td>24800000</td>
<td>Delay iterations of the inner delay function</td>
</tr>
</tbody>
</table>

Table 6.2: Execution time required by the delay function for various iterations
6.2 Results

Figure 6.1: Synthetic Benchmark With Outer Loop Delay : 0s

Figure 6.2: Synthetic Benchmark With Outer Loop Delay : 0.022s

55
Figure 6.3: Synthetic Benchmark With Outer Loop Delay : 0.041s

Figure 6.4: Synthetic Benchmark With Outer Loop Delay : 0.079s
6.3 Discussion

From the graphs we can observe that the threshold value is correct. When the delay time of the outer work is less than the calculated threshold (figures 6.1, 6.2, 6.3) parallelising the inner loop with 16 threads is still faster than parallelising the outer loop with 8 threads. The two decision functions of our runtime library show the correct behaviour by making the right decision. Depending on the amount of the execution time of the outer loop’s delay function the difference of the execution time between parallelising the outer loop with 8 threads and the inner loop with 16 threads varies. In the case where no work is performed in the outer loop, as shown in figure 6.1, parallelising the inner loop with 16 threads reduces the execution time in half, producing a speedup of 1.96. However, by introducing work in the outer loop, the difference between the execution time becomes smaller, since this work gets serialised by parallelising the inner loop. When the delay time of the outer loop is 0.022 seconds, there is a speedup of 1.3 in the execution time when parallelising the inner loop with 16 threads instead of the outer loop with 8. When the amount of work is 0.041 seconds, the speedup is only 1.065.

When a certain amount of work is introduced in-between the loops (figures 6.2, 6.3, 6.4, 6.5) parallelising the inner loop with less than 16 threads leads to an increase in the overall execution. In these cases, the distribution of the iterations is not balanced and so the execution time is limited by the slowest threads. Comparing the execution times to the case of 8 threads when parallelising the inner loop, we can observe that this load imbalance produces exactly the same behaviour. Since the threshold value of 0.045 seconds is only valid when 16 threads are used this results in an increase in
the execution time. Moreover, as the amount of work gets bigger the impact on the 
execution time when parallelising the inner loop is also increased, since more work is 
now being serialised. In these cases, the heuristics decider makes the wrong choice 
since its decision only concerns the amount of iterations of the loops and the available 
threads. In contrast to this, when profiling is used in the decision function, it is correctly 
detected that the fastest execution time is achieved by not parallelising the inner loop.

In the case where the amount of work of the outer loop exceeds the calculated threshold 
(figures 6.4 6.5) parallelising the inner loop, even with 16 threads, increases the total 
execution time. The benefit from using 16 threads to parallelise the inner loop is not 
enough to justify the work that is serialised. The heuristics decider still makes the wrong 
choice, whereas the heuristics-profiler decider once again manages to correctly identify 
the fastest parallelisation strategy.

When the benchmark is executed with a number of threads up to 8, both deciders make 
the right decision since the outer loop has enough iterations to utilise the available 
threads. However, an interesting behaviour is observed in figures 6.1 and 6.2. When 6 
threads are available, it is better to parallelise the inner loop instead of the outer loop 
despite the fact that the latter has enough iterations for all of the threads. This behaviour 
is the result of uneven distribution of the loop’s iterations to the threads. The distribution 
of 8 iterations to 6 threads results in all of the threads to get assigned 1 iteration of the 
outer loop each, and 2 of the threads get assigned from 1 more iteration. The total 
execution time in this case is limited by the slowest threads, which is the time of 32 
itations; 2 iterations of the outer loop multiplied by 16 iterations of the inner one. 
Parallelising the inner loop with 6 threads however, 2 of the thread get from 2 iterations 
whereas the rest 4 of the threads get 3 iterations each. In this case, the total execution 
time of the parallel loops is the amount of time required by 24 iterations; 3 iterations of 
the inner loop multiplied by 8 iterations of the outer loop. Since both decision functions 
only utilise the heuristics decision they cannot exploit this opportunity.
Chapter 7

CFD Extract Code Benchmark

7.1 Comparison Of Code Generation Modes

7.1.1 Parameters

This test case was conducted with the GCC compiler. The parameters which were used for the experiment are shown in table 7.1. We used as a reference the timings when parallelising the n_blocks, n_harms and n_cell_j loops individually, and compare the execution time of the heuristics decision function for the two code generation modes of our compiler. In order to avoid cases of the iterations not being evenly distributed to the threads, we only consider cases of 2, 4, 8, 12 and 16 threads.

We also consider cases where blocks do not have the same shape by altering the values of the n_cell_j and n_cell_i loops. No alterations indicate that all of the blocks have a grid shape of 2496x8(j_cell x i_cell). An alteration of 1 denotes that the first block has a grid shape of 8x2496 and the rest blocks have a grid shape of 2496x8. Similarly, an alteration of 2 means that the first and third blocks have a grid shape of 8x2496 whereas the second and forth blocks have a shape of 2486x8.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>n_iters</td>
<td>500</td>
</tr>
<tr>
<td>n_blocks</td>
<td>4</td>
</tr>
<tr>
<td>n_harms</td>
<td>2</td>
</tr>
<tr>
<td>n_cell_j</td>
<td>2496 or 8</td>
</tr>
<tr>
<td>n_cell_i</td>
<td>8 or 2496</td>
</tr>
<tr>
<td>repeats</td>
<td>1 and 5</td>
</tr>
</tbody>
</table>

Table 7.1: CFD Extract Code Parameters For The If Clause
7.1.2 Results

One Repetition Of Core Calculations

Figure 7.1: Comparison of code generation modes - 4 blocks, 4 harmonics, 1 repetition, 0 alterations
The results from the graphs show a tremendous difference between the timings when the if clause is used and the code duplication mode. Not only is the case of the if
clause slower, but it also increases the overall execution time of the code. When 2 and 4 threads are available, only the loop of the outer level is parallelised in both code generation modes. However, the if clause mode produces a slower execution time than the code duplication mode. In both cases of 2 and 4 threads, using code duplication provides a speedup of 1.2 when compared to the time of the if clause mode.

When more than 4 threads are used, the parallelisation is applied on the \( n_{\text{cell}_j} \) loop. In contrast to the code duplication mode which produces an execution time similar to the case of statically parallelising the loop, the if clause mode is still slower. Comparing the times of the two code generation modes, when 8 threads are used the code duplication mode requires 17 seconds of execution time whereas the if clause mode requires 63 seconds. This results in a speedup of 3.7 when using the code duplication mode. Similarly, for 16 threads code duplication has a speedup of 4.8 when compared to the execution time of the if clause mode.

Moreover, in the presence of alterations in the shape of the blocks, as shown in figures 7.1.2 and 7.1.2, the if clause mode produces an even slower execution time. Parallelising the innermost loop on the blocks with a shape of 8x2496 imposes a negative impact on the execution time. On the other hand, the code duplication mode can exploit this opportunity in order to utilise all of the available threads by applying parallelism on the \( n_{\text{cell}_i} \) loop.

**Five Repetitions Of Core Calculations**

![Figure 7.4: Comparison of code generation modes - 4 blocks, 4 harmonics, 5 repetitions, 0 alterations](image-url)
Increasing the number of repetitions for each core calculation has a positive effect on the if clause code generation mode. We can observe from figure 7.1.2 that although it
still has a poor performance, when more than 4 threads are available the execution time remains below the execution time which is required when 2 threads are used. However the increase of the number of threads results in performance degradation. Comparing the execution times of the two modes, when 2 and 4 threads are used the code duplication mode achieves a speedup of 1.08 and 1.06 respectively.

Despite this, when 8 threads are used and the \( n_{cell \_j} \) loop is parallelised with code duplication an execution time of 50.8 seconds is achieved whereas with the if clause the execution time becomes 95.4 seconds. Therefore, using code duplication provides a speedup of 1.87. Furthermore, when more threads are available the code duplication results in a speedup of 3.43 when compared to the time required by the if clause mode with 12 threads, and 4.6 with 16 threads. Similar to the case of 1 repetition of the core calculations, when there are blocks whose shape is altered, the if clause code generation mode provides even slower execution times. In the case of 1 alteration, using code duplication provides a gain of 4.6 speedup for 12 threads, and 6.1 for 16 threads. When 2 alterations are used, code duplication has a speedup of 5.7 for 12 threads and 7.8 for 16 threads.

**Impact on performance**

![Figure 7.7: Speedup of code generation modes - 4 blocks, 4 harmonics, 1 repetition of core calculations](image)

Figure 7.7: Speedup of code generation modes - 4 blocks, 4 harmonics, 1 repetition of core calculations

64
Figures 7.1.2 and 7.1.2 present the achieved speedup of the overall execution time when the two modes are applied. The serial version is considered to be the execution of the program when the n_blocks loop is parallelised with 1 thread.

In the case of 1 repetition of the core calculations, as shown in 7.1.2, we can observe that the best speedup is achieved by the code duplication mode. There is a constant increase in the speedup until the point of 12 threads is reached, which gives a speedup of 7 to 8, depending on the case of alterations. It is not clear as to the reason of why the case of 0 alterations produces less speedup than the cases where alterations are present.

In normal circumstances, not having any alterations in the shape of the blocks would give better performance since only the n_cell_j loop is parallelised. On the other hand, having any alteration causes the decision function to parallelise the n_cell_i loop which is one level deeper. This behaviour is possibly the cause of cache effects since each dimension of the arrays of the code is dynamically allocated individually.

When the if clause mode is used the speedup is increased up to the point of 4 threads, where the outermost loop (n_blocks) is parallelised. Using more threads causes the decision function to apply parallelism on the n_cell_j loop. This results in a drop of the achieved speedup on behalf of the if clause mode. An additional drop is also observed in the cases where blocks do not have the same shape.

When the core calculations are repeated for 5 times, as shown in figure 7.1.2, the speedup of the code duplication mode is linear. The amount of work of the calculations remains big enough and the benefit of using more threads for parallel execution is greater than the parallel overheads which are produced. On the other hand, the if clause
mode still suffers from performance degradation.

### 7.1.3 Discussion

The execution times of the *if clause* code generation mode raised some concerns whether the code behaved as expected. In order to ensure that both versions of our compiler perform the same operations we have compiled our runtime library with the `-DDEBUG` flag causing it to produce debugging information during the execution of the code. We have used the case of 4 blocks with no alterations in the shape of the blocks and reduced the number of iterations of the external loop from 500 to 10. The code was executed for both code generation modes, *code duplication* and *if clause* using 8 threads in order to compare the decisions of our runtime library in each case. The tests shown in appendix C indicate that parallelisation was applied only on the n_cell_j loop in both of the cases and threads where created as expected.

Since we were certain that both code generation modes produced the same behaviour, the next step was to assume that the *if clause* mode is greatly affected by the parallel overheads of the OpenMP runtime library of the GCC compiler. Other authors[11] have already studied the overheads of nested parallelism on various compilers, including a more recent version of the GCC compiler than the one used in this work. Their findings suggest that the implementation of nested parallel regions of the GCC compiler has great overheads. Specifically, it is stated that this might be the result from using kernel-level threads (by creating more threads than the available processors), which compete for the available hardware resources. What is not presented in their work is whether or not the use of the *if clause* on nested parallel regions produces the same overheads.

In order to ensure that the behaviour we observed in our results is the cause of nested parallel regions and not the presence of the *if clause*, we have constructed four versions of a benchmark code with three nested loops. The code is simple. There are three nested loops and the delay function of the EPCC Micro-benchmark Suit is used in the block of the innermost loop. The outermost loop is used for repetitions and always executes sequentially. The first version of the benchmark creates a parallel region on the loop of the second level. The second version performs the same operation on the innermost loop. The third version uses the *if clause* on both loops by serialising the outer loop with a value of 0 and parallelising the inner loop with a value of 1. Finally, the last version creates a parallel region on both of these loops, however we force the number of threads on the thread team of the outer loop to 1 using the `num_threads` clause. This way, we manage to reproduce the same behaviour as with the *if clause* code case when the inner loop is parallelised.

The number of iterations of the parallel loops is the same as the number of available threads. We have executed this benchmark with 8 threads, using a value of 24800000 for the delay function, and measured the execution time of each case. Table 7.2 presents the execution times of each case. In our small benchmark the difference of the nested parallelism with the inner approach is small, however it does show that parallelising the
inner loop with nested parallel regions takes 10 seconds longer than parallelising the inner loop manually. Moreover, the two versions that contain nested parallel regions achieve the same execution times. From this test we conclude that the behaviour we observed from the *if clause* code generation mode is affected by the overheads of the implementation of the GCC compiler for nested parallel regions.

<table>
<thead>
<tr>
<th>Parallel loop</th>
<th>Execution time in seconds</th>
</tr>
</thead>
<tbody>
<tr>
<td>Outer</td>
<td>38.845619</td>
</tr>
<tr>
<td>Inner</td>
<td>153.06809</td>
</tr>
<tr>
<td>Nested (with if_clause)</td>
<td>163.05681</td>
</tr>
<tr>
<td>Nested (with num_threads clause)</td>
<td>162.85479</td>
</tr>
</tbody>
</table>

Table 7.2: Overheads of GCC’s nested parallelism

### 7.2 Comparison Of Decision Functions

#### 7.2.1 Parameters

This section of the dissertation compares the two decision functions of the runtime library with code duplication mode. We consider cases of 4 and 8 blocks with varying numbers of values for the harmonics, for 2, 4, 8, 12 and 16 threads. Similar to the case of the comparison between the code generation modes, we also consider cases where the blocks do not have the same shape by altering the values of the n_cell_j and n_cell_i loops.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>n_iters</td>
<td>500</td>
</tr>
<tr>
<td>n_blocks</td>
<td>4, 8</td>
</tr>
<tr>
<td>n_harms</td>
<td>2, 4</td>
</tr>
<tr>
<td>n_cell_j</td>
<td>2496 or 8</td>
</tr>
<tr>
<td>n_cell_i</td>
<td>8 or 2496</td>
</tr>
<tr>
<td>repeats</td>
<td>1 and 5</td>
</tr>
</tbody>
</table>

Table 7.3: CFD Extract Code parameters when comparing the two decision functions
7.2.2  Test Case: 4 Blocks, 4 Harmonics (2 n_harms)

One repetition of the core calculations

Figure 7.9: Comparison of decision functions - 4 blocks, 4 harmonics, 1 repetition, 0 alterations
Figures 7.9, 7.10 and 7.11 present the execution time of the code when the core calculations are performed only one time. We can observe from the graphs that depending
on the number of threads that are used the best parallelisation strategy varies. For up to 4 threads, it is best to parallelise the outer loop, as it has enough iterations to utilise all the available threads. In these cases, both the decision functions make the right choice. However, the overheads of the profiling logic have a negative impact on the overall execution time. Although profiling is not being performed, the functions which are inserted before and after the execution of each loop which count the amount of work performed at each loop level increase the overall time. Comparing the times of the two decision functions, the decision function with profiling has an overhead of 10 seconds when compared to the time achieved by the heuristics function, which results in a gain of 1.2 speedup when code duplication is used.

In the case of 8 threads being available, the best execution time is achieved by parallelising the loop which corresponds to the n_cell_j loop of each block. The loops of the first two levels only have 4 iterations limiting the amount of threads that can be used. Both the decision functions detect this behaviour successfully and parallelise the n_cell_j loop instead. The overheads of the profiler however produce a slightly worst execution time, ranging from 2 to 10 seconds more than the heuristics decision function.

When 12 and 16 threads are used, the best parallelisation strategy is still for the n_cell_j loop. However, when we compare it with the execution time when 8 threads are used there is only a slight reduction in the execution time. This is the result of the parallel overheads dominating the execution time of the program, since the amount of the time needed for the calculations is relatively small. Moreover, we can observe that the profiler actually chooses to parallelise the harmonics loop instead, whereas the heuristics decider produces the correct behaviour. The timings which are performed during the profiling of a loop, in this case the harmonics loop, also include the overhead from the counting of the work which is performed by the enter and exit functions which are placed before and after each internal loop (n_cell_j and n_cell_i). These overheads affect the decision of the function as the amount of time which is required by the calculations is small. In this case these overheads have greater impact on the timings during profiling than the actual execution time of the calculations.
Five repetition of the core calculations

Figure 7.12: Comparison of decision functions - 4 blocks, 4 harmonics, 5 repetitions, 0 alterations

Figure 7.13: Comparison of decision functions - 4 blocks, 4 harmonics, 5 repetitions, 1 alteration
Figures 7.12, 7.13 and 7.14 present the results of the same benchmark when each core calculation is repeated for 5 times. The overall behaviour remains the same as with the case of 1 repetition. When 2 and 4 threads are available, the outer most parallelisation strategy is the fastest. However, we can still notice a difference between the execution time which is achieved by the two decision functions. In the case of 2 threads, the profiler requires 20 seconds more time when compared to that of the heuristics decision function, which gives a gain of 1.08 speedup in favour of the code duplication mode. When 8 or more threads are available, parallelising the n_cell_j loop provides the best performance since it is the only loop that can utilise all of the threads. In addition to this we can also observe that as the number of threads is increased there is a reduction in the execution time. Increasing the amount of repetitions for each core calculation reduces the significance of the parallel overheads on the execution time. Moreover, we can observe that in this case the decision function with profiling manages to identify the correct behaviour. In this case, the amount of time which is required by the core calculations has more significance on the time measurements during profiling. This results in more stable decisions to be made by the function. However, the heuristics decider still outperforms the profiler by a range of around 5 seconds.

When alterations are present in the shape of the blocks and more than 4 threads are available, as shown in figures 7.13 and 7.14, the heuristics decider manages to exploit this opportunity by parallelising the innermost loop of the blocks whose shape is 8x2496. On the other hand, if we compare the behaviour of the profiler with the execution time of the n_cell_j loop, we can observe that the profiler chooses to apply parallelism on this loop instead. Similar to the case of 1 repetitions, the timings during the profiling of
the n_cell_j are affected by the additional logic which is placed in the code, affecting the decision of the function.

**Impact on performance**

Figure 7.15: Speedup of decision functions - 4 blocks, 4 harmonics, 1 repetition of core calculations
Figure 7.16: Speedup of decision functions - 4 blocks, 4 harmonics, 5 repetition of core calculations

Figure 7.2.2 presents the overall speedup of the application when the two decision functions are used in the case of 1 repetition of the core calculations. We can observe that the decision function which is based on heuristics outperforms the decision function with profiling. Moreover, the parallel overheads limit the maximum achieved speedup to 8.5. The same behaviour as in the comparison between the code generation modes is also present in the cases of alterations, where having no alterations gives less speedup.

By increasing the amount of time needed for the computations, as shown in figure 7.2.2, this has a positive affect on both decision functions. In the case of the heuristics decision function, the execution time provides a linear speedup. On the other hand, the function which is based on profiling is affected by the presence of alterations in the shape of the blocks.

Observing the cases of the decision function with profiling we can notice a drop in the speedup when more that 8 threads are used. In the case of 5 repetitions of the core calculations, the speedup drops when alterations are present in the shape of the blocks. As stated previously, the decision function is not able to exploit this opportunity as there is a constant overhead of the additional code which is placed before and after each loop. This causes the function to apply parallelism on the n_cell_j loop instead which results in only 8 threads to be utilised on the blocks whose shape is altered. On the other hand, when each core calculation is performed for 1 time, the drop of the speedup when 12 and 16 threads are used is the result of a wrongfully made decision to apply parallelism on the harmonics loop.
7.2.3 Test Case: 4 Blocks, 8 Harmonics (4 n_harms)

One repetition of the core calculations

Figure 7.17: Comparison of decision functions - 4 blocks, 8 harmonics, 1 repetition, 0 alterations
When 2 and 4 threads are available, the best strategy is to apply parallelism on the outer loop. Both the decision functions make the right choice, however the additional
logic which is performed by the profiler result in an increase of its execution time when compared to that of the heuristics decider. The difference between the execution times ranges from 10 to 20 seconds depending on the case. Therefore, the gain of using the heuristics decision function instead of the profiling decision function gives a speedup of 1.12 to 1.25.

When 8 threads are used, parallelising the harmonics loop produces the best execution time since the outer loops limits the amount of threads that can be utilised to 4. Once again, the two decision functions manage to detect this. The heuristics decider still outperforms the profiling decider by 10 seconds.

In situations where 12 and 16 threads are available, parallelising the n_cell_j loop produces a slightly better execution time, since the harmonics loop can only utilise up to 8 threads. However, the parallel overheads dominate the execution time of the code. The heuristics decider makes the right choice. On the other hand the profiler has a constant overhead which results in parallelising the harmonics loop instead. Consequently, this also prevents it from achieving an optimal performance.

**Five repetitions of the core calculations**

![Comparison of decision functions](image)

Figure 7.20: Comparison of decision functions - 4 blocks, 8 harmonics, 5 repetitions, 0 alterations
In the same manner as with the previous case, for 2 and 4 threads it's best to parallelise the outer most loop. Both decision functions make the right choice, with the heuristics decider outperforming the profiler. It is not clear as to why the heuristics decider
achieves a faster execution time from the case of manually parallelising the loop. There is the possibility that this is the result of the threads not reaching the implicit barrier of the parallel region at the same time and the execution time is limited by the slowest thread.

When 8 threads are available, parallelising the harmonics loop is the best choice in order to achieve the optimal execution time since it can utilise all of the threads. Both decision functions make the right choice with the profiler producing a slightly worse execution time of around 10 seconds.

Using 12 and 16 threads, it is better to parallelise the n_cell_j loop which can utilise all of the available threads. The time needed for the calculations dominates the execution time and the benefit of utilising all of the available threads becomes more significant than the produced parallel overheads.

In the case of 16 threads, the profiler decides to parallelise the harmonics loop instead of the n_cell_j. Similar to the previous case of 4 blocks with 4 harmonics, this indicates that any effects on the timings of each loop during profiling can result in a suboptimal choice to be taken.

**Impact on performance**

![Speedup of decision functions - 4 blocks, 8 harmonics, 1 repetition of core calculations](image)

Figure 7.23: Speedup of decision functions - 4 blocks, 8 harmonics, 1 repetition of core calculations
Figure 7.24: Speedup of decision functions - 4 blocks, 8 harmonics, 5 repetition of core calculations

From the figures we can observe a similar behaviour to the previous case of 4 blocks with 4 harmonics. In both cases of 1 and 5 repetitions, we can observe that the heuristics decision function outperforms the decision function with profiling. Moreover, in figure 7.2.3, the wrong decision taken by the profiling function when 16 threads are used causes to a sudden drop in the speedup. Comparing the speedup trend of the two cases, when 1 repetition of each core calculation is performed the maximum speedup of the heuristics decision function is 9, where is the point of the parallel overheads dominating the execution time. In the case of 5 repetitions, the maximum achieved speedup is 14.5.
7.2.4 Test Case: 8 Blocks, 4 Harmonics (2 n_harms)

One repetition of the core calculations

Figure 7.25: Comparison of decision functions - 8 blocks, 4 harmonics, 1 repetition, 0 alteration
Figure 7.26: Comparison of decision functions - 8 blocks, 4 harmonics, 1 repetition, 1 alteration

Figure 7.27: Comparison of decision functions - 8 blocks, 4 harmonics, 1 repetition, 2 alterations
The graphs show a similar behaviour with the previous cases when 1 repetition was used for the core calculations. When up to 8 threads are used, the best strategy is to parallelise the outer loops which is responsible for the number of blocks, since it has 8 iterations. Despite the fact that both the decision functions make the correct choice, the heuristics decider has a faster execution time than the profiler.

When 12 and 16 threads are available, parallelising the n_cell_j loop provides a slight reduction in the execution time when compared to that of the n_blocks loop. The difference between the execution time is just 3 seconds in the case of 12 threads, and 1 second in the case of 16. Unlike the previous cases of 4 blocks, increasing this number to 8 results in more parallel overheads to appear when parallelising the n_cell_j loop. Since the workload is small, the benefit of using more threads to parallelise the computations has less significance in the execution time. Despite this, the heuristics decision function manages to identify the correct loop. On the other hand we can observe that the profiler has a constant overhead which prevents it from achieving the best possible execution time.
Five repetitions of the core calculations

Figure 7.29: Comparison of decision functions - 8 blocks, 4 harmonics, 5 repetitions, 0 alterations

Figure 7.30: Comparison of decision functions - 8 blocks, 4 harmonics, 5 repetitions, 1 alteration
Figure 7.31: Comparison of decision functions - 8 blocks, 4 harmonics, 5 repetitions, 2 alterations

For 2, 4 and 8 threads, both of the decision functions make the right choice with the heuristics decision function producing faster execution times. By increasing the amount
of time that each core calculation takes to complete, we can observe that parallelising \texttt{n\_cell\_j} loop with 12 and 16 threads provides a noticeable reduction in the execution time of the program, since the benefit from parallelising the computations has a greater impact on the execution time than the parallel overheads. Moreover, when alterations are present in the shape of the loops, as shown in figures 7.30, 7.31 and 7.32, the heuristics decider manages to adapt its behaviour into parallelising the innermost loop in order to utilise more threads. On the other hand, observing the execution time of the \texttt{n\_cell\_j} loop when it is parallelised manually, we can notice a slight increase in the execution time as the number of alterations gets bigger. The reason for this is that in the cases where a block with altered values is being processed, the \texttt{n\_cell\_j} loop can only utilise up to 8 threads for that block. In the case of the profiler however, the overheads of the decision function have a negative impact on its execution time.

**Impact on performance**

Figure 7.33: Speedup of decision functions - 8 blocks, 4 harmonics, 1 repetition of core calculations
Figure 7.34: Speedup of decision functions - 8 blocks, 4 harmonics, 5 repetition of core calculations

The impact on the overall performance of the two decision functions remains the same with the previous cases. The heuristics decision function outperforms the decision function with profiling in both cases of 1 and 5 repetitions of the core calculations.

Similar to the previous cases, in figure 7.2.4 we can observe that a maximum speedup of 8 is achieved by the heuristics decider. On the other hand, the maximum speedup of the decision function with profiling is 5.5.

Increasing the repetitions of core calculations, as graphed in figure 7.2.4, causes the heuristics decision function to achieve a linear speedup, reaching a value of 14.5 when 16 threads are used. Similarly, the decision function with profiling achieves a maximum speedup of 12.5. We can still notice a drop in the speedup when 16 threads are used, in the case of 2 and 4 blocks have their shape altered.

7.2.5 Discussion

All of the test cases produce a similar behaviour. The best parallelisation strategy is the first loop of the parallel region that has enough iterations to utilise all of the available threads. However, depending on the workload of the internal loops, the benefit of using more threads when parallelising an internal loop varies. In the cases where only 1 repetition of the core calculations is performed, the gain from reducing the execution time of the code is less than the case of 5 repetitions. Moreover, as the number of the iterations of the n_blocks loop is increased, the reduction in the execution time when using more threads is small, as this causes an increase in the parallel overheads.
In all of the test cases, the decision function which is based on profiling provides slower execution times than the decision function which is based on heuristics. Moreover, the additional logic which is included in the decision function with profiling caused a wrongful decision to be made in some situations.

As previously stated in chapter 5, we only considered the worst case from the timings of the conducted benchmarks since this is the limiting factor. It is worth noting that there were fluctuations in the timings of each benchmark. In the case of the decision function with profiling where a wrong decision was made, in some executions the function actually managed to produce the correct behaviour. However, this shows that the timings from the profiling are affected by the presence of any overheads.
Chapter 8

Relaxed Version Of Decision Function With Profiling

The results from the previous benchmarks lead to considerations of the reasons behind the poor execution of the decision function which performs profiling. Comparing the functionality of this function with the simple case of the heuristics decision function there are two sources of additional overheads.

The first one is the logic of profiling each version of a loop. In order to make a choice between the two versions of a loop, the slow version must also be executed. The second source of overheads is the inclusion of additional function calls before and after each loop in order to measure the time of the execution and count the amount of work performed.

The elimination of the functionality for taking the slow path is not possible since this is the essence of profiling. Both versions of a loop must be executed in order to make a comparison between their execution time. However, we can relax the conditions on the validity of the timings.

If we only consider the number of the iterations of the specific loop which is being profiled, then we can eliminate all the logic that performs the counting of the work for the internal loops. When the decision function decides that a version of a loop should be profiled (after the failure of the heuristics conditions) the number of the iterations of the version of the loop that is going to be executed is saved in the state of the loop at that exact point. This way, the code of the function calls which are placed before and after each loop remains simple, only adjusting the loop level counter of the master thread (since no work is being counted by the other threads), as well as marking the starting and ending times of the execution of a loop which is being profiled.

In order to test our theory we have created a new version of the runtime library which includes the above modifications.
8.1 Results

The following results present the benchmarks with the relaxed version of the function with profiling. The parameters which were used are the same as the ones presented in chapter 7. We have only included the results when the decision function with the counting of the work of the internal loops made a wrong decision. The graphs for the rest of the benchmarks are included in Appendix D.

8.1.1 Test Case: 4 Blocks, 4 Harmonics (2 n_harms)

Figure 8.1: Comparison of decision functions with the relaxed version of the profiling decision function - 4 blocks, 4 harmonics, 1 repetition, 0 alterations
From the graphs we can observe that the removal of the additional logic which performs the counting benefits the decision function with profiling. Not only did it reduce the
execution time when profiling is performed, as shown in cases of 2 and 4 threads, but also managed to detect the correct loop in the cases of 12 and 16 threads.

When no profiling is performed (2 and 4 threads), the relaxed version of the decision function has a reduction of 5 seconds in the execution time when compared to that of the accurate version. In the case of 8 threads, there is still a noticeable drop in the execution time of 2 seconds in each case.

Comparing the execution time of the new version of the decision function with profiling to the execution time of the heuristics decision function, the latter still produces a faster execution time. The difference ranges between 1 and 3 seconds depending on the case. This behaviour is expected, since the presence of profiling introduces additional computations within the code itself from the functions which are placed before and after each loop. Moreover, in the cases where the parallelisation is applied on a nested loop, the decision function must execute both versions of a loop, one of them being the slow version, in order to make a decision.

8.1.2 Test Case: 8 Blocks, 4 Harmonics (2 n_harms)

Figure 8.4: Comparison of decision functions with the relaxed version of the profiling decision function - 8 blocks, 4 harmonics, 5 repetition, 0 alterations
Figure 8.5: Comparison of decision functions with the relaxed version of the profiling decision function - 8 blocks, 4 harmonics, 5 repetition, 1 alterations

Figure 8.6: Comparison of decision functions with the relaxed version of the profiling decision function - 8 blocks, 4 harmonics, 5 repetition, 2 alterations
We can observe from the graphs that the relaxed version of the decision function with profiling also manages to adapt its behaviour in the presence of alterations in the shapes of the blocks. In addition to the reduction of the execution time when compared to that of the accurate version, the achieved execution times of the relaxed version when the amount of the core computations is repeated for 5 times matches the execution times of the heuristics decision function. The gain in the execution time of the relaxed version ranges from 6 to 20 seconds when compared to the execution times of the accurate version. This produces a speedup ranging from 1.08 to 1.11 when the relaxed version is used instead of the accurate version. When compared to the execution time of the heuristics decision function, the additional required time ranges from 1 to 3 seconds, resulting in a gain of 1.02 to 1.03 speedup in favour of the heuristics decision function.

8.1.3 Test Case: 15 Blocks, 14 Harmonics (7 n_harms)

The promising results from the previous section lead us to consider cases where the heuristics decision function will fail to identify the correct behaviour whereas the relaxed version of the profiling decision function would identify the best possible parallelisation strategy. If we consider the parallel overheads as a source of additional execution time which is introduced by applying parallelism into an inner loop, then depending on the parameters of the simulation these overheads can increase the overall execution time.

Expressing the parallel overheads in terms of $T_{outer\_work}$, as presented in the equation
6.4 shown in Chapter 6, we can reduce the maximum allowed threshold value by selecting specific simulation parameters so that the execution time will be increased. In order to achieve that we have selected the following parameters for the CFD extract code:

- **Block size**: 156x8
- **Repetitions**: 1
- **n_blocks**: 15
- **n_iters**: 5000

![Figure 8.8: Comparison of decision functions with the relaxed version of the profiling decision function - 15 blocks, 7 harmonics, 1 repetition, 0 alterations](image)

Figure 8.8 presents the results of the CFD extract code. When 15 threads are available, the best parallelisation strategy is to apply parallelism on the outermost n_blocks loop, since it can utilise all of the available threads. In this case, both the heuristics decision function and the relaxed version of the runtime library make the right decision. However, when 16 threads are available, the heuristics decision function will apply parallelism on the n_cell_j loop. The produced parallel overheads have a negative impact on the execution producing a time of 142.5 seconds. On the other hand, the decision function with profiling manages to detect the correct behaviour by not applying parallelism on the inner loop. This choice results in an execution time of 116 seconds.
Chapter 9

Conclusions and Future Work

9.1 Conclusions

The main focus of this work was to investigate the possibility of dynamically choosing at runtime the best loop of a nested loop region which best utilises the available threads. We have successfully created a source to source compiler and a runtime library in order to automatically transform the original source code, allowing a dynamic choice to be made at runtime. In addition to this, the runtime parameters of each loop are constantly being monitored in order to adapt the decision dynamically in case of changes to these parameters. Furthermore, our solution uses a directive based approach similar to OpenMP which requires minimum effort from the user’s point of view in order to utilise our functionality.

From the two code generation modes supported by our compiler, we have found that the code duplication mode outperformed the if clause mode. In situations where the decision function chose to parallelise an internal loop, using the if clause increased the execution time of the code dramatically. Despite the fact that this behaviour is the result of the inefficient implementation of the GCC compiler which was used in this work, the same compiler with the code duplication mode was able to provide additional speedup in the execution time of the code. From this we conclude that by relying on the OpenMP runtime library to perform loop nesting, the execution time is limited by the compiler’s implementation of nested parallel regions. Although code duplication is considered to be a bad programming practice, when it is done automatically, it can eliminate unnecessary parallel overheads.

The runtime library implements two decision algorithms. The first one is based on heuristics which only takes into consideration the amount of iterations of a loop and the number of available threads. The second one is an extension to this algorithm where profiling is performed in order to make a decision based on time measurements. The suitability of each decision function depends on the structure of the nested loop region. The decision based on heuristics produced better results in the cases of perfectly nested loops where parallelising an internal loop did not serialised any computations. On the
other hand, the decision based on profiling was able to capture situations where parallelising an internal loop produced worse execution time from serialising computations which appear in-between the loops.

Despite this, we have found that the decision based on profiling produced a lot of overheads when trying to accurately count the amount of computations performed by the execution of each loop. In some situations, this prevented the method from making the right decision since the additional logic affected its timings. When a more relaxed version of this method was developed, the decision function was able to produce similar results to that of the heuristics decision function. Moreover, the relaxed version was able to identify situations where the presence of excessive parallel overheads when parallelising an internal loop increased the overall execution time. From this we conclude that the additional logic which is required by our implementation in order to make an accurate decision on the validity of the timings when profiling a loop does not justify its overheads.

The idea of dynamic loop nesting does have its benefits. From our results, we have found that when the amount of the workload in the core calculations remains big enough, the impact of the parallel overheads from the creation of parallel regions in nested loops has less significance in the overall execution time. Moreover, based on the parameters during the execution of the code, there is an acceptable amount of work that can be serialised. When such case is presented and the number of iterations of the external loops limit the amount of processors which can be utilised, applying parallelism on the internal loops provides additional speedup to the overall execution.

9.2 Future Work

Despite the fact that our project is considered to be a success, there is still work that can be done in order to enhance it. Given the available time budget however, we were only able to investigate only a handful of the full possibilities.

The most important of all being the deployment of our solution to a large scientific code in order to measure the effects it has in real situations. Although the benchmark code we have used is an extract from a real application, we were only able to measure the effects our solution has on that particular region.

Another area of investigation we had in mind is the concept of the nested parallelisation strategy. Although this is possible with the current implementation of OpenMP and the use of dynamic thread allocation to the regions, this still requires the application of a parallel construct on each loop. However, in the case of codes which have regions of deeply nested loops, there is the potential of exhausting all of the available threads at the loops of the outer levels. However, unnecessary parallel regions will still be created on the loops of the internal levels, which might lead to situations similar to the case of the if clause. By extending our runtime library to handle nested parallelism, we can avoid the creation of these parallel regions.
Furthermore, the current implementations of our decision functions can still make use of some additional refinements. The heuristics decision function currently assumes that there is at-least one loop in the loop region that can utilise all of the available threads. The logic can be extended in order to make a second-best choice in situations where no loop can utilise all of the available threads. Another extension to the function is to detect cases of uneven distribution of the iterations to the threads, similar to the case of the synthetic benchmark code when 6 threads were available. These two additions will require from the decision function to maintain some state for the loops. However, considering our findings from the relaxed version of the profiling function, as long as the logic remains simple then it will not introduce much overheads.

Furthermore, our current implementations of the two decision functions which are based on profiling do not keep track of the frequency a loop has been profiled. This can impact the execution time with the presence of loops which require profiling and their number of iterations is constantly being changed, since profiling will always be performed on the loops.

Moreover, our prototype source to source compiler does not implement any semantic checks on the code which is parsed. Although this is not a problem in itself as any errors will be captured by the C compiler which is used at the final step of the compilation process, it would be beneficial if the compiler was extended to support this functionality.
Appendix A

Example Outputs From Source-To-Source Compiler

This section presents some example outputs of our source-to-source compiler. For clarity reasons, some of the details have been removed in order to make the examples more readable.

A.1 Using The If-Clause mode

```c
/* ***************************************************************************
 * Input Code
 * ***************************************************************************

#pragma precomp parallel for
for (i = 0; i < I; i++) {
#pragma precomp parallel for
    for (j = 0; j < J; j++) {
        work();
    }
}

/* ***************************************************************************
 * Output Code
 * ***************************************************************************

#pragma omp parallel for if( decision_function( 0, /* loop_start */
        1, /* loop_end */
        1, /* increment */
        0, /* inclusiveness */
        0, /* subtract */
        1.0 /* threshold */
))
```
for (i = 0; i < I; i++) {
#pragma omp parallel for if (decision_function(0, /* loop_start */
J, /* loop_end */
1, /* increment */
0, /* inclusiveness */
0, /* subtract */
1.0 /* threshold */
))
for (j = 0; j < J; j++) {
    work();
}
}

A.2 Using the Code Duplication Mode

/*-----------------------------------------------------------------------------------*/
/* Input Code                                                                         */
/*-----------------------------------------------------------------------------------*/
#pragma preomp parallel for
for (i = 0; i < I; i++) {
#pragma preomp parallel for
    for (j = 0; j < J; j++) {
        work();
    }
}

/*-----------------------------------------------------------------------------------*/
/* Output Code                                                                        */
/*-----------------------------------------------------------------------------------*/
if (decision_function(0, /* loop_start */
J, /* loop_end */
1, /* increment */
0, /* inclusiveness */
0, /* subtract */
1.0 /* threshold */
))
{
    #pragma omp for
    for (i = 0; i < I; i++)
    {
        if (decision_function(0, /* loop_start */
J, /* loop_end */
1, /* increment */
0, /* inclusiveness */
0, /* subtract */

1.0 /* threshold */
))
{
    # pragma omp for
    for ( j = 0; j < J; j++ )
    {
        work();
    }
} else {
    for ( j = 0; j < J; j++ )
    {
        work();
    }
}
}
else {
    for ( i = 0; i < I; i++ )
    {
        if( decision_function( 0, /* loop_start */
                J, /* loop_end */
                1, /* increment */
                0, /* inclusiveness */
                0, /* subtract */
                1.0 /* threshold */
            ))
        {
            # pragma omp for
            for ( j = 0; j < J; j++ )
            {
                work();
            }
        } else {
            for ( j = 0; j < J; j++ )
            {
                work();
            }
        }
    }
}
Appendix B

Detailed Solution Of The Performance Model For The Maximum Allowed Execution Time Of The Outer Loop’s Work

Given a region of two nested loops with computations in-between the loops, as shown in B.1.

```c
for (i = 0; i < outer_iters; i++) {
    outer_work();
    for (j = 0; i < inned_iters; j++) {
        inner_work();
    }
}
```

Listing B.1: Two nested loops with work in-between each loop.

The execution time of the two nested loops ($T_{pOuter}$) when parallelising the outer loop using a number of threads ($outer\_threads$) is shown in equation B.1.

$$T_{pOuter} = \frac{outer\_iters \times (T_{outer\_work} + (inner\_iters \times T_{inner\_work}))}{outer\_threads}$$  \hspace{1cm} (B.1)

The execution time of the two nested loops ($T_{pInner}$) when parallelising the inner loop using a number of threads ($inner\_threads$) is shown in equation B.2.

$$T_{pInner} = outer\_iters \times (T_{outer\_work} + \frac{inner\_iters \times T_{inner\_work}}{inner\_threads})$$  \hspace{1cm} (B.2)

The constrain that must be satisfied in order to get any performance increase when parallelising the inner loop is shown in equation B.3.
\[ T_{p_{\text{Inner}}} < T_{p_{\text{Outer}}} \]  \hspace{1cm} (B.3)

From the equations B.1, B.2 and B.3 we get the maximum allowed threshold of the execution time for the work of the outer loop \((T_{\text{outer\_work}})\) as shown in B.12.

\[
\begin{align*}
\text{outer\_iters} \times (T_{\text{outer\_work}} + \frac{\text{inner\_iters} \times T_{\text{inner\_work}}}{\text{inner\_threads}}) & < \frac{\text{outer\_iters} \times (T_{\text{outer\_work}} + (\text{inner\_iters} \times T_{\text{inner\_work}}))}{\text{outer\_threads}} \\
(\text{outer\_iters} \times T_{\text{outer\_work}}) + (\text{outer\_iters} \times \frac{\text{inner\_iters} \times T_{\text{inner\_work}}}{\text{inner\_threads}}) & < \frac{(\text{outer\_iters} \times T_{\text{outer\_work}}) + (\text{outer\_iters} \times \text{inner\_iters} \times T_{\text{inner\_work}})}{\text{outer\_threads}} \\
(\text{outer\_iters} \times T_{\text{outer\_work}}) + (\text{outer\_iters} \times \frac{\text{inner\_iters} \times T_{\text{inner\_work}}}{\text{inner\_threads}}) & < \frac{\text{outer\_iters} \times T_{\text{outer\_work}}}{\text{outer\_threads}} + \frac{\text{outer\_iters} \times \text{inner\_iters} \times T_{\text{inner\_work}}}{\text{outer\_threads}} \\
(\text{outer\_iters} \times T_{\text{outer\_work}}) - \frac{\text{outer\_iters} \times T_{\text{outer\_work}}}{\text{outer\_threads}} & < \frac{\text{outer\_iters} \times \text{inner\_iters} \times T_{\text{inner\_work}}}{\text{outer\_threads}} - \frac{\text{outer\_iters} \times \text{inner\_iters} \times T_{\text{inner\_work}}}{\text{inner\_threads}} \hspace{1cm} (B.7)
\end{align*}
\]

\[
\begin{align*}
T_{\text{outer\_work}} \times (\text{outer\_iters} - \frac{\text{outer\_iters}}{\text{outer\_threads}}) & < \frac{\text{outer\_iters} \times \text{inner\_iters} \times T_{\text{inner\_work}}}{\text{outer\_threads}} - \frac{\text{outer\_iters} \times \text{inner\_iters} \times T_{\text{inner\_work}}}{\text{inner\_threads}} \hspace{1cm} (B.8)
\end{align*}
\]
\[ T_{outer\_work} < \frac{outer\_iters \times inner\_iters \times T_{inner\_work}}{outer\_threads} - \frac{outer\_iters \times inner\_iters \times T_{inner\_work}}{inner\_threads} \] (B.9)

\[ T_{outer\_work} < \frac{outer\_iters \times inner\_iters \times (T_{inner\_work} - \frac{T_{inner\_work}}{outer\_threads} - \frac{T_{inner\_work}}{inner\_threads})}{outer\_iters - \frac{outer\_iters}{outer\_threads}} \] (B.10)

\[ T_{outer\_work} < \frac{outer\_iters \times inner\_iters \times (T_{inner\_work} - \frac{T_{inner\_work}}{outer\_threads} - \frac{T_{inner\_work}}{inner\_threads})}{outer\_iters \times (1 - \frac{1}{outer\_threads})} \] (B.11)

\[ T_{outer\_work} < \frac{inner\_iters \times T_{inner\_work} \times (\frac{1}{outer\_threads} - \frac{1}{inner\_threads})}{1 - \frac{1}{outer\_threads}} \] (B.12)
Appendix C

Testing Of Decisions For The Code Duplication Mode And The If Clause Mode

```bash
# Test total number of debugging information
# for the code duplication mode
$ cat cfd_duplicate.bash.e7275 | wc -l
$ 399571

# Test number of available threads
# for the code duplication mode
$ cat cfd_duplicate.bash.e7275 | grep "threads"
$ preomp_init(): Running with 8 threads

# Test total number of parallel decisions for
# the code duplication mode
$ cat cfd_duplicate.bash.e7275 | grep "Decision_1" | wc -l
$ 160

# Test number of parallel decisions
# on the n_cell_j loop for the code duplication mode
$ cat cfd_duplicate.bash.e7275 | grep "Decision_1.2496" | wc -l
$ 160

# Test total number of sequential decisions
# for the code duplication mode
$ cat cfd_duplicate.bash.e7275 | grep "Decision_0" | wc -l
$ 399410

# Test total number of debugging information
# produced by thread 7 for the code duplication
$ cat cfd_duplicate.bash.e7275 | grep "[7]" | wc -l
$ 49920

# Test total number of debugging
# information for the if clause mode
```
Listing C.1: Testing the if clause and code duplication modes

```bash
$ cat cfd_if_clause.bash.e7273 | wc -l
$ 399571

# Test number of available threads
# for the if clause mode
$ cat cfd_if_clause.bash.e7275 | grep "threads"
$ preomp_init(): Running with 8 threads

# Test total number of parallel decisions for
# the if clause mode
$ cat cfd_if_clause.bash.e7273 | grep "Decision_0" | wc -l
$ 160

# Test number of parallel decisions on
# the n_cell_j loop for the if clause mode
$ cat cfd_if_clause.bash.e7273 | grep "Decision_1.*2496" | wc -l
$ 160

# Test total number of sequential decisions
# for the if clause mode
$ cat cfd_duplicate.bash.e7275 | grep "Decision_0" | wc -l
$ 399410

# Test total number of debugging information
# produced by thread 7 for the if clause mode
$ cat cfd_if_clause.bash.e7273 | grep "[7]" | wc -l
$ 49920
```
Appendix D

Graphs From The Benchmarks Of The Relaxed Version Of The Profiling Decision Function

D.1 Test Case: 4 Blocks, 4 Harmonics (2 n_harms)

![Graph comparison of decision functions - 4 blocks, 4 harmonics, 5 repetition, 0 alterations](image)

Figure D.1: Comparison of decision functions - 4 blocks, 4 harmonics, 5 repetition, 0 alterations
Figure D.2: Comparison of decision functions with the relaxed version of the profiling decision function - 4 blocks, 4 harmonics, 5 repetition, 1 alteration

Figure D.3: Comparison of decision functions with the relaxed version of the profiling decision function - 4 blocks, 4 harmonics, 5 repetition, 2 alterations
D.2 Test Case: 8 Blocks, 4 Harmonics (2 n_harms)

Figure D.4: Comparison of decision functions with the relaxed version of the profiling decision function - 8 blocks, 4 harmonics, 1 repetition, 0 alterations

Figure D.5: Comparison of decision functions with the relaxed version of the profiling decision function - 8 blocks, 4 harmonics, 1 repetition, 1 alterations
Figure D.6: Comparison of decision functions with the relaxed version of the profiling decision function - 8 blocks, 4 harmonics, 1 repetition, 2 alterations

Figure D.7: Comparison of decision functions with the relaxed version of the profiling decision function - 8 blocks, 4 harmonics, 1 repetition, 4 alterations
Bibliography


