Grammatical Evolution + Multi-Cores

= Automatic Parallel Programming!

by

Gopinath Chennupati

B.Tech., M.Sc.

Supervisor: Prof. Conor Ryan

Co-Supervisor: Dr. R. Mohammad Atif Azad

External Examiner: Dr. John R. Woodward

A thesis for the PhD Degree

Submitted to the University of Limerick

2015
Declaration

I hereby declare that the work presented in this thesis is original except where an acknowledgement is made or a reference is given to other work. I confirm that I have read and accept the Handbook of Academic Regulations and Procedures.
The thesis work was conducted from 2012 to 2015 under the supervision of Prof. Conor Ryan at University of Limerick.

Student: Gopinath Chennupati

Signature: 

Date: September, 2015

Supervisor: Prof. Conor Ryan

Signature: 

Date: September, 2015
Abstract

Multi-core processors are shared memory multiprocessors integrated on a single chip which offer significantly higher processing power than traditional, single core processors. However, as the number of cores available on a single processor increases, efficiently programming them becomes increasingly more complex, often to the point where the limiting factor in speeding up tasks is the software.

This thesis presents Grammatical Automatic Parallel Programming (GAPP) which uses Grammatical Evolution to automatically generate natively parallel code on multi-core processors by directly embedding GAPP OpenMP parallelization directives in problem-specific Context Free Grammars. As a result, it obviates the need for programmers to think in a parallel manner while still letting them produce parallel code.

We first perform a thorough analysis on the computational complexity of Grammatical Evolution using standard benchmark problems. This analysis results in an interesting experiment which produces a system capable of predicting on-the-fly the likelihood of a particular GE run being successful.

A number of difficult proof of concept problems are examined in evaluating GAPP. The performance of the system on these informs the further optimization of both the design of grammars and fitness function to extract further parallelism. We demonstrate a surprising side effect of uncontrolled parallelism, which leads to the under-utilisation of the cores. This is addressed through the automatic generation of programs with controlled degree of parallelism. In this case, the automatically generated programs adapt to the number of cores on which they are scheduled to execute.
Finally, GAPP is applied to Automatic Lockless Programming, an enormously difficult design problem, resulting in parallel code guaranteed to avoid locks on shared resources, thereby further optimizing the execution time. We then draw conclusions and make future recommendations on the use of evolutionary systems in the generation of highly constrained parallel code.
Acknowledgements

Let me begin my gratitudes with the reminiscence of the Unknown Creator behind the creation, and her Mother and Father. As the whole World wants to know about her true identity, so does I ...

In this auspicious journey, first, I would like to cordially thank my supervisor, Prof. Conor Ryan for supporting a know-nothing-EC boy to grow as big as to achieve one of the prestigious honours of the field, Humies Silver Award. I cherish his supervision, encouragement, domain expertise, managerial skills, group engagements, and lastly, the most amusing outings during Christmas! Conor’s spirit of allowing me to figure out certain things on my own and rightly directing me whenever I was in need are both enjoyable and rewarding experiences.

Dr. Atif Azad is the next big pillar in educating me with a number of friendly and informal discussions about GE and EC, on a whole. I appreciate his support in writing this thesis and publishing most of this research work in reputed venues. His critical thinking and thoughtful comments are a boost not only to me but also to the entire group. His co-supervisory skills are intriguing and applaudable. Although, I have a little knowledge, I enjoyed our discussions on the India and Pakistan relationships, the border issues, the brutal and ruthless British empire’s rule over subcontinent, and the historical wars. I must admit that I missed your invitation for Eid due to my practice of vegetarianism on Mondays. May be I will get it straight next time ...!

I want to thank all the members, past and present, of the Bio-computing and Developmental Systems (BDS) group for guidance, support, and the
welcoming environment that they provided. Dr. Tom Arbuckle, Dr. Jean-nie Fitzgerald, Dr. Daniel Howard, were great as post-docs and very open to discuss anything. Dr. Damien T. Hogan who graduated and went onto join Conor’s start-up was very helpful with a lot of Linux issues in my early days at BDS. Rezaul Karim left the group because of some health issues, with whom had some interesting discussions. James Patten, Fergal Lane, and David Medernach are awesome pals to have all the discussions about double Irish, Irish economy, recession and European Union, etc. Lately, James has been educating me about the data science and high performance computing scientist jobs that are booming in Ireland. Especially, I am very fortunate to have David around, with whom I attended a number of conferences, had fun, took a lot of photos and went to gym everyday. Our discussions on French politics, the funny nationalist party, and the social-ists all enhance my understanding of European politics. Recently, he was getting a bit grumpy with the timings and delays, but, that is grand, we are back to normal with the start of the new semester.

J.J. Collins has been very handy whenever I was in need of a few clar-ifications on parallel computing. I must admit that all our discussions have helped me a lot in gaining insights and improving my knowledge of parallel programming.

The department of Computer Science and Information Systems (CSIS) has been a great place to work during this time by accommodating to work irrespective of weekends, and public holidays for the last one year. The head of the department Annette McElligott was very nice in helping me out in all the ways that she possibly can. She was so generous in cancelling her holiday to sort out a letter, which was a real emergency when I applied for a visa to Portugal to attend a conference. The administrative staff Nuala Kit-son, Gemma Swift, and Joan Imray were always there for all the support with a lot of patience. Specially, Nuala was real second to none in helping with a lot of letters and sorting out the financials of the trips to conferences.

I must remind the help of Liam O’Riordan in setting up my machine in the CSIS network and giving access to more than three ports, which ultimately became handy for all my learning stunts with Hadoop, and CUDA.
**Redmond O’Brien** has come to my rescue when the e-mail was hacked, when I was in need of a ethernet hub for my stupid configuration of a Beowulf cluster. **Darragh Pigott** has generously loaned me the windows operating system, of course, which I have never used for my research but kept it for dual booting.

**Danny Browne**, the porter (old man!), is another chatting buddy who always had time for a nice discussion on the Nature, Irish culture, his plantation, and the Gaelic football. It was great to have him here especially, during high time at work. In fact, his Banjo lessons were a great relief of huge pressure. I must thank him for his gift of the Banjo, and for being the Banjo Guru with his Irish classical tunes.

**Dr. Bill Langdon** is another great resource who was very welcoming when I first met him in GECCO’14 held at Vancouver, Canada. I admire his comments, suggestions, and generous advices on the work in this thesis when it was first presented in the Student workshop of GECCO’14. Also, his continued support in the subsequent conferences and through personal communication (e-mails and letters) are unforgettable.

Finally, I would like to thank **Dr. John R. Woodward** for agreeing to be the external and generously reorganising the initially set viva dates. Also, it was a great fun having met him in GECCO’15 held at Madrid Spain.
Dedication

An endless journey through darkness and a prosperous endeavour. Dedicated to the Mother, Father, and the Creator behind their creation.
Contents

List of Tables xiv

List of Figures xviii

I Introduction and Background 1

1 Introduction 2
  1.1 Motivation ................................................. 3
  1.2 Research Aims ............................................. 5
    1.2.1 Research Questions ................................. 5
    1.2.2 Objectives ........................................... 6
  1.3 Contributions ............................................. 7
  1.4 Thesis Structure ........................................... 8

2 Evolutionary Automatic Programming 11
  2.1 Introduction ................................................. 11
  2.2 Evolutionary Automatic Programming ...................... 13
  2.3 Genetic Algorithms ....................................... 15
  2.4 Evolution Strategies ..................................... 18
  2.5 Evolutionary Programming ................................ 19
  2.6 Genetic Programming .................................... 19
  2.7 Context Free Grammar .................................... 22
5.3.1 System Overview ........................................ 121
5.3.2 Design of Grammar ...................................... 123
5.3.3 Parallelization Strategy ................................. 124
5.4 Experiments .................................................. 125
  5.4.1 Experimental Setup ..................................... 125
  5.4.2 Experimental Results ................................... 126
5.5 Discussion .................................................... 130
  5.5.1 Efficiency ................................................ 130
  5.5.2 Genome Lengths ......................................... 131
5.6 Summary ....................................................... 134

6 Proof of Concept ............................................ 135
  6.1 Introduction ............................................... 135
  6.2 Recursion .................................................. 136
    6.2.1 Scope for Parallel Recursion ....................... 136
    6.2.2 Evolutionary Approaches for Recursion .......... 138
  6.3 GAPP Grammars for Recursion ......................... 140
  6.4 Experiments ............................................... 142
    6.4.1 Experimental Setup ................................. 142
    6.4.2 Experimental Problems ............................. 144
    6.4.3 Terminating Recursion .............................. 145
    6.4.4 Results .............................................. 146
  6.5 Discussion ................................................ 149
  6.6 Iterative Sorting ........................................ 150
    6.6.1 Evolutionary Approaches for Sorting ............ 151
  6.7 GAPP Grammars for Iterative Sorting ................. 153
  6.8 Experiments ............................................... 154
    6.8.1 Experimental Setup ................................. 154
    6.8.2 Experimental Problems ............................. 155
    6.8.3 Results .............................................. 157
  6.9 Discussion ................................................ 158
  6.10 Summary .................................................. 159
List of Tables

2.1 A list of open source software libraries for GE in different languages. ........................................... 42

4.1 The frequency of runs with various success intervals both in the presence and absence of wrapping on Santa Fe Ant Trail and binomial-3 problems. ................................. 92

4.2 The problem set considered. The “Yes” column represents the number of runs producing results above the threshold; the “No” column stands for the runs that remained below the threshold. The success threshold is decided on the Best Fitness (BF). ................................................................. 104

4.3 A brief description of the experimental datasets. BFC (Best Fitness Change), AFC (Average Fitness Change), AALC (Average Actual Length Change), AELC (Average Effective Length Change) are the attributes of the dataset. The data points represent the change from generation 1 to 10. ............................... 105

4.4 Comparison of predictive accuracy (mean [standard deviation]) in %, among J48, ZeroR, cAnt – MinerPB, PART and JRip measured by applying 10-fold cross-validation. The value of the most accurate algorithm is shown in bold for a given data set with Wilcoxon Signed Rank Sum tests at $\alpha = 5\%$. ................................................................. 107

4.5 Manually selected rules in predicting the performance of a GE run for all the four experimental problems. .......................... 109
4.6 Comparing the success rate of the standard GE versus RPM+GE measured for 30 evolutionary runs.

4.7 Wilcoxon tests at $\alpha = 5\%$ on the execution time of GE ($T_{GE}$) versus RPM+GE ($T_{RPM+GE}$) over 30 runs. Note, “[□]” shows that RPM+GE outperforms standard GE, whereas “[×]” shows that standard GE outperforms RPM+GE. When there is a significant difference, A measure is shown, otherwise (–).

4.8 Predictions of RPM+GE on the four experimental problems.

4.9 Total number of individuals processed and the mean best-of-run fitness (mean [standard deviation]) for both GE and RPM+GE.

5.1 The execution time (average [standard deviation]) of GAPP evolved parallel multiplexer programs averaged across 30 runs with 64, 256 and 2048 fitness cases, that is 6, 8 and 11 multiplexers.

5.2 The execution time (average [standard deviation]) of GAPP evolved parallel sextic polynomial regression programs averaged across 30 runs for 64, 512 and 1024 fitness cases that lie in the range $[-1.5, 1.5]$.

6.1 Experimental parameter settings

6.2 Experimental hardware and software settings.

6.3 The problems used in the experiments and their properties. Input and Return represent the data type of the input arguments and return results respectively. Local Variables (LV) are the temporary variables while Range is the bounds from which the input is drawn randomly.

6.4 Mean best execution time (MBT [standard deviation]) of GAPP at different number of cores of an Intel Xeon processor.

6.5 The problem set considered, input type and the local variables (LV).
6.6 Mean best execution time (MBT [standard deviation]) of four problems averaged across 50 runs of GAPP for 100 generations. ................................................. 158

7.1 Friedman statistical tests with Hommel’s post-hoc analysis on speed-up of all the four GAPP variants. The boldface shows the significance at $\alpha = 0.05$, while asterisk (*) shows the best variant. ................................................. 167

7.2 The mean best generation (MBG $\pm$ [standard deviation]) of all the four GAPP (Unoptimized, Grammar, Time, Combined) variants. The lowest value is in boldface. ............... 169

7.3 Friedman tests with Hommel’s post-hoc analysis on the speed-up of the four GAPP variants for 4, 8 and 16 cores. The boldface shows the significantly different results at $\alpha = 0.05$ and asterisk (*) shows the best variant. ......................... 174

7.4 The mean best generation (MBG $\pm$ [standard deviation]) of all the GAPP (Unoptimized, Grammar, Time, Combined) variants. The lowest generation is in boldface. ............... 175

7.5 The mean best execution time (in secs) (MBT [standard deviation]) of GAPP (Unoptimized, Combined) and the optimization flags (O1, O2, O3). The boldface represents the lowest execution time. ................................. 177

7.6 Friedman tests with Hommel’s post-hoc analysis on MBT of GAPP (Unoptimized, Combined) and O1, O2, O3 at $\alpha = 0.05$. The asterisk (*) shows the best method, while significantly different methods are in boldface. ............... 178

8.1 The mean best generation (MBG $\pm$ [standard deviation]) of GAPP (Grammar, Combined, Scaled). The lowest value is in boldface. ................................................. 189

8.2 Friedman tests with Hommel’s post-hoc analysis on speed-up and MBG of GAPP (Unoptimized, Grammar, Time, Combined, Scaled). Boldface shows the significance (at $\alpha = 0.05$) and asterisk (*) shows the best variant. ............... 190
8.3 The computational complexity of the evolved parallel programs compared with that of the ideal \((n\text{ processors})\) as well as the state-of-the-art evolutionary implementations on the four iterative sorting benchmarks.

8.4 GAPP (Chunk) evolved chunk size (mean ± [standard deviation]), averaged across 50 runs for all the four experimental problems.

8.5 Significance tests (at \(\alpha = 0.05\)) show that GAPP (Chunk) outperforms GAPP (Combined) for 8 and 16 cores. Note that “\(\checkmark\)” states that the results are significant \((p\text{-value} < 0.05)\). A measure shows the probability at which, GAPP (Chunk) is better over GAPP (Combined).

8.6 The mean best generation (mean [standard deviation]) of all the GAPP (Unoptimized, Grammar, Time, Combined, Scaled) variants. The lowest generation is in boldface.

9.1 Mean best execution time (in secs) (MBT [standard deviation]) of ALP on different cores versus GE that generates sequential programs.

9.2 Wilcoxon Signed Rank Sum tests (at \(\alpha = 0.05\)) show that ALP outperforms serial programs on all the cores except 2. “\(\checkmark\)” indicates that the results are significant. A measure shows the probability at which, ALP is better over GE.

9.3 Wilcoxon Signed Rank Sum tests (at \(\alpha = 0.05\)) show that ALP outperforms GAPP on all the cores except 2 and 4. “\(\checkmark\)” states the results are significant and “\(\not\checkmark\)” stands for insignificant difference. A measure shows the probability at which, ALP is better than GAPP.

9.4 Efficiency of ALP and GAPP (Combined).

9.5 Efficiency of the best programs of ALP versus human written programs for 8 and 16 cores with the significance tests.
List of Figures

2.1 Flowchart of a basic evolutionary algorithm. ....................................... 14
2.2 Examples of GA binary string individuals undergoing crossover and mutations. Here, parent 1 and parent 2 exchange genetic material at the specified crossover points producing offspring (child 1, child 2). The offspring after performing the mutation are also presented with the specified mutation points. ........................................ 16
2.3 Sub-tree crossover in GP, where the crossing over sub-trees are in boldface. ................................................................. 21
2.4 Point mutation of parse trees in GP, where the mutating node is in boldface. ................................................................. 22
2.5 An example CFG in BNF that generates a binary string. .... 24
2.6 A comparison between the transformation of biological DNA to protein with that of GE transforming a binary string to a computer program. ........................................ 29
2.7 One point crossover in GE with a flexible selection of crossover points. The exchanged genetic material is underlined in the resulting offspring. ........................................ 33
2.8 A point mutation in GE on an integer string, where the codon changes in the integer string are highlighted. .... 33
2.9 An example CFG for symbolic regression. ........................ 35
2.10 An example GE codon string. ........................................ 35
3.1 Flynn’s taxonomy of extended computer architectures: single instruction single data (SISD), multiple instruction single data (MISD), single instruction multiple data (SIMD), and multiple instructions multiple data (MIMD). ........... 48

3.2 The block diagram of a symmetric multiprocessor (SMP) system. ................................. 51

3.3 A multi-core processor with a shared L3 cache. .......................... 52

3.4 An example of a main thread forking two threads and joining them back into main thread after completion of their task. ....... 62

3.5 OpenMP fork-join programming model. ...................... 63

3.6 Classification of automatic parallel code generation ........... 70

3.7 A high level overview of auto-parallelization process. ......... 72

3.8 A survey of evolutionary automatic parallelization techniques. 77

4.1 The time complexity graphs of Santa Fe Ant Trail (shown left) and Binomial-3 (shown right) problems both in the presence and in the absence of wrapping on a log-log scale. 89

4.2 Average Terminating Generation (ATG), Average Actual Length (AAL), and Average Effective Length (AEL) of SantaFeAnt-Trail with wrapping OFF (left), wrapping ON (right) at various cumulative probability of successes across different population sizes. ................................. 94

4.3 Average Terminating Generation (ATG), Average Actual Length (AAL), and Average Effective Length (AEL) of Binomial-3 with wrapping OFF (left), wrapping ON (right) at various cumulative probability of successes across different population sizes. ................................. 95

4.4 A block diagram of the run prediction model (RPM) with GE. 100

4.5 The flowchart of Run Prediction Model (RPM) that predicts the performance of GE runs. ...................... 102

4.6 ACO discovered predictive model for problem \( f_1 \) ............... 107
4.7 The best fitness of standard GE (shown in squares) versus RPM+GE (shown in triangles) with the terminated runs (shown in circles) for the four symbolic regression problems. The horizontal dashed line represents the pre-defined success threshold. The end of run results of all four problems illustrate an improvement in the quality of runs after applying the prediction model.

5.1 An overview of GAPP parallel program generating system.
5.2 Design of GAPP grammar for symbolic regression problem.
5.3 Parallel evaluation strategy of the evolved parallel program, Program X, on a multi-core CPU, where different data is processed on each core.
5.4 An example of the best-of-run parallel sextic polynomial regression program.
5.5 Efficiency of the evolved parallel programs for 11-multiplexer and sextic polynomial regression (1024 fitness cases).
5.6 A comparison of average actual and effective genome lengths of standard GE versus GAPP. The results are presented for 11-multiplexer and sextic polynomial regression (1024 fitness cases), where standard GE uses a single core processor while GAPP uses a 16 core processor. The actual lengths of both the approaches are significantly larger than the effective lengths.

6.1 Recursive Fibonacci program generates a sequence of integers.
6.2 Control flow structure of Fibonacci program recursive calls.
6.3 GAPP grammar to evolve parallel recursive Fibonacci programs.
6.4 Mean best fitness (with standard deviation) of GAPP evolved recursive programs that is averaged across 50 runs of 100 generations. The results are for const-10, cond and const-limit.
6.5 An example evolved parallel recursive Fibonacci program that uses OpenMP parallel work-sharing construct.

6.6 An example best evolved task parallel recursive Fibonacci program that uses OpenMP sections work-sharing construct.

6.7 GAPP grammar that produces a parallel iterative Quicksort program. It contains various OpenMP parallelization primitives. The grammar allows the evolution of loops, branches, array indexes, etc.

6.8 The mean best fitness (with standard deviation) of GAPP evolved parallel iterative sorting programs that is averaged across 50 runs of 100 generations, where the graphs represent the results of const-10, and const-limit.

7.1 Segregation of the task and data parallel primitives in the design of the GAPP grammars that generate efficient parallel recursive programs. The complete Fibonacci grammar is presented in Figure B.10 of Appendix B.

7.2 The speed-up of GAPP (Unoptimized, Grammar, Time, Combined) variants for all the six experimental problems. The number of cores vary as 2, 4, 8 and 16. The horizontal dashed line (- -) represents the speed-up of 1 and acts as a reference for the remaining results.

7.3 GAPP grammar with the separation of task and data parallel primitives to evolve efficient parallel iterative sorting programs. Complete Odd-Even sort grammar is shown in Figure B.11 of Appendix B.

7.4 The speed-up of all the four GAPP (Unoptimized, Grammar, Time, Combined) variants on the four iterative sorting problems for 2, 4, 8, and 16 cores. The horizontal dashed line (- -) shows the speed-up of 1.

8.1 Actual and effective lengths of GE and GAPP (Unoptimized, Combined) genotypes. The lengths are averaged across 50 runs of 100 generations.
<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>8.2</td>
<td>The enhanced GAPP grammars to generate a parallel recursive Fibonacci program that is both serial and parallel.</td>
</tr>
<tr>
<td>8.3</td>
<td>GAPP (Scaled) evolved program that combines both parallel and serial execution to increase the performance.</td>
</tr>
<tr>
<td>8.4</td>
<td>GAPP (Scaled) evolved thread limiting constants for all the six recursive benchmarks averaged across 50 runs.</td>
</tr>
<tr>
<td>8.5</td>
<td>The performance of GAPP (Scaled) evolved programs for all the six recursive benchmarks on 2, 4, 8, and 16 cores.</td>
</tr>
<tr>
<td>8.6</td>
<td>The actual and effective lengths of GE, GAPP (Unoptimized, Combined) of the four iterative sorting benchmarks.</td>
</tr>
<tr>
<td>8.7</td>
<td>Enhanced GAPP grammars that generate an adaptable chunk size for thread scheduling.</td>
</tr>
<tr>
<td>8.8</td>
<td>Evolved Odd-Even program that combines both parallel and serial execution to increase the speed-up.</td>
</tr>
<tr>
<td>8.9</td>
<td>Performance of GAPP (Scaled).</td>
</tr>
<tr>
<td>9.1</td>
<td>Motivating example: Fibonacci program in OpenMP.</td>
</tr>
<tr>
<td>9.2</td>
<td>Enhanced GAPP grammar to evolve lock-free parallel recursive Fibonacci programs. Complete grammar is in Figure B.12 of Appendix B.</td>
</tr>
<tr>
<td>9.3</td>
<td>Speed-up in mean best execution time (MBT) of ALP (left) versus GAPP (right) for 2, 4, 8, and 16 cores.</td>
</tr>
<tr>
<td>9.4</td>
<td>ALP evolved parallel lock-free recursive Fibonacci program.</td>
</tr>
<tr>
<td>A.1</td>
<td>ACO discovered predictive model for problem $f_2$.</td>
</tr>
<tr>
<td>A.2</td>
<td>ACO discovered predictive model for problem $f_3$.</td>
</tr>
<tr>
<td>A.3</td>
<td>ACO discovered predictive model for problem $f_4$.</td>
</tr>
<tr>
<td>B.1</td>
<td>Design of the GAPP grammar for a 11-multiplexer problem, where A in &lt;var&gt; stands for address bits and D refers to the data bits.</td>
</tr>
<tr>
<td>B.2</td>
<td>GAPP grammar that generates a parallel recursive Sum-of-N program.</td>
</tr>
<tr>
<td>B.3</td>
<td>GAPP grammar that generates a parallel recursive Factorial program.</td>
</tr>
</tbody>
</table>
B.4 GAPP grammar that generates a parallel recursive *Binary-Sum* program. ........................................ 229

B.5 GAPP grammar that generates a parallel recursive *Reverse* program. ........................................ 230

B.6 GAPP grammar that generates a parallel recursive *Quick-sort* program. ................................. 231

B.7 GAPP grammar to generate parallel iterative *Bubble sort* program. ........................................ 232

B.8 GAPP grammar to generate parallel iterative *Odd-Even sort* program. ................................. 233

B.9 GAPP grammar to generate parallel iterative *Rank sort* program. ........................................ 234

B.10 GAPP grammar to generate efficient parallel recursive Fibonacci program. These changes are common to the problems in Figure B.2–B.6. ........................................ 235

B.11 GAPP grammar to evolve an efficient parallel iterative *Odd-Even sort* algorithm. These changes are common to the problems in Figure B.7–B.9 ........................................ 236

B.12 ALP grammar for lockfree parallel recursive Fibonacci programs. ........................................ 237
Publications

Most of this research has appeared in various reputed venues as listed below.


**Technical Reports**


Part I

Introduction and Background
Chapter 1

Introduction

“Well begun is half done.”

— Aristotle, Philosopher

The use of Shared Memory Processor (SMP) [Kalla et al., 2004], [Clabes et al., 2004], [Kongetira et al., 2005], [Le et al., 2007], [Sinharoy et al., 2011] architectures is on the rise, and promises potentially unlimited resources as the number of cores available on a single processor scale up. Unfortunately, producing software to exploit the hardware is a non-trivial task, and existing sequential software generally does not take parallel architectures into consideration.

This thesis presents a novel system, Grammatical Automatic Parallel Programming (GAPP) that automatically generates the native parallel code for these processors. Meaning that GAPP directly evolves compilable source code. The GAPP synthesized programs efficiently exploit the computational capacity of the on-chip multi-core processors. The programs effectively balance the workload, avoid locks, and scale better to the underlying multi-cores. The hypothesis of the thesis is that a machine learning system, specifically, Grammatical Evolution (GE), can be used to automatically generate parallel code that is competitive with that of human produced code. The
primary motivation to use GE over other evolutionary techniques lies in exploiting the flexibility offered through Context Free Grammars (CFGs), this is detailed in Chapter 2.

The rest of this chapter is organised as follows: section 1.1 briefly describes the motivation behind this research; section 1.2 presents the research questions and the objectives addressed in the thesis; section 1.3 accounts the contributions, and finally, section 1.4 presents the structure of the thesis.

1.1 Motivation

Improved performance of single core processors began to stagnate (in the early 1990s) due to narrow instruction level parallelism, and more importantly, high power consumption that resulted from a higher number of transistors. Consequently, a new architecture, Chip Multiprocessor\(^1\), which integrated multiple cores on a single chip [Olukotun et al., 1996], was introduced. These processors share memory, operate independently and cooperate with each other in solving a single problem or different problems.

As these multi-core processors become the norm, researchers are motivated to build dozens, hundreds, and even thousands of cores on a single chip [Bell et al., 2008], [Howard et al., 2010], [Esmailzadeh et al., 2011]. On the other hand, high performance programmers identified that the software is trailing behind the rise of multi-cores [Mattson and Wrinn, 2008], [Stephen, 2010], [Patterson, 2010]. Software developers can no more rely on the clock speed of a processor to improve their application performance. This fact makes necessary for the programmers to write parallel programs that can take better advantage of multi-cores.

In exploiting parallelism, multiple threads need to be mapped onto multiple cores. Such parallelism is non-deterministic due to the unpredictable thread scheduling by an operating system. This non-determinism can be treated with semaphores, mutual exclusion, monitors, and locks. However,\(^1\)

\(^1\)They are commonly known as multi-cores, hereafter, we refer them with that term.
it is widely accepted that the multi-threaded code is complex to understand and error prone with concurrency bugs [Lee, 2006]. Standards such as MPI [Gropp et al., 1996] and OpenMP [Dagum and Menon, 1998] are used for parallel processing. MPI is used to exploit the parallelism on distributed memory processors while OpenMP exploits that on shared memory processors. Since multi-cores are shared memory processors, this thesis focuses predominantly on the use of OpenMP. Its functionalities include, loop and task level parallelism, synchronization, and load balancing. They offer much better higher level abstractions than that of the multi-threaded programming.

Parallel programming is challenging when compared to sequential programming [O’Boyle and Bull, 1996]. Challenges include identifying the available parallelism, configuring the shared data, use of locks to guarantee mutual exclusion, synchronizing and balancing the workload among multiple processes. In addition, architectural considerations such as cache memory require expert programming skills. That is, to control multiple threads from accessing the closely aligning data on the same cache line.

An alternative to explicit parallel programming is automatic parallelization, that is, the transformation of an existing sequential application into a semantically equivalent parallel code. Some automatic parallelization compilers include Polaris [Blume et al., 1994], SUIF [Amarasinghe et al., 1995], and Vienna Fortran Compiler [Benkner, 1999]. Automatic parallelization is still difficult, where the burden moves from the software developer to a compiler engineer. One great difficulty with automatic parallelization is that the automatically transformed code can be virtually unrecognisable from the original sequential code, and often quite difficult to read [O’Boyle and Bull, 1996]. However, machine learning methods [Wang and O’Boyle, 2009], [Tournavitis et al., 2009], [Wang and O’Boyle, 2010] have come to the fore replacing those inflexible methods.

Realizing the difficulties of automatic parallelization, researchers are stressing the need for the development of new tools [Amarasinghe, 2008] to better exploit multi-cores. Also, automatic generation of native parallel code is little explored. To this end, this thesis proposes GAPP that applies
an evolutionary algorithm to synthesise parallel code. The following section elaborates further on the aims of the thesis.

1.2 Research Aims

This thesis aims to tackle some of the most important questions regarding the effective use of available computational resources, the synthesis of parallel programs, the analyse how different alterations in grammar and evolutionary parameters enhance or reduce the performance of those parallel programs, and how to further optimize their performance.

1.2.1 Research Questions

In achieving the core aim of the synthesis of optimal parallel recursive programs, this research applies Grammatical Evolution (GE) [Ryan et al., 1998]. In relation to the set goal, the following questions are raised.

Research Question 1: How much time does it take to complete the search process of a GE run?

Research Question 2: Can a GE run optimally utilize the available computational resources?

Research Question 3: Is it possible to predict the success or otherwise of a GE run?

Research Question 4: Can GE be applied for parallel program synthesis?

Research Question 5: How does a parallel grammar influence the search space of GE?

Research Question 6: How does the grammar design affect the evolvability of GE?
Research Question 7: How does the modifications in fitness evaluation influence the performance of GE?

Research Question 8: Does the design alterations in grammar facilitate the synthesis of optimal parallelism?

1.2.2 Objectives

This following objectives address the aims and research questions.

1. Conduct a survey into the state-of-the-art program synthesis techniques in evolutionary computing.

2. Conduct a survey into the ad hoc parallel programming techniques used in industry.

3. Calculate and optimize the execution time of GE. Optimizing is by reducing the time taken for fitness calculation.

4. Predict the performance (success or otherwise) of GE.

5. Design parallel-aware grammars for GE.

6. Speed up execution by executing each individual across multiple cores.

7. Empirically test the grammars on a selection of problems, including those that are inherently parallel and standard problems in literature.

8. Test grammars for the absence of cycles and race conditions.

9. Combine the grammars with existing coarse and fine grained parallel techniques to ascertain the true speed up.

10. The output programs must be provably correct with respect to their serial equivalents due to the use of grammars to map the individuals.

11. Synthesis of optimal parallel programs that are human competitive.
1.3 Contributions

Many of the contributions of this thesis have given rise to a number of publications, which are enumerated on page xxvi. Following are the research contributions of this work in the order of appearance of this thesis.

**Survey of Evolutionary Automatic Programming:** An extensive review of various evolutionary attempts for automatic programming, which accomplishes *Objective 1* and are discussed in Chapter 2.

**Survey of Automatic Parallel Code Generation:** A brief literature review on the rise of multi-core processors, parallel programming for these processors and automatic parallel code generation techniques, which achieves *Objective 2*. This also reviewed the evolutionary attempts for automatic parallel code generation that are presented in Chapter 3.

**Analysis on the Time Complexity of GE:** An analysis on the computational complexity of GE in solving the problems both in control and classification domains. It fulfils *Objective 3*, which is described in Chapter 4.

**Predicting the Performance of a GE Run:** Applied an Ant Colony Optimization (ACO) based classifier to predict the problem solving ability of a GE run. It is demonstrated that terminating a GE run that fails to solve the problem has actually improved the search results in less time. This contribution achieves *Objective 4* as shown in Chapter 4.

**GAPP for the Synthesis of Parallel Programs:** Automatically generated native parallel programs by defining OpenMP parallelization primitives in Context Free Grammars (CFG) of GE. This is the first time an evolutionary algorithm has been used in a generic to produce fine grained parallel programs. It implements the *objectives 5, and 6* in Chapter 5.

**GAPP for the Synthesis of Tractable Programs:** Designed the grammars such that native task and data parallel programs are automatically generated using GAPP. This measured the quality of the evolved programs among various alternatives of grammars, which satisfies *Objective 7* as shown in Chapter 6.
Execution Time and Grammatical Bias in GAPP: Considered the execution time of the evolving parallel programs in their fitness evaluation and induced bias through grammar. Thus, the degree of parallelism has improved, thereby improved the performance and eased the evolvability of both task and data parallel programs. This contribution meets the objectives 8, 9, and 10 in Chapter 7.

Optimized the Performance of GAPP: Reduced the excessive creation of threads on multi-cores, which further optimized the performance of the evolving programs. This showed that the evolved programs scale better at higher number of cores and are human competitive, fulfilling the objectives 9, 10, and 11 as listed in Chapter 8.

Application of GAPP for Lockfree Programs: Applied GAPP to synthesize lock-free parallel recursive programs, a much harder task. This work has demonstrated that the evolving programs are competing with human written programs. It realizes the Objective 11 as in chapters 8 and 9.

Rich Repertoire of Parallel Grammars: This thesis created a rich repository of parallel grammars that contain OpenMP primitives to evolve both task and data parallel programs. The resultant parallel grammars can be seen in Appendix B.

The advantage of a system like GAPP is that it uses grammars to create its individuals/programs; these grammars can be used to map the individuals onto multiple cores and to enforce various parallel constraints.

1.4 Thesis Structure

The research carried out in this thesis is organized into four parts.
Part I: Introduction and Background  The first part of the thesis consists of three chapters which introduce the problem space and the related technologies.

Chapter 1 briefly introduces the multi-core processors and the problems associated in programming them. This chapter outlines the aims, research questions, objectives and the research contributions of this thesis.

Chapter 2 is a historical review of evolutionary automatic programming techniques. It also describes the inner workings of GE algorithm, search operators, applications and the extensions of GE.

Chapter 3 describes Flynn’s taxonomy of parallel architectures, particularly, the surge of shared memory multi-core systems in the processor industry. This chapter then discusses the challenges in programming these systems. Finally, this chapter reviewed both the conventional and evolutionary attempts for automatic parallel code generation.

Part II: Analysis of GE and Parallel Program Synthesis  The second part of the thesis consists of three chapters which describe the complexity analysis, GAPP methodology, and the proof of concept of GAPP on tractable problem domains.

Chapter 4 analyses the computational complexity of GE to gain insights into its operation. It observes the quality of the evolved solutions that vary across runs. This chapter then proposes a reasonably good prediction system that identifies how a particular run will perform. The prediction system learns from the early generations of GE to decide the success or otherwise of a run.

Chapter 5 presents GAPP, an application of GE for the evolution of native parallel programs that exploit the computational capabilities of on-chip multi-cores. This chapter incorporates OpenMP parallelization primitives in GE grammars, so that a single program executes on multiple cores while operating on multiple data items.

Chapter 6 examines GAPP in two different computable problem domains: recursion and iterative sorting. This chapter compares different
variations of GAPP that describes the quality and the performance of the evolving parallel programs in both the problem domains.

**Part III Synthesis of Optimal Parallel Programs**  The third part of the thesis is comprised of three chapters that show the design modifications in parallel grammars, a study on the effect as a result of these tweaks, and the synthesis of optimal parallel programs.

*Chapter 7* extends GAPP presented in previous chapters by changing the design of the grammars and the fitness evaluation function. These changes are designed into four different variants of GAPP. These changes contribute to improve the performance of the evolving parallel programs in both the problem domains on multi-core machines.

*Chapter 8* deepens the study carried out in the previous chapter with an analysis on the code growth of the evolving individuals. It analyses the scheduling issues that arise in the resultant OpenMP programs. In turn, it presents a few modifications in the grammar that improve the scalability of the evolving parallel programs.

*Chapter 9* applies GAPP to automatically generate lock-free parallel recursive programs. These programs perform significantly better over the sequential programs as well as the parallel programs that include locks.

**Part IV Conclusions and Future Directions**  The fourth part contains a single chapter that concludes and recommends future research aspirations.

*Chapter 10* summarizes the contributions of this thesis and suggests possible future directions.
Chapter 2

Evolutionary Automatic Programming

“Study the past if you would define the future.”

— Confucius, Teacher

This chapter briefly discusses evolutionary computing techniques, particularly those that can be used for the task of automatic programming. It also explains the role of context free grammars in evolutionary computing, before describing the inner workings of Grammatical Evolution and the key advances in the field since its inception.

2.1 Introduction

Manual programming is the conventional approach to produce useful computer programs, and, since the late 1970s, has been treated as the “craftsman approach” for programming. With the introduction of von Neumann architecture [von Neumann and Taub, 1945] in 1945, this programming style assisted in executing the machine instructions of a program directly on a processor.
Later, in 1959 Arthur Samuel [Samuel, 1959] preached, “Tell the computer what to do, not how to do it.” That is, to automatically generate computer programs depending on the high-level description of a given problem at hand. The definition of what is considered “automatic” has varied over time. For example, in the early days of computing, automatically generating machine code from an assembly language program was considered to be Automatic Programming. At that time, an assembler automated the process of physically writing the binary codes. Thereafter, Fortran programming language compilers were treated as Automatic Programming. Today, with the advancements in computer science, no one would call the use of an assembler or a compiler as automatic programming [Rich and Waters, 1988], [Rich and Waters, 1993].

In 1958, the first attempt by Friedberg [Friedberg, 1958] evolved simple machine code programs for arithmetic calculations such as addition of two numbers. In this approach, the binary string structures were changed randomly while the instructions in a program were also swapped. Later, this approach was extended [Friedberg et al., 1959] to guarantee an improved performance in generating the programs. In the extended method, the instructions were changed rigorously. Next, Fogel et al., [Fogel et al., 1966] evolved finite state machines that was followed by Crammer [Cramer, 1985] with a goal to evolve a programming language. A detailed description of these two attempts are given in section 2.5 and section 2.6 respectively.

Following the lines of developing more sophisticated compilers, in 1990’s came the automatic parallelization of code, see for example [Lovely, 1992], [Foster, 1991], [Loveman, 1992], [Blume and Eigenmann, 1992], [Wilson et al., 1994], [Blume et al., 1994], and [Benkner, 1999], most of which use conventional computation techniques. At around the same time, a few researchers [Williams, 1998], [Nisbet, 1998], and [Ryan, 1999] leveraged evolutionary computing techniques for automatic code parallelization.

Koza described a number of characteristics [Koza et al., 1999] for an automatic programming approach. They include a high-level description of a problem, automatically determining the program size and reordering the instructions so that they can be reused in a program, problem independence,
ability to produce human competitive results. He identified that evolutionary automatic programming approaches rightly suit in realising these characteristics.

Taking advantage of Koza’s findings, more recent contributions leveraged these techniques to achieve more sophisticated goals in automatic programming. Example activities are automatic program repairs [Weimer et al., 2010], evolutionary improvement of programs [White et al., 2011], automated the design of algorithms [Haraldsson and Woodward, 2014], optimizing the existing software [Langdon and Harman, 2015].

Before delving further into evolutionary automatic parallel code generation, it is important to discuss some of the evolutionary algorithms that contributed towards automatic programming.

### 2.2 Evolutionary Automatic Programming

The field of evolutionary computation (EC) was inspired from Darwin’s theory of biological evolution through natural selection [Darwin, 1859] and assimilates Spencer’s notion of survival of the fittest [Spencer, 1864]. The EC techniques hold a population of candidate solutions for a given problem, with each candidate solution being evaluated to measure its quality. This quality of a candidate solution on the problem is referred to as its fitness. The candidates with a better fitness value are considered fitter, and are more likely to survive to the next generation. Conversely, poorly fit candidates are less likely to propagate to the next generation.

Figure 2.1 shows the step-by-step procedure of an evolutionary algorithm. The population of an EA must be initialised before evolution can commence; typically, the initial set of solutions in the population are generated randomly.

After the initialisation, the remaining steps in the evolutionary cycle of an EA are: evaluation, selection, genetic manipulation, and replacement. In evaluation, the quality of the population of individuals is estimated; that is, the fitness or the ability to solve the target problem. It is followed by the
Figure 2.1: Flowchart of a basic evolutionary algorithm.

selection of parent individuals based on their fitness. Selection of parents to generate offspring is an interesting phenomenon both in nature and in EAs. In nature, selecting parents from the same gene pool results in producing blotched offspring. In EAs, two fit parents (mom, dad) may produce fitter offspring, but also one fit and one less fit parents may also do so. Sometimes, only selecting the fittest individuals may result in premature convergence.

The parents undergo certain genetic manipulations and, of course, mate to produce offspring. Depending on the fitness of the parents, children and the other members of the current population, a new population of individuals are created for the next generation. This process is terminated upon satisfying the stopping criterion. Typically, a terminating condition is: i) coming to a maximum number of generations, ii) exhausting a time limit, or iii) the population converging to an optimal point in the search space.

An entire family of EAs exists that varies in some aspects yet shares several other characteristics. One uniformly prevalent characteristic though is that all the EAs are based on some form of the survival of the fittest. The primary members of this family are: Evolution Strategies (ES), Evolutionary
Programming (EP), Genetic Algorithm (GA), Genetic Programming (GP), and Grammatical Evolution (GE).

The conventional difference among the EAs is the adopted representation for the respective algorithm. Generally, the population of individuals are fixed-length binary strings in GAs, real-valued vectors in ES, finite state machines in EP, lisp S-expressions in GP, and variable-length integer strings in GE. Also, the best strategy to solve a problem is to mix the best characteristic from each of these algorithms [De Jong, 1999], rather differentiating among the EA family members.

As the primary focus of this thesis is to automatically generate parallel programs, first, we discuss those EAs that facilitate Evolutionary Automatic Programming.

2.3 Genetic Algorithms

The best known EA is Genetic Algorithm (GA), introduced by Holland [Holland, 1975] and Goldberg further popularized by formalizing through schema theory [Goldberg, 1989].

The original GAs operate on fixed length binary strings, where these strings are coded to suit the requirements of the target problem. The genetic operations that are applied commonly are roulette wheel selection, one-point crossover and bit-flipping mutation.

**Initialisation, Evaluation and Selection** Evolution in GA commences with an initial random population of binary strings. The fitness of these individuals is calculated, which refers to the problem solving ability of each individual in a population. The two parents are chosen using a fitness proportional selection strategy, typically *roulette wheel selection*, although other types may also be used. These schemes generally are stochastic processes, in which parents are drawn as per their fitness in comparison with other individuals in the population. More specifically, the probability of selecting
Figure 2.2: Examples of GA binary string individuals undergoing crossover and mutations. Here, parent 1 and parent 2 exchange genetic material at the specified crossover points producing offspring (child 1, child 2). The offspring after performing the mutation are also presented with the specified mutation points.
an individual \((i)\) is given as shown in equation (eq.) 2.1 below.

\[
P_i = \frac{f_i}{\sum_{k=1}^{N} f_k}
\]  

where, \(f_i\) stands for fitness of an individual \(i\) and \(N\) is the total number of individuals in a population. Conceptually, each individual gets a portion of the roulette in proportion to its fitness. Upon spinning the roulette, an individual is selected, where individuals with larger portion on the roulette will have more chance.

Another method is \textit{tournament selection} [Miller and Goldberg, 1995], which does not need the fitness details of the whole population as in roulette wheel. The competition among the individuals is in a small sub set of population rather than an entire set of population. A tournament size is randomly chosen, from which the best individual is selected for reproduction. This method prevents the immediate selection of a best individual in the population. Other selection methods such as \textit{truncation} [Mühlenbein and Schlierkamp-Voosen, 1993], \textit{ranking} [Whitley, 1989] selections also exist, but, none of them are of our interest in this thesis.

\textbf{Genetic Manipulations} \hspace{1em} The two selected parents are recombined using crossover to produce two children. A crossover point is selected randomly on the two individuals. The genetic material is exchanged between both the parents at the selected crossover points. Mutation slightly changes an individual, because of which, diversity is guaranteed. The mutation operator acts on a single individual at a time. Usually mutation changes a bit in a GA individual. Figure 2.2 shows these two genetic operations in GA.

\textbf{Replacement Strategy} \hspace{1em} Population size of any EA is fixed, but, the genetic operations produce offspring, thereby increasing the population size. In order to maintain fixed size throughout the evolutionary cycle, the current population of individuals has to be replaced by the offspring. Prominently, GA uses one of the two replacement strategies, \textit{Generational} and \textit{Steady State}.

17
In generational replacement, the offspring resulted from the genetic operations form a new population replacing the current population (parent) of individuals. In this method, the new population of individuals may also be less fitter than their parent generation. Thus the population may lose the best individual. To avoid this, the best individuals in the parent population are simply copied (elitism) to the new population.

Steady state replacement strategy follows a simple principle of survival of the fittest. When a genetic operation generates offspring, then they are evaluated for their fitness. If their fitness is better than that of the worst in the population, then the worst fit individuals are replaced by the new individuals. GA replacement strategies are in [Syswerda, 1991].

Thus, the whole evolutionary process continues to enhance the fitness of the solutions through the combination of different genetic operations. GA is detailed in [Goldberg, 1989], [Mitchell, 1998].

GAs were not designed for the evolution of programs. Smith [Smith, 1980] addressed this difficulty with variable length strings to evolve complete rule based programs. This invention was important for the genesis of Genetic Programming [Cramer, 1985], [Koza, 1989].

### 2.4 Evolution Strategies

Evolution strategies (ES) [Rechenberg, 1973], [Bäck, 1996] is another interesting EA, developed by Rechenberg and Schwefel in 1960s at Technical University of Berlin. ESs initially dealt with the hydrodynamic problems, at which, the shapes of the equipment (bended pipes, burden on joint plates) were optimized. In an ES, \( \mu \) stands for population, and \( \lambda \) represents the offspring. The pool of the candidates for the next generation is indicated with either a ‘+’ and ‘,’. When comma is used, the individuals for the next generation are selected from the generated offspring; no members of the original population are selected. For a plus, next generation can be selected from the combined pool of the parents and the offspring. In contrast to GAs, ESs work on floating point strings.
2.5 Evolutionary Programming

Evolutionary programming (EP) is a predecessor of GP and was introduced by Fogel, Owens, and Walsh [Fogel et al., 1966]. EP was successfully used in sequence prediction through finite state machines (FSM). An FSM is a computer program that contains the graph of nodes (states). An EP population is randomly initialised, mutated and reproduced based on fitness.

Quite distinctively, EP population contains FSMs as individuals that change their state according to the input. The fitness of an FSM is evaluated with respect to the compliance of output symbols for a given input symbol. EP is a mutation rich EA, where offspring are generated by applying mutations with a uniform probability distribution on the copies of the current population. Gaussian distributions are used on numerical components, while a Poisson distribution decides the number of mutations to generate a new FSM. The mutations replace the available states such as add state, delete state, change initial state, and change the transition. In fact, EP does not use crossover; this was one of the big distinctions, and ultimately, something that made it less popular.

2.6 Genetic Programming

Inspired by Smith’s advancements in evolving a complete rule-based program, Cramer [Cramer, 1985] proposed a tree structure to represent a program. Following that, similar methods [Hicklin, 1986], [Fujiko and Dickinson, 1987] evolved LISP programs. However, Koza [Koza, 1989], in his 1989 paper, identified the importance of GP for the task of automatic programming. A complete description of GP is in [Koza, 1992], [Koza, 1994], [Koza et al., 1999], [Koza et al., 2003], [Banzhaf et al., 1998], [Poli et al., 2008].

GP overcomes the program evolving difficulty of GAs by representing the individuals as parse trees. The key in evolving solutions for the given
problems with GP is to identify the primitives, referred by Koza as function and terminal sets.

An example terminal set, denoted by $T$, is as follows:

$$T = \{x, y, z, 1.0\}$$

where, $T$ contains three variables $x, y, z$ and a constant 1.0. The function set, denoted by $F$, consists of arithmetic operators and trigonometric functions. For example:

$$F = \{+, -, *, /, \sin, \cos\}$$

A function set can also contain boolean operators, subroutines, pre-defined functions, etc. In solving a problem, it is necessary to satisfy sufficiency and closure properties.

Sufficiency is the supplied terminal and function sets must be able to represent a solution to a problem. Closure must hold on the function set, that is, all the elements of a function set are able to accept all possible input values generated by other functions or terminals. Closure ensures that only syntactically correct offspring are generated while exchanging the sub-trees between the parents. Although inessential, it is wise to have a prudent function set, as an unnecessarily large function set increases the search space [Koza et al., 1999] and thereby, slows down the evolutionary search.

After determining the primitives, the standard GP parameters such as genetic operators, population size, maximum tree depth, selection and replacement strategies are applied to a given run. Additionally, unlike random initialisation in GAs, Koza outlined a ramped half-and-half [Koza, 1992] initialisation method. This method facilitates structural diversity in a population. The population is divided equally among the individuals of a range of tree depths. For a particular depth of individuals, half of them are created using grow and the other half with full techniques. The full technique allows every tree branch to grow to its full depth resulting in an uniform tree, while with the grow method, tree branches grow with varying depths up to the maximum depth specified for that group.
Although the evolutionary cycle of GP is similar to that of GA, it employs a different crossover scheme due to its representation. In GP crossover, the sub-trees are swapped between the two parents, whereas sub-tree mutation can regenerate an entire sub-tree. Thus, the GP genetic manipulations produce a syntactically correct program.

![Sub-tree crossover in GP](image)

**Figure 2.3:** Sub-tree crossover in GP, where the crossing over sub-trees are in boldface.

Figures 2.3 and 2.4 exemplify crossover and mutation operations in GP respectively. Following the genetic manipulations, a new population of individuals is formed using one of the replacement strategies explained in section 2.3. This way, GP continues to evolve feasible solutions to a target problem.

GP has been applied to a wide range of problems: these include robotic control, design of analogue circuits, financial prediction, design of neural networks, and more recently, design of heuristics [Burke et al., 2009], [Hong et al., 2013].

There exists a number of extensions to standard tree-based GP, such as the use of Automatically Defined Functions (ADFs) [Koza, 1994], Indexed...
memory [Teller, 1994], and strongly typed GP [Montana, 1995] that tries to handle the complex data types with a restricted sub-tree crossover during the evolution. Whigham [Whigham and McKay, 1995], [Whigham, 1996] identified the use of grammars in a tree based GP as a means to deal with the data typing and to induce learnt bias. It resulted in generating individuals with more expressive syntax.

We will now discuss some of the evolutionary algorithms whose search is guided through context free grammars (CFGs). In fact, the CFGs play a vital role in evolving parallel programs during the course of this thesis. Hence, we describe CFGs, before discussing the grammar based evolutionary programming techniques.

### 2.7 Context Free Grammar

In formal language theory, a grammar is a mechanism to produce a set of strings in a particular language [Harrison, 1978]. Grammars consist of a set of rules that define the structure of the rules and words of a language. These rules allow the construction of syntactically correct strings. A rule can contain both terminal and non-terminal symbols. Terminal symbols are the alphabets in the language that can not be broken down further into
smaller parts by the rules of the grammar and can appear in legal sentences of the language, while non-terminal symbols are those parts of a language that must be replaced with the terminal symbols, and, will not be present in the final strings. In a derivation process, the rules are used to transform types of strings with both the symbols into the strings with terminal symbols only.

The most commonly used grammar in evolutionary algorithms is a Context Free Grammar (CFG). CFGs are a subset of Chomsky hierarchy, that is of type-2 formal grammar. A CFG is defined as follows:

**Definition 1 (Context Free Grammar (CFG))** A context free grammar, $G$, is denoted as a quad tuple and it is defined as follows.

$$G = \{N, T, P, S\}$$

where,

- $N$ is a finite non-empty set of non-terminal symbols;
- $T$ is a finite non-empty set of terminal symbols, where $N \cap T = \emptyset$;
- $P$ is a finite set of production rules of the form $P : N \rightarrow V^* \cdot V^*$ contains the set of all finite-length strings constructed from $N \cup T$ and $V^*$ can have an empty string.
- $S$ is a start symbol, where $S \in N$

The production rule, $A \rightarrow \alpha$ is termed as an expression, where, $A \in N$, and $\alpha \in V^*$, a non-empty string. It means that the non-terminal symbol $A$ can be rewritten as $\alpha$, irrespective of the context that it occurs. This property of the grammars is known as context free.

In general, a CFG can be represented in many forms, for example, Chomsky Normal Form (CNF) [Chomsky, 1959], Backus Naur Form (BNF) [Knuth, 1964]. The BNF is a commonly used notation, and the grammars in this thesis follow this notation. In a BNF, non-terminals are enclosed in between
Figure 2.5: An example CFG in BNF that generates a binary string.

angular brackets (<>), while terminals are represented normally. Each rule is separated with “|” symbol.

Figure 2.5 presents an example CFG in BNF that generates a binary string. In the example, the start symbol is <string>; thus, the derivation always begins with a rule of this symbol. There are two rules (0, 1) for the non-terminal <string>. If we choose rule 1, then the derivation becomes: <bit><string>

Now we have two non-terminals to derive. If we derive the left most non-terminal (<bit>), it has two rules (2, 3). If we choose rule 2 then the expression becomes: 0<bit><string>. Again, for the left most non-terminal, <string>, if we choose rule 1, then the expression becomes: 0<bit><string>

Now, for the left most non-terminal, <bit>, if we choose rule 3, the expression becomes: 01<string>. Then, for the non-terminal, <string>, if we choose rule 0, the expression becomes: 01<bit>

Finally, for <bit>, if we choose rule 2, the expression produces a binary string 010. This way, the grammar can produce a variable length binary string.
A sequence of non-terminals uses production rules to derive a string of terminal symbols. This string can be a sequence of binary numbers, a sentence with alphabets, mathematical expressions, or a computer program.

To summarize, this section has defined Context Free Grammar (CFG) for the generation of bit-strings. We now continue discussing the use of grammars for the task of automatically evolving programs with evolutionary algorithms.

2.8 Grammar Based Evolutionary Algorithms

Perhaps the first instance of employing grammar for automatic programming is by Hicklin [Hicklin, 1986], used with genetic algorithm. Antonisse [Antonisse, 1991] proposed a grammar based GA that evolved linear strings. Johnson and Feyock [Johnson and Feyock, 1991] automatically generated LISP-like expert system rule bases through the use of variable length integer strings and a CFG.

Banzhaf [Banzhaf, 1994], inspired by the neutral theory of molecular evolution ¹ [Motoo, 1983], adopted a genotype-phenotype mapping (GPM) process in GP. This separates the search and solution spaces. The genetic search continues in search space, whereas the GPM translates a genotype to a phenotype (computer program).

Grammar based evolutionary search has attracted more attention both in tree based and linear evolutionary algorithmic techniques. We now discuss some of these approaches.

2.8.1 Tree Dependant EAs

Evolutionary algorithms that use derivation trees and employ grammars for program generation are discussed below.

¹different genotypes (genetic representation) can be transformed to the same phenotype (observed traits of an individual).
Whigham [Whigham, 1996] constrained the evolution to generate syntactically correct programs with the use of grammars. GP population is represented as derivation trees, at which, the genetic operations are applied.

Wong and Kwong-Sak [Wong and Kwong-Sak, 1996] presented a Generic Genetic Programming (GGP) system that employed logic grammars. Logic grammars are more expressive than CFGs, so that they induce both context-sensitive and domain knowledge to evolve the target program. GGP combines both GP and Inductive Logic Programming, that is the use of mathematical logic as a representation in order to learn programs in different languages.

Tree Adjunct Grammar Guided GP (TAG-GP) [Hoai and McKay, 2001] is a genetic programming system that employs Tree adjoining grammars [Joshi and Schabes, 1997]. TAG-GP extended Whigham’s tree based GP by incorporating tree adjunct grammars. In this approach individuals are derivation trees that are represented as a linear genome. Like Whigham [Whigham, 1996], TAG-GP also generates syntactically valid programs by applying GA style genetic operations. Then, Hoai et al., [Hoai et al., 2003] extended TAG-GP by changing the representation from linear genome to a derivation tree. The phenotype is termed as derived tree, where TAG facilitates the context sensitive bias. Later, Hoai et al., [Hoai et al., 2006] investigated the representation and structural difficulty in GP using TAG that solved the difficult GP problems quite easily.

The Logic based GP [Ross, 2001] system uses Definite Clause Translation Grammars (DCTG) [Abramson, 1984]. DCTG is a logical form of attribute grammar [Knuth, 1990], which is a CFG that also supports the context-sensitive information of a language. This system allows the important semantic aspects of a language to improve the evolutionary search. In fact, it also applied in [Ross, 2002] to study the stochastic regular motif language for protein sequences.

Stochastic Context Free Grammar GP [Ratle and Sebag, 2001] automatically generates programs using a stochastic generative grammar. Along with the stochastic grammars, it employs a method to update the probabilities of applying each rule of the grammar. This depends on the use of the
production rules in the best program in the previous generation. A simple
variant of this approach maintains a single vector to store the probabilities
for the grammar. A maturer variant keeps a depth dependent production
probabilities vector in a derivation tree. Another method favours recursive
productions.

Bosman and De Jong [Bosman and de Jong, 2004] investigated a specific
estimation of evolutionary algorithms (EDA) for GP to optimize the struc-
tural feature of a design problem. They developed a probabilistic model that
resulted local structures through the application of CFGs.

Shan [Shan, 2005] proposed an EDA for GP to estimate the evolution
of tree solutions using grammars. It is termed as Program Evolution with
Explicit Learning (PEEL), which is built incrementally by keeping a table
for the production rules. These rules are modified as per the best individual
at each step. It does not use any of the genetic operations early in the deriva-
tion while the non-recursive ones are more likely to be selected at greater
depths.

Hasegawa and Iba [Hasegawa and Iba, 2006] proposed a Probabilistic
Model Building GP (PMBGP) that employed Bayesian networks to model
the evolving population in GP. This approach has been extended as Prob-
bilistic Context Free Grammar (PCFG) [Hasegawa and Iba, 2009] in order
to induce latent variable model for EDAs; that is, a variable is ascertained
from other observed variables.

Piaseczny et al., [Piaseczny et al., 2005] introduced Chemical GP for opti-
mizing the genotype-phenotype mapping. It evolves a cell representing the
dNA code by applying genetic modifications while the genotype-phenotype
translation is co-evolved. Similarly, Suzuki et al., [Suzuki et al., 2006] pre-
sented Chemical GA that co-evolved the DNA codes and small molecular
cells, where the cells fitness evaluation was optimized.

Grammar (LPCSG) with GP for the evolution of simulated snake-like robot.
LPCSG maintains a table of probability distributions for various productions
of a rule. These context-sensitive probabilities are updated with evolution.
This process helps to learn about both the context of the rules and their probabilities.

Keller and Poli [Keller and Poli, 2007] improved the performance of search on the heuristic space and resource utilisation using a grammar based GP hyper-heuristic. Poli and McPhee [Poli and McPhee, 2008] introduced an n-gram GP system that is an estimation of distribution algorithm for the linear program evolution. An n-gram is an ordered list of n symbols for a language defined with some set of symbols. For example, in an English language sentence: “Peace cannot be kept by force”, the words, “kept by” and “be kept by” are 2-gram and 3-gram respectively. An n-gram model learns from the probabilities of possible n-grams from a given language. Later, n-gram GP was extended [McPhee et al., 2009] to allow developmental plasticity for the evolution of linear GP programs.

2.8.2 Linear EAs

Following the GPM in [Banzhaf, 1994], Keller and Banzhaf [Keller and Banzhaf, 1996] introduced a system called Binary GP (BGP). The genotypes in the BGP were linearly encoded strings of symbols that may or may not be syntactically correct. A mapping process was applied on each genotype to translate it into a syntactically correct phenotype; the mapping process used a CFG.

Genetic Algorithm for Deriving Software (GADS) [Paterson et al., 1996], [Paterson and Livesey, 1997], [Paterson, 2002] used CFGs for the automatic evolution of programs. Like BGP, GADS employed fixed-length binary string genomes in the genotype-phenotype mapping. GADS mapping process terminates upon reaching the end of the genome, at which point, the remaining non-terminals are replaced with their default symbols.

Similar to GADS, Freeman [Freeman, 1998] proposed a Context Free Grammars GP (CFG/GP). Her system uses fixed length linear string genomes where genes are integers. The gene values are restricted to the production rules in the grammar, while each gene can be used to map any non-terminal
in the partially mapped individual. As in GADS, this system also requires default rules in the case of mapping failures.

To avoid these issues of repairing an individual or using the default rules for mapping an individual, Ryan et al., [Ryan et al., 1998] proposed a novel genotype-phenotype mapping system, Grammatical Evolution (GE). This uses a variable length integer string genome as an individual and CFG to produce a computer program. We now discuss GE in detail.

![Diagram](image)

**Figure 2.6:** A comparison between the transformation of biological DNA to protein with that of GE transforming a binary string to a computer program.

## 2.9 Grammatical Evolution

GE [Ryan et al., 1998] evolves computer programs in an arbitrary language. GE differs with GP in many respects. As opposed to parse tree based indi-
viduals in GP, GE uses a linear genome representation. A GE individual is a variable length integer string, where GA style genetic manipulations are employed. Some advanced implementations of GE [O’Neill et al., 2008] support GP style operations that operate directly on derivation tree.

The integer strings are transformed into a computer program via a biologically inspired Genotype-Phenotype Mapping process. The mapping process characterizes the translation of DNA to a protein. Figure 2.6 compares the motivational biological approach with GE. In a manner analogous to DNA translating into a protein using amino acids, GE governs the integer string transformation into a program using a CFG specified in BNF. This allows a many-to-one mapping such that many individuals can be mapped to the same program. Note, for the experiments in the thesis the genotype is an integer string while the phenotype is the GE evolved computer program. We now discuss some of the inner workings of GE.

Algorithm 1 A pseudo algorithm of Grammatical Evolution (GE)

1: procedure GE(P, stop criterion)
2:     init_population(P); map(P); Evaluate(P);
3:     while (!stop) do
4:         parent1, parent2 ← Select(P);
5:         child1, child2 ← Crossover(mom, dad);
6:         child1, child2 ← Mutate(child1, child2);
7:         child1, child2 $\xrightarrow{CFG}$ $Z_1, Z_2$. /* map offspring */
8:         fitness = Evaluate($Z_1, Z_2$); /* map offspring */
9:     Replace_population_with_bestof(parents, offspring);
10:    endwhile return Best_Program(Z).
11: end procedure
Algorithm

Most of the steps in the evolutionary cycle of GE are similar to those in a GA or GP. The basic idea is to initialise a population of individuals and evaluate their fitness. At this stage, if a solution is found, evolution terminates, else genetic operations produce a next generation of individuals. This process is repeated until a solution is found or a terminating condition is satisfied.

Algorithm I shows the step by step procedure of GE evolutionary cycle.

Initialisation

The quality of the initial population of an evolutionary run can largely impact the success of that run. If the population contains individuals that are too small to represent a solution, then, clearly, the algorithm will fail to find a valid solution. Likewise, if the population is located in a narrow region of the solution space, then, the chance of convergence to local optimum is high. Each of these phenomena can lead to suboptimal performance. Ryan and Azad [Ryan and Azad, 2003] investigated these issues and proposed sensible initialization, which modelled ramped half and half method for derivation trees. More recently, Harper [Harper, 2009], [Harper, 2010] conducted an exhaustive investigation on GE initialisation. Harper concluded that certain distributions of tree shapes and sizes improved the GE performance. In general, the two most commonly used GE initialisation methods are described as follows.

Random Initialisation. As with most EAs, one of the commonly used initialisation for GE is random initialisation. In this method, end user specifies an initial length of the individuals. The random initialiser randomly generates a population of individuals that are of the size equal to the user supplied length. The quality and the variety of individuals in the population relies on the random number generator.
Sensible Initialisation. Often, in GE, random population of individuals perform poorly by converging to a sub-optimal solution or produce invalid individuals that fail to generate a complete phenotype. To address these issues Ryan and Azad [Ryan and Azad, 2003] introduced the sensible initialisation method, which is analogous to Ramped-Half-and-Half initialisation in GP. Since GE uses CFG, the growth of the derivation trees both in grow and full methods depend on the minimum depth and the recursive nature of the production rules. In this method, the tree grows by choosing the productions whose minimum depth is less than or equal to the remaining allowed depth. In the full method, if possible, only recursive rules are chosen, whereas in grow, both recursive and non-recursive rules are given equal probability. This method is inexpensive, and improves the GE search with a structurally more diverse population than random initialisation. On the other hand, tree representations that use GE implementation in [O’Neill et al., 2008] borrow Koza’s Ramped-Half-and-Half initialisation.

Selection

Like in other EAs, selection is an important step in GE. For almost all of the experiments in this thesis, we adopted the most frequently used selection method: fitness proportionate Roulette Wheel selection that has already been described in section 2.3.

Crossover

Like in any other EAs, discussed earlier, in GE, two parents have sex (crossover) to produce offspring (child1, child2). Crossover in GE is similar to that in GA, except the fact that it happens on variable length integer strings and results in a variable length offspring due to the selection of different crossover points in both the parents. Unlike GP, where crossover exchanges the sub-trees, GE exchanges the integer strings.

Figure 2.7 presents an example crossover on integer strings in GE. Once the two parents are selected, a crossover point is chosen on each individ-
One point crossover in GE with a flexible selection of crossover points. The exchanged genetic material is underlined in the resulting offspring.

Mutation

Unlike node replacement of a program tree in GP, the entire GE individual undergoes mutation in a fashion similar to that in a GA.

Figure 2.8: A point mutation in GE on an integer string, where the codon changes in the integer string are highlighted.

Figure 2.8 shows the integer mutation on an example individual in GE. A predefined mutation probability is applied on the whole integer string.
GE mutation has also been thoroughly investigated in [Keijzer et al., 2001], [Hugosson et al., 2010], [Hemberg, 2010].

**Replacement Strategy**

Like in a GA, GE uses one of the two replacement strategies, *Generational* and *Steady State*, explained in section 2.3.

**Steady State**  Ryan and O’Neill [Ryan and O’Neill, 1998] investigated the use of steady state replacement strategy in GE, which follows a simple principle of survival of the fittest. This study showed an improved GE performance by arresting the invalid GE individuals from progressing to the next generation, as is the case in generational strategy. GE often uses this method to form the next generation of individuals. We apply this strategy for all the experiments in this thesis.

### 2.9.1 Mapping

After the genetic manipulations, the variable length strings are translated to computer programs (Z). GE uses a simple *mapping* process to convert the genotypes to phenotypes. In the mapping process, inspired from nature as shown in Figure 2.6, the BNF grammar and a *mod rule* govern what legal structures can be produced. In GE, a genotype is a string of 8 bit integers and each integer is termed a *codon*; together these codons map a derivation tree by selecting the production rules from the CFG. GE mapping of a genotype to a phenotype is an iterative process. To facilitate this, each codon selects a production rule from the given grammar as shown in the following eq. 2.2:

\[
Rule = (\text{Codon Value}) \mod (\# \text{ of rules of non-terminal})
\]  

(2.2)

where *Rule* is an index to a production rule that is applicable in the present context of mapping a derivation. Next, we exemplify the mapping process.
Example Individual

We demonstrate the GE mapping process in this section. Figure 2.9 presents the grammar that generates arithmetic expressions. The symbols enclosed within angular brackets (<>) are non-terminals, others are terminals. The production rules are separated by |, and are indexed starting from 0 for each non-terminal. These indexes are used in mod rule calculation for any given non-terminal.

Figure 2.9: An example CFG for symbolic regression.

Figure 2.10: An example GE codon string.

Figure 2.10 represents an example GE genotype, that is, a string of codons. In GE, mapping begins with the start symbol, S in this example. Since S has only one production (<expr>), the first decision is how to pick
one of the three productions for $<\text{expr}>$. To decide, the genome emits the first codon 255 and $255 \mod 3 = 0$; thus, choosing $0^{th}$ production generates a developing phenotype $<\text{expr}> <\text{op}> <\text{expr}>$. Sticking to the left most non-terminal, and reading the next codon 158 modifies the phenotype to be: $<\text{var}> <\text{op}> <\text{expr}>$ (since $158 \mod 3 = 2$). Now $<\text{var}>$ has only two productions, the next codon 108 selects $X$ ($108 \mod 2 = 0$); the phenotype changes to $X <\text{op}> <\text{expr}>$. The process continues until we have a valid expression, that is:

\[ X * X + 1.0 \]

Naturally, the codons that appear earlier in the genome shape the derivation tree so that the codons appearing later only select rules as required by the partially generated derivation tree. Thus, there is a left to right dependency in how the codons in a GE genome are interpreted by the GE mapping process; therefore, if we change a codon at the start of the genome, the interpretation of the codons appearing later changes. Thus the effect of the change at the start of the genome $ripples$ through the rest of the genome; correspondingly, the one point crossover in GE is called the $ripple$ $crossover$ [Keijzer et al., 2001].

Since the genome sizes evolve in GE, usually enough codons are available to complete the mapping, but not always, sometimes it does not produce a complete individual. There are many reasons for the occurrence of such an undesirable result: For example, when the individuals are short, when a poorly designed grammar generates a never ending recursive derivation tree. Thus, it is possible that the mapping can terminate before using the entire genotype. Alternatively, the mapping remains incomplete, then a special operator, $wrapping$, can be used to reuse the genotype from the start. Thus, in GE, the $actual$ $length$ of a genotype may be different from the $effective$ $length$ which is the number of codons mapped. The actual length of a genome is usually longer than the effective length, on any problem. We conduct an extensive analysis on these two lengths in all the experiments conducted in this thesis. The ineffective part of the actual length of a genotype is referred to as a $tail$. 36
In order to prevent the mapping from entering into an infinite loop, we can set a pre-defined limit on the number of wrapping events as shown in [Ryan et al., 2003]. If the mapping fails even after reaching the set limit then the individual is considered as a worst fit individual. Nicolau et al., [Nicolau et al., 2012a] extensively investigated the mapping termination problem and established a link between the poor grammar design and mapping termination.

Recently, Fagan [Fagan, 2013] analysed the genotype-phenotype mapping in GE. This in-depth analysis resulted in adopting a breadth-first derivation instead of the depth-first derivation used in canonical GE. We now discuss several extensions and improvements of GE over the past number of years.

### 2.9.2 Extensions of GE

In this section we discuss various extensions to the canonical GE system. These extensions are listed in the chronological order of their invention.

**ALP** Unlike the derivation tree representation of individuals in [Wong and Kwong-Sak, 1996] and [Ross, 2001], Adaptive Logic Programming [Keijzer, 2002] employs GE for logic programming in Prolog. ALP uses logic programs instead of the CFGs of GE, but, follows the same individual representation and search strategy.

**Chorus** Chorus [Ryan et al., 2002a], [Azad, 2003], a position independent grammar based EA. Chorus used a modified mod rule such that the total number of rules are used, but not just the applicable rules as in GE.

**piGE** O’Neill et al., [O’Neill et al., 2004a] presented Position Independent GE that eliminates the position dependency in canonical GE mapping. This system uses two codons: one to determine the position at which the derivation expands; second, like in canonical GE, the codon selects the production rule.

GAuGE  Genetic Algorithms using Grammatical Evolution [Ryan et al., 2002b], [Nicolau, 2006] employs GE to evolve the GA coding gene positions. The system favours position independent genetic algorithms and exploits the mod operation to generate redundant coding. However, it is essentially a GA, and not an algorithm for automatic program generation.

BAP  Regolin and Pozo [Regolin and Pozo, 2005] combined Grammatical Evolution and Bayesian probabilistic models to induce programs. This system is termed as Bayesian Automatic Programming (BAP). Bayesian network in BAP considers the relationship among production rules in generating a program.

Shared GE  Luerssen and Powers [Luerssen and Powers, 2007], [Luerssen and Powers, 2008] presented Shared Grammatical Evolution that combined the grammatical development with grammars in GP in order to entrench declarative bias. They employed a global CFG that is extended with a user-defined grammar for program evolution. The production rules and the encapsulated sub-routines are shared among programs to allow reuse and reduce evaluations.
mgGE  Hemberg [Hemberg, 2010] has exclusively worked to assimilate meta-grammars in GE. He proposed two systems $GE^2$, Grammatical Evolution for Grammatical Evolution and meta Grammar Genetic Algorithm (mGGA). $GE^2$ applied GE in evolving grammar, which was incorporated back into GE in solving the target problem. mGGA applied meta grammar to identify modularity in GA.

CGE  Georgiou [Georgiou, 2012] extended GE as Constituent Grammatical Evolution, which incorporated constituent genes and conditional behaviour switching. This has reduced the genotype bloat, ripple crossover events, thereby the search space is limited towards optimal solutions.

SEOIGE  Byrne [Byrne, 2012] proposed Structural Engineering Optimization in GE for the interactive evolutionary design. This system interactively designs structures for some engineering problems using shape grammars.

PODI  Mc Dermott and Carroll [McDermott and Carroll, 2013] introduced Program Optimization with Dependency Injection. PODI evolves an optimal non-deterministic heuristic for many real world problems. They used the GE concepts to generate a general heuristic. Unlike GE mapping, PODI explores through the feasible space of non-deterministic programs, thus, the programs can be viewed as a result of the genotype-phenotype mapping.

TAGE  Another interesting extension, Tree Adjunct GE system was proposed by Murphy [Murphy, 2013]. TAGE uses tree adjunct grammars instead of the CFGs in GE. It was showed that TAGE has proved to generate better solutions than GE with a far better expressive power to represent the solutions.

Modular GE  Swafford [Swafford, 2013] extended GE to identify modules during the evolutionary run. The resultant best modules are incorporated back in the grammars to use in the subsequent run.
GSGE  Geometric Semantic Grammatical Evolution [Moraglio et al., 2014] is an extension of GE, in which, the search continues in the semantic space of the program, like in Geometric Semantic Genetic Programming [Moraglio et al., 2011]. In GSGE, semantic crossover and mutation operations are applied on the integer strings to avoid the disrupt behaviour [Keijzer et al., 2001], [Rothlauf and Oetzel, 2006] of search operators as well as to preserve the genotype-phenotype mapping modularity. i.e., phenotype modules (sub-trees) correspond to genotype modules (sub strings).

In summary, this section has provided a brief overview of advancing canonical GE in many a number of directions.

2.9.3 Applications of GE

This section describes some notable uses of GE in solving a number of problems from various domains.

As GE originated, it was applied on control, classification, regression problems, and to evolve caching algorithms for processors [O’Neill and Ryan, 2003]. From then onwards, a lot of the work has focused on financial modelling [Brabazon and O’Neill, 2006a]. These problems include GE’s applicability are: evolving classification models for corporate stability [Brabazon et al., 2002], anticipating bankruptcy [Brabazon and O’Neill, 2003], evolution of trading strategies [Brabazon and O’Neill, 2004], and forecasting the credit ratings [Brabazon and O’Neill, 2006b].

GE has also been used to tackle biological and medical domains. Azad et al., [Azad et al., 2004] employed Chorus to predict the Wall Shear Stress distributions in grafted artery. Escuela et al., [Escuela et al., 2005] used GE for protein structure prediction and inverse folding problems. Georgoulas et al., [Georgoulas et al., 2007] classified the fetal heart rate monitoring data. Their method extracted the correlation between the linear and non-linear features, which are fed back to the classifier. Smart et al., [Smart et al., 2011] detected epileptic oscillations in intracranial electroencephalograms of a patient using GE.
Another interesting domain is the design of structures. Hemberg and O’Reilly [Hemberg and O’Reilly, 2004] applied GE to get a generative model that evolves the design of digital surfaces of organic quality. O’Neill et al., [O’Neill et al., 2010] applied GE and shape grammars in designing shelters. Byrne [Byrne, 2012] evolved the design of bridge structures.

Computer gaming field has also received some attention from the GE community. Murphy et al., [Murphy et al., 2009] employed GE in animating the horse gait optimisation for graphical models. Galván-López et al., [Galván-López et al., 2010] applied GE to manoeuvre Ms.Pac-Man game controllers. Perez et al., [Perez et al., 2011] applied GE to evolve behavioural trees in creating the controllers for the Mario AI games.

We now briefly discuss applications of GE from a diverse array of domains. Reddin et al., [Reddin et al., 2009] and Shao et al., [Shao et al., 2010] evolved music using grammars. Sen [Sen and Clark, 2010] and Wilson [Wilson, 2008] applied GE to intrusion detection on different networking problems. Nicolau et al., [Nicolau et al., 2012b] employed GE to measure the Net Ecosystem CO2 Exchange between atmosphere and biosphere. Drake [Drake, 2014] evolved variable length neighbourhood in vehicle routing. Burke et al., [Burke et al., 2012] evolved hyper heuristics using GE for bin packing problem. Chen [Chen, 2011] studied reservoir inflow using GE in modelling non-linear time series. Recently, GE has been used as a program synthesis tool [O’Neill et al., 2014]; much like this thesis, however, here we present GE as a parallel program synthesis tool. The next section describes the open source implementations of GE.

2.9.4 Open Source Implementations

With the growth and wide acceptability of GE in many disciplines, availability of open source implementations have also increased. Now, there are a number of open source implementations of GE. Table 2.1 lists some popular implementations. Some of these implementations include not only GE but also a few other EAs discussed in this chapter.
Table 2.1: A list of open source software libraries for GE in different languages.

<table>
<thead>
<tr>
<th>Software</th>
<th>Language</th>
<th>URL</th>
</tr>
</thead>
<tbody>
<tr>
<td>ponyGE</td>
<td>Python</td>
<td><a href="http://code.google.com/p/ponyge">http://code.google.com/p/ponyge</a></td>
</tr>
<tr>
<td>PyNeurGen</td>
<td>Python</td>
<td><a href="http://pyneurgen.sourceforge.net">http://pyneurgen.sourceforge.net</a></td>
</tr>
<tr>
<td>GEM</td>
<td>Matlab</td>
<td><a href="http://ncra.ucd.ie/GEM">http://ncra.ucd.ie/GEM</a></td>
</tr>
<tr>
<td>GERET</td>
<td>Ruby</td>
<td><a href="http://geret.org">http://geret.org</a></td>
</tr>
<tr>
<td>DRP</td>
<td>Ruby</td>
<td><a href="http://drp.rubyforge.org">http://drp.rubyforge.org</a></td>
</tr>
</tbody>
</table>
2.10 Summary

This chapter has introduced Evolutionary Algorithms and discussed them in varying levels of detail. This chapter, by no means offers a detailed description of the EAs discussed, rather it serves as a collective representation of the state-of-the-art Evolutionary Automatic Programming techniques, a topic that is of particular relevance to the developments outlined in this thesis. Also, the chapter discussed the inner workings of Grammatical Evolution with greater details than the remaining evolutionary automatic programming techniques. It is because this thesis exclusively uses GE to achieve the desired goals.
Chapter 3

Multi-Cores, Parallel Computing and Automatic Parallelization

“A problem well put is half solved.”

— John Dewey, Philosopher

With the recent advancements in computational machinery, once again parallel computing has come to the fore. Accordingly, this chapter discusses the origins of hardware architectures and their evolution as powerful parallel processors over time. We discuss classification of parallel hardware, paying particular attention to shared memory processors, specifically the recent desktop multi-cores, before examining different shared memory programming alternatives. Finally, we review the literature on automatic parallel code generation both in conventional and evolutionary computing domains.

3.1 Introduction

Multi-core processors are shared memory multiple processing elements integrated on a single chip that offer high processing power. The advent of
multi-core (2 or more) processors for PCs has been little short of a revolution in terms of price/performance ratio. For example, the recent desktop processors such as Intel, AMD and IBM Cell are shipped with four and eight cores respectively. This shift in the number of processing elements is not all that surprising, particularly when we consider Moore’s law [Moore, 1965], which predicted the doubling of transistors on a microchip for every 18 or 24 months. Generally, the increase of transistors must increase the processor speed. But, because of the limitations in the physical dimension of the transistors, processor industry is driving towards manufacturing multiple processing elements on a single chip.

As the technology is evolving day to day so does the advancements in multi-core systems, and they will keep advancing for quite sometime. No matter how powerful these multi-cores are, their computational power needs to be exploited by the software applications explicitly. Following Gustafson’s law [Gustafson, 1988], it is possible to effectively exploit the computational capabilities of multi-cores. To some extent, compilers exploited the internal parallelism, through such mechanisms as pipelining, but, unfortunately, it is difficult for them to do so on multi-cores because of their inability to detect parallel parts of an application.

Since most of the programs are written for uni-core processors, they fail to exploit the presence of multi-cores. In order to make these programs run faster using the power of all the available cores, it is now necessary to write parallel programs or convert the existing sequential code to parallel.

Writing parallel programs on multi-cores improves the performance. The overall quality and speed of a parallel program often depends on the skill of the programmer. With a small number of cores, single processes or Virtual Machines can occupy each, but, as multi-core becomes many-core, the Intel Polaris and picoChip PC200 have 80 and 200+ cores respectively, the operating systems face difficulty in utilizing the power of all the cores.

On the other hand, converting a sequential program into a parallel one looks promising as it helps to avoid some level of human effort through the automation. However, researchers had limited success [Zima et al., 1988], [Vandierendonck et al., 2010] in this direction while the translation process
is also quite complicated and time consuming. However, the idea of automating the process of parallel program generation is always encouraging to discover novel approaches that reduce the human effort.

Before discussing the current automatic parallel programming methods, we discuss the existing parallel hardware architectures and their origins.

3.2 On the Origins of Computing Machines

This section discusses the hardware architecture of the computing machines that are both serial and parallel. Initially, the most famous serial machines are von Neumann architectures [von Neumann and Taub, 1945], [Burks et al., 1946]. They contain main memory, a central processing unit (CPU) or processor or core, and an interconnection between the memory and the CPU. A von Neumann machine executes one instruction at a time. When operating on voluminous data and a number of instructions, CPU interconnection transfers get jammed due to mismatch in the rate of CPU processing and memory, von Neumann bottleneck. Thus, the CPU remains idle till the data and/or instructions become available for processing. Computer scientists modified the basic von Neumann architecture for a number of times to overcome this bottleneck. For example, increasing the size of the interconnection, introducing cache [Handy, 1993] and virtual [Gorman, 2004] memories to reduce the distance between memory and CPU. Many of these modifications focussed only on the von Neumann bottleneck, whereas others focused on increasing the CPU speed. Since this thesis aims at performance, we briefly discuss some of those enhancements.

3.2.1 Improving Processor Performance

This section presents the enhancements to the von Neumann architecture in improving the processor performance. These modifications are mainly commissioned with the improvements in low-level parallelism and hardware multi-threading.
**Instruction-level Parallelism.** Instruction level parallelism (ILP) tries to improve the processor performance by simultaneously executing the instructions on multiple processor components. They are also called as *functional units*. ILP has two main approaches: *pipelining*, and *multiple issue*.

In pipelining, the functional units are aligned in stages. Their working principle is simple, where each functional unit is assigned a different task in regular CPU time intervals. In general, an $n$ staged pipeline may not offer an $n$-fold improvement in performance, because the functional units wait for other units to finish their tasks or for the availability of a resource.

In multiple issue, multiple instructions are initiated simultaneously. These processors replicate functional units and execute different instructions of a program simultaneously. It again has two modes of operation. If the compiler schedules the functional units then that multiple issue system is *static*. If they are scheduled at run-time then it is called *dynamic*. A processor that supports dynamic multiple issue is termed as *superscalar*. It is the job of the multiple issue system to find the independently executable instructions of a program. For that it follows a technique called *speculation*, where the compiler or the processor executes an instruction based on a guess. In the compiler speculation, compiler takes care of the correctness of a program. In processor speculation, a buffer stores the intermediate results before writing the final result in memory.

**Hardware Multi-threading**  Often, ILP can be hard to exploit, particularly when a program is composed of highly dependent instructions. Hardware multi-threading enables processors to do useful work even when their current execution gets stalled. For example, sometimes processors wait for a data item to be loaded from memory. In such instances, a new thread is created to run the same or a different task on a different data item. This type of thread-level parallelism offers better performance over ILP. This multi-threading is exceptional when the threads switch efficiently among themselves.

In thread level parallelism, the simultaneous execution of individual instructions, *fine-grained multi-threading* and program units, *coarse-grained*
multi-threading. In fine-grained parallelism, processor switches between the threads after executing each instruction. In this parallelism, more time consuming instructions have to wait for all the less time consuming instructions to finish their execution. Coarse-grained parallelism avoids this problem by switching the threads that wait for time-consuming instructions. Switching threads is not instantaneous and processors can stand idle. Simultaneous multi-threading exploits the superscalar processors by running multiple threads on multiple functional units.

The performance improvements through the above discussed parallelism techniques are invisible to the programmer. Programmers try to exploit the hardware by modifying their source code. For such purposes, we now continue our discussion on the classification of the parallel hardware.

**Figure 3.1:** Flynn’s taxonomy of extended computer architectures: single instruction single data (SISD), multiple instruction single data (MISD), single instruction multiple data (SIMD), and multiple instructions multiple data (MIMD).

### 3.2.2 Classification of Parallel Hardware

In 1966, Michael Flynn [Flynn, 1966] taxonomized computer architectures depending on the concurrent processing of data and instruction streams. In
this context, a stream is a sequence of data or instructions, which computer processes. His taxonomy was classified into four classes. This classification has been further refined with the advances in the computer architectures over the years, especially, with the relatively new parallel hardware such as the multi and many-core machines.

Figure 3.1 presents the Flynn’s extended classification of computer architectures with parallel systems and are described as follows:

**Single Instruction, Single Data (SISD)** The SISD system contains a single stream of instructions that handle single stream of data. A simple sequential machine operates one instruction on one data item at a time. This is commonly used in all single processor computers and the classical von Neumann systems are SISD systems while most of the contemporary machines do not belong to SISD.

**Multiple Instructions, Single Data (MISD)** The MISD system appears to be an idiosyncrasy. Precisely, when the question is: how does multiple instructions can be applied on single data stream? Usually, such a scenario is impractical. Systolic Arrays [Johnson et al., 1993] are considered as an example of this category, however, it is always debatable and not in the scope of our thesis.

**Single Instruction, Multiple Data (SIMD)** In a SIMD system, a single instruction stream operates on multiple data streams. In this configuration, same instruction is executed on different processors operating on a different data stream. An SIMD system can be thought of a single control unit with multiple ALUs. The control unit simulcasts an instruction to all the ALUs. These ALUs execute the instruction on a different data item. In classical SIMD systems, ALUs operate synchronously by waiting for the next instruction to be sent to all of them.

In late 1990s vector processors [Espasa et al., 1998] become widely available. Vector processors operate on an arrays or vectors of data and are considered as the first SIMD systems. Vector processors are very fast
on many applications. Vectorizing compilers also is a prosperous endeavour. Furthermore, if they fail to vectorize a loop, the reasons will be made clear for why not a loop can not be vectorized. This information can act as a guide for the prospective users. Another example is the Graphics Processing Units (GPUs), that contain many-cores operated by lightweight threads [Fernando, 2004].

**Multiple Instructions, Multiple Data (MIMD)** The MIMD system is a versatile configuration. It operates multiple instruction streams simultaneously on multiple data streams. In a MIMD system, each processing element processes its own set of instructions operating on its own data. These systems maintain independent processing units, each of which contain its own control unit and ALU.

Principally, MIMD systems are of two types: *shared memory systems* and *distributed memory systems*. This thesis is concerned about the shared memory systems and is detailed in section 3.3. A distributed memory system consists of multiple processors with each one paired with its own private memory. These pairs communicate with each other through a common interconnect, and can also access one another’s memory by sending special signals. As this is the architecture we are most concerned with, we now discuss the shared memory machines.

### 3.3 Shared Memory Machines

Shared memory machines can be further classified (as shown in Figure 3.1) as master-slave and symmetric multi-processors (SMPs). Master-slave systems have more than one processor, of which one acts as a master controlling all the remaining slave processors. In SMPs, all the processors are equal and are capable of executing any program.
3.3.1 Symmetric Multiprocessor (SMP)

All the processors in a shared memory system share the same address space and communicate with each other by accessing the shared resources.

![Block Diagram of a SMP System](image)

Figure 3.2: The block diagram of a symmetric multiprocessor (SMP) system.

Multiple processors operate independently while sharing the memory resources. Changes to a memory location by one processor are visible to all the remaining processors. Figure 3.2 presents the block diagram of a shared memory multiprocessor system. Multiple processors access the data stored in memory through a common interconnect. SMPs are further classified as *Uniform Memory Access* (UMA) and *Non-Uniform Memory Access* (NUMA) machines.

**UMA** In an UMA, all processors share memory in such a way that each of them can access any memory location in an approximately equal time with same speed. One drawback of UMAs is that they may not scale for large number of processors due to the interconnect bandwidth limitations.

**NUMA** A NUMA system is made of interconnecting more than one SMP. In a NUMA system, the memory locations of one SMP can be directly accessed by other SMPs. As opposed to UMA, in NUMA, the access time to various locations of the memory is different for each processor. This time depends on the region of the main memory. To diminish the non-uniformity
in memory accesses, a cache is maintained for each processor. These caches are coherent for an update by any processor to a memory location. Such systems are called as cache-coherent NUMA (cc-NUMA).

The current generation of multi-core processors are SMPs; we now continue to discuss multi-core architectures.

Figure 3.3: A multi-core processor with a shared L3 cache.

3.3.2 Multi-core Systems

Multi-core computers are also named as chip multiprocessors. A multi-core is that on which two or more processors or cores are combined on a single silicon chip. Typically, each core is treated as an independent processor, as it contains all the elements of a processor such as registers, ALU, control unit, cache, etc. Each processor in a multi-core system can either directly access the main memory or have a dedicated block of memory. Any processor can independently access these dedicated memory blocks.
Figure 3.3 shows the block diagram of a multi-core system that contain \( n \) cores per chip. Multi-core systems can also be built as UMA and NUMA architectures. All the attributes of both the architectures can be attributed to the multi-core systems that belong to the respective architectures.

Multi-core processors from various vendors differ primarily with their design of cache memory. Cache memory reduces the time required to access main memory. It can be placed on the same chip as the CPU or on a separate chip close to the CPU. Universally, the CPU tends to access the data and instructions close to recently used ones. In order to accomplish this principle of *locality*, cache blocks are effectively used rather than accessing individual data and instruction. Cache blocks typically contain blocks of data and instructions. Cache is divided into different levels (L1, L2, L3), where the lower level (L1) is faster and higher levels (L2, L3) are relatively slow. In accessing an instruction or data, the CPU first checks L1 then L2 and so on. The availability of the required information during this check is a *cache hit*, and the unavailability is a *cache miss*.

In general, multi-cores support two level (L1, L2) caches while the recent processors are supporting three level (L3) cache also. The L1 cache is divided into two, to store both instruction and data streams. There are different alternatives in the design of these caches. Some multi-core processors are designed in such a way that a dedicated L1 cache is kept on the chip with a shared L2 cache. Similarly, the growing need for increased cache memory due to increase in the number of cores, dedicated L1 and L2 caches per chip are manufactured to share an L3 cache. Figure 3.3 shows the design of dedicated L1 and L2 caches with a shared L3 cache. A potential advantage of having a dedicated L2 cache enjoys a more rapid access to its private memory through the principle of locality. As the number of cores increase, having a shared L3 cache is likely to offer better performance over simply a massive L2 cache.

Another major design difference in a multi-core system is whether the individual cores are superscalar or support simultaneous multi-threading. For example, the Intel Core Duo uses superscalar cores, while Intel Core i7 uses simultaneous multi-threading cores, also known as *hyper-threading*. 
Intel Core i7 simultaneous multi-threading scales up the number of hardware level threads such that four cores support two more simultaneous threads in each core. Thus, a quad-core processor appears as an 8 core processor for a program. The simultaneous multi-threading is more efficient than superscalar approach.

A number of processor vendors manufactured multi-core systems that basically differ in the design of cache, processing elements, memory system, etc. All of them contribute to the common cause of improving the performance. We now discuss the evolution of multi-core processors.

### 3.3.3 Evolution of Multi-core Processors

In recent years, various processor vendors proposed a wide variety of multi-core architectures. They have targeted every realm of the market starting from desktop, embedded, to server segments. In this section, we focus on four major processor manufacturers: IBM, Intel, AMD, SPARC. A complete description of multi-core systems can be seen in [Stallings, 2010].

**IBM** In late 1990s, IBM received attention in multi-core industry through the design of *IBM POWER4* multi-core processor for servers. IBM Power4 micro-architecture [Tendler et al., 2002] uses 64 bit instruction set. It has two cores manufactured on a single silicon die that offer a speed of 1.1–1.9GHz. It has a dedicated L1 and L2 (1.4MB) cache and a shared L3 (32MB) cache. This has been improved to *IBM POWER5* [Clabes et al., 2004], which is also a dual-core server that offers simultaneous multi-threading. It has further refined as IBM POWER6. Then, *IBM POWER7* [Sinhary et al., 2011] contains 4, 6 and 8 cores. Recently, massively multi-threaded processors, IBM POWER8, POWER9 and POWER10 are released that have an increased number of cores of up to 12 and offer a processor speed of 2.5–5.0GHz. They also include an extra shared L4 (16MB) cache along with a shared L3 (8MB) cache.
Intel  Intel has released a number of multi-core processors. We simply describe the Intel Core Duo and i7 processors. First, Intel introduced a dual core processor, Intel Core Duo [Gochman et al., 2006] in 2006. It has two x86\(^1\) superscalar processors with a shared L2 cache (2\(MB\)) and a dedicated L1 cache for each core. The L1 cache is divided into two, a 32\(KB\) instruction cache and a 32\(KB\) data cache. It has an advance programmable interrupt controller, because of which, a thread on one core can generate an interrupt which communicates with the thread on the other core. As the number of transistors increase on a single chip, power management becomes an important issue, especially, when the computing devise is a laptop or a mobile system. Each core of Intel Core Duo has an independent thermal control unit for chip heat dissipation in order to improve the processor performance. The power management control reduces the power consumption by monitoring the thermal conditions of cores and CPU activity to adjust the voltage levels approximately.

Later in 2008, Intel introduced four core Intel Core i7 processors. It supports x86 simultaneous multi-threading with each core having a dedicated L2 cache and a shared L3 (8\(MB\)) cache. Intel makes its caching effective by pre-fetching, where hardware examines the memory patterns and speculates the data that is likely to be filled in a cache. The i7 processor supports two forms of communication among chips: DDR3 memory controller and QuickPath Interconnect. The double data rate type three (DDR3) memory controller supports a data rate of up to 32 GB/s with a total bus width of 192 bits. QuickPath Interconnect is a link based on point-to-point electrical interconnect that enables 12.8 GB/s transfers between chips. These links are bidirectional with a total bandwidth of 25.6 GB/s with a processor speed of 2.66–3.33\(GHz\). A complete description of Intel architectures can be seen in [Intel, 2015].

AMD  AMD is another processor manufacturer, who introduced their first multi-core processor, AMD Opteron in 2005, targeting the server and workstation markets. AMD Opteron was built using Direct Connect Architecture.

---

\(^1\)Note that x86 is a 32 bit instruction set while x86_64 is a 64 bit instruction set.
This novel design eliminates the bottlenecks in traditional interconnecting front-side bus architectures. It directly connects the memory controllers to the CPU to improve overall system performance and efficiency. It exploits the AMD64 instruction set architecture with the ability to run both 32 and 64 bit instructions.

AMD Athlon II dual core processors are released for desktop computing with each core having a dedicated L1 and L2 caches. It also enables independent core access to L2 cache. A multi-point thermal control reduces the processor speed and heat when a predefined heat threshold exceeds. Later, AMD Phenom quad-core processors are manufactured with a shared L3 cache (2–6MB) and a dedicated L1 and L2 cache. The processor speed of these systems is approximately 2.5–3.0GHz. A detailed description of high performance through AMD multi-cores can be found in [AMD, 2008].

**SPARC** Sun Microsystems developed the *scalar processor architecture* that uses the reduced instruction set computing (RISC) [Patterson and Ditzel, 1980], [Patterson and Sequin, 1981], [Stallings, 1988], version 9. RISC is designed to process smaller set of computer instructions, thereby improving the speed of processor. In 2006, Fujitsu and Sun joint ventured to release a dual core processor, *SPARC64 VI*. A shared L2 (6MB) cache with a dedicated L1 cache. It uses coarse-grained multi-threading. Each core of this processor can execute two concurrent threads, one after the other, later extended to *SPARC64 VII* in 2008, which is a quad-core processor. In these systems, coarse-grained multi-threading is replaced with simultaneous multi-threading. SPARC VII has been further extended to *SPARC VII+* with each core running 8 threads simultaneously. The processor speed has been increased to 3GHz while the L2 cache is 12MB.

The architecture of the above mentioned chips are based on a conventional uniprocessor. A copy of a number of identical cores are maintained with a large cache sizes. A complete description on the design and a survey of multi-core processors that are surging in various realms of the markets from desktops, embedded systems, mobile, server segments can be found in [Blake et al., 2009].
There are a number of advantages for having multiple cores on a single chip. Foremost of which is the processing speed, as each core acts as an individual processor, the combined effect of all the processors enhances the processing speed. This multiple processors on-chip design enhances the cache-coherency so that memory transactions do not have to travel off-the-chip. Thus, more data will be processed within the same clock cycle time as that of the single core processor. Another advantage, also related to performance, is multitasking, where two different cores perform two independent tasks simultaneously. Unlike single core processors, multi-cores rarely starve for processing power. Also, fabricating two processors on a same die slightly reduces the power consumption when compared to two coupled single core processors. It is because the cores share some common hardware such as cache. This reduces the external memory transfers that requires a bit of extra electrical energy. A dual-core processor acts as a quad-core processor when multi-threading is enabled. We now discuss some of the recent advancements in multi-cores.

3.3.4 Deluge of Multi-cores

In fact, following the genesis of dual-core processors, Justin [Justin, 2005] related the growth of number of cores on a single chip of a desktop machine become ten by 2010, based on Moore’s law. He also expressed concerns on the challenges that the software community will face in exploiting the power of multi-cores. Following these lines of prediction, processor manufacturing vendors, predominantly Intel and AMD, fabricated multi-core processors. Examples of such processors include: quad-core (Intel i5, i7 and AMD Phenom II X4), hexa-core (Intel Core i7 Extreme Edition 980X and AMD Phenom II X6), octo-core (Intel Xeon E7-2820 and AMD FX-8350) and ten cores (Intel Xeon E7-2850). These processors are available for desktop computers.

Along with these advancements, processor industry has been keen on producing high performance computing machines with tens and hundreds of cores on a single chip. Of those processors Intel announced an 80 core
processor in 2007, released as the Intel Polaris [Ferguson, 2008] an 80 core processor that employs much simpler design than the classic dual and quad-cores. In 2012, a semiconductor company, PicoChip integrated 250–300 digital signal processing cores onto a single die, named PC200. IBM integrated 4096 cores in TrueNorth [Merolla et al., 2014] a brain-like highly scalable and efficient non-von Neumann architecture which mimics one million human neurons and 256 million synapses. It was proved to be suitable for the applications that use complex neural networks for multi-object detection and classification. Likewise, other processor manufacturing vendors (AMD, SPARC, etc) are investing in fabricating multi-core processors. Recently, Intel introduced Xeon Phi series of processors with a maximum of 61 cores. These multi-core processors are suitable for applications that are massively parallel. For example, a large scale data mining application requires a computational power of tens and hundreds of cores in processing a large amount of data.

Along with the advantages of multi-core systems, some challenges such as programming multi-cores remain critical. The software needs to be designed explicitly to realize the true potential of these cores, rather than relying on operating system (OS) scheduling to keep them occupied. Amarasinghe [Amarasinghe, 2008] described this deluge of multi-core menace as the third software crisis. The growing number of cores may fail to guarantee performance improvements without the extra computational power being exploited explicitly. Thereafter, research analyst, Gartner also identified this problem and suggested the software community to develop multi-core compatible applications [Barker, 2009].

### 3.4 Shared Memory Programming

This section presents the programming models to exploit the shared memory multi-core processors. It is evident that with as many advantages as

---

1 A synapse is a structure in nervous system, which permits one neuron or a nerve cell to send an electrical or chemical signal to another cell or a neuron [Tansey, 1997].
possible, apparently, there are a few concerns included with the multi-cores. Of which, the primary concern is with the software applications failure to exhibit performance gains when executed on multi-cores. Majority of these failures are because of the fact that they are programmed to run on single core machines [Dempsey, 2009] but not on multi-core processors. Like Gartner, Stephen [Stephen, 2010] rightly pointed out that applications do not automatically get faster simply with an increase in the number of cores. Programmers must employ their skill in writing software applications to exploit the computational power of multi-cores, however, this must not stretch the time in developing the software [Patterson, 2010].

Therefore, it is becoming increasingly difficult to increase the speed of a program on multi-cores. Also, given the trend in the multi-core industry it is highly possible that the desktop machines will be shipped with an increased number of processors – at least for a while. If we fail to utilize this computational power, then the existence of multiple cores is mere waste and they will effectively cease to exist. Hence, the answer is to program for the multi-cores, parallel programming. Most programs are written for conventional single core systems that fail to exploit the multi-cores. In those instances, one possibility is to run multiple instances of these programs, which often results in little gains [Biswas et al., 2009]. The alternative for this is to transform the serial programs into parallel programs.

First, we look at different existing possibilities to write parallel programs on shared memory systems. Recall that in a shared memory system, all the cores can access all the memory locations. Naturally, when programming these systems, coordinating multiple cores by specifying certain shared memory locations is non-trivial. There are a number of alternatives to program these processors. They range from processors switching between the processes, using threads [Nichols et al., 1996], using a new parallel programming language [Blumofe et al., 1995], using an existing language and compiler directives that offer parallelism [Dagum and Menon, 1998] and amending the syntax of an existing language to create parallel programming language [El-Ghazawi et al., 2003]. Along with these programming
alternatives, shared memory programming poses a number of challenges, we now discuss some of them.

### 3.4.1 Shared Memory Parallel Programming Challenges

We discuss some of the parallel programming challenges in a shared memory environment.

**Load balancing** is an important issue in parallel programming that can cause some performance implications. In load balancing, the work has to be distributed equally (approximately) among all the processors (cores/threads). For example, when all the threads are synchronized at a barrier\(^1\), the slowest thread determines the overall performance of the application. Such a situation can be avoided by evenly dividing the loop iterations or array computations among all the threads under execution. In some cases, equal distribution may not guarantee the load balancing, rather, assigning the work dynamically among threads can be a solution.

**Synchronization** In a parallel program, non-deterministic behaviour of scheduled tasks may result in changing order of execution. For example, one thread might finish reading a variable without the other thread writing it. At times, this order might have huge implications on the correctness of the program. This is a critical design consideration in writing parallel programs. It is the programmer’s job to ensure the synchronous execution of the threads. Such an execution often requires serialization of some blocks of a program. Using barriers, semaphores and locking the data objects are the ways to achieve synchronization.

**Race conditions** One of the major concerns of developing parallel programs is the possibility to occur race conditions. It occurs when multiple threads operate on a same shared resource, at least one of the operations

\(^1\) An obstacle that prevents the further movement of threads.
is an update. For example, thread one writes a value to a shared variable, which, is updated again in another thread to a different value. Thus, multiple threads race towards modifying the same variable. That means only one thread must be able to update the value at a time. The critical and atomic actions offer mutually exclusive execution of threads; the execution of one thread is independent of other threads.

**Deadlocks** is a peculiar issue in parallel programming. Deadlock occurs when there is a cycle of tasks, in which, one thread is blocked waiting for another thread to proceed for execution. In this case, all threads are blocked forever, where each thread waits for the other thread to do something. Fortunately, deadlocks are easy to detect as the executing tasks will halt at a deadlock point.

A few existing shared memory programming models efficiently addressed these challenges. Also, the scope of the thesis is defined such that it neither explores through the design of a new parallel programming language nor modifies the syntax of an existing language. Instead, we take advantage of the existing programming languages that employ compiler directives for parallelization. Henceforth, we discuss shared memory parallel programming with threads and compiler directives.

### 3.4.2 Pthreads

One of the many alternatives to program multi-cores is the use of POSIX threads or Pthreads. A thread in a shared memory programming is analogous to an instance of a program running on a processor. It is also considered as a light-weight process. It consists of a block of memory for the stack, heap, file descriptors, information about the hardware and software resources that it can access and the state of a process, whether it is ready to run or waiting for a resource. POSIX [Eißfeldt, 1997] is an application programming Interface (API) for Unix-like operating systems (OS). This library is used for multi-threaded programming in languages like C.
Figure 3.4: An example of a main thread forking two threads and joining them back into main thread after completion of their task.

Figure 3.4 shows the forking and joining mechanism of the Pthreads. Along with the advantages of Pthreads, the challenges such as load balancing, synchronization, race conditions, and deadlocks need a proper treatment. In fact, Pthreads resolve these challenges with the use of semaphores, mutexes, critical sections, locks, and barriers. A detailed description of Pthreads programming can be seen in [Nichols et al., 1996].

Pthreads is a low level programming library that makes programmer to explicitly define the behaviour of each thread. Although it is powerful to specify the thread behaviour, it demands human expertise considering the fact that programming the low-level thread interactions is often difficult.

### 3.4.3 OpenMP

Like Pthreads, OpenMP (Open multiprocessing) [OpenMP API, 2008] is another interesting shared memory programming interface (API). In 1997, a group of computer scientists (Architecture Review Board (ARB) [ARB, 2015]) accepted the OpenMP library as an informal standard for parallel processing on shared memory systems [Dongarra et al., 2003].

OpenMP is a portable, scalable, directive based specification to write parallel programs on shared memory systems. It consists of compiler directives, environment variables, and run time libraries that are used to designate parallelism in C/C++ and Fortran, whereas this thesis focuses only
Figure 3.5: OpenMP fork-join programming model.

on C/C++. These directives are special preprocessor instructions termed as pragmas. It works on the fork-join programming model [Dennis and Van Horn, 1966], which is shown in Figure 3.5. In this model, the programs start to execute sequentially within a master thread. When this thread encounters an OpenMP directive during the execution, it forks off a team of threads, all of which terminate at the end of the parallel directive. The main thread collaborates with the other members of the team, after the join stage, the main thread continues the execution.

OpenMP requires the programmer to specify a high-level specification of parallelism in a program. Thus, OpenMP provides a means to create team of threads, share work among the members of a team, declare shared and private variables and synchronize threads. The shared variables are accessible for all the threads while the private variables are permitted to the threads that own them. The general syntax of an OpenMP directive can be seen as follows:

```plaintext
#pragma omp directive_name [clause[[,] clause] ...]
{
    <statement block>
}
```

where, for example, a `directive_name` can be replaced with any one of the constructs: parallel construct (parallel for), parallel work sharing construct (parallel sections), shared (shared) and synchronization (critical) directive, etc. The number of threads allowed in a given parallel region depends on the clauses that a directive allows to control.
When the OpenMP program starts executing, it uses a single thread (*master thread*), upon reaching the *parallel* directive, a team of threads (*slave threads*) and the master thread continue executing in the parallel region. Each slave thread executes the code in parallel block to print the above message. There is an *implicit barrier* at the end of the block. When a thread reaches the end of the parallel block, it waits for the remaining threads in the team to complete their task. Then, all the slave threads terminate, handing over the execution to the master thread.

OpenMP library offers a number of features that are applicable to share work among different cores. These features are work-sharing constructs such as loop and data sharing. It also consists of environment variables and clauses that can control the manner in which the OpenMP programs are executed. Since it is not our objective to replicate the entire OpenMP specification, we briefly discuss some of these constructs and clauses that are detailed in [OpenMP API, 2008], [ARB, 2015].

**Work-sharing Constructs**

Work-sharing constructs are the most important feature of OpenMP as they distribute the computation among the threads in a team. There are two main rules for a team of threads that share work in a parallel block: i) each thread of a team must execute the work-sharing construct or not at all, ii) the sequence of work-sharing regions and barriers must be same for all the threads. Let us now look at a few work-sharing constructs.

**parallel for** is a *loop construct* that distributes the iterations of a loop among the threads. The use of a *parallel for* construct is limited to a *for* loop that has defined boundaries, that is a loop with a terminating condition.

**parallel sections** is a construct that makes different threads to perform different actions. The *sections* construct defines a parallel region, in which, each independent task is handled by *section* construct. The second construct can be an optional. Each thread executes only one *section* block of code at
once. If there are more number of threads than the independent blocks, the remaining threads will be idle, otherwise, all the threads execute multiple code blocks. Assigning code blocks to threads depend on the implementation of a program.

**parallel task** is also another work-sharing construct that works similar to **section** construct. Notice that the use of **omp task** is not optional.

**single** construct specifies a block of code to be executed by only one thread. At run time, it does not state which thread of a team to execute the block of code that it encircles. Indeed, the thread can vary from one run to the other. When this construct is used, all the threads wait at the barrier for the thread executing the **single construct** to complete its task.

**Clauses**

The OpenMP work-sharing constructs support a number of optional clauses that control the behaviour of the constructs. Usually, these clauses are used along with the work-sharing constructs. They are processed before entering into the construct that they are associated to, a few of these clauses are described as follows.

**shared** clause specifies the data to be shared among the threads associated with the parallel region. With this clause, a unique instance of a variable, which each thread can access (read/write) freely, is created. Quite importantly, with the sharing of the resources, multiple threads might simultaneously attempt to update the same memory location. It is the developers duty to avoid such anomalies while developing an application.

**private** clause maintains the thread-specific variables in a parallel region. Each thread must deal with a unique iteration of the loop so that modifications will be safe. Otherwise, the change made by one thread will be altered by another thread. With the use of a **private** clause, the changes made by
one thread are invisible to the other threads. The updates by the last thread on the private variable are thread specific, where each thread in the parallel region keeps a copy of that variable.

**lastprivate**  If we need the updated value even after the parallel region, OpenMP offers another clause, **lastprivate**. It \(\text{lastprivate}(\text{variable})\) helps to access the last thread updated value of variable outside the parallel construct. This might cause some performance implications as the OpenMP library has to keep track of the thread that executes the last iteration.

**firstprivate**  Usually, private variables are uninitialised before entering into the parallel construct. At times, there may be a case, to use the pre-initialized private variables in the parallel region. OpenMP offers **firstprivate** clause to fulfil this demand. For example, in a parallel region, if each thread is constrained not to access the first four elements of an array, an offset variable with a value of four can be made **firstprivate** so that a pre-initialised value can be carried into a parallel block of code.

All these three types of private clauses are supported by the **parallel for** and **parallel sections** work-sharing constructs.

**schedule**  is another interesting clause that is supported only on the loop construct. The **schedule** clause specifies the distribution of the loop iterations among the team of threads. It has five different types: **static**, **dynamic**, **guided**, **auto** and **runtime**. These types operate on certain number of iterations of a loop, termed as **chunk_size**, which is a positive integer.

The **static** type statically divides the loop iterations into chunks with a **chunk_size**. These chunks are assigned to threads in a round-robin style. The last chunk can have less number of iterations. If no **chunk_size** is specified, then the iterations are divided into approximately equal size. The default **chunk_size** is 1.

The **dynamic** type assigns iterations as the threads request them. A thread executes a chunk of iterations, requests another chunk and continues till none remain to schedule. Also, in this type the last chunk can have
less iterations than the chunk_size. In case, no chunk_size is mentioned, the default chunk will have 1 iteration per thread.

In guided scheduling, the iterations are assigned as the threads request them. The thread executes a chunk and requests the next, and so on till no chunks remain to work. For a chunk_size of 1 the chunk of iterations are proportional to the ratio of the number of unassigned iterations over the number of threads, decreasing to 1. For a chunk_size of “k” (k>1), the chunk iterations are determined similarly, with a condition that the iterations in a chunk will not be less than “k”, except for the last chunk. In this type also, the default chunk_size is 1. In auto, the compiler and/or the run-time system delegates the scheduling type.

Finally, runtime scheduling decides the type of scheduling and the optional chunk sizes during runtime. In this the schedule type and the chunk size need to be set with an environment variable OMP_SCHEDULE.

Synchronisation Constructs

Some shared resources require organized accesses from multiple threads. For example, a program may require multiple threads to orchestrate updates to a shared variable in a controlled order, or simply, two threads are not allowed to simultaneously write to a shared resource. OpenMP offers a few synchronization constructs, which work together with the work-sharing constructs that guarantee the controlled execution of multiple threads.

**critical** directive offers mutually exclusive execution of threads in the block of code that it encircles. Mutual exclusion serializes the thread execution, whereby at any point of time, only one thread executes in the critical section. All the remaining threads are enqueued before the critical region. Thus, the anomalous thread reads and writes are avoided. OpenMP offers named critical sections, where a program can have more than one critical section with a different name for each section. All of them are treated as one composite block, because of which, program performance deteriorates significantly.
the atomic directive enables multiple threads to update shared data, which is an efficient alternative to critical directive. The atomic directive also guarantees mutual exclusion, which is limited to small operations such as addition, subtraction, increment and decrement. With this directive, if a thread is atomically updating a value then the remaining threads will wait.

barriers is a point in a program, beyond which, no other threads progress their execution until all the threads reach that point. OpenMP offers an implicit barrier at the end of each construct (for example, parallel for and parallel sections). Thus, it is often unnecessary to have an explicit barrier. However, it purely depends on the requirements of the program.

To summarise, this section has briefly explained the OpenMP parallel programming model, different work-sharing and synchronisation constructs, and control clauses. A complete description of OpenMP API can be found in [Rohit et al., 2001], [Quinn, 2003], [Mattson et al., 2004], [Chapman et al., 2007], [Pacheco, 2011].

3.4.4 Performance Measures

Of course, an important aspiration of parallel programming is usually to extract better performance in terms of the execution time. The two widely accepted performance measures are speedup and efficiency.

Speedup In general, the parallel program should be able to show better performance over its serial counterparts in terms of the execution time. This improvement in execution time is expressed as speedup. It is formally defined as the ratio of the execution time of sequential program to that of the parallel program. It is given as shown in the following eq. 3.1

\[
\text{Speedup} = \frac{T_{\text{serial}}}{T_{\text{parallel}}}
\]

where \( T_{\text{serial}} \) and \( T_{\text{parallel}} \) stand for the execution time of a serial and parallel forms of a program respectively. Both the execution times are wall clock times that are usually influenced by a number of factors.
The execution time depends on many factors, such as the expertise of
the human developer, the compiler, use of any compiler optimizations, the
OS, type of the file system that stores the input data, number of users using
the same machine, different workloads, and network traffic, etc. However,
to have some confidence on the recorded execution times, it is necessary to
test both the sequential and parallel forms of a program in identical software
and hardware environments with similar conditions.

**Efficiency** Speedup can tell us the feasibility of parallelising a program.
The second measure *efficiency* states the amount of resources required for
parallelization. Efficiency is defined as the ratio of speedup over the number
of cores. It is given by the following eq. 3.2.

\[
\text{Efficiency} = \frac{\text{Speedup}}{n} = \frac{T_{serial}}{n \cdot T_{parallel}}
\]  

(3.2)

where \(n\) is the number of cores employed by the parallel program.

If efficiency is equal to 1 (i.e., 100%) then \(\text{Speedup} = n\), which is a
linear speedup, and typically the *ideal* case. In general, coordination among
multiple cores during parallel processing hinders to exhibit a speedup of \(n\),
thus the efficiency will not reach ideal.

### 3.4.5 Difficulties in Parallel Programming

Multiple cores offer high computational power, unfortunately, the difficulty
in programming these on chip multi-core architectures is an interesting and
a daunting challenge [Amarasinghe, 2008], [Stephen, 2010]. There are a
number of potential reasons for this. Since majority of programmers have
been trained to think and write sequential programs for decades, program-
meg for single core processors is instinctive.

Particularly, when we consider the roadblocks that a human program-
mer encounters in developing the parallel code: data restructuring, race and
deadlock occurrences, synchronizing the communication among threads,
dealing with thread specific data, scheduling the workload among cores,
debugging parallel programs and finally, attaining behavioural equivalence of both sequential and parallel programs, etc. Manually fixing these challenges, which in itself is a daunting task.

However, we can think of two different alternatives in developing the parallel programs: i) to write a sequential program then parallelize it; ii) implement a parallel program from scratch. The former approach, at times, may become infeasible (or impossible) to detect the parallelism by hand. In the latter approach, thinking parallel to develop an application from scratch might lead to incorrect programs. In any case, both the approaches are tedious and requires pragmatic human skill.

Hence, a number of automatic approaches were presented in literature to address many of these challenges in both the alternatives. We now discuss these alternatives for automatic parallel code generation.

### 3.5 Automatic Parallel Code Generation

This section describes the existing research in automatic generation of parallel programs. In general, automatic generation of parallel programs can be divided into two types: auto-parallelization of serial code and the generation of native parallel code. Figure 3.6 shows the classification.

![Figure 3.6: Classification of automatic parallel code generation](image-url)
Many conventional techniques in the literature tried for automatic parallel code generation. They range from fully automated [Bondhugula et al., 2008a], [Bondhugula et al., 2008b], semi-automated [Zima et al., 1988], [Vandierendonck et al., 2010] to parallel programming languages such as Cilk [Blumofe et al., 1995], Scala [Odersky et al., 2008], and Go [Baugh, 2010], etc. We now discuss some of these attempts that fall into both the categories, especially, those who experimented on multi-core architectures.

3.5.1 Automatic parallelization

Auto-parallelization is the re-engineering process that transcribes the serial code to execute on a parallel architecture by maintaining its authenticity. That is, the auto-parallelization approaches rely on the existence of a working program that they modify. Auto-parallelization research was began with auto-vectorization [Kuck et al., 1980], [Kuck et al., 1981] of loops on vector processors. From then onwards, automatic parallelization tools for MIMD architectures with shared memory systems [Allen et al., 1988], [Wolfe, 1990], [Wolf and Lam, 1991] were designed.

Similarly, novel compiler parallelization techniques were proposed, some of them were surveyed in [Trew and Wilson, 1991]. Many of these techniques work on extracting parallelism in small applications. Later, tools such as Polaris [Blume et al., 1994], Vienna Fortran Compiler [Benkner, 1999] focused on extracting parallelism in large scale applications. That is automatic parallelization of large scale applications that are written in Fortran and C. Then, Amarasinghe et al., [Amarasinghe et al., 1995] extended the compilers for automatic parallelization on scalable parallel machines. Also, a high-level representation in StreamIt compiler [Thies et al., 2002] improved the program performance on multi-cores for applications in streaming domain.

All these attempts realised the need for automated techniques for the task of parallelization. In realising that, it is necessary to follow a standard formalism for automatic parallelization. Initially, Kennedy et al., [Kennedy
et al., 2003] described the steps involved in auto-parallelization process, which comprised of four phases. Figure 3.7 represents them.

<table>
<thead>
<tr>
<th>Identify</th>
<th>Facilitation</th>
<th>Transformation</th>
<th>Verification</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parse, Analyse, Dependency, Profiling</td>
<td>Concurrent block scheduling</td>
<td>Code optimizations, Transforms</td>
<td>Program debugging, Verification</td>
</tr>
</tbody>
</table>

**Figure 3.7:** A high level overview of auto-parallelization process.

**Identify**  The sequential code is analysed in order to configure the concurrently executable parts. The analysis is often performed by parsing through the code for dependency checks and sometimes through profiling. Manual inspection is also possible but, for small programs. It involves greater degree of difficulty as in most of the cases the programmers that parallelize the code are unfamiliar with the original code and/or the application domain. This type of finer details makes it increasingly difficult even for an experienced programmer.

**Facilitation**  In this phase, the identified concurrent code blocks need to be executed on multiple processors/cores (in our context, it is a multi-core CPU). It can be performed in many ways depending upon the architecture of the experimental machine, the shared/distributed memory configuration. They are system level pthreads, a compiler, externally supported APIs such as OpenMP, MPI, etc. In this thesis we use OpenMP that has been explained in section 3.4.3.

**Transformations**  The parallel code produced in the previous phase can be suboptimal. Hence, the code is further analysed to remove the dependencies, if any, paying the way forward for an increased parallelism. Example transformations: loop interchange (changing the order of two different loops to obtain increased parallelism), loop distribution (separating the sequential parts that can not be parallelized and can be parallelized into two different
loops), loop fusion (merge independent tasks of different loops into a single loop), loop skewing (in a nested loop, rearranging the inner loop iterations so that the dependencies are in the iterations of outer loop) and statement reordering (swapping the statements such as if with the other statements of a program).

**Verification** The final phase tests the functional equivalence of the parallel code with that of the serial code and, the parallel debugging also happens in this phase.

Almost all the auto-parallelization tools adapt this four phase formalism in one way or the other. We discuss some important attempts that contributed to that task of automatic parallelization of source code on multi-core machines.

With the advent of multi-core processors, automatic parallelization has re-emerged as an important tool. In this direction, Franchetti et al., [Franchetti et al., 2006] derived Fast Fourier Transform algorithms [Frigo and Johnson, 2005] on multi-core platforms. These algorithms offered efficient load-balancing and loop optimizations. Again, researchers devoted to extend Polaris. In that, a sensitivity analysis on the behaviour input for a program [Rus et al., 2007] achieved the dynamic parallelization of loops on multi-cores.

Datta et al., [Datta et al., 2008] optimized the stencil computations on a range of multi-core architectures from different vendors, some of which have been discussed in section 3.3.3. Their approach derived a number of efficient optimization approaches, in which, an auto-tuning method searched for minimizing the runtime while maximizing the performance.

Another automatic parallelization tool, Pluto [Bondhugula et al., 2008c] that works on polyhedral model [Bondhugula et al., 2008b] for compiler optimization, which offers high-level abstractions to perform loop transformations. Pluto offers coarse-grained parallelism in source-to-source transformation of C programs into OpenMP programs that run on multi-cores. More recently, this framework has been further extended as Pluto+[Acharya and Bondhugula, 2015] that offer an efficient loop restructuring through the inclusion of certain useful transformations.
Similar to Polaris, Cetus [Dave et al., 2009] provides an infrastructure for compiler optimizations on multi-cores that highlight the automatic parallelization. Unlike Fortran transformations in Polaris, Cetus supports the source-to-source transformations in C programming language.

Refactoring [Dig, 2011] has been considered as a tool to extract efficient parallelism on multi-cores. Refactoring is a controlled process of applying small behaviour preserving transformations to observe significant effect in the performance of the end programs [Wloka et al., 2009]. It improved the program performance, portability and developer productivity.

Recently, a lock-free buffering scheme for kernel tracing [Desnoyers and Dagenais, 2012] on multi-cores improved the performance using local atomic operations that operate on local CPU variables. Tordini et al., [Tordini et al., 2012] proposed non-blocking high-level programming patterns that help programmers omit the low-level details about synchronization and memory management. Al Bahra [Al Bahra, 2013] explored different alternatives for lock-based synchronization to improve the scalability of non-blocking algorithms on multi-cores. Dechev et al., [Dechev et al., 2013] introduced a novel lockless containers/data concurrency library to design non-blocking algorithms on multi-cores.

All these techniques try to automate the sequential code to parallel. Thus, in the next section we discuss the creation of native parallel code.

### 3.5.2 Native Parallel Code Generation

Contrast to the above attempts, *natively parallel code generation* solves two problems together: generate a working program which is also parallel. In 1995, Ian Foster introduced Partitioning, Communication, Agglomeration, and Mapping (PCAM) [Foster, 1995], a four step process to design parallel programs. This methodology is still relevant for multi-cores.

**Partitioning** The first step is to divide the computation into smaller pieces. This division emphasizes any existing parallelism in a program. This division can be either of the two forms, *functional decomposition* and *data
decomposition. Functional decomposition is separating different steps in a program, whereas the data decomposition splits the data among the processing elements. The general rule of thumb is that the number of pieces in the division must be one or two times bigger than the number of cores.

**Communication** Ideally, the resultant individual tasks of the partitioning should be independent. Nonetheless, these tasks may have interdependencies. For one task to start executing, it may have to wait for the completion of the other task. These interdependencies may also arise in the form of data. Communication step decides the amount of data to be distributed among different tasks. It results in a task dependency graph, in which, tasks are the vertices and the edges are the amount of communicating data.

**Agglomeration** Communication obstructs the parallel processing and the solution is to group the tasks. Each group is assigned to a core, thereby the communication within the group is eliminated. A rule of thumb at this stage is to have more number of groups than the number of cores.

**Mapping** At this stage the task groups need to be assigned to the available cores. The objective is that all the cores must have approximately equal amount of work to perform. Again, the communication overhead needs to be reduced further by mapping the groups that often exchange data to the same cores. Devising these steps is always program dependent.

These four steps are a common practice in developing a parallel application. However, this branch of the automatic parallel code generation has a limited repertoire of attempts in the literature.

One of the few implementations is to automatically generate parallel code for computational fluid dynamics [Ierotheou et al., 1996] problems. This focused on generating parallel programs on distributed systems in Fortran. Later, ParaWise [Johnson et al., 2005] was introduced to generate more sophisticated code with an efficient and scalable performance. It works by taking information about the application code and performs code
alterations, giving accurate information about the application code at each step of the parallelization process described in [Foster, 1995].

Another approach, automatic generation of SIMD programs for an arbitrary architecture [McAllister and Ramirez, 2013], a US Patent. This approach defined a parallel architecture independent annotation standard that adopted a high-level abstraction for SIMD operations. It avoided the programmer from thinking to program to the hardware, while guaranteeing the portability in a parallel SIMD environment.

An interesting native parallel code generating tool, Paralldroid [Acosta and Almeida, 2014b] enables parallelism in android. It tries to abstract programming difficulties of a developer by adapting OpenMP directives. It generates parallel programs in Java, C, OpenCL [Stone et al., 2010] and RenderScript [Guihot, 2012]. This way the automatically generated programs are further scrutinized in [Acosta and Almeida, 2014a] for efficiency. This framework efficiently exploited the computational capabilities of multi-cores in mobile platforms.

We discussed the related work both in auto-parallelization and native parallel code generation. In the context of this thesis, it is important to discuss the existing attempts in evolutionary domain also.

3.6 Evolutionary Automatic Parallel Code Generation

Most of these traditional approaches are fairly complex and time consuming while they also demand an enormous amount of domain knowledge. On the other hand, evolutionary techniques lead their way into the automatic parallel code generation domain with their simplicity. Besides the ease of use, they demand casual background knowledge, yet times, no background information and no risk of dependency analysis. This section describes the state-of-the-art evolutionary attempts in both the categories of automatic parallel code generation.
3.6.1 Evolutionary Automatic parallelization

Although GA, ES and EP were discovered before GP, it was first applied in mid 1990s for automatic parallelization. Later, GA and ES were used in this regard. Overall, only a few evolutionary researchers employed EAs for automatic parallelization. Figure 3.8 shows the existing evolutionary automatic parallelization approaches.

Figure 3.8: A survey of evolutionary automatic parallelization techniques.

Evolutionary auto-parallelization of serial code was initiated by Walsh and Ryan, PARAGEN-I [Walsh and Ryan, 1995], [Walsh and Ryan, 1996], to produce parallel code without any dependency analysis. This system disassembles individual instructions of a program and automatically maps them onto parallel hardware using GP. Although, PARAGEN-I succeeded in transforming sequential loops, handling complex loops with inter-dependencies remained a major challenge; nested loops are an example of such a scenario. These challenges were dealt in PARAGEN-II [Ryan and Walsh, 1997a], [Ryan and Walsh, 1997b], that evolved a sequence of transformations. Each transformation resulted in some modification of a program, which was passed
to the subsequent transformation. They were applied to first analyse serial-
dependency of instructions; this analysis was performed during the fitness
evaluation. Finally, the transformations produced a functionally equivalent
parallel program. Then, this approach was extended to automatically par-
allelize loops on Beowulf cluster of machines [Walsh and Ryan, 1998] that
use distributed memory model. Another attempt [Ryan and Ivan, 1999] ex-
tended PARAGEN-II that operates in two modes: Atom and Loop, where
auto-parallelization of individual instructions and loops were melded into a
single coherent operation. Atom mode parallelized the independent instruc-
tions, upon encountering the loops, Loop mode parallelized the sequential
loops. Also, it performed loop fusion. A detailed description of all the auto-
parallelization attempts using GP are in [Ryan, 1999].

Similar approaches for GAs were presented. Nisbet [Nisbet, 1998] pre-
sented the Genetic Algorithm Parallelization System (GAPS) that selected
the optimal transformations. GAPS considered the selection of an optimal
transformation (recall the list of transformations discussed in section 3.5.1)
as an NP-complete problem. GA was used to search for a better sequence
of transformations to minimize the execution time of Fortran programs that
run in an SIMD style.

Later, Williams [Williams, 1998] presented a genetic compiler, Reading
Evolutionary Restructurer (REVOLVER), for automatic parallelization
of sequential Fortran programs using GA. REVOLVER used two represen-
tations gene-transformation, gene-statement [Williams and Williams, 1999]
in generating parallel programs. In gene-transformation representation, each
gene of an individual corresponds to a transformation. Thus an individual
represents a sequence of transformations to be applied on a sequential prob-
lem. This representation allowed all the genetic operations. In the gene-
statement representation, each gene represents a statement (if, do, etc.) in
a program. Only mutation was allowed for a better statement restructuring.
At the end, higher quality parallel programs were evolved.
3.6.2 Evolutionary Native Parallel Code Generation

GP has been exclusively applied for automatic generation of natively parallel code. For example, Poli [Poli, 1997] introduced *Parallel Distributed Genetic Programming* (PDGP) for the development of natively parallel program with a high degree of parallelism. The resultant programs were graphs with each node having the functions and terminals, whereas the links were meant for the flow of control and results. In fact, the evolved programs were simulated in a virtually parallel environment with an assumption of their adaptation to the real computer architectures.

Trenaman [Trenaman, 1999] introduced *Concurrent GP* (CGP) that used multi-tree GP representation, where a number of GP trees were evaluated simultaneously. CGP was applied to automatically design controllers for autonomous agents in Tartarus world [Teller, 1994]. CGP produced low complex and highly fit agents in Tartarus, which exhibited an interleaved evaluation of program trees.

Ross [Ross, 1999] used GP to automatically evolve concurrent systems, where process algebra [Hennessy, 1988] was the target language. The findings of this attempt were that GP was capable of evolving code for concurrent systems, particularly when controlling the non-deterministic behaviour of large computational spaces in process algebra.

Jackson [Jackson, 2004] presented the automatic evolution of concurrent programs using GP for “dining philosophers” problem. In this, the evolved programs were executed in a pseudo parallel style for a pre-defined number of philosophers. The concurrent execution of programs minimized the cost of fitness evaluation. The quality of the resultant programs is approximately equal to that of the human developed programs.

In general, relatively little contribution is there in evolving parallel programs. Most of the attempts focused on the concurrent program generation with no preference of exploiting the multi-core architectures. This in itself points out to the need for more innovative solutions to automatically program the multi-cores. In this context, the attempts in this thesis can be accommodated in the branch of evolutionary native parallel code generation.
3.7 Summary

To summarize, this chapter described the initial von Neumann architecture. The modifications to this basic architecture were discussed that contributed in improving the performance. We discussed the Flynn’s taxonomy of parallel hardware, explained the architecture of the shared memory multi-core systems and their growth in the processor industry. Thereafter, highlighted the shared memory programming considerations and different human programming alternatives. We presented the difficulties of developers in manually programming the multi-core processors. We then, discussed the conventional state-of-the-art automatic parallel code generation approaches besides the evolutionary attempts.
Part II

Grammatical Evolution and Parallel Program Synthesis
Chapter 4

Complexity and Predictability of Grammatical Evolution

“Not everything that can be counted counts, and not everything that counts can be counted.”
— Albert Einstein, Physicist

The purpose of this chapter is to explore GE to gain insights into its operation. In particular, we analyze the computational complexity of GE, which is detailed in section 4.3. Computational complexity analysis of an EA can provide crucial information about how it works. While relatively straightforward for fixed length structures, it is less so for variable length structures, although initial work has already been conducted on tree based GP [Neumann et al., 2011], [Urli et al., 2012] algorithms. GE is a variable length string based EA that evolves arbitrarily complex programs through complex gene interactions, but thus far, no such analysis has been conducted. As a result of this, several observations are made estimating the terminating generation, actual length and effective lengths of individuals based on the quality of the solution. The quality of the evolved solutions of an EA varies across different runs and a significant percentage of those runs can produce solutions of undesirable quality. These runs are a waste of computational
resources, particularly in difficult problems where practitioners have time bound limitations in repeating runs. The complexity analysis provides a way to produce a reasonably good prediction system of how a particular run will perform, and we provide details of how one can predict the success or otherwise of a GE run in the early generations with the amount of data collected. The prediction system is detailed in section 4.6.

4.1 Introduction

Time complexity analysis is crucial for any search algorithm, as it dictates the general applicability and cost of the algorithm. The more predictable an algorithm is in terms of time cost, the more useful it is. Execution time has long been a challenge to the field of evolutionary computation (EC) [Tanese, 1989], [Openshaw and Turton, 1994], [Andre and Koza, 1996], [Cantú-Paz and Goldberg, 1997], [Luke, 2001], [Cantú-Paz and Goldberg, 2003]. In particular, the stochastic and iterative nature of the underlying algorithms often arouse suspicion in those more accustomed to dealing with deterministic algorithms. However, if one could prove that EC methods can provide competitive solutions in less time than their deterministic counterparts, then their acceptance into more general Computer Science and Engineering fields would be accelerated.

Although there has been much computational complexity analysis on general EAs [Chen et al., 2010], [Oliveto et al., 2009], [Chen et al., 2009], no such analysis has been conducted on GE, with most of them concentrating on the performance impact of search operators such as crossover [O’Neill et al., 2003]. For example, Durrett et al., [Durrett et al., 2011] analysed GP with an investigation on the impact of neutral moves and the importance of a local mutation operator. A simple analysis on mutation in GE [Byrne et al., 2010] produced interesting results, but without any computational complexity analysis. Although there has been some work in analyzing the time complexity of GP [Neumann et al., 2011], [Urli et al., 2012], GE varies from GP with the introduction of grammars and mapping. We discuss the existing
attempts in analysing the computational complexity of various evolutionary techniques.

4.2 Related Work

Initially, Beyer et al., [Beyer et al., 2002] analyzed EAs to predict the time complexity and showed that it basically depends upon the time to complete the execution. A general approach in [Chen et al., 2009] analysed the average time complexity of evolutionary algorithms on uni-modal problems by generalizing the concept of takeover time to EAs with mutation. This showed that EAs with bitwise mutation and elitist selection strategies need $O(n \ln n + n^2/N)$ and $O(n \ln \ln n + n \ln n/N)$ generations respectively to find the global optimum. Works such as [Oliveto et al., 2009], [Chen et al., 2010] theoretically calculated the computational complexity of evolutionary algorithms when applied on combinatorial optimization problems. They used mathematical models and analysis tools such as Markov chains [Rudolph, 1998] and Martingale models with Stochastic Lyapunov function [He and Yao, 2004] to analyse the evolutionary algorithms. Markov chain models summarized the results on the limit and finite time behaviour of EAs with finite search spaces and discrete time scale, while Martingale models estimated upper and lower bounds of the mean hitting times of EAs to classify easy and hard problems.

The EC literature contains a few examples of measuring and accounting for the execution times of EAs. For example, Mutoh et al., [Mutoh et al., 2003] reduced the execution time of GAs in real world applications using fitness prediction. In GP, the run time is the product of the number of runs, the number of generations, the size of the population, the average size of the programs and the number of fitness cases [Poli et al., 2008]. Initial computational complexity results of GP [Neumann et al., 2011] showed how design choices of algorithm components impact its success, while computational complexity of GP on multi-objective models [Urli et al., 2012] observed maximum tree size and population size.
Given that GE has an extra step to GP, in the form of its mapping, it is not clear that these results are directly transferable to GE, partly due to the extra step, but also, probably more importantly, the complex relationship between genome length and the time required for mapping [Ryan et al., 2003]. That means a GE individual will not necessarily take longer to map.

Our contribution is to estimate the time complexity of GE in terms of its population size. We empirically study the time complexity of GE using simple algorithm analysis approaches such as power law [Stevens, 1957] and doubling hypothesis [Sedgewick and Wayne, 2011].

### 4.3 Computational Complexity of GE

We take a simple approach to analyse the time complexity of GE using doubling hypothesis and a power law, a traditional way of empirically analysing algorithms. We take the \( \log \) value graphs of execution time to analyse the time complexity of GE. Poli et al., [Poli et al., 2010] emphasized the need for computational complexity techniques in GP to better understand it. This analysis is an attempt towards that direction, believing the fact that GE also needs such an analysis. Based on this analysis we can produce new methods of prediction systems to optimize the performance of GE execution. Before discussing further into the analysis, we describe power law and doubling hypothesis.

#### 4.3.1 Power Law

A power law is commonly used approach to estimate the time complexity of algorithms. It is a mathematical relationship between two quantities; if the frequency of a quantity varies as power of the other quantity then it is said to obey a power law [Mitzenmacher, 2005]. The equation of a power law can be written as follows:

\[
T(N) = aN^b
\]  

(4.1)
where, the exponent $b$ typically depends on the algorithm, while $N$ is the input size, in our experiments, it is the population size; the leading constant $a$ depends on the system (i.e. hardware), the algorithm, and the input data. $T(N)$ is the execution time of GE at a given input of size $N$.

Since eq. 4.1 is non-linear, applying log on both sides of the equation converts it into linear form, which is shown below:

$$\log(T(N)) = b \cdot \log(N) + \log(a)$$

Plotting a log-log graph for the given input produces a straight line, and the slope of the straight line drawn can be determined with the following formula (eq. 4.3):

$$b = \frac{\log(T(N_2)) - \log(T(N_1))}{\log(N_2) - \log(N_1)}$$

where, $N_1, N_2$ are different values of $N$, population size.

In analyzing the computational complexity of the algorithms, another frequently used hypothesis is the doubling hypothesis, a way of measuring the time ratio as the input size gets doubled.

### 4.3.2 Doubling Hypothesis

A simple way to develop the doubling hypothesis is to double the input size and observe the effect on execution time [Sedgewick and Wayne, 2011]. If a function $T(N)$ obeys power law then:

$$\frac{T(2N)}{T(N)} \sim 2^z$$

eq. 4.4 is the doubling ratio, where $z$ indicates the exact relationship, order of growth of the algorithm. For example, if $z$ is 1, then time increases linearly, but if it is larger than 1, then the increase in time will be more than double. Thus, a simple and a quick way to calculate the doubling ratio is to find out the ratio of the execution times. For example, if the algorithm is
run \( n \) times doubling the input size for each time, then the doubling ratio is given as follows:

\[
doubling\ ratio = \frac{\text{Time taken for } n^{th} \text{ run}}{\text{Time taken for } (n-1)^{th} \text{ run}}
\]  (4.5)

Then, \( \log \) of the doubling ratio gives the order of growth of the algorithm. For example, if the doubling ratio is 8 then the order of growth of the algorithm would be 3, which is \( \log_2 8 \).

### 4.3.3 Empirical Analysis

An empirical analysis is performed by running the program for various input sizes. The output, in our case is time, can then be used to help determine the constants \( a \) and \( b \). Usually, such empirical analysis is carried out in the following manner: an initial hypothesis is made which assumes that the run time obeys the power law shown in eq. 4.1. Next, the data obtained by plotting a graph between run time versus input size is examined on a log-log scale. Consequently, the power law yields a straight line with some slope \( b \). Finally, the slope of the line decides the growth of the running time of an algorithm. For example, if the slope is 3 then, the run time grows as the \textit{cube} of the input size: \( aN^3 \). We determine the time complexity of GE with the help of the doubling hypothesis and empirical analysis.

### 4.4 Experiments

This section shows how time complexity varies with respect to the population size on two GE archetypal problems: Santa Fe Ant Trail and Symbolic Regression. The symbolic regression problem is binomial-3 [Daida et al., 2001] \( (1+x)^3 \). These two archetypal problems, although relatively simple and well known, are useful because they are representative of the sort of problems that GE can be used for, that is, control and classification.
4.4.1 Experimental Setup

The experiments in this chapter are conducted using libGE [Nicolau and Slattery, 2006], an open source framework for Grammatical Evolution in C++. We use the standard GE parameters for these experiments. It might be possible for a better set of parameter to solve the problems, however, we are concerned about the execution time rather solving the problem. The following are the parameters: crossover (0.9 probability), mutation (0.01 probability per bit), and a random initialiser was used to create an initial population with minimum and maximum genotype depth of 15 and 25 respectively. A steady state replacement strategy [Ryan and O’Neill, 1998] is used to generate next generation of individuals. Since the length of GE genotypes vary due to wrapping [Ryan et al., 2003], the experiments are repeated with and without wrapping. The behaviour of wrapping differs between Santa Fe Ant Trail and symbolic regression problems as explained in [Byrne et al., 2010] (wrapping was detailed in Chapter 2). The fitness of GE phenotypes is calculated with an evaluator, the exact form of which can be chosen at run time. We varied the evaluators, using Tiny C (TCC) [Bellar, 2004] for the Santa Fe Ant Trail problem and S-Lang [Davis, 2013] for the binomial-3 problem.

The experiment exclusively focused on analysing the time complexity of GE on different problems. Thus, the population sizes change on both the problems. Initially, we kept the population size (popsize) at 25 and executed GE for 1000 runs, and recorded the execution time for all 1000 runs. Next, we applied the doubling hypothesis on the population size, whereby popsize became 50 and executed GE for 1000 runs, and recorded the execution time. This process is repeated up to a population size of 400. That is the population sizes that are used in the experiment are: {25, 50, 100, 200, 400}. A run is terminated based on the convergence of the population in order to consider the quality of the solution; that is there was no explicit limit on the number of generations in a given run. This way we recorded the execution times of GE of Santa Fe Ant Trail, and binomial-3 problems, both in the presence and in the absence of the wrapping operator.
4.4.2 Experimental Results

This section investigates the time complexity of GE using doubling hypothesis and empirical analysis. We analyse how the execution time changes with respect to doubling the population size. Figure 4.1 shows the execution times of Santa Fe Ant Trail and binomial-3 problems on a log-log scale. The graphs are plotted with log of the population size (25, 50, 100, 200, 400) on x-axis and the log of the total execution time to finish 1000 runs at each population size on y-axis. These graphs help to empirically analyse the computational complexity as explained in section 4.3.3.

![Figure 4.1](image)

**Figure 4.1:** The time complexity graphs of Santa Fe Ant Trail (shown left) and Binomial-3 (shown right) problems both in the presence and in the absence of wrapping on a log-log scale.

We assume that the execution time of GE follows power law shown in eq. 4.1, and empirically record the change in execution time with respect to doubling the input size, i.e., the population size. We repeated this process on both the problems. The first step is to measure the total time spent by the algorithm to complete 1000 runs at each population size, and the second step is to calculate the time complexity by plotting the data on a log-log scale.

Figure 4.1 shows the computational complexity graphs of both the experimental problems. These plots are almost straight lines, which satisfy the
last step of the empirical analysis described in section 4.3.3, hence, these lines are in the form of eq. 4.2.

Next, we calculate the slope \((b)\) of the straight line using eq. 4.3 for any two points on the straight lines shown in Figure 4.1 for both the problems. When wrapping is disabled the slope of the Santa Fe Ant Trail is 1.04. With these mathematical calculations, the power law equation becomes:

\[
T(N) = a.N^{1.04}
\]

On Santa Fe Ant Trail, for the remaining straight line, when wrapping is enabled, the slope is 1.06 and makes the power law equation as follows:

\[
T(N) = a.N^{1.06}
\]

The slopes both in the presence and in the absence of wrapping converge to 1, meaning that the exponent is approximately equal to 1. Since, the leading constant \(a\) is a multiplication factor, that can be neglected. Hence, the final equation can be rewritten as

\[
T(N) \sim N
\]

This shows that the time complexity of GE on Santa Fe Ant Trail irrespective of wrapping is \(O(n)\) in terms of population size.

Similarly, for binomial-3 problem, the slopes in the absence and in the presence of wrapping are 2.08 and 2.26 respectively. These values produce the following two mathematical equations.

\[
T(N) = a.N^{2.08}
\]

\[
T(N) = a.N^{2.26}
\]

Here, the slopes are leaning towards 2, as the leading constant can be neglected, the power law (eq. 4.1) is rewritten as

\[
T(N) \sim N^2
\]

This shows that the time complexity of GE on binomial-3 problem is \(O(n^2)\).
Thus, the time complexity of GE on Santa Fe Ant Trail is stated as $O(n)$, while that on symbolic regression problem is $O(n^2)$, in terms of population size. Hence, this analysis shows that, doubling the population size linearly increases the total execution time of GE on the Santa Fe Ant Trail, and it increases in the power of 2 for binomial-3, while wrapping has negligible effect on GE time complexity.

In this analysis we observe the success rate, the number of generations required to solve a problem and the length of GE genotypes on both the problems. The following section discusses these observations.

4.5 Discussion

This section analyses two different aspects of the runs; the frequency of runs that belong to a particular success interval at a given population size and the average of certain useful GE characteristics. For this analysis, we determine nine different cumulative probability of successes (21%, 31%, 41%, 51%, 61%, 71%, 81%, 99% and 100%) for each population size. For example, consider a Santa Fe Ant Trail problem that contains 89 trails in a grid, in which, 81% success is equal to 72.09 trails (89*81/100); that is the ant covers 72 of the total available (89) trails.

4.5.1 Frequency of Success

We estimate the number of runs that belong to a particular success interval for a given population size. Table 4.1 presents the number of runs that fall into a specified interval of success out of 1000 runs at each population size for both instances of wrapping. The results show that GE fails to generate a qualitative (81 – 100% success) solution for a problem when the population size (25) is small. It is because of the convergence of the algorithm to a suboptimal solution with the lack of enough population of individuals. Nonetheless, the number of runs that generate a qualitative solution are increasing with respect to the population size. For example, at a population
Table 4.1: The frequency of runs with various success intervals both in the presence and absence of wrapping on Santa Fe Ant Trail and binomial-3 problems.

<table>
<thead>
<tr>
<th>Problem</th>
<th>% of Success</th>
<th>Count at Pop Size (Wrap OFF)</th>
<th>Count at Pop Size (Wrap ON)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>25</td>
<td>50</td>
</tr>
<tr>
<td>Santa Fe Ant Trail</td>
<td>0 – 21%</td>
<td>865</td>
<td>395</td>
</tr>
<tr>
<td></td>
<td>21 – 51%</td>
<td>83</td>
<td>219</td>
</tr>
<tr>
<td></td>
<td>51 – 81%</td>
<td>52</td>
<td>275</td>
</tr>
<tr>
<td></td>
<td>81 – 100%</td>
<td>–</td>
<td>111</td>
</tr>
<tr>
<td>Binomial-3</td>
<td>0 – 21%</td>
<td>904</td>
<td>736</td>
</tr>
<tr>
<td></td>
<td>21 – 51%</td>
<td>63</td>
<td>104</td>
</tr>
<tr>
<td></td>
<td>51 – 81%</td>
<td>33</td>
<td>153</td>
</tr>
<tr>
<td></td>
<td>81 – 100%</td>
<td>–</td>
<td>7</td>
</tr>
</tbody>
</table>
size of 400, Santa Fe Ant Trail without wrapping has 958 qualitative runs; that is a success percentage of 95.8%. On the other hand with the presence of wrapping, it is 99.5%; that is, almost all the runs generate a qualitative solution. Similarly, binomial-3 with and without wrapping exhibits a success percentage of 17.9% and 22.1% respectively.

One interesting finding is that we are able to solve (exactly 100% solution) the Santa Fe Ant Trail problem with a probability of 0.924 in the absence of wrapping and 0.977 probability in the presence of wrapping, which are counted against 1000 runs for a population size is 400.

Similarly, on binomial-3 for both with and without wrapping, the probabilities are 0.009 and 0.015 respectively. That is almost a negligible impact of wrapping on a symbolic regression problem. This again shows that wrapping has no/negligible effect on the performance of GE in the case of symbolic regression, which is explained in [O’Neill and Ryan, 2003].

### 4.5.2 Impact of GE Characteristics

We now assess three different characteristics of GE: i) **Average Terminating Generation (ATG)**, ii) **Average Actual Length (AAL)**, iii) **Average Effective Length (AEL)**. ATG is the generation at which a run terminates, AAL the total length of GE genotypes and AEL the effective length the genotypes; all of these are averaged over 1000 runs. A detailed description of these characteristics can be seen in Chapter 2. Since quality of the solution is important, these three measures are reported for the above mentioned nine cumulative probability of success rates.

Figure 4.2 and Figure 4.3 represent the ATG, AAL, AEL of the two GE problems at different cumulative probability of successes, with and without wrapping. These graphs are averaged across 1000 runs at a respective population size. On both the problems, we clearly observe that the curves are clustered together in patterns irrespective of wrapping.

On Santa Fe Ant Trail, shown in Figure 4.2, ATG, AAL, and AEL converge towards a particular value in both the cases of wrapping as population size increases. In the presence of wrapping, although ATG curves terminate
Figure 4.2: Average Terminating Generation (ATG), Average Actual Length (AAL), and Average Effective Length (AEL) of SantaFeAntTrail with wrapping OFF (left), wrapping ON (right) at various cumulative probability of successes across different population sizes.
Figure 4.3: Average Terminating Generation (ATG), Average Actual Length (AAL), and Average Effective Length (AEL) of Binomial-3 with wrapping OFF (left), wrapping ON (right) at various cumulative probability of successes across different population sizes.
at approximately the same generation for a population size of 400, the frequency of the runs changes with respect to the success intervals as shown in Table 4.1. The AAL curves of 100% success are smaller than that of 99%, 81% and 71% for up to the population size of 200, and from then onwards, they converge to a common point, approximately. In the presence of wrapping, the AAL curves at 99%, 81%, 71% and 61% are bigger than that of 100% curves, which again shows that the entire genome is not used. Similar kind of observations can be seen in case of AEL graphs.

On binomial-3, the patterns suggest that ATG, AAL, AEL curves are clustered together in groups at different regions. As shown in Figure 4.3, ATG has three clusters: the curves with cumulative success rate of 100%, 99%, 81% converge to one common point. The curves with the success rate of 61%, 51%, 41% and 21% are converging to another common point, while the curve with 71% success rate alone is converging to a different point, which lies between both the above points. We observe a similar phenomena even in the presence of wrapping. For both the problems, the AEL curves are much smaller than the AAL curves. These observations suggest that in any case, the entire genome is not always used in transforming a genotype to a phenotype. However, this information can be useful in estimating the quality of a GE run.

4.5.3 Is It Possible to predict the quality of a run?

The clustered patterns strongly suggest that the manner in which these graphs increase early on during the runs could have an impact on the success of a GE run. That is, there may be runs that do not have considerable useful contribution, which can be terminated early instead of waiting for the run to complete. These useless runs can be determined with a predictor that can be generated with the help of a simple classifier system.

We gathered a large amount of data from the results of these experiments. We can use this data as an input to a classifier to produce a prediction system. The input data contains the initial changes in best fitness, average fitness, actual length and effective length of a GE run in order to determine...
its usefulness. This enables us to come up with a reasonably good predictor optimizing the performance of GE and extend that to other problems; this is described in the following section.

In summary, we addressed the time complexity of GE on two archetypal problems, Santa Fe Ant Trail and symbolic regression. The time complexity of GE grows linearly on Santa Fe Ant Trail and quadratic in nature on symbolic regression. A few other interesting observations such as ATG, AAL and AEL of a GE genome depict patterns that are useful for the prediction of success or otherwise of a GE run.

4.6 Grammatical Evolution Run Prediction

Experiments in EC follow a common practice of running the algorithm multiple times in order to obtain statistically significant results, particularly when the likelihood of achieving a solution of a certain quality is only estimated through a large number of runs. However, as problems get increasingly computationally expensive, the computational resources are best utilised if the EA consistently generates solutions of desirable quality and the runs producing poor solutions are minimised. For example, [Ryan et al., 2005], [O’Reilly et al., 2013], [Wilson et al., 2013] are all more concerned with producing a single useful result rather making statistical affirmations about their ability to emulate the same result.

This pilot study is a mechanism to better utilise resources for EC by predicting which runs are most likely to fail and discontinuing them early on during the evolution. Our technique will work with essentially all other techniques for improving GE or GP, such as Linear Scaling [Keijzer, 2004] and No Same Mates (NSM) [Gustafson et al., 2005]. The induced prediction model in this study estimates the end of run performance based on the information gathered in the first few generations of GE. Then, we only allow the run to continue if it is predicted to be successful and terminate the remaining runs. This improves the probability of success while reducing the total amount of time significantly.
4.6.1 Run Improvement

The efforts to improve the performance of GP date back to its inception; GE being a form of GP, can also benefit from a number of such approaches. Typically, this work tries to improve the success rate of the results, which, in turn, impacts the total execution time.

For example, Keijzer [Keijzer, 2003] improved GP based symbolic regression results using interval arithmetic and linear scaling. An improved GP search was presented in [Gustafson et al., 2005] for the symbolic regression problems with an analysis on crossover and diversity. GP bloating individuals that occur at the later stages of the evolutionary cycle were killed [Poli, 2003], [Koza et al., 2005], thereby, improved GP performance. Tsakonas et al. [Tsakonas et al., 2004] applied kill tournaments [Smith and Vavak, 1999], replacing the worst fit individuals with the best, that predicted the classification rules with GP. Also, work shown in [Azad and Ryan, 2010], [Fitzgerald et al., 2013] improved the quality of GP applied on symbolic regression problems.

Although this is a tiny subset of vast amount of research, in general, they all have the same focus, that is to make the best out of the available genetic material. Our approach tries to ensure availability of genetic material that is as good as it can be, so that all these methods will get benefited from it.

4.6.2 Predicting The Quality of Runs

There are many possible reasons for a run to fail to achieve good quality solutions, but many manifest themselves in the form of the premature convergence [Koza, 1992] of the population to a sub-optimal solution leaving little diversity in the population to explore new solutions. In GE, Keijzer et al. [Keijzer et al., 2001] showed that the ripple crossover delays premature convergence. However, like other EAs, GE also suffers from the problem of locality [Rothlauf and Oetzel, 2006] that worsens the search results.

Some studies suggested that multiple independent runs could result in useful solutions. For example, Shonkwiler [Shonkwiler, 1993] argued that
multiple small runs can reach global solutions with fewer function evaluations than a single longer run for a GA.

The quality can be improved by modifying the search process as explained in literature or through multiple isolated runs. However, a different approach is to some how predict the runs that fail to produce high-quality solutions and terminate them. We achieve this goal by predicting the success or otherwise of a GE run using an ant colony optimization (ACO) [Dorigo and Stützle, 2004] based machine learning algorithm. As a result, we find that, along with an improvement in the final result of a run, our approach reduces the execution time using fitness prediction that leverage the difference between the individuals in the current and past generations.

A key difference with this work is that we are more concerned with producing a small number of high quality runs than producing a large number of statistically useful runs. We now discuss the GE run prediction model.

4.7 The Run Prediction Model

Our approach in predicting the run time performance is novel: to date, this is the first attempt to employ an ACO classifier to predict the performance of an EA run. We use cAnt – MinerPB [Medland and Otero, 2012], an ACO based classifier, to produce a rule based model that predicts the performance of GE. The selection of an ACO classifier is an ad hoc choice, whereby the goal here is to conduct a feasibility study on the run prediction rather the choice of the classifier. In fact, the ACO classifier can always be replaced with other algorithms that generate a model. Effectively, this model improves the proportion of successful GE runs on a selection of symbolic regression problems. The training data required for the classifier describes the changes in different parameters (such as fitness, genome lengths) at the early stages of an evolutionary cycle of GE, that is, how fast does a particular measure change. This model is further scrutinized to select the best set of rules to predict the performance. The predictive model thus produced is used in the prediction of an evolutionary cycle of a given GE run. Finally,
the runs that are predicted to be poor are immediately halted, freeing more resources for higher quality runs.

Figure 4.4: A block diagram of the run prediction model (RPM) with GE.

Figure 4.4 shows an overview of the run prediction model (RPM) applied to GE. We now discuss the role of ACO in discovering the classification models for prediction.

4.7.1 ACO Classifier

ACO is an optimization meta-heuristic inspired by the foraging and pheromone strategies of real ants. In essence, ants cooperate each other in the form of colonies in order to find an optimal solution to a given problem. For example, if we consider the travelling salesman problem (TSP): ants start from a random city (vertex) and then select the routes (edges) to add new cities that iterative process, in return, produces a solution in the form of pheromones present on the edges. Many applied ACO to a number of optimization problems [Dorigo et al., 1999], [Stützle and Dorigo, 1999], [Junjie and Dingwei, 2006], [Korb et al., 2006], [Dorigo and Stützle, 2010] and classification [Parpinelli et al., 2002], [Martens et al., 2007] is one of them.

The first ACO based classification algorithm, Ant-Miner, was introduced in [Parpinelli et al., 2002] that dealt only with the nominal attributes of
a dataset. Ant-Miner produces a list of IF-THEN classification rules of the form \( \text{IF BFC } \leq 0.0268 \text{ THEN No.} \) Following this, several variations were proposed, some dealt with pheromone update strategies and heuristic information [Martens et al., 2007], others discovered fuzzy classification rules [Galea and Shen, 2006]. One such extension is \( cAnt - Miner_{PB} \) [Medland and Otero, 2012], showed much better accuracy with a list quality function, that can deal with numerical attributes also.

\( cAnt - Miner_{PB} \) algorithm starts with an empty rule list and a training set in each iteration and, repeated for a maximum number of iterations or as long as the algorithm converges. In this process, the ant creates a rule from the data points in the training set. Thereafter, the ant prunes the rule and removes all the covered data points from the training set. This process is repeated as far as the remaining data points in the training set become less than the user specified number of uncovered training points. This way a rule list is prepared in each iteration. Following that, the quality of the current list of rules is compared with the previous list of rules, and replaces if the current list is better than the best so far. This way, ants construct a list of best rules. A data point is considered correctly classified if the predicted class value is same as the actual class value.

We now employ \( cAnt - Miner_{PB} \) to improve the quality of GE results. The following section describes the ACO learning process in predicting the GE runs and the implementation details of the novel approach.

### 4.7.2 RPM+GE

The ACO based learning algorithm uses the training data, as shown in Figure 4.4, in discovering the prediction model. The model operates by taking GE run as an input. It tries to identify the quality of the solution attainable at the end of a GE run. This way, the model allows only the promising runs to continue while terminates the remaining runs at the early generations.

Figure 4.5 presents the flowchart of run prediction model (RPM) applied on GE. The proposed approach follows the standard GE algorithm, except for a minor interruption through the prediction model, which decides the
Figure 4.5: The flowchart of Run Prediction Model (RPM) that predicts the performance of GE runs.
quality of a GE run. In the flowchart, the variables run and gen represent
the currently executing run and the generation respectively.

This approach starts with an initial random population of individuals. As
with GE, the next generation of individuals are evolved iteratively perform-
ing genetic operations, mapping the genotypes to phenotypes, evaluating
the fitness of the phenotypes, and increases the generation count. In this
process, the algorithm records the values of four parameters: best fitness,
average fitness, average actual length and, average effective length; at gen-
eration 1 and 10. We note the change in these four parameters, that is the
difference in the respective parameter from generation 1 to 10. This change
helps to identify the success or otherwise of a GE run.

An interesting observation is, what happens if we consider the dif-
ference between generation 1 and 5 or generation 1 and 20. In fact, the for-
mer choice achieves a better optimization in terms of execution time of a
GE run. However, the calculated change resulted in an approximately zero
difference in most of the runs, which is not useful for prediction, thus omit-
ted this choice. With the latter possibility each run needs to be executed
for 20 generations, that increases the total execution time, which is time-
consuming, thus neglected this alternative. Taking these two factors into
consideration, as well as the observed changes in the respective parameter
values, we leaned towards experimenting with the difference between gen-
nerations 1 and 10.

If the observed changes in any one or all of the recorded parameters
satisfy the conditions in the ACO classifier discovered model then, that run
is terminated reporting it as the best so far fitness. Otherwise, the algorithm
continues to run reporting the best of run result. This process is repeated for
a user defined number of runs. Although RPM is applied on GE, it can be
operated in tandem with virtually any system from the literature.
4.8 Experiments

This section evaluates the above presented method on a set of benchmark regression problems. The following sections present the generated training data of the problem, the experimental setup and the results.

4.8.1 Benchmarks and Training data

The experiments use four symbolic regression problems. Table 4.2 describes these four problems. The first column stands as an index to the problem; henceforth, we use this index to refer to the corresponding problem throughout this chapter. The second column gives the target function followed by a citation to the source of the problem from the GP literature. The third column shows the classification of data points and the fourth column represents success threshold, the classification criterion.

<table>
<thead>
<tr>
<th>Problem:Source</th>
<th>Class</th>
<th>Success Threshold</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1 + x)^3;[Daida et al., 2001]</td>
<td>590 410</td>
<td>0.7</td>
</tr>
<tr>
<td>x^4 - x^2 - y^2 - y;[Topchy and Punch, 2001]</td>
<td>606 394</td>
<td>0.65</td>
</tr>
<tr>
<td>x^3 - y^3 - y - x;[Topchy and Punch, 2001]</td>
<td>848 152</td>
<td>0.7</td>
</tr>
<tr>
<td>x^3;[Streeter and Becker, 2003]</td>
<td>518 482</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 4.2: The problem set considered. The “Yes” column represents the number of runs producing results above the threshold; the “No” column stands for the runs that remained below the threshold. The success threshold is decided on the Best Fitness (BF).
Table 4.3: A brief description of the experimental datasets. BFC (Best Fitness Change), AFC (Average Fitness Change), AALC (Average Actual Length Change), AELC (Average Effective Length Change) are the attributes of the dataset. The data points represent the change from generation 1 to 10.

<table>
<thead>
<tr>
<th>BFC</th>
<th>AFC</th>
<th>AALC</th>
<th>AELC</th>
<th>Class label</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0220119</td>
<td>0.0481755</td>
<td>2.005</td>
<td>-18.995</td>
<td>No</td>
</tr>
<tr>
<td>0.0530694</td>
<td>0.0641799</td>
<td>6.105</td>
<td>-12.565</td>
<td>Yes</td>
</tr>
<tr>
<td>0.0868888</td>
<td>0.0622562</td>
<td>10.945</td>
<td>-17.825</td>
<td>No</td>
</tr>
</tbody>
</table>

For each problem, a training set has been prepared to train the ACO classifier. These datasets are prepared using the experimental procedure explained in section 4.4, that is executing GE for 1000 runs on each of the four problems. Each dataset contains 1000 data points that are classified into two labels, Yes and No. The number of Yes and No class labels are presented in Table 4.2 for the respective problem. The data points are classified depending on the user defined success threshold, Best Fitness (BF). For example, in the case of $f_1$, if $BF > 0.7$ then those data points belong to Yes class, No otherwise. Since the regression problems are hard to solve and the evolved solution of an EA varies largely from a problem to problem, we consider a different threshold value as a success measure for each problem. Thus, the runs that belong to Yes class are promising runs and No class are poor runs.

Table 4.3 presents an example description of $f_1$ training set. The data set contains four attributes: BFC, AFC, AALC, AELC that represent the change in generations 1 and 10 for respective parameters of a problem. The two fitness related attributes (BFC and AFC) are intuitive; however, these are not the only possible attributes, and future work will examine more. All
the attributes in the resulting datasets are continuous with no duplicates. Although the computational effort in preparing the training sets is rather high for this preliminary investigation, it has resulted highly in improving the success rate and optimizing the execution time with the help of the proposed run prediction model. Also, it resulted in the abundant availability of datasets for future research.

4.8.2 Experimental Setup and Results

We have conducted two sets of experiments. In the first set, we compare the predictive accuracy of the ACO based classifier with those produced by a number of contemporary rule based machine learning algorithms. We find that the ACO based classifier outperforms the benchmark machine learning methods. In the second set, we apply the ACO devised predictive models to actual GE runs after a manual inspection in order to detect and terminate poor runs and present the results showing the efficacy of the proposed approach.

Discovered Prediction Models

We ran $c\text{Ant} – \text{Miner}_{PB}$ mining algorithm on all the four datasets to generate four different predictive models, one for each problem. The set parameters for the algorithm are: $\text{ant\_colony\_size} = 5$, $\text{iterations} = 500$, $\text{minimum\_cases\_per\_interval} = 10$ and $\text{evaporation\_probability} = 0.9$. A complete description of these parameters can be found in [Parpinelli et al., 2002].

We compare the predictive accuracy of $c\text{Ant} – \text{Miner}_{PB}$ with that of four state-of-the-art machine learning algorithms on the four datasets considered in this study. For these machine learning algorithms, default parameters are used, details of which can be seen in [Witten et al., 2011].

Table 4.4 shows the mean accuracy followed by standard deviation of a 10-fold cross-validation. $c\text{Ant} – \text{Miner}_{PB}$ exhibited significantly better predictive accuracy than its counterparts when performed the Wilcoxon Signed Rank Sum significance tests at a significance level of $\alpha = 5\%$. Overall,
Table 4.4: Comparison of predictive accuracy (mean [standard deviation]) in %, among J48, ZeroR, cAnt – MinerPB, PART and JRip measured by applying 10-fold cross-validation. The value of the most accurate algorithm is shown in bold for a given data set with Wilcoxon Signed Rank Sum tests at \( \alpha = 5\% \).

<table>
<thead>
<tr>
<th>Problem</th>
<th>J48</th>
<th>ZeroR</th>
<th>cAnt – MinerPB</th>
<th>PART</th>
<th>JRip</th>
</tr>
</thead>
<tbody>
<tr>
<td>(f_1)</td>
<td>58.4 [0.49]</td>
<td>54.12 [0.95]</td>
<td><strong>78.72 [0.84]</strong></td>
<td>57.85 [0.63]</td>
<td>59.18 [0.81]</td>
</tr>
<tr>
<td>(f_2)</td>
<td>60.61 [0.89]</td>
<td>60.25 [1.32]</td>
<td><strong>69.32 [0.61]</strong></td>
<td>61.19 [1.58]</td>
<td>62.60 [1.02]</td>
</tr>
<tr>
<td>(f_3)</td>
<td>79.44 [0.35]</td>
<td>78.43 [0.26]</td>
<td><strong>86.96 [0.45]</strong></td>
<td>80.88 [1.01]</td>
<td>80.24 [0.56]</td>
</tr>
<tr>
<td>(f_4)</td>
<td>54.21 [0.54]</td>
<td>51.82 [0.64]</td>
<td><strong>68.12 [0.99]</strong></td>
<td>56.10 [0.38]</td>
<td>58.50 [0.47]</td>
</tr>
</tbody>
</table>

IF AALC <= –12.8875 THEN No;
IF BFC <= 0.0268 THEN No;
IF AELC <= –23.31 THEN Yes;
IF AFC > 0.0731 THEN Yes;
IF 0.0614 < AFC <= 0.0633 THEN No;
IF AELC <= –21.725 THEN Yes;
IF AELC <= –20.99 THEN No;
IF AALC <= –6.035 THEN Yes;
IF 7.2475 < AALC <= 8.3425 THEN No;
IF BFC <= 0.0268 THEN No;
IF AFC <= 0.0534 THEN No;
IF BFC > 0.0656 THEN yes;
IF AELC > –17.7924 AND AALC > –1.5025 THEN Yes;
IF BFC > 0.0558 THEN No;
IF BFC > 0.0496 AND AFC <= 0.0613 THEN Yes;
IF AFC > 0.0558 AND AELC > –22.855 THEN No;
IF BFC > 0.0292 THEN Yes;
IF <empty> THEN Yes;

Figure 4.6: ACO discovered predictive model for problem \(f_1\)
cAnt – MinerPB models exhibit significantly better predictive accuracy outperforming the remaining classifiers on all the four datasets.

The ACO learning algorithm discovered four prediction models, one for each regression problem. For example, Figure 4.6 shows the discovered model of $f_1$. This model is selected depending on the predictive accuracy, where it showed better over that of the other state-of-the-art machine learning algorithms. In the model, a line that starts with an IF and ends with a “;” (semi-colon) represents a rule. The model consists of the best list of rules discovered by ants.

**Rule Selection**

Examining the classification model shown in Figure 4.6, the rules in the model are simple enough for a manual inspection and analysis, where ants devised a manageable number of rules with a single attribute alone or in combination of more than one attribute and a class. As our focus is to predict the poor runs, therefore, we consider the rules that predict the class value No from the discovered rule list. Of those rules, it is difficult to identify clearly which rule of the model suits best in run prediction, we manually explored different possibilities of these rules on the training sets of the respective problem in order to identify the best combination of rules. We focused on the rules that can identify the poor runs only. This analysis is fairly simple and the devised rules are applied on an unseen data while running GE.

Table 4.5 presents the selected rules through the manual analysis. In fact they may sometimes fail to exactly identify the poor runs, likewise, terminate a few promising runs. We treat this as the error exerted by the predictor, which is explained later in section 4.8.3.

**Impact of Predictors**

With the predictors generated, the next step is to assess their effectiveness through the second set of experiments. In this set, we compare standard GE results with that of the RPM+GE. For standard GE, we report the best
Table 4.5: Manually selected rules in predicting the performance of a GE run for all the four experimental problems.

<table>
<thead>
<tr>
<th>Problem</th>
<th>Selected Rule(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$f_1$</td>
<td>if(bfc $\leq$ 0.027)</td>
</tr>
<tr>
<td>$f_2$</td>
<td>if((bfc $&gt;$ 0.022) &amp;&amp; (afc $&gt;$ 0.087))</td>
</tr>
<tr>
<td>$f_3$</td>
<td>if(aalc $&gt;$ 5.97) &amp;&amp; if((bfc $\leq$ 0.19) &amp;&amp; (aelc $&gt;$ 3.84))</td>
</tr>
<tr>
<td>$f_4$</td>
<td>if(bfc $&gt;$ 0.035) &amp;&amp; if((aalc $&lt;$ -17.79) &amp;&amp; (aelc $&gt;$ -22.85))</td>
</tr>
</tbody>
</table>

Fitness at the end of run for all four problems. In case of RPM+GE, we report the best fitness of both model allowing and terminating runs. Finally, we compare the execution times of both the approaches.

The following parameters are kept consistent for both the approaches. They are: number of runs = 30, population size = 500, number of generations = 120, crossover probability = 0.9, mutation probability = 0.01 per gene, randomly initialized the population with a minimum and maximum genotype length of 15 and 25 respectively, a random seed value is incremented for each run, and a steady state replacement strategy. We used an S-Lang interpreter for fitness evaluation. A run in this experimental settings is considered long (with 120 generations) when compared to the standard GE settings (50 generations). The main reason for this setting is that a single large run reaches the global solution faster than multiple small runs [Cantú-Paz and Goldberg, 2003], thus saves the computational resources.

Figure 4.7 shows the best fitness of standard GE versus RPM+GE for 30 runs. Standard GE results are in squares, whereas the RPM+GE results are in triangles and circles, of which, the triangles represent the model al-
Figure 4.7: The best fitness of standard GE (shown in squares) versus RPM+GE (shown in triangles) with the terminated runs (shown in circles) for the four symbolic regression problems. The horizontal dashed line represents the pre-defined success threshold. The end of run results of all four problems illustrate an improvement in the quality of runs after applying the prediction model.
allowed runs to complete their execution, whereas the circles represent the terminated runs. There is a horizontal dashed line parallel to \( x\text{-axis} \), which shows the success measure (defined in section 4.8.1) of the respective problem. Some of the squares and triangles overlap each other. This is because the RPM+GE predicts the performance of a GE run without modifying the search process. This results in improving the success rate of GE results.

**Table 4.6:** Comparing the success rate of the standard GE versus RPM+GE measured for 30 evolutionary runs.

<table>
<thead>
<tr>
<th>Problem</th>
<th>GE</th>
<th>RPM+GE</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>promising runs</td>
<td>% of success</td>
</tr>
<tr>
<td>( f_1 )</td>
<td>18</td>
<td>56.66</td>
</tr>
<tr>
<td>( f_2 )</td>
<td>7</td>
<td>23.33</td>
</tr>
<tr>
<td>( f_3 )</td>
<td>6</td>
<td>20.00</td>
</tr>
<tr>
<td>( f_4 )</td>
<td>17</td>
<td>56.66</td>
</tr>
</tbody>
</table>

Table 4.6 summarizes the number of promising runs and, the success rate of standard GE and RPM+GE approaches over 30 runs. In addition to this the count of terminated runs resulted from the run prediction model are reported. The RPM+GE exhibits significant improvement in the % of promising results over standard GE on all the four problems. While \( f_3 \) enjoys only a small, albeit still statistically significant, improvement of 3.81%, the remaining three problems \( f_1 \), \( f_2 \) and \( f_4 \) exhibit 14.77%, 41.82%, and 13.93% respectively. Notice that \( f_2 \) exhibits bigger jump in the success rate; it is because, the model has terminated 23 runs identifying them as poor runs. *This behaviour of the model requires us to consider how many good runs are killed by the model?* We analyze such an inherent error of the new prediction approach in section 4.8.3.
Table 4.7: Wilcoxon tests at $\alpha = 5\%$ on the execution time of GE ($T_{GE}$) versus RPM+GE ($T_{RPM+GE}$) over 30 runs. Note, “✓” shows that RPM+GE outperforms standard GE, whereas “✗” shows that standard GE outperforms RPM+GE. When there is a significant difference, $A$ measure is shown, otherwise (−).

<table>
<thead>
<tr>
<th>Problem</th>
<th>$p$ value</th>
<th>Significant</th>
<th>$A$ measure</th>
<th>$T_{GE}$ (in sec)</th>
<th>$T_{RPM+GE}$ (in sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$f_1$</td>
<td>0.0556</td>
<td>✓</td>
<td>–</td>
<td>91.83</td>
<td>66.78</td>
</tr>
<tr>
<td>$f_2$</td>
<td>0.0008</td>
<td>✓</td>
<td>0.831</td>
<td>88.24</td>
<td>25.43</td>
</tr>
<tr>
<td>$f_3$</td>
<td>0.0300</td>
<td>✓</td>
<td>0.661</td>
<td>87.65</td>
<td>61.93</td>
</tr>
<tr>
<td>$f_4$</td>
<td>0.0100</td>
<td>✓</td>
<td>0.692</td>
<td>87.73</td>
<td>52.31</td>
</tr>
</tbody>
</table>

Meanwhile, the amount of processing time required to evolve the solution is a major concern in any EA, so does for GE. Table 4.7 shows the execution time results of both the approaches with their statistical significance. $T_{GE}$ and $T_{RPM+GE}$ stands for total execution time of standard GE and RPM+GE respectively. We observe that RPM+GE significantly reduces the execution time over standard GE for all the problems except for $f_1$. The $p$-value of the Wilcoxon tests show the significance of each problem.

The non-parametric Wilcoxon statistical significance tests state whether the model applied GE is significantly better or not, but they do not state how much better one approach over the other. We use *Vargha-Delaney A* [Vargha and Delaney, 2000] measure that tells us the probability of RPM+GE achieving better performance over standard GE. When the $A$ measure is above 0.5, standard GE outperforms the RPM+GE. When it is 0.5, both are equal, otherwise, RPM+GE outperforms the standard GE. The $A$ measure shows that RPM+GE achieves better performance over standard GE with 83.16%, 66.11% and, 69.17% of times on $f_2$, $f_3$ and $f_4$ respectively. Since the
Wilcoxon tests showed an insignificant difference between the two approaches for $f_1$, there is no need to estimate $A$ measure for $f_1$.

Finally, the results in Table 4.6 and Table 4.7 show that the introduction of an ACO based predictive model has significantly improved the rate of success of GE in terms of the promising evolutionary runs, while reduced the total execution time on the respective regression problems. Although the prediction results are encouraging, like any other predictor, RPM also induces error in predicting the performance of GE runs.

### 4.8.3 Analysis of RPM Prediction Error

This section analyses the RPM induced prediction error. RPM+GE classifies a few GE runs as promising or poor quality runs and terminates or allows to continue their execution, erroneous classification of RPM.

<table>
<thead>
<tr>
<th>Problem</th>
<th>TP</th>
<th>FP</th>
<th>FN</th>
<th>TN</th>
</tr>
</thead>
<tbody>
<tr>
<td>$f_1$</td>
<td>15</td>
<td>6</td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>$f_2$</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>19</td>
</tr>
<tr>
<td>$f_3$</td>
<td>5</td>
<td>16</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>$f_4$</td>
<td>12</td>
<td>5</td>
<td>5</td>
<td>8</td>
</tr>
</tbody>
</table>

Table 4.8 shows the prediction results of RPM+GE on all the four regression problems. The terms are defined as follows: $TP$ is the actual number of correctly predicted promising runs; $FP$ is the number of runs that are actually of poor quality, but predicted as promising runs; $FN$ is the number of runs that are actually promising, but predicted as poor runs; $TN$ is the actual number of correctly predicted poor runs. Although the results in Table 4.6 confirm the improvement in the success rate of GE, Table 4.8 shows
that a few number of promising runs (FN) are terminated. Similarly, a few number of poor runs (FP) are allowed to complete their execution. From these two observations, it is now clear that there is a scope for increasing the predictive accuracy of the model by minimizing the FPs and FNs.

Surprisingly, problem $f_3$ has high FP rate. To better justify this, if we refer back to Figure 4.7, note that most of these runs fall just short of crossing the threshold. Therefore, the need for a too sensitive predictor is evident in classifying such runs as poor runs. Nonetheless, this performance is not a disaster if we allow for a small margin of error around the threshold.

**Table 4.9:** Total number of individuals processed and the mean best-of-run fitness (mean [standard deviation]) for both GE and RPM+GE.

<table>
<thead>
<tr>
<th>Problem</th>
<th>GE</th>
<th>RPM+GE</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Total individuals processed</td>
<td>Mean best-of-run fitness</td>
</tr>
<tr>
<td>$f_1$</td>
<td>1800000</td>
<td>0.73 [0.13]</td>
</tr>
<tr>
<td>$f_2$</td>
<td>1800000</td>
<td>0.54 [0.15]</td>
</tr>
<tr>
<td>$f_3$</td>
<td>1800000</td>
<td>0.61 [0.05]</td>
</tr>
<tr>
<td>$f_4$</td>
<td>1800000</td>
<td>0.94 [0.08]</td>
</tr>
</tbody>
</table>

This error rate of RPM+GE is compensated through the amount of effort required to generate best fitness. Table 4.9 shows the total number of individuals processed and the mean best-of-run fitness for both standard GE and RPM+GE. The total individuals processed for GE are calculated as: the total number of individuals processed = the number of runs * population size * the number of generations. Similarly, for RPM+GE, it is calculated as: the total number of individuals processed = (the non-terminated number
of runs * population size * the number of generations) + (the number of terminated runs * population size * the number of generations allowed before termination). The results indicate that RPM+GE achieves better fitness by processing a less number of individuals than standard GE. This outweighs the prediction error and shows that it is possible to improve the quality of a solution.

This analysis clearly suggests that the ACO discovered models are simple enough to analyse and easy to use in predicting the performance of a GE run. Combined use of ACO classifier with GE produced a reasonably good prediction system. As a result, a significant improvement has been observed in the number of promising runs while significantly reducing the total execution time.
4.9 Summary

This chapter empirically analysed the computational complexity of GE on two standard benchmark problems: Santa Fe Ant Trail and binomial-3. This analysis suggested that the time complexity of GE is linear for Santa Fe Ant Trail and quadratic for binomial-3. It also showed certain patterns of alignment in four different characteristics of GE, which motivated to propose a prediction system.

We have successfully produced a reasonably good run prediction model that depends on an ACO classifier. This model identified and terminated the poor quality GE runs, thereby, improved the performance of GE both in terms of the number of quality runs and execution time. In fact, this work can have huge implications for scaling EC algorithms which, due to their stochastic nature, often confront prediction when it comes to the length of the runs. One assumption that we have made is the existence of good predictive data.

The next step will be concerned with bootstrapping the system, and we are currently examining two approaches. First, the use of data produced on smaller but related versions of the problem, similar to the technique used in [Keijzer et al., 2004] will be examined. Although not perfect, this could get over the initial bootstrapping phase, particularly if the problem being tackled is a scaled up version of the one that the data came from.

Secondly, and probably in tandem with the first method, we will investigate rapidly retraining the predictors during the runs as increasingly more data becomes available, similar to work by [Ryan et al., 2005], [Costelloe and Ryan, 2009], [Krawiec and Swan, 2013]. This would facilitate increasingly accurate predictors as more data becomes available.
Chapter 5

Grammatical Automatic Parallel Programming

“Grammar, which knows how to control even kings”

— Molière, Playwright

This chapter describes the novel Grammatical Automatic Parallel Programming (GAPP) [Chennupati et al., 2014a], [Chennupati et al., 2014b] for utilizing the on-chip multiple CPU architectures to automatically evolve parallel computer programs. These programs have the capability of exploiting the computational efficiency of the multi-core machines. We use OpenMP primitives through grammars, such that not only can we exploit parallelism while evolving individuals, but the final individuals produced can also be executed on parallel architectures even outside the evolutionary system.

This chapter is organised as follows. Section 5.3 describes the proposed approach in evolving native parallel programs. Section 5.4 evaluates our approach on two difficult GP benchmark problems. Section 5.5 shows its efficiency in exploiting the power of multi-core architectures, and section 5.6 concludes.
5.1 Introduction

Multi-core processors in desktops and PCs have efficient computational capabilities that can improve the performance of applications at a reasonable cost. As the number of cores on a single chip increase, however, operating systems find it increasingly difficult to occupy all the cores. Thus, developing applications that can fully realise the computational power of these processors is an apt solution.

EC, on the other hand, has less baggage than its human counterparts, making parallel programs an ideal target. Although automatic programming is a familiar terrain for the GP community, automatic parallel programming has attracted relatively less attention. While parallelising of sequential software has some precedent in GP, generating natively parallel code remains unprecedented. The work in this chapter proposes a novel approach, GAPP, an application of Grammatical Evolution in evolving natively parallel programs. It uses OpenMP primitives with the GE’s problem specific CFGs to evolve parallel programs.

Utilizing the power of multi-cores can optimize the execution time significantly. Thus, EC literature has several examples, where the evolutionary cycle is optimized to adapt to the multi and many cores. The following section discusses some of them. However, note that these attempts did not generate parallel programs.

5.2 Background

This section discusses the existing attempts in parallelising the evolutionary process and distinguishes our approach from them. Most of the EAs are inherently parallel or contain the design that is suitable for explicit parallelization. Explicit parallelization attempts range from evaluating an individual on independent fitness cases to executing the evolutionary runs on multiple independent parallel hardware. This capability of EC to exploit parallel hardware is often noted as “embarrassingly parallel”.
Cantú-Paz [Cantú-Paz, 1998] surveyed different parallel implementations of GAs, where the prime focus was on those that maintain multiple populations, evolve and exchange individuals among themselves. A survey of parallel distributed genetic algorithms [Alba and Troya, 1999], outlined the way that many sub-algorithms operate on sparse migrations of individuals. Different parallel models and tools of EAs in building the parallel evolutionary algorithms were surveyed in [Alba and Tomassini, 2002], which discussed the algorithmic implementation difficulties.

GP parallelization works date back to 1996, when an SIMD GP system [Juillé and Pollack, 1996] was implemented on a MasPar MP-2 super computer, which consists of thousands of processing elements. At around the same time, Cray super computers were used to parallelize GP written in Fortran [Turton et al., 1996]. The GP population was split to evolve separately on Beowulf cluster of machines [Andre and Koza, 1998], [Bennett III et al., 1999], where emigration among the computational nodes was limited. Also, works in [Chong and Langdon, 1999], [Groß et al., 2002] used Java and internet to distribute GP populations globally. Later, GP population of individuals with different configurations exploited the power of multiple computers [Washita and Iba, 2002]. A number of approaches that can accelerate GP are outlined in [Poli et al., 2008].

GP acceleration techniques span into exploiting the recently emerging graphics processing units (GPUs) [Fernando, 2004], highly parallel computational resources. Chitty [Chitty, 2007] pre-compiled the GP programs on a CPU before porting them onto GPU for fitness evaluation. Harding and Banzhaf [Harding and Banzhaf, 2007] parallelized the fitness evaluations of Cartesian GP (CGP) [Poli et al., 2000] on a GPU. Later, GP programs were interpreted as Single Program Multiple Data (SPMD) instructions on GPU cards [Langdon and Banzhaf, 2008]; this approach was later used for bio-informatics data mining [Langdon and Harrison, 2008]. Robilliard et al. [Robilliard et al., 2008], [Robilliard et al., 2009] tried to exploit the power of GPUs in parallelising GP fitness evaluations in SPMD style. Population parallel GP [Chitty, 2012], a parallel implementation of GP on CPU, the performance of which was competitive when executed on a GPU.
Like GP, GE is suitable for parallelization. The first parallelization attempt on GE was made in [Stallard, 2006] with an island model that was implemented in Python programming language. It optimized the total execution time when tested on a 17 node cluster with each node containing an Intel Pentium IV dual core processor. Osmera et al., [Osmera et al., 2008] proposed a two level population parallel approach with GE in the first level and Differential Evolution [Storn and Price, 1997] in the next level in a hierarchical master-slave configuration, where the slaves port their best candidate solutions to the master. They tested it on a group of 6 computers networked together with one master and five slaves with two populations (female, male) on each.

Recently, an accelerating GE [Pospichal et al., 2011] reported promising speedup by running the entire GE algorithm on GPU; it evolved regression programs. Although the proposed approach improved the execution time, it has a serious limitation in the use of GPU memory that would decelerate the speed of execution when the population size increases.

Parallel GE implementations are relatively few in number due to the need to optimize the code to the specialized parallel hardware such as GPUs. However, modern desktop CPU computers have multiple cores and their number will increase. The computational power of these architectures has been little explored leaving a lot of scope for us to investigate. This work, for the first time employs grammars to exploit the computational capabilities of multi-core architectures by automatically evolving parallel programs.

The evolutionary automatic parallel code generating approaches [Ryan and Walsh, 1997a], [Williams, 1998], discussed in Chapter 3 are different to the approach discussed here; both try to optimize the non-functional property of applications, that is, time. Another common feature in both the branches of the literature is the SPMD style of programming. However, GAPP takes advantage of this characteristic in synthesizing the parallel programs, as opposed to parallelising an existing sequential program.
5.3 Grammatical Automatic Parallel Programming (GAPP)

GAPP presents the first instance of using grammars for the task of automatic parallel programming. This system provides an alternative to the *craftsman* approach of parallel programming. This is significantly different from other parallel EC approaches, because not only do we produce individuals that, in their *final form*, can exploit parallel architectures, we can also exploit the same parallel architecture during evolution to reduce the execution time. OpenMP primitives are an integral part of the grammars. Next, section 5.3.1 presents an overview of GAPP; section 5.3.2 shows the multi-core grammars and section 5.3.3 explains the SPMD strategy of the programs.

5.3.1 System Overview

![An overview of GAPP parallel program generating system.](image)

**Figure 5.1:** An overview of GAPP parallel program generating system.
Figure 5.1 presents an overview of GAPP. As with any application of GE, it uses the typical search process, genetic operations, and genotype-phenotype mapping. GAPP operates on a string of codons that separate the search and the solution spaces. Note, these integer strings are genotypes while the resulting parallel programs are phenotypes.

```plaintext
<stdio> ::= <omp pragma><for loop><evolve exp>
<omp pragma> ::= #pragma omp parallel shared(Evolve, CHUNK)<newline>{<newline><omp for>
<omp for> ::= #pragma omp for schedule(dynamic, CHUNK) private(i, temp)<newline>{
<for loop> ::= for(i=0;i<FITNESS_CASES;i=i+1) {
<evolve exp> ::= temp=<expr>;<newline><assign>
<expr> ::= <expr><op><expr>|(<expr><op><expr>)
 | <pre-op> ( <expr> ) | <var>
<op> ::= + | - | * | /=
<pre-op> ::= Sin | Cos | Exp | Log
<var> ::= X[i] | 1.000 | <const> | -<const>
<const> ::= 0.<digit><digit><digit>
<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<assign> ::= Evolved[i] = temp;
 } // end for loop
 } // end parallel for
 } // end parallel region
<newline> ::= \n
<stdio>
```

Figure 5.2: Design of GAPP grammar for symbolic regression problem.

After sensible initialization of the population, standard genetic operations such as crossover and mutation work on the strings. The exchange of genetic material through one-point crossover with a probability of 0.9 occurs at the codon boundaries. A point mutation with a probability of 0.01 per bit occurs on the genome. These parameters are chosen following the standard GP [Koza, 1992] and GE [O’Neill and Ryan, 2003] practices. A
roulette wheel selection with steady state replacement are used to breed the next generation. These genetic operations are detailed in Chapter 2.

The individuals resulting from the genetic manipulation are transformed into parallel programs by employing the standard GE mapping through the problem specific OpenMP grammar. These programs are evaluated on a number of fitness cases such that the best among them is identified as an end parallel program. In this process the design of the grammar plays a vital role; the following section discusses the design of these grammars.

5.3.2 Design of Grammar

This section describes the design of grammar that includes the OpenMP parallelization pragmas in BNF. As each core of a multi-core machine is a single unique CPU, under any operating system environment, calling OpenMP enabled C/C++ function creates the user requested number of threads. These parallel programming pragmas can execute independent threads on cores.

Figure 5.2 presents an example multi-core grammar that evolves the parallel symbolic regression programs. The grammar uses `omp parallel` and `omp for` with the `shared`, `schedule` and `private` clauses. They operate on the following variables: `Evolved`, `CHUNK`, `i` (array index) and `temp`. `Evolved` is an array that stores the result of an expression; `temp` holds the result of the evolved expression, which is stored in `Evolved` array at an index `i`, while the variables, `i` and `temp` are thread-specific private variables that guarantee the correctness of parallel execution. `CHUNK` is a user defined integer constant used for scheduling the loop iterations.

Of the three OpenMP scheduling alternatives, for these preliminary investigations, we use `dynamic` scheduling, later, Chapter 6 details the way that GE selects one of them automatically. The evolved programs contain an expression that works on an array `X`, which accepts an input from the given dataset. The loop iterates over `FITNESS_CASES` number of fitness cases. For example, a `CHUNK` number of examples are dynamically (as explained in Chapter 3) assigned to the threads and the remaining of which are
assigned to the threads that are idle. Sometimes, finding an optimal value of \textit{CHUNK} becomes a challenge, a detailed investigation is in Chapter 8.

Thus, the evolved programs follow a \textit{data parallel} approach; the following section discusses the parallelization strategy of the evolved programs.

5.3.3 Parallelization Strategy

The synthesized parallel programs exploit the computational power of multi-cores executing the same program across multiple fitnesses on all the cores.

![Parallel Evaluation Strategy](image)

\textbf{Figure 5.3:} Parallel evaluation strategy of the evolved parallel program, Program X, on a multi-core CPU, where different data is processed on each core.

Figure 5.3 shows the evaluation strategy of the evolved parallel program on a four core CPU. It operates on \( k \) number of fitness cases stored in an input array, \( X \), that are shared among all the cores. The main characteristic of this approach is: a \textbf{single} evolved program is executed across all the cores with a distinct set of fitness cases, thus making the execution Single Program
Multiple Data (SPMD). For example, on a 4 core processor, ideally each core evaluates $1/4^{th}$ of the fitness cases, which might vary due to dynamic scheduling, and must report an evaluation speedup by a factor of 4. This is the ideal case, which, unfortunately, is most unlikely to happen due to the inability of an operating system to scale linearly.

In this scheme some threads finish their execution early while the others fall behind, so that there is an issue of thread synchronization. Also, there is a chance for the occurrence of race conditions. All such vulnerabilities are treated later in Chapter 6. Finally, all the cores write their outputs in the Evolved array. The following section examines the proposed approach.

5.4 Experiments

We examine GAPP on two standard benchmark problems: a boolean multiplexer [Koza, 1991] problem and a sextic polynomial ($x^6 - 2x^4 + x^2$) regression [Koza, 1994]. The first benchmark has been tested on 6, 8 and 11 bit multiplexers that contain 64, 256 and 2048 fitness cases, while the second benchmark contains 64, 512 and 1024 fitness cases in the range [-1.5, +1.5]. Figure 5.2 presents the multi-core grammar for the sextic regression problem, while Figure B.1 in Appendix B shows the multiplexer grammar. A complete description of the serial implementation of these grammars can be found in [Azad and Ryan, 2005]. The fitness function is different for both the problems. For multiplexer, it is the number of hits, where the evolved outputs are same as the target outputs. For sextic polynomial regression, it is the normalized mean squared error of the evolved and the target outputs.

5.4.1 Experimental Setup

The experiments are performed on an Intel (R) Xeon (R) CPU E7-4820 with a processor speed of 2 GHz containing 16 cores, each of which can act as 2 independent cores when hyper-threading is enabled and 18 Mb of L3 cache memory was shared among all the cores. The experimental parameters are
set as follows. We use libGE [Nicolau and Slattery, 2006], a C++ implementation of GE. The population of individuals is initialized randomly with minimum and maximum depths of 15 and 25 respectively.

It is interesting to see the performance of the proposed approach with different population sizes and different number of CPU cores, because scaling the performance is challenging. The experimental population sizes are \{25, 50, 100, 200, 400\}, while the number of CPU cores are \{2, 4, 8, 16\}. One point crossover with a probability of 0.9, point mutation with a probability of 0.01 per bit are used. We use steady state GA where the best individuals replace the worst in the population. The number of generations are 50 while the number of runs are 30. GNU GCC compiler is used to compile the evolved programs, which are then executed to evaluate them.

### 5.4.2 Experimental Results

We compare the execution time of the sequential programs evolved using standard GE with that of the parallel programs produced through GAPP. In both the cases, we report the average execution time of all the evolved programs across 30 runs. These execution time results are measured using the OpenMP platform independent timer utility function, `omp_get_wtime()`.

Figure 5.4 presents the evolved parallel sextic polynomial regression program. Tables 5.1 and 5.2 show the execution time of all the GAPP evolved parallel programs averaged across 30 runs when executed on 2, 4, 8, and 16 cores of a processor with different population sizes and fitness cases on both the benchmarks. The 1-core results represent a standard GE application that evolves sequential programs. These results show that the evolved parallel programs scale with the number of cores irrespective of the population size for both the problems, thus the execution time decreases. Nonetheless, for 2 cores, there is no significant difference in the execution time between the sequential and parallel programs.

In fact, at some instances there is a slight increase in the time, for example, in Table 5.1 at a population size of 50 for 64 fitness cases, the 2-core...
Table 5.1: The execution time (average [standard deviation]) of GAPP evolved parallel multiplexer programs averaged across 30 runs with 64, 256 and 2048 fitness cases, that is 6, 8 and 11 multiplexers.

<table>
<thead>
<tr>
<th>Fitness Cases</th>
<th>Pop Size</th>
<th>Cores</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>64</td>
<td>335.84 [27.32]</td>
<td>177.11 [24.81]</td>
</tr>
<tr>
<td>100</td>
<td>1099.07 [29.46]</td>
<td>456.42 [28.16]</td>
</tr>
<tr>
<td>50</td>
<td>610.06 [24.73]</td>
<td>63.91 [21.67]</td>
</tr>
<tr>
<td>100</td>
<td>2487.88 [33.46]</td>
<td>161.32 [24.04]</td>
</tr>
</tbody>
</table>
Table 5.2: The execution time (average [standard deviation]) of GAPP evolved parallel sextic polynomial regression programs averaged across 30 runs for 64, 512 and 1024 fitness cases that lie in the range $[-1.5, 1.5]$.

<table>
<thead>
<tr>
<th>Fitness cases</th>
<th>Pop Size</th>
<th>Cores</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>100</td>
<td>482.91 [34.37]</td>
<td>601.77 [33.43]</td>
</tr>
<tr>
<td>200</td>
<td>4217.48 [32.72]</td>
<td>4147.83 [29.89]</td>
</tr>
</tbody>
</table>
execution time is slightly higher than that of 1-core. Chapman et al., [Chapman et al., 2007] noted that this is a common pattern in many OpenMP experiments that results due to thread scheduling. In this case, the OpenMP pragmas in the BNF grammar induce a slight overhead. This increases the grammar size, thereby the effective length of GE individuals increases, which of course, has no influence on the execution time (see section 5.5.2). Thus, the performance degradation on 2 cores is due to thread scheduling, apart from that, it has no influence on 4, 8 and 16 cores.

In fact, the execution time has reduced significantly over the sequential programs for 4, 8, and 16 core. Overall, the multi-core parallel programs report promising performance in terms of the execution time. The results show a speedup of up to 11 in the case of 11-multiplexer and, 9 in the case of sextic polynomial regression with 1024 fitness cases. Notice that the decrease in execution time is even bigger with the larger population sizes.

Although the performance of the programs is promising, it is necessary to estimate the efficiency of the results. Since the size of grammars increase, it is interesting to study its effect on the length of individuals.

Figure 5.4: An example of the best-of-run parallel sextic polynomial regression program.
5.5 Discussion

In this section we analyse both the efficiency of the evolved parallel programs and the impact of the multi-core grammars on the size of the genotypes. It is interesting to see the length of the genotypes as it is clear from the presented grammars that the size of the grammars is larger than their serial version for the respective problem. Note that the sequential grammars for these problems can be seen in [Azad and Ryan, 2005].

5.5.1 Efficiency

We analyze the efficiency (defined in Chapter 3) of the evolved parallel programs on both the problems. Also achieving an ideal efficiency for any parallelization strategy is difficult and highly unlikely due to operating system scalability issues [Boyd-Wickizer et al., 2010]. We observe that the evolved parallel programs scale effectively on most of the hardware configurations.

Figure 5.5: Efficiency of the evolved parallel programs for 11-multiplexer and sextic polynomial regression (1024 fitness cases).

Figure 5.5 presents the efficiency results for the two problems, that is, 11-multiplexer and the sextic polynomial regression with 1024 fitness cases. The results indicate that the evolved programs achieve significant speedup
on both the benchmark problems studied. As explained earlier, the efficiency of the evolved programs for 2 cores is less satisfying. When the number of cores increases from 4 to 8, the efficiency increases for the respective population sizes on both the problems.

However, the efficiency decreases as the population size increases for each number of cores. For example, for 8 cores of a 11-multiplexer, the efficiency is 0.91 at a population size of 25, which is 0.61 at a population size of 400. A similar phenomenon is observed in the case of sextic polynomial regression. There are a number of reasons that can be both OpenMP and evolutionary specific. First of all, recall that the total execution time is measured for all the evolved parallel programs in an evolutionary cycle across 30 runs, of which, majority of the parallel programs may not guarantee greater parallelism. Second, a higher number of threads on 16 cores come in one another’s way.

In accordance with this, the efficiency drops with an increase in the number of cores from 8 to 16. In fact, a more general expectation is that increasing the computational power will improve the efficiency of the evolving programs. However, the efficiency actually decreases for these two problems, thus, the system has a natural law of diminishing returns. It is because of the fact that the parallel programs are unable to scale after a certain number of cores, which is caused due to the inability of an operating system in scheduling the work load among the available threads. The current limitations of the system identified in this analysis are addressed later in Chapter 8.

### 5.5.2 Genome Lengths

We discuss the impact of GAPP grammar design on the genotype lengths of the evolving programs. It is obvious that the inclusion of OpenMP parallelization pragmas with the problem specific grammar increases the number of non-terminals over that of the sequential grammar. This in turn, affects the size of the evolving genotypes, most likely to increase their lengths due to an increase in the number of non-terminals.
*Figure 5.6:* A comparison of average actual and effective genome lengths of standard GE versus GAPP. The results are presented for 11-multiplexer and sextic polynomial regression (1024 fitness cases), where standard GE uses a single core processor while GAPP uses a 16 core processor. The actual lengths of both the approaches are significantly larger than the effective lengths.
Figure 5.6 compares the average actual and effective genome lengths of standard GE with that of GAPP. In the graphs, GE-actual, GAPP-actual and GE-effective, GAPP-effective stand for actual and effective lengths of GE and GAPP genotypes respectively. The graphs show that the actual lengths of both the approaches on both the problems are significantly larger than the effective lengths. Wilcoxon Signed Rank Sum statistical tests at a significance level of $\alpha = 5\%$ show their significance with a $p$-value of 0.005 and 0.007 between GE-actual versus GE-effective and GAPP-actual versus GAPP-effective respectively. It is expected since the entire genome is not used in the mapping process (see Chapter 2).

The actual lengths of GAPP genotypes (GAPP-actual) increase significantly over the actual lengths of GE (GE-actual) on both the problems at $\alpha = 5\%$ with a $p$-value of 0.03. Like the actual lengths, effective lengths of both the approaches on both the problems differ significantly at $\alpha = 5\%$ with a $p$-value of 0.04. That, in other words, GAPP-effective is larger than GE-effective on both the problems. This is an expected phenomenon. Since the inclusion of OpenMP parallelization pragmas increase the non-terminals in a grammar, this results in extra mapping steps; thus, the effective lengths of GAPP grow. Although the individuals are longer, the results in Table 5.1, Table 5.2 and Figure 5.5 show that it does not effect the performance of the evolving programs. In fact, the power of the multi-cores supersedes this slight increase in the length of genotypes.
5.6 Summary

In this chapter, we presented a novel method, GAPP, that uses Grammatical Evolution to automatically generate natively parallel programs. To do this, GAPP used the multi-core grammars that include OpenMP parallelization primitives. Thus, the evolving parallel programs followed a data parallel approach, that is, Single Program Multiple Data (SPMD). In this strategy, a single program operates on a different data item on each core of a multi-core processor.

GAPP has been tested on two standard GP benchmarks: multiplexer and sextic polynomial regression. The evolving parallel programs of both the problems significantly reduced the execution time over their sequential counterparts. Although this improvement was not uniform as the number of cores and population size grow. Thus, the parallel programs effectively exploited the modern multi-core architectures through simultaneous evaluation of different fitness cases. The efficiency of these programs seems promising and scales with an increase in the fitness cases.

The size of the population of individuals has increased with the introduction of the parallel pragmas in the problem specific BNF grammar. The increase in the size of the candidate solutions has negligible impact on the performance of the evolving programs. Nonetheless, we note these aspects later in Chapter 8 that present methods to mitigate these problems.

Although the efficiency of these programs is promising, the optimization in execution time has resulted from the parallelization or merely because of the infeasible solutions that finish their execution more quickly. Especially, it is because the selected benchmarks in this chapter are hard to evolve. Therefore, Chapter 6 experiments with a class of computable problems, those that can potentially evolve solutions.
Chapter 6

Proof of Concept

“Science replaces private prejudice with public, verifiable evidence.”

— Richard Dawkins, Ethologist

Chapter 5 introduced GAPP, which can synthesise parallel programs. That chapter detailed the application of GAPP on two GP benchmark regression problems and discussed a few potential pitfalls, which allows us to explore different problem domains. In this chapter, we employ GAPP on two different computable problem domains: recursion [Chennupati et al., 2015d] and iterative sorting [Chennupati et al., 2015e] to prove that the GAPP successfully synthesizes parallel programs in other domains.

6.1 Introduction

We describe the application of GAPP to two different problem domains: recursion and iterative sorting. The recursive problem involves the discovery of independently operating parallel recursive calls, while the iterative sorting problem involves generating parallel programs with loops that sort an array of integer elements. Writing recursive programs for fine-grained task
parallelism and sorting algorithms for data parallelism are excellent test-beds to exploit the multi-core processors, as they typically require skilled parallelization knowledge to fully realize the true potential of multi-cores.

We automate the evolution of *natively* parallel recursive and iterative programs through GAPP to utilise the computational potential of multi-cores. GAPP automates this process using GE with OpenMP `#pragma` compiler directives. Both the domains evolve suitable branch statements, while iterative sorting evolves loops also. We use these problems to illustrate that GAPP is indeed capable of generating parallel code.

We evaluate this approach on six recursive and four iterative sorting programs, all of which are to be evolved in C. For each benchmark, we describe the problem, grammar and keep the experimental settings consistent. We show that the generated programs can not only solve the problem, but that they do so in a parallel fashion.

We compare the fitness of different GAPP variants in each domain and present the performance (in terms of execution time) of the evolved parallel programs and show that it is significantly reduced when compared with the sequential counterparts. We now examine the proposed approach in each problem domain.

### 6.2 Recursion

Recursion is a process where a routine makes self referential calls. It breaks the problem into smaller sub-problems, a divide-and-conquer strategy. A recursive program has two types of cases: the *base case* and *recursive case*. In base case, the routine does not call itself, but the recursive case does it, whereas the base case terminates the recursion.

#### 6.2.1 Scope for Parallel Recursion

Consider an example recursive program, Fibonacci. It generates a sequence of integers, where the next integer of the sequence results through the addi-
tion of the previous two integers. Figure 6.1 shows the Fibonacci program and an example sequence. The procedure \textit{fib} terminates upon fulfilling the base case. The two independent recursive calls follow with input decreasing by 1 and 2 in the first and second call respectively.

```c
int fib(int n) {
    if(n <= 2)
        return n;
    else
        return fib(n-1) + fib(n-2);
}
```

**Figure 6.1:** Recursive Fibonacci program generates a sequence of integers.

**Figure 6.2:** Control flow structure of Fibonacci program recursive calls.

These recursive calls can be computed simultaneously. For example, Figure 6.2 shows the control flow of the Fibonacci recursive calls. The recursive calls: call 1 and call 2 at level 4 of the execution trace can be executed simultaneously. That allows to execute the two calls in parallel. The
parallelism exploited can be fine-grained, where both the calls are computed before the final addition happens. Since the two calls generate different execution traces, their concurrent execution is task level parallelism. The challenge then is to automatically discover such parallelism through GAPP. First, we review the evolution of recursion regardless of parallelism.

### 6.2.2 Evolutionary Approaches for Recursion

Initially, Koza [Koza, 1992] introduced of a sequence reference function, \( SRF K D \) for Fibonacci sequence generation using GP. The function referenced the previously evaluated values from a table and returned the \( K^{th} \) value if available; otherwise, it returned the default value \( D \). This approach successfully evolved a Fibonacci sequence averting the needless recursive calls through referencing the pre-calculated values. The first 20 values of the sequence were used as input fitness cases. He used raw fitness measure, that is, the absolute difference between the evolved and the target outputs.

Whigham and McKay [Whigham and McKay, 1995] used tree based GP to learn recursive functions that take as input an element of a list and its position. The function returned true if the element was found in the list, and NIL (false) otherwise. However, they concluded that evolving recursion was inappropriate for automated learning because of infinite calls.

Brave [Brave, 1996] explored tree search on a simple planning problem through a restrictive form of recursion using Automatically Defined Functions (ADFs). To prevent excessive recursive calls, the ADFs used were only allowed as many recursive calls as the tree depth. He investigated the scalability of GP programs with and without recursion and ADFs and showed that the required effort to evolve a solution with ADFs and recursion remain constant irrespective of the size of the problem.

Huelsbergen [Huelsbergen, 1997] evolved programs that generate recursive sequences. He used the arithmetic operations such as addition, subtraction and multiplication without including an explicit recursion operator.

In recursion, due to code reuse, a new copy of the program code is created within the program. This leads to compact programs, which facili-
Realising that, Wong and Leung [Wong and Leung, 1996] used Generic Genetic Programming (GGP) for the evolution of recursive functions for even-n-parity problem. They identified that GGP accelerated the learning speed by including domain specific knowledge and improved the quality of induced programs. Later, in an attempt to discourage infinite recursion, Wong and Mun [Wong and Mun, 2005] used an adaptive grammar based GP by adjusting the weights associated with the production rules of the grammar. This approach increased the probability of success and decreased the number of infinite-recursive programs.

Yu and Clark [Yu and Clark, 1998] evolved modular recursive programs. They encountered difficulty with non-terminating calls, which was treated with the inclusion of implicit recursion, that resulted in performance gains in GP. This approach introduced a higher order function that took two arguments, a binary operator and a list of values; the operator then is placed in between successive pairs of items of the list and evaluated from left to right.

Koza et al., [Koza et al., 1999] proposed Automatically Defined Recursion (ADR). ADRs contained a recursion condition branch (RCB), a recursion body branch (RBB), a recursion update branch (RUB) and a recursion ground branch (RGB). An ADR first executes RCB. The recursion continues to execute RBB, which can trigger the recursive calls, after this, RUB is executed. When RCB terminates the recursion, the RGB is executed; otherwise, RBB is executed. Since the four branches change during the evolution, the use of an ADR separates the terminating condition and recursion. Similarly, Automatically Defined Nodes (ADN) [Shirakawa and Nagao, 2009] evolved recursive programs using graph structured program evolution.

Yu [Yu, 2001], [Yu, 2005] evolved even-parity recursive programs through hierarchical processing of higher order abstractions. A special structure abstraction designates the promising area of the search space in the initial generation. The abstraction hierarchically processes the designated area to evolve efficient even-parity programs. However, the efficiency of the results relied on the appropriateness of the higher-order function.

Woodward [Woodward, 2003], [Woodward, 2004] showed that the complexity of an evolving function is independent of the GP primitive set given
that it is expressive and the representation expresses the modules such as ADFs. In an extension to that, two primitive recursive function (PRF) representations were examined [Woodward, 2006]. He showed that the complexity of a PRF is independent of the representation and depends only on the target function. That is, the amount of memory required for a PRF is independent of the computational model used in expressing it, and is analogous to Kolmogorov complexity [Li and Vitnyi, 2008].


More recently, Moraglio et al., [Moraglio et al., 2012] presented a general non-recursive scaffolding method that evolved a recursive list reversal program using a CFG based GP. This approach evolved non-recursive programs, followed by converting the optimum non-recursive program to an associated optimum recursive program.

### 6.3 GAPP Grammars for Recursion

This section discusses the design of multi-core grammars in evolving the parallel recursive programs. Unlike the data parallelism (see Chapter 5) of regression programs, these recursive programs exhibit fine-grained task parallelism. That is, different sub-tasks execute simultaneously in parallel, where each core is occupied by a sub-task of the recursive execution.

As explained in Chapter 5, GAPP does not make any changes to the design of GE. Instead, it relies on grammars so designed as to embody the knowledge that allows GE to produce parallel recursive programs. The grammars allow GE to select from various OpenMP pragma directives. Unlike PARAGEN-II [Ryan, 1999], discussed in Chapter 3, GAPP does not
utilise dependency analysis; instead, GE works the data interdependencies out by selecting pragmas that suite program correctness.

```plaintext
<omppragma> ::= #pragma omp parallel for <newline> {<parcode> | 
#pragma omp parallel <newline> {<parcode> | 
#pragma omp parallel sections<newline> {<newline> <parblocks>
<parcode> ::= if( <var> <lop> <const> ) {<newline> 
    int a = <expr>; res<bop>=a; <newline> } else {<newline> int 
    a = <expr>; <newline> res<bop>=a; } <newline> <newline> <result> <newline> <result> ::= return <var>;
<expr> ::= <var> | <stmt> | <stmt><bop><stmt>
<stmt> ::= fib(<var> <bop> <const>)
<bop> ::= + | - | * | /
<lop> ::= >= | <= | > | < | ==
<const> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<var> ::= n | res <newline> ::= \n
Figure 6.3: GAPP grammar to evolve parallel recursive Fibonacci programs.
```

Figure 6.3 presents one such grammar for the Fibonacci program described in section 6.2. The non-terminal `<omppragma>` contains different OpenMP parallelization pragmas that execute the recursive calls in parallel. The non-terminal `<parcode>` generates the code that can be executed in a parallel region, which is created with `#pragma omp parallel`. Similarly, the non-terminal `<parblocks>` produces the parallel code blocks that are
produced through <blocks>. The parallel code blocks ensure task level parallel execution. Any choice of the non-terminal <omppragma> ensures that the recursive calls they enclose execute in parallel, but, the last choice (i.e., #pragma parallel sections) particularly suites task level parallelism. The syntax of this pragma requires the use of a special pragma (#pragma omp section) that designates the blocks of code to execute as separate tasks, of course, it is down to evolution to select this directive preferentially.

The non-terminal <stmt> refers to the Fibonacci recursive call. The symbols <bop> and <lop> refer to the binary arithmetic and logical operators respectively. The non-terminal <var> maps to an input (\(n\)) or an output variable (\(res\)). The <const> maps the space of integer constants. The base case is generated from the input variable, logical operators and constants generating non-terminals. The recursive calls are expressed through the non-terminal <expr>, which is called in <parcode> and <blocks>.

The evolving parallel programs of this design follow SPMD parallelization strategy in the form of executing each recursive call, which operates on a different input value. We evolve programs in C; however, GAPP is general enough to apply to the programming languages that offer OpenMP like parallelization. For example, JOMP [Bull and Kambites, 2000] is an OpenMP API for JAVA, hence can synthesize parallel programs in JAVA.

6.4 Experiments

This section describes the experimental setup and the evaluation procedure.

6.4.1 Experimental Setup

We use the experimental settings shown in Table 6.1, where population of individuals are initialised using Sensible Initialisation [Ryan and Azad, 2003], detailed in Chapter 2. The minimum and maximum derivation tree depths are 15 and 25 respectively. We use one-point crossover with a probability of 0.9, and a per bit mutation with a probability of 0.01. The exper-
iment are conducted on a 16-core Intel (R) Xeon (R) E7-4820 processor. Table 6.2 shows the hardware and software settings of the experiments.

Table 6.1: Experimental parameter settings

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initialization</td>
<td>Sensible</td>
</tr>
<tr>
<td>Crossover and Mutation</td>
<td>{0.9 and 0.01}</td>
</tr>
<tr>
<td>Selection and Replacement</td>
<td>{Roulette Wheel and Steady State}</td>
</tr>
<tr>
<td>Minimum and maximum depth</td>
<td>{15 and 25}</td>
</tr>
<tr>
<td>Wrapping</td>
<td>disabled</td>
</tr>
<tr>
<td>Population, Generations and Runs</td>
<td>{500, 100 and 50}</td>
</tr>
</tbody>
</table>

Table 6.2: Experimental hardware and software settings.

<table>
<thead>
<tr>
<th>Hardware/Software</th>
<th>Requirements</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>Intel (R) Xeon (R) E7-4820, 16 cores</td>
</tr>
<tr>
<td>OS</td>
<td>Debian Linux v 2.6.32, 64-bit</td>
</tr>
<tr>
<td>C++</td>
<td>GNU GCC v 4.4.5, libGE v 0.26</td>
</tr>
<tr>
<td>OpenMP and Time utility</td>
<td>libgomp v 3.0 and omp_get_wtime()</td>
</tr>
</tbody>
</table>

We calculate the normalized mean absolute difference between the target and evolved outputs (eq. 6.1) that lies in between 0 and 1.

\[
f_{prog} = \frac{1}{1 + \frac{1}{N} \sum_{i=1}^{N} |y_i - \hat{y}_i|} \quad (6.1)
\]

where, \(N\) is the number of fitness cases; the terms \(y_i\) and \(\hat{y}_i\) represent the target and the evolved outputs respectively.
6.4.2 Experimental Problems

We examine our approach on six standard benchmark recursive problems shown in Table 6.3. The solutions for these problems require several interesting features of a parallel recursive solution such as branching, recursive calls, and temporary, thread specific shared and private variables. The results also indicate that the difficulty of the problems varies.

**Sum-of-N** [Albarghouthi et al., 2013] returns the sum of all whole numbers less than or equal to an input number $N$.

**Factorial** [Cormen et al., 2009] is a single input function that returns the product of all the whole numbers from 1 to $N$, a given input number.

Table 6.3: The problems used in the experiments and their properties. Input and Return represent the data type of the input arguments and return results respectively. Local Variables (LV) are the temporary variables while Range is the bounds from which the input is drawn randomly.

<table>
<thead>
<tr>
<th>#</th>
<th>Problem</th>
<th>Type</th>
<th>LV</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Input</td>
<td>Return</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Sum-of-N</td>
<td>int</td>
<td>int</td>
<td>[1,1000]</td>
</tr>
<tr>
<td>2</td>
<td>Factorial</td>
<td>int</td>
<td>unsigned long long</td>
<td>3 [1,60]</td>
</tr>
<tr>
<td>3</td>
<td>Fibonacci</td>
<td>int</td>
<td>unsigned long long</td>
<td>3 [1,60]</td>
</tr>
<tr>
<td>4</td>
<td>Binary-Sum</td>
<td>int [], int, int</td>
<td>int</td>
<td>2 [1,1000]</td>
</tr>
<tr>
<td>5</td>
<td>Reverse</td>
<td>int [], int, int</td>
<td>void</td>
<td>2 [1,1000]</td>
</tr>
<tr>
<td>6</td>
<td>Quicksort</td>
<td>int [], int, int</td>
<td>void</td>
<td>3 [1,1000]</td>
</tr>
</tbody>
</table>
Fibonacci [Koza, 1992] is a sequence of numbers, where the next number is the sum of the two immediately preceding numbers.

Binary-Sum [Blelloch, 1996] adds the adjacent elements of an array or a list, before returning the result of sum.

Reverse [Moraglio et al., 2012] orders a list in opposite direction.

Quicksort [Kruse, 1989] is a classical sorting algorithm that arranges an array of elements in an ascending or descending order.

For the problems that operate over an array, we generate 1000 random integers that are in the range 0 to 1000, while for the problems that operate on a single integer input, we randomly select a value from the range 1 to 60 in each run; this range is large enough to expand the execution trace. For example, to compute the Fibonacci sequence for 40 we have to get the sequence right for the smaller inputs as well. Figure 6.3 presents the BNF grammar for Fibonacci while figures B.2 – B.6, given in Appendix B, present the grammars for the remaining five problems. Note that these grammars are consistent across all the problems except for a few minor changes with respect to the problem. These changes include the number of arguments, input variables, use of temporary variables, etc.

6.4.3 Terminating Recursion

As explained in section 6.2.2 preventing infinite recursion is crucial in evolving recursive programs. To this end, we handle the infinite recursion by setting the exponential of the randomly selected input size as the limiting criterion of a respective problem. If a program exhausts this quota, the evolved function simply returns the input value and terminates; otherwise, the function returns the computed value. Although, the actual limit is an ad hoc choice, the grammars explicitly account for evolving a base case. In order to evolve a correct parallel program that solves a given problem, we investigate three different approaches in formulating the base case.
The first approach, referred to as \textit{const-10} henceforth, allows the condition as well as the recursive calls require a constant with any one of the 10 constants (\texttt{<const>}) ranging between 0 and 9 (both inclusive).

\textbf{cond} The second approach, termed as \textit{cond} evolves the terminating condition so that it always compares with 1, that is \texttt{<condition> := if(<var> <lop> 1)}. Thus, only one constant is available to compare against the terminating condition; however, the recursive calls are allowed to choose from all the 10 available constants in \texttt{<const>}.  

\textbf{const-limit} Finally, the third approach is termed as \textit{const-limit}, where the constants (\texttt{<const>}) range is reduced to (1, 2). Both the base case and the recursive calls can only choose one of the two available constants. However, the last two approaches incorporate problem specific knowledge into the grammar to facilitate evolution.

\subsection*{6.4.4 Results}

We report two key statistics in this section: the mean best fitness and the mean of the total execution time for all the best of generation programs.

\textbf{Mean Best Fitness}

Figure 6.4 compares the mean best fitness (MBF) for the 6 recursive benchmarks; the results are averaged across 50 runs at each generation for all the three variations. The Wilcoxon Signed Rank Sum test at $\alpha = 0.05$ shows insignificant difference between \textit{const-10} and \textit{cond}.

MBF of \textit{const-limit} proved to be \textit{significantly} different from the rest at $\alpha = 0.05$. In fact, \textit{const-limit} solved all the problems. Obviously, the limited constant space used for generating the base case solved the problem.

The results indicate that except for \textit{Sum-of-N}, the problems tackled here are not easy: even with \textit{const-limit} it takes at least 40 generations to find a
Figure 6.4: Mean best fitness (with standard deviation) of GAPP evolved recursive programs that is averaged across 50 runs of 100 generations. The results are for \textit{const-10}, \textit{cond} and \textit{const-limit}. 
correct solution. Although syntactically an almost-ideal solution can be generated quite early, finding the exact solution can take more number of generations. That is, GAPP finds a syntactically correct parallel program early in the evolution. However, finding a right parallel program takes slightly more number of generations. A detailed discussion is in section 6.5.

Table 6.4: Mean best execution time (MBT [standard deviation]) of GAPP at different number of cores of an Intel Xeon processor.

<table>
<thead>
<tr>
<th>Problem</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>2481.5 [37.09]</td>
<td>2988.1 [22.31]</td>
<td>781.2 [26.32]</td>
<td>724.2 [29.74]</td>
<td>487.7 [35.61]</td>
</tr>
<tr>
<td>3</td>
<td>3799.3 [41.14]</td>
<td>2821.3 [37.32]</td>
<td>1409.5 [34.11]</td>
<td>921.4 [21.92]</td>
<td>608.2 [49.17]</td>
</tr>
<tr>
<td>4</td>
<td>3336.2 [67.76]</td>
<td>2134.9 [29.16]</td>
<td>1683.8 [29.22]</td>
<td>738.9 [37.32]</td>
<td>482.5 [27.46]</td>
</tr>
<tr>
<td>5</td>
<td>3222.7 [64.76]</td>
<td>3348.6 [38.41]</td>
<td>1035.8 [43.36]</td>
<td>596.6 [35.17]</td>
<td>520.1 [63.19]</td>
</tr>
<tr>
<td>6</td>
<td>4644.8 [29.04]</td>
<td>2578.2 [27.11]</td>
<td>1540.1 [63.69]</td>
<td>705.8 [44.51]</td>
<td>577.2 [41.28]</td>
</tr>
</tbody>
</table>

Mean Best Execution Time

We report the mean best execution time (MBT) of the evolving programs. That is the total time required to execute all the best of generation programs produced in a run; we report the mean of this time.

We report the impact of parallelism on execution time across an increasing number of cores. Table 6.4 presents the mean best execution time of the six experimental problems that are averaged across 50 runs with each run containing 100 generations for the number of cores from 2 to 16.

Overall, with the exception of when 2 cores are used, the execution time of the parallel programs reduces significantly as the number of cores in-
crease at $\alpha = 0.05$. Note, the decrease in the execution time of the evolving parallel recursive programs is merely due to the presence of OpenMP parallelization primitives. However, the execution time does not show promising improvement on 2 cores due to the scheduling issues discussed in Chapter 5.

### 6.5 Discussion

This section focuses on the quality of the evolved parallel recursive programs. In the case of *const-limit*, the best evolved Fibonacci program at generation 17 is shown in Figure 6.5 has a fitness value of 0.36, and at generation 98, we find the program shown in Figure 6.6 has a fitness of 1.

```c
#pragma omp parallel
{
    if(n > 2) {
        int a = n; res += a;
    }
    else {
        int a = fib(n-2) * fib(n-1); res += a;
    }
} return res;
```

**Figure 6.5:** An example evolved parallel recursive *Fibonacci* program that uses OpenMP *parallel* work-sharing construct.

From Figure 6.5 to Figure 6.6, the logical operator $>$ changed to $\leq$ in the base condition and, the binary operator $*$ changed to $+$, and now sums the result of recursive calls. Moreover, the parallelism exerting pragmas are also different: the pragma (*parallel*) in the first program creates multiple threads and executes the program code that it encloses in parallel.

Although the program in Figure 6.5 is not an optimal as it multiplies (*) the two recursive calls and fails to execute the two calls independently. The program in Figure 6.6 uses a correct pragma, and executes the tasks
in separate threads; these tasks are enclosed with the pragma `section`. The shared variable `res` is updated atomically, thus prevents race conditions and perfectly achieves task parallelism. These changes are not a simultaneous happening rather they occur much later in the evolutionary cycle as the run progresses. We now discuss GAPP for the evolution of iterative sorts.

```c
#pragma omp parallel sections
{
    if(n <= 2) {
        #pragma omp section
        {
            int a=n; #pragma omp atomic res+=a;
        }
    } else {
        #pragma omp section
        {
            int a=fib(n-1); #pragma omp atomic res+=a;
        }
        #pragma omp section
        {
            int a=fib(n-2); #pragma omp atomic res+=a;
        }
    }
} } return res;
```

**Figure 6.6:** An example best evolved task parallel recursive Fibonacci program that uses OpenMP `sections` work-sharing construct.

### 6.6 Iterative Sorting

Sorting reorganizes a sequence in a logical (ascending/descending) order. It is crucial in scientific and commercial applications that include weather prediction, finance, internet, etc. and is good candidate for parallelization.
To that end, the challenge then is to exploit that parallelism using GAPP. First, we review the existing work in the evolution of sorting algorithms.

### 6.6.1 Evolutionary Approaches for Sorting

Although we aim to generate parallel code for non-recursive sorting, in this section, we review both recursive and non-recursive sorting attempts regardless of parallelization. Hillis [Hillis, 1990] applied simulated evolution on sorting networks through the co-evolution of test sequences. It effectively evolved a minimal 16-input network and improved the performance.

O’Reilly and Oppacher [O’Reilly and Oppacher, 1992] attempted to evolve an iterative sorting algorithm using GP and failed. They used variable decrementing, element indexing, loops as primitive functions while the array size, constant 1 and, loop variables as terminals. However, they later [O’Reilly and Oppacher, 1996] succeeded with the use of a swapping primitive function that exchanges the elements.

Kinnear [Kinnear, 1993a], [Kinnear, 1993b] successfully generated a bubble sort by swapping the disordered adjacent elements, similar to the compare operator [Hillis, 1990] used in sorting networks.

Abbott [Abbott and Parviz, 2003] applied Object Oriented Genetic Programming (OOGP) for an integer sorting in Java. It included Java’s built-in and the user defined methods to generate both insertion and bubble sort. However, the evolution of bubble sort through this approach was unclear.

Agapitos and Lucas [Agapitos and Lucas, 2006], [Agapitos and Lucas, 2007] also applied OOGP to evolve recursive sorting. List processing methods and a higher order primitive function (*filter*) were used in [Agapitos and Lucas, 2006], and different fitness measures were analysed. Later, evolution of modular recursive sorting [Agapitos and Lucas, 2007] showed that OOGP generated better programs than the undirected random search.

A recent attempt by O’Neill et al., [O’Neill et al., 2014] presented an application of GE for program synthesis through the evolution of bubble sort in Python, where a series of loops and nested loops with swapping successfully solved the integer sorting problem.
Figure 6.7: GAPP grammar that produces a parallel iterative Quicksort program. It contains various OpenMP parallelization primitives. The grammar allows the evolution of loops, branches, array indexes, etc.
6.7 GAPP Grammars for Iterative Sorting

Figure 6.7 presents the GAPP grammar in evolving the parallel iterative Quicksort programs. The resultant sorts follow the data parallelism (see Chapter 5). The key difference is the use of richer grammars that include multiple OpenMP pragmas, iterative instructions, nested loops as in [O’Neill et al., 2014], branches and array indexes; these additions improve the generalizability of the system. Nevertheless, it produces high quality results.

Parallelism

In Figure 6.7, the non-terminal <omppragma> contains different alternatives for pragmas, from which, GAPP selects an apt choice that executes the code it encircles in parallel. The pragmas parallel for and parallel offer efficient parallelism, but their selection is the prerogative of evolution. The manner in which these pragmas operate depends on the shared and private variables, where the symbols <shared> and <private> choose them respectively.

The shared variables are input array (A) and the auxiliary stack (S); A contains integers, S fixes the order of those integers. Similarly, the symbol, <index> refers to a thread private variable, an index of A or S. For example, A is shared among the OpenMP processes to execute simultaneously while its indices are made private to ensure correctness of computations. Thus, the same program operates on multiple data, an SPMD parallel sorting.

Problem Specific Knowledge

The Quicksort problem specific grammars include nested loops, branches, exchanging the array elements, and array indexing. We keep the loop structures and initializations systematic in order to prevent both infinite loops and loop entry failures. The symbol <forstack> is the outer loop that pops the stack till it becomes empty while <partition> partitions the elements of an array. The non-terminal <code> shows the OpenMP pragmas.

The non-terminals <partcond> and <partitionele> show the functionality of partition and how to fix the pivot respectively. The prime reasons
to include such a problem specific information is due to the lessons learnt from literature. That is, the evolution of a simple bubble sort included the adjacent element swap [Kinnear, 1993b] after observing the failed attempts in [O’Reilly and Oppacher, 1992]. Similarly, evolution of loops [O’Neill et al., 2014] failed without swapping. Our initial results showed a similar behaviour without partitioning, hence such domain knowledge is needed.

The elements of an array are swapped (<swap>) while <aelement> refers to an absolute index of an array which prevents the out of index memory accesses. The constants (<const>) are in the range 0...9. The logical and binary arithmetic operators are in <lop> and <bop> respectively. The symbol <pushpop> represents the push and pop over the auxiliary stack.

**Generality of the Grammars**

The grammars for the other benchmarks are 90% similar, where all of them have common OpenMP pragmas while they differ in some domain specific knowledge. Converting the outer loop (over a stack) to iterate over an input array and excluding the partition functionality from Quicksort grammar makes it general enough for other iterative sorts. The grammars then resemble the ones in [O’Neill et al., 2014], here, we do not use an output array; instead we use an auxiliary stack (S) only for Quicksort. As in section 6.3, we can generalize to other languages that support OpenMP like parallelism. However, future research can leverage ways for more general grammars.

**6.8 Experiments**

This section describes the experimental setup, parameters, and the results.

**6.8.1 Experimental Setup**

We use the experimental parameter settings shown in Table 6.1 and the experimental environment presented in Table 6.2.
We compare two different approaches in terms of fitness of the GAPP evolving parallel programs. Of the two: the first approach evolves the programs using the grammars described in section 6.7 and referred to as *const-10* hereafter, while the second approach reduces the constants (<const>) from (0,...,9) to just (0,1) and referred to as *const-limit* hereafter.

**Table 6.5**: The problem set considered, input type and the local variables (LV).

<table>
<thead>
<tr>
<th>#</th>
<th>Problem</th>
<th>Input</th>
<th>LV</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Bubble sort</td>
<td>int [ ], int</td>
<td>4</td>
<td>[1:1000]</td>
</tr>
<tr>
<td>2</td>
<td>Quicksort</td>
<td>int [ ], int, int</td>
<td>5</td>
<td>[1:1000]</td>
</tr>
<tr>
<td>3</td>
<td>Odd-Even sort</td>
<td>int[ ], int</td>
<td>4</td>
<td>[1:1000]</td>
</tr>
<tr>
<td>4</td>
<td>Rank sort</td>
<td>int[ ], int</td>
<td>4</td>
<td>[1:1000]</td>
</tr>
</tbody>
</table>

### 6.8.2 Experimental Problems

A total of four problems were used (including Quicksort) that are summarized in Table 6.5. The problems contain a mixture of structures that include: loops, nesting, loops over arrays, branching, use of temporary variables, arrays, shared and/or private (OpenMP specific) variables.

**Bubble sort**  [Knuth, 1998] loops through an array of elements compares and swaps the adjacent elements in ascending order.

**Quicksort**  [Bailey et al., 1991] partitions an array over a pivotal element, and then, rearranges the smaller elements to the left and the bigger to the right of the pivot.

**Odd-Even sort**  [Knuth, 1998] operates in two phases, *Odd* and *Even*. In both the phases, the elements exchange with that of their right.
**Rank sort**  [Wilkinson and Allen, 1999] orders the array with the rank (number of elements less than the selected) of an element.

The grammars for all the four problems are relatively similar to each other except for Quicksort. It uses slightly more local variables (5) over the other problems as it contains a stack and partition that use 4 extra local variables (top, start, last, pivot) specified in the symbol, `<index>`. Figure 6.7 shows the *Quicksort* grammar, while Figure B.7–B.9 represent the grammars of the remaining three benchmarks. As in section 6.4, the fitness is normalized in between 0 and 1.

![Graphs showing mean best fitness for various sorting algorithms](image)

**Figure 6.8:** The mean best fitness (with standard deviation) of GAPP evolved parallel iterative sorting programs that is averaged across 50 runs of 100 generations, where the graphs represent the results of *const-10*, and *const-limit*.
6.8.3 Results

This section reports the two key statistics: mean best fitness and mean best execution time of the GAPP evolving parallel iterative sorting programs. These two measures are defined in section 6.4.4.

Mean Best Fitness

Figure 6.8 compares the mean best fitness (MBF) of two variations, (const-10 and const-limit) on all the four benchmarks. On these problems, the Wilcoxon Signed Rank Sum tests show a significant difference in the MBF between both the variants at $\alpha = 0.05$. Here, the const-limit variation exhibits a significant improvement in the quality of the solution.

However, const-10 variant fails to sort the sequence in all the four problems, while const-limit has solved all the problems in all the runs. This is because of the limited search space. The results indicate that the problems are not easy to solve. For example, Quicksort, even with const-limit, requires 47 generations to produce optimal solution. Although syntactically ideal solution is generated early, finding an optimal solution takes longer.

Mean Best Execution Time

As explained in section 6.4.4, the purpose of the evolved parallel programs is to exploit the performance boost offered by multi-cores. Therefore, we report mean best execution time (MBT) for the evolving programs.

Table 6.6 shows the mean best execution time of the evolved parallel iterative sorting programs of GAPP that are compared with a varying number of cores from 2 to 16 on an Intel Xeon multi-core processor. The results demonstrate that the execution time has been reduced significantly with an increase in the number of cores as confirmed by the Wilcoxon Signed Rank Sum test at $\alpha = 0.05$. However, the evolved programs that contain a parallel sections directive reported an incorrect solution while the programs that contain parallel or parallel for produced correct results. This is due to the fact that the programs with the former pragma creates a parallel region.
Table 6.6: Mean best execution time (MBT [standard deviation]) of four problems averaged across 50 runs of GAPP for 100 generations.

<table>
<thead>
<tr>
<th>Problem</th>
<th>Cores</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td></td>
<td>3197.3 [24.3]</td>
<td>2396.5 [22.1]</td>
<td>1146.2 [26.4]</td>
<td>435.3 [26.3]</td>
<td>257.2 [23.2]</td>
</tr>
</tbody>
</table>

with multiple threads where the work-sharing construct (sections) tries to identify independently executable functions. Hence, the programs with this choice are treated as the worst fit individuals. While the programs with the parallel for pragmas create multiple threads and identify an iterative work-sharing construct that executes the assigned task in parallel. In this case, the work in the for loop is scheduled across all the available cores that ensures faster execution of the programs.

### 6.9 Discussion

As with section 6.4.4, the execution time of the evolving sorting programs is insignificant for 2 cores. This is due to the extra overhead added through the OpenMP scheduling issues, which are detailed later in Chapter 8.
6.10 Summary

In summary, we showed the evolution of parallel programs that are of both recursive and iterative nature. The presented results explored various possibilities of evolving parallel programs while also maintained solution correctness; this was a dual challenge, a challenge that an EC approach tackled successfully given the problem suite. We noticed that the attempts to prevent infinite recursion, although ad hoc, did not prevent evolution from exploring the high quality solution space. This technique obviates the need for the dependency analysis in native parallel code generation. We also presented the computational effort exploited by the evolved parallel programs on multi-cores.

In Chapter 7, we use evolution to promote parallelism: in this chapter we did not prefer individuals that exploited maximum parallelism. We modify GAPP in order to evolve programs that exhibit higher degree of parallelism.
Part III

Optimizing the Performance of Programs
Chapter 7

Performance Optimization of GAPP

“If you’re not getting better you’re getting worse.”

— Joe Paterno, Football coach

GAPP is a system that natively produces parallel code. This chapter extends GAPP by including practical constraints in the grammars and fitness functions, such as increased control over the level of parallelism for each program. These changes execute the best-of-generation programs faster than the original GAPP generated parallel programs. We use evolution to optimize the performance of these parallel programs in terms of their execution time, and our results demonstrate [Chennupati et al., 2015a] [Chennpati et al., 2015] [Chennupati et al., 2015c] a significant performance optimization.

This chapter is organised as follows. Sections 7.2 and 7.3 describe and examine the proposed extensions in evolving the optimal parallel recursive programs respectively. Sections 7.4 and 7.5 extends and experiments in evolving the optimal parallel iterative sorting programs. Section 7.6 discusses the effect of proposed extensions, and finally, section 7.7 concludes.
7.1 Introduction

Although GAPP is a powerful tool for synthesising native parallel programs, generating \textit{optimal} parallel programs remains a non-trivial challenge. The minimal execution time of the parallel programs in Chapter 6 was essentially due to the presence of OpenMP pragmas which automatically map threads to cores. However, the use of a different OpenMP pragma would alter the performance of the parallel program, and skilled parallel programmers carefully choose the pragmas when writing code. It is harder to achieve optimal parallelism while obeying all these parallel design considerations such as preferentially select an apt OpenMP pragma.

To alleviate this difficulty, we extend GAPP in two ways: first, we restructure the grammars so \textit{task} and \textit{data parallelism} is separate, and second, we explicitly penalize long execution times. We re-design the GAPP grammars such that they now partition task and data parallelism under different production rules, making it convenient for evolution to select as appropriate. These design changes offer greater flexibility and ease in evolving the parallel programs. Furthermore, we modify the fitness function to explicitly take account of the execution time of the individuals.

We examine the proposed modification on six recursive and four iterative sorting benchmarks described in Chapter 6. These changes combine to give an average speed-up factor of 8.13 on recursive problems and 11.03 on iterative sorting problems. The following section demonstrates the proposed extensions on recursion.

7.2 Recursion

We describe the extensions of GAPP: first, the design of grammars (section 7.2.1), which are much richer and categorize rules better so as to accelerate evolution; second, through a modified fitness function (section 7.2.2), which considers time in its evaluation. Both of these help optimize the performance (execution time) of the evolving parallel recursive programs.
The programs take an integer input (<input>), while the variable res returns the result of the parallel evaluation. Moreover, the two local variables
(temp, a) store the intermediate results of recursive calls. The input and the two variables (temp, res) are shared (shared) among the threads, while “a” is a thread private (private) variable. Evolution selects a private clause from the three OpenMP private (private) clauses.

Of the three private clauses: private(a) makes a variable thread-specific, such that any changes on the variable are invisible after the parallel region; firstprivate(a) maintains a constant value throughout the program; lastprivate(a) keeps the changes of the last thread in the parallel region. These three private clauses are better explained in Chapter 3. Since the variable updates are thread specific, programs with private(a) are the best programs. Note, the other clauses degrade the fitness as they evolve incorrect solutions.

We now discuss the criteria for the selection of OpenMP primitives.

### 7.2.2 Performance Optimization

We encourage parallelism with the inclusion of execution time in fitness. It helps to select an appropriate pragma. Thus, the fitness function is the product of two factors: execution time and the mean absolute error, both are normalized in the range [0, 1] – a maximization function. The following equation represents the fitness of the evolved program \( f_{p prog} \):

\[
f_{p prog} = \frac{1}{(1+t)} \cdot \frac{1}{\left(1 + \frac{1}{N} \sum_{i=1}^{N} |y_i - \hat{y}_i| \right)}
\]  \hspace{1cm} (7.1)

where, \( t \) is the execution time of a program over \( N \) fitness cases; \( y_i \) and \( \hat{y}_i \) are the actual and evolved outputs respectively. The choice of an OpenMP pragma can significantly impact the execution time of a program. Hence, the first term, normalized execution time in eq. 7.1 helps to select the correct pragma. That is, the changes in the time component influence the performance of the resultant programs, whereby, those with minimum execution time become the best parallel programs. Meanwhile, the second term, normalized mean absolute error enforces program correctness. Together, the two objectives push for a correct and efficient parallel program.
### 7.3 Experiments

We reuse the six recursive benchmarks and the parameters in Chapter 6.

#### 7.3.1 GAPP Variants

The enhancements in section 7.2 influence the evolvability and fitness evaluation of the parallel programs, we study that with four GAPP variations. Of those four variations: the first variant, named as GAPP (Unoptimized) hereafter, does not separate task and data parallelism; instead, all the rules in <omptask> and <ompdata> are referred through <omppragma>. It is difficult to omit data parallelism when it only requires task parallelism, while the fitness function is only the normalized mean absolute error. The second variant, GAPP (Grammar), uses the grammars shown in Figure 7.1, while the fitness function is the normalized mean absolute error. The third variant, GAPP (Time), uses the same grammars as with the first variant (GAPP (Unoptimized)), but \( f_{pprog} \) (eq. 7.1) for fitness evaluation. Finally, the fourth variant, named as GAPP (Combined) hereafter, uses both the grammars as in Figure 7.1 and the fitness function \( f_{pprog} \), that is the case when both the enhancements proposed in section 7.2 work together.

#### 7.3.2 Results

We report two measures: speed-up and mean best generation (MBG).

**Speed-up**

The speed is the ratio of mean best execution time (MBT) of evolving programs on 1-core to \( n \)-cores. Eq 7.2 shows the speed-up:

\[
\text{Speed-up} = \frac{T_{MBT-1-core}}{T_{MBT-n-cores}} \quad (7.2)
\]

where, \( T_{MBT-1-core} \) is the mean best execution time on a single core, while \( T_{MBT-n-cores} \) is that of \( n \)-cores of a processor.
Figure 7.2: The speed-up of GAPP (Unoptimized, Grammar, Time, Combined) variants for all the six experimental problems. The number of cores vary as 2, 4, 8 and 16. The horizontal dashed line (- -) represents the speed-up of 1 and acts as a reference for the remaining results.
Figure 7.2 presents the speed-up of each of the four GAPP (Unoptimized, Grammar, Time, Combined) variants at different cores for all the six recursive benchmarks. The results indicate that the performance of GAPP improves as the number of cores increase. The non-parametric Friedman tests [Demšar, 2006] justify the significance of these results.

Table 7.1: Friedman statistical tests with Hommel’s post-hoc analysis on speed-up of all the four GAPP variants. The boldface shows the significance at $\alpha = 0.05$, while asterisk (*) shows the best variant.

<table>
<thead>
<tr>
<th>Cores</th>
<th>GAPP variant</th>
<th>Average Rank</th>
<th>p-value</th>
<th>p-Hommel</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>Unoptimized</td>
<td>3.25</td>
<td>0.0025</td>
<td>0.0167</td>
</tr>
<tr>
<td></td>
<td>Grammar</td>
<td>3.1667</td>
<td>0.0036</td>
<td>0.025</td>
</tr>
<tr>
<td></td>
<td>Time</td>
<td>2.5833</td>
<td>0.0331</td>
<td>0.05</td>
</tr>
<tr>
<td></td>
<td>Combined*</td>
<td>1.0</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>8</td>
<td>Unoptimized</td>
<td>3.1667</td>
<td>0.0036</td>
<td>0.025</td>
</tr>
<tr>
<td></td>
<td>Grammar</td>
<td>3.6667</td>
<td>3.47E-4</td>
<td>0.0167</td>
</tr>
<tr>
<td></td>
<td>Time</td>
<td>2.1667</td>
<td>0.1175</td>
<td>0.1</td>
</tr>
<tr>
<td></td>
<td>Combined*</td>
<td>1.0</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>16</td>
<td>Unoptimized</td>
<td>3.4999</td>
<td>7.96E-4</td>
<td>0.0167</td>
</tr>
<tr>
<td></td>
<td>Grammar</td>
<td>3.4999</td>
<td>7.96E-4</td>
<td>0.025</td>
</tr>
<tr>
<td></td>
<td>Time</td>
<td>1.9999</td>
<td>0.1797</td>
<td>0.05</td>
</tr>
<tr>
<td></td>
<td>Combined*</td>
<td>1.0</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

Table 7.1 shows the non-parametric Friedman tests with Hommel’s post-hoc [García and Herrera, 2008] analysis on the speed-up of the four GAPP
variants for all the six problems at $\alpha = 0.05$. The first column indicates the number of cores. The second column shows the GAPP variant, while the third column presents the average rank. The fourth and the fifth columns show the $p$-value and Hommel’s critical value respectively. The lowest average rank shows the best (GAPP (Combined)) variant, and is marked with an asterisk (*). A variant is significantly different from the best variant if $p$-value is less than $p$-Hommel at $\alpha = 0.05$, and is in boldface.

The performance on 2 cores is insignificant as the thread overhead offsets the performance gains. For 4 cores, GAPP (Combined) significantly outperforms the remaining three variants (Unoptimized, Grammar, Time). For 8 and 16 cores, GAPP (Combined) outperforms the two GAPP (Unoptimized, Grammar) variants significantly, and the difference with GAPP (Time) is insignificant. This is due to the fact that both GAPP (Time) and GAPP (Combined) include execution time in their fitness evaluation. However, both these variants differ in terms of the number of generations. We now discuss the evolvability of GAPP.

**Mean Best Generation (MBG)**

We investigate the effect of restructuring grammars on the evolvability (ease of evolving) of the correct parallel recursive programs. To this end, we measure the mean number of generations required to converge to the best fitness, with a pre-condition that the program under consideration must be correct, averaged across 50 runs; we call it the mean best generation (MBG).

Table 7.2 shows the MBG of all the four GAPP variants and the Friedman tests with Hommel’s post-hoc analysis at $\alpha = 0.05$. A GAPP variant with the lowest MBG is in boldface. GAPP (Grammar) significantly outperforms GAPP (Unoptimized, Time, Combined), that is, GAPP (Grammar) requires less number of generations over the remaining three variants in producing the best fit programs, which is due to the grammatical bias discussed in section 7.2.1.

However, the performance results (Figure 7.2) indicate that although GAPP (Grammar) takes less number of generations to evolve the paral-
Table 7.2: The mean best generation (MBG ± [standard deviation]) of all the four GAPP (Unoptimized, Grammar, Time, Combined) variants. The lowest value is in boldface.

<table>
<thead>
<tr>
<th>Problem</th>
<th>GAPP variant</th>
<th>Average Rank</th>
<th>p-value</th>
<th>p-Hommel</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Unoptimized</td>
<td>3.0</td>
<td>5.3205E-4</td>
<td>0.001596</td>
</tr>
<tr>
<td>2</td>
<td>Grammar</td>
<td>1.16666</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>3</td>
<td>Time</td>
<td>2.33333</td>
<td>0.0321438</td>
<td>0.042</td>
</tr>
<tr>
<td>4</td>
<td>Combined</td>
<td>1.99999</td>
<td>0.0024787</td>
<td>0.02</td>
</tr>
</tbody>
</table>
parallel recursive programs, it has been outperformed by GAPP (Time, Combined). That is, the GAPP (Grammar) generated parallel programs are not efficient, whereas GAPP (Time, Combined) generate efficient parallel recursive programs. Also, GAPP (Combined) outperforms GAPP (Time), that is, GAPP (Combined) quickly generates efficient parallel recursive programs than GAPP (Time). Hence, GAPP (Combined) is the best variant that reports an average (on all the problems) speed-up of 8.13 on 16 cores, a significant improvement of 23.86% over GAPP (Unoptimized) that reports 6.19 speed-up. We now discuss the enhancements on iterative sorting.

### 7.4 Iterative Sorting

We describe the changes in the grammar design and fitness function. These changes are applied on the iterative sorting benchmarks shown in Chapter 6.

```plaintext
<ompprogram> ::= <ompppragma> <sharedprivate> <schedule> <newline> <for_in>
<ompppragma> ::= <ompdata> | <omptask>
<ompdata> ::= #pragma omp parallel | #pragma omp parallel for
<omptask> ::= #pragma omp parallel sections | #pragma omp task
<sharedprivate> ::= shared(<var>, <index>, length)
<private> ::= private(<index>) | firstprivate(<index>) | lastprivate(<index>)
<schedule> ::= schedule(<type>, CHUNK)
<type> ::= static | dynamic | guided
```

**Figure 7.3:** GAPP grammar with the separation of task and data parallel primitives to evolve efficient parallel iterative sorting programs. Complete Odd-Even sort grammar is shown in Figure B.11 of Appendix B.
7.4.1 Design of Grammars

The selection of an appropriate pragma is crucial to the overall performance of the programs, while it is important for their quick generation. We achieve this automatically by separating the data and task parallel pragmas.

Figure 7.3 shows the separation of the task and data parallel pragmas, the use of schedule, shared and private clauses. A complete Odd-Even sort grammar that evolve an efficient parallel sorting program is in Figure B.11 of Appendix B. This program works by swapping the adjacent elements in two phases. The non-terminal <omppragma> has separate rules for task (<omptask>) and data parallelism (<ompdata>); evolution selects one of them. The best evolved programs contain the <ompdata> pragmas.

The input (<var>), index (<index>), and the size (length) of the array are shared (<sharedprivate>) amongst all the cores. The production rules of <private> represent the three OpenMP private clauses, of which, private allows variable read/write operations private to the thread, firstprivate keeps initial value of a variable irrespective of the parallel region and lastprivate holds the last change of a variable in the parallel region. The temporary variable (temp in <index>) is private to the thread. Since the last two rules of <private> generate a bad individual as temp in <swap> holds a different element in each iteration. Hence, evolution keeps private (<index>) clause in the best evolved program through its fitness evaluation.

Similarly, in scheduling the parallel <for_in> loop, OpenMP offers three clauses (<type>): static, dynamic and, guided; static divides the work among threads before the loop execution and dynamic allocates the work during the execution. The guided also divides the work during the execution but, the allocation begins with the given chunk size (described in Chapter 3) and decreases for the next requests.

These scheduling types are detailed in Chapter 3; they operate with a default chunk size of 1, but we use chunk=10 for the experiments in this chapter. A study on an ideal chunk size is laid out later in Chapter 8. A program with dynamic clause is the best fit than the ones with other schedule clauses and preferentially selecting this clause is up to the evolution.
The grammar has arithmetic operations (\(<bop>\), \(<lop>\)) and constants (\(<\text{const}>\)). We use absolute values (\(\text{abs}\) in \(<\text{for\_in\_line}>\) and \(<\text{swap}>\)) to prohibit negative array indexing. Next, we discuss the fitness evaluation.

### 7.4.2 Performance Optimization

Since the use of different pragmas can impact performance, considering the execution time in fitness evaluation helps to choose an apt pragma. Thus, the fitness function is a product of the execution time and program accuracy. The program accuracy is the mean inversions (out-of-ordered pairs). For example, if \(a_1a_2a_3...a_n\) is a permutation of the set \(1, 2, \ldots, n\) then the pair \((a_i, a_j)\) is an inversion of the permutation iff \(i < j\) and \(a_i > a_j\) [Knuth, 1998].

Both the fitness components are normalized in the range \([0, 1]\) – maximization function. The fitness function \((f_{\text{prog}})\) is given as shown in eq. 7.3.

\[
\begin{align*}
\quad f_{\text{prog}} &= \frac{1}{(1+t)} \times \frac{1}{\left[ \sum_{i=1}^{N} n(I(A_i)) \right] + \frac{\sum_{i=1}^{N} n(I(A_i))}{TP}} \tag{7.3}
\end{align*}
\]

where, \(t\) stands for the execution time of the evolved parallel program over all the fitness cases \((N)\); \(n(I(A_i))\) is the number of inversions in the \(i^{th}\) array \((A_i)\); total, \(N\) arrays); and \(TP\) is the total number of pairs in all the fitness cases \((N)\).

Selecting an inapt pragma increases the execution time of the evolved program. The time component of \(f_{\text{prog}}\), thus ensures to select an apt pragma. Meanwhile, normalized mean inversions assures the accuracy of sorting. Thus, we obtain a correct sorting program that is optimized for multi-cores.

### 7.5 Experiments

We examine the proposed approach with the experimental settings and four iterative sorting benchmarks detailed in Chapter 6. We experiment with the four GAPP variants (in section 7.3.1) on these benchmarks.
7.5.1 Results

The results in this section are twofold: first, we compare the speed-up and mean best generation (MBG) among the four GAPP variants; and second, we compare the mean best execution time (MBT) of GAPP (Unoptimized, Combined) with that of the GNU GCC optimization flags.

**Figure 7.4**: The speed-up of all the four GAPP (Unoptimized, Grammar, Time, Combined) variants on the four iterative sorting benchmarks for 2, 4, 8, and 16 cores. The horizontal dashed line (- -) shows the speed-up of 1.

**Speed-up**

Figure 7.4 shows the speed-up of GAPP (Unoptimized, Grammar, Time, Combined) on the four iterative sorting benchmarks for 2, 4, 8 and 16 cores. The horizontal dashed line (- -) represents a speed-up of 1.
Table 7.3 shows the Friedman statistical tests with Hommel’s post-hoc analysis on speed-up of GAPP (Unoptimized, Grammar, Time, Combined) for 4, 8, and 16 cores. A variant with the lowest average rank is the best variant (GAPP(Combined)) and marked with an asterisk (*).

**Table 7.3:** Friedman tests with Hommel’s post-hoc analysis on the speed-up of the four GAPP variants for 4, 8 and 16 cores. The boldface shows the significantly different results at $\alpha = 0.05$ and asterisk (*) shows the best variant.

<table>
<thead>
<tr>
<th>Cores</th>
<th>GAPP variant</th>
<th>Average Rank</th>
<th>$p$-value</th>
<th>$p$-Hommel</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>Unoptimized</td>
<td>3.75</td>
<td><strong>0.006169</strong></td>
<td>0.0166</td>
</tr>
<tr>
<td></td>
<td>Grammar</td>
<td>3.25</td>
<td>0.028459</td>
<td>0.025</td>
</tr>
<tr>
<td></td>
<td>Time</td>
<td>1.75</td>
<td>0.58388</td>
<td>0.05</td>
</tr>
<tr>
<td></td>
<td>Combined*</td>
<td>1.25</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>8</td>
<td>Unoptimized</td>
<td>4.0</td>
<td><strong>0.007383</strong></td>
<td>0.0166</td>
</tr>
<tr>
<td></td>
<td>Grammar</td>
<td>3.0</td>
<td><strong>0.014825</strong></td>
<td>0.0233</td>
</tr>
<tr>
<td></td>
<td>Time</td>
<td>2.0</td>
<td>0.232147</td>
<td>0.05</td>
</tr>
<tr>
<td></td>
<td>Combined*</td>
<td>1.0</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>16</td>
<td>Unoptimized</td>
<td>4.0</td>
<td><strong>0.001015</strong></td>
<td>0.0166</td>
</tr>
<tr>
<td></td>
<td>Grammar</td>
<td>3.0</td>
<td><strong>0.0204597</strong></td>
<td>0.025</td>
</tr>
<tr>
<td></td>
<td>Time</td>
<td>2.0</td>
<td>0.2733216</td>
<td>0.05</td>
</tr>
<tr>
<td></td>
<td>Combined*</td>
<td>1.0</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

The results are insignificant on 2 cores for the four variants. For 4 cores, GAPP (Combined) outperforms GAPP (Unoptimized) while it is insignificant from GAPP (Grammar, Time). For 8 and 16 cores, GAPP (Combined) outperforms GAPP (Unoptimized, Grammar) and is insignificant over GAPP (Time). GAPP (Combined) shows an average speed-up of 11.03, an improvement of 15.75% over GAPP (Unoptimized) (speed-up 9.29).
Table 7.4: The mean best generation (MBG ± [standard deviation]) of all the GAPP (Unoptimized, Grammar, Time, Combined) variants. The lowest generation is in boldface.

<table>
<thead>
<tr>
<th>Problem</th>
<th>GAPP</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Unoptimized</td>
<td>Grammar</td>
<td>Time</td>
<td>Combined</td>
</tr>
<tr>
<td>mean best generation</td>
<td>mean best generation</td>
<td>mean best generation</td>
<td>mean best generation</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>67.19 ± [4.16]</td>
<td>37.63 ± [3.19]</td>
<td>73.27 ± [3.31]</td>
<td>41.27 ± [0.81]</td>
</tr>
</tbody>
</table>

Friedman statistical tests with Hommel’s post-hoc analysis. Boldface represents the significance at $\alpha = 0.05$, while asterisk (*) shows the best variant among all the four GAPP variants.

<table>
<thead>
<tr>
<th>GAPP variant</th>
<th>Average Rank</th>
<th>$p$-value</th>
<th>$p$-Hommel</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unoptimized</td>
<td>3.25</td>
<td>0.0284597</td>
<td>0.025</td>
</tr>
<tr>
<td>Grammar*</td>
<td>1.25</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Time</td>
<td>3.75</td>
<td>0.0061698</td>
<td>0.0166</td>
</tr>
<tr>
<td>Combined</td>
<td>1.75</td>
<td>0.5838824</td>
<td>0.05</td>
</tr>
</tbody>
</table>
**Mean Best Generation (MBG)**

Table 7.4 compares the mean best generation (MBG) of GAPP (Unoptimized, Grammar, Time, Combined) and their statistical significance tests. Here, GAPP (Grammar) outperforms GAPP (Unoptimized, Time) while it is insignificant over GAPP (Combined). That is, GAPP (Grammar) produces parallel iterative sorting programs in fewer generations, while GAPP (Time) takes more generations. It is because of the alterations in the design of the grammars among GAPP variants, which effects the evolution of the programs. However, GAPP (Combined) evolves efficient programs over GAPP (Grammar) (see Figure 7.4). Hence, GAPP (Combined) is the best variant for the evolution of efficient parallel iterative sorting programs.

**Compiler Optimizations**

We compare the mean best execution time (MBT) of GAPP (Unoptimized, Combined) with that of the GNU GCC optimization flags. These flags\(^1\) (–O1, –O2, –O3) try to minimize the code, and reduce the execution time. Of these flags, O1, optimizes the source code with conditional branching, copy propagations, etc., and moderately reduces the time with no changes in compile time. O2, along with O1, offers aliasing, cross jumps, etc., to reduce the time with a slight increase in compile time. O3, along with O2, offers auto-vectorization, function in-lining, etc., to fully reduce the execution time. GE evolves serial programs that are compiled with these flags.

The experimental settings in Chapter 6 are reused for these experiments. Table 7.5 compares the MBT of the two GAPP (Unoptimized, Combined) variants and the three flags for 2, 4, 8, and 16 cores on the four iterative sorting benchmarks. The lowest execution time is in boldface for the corresponding method on a given problem. If we observe MBT of the optimization flags alone, the results indicate that it is reducing from O1 to O3, which is an expected phenomenon. Although these flags reduce the time, GAPP (Unoptimized, Combined) variants exhibit better reduction.

\(^{1}\)https://gcc.gnu.org/onlinedocs/gcc-4.4.5/gcc/Optimize-Options.html
Table 7.5: The mean best execution time (in secs) (MBT [standard deviation]) of GAPP (Unoptimized, Combined) and the optimization flags (O1, O2, O3). The boldface represents the lowest execution time.

<table>
<thead>
<tr>
<th>Cores</th>
<th>Problem</th>
<th>mean best execution time (MBT)</th>
<th>GAPP (Unoptimized)</th>
<th>GAPP (Combined)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>O1</td>
<td>O2</td>
<td>O3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>3987.4 [27.2]</td>
<td>3999.2 [29.3]</td>
<td>3819.3 [21.9]</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>3219.5 [24.2]</td>
<td>3331.6 [31.2]</td>
<td>2883.7 [22.2]</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>2744.7 [29.6]</td>
<td>2234.6 [30.2]</td>
<td>2584.5 [29.3]</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>2262.5 [31.4]</td>
<td>3109.3 [28.3]</td>
<td>2396.2 [29.2]</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>2435.3 [22.3]</td>
<td>2319.9 [33.4]</td>
<td>2445.5 [29.2]</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>2347.1 [25.4]</td>
<td>2835.8 [27.3]</td>
<td>3177.1 [24.8]</td>
</tr>
</tbody>
</table>
Table 7.6 shows the non-parametric Friedman statistical tests with Hommel’s post-hoc analysis on MBT of GAPP (Unoptimized, Combined) and the three optimization flags for 4, 8 and 16 cores.

**Table 7.6**: Friedman tests with Hommel’s post-hoc analysis on MBT of GAPP (Unoptimized, Combined) and O1, O2, O3 at $\alpha = 0.05$. The asterisk (*) shows the best method, while significantly different methods are in boldface.

<table>
<thead>
<tr>
<th>Cores</th>
<th>Method</th>
<th>Average Rank</th>
<th>$p$-value</th>
<th>$p$-Hommel</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>O1</td>
<td>4.5</td>
<td>0.00729</td>
<td>0.0125</td>
</tr>
<tr>
<td></td>
<td>O2</td>
<td>4.5</td>
<td>0.00729</td>
<td>0.0166</td>
</tr>
<tr>
<td></td>
<td>O3</td>
<td>2.75</td>
<td>0.02357</td>
<td>0.025</td>
</tr>
<tr>
<td></td>
<td>Unoptimized</td>
<td>2.15</td>
<td>0.02306</td>
<td>0.05</td>
</tr>
<tr>
<td></td>
<td>Combined*</td>
<td>1.5</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>8</td>
<td>O1</td>
<td>5.0</td>
<td>3.65E-3</td>
<td>0.0025</td>
</tr>
<tr>
<td></td>
<td>O2</td>
<td>4.0</td>
<td>0.00961</td>
<td>0.0196</td>
</tr>
<tr>
<td></td>
<td>O3</td>
<td>2.5</td>
<td>0.0107</td>
<td>0.020</td>
</tr>
<tr>
<td></td>
<td>Unoptimized</td>
<td>2.25</td>
<td>0.01952</td>
<td>0.025</td>
</tr>
<tr>
<td></td>
<td>Combined*</td>
<td>1.75</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>16</td>
<td>O1</td>
<td>4.85</td>
<td>7.32E-4</td>
<td>0.0012</td>
</tr>
<tr>
<td></td>
<td>O2</td>
<td>4.15</td>
<td>0.00254</td>
<td>0.0107</td>
</tr>
<tr>
<td></td>
<td>O3</td>
<td>2.5</td>
<td>0.00134</td>
<td>0.020</td>
</tr>
<tr>
<td></td>
<td>Unoptimized</td>
<td>1.95</td>
<td>0.02012</td>
<td>0.025</td>
</tr>
<tr>
<td></td>
<td>Combined*</td>
<td>1.15</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

The best approach (GAPP (Combined)) is marked with an asterisk (*). A value is in *boldface* if it is significantly different from the best method. Although it is expected that increasing the number of cores should reduce the time to execute a parallel program, that is not true for all these results.
Instead, the results indicate that GAPP (Combined) outperforms all its counterparts except for 2 cores.

Finally, GAPP (Combined) evolving parallel sorting programs record better performance over the GE evolving optimized serial sorting programs. Next, we discuss the factors that influence the performance.

### 7.6 Discussion

It is interesting to analyze the factors that influence the performance of the evolving parallel programs when they are executed on multiple cores of a processor. GAPP (Combined) optimizes the performance of the evolving parallel programs on all the experiments except for those 2 cores. It is because of the OpenMP specific thread scheduling issues.

Although GAPP (Combined) boosts up the performance over the original system, it fails to fully utilize the power of multi-cores. That is, when the evolving programs are scheduled to execute on 16 cores, it reported a speed-up of 8.13 for recursive benchmarks and 11.03 for iterative sorting benchmarks, whereas the ideal case should be 16. Mainly factors such as thread creation and scheduling contribute in obviating to achieve the ideal scale-up. We analyse these issues for both the problem domains and further enhance the performance of the evolving programs in Chapter 8.
7.7 Summary

In summary, this chapter enhanced GAPP to automatically generate efficient parallel programs. This study offered a separation between the task and data parallelism in the design of the grammars while considered the execution time in fitness evaluation. The modifications in the grammar favoured quick generation of programs, while the execution time helped in improving their performance.

Next, Chapter 8 expands in several directions: we intend to see the impact of parallelism on code growth so as to delineate the computational overhead of scheduling threads. Also, there are challenges that associated with avoiding excessive parallelism which can result from producing too many threads that accomplish too little individually, and maintaining correctness of the evolving solution while promoting parallelism.
Chapter 8

Analysis and Extension of GAPP

“Excellence is a continuous process and not an accident”

— A.P.J. Abdul Kalam, Scientist

The enhancements of GAPP in the previous chapter have improved the performance of the evolving parallel programs. Nevertheless, we identify a lot of scope to further improve their performance. In realizing this goal, this chapter scrutinizes both the evolutionary and non-evolutionary factors of GAPP that have potential influence on the performance of the evolving parallel programs. The evolutionary factor considered in the study is code growth while the non-evolutionary factors are time complexity and thread scheduling. Parts of this research has been published [Chennpati et al., 2015], [Chennupati et al., 2015a], [Chennupati et al., 2015c].

8.1 Introduction

This chapter analyses the effect of code growth, time complexity and OpenMP thread scheduling on performance of the GAPP evolved parallel programs, both in recursion and iterative sorting domains. We find that code growth in GAPP is surprisingly insignificant, therefore it does not affect the program
execution. The computational complexity analysis shows the creation of threads and the resultant degree of parallelism in the end programs. Thus, scheduling these threads is a significant challenge with its load balancing. It has a distinct influence on both the problem domains.

For example, one risk in evolving parallel recursion is excessive multithreading that spawns a thread at every recursive call; thus, each thread does little useful work. This degrades the performance due to the overhead in first creating and then scheduling these threads. Scheduling can be especially expensive as a large number of threads compete to get a slice of CPU time. On the other hand, for iterative sorting, the amount of work divided among the threads varies with the number of cores and the number of threads. We need a controlling mechanism that deals with these challenges effectively.

Therefore, we further tweak the grammars such that the evolved programs can control the creation of threads after a certain depth in their recursive trace. Thus, the lower level recursive calls run in serial by arresting the thread creation, while the top level calls run in parallel. For iterative sorting, the tweaks to the grammar schedule the work load efficiently among the threads, also control the invalid cache line accesses. These modifications lead to scale up the performance of the parallel programs. These programs from both the domains adapt to the underlying hardware architectures. We now analyze and extend GAPP in both the problem domains.

8.2 Recursion

This section analyses the code growth, time complexity and the thread scheduling issues for the six recursive benchmarks.

8.2.1 Code Growth

We compare the size of standard GE (without OpenMP pragmas; it evolves serial programs) with that of GAPP (Unoptimized, Combined) genotypes. Recall that GE genomes have two sizes (see Chapter 2): actual and effective.
Figure 8.1: Actual and effective lengths of GE and GAPP (Unoptimized, Combined) genotypes. The lengths are averaged across 50 runs of 100 generations.
Figure 8.1 presents the actual and effective lengths of GE and GAPP (Unoptimized, Combined), that are averaged across 50 runs of 100 generations for each of the six recursive benchmarks. As expected, the actual and effective lengths vary significantly for the particular set-ups, with Wilcoxon Singed Rank Sum tests at $\alpha = 0.05$. This means, for a given set up, there is always a statistically significant difference between the actual and effective lengths of the genomes that it uses. For example, in the case of GE, there is a significant difference between GE-actual and GE-effective, which is similar for GAPP (Unoptimised, Combined).

When we compare across the approaches, surprisingly, the statistical tests show insignificant difference between the actual length of GE and GAPP (Unoptimized, Combined). This is unexpected, although a recent study [Azad and Ryan, 2014] reported a similar behaviour on different problems. There exists no investigations that analysed the reasons for such a behaviour. We hypothesis this as the fact that, GE generates larger genotypes than required, thus, the GAPP grammars do not influence the actual length, as both use the same search engine. Instead, GAPP uses some of the actual size to map the OpenMP pragmas, hence, effective length increases.

The effective lengths of GAPP (Unoptimized, Combined) are significantly larger than that of GE at $\alpha = 0.05$ due to the extra mapping steps. However, among the GAPP variants, the effective lengths do not vary significantly. It shows the fact that the changes in the design of GAPP grammars forfeit the overhead in code growth. This slight increase in the effective length may impact the performance of the programs only for 2 cores. However, for 4 cores and above, it has negligible implications, because now the impact of the extra cores is greater than that of the extra code. Also note, the code growth is independent of the number of cores under execution.

**8.2.2 Computational Complexity**

We empirically analyze the time complexity of GAPP (Combined) generated task parallel recursive programs. In this analysis, we consider the number of recursive calls of a program over the given input.
This analysis shows that the problems Sum-of-N, Factorial, Reverse exhibit $O(n)$ (linear) complexity whereas, Binary-Sum and Quicksort exhibit $O(\log(n))$ and, $O(n \log(n))$ complexity respectively, while Fibonacci shows $O(2^n)$ complexity. That is, the evolving programs are competitive with that of the conventional human written programs. Note, the use of parallel hardware can only reduce the computational load by dividing it among the existing processing elements, but not the computational complexity. To that end, Binary-Sum exhibits the lowest complexity, $O(\log(n))$.

However, Fibonacci required an exponential number of recursive calls. In such cases the performance fails to scale up due to excessive parallelism, because an exponential number of recursive calls create an exponential number of threads, where too many threads operate to perform too little individually. Clearly, this means that the degree of parallelism needs to be managed better. Hence, in this chapter, we further optimize the performance. We now discuss the challenges in scheduling these threads.

### 8.2.3 Thread Scheduling

The quality of parallel code is difficult to quantify as execution time often depends on the ability of OS to efficiently schedule the tasks. This job itself is complicated by other parallel threads (from other programs) running at the same time. OpenMP abstracts away much of these concerns from programmers, which makes it easier at the cost of some of fine control. We compensate this by adapting to program to the hardware.

Hardware can cap the maximum number of threads; however, in the given grammars each recursive call spawning a new thread. Then, the OS-specific factors for the Linux kernels, which eventually fail to scale in scheduling the very high number of threads [Boyd-Wickizer et al., 2010]. Moreover, a parent thread spawns a child thread, it sleeps until all the child threads have finished. This process is expensive, when a large number of threads are involved. Also, memory access restrictions over shared and private variables can add to the complexity of the executing code. Complexity in this instance
comes from the vagaries of scheduling what can be a very high number of
treads. We extend GAPP to overcome these limitations.

8.3 Extending GAPP for Recursion

Armed with the knowledge of excessive parallelism, we constrain the sys-
tem so as to optimize the degree of parallelism. We combine parallel and
serial implementations of the evolved programs, which, further improves
the performance. This reduces the overhead caused due to excessive par-
allelism as the top level recursive calls distribute load across a number of
threads, whereas the lower level calls appropriately carry out the work in-
stead of merely invoking more threads. Evolution detects the appropriate
\textit{level} at which recursion switches from parallel to serial.

\begin{center}
\texttt{<condition> ::= if(<input><lop><const>) \{ \newline
<line1>;<newline><line2>;<newline> \}}
\end{center}

is altered to appear as
\begin{center}
\texttt{<condition> ::= if(<input><lop><const>) \{ \newline
<line1>;<newline><line2>;<newline> \} \newline
else if(<input><lop><const><const>)\{<newline><line1>; \newline
<line2>; <newline> \}}
\end{center}

\textbf{Figure 8.2:} The enhanced GAPP grammars to generate a parallel recursive
Fibonacci program that is both serial and parallel.

The GAPP grammars used for recursive benchmarks in Chapter 7 are
modified as shown in Figure 8.2, termed as \textit{GAPP (Scaled),} hereafter. We
alter the non-terminal \texttt{<condition>} to generate nested \textit{if-else} condition
blocks. The changes generate a program that reduces the execution time
of the evolved programs. It generates a two digit \textit{thread limiting constant}
automatically, at which, the program starts to execute sequentially.

Figure 8.3 shows an example of the GAPP (Scaled) generated Fibonacci
program. It evolves a thread limiting constant for a given problem and
the computational environment; this constant, arrests the further creation of threads and continues to execute serially. The intermediate result (temp in else if) is shared among the threads, thus, further optimizes the execution time, thereby, efficiently exploits the power of multi-cores.

```c
if (n <= 2) {
    temp = n; res += temp;
}
else if (n <= 39) {
    temp = fib(n-1)+fib(n-2); res += temp;
}
else {
    #pragma omp parallel sections private(a) shared(n, temp, res)
    {
        #pragma omp section
        {
            a = fib(n-1);
            #pragma omp atomic res += temp+a;
        }
        #pragma omp section
        {
            a = fib(n-2);
            #pragma omp atomic res += temp+a;
        }
    }
    return res;
}
```

**Figure 8.3:** GAPP (Scaled) evolved program that combines both parallel and serial execution to increase the performance.

The constant (39) in the else if (in Figure 8.3) condition is the thread limiting constant for 16 cores. Figure 8.4 shows the thread limiting constants range (with standard deviation) with respect to the number of cores for all the six benchmarks. The evolved constants for each problem adapt to the underlying hardware architectures. For example, the constant (39),
Figure 8.4: GAPP (Scaled) evolved thread limiting constants for all the six recursive benchmarks averaged across 50 runs.

Figure 8.5: The performance of GAPP (Scaled) evolved programs for all the six recursive benchmarks on 2, 4, 8, and 16 cores.
which, at a large input (say, a 1000000 element array) may not be an optimal value, that can be a bigger constant. This can be addressed with digit concatenation grammars [O’Neill and Ryan, 2003, Chapter-5].

Figure 8.5 shows the speed-up of GAPP (Scaled) over all the six benchmarks for 2, 4, 8, and 16 cores. Like the other GAPP variants, the speed-up of GAPP (Scaled) improves with an increase in the number of cores. Especially, the speed-up of GAPP (Scaled) can be better seen, where the performance is much better than its counterparts.

Table 8.1: The mean best generation (MBG ± [standard deviation]) of GAPP (Grammar, Combined, Scaled). The lowest value is in boldface.

<table>
<thead>
<tr>
<th>Problem</th>
<th>Unoptimized</th>
<th>Grammar</th>
<th>Combined</th>
<th>Scaled</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>mean best generation</td>
<td>mean best generation</td>
<td>mean best generation</td>
<td>mean best generation</td>
</tr>
<tr>
<td>6</td>
<td>49.25 [4.57]</td>
<td><strong>40.49 [5.23]</strong></td>
<td>52.49 [2.58]</td>
<td>47.28 [3.15]</td>
</tr>
</tbody>
</table>

Table 8.1 presents the mean best generation (MBG) of GAPP (Unoptimized, Grammar, Time, Combined, Scaled) variants. The results show that GAPP (Grammar) generates a program faster than the two GAPP (Combined, Scaled) variants because of the grammatical bias. However, the last two GAPP (Combined, Scaled) variants use execution time in fitness evaluation. Thus, it makes the evolution slightly hard regardless of that, it generates the efficient task parallel recursive programs.
Table 8.2 shows the non-parametric Friedman tests with Hommel’s post-hoc analysis on speed-up and MBG of GAPP (Unoptimized, Grammar, Time, Combined, Scaled). The best variant with the lowest rank is marked with an asterisk (*), and significantly different variants are in boldface.

**Table 8.2:** Friedman tests with Hommel’s post-hoc analysis on *speed-up* and *MBG* of GAPP (Unoptimized, Grammar, Time, Combined, Scaled). Boldface shows the significance (at $\alpha = 0.05$) and asterisk (*) shows the best variant.

<table>
<thead>
<tr>
<th>GAPP variant</th>
<th>Average Rank</th>
<th>p-value</th>
<th>p-Hommel</th>
</tr>
</thead>
<tbody>
<tr>
<td>Speed-up</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Unoptimized</td>
<td>4.5</td>
<td>1.2604E-4</td>
<td>0.0125</td>
</tr>
<tr>
<td>Grammar</td>
<td>4.5</td>
<td>1.2604E-4</td>
<td>0.0166</td>
</tr>
<tr>
<td>Time</td>
<td>3.0</td>
<td>0.0284597</td>
<td>0.025</td>
</tr>
<tr>
<td>Combined</td>
<td>1.9998</td>
<td>0.0347332</td>
<td>0.05</td>
</tr>
<tr>
<td>Scaled*</td>
<td>0.99999</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Mean Best Generation</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Unoptimized</td>
<td>4.333333</td>
<td>0.0019107</td>
<td>0.0125</td>
</tr>
<tr>
<td>Grammar*</td>
<td>1.5</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Time</td>
<td>3.833333</td>
<td>0.0105871</td>
<td>0.01667</td>
</tr>
<tr>
<td>Combined</td>
<td>2.49998</td>
<td>0.0355132</td>
<td>0.05</td>
</tr>
<tr>
<td>Scaled</td>
<td>3.666667</td>
<td>0.0176221</td>
<td>0.025</td>
</tr>
</tbody>
</table>

For *speed-up*, GAPP (Scaled) outperforms the remaining four GAPP variants. Note, these results are for 16 cores of a processor, and are similar for 8 cores, while they are insignificant with 4 cores and below. On average, for 16 cores, GAPP (Scaled) speeds up by a factor of 9.97, which improves over GAPP (Combined) and GAPP (Unoptimized) by 17.45% and 37.91% respectively.

190
Figure 8.6: The actual and effective lengths of GE, GAPP (Unoptimized, Combined) of the four iterative sorting benchmarks.

For MBG, the GAPP (Grammar) outperforms the remaining four GAPP variants. Note, these results are for 16 cores, while they are similar for 8 cores and below. Although GAPP (Scaled) requires slightly more number of generations over other variants, GAPP (Scaled) is the best, as it generates efficient parallel recursive programs.

However, a similar solution is to keep a table that records the result of a recursive call in its first evaluation, then, refer the table for the repeated recursive calls, similar to Koza [Koza, 1992]. But it has often been criticized [Moraglio et al., 2012] for not being an exact recursion. We now analyze and extend GAPP for iterative sorting domain.
8.4 Iterative Sorting

This section analyses the code growth, time complexity and thread scheduling of GAPP for the four iterative sorting benchmarks.

8.4.1 Code Growth

Figure 8.6 shows the actual and effective lengths of GE, GAPP (Unoptimized, Combined) for the four iterative sorting benchmarks. Like in recursion, both the lengths in iterative sorting differ significantly with the Wilcoxon Signed Rank Sum tests at $\alpha = 0.05$ for a given approach.

Like in recursion, there is no significant difference between the actual lengths of GE and GAPP (Unoptimized, Combined). We hypothesize this as GE generates larger genotypes than the required, which are unaffected even with the use of parallelization pragmas. Rather, a part of the genotype generates a parallel program, as a result, the effective length increases. The effective lengths of GE and GAPP differ significantly at $\alpha = 0.05$ due to the fact that GAPP requires extra number of mappings to evolve a parallel program. However, the effective size differs insignificantly on both the GAPP (Unoptimized, Combined) variants at $\alpha = 0.05$.

Although, the effective lengths increase significantly, the difference is only marginal; it may offset the gains for 1 to 2 cores. However, for 4 cores and above, this increase is not much of an issue given the power of the multiple cores. However, we find that GE does not bloat as much as GP, a similar happening as in [Azad and Ryan, 2014]. The reasons for such nature requires further analysis, a matter of future research.

8.4.2 Computational Complexity

This section presents an empirical analysis on the computational complexity of GAPP (Combined) evolving parallel iterative sorting programs. The empirical analysis is conducted in terms of the amount of time taken by the
best evolved program for sorting an input. Paralleling an algorithm do not affect the complexity, nevertheless, it optimizes the execution time.

**Table 8.3:** The computational complexity of the evolved parallel programs compared with that of the ideal \( (n \text{ processors}) \) as well as the state-of-the-art evolutionary implementations on the four iterative sorting benchmarks.

<table>
<thead>
<tr>
<th>Problem</th>
<th>Ideal</th>
<th>GAPP</th>
<th>[Agapitos and Lucas, 2006]</th>
<th>[Spector et al., 2005]</th>
<th>[Kinnear, 1993b]</th>
<th>[O’Neill et al., 2014]</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>( O(n) )</td>
<td>( O(n^2) )</td>
<td>-</td>
<td>( O(n^2) )</td>
<td>( O(n^2) )</td>
<td>( O(n^2) )</td>
</tr>
<tr>
<td>2</td>
<td>( O(n) )</td>
<td>( O(n \log n) )</td>
<td>( O(n \log n) )</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>3</td>
<td>( O(n) )</td>
<td>( O(n^2) )</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>4</td>
<td>( O(n) )</td>
<td>( O(n^2) )</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

Table 8.3 compares the time complexity of GAPP (Combined) evolved parallel iterative sorting programs with that of the ideal and the state-of-the-art evolutionary attempts (see Chapter 6). The ideal case is when the programs are executed on \( n \) processors. The results show that the ideal case is hard to achieve, while it is competitive with the evolutionary attempts.

### 8.4.3 Thread Scheduling

The non-evolutionary factors such as OpenMP scheduling play a vital role in optimizing the performance. Interestingly, OpenMP hides these details from the developer, which makes it easy to use, at the same time hard to realize its full potential. Load balancing by parallel threads is a serious concern on shared memory processors. OpenMP scheduling strategies (*static, dynamic, guided*) (described in Chapter 3) answer these performance issues effectively. However, it becomes complicated in setting the optional *chunk size* (*chunk*) explicitly, as the ideal value often requires the problem specific knowledge. That is, it changes with respect to the loop iterations, number of cores, and the threads under execution. Also, smaller chunks of data leads to a well known parallel programming challenge of *false sharing*. 

193
**False Sharing**

False sharing is a performance challenge that occurs when threads on different cores modify variables that reside on the same cache line [Torrellas et al., 1994]. This invalidates the cache line and forces a memory update, thereby reduces the performance.

Precisely, if one core tries to load the same cache line loaded by another core, then that line is marked with “shared” access. If this core stores shared cache line, then that line is marked as “modified” and all the remaining cores will receive a cache line “invalid” message. Now, if any core tries to access the cache line marked with *modified*, then that line will be written back to the memory and marks it as “shared”. The other cores that try to access the same cache line will incur a cache miss. This frequent coordination among the cores, cache lines and memory that caused due to false sharing significantly degrades the performance of an application.

False sharing can be avoided by placing the variables far apart in the memory (using some complier directives) so that they do not align in the same cache line. In the case of arrays, it can be avoided by aligning the array of elements on the cache line boundary. If this is impossible, we can set the array size to double the cache line, which is possible when dynamically allocating the array sizes. Our extensions of GAPP ensure that controlling the array sizes helps to deal with false sharing in improving the performance of the evolving programs.

### 8.5 Extending GAPP for Iterative Sorting

This section further extends GAPP to evolve more efficient parallel iterative sorting programs. We overcome the problem of ideal load balancing by evolving an appropriate *chunk size* that is independent of the problem and the number of cores that it executes. We adopt the digit concatenation grammars [O’Neill and Ryan, 2003, Chapter-5] that are used in solving the symbolic regression problems, termed as GAPP (Scaled), hereafter.
is modified to appear as

<schedule> ::= schedule(<type>, <const1>)
<const1> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | <const1><const1>

Figure 8.7: Enhanced GAPP grammars that generate an adaptable chunk size for thread scheduling.

```c
for(i=0; i < length; i++) {
    if (i%2 == 0) { //start of even phase
        #pragma omp parallel for shared(A,length) private (j,temp) schedule(dynamic, 89)
        {
            for(j=1; j < length-1; j+=2) {
                if(A[abs(j-1)]<A[abs(j-0)]) {
                    temp=A[abs(j-1)]; A[abs(j-1)]=A[abs(j-0)];
                    A[abs(j-0)]=temp;
                }
            }
        }
    } else {
        //start of odd phase
        #pragma omp parallel for shared(A,length) private (j,temp) schedule(dynamic, 87)
        {
            for(j=1; j < length-1; j+=2) {
                if(A[abs(j)] > A[abs(j+1)]) {
                    temp=A[abs(j+1)]; A[abs(j+1)]=A[abs(j+0)];
                    A[abs(j+0)] = temp;
                }
            }
        }
    }
} // end for loop
```

Figure 8.8: Evolved Odd-Even program that combines both parallel and serial execution to increase the speed-up.
Figure 8.7 shows the modified GAPP grammar that automatically generates a sequence of digits. The evolved chunk size adapts to the number of cores, amount of load, and the number of threads. The proposed enhancements evolve more efficient programs.

Figure 8.8 presents the successfully evolved parallel iterative Odd-Even sort program using GAPP (Scaled) grammars. Note, the program contains two constants (89, 87) as it operates in two phases (odd and even).

Table 8.4: GAPP (Chunk) evolved chunk size (mean ± [standard deviation]), averaged across 50 runs for all the four experimental problems.

<table>
<thead>
<tr>
<th>Problem</th>
<th>chunk size (CHUNK)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>8 cores</td>
</tr>
<tr>
<td>Quicksort</td>
<td>159.34 ± [22.71]</td>
</tr>
<tr>
<td>Odd-Even sort</td>
<td>166.81 ± [17.33]</td>
</tr>
</tbody>
</table>

Table 8.4 shows the GAPP (Scaled) evolved constants (chunk size). It is the average of the evolved best of run programs averaged across 50 runs. The chunk results are reported for 8, and 16 cores. They showed significant performance optimization while, 2 and 4 are insignificant, hence, neglected. As a result, the evolved constants balance the load effectively. These chunk sizes created larger data arrays as opposed to the arrays with a default chunk size of 10 (see Chapter 6). Creation of larger chunks of data helped in controlling the false sharing. The effect is clearly evident at the higher number of cores, thus, the performance of the evolving programs improves. We now report the speed-up of the enhancements.

Figure 8.9 shows the speed-up of GAPP (Scaled) evolved programs. The results indicate that the performance improves with an increase in the num-
Bubble sort
Quick sort
Odd-Even sort
Rank sort

Problem

4
8
12
16

1
2
4
8
16

Speed up

GAPP (Scaled)

Figure 8.9: Performance of GAPP (Scaled)

Table 8.5: Significance tests (at $\alpha = 0.05$) show that GAPP (Chunk) outperforms GAPP (Combined) for 8 and 16 cores. Note that “✓” states that the results are significant ($p$-value < 0.05). A measure shows the probability at which, GAPP (Chunk) is better over GAPP (Combined).

<table>
<thead>
<tr>
<th>Cores</th>
<th>Problem</th>
<th>Wilcoxon Signed Rank Sum Test</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Rank</td>
<td>$p$ value</td>
</tr>
<tr>
<td>8</td>
<td>Sum</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>2089</td>
<td>0.00632</td>
</tr>
<tr>
<td>2</td>
<td>2798</td>
<td>0.03183</td>
</tr>
<tr>
<td>3</td>
<td>3321</td>
<td>0.01119</td>
</tr>
<tr>
<td>4</td>
<td>2479</td>
<td>0.04178</td>
</tr>
<tr>
<td>16</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>2250</td>
<td>0.04261</td>
</tr>
<tr>
<td>2</td>
<td>2701</td>
<td>0.00018</td>
</tr>
<tr>
<td>3</td>
<td>3253</td>
<td>0.00461</td>
</tr>
<tr>
<td>4</td>
<td>2221</td>
<td>0.03516</td>
</tr>
</tbody>
</table>

197
ber of cores. It shows an average speed-up of 12.52 for 16 cores, a better improvement of 11.91% over GAPP (Combined), an improvement of 25.79% over GAPP (Unoptimized).

Table 8.6: The mean best generation (mean [standard deviation]) of all the GAPP (Unoptimized, Grammar, Time, Combined, Scaled) variants. The lowest generation is in boldface.

<table>
<thead>
<tr>
<th>Problem</th>
<th>Unoptimized</th>
<th>Grammar</th>
<th>Time</th>
<th>Combined</th>
<th>Scaled</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>mean best generation</td>
<td>mean best generation</td>
<td>mean best generation</td>
<td>mean best generation</td>
<td>mean best generation</td>
</tr>
<tr>
<td>1</td>
<td>67.19 [4.16]</td>
<td><strong>37.63 [3.19]</strong></td>
<td>73.27 [3.31]</td>
<td>41.27 [0.81]</td>
<td>49.57 [2.15]</td>
</tr>
</tbody>
</table>

Friedman tests with Hommel’s post-hoc analysis. Boldface shows the significance at $\alpha = 0.05$, while asterisk (*) shows the best GAPP variants.

<table>
<thead>
<tr>
<th>GAPP variant</th>
<th>Average Rank</th>
<th>$p$-value</th>
<th>$p$-Hommel</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unoptimized</td>
<td>4.25</td>
<td>0.0084597</td>
<td>0.0166</td>
</tr>
<tr>
<td>Grammar*</td>
<td>1.50</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Time</td>
<td>4.55</td>
<td>0.0061698</td>
<td>0.003</td>
</tr>
<tr>
<td>Combined</td>
<td>2.50</td>
<td>0.0218124</td>
<td>0.025</td>
</tr>
<tr>
<td>Scaled</td>
<td>3.25</td>
<td>0.0023824</td>
<td>0.0166</td>
</tr>
</tbody>
</table>

Table 8.5 represents the Wilcoxon Signed Rank Sum significance tests between GAPP (Scaled) and GAPP (Combined) at $\alpha = 0.05$. It contains the $p$-value for the corresponding problem while “✓” states that the difference between the results of both the methods is significant; i.e., $p < 0.05$. 

198
Vargha-Delaney [Vargha and Delaney, 2000] A measure states how often that GAPP (Scaled) outperforms GAPP (Combined). A measure lies in between 0 and 1: when it is above 0.5, GAPP (Scaled) is better than GAPP (Combined); when it is 0.5, then both are equal; when it is less than 0.5 GAPP (Combined) is better than GAPP (Scaled); if it is close to 0.5 then the difference is small, otherwise the difference is large. For example, on Bubble sort with 16 cores, 35% of the time, GAPP (Scaled) performs better than GAPP (Combined). In other words, 65% of the time, GAPP (Combined) performs better than GAPP (Scaled). Overall, GAPP (Scaled) performs better than GAPP (Combined), in terms of the execution time.

Table 8.6 compares the mean best generation of the five GAPP (Un-optimized, Grammar, Time, Combined, Scaled) variants. The Friedman significance tests show that GAPP (Grammar) takes less number of generations to evolve a parallel program. However, GAPP (Scaled) exhibits better performance with the expense of a few extra generations.
8.6 Summary

This chapter analysed the code growth, computational complexity and thread scheduling in GAPP for both recursive and iterative sorting domains. Code growth has negligible effect on the performance of the evolving parallel programs. The computational complexity of the evolving programs has revealed interesting details both in recursion and iterative sorting. In recursion, the problem of excessive parallelism has been restricted with a little tweak in the GAPP grammars for recursion. In turn, this improved the degree of parallelism of the evolving parallel recursive programs. As a result, the performance of the evolving parallel programs has been further optimized.

For iterative sorting, the problem of optimal load balancing false sharing has been addressed with an enhanced GAPP grammars. In turn, the evolved parallel iterative sorting programs exhibited efficient performance.

Most interestingly, in both the problem domains, the evolved constants adapt to the experimental hardware environment, with which, the system has evolved architecture-aware programs. Finally, we noted that the time complexity of the programs is competing with the attempts in the literature.
Chapter 9

GAPP for Automatic Lock-free Programming

“Freedom is not worth having if it does not include the freedom to make mistakes.”
— Mahatma Gandhi, Leader

Writing correct and efficient parallel programs is an unavoidable challenge; the challenge becomes arduous with lock-free programming. This chapter presents an automated approach, Automatic Lock-free Programming (ALP) [Chennupati et al., 2015b], an application of GAPP. This avoids the programming difficulties presented to the typical programmer due to locks. ALP evolves the parallel lock-free recursive programs. These programs perform significantly better than parallel programs with locks, while they are competitive with the human written programs.

9.1 Introduction

As multi-core processors become the norm, it is compelling to develop parallel programs that efficiently coordinate processing shared resources. Al-
though powerful, parallel programming is a non-trivial task, as sequential developers still combat with the parallel design issues such as synchronization, locks, correctness and behavioural equivalence. Typically, locks ensure mutual exclusion among the shared objects; however, while locks assure program correctness, the performance of an application degrades when the contention for the shared resource is high, or when it is used often.

While avoiding locks is desirable [Harris, 2001], ensuring correct lock-free programs is often challenging even for the experts. Lock-free programming redesigns program semantics relying primarily on two operations\(^1\): first, atomic instructions such as read-modify-write, and second, memory barriers such as load and store. This chapter focuses on atomic operations, where thread waiting due to locks is eschewed to improve the performance of parallel programs.

We introduce Automatic Lock-free Programming (ALP), an application of GAPP, to automatically generate natively parallel recursive programs that avoid locks on shared resources. In fact, recursion is an interesting problem with its complexity in achieving atomicity, when the recursive calls are executed simultaneously. Thus, we synthesise the first natively parallel lock-free recursive programs.

### 9.2 Lock-free Programming

Lock-free programming employ multiple threads to operate on shared data, while refraining the threads from blocking each other. The atomic operations ensure that the tasks of one thread are invisible to other threads. It stops from accessing the inconsistent values, that is, half modified values of the thread under execution. It is important to avoid such errors in lock-free programming. This chapter focuses on the atomic read-modify-write operations on multi-cores. The following section describes this challenge with an example OpenMP recursive program.

\(^1\)https://msdn.microsoft.com/en-us/library/windows/desktop/ee418650(v=vs.85).aspx
9.2.1 Motivating Example

Figure 9.1 implements a parallel recursive Fibonacci program [Cormen et al., 2009] in C. It illustrates the importance of acquiring and/or releasing locks on shared objects, this section describes such a scenario.

```c
/* Fibonacci with parallel recursion */
int fib(int n) {
    int i, j;  if (n <= 2) return n;
    else {
        #pragma omp parallel sections shared(i, j, res)
        private(n)
        { //more than one thread in operation
            #pragma omp section
            { // update result in first call
                #pragma omp critical
                { i = fib(n-1); res += i; }
            }
            #pragma omp section
            { // update result in second call
                #pragma omp critical
                { j = fib(n-2); res += j; }
            }
        }
        return res; }
}
```

Figure 9.1: Motivating example: Fibonacci program in OpenMP.

The result of the two parallel recursive calls is saved in temporary variables \((i, j)\). To guarantee mutual exclusion among threads, the OpenMP critical construct stores the intermediate recursive calls and updates the end result \((res)\). Under-the-hood, the critical directive (see Chapter 3) locks the shared data to the thread in the region and restricts other threads from accessing it. As a result, the program fails to realize the true potential of the processor. It is necessary to avoid such locks for better performance.

In general, lock-free programming requires the knowledge of both hardware and compiler. Since even programming experts proposed erroneous [Val-
Automatic Lock-free Programming (ALP)

Automatic Lock-free Programming (ALP) generates parallel lock-free recursive using GAPP. These programs avoid the locking behaviour shown in Figure 9.1. ALP automates lock avoidance using GNU GCC atomic primitives, an established practice which is hard to use for a programmer.

9.3.1 Grammar Design

This section describes the design of the ALP grammar. Figure 9.2 shows the modifications in the GAPP grammar that was used to evolve parallel recursive programs (see Chapter 6). The modifications define a feasible lock-free space, which GAPP searches for an optimal solution. The program accepts an integer ($n$) input ($<input>$), while $res$ returns the result.

```
<parblocks> ::= <secblocks> | <taskblocks>
<secblocks> ::= #pragma omp section <newline> { <newline><line1>;<newline><lock-free> <newline> <line2> <newline> }
<taskblocks> ::= #pragma omp task <newline> { <newline> <line1>;<newline> <lock-free><line2> <newline> }
<lock-free> ::= __sync_fetch_and_add | __sync_fetch_and_sub | __sync_add_and_fetch | __sync_sub_and_fetch
<line1> ::= a=<expr>; <line2> ::= (&res, a);
```

Figure 9.2: Enhanced GAPP grammar to evolve lock-free parallel recursive Fibonacci programs. Complete grammar is in Figure B.12 of Appendix B.
The non-terminal `<ompppragma>` allows to choose between OpenMP task or data parallel directives, and further specialises with a choice between `<taskblocks>` and `<secblocks>`. The non-terminal `<taskblocks>` refers to the `task` work-sharing construct while `<secblocks>` refers to the `section` construct. The non-terminals `<private>` and `<shared>` map to thread-private (`n`) and thread-shared (`res`) variables respectively.

The non-terminal `<lock-free>` refers to the built-in GNU GCC atomic memory access primitives. Thus, the evolution suitably selects from these atomic `read-and-write` operations for addition (`__sync_fetch_and_add`, `__sync_add_and_fetch`) and the subtraction (`__sync_fetch_and_sub`, `__sync_sub_and_fetch`), which returns either an old or a new value as defined. The `<expr>` returns a simple integer value or a recursive call (`<stmt>`) of the program. The use of an inapt directive and/or an atomic primitive degrades the fitness of the evolved program.

### 9.3.2 Fitness Function

The goal is to generate a correct lock-free parallel recursive program while minimising the time to execute it. Although the use of any OpenMP pragma will result in a parallel program, the execution time changes significantly. To assure the right degree of parallelism, we use the fitness function ($f_{pprog}$) introduced in Chapter 6 for recursion.

Thus, the fitness function ($f_{pprog}$):

$$f_{pprog} = \frac{1}{(1 + t)^*} \cdot \frac{1}{1 + \frac{1}{N} \sum_{i=1}^{N} |y_i - \hat{y}_i|}$$

where, $t$ is the execution time of the program over $N$ fitness cases; $y_i$ and $\hat{y}_i$ are the desired and the evolved outputs of a program respectively. A detailed description of the $f_{pprog}$ is in Chapter 7. In eq. 9.1, the first term, *normalized execution time*, encourages the selection of an optimal OpenMP pragma, while the second term *normalized mean absolute error* helps to solve the problem. Thus, the fitness function promotes both efficient parallelization and correctness of a program.

205
9.4 Experiments

We report experiments that are designed to: 1) evaluate the performance of automatically generated lock-free benchmark recursive programs over its serial counterparts; and 2) measure speed-up of the lock-free programs over the GAPP (Combined) evolved parallel programs.

9.4.1 Experimental Setup

We evaluate ALP on five benchmark recursive programs in C, which vary in their complexity. The benchmarks are: Sum-of-N, Factorial, Fibonacci, Binary-Sum, and Reverse. These benchmarks are described in Chapter 6. Also, we use the same experimental parameters as well as the hardware and software specifications that are listed in Chapter 6 of Section 6.4.2.

Table 9.1: Mean best execution time (in secs) (MBT [standard deviation]) of ALP on different cores versus GE that generates sequential programs.

<table>
<thead>
<tr>
<th>Problem</th>
<th>Cores</th>
<th>Standard GE (serial programs)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>1</td>
<td>4607.8 [58.4]</td>
<td>1518.7 [43.4]</td>
</tr>
<tr>
<td>2</td>
<td>3745.6 [67.3]</td>
<td>1175.7 [71.6]</td>
</tr>
<tr>
<td>3</td>
<td>2456.1 [55.5]</td>
<td>1194.5 [44.1]</td>
</tr>
<tr>
<td>5</td>
<td>3467.1 [24.1]</td>
<td>1277.9 [23.9]</td>
</tr>
</tbody>
</table>

9.4.2 Results

We report two statistics of the proposed approach; mean best execution time (MBT) and speed-up.
Mean Best Execution Time

We compare the MBT of the evolved lock-free programs against two benchmarks; first, we compare against the serial programs; and then we compare against GAPP (Combined) evolved programs that include locks.

Table 9.1 compares the MBT of the automatically generated lock-free programs (on different number of cores) with that of the standard GE generated serial programs for all the five benchmarks. Note that the serial programs are generated automatically by omitting the OpenMP parallelization pragmas from the grammar shown in Figure 9.2. It is important to compare the performance of parallel programs with that of the serial counterparts, as poorly crafted parallel programs often obviate to achieve the promised gains [Herlihy, 1991]. That being the fact, it is always interesting when the parallel lock-free programs are produced automatically. The results show no difference in the execution time between the lock-free programs scheduled to execute on 2 cores and serial programs.

Table 9.2 shows the Wilcoxon Signed Rank Sum tests (at $\alpha = 0.05$) between ALP with different cores and GE. As expected, ALP significantly outperforms the standard GE when the resultant programs are scheduled to execute on 4, 8, and 16 cores. There is no significant difference between the two methods for 2 cores. In other words, a slight increase in the processing power has negligible effect on the performance of the ALP resultant programs, which is due to the OpenMP scheduling issues (see Chapter 7).

The $A$-measure describes the probability at which, ALP is better than GE. This measure is applicable only on the results that are proved to be statistically significant. In essence, when the $A$-measure $> 0.5$ then ALP is better than GE; when the $A$-measure $< 0.5$ then ALP is worse than the other; when $A$-measure $= 0.5$ then both are equal. The greater the magnitude of $A$-measure, better is the performance difference. If the $A$-measure is close to 0.5, then closer the difference; if it is far from 0.5, then farther the difference. Overall, auto-generated parallel recursive lock-free programs are significantly better than the serial counterparts. Particularly, the generated programs automate the use of atomic updates on shared variables.
Table 9.2: Wilcoxon Signed Rank Sum tests (at $\alpha = 0.05$) show that ALP outperforms serial programs on all the cores except 2. “✓” indicates that the results are significant. A measure shows the probability at which, ALP is better over GE.

<table>
<thead>
<tr>
<th>Cores</th>
<th>Program</th>
<th>Wilcoxon Signed Rank Sum Test</th>
<th>A measure</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Rank Sum $p$ value Significant</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Sum-of-N</td>
<td>2043 0.0426 ✓</td>
<td>0.3545</td>
</tr>
<tr>
<td>4</td>
<td>Factorial</td>
<td>2793 0.0187 ✓</td>
<td>0.8751</td>
</tr>
<tr>
<td>4</td>
<td>Fibonacci</td>
<td>3253 0.0361 ✓</td>
<td>0.6559</td>
</tr>
<tr>
<td>4</td>
<td>Binary-Sum</td>
<td>2529 0.0351 ✓</td>
<td>0.5215</td>
</tr>
<tr>
<td>4</td>
<td>Reverse</td>
<td>2471 0.0418 ✓</td>
<td>0.2851</td>
</tr>
<tr>
<td>8</td>
<td>Sum-of-N</td>
<td>2189 0.0063 ✓</td>
<td>0.6183</td>
</tr>
<tr>
<td>8</td>
<td>Factorial</td>
<td>2815 0.0318 ✓</td>
<td>0.3917</td>
</tr>
<tr>
<td>8</td>
<td>Fibonacci</td>
<td>3321 0.0112 ✓</td>
<td>0.7392</td>
</tr>
<tr>
<td>8</td>
<td>Binary-Sum</td>
<td>2079 0.0417 ✓</td>
<td>0.2719</td>
</tr>
<tr>
<td>8</td>
<td>Reverse</td>
<td>2479 0.0318 ✓</td>
<td>0.3851</td>
</tr>
<tr>
<td>16</td>
<td>Sum-of-N</td>
<td>1952 0.0426 ✓</td>
<td>0.3102</td>
</tr>
<tr>
<td>16</td>
<td>Factorial</td>
<td>2409 0.0018 ✓</td>
<td>0.6751</td>
</tr>
<tr>
<td>16</td>
<td>Fibonacci</td>
<td>3253 0.0046 ✓</td>
<td>0.8919</td>
</tr>
<tr>
<td>16</td>
<td>Binary-Sum</td>
<td>2221 0.0352 ✓</td>
<td>0.4556</td>
</tr>
<tr>
<td>16</td>
<td>Reverse</td>
<td>2008 0.0375 ✓</td>
<td>0.3184</td>
</tr>
</tbody>
</table>
Speed-up

We compare the speed-up of ALP resultant programs with that of GAPP (Combined). Figure 9.3 shows the speed-up of both approaches on all the five benchmarks for 2, 4, 8, and 16 cores. In both methods, the speed-up improves as the number of cores increase.

![Graph showing speed-up comparison between ALP and GAPP](image)

**Figure 9.3:** Speed-up in mean best execution time (MBT) of ALP (left) versus GAPP (right) for 2, 4, 8, and 16 cores.

Table 9.3 shows the Wilcoxon Signed Rank Sum tests (at $\alpha = 0.05$) with A-measure between ALP and GAPP. ALP significantly outperforms GAPP on 8 and 16 cores. There is no difference between both the approaches on 2 and 4 cores. That is, the impact of lock-free programs perform better at higher number of cores. However, an analysis on the efficiency of these programs better explains the impact of lock-free programming.

Efficiency

We compare the efficiency (defined in Chapter 3) of ALP and GAPP for 2, 4, 8, and 16 cores, as this is a useful way to analyze the scaling properties of the methods.

Table 9.4 presents the efficiency of ALP and GAPP for all five benchmarks on different cores. It is evident from the results that the efficiency of
Table 9.3: Wilcoxon Signed Rank Sum tests (at \( \alpha = 0.05 \)) show that ALP outperforms GAPP on all the cores except 2 and 4. “✓” states the results are significant and “✗” stands for insignificant difference. A measure shows the probability at which, ALP is better than GAPP.

<table>
<thead>
<tr>
<th>Cores</th>
<th>Program</th>
<th>Wilcoxon Signed Rank Sum Test</th>
<th>A measure</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Rank Sum value</td>
<td>Significant</td>
</tr>
<tr>
<td>4</td>
<td>Sum-of-N</td>
<td>2139   0.0926</td>
<td>✓</td>
</tr>
<tr>
<td>4</td>
<td>Factorial</td>
<td>2851   0.0857</td>
<td>✓</td>
</tr>
<tr>
<td>4</td>
<td>Fibonacci</td>
<td>3298   0.0618</td>
<td>✓</td>
</tr>
<tr>
<td>4</td>
<td>Binary-Sum</td>
<td>2711   0.5161</td>
<td>✓</td>
</tr>
<tr>
<td>4</td>
<td>Reverse</td>
<td>2628   0.5119</td>
<td>✓</td>
</tr>
<tr>
<td>8</td>
<td>Sum-of-N</td>
<td>2189   0.0432</td>
<td>✓</td>
</tr>
<tr>
<td>8</td>
<td>Factorial</td>
<td>2815   0.0318</td>
<td>✓</td>
</tr>
<tr>
<td>8</td>
<td>Fibonacci</td>
<td>3321   0.0411</td>
<td>✓</td>
</tr>
<tr>
<td>8</td>
<td>Binary-Sum</td>
<td>2079   0.0227</td>
<td>✓</td>
</tr>
<tr>
<td>8</td>
<td>Reverse</td>
<td>2479   0.0428</td>
<td>✓</td>
</tr>
<tr>
<td>16</td>
<td>Sum-of-N</td>
<td>2250   0.0311</td>
<td>✓</td>
</tr>
<tr>
<td>16</td>
<td>Factorial</td>
<td>2701   0.0182</td>
<td>✓</td>
</tr>
<tr>
<td>16</td>
<td>Fibonacci</td>
<td>3253   0.0461</td>
<td>✓</td>
</tr>
<tr>
<td>16</td>
<td>Binary-Sum</td>
<td>2221   0.0351</td>
<td>✓</td>
</tr>
<tr>
<td>16</td>
<td>Reverse</td>
<td>2008   0.0474</td>
<td>✓</td>
</tr>
</tbody>
</table>

210
Table 9.4: Efficiency of ALP and GAPP (Combined).

<table>
<thead>
<tr>
<th>Problem</th>
<th>ALP</th>
<th>GAPP (Combined)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Cores</td>
<td>Cores</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>1</td>
<td>0.57</td>
<td>0.70</td>
</tr>
<tr>
<td>2</td>
<td>0.46</td>
<td>0.74</td>
</tr>
<tr>
<td>3</td>
<td>0.77</td>
<td>0.79</td>
</tr>
<tr>
<td>4</td>
<td>0.68</td>
<td>0.74</td>
</tr>
<tr>
<td>5</td>
<td>0.48</td>
<td>0.65</td>
</tr>
</tbody>
</table>

both the methods on 2 and 4 cores is similar, whereas ALP is better than GAPP for 8 and 16 cores. Clearly, an improvement in the performance at higher number of cores with the lock-free programs. Next, we discuss ALP.

9.5 Discussion

We discuss the quality and the competence of the resultant lock-free programs. The quality is justified by focusing on the correctness of the resultant programs, while the competence is measured by comparing their performance with that of the human written parallel programs.

9.5.1 Best Program

Figure 9.4 shows the ALP generated parallel lock-free recursive Fibonacci program. The result (res) is shared among the threads, while the variable n is thread-private. The recursive calls are computed independently in parallel, where res is updated atomically by using the GCC atomic
__sync_add_and_fetch primitive. On executing this primitive, res gets a new value, thus avoids the locking problem.

```c
int fib(int n) {
    if (n <= 2) {
        int a=n;
        ___sync_fetch_and_add(&res, a);
    }
  else {
      #pragma omp parallel sections shared(res)private(n)
      { //more than one thread in operation
          #pragma omp section
          {
              int a = fib(n-1);
              ___sync_add_and_fetch(&res, a);
          }
      #pragma omp section
      {  
          int a = fib(n-2);
              ___sync_add_and_fetch(&res, a);
      }
     } return res; }
```

Figure 9.4: ALP evolved parallel lock-free recursive Fibonacci program.

Correctness

The correctness of the automatically generated lock-free programs is crucial. Since evolution searches through the feasible solution space, all the solutions except the best, become sub-optimal solutions. For example, line 12 of Figure 9.4 contains __sync_add_and_fetch, a sub-optimal solution of this is __sync_sub_and_fetch, which in fact is an incorrect solution.

Notice the non-terminal <lockless> shown in Figure 9.2, that contains four production rules for atomic operations. For Fibonacci program,
__sync_add_and_fetch is part of an optimal solution while the programs with the remaining three rules are sub-optimal. The best programs in the earlier generations of the evolutionary cycle often contain such sub-optimal solutions. However, ALP generates an optimal solution (Figure 9.4) in all the evolutionary runs for all the programs.

Table 9.5: Efficiency of the best programs of ALP versus human written programs for 8 and 16 cores with the significance tests.

<table>
<thead>
<tr>
<th>Problem</th>
<th>ALP</th>
<th>Human</th>
<th>Wilcoxon</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Cores</td>
<td>Cores</td>
<td>p-value</td>
<td>measure</td>
</tr>
<tr>
<td>8 16 8 16</td>
<td>8 16 8 16</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 0.78 0.59 0.81 0.67</td>
<td>0.57 0.09</td>
<td>–</td>
<td>–</td>
<td></td>
</tr>
<tr>
<td>2 0.54 0.61 0.64 0.65</td>
<td>0.10 0.60</td>
<td>–</td>
<td>–</td>
<td></td>
</tr>
<tr>
<td>3 0.76 0.63 0.60 0.54</td>
<td><strong>0.03</strong>  <strong>0.04</strong> 0.47 0.32</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4 0.81 0.75 0.68 0.62</td>
<td><strong>0.01</strong>  <strong>0.03</strong> 0.59 0.38</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5 0.73 0.58 0.69 0.61</td>
<td>0.07 0.1</td>
<td>–</td>
<td>–</td>
<td></td>
</tr>
</tbody>
</table>

9.5.2 Human Competitive

While it is obvious to a human programmer to utilise the critical directive to ensure correctness of the program, if not obvious how to do so, evolution in GAPP entails a learning curve where it learns about program semantics as well as the most suitable parallel constructs. Therefore, we present a valuable proof of concept even if the target programs have expert human programmed solutions. Our further work will explore applications where
forgoing locks adds significant programming complexity and/or when the program clearly requires performance optimisation.

Table 9.5 compares the efficiency of all the five best lock-free programs with that of the human written programs on 8 and 16 cores. The Wilcoxon Signed Rank Sum tests with Vargha-Delaney A-measure. The $p$-value is in boldface for the corresponding program when the efficiency of ALP programs is significantly better than human written programs. Overall, the results indicate that the automatically generated lock-free programs are competitive with the human written programs. However, they are limited by the Linux kernel scalability issues [Boyd-Wickizer et al., 2010] in reaching from ideal efficiency.

The auto-generated lock-free programs have only just begun competing with the human written programs; there is more ground to be made up. This is due to the fact that the latter uses OpenMP atomic that is native to that library. It does not require any extra overhead for built-in calls as opposed to a system call for the GCC atomic memory calls. However, OpenMP atomic can deal only with small operations. In case of larger operations, it fails to guarantee same scale of performance, whereas ALP overcomes such limitations. In fact, this research of synthesising the parallel recursive (GAPP) and the lock-free (ALP) programs has allowed us to win the prestigious Humies Silver Award\textsuperscript{1} in Genetic and Evolutionary Computation Conference 2015.

### 9.5.3 Limitations

We list some of the limitations of ALP. As explained in section 9.5.1 use of an inapt atomic primitive results in a poor solution. Quite rarely, use of a correct primitive may still fail to produce a correct solution. For example, in some cases the target CPU architecture does not support 16-byte types, then there is no guarantee for the atomicity of read-modify-write operations.

\textsuperscript{1}Note, that this award was introduced by the creator of Genetic Programming to encourage the growth of evolutionary techniques into various scientific domains.
9.6 Summary

In summary, we presented an approach that effectively exploits the computational capability of the multi-core processors using GAPP. The automatically produced lock-free programs used both OpenMP directives and GNU GCC atomic primitives. These programs proved to significantly outperform the serial programs, while they are better than the state-of-the-art implementations at higher number of cores. Also, the performance of these programs are competitive with the human written programs.
Part IV

Conclusions and Future Directions
Chapter 10

Conclusions and Future Recommendations

“Arise! Awake! and stop not till the goal is reached”

— Swami Vivekananda, Monk

The hypothesis of this thesis was to use Grammatical Evolution (GE) for the synthesis of parallel programs to exploit the computational capabilities of multi-cores. We proposed GAPP for the evolution of native parallel programs with the design of grammars that include OpenMP pragmas. GAPP was analysed and applied for the evolution of lock-free parallel programs. The remainder of this chapter is organised as follows. Section 10.1 summarizes the work in this thesis. Finally, section 10.2 presents the potential opportunities for future research based on the findings of this thesis.

10.1 Summary

The aim of this thesis was to examine the use of GE as a method for the synthesis of parallel programs on multi-core processors. Specifically, this
research focused on the design of grammars that use OpenMP primitives in
the evolution of those parallel programs. In achieving this goal a number of
research questions were raised, which were presented in Chapter 1. Also,
this chapter explained the objectives and contributions of this thesis.

A survey on the evolutionary automatic programming techniques was
presented in Chapter 2, where, the inner workings of GE were detailed.
Thereafter, Chapter 3 discussed the growth of multi-cores, reviewed the con-
ventional automatic parallel code generation techniques, in which, the focus
was drawn towards the evolutionary automatic parallel code generation.

In Chapter 4, the computational complexity of GE was analysed on two
archetypal problems: Santa Fe Ant Trail and Binomial-3, which were the
representation of both control and classification. The analysis showed that
the complexity of GE on both the problems was linear and quadratic in na-
ture for a given population size. This behaviour was analysed on the fitness
evaluation phase of GE, meaning that the complexity varies with the exper-
imental problem. A theoretical analysis on the complete evolutionary cycle
of GE is of future interest. This study resulted in several interesting pat-
tterns in the parameters such as terminating generation, actual and effective
lengths of genotypes. These observations allowed us to present a evolution-
ary run prediction model (RPM), which learned from the initial generations
of a GE run. RPM predicted whether a given GE run produces a qualitative
solution or otherwise with an average accuracy of 63.31%. RPM+GE im-
proved the mean best fitness over standard GE by processing less number of
individuals while reduced the overall execution time of GE. An underscored
result of this research was the creation of the novel datasets that contain the
information regarding the early generation of an evolutionary run. This data
can be leveraged in future to further enhance the current system.

Grammatical Automatic Parallel Programming (GAPP) system was in-
troduced in Chapter 5 that synthesized parallel regression programs. GAPP
was examined on two standard benchmarks: multiplexer and sextic regres-
sion. We designed the parallel aware grammars using OpenMP pragmas.
The evolved parallel programs exhibited a single program multiple data
(SPMD) parallelization strategy, where a single program was operated on

218
different data items across multiple cores. Although the reduction in the execution time of the evolving programs was promising, it fails to scale as the number of fitness cases and the cores increase. Apart from that, the selected regression benchmarks are hard to solve, thereby it became more difficult to analyse the real impact of parallelization.

In order to synthesise parallel programs that can solve a given problem, GAPP was examined on a number of tractable problems in Chapter 6. These computable problems belong to both recursive and iterative sorting domains. The designed grammars explored various parallel programming possibilities while maintained the solution correctness. The resultant parallel programs have significantly reduced the execution time on multi-cores.

In this investigation to optimize the performance of the evolving programs, Chapter 7 presented a few ancillary extensions both for the design of the grammars and fitness evaluation. The changes included the separation of the task and data parallel constructs and the fitness function included the execution time of the evolving programs. These extensions facilitated the quick generation of efficient parallel programs.

However, these efficient parallel programs posed a few challenges in both the computable problem domains. We analysed GAPP on these two domains in Chapter 8 and observed a few interesting characteristics in the growth of the GAPP individuals. Effective size of GAPP individuals increased over that of GE while actual size remained same for both the approaches. For recursion, the excessive creation of threads blocked the performance of the evolving parallel programs. The performance of these programs was optimized by arresting the excessive creation of the parallel threads. For iterative sorting, creation of small data arrays lead to false sharing, which was dealt with the creation of larger data arrays that were ceased from staying on the same cache line. A sublime feature was that the resultant parallel programs adapt to the experimental problem and the underlying hardware architecture.

Finally, in Chapter 9 an application of GAPP was presented for the automatic evolution of parallel lock-free recursive programs. Especially, when writing lock-free programs in itself is hard even for the experts, evolving
lock-free parallel programs had gained an unavoidable attention. The grammars were modified to include GNU GCC atomic primitives that guaranteed the lock-freedom. The resultant programs exhibited a significant improvement in the execution time over the sequential and parallel programs (contains locks). Also, the performance of these programs was competitive with that of the human written programs.

10.2 Future Recommendations

This thesis described the synthesis of parallel programs on multi-core architectures. However, as with most research studies, there are multiple avenues for future research.

Enhancing Run Prediction Model

Along with a bootstrapping of the RPM system (in Chapter 4) and rapidly retraining the predictors during the evolutionary run (as opposed to the off-line learning in RPM), we can enhance RPM in several directions. One direction is to extrapolate [Dym and McKean, 1970] an evolutionary run after a few initial generations, that is, forecasting the quality of a run. Although extrapolating an evolutionary run will not generate a model, we still can get an estimate of the fitness value, which, in fact is sufficient to decide the progression of the run. Another interesting way is, instead of preparing a dataset with the raw values of changes in certain characteristics, it is better to include the percentage of changes so that the resultant prediction system can be more general to apply on an unseen regression problems.

Multi-Objective Grammatical Evolution

The fitness function of GAPP uses a dual objective of execution time and normalized error. It might improve the search process, if the underlying search algorithm of GAPP is replaced with a multi-objective evolutionary
algorithm. Similar to [Nasuf et al., 2011], [Byrne et al., 2011], which will further enhance the evolvability and better optimize both the objectives.

**Behavioural Program Synthesis**

The current GAPP system maintains a separate grammar for each of the benchmarks used in the experiments. An interesting exploration would be to mix the grammars of all the problems, thereby observe how the increase in the search space influences the evolution of a particular program. For example, if we mix the grammars of *Bubble sort* and *Rank sort*, it would be interesting to see the behaviour of the resulting programs. That is, finding out whether the evolved program sorts an array by following a Bubble sort procedure or a Rank sort procedure. It is worth identifying the fact that whether GE prefers to evolve a behaviourally simple programs or computationally easier programs. Remember a behaviourally easier program can be complex computationally and computationally easier program can be complex behaviourally. Hence, an interesting future direction is to explore the synthesis of behavioural programs.

**Code Growth**

We observed that the effective size of GAPP individuals increased relative to those of GE while, surprisingly, the actual size remained same. A similar behaviour was observed in [Azad and Ryan, 2014]. The reasons for such an effect are, as yet unknown, and deserve further investigation.

**Automatic Memory Re-ordering**

Another interesting future direction is to automate the memory reordering operations. We also recommend to address the difference between the wait-free synchronization [Herlihy, 1991] as well as the lock-free programming through program synthesis.
Next, an intriguing application is the synthesis of lock-free programs on Xbox gaming console [Xbox 360, 2012]. Xbox uses a Windows based operating system (OS), Xbox One. In general, synchronisation of the threads on Windows varies greatly depending on the processor architecture and configuration. The Xbox OS is a stripped down version of the traditional desktop based Windows OS that runs virtually on Xbox console. Therefore, Xbox OS boosts the performance of the applications when compared to the desktop OS.

However, the challenge becomes more difficult if we are to synthesise lock-free programs for Xbox gaming console, because of the fact that it does not support instruction re-ordering, instead supporting write re-orderings, for which, writing lock-free programs will be an interesting endeavour. Also, the scale of the experimental problem increases while synthesising the programs for Xbox console.

Scalability

We examined GAPP on difficult standard benchmarks in the parallel programming domain. The next step will be to ramp up the difficulty and scale of the problems to establish how far this work can go in an industrial setting.

Multi-staged Approach

Another promising set of future experiments is to implement a novel multi-staged parallelization approach. That is, first we can find a correct solution and then parallelize it. We can use a multi-staged approach where the first stage finds the correct solution while the second stage maximizes the parallelism.

Arbitrary Program Synthesis

A final vein of future research is to extend beyond C and explore the potential of interpreted languages for evolving parallel code. This will decrease
the time taken to complete the evolutionary simulations that currently rely on external system calls to write the evolving programs on to the disk, compile and then execute it as an external process. In similar vein, GAPP can be generalized to evolve parallel programs in other programming environments that support OpenMP like fork/join parallelism. An example of such programming environment is the use of JOMP [Bull and Kambites, 2000] library, an OpenMP like API for JAVA. In a similar vein, synthesising programs using Intel and Cray parallel programming APIs will foster the speed-up over GCC while it also fulfils the purpose of arbitrary programming.
Appendix A: Prediction Models

This section presents the ACO discovered classification models for the symbolic regression benchmarks used in Chapter 4.

\[
\begin{align*}
\text{IF } \text{AELC} &> 5.18 \quad \text{THEN } \text{No}; \\
\text{IF } \text{AALC} &> 7.5875 \quad \text{THEN } \text{No}; \\
\text{IF } \text{AFC} &\leq 0.0851 \quad \text{AND} \quad \text{AELC} \leq 3.995 \quad \text{THEN } \text{No}; \\
\text{IF } \text{AELC} &\leq 0.1075 \quad \text{THEN } \text{Yes}; \\
\text{IF } \text{AELC} &\leq 0.3025 \quad \text{THEN } \text{No}; \\
\text{IF } 6.3825 &< \text{AALC} \leq 6.5425 \quad \text{THEN } \text{No}; \\
\text{IF } \text{AELC} &> 3.1974 \quad \text{THEN } \text{No}; \\
\text{IF } \text{AELC} &\leq 0.45 \quad \text{THEN } \text{Yes}; \\
\text{IF } \text{BFC} &> 0.0904 \quad \text{THEN } \text{Yes}; \\
\text{IF } \text{BFC} &> 0.0513 \quad \text{THEN } \text{Yes}; \\
\text{IF } \text{BFC} &> 0.0227 \quad \text{AND} \quad \text{AFC} > 0.0874 \quad \text{THEN } \text{No}; \\
\text{IF } \text{BFC} &> 0.0296 \quad \text{THEN } \text{Yes}; \\
\text{IF } \text{AELC} &> 2.4074 \quad \text{THEN } \text{No}; \\
\text{IF } \text{AELC} &> 1.87 \quad \text{THEN } \text{No}; \\
\text{IF } \text{BFC} &\leq 0.0 \quad \text{THEN } \text{No}; \\
\text{IF } \text{AFC} &> 0.1031 \quad \text{THEN } \text{No}; \\
\text{IF } \text{AELC} &> 0.7675 \quad \text{THEN } \text{Yes}; \\
\text{IF } <\text{empty}> &\quad \text{THEN } \text{No};
\end{align*}
\]

**Figure A.1:** ACO discovered predictive model for problem $f_2$
IF BFC < 0.1972 THEN No;
IF AFC > 0.274 THEN No;
IF AELC > 6.6425 THEN No;
IF AALC <= 4.5175 THEN No;
IF AFC <= 0.1897 AND AELC > 2.8475 THEN No;
IF AFC <= 0.1541 THEN No;
IF BFC <= 0.1965 AND AELC > 3.8425 THEN No;
IF AELC <= 2.81 AND AALC > 4.825 AND BFC > 0.0069 THEN No;
IF BFC <= 3.0E−4 THEN yes
IF BFC > 0.1422 AND AFC <= 0.2131 THEN Yes;
IF AALC > 5.975 THEN No;
IF <empty> THEN Yes;

Figure A.2: ACO discovered predictive model for problem $f_3$

IF AALC > 5.76 THEN No;
IF BFC > 0.0406 THEN Yes;
IF BFC < 0.0358 THEN No;
IF AFC < 0.0583 THEN No;
IF AALC <= 2.4875 THEN No;
IF AALC <= 3.3675 THEN Yes;
IF AELC > 0.6675 AND AFC > 0.0445 THEN No;
IF AFC < 0.0551 THEN No;
IF AFC > 0.0439 AND AALC > 3.845 THEN Yes;
IF BFC < 0.0351 THEN No;
IF AELC <= −0.0925 THEN No;
IF AELC <= 0.2625 THEN Yes;
IF AFC > 0.0467 THEN Yes;
IF BFC > 0.0142 AND AFC > 0.0407 THEN Yes;
IF AFC > 0.0402 AND BFC > 0.0051 THEN No;
IF AELC > 1.6425 THEN Yes;
IF <empty> THEN No;

Figure A.3: ACO discovered predictive model for problem $f_4$
Appendix B: Parallel Grammars

This section presents the parallel grammars for the experiments described in the chapters 5, 6, 7, 8, and 9.

```
<start> ::= <omp_prag> <for_loop> <data> <evolve_exp>
<omp_prag> ::= #pragma omp parallel shared(Evolve, CHUNK)<newline>{<newline> <omp_for>
<omp_for> ::= #pragma omp for schedule(dynamic, CHUNK) private(i, temp)<newline>{
<for_loop> ::= for(i=0;i<FITNESS_CASES;i=i+1) {
<data> ::= A0 = X[i][0];<newline> A1 = X[i][1];
A2 = X[i][2];<newline> D0 = X[i][3];
D1 = X[i][4];<newline> D2 = X[i][5];
D3 = X[i][6];<newline> D4 = X[i][7];
D5 = X[i][8];<newline> D6 = X[i][9];
D7 = X[i][10];<newline>
<evolve_exp> ::= temp = <expr>; <newline> <assign>
<expr> ::= (<expr><op><expr>)|<pre-op>|<var>
<op> ::= AND | OR <newline> ::= 
<pre-op> ::= NOT(<expr>) | IF((<expr>), (<expr>), (<expr>))
<var> ::= A0 | A1 | A2 | D0 | D1 | D2 | D3 |
D4 | D5 | D6 | D7
<assign> ::= Evolved[i] = temp;
} } } // end parallel region
```

Figure B.1: Design of the GAPP grammar for a 11-multiplexer problem, where A in <var> stands for address bits and D refers to the data bits.
<ompppragma> ::= #pragma omp parallel for <newline> { <parcode> | #pragma omp parallel <newline> { <parcode> | #pragma omp parallel sections <newline> { <newline> <parblocks> }

<parcode> ::= if( <var> <lop> <const> ) { <newline> int a=<expr>; res<bop>=a; <newline> } else { <newline> int a=<expr>; <newline> res<bop>=a; } <newline> <result> <newline> }

<parblocks> ::= if( <var> <lop> <const> ) {<newline> int a=<expr>; <newline> } else { <newline> <blocks> <newline> }

<blocks> ::= <blocks> | <blocks><newline><blocks> | #pragma omp section <newline> { <newline> int a=<stmt>; #pragma omp atomic <newline> res <bop>= a; } <newline> <newline> <result> <newline>

<result> ::= return <var>;

<expr> ::= <var> | <stmt> | <var> <bop> <stmt>

<stmt> ::= sumofn(<var> <bop> <const>)

<bop> ::= + | - | * | /

<lop> ::= >= | <= | > | < | ==

<const> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<var> ::= n | res

<newline> ::= \n

Figure B.2: GAPP grammar that generates a parallel recursive Sum-of-N program.
Figure B.3: GAPP grammar that generates a parallel recursive Factorial program.
Figure B.4: GAPP grammar that generates a parallel recursive *Binary-Sum* program.
Figure B.5: GAPP grammar that generates a parallel recursive Reverse program.
Figure B.6: GAPP grammar that generates a parallel recursive Quicksort program.
Figure B.7: GAPP grammar to generate parallel iterative Bubble sort program.
<program> ::= <for_out><newline><condition>

<condition> ::= if(<index><bop><const><lop><const>){
                <ompprogram> } else {<ompprogram>}

<ompprogram> ::= <omppragma> <shared> <private>
                <newline><for_in>

<omppragma> ::= #pragma omp parallel for | #pragma omp parallel | #pragma omp parallel sections

<shared> ::= shared(<var>) { <newline>

<private> ::= private(<index>) <var> ::= A

<for_out> ::= for(i=0; i<length; i++) {

<for_in> ::= for(j=1; j<length-1; j+=2){<newline>

<for_in_line> ::= if(<var>[abs(<index><bop><const>)])
                <lop> <var>[abs(<index><bop>
                <const>)]) { <newline> <swap> }

<swap> ::= temp=<var>[abs(<index><bop><const>)]
         ;<newline> <var>[abs(<index><bop>
         <const>)=<var>[abs(<index> <bop>
         <const>)];<newline> <var> [abs
         (<index><bop><const>)] = temp;

<const> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<bop> ::= + | - | * | / | %

<index> ::= i | j
<lop> ::= >= | <= | < | >

<newline> ::= \n

Figure B.8: GAPP grammar to generate parallel iterative Odd-Even sort program.
| <program>    | ::= <ompppragma><private><shared><code> |
| <ompppragma> | ::= #pragma omp parallel for | #pragma omp parallel | #pragma omp parallel sections |
| <shared>     | ::= shared(<var>) { <newline> |
| <private>    | ::= private(<index>) |
| <code>       | ::= for(i=1; i<length; i++) { <newline> |
| <for_j>      | ::= for(j=0; j<i; j++) { <newline> |
| <condition>  | ::= if( <for_var> <lop> <for_var> ) { <newline> |
| <for_var>    | ::= A[<index>] | R[<index>] |
| <for_setop>  | ::= E[<for_var>] = <for_var> |
| <for_c>      | ::= for(i=0; i<length; i++) <newline> |
| <bop>        | ::= + | - |
| <lop>        | ::= >= | <= | > | < | == |
| <const>      | ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| <index>      | ::= i | j |
| <newline>    | ::= \n |

Figure B.9: GAPP grammar to generate parallel iterative Rank sort program.
Figure B.10: GAPP grammar to generate efficient parallel recursive Fibonacci program. These changes are common to the problems in Figure B.2–B.6.
Figure B.11: GAPP grammar to evolve an efficient parallel iterative Odd-Even sort algorithm. These changes are common to the problems in Figure B.7–B.9
Figure B.12: ALP grammar for lockfree parallel recursive Fibonacci programs.
Bibliography


M. Galea and Q. Shen. Simultaneous ant colony optimization algorithms for learning linguistic fuzzy rules. In A. Abraham, C. Grosan, and V. Ramos, editors, *Swarm Intelligence in Data Mining*, volume 34 of


262


H. Suzuki, H. Sawai, and W. Piaseczny. Chemical genetic algorithms—
evolutionary optimization of binary-to-real-value translation in genetic

J. M. Swafford. *Analyzing the Discovery and the use of Modules in Gram-

G. Syswerda. A study of reproduction in generational and steady state ge-

R. Tanese. Distributed genetic algorithms. In *Proceedings of the Third Inter-

I. Tanev. Genetic programming incorporating biased mutation for evolu-
tion and adaptation of snakebot. *Genetic Programming and Evolvable

E. M. Tansey. Not committing barbarisms: Sherrington and the synapse,

A. Teller. The evolution of mental models. In K. E. Kinnear Jr., editor,
*Advances in Genetic Programming*, pages 199–217. MIT Press, Cam-
bridge, MA, USA, 1994.

system microarchitecture. *IBM Journal of Research and Development*,

W. Thies, M. Karczmarek, and S. Amarasinghe. Streamit: A language for
streaming applications. In *Proceedings of the 11th International Con-
ference on Compiler Construction*, CC’02, pages 179–196. Springer,
2002.

A. Topchy and W. F. Punch. Faster genetic programming based on local
gradient search of numeric leaf values. In L. Spector, E. D. Good-
man, A. Wu, W. B. Langdon, H.-M. Voigt, M. Gen, S. Sen, M. Dorigo,


